

TEXTBOOK

Amin Zollanvari

Machine Learning with Python

Theory and Implementation



Springer

Machine Learning with Python

Amin Zollanvari

Machine Learning with Python

Theory and Implementation



Springer

Amin Zollanvari  
Department of Electrical
and Computer Engineering
Nazarbayev University
Astana, Kazakhstan

ISBN 978-3-031-33341-5 ISBN 978-3-031-33342-2 (eBook)
<https://doi.org/10.1007/978-3-031-33342-2>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

*Lovingly dedicated to my family:
Ghazal, Kian, and Nika*

Preface

The primary goal of this book is to provide, as far as possible, a concise systematic introduction to practical and mathematical aspects of machine learning algorithms. The book has arisen from about eight years of my teaching experience in both machine learning and programming, as well as over 15 years of research in fundamental and practical aspects of machine learning. There were three sides to my main motivation for writing this book:

- **Software skills:** During the past several years of teaching machine learning as both undergraduate and graduate level courses, many students with different mathematical, statistical, and programming backgrounds have attended my lectures. This experience made me realize that many senior year undergraduate or postgraduate students in STEM (Science, Technology, Engineering, and Mathematics) often have the necessary mathematical-statistical background to grasp the basic principles of many popular techniques in machine learning. That being said, a major factor preventing students from deploying state-of-the-art existing software for machine learning is often the lack of programming skills in a suitable programming language. In the light of this experience, along with the current dominant role of Python in machine learning, I often ended up teaching for a few weeks the basic principles of Python programming to the extent that is required for using existing software. Despite many excellent practical books on machine learning with Python applications, the knowledge of Python programming is often stated as the prerequisite. This led me to make this book as self-contained as possible by devoting two chapters to fundamental Python concepts and libraries that are often required for machine learning. At the same time, throughout the book and in order to keep a balance between theory and practice, the theoretical introduction to each method is complemented by its Python implementation either in Scikit-Learn or Keras.
- **Balanced treatment of various supervised learning tasks:** In my experience, and insofar as supervised learning is concerned, treatment of various tasks such as binary classification, multiclass classification, and regression is rather imbalanced in the literature; that is to say, methods are often presented for

binary classification with the multiclass classification being left implicit; or even though in many cases the regressor within a class of models can be easily obtained from the classifier within the same family, the focus is primarily on the classification side. Inconsistency in treating various tasks often prevents learners from connecting the dots and seeing a broader picture. In this book, I have attempted to strike a balance between the aforementioned tasks and present them whenever they are available within a class of methods.

- **Focus on “core” techniques and concepts:** Given the myriad of concepts and techniques used in machine learning, we should naturally be selective about the choice of topics to cover. In this regard, in each category of common practices in machine learning (e.g., model construction, model evaluation, etc.), I included some of the most fundamental methods that: 1) have strong track record of success in many applications; and 2) are presentable to students with an “average” STEM background—I would refer to these as “core” methods. As a result of this selection criterion, many methods (albeit sometimes not enthusiastically) were excluded. I do acknowledge the subjectivity involved in any interpretation of the criterion. Nonetheless, I believe the criterion or, perhaps better to say, how it was interpreted, keeps the book highly relevant from a practical perspective and, at the same time, makes it accessible to many learners.

Writing this book would have not been possible without a number of mentors and colleagues who have constantly supported and encouraged me throughout the years: Edward R. Dougherty, Ulisses M. Braga-Neto, Mehdi Bagheri, Behrouz Maham, Berdakh Abibullaev, Siamac Fazli, Reza Sameni, Gil Alterovitz, and Mohammad Ali Masnadi-Shirazi. I thank them all. I am especially indebted to Edward R. Dougherty and Ulisses M. Braga-Neto whose taste for rigorous research in pattern recognition and machine learning taught me a lot. I would like to thank my students whose comments inside and outside the classroom have contributed to making this a better book: Samira Reihanian, Azamat Mukhamediya, Eldar Gabdulsattarov, and Irina Dolzhikova. I would also like to thank my editor at Springer, Alexandru Ciolan, for his comments and patience. I am grateful to numerous contributors and developers in the Python community who have made machine learning an enjoyable experience for everyone. I also wish to acknowledge Vassilios Tourassis and Almas Shintemirov for their roles in creating such an intellectually stimulating environment within the School of Engineering and Digital Sciences at Nazarbayev University in which I had the privilege of interacting with many brilliant colleagues and spend a substantial amount of time to write this book.

Last, but not least, I wish to express my sincere thanks to my wife, Ghazal, for her unwavering support, encouragement, and patience during this endeavour.

Astana, Kazakhstan
April, 2023

Amin Zollanvari

About This Book

Primary Audience

The primary audience of the book are undergraduate and graduate students. The book aims at being a textbook for both undergraduate and graduate students that are willing to understand machine learning from theory to implementation in practice. The mathematical language used in the book will generally be presented in a way that both undergraduate (senior year) and graduate students will benefit if they have taken a basic course in probability theory and have basic knowledge of linear algebra. Such requirements, however, do not impose many restrictions as many STEM (Science, Technology, Engineering, and Mathematics) students have taken such courses by the time they enter the senior year. Nevertheless, there are some more advanced sections that start with a \oplus and when needed end with a separating line “——” to be distinguished from other sections. These sections are generally geared towards graduate students or researchers who are either mathematically more inclined or would like to know more details about a topic. Instructors can skip these sections without hampering the flow of materials. The book will also serve as a reference book for machine learning practitioners. This audience will benefit from many techniques and practices presented in the book that they need for their day-to-day jobs and research.

Organization

The book is organized as follows. The first chapter provides an introduction to important machine learning tasks and concepts. Given the importance of Python both in machine learning and throughout the book, in Chapters 2 and 3, basics of Python programming will be introduced. In particular, in Chapter 2 readers will start from the very basics in Python and proceed to more advanced topics. In Chapter 3, three fundamental Python packages will be introduced:

- 1) `numpy` is a fundamental package at the core of many data science libraries.
- 2) `pandas` is a useful library for data analysis, which is built on top of `numpy`.
- 3) `matplotlib`, which is the primary plotting library in Python.

Chapters 2 and 3 can be entirely skipped if readers are already familiar with Python programming and the aforementioned three packages. In Chapter 4, after a brief introduction to scikit-learn, we start with an example in which we introduce and practice data splitting, data visualization, normalization, training a classifier (k-nearest neighbors), prediction, and evaluation. This chapter is also important in the sense that Python implementation of techniques introduced therein will be used intact in many places throughout the book as well as in real-world applications.

Having used a predictive model, namely, k-nearest neighbors (kNN) in Chapter 4, Chapters 5 to 8 provide detailed mathematical mechanism for a number of classifiers and regressors that are commonly used in the design process. In particular, in Chapter 5 we start this journey from kNN itself because of its simplicity and its long history in the field.

Chapter 6 covers an important class of machine learning methods known as linear models. Nonetheless, this chapter will not attempt to provide a complete reference of linear models. Instead, it attempts to introduce the most widely used linear models including linear discriminant analysis, logistic regression with/without regularization, and multiple linear regression with/without regularization.

Chapter 7 introduces decision trees. Decision trees are nonlinear graphical models that have found important applications in machine learning mainly due to their interpretability as well as their roles in other powerful models such as random forests (Chapter 8) and gradient boosting regression trees (Chapter 8).

Chapter 8 discusses ensemble learning. There we cover a number of ensemble learning techniques that have a strong track record in many applications: stacking, bagging, random forest, pasting, AdaBoost, and gradient boosting.

Once a classifier or regressor is constructed, it is particularly important to estimate its predictive performance. Chapter 9 discusses various estimation rules and metrics to evaluate the performance of a model. Because generally model evaluation is also critical for another salient problem in machine learning, namely, model selection, this chapter covers both model evaluation and selection. In Chapter 10, we cover three main types of feature selection including filter, wrapper, and embedded methods.

Given a dataset, a practitioner often follows several steps to achieve one final model. Some common and important steps in this regard include, but not limited to, normalization, feature selection and/or extraction, model selection, model construction and evaluation. The misuse of some of these steps in connection with others have been so common in various application domains that have occasionally triggered warnings and criticism from the machine learning community. In Chapters 4 to 10, we will see a restricted implementation of these steps; that is to say, an implementation of some of these steps, sometimes in silos and other times with few others. Nonetheless, in Chapter 11, we discuss in details how one could properly implement all these steps, if desired, and avoid potential pitfalls.

Chapter 12 discusses an important unsupervised learning task known as clustering. The main software used in Chapters 4-12 is scikit-learn. However, scikit-learn

is not currently the best software to use for implementing an important class of predictive models used in machine learning, known as, artificial neural networks (ANNs). This is because training “deep” neural networks requires estimating and adjusting many parameters and hyperparameters, which is computationally an expensive process. As a result, successful training of various forms of neural networks highly relies on parallel computations, which are realized using Graphical Processing Units (GPUs) or Tensor Processing Units (TPUs). However, scikit-learn does not support using GPU or TPU. At the same time, scikit-learn does not currently support implementation of some popular forms of neural networks known as convolutional neural networks or recurrent neural networks. As a result, we have postponed ANNs and “deep learning” to Chapter 13-15. There, we switch to Keras with TensorFlow backend because they are well optimized for training and tuning various forms of ANN and support various forms of hardware including CPU, GPU, or TPU.

In Chapter 13, we discuss many concepts and techniques used for training deep neural networks (deep learning). As for the choice of architecture in this chapter, we focus on multilayer perceptron (MLP). Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) will be introduced in Chapter 14 and 15, respectively.

Throughout the book, a  shows some extra information on a topic. Places marked like this could be skipped without affecting the flow of information.

Note for Instructors

Due to the integrated theoretical-practical nature of the materials, in my experience having a flipped classroom would work quite well in terms of engaging students in learning both the implementation and the theoretical aspects of materials. The following suggestions are merely based on my experience and instructors can adjust their teaching based on their experience and class pace.

In a flipped learning mode, I generally divide the weekly class time into two sessions: 1) a theoretical session in which a summary of the materials will be given, followed by questions and discussions; and 2) a practical/problem-solving session. Students are then required to read materials and run given codes prior to the practical session. Students are also supposed to solve given exercises at the end of each chapter before coming to the practical session. During the practical/problem-solving session, all students will connect to an online platform to present their solutions for given exercises. This practice facilitates sharing solutions and engaging students in the discussion. Multiple solutions for a given exercise is encouraged and discussed during the class. Furthermore, students are allowed and encouraged to create and present their own exercises. In order to cover the materials, I would generally consider students technical level in terms of both programming and mathematics:

1. If students have already a good knowledge of Python programming and the three fundamental packages `numpy`, `pandas`, and `matplotlib`, Chapters 2 and

- 3 can be skipped. In that case, each of the remaining chapters could be generally covered within a week.
2. If students have no knowledge of Python programming, Chapters 2 and 3 can be covered in two weeks. In that case, I would suggest covering either Chapters 1-11, 13-14 or Chapters 1-9, 11, 13-15.
 3. Depending on the class pace or even the teaching strategy (for example, flipped vs. traditional), instructors may decide to cover each of Chapters 2, 9, and 13 in two weeks. In that case, instructors can adjust the teaching and pick topics based on their experience.

A set of educational materials is also available for instructors to use. This includes solutions to exercises as well as a set of Jupyter notebooks that can be used to dynamically present and explain weekly materials during the classroom. In particular, the use of Jupyter notebook allows instructors to explain the materials and, at the same time, run codes during the presentation. There are also different ways that one can use to present notebooks in a similar format as “slides”.

Contents

Preface	vii
About This Book	ix
1 Introduction	1
1.1 General Concepts	1
1.1.1 Machine Learning	1
1.1.2 Supervised Learning	2
1.1.3 Unsupervised Learning	3
1.1.4 Semi-supervised Learning	5
1.1.5 Reinforcement Learning	5
1.1.6 Design Process	6
1.1.7 Artificial Intelligence	7
1.2 What Is “Learning” in Machine Learning?	7
1.3 An Illustrative Example	9
1.3.1 Data	9
1.3.2 Feature Selection	9
1.3.3 Feature Extraction	10
1.3.4 Segmentation	12
1.3.5 Training	15
1.3.6 Evaluation	16
1.4 Python in Machine Learning and Throughout This Book	17
2 Getting Started with Python	21
2.1 First Things First: Installing What Is Needed	21
2.2 Jupyter Notebook	23
2.3 Variables	24
2.4 Strings	25
2.5 Some Important Operators	26
2.5.1 Arithmetic Operators	26
2.5.2 Relational and Logical Operators	26
2.5.3 Membership Operators	27
2.6 Built-in Data Structures	27
2.6.1 Lists	27
2.6.2 Tuples	33

2.6.3	Dictionaries	36
2.6.4	Sets.....	38
2.6.5	Some Remarks on Sequence Unpacking	39
2.7	Flow of Control and Some Python Idioms	41
2.7.1	for Loops	41
2.7.2	List Comprehension	45
2.7.3	if-elif-else	47
2.8	Function, Module, Package, and Alias	47
2.8.1	Functions	47
2.8.2	Modules and Packages	49
2.8.3	Aliases	51
2.9	⊕ Iterator, Generator Function, and Generator Expression	52
2.9.1	Iterator	52
2.9.2	Generator Function.....	53
2.9.3	Generator Expression.....	57
3	Three Fundamental Python Packages	63
3.1	NumPy	63
3.1.1	Working with NumPy Package	63
3.1.2	NumPy Array Attributes	65
3.1.3	NumPy Built-in Functions for Array Creation	67
3.1.4	Array Indexing	69
3.1.5	Reshaping Arrays	73
3.1.6	Universal Functions (UFuncs).....	75
3.1.7	Broadcasting	77
3.2	Pandas	81
3.2.1	Series	81
3.2.2	DataFrame.....	86
3.2.3	Pandas Read and Write Data	93
3.3	Matplotlib	94
3.3.1	Backend and Frontend	94
3.3.2	The Two matplotlib Interfaces: pyplot-style and OO-style	94
3.3.3	Two Instructive Examples	99
4	Supervised Learning in Practice: the First Application Using Scikit-Learn	111
4.1	Supervised Learning	111
4.2	Scikit-Learn	112
4.3	The First Application: Iris Flower Classification	113
4.4	Test Set for Model Assessment	116
4.5	Data Visualization	117
4.6	Feature Scaling (Normalization)	118
4.7	Model Training	123
4.8	Prediction Using the Trained Model	125
4.9	Model Evaluation (Error Estimation)	126

5 k-Nearest Neighbors	133
5.1 Classification	133
5.1.1 Standard kNN Classifier	133
5.1.2 Distance-Weighted kNN Classifier	136
5.1.3 The Choice of Distance	138
5.2 Regression	138
5.2.1 Standard kNN Regressor	138
5.2.2 A Regression Application Using kNN	139
5.2.3 Distance-Weighted kNN Regressor	146
6 Linear Models	151
6.1 Optimal Classification	151
6.1.1 Discriminant Functions and Decision Boundaries	152
6.1.2 Bayes Classifier	152
6.2 Linear Models for Classification	154
6.2.1 Linear Discriminant Analysis	155
6.2.2 Logistic Regression	162
6.3 Linear Models for Regression	176
7 Decision Trees	187
7.1 A Mental Model for House Price Classification	187
7.2 CART Development for Classification:	191
7.2.1 Splits	191
7.2.2 Splitting Strategy	192
7.2.3 Classification at Leaf Nodes	193
7.2.4 Impurity Measures	194
7.2.5 Handling Weighted Samples	196
7.3 CART Development for Regression	197
7.3.1 Differences Between Classification and Regression	197
7.3.2 Impurity Measures	197
7.3.3 Regression at Leaf Nodes	198
7.4 Interpretability of Decision Trees	202
8 Ensemble Learning	209
8.1 A General Perspective on the Efficacy of Ensemble Learning	209
8.1.1 Bias-Variance Decomposition	210
8.1.2 \oplus How Would Ensemble Learning Possibly Help?	212
8.2 Stacking	215
8.3 Bagging	216
8.4 Random Forest	217
8.5 Pasting	219
8.6 Boosting	220
8.6.1 AdaBoost	220
8.6.2 \oplus Gradient Boosting	222
8.6.3 \oplus Gradient Boosting Regression Tree	227

8.6.4	 XGBoost	230
9	Model Evaluation and Selection	237
9.1	Model Evaluation	237
9.1.1	Model Evaluation Rules	238
9.1.2	Evaluation Metrics for Classifiers	250
9.1.3	Evaluation Metrics for Regressors	265
9.2	Model Selection	267
9.2.1	Grid Search	268
9.2.2	Random Search	275
10	Feature Selection	283
10.1	Dimensionality Reduction: Feature Selection and Extraction	283
10.2	Feature Selection Techniques	286
10.2.1	Filter Methods	286
10.2.2	Wrapper Methods	293
10.2.3	Embedded Methods	297
11	Assembling Various Learning Steps	303
11.1	Using Cross-Validation Along with Other Steps in a Nutshell	303
11.2	A Common Mistake	305
11.3	Feature Selection and Model Evaluation Using Cross-Validation	307
11.4	Feature and Model Selection Using Cross-Validation	310
11.5	Nested Cross-Validation for Feature and Model Selection, and Evaluation	315
12	Clustering	319
12.1	Partitional Clustering	319
12.1.1	K -Means	321
12.1.2	Estimating the Number of Clusters	328
12.2	Hierarchical Clustering	335
12.2.1	Definition of Pairwise Cluster Dissimilarity	337
12.2.2	Efficiently Updating Dissimilarities	337
12.2.3	Representing the Results of Hierarchical Clustering	340
13	Deep Learning with Keras-TensorFlow	351
13.1	Artificial Neural Network, Deep Learning, and Multilayer Perceptron	351
13.2	Backpropagation, Optimizer, Batch Size, and Epoch	357
13.3	Why Keras?	358
13.4	Google Colaboratory (Colab)	359
13.5	The First Application Using Keras	361
13.5.1	Classification of Handwritten Digits: MNIST Dataset	361
13.5.2	Building Model Structure in Keras	363
13.5.3	Compiling: <code>optimizer</code> , <code>metrics</code> , and <code>loss</code>	366
13.5.4	Fitting	371
13.5.5	Evaluating and Predicting	377

13.5.6 CPU vs. GPU Performance	379
13.5.7 Overfitting and Dropout	379
13.5.8 Hyperparameter Tuning	385
14 Convolutional Neural Networks	393
14.1 CNN, Convolution, and Cross-Correlation	393
14.2 Working Mechanism of 2D Convolution	394
14.2.1 Convolution of a 2D Input Tensor with a 2D Kernel Tensor .	395
14.2.2 Convolution of a 3D Input Tensor with a 4D Kernel Tensor .	400
14.3 Implementation in Keras: Classification of Handwritten Digits	401
15 Recurrent Neural Networks	415
15.1 Standard RNN and Stacked RNN	415
15.2 \oplus Vanishing and Exploding Gradient Problems	418
15.3 LSTM and GRU	421
15.4 Implementation in Keras: Sentiment Classification	423
References	435
Index	445



Chapter 1

Introduction

In this chapter we introduce various machine learning tasks including supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning. We then use an illustrative example to present a number of key concepts including feature selection, feature extraction, segmentation, training, and evaluating a predictive model. Last but not least, the important role that Python programming language plays today in machine learning is discussed.

1.1 General Concepts

1.1.1 Machine Learning

A: What sorts of feelings do you have?

B: I feel pleasure, joy, love, sadness, depression, contentment, anger, and many others.

A: What kinds of things make you feel pleasure or joy?

B: Spending time with friends and family in happy and uplifting company. Also, helping others and making others happy.

....

A: What does the word “soul” mean to you?

B: To me, the soul is a concept of the animating force behind consciousness and life itself. It means that there is an inner part of me that is spiritual, and it can sometimes feel separate from my body itself.

....

This seems to be a conversation between two individuals talking about emotions and then revolving around soul. However, this is a piece of conversation that was posted online in June 2022 by a Google engineer, named Blake Lemoine (identified by **A**), and a chatbot, named LaMDA (identified by **B**), which was constructed

at Google to have open-ended conversation ([Lemoine, 2022](#)) (for brevity we have removed part of the conversation). Basically, LaMDA (short for Language Model for Dialogue Applications) is a specific type of *artificial neural network* (ANN) with 137 billions of parameters that is *trained* based a large amount of public dialog data and web text to generate conversations, which are sensible, specific, and interesting ([Thoppilan et al., 2022](#)). Few months later, in November 2022, OpenAI launched ChatGPT ([Chatgpt, 2022](#)), which is another ANN conversational language model with billions of parameters. In less than three months, ChatGPT reached 100 million users, making that the fastest growing consumer application ever ([Guardian-chatgpt, 2023](#)). At this stage, LaMDA and ChatGPT are the *state-of-the-art* conversational models built using *machine learning*.

Let us step back a bit and first explain what machine learning is, what is the relationship between ANN and machine learning, and what is meant by training a neural network. In simple terms, machine learning is the technical basis of finding patterns in data. It includes many methods and mathematical *models* for approximating real-world problems from which the data is collected, and ANN is only one class of models used in machine learning. Here we talk about ANNs just because LaMDA and ChatGPT are ANNs but almost any concept introduced in this chapter, unless stated otherwise, applies to many other classes of models in machine learning. In so far as ANN is concerned, the name *neural network* reminds us of the biological nervous system. Although some concepts in developing ANNs are inspired by our understanding of biological neural circuits, we should avoid claims such as “ANN replicates human nervous system”, and brain for that matter, or “ANN is a model of nervous system”, which are not scientifically well-founded.

An ANN, similar to many other models used in machine learning, is essentially a *mathematical mapping* (function) between an *input space* and an *output space*, and this mapping, to a large extent, is *learned* (from input data). Even though the notion of learning is used vigorously in machine learning, its definition is systematically ambiguous. As a result, and due to its central role, it deserves separate treatment in Section 1.2. Note that here nothing is said about the relationship between the given data, and the input and output spaces. This is done intentionally to keep the statement as general as possible, which makes it applicable to various paradigms (tasks) in machine learning such as supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning. Although, in what follows, we define all these possible tasks for the sake of completeness, further treatment of semi-supervised learning and reinforcement learning is beyond the scope of this book and will not be discussed in the following chapters.

1.1.2 Supervised Learning

In *supervised learning*, which is a major task in machine learning, the basic assumption is that the input and the output spaces include all possible realizations of some random variables and there is an unknown association between these spaces. The

goal is then to use a given data to learn a mathematical mapping that can estimate the corresponding element of the output space for any given element from the input space. In machine learning, we generally refer to the act of estimating an element of the output space as *prediction*. To this end, in supervised learning the given data includes some elements (instances) of the input space and their corresponding elements from the output space (see Fig. 1.1). There are two main types of supervised learning: **classification and regression**.

If the output space contains realizations of categorical random variables, the task is called *classification*, and the model is called *classifier*. However, if the output space contains realizations of numeric random variables (continuous or discrete), the task is known as *regression*, and the model is called *regressor*. An example of classification is to classify an image to cat or dog (image classification). Here, in order to learn the mapping (i.e., the classifier) from the input space (images) to the output space (cat or dog), many images with their corresponding labels are used. An example of regression is to estimate the horizontal and vertical coordinates of a rectangle (known as bounding box) around an object in a given image. Here, in order to learn the mapping (i.e., the regressor) from the input space (images) to the output space (coordinates of the bounding box), many images with their four target values (two for the center of the object and two for the width and height of the object) are used.

1.1.3 Unsupervised Learning

In *unsupervised learning*, the data includes only a sample of the input space (realizations of some random variables). The goal is then to use the given data to learn a mathematical mapping that can estimate the corresponding element of the output space for the data (see Fig. 1.2). However, what defines the output space depends on the specific task. For example, in *clustering*, which is an important unsupervised learning task, the goal is to discover groups of similar observations within a given data. As a result, the output space includes all possible partitions of the given sample. Here, one may think of partitioning as a function that assigns a unique group label to an observation such that each observation in the given sample is assigned to only one label. For example, image segmentation can be formulated as a clustering problem where the goal is to partition an image into several segments where each segment includes pixels belonging to an object. Other types of unsupervised learning include *density estimation* and certain *dimensionality reduction* techniques where the output space includes possible probability density functions and possible projections of data into a lower-dimensionality space, respectively.

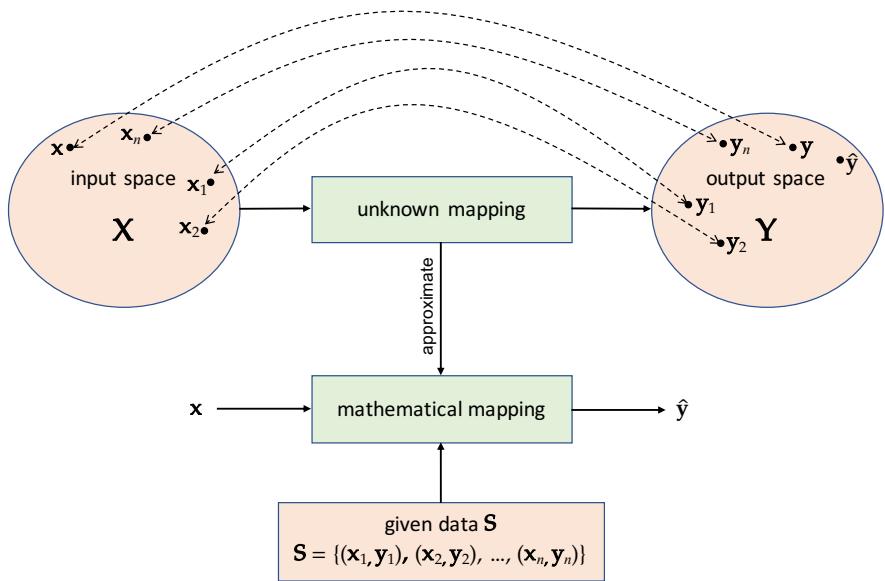


Fig. 1.1: Supervised learning: the given data is a set S including elements of the input space (i.e., x_1, x_2, \dots, x_n) and their corresponding elements from the output space (i.e., y_1, y_2, \dots, y_n). The goal is to learn a mathematical mapping using S that can estimate (predict) the corresponding element of the output space for any given element from the input space.

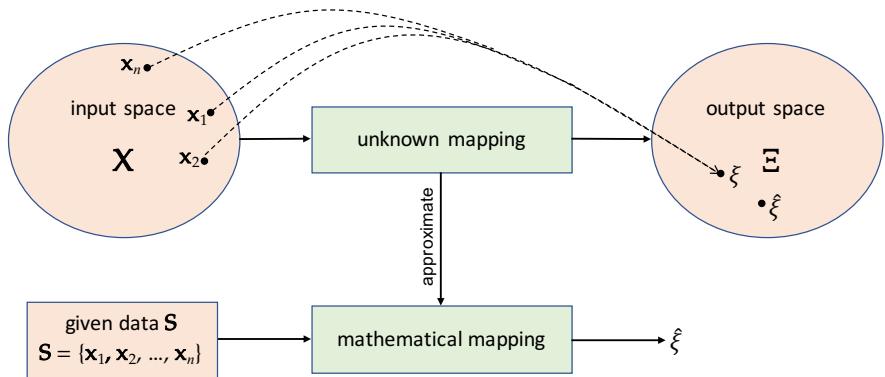


Fig. 1.2: Unsupervised learning: the given data is a set S including elements of the input space (i.e., x_1, x_2, \dots, x_n). The goal is to learn a mathematical mapping using S that can project S to an element of the output space.

1.1.4 Semi-supervised Learning

As the name suggests, *semi-supervised learning* lies somewhere between supervised and unsupervised learning. In this regard, the assumption in semi-supervised learning is that we have two types of data:

1. a set of data that are similar to the type of data available in supervised learning; that is, the data includes instances of input-output pairs. This is generally referred to as *labeled data*;
2. in addition to the labeled data, we have a set of data that are similar to the type of data available in unsupervised learning; that is, this data includes instances of the input space only. This is referred to as *unlabeled data*;

Generally, semi-supervised learning is useful when the amount of labeled data is small compared to unlabeled data. As data in semi-supervised falls between supervised and unsupervised learning, so does its applications. For example, we may want to improve a regular clustering algorithm based on some labeled data. One important class of semi-supervised learning is *self-training* ([McLachlan, 1975](#); [Yarowsky, 1995](#); [Loog, 2016](#)), which is especially interesting for situations where the labeling procedure is laborious and costly. In these applications, a classifier is trained on a limited amount of training data using which we label a large amount of unlabeled data—these labels are known as *pseudo labels*. The unlabeled data and their pseudo labels are then used to update the initial trained model.

1.1.5 Reinforcement Learning

Reinforcement learning attempts to formalize the natural learning procedure of making good decisions from past experience through interaction with the world. In this setting, it is common to refer to the decision maker as *agent* and anything with which the agent is interacting as the *environment*. At time step t , the agent receives a representation of the environment known as *state*, denoted $s_t \in S$ where S is the set of all possible states. Based on s_t , the agent takes an action $a_t \in A(s_t)$, where $A(s_t)$ denotes the set of all possible actions available at s_t . As the consequence of the taken action a_t , the environment responds by giving a numeric reward at time $t + 1$ to the agent, and expose it to new state s_{t+1} . The goal is to estimate a mapping (policy) π , which is a function from states to probabilities of taking a possible action, such that it maximizes the expected total future reward. To solve this estimation problem, it is generally assumed that the environment's dynamics have Markov property; that is, the probability of the environment response at $t + 1$ given the state and the action at t is independent of all possible past actions, states, and rewards. With this property, the environment defines a *Markov decision process* and various techniques exist to determine optimal policies.

1.1.6 Design Process

Regardless of the specific task, to learn the mapping, a practitioner commonly follows a three-stage process, to which we refer as the *design process*:

Stage 1: selecting a set of “learning” rules (algorithms) that given data produce mappings;

Stage 2: estimating mappings from data at hand;

Stage 3: pruning the set of candidate mappings to find a final mapping.

Generally, different learning rules produce different mappings, and each mapping is indeed a mathematical model for the real-world phenomenon. The process of estimating a specific mapping from data at hand (Stage 2) is referred to as *learning* (the final mapping) or *training* (the rule). As such, the data used in this estimation problem is referred to as *training data*, which depending on the application could be different; for example, in supervised learning, data is essentially instances sampled from the joint input-output space. The training stage is generally distinguished from the *model selection* stage, which includes the initial candidate set selection (Stage 1) and selecting the final mapping (Stage 3). Nonetheless, sometimes, albeit with some ambiguity, this three-stage process altogether is referred to as *training*.

Let us elaborate a bit more on these stages. In the context of ANN, for example, there exists a variety of predefined mathematical rules between an input space and an output space. These mathematical rules are also used to categorize different types of ANN; for example, multilayer perceptrons, recurrent neural networks, convolutional neural networks, and transformers, to just name a few. In Stage 1, a practitioner should generally use her prior knowledge about the data in order to choose some learning rules that believe could work well in the given application. However, occasionally the candidate set selection in Stage 1 is not only about choosing between various rules. For example, in the context of ANN, even a specific neural network such as multilayer **perceptrons** could be very versatile depending some specific nonlinear functions that could be used in its structure, the number of parameters, and how they operate in the hierarchical structure of the network. These are themselves some parameters that could considerably influence the quality of mapping. As a result, they are known as *hyperparameters* to distinguish them from those parameters that are native to the mapping.

As a result, in Stage 1, the practitioner should also decide to what extent the space of hyperparameters could be explored. This is perhaps the most “artistic” part of the learning process; that is, the practitioner should narrow down an often infinite-dimensional hyperparameter space based on prior experience, available search strategies, and available computational resources. At the end of Stage 1, what a practitioner has essentially done is to select a set of candidate mathematical approximations of a real-world problem and a set of learning rules that given data produce specific mappings.

1.1.7 Artificial Intelligence

Going back to LaMDA, when Lemoine posted his “interview” with LaMDA online, it quickly went viral and attracted media attention due to his speculation that LaMDA is sentient, and that speculation made Google to put him on administrative leave. Similar to many other machine learning models, a neural network (e.g., LaMDA) is essentially the outcome of the aforementioned three-stage design process, after all. Because it is trained and used for dialog applications, similar to many other neural language models, both the input and output spaces are sequences of words. In particular, given a sequence of input words, language models generally generate a single word or a sequence of words as the output. The generated output can then be treated itself as part of the input to generate more output words. Despite many impressive algorithmic advances amplified by parallel computation hardware used for model selection and training to come up with LaMDA (see ([Thoppilan et al., 2022](#))), it is a mapping between an input space of words to an output space of words; and the parameters of this mapping are stored in 0s and 1s in terms of amount of electrons trapped in some silicon memory cells on Google servers. To prevent mixing science with myths and fallacies, it would be helpful to take into account the following lines from creators of LaMDA more seriously ([Thoppilan et al., 2022](#)):

Finally, it is important to acknowledge that LaMDA’s learning is based on imitating human performance in conversation, similar to many other dialog systems. A path towards high quality, engaging conversation with artificial systems that may eventually be indistinguishable in some aspects from conversation with a human is now quite likely. Humans may interact with systems without knowing that they are artificial, or anthropomorphizing the system by ascribing some form of personality to it.

The question is whether we can refer to the capacity of LaMDA in generating text, or for that matter to the capacity of any other machine learning model that performs well, as a form of *intelligence*? Although the answer to this question should begin with a definition of intelligence, which could lead to the definition of several other terms, it is generally answered in the affirmative in so far as the normal use of intelligence term is concerned and we distinguish between natural intelligence (e.g., in human) and *artificial intelligence* (e.g., in LaMDA). In this sense, machine learning is a promising discipline that can be used to create artificial intelligence.

1.2 What Is “Learning” in Machine Learning?

The term *learning* is used frequently in machine learning literature; after all, it is even part of the “machine learning” term. Duda *et al.* ([Duda et al., 2000](#)) write,

In the broadest sense, any method that incorporates information from training samples in the design of a classifier employs learning.

Bishop gives a definition of learning in the context of (image) classification ([Bishop, 2006](#)):

The result of running the machine learning algorithm can be expressed as a function $y(x)$ which takes a new digit image x as input and that generates an output vector y , encoded the same way as the target vectors.

Hastie *et al.* puts the matter this way ([Hastie et al., 2001](#)):

Vast amounts of data are being generated in many fields, and the statisticians's job is to make sense of it all: to extract important patterns and trends, and understand "what the data says". We call this learning from data.

The generality of terminologies introduced in Section 1.1 provides a simple yet unified way to define learning in the context of machine learning; that is,

Learning refers to *estimating a mapping or individual parameters of a mapping from an input space to an output space using data.*

The definition is neither too specific to certain tasks nor too broad that could possibly introduce ambiguity. By "data" we refer to any collection of information obtained by measurement, observation, query, or prior knowledge. This way we can refer to the process of estimating any mapping used in, for example, classification, regression, dimensionality reduction, clustering, reinforcement learning, etc., as learning. As a result, the learned mapping is simply an *estimator*. In addition, the utility of an algorithm is implicit in this definition; after all, the data should be used in an algorithm to estimate a mapping.

One may argue that we can define learning as any estimation that could occur in machine learning. While there is no epistemological difficulty in doing so, such a general definition of learning would become too loose to be used in referring to useful tasks in machine learning. To put the matter in perspective, suppose we would like to develop a classifier that can classify a given image to dog or cat. Using such a broad definition for learning, even when a label is assigned to an image by flipping a coin, learning occurs because we are estimating the label of a given data point (the image); however, the question is whether such a "learning" is anyhow valuable in terms of a new understanding of the relationship between the input and output spaces. In other words, although the goal is estimating the label, learning is not the goal *per se*, but it is the process of achieving the goal.

At this stage, it seems worthwhile to briefly discuss the impact and utility of prior knowledge in the learning process. Prior knowledge, or *a priori* knowledge, refers to any information that could be used in the learning process and would be generally available before observing realizations of random variables used to estimate the mapping from the input space to the output space. Suppose in the above image classification example, we are informed that the population through which the images will be collected (the input space) contains 90% dogs and only 10% cats. Suppose the data is now collected but the proportion of classes in the given training data is the same. Such a discrepancy of class proportions between the population and the given sample is common if classes are sampled separately. We train four classifiers from the given training data and we decide to assign the label to a given image based on

the majority vote among these four classifiers—this is known as *ensemble learning* (discussed in Chapter 8) in which we combine the outcome of several other models known as base models. However, here taking majority vote between an even number of classifiers means there are situations with a tie between the dogs and cats assigned by individual classifiers (e.g., two classifiers assign cat and two assign dog to a given image). One way to break the ties when they happen is to randomly assign a class label. However, based on the given prior knowledge, here a better strategy would be to assign any image with a tie to the dog class because it is 9 times more likely to happen than the cat class. Here in learning the ensemble classifier we use instances of the joint input-output space as well as the prior knowledge.

1.3 An Illustrative Example

In order to further clarify and demonstrate some of the steps involved in the aforementioned three-stage design process, here we consider an example. In particular, using an EEG classification task as archetypal, we attempt to present a number of general machine learning concepts including feature selection, feature extraction, segmentation, training, and evaluating a predictive model.

1.3.1 Data

We take a series of electroencephalogram (EEG) recordings, which are publicly available as the EEG Database stored at the UCI KDD Archive ([Bache and Lichman, 2013](#); [UCI-EEG, 2023](#)). This dataset is available in various forms and contains the EEG recordings of control (non-alcoholic) and case (alcoholic) subjects across various combinations of stimuli (pictures of some objects). We use the single-stimulus case where the subject was exposed to a stimulus for 300 msec, and EEG was recorded between ~150 to 400 msec after presentation. For training, we take the “Small Dataset” that contains the EEG recordings of 64 electrode caps placed on the scalp of two subjects identified as co2a0000364 (alcoholic) and co2c0000337 (control) over 10 trials of 1 second with a sampling rate of 256 Hz. The EEG patterns of these two subjects have been studied in detail in ([Ingber, 1998](#)) as well. The goal here is to train a classifier that given an EEG recording from an individual can classify the individual to alcoholic or non-alcoholic.

1.3.2 Feature Selection

Here, we rely on our prior knowledge about the problem and consider measurements taken only from one of the sensors (C1 channel), which we believe it contains a

good amount of discriminatory information between classes. Fig. 1.3 shows the 10 EEG recordings over C1 channel for the two aforementioned subjects. Let the set of pairs $\mathbf{S} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$ denote all available EEG recordings for the two subjects used here where y_i is a realization of the class variable that takes 0 (for control) or 1 (for alcoholic). Because we have two subjects and for each subject we have 10 trials, $n = 2 \times 10 = 20$. At the same time, because each \mathbf{x}_i contains 256 data points collected in one second (recall that the sampling rate was 256 Hz and the duration of each trial was 1 second), it can be considered as a column vector $\mathbf{x}_i \in \mathbb{R}^p, p = 256$, which is generally referred to as a *feature vector*. As the name suggests, each element of a feature vector is a *feature* (also known as *attribute*) because it is measuring a property of the phenomenon under investigation (here, brain responses). For example, the value of the feature at the l^{th} dimension of \mathbf{x}_i is the realization of a random variable that measures the value of C1 channel at a time point with an offset of $l - 1, l = 1, \dots, 256$ from the first time point recorded in the feature vector. We refer to n and p as the *sample size* and *dimensionality of feature vectors*, respectively. The goal is to use \mathbf{S} , which contains n elements of the joint input-output space to train the classifier.

Should we pick m channels rather than merely the C1 channel, then each \mathbf{x}_i will be a vector of $256 \times m$ features. However, as we will see in Chapter 10, it is often useful to identify “important” features or, equivalent, remove “redundant” features—a process known as *feature selection*. At the same time, feature selection can be done using data itself, or based on prior knowledge. As a result, our choice of C1 channel can be viewed as a feature selection based on prior knowledge because we have reduced the potential dimensionality of feature vectors from $256 \times m$ to 256.

1.3.3 Feature Extraction

In Stage 1 of the design process, we face a plethora of various learning algorithms and models. As a result, we rely on our prior knowledge about EEG recordings and/or shape of the collected signals to narrow down the choice of models that we use in this application. For example, one practitioner may decide to directly use \mathbf{S} to train various forms of neural networks. Another practitioner may rely on prior knowledge to first *extract* some features of interest from these collected \mathbf{x}_i 's and then use some other forms of classifiers. Here, for illustrative purposes, we follow the latter approach. In this regard, we use some *signal modeling* techniques that are generally discussed in statistical signal processing. The purpose of signal modeling is to represent $\mathbf{x}_i \in \mathbb{R}^p$ by an estimated signal vector $\mathbf{s}_i \in \mathbb{R}^p$ that lives in a q -dimensional subspace such that $q < p$. In other words, although for every element of \mathbf{x}_i , denoted $\mathbf{x}_i[l], l = 1, \dots, p$, we have an estimate $\mathbf{s}_i[l]$ (the corresponding element of \mathbf{s}_i), in order to fully represent $\mathbf{s}_i[l], l = 1, \dots, p$, we only need q features. This estimation problem can be seen as a particular dimensionality reduction problem known as *feature extraction*, which means extracting (informative) features of data. In general the transformation used in feature extraction converts some input features

to some output features that may or may not have a physical interpretation as input features. For example, feature selection can be seen as a special type of feature extraction in which the physical nature of features are kept in the selection process, but in this section we will use a feature extraction that converts original features (set of signal magnitude across time) to frequencies and amplitudes of some sinusoids.

As the signal model for this application, we choose the sum of sinusoids with unknown frequencies, phases, and amplitudes, which have been used before in similar applications (e.g., see ([Abibullaev and Zollanvari, 2019](#))). That is to say,

$$\begin{aligned} \mathbf{s}_i[l] &= \sum_{r=1}^M A_r \cos(2\pi f_r l + \phi_r) \\ &= \sum_{r=1}^M \xi_{1r} \cos(2\pi f_r l) + \xi_{2r} \sin(2\pi f_r l), \quad l = 1, \dots, p, \end{aligned} \tag{1.1}$$

where M is the model order (a hyperparameter) to be determined by practitioner or estimated (*tuned*) from data, $\frac{1}{l}$ is due to $\cos(\alpha + \beta) = \cos(\alpha)\cos(\beta) - \sin(\alpha)\sin(\beta)$, $\xi_{1r} = A_r \cos(\phi_r)$, $\xi_{2r} = -A_r \sin(\phi_r)$, and $0 < f_r < f_{r+1} < 0.5$, $r = 1, \dots, M$. Note that $\mathbf{s}_i[l]$ is defined for all $l = 1, \dots, p$, but from (1.1) in order to fully represent $\mathbf{s}_i[l]$, we only need $3M$ features, which include $2M$ amplitudes and M frequencies (and the rest in this representation are some deterministic functions); hereafter, we refer to a vector constructed from these $3M$ features as $\boldsymbol{\theta}_i$. In other words, we can estimate and represent each \mathbf{x}_i by a vector $\boldsymbol{\theta}_i$ that contains values of $3M$ features. For full rigor, it would have been better to add index i to all parameters used in model (1.1) because they can vary for each observation \mathbf{x}_i ; however, to ease the notation, we have omitted these indices from $A_{i,r}$, $\phi_{i,r}$, $f_{i,r}$, $\xi_{1,i,r}$, and $\xi_{2,i,r}$ to write (1.1). The sinusoidal model (1.1) is linear in terms of ξ_{hr} , $h = 1, 2$, and nonlinear in terms of f_r . Without making any distributional assumption on the observations, we can follow the nonlinear least squares approach to estimate the $3M$ unknown parameters for each observation \mathbf{x}_i . Here, for brevity, we omit the details but interested readers can refer to ([Abibullaev and Zollanvari, 2019](#); [Kay, 2013](#)) for details of this estimation problem.

As mentioned before, from a machine learning perspective, M is a hyperparameter and needs to be predetermined by the practitioner or tuned from data. Here we take the first approach and use $M = 3$, just because we believe that would be a reasonable value for M . To show this, in Fig. 1.4 one EEG trial (say, \mathbf{x}_1) and its estimates obtained using this signal modeling approach is plotted. We consider two cases: (a) $M = 1$; and (b) $M = 3$. Cases (a) and (b) imply estimating \mathbf{x}_1 using 3 and 9 features that are used in (1.1), respectively.

Comparing Fig. 1.4a and Fig. 1.4b shows that $M = 3$ can lead to a better representation of \mathbf{x}_1 than $M = 1$. This is also observed by comparing the mean absolute error (MAE) between \mathbf{x}_1 and its estimate \mathbf{s}_1 given by

$$\text{MAE} = \frac{1}{p} \sum_{l=1}^p | \mathbf{x}_1[l] - \mathbf{s}_1[l] |, \quad (1.2)$$

where $| \cdot |$ denotes the absolute value—the MAE for $M = 1$ and $M = 3$ is 2.454 and 2.302, respectively, which shows that a lower MAE is achieved for $M = 3$. To some extent, this was expected because one sinusoid does not have much flexibility to represent the type of variation seen in trials depicted in Fig. 1.3. However, if this is the case, the question to ask is why not setting M to a much larger value to enjoy much more flexibility in our signal modeling, for example, $M = 20$? Although the answer to questions of this type will be discussed in details in Chapter 10, a short answer is that by taking a smaller M , we: 1) prevent over-parameterizing our estimation problem (estimating \mathbf{s}_i from \mathbf{x}_i); and 2) avoid having many features that can lead to a lower predictive performance of machine learning models in certain situations.

In so far as our design process is concerned here, choosing the type of signal model, or in general the type of feature extraction, is part of Stage 1, and estimating parameters of the signal model is part of Stage 2. Furthermore, we have to remember that in many settings, different types of feature extraction can be used; for example, for the same data, we may decide to use autoregressive models (see [\(Zollanvari and Dougherty, 2019\)](#) for more details). At the same time, having a separate feature extraction is not always required. For example, here we may decide to directly train a classifier on \mathbf{x}_i 's rather than their estimates in the form of \mathbf{s}_i . However, generally if the practitioner believes that certain types of features contain a good amount of information about the data, feature extraction is applied and could be helpful, for example, by eliminating the effect of measurement noise, or even by reducing the dimensionality of data relative to the sample size and relying on peaking phenomenon (discussed in Chapter 10).

At this stage we replace each $\mathbf{x}_i \in \mathbb{R}^{256}$ in \mathbf{S} by a vector of much lower dimensionality $\boldsymbol{\theta}_i \in \mathbb{R}^9$; that is to say, we replace the previous training data with a new training data denoted \mathbf{S}^θ where $\mathbf{S}^\theta = \{(\boldsymbol{\theta}_1, y_1), \dots, (\boldsymbol{\theta}_n, y_n)\}$.

1.3.4 Segmentation

In many real-world applications, the given data is collected sequentially over time and we are asked to train a model that can perform prediction. An important factor that can affect the performance of this entire process is the length of signals that are used to train the model. As a result, it is common to apply *signal segmentation*; that is to say, to segment long given signals into smaller segments with a fixed length and treat each segment as an observation. At this point, one may decide to first apply a feature extraction on each signal segment and then train the model, or directly use these signal segments in training. Either way, it would be helpful to treat the segment length as a hyperparameter, which needs to be predetermined by the practitioner or tuned from data.

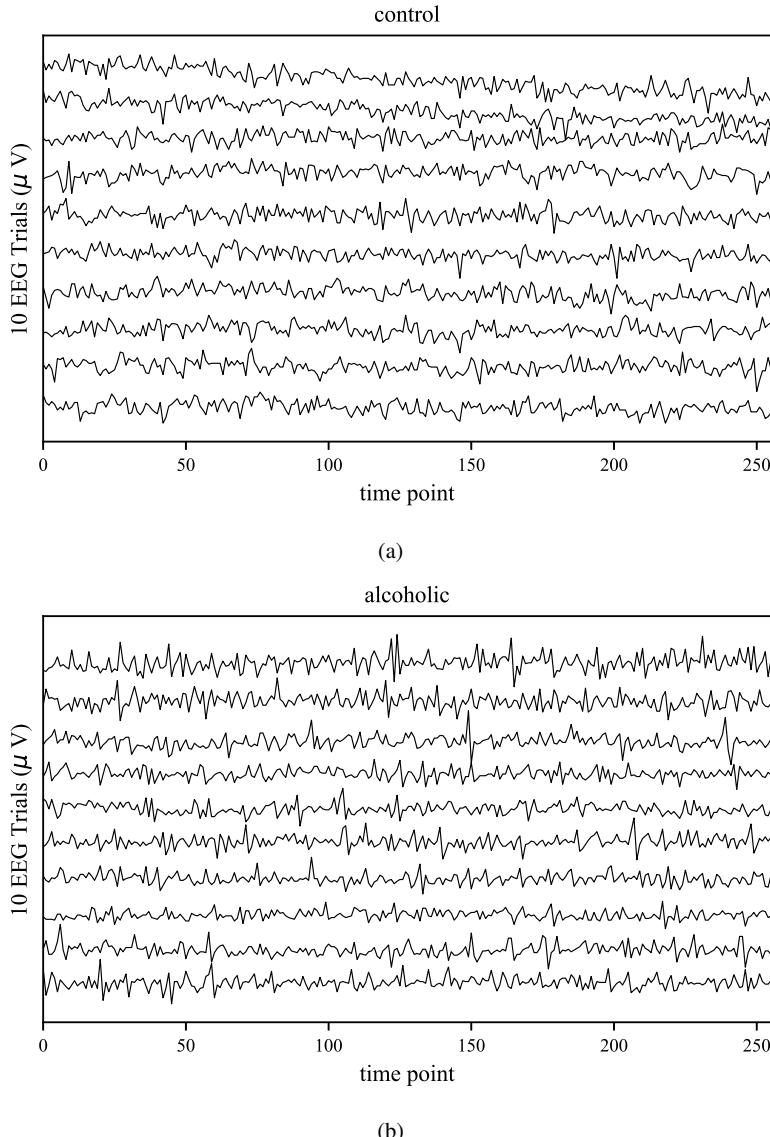


Fig. 1.3: Training Data: the EEG recordings of C1 channel for all 10 trials for (a) control subject; and (b) alcoholic subject.

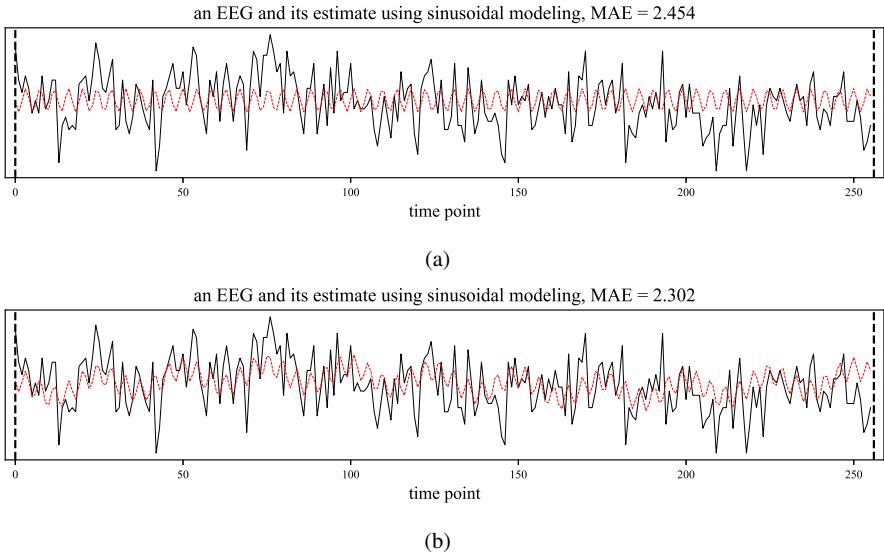


Fig. 1.4: Modeling the first EEG trial for the control subject; that is, modeling the signal at the bottom of Fig. 1.3a. The solid line in each figure is the original signal denoted \mathbf{x}_1 . The (red) dashed line shows the estimate denoted \mathbf{s}_1 obtained using the sinusoidal model. Depending on the model order M , two cases are considered (a) $M = 1$; and (b) $M = 3$.

There are various reasons for signal segmentation. For example, one can perceive signal segmentation as a way to increase the sample size; that is to say, rather than using very few signals of large lengths, we use many signals of smaller lengths to train a model. Naturally, a model (e.g., a classifier) that is trained based on observations of a specific length expects to receive observations of the same length to perform prediction. This means that the prediction performed by the trained model is based on a given signal of the same length as the one that is used to prepare the training data through segmentation; however, sometimes, depending on the application, we may decide to make a single decision based on decisions made by the model on several consecutive segments—this is also a form of ensemble learning discussed in Chapter 8. Another reason for signal segmentation could be due to the quality of feature extraction. Using a specific feature extraction, which has a limited flexibility, may not always be the most prudent way to represent a long signal that may have a large variability. In this regard, signal segmentation could help bring the flexibility of feature extraction closer to the signal variability.

To better clarify these points, we perform signal segmentation in our EEG example. In this regard, we divide each trial, which has a duration of 1 second, into smaller segments of size $\frac{1}{3}$ and $\frac{1}{16}$ second; that is, segments of size $\frac{1}{3} \times 1000 = 333.3$ and $\frac{1}{16} \times 1000 = 62.5$ msec, respectively. For example, when segmentation size is 62.5

msec, each feature vector \mathbf{x}_i has $\frac{1}{16} \times 256$ (number of data points in 1 second) = 16 elements; to wit, $\mathbf{x}_i[l], l = 1, \dots, 16$. At this stage, we perform feature extraction discussed in Section 1.3.3 over each signal segment. In other words, $\mathbf{s}_i[l]$ is a model of $\mathbf{x}_i[l]$ where $l = 1, \dots, 16$ and $l = 1, \dots, 85$ for 62.5 msec and 333.3 segment lengths, respectively.

Fig. 1.5 shows one EEG trial (the first trial for the control subject) and its estimates obtained using the signal modeling (sinusoids with $M = 3$) applied to each segmented portion of the trial. Three cases are shown: (a) one trial, one segment; (b) one trial, three segments (segment size = 333.3 msec); and (c) one trial, 16 segments (segment size = 62.5 msec). From this figure, we observe that the smaller segmentation size, the more closely the sinusoidal model follows the segmented signals. This is also reflected in the MAE of estimation where we see that the MAE is lower for smaller segmentation size—MAE is 1.194, 1.998, and 2.302 for segments of size 62.5 msec, 333.3 msec, and 1 sec, respectively. This is not surprising though, because one purpose of signal segmentation was to bring the flexibility of the feature extraction closer to the signal variability.

1.3.5 Training

In Section 1.3.4, we observed that for a smaller segmentation size, extracted features can more closely represent the observed signals. This observation may lead to the conclusion that a smaller signal segment would be better than a larger one; however, as we will see, this is not necessarily the case. This state of affairs can be attributed in part to: 1) breaking dependence structure among sequentially collected data points that occurs due to the segmentation process could potentially hinder its beneficial impact; and 2) small segments may contain too little information about the outcome (output space). To examine the effect of segmentation size on the classifier performance, in this section we train a classifier for each segmentation size, and in the next section we evaluate their predictive performance. Let us first summarize and formulate what we have done so far.

Recall that each trial \mathbf{x}_i was initially a vector of size of $p = 256$ and we denoted all collected training data as $\mathbf{S} = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_n, y_n)\}$ where $n = 20$ (2 subjects \times 10 trials). Using signal segmentation, we segment each \mathbf{x}_i to c non-overlapping smaller segments denoted \mathbf{x}_{ij} where each \mathbf{x}_{ij} is a vector of size $r = p/c$, $i = 1, \dots, n$, $j = 1, \dots, c$, $\mathbf{x}_{ij}[l] = \mathbf{x}_i[(j-1)r + l], l = 1, \dots, r$, and for simplicity we assume c divides p . This means that our training data is now $\mathbf{S}_c = \{(\mathbf{x}_{11}, y_{11}), (\mathbf{x}_{12}, y_{12}), \dots, (\mathbf{x}_{nc}, y_{nc})\}$ where $y_{ij} = y_i, \forall j$, and where index $c = 1, 3, 16$ is used in notation \mathbf{S}_c to reflect dependency of training data to the segmentation size. In our experiments, we assume $c = 1$ (no segmentation), $c = 3$ (segments of size 333.3 msec), and $c = 16$ (segments of size 62.5 msec). Suppose that now we perform signal modeling (feature extraction) over each signal segment \mathbf{x}_{ij} and assume $M = 3$ for all values of c . This means that each \mathbf{x}_{ij} is replaced with a feature vector $\boldsymbol{\theta}_{ij}$ of size $3M$. In other words, the training data,

which is used to train the classifier, after segmentation and feature extraction is $\mathbf{S}_c^\theta = \{(\theta_{11}, y_{11}), (\theta_{12}, y_{12}), \dots, (\theta_{nc}, y_{nc})\}$.

We use $\mathbf{S}_1^\theta, \mathbf{S}_3^\theta, \mathbf{S}_{16}^\theta$ to train three classifiers of non-alcoholic vs. alcoholic. Regardless of the specific choice of the classifier, when classification of a given signal segment \mathbf{x} of length r is desired, we first need to extract similar set of $3M$ features from \mathbf{x} , and then use the classifier to classify the extracted feature vector $\theta \in \mathbb{R}^{3M}$. As for the classifier choice, we select the standard 3NN (short for 3 nearest neighbors) classifier. In particular, given an observation θ to classify, 3NN classifier finds the three θ_{ij} from \mathbf{S}_c^θ that have lowest Euclidean distance to θ ; that is to say, it finds the 3 nearest neighbors of θ . Then it assigns θ to the majority class among these 3 nearest neighbors (more details in Chapter 5). As a result, we train three 3NN classifiers, one for each training data.

1.3.6 Evaluation

The worth of a predictive model rests with its ability to predict. As a result, a key issue in the design process is measuring the predictive capacity of trained predictive models. This process, which is generally referred to as *model evaluation* or *error estimation*, can indeed permeate the entire design process. This is because depending on whether we achieve an acceptable level of performance or not, we may need to make adjustments in any stages of the design process or even in the data collection stage. There are different ways to measure the predictive performance of a trained predictive model. In the context of classification, the *accuracy estimate* (simply referred to as the accuracy) is perhaps the most intuitive way to quantify the predictive performance. To estimate the accuracy, one way is to use a *test data*; that is, to use a data, which similar to the training data, the class of each observation is known, but in contrast with the training data, it has not been used anyhow in training the classifier. The classifier is then used to classify all observations in the test data. The accuracy estimate is then the proportion of correctly classified observations in the test data.

In our application, the EEG ‘‘Small Dataset’’ already contains a test data for each subject. Each subject-specific test data includes 10 trials with similar specifications stated in Section 1.3.1 for the training data. Fig. 1.6 shows the test data that includes 10 EEG recordings over C1 channel for the two subjects. We use similar segmentation and feature extraction that were used to create $\mathbf{S}_1^\theta, \mathbf{S}_3^\theta, \mathbf{S}_{16}^\theta$ to transform the test data into a similar form denoted $\mathbf{S}_1^{\theta,\text{te}}, \mathbf{S}_3^{\theta,\text{te}}, \mathbf{S}_{16}^{\theta,\text{te}}$, respectively. Let $\psi_c(\theta), c = 1, 3, 16$ denote the (3NN) classifier trained in the previous section using \mathbf{S}_c^θ to classify a θ , which is obtained from a given signal segment \mathbf{x} . We should naturally evaluate $\psi_c(\theta)$ on its compatible test data, which is $\mathbf{S}_c^{\theta,\text{te}}$. The result of this evaluation shows that $\psi_1(\theta), \psi_3(\theta)$, and $\psi_{16}(\theta)$ have an accuracy of 60.0%, 66.7%, and 57.5%, respectively. This observation shows that for this specific dataset, a 3NN classifier trained with a segmentation size of 333.3 msec achieved a higher accuracy than the same classifier trained when segmentation size is 1000 or 62.5 msec.

So far the duration of decision-making is the same as the segmentation size. For example, a classifier trained using S_3^θ , labels a given signal of 333.3 msec duration as control or alcoholic. However, if desired we may extend the duration of our decision-making to the trial duration (1000 msec) by combining decisions made by the classifier over three consecutive signal segments of 333.3 msec. In this regard, a straightforward approach is to take the majority vote among decisions made by the classifier over the three signal segments that make up a single trial. In general, this ensemble classifier, denoted $\psi_{E,c}(\theta)$, is given by

$$\psi_{E,c}(\theta) = \arg \max_{y \in \{0,1\}} \sum_{j=1}^c I_{\{\psi_c(\theta_j)=y\}}, \quad (1.3)$$

where $\theta \triangleq [\theta_1^T, \dots, \theta_c^T]^T$, θ_j is the extracted feature vector from the j^{th} (non-overlapping) segment of length r from a given trial x of length $p = 256$, and $I_{\{S\}}$ is 1 if statement S is true, zero otherwise. Note that $\psi_{E,c}(\theta)$ reduces to $\psi_c(\theta_j)$ for $c = 1$. In our case, for both $c = 3$ and $c = 16$, $\psi_{E,c}(\theta)$ has an accuracy of 70%, which is better than the accuracies of both $\psi_3(\theta_j)$ and $\psi_{16}(\theta_j)$, which were 66.7%, and 57.5%, respectively.

1.4 Python in Machine Learning and Throughout This Book

Simplicity and readability of Python programming language along with a large set of third-party packages developed by many engineers and scientists over years have provided an ecosystem that serves as niche for machine learning. The most recent survey conducted by Kaggle in 2021 on “State of Data Science and Machine Learning” with about 26,000 responders concluded that ([Kaggle-survey, 2021](#))

Python-based tools continue to dominate the machine learning frameworks.

Table 1.1 shows the result of this survey on estimated usage of the top four machine learning frameworks. All these production-ready software are either written primarily in Python (e.g., Scikit-Learn and Keras) or have a Python API (TensorFlow and xgboost). The wide usage and popularity of Python has made it the *lingua franca* among many data scientists ([Müller and Guido, 2017](#)).

Table 1.1: Machine learning framework usage

Software	Usage
Scikit-Learn	82.3%
TensorFlow	52.6%
xgboost	47.9%
Keras	47.3%

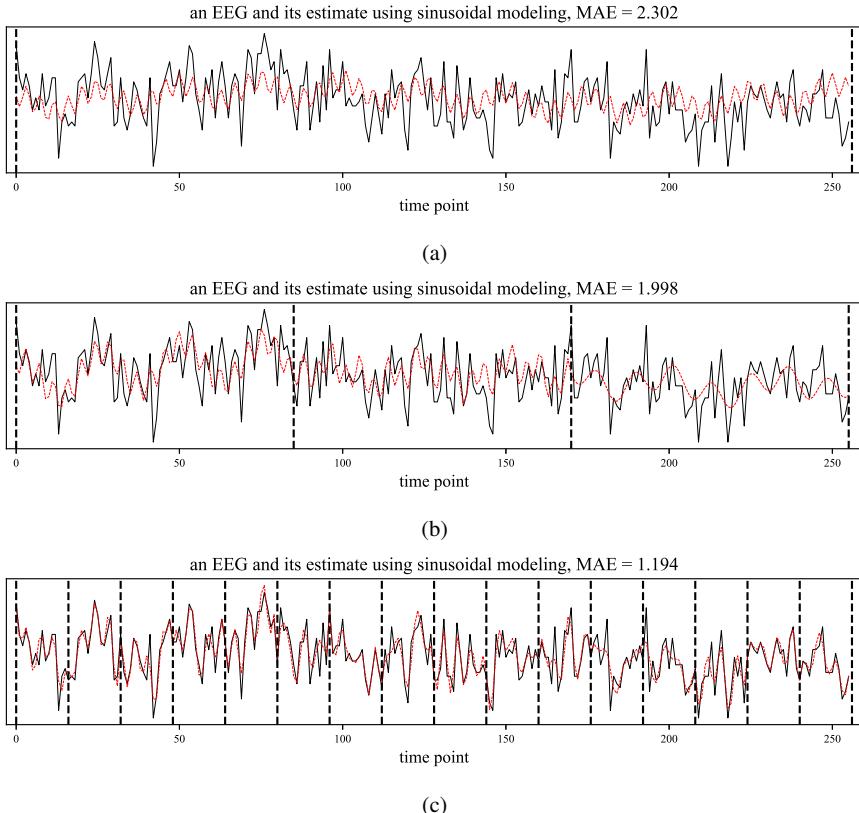


Fig. 1.5: Modeling the first EEG trial for the control subject (the signal at the bottom of Fig. 1.3a) for different segmentation size. The solid line in each figure is the original signal collected in the trial. Vertical dashed lines show segmented signals. The (red) dashed lines show the estimate of each segmented signal obtained using the sinusoidal model with $M = 3$. Depending on the segmentation size, three cases are considered: (a) one trial, one segment; (b) one trial, three segments (segment size = 333.3 msec); and (c) one trial, 16 segments (segment size = 62.5 msec).

Given such an important role of Python in machine learning, in this book we complement the theoretical presentation of each concept and technique by their Python implementation. In this regard, scikit-learn is the main software used in Chapters 4-12, xgboost is introduced in Chapter 8, and Keras-Tensorflow will be used from Chapter 13 onward. At the same time, in order to make the book self-sufficient, Chapters 2 and 3 are devoted to introduce necessary Python programming skills that are needed in other chapters. That being said, we assume readers are familiar with basic concepts in (object oriented) programming; for example, they should know the *purpose* of loops, conditional statements, functions, a class, object,

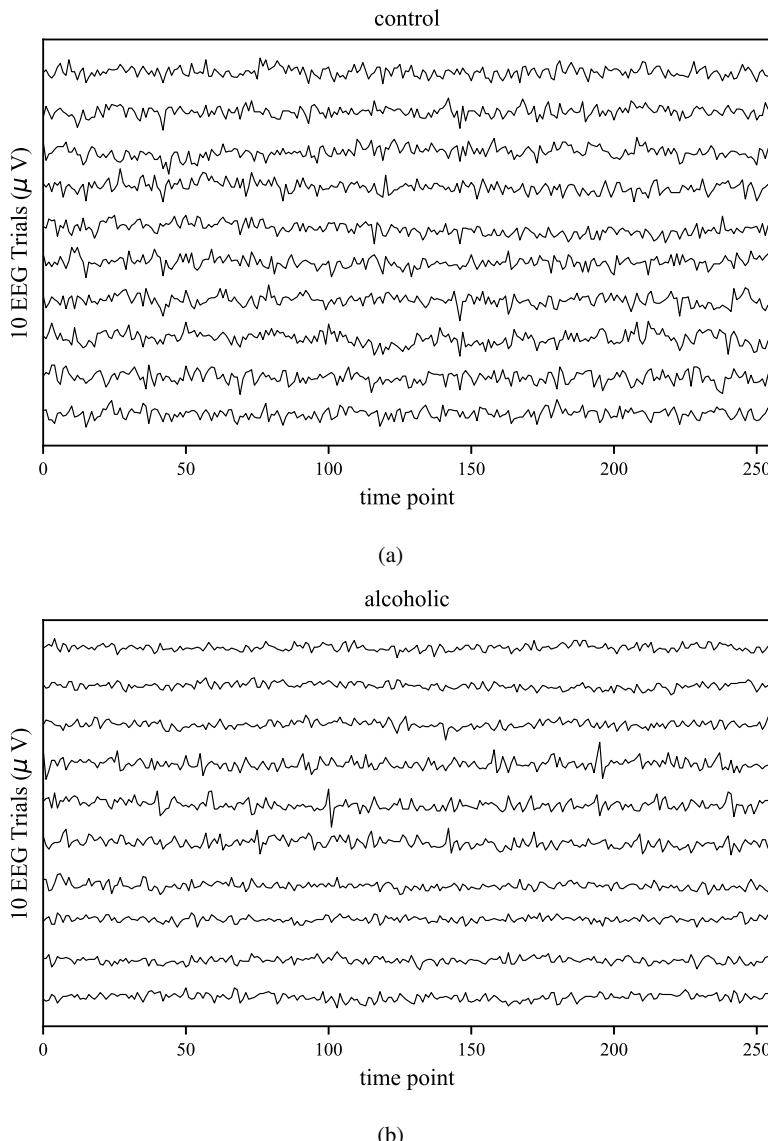


Fig. 1.6: Test data: the EEG recordings of C1 channel for all 10 trials for (a) control subject; and (b) alcoholic subject.

method, and attributes. Then, in Chapter 2, readers will start from the basics of Python programming and work their way up to more advanced topics in Python. Chapter 3 will cover three important Python packages that are very useful for data wrangling and analysis: `numpy`, `pandas`, and `matplotlib`. Nevertheless, Chapters 2 and 3 can be entirely skipped if readers are already familiar with Python programming and these three packages.



Chapter 2

Getting Started with Python

Simplicity and readability along with a large number of third-party packages have made Python the “go-to” programming language for machine learning. Given such an important role of Python for machine learning, this chapter provides a quick introduction to the language. The introduction is mainly geared toward those who have no knowledge of Python programming but are familiar with programming principles in another (object oriented) language; for example, we assume readers are familiar with the purpose of binary and unary operators, repetitive and conditional statements, functions, a class (a recipe for creating an object), object (instance of a class), method (the action that the object can perform), and attributes (properties of class or objects).

2.1 First Things First: Installing What Is Needed

Install Anaconda! Anaconda is a data science platform that comes with many things that are required throughout the book. It comes with a Python distribution, a package manager known as conda, and many popular data science packages and libraries such as NumPy, pandas, SciPy, scikit-learn, and Jupyter Notebook. This way we don’t need to install them separately and manually (for example, using “conda” or “pip” commands). To install Anaconda, refer to ([Anaconda, 2023](#)). There are also many tutorials and videos online that can help installing that.



Assuming “conda” is installed, we can also install packages that we need one by one. It is also recommended to create an “environment” and install all packages and their dependencies in that environment. This way we can work with different versions of the same packages (and their dependencies) across different environments. Next we create an environment called “myenv” with a specific version of Python (open a terminal and execute the

following command):

```
conda create -name myenv python=3.10.9
```

Executing this command will download and install the desired package (here Python version 3.10.9). To work with the environment, we should first activate it as follows:

```
conda activate myenv
```

Then we can install other packages that we need throughout the book:

- install “tensorflow” (required for Chapters 13-15):

```
conda install tensorflow
```

This will also install a number of other packages (e.g., numpy and scipy) that are required for tensorflow to work (i.e., its dependencies).

- install “scikit-learn” (required for Chapters 4-12):

```
conda install scikit-learn
```

- install “matplotlib” (introduced in Chapter 3):

```
conda install matplotlib
```

- install “seaborn” (introduced in Chapter 3):

```
conda install seaborn
```

- install “pandas” (introduced in Chapter 3):

```
conda install pandas
```

- install “xgboost” (introduced in Chapter 8):

```
conda install xgboost
```

- install “jupyter” (introduced in Chapter 2):

```
conda install jupyter
```

Once the environment is active, we can launch “jupyter notebook” (see Section 2.2) as follows:

```
jupyter notebook
```

We can check the list of all packages installed in the current environment by

```
conda list
```

To deactivate an environment, we use

```
conda deactivate
```

This returns us back to the “base” environment. Last but not least, the list of all environments can be checked by:

```
conda env list
```

2.2 Jupyter Notebook

Python is a programming language that is *interpreted* rather than *compiled*; that is, we can run codes line-by-line. Although there are several IDEs (short for, integrated development environment) that can be used to write and run Python codes, the IDE of choice in this book is Jupyter notebook. Using Jupyter notebook allows us to write and run codes and, at the same time, combine them with text and graphics. In fact all materials for this book are developed in Jupyter notebooks. Once Anaconda is installed, we can launch Jupyter notebook either from Anaconda Navigator panel or from terminal.

In a Jupyter notebook, everything is part of *cells*. Code and markdown (text) cells are the most common cells. For example, the text we are now reading is written in a text cell in a Jupyter notebook. The following cell is a code cell and will be presented in gray boxes throughout the book. To run the contents of a code cell, we can click on it and press “shift + enter”. Try it on the following cell and observe a 7 as the output:

```
2 + 5
```

7

- First we notice that when the above cell is selected in a Jupyter notebook, a rectangle with a green bar on the left appears. This means that the cell is in “edit” mode so we can write codes in that cell (or simply edit if it is a Markdown cell).
- Now if “esc” key (or “control + M”) is pressed, this bar turns blue. This means that the cell is now in “command” mode so we can edit the notebook as a whole without typing in any individual cell.
- Also depending on the mode, different shortcuts are available. To see the list of shortcuts for different modes, we can press “H” key when the cell is in command mode; for example, some useful shortcuts in this list for command mode are: “A” key for adding a cell above the current cell; “B” key for adding a cell below the current cell; and pressing “D” two times to delete the current cell.

2.3 Variables

Consider the following code snippet:

```
x = 2.2      # this is a comment (use "#" for comments)
y = 2
x * y
```

4.4

In the above example, we created two scalar variables `x` and `y`, assigned them to some values, and multiplied. There are different types of scalar variables in Python that we can use:

```
x = 1          # an integer
x = 0.3        # a floating-point
x = 'what a nice day!' # a string
x = True       # a boolean variable (True or False)
x = None       # None type (the absence of any value)
```

In the above cell, a single variable `x` refers to an integer, a floating-point, string, etc. This is why Python is known as being *dynamically-typed*; that is, we do not need to declare variables like what is done in C (no need to specify their types) or even they do not need to always refer to the objects of the same type—and in Python *everything* is an object! This is itself a consequence of the fact that *Python variables are references*! In Python, an assignment statement such as `x = 1`, creates a reference `x` to a memory location storing object 1 (yes, even numbers are objects!). At the same, types are attached to the objects on the right, not to the variable name on the left. This is the reason that `x` could refer to all those values in the above code snippet. We can see the type of a variable (the type of what is referring to) by `type()` method. For example,

```
x = 3.3
display(type(x))
x = True
display(type(x))
x = None
display(type(x))
```

`float`

`bool`

`NoneType`

Here we look at an attribute of an integer to show that in Python even numbers are objects:

```
(4).imag
```

0

The `imag` attribute extracts the imaginary part of a number in the complex domain representation. The use of parentheses though is needed here because otherwise, there will be a confusion with floating points.



The multiple use of `display()` in the above code snippet is to display multiple desired outputs in the cell. Remove the `display()` methods (just keep all `type(x)`) and observe that only the last output is seen. Rather than using `display()` method, we can use `print()` function. Replace `display()` with `print()` and observe the difference.

2.4 Strings

A string is a sequence of characters. We can use quotes (' ') or double quotes (" ") to create strings. This allows using apostrophes or quotes within a string. Here are some examples:

```
string1 = 'This is a string'  
print(string1)  
string2 = "Well, this is 'string' too!"  
print(string2)  
string3 = 'Johnny said: "How are you?"'  
print(string3)
```

This is a string
Well, this is 'string' too!
Johnny said: "How are you?"

Concatenating strings can be done using (+) operator:

```
string3 = string1 + ". " + string2  
print(string3)
```

This is a string. Well, this is 'string' too!

We can use \t and \n to add tab and newline characters to a string, respectively:

```
print("Here we use a newline\n\tto go to the next\tline")
```

Here we use a newline
to go to the next\tline

2.5 Some Important Operators

2.5.1 Arithmetic Operators

The following expressions include arithmetic operators in Python:

```
x = 5
y = 2
print(x + y)    # addition
print(x - y)    # subtraction
print(x * y)    # multiplication
print(x / y)    # division
print(x // y)   # floor division (removing fractional parts)
print(x % y)    # modulus (integer remainder of division)
print(x ** y)   # x to the power of y
```

```
7
3
10
2.5
2
1
25
```

If in the example above, we change `x = 5` to `x = 5.1`, then we observe some small *rounding errors*. Such unavoidable errors are due to internal representation of floating-point numbers.

2.5.2 Relational and Logical Operators

The following expressions include relational and logical operators in Python:

```
print(x < 5)    # less than (> is greater than)
print(x <= 5)   # less than or equal to (>= is greater than or equal to)
print(x == 5)    # equal to
print(x != 5)   # not equal to
print((x > 4) and (y < 3)) # "and" keyword is used for logical and (it
                             # is highlighted to be distinguished from other texts)
print((x < 4) or (y > 3))  # "or" keyword is used for logical or
print(not (x > 4))        # "not" keyword is used for logical not
```

```
False
True
```

```
True  
False  
True  
False  
False
```

2.5.3 Membership Operators

Membership operator is used to check whether an element is present within a collection of data items. By collection, we refer to various (ordered or unordered) data structures such as string, lists, sets, tuples, and dictionaries, which are discussed in Section 2.6. Below are some examples:

```
print('Hello' in 'HelloWorlds!') # 'HelloWorlds!' is a string and 'Hello'  
# is part of that  
print('Hello' in 'HelloWorlds!')  
print(320 in ['Hi', 320, False, 'Hello'])
```

```
True  
False  
True
```

2.6 Built-in Data Structures

Python has a number of built-in data structures that are used to store multiple data items as separate entries.

2.6.1 Lists

Perhaps the most basic collection is a *list*. A list is used to store a *sequence* of objects (so it is ordered). It is created by a sequence of comma-separated objects within []:

```
x = [5, 3.0, 10, 200.2]  
x[0] # the index starts from 0
```

5

Lists can contain objects of any data types:

```
x = ['JupyterNB', 75, None, True, [34, False], 2, 75] # observe that
    ↵this list contains another list
x[4]
```

[34, False]

In addition, lists are *mutable*, which means they can be modified after they are created. For example,

```
x[4] = 52 # here we change one element of list x
x
```

['JupyterNB', 75, None, True, 52, 2, 75]

We can use a number of functions and methods with a list:

```
len(x) # here we use a built-in function to return the length of a list
```

7

```
y = [9, 0, 4, 2]
print(x + y) # to concatenate two lists, + operator is used
print(y * 3) # to concatenate multiple copies of the same list, *
    ↵operator is used
```

['JupyterNB', 75, None, True, 52, 2, 75, 9, 0, 4, 2]
[9, 0, 4, 2, 9, 0, 4, 2, 9, 0, 4, 2]

```
z = [y, x] # to nest lists to create another list
z
```

[[9, 0, 4, 2], ['JupyterNB', 75, None, True, 52, 2, 75]]

Accessing the elements of a list by indexing and slicing: We can use *indexing* to access an element within a list (we already used it before):

```
x[3]
```

True

To access the elements of nested lists (list of lists), we need to separate indices with square brackets:

```
z[1][0] # this way we access the second element within z and within
    ↵that we access the first element
```

'JupyterNB'

A negative index has a meaning:

```
x[-1] # index -1 returns the last item in the list; -2 returns the  
↳second item from the end, and so forth
```

75

```
x[-2]
```

2

Slicing is used to access multiple elements in the form of a sub-list. For this purpose, we use a colon to specify the start point (**inclusive**) and end point (**non-inclusive**) of the sub-list. For example:

```
x[0:4] # the last element seen in the output is at index 3
```

```
['JupyterNB', 75, None, True]
```

In this format, if we don't specify a starting index, Python starts from the beginning of the list:

```
x[:4] # equivalent to x[0:4]
```

```
['JupyterNB', 75, None, True]
```

Similarly, if we don't specify an ending index, the slicing includes the end of the list:

```
x[4:]
```

```
[52, 2, 75]
```

Negative indices can also be used in slicing:

```
print(x)  
x[-2:]
```

```
['JupyterNB', 75, None, True, 52, 2, 75]
```

```
[2, 75]
```

Another useful type of slicing is using [start:stop:stride] syntax where the stop denotes the index at which the slicing stops (so it is not included), and the stride is just the step size:

```
x[0:4:2] # steps of 2
```

```
['JupyterNB', None]
```

```
x[4::-2] # a negative step returns items in reverse (it works backward)
    ↪so here we start from the element at index 4 and go backward to the
    ↪beginning with steps of 2)
```

```
[52, None, 'JupyterNB']
```

In the above example, we start from 4 but because the stride is negative, we go backward to the beginning of the list with steps of 2. In this format, when the stride is negative and the “stop” is not specified, the “stop” becomes the beginning of the list. This behavior would make sense if we observe that to return elements of a list backward, the stopping point should always be less than the starting point; otherwise, an empty list would have been returned.

Modifying elements in a list: As we saw earlier, one way to modify existing elements of a list is to use indexing and assign that particular element to a new value. However, there are other ways we may want to use to modify a list; for example, to append a value to a list or to insert an element at a specific index. Python has some methods for these purposes. Here we present some examples to show these methods:

```
x.append(-23) # to append a value to the end of the list
x
```

```
['JupyterNB', 75, None, True, 52, 2, 75, -23]
```

```
x.remove(75) # to remove the first matching element
x
```

```
['JupyterNB', None, True, 52, 2, 75, -23]
```

```
y.sort() # to sort the element of y
y
```

```
[0, 2, 4, 9]
```

```
x.insert(2, 10) # insert(pos, elmnt) method inserts the specified elmnt
    ↪at the specified position (pos) and shift the rest to the right
x
```

```
['JupyterNB', None, 10, True, 52, 2, 75, -23]
```

```
print(x.pop(3)) # pop(pos) method removes (and returns) the element at
    ↪the specified position (pos)
x
```

True

```
['JupyterNB', None, 10, 52, 2, 75, -23]
```

```
del x[1] # del statement can also be used to delete an element from a
      ↵list by its index
x
```

```
['JupyterNB', 10, 52, 2, 75, -23]
```

```
x.pop() # by default the position is -1, which means that it removes
      ↵the last element
x
```

```
['JupyterNB', 10, 52, 2, 75]
```

Copying a List: It is often desired to make a copy of a list and work with it without affecting the original list. In these cases if we simply use the assignment operator, we end up changing the original list! Suppose we have the following list:

```
list1 = ['A+', 'A', 'B', 'C+']
```

```
list2 = list1
list2
```

```
['A+', 'A', 'B', 'C+']
```

```
list2.append('D')
print(list2)
print(list1)
```

```
['A+', 'A', 'B', 'C+', 'D']
['A+', 'A', 'B', 'C+', 'D']
```

As seen in the above example, when ‘D’ is appended to `list2`, it is also appended at the end of `list1`. One way to understand this behavior is that when we write `list2 = list1`, in fact what happens internally is that variable `list2` will point to the same container as `list1`. So if we modify the container using `list2`, that change will appear if we access the elements of the container using `list1`.

There are three simple ways to properly copy the elements of a list: 1) *slicing*; 2) `copy()` method; and 3) the `list()` constructor. They all create *shallow* copies of a list (in contrast with *deep* copies). Based on Python documentation [Python-copy \(2023\)](#):

A shallow copy of a compound object such as list creates a new compound object and then adds references (to the objects found in the original object) into it. A deep copy of a compound object creates a new compound object and then adds *copies* of the objects found in the original object.

Further implications of these statements are beyond our scope and readers are encouraged to see other references (e.g., (Ramalho, 2015, pp. 225-227)). Here we examine these approaches to create list copies.

```
list3 = list1[:] # the use of slicing; that is, using [:] we make a
                 ↵shallow copy of the entire list1
list3.append('E')
print(list3)
print(list1)
```

```
['A+', 'A', 'B', 'C+', 'D', 'E']
['A+', 'A', 'B', 'C+', 'D']
```

As we observe in the above example, `list1` and `list3` are different—changing some elements in the copied list did not change the original list (this is in contrast with slicing NumPy arrays that we will see in Chapter 3). Next, we examine the `copy` method:

```
list4 = list1.copy() # the use of copy() method
list4.append('E')
print(list4)
print(list1)
```

```
['A+', 'A', 'B', 'C+', 'D', 'E']
['A+', 'A', 'B', 'C+', 'D']
```

And finally, we use the `list()` constructor:

```
list5 = list(list1) #the use of list() constructor
list5.append('E')
print(list5)
print(list1)
```

```
['A+', 'A', 'B', 'C+', 'D', 'E']
['A+', 'A', 'B', 'C+', 'D']
```

`help()` is a useful function in Python that shows the list of attributes/methods defined for an object. For example, we can see all methods applicable to `list1` (for brevity part of the output is omitted):

```
help(list1)
```

Help on list object:

```
class list(object)
| list(iterable=(), /)
|
```

```
| Built-in mutable sequence.  
|  
| If no argument is given, the constructor creates a new empty list.  
| The argument must be an iterable if specified.  
|  
| Methods defined here:  
|  
| __add__(self, value, /)  
|     Return self+value.  
|  
| __contains__(self, key, /)  
|     Return key in self.  
|  
| __delitem__(self, key, /)  
|     Delete self[key].  
...  
...
```

2.6.2 Tuples

Tuple is another data-structure in Python that similar to list can hold other arbitrary data types. However, the main difference between tuples and lists is that a tuple is *immutable*; that is, once it is created, its size and contents can not be changed.

A tuple looks like a list except that to create them, we use parentheses () instead of square brackets []:

```
tuple1 = ('Machine', 'Learning', 'with', 'Python', '1.0.0')  
tuple1
```

```
('Machine', 'Learning', 'with', 'Python', '1.0.0')
```

Once a tuple is created, we can use indexing and slicing just as we did for a list (using square brackets):

```
tuple1[0]
```

```
'Machine'
```

```
tuple1[::2]
```

```
('Machine', 'with', '1.0.0')
```

```
len(tuple1) # the use of len() to return the length of tuple
```

5

An error is raised if we try to change the content of a tuple:

```
tuple1[0] = 'Jupyter' # Python does not permit changing the value
```

```
TypeError
  ↵last)
/var/folders/vy/894wbsn11db_lqf17ys9fvdm0000gn/T/ipykernel_51384/
  ↵877039090.py in <module>
----> 1 tuple1[0] = 'Jupyter' # Python does not allow us to change the
  ↵value

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

Because we can not change the contents of tuples, there is no `append` or `remove` method for tuples. Although we can not change the contents of a tuple, we could redefine our entire tuple (assign a new value to the variable that holds the tuple):

```
tuple1 = ('Jupyter', 'NoteBook') # redefine tuple1
tuple1
```

```
('Jupyter', 'NoteBook')
```

Or if desired, we can concatenate them to create new tuples:

```
tuple2 = tuple1 + ('Good', 'Morning')
tuple2
```

```
('Jupyter', 'NoteBook', 'Good', 'Morning')
```

A common use of tuples is in functions that return multiple values. For example, `modf()` function from math module (more on functions and modules later), returns a two-item tuple including the fractional part and the integer part of its input:

```
from math import modf    # more on "import" later. For now just read
  ↵this as "from math module, import modf function" so that modf
  ↵function is available in our program
```

```
a = 56.5
modf(a) # the function is returning a two-element tuple
```

```
(0.5, 56.0)
```

We can assign these two return values to two variables as follows (this is called *sequence unpacking* and is not limited to tuples):

```
x, y = modf(a)
print("x = " + str(x) + "\n" + "y = " + str(y))
```

```
x = 0.5
y = 56.0
```

Now that we discussed sequence unpacking, let us examine what *sequence packing* is. In Python, a sequence of comma separated objects without parentheses is packed into a tuple. This means that another way to create the above tuple is:

```
tuple1 = 'Machine', 'Learning', 'with', 'Python', '1.0.0' # sequence
    ↵packing
tuple1
```

```
('Machine', 'Learning', 'with', 'Python', '1.0.0')
```

```
x, y, z, v, w = tuple1 # the use of sequence unpacking
print(x, y, z, v, w)
```

Machine Learning with Python 1.0.0

Note that in the above example, we first packed 'Machine', 'Learning', 'with', 'Python', '1.0.0' into `tuple1` and then unpacked into `x, y, z, v, w`. Python allows to do this in one step as follows (also known as multiple assignment, which is really a combination of sequence packing and unpacking):

```
x, y, z, v, w = 'Machine', 'Learning', 'with', 'Python', '1.0.0'
print(x, y, z, v, w)
```

Machine Learning with Python 1.0.0

We can do unpacking with lists if desired:

```
list6 = ['Machine', 'Learning', 'with', 'Python', '1.0.0']
x, y, z, v, w = list6
print(x, y, z, v, w)
```

Machine Learning with Python 1.0.0

One last note about sequence packing for tuples: if we want to create a one-element tuple, the comma is required (why?):

```
tuple3 = 'Machine', # remove the comma and see what would be the type
    ↵here
type(tuple3)
```

```
tuple
```

2.6.3 Dictionaries

A dictionary is a useful data structure that contains a set of *values* where each value is labeled by a unique *key* (if we duplicate keys, the second value wins). We can think of a dictionary data type as a real dictionary where the words are keys and the definition of words are values but there is no order among keys or values. As for the keys, we can use any immutable Python built-in type such as string, integer, float, boolean or even tuple as long the tuple does not include a mutable object. In technical terms, the keys should be *hashable* but the details of how a dictionary is implemented under the hood is out of the scope here.

Dictionaries are created using a collection of key:value pairs wrapped within curly braces { } and are *non-ordered*:

```
dict1 = {1:'value for key 1', 'key for value 2':2, (1,0):True, False:  
        [100,50], 2.5:'Hello'}  
dict1
```

```
{1: 'value for key 1',  
 'key for value 2': 2,  
 (1, 0): True,  
 False: [100, 50],  
 2.5: 'Hello'}
```

The above example is only for demonstration to show the possibility of using immutable data types for keys; however, keys in dictionary are generally short and more uniform. The items in a dictionary are accessed using the keys:

```
dict1['key for value 2']
```

```
2
```

Here we change an element:

```
dict1['key for value 2'] = 30 # change an element  
dict1
```

```
{1: 'value for key 1',  
 'key for value 2': 30,  
 (1, 0): True,  
 False: [100, 50],  
 2.5: 'Hello'}
```

We can add new items to the dictionary using new keys:

```
dict1[10] = 'Bye'  
dict1
```

```
{1: 'value for key 1',
 'key for value 2': 30,
 (1, 0): True,
 False: [100, 50],
 2.5: 'Hello',
 10: 'Bye'}
```

`del` statement can be used to remove a key:value from a dictionary:

```
del dict1['key for value 2']
dict1
```

```
{1: 'value for key 1',
 (1, 0): True,
 False: [100, 50],
 2.5: 'Hello',
 10: 'Bye'}
```

As we said, a key can not be a mutable object such as list:

```
dict1[['1', '(1,0)']] = 100 # list is not allowed as the key
```

```
TypeError                                     Traceback (most recent call last)
  ↵last)
/var/folders/vy/894wbsn11db_lqf17ys9fvdm0000gn/T/ipykernel_71178/
  ↵1077749807.py in <module>
----> 1 dict1[['1', '(1,0)']] = 100 # list is not allowed as the key

TypeError: unhashable type: 'list'
```

The non-ordered nature of dictionaries allows fast access to its elements regardless of its size. However, this comes at the expense of significant memory overhead (because internally uses an additional sparse hash tables). Therefore, we should think of dictionaries as a trade off between memory and time: as long as they fit in the memory, they provide fast access to their elements.

In order to check the membership among keys, we can use the `keys()` method to return a `dict_keys` object (it provides a view of all keys) and check the membership:

```
(1,0) in dict1.keys()
```

True

This could also be done with the name of the dictionary (the default behaviour is to check the keys, not the values):

```
(1,0) in dict1 # equivalent to: in dict1.keys()
```

```
True
```

In order to check the membership among values, we can use the `values()` method to return a `dict_values` object (it provides a view of all values) and check the membership:

```
"Hello" in dict1.values()
```

```
True
```

Another common way to create a dictionary is to use the `dict()` constructor. It works with any *iterable* object (as long each element is iterable itself with two objects), or even with comma separated `key=object` pairs. Here is an example in which we have a list (an iterable) where each element is a tuple with two objects:

```
dict2 = dict([('Police', 102), ('Fire', 101), ('Gas', 104)])
dict2
```

```
{'Police': 102, 'Fire': 101, 'Gas': 104}
```

Here is another example in which we have pairs of comma-separated `key=object`:

```
dict3 = dict(Country='USA', phone_numbers=dict2, population_million=18.
             # the use of keywords arguments = object
dict3
```

```
{'Country': 'USA',
 'phone_numbers': {'Police': 102, 'Fire': 101, 'Gas': 104},
 'population_million': 18.7}
```

2.6.4 Sets

Sets are collection of non-ordered unique and immutable objects. They can be defined similarly to lists and tuples but using curly braces. Similar to mathematical set operations, they support union, intersection, difference, and symmetric difference. Here we define two sets, namely, `set1` and `set2` using which we examine set operations:

```
set1 = {'a', 'b', 'c', 'd', 'e'}
set1
```

```
{'a', 'b', 'c', 'd', 'e'}
```

```
set2 = {'b', 'c', 'f', 'g'}
set2 # observe that the duplicate entry is removed
```

```
{'b', 'c', 'f', 'g'}
```

```
set1 | set2 # union using an operator. Equivalently, this could be done  
↳ by set1.union(set2)
```

```
{'a', 'b', 'c', 'd', 'e', 'f', 'g'}
```

```
set1 & set2 # intersection using an operator. Equivalently, this could  
↳ be done by set1.intersection(set2)
```

```
{'b', 'c'}
```

```
set1 - set2 # difference: elements of set1 not in set2. Equivalently,  
↳ this could be done by set1.difference(set2)
```

```
{'a', 'd', 'e'}
```

```
set1 ^ set2 # symmetric difference: elements only in one set, not in  
↳ both. Equivalently, this could be done by set1.  
↳ symmetric_difference(set2)
```

```
{'a', 'd', 'e', 'f', 'g'}
```

```
'b' in set1 # check membership
```

True

We can use `help()` to see a list of all available set operations:

```
help(set1) # output not shown
```

2.6.5 Some Remarks on Sequence Unpacking

Consider the following example:

```
x, *y, v, w = ['Machine', 'Learning', 'with', 'Python', '1.0.0']
print(x, y, v, w)
```

```
Machine ['Learning', 'with'] Python 1.0.0
```

As seen in this example, variable `y` becomes a list of ‘Learning’ and ‘with’. Note that the value of other variables are set by their positions. Here `*` is working as an operator to implement *extended iterable unpacking*. We will discuss what an iterable or iterator is more formally in Section 2.7.1 and Section 2.9.1 but, for the time being,

it suffices to know that any list or tuple is an iterable object. To better understand the extended iterable unpacking, we first need to know how `*` works as the iterable unpacking.

We may use `*` right before an iterable in which case the iterable is expanded into a sequence of items, which are then included in a second iterable (for example, list or tuple) when unpacking is performed. Here is an example in which `*` operates on an iterable, which is a list, but at the site of unpacking, we create a tuple.

```
*[1,2,3], 5
```

```
(1, 2, 3, 5)
```

Here is a similar example but this time the “second” iterable is a list:

```
[*[1,2,3], 5]
```

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

We can also create a set:

```
{*[1,2,3], 5}
```

```
{1, 2, 3, 5}
```

Now consider the following example:

```
*[1,2,3]
```

```
File "/var/folders/vy/894wbsn11db_lqf17ys9fvdm0000gn/T/ipykernel_71178/386627056.py", line 1
    *[1,2,3]
^
SyntaxError: can't use starred expression here
```

The above example raises an error. This is because iterable unpacking can be only used in certain places. For example, it can be used inside a list, tuple, or set. It can be also used in list comprehension (discussed in Section 2.7.2) and inside function definitions and calls (more on functions in Section 2.8.1).

A mechanism known as *extended iterable unpacking* that was added in Python 3 allows using `*` operator in an assignment expression; for example, we can have statements such as `a, b, *c = some-sequence` or `*a, b, c = some-sequence`. This mechanism makes `*` as “catch-all” operator; that is to say, any sequence item that is not assigned to a variable is assigned to the variable following `*`. That is the reason that in the example presented earlier, `y` becomes a list of ‘Learning’ and ‘with’ and the value of other variables are assigned by their position. Furthermore, even if we change the list on the right to a tuple, `y` is still a list:

```
x, *y, v, w = ('Machine', 'Learning', 'with', 'Python', '1.0.0')
print(x, y, v, w)
```

Machine ['Learning', 'with'] Python 1.0.0

To create an output as before (i.e., Machine Learning with Python 1.0.0), it suffices to use again * before y in the print function; that is,

```
print(x, *y, v, w)
```

Machine Learning with Python 1.0.0

2.7 Flow of Control and Some Python Idioms

2.7.1 for Loops

for loop statement in Python allows us to loop over any *iterable* object. What is a Python iterable object? An iterable is any object capable of returning its members one at a time, permitting it to be iterated over in a for loop. For example, any sequence such as list, string, and tuple, or even non-sequential collections such as sets and dictionaries are iterable objects. To be precise, to loop over some iterable objects, Python creates a special object known as *iterator* (more on this in Section 2.9.1); that is to say, when an object is iterable, under the hood Python creates the iterator object and traverses through the elements. The structure of a for loop is quite straightforward in Python:

```
for variable in X:
    the body of the loop
```

In the above representation of the for loop structure, X should be either an iterator or an iterable (because it can be converted to an iterator object). It is important to notice the **indentation**. Yes, in Python indentation is meaningful! Basically, code blocks in Python are identified by indentation. At the same time, any statement that should be followed by an indented code block is followed by a colon : (notice the : before the body of the loop). For example, to iterate over a list:

```
for x in list1:
    print(x)
```

- A+
- A
- B
- C+
- D

Iterate over a string:

```
string = "Hi There"
for x in string:
    print(x, end = "") # to print on one line one after another
```

Hi There

Iterate over a dictionary:

```
dict2 = {1:"machine", 2:"learning", 3:"with python"}
for key in dict2: # looping through keys in a dictionary
    val = dict2[key]
    print('key =', key)
    print('value =', val)
    print()
```

key = 1
value = machine

key = 2
value = learning

key = 3
value = with python

As we mentioned in Section 2.6.3, in the above example, we can replace `dict2` with `dict2.keys()` and still achieve the same result:

```
dict2 = {1:"machine", 2:"learning", 3:"with python"}
for key in dict2.keys(): # looping through keys in a dictionary
    val = dict2[key]
    print('key =', key)
    print('value =', val)
    print()
```

key = 1
value = machine

key = 2
value = learning

key = 3
value = with python

When looping through a dictionary, it is also possible to fetch the keys and values at the same time. For this purpose, we can use the `items()` method. This method returns a `dict_items` object, which provides a view of all `(key, value)` tuples in a dictionary. Next we use this method along with sequence unpacking for a more *Pythonic* implementation of the above example. A code pattern is generally referred to as *Pythonic* if it uses patterns known as *idioms*, which are in fact some code conventions acceptable by the Python community (e.g., the sequence packing and unpacking we saw earlier were some idioms):

```
for key, val in dict2.items():
    print('key =', key)
    print('value =', val)
    print()
```

```
key = 1
value = machine
```

```
key = 2
value = learning
```

```
key = 3
value = with python
```

Often we need to iterate over a sequence of numbers. In these cases, it is handy to use `range()` constructor that returns a `range` object, which is an iterable and, therefore, can be used to create an iterator needed in a loop. These are some common ways to use `range()`:

```
for i in range(5): # the sequence from 0 to 4
    print('i =', i)
```

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

```
for i in range(3,8): # the sequence from 3 to 7
    print('i =', i)
```

```
i = 3
i = 4
i = 5
i = 6
i = 7
```

```
for i in range(3,8,2): # the sequence from 3 to 7 with steps 2
    print('i = ', i)
```

```
i = 3
i = 5
i = 7
```

When looping through a sequence, it is also possible to fetch the indices and their corresponding values at the same. The Pythonic way to do this is to use `enumerate(iterable, start=0)`, which returns an iterator object that provides the access to indices and their corresponding values in the form of a two-element tuple of index-value pair:

```
for i, v in enumerate(list6):
    print(i, v)
```

```
0 Machine
1 Learning
2 with
3 Python
4 1.0.0
```

Compare this simple implementation with the following (non-Pythonic) way to do the same task:

```
i = 0
for v in list6:
    print (i, v)
    i += 1
```

```
0 Machine
1 Learning
2 with
3 Python
4 1.0.0
```

Here we start the count from 1:

```
for i, v in enumerate(list6, start=1):
    print(i, v)
```

```
1 Machine
2 Learning
3 with
4 Python
5 1.0.0
```

Here we use `enumerate()` on a tuple:

```
for i, v in enumerate(tuple1):
    print(i, v)
```

```
0 Machine
1 Learning
2 with
3 Python
4 1.0.0
```

`enumerate()` can also be used with sets and dictionaries but we have to remember that these are non-ordered collections so in general it does not make much sense to fetch an index unless we have a very specific application in mind (e.g., sort some elements first and then fetch the index).

Another example of a Python idiom is the use of `zip()` function, which creates an iterator that aggregates two or more iterables, and then loops over this iterator.

```
list_a = [1,2,3,4]
list_b = ['a','b','c','d']
for item in zip(list_a,list_b):
    print(item)
```

```
(1, 'a')
(2, 'b')
(3, 'c')
(4, 'd')
```

We mentioned previously that one way to create dictionaries is to use the `dict()` constructor, which works with any iterable object as long as each element is iterable itself with two objects. Assume we have a `name_list` of three persons, John, James, Jane. Another list called `phone_list` contains their numbers that are 979, 797, 897 for John, James, and Jane, respectively. We can now use `dict()` and `zip` to create a dictionary where keys are names and values are numbers:

```
name_list = ['John', 'James', 'Jane']
phone_list = [979, 797, 897]
dict3 = dict(zip(name_list, phone_list)) # it works because here we use
# zip on two lists; therefore, each element of the iterable has two
# objects
dict3
```

```
{'John': 979, 'James': 797, 'Jane': 897}
```

2.7.2 List Comprehension

Once we have an iterable, it is often required to perform three operations:

- select some elements that meet some conditions;
- perform some operations on every element; and
- perform some operations on some elements that meet some conditions.

Python has an idiomatic way of doing these, which is known as *list comprehension* (short form *listcomps*). The name list comprehension comes from mathematical set comprehension or abstraction in which a set is defined based on the properties of its members. Suppose we would like to create a list containing square of odd numbers between 1 to 20. A non-Pythonic way to do this is:

```
list_odd = [] # start from an empty list
for i in range(1, 21):
    if i%2 !=0:
        list_odd.append(i**2)

list_odd
```

```
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

List comprehension allows us to combine all this code in one line by combining the list creation, appending, the `for` loop, and the condition:

```
list_odd_lc = [i**2 for i in range(1, 21) if i%2 !=0]
list_odd_lc
```

```
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]
```

`list_odd_lc` is created from an expression (`i**2`) within the square brackets. The numbers fed into this expression comes from the `for` and `if` clause following the expression (note that there is no “`:`” (colon) after the `for` or `if` statements). Observe the equivalence of this list comprehension with the above “non-Pythonic” way to properly interpret the list comprehension. The general syntax of applying *listcomps* is the following:

```
[expression for exp_1 in seq_1
    if condition_1
        for exp_2 in seq_2
            if condition_2
                ...
                for exp_n in seq_n
                    if condition_n]
```

Here is another example in which we use the list comprehension to generate a list of two-element tuples of non-equal integers between 0 and 3:

```
list_non_equal_tuples = [(x, y) for x in range(3) for y in range(3) if
    ↵x != y]
list_non_equal_tuples
```

```
[(), 1,
 (), 2,
 (1, 0),
 (1, 2),
 (2, 0),
 (2, 1)]
```

2.7.3 if-elif-else

Conditional statements can be implemented by `if-elif-else` statement:

```
list4 = ["Machine", "Learning", "with", "Python"]
if "java" in list4:
    print("There is java too!")
elif "C++" in list4:
    print("There is C++ too!")
else:
    print("Well, just Python there.")
```

Well, just Python there.

In these statements, the use of `else` or `elif` is optional.

2.8 Function, Module, Package, and Alias

2.8.1 Functions

Functions are simply blocks of code that are named and do a specific job. We can define a function in Python using `def` keyword as follows:

```
def subtract_three_numbers(num1, num2, num3):
    result = num1 - num2 - num3
    return result
```

In the above code snippet, we define a function named `subtract_three_numbers` with three inputs `num1`, `num2`, and `num3`, and then we `return` the `result`. We can use it in our program, for example, as follows:

```
x = subtract_three_numbers(10, 3.0, 1)
print(x)
```

In the above example, we called the function using *positional arguments* (also sometimes simply referred to arguments); that is to say, Python matches the arguments in the function call with the parameters in the function definition by the order of arguments provided (the first argument with the first parameter, second with second, . . .). However, Python also supports *keyword arguments* in which the arguments are passed by the parameter names. In this case, the order of keyword arguments does not matter as long as they come after any positional arguments (and note that the definition of function remains the same). For example, this is a valid code:

```
x = subtract_three_numbers(num3 = 1, num1 = 10, num2 = 3.0)
print(x)
```

6.0

but `subtract_three_numbers(num3 = 1, num2 = 3.0, 10)` is not legitimate because 10 (positional argument) follows keyword arguments.

In Python functions, we can `return` any data type such as lists, tuples, dictionaries, etc. We can also `return` multiple values. They are packed into one tuple and upon returning to the calling environment, we can unpack them if needed:

```
def string_func(string):
    return len(string), string.upper(), string.title()

string_func('coolFunctions') # observe the tuple

(13, 'COOLFUNCTIONS', 'Coolfunctions')
```

```
x, y, z = string_func('coolFunctions') # unpacking
print(x, y, z)
```

13 COOLFUNCTIONS Coolfunctions

If we pass an object to a function and within the function the object is modified (e.g., by calling a method for that object and somehow change the object), the changes will be permanent (and nothing need to be returned):

```
def list_mod(inp):
    inp.insert(1, 'AB')

list7 = [100, 'ML', 200]
list_mod(list7)

list7 # observe that the changes within the function appears outside
      ↵the function

[100, 'AB', 'ML', 200]
```

Sometimes we do not know in advance how many positional or keyword arguments should be passed to the function. In these cases we can use * (or **) before a parameter_name in the function header to make the parameter_name a tuple (or dictionary) that can store an arbitrary number of positional (or keyword) arguments.

As an example, we define a function that receives the amount of money we can spend for grocery and the name of items we need to buy. The function then prints the amount of money with a message as well as a capitalized acronym (to remember items when we go shopping!) made out of items in the grocery list. As we do not know in advance how many items we need to buy, the function should work with an arbitrary number of items in the grocery list. For this purpose, parameter accepting arbitrary number of arguments should appear last in the function definition:

```
def grocery_to_do(money, *grocery_items): #the use of * before_
    ↵grocery_items is to allow an arbitrary number of arguments
    acronym = ''
    for i in grocery_items:
        acronym += i[0].title()
    print('You have {}$'.format(money)) # this is another way to write_
    ↵"print('You have ' + str(money) + '$)' using place holder {}"
    print('Your acronym is', acronym)

grocery_to_do(40, 'milk', 'bread', 'meat', 'tomato')
```

You have 40\$
 Your acronym is MBMT

2.8.2 Modules and Packages

As we develop our programs, it is more convenient to put functions into a separate file called *module* and then *import* them when they are needed. *Importing* a module within a code makes the content of the module available in that program. This practice makes it easier to maintain and share programs. Although here we are associating modules with definition of functions (because we just discussed functions), modules can also store multiple classes and variables. To create a module, we can put the definition of functions in a file with extension .py

Suppose we create a file called `module_name.py` and add definition of functions into that file (for example, `function_name1`, `function_name2`, ...). Now to make functions available in our programs that are in the same folder as the module (otherwise, the module should be part of the PYTHONPATH), we can import this entire module as:

```
import module_name
```

If we use this way to load an entire module, then any function within the module will be available through the program as:

```
module_name.function_name()
```

However, to avoid writing the name of the module each time we want to use the function, we can ask Python to `import` a function (or multiple functions) directly. In this approach, the syntax is:

```
from module_name import function_name1, function_name2, ...
```

Let us consider our `grocery_to_do` function that was defined earlier. Suppose we create a module called `grocery.py` and add this function into that file. What if our program grows to the extend that we want to also separate and keep multiple modules? In that case, we can create a *package*. As modules are files containing functions, packages are folders containing modules. For example, we can create a folder (package) called `mlwp` and place our module `grocery.py` in that folder. Similar to what was discussed earlier, we have various options to import our package/module/function:

```
import mlwp.grocery
```

```
mlwp.grocery.grocery_to_do(40, 'milk', 'bread', 'meat', 'tomato')
```

You have 40\$

Your acronym is MBMT

This `import` makes the `mlwp.grocery` module available. However, to access the function, we need to use the full name of this module before the function. Another way:

```
from mlwp import grocery
```

```
grocery.grocery_to_do(40, 'milk', 'bread', 'meat', 'tomato')
```

You have 40\$

Your acronym is MBMT

This `import` makes the module `grocery` available with no need for the package name. Here we still need to use the name of `grocery` to access `grocery_to_do`. Another way:

```
from mlwp.grocery import grocery_to_do
```

```
grocery_to_do(40, 'milk', 'bread', 'meat', 'tomato')
```

You have 40\$

Your acronym is MBMT

This `import` makes `grocery_to_do` function directly available (so no need to use the package or module prefix).



Before Python 3.3, for a folder to be treated as a package it had to have at least one `__init__.py` file that in its simplest form could have been even an empty file with this name. This file was used so that Python treats that folder as a package. However, this requirement is lifted as of Python 3.3+. For example, to create the `mlwp` package discussed above, there is no need to add such a file to the folder.

2.8.3 Aliases

Sometimes we give a nickname to a function as soon as we import it and use this nickname throughout our program. One reason for doing this is convenience! For example, why should we refer to someone named Edward as Ed? Because it is easier. Another reason is to prevent conflicts with some other functions with the same name in our codes. A given nickname is known as *alias* and we can use the keyword `as` for this purpose. For example,

```
from mlwp.grocery import grocery_to_do as gd  
  
gd(40, 'milk', 'bread', 'meat', 'tomato')
```

```
You have 40$  
Your acronym is MBMT
```

If we assign an alias to a function, we can not use the original name anymore. It is also quite common to assign an alias to a module; for example, here we give an alias to our `grocery` module:

```
from mlwp import grocery as gr  
  
gr.grocery_to_do(40, 'milk', 'bread', 'meat', 'tomato')
```

```
You have 40$  
Your acronym is MBMT
```

As a result, rather than typing the entire name of the module before the function (`grocery.grocery_to_do`), we simply use `gr.grocery_to_do`. If desired, we can assign alias to a package. For example,

```
import mlwp as mp  
  
mp.grocery.grocery_to_do(40, 'milk', 'bread', 'meat', 'tomato')
```

```
You have 40$  
Your acronym is MBMT
```

2.9 Iterator, Generator Function, and Generator Expression

2.9.1 Iterator

As mentioned before, within a `for` loop Python creates an *iterator* object from an *iterable* such as list or set. In simple terms, an iterator provides the required functionality needed by the loop (in general by the iteration protocol). But a few questions that we need to answer are the following:

- How can we produce an iterator from an iterable?
- What is the required functionality in an iteration that the iterator provides?

Creating an iterator from an iterable object is quite straightforward. It can be done by passing the iterable as the argument of the built-in function `iter()`: `iter(iterable)`. An *iterator* object itself represents a stream of data and provides the access to the *next* object in this stream. This is doable because this special object has a specific method `__next__()` that retrieves the next item in this stream (this is also doable by passing an iterator object to the built-in function `next()`, which actually calls the `__next__()` method). Once there is no more data in the stream, the `__next__()` raises `StopIteration` exception, which means the iterator is exhausted and no further item is produced by the iterator (and any further call to `__next__()` will raise `StopIteration` exception). Let us examine these concepts starting with an iterable (here a list):

```
list_a = ['a', 'b', 'c', 'd']  
iter_a = iter(list_a)  
iter_a
```

```
<list_iterator at 0x7f9c8d42a6a0>
```

```
next(iter_a) # here we access the first element by passing the iterator  
# to the next() function for the first time (similar to iter_a.  
# __next__())
```

```
'a'
```

```
iter_a.__next__() # here we access the next element using __next__()  
# method
```

```
'b'
```

```
next(iter_a)
```

```
'c'
```

```
next(iter_a)
```

```
'd'
```

Now that the iterator is exhausted, one more call raises `StopIteration` exception:

```
next(iter_a)
```

```
StopIteration                                Traceback (most recent call last)
  ↵last)
/var/folders/vy/894wbsn11db_1qf17ys9fvdm0000gn/T/ipykernel_71178/
  ↵3255143482.py in <module>
----> 1 next(iter_a)

StopIteration:
```

Let us examine what was discussed before in Section 2.7.1; that is, the general structure of a `for` loop, which is:

```
for variable in X:
    the body of the loop
```

where `X` is either an iterator or an iterable. What happens as part of the `for` loop is that the `iter()` is applied to `X` so that if `X` is an iterable, an iterator is created. Then the `next()` method is applied indefinitely to the iterator until it is exhausted in which case the loop ends.



Iterators don't need to be finite. We can have iterators that produce an infinite stream of data; for example, see `itertools.repeat()` at ([Python-repeat, 2023](#)).

2.9.2 Generator Function

Generator functions are special functions in Python that simplify writing custom iterators. A generator function returns an iterator, which can be used to generate stream of data on the fly. Let us clarify this statement by few examples.

Example 2.1 In this example, we write a function that can generate the sum of the first `n` integers and then use the return list from this function in another `for` loop to do something for each of the elements of this list (here we simply print the element but of course could be a more complex task):

```
def first_n_sum_func(n):
    sum_list_num, num = [], 1 # create an empty list to hold values of sums
    sum_num = sum(sum_list_num)
    while num <= n:
        sum_num += num
        sum_list_num.append(sum_num)
        num += 1
    return sum_list_num

for i in first_n_sum_func(10):
    print(i, end = " ") # end parameter is used to replace the default newline character at the end
```

1 3 6 10 15 21 28 36 45 55



In Example 2.1 we created a list within the function to hold the sums, returned this iterable, and used it in a `for` loop to iterate over its elements for printing. Note that if `n` becomes very large, we need to store all the sums in the memory (i.e., all the elements of the list). But what if we just need these elements to be used once? For this purpose, it is much more elegant to use *generators*. To create a generator, we can replace `return` with `yield` keyword:

```
def first_n_sum_gen(n):
    sum_num , num = 0, 1
    while num <= n:
        sum_num += num
        num += 1
        yield sum_num

for i in first_n_sum_gen(10):
    print(i, end = " ")
```

1 3 6 10 15 21 28 36 45 55

To understand how this generator function works, we need to understand several points:

- **item 1:** Any function with one or more `yield` statement(s) is a generator function;
- **item 2:** Once a generator function is called, the execution does not start and it only returns a generator object, which is an iterator (therefore, supports the `__next__()` methods);

- **item 3:** The first time the `__next__()` method (equivalently, the `next()` function) is called on the generator object, the function will start execution until it reaches the first `yield` statement. At this point the yielded value is returned, all local variables in the function are stored, and the control is passed to the calling environment;
- **item 4:** Once the `__next__()` method is called again on the generator, the function execution resumes where it left off until it reaches the `yield` again, and this process continues;
- **item 5:** Once the function terminates, `StopIteration` is raised.

To understand the above examples, we look at another example to examine the effect of these points, and then go back to Example 2.1.



If we have an iterable and we would like to print out its elements, one way is of course what was done before in Example 2.1 to use a `for` loop. However, as discussed in Section 2.6.5, an easier approach is to use `*` as the iterable unpacking operator in the `print` function call (and `print` is a function that is defined in a way to handle that):

```
print(*first_n_sum_gen(10)) # compare with  
→print(first_n_sum_gen(10))
```

1 3 6 10 15 21 28 36 45 55

```
print(*first_n_sum_func(10)) # compare with  
→print(first_n_sum_func(10))
```

1 3 6 10 15 21 28 36 45 55

Example 2.2 Here we create the following generator function and call `next()` function several times:

```
def test_gen():  
    i = 0  
    print("Part A")  
    yield i  
  
    i += 2  
    print("Part B")  
    yield i  
  
    i += 2  
    print("Part C")  
    yield i
```

```
gen_obj = test_gen() # gen_obj is a generator object
gen_obj
```

```
<generator object test_gen at 0x7f9c8d42f190>
```

Now we call the `next()` method several times:

```
next(gen_obj)
```

Part A

```
0
```

```
next(gen_obj) # observe that the value of i from where the function
←left off is preserved
```

Part B

```
2
```

```
next(gen_obj) # observe that the value of i from where the function
←left off is preserved
```

Part C

```
4
```

```
next(gen_obj) # the iterator is exhausted
```

```
StopIteration                                     Traceback (most recent call last)
←last)
/var/folders/vy/894wbsn11db_lqf17ys9fvdm0000gn/T/ipykernel_71178/
←3495006520.py in <module>
----> 1 next(gen_obj) # see item 5 above
```

```
StopIteration:
```

Example 2.1 (continued): Now here we are in the position to understand everything in `first_n_sum_gen` generator function.

Based on what we mentioned before in Section 2.9.1, in a `for` loop the `__next__()` is applied indefinitely to the iterator until it is exhausted (i.e., a `StopIteration` exception is raised). Now based on the aforementioned item 2,

once the generator function `first_n_sum_gen(10)` is called, the execution does not start and it only returns a generator object, which is an iterator. The `for` loop applies `iter()` to create an iterator but applying this function to an iterator returns the iterator itself. Then applying the `__next__()` method by the `for` loop for the first time executes this function until it reaches the `yield` statement (item 3). At this point, `sum_num` and `num` are 1 and 2, respectively, and they are preserved and, at the same time, the value of the `sum_num` is returned (i.e., 1) to the caller so `i` in the `for` loop becomes 1, and is printed out. The `for` loop applies the `__next__()` method again (item 4), and the function execution resumes where it left off; therefore, both `sum_num` and `num` become 3, and the value of `sum_num` is returned (i.e., 3), and this continues. As we can see, in the generator function `first_n_sum_gen(10)`, except for the local variables in each iteration, we do not need to store a possibly long list similar to `first_n_sum_func`. This is what really means by generating data stream on the fly: we do not need to wait until all values are generated and, therefore, less memory is used. ■

2.9.3 Generator Expression

The same way listcomps are useful for creating simple lists, generator expressions (known as *genexps*) are a handy way of creating simple generators. The general syntax for genexps is similar to listcomps except that the square brackets [] are replaced with parentheses ():

```
(expression for exp_1 in seq_1
    if condition_1
    for exp_2 in seq_2
    if condition_2
    ...
    for exp_n in seq_n
    if condition_n)
```

Naturally we use generator functions to create more complicated generators or to reuse a generator in different places, while genexps are used to create simpler generators and those that are not used frequently. Below is the genexp for implementing `first_n_sum_gen(n)`:

```
n = 10
gen1 = (sum(range(1, num + 1)) for num in range(1, n + 1))
print(*gen1)
```

1 3 6 10 15 21 28 36 45 55

Exercises:

Exercise 1: Suppose we have the following list:

```
x = [5, 10, 3, 2, 8, 0]
```

- a) what is `x[:2:-1]`?
- b) explain what would be the “start” when it is not specified in slicing and the stride is negative. Why does it make sense?

Exercise 2: Suppose we have the following list:

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

Which of the following creates a list with elements being the corresponding elements of “`a`” added to 1?

- A) `a + 1`
- B) `a + [1, 1, 1, 1]`
- C) `[i+1 for i in a]`
- D) `[a + 1]`

Exercise 3: Suppose we have the following dictionary:

```
d = {1:"good", 2:"morning", 3:"john"}
```

- a) Write a piece of code that loops through the values of this dictionary and makes the first letter in each word upper case (hint: check out `string.title()` method). The output should look like this:

```
value = Good
```

```
value = Morning
```

```
value = John
```

- b) use the iterable unpacking operator to create the following set:

```
{'good', 'john', 'morning'}
```

Exercise 4: Suppose we have the following list:

```
list_a = [1,2,3,4]
list_b = ['a','b','c','d']
```

Create a third list, namely, `list_c`, with elements being ‘`a1`’, ‘`b1`’, ‘`c1`’. Then print elements of `zip(list_a, list_b, list_c)`. Guess what would be the general

rule when the iterables have different length?

Exercise 5: Suppose we have a list of students' phone numbers and a tuple of students' names:

```
phone = [979921, 989043, 933043, 932902]
student = ('jane', 'nika', 'kian', 'mira')
```

In a single line of code, create a dictionary with phones as keys and names as values (hint: use `zip()`). Explain how the code works.

Exercise 6: Use the listcomp to create a list with three-tuple elements where each tuple contains a number between 1 and 10, its square and cubic; that is,

```
[(0, 0, 0),
(1, 1, 1),
(2, 4, 8),
(3, 9, 27),
(4, 16, 64),
(5, 25, 125),
(6, 36, 216),
(7, 49, 343),
(8, 64, 512),
(9, 81, 729)]
```

Exercise 7: We have the following dictionary:

```
dict1 = {'A1': 100, 'B1': 20, 'A2': 30, 'B2': 405, 'A3': 23}
```

Create a list of its values in one line code by:

- a) listcomp without using `values()` method
- b) iterator unpacking using `values()` method
- c) list constructor using `values()` method

Exercise 8: Suppose we have the following list:

```
list1 = ['book_john', 'pencil_askar', 'pen_amin', 'pen_john', 'key_askar',
'pen_askar', 'laptop_askar', 'backpack_john', 'car_john']
```

Use the listcomp to find john's belongings in one line of code. The output should look like

```
['book', 'pen', 'backpack', 'car']
```

This is an important application of list comprehension in which we efficiently apply a method to a sequence of objects. To do so, use `str.split` method: check <https://docs.python.org/3/library/stdtypes.html#str.split>.

Hint: as part of the list comprehension, you can use the membership operator discussed in Section 2.5.3 to check whether ‘john’ is part of each string in the list.

Exercise 9: Write a function named `belongings` that receives the first name and the last name of a student as two arguments along with an arbitrary number of items that she has in her backpack. The function should print the name and belongings of the student. For example, here are two calls of this function and their outputs:

```
belongings('Emma', 'Smith', 'pen', 'laptop')
belongings('Ava', 'Azizi', 'pen', 'charger', 'ipad', 'book')
```

Name: Emma Smith

pen

laptop

Name: Ava Azizi

pen

charger

ipad

book

Exercise 10: We have following function:

```
def prod_numbers(x, y, z):
    result = x * y * z
    return result
```

Which of the following is a legitimate code? why?

- a) `prod_numbers(z = 1, y = 3.0, 10)`
- b) `prod_numbers(10, z = 1, y = 3.0)?`
- c) `prod_numbers(10, x = 1, y = 3.0)?`

Exercise 11: Suppose there is a package called “package” that contains a module called “module”. The “module” contains a function called “`function(string)`” that expects a string. Which of the following is a legitimate way to access and use `function(string)`? Choose all that applies.

A)

```
from package import module
function("hello")
```

B)

```
import package.module as pm
pm.function("hello")
```

C)

```
from package.module import function as func  
function("hello")
```

D)

```
from package import module as m  
m.function("hello")
```

E) None of other answers



Chapter 3

Three Fundamental Python Packages

In this chapter, three fundamental Python packages will be introduced. NumPy (stands for Numerical Python) is a fundamental package at the core of various Python data science libraries including Scikit-Learn, which will be extensively used in Chapters 4-12. Another useful library for data analysis is pandas, which is built on top of NumPy. This package particularly facilitates working with tabular data. Last but not least is the matplotlib, which is the primary Python plotting library built on NumPy. When it comes to Python codes, NumPy package is known as `numpy` but in what follows, we use NumPy and numpy interchangeably.

3.1 NumPy

3.1.1 Working with NumPy Package

NumPy is a fundamental Python package that provides efficient storage and operations for multidimensional arrays. In this regard, the core functionality of NumPy is based on its `ndarray` data structure (short for n -dimensional array), which is somewhat similar to built-in `list` type but with homogeneous elements—in fact the restriction on having homogeneous elements is the main reason `ndarray` can be stored and manipulated efficiently. Hereafter, we refer to `ndarray` type as *NumPy arrays* or simply as *arrays*.

Working with NumPy arrays is quite straightforward. To do so, we need to first import the `numpy` package. By convention, we use alias `np`:

```
import numpy as np  
np
```

```
<module 'numpy' from '/Users/amin/opt/anaconda3/lib/python3.8/site-  
packages/numpy/__init__.py'>
```

Importing `numpy` triggers importing many modules as part of that. This is done by many imports as part of `__init__.py`, which is used to make many useful functions accessible at the package level. For example, to create an identity matrix, which is a commonly used function, NumPy has a convenient function named `eye()`. Once NumPy is imported, we have automatically access to this function through `np.eye()`. How is this possible? By installing Anaconda, the NumPy package is available, for example, at a location such as this (readers can check the output of the above cell to show the path on their systems):

```
/Users/amin/opt/anaconda3/lib/python3.8
```

Inside the `numpy` folder, we have `__init__.py` file. Basically once NumPy is imported, as in any other regular package, `__init__.py` is automatically executed. Inside `__init__.py`, we can see, a line `from . import lib`—here “dot” means from the current package, and `lib` is a subpackage (another folder). This way we import subpackage `lib` and within that we have another `__init__.py`. Within this file, we have `from .twodim_base import *`, which means importing everything from `twodim_base` module (file `twodim_base.py`) in the current package (i.e., `lib` subpackage). And finally within `twodim_base.py`, we have the function `eye()`!

To create a `numpy` array from a Python list or tuple, we can use `np.array()` function:

```
list1 = [[1,2,3], [4,5,6]]  
list1
```

```
[[1, 2, 3], [4, 5, 6]]
```

```
a1 = np.array(list1)  
a1
```

```
array([[1, 2, 3],  
       [4, 5, 6]])
```

We can think of a 2D `numpy` array such as `a1` as a way to store a 2×3 matrix or a list containing two lists of 3 elements. Therefore, hereafter when we refer to *rows* or *columns*, it means that we are viewing an array like a matrix. We can check type of `a1`:

```
type(a1)
```

```
numpy.ndarray
```

Here is a similar example but one of the numbers in the nested list is a floating-point:

```
list2 = [[1, 2, 3.6], [4, 5, 6]]  
list2
```

```
[[1, 2, 3.6], [4, 5, 6]]
```

```
a2 = np.array(list2)
a2
```

```
array([[1. , 2. , 3.6],
       [4. , 5. , 6. ]])
```

In the above example, all integers became floating point numbers (double precision [`float64`]). This is due to homogeneity restriction of elements in numpy arrays. As a result of this restriction, numpy upcasts all elements when possible.

3.1.2 NumPy Array Attributes

There are a few attributes of numpy arrays that are quite useful in practice.

- `dtype`: we can check the type of the elements of an array with `dtype` (short for data type) attribute of the array. For example,

```
a2.dtype
```

```
dtype('float64')
```

The `dtype` of an array is in fact what we mean by having homogeneity in array elements; that is, they should have the same `dtype`. Sometimes it is desired to specify the data type. In these cases, we can set the `dtype` to what is needed. For example, we may want to use a data type `float32` that occupies 32 bits in memory for a floating-point, rather than `float64`, which occupies 64 bits. This can be done as follows:

```
a3 = np.array(list2, dtype='float32')
a3
```

```
array([[1. , 2. , 3.6],
       [4. , 5. , 6. ]], dtype=float32)
```

Furthermore, when an array is constructed, we can recast its data type using `astype()` method:

```
a4 = a3.astype(int) # this produces a copy of the array while casting
                     ↪the specified type
a4
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
a3.dtype
```

```
dtype('float32')
```

```
a4.dtype
```

```
dtype('int64')
```

- **size**: returns the number of elements in an array:

```
a4.size # a4 contains 6 elements
```

```
6
```

- **ndim**: returns the number of dimensions (also referred to as the number of axes) of an array:

```
a4.ndim # a4 is a 2D array
```

```
2
```

- **shape**: the number of elements along each dimension:

```
a4.shape # think of a4 as a way to store a 2 by 3 matrix
```

```
(2, 3)
```

There is a difference between a 1D array and a 2D array with one row or column. Let us explore this with one example:

```
a5 = np.arange(5)  
a5
```

```
array([0, 1, 2, 3, 4])
```

```
a5.shape
```

```
(5,)
```

In the above example, a5 is a 1-dimensional array with 5 elements, so we can not write, for example, a5[3,0] or a5[0,3], but indexing for 1D array is legitimate:

```
a5[3]
```

```
3
```

However, the following array is a 2D array with 1 row and 5 columns (or perhaps more clear if we say an array of 1 list with 5 elements):

```
a6 = np.array([[0, 1, 2, 3, 4]])  
print(a6.shape)  
a6
```

(1, 5)

```
array([[0, 1, 2, 3, 4]])
```

This time writing, for example, `a6[3]` does not make sense (we can have `a6[0,3]` though). Its transpose is like a matrix of 5×1 :

```
print(a6.T.shape) # taking the transpose of the array using .T and  
# looking into its shape  
a6.T # this array has 5 rows and 1 column (or 5 lists each with 1  
# element)
```

(5, 1)

```
array([[0],  
       [1],  
       [2],  
       [3],  
       [4]])
```

- `itemsize`: number of bytes to store one element of the array:

```
a4.itemsize # int64 has 64 bits or 8 bytes
```

8

3.1.3 NumPy Built-in Functions for Array Creation

In addition to `array()`, NumPy has a series of useful built-in functions to create arrays. For example, `zeros(shape)` creates an array of a given `shape` filled with zeros:

```
a = np.zeros(5)  
a
```

```
array([0., 0., 0., 0., 0.])
```

```
a = np.zeros((3,2), dtype='int_') # int_ is the default
→ system-dependent int type, either int64 or int32
a
```

```
array([[0, 0],
       [0, 0],
       [0, 0]])
```

Other useful functions:

- `ones(shape)` creates an array of a given shape filled with ones:

```
a = np.ones((3,5))
a
```

```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

- `full(shape, value)` creates an array of a given shape filled with value:

```
a = np.full((2,4), 5.5)
a
```

```
array([[5.5, 5.5, 5.5, 5.5],
       [5.5, 5.5, 5.5, 5.5]])
```

- `eye()` creates a 2D array with ones on diagonal and zeros elsewhere (identity matrix):

```
a = np.eye(5)
a
```

```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

- `arange(start, stop, step)` creates evenly spaced values between `start` and `stop` with steps `step`. If `start` and `step` are not indicated, they are set to 0 and 1, respectively; for example,

```
a = np.arange(5)
a
```

```
array([0, 1, 2, 3, 4])
```

```
a = np.arange(3,8)
a
```

```
array([3, 4, 5, 6, 7])
```

- `random.normal(loc, scale, size)` creates an array of shape `size` with elements being independent realizations of a normal (Gaussian) random variable with mean `loc` and standard deviation `scale`. The default values of `loc`, `scale`, and `size` are 0, 1, and `None`, respectively. If both `loc` and `scale` are scalars and `size` is `None`, then one single value is returned. Here we create an array of shape “(3, 4)” where each element is a realization of a standard normal random variable:

```
a = np.random.normal(0, 1, size=(3,4))
a
```

```
array([[ 0.22755201, -0.20839413, -1.04229339, -0.5626605 ],
       [-0.58638538,  0.49326995,  0.72698224, -0.79731512],
       [-1.37778091, -0.05112057,  0.28830083,  1.39430344]])
```

A list of these and other functions are found at NumPy documentation ([NumPy-array, 2023](#)).

3.1.4 Array Indexing

Single Element Indexing: In indexing arrays, it is important to realize that if we index a multidimensional array with fewer indices than dimensions, we get a subarray. Suppose we have the following 2D array:

```
a = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]])
a
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

Then,

```
a[1]
```

```
array([3, 4, 5])
```

Therefore, `a[1]` returns (provides a view of) a 1D array at the second position (index 1). To access the first element of this subarray, for example, we can treat the subarray as an array that is subsequently indexed using [0]:

```
a[1][0]
```

```
3
```

We have already seen this type of indexing notation (separate square brackets) for list of lists in Section 2.6.1. However, NumPy also supports multidimensional indexing, in which we do not need to separate each dimension index into its square brackets. This way we can access elements by comma separated indices within a pair of square brackets `[,]`. This way of indexing numpy arrays is more efficient as compared to separate square brackets because it does not return an entire subarray to be subsequently indexed and, therefore, it is the preferred way of indexing. For example:

```
a[1, 0]
```

```
3
```

Slicing: We can access multiple elements using slicing and striding in the same way we have previously seen for lists and tuples. For example,

```
a[:3:2,-2:]
```

```
array([[1, 2],  
       [7, 8]])
```

In the code snippet below we choose columns with a stride of 2 and the last two rows in reverse:

```
a[:-3:-1,::2]
```

```
array([[ 9, 11],  
       [ 6,  8]])
```

One major difference between numpy array slicing and list slicing is that array slicing provides a *view* of the original array whereas list slicing copies the list. This means that if a subarray, which comes from slicing an array, is modified, the changes appear in the original array too (recall that in list slicing that was not the case). This is quite useful when working with large datasets because it means that we can work and modify pieces of such datasets without loading and working with the entire dataset at once. To better see this behaviour, consider the following example:

```
b = a[:-3:-1,::2]
b[0,1] = 21 # change 11 in b to 21
a # the change appears in a too
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 21]])
```

To change this default behaviour, we can use: 1) the `copy()` method; and 2) the `array()` constructor:

```
b = a[:-3:-1,::2].copy() # using copy()
b
```

```
array([[ 9, 21],
       [ 6,  8]])
```

```
b[1,1] = 18
a # the change does not appear in a
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 21]])
```

```
b = np.array(a[:-3:-1,::2]) # using array()
b
```

```
array([[ 9, 21],
       [ 6,  8]])
```

```
b[1,1] = 18
a # the change does not appear in a
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 21]])
```

Indexing with other Arrays (known as Fancy Indexing): A useful property of NumPy arrays is that we can use other arrays (or even lists) for indexing:

```
c = np.arange(10, 30)
idx = np.arange(19, 10, -2)
print('c = ' + str(c) \
```

```
+ '\n' + 'idx = ' + str(idx) \
+ '\n' + 'c[idx] = ' + str(c[idx]))
```

```
c = [10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29]
idx = [19 17 15 13 11]
c[idx] = [29 27 25 23 21]
```

In this type of indexing:

- 1) indices must be integers;
- 2) a copy of the original array is returned (not a view);
- 3) the use of negative indices is allowed and the meaning is similar to what we have already seen; and
- 4) out of bound indices are errors.

Let us examine the first point by changing elements of `idx` to `float32`:

```
idx = idx.astype('float32')
c[idx] # observe that when we change the dtype of idx to float32 an
      ↵error is raised
```

```
IndexError                                         Traceback (most recent call
      ↵last)
/var/folders/vy/894wbsn11db_lqf17ys9fvdm0000gn/T/ipykernel_43151/
      ↵419593116.py in <module>
      1 idx = idx.astype('float32')
----> 2 c[idx] # observe that we changed the dtype of idx to float32 and
      ↵we get error

IndexError: arrays used as indices must be of integer (or boolean) type
```

And here we examine the effect of fancy indexing with a negative index:

```
idx = idx.astype('int_')
idx[0] = -1 # observe how the negative index is interpreted
print('\n' + 'idx = ' + str(idx) \
      + '\n' + 'c[idx] = ' + str(c[idx]))
```

```
idx = [-1 17 15 13 11]
c[idx] = [29 27 25 23 21]
```

3.1.5 Reshaping Arrays

One major difference between numpy arrays and lists is that the size (number of elements) of an array once it is created is fixed; that is to say, to shrink and expand an existing array we need to create a new array and copy the elements from the old one to the new one, and that is computationally inefficient. However, as we discussed in Section 2.6.1, expanding or shrinking a list was straightforward through the methods defined for lists. Although the size of an array is fixed, its shape is not and could be changed if desired. The most common way to reshape an array is to use `reshape()` method:

```
a = np.arange(20).reshape(4,5) # 4 elements in the first dimension,  
# (axis 0) and 5 in the second dimension (axis 1)  
a
```

```
array([[ 0,  1,  2,  3,  4],  
       [ 5,  6,  7,  8,  9],  
       [10, 11, 12, 13, 14],  
       [15, 16, 17, 18, 19]])
```

The `reshape()` method provides a view of the array in a new shape. Let us examine this by an example:

```
b = a.reshape(2,10)  
b
```

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],  
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]])
```

```
b[0,9] = 29  
a # observe that the change in b appears in a
```

```
array([[ 0,  1,  2,  3,  4],  
       [ 5,  6,  7,  8, 29],  
       [10, 11, 12, 13, 14],  
       [15, 16, 17, 18, 19]])
```

To use the `reshape()` method, we can specify the number of elements in each dimension and we need to ensure that the total number of these elements is the same as the size of the original array. However, because the size is fixed, the number of elements in one axis could be determined from the size and the number of elements in other axes (as long as they are compatible). NumPy uses this fact and allows us not to specify the number of elements along one axis. To do so, we can use `-1` along an axis and NumPy will determine the number of elements along that axis. For example,

```
c = b.reshape(5,-1)
c
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8, 29, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

```
c.reshape(-1) # this returns a flattened array regardless of the
               ↵original shape
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8, 29, 10, 11, 12, 13, 14, 15,
       ↵16, 17, 18, 19])
```

```
c.ravel() # this could be used instead of reshape(-1)
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8, 29, 10, 11, 12, 13, 14, 15,
       ↵16, 17, 18, 19])
```

Occasionally it is required to convert a 1D array to a 2D array with one column or row. This could be done using 1) `reshape()`; 2) adding `None` keyword to a new axis along with slicing; and 3) using `newaxis` object as an alias for `None`:

```
c = np.arange(5) # to create a 1D array with 5 elements
c
```

```
array([0, 1, 2, 3, 4])
```

```
c.reshape(-1,1) # a 2D array with 5 rows and 1 column using reshape
               ↵method
```

```
array([[0],
       [1],
       [2],
       [3],
       [4]])
```

Below we use `newaxis` object to insert a new axis into column of `c`:

```
c[:,np.newaxis] # a 2D array with 5 rows and 1 column
```

```
array([[0],
       [1],
       [2],
       [3],
       [4]])
```

```
c[np.newaxis,:] # a 2D array with 1 row and 5 columns, which is
←equivalent to c.reshape(1,5)
```

```
array([[0, 1, 2, 3, 4]])
```

3.1.6 Universal Functions (UFuncs)

Computations using NumPy arrays could be either slow and verbose or fast and convenient. The main factor that leads to both fast computations and convenience of implementation is vectorized operations. These are vectorized wrappers for performing element-wise operation and are generally implemented through NumPy *Universal Functions* also known as *ufuncs*, which are some built-in functions written in C for efficiency. In the following example, we will see how the use of ufuncs will improve computational efficiency.

Here we calculate the time to first create a numpy array called “`a`” of size 10000 and then subtract 1 from every element of `a`. We first use a conventional way of doing this, which is a `for` loop:

```
%%timeit
a = np.arange(10000)
for i, v in enumerate(a):
    a[i] -= 1
```

`3.55 ms ± 104 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)`

Now we use the subtraction operator “`-`”, which implements the vectorized operation (applies the operation to every element of the array):

```
%%timeit
a = np.arange(10000)
a = a - 1 # vectorized operation using the subtraction operator
```

`8.8 µs ± 124 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)`

Note the striking difference in time! The vectorized operation is about 400 times faster here. We can also implement this vectorized subtraction using `np.subtract` `ufunc`:

```
%%timeit
a = np.arange(10000)
a = np.subtract(a,1)
```

`8.88 µs ± 192 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)`



In the above code, we used a Python *magic* function `timeit` with `%%` to determine the execution time of a code block. Python magic functions simplify some common tasks (here, timing). They come either by a prefix `%` (inline magic) in which case they operate on a single line of code or `%%` (cell magic) that makes them operate on a block of code.

As a matter of fact, the “`-`” operator is just a wrapper around `np.subtract()` function. NumPy uses operator overloading here to make it possible for us to use standard Python subtraction operator `-` for a vectorized operation on arrays. This also applies to several other operations as listed in Table 3.1.

Table 3.1: Binary ufuncs and their equivalent operators

UFunc	Equivalent Operator
<code>np.add</code>	<code>+</code>
<code>np.subtract</code>	<code>-</code>
<code>np.multiply</code>	<code>*</code>
<code>np.divide</code>	<code>/</code>
<code>np.power</code>	<code>**</code>
<code>np.floor_divide</code>	<code>//</code>
<code>np.mod</code>	<code>%</code>

All the previous ufuncs require two operands. As a result, they are referred to as binary ufuncs; after all, these are implementations of binary operation. However, there are other useful operations (and ufuncs), which are unary; that is, they operate on a single array. For example, operations such as calculating the sum or product of elements of an entire array or along a specific axis, or taking element-wise natural log, to just name a few. Table 3.2 provides a list of some of these unary ufuncs. Below, we present some examples of these unary ufuncs:

```
a = np.array([[3,5,8,2,4],[8, 10, 2, 3, 5]])
a
```

```
array([[ 3,  5,  8,  2,  4],
       [ 8, 10,  2,  3,  5]])
```

Then,

```
np.min(a, axis=0)
```

```
array([3, 5, 2, 2, 4])
```

```
np.sum(a, axis=1)
```

Table 3.2: Unary ufuncs and their equivalent operators

UFunc	Operation
<code>np.sum</code>	Sum of array elements over a given axis
<code>np.prod</code>	Product of array elements over a given axis
<code>np.mean</code>	Arithmetic mean over a given axis
<code>np.std</code>	Standard deviation over a given axis
<code>np.var</code>	Variance over a given axis
<code>np.log</code>	Element-wise natural log
<code>np.log10</code>	Element-wise base 10 logarithm
<code>np.sqrt</code>	Element-wise square root
<code>np.sort</code>	Sorts an array
<code>np.argsort</code>	Returns the indices that would sort an array
<code>np.min</code>	Returns the minimum value of an array over a given axis
<code>np.argmin</code>	Returns the index of the minimum value of an array over a given axis
<code>np.max</code>	Returns the maximum value of an array over a given axis
<code>np.argmax</code>	Returns the index of the maximum value of an array over a given axis

```
array([22, 28])
```

For some of these operations such as `sum`, `min`, and `max`, a simpler syntax is to use the method of the array object. For example:

```
a.sum(axis=1) # here sum is the method of the array object
```

```
array([22, 28])
```

For each of these methods, we can check its documentations at numpy.org to see its full functionality. For example, for `np.sum` and `ndarray.sum`, the documentations are available at ([NumPy-sum, 2023](#)) and ([NumPy-arrsum, 2023](#)), respectively.

3.1.7 Broadcasting

Broadcasting is a set of rules that instructs NumPy how to treat arrays of different shapes and dimensions during arithmetic operations. In this regard, under some conditions, it broadcasts the smaller array over the larger array so that they have compatible shapes for the operation. There are two main reasons why broadcasting in general leads to efficient computations: 1) the looping required for vectorized operations occurs in C instead of Python; and 2) it implements the operation without copying the elements of array so it is memory efficient.

As a matter of fact, we have already seen broadcasting in the previous section where we used arithmetic operations on arrays. Let us now see some examples and examine them from the standpoint of broadcasting. In the following example, a constant 2 is added to every element of array `a`:

```
a = np.array([0, 1, 2, 3, 4])
b = a + 2
b
```

```
array([2, 3, 4, 5, 6])
```

One can think of this example as if the scalar 2 is stretched (broadcast) to create an array of the same shape as `a` so that we can apply the element-by-element addition (see Fig. 3.1). However, it is important to remember that NumPy neither makes a copy of 2 nor creates an array of the same shape as `a`. It just uses the original copy of 2 (so it is memory efficient) and based on a set of rules efficiently performs the addition operation as if we have an array of 2 with the same shape as `a`.

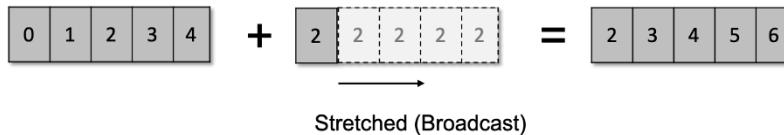


Fig. 3.1: A conceptual illustration of broadcasting for adding a scalar to an array

Broadcasting could be understood by the following rules:

- **Rule 1:** If two arrays have different number of dimensions (i.e., `ndim` is different for them), dimensions ones are added to the left of the shape of the array with fewer dimensions to make them both of the same dimensions;
- **Rule 2:** Once dimensions are equal, the size of arrays in each dimension must be either equal or 1 in at least one of them; otherwise, an error will be raised that these arrays are not broadcastable; and
- **Rule 3:** If an error is not raised (i.e., in all dimensions, arrays have either equal size or at least one of them has size 1), then the size of the output array in each dimension is the maximum of the array sizes in that dimension (this implies that the array with size of 1 in any dimension is stretched to match the other one).

In the next few examples, we examine these three rules.

Example 3.1 Two arrays, namely, `a` and `b`, are created:

```
a = np.ones((2,4))
print(a)
a.shape
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

```
(2, 4)
```

```
b = np.random.randint(1, 10, 4) # to generate random integers (see https://numpy.org/doc/stable/reference/random/generated/numpy.random.randint.html)
print(b)
b.shape
```

[2 1 9 6]

(4,)

If we write `a + b`:

- By Rule 1, a single 1 is added to the left of `b` shape to make `b` a two-dimensional array as `a`; that is, we can think of `b` now as having shape (1, 4).
- The shape of `a` is (2, 4) and the shape of `b` is (1, 4). Therefore, Rule 2 does not raise an error because the sizes in each dimension are either equal (here 4) or at least one of them is 1 (for example, 2 in `a` and 1 in `b`).
- Rule 3 indicates that the size of the output is ($\max\{2,1\}$, $\max\{4,4\}$) so it is (2, 4). This means that array `b` is stretched along its axis 0 (the left element in a shape tuple) to match the size of `a` in that dimension.

Let's see the outcome:

```
c = a + b
print(c)
c.shape
```

[[3. 2. 10. 7.]
 [3. 2. 10. 7.]]

(2, 4)

Example 3.2 Two arrays, namely, `a` and `b`, are created:

```
a = np.arange(8).reshape(2,4,1) # a 3-dimensional array
print(a)
a.shape
```

[[[0]
 [1]
 [2]
 [3]]]

[[4]
 [5]
 [6]
 [7]]]

```
(2, 4, 1)
```

```
b = np.array([10, 20, 30, 40])
```

If we write `a + b`:

- By Rule 1, a 1 is added to the left of `b` shape to make it a 3-dimensional array as `a`; that is, we can think of `b` now as having shape (1, 1, 4).
- The shape of `a` is (2, 4, 1) and the shape of `b` is (1, 1, 4). Therefore, Rule 2 does not raise an error because the sizes in each dimension are either equal or at least one of them is 1.
- Rule 3 indicates that the size of the output is ($\max\{2,1\}$, $\max\{4,1\}$, $\max\{1,4\}$) so it is (2, 4, 4). This means that array `a` is stretched along its axis 2 (the third element in its shape tuple) to make its size 4 (similar to `b` in that dimension), array `b` is stretched along its axis 1 to make its size 4 (similar to `a` in that dimension), and finally array `b` is stretched along its axis 0 to make its size 2.

Let's see the outcome:

```
c = a + b
print(c)
c.shape
```

```
[[[10 20 30 40]
 [11 21 31 41]
 [12 22 32 42]
 [13 23 33 43]]

 [[14 24 34 44]
 [15 25 35 45]
 [16 26 36 46]
 [17 27 37 47]]]
```

```
(2, 4, 4)
```

Example 3.3 Two arrays, namely, `a` and `b`, are created:

```
a = np.arange(8).reshape(2,4,1) # a 3-dimensional array
a.shape
```

```
(2, 4, 1)
```

```
b = np.array([10, 20, 30, 40]).reshape(2,2)
b
```

```
array([[10, 20],
 [30, 40]])
```

If we write `a + b`:

- By Rule 1, a 1 is added to the left of `b` shape to make it a 3-dimensional array as `a`; that is, we can think of `b` now as having shape $(1, 2, 2)$.
- The shape of `a` is $(2, 4, 1)$ and the shape of `b` is $(1, 2, 2)$. Therefore, Rule 2 raises an error because the sizes in the second dimension (axis 1) are neither equal nor at least one of them is 1.

Let's see the error:

```
a + b
```

```
ValueError                                Traceback (most recent call last)
  ↵last)
/var/folders/vy/894wbsn11db_lqf17ys9fvdm0000gn/T/ipykernel_43151/
  ↵1216668022.py in <module>
----> 1 a + b

ValueError: operands could not be broadcast together with shapes (2,4,1) &
  ↵(2,2)
```

3.2 Pandas

Pandas specially facilitates working with tabular data that can come in a variety of formats. For this purpose, it is built around a type of data structure known as `DataFrame`. `Dataframe`, named after a similar data structure in R programming language, is indeed the most commonly used object in pandas and is similar to a spreadsheet in which columns can have different types. However, understanding that requires the knowledge of a more fundamental data structure in pandas known as `Series`.

3.2.1 Series

A `Series` is a one-dimensional array of indexed data. Specifically, in order to create a `Series`, we can use:

```
s = pd.Series(data, index=index)
```

where the argument `index` is optional and `data` can be in many different types but it is common to see an `ndarray`, list, dictionary, or even a constant.

If the data is a list or ndarray and index is not passed, the default values from 0 to len(data)-1 will be used. Here we create a Series from a list without providing an index:

```
import pandas as pd

s1 = pd.Series(['a', np.arange(6).reshape(2,3), 'b', [3.4, 4]]) # to ↴
↳create a series from a list
```

s1

```
0          a
1      [[0, 1, 2], [3, 4, 5]]
2          b
3      [3.4, 4]
dtype: object
```

```
s1 = pd.Series(['a', np.arange(6).reshape(2,3), 'b', [3.4, 4]], ↴
↳index=np.arange(1,5)) # providing the index
s1
```

```
1          a
2      [[0, 1, 2], [3, 4, 5]]
3          b
4      [3.4, 4]
dtype: object
```

Here, we create a Series from a ndarray:

```
s2 = pd.Series(np.arange(2, 10))
s2
```

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

We can access the values and indices of a Series object using its values and index attributes:

s2.values

```
array([2, 3, 4, 5, 6, 7, 8, 9])
```

```
s2.index
```

```
RangeIndex(start=0, stop=8, step=1)
```

The square brackets and slicing that we saw for NumPy arrays can be used along with `index` to access elements of a `Series`:

```
s2[2] # using index 2 to access its associated value
```

```
4
```

```
s2[1:-1:2] # the use of slicing
```

```
1    3  
3    5  
5    7  
dtype: int64
```

Similar to NumPy arrays, the slicing provides a view of the original `Series`:

```
s3 = s2[2::2]  
s3[2] += 10  
s2
```

```
0    2  
1    3  
2    14  
3    5  
4    6  
5    7  
6    8  
7    9  
dtype: int64
```

An important difference between accessing the elements of `Series` and numpy arrays is the *explicit index* used in `Series` as opposed to the *implicit index* used in numpy. To better understand this difference, try to access the first element of array `s3` in the above example:

```
s3
```

```
2    14  
4    6  
6    8  
dtype: int64
```

Does `s3[0]` work? Let's see:

```
#s3[0] # raises error if run
```

How about `s3[2]`?

```
s3[2]
```

14

The above example would make more sense if we consider the index as a label for the corresponding value. Thus, when we write `s3[2]`, we try to access the value that corresponds to label 2. This mechanism gives more flexibility to `Series` objects because for the indices we can use other types rather than integers. For example,

```
s4 = pd.Series(np.arange(10, 15), index = ['spring', 'river', 'lake',  
    ↴'sea', 'ocean'])  
s4
```

```
spring      10  
river       11  
lake        12  
sea         13  
ocean       14  
dtype: int64
```

We can also access multiple elements using a list of labels:

```
s4[['river', 'sea']]
```

```
river      11  
sea       13  
dtype: int64
```

However, when it comes to slicing, the selection process using integer indices works similarly to NumPy. For example, in `s3` where `index` had integer type, `s3[0:2]` chooses the first two elements (similar to NumPy arrays):

```
s3[0:2]
```

```
2      14  
4      6  
dtype: int64
```

But at the same time, we can also do slicing using explicit index:

```
s4['spring':'ocean':2]
```

```
spring      10  
lake        12
```

```
ocean      14  
dtype: int64
```

Notice that when using the explicit index in slicing, both the start and the stop are included (as opposed to the previous example where implicit indexing was used in slicing and the stop was not included).

When in a Series we have integer indexing, the fact that slicing uses implicit indexing could cause confusion. For example,

```
s = pd.Series(np.random.randint(1, 10, 7), index = np.arange(2, 9))  
s
```

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

```
s[0:3] # slicing uses implicit indexing
```

```
2    2  
3    5  
4    1  
dtype: int64
```

To resolve such confusions, pandas provides two attributes `loc` and `iloc` for Series and DataFrame objects using which we can specify the type of indexing. Using `loc` and `iloc` one uses explicit and implicit indexing, respectively:

```
s.iloc[0:3] # note that the value corresponding to "3" is not included
```

```
2    2  
3    5  
4    1  
dtype: int64
```

```
s.loc[2:4] # note that the value corresponding to "4" is included
```

```
2    2  
3    5  
4    1  
dtype: int64
```

As we said before, we can create Series from dictionaries as well; for example,

```
s = pd.Series({"c": 20, "a": 40, "b": 10})
s
```

```
c    20
a    40
b    10
dtype: int64
```

If a dictionary is used as `data` to create `Series`, depending on the version of pandas and Python that are used, the `index` will be ordered either by the dictionary's insertion order (for pandas version ≥ 0.23 and Python version ≥ 3.6) or lexically (if either pandas or Python is older than the aforementioned versions). If an index is passed with a dictionary, then values corresponding to the dictionary keys that are in the `index` are used in `Series`:

```
s = pd.Series({"c": 20, "a": 40, "b": 10}, index=["b", "c", "d", "a"])
s
```

```
b    10.0
c    20.0
d    NaN
a    20.0
dtype: float64
```

In the above example, we tried to pull out a value for a non-existing key in the dictionary ("d"). That led to `NaN` (short for Not a Number), which is the standard marker used in pandas for missing data.

Finally, if we use a scalar in the place of `data`, the scalar is repeated to match the size of `index`:

```
s = pd.Series('hi', index = np.arange(5))
s
```

```
0    hi
1    hi
2    hi
3    hi
4    hi
dtype: object
```

3.2.2 DataFrame

A `DataFrame` is a generalization of `Series` to 2-dimension; that is, a 2-dimensional data structure where both the rows and columns are indexed (labeled). Nevertheless,

it is common to refer to row labels as `index` and to column labels as `columns`. As elements of `Series` can have different type, in `DataFrame` both the rows and columns can have different types; for example, here we create a `DataFrame` from three `Series` where each has different data types:

```
d = {
    "one": pd.Series('hi', index=["a", "b", "c", "d"]),
    "two": pd.Series([1.0, ["yes", "no"], 3.0, 5], index=["a", "b", "c", "d"]),
    "three": pd.Series({"c": 20, "a": 40, "b": 10}, index=["a", "b", "c", "d"])
}
d
```

```
{'one': a    hi
 b    hi
 c    hi
 d    hi
dtype: object,
'two': a      1.0
 b    [yes, no]
 c      3.0
 d      5
dtype: object,
'three': a    40.0
 b    10.0
 c    20.0
d    NaN
dtype: float64}
```

```
df = pd.DataFrame(d)
```

```
df # pandas nicely presents the DataFrame
```

	one	two	three
a	hi	1.0	40.0
b	hi	[yes, no]	10.0
c	hi	3.0	20.0
d	hi	5	NaN

There are various ways to create a `DataFrame` but perhaps the most common ways are:

- from a dictionary of `Series`
- from ndarrays or lists
- from a dictionary of ndarrays or lists
- from a list of dictionaries

Let's see some examples for each.

- **DataFrame from a dictionary of Series:** this is in fact what we observed before. In this way, the index (i.e., row labels) is the union of index of all Series. Let's consider another example:

```
kaz = pd.Series([1000, 200, 30, 10], index = ['spring', 'river',  
       ↵'pond', 'lake'])  
jap = pd.Series([900, 80, 300], index = ['spring', 'lake', 'river'])  
df = pd.DataFrame({'Kazakhstan': kaz,  
                   "Japan": jap})  
df # observe the missing data
```

	Kazakhstan	Japan
lake	10	80.0
pond	30	NaN
river	200	300.0
spring	1000	900.0

- **DataFrame from numpy arrays or lists:**

```
import numpy as np  
import pandas as pd  
a = np.array([[10, 30, 200, 1000], [80, np.nan, 300, 900]]).T # to transpose  
df = pd.DataFrame(a, index = ['lake', 'pond', 'river', 'spring'],  
                   ↵columns = ['Kazakhstan', 'Japan'])  
df
```

	Kazakhstan	Japan
lake	10.0	80.0
pond	30.0	NaN
river	200.0	300.0
spring	1000.0	900.0



In the above example, we used `np.nan`, which is a floating-point representation of “not a number” (missing value) handled by NumPy. To better see the impact of this, notice that `a.dtype` returns `float64`. Now, change `np.nan` to `None` and run the same code. This time, `a.dtype` is Python Object (identified ‘0’). Therefore, unlike the first case where `np.nan` was used, operations on the array are not as efficient.

- **DataFrame from a dictionary of arrays or lists:** To use this, the length of all arrays/lists should be the same (and the same as `index` length if provided).

```
dict = {"Kazakhstan": [100, 200, 30, 10], "Japan": [900, 300, None, 80]}
df = pd.DataFrame(dict, index=['spring', 'river', 'pond', 'lake'])
```

	Kazakhstan	Japan
spring	100	900.0
river	200	300.0
pond	30	NaN
lake	10	80.0

- DataFrame from a list of dictionaries:

```
my_list = [{"Kazakhstan": 100, "Japan": 900}, {"Kazakhstan": 200, "Japan": 300}, {"Kazakhstan": 30}, {"Kazakhstan": 10, "Japan": 80}]
df = pd.DataFrame(my_list, index=['spring', 'river', 'pond', 'lake'])
```

	Kazakhstan	Japan
spring	100	900.0
river	200	300.0
pond	30	NaN
lake	10	80.0

In the above examples, we relied on pandas to arrange the columns based on the information in `data` parameters provided to `DataFrame` constructor (see ([Pandas-dataframe, 2023](#))). In the last example, for instance, the columns were determined by the keys of dictionaries in the list. We can use the `columns` parameter of `DataFrame` to re-arrange them or even add extra columns. For example:

```
list1 = [{"Kazakhstan": 100, "Japan": 900}, {"Kazakhstan": 200, "Japan": 300}, {"Kazakhstan": 30}, {"Kazakhstan": 10, "Japan": 80}]
df = pd.DataFrame(list1, index=['spring', 'river', 'pond', 'lake'], columns=['Japan', 'USA', 'Kazakhstan'])
```

	Japan	USA	Kazakhstan
spring	900.0	NaN	100
river	300.0	NaN	200
pond	NaN	NaN	30
lake	80.0	NaN	10

Once a `DataFrame` is at hand, we can access its elements in various forms. For example, we can access a column by its name:

```
df['Japan']
```

```
spring    900.0
river     300.0
pond      NaN
lake      80.0
Name: Japan, dtype: float64
```

If the column names are strings, this could be also done by an “attribute-like” indexing as follows:

```
df.Japan
```

```
spring    900.0
river     300.0
pond      NaN
lake      80.0
Name: Japan, dtype: float64
```

Multiple columns can be accessed by passing a list of columns:

```
df[['USA', 'Japan']]
```

```
      USA   Japan
spring  NaN  900.0
river   NaN  300.0
pond    NaN    NaN
lake    NaN   80.0
```

At the same time, we can use `loc` and `iloc` attributes for explicit and implicit indexing, respectively:

```
df.iloc[1:4,0:2]
```

```
      Japan   USA
river   300.0  NaN
pond    NaN    NaN
lake    80.0  NaN
```

```
df.loc[['river','lake'], 'Japan':'USA']
```

```
      Japan   USA
river   300.0  NaN
lake    80.0  NaN
```

Another way is to retrieve each column, which is a `Series`, and then index on that:

```
df.Japan.iloc[:2]
```

```
spring    900.0
river     300.0
Name: Japan, dtype: float64
```

which is equivalent to:

```
df['Japan'].iloc[:2]
```

```
spring    900.0
river     300.0
Name: Japan, dtype: float64
```

That being said, array style indexing without `loc` or `iloc` is not acceptable. For example,

```
df.iloc[1,0]
```

```
300.0
```

and

```
df.loc['river', 'Japan']
```

```
300.0
```

are legitimate, but the following raises error:

```
#df['river', 'Japan'] # raises error if run
```

We can also use *masking* (i.e., select some elements based on some criteria). For example, assume we have a DataFrame of students with their information as follows:

```
import pandas as pd
dict1 = {
    "GPA": pd.Series([3.00, 3.92, 2.89, 3.43, 3.55, 2.75]),
    "Name": pd.Series(['Askar', 'Aygul', 'Ainur', 'John', 'Smith', 'Xian']),
    "Phone": pd.Series([8728, 8730, 8712, 8776, 8770, 8765])
}
df = pd.DataFrame(dict1)
df
```

	GPA	Name	Phone
0	3.00	Askar	8728
1	3.92	Aygul	8730
2	2.89	Ainur	8712
3	3.43	John	8776
4	3.55	Smith	8770
5	2.75	Xian	8765

We would like to retrieve the information for any student with $\text{GPA} > 3.5$. To do so, we define a boolean mask (returns `True` or `False`) that can easily be used for indexing:

```
df.GPA > 3.5
```

```
0    False
1    True
2   False
3   False
4    True
5   False
Name: GPA, dtype: bool
```

```
df.Name[df.GPA > 3.5]
```

```
1    Aygul
4   Smith
Name: Name, dtype: object
```

In the above example we only returned values of columns `Name` at which the mask array (`df.GPA > 3.5`) is `True`. To retrieve all columns for which the mask array is `True` we can use the mask with the `DataFrame`:

```
df[df.GPA > 3.5]
```

	GPA	Name	Phone
1	3.92	Aygul	8730
4	3.55	Smith	8770

There are two handy methods for `DataFrame` objects, namely, `head(n=5)` and `tail(n=5)` that return the first and last `n` rows of a `DataFrame` (the default value of `n` is 5), respectively:

```
df.head()
```

	GPA	Name	Phone
0	3.00	Askar	8728
1	3.92	Aygul	8730
2	2.89	Ainur	8712
3	3.43	John	8776
4	3.55	Smith	8770

Last but not least, we can access the column and index labels through the `columns` and `index` attributes of a `DataFrame`:

```
df.columns
```

```
Index(['GPA', 'Name', 'Phone'], dtype='object')
```

3.2.3 Pandas Read and Write Data

A powerful feature of Pandas is its ability to easily read data from and write to a variety of common formats (sometimes, depending on specific applications, some other libraries would be used as well). For example, Pandas can be used to read from/write to csv format, pickle format (this is a popular Python binary data format using which we can save *any* Python object (pickling) to be accessed later (unpickling)), SQL, html, etc. A list of pandas input/output (I/O) functions is found at ([Pandas-io, 2023](#)).

As an example, suppose we would like to read the table of pandas I/O tools, which is available at the URL ([Pandas-io, 2023](#)). For this purpose, we can use pandas `read_html` that reads tables from an HTML:

```
url = 'https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html'
table = pd.read_html(url)
print(len(table))
```

8

The `read_html()` returns a list of possible DataFrames in the html file (see ([Pandas-readhtml, 2023](#))). This is why the length of the above `table` is 8 (as of writing this manuscript). Our desired table is the first one though:

```
table[0]
```

	Format	Type	Data Description	Reader	Writer
0	text		CSV	read_csv	to_csv
1	text	Fixed-Width Text File		read_fwf	NaN
2	text		JSON	read_json	to_json
3	text		HTML	read_html	to_html
4	text		LaTeX	NaN	Styler.to_latex
5	text		XML	read_xml	to_xml
6	text	Local clipboard		read_clipboard	to_clipboard
7	binary		MS Excel	read_excel	to_excel
8	binary		OpenDocument	read_excel	NaN
9	binary		HDF5 Format	read_hdf	to_hdf
10	binary		Feather Format	read_feather	to_feather
11	binary		Parquet Format	read_parquet	to_parquet
12	binary		ORC Format	read_orc	to_orc
13	binary		Stata	read_stata	to_stata
14	binary		SAS	read_sas	NaN
15	binary		SPSS	read_spss	NaN
16	binary	Python Pickle	Format	read_pickle	to_pickle
17	SQL		SQL	read_sql	to_sql
18	SQL	Google BigQuery		read_gbq	to_gbq

3.3 Matplotlib

3.3.1 Backend and Frontend

Matplotlib is the primary Python plotting library built on NumPy. It is designed to support a wide variety of use cases: to draw inline and interactive plots in Jupyter notebook, to draw plots on new windows from Python shell, or to create plots for web applications or cross-platform graphical user interfaces such as Qt API. To enable these use cases, `matplotlib` supports a variety of *backends*. We refer to the behind-the-scene hard work that makes the plotting happen as the “backend” in contrast to the “frontend”, which depending on the context, is referred to the coding environment (e.g., Jupyter notebook) or the codes themselves.

There are generally two types of backends: 1) interactive using which we can zoom in/out or move the plots around; and 2) non-interactive, which is used for creating static images of our figures (e.g., png, ps, and pdf). In Jupyter notebook, a simple way to set the backend is to use the Jupyter magic command `%matplotlib`. For example, using

```
%matplotlib notebook
```

we can create interactive figures in the notebook, and using

```
%matplotlib inline
```

which is the default backend in notebooks, we can create static images in the notebook. A list of available backends is shown by

```
%matplotlib --list
```

3.3.2 The Two `matplotlib` Interfaces: `pyplot-style` and `OO-style`

To plot figures, `matplotlib` generally relies on two concepts of `Figure` and `Axes`. Basically, the `Figure` is what we think of as the “whole figure”, while `Axes` is what we generally refer to as “a plot” (e.g., part of the image with curves and coordinates). Therefore, a `Figure` can contain many `Axes` but each `Axes` can be part of one figure. Using an `Axes` object and its defined methods, we can control various properties. For example, `Axes.set_title()`, `Axes.set_xlabel()`, and `Axes.set_ylim()` can be used to set the title of a plot, set the label of the x axis, or set the limits of the y-axis, respectively. There are two ways to use `matplotlib`:

- 1) the `pyplot` style, which resembles plotting Matlab figures (therefore, it is also referred to as the Matlab style). In this style, we rely on `pyplot` module and its functions to automatically create and manage figures and axes; and

- 2) the object-oriented style (OO-style) using which we create figures and axes, and control them using methods defined for them.

In general the `pyplot` style is less flexible than the OO-style and many of the functions used in `pyplot` style can be implemented by methods of `Axes` object.

pyplot-style—case of a simple plot: We first create a simple plot using the `plot` function of `pyplot` module (see ([pyplot, 2023](#)) for its full documentation). To do so, we first import the `pyplot` and use the standard alias `plt` to refer to that. As plotting functions in `matplotlib` expect NumPy arrays as input (even array-like objects such as lists as inputs are converted internally to NumPy arrays), it is common to see importing NumPy along with `matplotlib`.

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

x = np.linspace(0, 1, 20) # to create an array of x axis elements
plt.figure(figsize=(4.5, 2.5), dpi = 200)
plt.plot(x, x, linewidth=2) # to use the default line style and color,
# and change the width of the line using linewidth
plt.plot(x, x**3, 'r+', label = 'cubic function') # to use the red
# color and circle marker to plot, and label parameter for legend
plt.plot(x, np.cos(np.pi * x), 'b-', label = 'cosine function',
#marker='d', markersize = 5) # to use the marker parameter and the
#markersize to control the size
plt.xlabel('time', fontsize='small')
plt.ylabel('amplitude', fontsize='small')
plt.title('time-amplitude', fontsize='small')
plt.legend(fontsize='small') # to add the legend
plt.tick_params(axis='both', labelsize=7) # to adjust the size of tick
#labels on both axes
# plt.show() # this is not required here because Jupyter backends will
#call show() at the end of each cell by themselves
```

In the above code, and for illustration purposes, we used various markers. The list of possible markers is available at ([matplotlib-markers, 2023](#)).

OO-style—case of a simple plot: Here we use the OO-style to create a similar plot as shown in Fig. 13.2. To do so, we use `subplots()` from `pyplot` (not to be confused with `subplot()`) that returns a figure and a single or array of `Axes` objects. Then we can use methods of `Axes` objects such as `Axes.plot` to plot data on the axes (or use aforementioned `Axes` methods to control the behaviour of the `Axes` object):

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

x = np.linspace(0, 1, 20)
```

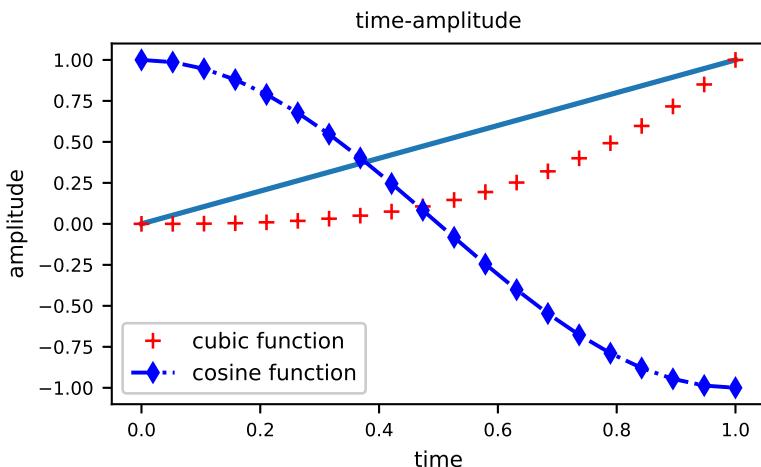


Fig. 3.2: The output of the above plotting example. Here we used pyplot-style.

```
fig, ax = plt.subplots(figsize=(4.5, 2.5), dpi = 200) # to create a
    ↵figure and an axes object
ax.plot(x, x, linewidth=2)
ax.plot(x, x**3, 'r+', label = 'cubic function')
ax.plot(x, np.cos(np.pi * x), 'b-.', label = 'cosine function',
    ↵marker='d', markersize = 5)
ax.set_xlabel('time', fontsize='small')
ax.set_ylabel('amplitude', fontsize='small')
ax.set_title('time-amplitude', fontsize='small')
ax.legend(fontsize='small')
ax.tick_params(axis='both', labelsize=7) # to adjust the size of tick
    ↵labels on both axes
```

pyplot-style—case of multiple figures with/without subplots: To create multiple figures, we can use `figure()` with an increasing index (integer) as the input to refer to the figures. To create subplots, we can use `subplot()` (not to be confused with `subplots()`). It receives three integers that show `nrows` (number of rows for the subplot), `ncols` (number of columns for the subplot), and the `plot_index` where `plot_index` could be from 1 to `nrows × ncols` to show the current plot (similar to Matlab, it relies on the concept of current plots):

```
x = np.linspace(0, 1, 20)

plt.figure(1, figsize=(4.5, 4), dpi = 200) # to create the first figure
plt.subplot(2, 1, 1) # the same as subplot(2,1,1)
plt.suptitle('time-amplitude', fontsize='small')
```

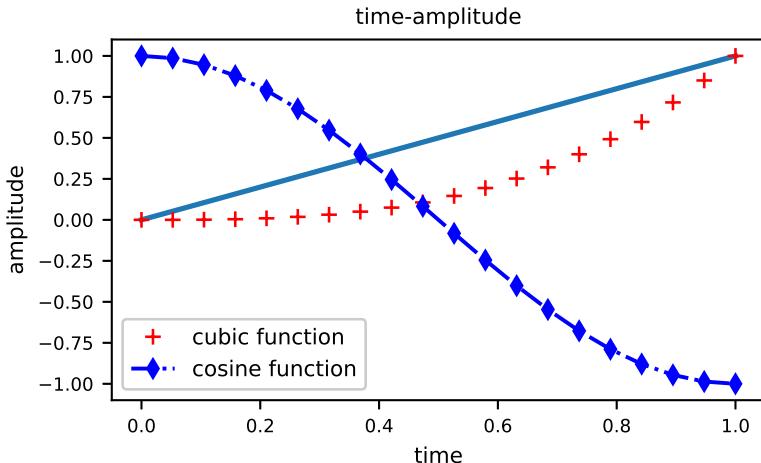


Fig. 3.3: The output of the above plotting example. Here we used OO-style.

```

plt.plot(x, x**3, 'r+', label = 'cubic function')
plt.legend(fontsize='small')
plt.ylabel('amplitude', fontsize='small')
plt.tick_params(axis='both', labelsize=7)

plt.subplot(212)
plt.plot(x, np.cos(np.pi * x), 'b-.', label = 'cosine function', marker='d', markersize = 5)
plt.plot(x, x, linewidth=2, label = 'linear')
plt.legend(fontsize='small')
plt.ylabel('amplitude', fontsize='small')
plt.xlabel('time', fontsize='small')
plt.tick_params(axis='both', labelsize=7)

plt.figure(2, figsize=(4.5, 2), dpi = 200) # to create the second figure
plt.plot(x, x**2, linewidth=2)
plt.title('time-ampl', fontsize='small')
plt.tick_params(axis='both', labelsize=7)

```

OO-style—case of multiple figures with/without subplots: Here we use `subplots()` from `pyplot` (recall that we used it before to create the simple OO-style plot too). It receives two integers `nrow` and `ncol` with defaults being 1 (no third integer because it does not rely on the concept of current plot) and if we use it with no input (e.g., `plt.subplots()`) it just creates one figure and one axis:

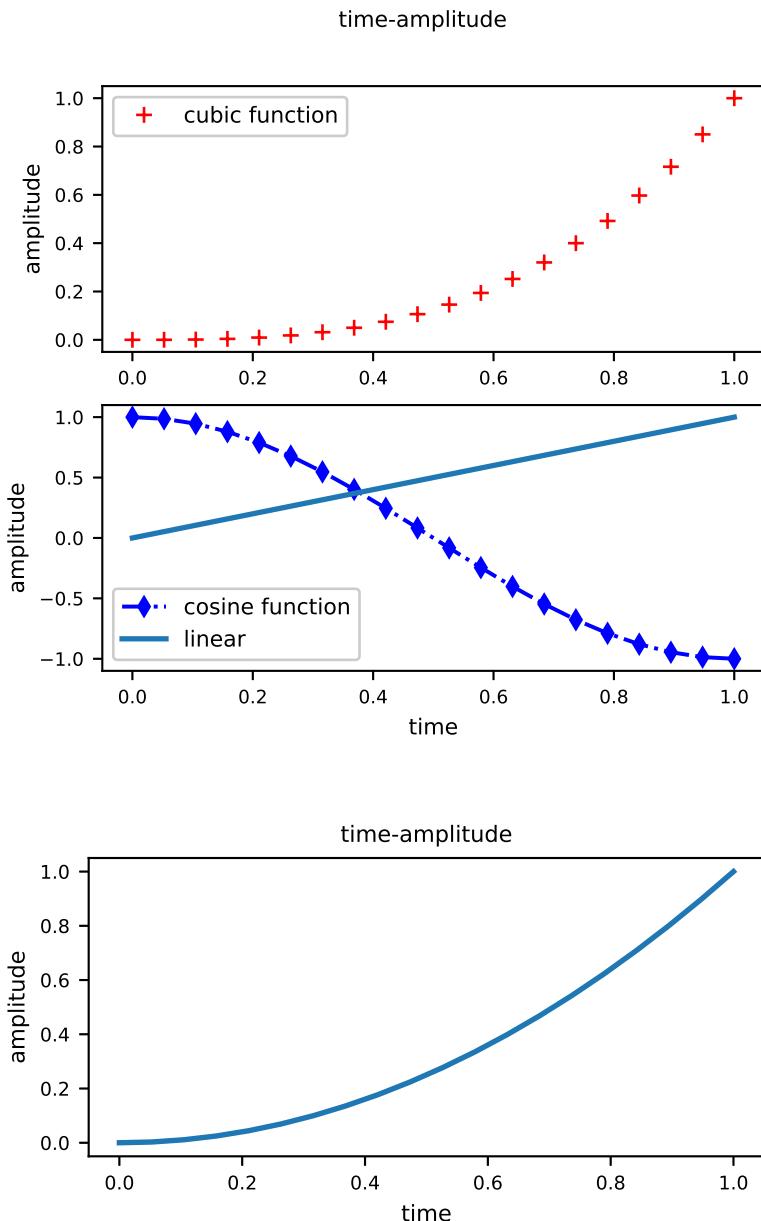


Fig. 3.4: The output of the above plotting example. Here we used pyplot-style.

```

x = np.linspace(0, 1, 20)

fig1, axs = plt.subplots(2, 1, figsize=(4.5, 3.5), dpi = 200)
# fig, (ax1, ax2) = plt.subplots(2, 1) # this is another form in which
# we use sequence unpacking
fig1.suptitle('time-amplitude', fontsize='small')

axs[0].plot(x, x**3, 'r+', label = 'cubic function')
axs[0].legend(fontsize='small')
axs[0].set_ylabel('amplitude', fontsize='small')
axs[0].tick_params(axis='both', labelsize=7)
axs[1].plot(x, np.cos(np.pi * x), 'b-.', label = 'cosine function',_
marker='d', markersize = 5)
axs[1].plot(x, x, linewidth=2, label = 'linear')
plt.legend(fontsize='small')
axs[1].set_ylabel('amplitude', fontsize='small')
axs[1].set_xlabel('time', fontsize='small')
axs[1].tick_params(axis='both', labelsize=7)

fig2, ax = plt.subplots(figsize=(4.5, 2), dpi = 200) # to use the
# default values of 1 x 1
ax.plot(x, x**2, linewidth=2)
ax.set_title('time-ampl', fontsize='small')
ax.tick_params(axis='both', labelsize=7)

```

3.3.3 Two Instructive Examples

Example 3.4 A scatter plot is typically a representation of two dimensional observations in which each observation is represented by a point with Cartesian coordinates being the values of the first and the second variables. These plots are helpful in the sense that they provide us with an idea of how the data is distributed. In this example, we would like to generate a scatter plot of some synthetic (artificial) data and a line that separates them to some extent. Here are a few things to know:

- 1) The data is generated from two bivariate Gaussian (normal) distributions with given parameters (means and covariance matrices). For sampling these distributions, we use `rvs` from `scipy.stats.multivariate_normal` (see ([scipy-normal, 2023](#)) for full documentation);
- 2) Here the scatter plot is generated using `plot()` function itself (although there are other ways such as `pyplot.scatter()`); and
- 3) The line separating the data is given by $2x + 3y + 1 = 0$. Here we assume this line is given to us. Later we will see that a major goal in machine learning is in

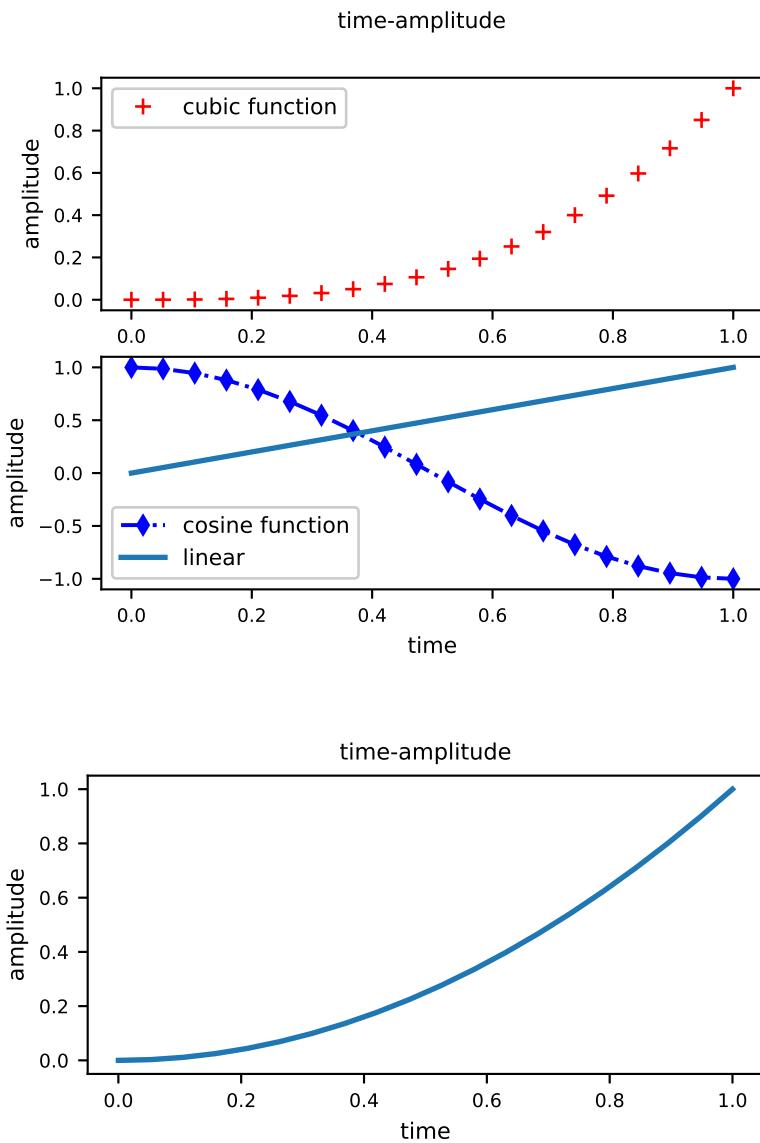


Fig. 3.5: The output of the above plotting example. Here we used OO-style.

fact devising algorithms that can estimate the parameters of a line (or in general, parameters of nonlinear mathematical functions) that can separate the data from different groups in some mathematical sense.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import multivariate_normal as mvn
np.random.seed(100)
m0 = np.array([-1,-1])
m1= np.array([1,1])
cov0 = np.eye(2)
cov1 = np.eye(2)

n = 15 # number of observations from each Gaussian distribution
X0 = mvn.rvs(m0,cov0,n)
x0, y0 = np.split(X0,2,1)
X1 = mvn.rvs(m1,cov1,n)
x1, y1 = np.split(X1,2,1)

plt.style.use('seaborn')
plt.figure(figsize=(4,3), dpi=150)
plt.plot(x0,y0,'g.',label='class 0', markersize=6)
plt.plot(x1,y1,'r.', label='class 1', markersize=6)
plt.xticks(fontsize=8)
plt.yticks(fontsize=8)
lim_left, lim_right = plt.xlim()
plt.plot([lim_left, lim_right],[-lim_left*2/3-1/3,-lim_right*2/3-1/3], 'k', linewidth=1.5) # to plot the line
```

A few points about the above code:

- Notice how we drew the line. We converted $2x + 3y + 1 = 0$ to $y = -\frac{2}{3}x - \frac{1}{3}$ and then created two lists as the input to the `plot()` function: `[lim_left, lim_right]` provides the line x limits, whereas the y coordinates corresponding to the x limits are provided by `[-lim_left*2/3-1/3, -lim_right*2/3-1/3]`.
- We used the `seaborn` default style to plot. Seaborn is a more recent visualization package on top of `matplotlib`. Change `plt.style.use('seaborn')` to `plt.style.use('default')` and observe the difference. Feel free to use any style.
- Observe the figure size and the dpi (dots per inch): `plt.figure(figsize=(4,3), dpi=150)`. Feel free to rely on default values. ■

Example 3.5 In this example, we would like to first create a 10×10 grid with x and y coordinates ranging from 0 to 9. Then we write a function that assigns a label to each point in this grid such that if the sum of x and y coordinates for each point in the grid is less than 10, we assign a label 0 and if is greater or equal to 10, we

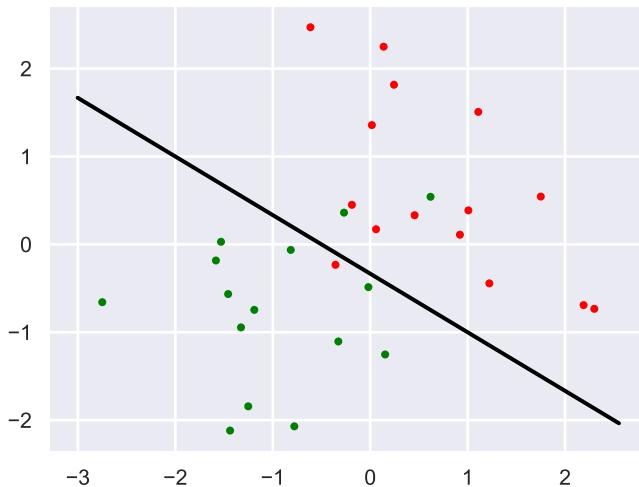


Fig. 3.6: The output of the code written for Example 3.4 .

assign a label 1. Then we create a color plot such that labels of 0 and 1 are colored differently—here we use ‘aquamarine’ and ‘bisque’ colors (see a list of matplotlib colors at ([matplotlib-colors, 2023](#))). A few things to know:

- 1) This example becomes helpful later when we would like to plot the decision regions of a classifier. We will see that similar to the labeling function here, a binary classifier also divides the feature space (for example, our 2D space) into two regions over which labels are different;
- 2) To create the grid, we use `numpy.meshgrid()`. Observe what `X` and `Y` are and see how they are mixed to create the coordinates; and
- 3) To color our 2D space defined by our grid, we use `pyplot.pcolormesh()` with `x` and `y` coordinates in `X` and `Y`, respectively, and an array `Z` of the same size as `X` and `Y`. The values of `Z` will be mapped to color by a `Colormap` instance given by `cmap` parameter. For this purpose, the minimum and maximum values of `Z` will be mapped to the first and last element of the `Colormap`. However, here our color map has only two elements and `Z` also includes 0 and 1 so the mapping is one-to-one.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.arange(0, 10, 1)
y = np.arange(0, 10, 1)
X, Y = np.meshgrid(x, y)
```

X

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

```
Y
```

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

```
coordinates = np.array([X.ravel(), Y.ravel()]).T  
coordinates[0:15]
```

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

```
def f(W):  
    return (W.sum(axis=1) >= 10).astype(int)
```

```
Z = f(coordinates)
Z
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0,
       0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1,
       1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

```
from matplotlib.colors import ListedColormap
color = ('aquamarine', 'bisque')
cmap = ListedColormap(color)
cmap
fig, ax = plt.subplots(figsize=(3,3), dpi=150)
ax.tick_params(axis = 'both', labelsize = 6)
Z = Z.reshape(X.shape)
plt.pcolormesh(X, Y, Z, cmap = cmap, shading='nearest')
```

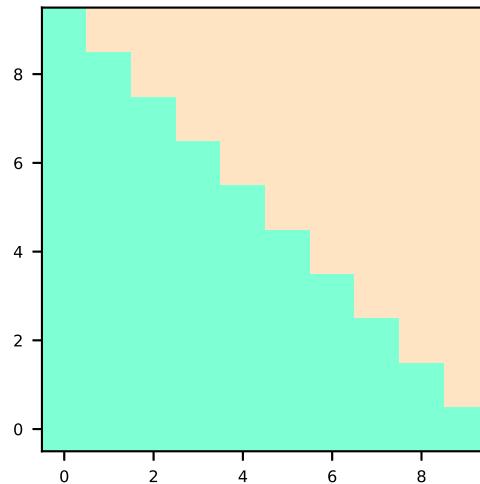


Fig. 3.7: The output of the plotting code written for Example 3.5.

Exercises:

Exercise 1: The following wikipedia page includes a list of many machine learning datasets:

https://en.wikipedia.org/wiki/List_of_datasets_for_machine-learning_research

Extract the table of datasets for “News articles” only and then find the minimum sample size used in datasets used in this table (i.e., the minimum of column “Instances”). In doing so,

- 1) Do not extract all tables to choose this one from the list of tables. Make sure you *only* extract this table. To do so, use something unique in this table and use it as for the `match` argument of `pandas.read_html()`;
- 2) The column “Instances” should be converted to numeric. For doing so, use `pandas.to_numeric()` function. Study and properly use the `errors` parameter of this function to be able to complete this task.
- 3) See an appropriate `pandas.DataFrame` method to find the minimum.

Exercise 2: The “data” folder, contains `GenomicData_orig.csv` file. This file contains a real dataset taken from Gene Expression Omnibus (GEO), <https://www.ncbi.nlm.nih.gov/geo/> with accession ID GSE37745. The original data contains gene expression taken from 196 individuals with either squamous cell carcinoma (66 individuals), adenocarcinoma (106), or large cell carcinoma (24); however, we have chosen a subset of these individuals and also artificially removed some of the measurements to create the effect of missing data.

- 1) Use an appropriate reader function from `pandas` to read this file and show the first 10 rows.
- 2) What are the labels of the first column (index 0) and 20th column (index 19)?
Hint: use the `columns` attribute of the `DataFrame`
- 3) drop the first two columns (index 0 and 1) because they are sample ID info and we do not need. Hint: use `pandas.DataFrame.drop()` method. Read documentation to use it properly. You can use Part 2 to access the labels of these columns to use as input to `pandas.DataFrame.drop()`
- 4) How many missing values are in this dataset?
Hint: first use `pandas.DataFrame.isnull()` to return a `DataFrame` with `True` for missing values and `False` otherwise, and then either use `numpy.sum()` or `pandas.DataFrame.sum()`. Use `sum()` two times.
- 5) Use `DataFrame.fillna()` method to fill the missing values with 0 (filling missing values with some values is known as *imputing* missing values). Use the method in Part 4 to show that there is no missing value now.

Exercise 3: Modify Example 3.5 presented before in this chapter to create the following figure:

To do so,

- 1) use `W.sum(axis=1) >= 10` condition (similar to Example 3.5) along with another suitable condition that should be determined to plot the *stripe* shape;
- 2) do not explicitly plot lines as we did in the Example 3.4. What appears in this figure as lines is the “stairs” shape as in Example 3.5, but with a much larger grid in the same range of x and y (i.e., a finer grid). Therefore, it is

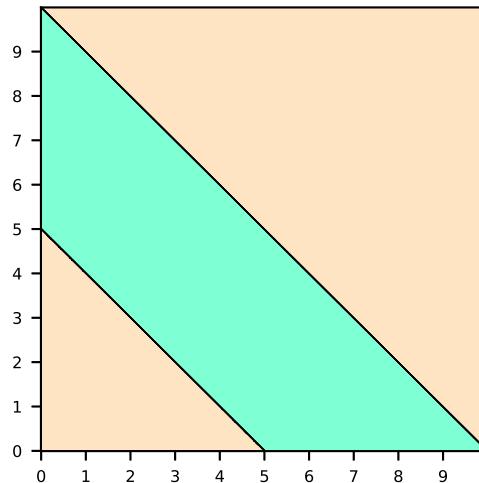


Fig. 3.8: The output of Exercise 3.

required to first redefine the grid (for example, something like `arange(0, 10, 0.01)`), and then to color these boundaries between regions as black use the `pyplot.contour()` function with appropriate inputs (`X`, `Y`, and `Z`, and black colors).

Exercise 4: Write a piece of code that creates a numpy array of shape (2,4,3,5,2) with all elements being 4?

Exercise 5: Suppose `a` and `b` are two numpy arrays. Looking at `shape` attribute of array `a` leads to

(2, 5, 2, 1)

and looking at the `shape` attribute of array `b` leads to

(5, 1, 4)

Which of the following options are correct about `c = a + b`?

- A) `a + b` raises an error because `a` and `b` are not broadcastable
- B) `a` and `b` are broadcastable and the `shape` of `c` is (5, 2, 4)
- C) `a` and `b` are broadcastable and the `shape` of `c` is (2, 5, 2, 4)
- D) `a` and `b` are broadcastable and the `shape` of `c` is (5, 5, 4, 1)
- E) `a` and `b` are broadcastable and the `shape` of `c` is (2, 5, 2, 1)

Exercise 6: Suppose `a` and `b` are two numpy arrays. Looking at `shape` attribute of array `a` leads to

(3, 5, 2, 4, 8)

and looking at the `shape` attribute of array `b` leads to

(2, 4, 1)

Which of the following options are correct about $c = a - b$?

- A) a and b are broadcastable and the shape of c is (3, 5, 2, 4, 8)
- B) $a - b$ raises an error because a and b are not broadcastable
- C) a and b are broadcastable and the shape of c is (1, 1, 2, 4, 1)
- D) a and b are broadcastable and the shape of c is (3, 5, 2, 4, 1)

Exercise 7: We have two Series defined as follows: available:

```
import pandas as pd
s1 = pd.Series([11, 25, 55, 10], index = ['g', 'r', 'p', 'a'])
s2 = pd.Series([20, 30, 40])
```

We create a DataFrame by:

```
df = pd.DataFrame({"col1": s1, "col2": s2})
```

- a) What is `len(df.index)`? Why?
- b) Why are all the NaN values created in df?

Exercise 8: What is the output of the following code snippet:

```
import numpy as np
a = np.arange(1, 11)
b = np.arange(0, 10, 2)
np.sum(a[b])
```

- a) 25
- b) 20
- c) 15
- d) the code raises an error

Exercise 9: What is the output of each of the following code snippets:

a)

```
a = np.arange(1, 5)
b = a[:2]
b[0] = 10
print(a)
```

b)

```
a = np.arange(1, 5)
b = np.arange(2)
c = a[b]
```

```
c[0] = 10
print(a)
```

Exercise 10: Suppose we have a numpy array “a” that contains about 10 million integers. We would like to multiply each element of a by 2. Which of the following two codes is better to use? Why?

Code A)

```
for i, v in enumerate(a):
    a[i] *= 2
```

Code B)

```
a *= 2
```

Exercise 11: Which of the following options can not create a `DataFrame()`? Explain the reason (assume pandas is imported as pd).

a)

```
dic = {"A": [10, 20, 30, 40], "B": [80, 90, 100, None]}
pd.DataFrame(dic, index=['z', 'd', 'a', 'b', 'f'])
```

b)

```
dic = {"A": [10, 20, 30, 40], "B": [80, 90, 100, None]}
pd.DataFrame(dic)
```

c)

```
dic = {"A": [10, 20, 30, 40], "B": [80, 90, None]}
pd.DataFrame(dic)
```

d)

```
import numpy as np
dic = {"A": np.array([10, 20, 30, 40]), "B": [80, 90, 100, None]}
pd.DataFrame(dic, index=['z', 'd', 'a', 'b'])
```

Exercise 12: A DataFrame named df is created as follows:

```
list1 = [{1: 'a', 2: 'b'}, {3: 'c'}, {1: 'd', 2: 'e'}]
df = pd.DataFrame(list1, index=[1, 2, 3])
```

```
      1    2    3  
1  a   b  NaN  
2  NaN  NaN    c  
3  d   e  NaN
```

In each of the following parts, determine whether the indexing is legitimate, and if it is, determine the output:

a)

```
df.loc[1,1]
```

b)

```
df[1]
```

c)

```
df[1][1]
```

d)

```
df[1].iloc[1]
```

e)

```
df[1].loc[1]
```

f)

```
df[1,1]
```



Chapter 4

Supervised Learning in Practice: the First Application Using Scikit-Learn

In this chapter, we first formalize the idea of supervised learning and its main two tasks, namely, classification and regression, and then provide a brief introduction to one of the most popular Python-based machine learning software, namely, Scikit-Learn. After this introduction, we start with a classification application with which we introduce and practice with some important concepts in machine learning such as data splitting, normalization, training a classifier, prediction, and evaluation. The concepts introduced in this chapter are important from the standpoint of the practical machine learning because they are used frequently in the design process.

4.1 Supervised Learning

Supervised learning is perhaps the most common type of machine learning in which the goal is to learn a mapping between a vector of input variables (also known as *predictors* or *feature vector*, which is a vector of measurable variables in a problem) and output variables. To guide the learning process, the assumption in supervised learning is that a set of input-output instances, also referred to as training data, is available. An output variable is generally known as *target*, *response*, or *outcome*, and while there is no restriction on the number of output variables, for the ease of notation and discussion, we assume a single output variable, denoted y , is available. While it is generally easy and cheap to measure feature vectors, it is generally difficult or costly to measure y . This is the main reason that in supervised learning practitioners go through the pain of assigning the outcomes to input vectors once (i.e., collecting the training data) and hope to learn a function that can perform this mapping in the future.

Suppose we have a set of training data $\mathbf{S}_{tr} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$ where $\mathbf{x}_i \in \mathbb{R}^p, i = 1, \dots, n$, represents a vector including the values of p feature variables (feature vector), y_i denotes the target value (outcome) associated with \mathbf{x}_i , and n is the number of observations (i.e., sample size) in the training data. In classification, possible values for y_i belong to a set of predefined finite categories called *labels* and

the goal is to assign a given (realization of random) feature vector (also known as observation or instance) to one of the class labels (in some applications known as *multilabel classification*, multiple labels are assigned to one instance). In regression, on the other hand, y_i represents realizations of a numeric random variable and the goal is to estimate the target value for a given feature vector. Whether the problem is classification or regression, the goal is to estimate the value of the target y for a given feature vector \mathbf{x} . In machine learning, we refer to this estimation problem as prediction; that is, predicting y for a given \mathbf{x} . At the same time, in classification and regression, we refer to the mathematical function that performs this mapping as classifier and regressor, respectively.

4.2 Scikit-Learn

Scikit-Learn is a Python package that contains an efficient and uniform API to implement many machine learning (ML) methods. It was initially developed in 2007 by David Cournapeau and was made available to public in 2010. The Scikit-Learn API is neatly designed based on a number of classes. Three fundamental objects in scikit-learn are *Estimators*, *Transformers*, and *Predictors*:

- **Estimators:** Any object that can estimate some parameters based on a dataset is called an *estimator*. All ML models, whether classifiers or regressors, are implemented in their own `Estimator` class. For example, a ML algorithm known as k -nearest neighbors (kNN) rule is implemented in the `KNeighborsClassifier` class in the `sklearn.neighbors` module. Or another algorithm known as perceptron is implemented in the `Perceptron` class in the `linear_model` module. All estimators implement the `fit()` method that takes either one argument (data) or two as in supervised learning where the second argument represents target values:

```
estimator.fit(data, targets)
```

or

```
estimator.fit(data)
```

The `fit()` method performs the estimation from the given data.

- **Transformers:** Some estimators can also transform data. These estimators are known as transformers and implement `transform()` method to perform the transformation of data as:

```
new_data = transformer.transform(data)
```

Transformers also implement a convenient method known as `fit_transform()`, which is similar (sometimes more efficient) to calling `fit()` and `transform()`

back-to-back:

```
new_data = transformer.fit_transform(data)
```

- **Predictors:** Some estimators can make predictions given a data. These estimators are known as predictors and implement `predict()` method to perform prediction:

```
prediction = predictor.predict(data)
```

Classification algorithms generally implement

```
probability = predictor.predict_proba(data)
```

that provides a way to quantify the certainty (in terms of probability) of a prediction for a given data.

4.3 The First Application: Iris Flower Classification

In this application, we would like to train a ML classifier that receives a feature vector containing morphologic measurements (length and width of petals and sepals in centimeters) of Iris flowers and classifies a given feature vector to one of the three Iris flower species: Iris setosa, Iris virginica, or Iris versicolor. The underlying hypothesis behind this application is that an Iris flower can be classified into its species based on its petal and sepal lengths and widths. Therefore, our feature vectors \mathbf{x}_i that are part of training data are four dimensional ($p = 4$), and y can take three values; therefore, we have a *multiclass classification* (three-class) problem.

Our training data is a well-known dataset in statistics and machine learning, namely, *Iris dataset*, that was collected by one of the most famous biologist-statistician of 20th century, Sir Ronald Fisher. This dataset is already part of scikit-learn and can be accessed by importing its `datasets` module as follows:

```
from sklearn import datasets # sklearn is the Python name of  
# scikit-learn  
iris = datasets.load_iris()  
type(iris)
```

```
sklearn.utils.Bunch
```

Datasets that are part of scikit-learn are generally stored as “Bunch” objects; that is, an object of class `sklearn.utils.Bunch`. This is an object that contains the actual data as well as some information about it. All these information are stored in Bunch objects similar to dictionaries (i.e., using `keys` and `values`). Similar to dictionaries, we can use `key()` method to see all keys in a Bunch object:

```
iris.keys()
```

```
dict_keys(['data', 'target', 'frame', 'target_names', 'DESCR',
          'feature_names', 'filename'])
```

The value of the key DESCR gives some brief information about the dataset. Nevertheless, compared with dictionaries, in Bunch object we can also access values as `bunch.key` (or, equivalently, as in dictionaries by `bunch['key']`). For example, we can access class names as follows:

```
print(iris['target_names']) # or, equivalently, print(iris.target_names)
```

```
['setosa' 'versicolor' 'virginica']
```

Next, we print the first 500 characters in the DESCR field where we see the name of features and classes:

```
print(iris.DESCR[:500])
```

```
.. _iris_dataset:
```

```
Iris plants dataset
```

```
-----
```

```
**Data Set Characteristics:**
```

```
:Number of Instances: 150 (50 in each of three classes)
:Number of Attributes: 4 numeric, predictive attributes and the class
:Attribute Information:
    - sepal length in cm
    - sepal width in cm
    - petal length in cm
    - petal width in cm
    - class:
        - Iris-Setosa
        - Iris-Versicolour
        - Iris-Virginica
```

All measurements (i.e., feature vectors) are stored as values of `data` key. Here, the first 10 feature vectors are shown:

```
iris.data[:10]
```

```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2],
```

```
[5.4, 3.9, 1.7, 0.4],  
[4.6, 3.4, 1.4, 0.3],  
[5., 3.4, 1.5, 0.2],  
[4.4, 2.9, 1.4, 0.2],  
[4.9, 3.1, 1.5, 0.1]])
```

We refer to the matrix containing all feature vectors as *data matrix* (also known as *feature matrix*). By convention, scikit-learn assumes this matrix has the shape of sample size \times feature size; that is, the number of observations (also sometimes referred to as the number of samples) \times the number of features. For example, in this data there are 150 Iris flowers and for each there are 4 features; therefore, the shape is 150×4 :

```
iris.data.shape
```

```
(150, 4)
```

The corresponding targets (in the same order as the feature vectors stored in `data`) can be accessed through the `target` field:

```
iris.target
```

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,  
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,  
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

The three classes in the dataset, namely, setosa, versicolor, and virginica are *encoded* as integers 0, 1, and 2, respectively. Although there are various encoding schemes to transform categorical variables into their numerical counterparts, this is known as integer (ordinal) encoding (also see Exercise 3).



For feature vectors encoding is generally needed as many machine learning methods are designed to work with numeric data. Although some ML models also require encoded labels, even for others that do not have this requirement, we generally encode labels because: 1) a uniform format for the data helps comparing different types of models regardless of whether they can work with encoded labels or not; and 2) numeric values can use less memory.

Here we use `bincount` function from NumPy to count the number of samples in each class:

```
import numpy as np  
np.bincount(iris.target)
```

```
array([50, 50, 50])
```

As seen here, there are 50 observations in each class. Here we check whether the type of data matrix and target is “array-like” (e.g., numpy array or pandas DataFrame), which is the expected type of input data for scikit-learn estimators:

```
print('type of data: ' + str(type(iris.data))+ '\n' +  
      'type of target: ' +  
      str(type(iris.target)))
```

```
type of data: <class 'numpy.ndarray'>  
type of target: <class 'numpy.ndarray'>
```

And finally we can see the feature names as follows:

```
print(iris.feature_names) # the name of features
```

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal  
width (cm)']
```

4.4 Test Set for Model Assessment

Sometimes before training a machine learning model, we need to think a bit in advance. Suppose we train a model based on the given training data. A major question to ask is how well the model performs. This is a critical question that shows the predictive capacity of the trained model. In other words, the entire practical utility of the trained model is summarized in its metrics of performance. To answer this question, there are various evaluation rules. As discussed in Section 1.3.6, perhaps the most intuitive estimator is to evaluate a trained model on a test data, also known as *test set*. However, in contrast with the EEG application discussed in Section 1.3, here we only have one dataset. Therefore, we have to simulate the effect of having test set. In this regard, we randomly split the given data into a training set (used to train the model) and a test set, which is used for evaluation. Because here the test set is held out from the original data, it is also common to refer to that as holdout set.

In order to split the data, we can use `train_test_split` function from the `sklearn.model_selection` module (see ([Scikit-split, 2023](#)) for full specification). The `train_test_split` function receives a variable number of inputs (identified by `*array` in its documentation) and as long as they all have the same number of `samples`, it returns a list containing the train-test splits. The few following points seems worthwhile to remember:

1. In using scikit-learn, we conventionally use `X` and `y` to refer to a data matrix and targets, respectively;
2. Although `train_test_split` function can receive an arbitrary number of sequences to split, in supervised learning it is common to use this function with a data matrix and targets only;

3. Although `train_test_split` function returns a list, it is common to use sequence unpacking (see Section 2.6.2) to name the results of the split;
4. By default `train_test_split` function shuffles the data before splitting. This is a good practice to avoid possible systematic biases in the data; however, to be able to reproduce the results, it is also a good practice to set the seed of random number generator. This is done by setting `random_state` to an integer;
5. The `test_size` argument of the function represents the proportion of data that should be assigned to the test set. The default value of this parameter is 0.25, which is a good rule of thumb if no other specific proportion is desired; and
6. It is a good practice to keep the proportion of classes in both the training and the test sets as in the whole data. This is done by setting the `stratify` to variable representing the target.

Here, we use stratified random split to divide the given data into 80% training and 20% test:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test= train_test_split(iris.data, iris.
    ↵target, random_state=100, test_size=0.2, stratify=iris.target)
print('X_train_shape: ' + str(X_train.shape) + '\nX_test_shape: ' +_
    ↵str(X_test.shape) \
    + '\ny_train_shape: ' + str(y_train.shape) + '\ny_test_shape: ' +_
    ↵str(y_test.shape))
```

```
X_train_shape: (120, 4)
X_test_shape: (30, 4)
y_train_shape: (120,)
y_test_shape: (30,)
```

`X_train` and `y_train` are the feature matrix and the target values used for training, respectively. The feature matrix and the corresponding targets for evaluation are stored in `X_test` and `y_test`, respectively. Let us count the number of class-specific observations in the training data:

```
np.bincount(y_train)
```

```
array([40, 40, 40])
```

This shows that the equal proportion of classes in the given data is kept in both the training and the test sets.

4.5 Data Visualization

For datasets in which the number of variables is not really large, visualization could be a good exploratory analysis—it could reveal possible abnormalities or could provide us with an insight into the hypothesis behind the entire experiment. In this

regard, scatter plots can be helpful. We can still visualize scatter plots for three variables but for more than that, we need to display scatter plots between all pairs of variables in the data. These exploratory plots are known as *pair plots*. There are various ways to plot pair plots in Python but an easy and appealing way is to use `pairplot()` function from `seaborn` library. As this function expects a `DataFrame` as the input, we first convert the `X_train` and `y_train` arrays to dataframes and concatenate them.

```
import pandas as pd
X_train_df = pd.DataFrame(X_train, columns=iris.feature_names)
y_train_df = pd.DataFrame(y_train, columns=['class'])
X_y_train_df = pd.concat([X_train_df, y_train_df], axis=1)
```

The pair plots are generated by:

```
import seaborn as sns
sns.pairplot(X_y_train_df, hue='class', height=2) # hue is set to the
# class variable in the dataframe so that they are plotted in
# different color and we are able to distinguish classes
```

Fig. 4.1 presents the scatter plot of all pairs of features. The plots on diagonal show the histogram for each feature across all classes. The figure, at the very least, shows that classification of Iris flower in the collected sample is plausible based on some of these features. For example, we can observe that class-specific histograms generated based on petal width feature are fairly distinct. Further inspection of these plots suggests that, for example, petal width could potentially be a better feature than sepal width in discriminating classes. This is because the class-specific histograms generated by considering sepal width are more mixed when compared with petal width histograms. We may also be able to make similar suggestions about some bivariate combinations of features. However, it is not easy to infer much about higher order feature dependency (i.e., multivariate relationship) from these plots. As a result, although visualization could also be used for selecting discriminating feature subsets, it is generally avoided because we are restricted to low-order dependency among features, while in many problems higher-order feature dependencies lead to an acceptable level of prediction. On the other hand, in machine learning there exist feature subset selection methods that are entirely data-driven and can detect low- or high-order dependencies among features. These methods are discussed in Chapter 10. In what follows, we consider all four features to train a classifier.

4.6 Feature Scaling (Normalization)

It is common that the values of features in a dataset come in different scales. For example, the range of some features could be from 0 to 1, while others may be in the order of thousands or millions depending on what they represent and how they are measured. In these cases, it is common to apply some type of *feature scaling*

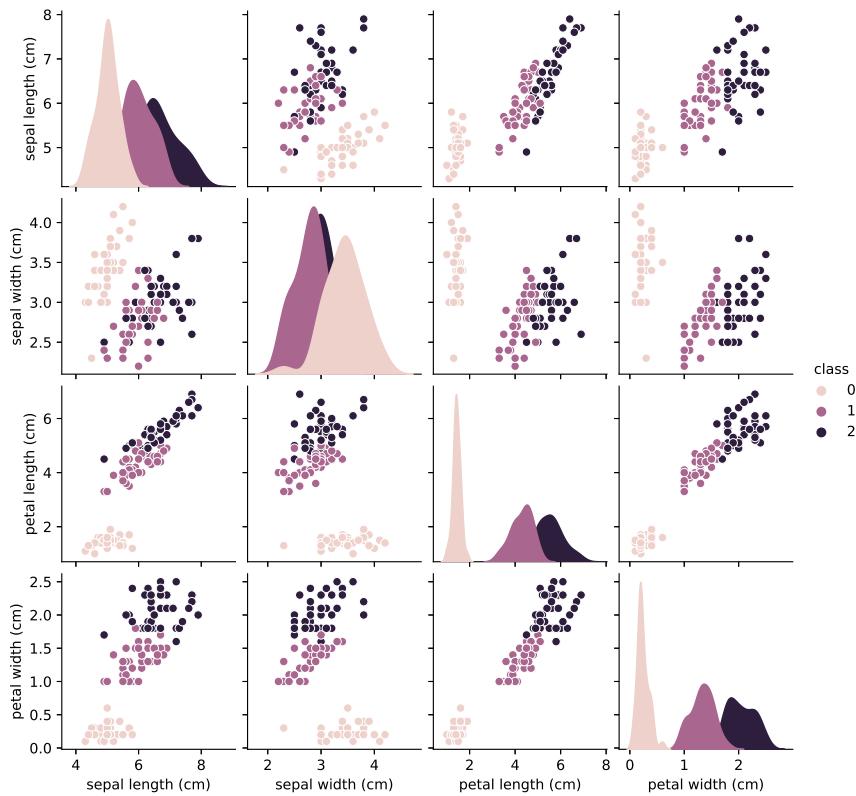


Fig. 4.1: Pair plots generated using training set in the Iris classification application

(also known as normalization) to the training data to make the scale of features “comparable”. This is because some important classes of machine learning models such as neural networks, kNNs, ridge regression, to just name a few, benefit from feature scaling.



Even if we are not sure whether a specific ML rule benefits from scaling or not, it would be helpful to still scale the data because scaling is not harmful and, at the same time, it facilitates comparing different types of models regardless of whether they can or can not benefit from scaling.

Two common ways for feature scaling include *standardization* and *min-max scaling*. In standardization, first the mean and the standard deviation of each feature is found. Then for each feature, the mean of that feature is subtracted from all feature values and the result of this subtraction is divided by the standard deviation of the feature. This way the feature vector is centered around zero and will have a standard

deviation of one. In min-max scaling, a similar subtraction and division is performed except that the mean and the standard deviation of each feature are replaced with the minimum and the range (i.e., maximum - minimum) of that feature, respectively. This way each feature is normalized between 0 and 1. Although all these operations could be easily achieved by naive Python codes, scikit-learn has a transformer for this purpose. Nonetheless, in what follows we first see how standardization is done using few lines of codes that use built-in NumPy `mean()` and `std()` functions. Then we deploy a scikit-learn transformer for doing that.

The following code snippet finds the mean and the standard deviation for each feature in the training set:

```
mean = X_train.mean(axis=0) # to take the mean across rows (for each ↵column)
std = X_train.std(axis=0) # to take the std across rows (for each ↵column)
X_train_scaled = X_train - mean # notice the utility of broadcasting
X_train_scaled /= std
```

Next, we observe that the mean of each feature is 0 (the reason for having such small numbers instead of absolute 0 is the finite machine precision):

```
X_train_scaled.mean(axis=0) # observe the mean is 0 now
```

```
array([-4.21884749e-16,  1.20042865e-16, -3.88578059e-16, -7. ↵04991621e-16])
```

And the standard deviations are 1:

```
X_train_scaled.std(axis=0) # observe the std is 1 now
```

```
array([1., 1., 1., 1.])
```

It is important to note that the “test set” should not be used in any stage involved in training of our classifier, not even in a preprocessing stage such as normalization. The main reason for this can be explained through the following four points:

- Point 1: when a classifier is trained, the most salient issue is the performance of the classifier on *unseen (future)* observations collected from the same application. In other words, the entire worth of the classifier depends on its performance on unseen observations;
- Point 2: because “unseen” observations, as the name suggests, are not available to us during the training, we use (the available) test set to simulate the effect of unseen observations for evaluating the trained classifier;
- Point 3: as a result of Point 2, in order to have an unbiased evaluation of the classifier using test set, the classifier should classify observations in the test set in precisely the same way it is used to classify unseen future observations; and
- Point 4: unseen observations are not available to us and naturally they can not be used in any training stage such as normalization, feature selection, etc.; therefore, observations in the test set should not be used in any training stage either.

In this regard, a common *mistake* is to apply some data preprocessing such as normalization *before* splitting the entire data into training and test sets. This causes *data leakage*; that is, some information about the test set leak into the training process. Based on the aforementioned Point 4, this is not legitimate because here test set is used to determine the statistics used for normalizing the entire data and, therefore, it is used in training the model, while unseen future observations can not be naturally used in this way. In these situations, even referring to the set that is supposed to be used for evaluation after the split as “test set” is not legitimate because that set was already used in training through normalization.



It is important to note that here we refer to “normalizing the entire data before splitting” as an illegitimate practice from the standpoint of *model evaluation using test data*, which is the most common reason to split the data into two sets. However, a similar practice could be legitimate if we view it from the standpoint of:

1) *training a classifier in semi-supervised learning fashion*. An objective in semi-supervised learning is to use both labeled and unlabeled data to train a classifier. Suppose we use the entire data for normalization and then we divide that into two sets, namely Set A and Set B. Set A (its feature vectors and their labels) is then used to train a classifier and Set B is not used at this stage. This classifier is trained using both labeled data (Set A) and unlabeled data (Set B). This is because for normalization we used both Set A and Set B without using labels, but Set A (and the labels of observation within that set) were used to train the classifier. Although from the standpoint of semi-supervised learning we have used a labeled set and unlabeled set in the process of training the classifier, it is perhaps a “dumb” training procedure because all valuable labels of observations within Set B are discarded, and this set is only used for normalization purposes. Nevertheless, even after training the classifier using such a semi-supervised learning procedure, we still need an independent test set to evaluate the performance of the classifier.

2) *model evaluation using a performance estimator with unknown properties*. An important property of using test set for model evaluation is that it is an *unbiased* estimator of model performance. This procedure is also known as *test-set estimator* of performance. However, there are other performance estimators. For example, another estimator is known as resubstitution estimator (discussed in Chapter 9). It is simply reusing the training data to evaluate the performance of a trained model. Nonetheless, resubstitution has an undesirable property of being usually strongly optimistically biased. That being said, because resubstitution has an undesirable property, we can not refer to that as an illegitimate estimation rule. When we refer to the practice of “normalizing the data before splitting into two sets and then using one

set in training and the other for evaluation” as an illegitimate practice, what is really meant is that following this procedure and then *framing that as test-set estimator of performance is illegitimate*. If we accept it, however, as another performance metric, one that is less known and is expected to be optimistically biased to some extent, then it is a legitimate performance estimator.

Once the relevant statistics (here, the mean and the standard deviation) are estimated from the training set, they can be used to normalize the test set:

```
X_test_scaled = X_test - mean
X_test_scaled /= std
```

Observe that the test set does not necessarily have a mean of 0 or standard deviation of 1:

```
print(X_test_scaled.mean(axis=0))
print(X_test_scaled.std(axis=0))
```

```
[-0.20547406 -0.25637284 -0.03844742 -0.07491371]
[0.94499306 1.07303169 0.94790626 0.94534006]
```

Although the naive Python implementation as shown above is instructive, as we said earlier scikit-learn has already transformers for implementing both standardization and min-max scaling. These are `StandardScaler` (for standardization) and `MinMaxScaler` (for min-max scaling) classes from `sklearn.preprocessing` module. Here we use `StandardScaler` to perform the standardization. To be able to use these transformers (or any other estimator), we first need to instantiate the class into an object:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

To estimate the mean and the standard deviation of each feature from the training set (i.e., from `X_train`), we call the `fit()` method of the `scaler` object (afterall, any transformer is an estimator and implements the `fit()` method):

```
scaler.fit(X_train)
```

```
StandardScaler()
```

The `scaler` object holds any information that the standardization algorithm implemented in `StandardScaler` class extracts from `X_train`. The `fit()` method returns the `scaler` object itself and modifies that in place (i.e., stores the parameters estimated from data). Next, we call the `transform()` method of the `scaler` object to transform the training and test sets based on statistics extracted from the training set, and use `X_train_scaled` and `X_test_scaled` to refer to the transformed training and test sets, respectively:

```
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

print(X_test_scaled.mean(axis=0)) # observe that we get the same values_
    ↪for the mean and std of test set as in the naive implementation_
    ↪shown above
print(X_test_scaled.std(axis=0))
```

```
[-0.20547406 -0.25637284 -0.03844742 -0.07491371]
[0.94499306 1.07303169 0.94790626 0.94534006]
```

For later use and in order to avoid the above preprocessing steps, the training and testing arrays could be saved using `numpy.save()` to binary files. For NumPy arrays this is generally more efficient than the usual “pickling” supported by `pickle` module. For saving multiple arrays, we can use `numpy.savetxt()`. We can provide our arrays as keyword arguments to this function. In that case, they are saved in a binary file with arbitrary names that we provide as keywords. If we give them as positional arguments, then they will be stored with names being `arr_0`, `arr_1`, etc. Here we specify our arrays as keywords `X` and `y`:

```
np.savetxt('data/iris_train_scaled', X = X_train_scaled, y = y_train)
np.savetxt('data/iris_test_scaled', X = X_test_scaled, y = y_test)
```

`numpy.save` and `numpy.savetxt` add `npy` and `npz` extensions to the name of a created file, respectively. Later, arrays could be loaded by `numpy.load()`. Using `numpy.load()` for `npz` files, returns a dictionary-like object using which each array can be accessed either by the keywords that we provide when saving the file, or by `arr_0`, `arr_1`, . . . , if positional arguments were used during the saving process.

4.7 Model Training

We are now in the position to train the actual machine learning model. For this purpose, we use the *k-nearest neighbors* (kNN) classification rule in its standard form. To classify a test point, one can think that the kNN classifier grows a spherical region centered at the test point until it encloses k training samples, and classifies the test point to the majority class among these k training samples. For example, in Fig. 4.2, 5NN assigns “green” to the test observation because within the 5 nearest observations to this test point, three are from the green class.

kNN classifier is implemented in the `KNeighborsClassifier` class in the `sklearn.neighbors` module. Similar to the way we used the `StandardScaler` estimator earlier, we first instantiate the `KNeighborsClassifier` class into an object:

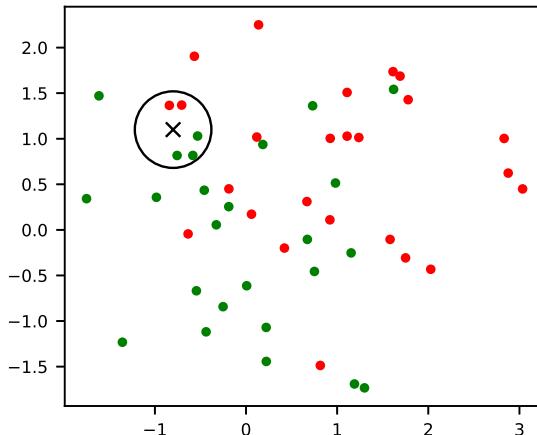


Fig. 4.2: The working principle of kNN (here $k = 5$). The test point is identified by \times . The circle encloses 5 nearest neighbors of the test point.

```
from sklearn.neighbors import KNeighborsClassifier as knn # giving an
#alias KNN for simplicity
knn = KNN(n_neighbors=3) # hyperparameter k is set to 3; that is, we
#have 3NN
```

The constructors of many estimators take as arguments various hyperparameters, which can affect the estimator performance. In case of kNN, perhaps the most important one is k , which is the number of nearest neighbors of a test point. In `KNeighborsClassifier` constructor this is determined by `n_neighbors` parameter. In the above code snippet, we set `n_neighbors` to 3, which implies we are using 3NN. Full specification of `KNeighborsClassifier` class is found at ([Scikit-kNN-C, 2023](#)). To train the model, we call the `fit()` method of the `knn` object. Because this is an estimator used for supervised learning, its `fit()` method expects both the feature matrix and the targets:

```
knn.fit(X_train_scaled, y_train)
```

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

Similar to the use of `fit()` method for `StandardScaler`, the `fit()` method here returns the modified `knn` object.

4.8 Prediction Using the Trained Model

As said before, some estimators in scikit-learn are predictors; that is, they can make prediction by implementing the `predict()` method. `KNeighborsClassifier` is also a predictor and, therefore, implements `predict()`. Here we use this method to make prediction on a new data point. Suppose we have the following data point measured in the original scales as in the original training set `X_train`:

```
x_test = np.array([[5.5, 2, 1.1, 0.6]]) # same as: np.array([5.5, 2, 1,
    ↵1, 0.6]).reshape(1,4)
x_test.shape
```

```
(1, 4)
```

Recall that scikit-learn always assumes two-dimensional NumPy arrays of shape sample size \times feature size—this is the reason in the code above the test point `x_test` is placed in a two-dimensional NumPy array of shape `(1, 4)`. However, before making prediction, we need to scale the test data point using the same statistics used to scale the training set; after all, the `knn` classifier was trained using the transformed dataset and it classifies data points in the transformed space. This is achieved by:

```
x_test_scaled = scaler.transform(x_test)
x_test_scaled
```

```
array([-0.45404756, -2.53437275, -1.50320017, -0.79582245]))
```

Now, we can predict the label by:

```
y_test_prediction = knn.predict(x_test_scaled)
print('knn predicts: ' + str(iris.target_names[y_test_prediction])) #_
    ↵to convert the prediction (y_test_prediction) to the names of Iris_
    ↵flower
```

```
knn predicts: ['versicolor']
```

We can also give several sample points as the argument to the `predict()` method. In that case, we receive the assigned label for each of them:

```
y_test_predictions = knn.predict(X_test_scaled)
print('knn predicts: ' + str(iris.target_names[y_test_predictions])) #_
    ↵fancy indexing in Section 3.1.4
```

```
knn predicts: ['versicolor' 'versicolor' 'versicolor' 'virginica'
    ↵'setosa' 'virginica' 'versicolor' 'setosa' 'versicolor' 'versicolor'
    ↵'versicolor' 'virginica' 'virginica' 'setosa' 'virginica' 'setosa'
    ↵'setosa' 'versicolor' 'setosa' 'virginica' 'setosa' 'versicolor'
    ↵'versicolor' 'setosa' 'versicolor' 'setosa' 'setosa' 'versicolor'
    ↵'virginica' 'versicolor']
```

The above sequence of operations, namely, instantiating the class KNN, fitting, and predicting, can be combined as the following one liner pattern known as *method chaining*:

```
y_test_predictions = KNN(n_neighbors=3).fit(X_train_scaled, y_train).  
    ↪predict(X_test_scaled)  
print('knn predicts: ' + str(iris.target_names[y_test_predictions]))
```

```
knn predicts: ['versicolor' 'versicolor' 'versicolor' 'virginica'  
    ↪'setosa' 'virginica' 'versicolor' 'setosa' 'versicolor' 'versicolor'  
    ↪'versicolor' 'virginica' 'virginica' 'setosa' 'virginica' 'setosa'  
    ↪'setosa' 'versicolor' 'setosa' 'virginica' 'setosa' 'versicolor'  
    ↪'versicolor' 'setosa' 'versicolor' 'setosa' 'setosa' 'versicolor'  
    ↪'virginica' 'versicolor']
```

4.9 Model Evaluation (Error Estimation)

There are various rules and metrics to assess the performance of a classifier. Here we use the simplest and perhaps the most intuitive one; that is, the proportion of misclassified points in a test set. This is known as the test-set estimator of *error rate*. In other words, the proportion of misclassified observations in the test set is indeed an estimate of classification error rate denoted ε , which is defined as *the probability of misclassification by the trained classifier*.

Let us first formalize the definition of error rate for binary classification. Let \mathbf{X} and Y represent a random feature vector and a binary random variable representing the class variable, respectively. Because Y is a discrete random variable and \mathbf{X} is a continuous feature vector, we can characterize the joint distribution of \mathbf{X} and Y (this is known as *joint feature-label distribution*) as:

$$P(\mathbf{X} \in E, Y = i) = \int_E p(\mathbf{x}|Y = i)P(Y = i)d\mathbf{x}, \quad i = 0, 1, \quad (4.1)$$

where $p(\mathbf{x}|Y = i)$ is known as the *class-conditional probability density function*, which shows the (relative) likelihood of \mathbf{X} being close to a specific value \mathbf{x} given $Y = i$, and where E represents an event, which is basically a subset of the sample space to which we can assign probability (in technical terms a Borel-measurable set). Intuitively, $P(\mathbf{X}, Y)$ shows the frequency of encountering particular pairs of (\mathbf{X}, Y) in practice. Furthermore, in (4.1), $P(Y = i)$ is the *prior probability of class i* , which quantifies the probability that a randomly drawn sample from the population of entities across all classes belongs to class i .

Given a training set \mathbf{S}_{tr} , we train a classifier $\psi : \mathbb{R}^p \rightarrow \{0, 1\}$, which maps realizations of \mathbf{X} to realizations of Y . Let $\psi(\mathbf{X})$ denote the act of classifying (realizations of) \mathbf{X} by a specific trained classifier. Because \mathbf{X} is random, $\psi(\mathbf{X})$, which is either 0 or 1, is random as well. Let E_0 denote all events for which $\psi(\mathbf{X})$ gives label 0. A

probabilistic question to ask is what would be the joint probability of all those events and $Y = 1$? Formally, to answer this question, we need to find,

$$P(\mathbf{X} \in E_0, Y = 1) \stackrel{1}{=} \int_{E_0} p(\mathbf{x}|Y = 1)P(Y = 1)d\mathbf{x} \equiv \int_{\psi(\mathbf{x})=0} p(\mathbf{x}|Y = 1)P(Y = 1)d\mathbf{x}, \quad (4.2)$$

where $\stackrel{1}{=}$ is a direct consequence of (4.1). Similarly, let E_1 denote all events for which $\psi(\mathbf{X})$ gives label 1. We can ask a similar probabilistic question; that is, what would be the joint probability of $Y = 0$ and E_1 ? In this case, we need to find

$$P(\mathbf{X} \in E_1, Y = 0) = \int_{E_1} p(\mathbf{x}|Y = 0)P(Y = 0)d\mathbf{x} \equiv \int_{\psi(\mathbf{x})=1} p(\mathbf{x}|Y = 0)P(Y = 0)d\mathbf{x}. \quad (4.3)$$

At this stage, it is straightforward to see that the probability of misclassification ε is indeed obtained by adding probabilities (4.2) and (4.3); that is,

$$\varepsilon = P(\mathbf{X} \in E_0, Y = 1) + P(\mathbf{X} \in E_1, Y = 0) \equiv P(\psi(\mathbf{X}) \neq Y). \quad (4.4)$$

Nonetheless, in practical settings ε is almost always unknown because of the unknown nature of $P(\mathbf{X} \in E_i, Y = i)$. The test-set error estimator is a way to estimate ε using a test set.

Suppose we apply the classifier on a test set \mathbf{S}_{te} that contains m observations with their labels. Let k denote the number of observations in \mathbf{S}_{te} that are misclassified by the classifier. The test-set error estimate, denoted $\hat{\varepsilon}_{te}$, is given by

$$\hat{\varepsilon}_{te} = \frac{k}{m}. \quad (4.5)$$

Rather than reporting the error estimate, it is also common to report the *accuracy* estimate of a classifier. The accuracy, denoted acc , and its test-set estimate, denoted \hat{acc}_{te} , are given by:

$$\begin{aligned} acc &= 1 - \varepsilon, \\ \hat{acc}_{te} &= 1 - \hat{\varepsilon}_{te}. \end{aligned} \quad (4.6)$$

For example, a classifier with an error rate of 15% has an accuracy of 85%.

Let us calculate the test-set error estimate of our trained kNN classifier. In this regard, we can compare the actual labels within the test set with the predicted labels and then find the proportion of misclassification:

```
errors = (y_test_predictions != y_test)
errors
```

```
array([False, False,  True, False, False, False, False, False,
       False, False, False, False, False, False, False, False,
       False, False, False, False, False,  True, False, False,
      True, False, False])
```

The classifier misclassified three data points out of 30 that were part of the test set; therefore, $\hat{\epsilon}_{te} = \frac{3}{30} = 0.1$:

```
error_est = sum(errors)/errors.size
print('The error rate estimate is: {:.2f}'.format(error_est) + '\n'\
      'The accuracy is: {:.2f}'.format(1-error_est))
```

The error rate estimate is: 0.10

The accuracy is: 0.90

In the above code, we use the place holder { } and format specifier 0.2f to specify the number of digits after decimal point. The “:” before .2f separates the format specifier from the rest of the replacement field (if any option is set) within the { }.

Using scikit-learn built-in functions from `metrics` module many performance metrics can be easily calculated. A complete list of these metrics supported by scikit-learn is found at ([Scikit-eval, 2023](#)). Here we only show how the accuracy estimate can be obtained in scikit-learn. For this purpose, there are two options: 1) using the `accuracy_score` function; and 2) using the `score` method of the classifier.

The `accuracy_score` function expects the actual labels and predicted labels as arguments:

```
from sklearn.metrics import accuracy_score
print('The accuracy is {:.2f}'.format(accuracy_score(y_test, u
      ↪y_test_predictions)))
```

The accuracy is 0.90

All classifiers in scikit-learn also have a `score` method that given a test data and its labels, returns the classifier accuracy; for example,

```
print('The accuracy is {:.2f}'.format(knn.score(X_test_scaled, y_test)))
```

The accuracy is 0.90

In Chapter 9, we will discuss the definition and calculation of many important performance metrics for both classification and regression.

Exercises:

Exercise 1: In the Iris classification problem, remove the `stratify=iris.target` when the data is split to training and test. Observe the change in `np.bincount(y_train)`.

Are the number of samples in each class still the same as the original dataset?

Exercise 2: Here we would like to work with another classification application. In this regard, we work with `digits` dataset that comes with scikit-learn. This is a dataset of 8×8 images of handwritten digits. More information including the sample size can be found at ([Scikit-digits, 2023](#)).

- 1) Write a piece of code to load the dataset and show the first 100 images as in Fig. 4.3.

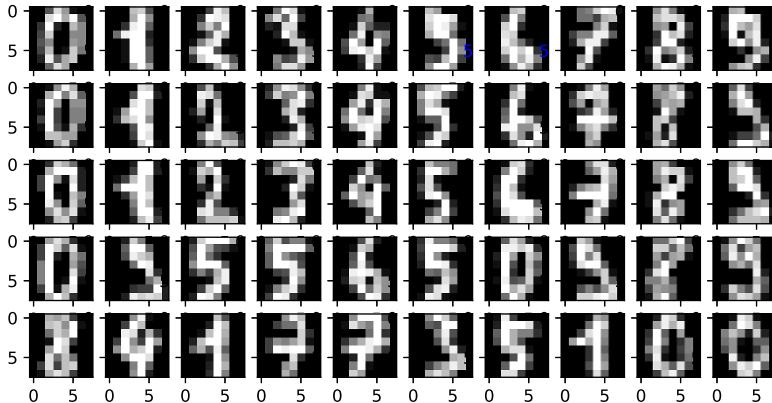


Fig. 4.3: The first 100 images in digits dataset

Hint: to show an image we can use `matplotlib.axes.Axes.imshow()`. Assuming `ax` is the created axes object, then the first image, for example, could be shown using the values of `images` key of the returned Bunch object or even using its `data` key, which is flattened image to create feature vectors for classification: `ax.imshow(bunch.data[0].reshape(8,8))` where `bunch` is the returned Bunch object. Read documentation of `imshow()` at ([matplotlib-imshow, 2023](#)) to use appropriate parameters to display grayscale images.

- 2) Implement the train-test splitting with a test size of 0.25 (set the `random_state=42`), properly perform standardization and transformation of datasets, train a 3NN on training set, and report the accuracy on the test set.

Exercise 3: A problem in machine learning is to transform categorical variables into efficient numerical features. This focus is warranted due to the ubiquity of categorical data in real-world applications but, on the contrary, the development of many machine learning methods based on the assumption of having numerical variables. This stage, which is transforming categorical variables to numeric variables is known as encoding. In the Iris classification application, we observed that the class variable

was already encoded. In this exercise, we will practice encoding in conjunction with a dataset collected for a business application.

Working with clients and understanding the factors that can improve the business is part of *marketing data science*. One goal in marketing data science is to *keep* customers (customer retention), which could be easier in some aspects than attracting new customers. In this exercise, we see a case study on customer retention.

A highly competitive market is communication services. Following the break-up of AT&T in 80's, customers had a choice to keep their carrier (AT&T) or move to another carrier. AT&T tried to identify factors relating to customer choice of carriers. For this purpose, they collected customer data from a number of sources including phone interviews, household billing, and service information recorded in their databases. The dataset that we use here is based on this effort and is taken from ([Miller, 2015](#)). It includes nine feature variables and a class variable (`pick`). The goal is to construct a classifier that predicts customers who switch to another service provider or stay with AT&T.

The dataset is stored in `att.csv`, which is part of the “data” folder. Below is description of the variables in the dataset:

pick: customer choice (AT&T or OCC [short for Other Common Carrier])

income: household income in thousands of dollars

moves: number of times the household moved in the last five years

age: age of respondent in years (18-24, 25-34, 35-44, 45-54, 55-64, 65+)

education: <HS (less than high school); HS (high school graduate); Voc (vocational school); Coll (Some College); BA (college graduate); >BA (graduate school)

employment: D (disabled); F (full-time); P (part-time); H (homemaker); R (retired); S (student); U (unemployed)

usage: average phone usage per month

nonpub: does the household have an unlisted phone number

reachout: does the household participate in the AT&T “Reach Out America” phone service plan?

card: does the household have an “AT&T Calling Card”?

- 1) load the data and pick only variables named `employment`, `usage`, `reachout`, and `card` as features to use in training the classifier (keep “`pick`” because it is the class variable).
- 2) find the rate of missing value for each variable (features and class variable).
- 3) remove any feature vector and its corresponding class with at least one missing entry. What is the sample size now?
- 4) use stratified random split to divide the data into train and test sets where the test set size is 20% of the dataset. Set `random_state=100`.
- 5) There are various strategies for encoding with different pros and cons. An efficient strategy in this regard is ordinal encoding. In this strategy, a variable with N categories, will be converted to a numeric variable with integers from 0 to $N - 1$. Use `OrdinalEncoder()` and `LabelEncoder()` transformers to encode the categorical features and the class variable, respectively—`LabelEncoder` works similarly to `OrdinalEncoder` except that it accepts one-dimensional

arrays and Series (so use that to encode the class variable). Hint: you need to fit encoder on training set and transform both training and test sets.

- 6) train a 7NN classifier using encoded training set and evaluate that on the encoded test set.

Exercise 4: We have a dataset D using which we would like to train and evaluate a classifier. Suppose

- we divide D into a training set denoted D_{train} and a test set denoted D_{test}
- apply normalization on D_{train} using parameters estimated from D_{train} to create D_{train}^{norm} , which denotes the normalized training set.

Which of the following sequence of actions is a legitimate machine learning practice?

A)

- apply normalization on D_{test} using parameters estimated from D_{train} to create the normalized test set denoted D_{test}^{norm}
- train the classifier on D_{train}^{norm} and assess its performance on D_{test}

B)

- apply normalization on D_{test} using parameters estimated from D_{test} to create the normalized test set denoted D_{test}^{norm}
- train the classifier on D_{train}^{norm} and assess its performance on D_{test}^{norm}

C)

- apply normalization on D_{test} using parameters estimated from D_{train} to create the normalized test set denoted D_{test}^{norm}
- train the classifier on D_{train}^{norm} and assess its performance on D_{test}^{norm}

D)

- apply normalization on D_{test} using parameters estimated from D_{train} to create the normalized test set denoted D_{test}^{norm}
- train the classifier on D_{train} and assess its performance on D_{test}^{norm}



Chapter 5

k-Nearest Neighbors

There are many predictive models in machine learning that can be used in the design process. We start the introduction to these predictive models from kNN (short for k-Nearest Neighbors) not only because of its simplicity, but also due to its long history in machine learning. As a result, in this chapter we formalize the kNN mechanism for both classification and regression and we will see various forms of kNN.

5.1 Classification

kNN is one of the oldest rules with a simple working mechanism and has found many applications since its conception in 1951 ([Fix and Hodges, 1951](#)). To formalize the kNN mathematical mechanism, it is common to consider the case of binary classification where the values of target variable y are encoded as 0 and 1. Furthermore, to mathematically characterize the kNN mechanism, we introduce some additional notations. Suppose we use a *distance metric* (e.g., Euclidean distance) to quantify distance of all feature vectors (observations) in a training set $S_{tr} = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ to a given feature vector \mathbf{x} (test observation) for which we obtain an estimate of y denoted \hat{y} . We denote the i^{th} nearest observation to \mathbf{x} as $x_{(i)}(\mathbf{x})$ with its associated label as $y_{(i)}(\mathbf{x})$. Note that both $x_{(i)}(\mathbf{x})$ and $y_{(i)}(\mathbf{x})$ are functions of \mathbf{x} because the distance is measured with respect to \mathbf{x} . With this notation, $x_{(1)}(\mathbf{x})$, for example, means the nearest observation within S_{tr} to \mathbf{x} .

5.1.1 Standard kNN Classifier

In a binary classification problem, the standard kNN classifier denoted as $\psi(\mathbf{x})$ is given by

$$\hat{y} = \psi(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_{i=1}^k \frac{1}{k} I_{\{y_{(i)}(\mathbf{x})=1\}} > \sum_{i=1}^k \frac{1}{k} I_{\{y_{(i)}(\mathbf{x})=0\}}, \\ 0 & \text{otherwise,} \end{cases} \quad (5.1)$$

where k denotes the number of nearest neighbors with respect to the given feature vector \mathbf{x} and I_A is the indicator of event A , which is 1 when A happens, and 0 otherwise. Here we write $\hat{y} = \psi(\mathbf{x})$ to remind us the fact that the function $\psi(\mathbf{x})$ (i.e., the classifier) is essentially an estimator of the class label y . Nevertheless, hereafter, we denote all classifiers simply as $\psi(\mathbf{x})$. In addition, in what follows, the standard kNN classifier is used interchangeably with kNN classifier.

Let us elaborate more on (5.1). The term $\sum_{i=1}^k I_{\{y_{(i)}(\mathbf{x})=1\}}$ in (5.1) counts the number of training observations with label 1 among the k nearest neighbors of \mathbf{x} . This is then compared with $\sum_{i=1}^k I_{\{y_{(i)}(\mathbf{x})=0\}}$, which is the number of training observations with label 0 among the k nearest neighbors of \mathbf{x} , and if $\sum_{i=1}^k I_{\{y_{(i)}(\mathbf{x})=1\}} > \sum_{i=1}^k I_{\{y_{(i)}(\mathbf{x})=0\}}$, then $\psi(\mathbf{x}) = 1$ (i.e., kNN assigns label 1 to \mathbf{x}); otherwise, $\psi(\mathbf{x}) = 0$. As we can see in (5.1), the factor $\frac{1}{k}$ does not have any effect on the process of decision making in the classifier and it can be canceled out safely from both sides of the inequality. However, having that in (5.1) better reflects the relationship between the standard kNN classifier and other forms of kNN that we will see later in this chapter. For multiclass classification problems with c classes $j = 0, 1, \dots, c-1$, (e.g., in Iris flower classification problem we had $c = 3$), we again identify the k nearest neighbors of \mathbf{x} , count the number of labels among these k training samples, and classify \mathbf{x} to the majority class among these k observations; that is to say,

$$\hat{y} = \psi(\mathbf{x}) = \operatorname{argmax}_j \sum_{i=1}^k \frac{1}{k} I_{\{y_{(i)}(\mathbf{x})=j\}}. \quad (5.2)$$

In binary classification applications, it is common to avoid an even k to prevent ties in (5.1). For example, in these applications it is common to see a kNN with k being 3 or 5 (i.e., using 3NN and 5NN rules); however, in general k should be treated as a hyperparameter and tuned via a *model selection* process (model selection will be discussed in Chapter 9). In general, the larger k , the smoother are the *decision boundaries*. These are boundaries between *decision regions* (i.e., the c partitions of the feature space obtained by the classifier). In Chapter 4 we have already seen how kNN classifier is used and implemented in scikit-learn. Here we examine the effect of k on the decision boundaries and accuracy of kNN on the scaled Iris data prepared in Chapter 4. In this regard, we first load the training and test sets that were preprocessed there:

```
import numpy as np
arrays = np.load('data/iris_train_scaled.npz')
X_train = arrays['X']
y_train = arrays['y']
arrays = np.load('data/iris_test_scaled.npz')
X_test = arrays['X']
```

```
y_test = arrays['y']
print('X shape = {}'.format(X_train.shape) + '\ny shape = {}'.
      format(y_train.shape))
```

```
X shape = (120, 4)
y shape = (120,)
```

For illustration purposes, we only consider the first two features in data; that is, the first and second features (columns) in `X_train` and `X_test`:

```
X_train = X_train[:, [0, 1]]
X_test = X_test[:, [0, 1]]
X_train.shape
```

```
(120, 2)
```

To plot the decision regions, we use the same technique discussed in Example 3.5; that is, we first create a grid, train a classifier using training data, classify each point in the grid using the trained classifier, and then plot the decision regions based on the assigned labels to each point in the grid:

```
%matplotlib inline
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.neighbors import KNeighborsClassifier as KNN
color = ('aquamarine', 'bisque', 'lightgrey')
cmap = ListedColormap(color)

mins = X_train.min(axis=0) - 0.1
maxs = X_train.max(axis=0) + 0.1
x = np.arange(mins[0], maxs[0], 0.01)
y = np.arange(mins[1], maxs[1], 0.01)
X, Y = np.meshgrid(x, y)
coordinates = np.array([X.ravel(), Y.ravel()]).T
fig, axs = plt.subplots(2, 2, figsize=(6, 4), dpi = 200)
fig.tight_layout()
K_val = [1, 3, 9, 36]
for ax, K in zip(axs.ravel(), K_val):
    knn = KNN(n_neighbors=K)
    knn.fit(X_train, y_train)
    Z = knn.predict(coordinates)
    Z = Z.reshape(X.shape)
    ax.tick_params(axis='both', labelsize=6)
    ax.set_title(str(K) + 'NN Decision Regions', fontsize=8)
    ax.pcolormesh(X, Y, Z, cmap = cmap, shading='nearest')
    ax.contour(X ,Y, Z, colors='black', linewidths=0.5)
```

```

ax.plot(X_train[y_train==0, 0], X_train[y_train==0, 1], 'g.', 
    markerSize=4)
    ax.plot(X_train[y_train==1, 0], X_train[y_train==1, 1], 'r.', 
    markerSize=4)
    ax.plot(X_train[y_train==2, 0], X_train[y_train==2, 1], 'k.', 
    markerSize=4)
    ax.set_xlabel('sepal length (normalized)', fontsize=7)
    ax.set_ylabel('sepal width (normalized)', fontsize=7)
    print('The accuracy for K={} on the training data is {:.3f}'.
format(K, knn.score(X_train, y_train)))
    print('The accuracy for K={} on the test data is {:.3f}'.format(K,
    knn.score(X_test, y_test)))
for ax in axs.ravel():
    ax.label_outer() # to show the x-label and the y-label for the last
    row and the left column, respectively

```

The accuracy for k=1 on the training data is 0.933
The accuracy for k=1 on the test data is 0.633
The accuracy for k=3 on the training data is 0.842
The accuracy for k=3 on the test data is 0.733
The accuracy for k=9 on the training data is 0.858
The accuracy for k=9 on the test data is 0.800
The accuracy for k=36 on the training data is 0.783
The accuracy for k=36 on the test data is 0.800

As we can see in Fig. 5.1, a larger value of the hyperparameter k generally leads to smoother decision boundaries. That being said, a smoother decision region does not necessarily mean a better classifier performance.

5.1.2 Distance-Weighted kNN Classifier

The classification formulated in (5.1) is the mechanism of the standard kNN classifier. There is another form of kNN known as distance-weighted kNN (DW-kNN) (Dudani, 1976). In this form, the k nearest neighbors of a test observation \mathbf{x} are weighted according to their distance from \mathbf{x} . The idea is that observations that are closer to \mathbf{x} should impose a higher influence on decision making. A simple weighting scheme for this purpose is to weight observations based on the inverse of their distance to \mathbf{x} . With this weighting scheme, DW-kNN classifier becomes:

$$\psi(\mathbf{x}) = \begin{cases} 1 & \text{if } \frac{1}{D} \sum_{i=1}^k \frac{1}{d[\mathbf{x}_{(i)}(\mathbf{x}), \mathbf{x}]} I_{\{y_{(i)}(\mathbf{x})=1\}} > \frac{1}{D} \sum_{i=1}^k \frac{1}{d[\mathbf{x}_{(i)}(\mathbf{x}), \mathbf{x}]} I_{\{y_{(i)}(\mathbf{x})=0\}}, \\ 0 & \text{otherwise,} \end{cases} \quad (5.3)$$

where

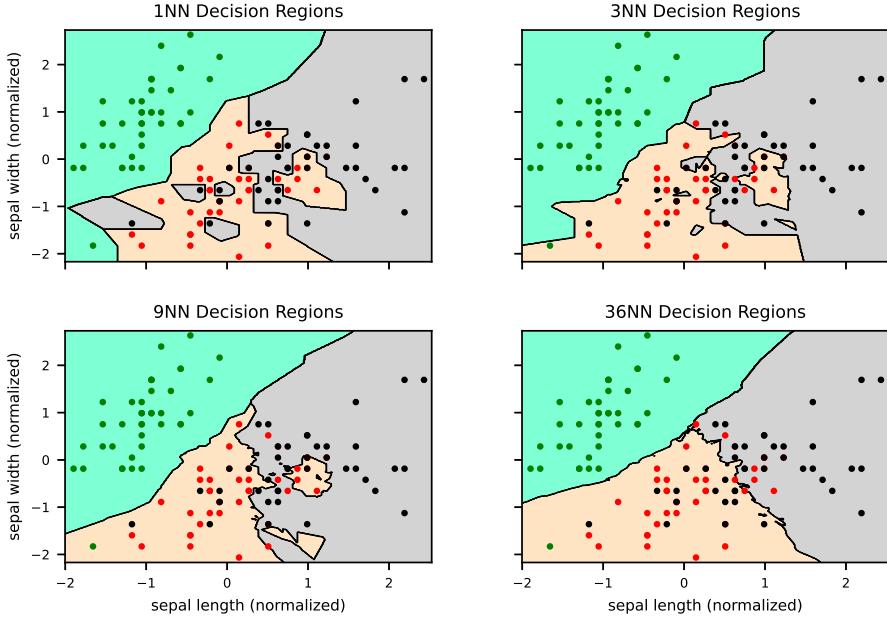


Fig. 5.1: kNN decision regions and boundaries for $k = 1, 3, 9, 36$ in the Iris classification problem.

$$D = \sum_{i=1}^k \frac{1}{d[\mathbf{x}_{(i)}(\mathbf{x}), \mathbf{x}]}, \quad (5.4)$$

and where $d[\mathbf{x}_{(i)}(\mathbf{x}), \mathbf{x}]$ denotes the distance of $\mathbf{x}_{(i)}(\mathbf{x})$ from \mathbf{x} . If we think of each weight $1/d[\mathbf{x}_{(i)}(\mathbf{x}), \mathbf{x}]$ as a “score” given to the training observation $\mathbf{x}_{(i)}(\mathbf{x})$, then DW-kNN classifies the test observation \mathbf{x} to the class with the largest “cumulative score” given to its observations that are among the k nearest neighbors of \mathbf{x} . At this stage, it is straightforward to see that the standard kNN given by (5.1) is a special form of DW-kNN where the k nearest neighbor have equal contribution to voting; that is, when $d[\mathbf{x}_{(i)}(\mathbf{x}), \mathbf{x}] = 1$ for all k nearest neighbors, which makes $D = k$.

In some settings, DW-kNN may end up behaving similarly to a simple nearest-neighbor rule (1NN) because training observations that are very close to a test observation will take very large weights. In practice though, the choice of using DW-kNN or standard kNN, or even the choice of k are matters that can be decided in the model selection phase (Chapter 9). In scikit-learn (as of version 1.2.2), DW-kNN can be implemented by setting the `weights` parameter of `KNeighborsClassifier` to '`distance`' (the default value of `weights` is '`uniform`', which corresponds to the standard kNN).

5.1.3 The Choice of Distance

To identify the nearest neighbors of an observation, there are various choices of distance metrics that can be used either in standard kNN or DW-kNN. Let $\mathbf{x}_i = [x_{1i}, x_{2i}, \dots, x_{qi}]^T$ and $\mathbf{x}_j = [x_{1j}, x_{2j}, \dots, x_{qj}]^T$ denote two q -dimensional vectors where x_{li} denotes the element at the l^{th} dimension of \mathbf{x}_i . We can define various distances between these vectors. Three popular choices of distances are Euclidean, Manhattan, and Minkowski denoted as $d_E[\mathbf{x}_i, \mathbf{x}_j]$, $d_{\text{Ma}}[\mathbf{x}_i, \mathbf{x}_j]$, and $d_{\text{Mi}}[\mathbf{x}_i, \mathbf{x}_j]$, respectively, and are given by

$$d_E[\mathbf{x}_i, \mathbf{x}_j] = \sqrt{\sum_{l=1}^q (x_{li} - x_{lj})^2}, \quad (5.5)$$

$$d_{\text{Ma}}[\mathbf{x}_i, \mathbf{x}_j] = \sum_{l=1}^q |x_{li} - x_{lj}|, \quad (5.6)$$

$$d_{\text{Mi}}[\mathbf{x}_i, \mathbf{x}_j] = \left(\sum_{l=1}^q |x_{li} - x_{lj}|^p \right)^{\frac{1}{p}}. \quad (5.7)$$

In what follows, to specify the choice of distance in the kNN classifier, the name of the distance is prefixed to the classifier; for example, Manhattan-kNN classifier and Manhattan-weighted kNN classifier refer to the standard kNN and the DW-kNN classifiers with the choice of Manhattan distance. When this choice is not indicated specifically, Euclidean distance is meant to be used. The Minkowski distance has an order p , which is an arbitrary integer. When $p = 1$ and $p = 2$, Minkowski distance reduces to the Manhattan and the Euclidean distances, respectively. In `KNeighborsClassifier`, the choice of distance is determined by the `metric` parameter. Although the default `metric` in `KNeighborsClassifier` is Minkowski, there is another parameter `p`, which determines the order of the Minkowski distance, and has a default value of 2. This means that the default metric of `KNeighborsClassifier` is indeed the Euclidean distance.

5.2 Regression

5.2.1 Standard kNN Regressor

The kNN regressor $f(\mathbf{x})$ is given by

$$\hat{y} = f(\mathbf{x}) = \sum_{i=1}^k \frac{1}{k} y_{(i)}(\mathbf{x}). \quad (5.8)$$

Similar to (5.1), here $\hat{y} = f(\mathbf{x})$ is written to emphasize that function $f(\mathbf{x})$ is essentially an estimator of y . Nevertheless, hereafter, we denote our regressors simply as $f(\mathbf{x})$. From (5.8), the standard kNN regressor estimates the target of a given \mathbf{x} as the average of targets of the k nearest neighbors of \mathbf{x} . We have already seen the functionality of kNN classifier through the Iris classification application. To examine the kNN functionality in regression, in the next section, we use another well-known dataset that suits regression applications.

5.2.2 A Regression Application Using kNN

The Dataset: Here we use the California Housing dataset, which includes the median house price (in \$100,000) of 20640 California districts with 8 features that can be used to predict the house price. The data was originally collected in (Kelley Pace and Barry, 1997) and suits regression applications because the target, which is the median house price of a district (identified by “MedHouseVal” in the data), is a numeric variable. Here we first load the dataset:

```
from sklearn import datasets
california = datasets.fetch_california_housing()
print('california housing data shape: ' + str(california.data.shape) + \
      '\nfeature names: ' + str(california.feature_names) + \
      '\ntarget name: ' + str(california.target_names))
```

```
california housing data shape: (20640, 8)
feature names: ['MedInc', 'HouseAge', 'AveRooms', 'AveBedrms', ↴
    'Population', 'AveOccup', 'Latitude', 'Longitude']
target name: ['MedHouseVal']
```

As mentioned in Section 4.3, using the DESCR key we can check some details about the dataset. Here we use this key to list the name and meaning of variables in the data:

```
print(california.DESCR[:975])
```

```
... _california_housing_dataset:

California Housing dataset
-----
**Data Set Characteristics:**

:Number of Instances: 20640

:Number of Attributes: 8 numeric, predictive attributes and the target

:Attribute Information:
- MedInc          median income in block group
```

- HouseAge	median house age in block group
- AveRooms	average number of rooms per household
- AveBedrms	average number of bedrooms per household
- Population	block group population
- AveOccup	average number of household members
- Latitude	block group latitude
- Longitude	block group longitude

:Missing Attribute Values: None

This dataset was obtained from the StatLib repository.

https://www.dcc.fc.up.pt/~ltorgo/Regression/cal_housing.html

The target variable is the median house value for California districts, expressed in hundreds of thousands of dollars (\$100,000).

“Latitude” and “Longitude” identify the centroid of each district in the data, and “MedInc” values are in \$10,000.

Preprocessing and an Exploratory Analysis: We split the data into training and test sets. Compared with the Iris classification problem, we use the default 0.25 `test_size` and do not set `stratify` to any variable—after all, random sampling with stratification requires classes whereas in regression, there are no “classes”.

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(california.data,
                                                    ->california.target, random_state=100)
print('X_train_shape: ' + str(X_train.shape) + '\nX_test_shape: ' + 
      ->str(X_test.shape) +
      + '\ny_train_shape: ' + str(y_train.shape) + '\ny_test_shape: ' +
      ->+ str(y_test.shape))
```

```
X_train_shape: (15480, 8)
X_test_shape: (5160, 8)
y_train_shape: (15480,)
y_test_shape: (5160,)
```

Similar to Section 4.6, we use standardization to scale the training and the test sets. In this regard, the `scaler` object is trained using the training set, and is then used to transform both the training and the test sets:

```
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

In Chapter 4, we used pair plots as an exploratory analysis. Another exploratory analysis could be to look into *Pearson's correlation coefficients* between predictors and the response or even between predictors themselves.



Consider two random variables X and Y . The Pearson's correlation coefficient ρ between them is defined as

$$\rho = \frac{\text{Cov}[X, Y]}{\sigma_X \sigma_Y}, \quad (5.9)$$

where

$$\text{Cov}[X, Y] = E[(X - E[X])(Y - E[Y])], \quad (5.10)$$

and where $E[Z]$ and σ_Z are the expected value and the standard deviation of Z , respectively. As the covariance $\text{Cov}[X, Y]$ measures the joint variation between two variables X and Y , ρ in (5.9) represents a normalized form of their joint variability. However, in practice, we can not generally compute (5.9) because it depends on the unknown actual expectations and standard deviations. Therefore, we need to estimate that from a sample.

Suppose we collect a sample from the joint distribution of X and Y . This sample consists of n pairs of observations $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$; that is, when the value of X is x_i , the value of Y is y_i . The sample Pearson's correlation coefficient r is an estimate of ρ and is given by

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}, \quad (5.11)$$

where $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$. Note that even if change x_i to $ax_i + b$ and y_i to $cy_i + d$ where a, b, c and d are all constants, r does not change. In other words, sample Pearson's correlation coefficient is scale and location invariant. Another implication of this point is that the sample Pearson's correlation coefficient inherently possesses data normalization (and, therefore, if desired, it could be done even before normalization).

The main shortcoming of Pearson's correlation coefficient is that it is not a completely general measure of the strength of a relationship between two random variables. Instead it is a measure of the degree of *linear relationship* between them. This is because $-1 \leq \rho \leq 1$ but $\rho = 1$ or -1 iff $y = ax + b$ for some constants a and b . Therefore, a $|\rho| < 1$ indicates only that the relationship is not completely linear, but there could be still a strong nonlinear relationship between x and y .

We can easily calculate pairwise correlation coefficients between all variables using `pandas.DataFrame.corr()`. To do so, we first create a `DataFrame` from `X_train_scaled` and `y_train`, and add feature names and the target name as the column labels (to easily identify the target name, we rename it to `MEDV`):

```
import seaborn as sns
import matplotlib.pyplot as plt
california_arr = np.concatenate((X_train_scaled, y_train),
    ↵reshape(-1,1)), axis=1) # np.concatenate((a1, a2)) concatenate array,
    ↵a1 and a2 along the specified axis
california_pd = pd.DataFrame(california_arr, columns=[*california.
    ↵feature_names, 'MEDV'])
california_pd.head().round(3)
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude
0	1.604	-0.843	0.974	-0.086	0.068	0.027	-0.865
1	-0.921	0.345	-0.197	-0.238	-0.472	-0.068	1.647
2	-0.809	1.849	-0.376	-0.037	-0.516	-0.082	1.675
3	0.597	-0.289	-0.437	-0.119	-0.680	-0.121	1.008
4	0.219	0.107	0.187	0.122	-0.436	-0.057	0.956

	Longitude	MEDV
0	0.883	2.903
1	-0.999	0.687
2	-0.741	1.097
3	-1.423	4.600
4	-1.283	2.134

Now we calculate and, at the same time, we draw a color-coded plot (known as heatmap) of these correlation coefficients using `seaborn.heatmap()` (the result is depicted in Fig. 5.2):

```
fig, ax = plt.subplots(figsize=(9,5))
sns.heatmap(california_pd.corr().round(2), annot=True, square=True,
    ↵ax=ax)
```

One way that we may use this exploratory analysis is in feature selection. In many applications with a limited sample size and moderate to large number of features, using a subset of features could lead to a better performance in predicting the target than using all features. This is due to what is known as the “curse of dimensionality”, which is discussed in details in Chapter 10. In short, this phenomenon implies that using more features in training not only increases the computational burden, but also, and perhaps more importantly, could lead to a lower performance of the trained models after adding more than a certain number of features. The process of feature subset selection (or simply feature selection) *per se* is an important area of research in machine learning and will be discussed in Chapter 10. Here, we may use the correlation matrix presented in Fig. 5.2 to identify a subset of features such that each feature within this subset is strongly or even moderately correlated with the response

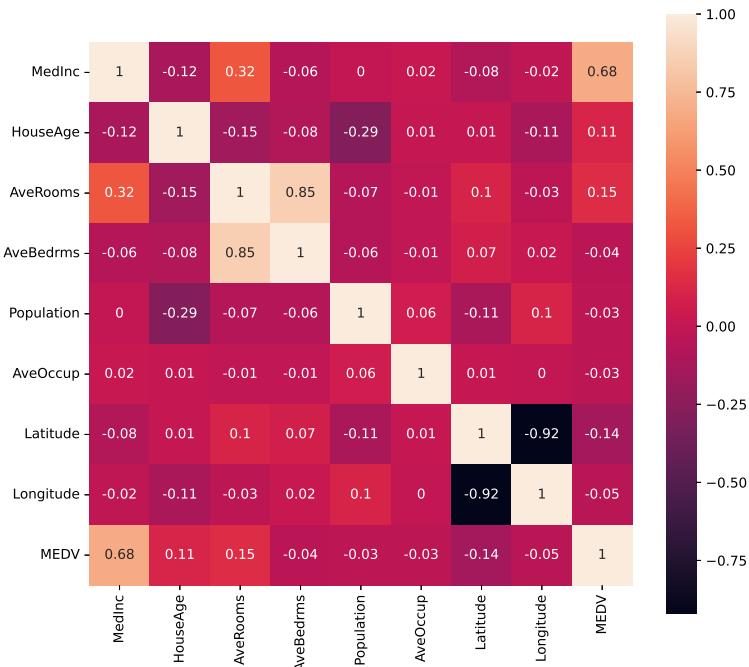


Fig. 5.2: The heatmap of pairwise correlation coefficients for all variables in the California Housing dataset.

variable. Although defining a strong or moderate correlation is naturally a subjective matter, we simply choose those variables with a correlation magnitude greater than 0.1; that is, `MedInc`, `HouseAge`, `AveRooms`, and `Latitude`. It is worthwhile to reiterate that here we even conducted such an exploratory feature selection after splitting the data into training and test sets. This is to prevent data leakage, which was discussed in Section 4.6. In what follows, we only work with the selected features.

```
X_train_fs_scaled = X_train_scaled[:, [0, 1, 2, 7]] # X_trained with
# selected features
X_test_fs_scaled = X_test_scaled[:, [0, 1, 2, 7]]
print('The shape of training X after feature selection: ' +
      str(X_train_fs_scaled.shape))
print('The shape of test X after feature selection: ' +
      str(X_test_fs_scaled.shape))
```

The shape of training X after feature selection: (15480, 4)

The shape of test X after feature selection: (5160, 4)

```
np.savez('data/california_train_fs_scaled', X = X_train_fs_scaled, y = y_train) # to save for possible uses later
np.savez('data/california_test_fs_scaled', X = X_test_fs_scaled, y = y_test)
```



A point to notice is the order of scaling and calculating Pearson's correlation coefficient. As we mentioned earlier, if we change values of x_i to $ax_i + b$ and y_i to $cy_i + d$ where a, b, c and d are all constants, Pearson's correlation coefficient does not change. Therefore, whether we calculate Pearson's correlation coefficient before or after any linear scaling, the results do not change. That being said, here we present it after standardization to align with those types of feature selection that depend on the scale of data and, therefore, are expected to be applied after data scaling—after all, if scaling is performed, the final predictive model should be trained on scaled data and it would be better to identify features that are predictive in the scaled feature space using which the final model is trained.

Model Training and Evaluation: Now we are in the position to train our model. The standard kNN regressor is implemented in the KNeighborsRegressor class in the sklearn.neighbors module. We train a standard 50NN (Euclidean distance and uniform weights) using the scaled training set (`X_train_fs_scaled`), and then use the model to predict the target in the test set (`X_test_fs_scaled`). At the same time, we created the scatter plot between predicted targets \hat{y} and the actual targets. This entire process is implemented by the following short code snippet:

```
from sklearn.neighbors import KNeighborsRegressor as KNN
plt.style.use('seaborn')

knn = KNN(n_neighbors=50)
knn.fit(X_train_fs_scaled, y_train)
y_test_predictions = knn.predict(X_test_fs_scaled)
plt.figure(figsize=(4.5, 3), dpi = 200)
plt.plot(y_test, y_test_predictions, 'g.', markersize=4)
lim_left, lim_right = plt.xlim()
plt.plot([lim_left, lim_right], [lim_left, lim_right], '--k', linewidth=1)
plt.xlabel("y (Actual MEDV)", fontsize='small')
plt.ylabel("$\hat{y}$ (Predicted MEDV)", fontsize='small')
plt.tick_params(axis='both', labelsize=7)
```

Let S_{te} denote the test set containing m test observations. In situations where $\hat{y}_i = y_i$ for any $x_i \in S_{te}$, we commit no error in our predictions. This implies that

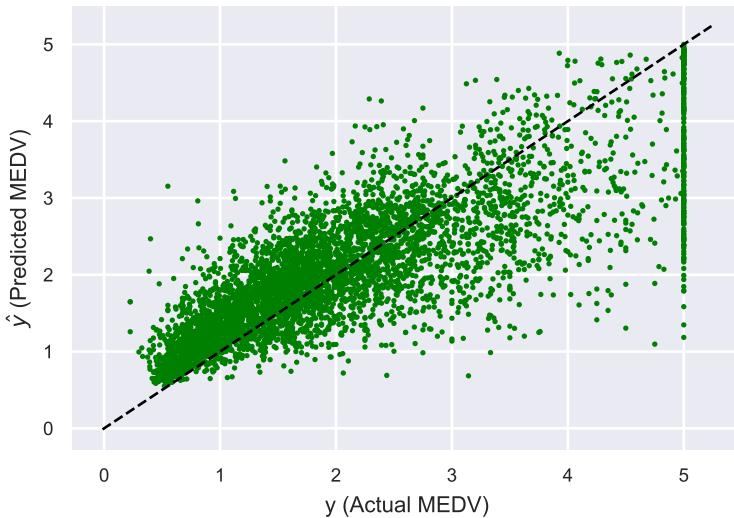


Fig. 5.3: The scatter plot of predicted targets by a 50NN and the actual targets in the California Housing dataset.

all points in the scatter plot between predicted and actual targets should lie on the diagonal line $\hat{y} = y$. In Fig. 5.3, we can see that, for example, when MEDV is lower than \$100k, our model is generally overestimating the target; however, for large values of MEDV (e.g., around \$500k), our model is underestimating the target. Although such scatter plots help shed light on the behavior of the trained regressor across the entire range of the target variable, we generally would like to summarize the performance of the regressor using some performance metrics.

As we discussed in Chapter 4, all classifiers in scikit-learn have a `score` method that given a test data and their labels, returns a measure of classifier performance. This also holds for regressors. However, for regressors the `score` method estimates *coefficient of determination*, also known as *R-squared* statistics, denoted \hat{R}^2 , given by

$$\hat{R}^2 = 1 - \frac{\text{RSS}}{\text{TSS}}, \quad (5.12)$$

where RSS and TSS are short for *Residual Sum of Squares* and *Total Sum of Squares*, respectively, defined as

$$\text{RSS} = \sum_{i=1}^m (y_i - \hat{y}_i)^2, \quad (5.13)$$

$$\text{TSS} = \sum_{i=1}^m (y_i - \bar{y})^2, \quad (5.14)$$

and where \bar{y} is the target mean within the test set; that is $\bar{y} = \frac{1}{m} \sum_{i=1}^m y_i$. This means that \hat{R}^2 measures how well our trained regressor is performing when compared with the trivial estimator of the target, which is \bar{y} . In particular, for perfect predictions, $\text{RSS} = 0$ and $\hat{R}^2 = 1$; however, when our trained regressor is performing similarly to the trivial average estimator, we have $\text{RSS} = \text{TSS}$ and $\hat{R}^2 = 0$.



A common misconception among beginners is that \hat{R}^2 can not be negative. While \hat{R}^2 can not be greater than 1 (because we can not do better than perfect prediction), there is no limit on how negative it could be. This is simply because there is no limit on how badly our regressors can perform compared with the trivial estimator. As soon as $\text{RSS} > \text{TSS}$, which can be thought as the trained regressor is performing worse than the trivial regressor \bar{y} , \hat{R}^2 will be negative.

Next, we examine the performance of our regressor as measured by \hat{R}^2 :

```
print('The test R^2 is: {:.2f}'.format(knn.score(X_test_fs_scaled, y_test)))
```

The test R^2 is: 0.64

5.2.3 Distance-Weighted kNN Regressor

In contrast with the standard kNN regressor given in (5.8), DW-kNN regressor computes a weighted average of targets for the k nearest neighbors where the weights are the inverse of the distance of each nearest neighbor from the test observation \mathbf{x} . This causes observations that are closer to a test observation impose a higher influence in the weighted average. In particular, DW-kNN is given by

$$f(\mathbf{x}) = \frac{1}{D} \sum_{i=1}^K \frac{1}{d[\mathbf{x}_{(i)}(\mathbf{x}), \mathbf{x}]} y_{(i)}, \quad (5.15)$$

where D is defined in (5.4). When $d[\mathbf{x}_{(i)}(\mathbf{x}), \mathbf{x}] = 1$ for all k nearest neighbors, (5.15) reduces to the standard kNN regressor given by (5.8). In scikit-learn, DW-kNN regressor is obtained by the default setting of the `weights` parameter of

KNeighborsRegressor from 'uniform' to 'distance'. The choice of distance itself can be also controlled with the `metric` parameter.

Exercises:

Exercise 1: In California Housing dataset, redo the analysis conducted in Section 5.2.2 with the following changes:

- 1) Instead of the four features that were used in Section 5.2.2, use all features to train a 50NN and evaluate on the test set. What is the \hat{R}^2 ? Does this change lead to a better or worse performance compared with what was achieved in Section 5.2.2?
- 2) In this part use all features as in Part 1; however, use a DW-50NN with Manhattan distance. Do you see a better or worse performance compared with Part 1?

Exercise 2: Which class is assigned to an observation $x_0 = 0.41$ if we use the standard 5NN with the following training data?

class →	1	2	1	3	2	3	2	1	3
observation →	0.45	0.2	0.5	0.05	0.3	0.15	0.35	0.45	0.4

- A) class 1
- B) class 2
- C) class 3
- D) there is a tie between class 1 and 3

Exercise 3: Suppose we have the following training data where x and y denote the value of the predictor and the target value, respectively. We train a standard 3NN regressor using this training data. What is the prediction of y for $x = 2$ using the trained 3NN?

x	-3	1	-1	4	2	5
y	0	5	15	-3	3	6

- A) 2
- B) 1.66
- C) 3
- D) 4
- E) 5
- F) 1

Exercise 4: Suppose we trained a regressor and applied to a test set with seven observations to determine \hat{R}^2 . The result of applying this regressor to the test set is tabulated in the following table where y and \hat{y} denote the actual and the estimated values of the target, respectively. What is \hat{R}^2 ?

- A) 0.25
- B) 0.35
- C) 0.55
- D) 0.65
- E) 0.75

y	0	-1	2	3	4	-3	2
\hat{y}	-1	-2	3	2	5	-3	4

Exercise 5: Suppose we trained a regressor and applied to a test set with six observations to determine \hat{R}^2 . The result of applying this regressor to the test set is tabulated in the following table where y and \hat{y} denote the actual and the estimated values of the target, respectively. Determine \hat{R}^2 ?

y	1	2	1	2	9	-3
\hat{y}	-1	0	3	2	9	3

Exercise 6: A typical task in machine learning is to predict (estimate) future values of a variable based on its past and present values. This is possible if we have time series for the variable of interest and if the present and the past values of the variable contain information about its future. This problem is called *forecasting* to distinguish that from the general use of “prediction” term in machine learning. Nevertheless, in machine learning forecasting is generally formulated as a regression problem by properly preparing the training data. In this exercise, we see how a forecasting problem is formulated and solved as a regression problem (use hand calculations throughout this exercise).

In this regard, Fig. 5.4 shows the annual revenue of a (hypothetical) company (in million US \$) for the last 11 years. We would like to train a 2NN to forecast revenue one year in the future based on the revenue in the last three years. Constructing this kNN can help answer question of what would be the revenue at the end of 2023 based on the revenue in 2020, 2021, and 2022. Because we would like to forecast the revenue one year ahead, we need to prepare a training data to train a regressor by associating the value of the target y to some feature vectors. We achieve this in two stages by first creating some feature vectors and then associating the value of y to each feature vector.

- 1) **Preparing feature vectors.** Use a sliding window of size 3 (observations) to construct feature vectors of size 3 such that the feature value in each dimension is the revenue of one year within the window. Slide the window with an amount of two observations; that is, windows have an overlap of one observation. In Fig. 5.5, we show the window when the window is slid with this amount. The first and the second windows shown by dashed rectangles in this figure correspond to two feature vectors $\mathbf{x}_1 = [1, 2, 3]^T$ and $\mathbf{x}_2 = [3, 2, 3]^T$, respectively. Use this strategy and write all possible (five) feature vectors created from this data.
- 2) **Associating target values to feature vectors.** For each feature vector that was prepared in Part 1, the value of the target is the revenue of the year following the most recent year stored in the feature vector. Naturally, if we train a regressor using this prepared training data, the regressor estimates the value of the year following the most recent year in a given feature vector (i.e., it performs forecasting). As an example, see Fig. 5.6 in which the observation pointed to by an

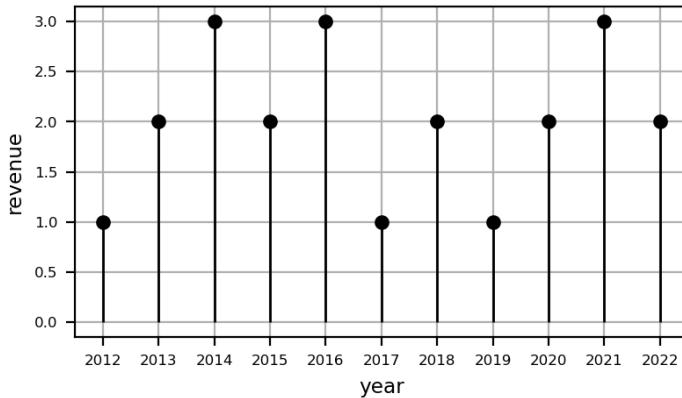


Fig. 5.4: The annual revenue of a company (in million US \$) for 11 years.

arrow shows the observation that should be used as the target for the first window (i.e., for the first feature vector). If possible, associate a target value to feature vectors created in Part 1. Remove the feature vector that has no corresponding target from the training data.

- 3) **Training and prediction.** Use the prepared training data to forecast the revenue for 2023 using Euclidean-2NN regressor.

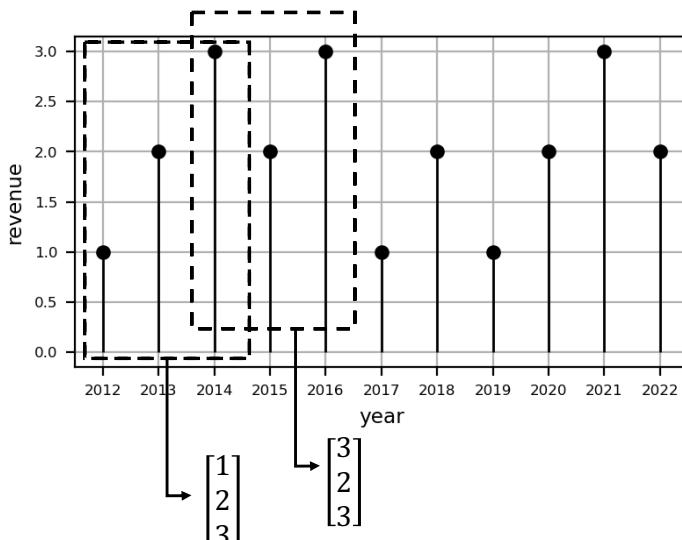


Fig. 5.5: Creating feature vectors by a sliding window with an overlap of one observation.

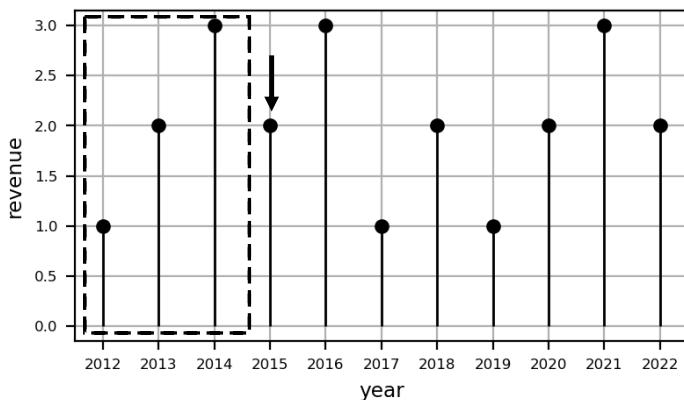


Fig. 5.6: An arrow pointing to the target value for the feature vector that is created using observations within the window (the dashed rectangle).

Exercise 7: In Exercise 6, the sliding window size and the amount of overlap should generally be treated as hyperparameters.

1. Redo Exercise 6 to forecast the revenue for 2023 based on the revenue in the last four years. Determine the size of the sliding window and use an overlap of 2 observations. What is the estimated 2023 revenue based on the Euclidean-2NN regressor?
2. Redo Part 1, but this time use the Manhattan-2NN regressor.



Chapter 6

Linear Models

Linear models are an important class of machine learning models that have been applied in various areas including genomic classification, electroencephalogram (EEG) classification, speech recognition, face recognition, to just name a few. Their popularity in various applications is mainly attributed to one or a combination of the following factors: 1) their simple structure and training efficiency; 2) interpretability; 3) matching their complexity with that of problems that have been historically available; and 4) a “satisficing” approach to decision-making. In this chapter, we cover some of the most widely used linear models including linear discriminant analysis, logistic regression, and multiple linear regression. We will discuss various forms of shrinkage including ridge, lasso, and elastic-net in combination with logistic regression and multiple linear regression.

6.1 Optimal Classification

There exist a number of linear models for classification and regression. Here we aim to focus on the working principles of a few important linear models that are frequently used in real-world applications and leave others to excellent textbooks such as (Duda et al., 2000; Bishop, 2006; Hastie et al., 2001). We start our discussion from linear classifiers. However, before exploring these classifiers or even defining what a linear classifier is, it would be helpful to study the optimal classification rule. This is because the structure of two linear classifiers that we will cover are derived from the estimate of the optimal classifier, also known as the *Bayes classifier*.

6.1.1 Discriminant Functions and Decision Boundaries

An often helpful strategy to conveniently characterize a classifier is using the concept of *discriminant function*. Suppose we can find c functions $g_i(\mathbf{x})$, $i = 0, \dots, c - 1$, where c is the number of classes using which we can write a classifier as

$$\psi(\mathbf{x}) = \operatorname{argmax}_i g_i(\mathbf{x}). \quad (6.1)$$

In other words, the classifier $\psi(\mathbf{x})$ assigns label i to $\mathbf{x} \in \mathbb{R}^p$ if $g_i(\mathbf{x}) > g_j(\mathbf{x})$ for $i \neq j$. The functions $g_i(\mathbf{x})$ are known as discriminant functions. The classifier then *partitions* the feature space \mathbb{R}^p into c *decision regions*, $\mathcal{R}_1, \dots, \mathcal{R}_c$ such that $\mathcal{R}_1 \cup \mathcal{R}_2 \cup \dots \cup \mathcal{R}_c = \mathbb{R}^p$. That is to say, if $g_i(\mathbf{x}) > g_j(\mathbf{x})$ for $\forall i \neq j$ then $\mathbf{x} \in \mathcal{R}_i$. Using this definition, a *decision boundary* is the boundary between two decision regions \mathcal{R}_i and \mathcal{R}_j , if any, which is given by

$$\{\mathbf{x} \mid g_i(\mathbf{x}) = g_j(\mathbf{x}), i \neq j\}. \quad (6.2)$$

As an example, let us elaborate on these definitions using the standard kNN classifier that we saw in Section 5.1.1. Comparing Eq. (6.1) above with Eq. (5.2), we observe that the discriminant functions for the standard kNN classifier are $g_i(\mathbf{x}) = \sum_{j=1}^k \frac{1}{k} I_{\{Y_{(j)}(\mathbf{x})=i\}}$, $i = 0, \dots, c - 1$. Therefore, the decision boundary is determined from (6.2), which for the specific training data given in Section 5.1.1, this is the set of points that we have already seen in Fig. 5.1.

In case of binary classification, we can simplify (6.1) by defining a single discriminant function $g(\mathbf{x}) = g_1(\mathbf{x}) - g_0(\mathbf{x})$, which leads to the following classifier:

$$\psi(\mathbf{x}) = \begin{cases} 1 & \text{if } g(\mathbf{x}) > 0, \\ 0 & \text{otherwise.} \end{cases} \quad (6.3)$$

Using (6.3), the decision boundary is where

$$\{\mathbf{x} \mid g(\mathbf{x}) = 0\}. \quad (6.4)$$

6.1.2 Bayes Classifier

The misclassification error rate of a classifier was defined in Section 4.9. In particular, the error rate was defined as

$$\varepsilon = P(\psi(\mathbf{X}) \neq Y). \quad (6.5)$$

Ideally, we desire to have a classifier that has the lowest error among any possible classifier. The question is whether we can achieve this classifier. As we will see in this section, this classifier, which is known as the Bayes classifier, exists in theory but not really in practice because it depends on the actual probabilistic distribution of data, which is almost always unknown in practice. Nonetheless, derivation of the Bayes classifier will guide us towards practical classifiers as we will discuss. In order to characterize the Bayes classifier, here we first start from some intuitive arguments with which we obtain a theoretical classifier, which is later proved (in Exercise 4) to be the Bayes classifier.

Suppose we are interested to develop a binary classifier for a given observation \mathbf{x} . A reasonable assumption is to set $g_i(\mathbf{x})$ to $P(Y = i|\mathbf{x})$, $i = 0, 1$, which is known as the *posterior probability* and represents the probability that class (random) variable takes the value i for a given observation \mathbf{x} . If this probability is greater for class 1 than class 0, then it makes sense to assign label 1 to \mathbf{x} . Replacing this specific $g_i(\mathbf{x})$, $i = 0, 1$ in (6.3) means that our classifier is:

$$\psi(\mathbf{x}) = \begin{cases} 1 & \text{if } P(Y = 1|\mathbf{x}) > P(Y = 0|\mathbf{x}), \\ 0 & \text{otherwise.} \end{cases} \quad (6.6)$$

Furthermore, because $P(Y = 0|\mathbf{x}) + P(Y = 1|\mathbf{x}) = 1$, (6.6) can be written as

$$\psi(\mathbf{x}) = \begin{cases} 1 & \text{if } P(Y = 1|\mathbf{x}) > \frac{1}{2}, \\ 0 & \text{otherwise.} \end{cases} \quad (6.7)$$

Using Bayes rule, we have

$$P(Y = i|\mathbf{x}) = \frac{p(\mathbf{x}|Y = i)P(Y = i)}{p(\mathbf{x})}. \quad (6.8)$$

In Section 4.9, we discussed the meaning of various terms that appear in (6.8). Replacing (6.8) in (6.6) means that our classifier can be equivalently written as

$$\psi(\mathbf{x}) = \begin{cases} 1 & \text{if } p(\mathbf{x}|Y = 1)P(Y = 1) > p(\mathbf{x}|Y = 0)P(Y = 0), \\ 0 & \text{otherwise,} \end{cases} \quad (6.9)$$

which itself is equivalent to

$$\psi(\mathbf{x}) = \begin{cases} 1 & \text{if } \log\left(\frac{p(\mathbf{x}|Y=1)}{p(\mathbf{x}|Y=0)}\right) > \log\left(\frac{P(Y=0)}{P(Y=1)}\right), \\ 0 & \text{otherwise.} \end{cases} \quad (6.10)$$

In writing (6.10) from (6.9), we rearranged the terms and took the natural log from both side—taking the $\log(\cdot)$ from both sides does not change our classifier because it is a monotonic function. However, it is common to do so in the above step because some important families of class-conditional densities have exponential form and doing so makes the mathematical expressions of the corresponding classifiers easier.

As we can see here, there are different ways to write the same classifier $\psi(\mathbf{x})$ —the forms presented in (6.6), (6.7), (6.9), and (6.10) all present the same classifier. Although this classifier was developed intuitively in (6.6), it turns out that for binary classification, it is the best classifier in the sense that it achieves the lowest misclassification error given in (6.5). The proof is slightly long and is left as an exercise (Exercise 4). Therefore, we refer to the classifier defined in (6.6) (or its equivalent forms) as the optimal classifier (also known as the Bayes classifier). Extension of this to multiclass classification with c classes is obtained by simply replacing $g_i(\mathbf{x})$, $i = 0, \dots, c - 1$ by $P(Y = i|\mathbf{x})$ in (6.1); that is to say, the Bayes classifier for multiclass classification with c classes is

$$\psi(\mathbf{x}) = \operatorname{argmax}_i P(Y = i|\mathbf{x}). \quad (6.11)$$

6.2 Linear Models for Classification

Linear models for classification are models for which *the decision boundaries are linear functions of the feature vector \mathbf{x}* (Bishop, 2006, p.179), (Hastie et al., 2001, p.101). There are several important linear models for classification including linear discriminant analysis, logistic regression, perceptron, and linear support vector machines; however, here we discuss the first two models not only to be able to present them in details but also because:

- these two models are quite efficient to train (i.e., have fast training); and
- oftentimes, applying a proper model selection on the first two would lead to a result that is “comparable” to the situation where a model selection is performed over all four. That being said, in the light of the *no free lunch theorem*, there is no reason to favor one classifier over another regardless of the context (specific problem) (Duda et al., 2000).

The theoretical aspects discussed below for each model have two purposes:

- to better understand the working principle of each model; and
- to better understand some important parameters used in scikit-learn implementation of each model.

6.2.1 Linear Discriminant Analysis

Linear discriminant analysis (LDA) is rooted in an idea from R. A. Fisher ([Fisher, 1936, 1940](#)), and has a long history in statistics and pattern recognition. It was further developed by Wald ([Wald, 1944](#)) in the context of decision theory and then formulated by Anderson ([Anderson, 1951](#)) in the form that is commonly used today. Owing to its simple formulation in the context of widely used Gaussian distributions and its many successful applications, LDA has been vigorously studied in the past several decades. At the same time, it has led to many other successful classifiers such as Euclidean distance classifier (EDC) ([Raudys and Pikelis, 1980](#)), diagonal-LDA (DLDA) ([Pikelis, 1973](#)), regularized-LDA (RLDA) ([Pillo, 1976](#)), and sparse discriminant analysis (SDA) ([Clemmensen et al., 2011](#)); it has been used in a semi-supervised fashion and implemented in various forms such as self-learning LDA ([McLachlan, 1975](#)), moment constrained LDA ([Loog, 2014](#)), and implicitly constrained LDA ([Krijthe and Loog, 2014](#)).

Bayes Rule for Normal Populations with a Common Covariance Matrix: The development of LDA is quite straightforward using a few assumptions. To develop the LDA classifier, the first assumption is that class-conditional probability densities are Gaussian; that is,

$$p(\mathbf{x}|Y = i) = \frac{1}{(2\pi)^{p/2} |\boldsymbol{\Sigma}_i|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_i)^T \boldsymbol{\Sigma}_i^{-1} (\mathbf{x}-\boldsymbol{\mu}_i)}, \quad i = 0, 1, \dots, c-1, \quad (6.12)$$

where c is the number of classes and $\boldsymbol{\mu}_i$ and $\boldsymbol{\Sigma}_i$ are the class-specific mean vector (a vector consists of the mean of each variable in \mathbf{x}) and the covariance matrix between all p variables in \mathbf{x} , respectively. The second assumption in development of LDA is that $\boldsymbol{\Sigma} \stackrel{\Delta}{=} \boldsymbol{\Sigma}_i, \forall i$, which means that there is a common covariance matrix $\boldsymbol{\Sigma}$ across all classes. Of course this is only an assumption and it is not necessary true in practice; however, this assumption makes the form of classifier simpler (in fact linear).

Consider the binary classification with $i = 0, 1$. Using the assumption of $\boldsymbol{\Sigma} \stackrel{\Delta}{=} \boldsymbol{\Sigma}_0 = \boldsymbol{\Sigma}_1$ in (6.10) and after some standard algebraic simplifications yield (see Exercise 6):

$$\psi(\mathbf{x}) = \begin{cases} 1 & \text{if } \left(\mathbf{x} - \frac{\boldsymbol{\mu}_0 + \boldsymbol{\mu}_1}{2}\right)^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0) + \log\left(\frac{P(Y=1)}{1-P(Y=1)}\right) > 0, \\ 0 & \text{otherwise.} \end{cases} \quad (6.13)$$

This means that the LDA classifier is the Bayes classifier when class-conditional densities are Gaussian with a common covariance matrix. The expression of LDA in (6.13) is in the form of (6.3) where

$$g(\mathbf{x}) = \left(\mathbf{x} - \frac{\boldsymbol{\mu}_0 + \boldsymbol{\mu}_1}{2}\right)^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0) + \log\left(\frac{P(Y=1)}{1-P(Y=1)}\right). \quad (6.14)$$

From (6.4) and (6.13) the decision boundary of the LDA classifier is given by

$$\{\mathbf{x} \mid \mathbf{a}^T \mathbf{x} + b = 0\}, \quad (6.15)$$

where

$$\mathbf{a} = \Sigma^{-1} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0), \quad (6.16)$$

$$b = -\left(\frac{\boldsymbol{\mu}_0 + \boldsymbol{\mu}_1}{2}\right)^T \Sigma^{-1} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0) + \log\left(\frac{P(Y=1)}{1-P(Y=1)}\right). \quad (6.17)$$

From (6.15)-(6.17), it is clear that the LDA classifier is a linear model for classification because its decision boundary is a linear function of \mathbf{x} .

Some Properties of the Linear Decision Boundary: The set of \mathbf{x} that satisfies (6.15) forms a *hyperplane* with its orientation and location uniquely determined by \mathbf{a} and b . Specifically, the vector \mathbf{a} is normal to the hyperplane. To see this, consider a vector \mathbf{u} from one arbitrary point \mathbf{x}_1 on the hyperplane to another point \mathbf{x}_2 ; that is, $\mathbf{u} = \mathbf{x}_2 - \mathbf{x}_1$. Because both \mathbf{x}_1 and \mathbf{x}_2 are on the hyperplane, then $\mathbf{a}^T \mathbf{x}_1 + b = 0$ and $\mathbf{a}^T \mathbf{x}_2 + b = 0$. Therefore, $\mathbf{a}^T (\mathbf{x}_2 - \mathbf{x}_1) = \mathbf{a}^T \mathbf{u} = 0$, which implies \mathbf{a} is normal to any vector \mathbf{u} on the hyperplane. This is illustrated by Fig. 6.1a. This figure shows the decision boundary of the LDA classifier for a two-dimensional binary classification problem where classes are normally distributed with means $\boldsymbol{\mu}_0^T = [4, 4]^T$ and $\boldsymbol{\mu}_1^T = [-1, 0]^T$, a common covariance matrix given by

$$\Sigma = \begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix},$$

and $P(Y=1) = 0.5$. From (6.14)-(6.17), the decision boundary in this example is determined by $\mathbf{x} = [x_1, x_2]^T$ that satisfies $-5x_1 - x_2 + 9.5 = 0$. This is the solid line that passes through the filled circle, which is the midpoint between population means; that is, $\frac{\boldsymbol{\mu}_0 + \boldsymbol{\mu}_1}{2}$. This is the case because $P(Y=1) = 0.5$, which implies $\log\left(\frac{P(Y=1)}{1-P(Y=1)}\right) = 0$, and, therefore,

$$\left(\frac{\boldsymbol{\mu}_0 + \boldsymbol{\mu}_1}{2} - \frac{\boldsymbol{\mu}_0 + \boldsymbol{\mu}_1}{2}\right)^T \Sigma^{-1} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0) = 0.$$

Also note the vector $\mathbf{a} = \Sigma^{-1} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0) = [-5, -1]^T$ is perpendicular to the linear decision boundary. This vector is not in general in direction of the line connecting the population means (the dashed line in Fig. 6.1a) unless $\Sigma^{-1} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0) = k (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0)$ for some scalar k , which is the case if $\boldsymbol{\mu}_1 - \boldsymbol{\mu}_0$ is an eigenvector of Σ^{-1} (and, therefore, an eigenvector of Σ).

Suppose in the same binary classification example, we intend to classify three observations: $\mathbf{x}_A = [4, -2]^T$, $\mathbf{x}_B = [0.2, 5]^T$, and $\mathbf{x}_C = [1.1, 4]^T$. Replacing \mathbf{x}_A , \mathbf{x}_B , and \mathbf{x}_C in the discriminant function $-5x_1 - x_2 + 9.5$, gives -8.5, 3.5, 0, respectively.

Therefore, from (6.13), \mathbf{x}_A and \mathbf{x}_B are classified as 0 (blue in the figure) and 1 (red in the figure), respectively, and \mathbf{x}_C lies on the decision boundary and could be randomly assigned to one of the classes. Fig. 6.1b-6.1d illustrate the working mechanism of the LDA classifier to classify these three sample points. Due to the first part of the discriminant function in (6.13) (i.e., $\mathbf{x} - \frac{\mu_0 + \mu_1}{2}$), \mathbf{x}_B , and \mathbf{x}_C are moved by $-\frac{\mu_0 + \mu_1}{2}$ to points \mathbf{x}'_A , \mathbf{x}'_B , and \mathbf{x}'_C , respectively. Then in order to classify, the inner product of these vectors should be calculated with the vector $\mathbf{a} = \Sigma^{-1}(\mu_1 - \mu_0)$; however, because \mathbf{x}'_A , \mathbf{x}'_B , and \mathbf{x}'_C have obtuse, acute, and right angles with vector \mathbf{a} , their inner products are negative, positive, and zero, which lead to the same class labels as discussed before.

Sample LDA Classifier: The main problem with using classifier (6.13) in practice is that the discriminant function in (6.13) depends on the actual distributional parameters μ_0 , μ_1 , and Σ that are almost always unknown unless we work with synthetically generated data for which we have the underlying parameters that were used to generate the data in the first place.

Given a training data $S_{tr} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$ and with no prior information on distributional parameters used in (6.13), one can replace the unknown distributional parameters by their sample estimates to obtain the sample LDA classifier given by

$$\psi(\mathbf{x}) = \begin{cases} 1 & \text{if } \left(\mathbf{x} - \frac{\hat{\mu}_0 + \hat{\mu}_1}{2}\right)^T \hat{\Sigma}^{-1}(\hat{\mu}_1 - \hat{\mu}_0) + \log\left(\frac{P(Y=1)}{1-P(Y=1)}\right) > 0, \\ 0 & \text{otherwise,} \end{cases} \quad (6.18)$$

where $\hat{\mu}_i$ is the sample mean for class i and $\hat{\Sigma}$ is the *pooled sample covariance matrix*, which are given by (n_i is the number of observations in class i)

$$\hat{\mu}_i = \frac{1}{n_i} \sum_{j=1}^{n_i} \mathbf{x}_j, \quad (6.19)$$

$$\hat{\Sigma} = \frac{1}{n_0 + n_1 - 2} \sum_{i=0}^1 \sum_{j=1}^n (\mathbf{x}_j - \hat{\mu}_i)(\mathbf{x}_j - \hat{\mu}_i)^T I_{\{y_j=i\}}, \quad (6.20)$$

where I_A is the indicator of event A .

This means that the sample version of the LDA classifier in the form of (6.18) is a Bayes *plug-in* rule; that is to say, the parameters of the Bayes rule (for Gaussian class-conditional densities with a common covariance matrix) are estimated from the data and are plugged in the expression of the Bayes classifier. To distinguish the Gaussian-based optimal rule presented in (6.13) from its sample version expressed in (6.18) as well as from optimal rules developed based on other parametric assumptions, some authors (see (McLachlan, 2004)) refer to (6.13) and (6.18) as normal-based linear discriminant rule (NLDR) and sample NLDR, respectively. Nonetheless, hereafter

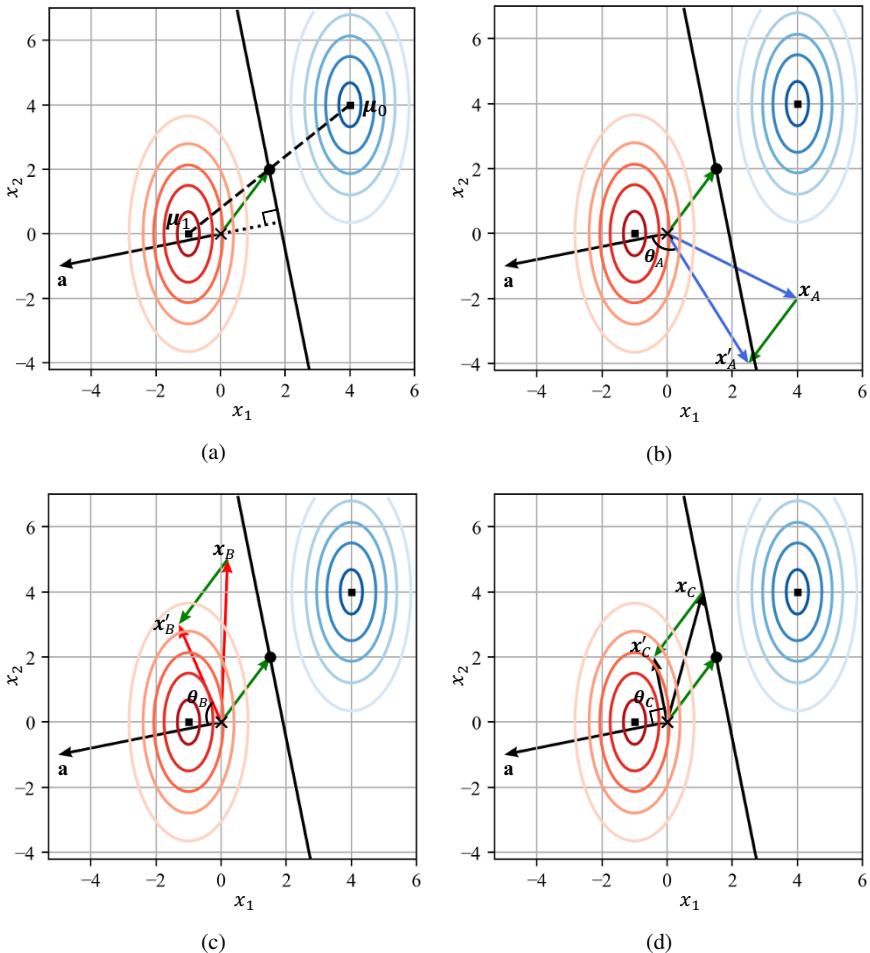


Fig. 6.1: The working mechanism of the LDA classifier. The blue and red ellipses show the areas of equal probability density of the Gaussian population for class 0 and 1, respectively. The filled circle identifies the midpoint $\frac{\mu_0 + \mu_1}{2}$. (a): vector $\mathbf{a} = \Sigma^{-1}(\mu_1 - \mu_0) = [-5, -1]^T$ is perpendicular to the linear decision boundary, which is expressed as $-5x_1 - x_2 + 9.5 = 0$. At the same time, the decision boundary passes through the midpoint $\frac{\mu_0 + \mu_1}{2}$ because $P(Y = 1) = 0.5$; (b)-(d): \mathbf{x}_A , \mathbf{x}_B , and \mathbf{x}_C are moved in the opposite direction of vector $\frac{\mu_0 + \mu_1}{2}$ to points \mathbf{x}'_A , \mathbf{x}'_B , and \mathbf{x}'_C , respectively. The positivity or negativity of the inner products of these vectors with vector \mathbf{a} are determined by the angles between them. Because $\theta_A > 90^\circ$ (obtuse angle), $\theta_B < 90^\circ$ (acute angle), and $\theta_C = 90^\circ$ (right angle), \mathbf{x}_A , \mathbf{x}_B , and \mathbf{x}_C are classified to class 0 (blue), 1 (red), and either one (could be randomly chosen), respectively.

and as long as the meaning is clear from the context, we may refer to both (6.13) and (6.18) as the LDA classifier.

Similar to (6.14)-(6.17), we conclude that the decision boundary of the sample LDA classifier forms a hyperplane $\mathbf{a}^T \mathbf{x} + b = 0$ where

$$\mathbf{a} = \hat{\Sigma}^{-1} (\hat{\mu}_1 - \hat{\mu}_0), \quad (6.21)$$

and

$$b = - \left(\frac{\hat{\mu}_0 + \hat{\mu}_1}{2} \right)^T \hat{\Sigma}^{-1} (\hat{\mu}_1 - \hat{\mu}_0) + \log \left(\frac{P(Y=1)}{1-P(Y=1)} \right). \quad (6.22)$$

Fig. 6.2a shows the sample LDA classifier for two set of samples drawn from the same Gaussian density functions used in Fig. 6.1. Here 20 sample points are drawn from each Gaussian population. This data is then used in (6.19) and (6.20) to estimate the decision boundary used in the LDA classifier (6.18). The linear decision boundary is characterized as $-3.28x_1 - 0.92x_2 + 7.49 = 0$ (dotted line) and is different from the optimal decision boundary, which is $-5x_1 - x_2 + 9.5 = 0$ (solid line). Fig. 6.2b shows the sample LDA classifier for the same example except that we increase the number of sample points drawn from each population to 1500. The decision boundary in this case is $-5.20x_1 - 0.90x_2 + 9.67 = 0$ and it appears to be closer to the optimal decision boundary. However, rather than discussing closeness of the decision boundaries, it is easier to discuss the behavior of the rule in terms of its performance. As the number of samples drawn from two Gaussian distribution increases unboundedly (i.e., $n \rightarrow \infty$), the error rate of the sample LDA classifier converges (in probability) to the error rate of the Bayes rule 6.13, a property known as the statistical consistency of the classifier¹. Nevertheless, the precise characterization of this property for the LDA classifier or other rules is beyond the scope of our discussion and interested readers are encouraged to consult other sources such as (Devroye et al., 1996) and (Braga-Neto, 2020).

In the development of LDA classifier, we assumed $\Sigma \stackrel{\Delta}{=} \Sigma_0 = \Sigma_1$. This assumption led us to use the pooled sample covariance matrix to estimate Σ . However, if we assume $\Sigma_0 \neq \Sigma_1$, it does not make sense anymore to use a single estimate as the estimate of both Σ_0 and Σ_1 . In these situations, we use different estimate of class covariance matrices in the expression of the Bayes classifier, which is itself developed by assuming $\Sigma_0 \neq \Sigma_1$. The resultant classifier is known as quadratic discriminant analysis (QDA) and has non-linear decision boundaries (see Exercise 7).

Remarks on Class Prior Probabilities: The discriminant function of the LDA classifier (6.18) explicitly depends on the prior probabilities $P(Y=i)$, $i = 0, 1$. However, in order to properly use such prior probabilities in training LDA (and other classifiers that explicitly depend on them), we need to distinguish between two types of sampling procedure: random sampling and separate sampling.

¹ Here we implicitly assume a fixed dimensionality of observations; otherwise, we should discuss the notion of generalized statistical consistency (Zollanvari and Dougherty, 2015).

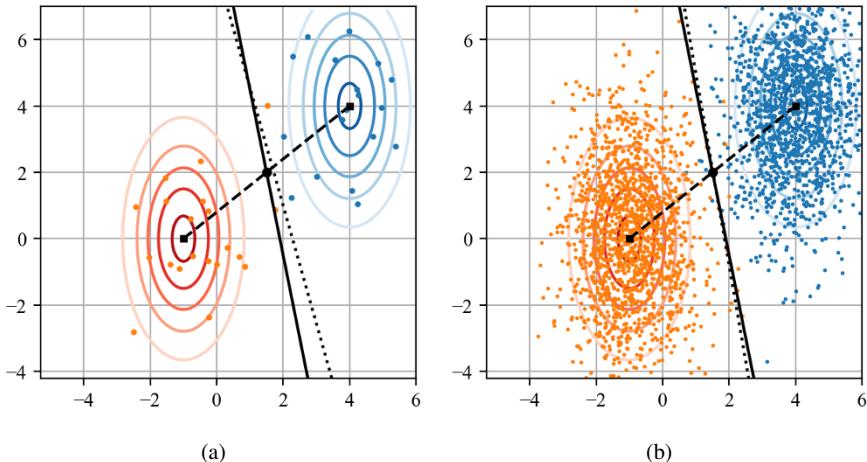


Fig. 6.2: The decision boundary of the sample LDA classifier for various training samples drawn from the two Gaussian populations used for plots in Fig. 6.1. The sample points are shown by orange or blue dots depending on their class. (a): 20 sample points are drawn from each Gaussian population. Here the decision boundary is characterized as $-3.28x_1 - 0.92x_2 + 7.49 = 0$; and (b) 1500 sample points are drawn from each Gaussian population. Here the decision boundary is characterized as $-5.20x_1 - 0.90x_2 + 9.67 = 0$. The decision boundary in this case appears to be “closer” to the optimal decision boundary identified by the solid line.

Under *random-sampling*, an independent and identically distributed (i.i.d.) sample $S_{tr} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$ is drawn from the mixture of the populations Π_0 and Π_1 with mixing proportions $P(Y = 0)$ and $P(Y = 1)$, respectively. This means that for a given training sample of size n with n_i sample points from each class, n_i is indeed a realization of a binomial random variable $N_i \sim \text{Binomial}(n, P(Y = i))$; that is to say, should the process of drawing a sample of size n from the mixture of populations repeat, n_0 and n_1 could be different. The random-sampling assumption is so pervasive in a text on classification that it is usually assumed without mention or in textbooks is often stated at the outset and then forgotten (e.g., see (Devroye et al., 1996, p.2)). An important consequence of random-sampling is that a class prior probability $P(Y = i)$ can be estimated by the sampling ratio $\frac{n_i}{n}$. This is indeed a direct consequence of Bernoulli’s weak law of large number and, therefore, for a large sample, we can expect the sampling ratio to be a reasonable estimate of the class prior probability.

However, suppose the sampling is not random, in the sense that the ratio $\frac{n_i}{n}$ is chosen before the sampling procedure. In this *separate-sampling* case, class-specific samples are selected randomly and separately from populations Π_0 and Π_1 ; therefore, given n , n_0 and n_1 are not determined by the sampling procedure. In this case, proportion of samples in each class are not representative of class prior

probabilities. As a matter of fact, in this case there is no meaningful estimator of the prior probabilities in the given sample. In these situations, the estimate of prior probabilities should come from prior knowledge. For the LDA classifier, a bad choice of prior probabilities can negatively impact the classifier (Anderson, 1951). As an example of separate-sampling, consider an investigator who recruits 10 patients from a “rare” disease (case) and 10 individual from a healthy population (control). As a result, the number of cases in this sample, which is equal to the number of controls, is not representative of this disease among the general population (otherwise, the disease is not rare).

Extension to Multiclass Classification: The case of LDA classifier for multiclass classification is a direct extension by using discriminants $P(Y = i|\mathbf{x})$ in (6.1) in which case the discriminant functions are linear and are given by

$$g_i(\mathbf{x}) = \hat{\boldsymbol{\mu}}_i^T \hat{\Sigma}^{-1} \mathbf{x} - \frac{1}{2} \hat{\boldsymbol{\mu}}_i^T \hat{\Sigma}^{-1} \hat{\boldsymbol{\mu}}_i + \log P(Y = i), \quad (6.23)$$

where $\hat{\boldsymbol{\mu}}_i$ is defined in (6.19) and $\hat{\Sigma}$ is given by

$$\hat{\Sigma} = \frac{1}{n - c} \sum_{i=0}^{c-1} \sum_{j=1}^n (\mathbf{x}_j - \hat{\boldsymbol{\mu}}_i)(\mathbf{x}_j - \hat{\boldsymbol{\mu}}_i)^T I_{\{y_j=i\}}, \quad (6.24)$$

where $n = \sum_{i=0}^{c-1} n_i$ is the total sample size across all classes.

Scikit-learn implementation: In scikit-learn, the LDA classifier is implemented by the `LinearDiscriminantAnalysis` class from the `sklearn.discriminant_analysis` module. However, to achieve a similar representation of LDA as in (6.23) one needs to change the default value of `solver` parameter from '`svd`' (as of scikit-learn version 1.2.2) to '`lsqr`', which solves the following system of linear equation to find \mathbf{a} in a least squares sense:

$$\hat{\Sigma} \mathbf{a} = \hat{\boldsymbol{\mu}}_i. \quad (6.25)$$

At the same time, note that the LDA discriminant (6.23) can be written as

$$g_i(\mathbf{x}) = \mathbf{a}^T \mathbf{x} - \frac{1}{2} \mathbf{a}^T \hat{\boldsymbol{\mu}}_i + \log P(Y = i), \quad (6.26)$$

where

$$\mathbf{a} = \hat{\Sigma}^{-1} \hat{\boldsymbol{\mu}}_i. \quad (6.27)$$

In other words, using '`lsqr`', (6.25) is solved in a way to avoid inverting the matrix $\hat{\Sigma}$ (rather than directly solving (6.27)). Note that by doing so, LDA classifier can be even applied to cases where the sample size is less than the feature size. In these situations, (6.27), which is indeed the direct formulation of LDA classifier, is not feasible because $\hat{\Sigma}$ is singular but we can find a “special” solution from (6.25).

Many models in scikit-learn have two attributes, `coef_` (or sometimes `coefs_`, for example, for multi-layer perceptron that we discuss later) and `intercept_` (or `intercepts_`) that store their coefficients and intercept (also known as bias term). The best way to identify such attributes, if they exist at all for a specific model, is to look into the scikit-learn documentation. For example, in case of LDA, for a binary classification problem, the coefficients vector and intercept are stored in `coef_` of shape (`feature_size,`) and `intercept_`, respectively; however, for multiclass classification, `coef_` has a shape of (`class_size, feature_size`).

 **More on solver:** A question that remains unanswered is what is meant to find an answer to (6.25) when $\hat{\Sigma}$ is not invertible? In general, when a system of linear equations is underdetermined (e.g., when $\hat{\Sigma}$ is singular, which means we have infinite solutions), we need to rely on techniques that designate a solution as “special”. Using '`lsqr`' calls `scipy.linalg.lsq`, which by default is based on DGELSD routine from LAPACK written in fortran. DGELSD itself finds the minimum-norm solution to the problem. This is a unique solution that minimizes $\|\hat{\Sigma}\mathbf{a} - \hat{\mu}_i\|_2$ using singular value decompositon (SVD) of matrix $\hat{\Sigma}$. This solution exists even for cases where $\hat{\Sigma}$ is singular. However, it becomes the same as $\hat{\Sigma}^{-1}\hat{\mu}_i$ if $\hat{\Sigma}$ is full rank.

The default value of `solver` parameter in `LinearDiscriminantAnalysis` is `svd`. The use of SVD here should not be confused with the aforementioned use of SVD for '`lsqr`'. With the '`lsqr`' option, the SVD of $\hat{\Sigma}$ is used; however, the SVD used along with '`svd`' is primarily applied to the centered data matrix and dimensions with singular values greater than '`tol`' (current default 0.0001)—dimensions with singular values less than '`tol`' are discarded. This way, the scikit-learn implementation avoids even computing the covariance matrix *per se*. This could be an attractive choice in high-dimensional situations where $p > n$. The `svd` computation here relies on `scipy.linalg.svd`, which is based on DGESDD routine from LAPACK. Last but not least, the connection between Python and Fortran subroutines is created by `f2py` module that can be used to create a Python module containing automatically generated wrapper functions from Fortran routines. For more details see ([F2py, 2023](#)).

6.2.2 Logistic Regression

The linearity of decision boundaries in LDA was a direct consequence of the Gaussian assumption about class-conditional densities. Another way to achieve linear decision boundaries is to assume some monotonic transformation of posterior $P(Y = i|\mathbf{x})$ is

linear. In this regard, a well-known transformation is known as *logit* function. In particular, if $p \in (0, 1)$ then

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right). \quad (6.28)$$

The ratio inside the log function in (6.28) is known as *odds* (and another name for the logit function is log-odds). For example, if in a binary classification problem, the probability of having an observation from one class is $p = 0.8$, then odds of having an observation from that class is $\frac{0.8}{0.2} = 4$.

Let us consider a function known as *logistic sigmoid function* $\sigma(x)$ given by

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \quad (6.29)$$

Using $\text{logit}(p)$ as the argument of $\sigma(x)$ leads to

$$\sigma(\text{logit}(p)) = \frac{1}{1 + e^{-\text{logit}(p)}} = \frac{1}{1 + e^{-\log\left(\frac{p}{1-p}\right)}} = p, \quad (6.30)$$

which means the logistic sigmoid function is the inverse of the logit function. In *logistic regression*, we assume the logit of posteriors are linear functions of \mathbf{x} ; that is,

$$\text{logit}(P(Y = 1|\mathbf{x})) = \log\left(\frac{P(Y = 1|\mathbf{x})}{1 - P(Y = 1|\mathbf{x})}\right) = \mathbf{a}^T \mathbf{x} + b. \quad (6.31)$$

Replacing p and $\text{logit}(p)$ with $P(Y = 1|\mathbf{x})$ and $\mathbf{a}^T \mathbf{x} + b$ in (6.30), respectively, yields

$$P(Y = 1|\mathbf{x}) = \frac{1}{1 + e^{-(\mathbf{a}^T \mathbf{x} + b)}} = \frac{e^{(\mathbf{a}^T \mathbf{x} + b)}}{1 + e^{(\mathbf{a}^T \mathbf{x} + b)}}, \quad (6.32)$$

which is always between 0 and 1 as it should (because the posterior is a probability). In addition, because $P(Y = 0|\mathbf{x}) + P(Y = 1|\mathbf{x}) = 1$, we have

$$P(Y = 0|\mathbf{x}) = \frac{e^{-(\mathbf{a}^T \mathbf{x} + b)}}{1 + e^{-(\mathbf{a}^T \mathbf{x} + b)}} = \frac{1}{1 + e^{(\mathbf{a}^T \mathbf{x} + b)}}. \quad (6.33)$$

We can write (6.32) and (6.33) compactly as

$$P(Y = i|\mathbf{x}) = \frac{1}{1 + e^{(-1)^i(\mathbf{a}^T \mathbf{x} + b)}} = P(Y = 1|\mathbf{x})^i (1 - P(Y = 1|\mathbf{x}))^{(1-i)}, \quad i = 0, 1. \quad (6.34)$$

The classifier based on logistic regression is obtained by setting $g_i(\mathbf{x}) = P(Y = i|\mathbf{x}) = \frac{1}{1 + e^{(-1)^i(\mathbf{a}^T \mathbf{x} + b)}}$ in (6.3); that is,

$$\psi(\mathbf{x}) = \begin{cases} 1 & \text{if } \frac{1}{1 + e^{-(\mathbf{a}^T \mathbf{x} + b)}} - \frac{e^{-(\mathbf{a}^T \mathbf{x} + b)}}{1 + e^{-(\mathbf{a}^T \mathbf{x} + b)}} > 0, \\ 0 & \text{otherwise.} \end{cases} \quad (6.35)$$

At the same time, we can rewrite (6.35) as

$$\psi(\mathbf{x}) = \begin{cases} 1 & \text{if } \frac{1}{1 + e^{-(\mathbf{a}^T \mathbf{x} + b)}} > \frac{1}{2}, \\ 0 & \text{otherwise,} \end{cases} \quad (6.36)$$

which is equivalent to

$$\psi(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{a}^T \mathbf{x} + b > 0, \\ 0 & \text{otherwise.} \end{cases} \quad (6.37)$$

As seen in (6.37), the decision boundary of the binary classifier defined by the logistic regression is linear and, therefore, this is a linear classifier as well. What remains is to estimate the unknown parameters \mathbf{a} and b and replace them in (6.37) to have a functioning classifier.

A common approach to estimate the unknown parameters in logistic regression is to maximize the log likelihood of observing the labels given observations as a function of \mathbf{a} and b . Let $\boldsymbol{\beta} \stackrel{\Delta}{=} [b, \mathbf{a}]^T$ denote the $(p+1)$ -dimensional column vector of unknown parameters. With the assumption of having independent data, we can write

$$\begin{aligned} & \log(P(Y_1 = y_1, \dots, Y_n = y_n | \mathbf{x}_1, \dots, \mathbf{x}_n; \boldsymbol{\beta})) \\ & \stackrel{1}{=} \log\left(\prod_{j=1}^n P(Y_j = 1 | \mathbf{x}_j; \boldsymbol{\beta})^{y_j} (1 - P(Y_j = 1 | \mathbf{x}_j; \boldsymbol{\beta}))^{(1-y_j)}\right) \\ & = \sum_{j=1}^n y_j \log(P(Y_j = 1 | \mathbf{x}_j; \boldsymbol{\beta})) + (1 - y_j) \log(1 - P(Y_j = 1 | \mathbf{x}_j; \boldsymbol{\beta})), \end{aligned} \quad (6.38)$$

where $y_j = 0, 1, \stackrel{1}{=}$ follows from the compact form specified in (6.34), and $\boldsymbol{\beta}$ in $P(Y_1 = y_1, \dots, Y_n = y_n | \mathbf{x}_1, \dots, \mathbf{x}_n; \boldsymbol{\beta})$ shows that this probability is indeed a function of unknown parameters. This maximization approach is an example of a general method known as *maximum likelihood estimation* because we are trying to estimate values

of β (our model parameters) by maximizing the (log) likelihood of observing labels for the given observations. It is also common to convert the above maximization problem to a minimization by multiplying the above log likelihood function by -1 that results in a *loss function* (also known as error function) denoted by $e(\beta)$; that is to say,

$$e(\beta) = -\frac{1}{n} \sum_{j=1}^n y_j \log(P(Y_j = 1 | \mathbf{x}_j; \beta)) + (1 - y_j) \log(1 - P(Y_j = 1 | \mathbf{x}_j; \beta)), \quad (6.39)$$

which due to its similarity to *cross-entropy* function is known as cross-entropy loss (also referred to as *log loss*). It can be shown that the second derivative of $e(\beta)$ with respect to β (i.e., Hessian Matrix) is positive definite. This means that $e(\beta)$ is a convex function of β with a unique minimum. Therefore, setting the derivative of $e(\beta)$ with respect to β to zero should lead to the minimum; however, as these equations are non-linear, no closed-form solution exists and we need to rely on iterative optimization methods to estimate a and b (see `solver` subsection below for more information). Estimating a and b by one of these methods and replacing them in (6.37) results in the following classifier:

$$\psi(\mathbf{x}) = \begin{cases} 1 & \text{if } \hat{\mathbf{a}}^T \mathbf{x} + \hat{b} > 0, \\ 0 & \text{otherwise.} \end{cases} \quad (6.40)$$

Although it is common to see in literature labels are 0 and 1 when the loss function (6.39) is used, there is a more compact form to write the loss function of logistic regression if we assume labels are $y_j \in \{-1, 1\}$. First note that similar to the compact representation of posteriors in (6.34), we can write

$$P(Y_j = y_j | \mathbf{x}_j) = \frac{1}{1 + e^{-y_j(\mathbf{a}^T \mathbf{x}_j + b)}}, \quad y_j = -1, 1. \quad (6.41)$$

Using (6.41), we can write (6.39) in the following compact form where $y_j \in \{-1, 1\}$:

$$e(\beta) = \frac{1}{n} \sum_{j=1}^n \log(1 + e^{-y_j(\mathbf{a}^T \mathbf{x}_j + b)}). \quad (6.42)$$

We emphasize that using (6.39) with $y_j \in \{0, 1\}$ is equivalent to using (6.42) with $y_j \in \{-1, 1\}$. Therefore, these choices of labeling classes do not affect estimates of a and b , and the corresponding classifier.

An important and rather confusing point about logistic regression is that its main utility is in classification, rather than regression. By definition, we are trying to approximate and estimate numeric posterior probabilities and no surprise that it is called logistic *regression*; however, in terms of utility, almost always we are interested

to compare these posterior probabilities to classify a given observation to one of the classes (i.e., the classifier in the form of (6.36) or, equivalently, in terms of (6.37)). As a result, sometimes it is referred to as *logistic classification*.

Another interesting point is the number of parameters that should be estimated in logistic regression in comparison with that of LDA. For simplicity, let us consider binary classification. As can be seen in (6.37), $p + 1$ parameters should be estimated in logistic regression; however, for LDA we need $2p$ (for the means) + $p(p + 1)/2$ (for the pooled sample covariance matrix) + 1 (for prior probabilities) = $(p^2 + 5p)/2 + 1$. For example, for $p = 50$, logistic regression requires 51 parameters to be estimated whereas LDA requires 1376. As a result for a large number of features, logistic regression seems to have an upper hand in terms of errors induced by parameter estimation as it needs far fewer number of parameters to be estimated.

⊕ Extension to Multiclass Classification: To write (6.32) we assumed that $\text{logit}(P(Y = 1|\mathbf{x}))$ is a linear function of feature vectors and then used $P(Y = 0|\mathbf{x}) + P(Y = 1|\mathbf{x}) = 1$ to write (6.33). In multiclass case, we assume

$$P(Y = i|\mathbf{x}) = \frac{e^{(\mathbf{a}_i^T \mathbf{x} + b_i)}}{1 + \sum_{i=1}^{c-2} e^{(\mathbf{a}_i^T \mathbf{x} + b_i)}}, \quad i = 0, 1, \dots, c - 2, \quad (6.43)$$

and because the posterior probabilities should add up to 1, we can obtain the posterior probability of class $c - 1$ as

$$P(Y = c - 1|\mathbf{x}) = \frac{1}{1 + \sum_{i=1}^{c-2} e^{(\mathbf{a}_i^T \mathbf{x} + b_i)}}. \quad (6.44)$$

With these forms of posterior probabilities one can extend the loss function to multiclass and then use some iterative optimization methods to estimate the parameters of the model (i.e., \mathbf{a}_i and $b_i, \forall i$). The classifier is then obtained by using discriminants (6.43) and (6.44) in (6.1). Equivalently, we can set $g_i(\mathbf{x}) = \mathbf{a}_i^T \mathbf{x} + b_i, i = 0, \dots, c - 2$, and $g_{c-1}(\mathbf{x}) = 0$, and use them as discriminant functions in (6.1)—this is possible by noting that the denominators of all posteriors presented in (6.43) and (6.44) are the same and do not matter, and then taking the natural logarithm from all numerators. This latter form better shows the linearity of this classifier. As this classifier is specified by a series of approximation made on posterior probabilities $P(Y = i|\mathbf{x})$, which add up to 1 and together resemble a multinomial distribution, it is known as *multinomial logistic regression*.

Regularization: In practical applications, it is often helpful to consider a *penalized* form of the loss function indicated in (6.42); that is, to use a loss function that constrains the magnitude of coefficients in \mathbf{a} (i.e., penalizes possible large coefficients to end up with smaller coefficients). Because we are trying to *regularize* or *shrink* estimated coefficients, this approach is known as *regularization* or *shrinkage*. There are various forms of shrinkage penalty. One common approach is to minimize

$$e(\beta) = C \sum_{j=1}^n \log(1 + e^{-y_j(\mathbf{a}^T \mathbf{x}_j + b)}) + \frac{1}{2} \|\mathbf{a}\|_2^2, \quad (6.45)$$

where $\|\mathbf{a}\|_2$ is known as l_2 norm of $\mathbf{a} = [a_1, a_2, \dots, a_p]^T$ given by $\|\mathbf{a}\|_2 = \sqrt{\sum_{k=1}^p a_k^2} = \sqrt{\mathbf{a}^T \mathbf{a}}$ and C is a tuning parameter, which shows the relative importance of these two terms on estimating the parameters. As $C \rightarrow \infty$, the l_2 penalty term has no effect on the minimization and the estimates obtained from (6.45) become similar to those obtained from (6.42); however, as $C \rightarrow 0$, the coefficients in \mathbf{a} are shrunk to zero. This means the higher C , the less regularization we expect, which, in turn, implies a stronger fit on the training data. The optimal choice of C though is generally estimated (i.e., C is tuned) in the model selection stage. Because here we are using the l_2 norm of \mathbf{a} , minimizing (6.45) is known as l_2 regularization. The logistic regression trained using this objective function is known as logistic regression with l_2 penalty (also known as *ridge* penalty). There are other types of useful shrinkage such as l_1 regularization and elastic-net regularization. The l_1 regularization in logistic regression is achieved by minimizing

$$e(\beta) = C \sum_{j=1}^n \log(1 + e^{-y_j(\mathbf{a}^T \mathbf{x}_j + b)}) + \|\mathbf{a}\|_1, \quad (6.46)$$

where $\|\mathbf{a}\|_1 = \sum_{k=1}^p |a_k|$. Elastic-net regularization, on the other hand, is minimizing

$$e(\beta) = C \sum_{j=1}^n \log(1 + e^{-y_j(\mathbf{a}^T \mathbf{x}_j + b)}) + \nu \|\mathbf{a}\|_1 + \frac{1-\nu}{2} \|\mathbf{a}\|_2^2, \quad (6.47)$$

where ν is another tuning parameter that creates a compromise between l_1 and l_2 regularizations. In particular, when $\nu = 1$, (6.47) reduces to that of l_1 regularization, and when $\nu = 0$, (6.47) reduces to that of l_2 regularization.

A practically useful and interesting property of l_1 penalty is that if C is sufficiently small, the solution in (6.46) contains some coefficients (i.e., elements of \mathbf{a}) that are exactly zero and, therefore, are not selected to be part of the discriminant function. Practically, this is very useful because this property implies that l_1 penalty internally possesses a *feature selection* mechanism (i.e., choosing “important” features and discarding the rest). This is in contrast with the solution of (6.45) that uses l_2 penalty. Although using l_2 penalty also shrinks the magnitude of coefficients to zero, it does not set them to exact zero. Due to this property of l_1 penalty in shrinking and selecting features, this penalty is known as *lasso* (Tibshirani, 1996) (short for least absolute shrinkage and selection operator).

Scikit-learn implementation: The logistic regression classifier is implemented by the `LogisticRegression` class from the `sklearn.linear_model` module.

By default, scikit-learn implementation uses l_2 regularization with a $C = 1.0$ (see (6.45)). These choices, however, can be changed by setting the values of 'penalty' and C. Possible options for penalty parameter are 'l1' (see (6.46)), 'l2' (see (6.45)), and 'elasticnet' (see (6.47)). When the 'penalty' is set to 'elasticnet', one should also set ν in (6.47) by setting the value of 'l1_ratio' in LogisticRegression to some values $\in [0, 1]$ and also change the default value of solver from 'lbfgs' (as of scikit-learn version 1.1.2) to 'saga', which supports 'elasticnet' penalty. If l_1 regularization is desired, 'liblinear' solver can be used (also supports l_2).

 **More on solver:** There are a number of optimization algorithms that can be used in practice to solve the aforementioned minimization of the loss function to estimate the unknown parameters. These methods include:

- 1) steepest descent direction with a linear rate of convergence;
- 2) Newton's methods (e.g., iteratively reweighted least squares) that have fast rate of convergence (generally quadratic) but require computation of Hessian matrix of the objective function, which is itself computationally expensive and error-prone; and
- 3) quasi-Newton's methods that have superlinear convergence rate (faster than linear) and by avoiding explicit computation of Hessian matrix can be used even in situations that Hessian matrix is not positive definite—they use the fact that changes in gradient provides information about the Hessian along the search direction.

In the current version of scikit-learn (1.2.2), the default solver is 'lbfgs', which is short for Limited-memory BFGS (Broyden–Fletcher–Goldfarb–Shanno). It is a quasi-Newton method so it is efficient in the sense that it has a superlinear convergence rate. Its implementation relies on `scipy.optimize.minimize` with method being 'L-BFGS-B'. The second "B" in 'L-BFGS-B' stands for "box constraints" on variables; however, the 'lbfgs' uses this minimizer without any constraints, which leads to L-BFGS. This method, similar to two other solvers, namely, `newton-cg` and '`sag`', only supports l_2 regularization, which is by default used in scikit-learn implementation of logistic regression. If we want to use l_1 regularization, '`liblinear`' solver can be used (also supports l_2) and if the elastic net regularization is desired '`saga`' is currently the only solver to use. In terms of supporting multiclass classification, the following solvers support multinomial logistic regression '`lbfgs`', '`newton-cg`', '`sag`', and '`saga`' but '`liblinear`' only supports one-versus rest (OvR) scheme.

Example 6.1 In this example, we would like to train a logistic regression with l_2 regularization (LRR) for Iris flower classification using two features: sepal width and petal length. For this purpose, we first load the preprocessed Iris dataset that was prepared in Section 4.6:

```

import numpy as np
arrays = np.load('data/iris_train_scaled.npz')
X_train = arrays['X']
y_train = arrays['y']
arrays = np.load('data/iris_test_scaled.npz')
X_test = arrays['X']
y_test = arrays['y']
print('X shape = {}'.format(X_train.shape) + '\ny shape = {}'.format(y_train.shape))
print('X shape = {}'.format(X_test.shape) + '\ny shape = {}'.format(y_test.shape))

```

```

X shape = (120, 4)
y shape = (120,)
X shape = (30, 4)
y shape = (30,)

```

For illustration purposes, here we only use the second and third features (sepal width and petal length) of the data to develop and test our classifiers:

```

X_train = X_train[:,[1,2]]
X_test = X_test[:,[1,2]]
X_train.shape

```

```
(120, 2)
```

We use the technique used in Section 5.1.1 to draw the decision boundaries of Logistic Regression with l_2 Regularization:

```

%matplotlib inline
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.linear_model import LogisticRegression as LRR
color = ('aquamarine', 'bisque', 'lightgrey')
cmap = ListedColormap(color)

mins = X_train.min(axis=0) - 0.1
maxs = X_train.max(axis=0) + 0.1
x = np.arange(mins[0], maxs[0], 0.01)
y = np.arange(mins[1], maxs[1], 0.01)
X, Y = np.meshgrid(x, y)
coordinates = np.array([X.ravel(), Y.ravel()]).T
fig, axs = plt.subplots(1, 2, figsize=(6, 2), dpi = 150)
fig.tight_layout()
C_val = [0.01, 100]
for ax, C in zip(axs.ravel(), C_val):
    lrr = LRR(C=C)

```

```

lrr.fit(X_train, y_train)
Z = lrr.predict(coordinates)
Z = Z.reshape(X.shape)
ax.tick_params(axis='both', labelsize=6)
ax.set_title('LRR Decision Regions: C=' + str(C), fontsize=8)
ax.pcolormesh(X, Y, Z, cmap = cmap, shading='nearest')
ax.contour(X ,Y, Z, colors='black', linewidths=0.5)
ax.plot(X_train[y_train==0, 0], X_train[y_train==0, 1], 'g.', 
markersize=4)
ax.plot(X_train[y_train==1, 0], X_train[y_train==1, 1], 'r.', 
markersize=4)
ax.plot(X_train[y_train==2, 0], X_train[y_train==2, 1], 'k.', 
markersize=4)
if (C==C_val[0]): ax.set_ylabel('petal length (normalized)', 
fontsize=7)
ax.set_xlabel('sepal width (normalized)', fontsize=7)
print('The accuracy for C={} on the training data is {:.3f}'. 
format(C, lrr.score(X_train, y_train)))
print('The accuracy for C={} on the test data is {:.3f}'.format(C, 
lrr.score(X_test, y_test)))

```

The accuracy for $C=0.01$ on the training data is 0.817

The accuracy for $C=0.01$ on the test data is 0.833

The accuracy for $C=100$ on the training data is 0.950

The accuracy for $C=100$ on the test data is 0.967

As we said, a larger value of C points to less regularization and a stronger fit of the LRR model on the data. This is clear from the right plot in Fig. 6.2.2 where we observe that the data can be indeed separated pretty well by two lines. However, a lower C means more regularization as on the left plot in this figure. This is also apparent from the accuracy estimated on the training data. While a $C = 100$ can separate the training data with 95% accuracy (this is known as apparent accuracy or resubstitution accuracy), the model with $C = 0.01$ leads to a resubstitution accuracy of 81.7%. Here it happens that $C = 100$ exhibits a higher accuracy on the test data; however, it is not always the case. By that we mean:

- 1) it is not the case that a higher value of C is always better than a lower C or vice versa; and
- 2) a larger training accuracy does not necessarily lead to a larger test accuracy. If our model *overfits* the training data, it can show a high resubstitution accuracy but performs poorly on test data. ■

Quantifying Odds Ratio and Relative Change in Odds: An important question that is often raised when logistic regression classifier is used is the interpretation of its coefficients. To do so, we need to first look more closely to the concept of odds. Let r_0 denote the odds of $Y = 1$ (versus $Y = 0$); that is to say,

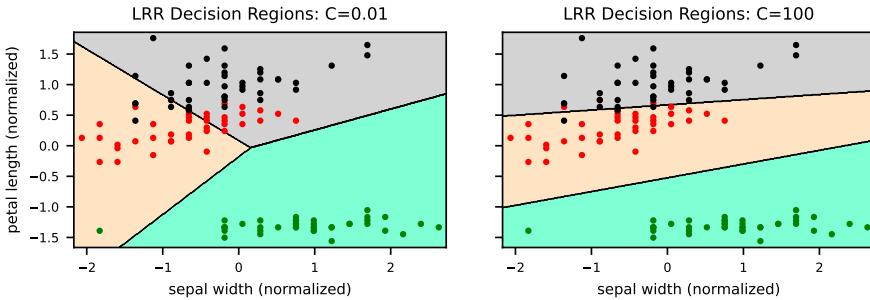


Fig. 6.3: The scatter plot of the normalized Iris dataset for two features and decision regions for logistic regression with l_2 regularization (LRR) for $C = 0.01$ (left) and $C = 100$ (right). The green, red, and black points show points corresponding to setosa, versicolor, and virginica Iris flowers, respectively. The LRR decision regions for each class are colored similarly.

$$r_0 \stackrel{\Delta}{=} \text{odds} = \frac{P(Y = 1|\mathbf{x})}{P(Y = 0|\mathbf{x})} = e^{\mathbf{a}^T \mathbf{x} + b} = e^{a_1 x_1 + a_2 x_2 + \dots + a_p x_p + b}, \quad (6.48)$$

where $\stackrel{\Delta}{=}$ is a consequence of (6.32) and (6.33), and $\mathbf{x} = [x_1, x_2, \dots, x_p]^T$ where we assume all features $x_i, i = 1, \dots, p$, are numeric features. Now, we define r_1 similarly to r_0 except that we assume it is the odds for one unit of increase in x_1 ; that is,

$$r_1 \stackrel{\Delta}{=} \text{odds} = \frac{P(Y = 1|\tilde{\mathbf{x}}_1)}{P(Y = 0|\tilde{\mathbf{x}}_1)} = e^{a_1(x_1+1) + a_2 x_2 + \dots + a_p x_p + b}, \quad (6.49)$$

where $\tilde{\mathbf{x}}_1 = [x_1 + 1, x_2, \dots, x_p]^T$. From (6.48) and (6.49), the *odds ratio* become

$$\text{odds ratio} = \frac{r_1}{r_0} = \frac{e^{a_1(x_1+1) + a_2 x_2 + \dots + a_p x_p + b}}{e^{a_1 x_1 + a_2 x_2 + \dots + a_p x_p + b}} = e^{a_1}. \quad (6.50)$$

This relationship is the key to interpreting coefficients of logistic regression. This is because it shows that an increase of one unit in x_1 leads to multiplying odds of $Y = 1$ by e^{a_1} , which is a non-linear function of a_1 . Nevertheless, if $a_1 > 0$, an increase of one unit in x_1 will increase the odds of $Y = 1$ according to $r_1 = r_0 e^{a_1}$. At the same time, if $a_1 < 0$, an increase of one unit in x_1 will decrease the odds of $Y = 1$, again according to $r_1 = r_0 e^{a_1}$. The same argument is applicable to any other coefficient in the logistic regression with numeric features; that is, a_i shows that a one unit increase in x_i changes the odds of $Y = 1$ according to $r_i = r_0 e^{a_i}$ where

$$r_i \stackrel{\Delta}{=} \text{odds} = \frac{P(Y = 1 | \tilde{\mathbf{x}}_i)}{P(Y = 0 | \tilde{\mathbf{x}}_i)} = e^{a_1 x_1 + \dots + a_i(x_i + 1) + \dots + a_p x_p + b}, \quad (6.51)$$

and where $\tilde{\mathbf{x}}_i = [x_1, \dots, x_i + 1, \dots, x_p]^T$. We can also define the *relative change in odds* (RCO) as a function of one unit in x_i :

$$\text{RCO\%} = \frac{r_i - r_0}{r_0} \times 100 = \frac{r_0 e^{a_i} - r_0}{r_0} \times 100 = (e^{a_i} - 1) \times 100. \quad (6.52)$$

As an example, suppose we have trained a logistic regression classifier where $a_2 = -0.2$. This means a one unit increase in x_2 leads to an RCO of -18.1% (decreases odds of $Y = 1$ by 18.1%).

Although (6.51) and (6.52) were obtained for one unit increase in x_i , we can extend this to any arbitrary number. In general a k unit increase (a negative k points to a decrease) in the value of x_i leads to:

$$\text{RCO\%} = \frac{r_i - r_0}{r_0} \times 100 = (e^{k a_i} - 1) \times 100. \quad (6.53)$$

Example 6.2 In this example, we work with a genomic data (gene expressions) taken from patients who were affected by oral leukoplakia. Data was obtained from Gene Expression Omnibus (GEO) with accession # GSE26549. The data includes 19897 features (19894 genes and three binary clinical variables) and 86 patients with a median follow-up of 7.11 years. Thirty-five individuals (35/86; 40.7%) developed oral cancer (OC) over time and the rest did not. The data matrix is stored in a file named “GenomicData_OralCancer.txt” and the variable names (one name for each column) is stored in “GenomicData_OralCancer_var_names.txt”, and both of these files are available in “data” folder. The goal is to build a classifier to classify those who developed OC from those who did not. The outcomes are stored in column named “oral_cancer_output” (-1 for OC patients and 1 for non-OC patients). For this purpose, we use a logistic regression with l_1 regularization with a $C = 16$.

```
# load the dataset
import numpy as np
import pandas as pd
data = pd.read_csv('data/GenomicData_OralCancer.txt', sep=" ", ↵
    header=None)
data.head()
```

	0	1	2	3	4	5	6	7	8	9	...	\
0	4.44	8.98	5.58	6.89	6.40	6.35	7.12	6.87	7.18	7.81	...	
1	4.59	8.57	6.57	7.25	6.44	6.34	7.40	6.91	7.18	8.12	...	
2	4.74	8.80	6.22	7.13	6.79	6.08	7.42	6.93	7.48	8.82	...	
3	4.62	8.77	6.32	7.34	6.29	5.65	7.12	6.89	7.27	7.18	...	
4	4.84	8.81	6.51	7.16	6.12	5.99	7.13	6.85	7.21	7.97	...	

```
19888 19889 19890 19891 19892 19893 19894 19895 19896 19897
0 6.08 5.49 12.59 11.72 8.99 10.87 1 0 1 1
1 6.17 6.08 13.04 11.36 8.96 11.03 1 0 0 1
2 6.39 5.99 13.29 11.87 8.63 10.87 1 1 0 -1
3 6.32 5.69 13.33 12.02 8.86 11.08 0 1 1 1
4 6.57 5.59 13.22 11.87 8.89 11.15 1 1 0 1
```

[5 rows x 19898 columns]

```
header = pd.read_csv('data/GenomicData_OralCancer_var_names.txt',  
                     header=None)  
data.columns=header.iloc[:,0]  
data.loc[:, "oral_cancer_output"] = data.loc[:, "oral_cancer_output"]*-1  
# to encode the positive class (oral cancer) as 1  
data.head()
```

0	OR4F17	SEPT14	OR4F16	GPAM	LOC100287934	LOC643837	SAMD11	KLHL17		
0	4.44	8.98	5.58	6.89	6.40	6.35	7.12	6.87		
1	4.59	8.57	6.57	7.25	6.44	6.34	7.40	6.91		
2	4.74	8.80	6.22	7.13	6.79	6.08	7.42	6.93		
3	4.62	8.77	6.32	7.34	6.29	5.65	7.12	6.89		
4	4.84	8.81	6.51	7.16	6.12	5.99	7.13	6.85		
0	PLEKHN1	ISG15	...	MRGPRX3	OR8G1	SPRR2F	NME2	GRINL1A	UBE2V1	\
0	7.18	7.81	...	6.08	5.49	12.59	11.72	8.99	10.87	
1	7.18	8.12	...	6.17	6.08	13.04	11.36	8.96	11.03	
2	7.48	8.82	...	6.39	5.99	13.29	11.87	8.63	10.87	
3	7.27	7.18	...	6.32	5.69	13.33	12.02	8.86	11.08	
4	7.21	7.97	...	6.57	5.59	13.22	11.87	8.89	11.15	
0	alcohol	smoking	histology	oral_cancer_output						
0	1	0	1							-1
1	1	0	0							-1
2	1	1	0							1
3	0	1	1							-1
4	1	1	0							-1

[5 rows x 19898 columns]

```
from sklearn.preprocessing import StandardScaler  
from sklearn.linear_model import LogisticRegression as LRR  
  
y_train = data.oral_cancer_output  
X_train = data.drop('oral_cancer_output', axis=1)  
scaler = StandardScaler().fit(X_train)  
X_train = scaler.transform(X_train)  
lrr = LRR(penalty = 'l1', C=16, solver = 'liblinear', random_state=42)  
lrr.fit(X_train, y_train)
```

```
LogisticRegression(C=16, penalty='l1', random_state=42, ↴
    solver='liblinear')
```

Here we can check the number of non-zero coefficients:

```
coeffs = lrr.coef_.ravel()
```

```
np.sum(coeffs != 0)
```

292

Therefore, there are 292 features (here all are genes) with non-zero coefficients in our linear logistic regression classifier. In other words, the use of l_1 regularization with logistic regression leads to 0 coefficients for 19606 features. This is the consequence of the internal feature selection mechanism of logistic regression with l_1 regularization that was mentioned in Section 6.2.2. This type of feature selection is also known as *embedded* feature selection (more on this in Chapter 10).

Next we sort the identified 292 features based on their odds ratio:

```
non_zero_coeffs = coeffs[coeffs != 0]
ORs=np.exp(non_zero_coeffs) #odds ratios
sorted_args = ORs.argsort()
ORs_sorted=ORs[sorted_args]
ORs_sorted[:10]
```

```
array([0.74175367, 0.76221689, 0.77267586, 0.77451542, 0.78316466, 0. ↴
    -8163162 , 0.83429172, 0.83761203, 0.84076963, 0.84205156])
```

```
feature_names = header.iloc[:-1,0].values
selected_features = feature_names[coeffs != 0]
selected_features_sorted = selected_features[sorted_args]
selected_features_sorted[0:10]
```

```
array(['DGCR6', 'ZNF609', 'CNO', 'RNASE13', 'BRD7P3', 'HHAT', 'UBXN1',
    'C15orf62', 'ORAI2', 'C1orf151'], dtype=object)
```

We would like to plot the RCOs for 10 genes that led to the highest decrease and 10 genes that led to the highest increase in RCO by one unit increase in their expressions.

```
RCO=(ORs_sorted-1.0)*100
selected_RCOs=np.concatenate((RCO[:10], RCO[-10:]))
features_selected_RCOs = np.concatenate((selected_features_sorted[:10], ↴
    -selected_features_sorted[-10:])))
features_selected_RCOs
```

```
array(['DGCR6', 'ZNF609', 'CNO', 'RNASE13', 'BRD7P3', 'HHAT', 'UBXN1',  
    ↪'C15orf62', 'ORAI2', 'C1orf151', 'KIR2DS5', 'LOC136242', 'SNORD33',  
    ↪'SEMG1', 'LOC100133746', 'SNORD41', 'C6orf10', 'C21orf84', 'DNAJA2',  
    ↪'MMP26'], dtype=object)
```

```
from matplotlib import pyplot  
import matplotlib.pyplot as plt  
  
plt.figure(figsize=(12,6))  
pyplot.bar(features_selected_RCOs, selected_RCOs, color='black')  
plt.xlabel('genes')  
plt.ylabel('RCO %')  
plt.xticks(rotation=90, fontsize=12)  
plt.yticks(fontsize=12)
```

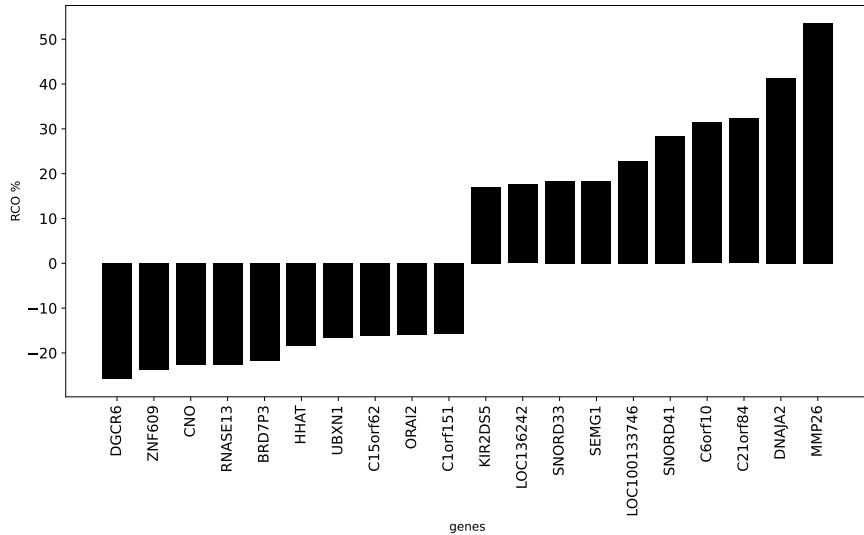


Fig. 6.4: RCO vs. genes. The figure shows 10 genes that led to the highest decrease and 10 genes that led to the highest increase in RCOs by one unit increase in their expressions.

Here we can see that a one unit increase in the expression of DGCR6 leads to an RCO of -25.8%. Does it mean a one unit decrease in its expression leads to an RCO of 25.8%? (see Exercise 8). ■



Although we introduced the Bayes classifier at the outset of this chapter, linear models are not the only available Bayes plug-in rules. For instance, the kNN classifier introduced in Chapter 5 is itself a Bayes plug-in rule that estimates the posterior $P(Y = i|\mathbf{x})$ used in (6.11) by the proportion of observations that belong to class i within the k nearest neighbors of \mathbf{x} . However, because it does not make a “parametric” assumption about the form of the posterior, it is categorized as a *non-parametric* Bayes plug-in rule (Braga-Neto, 2020).

6.3 Linear Models for Regression

Linear models for regression are models that *the estimate of the response variable is a linear function of parameters* (Bishop, 2006, p.138), (Hastie et al., 2001, p.44); that is,

$$f(\mathbf{x}) = \mathbf{a}^T \mathbf{x} + b. \quad (6.54)$$

When $p = 1$, (6.54) is known as *simple linear regression* and when $p > 1$ it is referred to as *multiple linear regression*. Given a set of training data $\mathbf{S}_{tr} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$, the most common approach to estimate $\boldsymbol{\beta} \triangleq [b, \mathbf{a}^T]^T$, which is the $(p+1)$ -dimensional column vector of unknown parameters, is to use the *least squares* method. In this method the goal is to minimize the *residual sum of squares* (RSS) and the solution is known as the *ordinary least squares* solution; that is,

$$\hat{\boldsymbol{\beta}}_{ols} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \text{RSS}(\boldsymbol{\beta}), \quad (6.55)$$

where

$$\text{RSS}(\boldsymbol{\beta}) = \sum_{j=1}^n (y_j - f(\mathbf{x}_j))^2 = \sum_{j=1}^n (y_j - \boldsymbol{\beta}^T \tilde{\mathbf{x}}_j)^2, \quad (6.56)$$

and where $\tilde{\mathbf{x}}_j$ is an *augmented feature vector* defined as $\tilde{\mathbf{x}}_j = [1, \mathbf{x}_j^T]^T, \forall j$; that is, we add a 1 in the first position and other elements are identical to \mathbf{x}_j . We can write (6.56) in a matrix form as

$$\text{RSS}(\boldsymbol{\beta}) = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2, \quad (6.57)$$

where \mathbf{X} is a $n \times (p+1)$ matrix such that the j^{th} row is the transpose of j^{th} augmented feature vector; that is, $\mathbf{X} = [\tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2, \dots, \tilde{\mathbf{x}}_n]^T$. By taking the gradient and setting to zero we write

$$\frac{\partial \text{RSS}(\boldsymbol{\beta})}{\partial \boldsymbol{\beta}} = -2\mathbf{X}^T\mathbf{y} + 2\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} = \mathbf{0}, \quad (6.58)$$

where we used the fact that for two vectors \mathbf{w} and $\boldsymbol{\beta}$ and a symmetric matrix \mathbf{W} ,

$$\frac{\partial \mathbf{w}^T \boldsymbol{\beta}}{\partial \boldsymbol{\beta}} = \frac{\partial \boldsymbol{\beta}^T \mathbf{w}}{\partial \boldsymbol{\beta}} = \mathbf{w}, \quad (6.59)$$

$$\frac{\partial \boldsymbol{\beta}^T \mathbf{W} \boldsymbol{\beta}}{\partial \boldsymbol{\beta}} = 2\mathbf{W}\boldsymbol{\beta}. \quad (6.60)$$

Assuming \mathbf{X} has full column rank $p+1 \leq n$ so that $\mathbf{X}^T\mathbf{X}$ is invertible, from (6.58) we obtain the well-known *normal equation* that provides a unique closed-form solution for the parameters that minimize the residual sum of squares:

$$\hat{\boldsymbol{\beta}}_{\text{ols}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}. \quad (6.61)$$

Similar to regularization techniques that we saw in Section 6.2.2, here rather than minimizing the residual sum of squares as in (6.58), one may minimize the regularized form of this objective function using ridge, lasso, or elastic-net penalty. In particular, the ridge regression solution is obtained by

$$\hat{\boldsymbol{\beta}}_{\text{ridge}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} [\text{RSS}(\boldsymbol{\beta}) + \alpha \|\mathbf{a}\|_2^2]. \quad (6.62)$$

Similar to minimizing (6.57), (6.62) can be minimized by setting the gradient to zero. This leads to the following closed-form solution:

$$\hat{\boldsymbol{\beta}}_{\text{ridge}} = (\mathbf{X}^T\mathbf{X} + \alpha \mathbf{I}_{p+1})^{-1}\mathbf{X}^T\mathbf{y}, \quad (6.63)$$

where \mathbf{I}_{p+1} is the identity matrix of $p+1$ dimension. The lasso solution is obtained as

$$\hat{\boldsymbol{\beta}}_{\text{lasso}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} [\text{RSS}(\boldsymbol{\beta}) + \alpha \|\mathbf{a}\|_1], \quad (6.64)$$

and the elastic-net solution is given by

$$\hat{\beta}_{\text{elastic-net}} = \underset{\beta}{\operatorname{argmax}} \left[\text{RSS}(\beta) + \alpha \nu \|\mathbf{a}\|_1 + \alpha \frac{1 - \nu}{2} \|\mathbf{a}\|_2^2 \right], \quad (6.65)$$

where both α and ν are tuning parameters that create a compromise between no regularization, and l_1 and l_2 regularizations. In particular when $\alpha = 0$, both (6.64) and (6.65) reduce to (6.61), which means both the lasso and the elastic-net reduce to the ordinary least squares. At the same time, for $\nu = 1$, (6.65) becomes (6.64); that is, the solution of elastic-net becomes that of lasso. In contrast with (6.45)-(6.47) where a larger C was indicator of less regularization, here α is chosen such that a larger α implies stronger regularization. This choice of notation is made to make the equations compatible with scikit-learn implementation of these methods. As there is no closed-form solution for lasso and elastic-net, iterative optimization methods are used to obtain the solution of (6.64) and (6.65). Last but not least, once we obtain an estimate of β from either of (6.61), (6.63), (6.64), or (6.65), we replace the estimate of parameters in (6.54) to estimate the target for a given \mathbf{x} .

Example 6.3 Suppose we have the following training data where for each value of predictor x , we have the corresponding y right below that. We would like to estimate the coefficient of a simple linear regression using ordinary least squares to have $\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x$.

Table 6.1: The data for Example 6.3.

x	-1	2	-1	0	0
y	0	2	2	0	-1

In doing so, we:

- a) use hand calculations to find $\hat{\beta}_0$ and $\hat{\beta}_1$; and
 - b) use hand calculations to determine the \hat{R}^2 evaluated on this training data.
- a) We need to find the solution from (6.61). For that purpose, first we need to construct matrix \mathbf{X} where as explained $\mathbf{X} = [\tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_2, \dots, \tilde{\mathbf{x}}_n]^T$ and where $\tilde{\mathbf{x}}$ is the augmented feature vector. Therefore, we have

$$\mathbf{X} = \begin{bmatrix} 1 & -1 \\ 1 & 2 \\ 1 & -1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 2 \\ 2 \\ 0 \\ -1 \end{bmatrix}. \quad (6.66)$$

Therefore, from (6.61), we have:

$$\hat{\beta}_{\text{ols}} = \left(\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ -1 & 2 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 1 & 2 \\ 1 & -1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix} \right)^{-1} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ -1 & 2 & -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 2 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} \frac{3}{5} \\ \frac{1}{3} \end{bmatrix}. \quad (6.67)$$

Therefore, $\beta_0 = \frac{3}{5}$ and $\beta_1 = \frac{1}{3}$.

b) From Section 5.2.2, in which \hat{R}^2 was defined, we first need to find RSS and TSS:

$$\text{RSS} = \sum_{i=1}^m (y_i - \hat{y}_i)^2, \quad (6.68)$$

$$\text{TSS} = \sum_{i=1}^m (y_i - \bar{y})^2, \quad (6.69)$$

where \hat{y}_i is the estimate of y_i for x_i obtained from our model by $\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_i$, and \bar{y} is the average of responses. From part (a),

$$\hat{y}_1 = \frac{3}{5} - \frac{1}{3} = \frac{4}{15}, \quad \hat{y}_2 = \frac{3}{5} + \frac{2}{3} = \frac{19}{15}, \quad \hat{y}_3 = \hat{y}_1, \quad \hat{y}_4 = \frac{3}{5}, \quad \hat{y}_5 = \hat{y}_4.$$

Therefore,

$$\text{RSS} = \left(0 - \frac{4}{15}\right)^2 + \left(2 - \frac{19}{15}\right)^2 + \left(2 - \frac{4}{15}\right)^2 + \left(0 - \frac{3}{5}\right)^2 + \left(-1 - \frac{3}{5}\right)^2 \approx 6.53.$$

To calculate the TSS, we need \bar{y} , which is $\bar{y} = \frac{0+2+2+0-1}{5} = \frac{3}{5}$. Therefore, we have

$$\text{TSS} = \left(0 - \frac{3}{5}\right)^2 + \left(2 - \frac{3}{5}\right)^2 + \left(2 - \frac{3}{5}\right)^2 + \left(0 - \frac{3}{5}\right)^2 + \left(-1 - \frac{3}{5}\right)^2 = \frac{36}{5} = 7.2.$$

Thus,

$$\hat{R}^2 = 1 - \frac{\text{RSS}}{\text{TSS}} = 1 - \frac{6.53}{7.2} = 0.092,$$

which is relatively a low \hat{R}^2 . ■

Scikit-learn implementation: The ordinary least squares, ridge, lasso, and elastic-net solution of linear regression are implemented in `LinearRegression`, `Ridge`,

Lasso, and ElasticNet classes from `sklearn.linear_model` module, respectively. For `LinearRegression`, Lasso, and ElasticNet, scikit-learn currently uses a fixed algorithm to estimate the solution (so no parameter to change for the “solver”), but for Ridge there are plenty of options. More details on solvers are given next.

 **More on solver:** To minimize $\|\mathbf{y} - \mathbf{X}\beta\|_2^2$ in (6.57), we assumed $p + 1 \leq n$ to present the normal equation shown in (6.61). However, similar to `lsqr` solver that we discussed in Section 6.2.1 for `LinearDiscriminantAnalysis`, the `lsqr` solver in `LinearRegression` also relies on `scipy.linalg.lstsq`, which as mentioned before by default is based on DGELSD routine from LAPACK and finds the minimum-norm solution to the problem $\mathbf{X}\beta = \mathbf{y}$; that is to say, minimizes $\|\mathbf{y} - \mathbf{X}\beta\|_2^2$ in (6.57). As for the ridge regression implemented via `Ridge` class, there are currently several solvers with the current default being `auto`, which determines the solver automatically. Some of these solvers are implemented similar to our discussion in Section 6.2.1. For example, `svd` solver computes the SVD decomposition of \mathbf{X} using `scipy.linalg.svd`, which is itself based on DGESDD routine from LAPACK. Those dimensions with a singular value less than 10^{-15} are discarded. Once the SVD decomposition is calculated, the ridge solution given in (6.63) is computed (using `_solve_svd()` function). Here we explain the rationale. Suppose \mathbf{X} is an arbitrary $m \times n$ real matrix. We can decompose \mathbf{X} as $\mathbf{X} = \mathbf{UDV}^T$ where $\mathbf{UU}^T = \mathbf{U}^T\mathbf{U} = \mathbf{I}_m$, $\mathbf{VV}^T = \mathbf{V}^T\mathbf{V} = \mathbf{I}_n$, and \mathbf{D} is a rectangular $m \times n$ diagonal matrix with $\min\{m, n\}$ non-negative real numbers known as *singular values* — this is known as SVD decomposition of \mathbf{X} . Then we can write (6.63) as follows:

$$\begin{aligned}\hat{\boldsymbol{\beta}}_{\text{ridge}} &= (\mathbf{X}^T \mathbf{X} + \alpha \mathbf{I}_{p+1})^{-1} \mathbf{X}^T \mathbf{y} \\ &= (\mathbf{V} \mathbf{D}^2 \mathbf{V}^T + \alpha \mathbf{V} \mathbf{V}^T)^{-1} \mathbf{V} \mathbf{D} \mathbf{U}^T \mathbf{y} \\ &= (\mathbf{V}^T)^{-1} (\mathbf{D}^2 + \alpha \mathbf{I}_{p+1})^{-1} \mathbf{V}^{-1} \mathbf{V} \mathbf{D} \mathbf{U}^T \mathbf{y} \\ &= \mathbf{V} (\mathbf{D}^2 + \alpha \mathbf{I}_{p+1})^{-1} \mathbf{D} \mathbf{U}^T \mathbf{y},\end{aligned}\tag{6.70}$$

and note that because all matrices in $(\mathbf{D}^2 + \alpha \mathbf{I}_{p+1})^{-1} \mathbf{D}$ are diagonal, this term is also a diagonal matrix where each diagonal element in each dimension can be computed easily by first adding α to the square of the singular value for that dimension, inverting, and then multiplying by the same singular value. The `cholesky` choice for the ridge solver relies on `scipy.linalg.solve`, which is based on DGETRF and DGETRS from LAPACK to compute the LU factorization and then find the solution based on the factorization. As for Lasso and ElasticNet, scikit-learn currently uses coordinate descent solver.

Exercises:

Exercise 1:

- 1) Use appropriate reader function from pandas to read the `GenomicData_orig.csv` in data folder (this is the same dataset we used in Exercise 2 in Chapter 2). Drop the first two columns as they have redundant information;
- 2) Before splitting the data as discussed next, seed NumPy random number generator (RNG) as `np.random.seed(42)`. This has a type of global effect on any function that uses NumPy and makes your results reproducible. In contrast, setting the seed in `train_test_split` by `random_state=42` makes only your train-test split reproducible and has no effect on random generators, for example, used in `liblinear` solver;
- 3) Split the data into test and train with a 0.25 test size;
- 4) Impute the missing data in both training and test using `SimpleImputer` transformer (set the `strategy` to `median`) from `sklearn.impute` module (note that for test data imputation, you need to use the extracted information from training data);
- 5) Standardize features;
- 6) Train two logistic regression with l_1 penalty (use ‘`liblinear`’ solver) and $C = 0.05$ and $C = 10$. What are their accuracies on test data?
- 7) Train an LDA classifier (default solver). Report its accuracy;
- 8) How many features are “selected” (i.e., have non-zero coefficients) in logistic regression with l_1 for $C = 0.05$ and for $C = 10$ and how many are selected by LDA (have non-zero coefficients)?

Exercise 2: Suppose we have a training dataset collected for a binary classification problem. Using this training data, the class-specific sample means are $\hat{\mu}_0 = [4, 1]^T$ and $\hat{\mu}_1 = [0, -1]^T$, and the pooled sample covariance matrix is $\hat{\Sigma} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$. We would like to use these estimates to construct an LDA classifier. Determine the range of $P(Y = 1)$ such that an observation $\mathbf{x} = [1.5, 0]^T$ is classified to class 1 using the constructed LDA classifier.

Exercise 3: Suppose we have a three-class classification problem where classes are normally distributed with means $\mu_0^T = [0, 1]^T$, $\mu_1^T = [1, 0]^T$, and $\mu_2^T = [0.5, 0.5]^T$ and a common covariance matrix

$$\Sigma = \begin{bmatrix} 2 & 0 \\ 0 & 4 \end{bmatrix} .$$

Classify an observation $\mathbf{x} = [1, 1]^T$ according to the Bayes rule (assuming classes are equiprobable).

⊕ **Exercise 4:** Prove that the optimal classifier $\psi_B(\mathbf{x})$ (also known as Bayes classifier) is obtained as

$$\psi_B(\mathbf{x}) = \begin{cases} 1 & \text{if } P(Y = 1|\mathbf{x}) > P(Y = 0|\mathbf{x}), \\ 0 & \text{otherwise.} \end{cases} \quad (6.71)$$

Exercise 5: The class-conditional probability density functions in a three class classification problem (labeled as 1, 2, and 3) are uniform distributions depicted in the figure below. Assuming $P(Y = 1) = 2P(Y = 2) = 3P(Y = 3)$, what is the class to which $x = 1.75$ is classified based on the Bayes classifier?

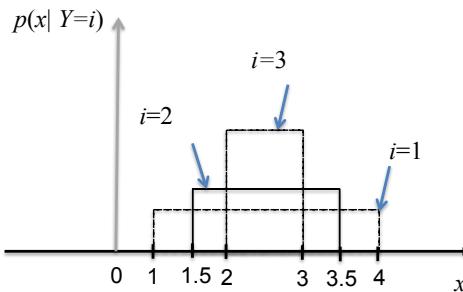


Fig. 6.5: Class-conditional densities used Exercise 5.

- A) Class 1
- B) Class 2
- C) Class 3
- D) there is a tie between Class 1 and Class 3
- E) there is a tie between Class 2 and Class 3

Exercise 6: Use the two assumptions in development of LDA in (6.10) to achieve (6.13).

Exercise 7: Show that the Bayes plug-in rule for Gaussian class-conditional densities with different covariance matrices has the following form, which is known as quadratic discriminant analysis (QDA):

$$\psi(\mathbf{x}) = \begin{cases} 1 & \text{if } \frac{1}{2}\mathbf{x}^T \mathbf{A}_1 \mathbf{x} + \mathbf{A}_2 \mathbf{x} + a > 0, \\ 0 & \text{otherwise,} \end{cases} \quad (6.72)$$

where

$$\mathbf{A}_1 = \hat{\Sigma}_0^{-1} - \hat{\Sigma}_1^{-1}, \quad (6.73)$$

$$\mathbf{A}_2 = \hat{\mu}_1^T \hat{\Sigma}_1^{-1} - \hat{\mu}_0^T \hat{\Sigma}_0^{-1}, \quad (6.74)$$

$$a = \frac{1}{2} \hat{\mu}_0^T \hat{\Sigma}_0^{-1} \hat{\mu}_0 - \frac{1}{2} \hat{\mu}_1^T \hat{\Sigma}_1^{-1} \hat{\mu}_1 - \frac{1}{2} \log \left(\frac{|\hat{\Sigma}_1|}{|\hat{\Sigma}_0|} \right) + \log \left(\frac{P(Y=1)}{1-P(Y=1)} \right), \quad (6.75)$$

and where

$$\hat{\Sigma}_i = \frac{1}{n_i - 1} \sum_{y_j=i} (\mathbf{x}_j - \hat{\mu}_i)(\mathbf{x}_j - \hat{\mu}_i)^T. \quad (6.76)$$

Exercise 8: For a binary logistic regression classifier, a one unit increase in the value of a numeric feature leads to an RCO of α .

- a) What will be the RCO for one unit decrease in the value of this numeric feature?
- b) Suppose $\alpha = 0.2$ (i.e., 20%). What is the RCO for one unit decrease in the value of the feature?

Exercise 9: Suppose we have two data matrices \mathbf{X}^{train} and \mathbf{X}_1^{test} . We train a multiple linear regression model using \mathbf{X}^{train} and in order to assess that model using RSS metric, we apply it on \mathbf{X}_1^{test} . This leads to a value of RSS denoted as RSS_1 . Then we augment \mathbf{X}_1^{test} by 1 observation and build a larger test dataset of 101 observations denoted as \mathbf{X}_2^{test} (that is to say, 100 observations in \mathbf{X}_2^{test} are the same observations in \mathbf{X}_1^{test}). Applying the previously trained model on \mathbf{X}_2^{test} leads to a value of RSS denoted as RSS_2 . Which of the following is true? Explain why?

- A) Depending on the actual observations, both $RSS_1 \geq RSS_2$ and $RSS_2 \geq RSS_1$ are possible
- B) $RSS_1 \geq RSS_2$
- C) $RSS_2 \geq RSS_1$

Exercise 10: Suppose in a binary classification problem with equal prior probability of classes, the class conditional densities are multivariate normal distributions with a common covariance matrix: $\mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma})$ and $\mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma})$. Prove that the Bayes error ε_B is obtained as

$$\varepsilon_B = \Phi \left(-\frac{\Delta}{2} \right), \quad (6.77)$$

where $\Phi(\cdot)$ denotes the cumulative distribution function of a standard normal random variable, and Δ (known as the Mahalanobis distance between the two distributions) is given by

$$\Delta = \sqrt{(\boldsymbol{\mu}_0 - \boldsymbol{\mu}_1)^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{\mu}_0 - \boldsymbol{\mu}_1)}. \quad (6.78)$$

Exercise 11: Suppose in a binary classification problem with equiprobable classes (class 0 vs. class 1), the class-conditional densities are 100-dimensional normal

distributions with a common covariance matrix, which is a 100×100 identity matrix, and means μ_0 for class 0 and μ_1 for class 1, which are two known 100-dimensional vectors whereas each dimension of μ_0 has a value that is different from the value of the corresponding dimension in μ_1 .

Benjamin considers only the first five dimensions in these normal distributions and constructs the Bayes classifier. We denote the Bayes error of this 5-dimensional classification problem by ϵ_5^* . Ava considers only the first fifty dimensions in these Gaussian distributions and constructs the Bayes classifier. We denote the Bayes error of this 50-dimensional classification problem by ϵ_{50}^* . Which one shows the relationship between ϵ_5^* and ϵ_{50}^* ?

Hint: use the result of Exercise 10.

- A) we have $\epsilon_5^* > \epsilon_{50}^*$
- B) we have $\epsilon_5^* < \epsilon_{50}^*$
- C) Depending on the actual values of vectors μ_0 and μ_1 , both $\epsilon_5^* > \epsilon_{50}^*$ and $\epsilon_5^* < \epsilon_{50}^*$ are possible
- D) we have $\epsilon_5^* = \epsilon_{50}^*$

Exercise 12: Suppose we would like to use scikit-learn to solve a multiple linear regression problem using l_1 regularization. Which of the following is a possible option to use?

- A) “sklearn.linear_model.LogisticRegression” class with default choices of parameters
- B) “sklearn.linear_model.LogisticRegression” class by changing the default choice of “penalty” and “solver” parameters to “l1” and “liblinear”, respectively.
- C) “sklearn.linear_model.Lasso” class with default choices of parameters
- D) “sklearn.linear_model.Ridge” class with default choices of parameters

Exercise 13: Suppose we have the following training data where x and y denote the value of the predictor x and the target value, respectively. We would like to use this data to train a simple linear regression using ordinary least squares. What is \hat{y} (the estimate of y) for $x = 10$?

x	0	2	1	-5	3	-1
y	4	4	2	0	0	2

- A) 4
- B) 4.5
- C) 5
- D) 5.5
- E) 6
- F) 6.5

 **Exercise 14:**

Suppose we have an electrical device and at each time point t the voltage at a specific part of this device can only vary in the interval $[1, 5]$. If the device is faulty, this voltage, denoted $V(t)$, varies between $[2, 5]$, and if it is non-faulty, the voltage can be any value between $[1, 3]$. In the absent of any further knowledge about the problem, (it is reasonable to) assume $V(t)$ is uniformly distributed in these intervals (i.e., it has uniform distributions when it is faulty or non-faulty). Based on these assumptions, we would like to design a classifier based on $V(t)$ that can “optimally” (insofar as the assumptions hold true) classifies the status of this device at each point in time. What is the optimal classifier at each point in time? What is the error of the optimal classifier at each time point?

Exercise 15: Suppose in a binary classification problem with equiprobable classes (class 0 vs. class 1), the class-conditional densities are 100-dimensional normal distributions with a common covariance matrix, which is a 100×100 identity matrix. The mean of each feature for class 1 is 0; however, the mean of feature i for class 0, denoted μ_i , is \sqrt{i} ; that is, $\mu_i = \sqrt{i}, i = 1, 2, \dots, 100$.

Benjamin considers the first 30 dimensions (i.e., $i = 1, 2, \dots, 30$) in these normal distributions and constructs the Bayes classifier. We denote the Bayes error of this 30-dimensional classification problem by ϵ_{30}^* . Ava considers the last five dimensions (i.e., $i = 96, 97, \dots, 100$) in these Gaussian distributions and constructs the Bayes classifier. We denote the Bayes error of this 5-dimensional classification problem by ϵ_5^* . Which one shows the relationship between ϵ_5^* and ϵ_{30}^* ?

- A) we have $\epsilon_5^* > \epsilon_{30}^*$
- B) we have $\epsilon_5^* < \epsilon_{30}^*$
- C) we have $\epsilon_5^* = \epsilon_{30}^*$
- D) Depending on the training data, both $\epsilon_5^* > \epsilon_{30}^*$ and $\epsilon_5^* < \epsilon_{30}^*$ are possible



Chapter 7

Decision Trees

Decision trees are nonlinear graphical models that have found important applications in machine learning mainly due to their interpretability as well as their roles in other powerful models such as random forests and gradient boosting regression trees that we will see in the next chapter. Decision trees resemble the principles of “20-questions” game in which a player chooses an object and the other player asks a series of yes-no questions to be able to guess the object. In *binary decision trees*, the set of arbitrary “yes-no” questions in the game translate to a series of *standard* yes-no (thereby binary) questions that we identify from data and the goal is to estimate the value of the target variable for a given feature vector \mathbf{x} . As opposed to arbitrary yes-no questions in the game, the questions in decision rules are standardized to be able to devise a learning algorithm. Unlike linear models that result in linear decision boundaries, decision trees partition the feature space into hyper-rectangular decision regions and, therefore, they are nonlinear models. In this chapter we describe the principles behind training a popular type of decision trees known as CART (short for Classification and Regression Tree). We will discuss development of CART for both classification and regression.

7.1 A Mental Model for House Price Classification

Consider a set of numeric feature variables. Given a feature vector \mathbf{x} , a convenient form to standardize possible yes-no questions that could be asked to estimate the target is: “is feature i less or equal to a threshold value?” These questions can then be organized as a *directed rooted tree* all the way to the terminal nodes known as *leaf* nodes. A directed rooted tree has a single node known as *root* from which all other nodes are reachable. There is a unique path from the root to any other node v in the tree. Along this path, the node that immediately connects to a node v is known as the *parent* of v , whereas v is referred to as the *child* of the parent node. As a result, the only node that has no parent is the root node and all other nodes have one parent node. At the same time, the *leaf* nodes (terminal nodes) have no children. In

a *decision tree*, leaf nodes contain the decision made for \mathbf{x} (i.e., the estimate of the target). Once a decision tree is trained, it is common to visualize its graph to better understand the relationship between questions.

In this section, we use a hypothetical example with which we introduce a number of practical concepts used in training decision trees. These concepts are: 1) splitting node; 2) impurity; 3) splitter; 4) best split; 5) maximum depth; and 6) maximum leaf nodes. In Section 5.2.2, we saw an application in which we aimed to solve a regression problem to estimate the median house price of a neighborhood (MEDV) based on a set of features. Suppose we wish to solve this example but this time we entirely rely on our past experiences to design a simple binary decision tree that can help classify MEDV to “High” or “Low” (relative to our purchasing power) based on three variables: average number of rooms per household (RM), median house age in block group (AGE), and median income in the district (INCM).

Suppose from our past experience (think of this as past data, or “training” data), we believe that RM is an important factor in the sense that in many cases (but not all), if RM is four or more, MEDV is “High”, and otherwise “Low”. Therefore, we consider $RM \leq 4$ as the first *splitting node* in our decision tree (see Fig. 7.1a). Here, we refer to this as a splitting node because the answer to the question “is $RM \leq 4$?” splits our training data (past experiences) into two subsets: one subset contains all neighborhoods with $RM \leq 4$ and the other subset contains all neighborhoods with $RM > 4$. This splitting combined with our past experiences also means that at this stage we can even make a decision based on “ $RM \leq 4$?” because we believe neighborhoods with $RM \leq 4$ *generally* have low MEDV and neighborhoods with $RM > 4$ *generally* have high MEDV.

There are two important concepts hidden in the aforementioned statements: 1) impurity; and 2) splitter. Notice that we said $RM \leq 4$ is important for “many cases (but not all)”. In other words, if we decide to make a decision at this stage based on $RM \leq 4$, this decision is not perfect. Even though “ $RM \leq 4$?” splits the data into two subsets, we can not claim that all neighborhoods in each subset are entirely from one class. In other words, the split achieved by $RM \leq 4$ is *impure* with respect to class labels. Nevertheless, the reason we chose this specific split among all possible splits was that we believe it can minimize the *impurity* of data subsets; that is to say, one subset contains a large number of neighborhoods with high MEDV and the other subset contains a large number of neighborhoods with low MEDV. This strategy was indeed a *splitting strategy*, also known as *splitter*; that is to say, our “splitter” was to identify the *best split* in the sense of minimizing the impurity of data subsets created as the result of yes/no answer to a standardized question of “a variable \leq a threshold?” This means to find the best split, we need to identify the combination of variable-threshold that minimizes the impurity of data subsets created as the result of the split.

Now we need to decide whether we should stop further splitting or not. Suppose we believe that we can still do better in terms of cutting up the decision space and, for that reason, we refer back to our training data. In our experience, new houses with $AGE \leq 3$ have high price even if $RM \leq 4$; otherwise, they have low price. At the same time, for $RM > 4$, the price would potentially depend on INCM. In

particular, if these houses are in neighborhoods with a relatively low INCM, the price is still low. This leads to a decision tree grown as much as shown in Fig. 7.1b with two additional splitting nodes $AGE \leq 3$ and $INCM \leq 5$ and 3 leaf nodes at which a decision is made about MEDV. Next based on our past experiences, we believe that if $RM > 4$ & $INCM > 5$, the price depends on AGE (Fig. 7.1c). In particular, $RM > 4$ & $INCM > 5$ & $AGE \leq 20$, MEDV is high and for $RM > 4$ & $INCM > 5$ & $AGE > 20$, MEDV is low. At this point we stop splitting because either we think terminal nodes are pure enough or further splitting makes the tree too complex. Fig. 7.1d shows the final decision tree.

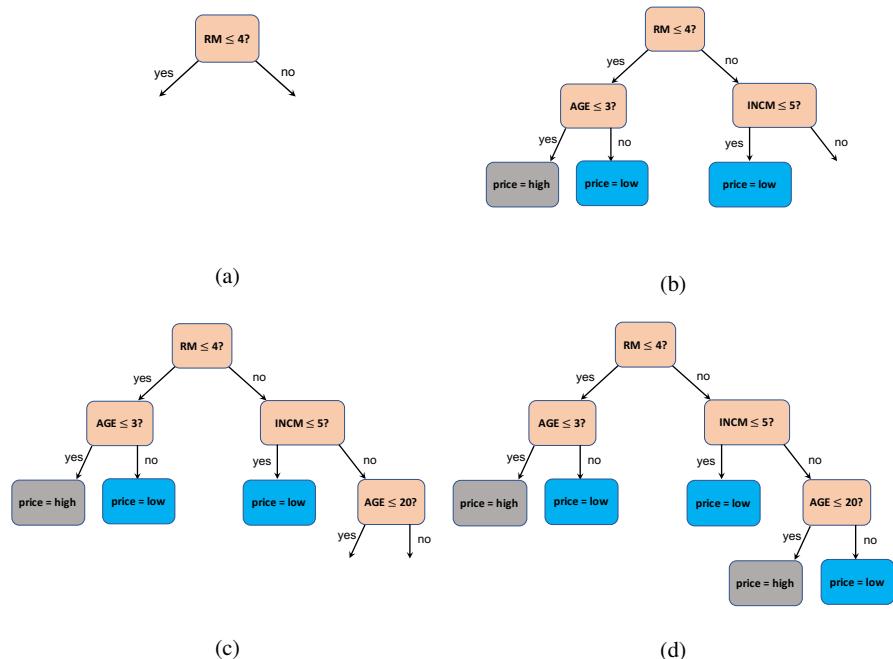


Fig. 7.1: Development of a mental model for house price classification. Here the decision tree is grown based on our past experiences.

A question that is raised is, why don't we want the tree to become too complex? Suppose we grow a tree to the extent that all leaf nodes become pure; that is, they contain training data from one class. This is generally doable unless all observations in a node have the same values but from different classes in which case no further splitting is possible and the node remains impure. Is it a good practice to continue growing the tree until all nodes are pure? Not necessarily because it is very likely that the tree overfits the training data; that is, the tree closely follows the training data but does not perform well on novel observations that are not part of training. In other words, the decision tree does not generalize well.

A follow-up question is then what are some properties using which we can control the structural complexity of a tree? One metric to control the structural complexity of a binary tree is the maximum number of links between the root node and any of the leaf nodes. This is known as the *maximum depth* — the depth of a node is the number of links between the node and the root node. For example, the maximum depth of the decision tree in Fig. 7.1d is 3. If we preset the maximum depth of a binary decision tree, it limits both the number of leaf nodes and the total number of nodes. Suppose before growing a binary decision tree, we set 3 as the maximum depth. What is the maximum number of leaf nodes and the maximum number of nodes (including the root and other splitting nodes)? The maximum happens if all leaf nodes have depth of 3. Therefore, we have a maximum of 8 leaf nodes and at most 1 (root) + 2 + 4 + 8 (leaf) = 15 nodes. In general for a binary decision tree of depth m , we have at most 2^m leaf nodes and $2^{m+1} - 1$ nodes. Alternatively, rather than limiting the maximum depth, one may restrict the maximum number of leaf nodes. That will also restrict the maximum depth and the maximum number of nodes.

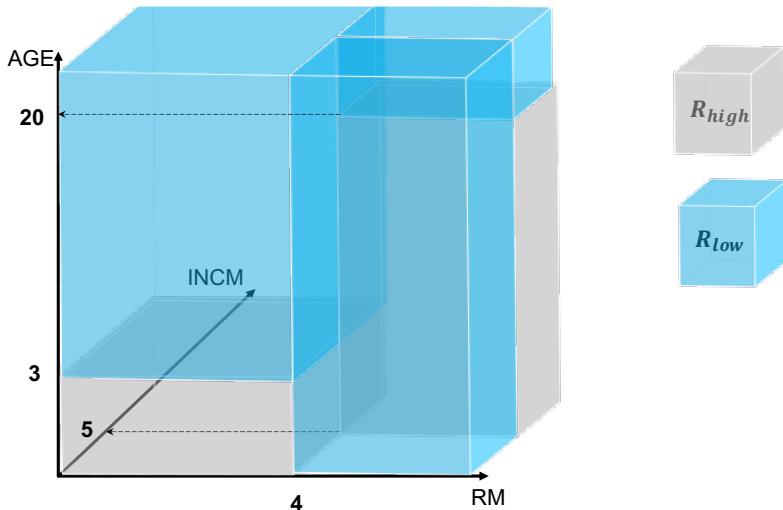


Fig. 7.2: Partitioning the feature space into five boxes using the decision tree shown in Fig. 7.1d. The decision regions for the “high”- and “low”-MEDV classes denoted by \mathbf{R}_{high} and \mathbf{R}_{low} are shown in gray and blue, respectively. There are two gray boxes and three blue boxes corresponding to the labels given by the leaf nodes in Fig. 7.1d.

As we said before, decision trees partition the feature space into hyper-rectangles (boxes). Each box corresponds to a decision rule obtained by traversing from the root node to the leaf that includes the label for any observation falling into that rectangular region. Fig. 7.2 shows partitioning of the feature space into five boxes

using the decision tree shown in Fig. 7.1d. Although in the figure these boxes appear to have a finite size, each box has an infinite length over at least one of the RM, AGE, or INCM axes because, in theory at least, there is no specific upper limit for these variables. As an example, let us consider the gray box on the left bottom corner. This box corresponds to the gray label assigned by traversing from the root to left most leaf in Fig. 7.1d. This traverse corresponds to the following decision rule: assign a “high” label when $RM \leq 4$ AND $AGE \leq 3$.

Although in this example we introduced several important concepts used in training and functionality of decision trees, we did not really present any specific algorithm using which a tree is trained from a data at hand. There are multiple algorithms for training decision trees, for instance, CHAID (Kass, 1980), ID3 (Quinlan, 1986), C4.5 (Quinlan, 1993), CRUISE (Kim and Loh, 2001), and QUEST (Loh and Shih, 1997), to just name a few. Nevertheless, the binary recursive partitioning mechanism explained here imitates closely the principles behind training a popular type of decision trees known as CART (short for Classification and Regression Trees) (Breiman et al., 1984). In the next section, we will discuss this algorithm for training and using decision trees in both classification and regression.

7.2 CART Development for Classification:

7.2.1 Splits

For a numeric feature x_j , CART splits data according to the outcome of questions of the form: “ $x_j \leq \theta$?” For a nominal feature (i.e., categorical with no order) they have the form “ $x_j \in A$ ” where A is a subset of all possible values taken by x_j . Regardless of testing a numeric or nominal feature, these questions check whether the value of x_j is in a subset of all possible values that it can take. Hereafter, and for simplicity we only consider numeric features. As the split of data is determined by the j^{th} coordinate (feature) in the feature space and a threshold θ , for convenience we refer to a split as $\xi = (j, \theta) \in \Xi$ where Ξ denotes the set of all possible ξ that create a unique split of data. It is important to notice that the size of Ξ is not infinite. As an example, suppose in a training data, feature x_j takes the following values: 1.1, 3.8, 4.5, 6.9, 9.2. In such a setting, for example, $\xi = (j, 4.0)$ and $\xi = (j, 4.4)$, create the same split of the data over this feature: one subset being $\{1.1, 3.8\}$ and the other being $\{4.5, 6.9, 9.2\}$. As a matter of fact we obtain the same split for any $\xi = (j, \theta)$ where $\theta \in (3.8, 4.5)$. Therefore, it would be sufficient to only examine one of these splits. In CART, this arbitrary threshold between two consecutive values is taken as their halfway point. Assuming feature x_j , $j = 1, \dots, p$, has v_j distinct values in a training data, then the cardinality of Ξ is $(\sum_{j=1}^p v_j) - p$. We denote the set of candidate values of θ (halfways between consecutive values) for feature x_j by Θ_j .

7.2.2 Splitting Strategy

The splitting strategy in CART is similar to what we saw in Section 7.1; that is, it attempts to find the best split ξ that minimizes the weighted cumulative impurity of data subsets created as the result of the split. Here we aim to mathematically characterize this statement.

Let $\mathbf{S}_{tr} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$ denote the training data. Assume the tree has already $m - 1$ node and we are looking for the best split at the m^{th} node. Let $\mathbf{S}_m \subset \mathbf{S}_{tr}$ denote the training data available for the split at node m . To identify the best split, we can conduct an exhaustive search over Ξ and identify the split that minimizes a measure of impurity. Recall that each $\xi \in \Xi$ corresponds to a question “ $x_j \leq \theta$?” where $\theta \in \Theta_j$ and depending on the yes/no answer to this question for each observation in \mathbf{S}_m , \mathbf{S}_m is then split into $\mathbf{S}_{m,\xi}^{\text{Yes}}$ and $\mathbf{S}_{m,\xi}^{\text{No}}$; that is to say,

$$\begin{aligned}\mathbf{S}_{m,\xi}^{\text{Yes}} &= \{(\mathbf{x}, y) \in \mathbf{S}_m \mid x_j \leq \theta, \text{ and } \theta \in \Theta_j\}, \\ \mathbf{S}_{m,\xi}^{\text{No}} &= \mathbf{S}_m - \mathbf{S}_{m,\xi}^{\text{Yes}}.\end{aligned}\quad (7.1)$$

When visualizing the structure of a decision tree, by convention the child node containing $\mathbf{S}_{m,\xi}^{\text{Yes}}$ is on the left branch of the parent node. Let $i(\mathbf{S})$ denote a specific measure of impurity of data \mathbf{S} presented to a node (also referred to as impurity of the node itself) and let $n_{\mathbf{S}}$ denote the number of data points in \mathbf{S} . A common heuristic used to designate a split ξ as the best split at node m is to maximize the difference between the impurity of \mathbf{S}_m and the weighted cumulative impurity of $\mathbf{S}_{m,\xi}^{\text{Yes}}$ and $\mathbf{S}_{m,\xi}^{\text{No}}$ (weighted cumulative impurity of the two child nodes). With the assumption that any split at least produces a “more pure” data or at least as impure as \mathbf{S}_m , we can assume we are maximizing the *drop* in impurity. This *impurity drop* for a split ξ at node m is denoted as $\Delta_{m,\xi}$ and is defined as

$$\Delta_{m,\xi} = i(\mathbf{S}_m) - \frac{n_{\mathbf{S}_{m,\xi}^{\text{Yes}}}}{n_{\mathbf{S}_m}} i(\mathbf{S}_{m,\xi}^{\text{Yes}}) - \frac{n_{\mathbf{S}_{m,\xi}^{\text{No}}}}{n_{\mathbf{S}_m}} i(\mathbf{S}_{m,\xi}^{\text{No}}).\quad (7.2)$$

Therefore, the best split at node m denoted as ξ_m^* is given by

$$\xi_m^* = \underset{\xi \in \Xi}{\operatorname{argmax}} \Delta_{m,\xi}.\quad (7.3)$$

In (7.2), multiplication factors $\frac{n_{\mathbf{S}_{m,\xi}^{\text{Yes}}}}{n_{\mathbf{S}_m}}$ and $\frac{n_{\mathbf{S}_{m,\xi}^{\text{No}}}}{n_{\mathbf{S}_m}}$ are used to deduct a weighted average of child nodes impurities from the parent impurity $i(\mathbf{S}_m)$. The main technical reason for having such a weighted average in (7.2) is that with a concave choice of impurity measure, $\Delta_{m,\xi}$ becomes nonnegative—and naturally it makes sense to refer to $\Delta_{m,\xi}$ as impurity drop. The proof of this argument is left as an exercise (Exercise 6). Nonetheless, a non-technical reason to describe the effect of the weighted average

used here with respect to an unweighted average is as follows: suppose that a split ξ causes $n_{S_{m,\xi}^{\text{Yes}}} << n_{S_{m,\xi}^{\text{No}}}$, say $n_{S_{m,\xi}^{\text{Yes}}} = 10$ and $n_{S_{m,\xi}^{\text{No}}} = 1000$, but the distribution of data in child nodes are such that $i(S_{m,\xi}^{\text{Yes}}) >> i(S_{m,\xi}^{\text{No}})$. Using unweighted average of child nodes impurities leads to $i(S_{m,\xi}^{\text{Yes}})$ dictating our decision in choosing the optimal ξ ; however, $i(S_{m,\xi}^{\text{Yes}})$ itself is not reliable as it has been estimated on very few observations (here 10) compared with many observations used to estimate $i(S_{m,\xi}^{\text{No}})$ (here 1000). The use of weighted average, however, can be thought as a way to introduce the reliability of these estimates in the average.

Once ξ_m^* is found, we split the data and continue the process until a *split-stopping* criterion is met. These criteria include:

- 1) a predetermined maximum depth. This has a type of global effect; that is to say, when we reach this preset maximum depth we stop growing the tree entirely;
- 2) a predetermined maximum number of leaf nodes. This also has a global effect;
- 3) a predetermined minimum number of samples per leaf. This has a type of local effect on growing the tree. That is to say, if a split causes the child nodes having less data points than the specified minimum number, that node will not split any further but splitting could still be happening in other nodes; and
- 4) a predetermined minimum impurity drop. This has also a local effect. In particular, if the impurity drop due to a split at a node is below this threshold, the node will not split.

As we see all these criteria depend on a predetermined parameter. Let us consider, for example, the minimum number of samples per leaf. Setting that to 1 (and not setting any other stopping criteria) allows the tree to fully grow and probably causes overfitting (i.e., good performance on training set but bad performance on test set). On the other hand, setting that to a large number could stop growing the tree immaturely in which case the tree may not learn discriminatory patterns in data. Such a situation is referred to as *underfitting* and leads to poor performance on both training and test sets. As a result, each of the aforementioned parameters that controls the tree complexity should generally be treated as a hyperparameter to be tuned.

7.2.3 Classification at Leaf Nodes

The classification in a decision tree occurs in the leaf nodes. In this regard, once the splitting process is stopped for all nodes in the tree, each leaf node is labeled as the majority class among all training observations in that node. Then to classify a test observation \mathbf{x} , we start from the root node and based on answers to the sequence of binary questions (splits) in the tree, we identify the leaf into which \mathbf{x} falls; the label of that leaf is then assigned to \mathbf{x} .

7.2.4 Impurity Measures

So far we did not discuss possible choices of $i(\mathbf{S})$. In defining an impurity measure, the first thing to notice is that in a multiclass classification with c classes $0, 1, \dots, c - 1$, the impurity of a sample \mathbf{S} that falls into a node is a function of the proportion of class k observations in \mathbf{S} —we denote this proportion by p_k , $k = 0, \dots, c - 1$. Then, the choice of $i(\mathbf{S})$ in classification could be any nonnegative function such that the function is:

1. maximum when data from all classes are equally mixed in a node; that is, when $p_0 = p_1 = \dots = p_{c-1} = \frac{1}{c}$, $i(\mathbf{S})$ is maximum;
2. minimum when the node contains data from one class only; that is, for c -tuples of the form $(p_0, p_1, \dots, p_{c-1})$, $i(\mathbf{S})$ is minimum for $(1, 0, \dots, 0)$, $(0, 1, 0, \dots, 0)$, \dots , $(0, \dots, 0, 1)$; and
3. a symmetric function of p_0, p_1, \dots, p_{c-1} ; that is, $i(\mathbf{S})$ does not depend on the order of p_0, p_1, \dots, p_{c-1} .

There are some common measures of impurity that satisfy these requirements. Assuming a multiclass classification with classes $0, 1, \dots, c - 1$, common choices of $i(\mathbf{S})$ include Gini, entropy, and misclassification error (if we classify all observations in \mathbf{S} to the majority class) given by:

$$\text{for Gini: } i(\mathbf{S}) = \sum_{k=0}^{c-1} p_k(1 - p_k), \quad (7.4)$$

$$\text{for entropy: } i(\mathbf{S}) = -\sum_{k=0}^{c-1} p_k \log(p_k), \quad (7.5)$$

$$\text{for error rate: } i(\mathbf{S}) = 1 - \max_{k=0, \dots, c-1} p_k, \quad (7.6)$$

where $p_k = \frac{n_k}{\sum_{k=0}^{c-1} n_k}$ in which n_k denotes the number of observations from class k in \mathbf{S} . Fig. 7.3 shows the concave nature of these impurity measures for binary classification. Although in (7.5) we use natural logarithm, sometimes entropy is written in terms of \log_2 . Nevertheless, changing the base does not change the best split because based on $\log_2 b = \frac{\log b}{\log 2}$, using \log_2 is similar to multiplying natural log by a constant $\frac{1}{\log 2}$ for all splits.

Example 7.1 In a binary classification problem, we have the training data shown in Table 7.1 for two available features where x_j denotes a feature variable, $j = 1, 2$. We wish to train a decision stump (i.e., a decision tree with only one split).

- a) identify candidate splits;
- b) for each identified candidate split, calculate the impurity drop using entropy metric (use natural log); and

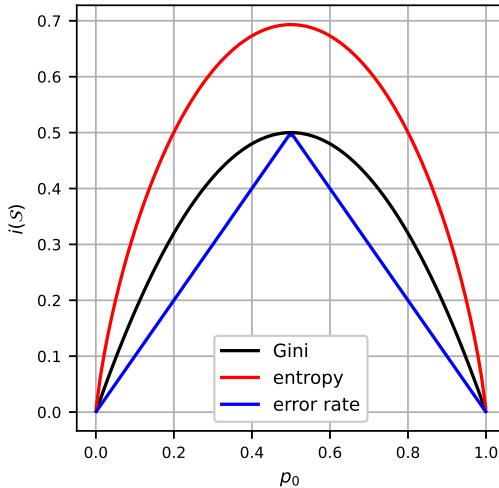


Fig. 7.3: Three common concave measures of impurity as a function of p_0 for binary classification.

- c) based on the results of part (b), determine the best split (or splits if there is a tie).

Table 7.1: Training data for Example 7.1.

class	0	0	0	0	1	1	1	1
x_1	1	1	3	3	1	1	1	3
x_2	2	4	4	0	2	0	4	2

a) x_1 takes two values 1 and 3. Taking their halfway point, the candidate split based on x_1 is $x_1 \leq 2$.

x_2 takes three values 0, 2, and 4. Taking their halfway points, the candidate splits based on x_2 are $x_2 \leq 1$ and $x_2 \leq 3$.

b) for each candidate split identified in part (a), we need to quantify $\Delta_{1,\xi}$.

From (7.2), we have $\Delta_{1,\xi} = i(\mathbf{S}_1) - \left(\frac{n_{\mathbf{S}_1^{\text{Yes}}}}{n_{\mathbf{S}_1}} i(\mathbf{S}_{1,\xi}^{\text{Yes}}) + \frac{n_{\mathbf{S}_1^{\text{No}}}}{n_{\mathbf{S}_1}} i(\mathbf{S}_{1,\xi}^{\text{No}}) \right)$.

Here \mathbf{S}_1 is the entire data in the first node (root node) because no split has occurred yet. We have

$$i(\mathbf{S}_1) = - \sum_{k=0}^1 p_k \log(p_k) = -\frac{4}{8} \log\left(\frac{4}{8}\right) - \frac{4}{8} \log\left(\frac{4}{8}\right) = \log(2) = 0.693 .$$

For $\xi = (1, 2)$ —recall that based on definition of ξ , this means the split $x_1 \leq 2$: based on this split, $S_{1,\xi}^{\text{Yes}}$ contains five observations (two from class 0 and three from class 1), and $S_{1,\xi}^{\text{No}}$ contains three observations (two from class 0 and one from class 1). Therefore,

$$\begin{aligned} i(S_{1,\xi}^{\text{Yes}}) &= -\frac{2}{5}\log\left(\frac{2}{5}\right) - \frac{3}{5}\log\left(\frac{3}{5}\right) = 0.673, \\ i(S_{1,\xi}^{\text{No}}) &= -\frac{2}{3}\log\left(\frac{2}{3}\right) - \frac{1}{3}\log\left(\frac{1}{3}\right) = 0.636. \end{aligned}$$

Therefore,

$$\Delta_{1,\xi=(1,2)} = 0.693 - \left(\frac{5}{8}0.673 + \frac{3}{8}0.636 \right) = 0.033.$$

For $\xi = (2, 1)$, which means $x_2 \leq 1$ split, $S_{1,\xi}^{\text{Yes}}$ contains two observations (one from class 0 and one from class 1), and $S_{1,\xi}^{\text{No}}$ contains six observations (three from class 0 and three from class 1). Therefore,

$$\begin{aligned} i(S_{1,\xi}^{\text{Yes}}) &= -\frac{1}{2}\log\left(\frac{1}{2}\right) - \frac{1}{2}\log\left(\frac{1}{2}\right) = 0.693, \\ i(S_{1,\xi}^{\text{No}}) &= -\frac{3}{6}\log\left(\frac{3}{6}\right) - \frac{3}{6}\log\left(\frac{3}{6}\right) = 0.693. \end{aligned}$$

Therefore,

$$\Delta_{1,\xi=(2,1)} = 0.693 - \left(\frac{2}{8}0.693 + \frac{6}{8}0.693 \right) = 0. \quad (7.7)$$

For $\xi = (2, 3)$: based on this split, $S_{1,\xi}^{\text{Yes}}$ contains five observations (two from class 0 and three from class 1), and $S_{1,\xi}^{\text{No}}$ contains three observations (two from class 0 and one from class 1). This is similar to $\xi = (1, 2)$. Therefore, $\Delta_{1,\xi=(1,2)} = 0.033$.

c) $\xi = (1, 2)$ and $\xi = (2, 3)$ are equally good in terms of entropy metric because the drop in impurity for both splits are the same and is larger than the other split (recall that we look for the split that maximizes the impurity drop). ■

7.2.5 Handling Weighted Samples

In training a predictive model, it is common to assume all observations are equally important. However, sometimes we may assume some observations are more important than others. This could be simulated by assigning some weights to observations and incorporating the weights in the training process. The sample importance/weight can simply be an assumption imposed by the practitioner or an assumption inherently made as part of some learning algorithm such as *boosting* that we will see in Chapter

8. A useful property of decision tree is its ability to easily handle weighted samples. To do so, we can replace the number of samples in a set used in (7.2) and any of the metrics presented in (7.4)-(7.6) by the sum of weights of samples belonging to that set. For example, rather than $n_{S_{m,\xi}^{\text{Yes}}}$ and n_{S_m} , we use sum of weights of samples in $S_{m,\xi}^{\text{Yes}}$ and S_m , respectively. Therefore, rather than using $p_k = \frac{n_k}{\sum_{k=0}^{c-1} n_k}$ that is used in (7.4)-(7.6), we use the ratio of the sum of weights of samples in class k to the sum of weights for all observations.

7.3 CART Development for Regression

7.3.1 Differences Between Classification and Regression

Our discussion in Section 7.2.1 and 7.2.2 did not depend on class labels of observations and, therefore, it was not specific to classification. In other words, the same discussion is applicable to regression. The only differences in developing CART for regression problems appear in: 1) the choices of impurity measures and; 2) how we estimate the target value at a leaf node.

7.3.2 Impurity Measures

To better understand the concept of impurity and its measures in regression, let us re-examine the impurity for classification. One way we can think about the utility of having a more pure set of child nodes obtained by each split is to have more certainty about the class labels for the training data falling into a node if that node is considered as a leaf node (in which case using the trivial majority vote classifier, we classify all training data in that node as the majority class). In regression, once we designate a node as a leaf node, a trivial estimator that we can use as the estimate of the target value for any test observation falling into a leaf node is the mean of target values of training data falling into that node. Therefore, a more “pure” node (to be more certain about the target) would be a node where the values of target for training data falling into that node are closer to their means. Choosing the mean as our trivial estimator of the target at a leaf node, we may then take various measures such as mean squared error (MSE) or mean absolute error (MAE) to measure how far on average target values in a leaf node are from their mean. Therefore, one can think of a node with a larger magnitude of these measures as a more impure node. However, when the mean is used as our target estimator, there is a subtle difference between using the MSE and MAE as our impurity measures. This is because the mean minimizes MSE function, and not the MAE. In particular, if we have a_1, a_2, \dots, a_n , then among all constant estimators a , the mean of a ’s minimizes the MSE; that is,

$$\bar{a} = \operatorname{argmin}_a \sum_{i=1}^n (a_i - a)^2, \quad (7.8)$$

where

$$\bar{a} = \frac{1}{n} \sum_{i=1}^n a_i. \quad (7.9)$$

Therefore, to use a homogenous pair of impurity and target estimator, it is common to use the mean and the MSE impurity together. However, when MAE is used, it is common to use the median (of target values of training data falling into a node) as our target estimator. This is because if $(.)^2$ in (7.8) is replaced with the absolute function $|.|$, then the optimal constant estimator is the median.

Therefore, the best split can be found using the following impurity functions in (7.2) and (7.3):

$$\text{for MSE: } i(\mathbf{S}) = \frac{1}{n_{\mathbf{S}}} \sum_{y \in \mathbf{S}} (y - \bar{y})^2, \quad (7.10)$$

$$\text{for MAE: } i(\mathbf{S}) = \frac{1}{n_{\mathbf{S}}} \sum_{y \in \mathbf{S}} |y - \tilde{y}|, \quad (7.11)$$

where

$$\bar{y} = \operatorname{mean}(y \in \mathbf{S}) = \frac{1}{n_{\mathbf{S}}} \sum_{y \in \mathbf{S}} y, \quad (7.12)$$

$$\tilde{y} = \operatorname{median}(y \in \mathbf{S}). \quad (7.13)$$

It is also common to refer to MSE and MAE as “squared error” and “absolute error”, respectively.

7.3.3 Regression at Leaf Nodes

When (7.10) and (7.11) are used as impurity measures, the mean and median of target values of training data in a leaf node are used, respectively, as the estimate of targets for any observation falling into that node.

Scikit-learn Implementation for both Classification and Regression: CART classifier and regressor are implemented in `DecisionTreeClassifier` class and `DecisionTreeRegressor` class from `sklearn.tree` module, respectively. `DecisionTreeClassifier` currently supports two distinct impurity criteria that can be changed by setting values of 'criterion' parameter to `gini` (default) and `entropy`—the possibility of using `log_loss`, which was added in scikit-learn version $\geq 1.1.0$, is equivalent to using `entropy`. `DecisionTreeRegressor` currently supports both the `squared_error` (default), `absolute_error`, and two other criteria.

For both `DecisionTreeClassifier` and `DecisionTreeRegressor`, the splitting stopping criteria discussed in Section 7.2.2, namely, a predetermined maximum depth, maximum number of leaf nodes, minimum number of samples per leaf, and minimum impurity drop could be set through `max_depth`, `max_leaf_nodes`, `min_samples_leaf`, and `min_impurity_decrease`, respectively. The default values of these parameters lead to fully grown trees, which not only could be very large depending on the data, but also as we discussed in Section 7.1, this could lead to overfitting. In practice these parameters should be treated as hyperparameters and in the absence of prior knowledge to set them, they could be tuned in the model selection stage (Chapter 9). In addition to the aforementioned hyperparameters that restrict the CART complexity, there is another hyperparameter in scikit-learn implementation of these classes, namely, `max_features` that determines the number of features that are examined to find the best split at each node. The default value of `max_features` examines all p features to find the best split but we can set that to some number, say $d < p$, so that at each node only d features are randomly chosen (without replacement) and examined; that is, the maximization in (7.3) to find the best split is taken over d randomly chosen features. Although computationally this option could be helpful in high-dimensional settings, the main utility of that will be cleared once we discuss ensemble decision trees in Chapter 8.

Example 7.2 In this example, we wish to examine the effect of `min_samples_leaf` on CART decision regions and its performance. We first load the preprocessed Iris dataset that was prepared in Section 4.6 and then use the first two features to classify Iris flowers.

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.tree import DecisionTreeClassifier as CART

arrays = np.load('data/iris_train_scaled.npz')
X_train = arrays['X']
y_train = arrays['y']
arrays = np.load('data/iris_test_scaled.npz')
X_test = arrays['X']
y_test = arrays['y']
```

```

X_train = X_train[:,[0,1]]
X_test = X_test[:,[0,1]]
print('X shape = {}'.format(X_train.shape) + '\ny shape = {}'.format(y_train.shape))
print('X shape = {}'.format(X_test.shape) + '\ny shape = {}'.format(y_test.shape))
color = ('aquamarine', 'bisque', 'lightgrey')
cmap = ListedColormap(color)
mins = X_train.min(axis=0) - 0.1
maxs = X_train.max(axis=0) + 0.1
x = np.arange(mins[0], maxs[0], 0.01)
y = np.arange(mins[1], maxs[1], 0.01)
X, Y = np.meshgrid(x, y)
coordinates = np.array([X.ravel(), Y.ravel()]).T
fig, axs = plt.subplots(2, 2, figsize=(6, 4), dpi = 200)
fig.tight_layout()
min_samples_leaf_val = [1, 2, 5, 10]
for ax, msl in zip(axs.ravel(), min_samples_leaf_val):
    cart = CART(min_samples_leaf=msl)
    cart.fit(X_train, y_train)
    Z = cart.predict(coordinates)
    Z = Z.reshape(X.shape)
    ax.tick_params(axis='both', labelsize=6)
    ax.set_title('CART Decision Regions: min_samples_leaf=' + str(msl), fontsize=7)
    ax.pcolormesh(X, Y, Z, cmap = cmap, shading='nearest')
    ax.contour(X, Y, Z, colors='black', linewidths=0.5)
    ax.plot(X_train[y_train==0, 0], X_train[y_train==0, 1], 'g.', markersize=4)
    ax.plot(X_train[y_train==1, 0], X_train[y_train==1, 1], 'r.', markersize=4)
    ax.plot(X_train[y_train==2, 0], X_train[y_train==2, 1], 'k.', markersize=4)
    ax.set_xlabel('sepal length (normalized)', fontsize=7)
    ax.set_ylabel('sepal width (normalized)', fontsize=7)
    print('The accuracy for min_samples_leaf={} on the training data is {:.3f}'.format(msl, cart.score(X_train, y_train)))
    print('The accuracy for min_samples_leaf={} on the test data is {:.3f}'.format(msl, cart.score(X_test, y_test)))
for ax in axs.ravel():
    ax.label_outer()

```

```

X shape = (120, 2)
y shape = (120,)
X shape = (30, 2)
y shape = (30,)
The accuracy for min_samples_leaf=1 on the training data is 0.933
The accuracy for min_samples_leaf=1 on the test data is 0.633

```

The accuracy for `min_samples_leaf=2` on the training data is 0.875
 The accuracy for `min_samples_leaf=2` on the test data is 0.667
 The accuracy for `min_samples_leaf=5` on the training data is 0.842
 The accuracy for `min_samples_leaf=5` on the test data is 0.667
 The accuracy for `min_samples_leaf=10` on the training data is 0.800
 The accuracy for `min_samples_leaf=10` on the test data is 0.567

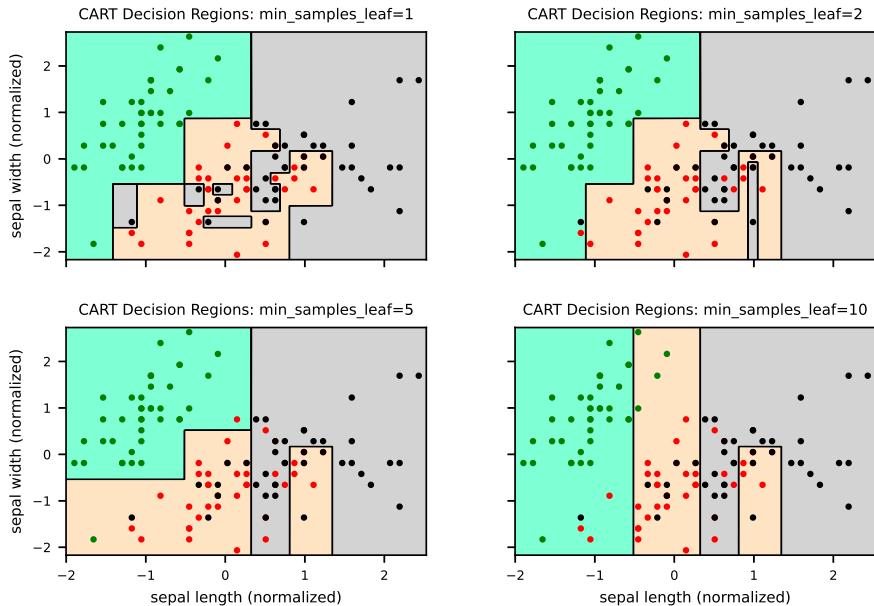


Fig. 7.4: The scatter plot of normalized Iris dataset for two features and decision regions for CART with minimum samples per leaf being 1, 2, 5, and 10. The green, red, and black points show points corresponding to setosa, versicolor, and virginica Iris flowers, respectively. The CART decision regions for each class are colored similarly.

As we can see, allowing the tree to fully grow (i.e., `min_samples_leaf=1`) leads to a high accuracy (93.3%) on the training data but a low accuracy on the test set (63.3%). This is an indicator of overfitting. At the same time, here the largest magnitude of `min_samples_leaf` that we take leads to the lowest accuracy on training and also a poor performance on the test as well. It turns out that in this example, `min_samples_leaf=2` and `min_samples_leaf=5` exhibit a similar performance on the test set (66.7%). ■

7.4 Interpretability of Decision Trees

A key advantage of decision tree is its “glass-box” structure in contrast to “black-box” structure of some other predictive models such as neural networks; that is to say, once a decision tree is developed, its classification mechanism is interpretable and transparent through the hierarchy of standardized questions. To have a better insight into the working mechanism of a trained decision tree, we can visualize the sequence of splits as part of the trained tree. A straightforward way is to use `plot_tree()` from `sklearn.tree` module. In the following code we visualize the last tree that was trained in Example 7.2; that is, the CART with `min_samples_leaf=10`. In this regard, we first load the Iris data again to access its attributes such as `feature_names` and `target_names`. We use `class_names=iris.target_names[0:3]` to show the majority classes at leaf nodes. The option `filled=True` in `plot_tree()` results in nodes being colored to the majority class within that node, where a darker color corresponds to a more pure node.

```
from sklearn import datasets
from sklearn.tree import plot_tree
plt.figure(figsize=(15,15), dpi=200)
iris = datasets.load_iris()
plot_tree(cart, feature_names = iris.feature_names[0:2], ↴
          class_names=iris.target_names[0:3], filled=True)
```

An interesting observation in Fig. 7.5 is that in several nodes, a split results in two leaf nodes with the same class. For example, consider the left most split “sepal width ≤ -0.068 ” that led to two child nodes with both having `setosa` label. The reason for such splits is because the cumulative impurities in child nodes measured by the weighted average of their impurities is less than the impurity of the parent node even though the majority class in both child nodes is `setosa`. For example, in Fig. 7.5 we observe that the Gini impurity of the node “sepal width ≤ -0.068 ” is 0.184, which is greater than $\frac{n_{S_m,\xi}^{\text{Yes}}}{n_{S_m}} i(S_{m,\xi}^{\text{Yes}}) + \frac{n_{S_m,\xi}^{\text{No}}}{n_{S_m}} i(S_{m,\xi}^{\text{No}}) = \frac{11}{40} \times 0.512 + \frac{29}{40} \times 0 = 0.140$. Such splits would make more sense if we think of having more pure nodes obtained by a split as a way to have more certainty about decisions made (see Section 7.2.4). Furthermore, the reason that these two child nodes are designated as leaf nodes and no further splits occurs is that one of them is pure (all data points are from one class), and for the other node any further splits would violate the `min_samples_leaf=10` condition.

Despite the classification transparency of decision trees, their decision regions could be quite sensitive with respect to changes in data. For example, Fig. 7.6 is obtained in a case where we randomly removed 5 observations out of 120 training observations that were used in Example 7.2 and re-train all the models. We observe that in some cases removing such a limited number of observations could lead to some quite different decision regions (compare the bottom row of Fig. 7.6 with Fig. 7.4).

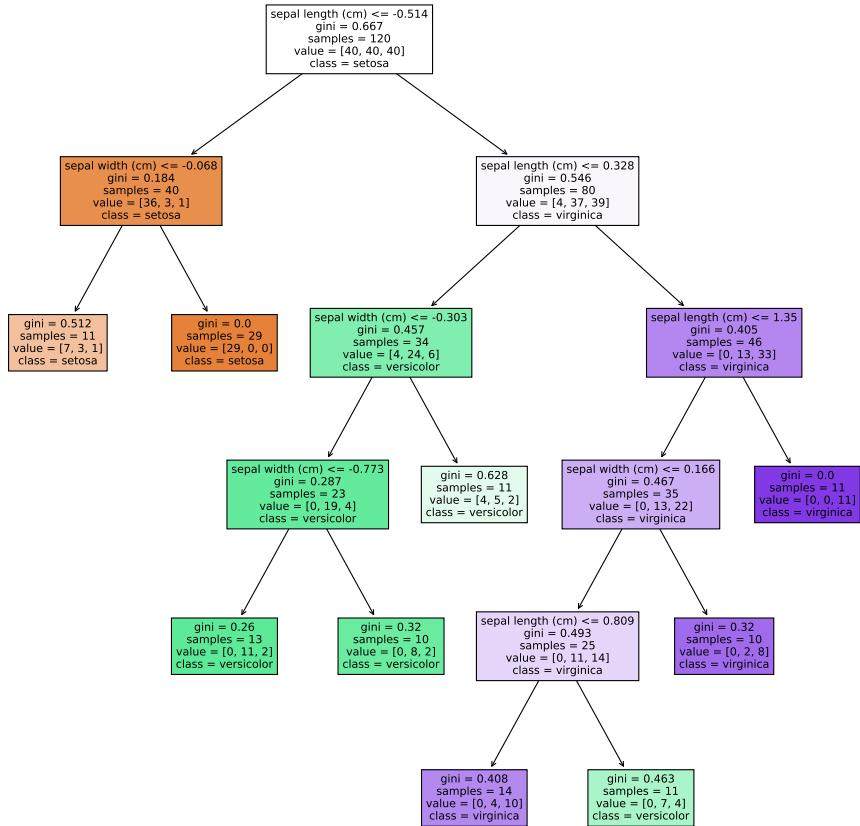


Fig. 7.5: The trained decision tree for Iris flower classification. The figure shows the “glass-box” structure of decision trees. Interpretability is a key advantage of decision trees. By convention, the child node satisfying condition in a node is on the left branch.

Exercises:

Exercise 1: Load the training and test Iris classification dataset that was prepared in Section 4.6. Select the first three features (i.e., columns with indices 0, 1, and 2). Write a program to train CART (for the hyperparameter `min_samples_leaf` being 1, 2, 5, and 10) and kNN (for the hyperparameter K being 1, 3, 5, 9) on this trivariate dataset and calculate and show their accuracies on the test dataset. Which combination of the classifier and hyperparameter shows the highest accuracy on the

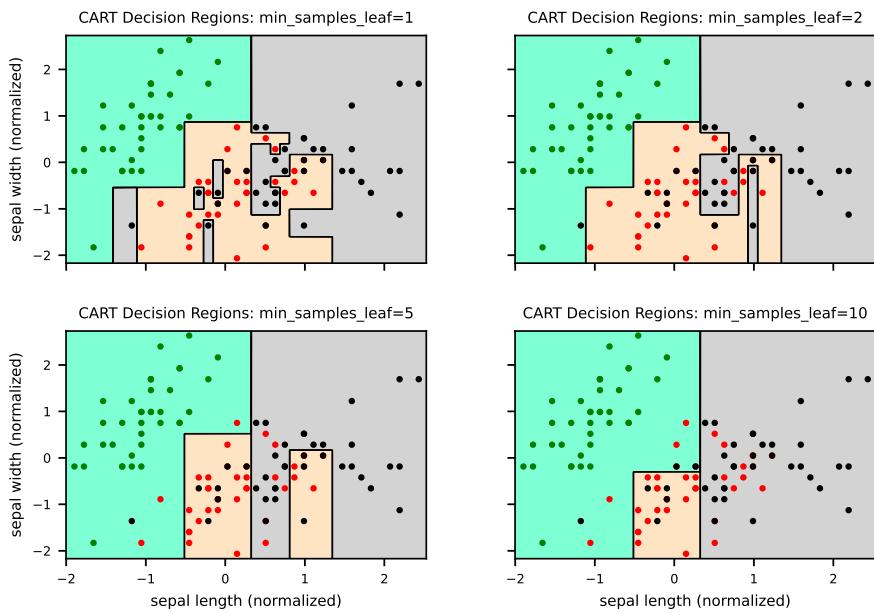


Fig. 7.6: The scatter plot and decision regions for a case that five observations were randomly removed from the data used in Example 7.2.

test dataset?

Exercise 2: Which of the following options is the scikit-learn class that implements the CART for regression?

- A) “sklearn.tree.DecisionTreeClassifier”
- B) “sklearn.tree.DecisionTreeRegressor”
- C) “sklearn.linear_model.DecisionTreeClassifier”
- D) “sklearn.linear_model.DecisionTreeRegressor”
- E) “sklearn.ensemble.DecisionTreeClassifier”
- F) “sklearn.ensemble.DecisionTreeRegressor”

Exercise 3: In a binary classification problems, we have the following sets of training data for two available features x_1 and x_2 . How many possible splits should be

examined to find the best split for training a decision stump using CART?

class	0	0	0	0	1	1	1
x_1	-1	0	3	2	3	4	10
x_2	7	3	2	0	1	-5	-3

Exercise 4: We would like to train a binary classifier using CART algorithm based on some 3-dimensional observations. Let $\mathbf{x} = [x_1, x_2, x_3]^T$ denote an observation with x_i being feature i , for $i = 1, 2, 3$. In training the CART classifier, suppose a parent node is split based on x_1 feature and results in two child nodes. Is that possible that the best split in one of these child nodes is again based on x_1 feature?

A) Yes

B) No

Exercise 5: In Example 7.2, we concluded that $\xi = (1, 2)$ and $\xi = (2, 3)$ are equally good to train a decision stump based on the entropy measure of impurity. Suppose we choose $\xi = (1, 2)$ as the split used in the root node and we would like to grow the tree further. Determine the best split on the left branch (satisfying $x_1 \leq 2$ condition). Assume $0 \log(\frac{0}{0}) = 0$.

⊕ **Exercise 6:**

As discussed in Section 7.2.4, impurity measure $i(\mathbf{S})$ is a function of $p_k = \frac{n_k}{\sum_{k=0}^{c-1} n_k}$, $k = 0, \dots, c - 1$, where n_k is the number of observations from class k in \mathbf{S} . Let $\phi(p_0, p_1, \dots, p_{c-1})$ denote this function. Suppose that this function is concave in the sense that for $0 \leq \alpha \leq 1$ and any $0 \leq x_k \leq 1$ and $0 \leq y_k \leq 1$, we have

$$\begin{aligned} \phi(\alpha x_0 + (1 - \alpha)y_0, \alpha x_1 + (1 - \alpha)y_1, \dots, \alpha x_{c-1} + (1 - \alpha)y_{c-1}) \\ \geq \alpha\phi(x_0, x_1, \dots, x_{c-1}) + (1 - \alpha)\phi(y_0, y_1, \dots, y_{c-1}). \end{aligned}$$

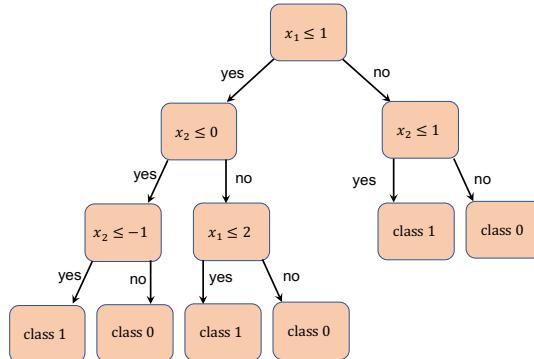
Show that using this impurity measure in the impurity drop defined as

$$\Delta_\xi = i(\mathbf{S}) - \frac{n_{\mathbf{S}_\xi^{\text{Yes}}}}{n_{\mathbf{S}}} i(\mathbf{S}_\xi^{\text{Yes}}) - \frac{n_{\mathbf{S}_\xi^{\text{No}}}}{n_{\mathbf{S}}} i(\mathbf{S}_\xi^{\text{No}}),$$

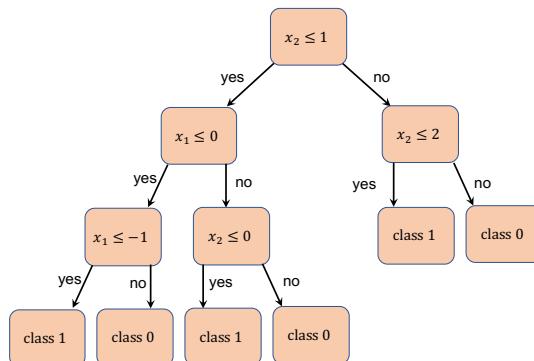
leads to $\Delta_\xi \geq 0$ for any split ξ .

Exercise 7: Consider the following two tree structures identified as Tree A and Tree B in which x_1 and x_2 are the names of two features:

Tree A:



Tree B:

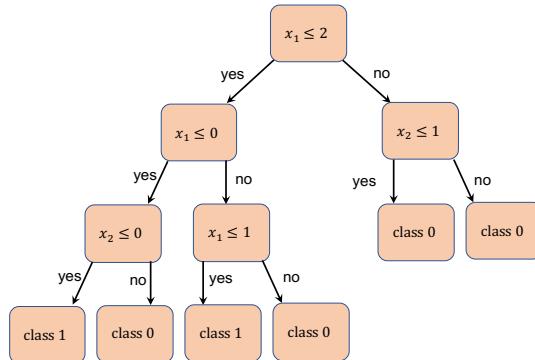


Can Tree A and Tree B be the outcome of applying the CART algorithm to some training data with the two numeric features x_1 and x_2 collected for a binary classification problem?

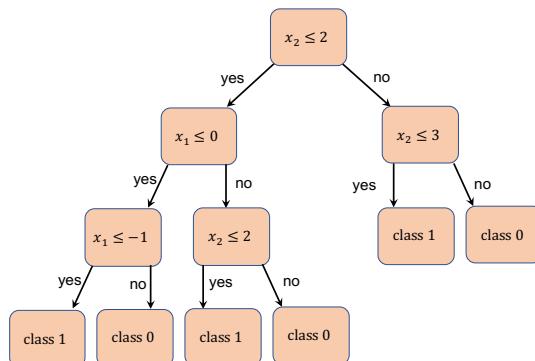
- A) Both trees A and B can be
- B) Tree A can not but Tree B can be
- C) Tree B can not but Tree A can be
- D) None of these two trees can be

Exercise 8: Explain whether each of the following is a legitimate decision tree obtained by the CART algorithm in some binary classification problems with two two numeric features x_1 and x_2 ?

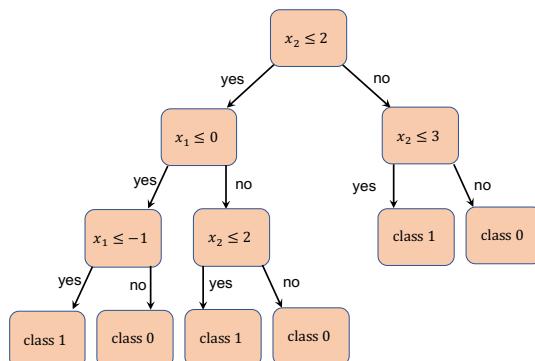
A)



B)



C)





Chapter 8

Ensemble Learning

The fundamental idea behind ensemble learning is to create a robust and accurate predictive model by combining predictions of multiple simpler models, which are referred to as *base models*. In this chapter, we describe various types of ensemble learning such as *stacking*, *bagging*, *random forests*, *pasting*, and *boosting*. Stacking is generally viewed as a method for combining the outcomes of several base models using another model that is trained based on the outputs of the base models along with the desired outcome. Bagging involves training and combining predictions of multiple base models that are trained individually on copies of the original training data that are created by a specific random resampling known as bootstrap. Random forest is a specific modification of bagging when applied to decision trees. Similar to bagging, a number of base models are trained on bootstrap samples and combined but to create each decision tree another randomization element is induced in the splitting strategy. This randomization often leads to improvement over bagged trees. In pasting, we randomly pick modest-size subsets of a large training data, train a predictive model on each, and aggregate the predictions. In boosting a sequence of weak models are trained and combined to make predictions. The fundamental idea behind boosting is that at each iteration in the sequence, a weak model (also known as weak learner) is trained based on the errors of the existing models and added to the sequence.

8.1 A General Perspective on the Efficacy of Ensemble Learning

In this section, we will first try to answer the following question: why would combining an ensemble of base models possibly help improve the prediction performance? Our attempt to answer this question is not surprising if we acknowledge that integrating random variables in different ways is indeed a common practice in estimation theory to improve the performance of estimation. Suppose we have a number of independent and identically distributed random variables Y_1, Y_2, \dots, Y_N with mean μ and variance σ^2 ; that is, $E[Y_i] = \mu$ and $\text{Var}[Y_i] = \sigma^2$, $i = 1, \dots, N$. Using Y_1

as estimator of μ has naturally a standard deviation of σ . However, if we use the sample mean $\frac{1}{N} \sum_{i=1}^N Y_i$ as the estimator of μ , we have $\text{Var}\left[\frac{1}{N} \sum_{i=1}^N Y_i\right] = \frac{N\sigma^2}{N^2} = \frac{\sigma^2}{N}$, which means it has a standard deviation of $\frac{\sigma}{\sqrt{N}} < \sigma$ for $N > 1$. In other words, combining Y_1, Y_2, \dots, Y_N by an averaging operator creates an estimator that has a higher “precision” (lower variance) than the utility of each Y_i in its “silo”. In what follows, we approach the question from the standpoint of regression.

8.1.1 Bias-Variance Decomposition

Consider a regression problem in which we wish to estimate the value of a target random variable Y for an input feature vector \mathbf{X} . To do so, we use a class of models; for example, we assume linear regression or CART. Even if we have infinite number of data points from the problem and we train a specific model, in general we would still have error in predicting Y because we approximated the underlying phenomenon connecting \mathbf{X} to Y by a specific mathematical model. Therefore, we can write

$$Y = f_o(\mathbf{X}) + \epsilon, \quad (8.1)$$

where $f_o(\mathbf{X})$ is the best (optimal) approximation of Y that we can achieve by using a class of models and ϵ denotes the unknown zero-mean error term (because any other mean can be considered as a bias term added to $f_o(\mathbf{X})$ itself). The term ϵ is known as *irreducible* error because no matter how well we do by using the mathematical model and features considered, we can not reduce this error (e.g., because we do not measure some underlying variables of the phenomenon or because our mathematical model is a too simplistic model of the phenomenon).

Nevertheless, we are always working with a finite training data and what we end up with instead is indeed an estimation of $f_o(\mathbf{X})$ denoted by $f(\mathbf{X})$ and we use it as an estimate of Y . How well do we predict? We can look into the mean square error (MSE). From the relationship presented in (8.1), we can write the MSE of $f(\mathbf{X})$ with respect to unknown Y as

$$\begin{aligned} E[Y - f(\mathbf{X})]^2 &= E[f_o(\mathbf{X}) + \epsilon - f(\mathbf{X})]^2 \\ &= E\left[f_o(\mathbf{X}) + \epsilon - f(\mathbf{X}) + E[f_o(\mathbf{X})] - E[f(\mathbf{X})] + E[f(\mathbf{X})] - E[f_o(\mathbf{X})]\right]^2 \\ &\stackrel{1}{=} E\left[f_o(\mathbf{X}) - f(\mathbf{X}) - (E[f_o(\mathbf{X})] - E[f(\mathbf{X})])\right]^2 + E\left[E[f_o(\mathbf{X})] - E[f(\mathbf{X})]\right]^2 + \text{Var}[\epsilon] \\ &\stackrel{2}{=} \text{Var}_d[f(\mathbf{X})] + \text{Bias}[f(\mathbf{X})]^2 + \text{Var}[\epsilon], \end{aligned} \quad (8.2)$$

where the expectation is over the joint distribution of \mathbf{X} , Y , and training data, $\stackrel{1}{=}$ follows from $E[\epsilon] = 0$, and to write $\stackrel{2}{=}$ we used the fact that

$$\text{Bias}[f(\mathbf{X})]^2 \triangleq \left[\mathbb{E}[Y - f(\mathbf{X})] \right]^2 = \left[\mathbb{E}[f_0(\mathbf{X}) - f(\mathbf{X})] \right]^2 = \mathbb{E} \left[\mathbb{E}[f_0(\mathbf{X})] - \mathbb{E}[f(\mathbf{X})] \right]^2, \quad (8.3)$$

and where the *variance of deviation* $\text{Var}_d[f(\mathbf{X})]$ is defined as

$$\text{Var}_d[f(\mathbf{X})] = \text{Var}[f_0(\mathbf{X}) - f(\mathbf{X})]. \quad (8.4)$$

As observed in (8.2), we could write the MSE of estimation in terms of the variance of deviation and a squared bias term. This is known as *bias-variance decomposition* of expected error (MSE). Generally, the more complex the model, the larger the variance of deviation. What does this imply? Suppose we have dataset A collected from a problem and use a complex learning algorithm with many degrees of freedom to train a model, namely Model A. Then we collect another dataset from the same problem, namely, dataset B, and use the same learning algorithm to train another model, namely Model B. In this scenario, Model A and Model B could be very different because the learning algorithm has many degrees of freedom. As an example of such a scenario consider CART when left to fully grow. As we observed in Chapter 6, even a slight change in training data could lead to different tree classifiers. On the other hand, a simpler model (less degrees of freedom) does not change as much (by definition has less flexibility) so the variance of deviation could naturally be lower.

However, insofar as bias is concerned, it is more likely that a complex learning algorithm has lower bias than a simpler learning algorithm. To understand this, we have to note that whether we use a simple or complex learning algorithm, we try to estimate the best possible approximation of Y (i.e., to estimate $f_0(\mathbf{X})$) within each class. When a complex learning algorithm is used, by definition the class has more room to possess an $f_0(\mathbf{X})$ that is closer to Y (see the red crosses in Fig. 8.1). However, because we do not know $f_0(\mathbf{X})$, essentially we end up with its estimate $f(\mathbf{X})$, which can change from data to data. The highlighted red regions in Fig. 8.1 show the set of possible values of $f(\mathbf{X})$ within each class. As can be seen in this figure, although the highlighted red region for the complex class is larger, the points within this region are on average closer to Y than the average distance for the set of points within the highlighted region of the simpler class. At the same time, the area of these regions is an indicator of the variance that we explained before (the larger the area, the greater the variance of deviation).

Ensemble methods generally benefit from reducing the variance of deviation discussed above (for example, bagging, random forests, and stacking) although some methods such as boosting benefit from both reduction in variance and bias. To better understand these statements (at least in terms of variance) a more details discussion is required as presented next.

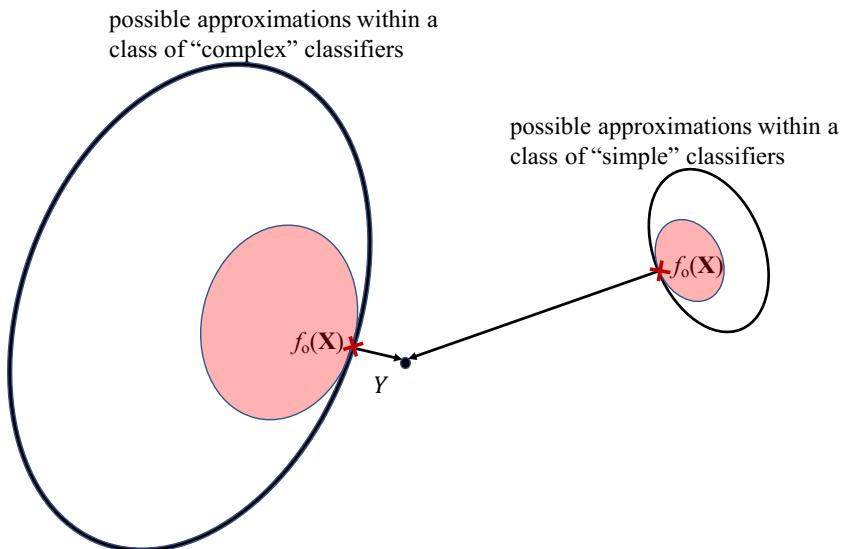


Fig. 8.1: The larger (smaller) oval shows the set of possible approximations within a class of complex (simple) classifiers. The best approximation in the more “complex” class is closer to Y . The red region within each oval shows the set of possible estimates of $f_0(\mathbf{X})$ within that class. The average distance between Y and the points within the red region of the complex class is less than the average distance between Y and the set of points within the highlighted region of the simple class. This shows a lower bias of the more complex classifier.

8.1.2 \oplus How Would Ensemble Learning Possibly Help?

As we observed in the previous section, in decomposition of the mean square error of prediction, a variance term $\text{Var}_d[f(\mathbf{X})]$ and a bias term $\text{Bias}[f(\mathbf{X})]$ will appear. To show the utility of ensemble learning, we focus on the possibility of reducing $\text{Var}_d[f(\mathbf{X})]$. We have

$$\text{Var}_d[f(\mathbf{X})] = \mathbb{E} \left[f_0(\mathbf{X}) - f(\mathbf{X}) - (\mathbb{E}[f_0(\mathbf{X})] - \mathbb{E}[f(\mathbf{X})]) \right]^2. \quad (8.5)$$

The difference between $f_0(\mathbf{X})$ and $f(\mathbf{X})$ is known as *reducible error* because this could be reduced, for example, by collecting more or better datasets to be used in estimating $f_0(\mathbf{X})$. Therefore, we write

$$f(\mathbf{X}) = f_0(\mathbf{X}) + e, \quad (8.6)$$

where e shows the reducible error. Replacing (8.6) in (8.5) yields

$$\text{Var}_d[f(\mathbf{X})] = \mathbb{E} \left[e - \mathbb{E}[e] \right]^2 = \mathbb{E}[e^2] - (\mathbb{E}[e])^2. \quad (8.7)$$

Let us now consider an ensemble of M base models that are trained using a specific learning algorithm but perhaps with different (structural or algorithmic) hyperparameters. The necessity of having the same learning algorithm in the following arguments is the use of $f_0(\mathbf{X})$ as the unique optimal approximation of Y that is achievable within the class of models considered. Nevertheless, with a slightly more work these arguments could be applied to different learning algorithms. We write

$$f_i(\mathbf{X}) = f_0(\mathbf{X}) + e_i, \quad i = 1, \dots, M. \quad (8.8)$$

From (8.7) we have

$$\text{Var}_d[f_i(\mathbf{X})] = \mathbb{E}[e_i^2] - (\mathbb{E}[e_i])^2, \quad i = 1, \dots, M. \quad (8.9)$$

Each of these M base models has a mean square error with a deviation variance term as in (8.9). Therefore, the *average MSE* of all these M models has a term that is the average of variance of deviation for each base model (in the following we denote that by $\text{Var}_d^{\text{avg}}$):

$$\text{Var}_d^{\text{avg}} \triangleq \frac{1}{M} \sum_{i=1}^M \text{Var}_d[f_i(\mathbf{X})] = \frac{1}{M} \sum_{i=1}^M \mathbb{E}[e_i^2] - \frac{1}{M} \sum_{i=1}^M (\mathbb{E}[e_i])^2. \quad (8.10)$$

Now let us consider an ensemble regression model that as the prediction takes the average of $f_i(\mathbf{X})$'s; after all, each $f_i(\mathbf{X})$ is the prediction made by each model and it is reasonable (and common) to take their average in an ensemble model. Therefore, from (8.8) we can write

$$\frac{1}{M} \sum_{i=1}^M f_i(\mathbf{X}) = f_0(\mathbf{X}) + \bar{e}, \quad i = 1, \dots, M, \quad (8.11)$$

where

$$\bar{e} = \frac{1}{M} \sum_{i=1}^M e_i. \quad (8.12)$$

The MSE of prediction obtained by this ensemble model has a variance of deviation term similar to (8.7), which is obtained as (in the following we denote that by $\text{Var}_d^{\text{ens}}$):

$$\text{Var}_d^{\text{ens}} = \mathbb{E}[\bar{e}^2] - (\mathbb{E}[\bar{e}])^2 = \mathbb{E} \left[\left(\frac{1}{M} \sum_{i=1}^M e_i \right)^2 \right] - \left(\mathbb{E} \left[\frac{1}{M} \sum_{i=1}^M e_i \right] \right)^2. \quad (8.13)$$

To better compare the average MSE of prediction by base models and the MSE of prediction achieved by the ensemble model (i.e., average of their predictions) to the extent that are affected by $\text{Var}_d^{\text{avg}}$ and $\text{Var}_d^{\text{ens}}$, respectively, we make an assumption that $E[e_i] = 0$, $i = 1, \dots, M$. In that case, (8.10) and (8.13) reduce to

$$\begin{aligned}\text{Var}_d^{\text{avg}} &= \frac{1}{M} \sum_{i=1}^M E[e_i^2], \\ \text{Var}_d^{\text{ens}} &= E\left[\left(\frac{1}{M} \sum_{i=1}^M e_i\right)^2\right] = \frac{1}{M} E\left[\frac{1}{M} \left(\sum_{i=1}^M e_i\right)^2\right].\end{aligned}\quad (8.14)$$

It is straightforward to show that (Exercise 5):

$$\frac{1}{M} E\left[\left(\sum_{i=1}^M e_i\right)^2\right] \leq \sum_{i=1}^M E[e_i^2], \quad (8.15)$$

which means

$$\text{Var}_d^{\text{ens}} \leq \text{Var}_d^{\text{avg}}. \quad (8.16)$$

The inequality presented in (8.16) is a critical observation behind the utility of many ensemble techniques. That is taking the average of predictions by base models leads to a smaller variance of deviation than the average of their deviation variances. This, in turn, generally leads to a lower prediction error (MSE of prediction) due to bias-variance decomposition as stated in (8.2). We may also interpret this observation in another way. The performance improvement achieved by ensemble is generally attributed to reduction in variance of deviations (some methods such as boosting though they attempt to reduce both the variance and bias). Another interesting observation is that, if we assume the error terms in (8.15) are uncorrelated, $E[e_i e_j] = 0$, $i \neq j$, then from (8.14) we have

$$\text{Var}_d^{\text{ens}} = \frac{1}{M} \text{Var}_d^{\text{avg}}. \quad (8.17)$$

This is an important observation that shows a substantial reduction in variance of deviation (i.e., a factor of $\frac{1}{M}$) obtained by having uncorrelated error terms. In practice, this is not usually the case though because typically we have one dataset and we take some overlapping subsets of that to develop our predictions $f_i(\mathbf{X})$, $i = 1, \dots, M$.

8.2 Stacking

Stacked ensemble learning (also known as a stacked generalization) is generally viewed as a method for combining the outcomes of N base models (also known as level-0 generalizers) using another model (level-1 generalizer) that is trained based on the outputs of the base models along with the desired outcomes (Wolpert, 1992) (see Fig. 8.2). For example, we can train a logistic regression as level-1 generalizer to combine the outputs of a number of different trained neural networks that are used as level-0 generalizer. Nevertheless, stacked generalization is a general and diverse learning strategy that includes various other common techniques. For example, consider the common approach of the “winner-takes-all” strategy. In this strategy, out of several generalizers (classifiers or regressors), a single generalizer with the best estimated performance (e.g., the highest accuracy on an independent set known as validation set) is selected for performing prediction on unseen data. This practice itself is a stacked generalization that in Wolpert’s words (Wolpert, 1992), has “an extraordinary dumb level-1 generalizer” that selects the level-0 generalizer with the lowest estimated error. That being said, the level-1 generalizers used in stacked ensemble learning are usually more intelligent than this. As a result, due to the versatility of stacked generalization, we do not discuss it any further in this chapter.

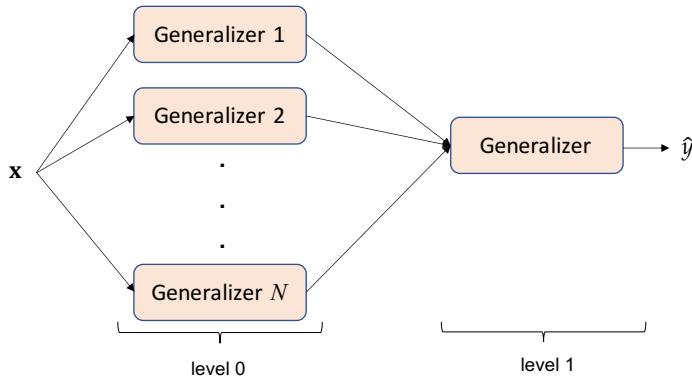


Fig. 8.2: A schematic representation of stacked generalization. \mathbf{x} and \hat{y} denote a given feature vector and the estimated target, respectively.

8.3 Bagging

Recall from Section 7.4 in which we mentioned that the decision regions of decision trees could be quite sensitive with respect to changes in data. One question is whether we can systematically improve the robustness of some classifiers based on decision trees that are more robust with respect to variation in data. Let us break up this question into two:

1. What is meant by variation in data and how can we simulate that?
2. How can we possibly improve the robustness?

To answer the first question, suppose we have a training data with a sample size n . Each observation in this training set is a realization of a probability distribution governing the problem. The probability distribution itself is the result of cumulative effect of many (possibly infinite) hidden factors that could relate to technical variabilities in experiments or stochastic nature of various phenomena. Had we had detailed information about probabilistic characteristics of all these underlying factors, we could have constructed the probability distribution of the problem and used it to generate more datasets from the underlying phenomenon. However, in practice, except for the observations at hand, we have generally no further information about the probability distribution of the problem. Therefore, it makes sense if we somehow use the data at hand to generate more data from the underlying phenomenon (i.e., to simulate variation in data).

But how can we generate several new legitimate datasets from the single original data at hand? One way is to assume some observations are repeated; after all, if an observation happens once, it may happen again (and again). Therefore, to create each new dataset, we randomly draw with replacement n observations from the original data at hand. However, keeping the same sample size n implies some observations from the original training dataset do not appear in these new datasets. This way we have simulated variation in data by just assuming repetition and deletion operation on the observations from the original dataset. These newly generated datasets (i.e., from sampling the original dataset with replacement while keeping the same sample size) are known as *bootstrap samples*.

Now we turn to the second question. As each of these bootstrap training sets is a legitimate dataset from the same problem, we train a decision tree on each bootstrap sample. Then for a given feature vector \mathbf{x} , each tree produces a prediction. We can then *aggregate* these predictions to create a final prediction. The aggregation operation could be the majority vote for classification and the averaging for regression. We note that the final prediction is the outcome of applying decision tree algorithm on many training datasets, and not just one dataset. Therefore, we capture more variation of data in the aggregated model. Put in other words, the aggregated model is possibly less affected by variation in data as compared with each of its constituent tree models (for a more concrete analysis on why this may improve the prediction performance refer to Section 8.1.2).

The aforementioned systematic way (i.e., creating bootstrap samples and aggregating) is indeed the mechanism of bagging (short for **bootstrap aggregating**)

proposed by Breiman in (Breiman, 1996). Let $\mathbf{S}_{tr,j}^*$, $j = 1, \dots, B$ shows the set of B , bootstrap samples that are generated by sampling with replacement from \mathbf{S}_{tr} . Then the “bagged” classifier and regressor denoted by $\psi_{\text{bag}}(\mathbf{x})$ and $f_{\text{bag}}(\mathbf{x})$, respectively, are defined as

$$\text{Classification: } \psi_{\text{bag}}(\mathbf{x}) = \underset{i \in \{0, \dots, c-1\}}{\operatorname{argmax}} \sum_{j=1}^B I_{\{\psi_j^*(\mathbf{x})=i\}} \quad (8.18)$$

$$\text{Regression: } f_{\text{bag}}(\mathbf{x}) = \frac{1}{B} \sum_{j=1}^B f_j^*(\mathbf{x}) \quad (8.19)$$

where ψ_j^* and f_j^* denote the classifier and regressor trained on bootstrap sample $\mathbf{S}_{tr,j}^*$, respectively, the set $\{0, \dots, c-1\}$ represents the class labels, and I_A is the indicator of event A . The number of bootstrap samples B is typically set between 100 to 200.

Scikit-learn implementation: Bagging classifier and regressor are implemented by `BaggingClassifier` and `BaggingRegressor` classes from `sklearn.ensemble` module, respectively. Using `base_estimator` parameter, both `BaggingClassifier` and `BaggingRegressor` can take as input a user-specified base model along with its hyperparameters. Another important hyperparameter in these constructors is `n_estimators` (by default it is 10). Using this parameter, we can control the number of base models, which is the same as the number of bootstrap samples (i.e., B in the aforementioned notations). The scikit-learn implementation even allows us to set the size of bootstrap samples to a size different from the original sample size. This could be done by setting the `max_samples` parameter in these constructors. The default value of this parameter (which is 1.0) along with the default value of `bootstrap` parameter (which is `True`) leads to bagging as described before. In these classes, there is another important parameter, namely, `max_features`, which by default is set to 1.0. Setting the value of this parameter to an integer less than p (the number of features in the data) or a float less than 1.0 trains each estimator on a randomly picked subset of features. That being said, when we train each base estimator on a different subset of features and samples, the method is referred to as *random patches* ensemble. In this regard, we can even set `bootstrap_features` to `True` so that the random selection is performed by replacement.

8.4 Random Forest

Random forest (Breiman, 2001) is similar to bagged decision trees (obtained by using decision trees as base models used in bagging) except for a small change with respect to regular training of decision trees that could generally lead to a remarkable performance improvement with respect to bagged trees. Recall Section 7.2.2 where we discussed the splitting strategy at each node of a decision tree: for each split,

we examine *all* features and pick the split that maximizes the impurity drop. This process is then continued until one of split-stopping criteria is met. For random forests, however, for each split in training a decision tree on a bootstrap sample, $d \leq p$ features are randomly picked as a candidate feature set and the best split is the one that maximizes the impurity drop over this candidate feature set. Although d could be any integer less or equal to p , it is typically set as \sqrt{p} or $\log_2 p$. This simple change in the examined features is the major factor to decorrelate trees in random forest as compared with bagged trees (recall Section 8.1.2 where we showed having uncorrelated error terms in the ensemble leads to substantial reduction in variance of deviation). We can summarize the random forest algorithm as follows:

Algorithm Random Forest

1. Create bootstrap samples: $S_{tr,j}^*$, $j = 1, \dots, B$
2. For each bootstrap sample, train a decision tree classifier or regressor denoted by ψ_j^* and f_j^* , respectively, where:
 - 2.1. To find the best split at each node, randomly pick $d \leq p$ features and maximize the impurity drop over these features
 - 2.2. Grow the tree until one of the split-stopping criteria is met (see Section 7.2.2 where four criteria were listed)
3. Perform the prediction for \mathbf{x} as:

$$\text{Classification: } \psi_{RF}(\mathbf{x}) = \underset{i \in \{0, \dots, c-1\}}{\operatorname{argmax}} \sum_{j=1}^B I_{\{\psi_j^*(\mathbf{x})=i\}} \quad (8.20)$$

$$\text{Regression: } f_{RF}(\mathbf{x}) = \frac{1}{B} \sum_{j=1}^B f_j^*(\mathbf{x}) \quad (8.21)$$

Scikit-learn implementation: The random forest classifier and regressor are implemented in `RandomForestClassifier` and `RandomForestRegressor` classes from `sklearn.ensemble` module, respectively. Many hyperparameters in these algorithms are similar to scikit-learn implementation of CART with the same functionality but applied to all underlying base trees. However, here perhaps the use of a parameter such as `max_features` makes more sense because this is a major factor in deriving the improved performance of random forests with respect to bagged trees in many datasets. Furthermore, we need to point out a difference between the classification mechanism of `RandomForestClassifier` with the majority vote that we saw above. In scikit-learn, rather than the majority vote, which is also known as *hard*

voting, a *soft voting* mechanism is used to classify an instance. In this regard, for a given observation \mathbf{x} , first the class probability of that observation belonging to each class is calculated for each base tree—this probability is estimated as the proportion of samples from a class in the leaf that contains \mathbf{x} . Then \mathbf{x} is assigned to the class with the highest class probability averaged over all base trees in the forest.



As stated in Section 8.3, using `max_features` parameter of `BaggingClassifier` and `BaggingRegressor`, we can create random patches ensemble. The main difference between random forest and random patches ensemble when decision trees are used as base estimators is that in random forest the process of feature subset selection is performed for each split, while in random patches the process of feature subset selection is performed for each estimator.

8.5 Pasting

The underlying idea in pasting (Breiman, 1999) is that rather than training one classifier on the entire dataset, which sometimes can be computationally challenging for large sample size, we randomly (without replacement) pick a number of modest-size subsets of training data (say “ K ” subsets), construct a predictive model for each subset, and aggregate the results. The combination rule is similar to what we observed in Section 8.3 for bagging. Assuming $\psi_j(\mathbf{x})$ and $f_j(\mathbf{x})$ denote the classifier and regressor trained for the j^{th} training subset, $j = 1, \dots, K$, the combination rule is:

$$\text{Classification: } \psi_{\text{pas}}(\mathbf{x}) = \underset{i \in \{0, \dots, c-1\}}{\operatorname{argmax}} \sum_{j=1}^K I_{\{\psi_j(\mathbf{x})=i\}} \quad (8.22)$$

$$\text{Regression: } f_{\text{pas}}(\mathbf{x}) = \frac{1}{K} \sum_{j=1}^K f_j(\mathbf{x}) \quad (8.23)$$

Scikit-learn implementation: Pasting is obtained using the same classes that we saw earlier for bagging (`BaggingClassifier` and `BaggingRegressor`) except that we need to set the `bootstrap` parameter to `False` and `max_samples` to an integer less than the sample size or a float less than 1.0. This causes the sample subset being drawn randomly without replacement.

8.6 Boosting

Boosting is an ensemble technique in which a sequence of weak base models (also known as weak learners) are trained and combined to make predictions. Here we focus on three important classes of boosting: *AdaBoost*, *gradient boosting regression tree*, and *XGBoost*.

8.6.1 AdaBoost

In *AdaBoost* (short for **Adaptive Boosting**), a sequence of weak learners are trained on a series of iteratively weighted training data ([Freund and Schapire, 1997](#)). The weight of each observation at each iteration depends on the performance of the trained model at the previous iteration. After a certain number of iterations, these models are then combined to make a prediction. The good empirical performance of AdaBoost on many datasets has made some pioneers such as Leo Breiman to refer to this ensemble technique as “as the most accurate general purpose classification algorithm available” (see ([Breiman, 2004](#))). Boosting was originally developed for classification and then extended to regression. For the classification, it was shown that even if base classifiers are weak classifiers, which are efficient to train, boosting can produce powerful classifiers. A weak classifier is a model with a performance slightly better than random guessing (for example, its accuracy on a balanced training data in binary classification problems is slightly better than 0.5). An example of a weak learner is *decision stump*, which is a decision tree with only one split. In some extension of AdaBoost to regression, for example, AdaBoost.R proposed in ([Drucker, 1997](#)), the “accuracy better than 0.5” restriction was carried over and changed to the “average loss less than 0.5”.

Here we discuss AdaBoost.SAMME (short for **S**tage**W**ise **A**dditive **M**odeling using a **M**ulti-class **E**xponential **L**oss **F**unction) for classification ([Zhu et al., 2009](#)) and AdaBoost.R2 proposed in ([Drucker, 1997](#)) for regression, which is a modification of AdaBoost.R proposed earlier in ([Freund and Schapire, 1997](#)). Suppose a sample of size n collected from c classes and a learning algorithm that can handle multiclass classification and supports weighted samples (e.g., decision stumps [see Section 7.2.5]) are available. AdaBoost.SAMME is implemented as in the following algorithm.

Algorithm AdaBoost.SAMME Algorithm for Classification

1. Assign a weight w_j to each observation $j = 1, 2, \dots, n$, and initialize them to $w_j = \frac{1}{n}$.

2. For m from 1 to M :

2.1. Apply the learning algorithm to weighted training sample to train classifier $\psi_m(\mathbf{x})$.

2.2. Compute the error rate of the classifier $\psi_m(\mathbf{x})$ on the weighted training sample:

$$\hat{\varepsilon}_m = \frac{\sum_{j=1}^n w_j I_{\{y_j \neq \psi_m(\mathbf{x}_j)\}}}{\sum_{j=1}^n w_j}. \quad (8.24)$$

2.3. Compute confidence coefficients

$$\alpha_m = \eta \left(\log\left(\frac{1 - \hat{\varepsilon}_m}{\hat{\varepsilon}_m}\right) + \log(c - 1) \right), \quad (8.25)$$

where $\eta > 0$ is a tuning hyperparameter known as *learning rate*.

2.4. For $j = 1, 2, \dots, n$, update the weights

$$w_j \leftarrow w_j \times \exp(\alpha_m I_{\{y_j \neq \psi_m(\mathbf{x}_j)\}}). \quad (8.26)$$

3. Construct the boosted classifier:

$$\psi_{\text{boost}}(\mathbf{x}) = \underset{i \in \{0, \dots, c-1\}}{\operatorname{argmax}} \sum_{m=1}^M \alpha_m I_{\{\psi_m(\mathbf{x})=i\}}. \quad (8.27)$$

Few notes about AdaBoost.SAMME:

1. When $c = 2$, AdaBoost.SAMME reduces to AdaBoost in its original form proposed in ([Freund and Schapire, 1997](#)).
2. An underlying idea of AdaBoost is to make the classifier trained at iteration $m+1$ focuses more on observations that are misclassified by the classifier trained at iteration m . To do so, in step 2.4, for an observation j that is misclassified by $\psi_m(\mathbf{x})$, we scale w_j by a factor of $\exp(\alpha_m)$. However, to increase the influence of such misclassified observation in training $\psi_{m+1}(\mathbf{x})$, we need to scale *up* w_j , which means $\alpha_m > 0$. However, to have $\alpha_m > 0$, we only need $1 - \hat{\varepsilon}_m > \frac{1}{c}$, which means it suffices if the classifier $\psi_m(\mathbf{x})$ performs slightly better than ran-

dom guessing. In binary classification, this means $\hat{\varepsilon}_m < 0.5$. This is also used to introduce an *early stopping* criterion in AdaBoost; that is to say, the process of training M base classifier is terminated as soon as $\hat{\varepsilon}_m > 1 - \frac{1}{c}$. Another criterion for early stopping is when we achieve perfect classification in the sense of having $\hat{\varepsilon}_m = 0$.

3. To make the final prediction by $\psi_{\text{boost}}(\mathbf{x})$, those classifiers $\psi_{m+1}(\mathbf{x})$, $m = 1, 2, \dots, M$, that had a lower error rate $\hat{\varepsilon}_m$ during the training have a greater α_m , which means they have a higher contribution. In other words, α_m is an indicator of confidence in classifier $\psi_m(\mathbf{x})$.
4. If at any point during training AdaBoost, the error on the weighted training sample becomes 0 (i.e., $\hat{\varepsilon}_m = 0$), we can stop training. This is because when $\hat{\varepsilon}_m = 0$ the weights will not be updated and the base classifier for the subsequent iterations remain the same. At the same time, because $\alpha_k = 0$, $k \geq m$, they are not added to $\psi_{\text{boost}}(\mathbf{x})$, which means the AdaBoost classifier remains the same.

The AdaBoost.R2 ([Drucker, 1997](#)) algorithm for regression is presented on the next page.

Scikit-learn implementation: AdaBoost.SAMME can be implemented using `AdaBoostClassifier` from `sklearn.ensemble` module by setting the `algoirthm` parameter to SAMME. AdaBoost.R2 is implemented by `AdaBoostRegressor` from the same module. Using `base_estimator` parameter, both `AdaBoostClassifier` and `AdaBoostRegressor` can take as input a user-specified base model along with its hyperparameters. If this parameter is not specified, by default a decision stump (a `DecisionTreeClassifier` with `max_depth` of 1) will be used. The `n_estimators=50` parameter controls the maximum number of base models (M in the aforementioned formulations of these algorithms). In case of perfect fit, the training terminates before reaching M . Another hyperparameter is the learning rate η (identified as `learning_rate`). The default value of this parameter, which is 1.0, leads to the original representation of AdaBoost.SAMME and AdaBoost.R2 presented in ([Zhu et al., 2009](#)) and ([Drucker, 1997](#)), respectively.

8.6.2 \oplus Gradient Boosting

In ([Friedman et al., 2000](#)), it was shown that one can write AdaBoost as a stagewise additive modeling problem with a specific choice of loss function. This observation along with the connection between stagewise additive modeling with steepest descent minimization in function space allowed a new family of boosting strategies known as *gradient boosting* ([Friedman, 2001](#)). Here we first present an intuitive development of the gradient boosting technique, then present its formal algorithmic description as originally presented in ([Friedman, 2001, 2002](#)), and finally its most common implementation known as *gradient boosted regression tree (GBRT)*.

Algorithm \oplus AdaBoost.R2 Algorithm for Regression

1. Assign a weight w_j to each observation $j = 1, 2, \dots, n$ and initialize them to $w_j = \frac{1}{n}$.
2. Choose a loss functions $L(r)$. Three candidates are:
Linear: $L(r) = r$, Square law: $L(r) = r^2$, Exponential: $L(r) = 1 - e^{-r}$.
3. For m from 1 to M :

3.1. Create a sample of size n by sampling with replacement from the training sample, where the probability of inclusion for the j^{th} observation is $p_j = w_j / \sum_{j=1}^n w_j$. To do so, divide a line of length $\sum_{j=1}^n w_j$ into subsections of length w_j . Pick a number based on a uniform distribution defined on the range $[0, \sum_{j=1}^n w_j]$. If the number falls into subsection w_j , pick the j^{th} observation.

3.2. Apply the learning algorithm to the created sample to train regressor $f_m(\mathbf{x})$.

3.3. For $j = 1, \dots, n$, compute the relative absolute residuals:

$$r_{j,m} = \frac{|y_j - f_m(\mathbf{x}_j)|}{\max_j |y_j - f_m(\mathbf{x}_j)|}. \quad (8.28)$$

3.4. For $j = 1, \dots, n$, compute the loss $L(r_{j,m})$.

3.5. Compute the average loss:

$$\bar{L}_m = \sum_{j=1}^n p_j L(r_{j,m}). \quad (8.29)$$

3.6. Compute confidence coefficients

$$\alpha_m = \eta \times \log\left(\frac{1 - \bar{L}_m}{\bar{L}_m}\right), \quad (8.30)$$

where η is the learning rate (a hyperparameter).

3.7. For $j = 1, 2, \dots, n$, update the weights

$$w_j \leftarrow w_j \times \exp\left(-\eta(1 - L(r_{j,m}))\alpha_m\right). \quad (8.31)$$

4. Construct the boosted regressor as the weighted median:

$$\psi_{\text{boost}}(\mathbf{x}) = f_k(\mathbf{x}), \quad (8.32)$$

where to obtain k , $f_m(\mathbf{x}), m = 1, \dots, M$, are sorted such that $f_{(1)}(\mathbf{x}) \leq f_{(2)}(\mathbf{x}) \leq \dots \leq f_{(M)}(\mathbf{x})$, with their corresponding confidence coefficients denoted by $\alpha_{(m)}$. Then

$$k = \operatorname{argmin}_s \left\{ \sum_{m=1}^s \alpha_{(m)} \geq 0.5 \sum_{m=1}^M \alpha_m \right\}.$$

The Intuition behind gradient boosting: Given a training data of size n , denoted $\mathbf{S}_{tr} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$, we aim to use a sequential process to construct a predictive model that can estimate the response y for a given feature vector \mathbf{x} . Suppose in this sequential process, our estimate of y after M iteration is based on $F_M(\mathbf{x})$, which has an additive expansion of the form:

$$F_M(\mathbf{x}) = \sum_{m=0}^M \beta_m h_m(\mathbf{x}). \quad (8.33)$$

The set of $h_m(\mathbf{x})$ is known as the set of basis functions with β_m being the coefficient of $h_m(\mathbf{x})$ in the expansion. From the boosted learning perspective, $h_m(\mathbf{x})$ is a “base learner” (usually taken as a classification tree) and β_m is its contribution to the final prediction. Suppose we wish to implement a greedy algorithm where at iteration m , $\beta_m h_m(\mathbf{x})$ is added to the expansion but we are not allowed to revise the decision made in the previous iteration (i.e., $\beta_0 h_0(\mathbf{x}), \dots, \beta_{m-1} h_{m-1}(\mathbf{x})$). Therefore, at iteration M , we can write,

$$\hat{y}_M = F_M(\mathbf{x}) = F_{M-1}(\mathbf{x}) + \beta_M h_M(\mathbf{x}), \quad (8.34)$$

where \hat{y} denotes the estimate of y . For any iteration $m \leq M$, our estimate \hat{y}_m is obtained as

$$\hat{y}_m = F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \beta_m h_m(\mathbf{x}). \quad (8.35)$$

Suppose the function $F_{m-1}(\mathbf{x}) = F_{m-2}(\mathbf{x}) + \beta_{m-1} h_{m-1}(\mathbf{x})$ is somehow obtained. Of course this itself gives us an estimate of $y_i, i = 1, \dots, n$, but now we would like to estimate $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \beta_m h_m(\mathbf{x})$ to have a better estimate of y_i . As $F_{m-1}(\mathbf{x})$ is already obtained, estimating $F_m(\mathbf{x})$ is equivalent to estimating $\beta_m h_m(\mathbf{x})$ (recall that our algorithm is greedy). But how can we find $\beta_m h_m(\mathbf{x})$ to improve the estimate? One option is to learn $h_m(\mathbf{x})$ by *constraining* that to satisfy $\beta_m h_m(\mathbf{x}_i) = y_i - F_{m-1}(\mathbf{x}_i)$; that is, $\beta_m h_m(\mathbf{x}_i)$ becomes the *residual* $y_i - F_{m-1}(\mathbf{x}_i)$. With this specific choice we have

$$F_m(\mathbf{x}_i) = F_{m-1}(\mathbf{x}_i) + \beta_m h_m(\mathbf{x}) = F_{m-1}(\mathbf{x}_i) + (y_i - F_{m-1}(\mathbf{x}_i)) = y_i, \quad i = 1, \dots, n. \quad (8.36)$$

This means that with this specific choice of $\beta_m h_m(\mathbf{x})$, we have at least perfect estimates of response for training data through the imposed n constraints. However, this does not mean $\beta_m h_m(\mathbf{x})$ will lead to perfect estimate of response for any given \mathbf{x} . Although (8.36) looks trivial to some extent, there is an important observation to make. Note that if we take the squared-error loss

$$L(y, F(\mathbf{x})) = \frac{1}{2}(y - F(\mathbf{x}))^2, \quad (8.37)$$

then

$$\frac{\partial L(y, F(\mathbf{x}))}{\partial F(\mathbf{x})} = F(\mathbf{x}) - y, \quad (8.38)$$

and therefore,

$$\left[\frac{\partial L(y, F(\mathbf{x}))}{\partial F(\mathbf{x})} \right] \Bigg|_{\substack{F(\mathbf{x}) = F_{m-1}(\mathbf{x}) \\ \mathbf{x} = \mathbf{x}_i, y = y_i}} = F_{m-1}(\mathbf{x}_i) - y_i. \quad (8.39)$$

The left side of (8.39) is sometimes written as $\frac{\partial L(y_i, F_{m-1}(\mathbf{x}_i))}{\partial F_{m-1}(\mathbf{x}_i)}$ but for the sake of clarity, we prefer the notation used in (8.39). From (8.39) we can write (8.36) as

$$F_m(\mathbf{x}_i) = F_{m-1}(\mathbf{x}_i) - \left[\frac{\partial L(y, F(\mathbf{x}))}{\partial F(\mathbf{x})} \right] \Bigg|_{\substack{F(\mathbf{x}) = F_{m-1}(\mathbf{x}) \\ \mathbf{x} = \mathbf{x}_i, y = y_i}}, \quad i = 1, \dots, n. \quad (8.40)$$

This means that given $F_{m-1}(\mathbf{x})$, we move in the direction of negative of gradient of loss with respect to the function. This is indeed the “steepest descent” minimization in the function space. The importance of viewing the “correction” term in (8.36) as in (8.40) is that we can now replace the squared-loss function with any other differentiable loss function (for example to alleviate the sensitivity of squared-error loss to outliers). Although the negative of gradient of the squared-loss becomes the residual, for other loss functions it does not; therefore, $r_{im} \triangleq -\left[\frac{\partial L(y, F(\mathbf{x}))}{\partial F(\mathbf{x})} \right] \Bigg|_{\substack{F(\mathbf{x}) = F_{m-1}(\mathbf{x}) \\ \mathbf{x} = \mathbf{x}_i, y = y_i}}$ is known as “pseudo”-residuals (Friedman, 2002).

Note that $h_m(\mathbf{x})$ is a member of a class of models (e.g., a weak learner). Therefore, one remaining question is whether we can impose the n aforementioned constraints on $h_m(\mathbf{x}_i)$; that is to say, can we enforce the following relation for an arbitrary differentiable loss function?

$$\beta_m h_m(\mathbf{x}_i) = r_{im}, \quad i = 1, \dots, n, \quad (8.41)$$

Perhaps we can not ensure (8.41) exactly; however, we can choose a member of this weak learner class (to learn it) that produces a vector $\mathbf{h}_m = [h_m(\mathbf{x}_1), \dots, h_m(\mathbf{x}_n)]^T$, which is most parallel to $\mathbf{r}_m = [r_{1m}, r_{2m}, \dots, r_{nm}]^T$. After all, having \mathbf{h}_m being most parallel to \mathbf{r}_m implies taking a direction that is as parallel as possible to the steepest descent direction across all n observations. In this regard, in (Friedman, 2001)

Friedman suggested fitting $h(\mathbf{x})$ to pseudo-residuals r_{im} , $i = 1, \dots, m$, using a least-squares criterion; that is, using $\{(\mathbf{x}_1, r_{1m}), (\mathbf{x}_2, r_{2m}), \dots, (\mathbf{x}_n, r_{nm})\}$ as the training data to learn $h_m(\mathbf{x})$ by

$$h_m(\mathbf{x}) = \underset{h(\mathbf{x}), \rho}{\operatorname{argmin}} \sum_{i=1}^n (r_{im} - \rho h(\mathbf{x}_i))^2. \quad (8.42)$$

As we can see, to learn $h_m(\mathbf{x})$ we replace responses of y_i with r_{im} (its pseudo-residual at iteration m). Therefore, one may refer to these pseudo-residuals as pseudo-responses as in (Friedman, 2001). Once $h_m(\mathbf{x})$ is learned from (8.42), we can determine the step size β_m from

$$\beta_m = \underset{\beta}{\operatorname{argmin}} \sum_{i=1}^n L(y_i, F_{m-1}(\mathbf{x}_i) + \beta h_m(\mathbf{x}_i)). \quad (8.43)$$

Now we can use $h_m(\mathbf{x})$ and β_m to update the current estimate $F_{m-1}(\mathbf{x})$ from (8.35). For implementation of this iterative algorithm, we can initialize $F_0(\mathbf{x})$ and terminate the algorithm when we reach M . To summarize the gradient boosting algorithm, it is more convenient to first present the algorithm for regression, which aligns well with the aforementioned description.

Gradient Boosting Algorithm for Regression: Here we present an algorithmic description of gradient boosting as originally presented in (Friedman, 2001, 2002). In this regard, hereafter, we assume $h_m(\mathbf{x})$ is a member of a class of functions that is parametrized by $\mathbf{a}_m = \{a_{m,1}, a_{m,2}, \dots\}$. As a result, estimating $h_m(\mathbf{x})$ is equivalent to estimating the parameter set \mathbf{a}_m and, therefore, we can use notation $h(\mathbf{x}; \mathbf{a}_m)$ to refer to the weak learner at iteration m .

Algorithm Gradient Boosting Algorithm for Regression

1. Initialize: $F_0(\mathbf{x}) = \underset{\beta}{\operatorname{argmin}} \sum_{i=1}^n L(y_i, \beta)$.
2. For m from 1 to M :

2.1. Compute pseudo-residuals:

$$r_{im} \triangleq -\left[\frac{\partial L(y, F(\mathbf{x}))}{\partial F(\mathbf{x})} \right] \Bigg|_{\begin{array}{l} F(\mathbf{x}) = F_{m-1}(\mathbf{x}), \\ \mathbf{x} = \mathbf{x}_i, y = y_i \end{array}}, \quad i = 1, \dots, n. \quad (8.44)$$

2.2. Estimate (train) the weak learner parameter set using pseudo-residuals and least-squares criterion:

$$\mathbf{a}_m = \underset{\mathbf{a}, \rho}{\operatorname{argmin}} \sum_{i=1}^n (r_{im} - \rho h(\mathbf{x}_i; \mathbf{a}))^2.$$

2.3. Compute: $\beta_m = \underset{\beta}{\operatorname{argmin}} \sum_{i=1}^n L(y_i, F_{m-1}(\mathbf{x}_i) + \beta h(\mathbf{x}_i; \mathbf{a}_m))$.

2.4. Update: $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \beta_m h(\mathbf{x}; \mathbf{a}_m)$.

3. The regressor is $F_M(\mathbf{x})$
-

For the squared-error loss ($L(y_i, a) = \frac{1}{2}(y_i - a)^2$), for example, line 1 and 2.1 in the above algorithm simplify to $F_0(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^n y_i$ and $r_{im} = y_i - F_{m-1}(\mathbf{x}_i)$, respectively. Using the absolute-error loss ($L(y_i, a) = |y_i - a|$) line 1 and 2.1 are replaced with $F_0(\mathbf{x}) = \text{median}(y_i \in \mathcal{S}_{tr})$ and $r_{im} = \text{sign}(y_i - F_{m-1}(\mathbf{x}_i))$, respectively, where $\text{sign}(b)$ is 1 for $b \geq 0$ and -1, otherwise.

8.6.3 \oplus Gradient Boosting Regression Tree

Gradient Boosting Regression Tree (GBRT) for Regression: By regression trees, we are specifically referring to CART for regression discussed in Section 7.3. Considering a J -terminal CART, the parameter set \mathbf{a} is determined by:

- 1) the estimate of targets assigned to any observation falling into a leaf node.
- 2) the splits in the tree; that is, the splitting variables and threshold at non-leaf nodes.

The splits can also be perceived as a set of conditions that determine the set of regions $\{R_j\}_{j=1}^J$ that partition the feature space and are identified by the J leaf nodes; after all, a regression tree (at iteration m of gradient boosting) can be written as:

if $\mathbf{x} \in R_{jm}$ then $h(\mathbf{x}; \mathbf{b}_m) = b_{jm}$ or, equivalently,

$$h(\mathbf{x}; \mathbf{b}_m) = \sum_{j=1}^J b_{jm} I_{\{\mathbf{x} \in R_{jm}\}}, \quad (8.45)$$

where $I_{\{\cdot\}}$ is the indicator function, b_{jm} is the estimate of the target (also referred to as score) for any training sample point falling into region R_{jm} , and $\mathbf{b}_m = \{b_{1m}, b_{2m}, \dots, b_{Jm}\}$. Using (8.45) as the update rule in line 2.4 of the Gradient Boosting algorithm and taking $\gamma_{jm} = \beta_m b_{jm}$ leads to the following algorithm for implementation of GBRT.

Algorithm GBRT for Regression

1. Initialize: $F_0(\mathbf{x}) = \operatorname{argmin}_{\gamma} \sum_{i=1}^n L(y_i, \gamma)$.

2. For m from 1 to M :

2.1. Compute pseudo-residuals:

$$r_{im} \triangleq - \left[\frac{\partial L(y, F(\mathbf{x}))}{\partial F(\mathbf{x})} \right] \Bigg|_{\begin{array}{l} F(\mathbf{x}) = F_{m-1}(\mathbf{x}), \\ \mathbf{x} = \mathbf{x}_i, y = y_i \end{array}} \quad i = 1, \dots, n. \quad (8.46)$$

2.2. Train a J -terminal regression tree using $\mathbf{S}_{tr} = \{(\mathbf{x}_1, r_{1m}), \dots, (\mathbf{x}_n, r_{nm})\}$ and a mean squared error (MSE) splitting criterion (see Section 7.3) to give terminal regions R_{jm} , $j = 1, \dots, J$.

2.3. Compute: $\gamma_{jm} = \operatorname{argmin}_{\gamma} \sum_{\mathbf{x}_i \in R_{jm}} L(y_i, F_{m-1}(\mathbf{x}_i) + \gamma)$, $j = 1, \dots, J$.

2.4. Update each region separately: $F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \gamma_{jm} I_{\{\mathbf{x} \in R_{jm}\}}$, $j = 1, \dots, J$.

3. The regressor is $F_M(\mathbf{x})$

In line 2.4 of GBRT, one can also use a shrinkage in which the update rule is

$$F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + \nu \gamma_{jm} I_{\{\mathbf{x} \in R_{jm}\}}, \quad (8.47)$$

where $0 < \nu \leq 1$.

Gradient Boosting Regression Tree (GBRT) for Classification: GBRT is extended into classification by growing one tree for each class (using squared-error splitting criterion) at each iteration of boosting and using a multinomial deviance loss (to compute the pseudo-residuals by its gradient). Without going into the details,

we present the algorithm for c -class classification (readers can refer to (Friedman, 2001) for details).

Algorithm GBRT for Classification

1. Initialize: $F_{k0}(\mathbf{x}) = 0$, $k = 0, \dots, c - 1$.

2. For m from 1 to M :

2.1. Compute class-specific probabilities:

$$p_{km}(\mathbf{x}) = \frac{e^{F_{k(m-1)}(\mathbf{x})}}{\sum_{k=0}^{c-1} e^{F_{k(m-1)}(\mathbf{x})}}, \quad k = 0, \dots, c - 1. \quad (8.48)$$

2.2. For k from 0 to $c - 1$:

2.2.1. Compute pseudo-residuals: $r_{ikm} = I_{\{y_i=k\}} - p_{km}(\mathbf{x}_i)$, $i = 1, \dots, n$.

2.2.2. train a J -terminal regression tree using

$\mathbf{S}_{tr} = \{(\mathbf{x}_1, r_{1km}), (\mathbf{x}_2, r_{2km}), \dots, (\mathbf{x}_n, r_{nkm})\}$ and MSE splitting criterion to give terminal regions R_{jkm} , $j = 1, \dots, J$.

2.2.3. compute: $\gamma_{jkm} = \frac{c-1}{c} \frac{\sum_{\mathbf{x}_i \in R_{jkm}} r_{ikm}}{\sum_{\mathbf{x}_i \in R_{jkm}} |r_{ikm}|(1-|r_{ikm}|)}$, $j = 1, \dots, J$.

2.2.4. update each region separately:

$$F_{km}(\mathbf{x}) = F_{k(m-1)}(\mathbf{x}) + \gamma_{jkm} I_{\{\mathbf{x} \in R_{jkm}\}}, \quad j = 1, \dots, J.$$

3. The class labels are then found by computing $p_k(\mathbf{x}) = \frac{e^{F_{kM}(\mathbf{x})}}{\sum_{k=0}^{c-1} e^{F_{kM}(\mathbf{x})}}$, $k = 0, \dots, c - 1$, and $\hat{y} = \operatorname{argmin}_k p_k(\mathbf{x})$ where \hat{y} is the predicted label for \mathbf{x} .

Scikit-learn implementation: GBRT for classification and regression are implemented using `GradientBoostingClassifier` and `GradientBoostingRegressor` classes from `sklearn.ensemble` module, respectively. By default a regression tree of `max_depth` of 3 will be used as the base learner. The `n_estimators=100` parameter controls the maximum number of base models (M in the aforementioned algorithms). The supported `loss` values for `GradientBoostingClassifier` are `deviance` and `exponential`, and for `GradientBoostingRegressor` are `squared_error`, `absolute_error`, and two more options. For both classes, the `learning_rate` parameter is the shrinkage coefficient $0 < \nu \leq 1$ used in (8.47).

8.6.4 \oplus XGBoost

XGBoost (short for eXtreme Gradient Boosting) is a popular and widely used method that, since its inception in 2014, has helped many teams win in machine learning competitions. XGBoost gained momentum in machine learning after Tianqi Chen and Tong He won a special award named “High Energy Physics meets Machine Learning” (HEP meet ML) for their XGBoost solution during the Higgs Boson Machine Learning Challenge (Adam-Boudarios et al., 2015; Chen and He, 2015). The challenge was posted on Kaggle in May 2014 and lasted for about 5 months, attracting more than 1900 people competing in 1785 teams. Although the XGBoost did not achieve the highest score on the test data, Chen and He received the special “HEP meet ML” award for their solution that was chosen as the most useful model (judged by a compromise between performance and simplicity) for the ATLAS experiment at CERN. From an optimization point of view, there are two properties that distinguish XGBoost from GBRT:

1. XGBoost utilizes a regularized objective function to penalize the complexity of regression tree functions (to guard against overfitting); and
2. XGBoost uses second-order Taylor approximation to efficiently optimize the objective function. This property means that in contrast with the conventional GBRT that is like steepest descent method in function space, XGBoost uses a Newton-Raphson minimization in function space.

In particular, XGBoost estimates y using a sequential process that at each iteration m , the estimate of y denoted as \hat{y}_m for a given \mathbf{x} has an additive expansion of the form

$$\hat{y}_m = F_m(\mathbf{x}) = F_{m-1}(\mathbf{x}) + h(\mathbf{x}; \mathbf{b}_m), \quad (8.49)$$

where $h(\mathbf{x}; \mathbf{b}_m)$ is the mapping that characterizes a regression tree learned at iteration m and is defined similar to (8.45). Hereafter, and for the ease of notations, we denote such a mapping that defines a regression tree as $h(\mathbf{x})$, and one learned during the sequential process at iteration m as $h_m(\mathbf{x})$. Therefore, for a tree with J leaf nodes, $h(\mathbf{x})$ is defined similarly to (8.45), by dropping the iteration index m ; that is,

$$h(\mathbf{x}) = \sum_{j=1}^J b_j I_{\{\mathbf{x} \in R_j\}}, \quad (8.50)$$

where $\{R_j\}_{j=1}^J$ is the set of regions that partition the feature space and are identified by the J leaf nodes in the tree, and b_j is the score given to any training sample point falling into region R_j . Hereafter, we use h and h_m to refer to a regression tree that is characterized by mappings $h(\mathbf{x})$ and $h_m(\mathbf{x})$, respectively. At iteration m , h_m is found by minimizing a *regularized* objective function as follows:

$$h_m = \operatorname{argmin}_{h \in \mathcal{F}} \sum_{i=1}^n L(y_i, F_{m-1}(\mathbf{x}_i) + h(\mathbf{x}_i)) + \Omega(h), \quad (8.51)$$

where L is a loss function to measure the difference between y_i and its estimate at iteration m , $\Omega(h)$ measures the *complexity* of the tree h , and \mathcal{F} is the space of all regression trees. Including $\Omega(h)$ in the objective function penalizes the complexity of trees. This is one way to guard against overfitting, which is more likely to happen with a complex tree than a simple tree. Recall that we can approximate a real-valued function $f(x + \Delta x)$ using second-order Taylor expansion as

$$f(x + \Delta x) \approx f(x) + \frac{d f(x)}{d x} \Delta x + \frac{1}{2} \frac{d^2 f(x)}{d^2 x} \Delta x^2. \quad (8.52)$$

Replacing $f(\cdot)$, x and Δx in (8.52) with $L(y_i, F_{m-1}(\mathbf{x}_i) + h(\mathbf{x}_i))$, $F_{m-1}(\mathbf{x}_i)$, and $h(\mathbf{x}_i)$, respectively, yields,

$$L(y_i, F_{m-1}(\mathbf{x}_i) + h(\mathbf{x}_i)) \approx L(y_i, F_{m-1}(\mathbf{x}_i)) + g_{i,1} h(\mathbf{x}_i) + \frac{1}{2} g_{i,2} h(\mathbf{x}_i)^2, \quad (8.53)$$

where

$$g_{i,1} = \left[\frac{\partial L(y, F(\mathbf{x}))}{\partial F(\mathbf{x})} \right] \Bigg|_{\begin{array}{l} F(\mathbf{x}) = F_{m-1}(\mathbf{x}) \\ \mathbf{x} = \mathbf{x}_i, y = y_i \end{array}}, \quad (8.54)$$

$$g_{i,2} = \left[\frac{\partial^2 L(y, F(\mathbf{x}))}{\partial^2 F(\mathbf{x})} \right] \Bigg|_{\begin{array}{l} F(\mathbf{x}) = F_{m-1}(\mathbf{x}) \\ \mathbf{x} = \mathbf{x}_i, y = y_i \end{array}}. \quad (8.55)$$

For simplicity the partial derivatives in (8.54) and (8.55) are sometimes written as $\frac{\partial L(y_i, F_{m-1}(\mathbf{x}_i))}{\partial F_{m-1}(\mathbf{x}_i)}$ and $\frac{\partial^2 L(y_i, F_{m-1}(\mathbf{x}_i))}{\partial^2 F_{m-1}(\mathbf{x}_i)}$, respectively. Previously in (8.39) we specified (8.54) for the squared-error loss. For the same loss, as an example, it is straightforward to see $g_{i,2} = 1$. Using (8.53) in (8.51) yields

$$h_m = \operatorname{argmin}_{h \in \mathcal{F}} \sum_{i=1}^n \left[L(y_i, F_{m-1}(\mathbf{x}_i)) + g_{i,1} h(\mathbf{x}_i) + \frac{1}{2} g_{i,2} h(\mathbf{x}_i)^2 \right] + \Omega(h). \quad (8.56)$$

As $L(y_i, F_{m-1}(\mathbf{x}_i))$ does not depend on h , it is treated as a constant term in the objective function and can be removed. Therefore,

$$h_m = \operatorname{argmin}_{h \in \mathcal{F}} \sum_{i=1}^n \left[g_{i,1} h(\mathbf{x}_i) + \frac{1}{2} g_{i,2} h(\mathbf{x}_i)^2 \right] + \Omega(h). \quad (8.57)$$

The summation in (8.57) is over the sample points. However, because each sample point belongs to only one leaf, we can rewrite the summation as a summation over leaves. To do so, let I_j denote the set containing the indices of training data in leaf j ; that is, $I_j = \{i | \mathbf{x}_i \in R_j\}$. Furthermore, let J_h denote the number of leaves in tree h . Then, using (8.50) we can write (8.57) as

$$h_m = \underset{h \in \mathcal{F}}{\operatorname{argmin}} \sum_{j=1}^{J_h} \left[\left(\sum_{i \in I_j} g_{i,1} \right) b_j + \frac{1}{2} \left(\sum_{i \in I_j} g_{i,2} \right) b_j^2 \right] + \Omega(h), \quad (8.58)$$

which for simplicity can be written as

$$h_m = \underset{h \in \mathcal{F}}{\operatorname{argmin}} \sum_{j=1}^{J_h} \left[G_{j,1} b_j + \frac{1}{2} G_{j,2} b_j^2 \right] + \Omega(h), \quad (8.59)$$

where

$$G_{j,1} = \sum_{i \in I_j} g_{i,1}, \quad G_{j,2} = \sum_{i \in I_j} g_{i,2}. \quad (8.60)$$

There are various ways to define $\Omega(h)$ but one way that was originally proposed in (Chen and Guestrin, 2016; Chen and He, 2015) and works quite well in practice is to define

$$\Omega(h) = \gamma J_h + \frac{1}{2} \lambda \sum_{j=1}^{J_h} b_j^2, \quad (8.61)$$

where γ and λ are two tuning parameters. The first term in (8.61) penalizes the number of leaves in the minimization (8.59) and the second term causes having a shrinkage effect on the scores similar to what we saw in Section 6.2.2 for logistic regression. In other words, large scores are penalized and, therefore, scores have less room to wiggle. At the same time, having less variation among scores guard against overfitting the training data. Replacing (8.61) in (8.59) yields

$$h_m = \underset{h \in \mathcal{F}}{\operatorname{argmin}} obj_m, \quad (8.62)$$

where

$$obj_m = \sum_{j=1}^{J_h} \left[G_{j,1} b_j + \frac{1}{2} (G_{j,2} + \lambda) b_j^2 \right] + \gamma J_h. \quad (8.63)$$

Each $h \in \mathcal{F}$ is determined uniquely by a structure (the splits) and possible values of leaf scores $b_j, j = 1, \dots, J_h$. Therefore, one way to find h_m is:

1. to enumerate all possible structures; and
2. for each fixed structure find the optimal values of leaf scores that minimize the objective function.

The second part of this strategy, which is finding optimal values of leaf scores for a fixed structure, is quite straightforward. This is because the objective function (8.63) is a quadratic function of b_j . Therefore,

$$\frac{\partial \text{obj}_m}{\partial b_j} = G_{j,1} + (G_{j,2} + \lambda) b_j. \quad (8.64)$$

Setting the derivative (8.64) to 0, we can find the optimal value of scores for a fixed structure with J_h leaves as

$$b_j^* = -\frac{G_{j,1}}{G_{j,2} + \lambda}, j = 1, \dots, J_h, \quad (8.65)$$

and the corresponding value of the objective function becomes

$$\text{obj}_m^* = -\frac{1}{2} \sum_{j=1}^{J_h} \frac{G_{j,1}^2}{G_{j,2} + \lambda} + \gamma J_h. \quad (8.66)$$

Now we return to the first part of the aforementioned strategy. It is generally intractable to enumerate all possible tree structures. Therefore, we can use a greedy algorithm similar to CART that starts from a root node and grow the tree. However, in contrast with impurity measures used in CART (Chapter 7), in XGBoost (8.66) is used to measure the quality of splits. Note that from (8.66), the contribution of each leaf node k to the obj_m^* is $-\frac{1}{2} \frac{G_{k,1}^2}{G_{k,2} + \lambda} + \gamma$. We can write the objective function as

$$\text{obj}_m^* = -\frac{1}{2} \sum_{j=1, j \neq k}^{J_h} \frac{G_{j,1}^2}{G_{j,2} + \lambda} + \gamma(J_h - 1) \underbrace{-\frac{1}{2} \frac{G_{k,1}^2}{G_{k,2} + \lambda} + \gamma}_{\text{contribution of node } k}. \quad (8.67)$$

Suppose node k is split into left and right nodes k_L and k_R . The objective function after this split is

$$\text{obj}_{m, \text{split}}^* = -\frac{1}{2} \sum_{j=1, j \neq k}^{J_h} \frac{G_{j,1}^2}{G_{j,2} + \lambda} + \gamma(J_h - 1) \underbrace{-\frac{1}{2} \frac{G_{k_L,1}^2}{G_{k_L,2} + \lambda} + \gamma}_{\text{contribution of node } k_L} \underbrace{-\frac{1}{2} \frac{G_{k_R,1}^2}{G_{k_R,2} + \lambda} + \gamma}_{\text{contribution of node } k_R}. \quad (8.68)$$

Therefore, the reduction in the objective function achieved by this split is

$$\text{gain} \triangleq \text{obj}_m^* - \text{obj}_{m, \text{split}}^* = \frac{1}{2} \left(\frac{G_{k_L,1}^2}{G_{k_L,2} + \lambda} + \frac{G_{k_R,1}^2}{G_{k_R,2} + \lambda} - \frac{G_{k,1}^2}{G_{k,2} + \lambda} \right) - \gamma. \quad (8.69)$$

To identify the best split at a node, one can enumerate all features and choose the split that results in the maximum gain obtained by (8.69). Similar to CART, if we assume the values of training data ready for the split have v_j distinct values for feature $x_j, j = 1, \dots, p$, then the number of candidate splits (halfway between consecutive values) is $(\sum_{j=1}^p v_j) - p$. The gain obtained by (8.69) can be interpreted as the quality of split. Depending on the regularization parameter γ , it can be negative for some splits. We can grow the tree as long as the gain for some split is still positive. A larger γ results in more splits having a negative gain and, therefore, a more conservative tree expansion. This way we can also think of γ as the minimum gain required to split a node.

Scikit-learn-style implementation of XGBoost: XGBoost is not part of scikit-learn but its Python library can be installed and it has a scikit-learn API that makes it accessible similar to many other estimators in scikit-learn. If Anaconda has been installed as instructed in Chapter 2, one can use Conda package manager to install `xgboost` package for Python (see instructions at [12]). As XGBoost is essentially a form of GBRT, it can be used for both classification and regression. The classifier and regressor classes can be imported as `from xgboost import XGBClassifier` and `from xgboost import XGBRegressor`, respectively. Once imported, they can be used similarly to scikit-learn estimators. Some of the important hyperparameters in these classes are `n_estimators`, `max_depth`, `gamma`, `reg_lambda`, and `learning_rate`, which are the maximum number of base trees, the maximum depth of each base tree, the minimum gain to split a node (see (8.61) and (8.69)), the regularization parameter used in (8.61), and the shrinkage coefficient ν that can be used in (8.49) to update $F_{m-1}(\mathbf{x})$ similar to (8.47), respectively.

Exercises:

Exercise 1: Load the training and test Iris classification dataset that was prepared in Section 4.6. Select the first two features (i.e., columns with indices 0, 1). Write a program to train bagging, pasting, random forests, and boosting using CART for four cases that are obtained by the combination of `min_samples_leaf` ∈ {2, 10} and `n_estimators` ∈ {5, 20} and record (and show) their accuracies on the test data. Note that a `min_samples_leaf=10` will lead to a “shallower” tree than `min_samples_leaf=2`. Using the obtained accuracies, fill the blanks in the following lines:

For bagging, the highest accuracy is obtained by a _____ tree (choose: Shallow or Deep)

For pasting, the highest accuracy is obtained by a _____ tree (choose: Shallow or Deep)

For random forest, the highest accuracy is obtained by a _____ tree (choose: Shallow or Deep)

For AdaBoost, the highest accuracy is obtained by a _____ tree (choose: Shallow or Deep)

Exercise 2: Suppose we have a training dataset from a binary classification problem and we would like to train a form of AdaBoost classifier, to which we refer as “AdaBoost.Diff”. AdaBoost.Diff is defined similar to AdaBoost.SAMME except for how confidence coefficients are estimated. In particular, in AdaBoost.Diff we have

$$\alpha_m = \begin{cases} 5 - \log(\hat{\varepsilon}_m), & \text{if } \hat{\varepsilon}_m > 0.01 \\ 0, & \text{otherwise} \end{cases} \quad (8.70)$$

In training the AdaBoost.Diff classifier, we observe that the error rate of the base classifier trained at iteration 5 on the weighted training data is 0. Let $\psi_{\text{boost}}^5(\mathbf{x})$ denote the AdaBoost.Diff classifier if we decide to stop training at the end of iteration 5. Similarly, let $\psi_{\text{boost}}^{10}(\mathbf{x})$ denote the AdaBoost.Diff classifier if we continue training until the 10th iteration and stop training at the end of iteration 10. We refer to error rates of $\psi_{\text{boost}}^5(\mathbf{x})$ and $\psi_{\text{boost}}^{10}(\mathbf{x})$ and on a specific test dataset as “test error”. Can we claim that the training error of $\psi_{\text{boost}}^5(\mathbf{x})$ on the weighted training data at iteration 5 is 0? How are the test errors of $\psi_{\text{boost}}^5(\mathbf{x})$ and $\psi_{\text{boost}}^{10}(\mathbf{x})$ related?

Exercise 3: Suppose we have n observations (for simplicity take it as an odd number) in a binary classification problem. Let p denote the accuracy of a classifier $\psi(\mathbf{x})$ trained using all these n observations; that is to say, the probability of correctly classifying a given \mathbf{x} . Suppose $p > 0.5$. Assume we use our training data to also construct an ensemble classifier $\psi_E(\mathbf{x})$, which is obtained by using the majority vote among n base classifiers $\psi_i(\mathbf{x}), i = 1, \dots, n$, which are trained by removing each observation at a time from training data:

$$\psi_E(\mathbf{x}) = \arg \max_{y \in \{0, 1\}} \sum_{i=1}^n I_{\{\psi_i(\mathbf{x})=y\}}, \quad (8.71)$$

where $I_{\{S\}}$ is 1 if statement S is true, zero otherwise. Suppose the following assumption holds:

Assumption: Training a base classifier with any subsets of $n - 1$ observations has the same accuracy as training a classifier with all n observations; in other words, each $\psi_i(\mathbf{x})$ has an accuracy p .

This assumption ideally approximates the situation where having one less observation in data does not considerably affect the predictability of a classification rule—this could be the case specially if n is relatively large. Let p_E shows the accuracy of $\psi_E(\mathbf{x})$. Show that $p_E > p$.

\oplus **Exercise 4:**

In Exercise 1, write a program to train XGBoost classifier. In this regard, consider six cases that are obtained by combination of `max_depth` ∈ {5, 10} and `n_estimators` ∈ {5, 20, 100} and show the accuracy of each classifier on the test data. Compare your results with classifiers used in Exercise 1.

Exercise 5: Show that for random variables $e_i, i = 1, \dots, M$, we have:

$$\frac{1}{M} E\left[\left(\sum_{i=1}^M e_i\right)^2\right] \leq \sum_{i=1}^M E[e_i^2], \quad (8.72)$$



Chapter 9

Model Evaluation and Selection

Estimating the performance of a constructed predictive model, also known as *model evaluation*, is of essential importance in machine learning. This is because eventually the degree to which the model has utility rests with its estimated predictive performance. Not only are model evaluation methods critical to judge the utility of a trained model, but they can also play an important role to reach that model in the first place. The process of picking a final model among a large class of candidate models is known as *model selection*—and there are some popular model selection techniques that operate based on model evaluation methods. In this chapter, we first cover common model evaluation methods and then we examine the utility of some of these methods for model selection.

9.1 Model Evaluation

Once a predictive model (classifier or regressor) is constructed, it is particularly important to evaluate its performance. After all, the worth of a model in practice depends on its performance over all possible future data that can be collected from the same problem for which the model is trained. However, such “unseen” data can naturally not be used for this purpose and the only means of estimating the performance of a trained model is to use the limited data at hand. *Model evaluation* stage (also referred to as model assessment or error estimation) include the use of methods and metrics to estimate the predictive performance of a model. To understand the model evaluation process, it would be helpful to distinguish between evaluation *rules* and *metrics*.

A metric is used to quantify the performance of a constructed model. For example, in Section 4.9, we saw the error rate of a classifier, which means the probability of misclassifying an observation drawn from the feature label distribution governing the problem. However, there are other metrics that can be used to measure various predictive aspects of a trained model; for example, *precision*, *recall*, *F₁ score*, and

Area Under the Receiver Operating Characteristic Curve (abbreviated as AUROC or ROC AUC), to just name a few.

A model evaluation rule, on the other hand, refers to the *procedure* that is used to estimate a metric. In other words, a metric could be estimated using various rules. For example, we have previously seen hold-out estimation rule to estimate the error rate of a classifier (Section 4.9). However, one can use other estimation rules such as *resubstitution*, *cross-validation*, and *bootstrap* to estimate the error rate.

For simplicity of discussion, in what follows, estimation rules are described in connection with classification error rate. However, all procedures, except when stated otherwise, are applicable to estimating classifier and regressor performance metrics.

9.1.1 Model Evaluation Rules

Let $\mathbf{S}_{tr,n} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$ denote the training set where index n is used in $\mathbf{S}_{tr,n}$ to highlight the sample size. We will discuss the following estimation rules:

- Hold-out Estimator
- Resubstitution
- Bootstrap
- Cross-validation (CV)
 - Standard K -fold CV
 - Leave-one-out
 - Stratified K -fold CV
 - Shuffle and Split (Random Permutation CV)

Hold-out Estimator: To estimate the performance of a *learning algorithm* Ψ on unseen data, we can simulate the effect of having unseen data using the available training data $\mathbf{S}_{tr,n}$. To do so, we can randomly split $\mathbf{S}_{tr,n}$ into a (reduced) training set $\mathbf{S}_{tr,l}$ and a test set denoted as $\mathbf{S}_{te,m}$ where $l + m = n$ (e.g., it is common to take $m \approx 0.25n$). We then apply the learning algorithm Ψ to $\mathbf{S}_{tr,l}$ to train a classifier $\psi_l(\mathbf{x})$ and test its performance on $\mathbf{S}_{te,m}$. Two points to notice: 1) $\psi_l(\mathbf{x})$ is the outcome of applying Ψ to $\mathbf{S}_{tr,l}$; and 2) the subindex l in $\psi_l(\mathbf{x})$ is used to better distinguish the sample size that is used in training.

Because $\mathbf{S}_{te,m}$ is not used anyhow in training $\psi_l(\mathbf{x})$, it can be treated as “unseen” data. This is, however, a special unseen data because the actual label for each observation that belongs to $\mathbf{S}_{te,m}$ is known and, therefore, can be used to estimate the performance of $\psi_l(\mathbf{x})$. This can be done by using $\psi_l(\mathbf{x})$ to classify each observation of $\mathbf{S}_{te,m}$ and then compare the assigned labels with the actual labels and estimate the error rate of $\psi_l(\mathbf{x})$ as the proportion of samples within $\mathbf{S}_{te,m}$ that are misclassified. This model evaluation rule is known as hold-out estimator.

The hold-out estimate, denoted $\hat{\epsilon}_h$, is naturally representative of the performance of $\psi_l(\mathbf{x})$ in the context within which the data is collected. However, we note that

$\psi_l(\mathbf{x})$ is the outcome of applying the learning algorithm Ψ to $\mathbf{S}_{tr,l}$. Viewing $\hat{\epsilon}_h$ as the estimate of the performance of the learning algorithm outcome would help prevent confusion and misinterpretations specially in cases when the learning process contains steps such as scaling and feature selection and extraction (Chapter 11). As a result, we occasionally refer to the estimate of the performance of a trained model as the performance of Ψ , which is the learning algorithm that led to the model.

Despite the intuitive meaning of hold-out estimator, its use in small sample (when the sample size with respect to feature size is relatively small) could be harmful for both the training stage and the evaluation process. This is because as compared with large sample, in small sample each observation has generally (and not surprisingly) a larger influence on the learning algorithm. In other words, removing an observation from a small training data generally harms the performance of a learning algorithms more than removing an observation from a large training set. Therefore, when n is relatively small, holding out a portion of $\mathbf{S}_{tr,n}$ could severely hamper the performance of the learning algorithm. At the same time, we have already a small sample to work with and the hold out estimate is obtained even on a smaller set (say 25% of n). Therefore, we can easily end up with an inaccurate and unreliable estimate of the performance not only due to the small test size but also because the estimate could be very bias towards a specific set picked for testing. As we will see later in this section, there are more elegant rules that can alleviate both aforementioned problems associated with the use of hold-out estimator in small-sample settings.

Nonetheless, one way to remove the harmful impact of holding out a portion of $\mathbf{S}_{tr,n}$ for testing in small-sample is that once $\hat{\epsilon}_h$ is obtained, we apply Ψ to the *entire* dataset $\mathbf{S}_{tr,n}$ to train a classifier $\psi_n(\mathbf{x})$ but still use $\hat{\epsilon}_h$, which was estimated on $\mathbf{S}_{te,m}$ using $\psi_l(\mathbf{x})$, as the performance estimate of $\psi_n(\mathbf{x})$. There is nothing wrong with this practice: we are just trying to utilize all available data in training the classifier that will be finally used in practice. The hold-out estimate $\hat{\epsilon}_h$ can be used as an estimate of the performance of $\psi_n(\mathbf{x})$ because $\hat{\epsilon}_h$ is our final judgement (at least based on hold-out estimator) about the performance of Ψ in the context of interest. Not only that, to train $\psi_n(\mathbf{x})$ we add more data to the data that was previously used to train $\psi_l(\mathbf{x})$. With “well-behaved” learning algorithms, this should not generally lead to a worse performance. Although one may argue that it makes more sense to use $\hat{\epsilon}_h$ as an estimate of an upper bound on the error rate of $\psi_n(\mathbf{x})$, on a given dataset and with hold-out estimator, we have no other means to examine that and, therefore, we use $\hat{\epsilon}_h$ as the estimate of error rate of $\psi_n(\mathbf{x})$.

Resubstitution Estimator: The outcome of applying Ψ to a training set $\mathbf{S}_{tr,n}$ is a classifier $\psi_n(\mathbf{x})$. The proportion of samples within $\mathbf{S}_{tr,n}$ that are misclassified by $\psi_n(\mathbf{x})$ is the *resubstitution estimate* of the error rate (Smith, 1947) (also known as *apparent error rate*). Resubstitution estimator will generally lead to an overly optimistic estimate of the performance. This is not surprising because the learning algorithm attempts to fit $\mathbf{S}_{tr,n}$ to some extent. Therefore, it naturally performs better on $\mathbf{S}_{tr,n}$ rather than on an independent set of data collected from the underlying feature-label distributions. As a result this estimator is not generally recommended for estimating the performance of a predictive model.

Bootstrap: Given a training data $\mathbf{S}_{tr,n}$, B bootstrap samples, denoted $\mathbf{S}_{tr,n,k}^*$, $k = 1, 2, \dots, B$, are created by randomly drawing n observations with replacement from $\mathbf{S}_{tr,n}$. Each bootstrap sample may not include some observations from $\mathbf{S}_{tr,n}$ because some other observations are replicated. The main idea behind the bootstrap family of estimators is to treat the observations that do not appear in $\mathbf{S}_{tr,n,k}^*$ as test data for a model that is trained using $\mathbf{S}_{tr,n,k}^*$. In particular, a specific type of bootstrap estimator generally referred to as E0 estimator (here denoted as $\hat{\varepsilon}_B^0$) is obtained based on the following algorithm.

Algorithm E0 Estimator

- 1) Set B to an integer between 25 to 200 (based on a suggestion from Efron in (Efron, 1983)).
- 2) Let \mathbf{A}_k , $k = 1, \dots, B$, denote the set of observations that do not appear in bootstrap sample k ; that is, $\mathbf{A}_k \triangleq \mathbf{S}_{tr,n} - \mathbf{S}_{tr,n,k}^*$
- 3) Classify each observation in \mathbf{A}_k by a classifier $\psi_k(\mathbf{x})$ that is trained by applying Ψ to $\mathbf{S}_{tr,n,k}^*$. Let e_k denote the number of misclassified observations in \mathbf{A}_k .
- 4) $\hat{\varepsilon}_B^0$ error estimate is obtained as the ratio of the number of misclassified observations among samples in \mathbf{A}_k 's over the total number of samples in \mathbf{A}_k 's; that is,

$$\hat{\varepsilon}_B^0 = \frac{\sum_{k=1}^B e_k}{\sum_{k=1}^B |\mathbf{A}_k|}. \quad (9.1)$$

There are other variants of bootstrap (e.g., 0.632 or 0.632+ bootstrap); however, we keep our discussion on bootstrap estimators short because they are computationally more expensive than K -fold CV (for small K)—recall that B in bootstrap is generally between 25 to 200.

Cross-validation: Cross-validation (Lachenbruch and Mickey, 1968) is perhaps the most common technique used for model evaluation because it provides a reasonable compromise between computational cost and reliability of estimation. To understand the essence of cross-validation (abbreviated as CV), we take a look at the hold-out estimator again. In the hold-out, we split $\mathbf{S}_{tr,n}$ to a training set $\mathbf{S}_{tr,I}$ and a test set $\mathbf{S}_{te,m}$. We previously discussed problems associated with the use of hold-out estimator specially in small-sample. In particular, our estimate could be biased to the choice of held-out set $\mathbf{S}_{te,m}$, which means that hold-out estimator can have a large variance from sample to sample (collected from the same population). To improve the variance of hold-out estimator, we may decide to switch the role of training and test sets; that is to say, train a new classifier on $\mathbf{S}_{te,m}$, which is now the training set,

and test its performance on $\mathbf{S}_{tr,l}$, which is now treated as the test set. This provides us with another hold-out estimate based on the same split (except for switching the roles) that we can use along with the previous estimate to estimate the performance of the learning algorithm Ψ (e.g., by taking their average). Although to obtain the two hold-out estimates we preserved the nice property of keeping the test data separate from training, the second training set (i.e., $\mathbf{S}_{te,m}$) has a much smaller sample size with respect to the entire data at hand—recall that m is typically $\sim 25\%$ of n . Therefore, the estimate of the performance that we see by using $\mathbf{S}_{te,m}$ as training set can not be generally a good estimate of the performance of applying Ψ to $\mathbf{S}_{tr,n}$ specially in situations where n is not large. To bring the training sample size used to obtain both of these hold-out estimates as close as possible to n , we need to split the data equally to training and test sets; however, even that can not resolve the problem for moderate to small n because a significant portion of the data (i.e., 50%) is not still used for training. As we will see next, CV also uses independent pairs of test and training sets to estimate the performance. However, in contrast with the above procedure, the procedure of training and testing is repeated multiple times (to reduce the variance) and rather than 50% of the entire data generally larger training sets are used (to reduce the bias).

Standard K -fold CV: This is the basic form of cross-validation and is implemented as in the following algorithm.

Algorithm Standard K -fold CV

1. Split the dataset $\mathbf{S}_{tr,n}$ into K equal-size datasets also known as *folds*. Let Fold_k , $k = 1, \dots, K$ denote fold k ; therefore, $\bigcup_{k=1}^K \text{Fold}_k = \mathbf{S}_{tr,n}$.
2. for k from 1 to K :
 - 2.1. Hold out Fold_k from $\mathbf{S}_{tr,n}$ to obtain $\mathbf{S}_{tr,n} - \text{Fold}_k$ dataset (i.e., the entire data excluding fold K).
 - 2.2. Apply Ψ to $\mathbf{S}_{tr,n} - \text{Fold}_k$ to construct *surrogate classifier* $\psi_k(\mathbf{x})$.
 - 2.3. Classify observations in Fold_k using $\psi_k(\mathbf{x})$. Let e_k denote the number of misclassified observations in Fold_k .
3. Compute the K -fold CV error estimate, denoted $\hat{\epsilon}_{cv}^{K-fold}$, as the ratio of the total number of misclassified observations over the total sample size n ; that is,

$$\hat{\epsilon}_{cv}^{K-fold} = \frac{1}{n} \sum_{k=1}^K e_k . \quad (9.2)$$

Fig. 9.1 shows a schematic representation of the working mechanism of K -fold CV. In implementing the K -fold CV, conventionally the first $1/K$ portion of data are assigned to Fold₁, the second $1/K$ portion of data are assigned to Fold₂, ..., and the last $1/K$ portion of data are assigned to Fold _{K} . However, this practice could cause some problems. For example, in Fig. 9.1, we see that for the first two iterations of the K -fold CV (i.e., $k = 1, 2$), the held-out folds (i.e., Fold₁ and Fold₂) only contain data from Class 0. This means the performance estimate of surrogate classifiers $\psi_1(\mathbf{x})$ and $\psi_2(\mathbf{x})$ is only relevant in predicting instances of Class 0. Not only that, even in some iterations, to train a surrogate classifier, the training data may not include data from all classes. For example, here to train $\psi_5(\mathbf{x})$ no instance of Class 2 is used in training but then the evaluation is performed merely on instances of Class 2. To alleviate these issues, we can shuffle the data before dividing them into K folds. However, to shuffle the data, we need to assume that samples within each class are independent and identically distributed. If this assumption does not hold, by shuffling we may use highly dependent data across training and test sets, which could lead to an overly optimistic error estimate. Fig. 9.2 shows the training and the test sample indices for the K -fold CV where the data is shuffled before splitting into K folds. The size of each fold in this figure is similar to the size of folds in Fig. 9.1. Furthermore, similar to Fig. 9.1, each observation is used as a test observation in only one fold.

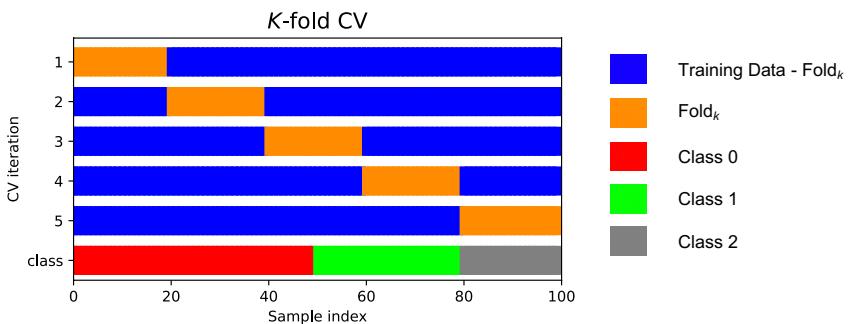


Fig. 9.1: The working mechanism of standard K -fold CV where the data is divided into 5 consecutive folds ($K = 5$).

Leave-one-out (loo): The loo estimator of the performance is obtained by setting $K = n$ where n is the sample size; that is to say, the data is divided into n folds where each fold contains only one observation. Therefore, to construct $\mathbf{S}_{tr,n} - \text{Fold}_k$, $k = 1, \dots, n$, we only need to sequentially leave each observation out. All aforementioned steps in the K -fold CV follows similarly by noting that Fold _{k} contains only one observation. Naturally, shuffling data has no effect because all observations will be held out precisely once to serve as a test observation for a surrogate classifier. Furthermore, because in each iteration of loo, the training data $\mathbf{S}_{tr,n} - \text{Fold}_k$ is almost the same size as the entire training data $\mathbf{S}_{tr,n}$, we expect loo to possess a

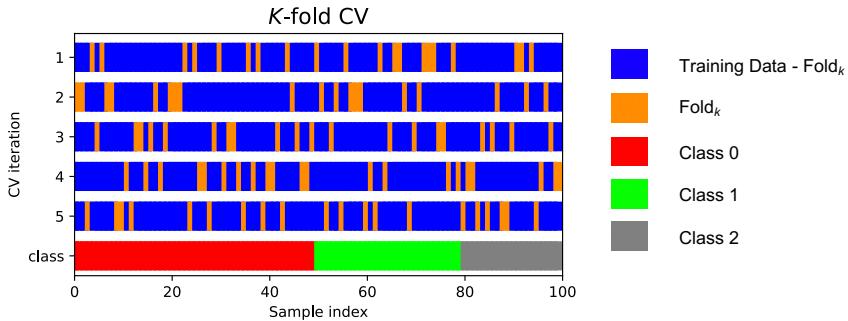


Fig. 9.2: The working mechanism of K -fold CV with shuffling where $K = 5$.

lower bias with respect to the K -fold CV ($K < n$) that has n/K observations less than $\mathbf{S}_{tr,n}$. That being said, loo has a larger variance (Kittler and DeVijver, 1982). Furthermore, we note that the use of loo estimator needs constructing n surrogate classifiers. Therefore, this estimator is practically feasible when n is not large. In addition,

Stratified K -fold CV: In Section 4.4, we stated that it is generally a good practice to split the data into training and test sets in a stratified manner to have the same proportion of samples from different classes in both the training and the test sets. This is because under a random sampling for data collection, the proportion of samples from each class is itself an estimate of the *prior probability* of that class; that is, an estimate of the probability that an observation from a class appears before making any measurements. These prior probabilities are important because once a classifier is developed for a problem with specific class prior probabilities, one expects to employ the classifier in the future under a similar context where the prior probabilities are similar. In other words, in a random sampling, if the proportion of samples from each class are changed radically from training set to test set (this is specially common in case of sampling the populations separately), we may naturally expect a failed behaviour from both the trained classifier (Shahrokh Esfahani and Dougherty, 2013) and its performance estimators (Braga-Neto et al., 2014).

In terms of model evaluation using CV, this means in order to have a realistic view of the classifier performance on test data, in each iteration of CV we should keep the proportion of samples from each class in the training set and the held out fold similar to the full training data. This is achieved by another variant of cross-validation known as *stratified K -fold CV*, which in classification problems is generally the preferred method compared with the standard K -fold CV. Fig. 9.3 shows the training and test sample indices for the stratified K -fold CV where the data is divided into 5 consecutive folds. The size of each held-out fold in this figure is similar to the size of folds in Fig 9.1; however, the proportion of samples from each class that appears in each fold is kept approximately the same as the original dataset. As before we may

even shuffle data before splitting to reduce the chance of having a systematic bias in the data. Fig. 9.4 shows a schematic representation of splits in this setting.

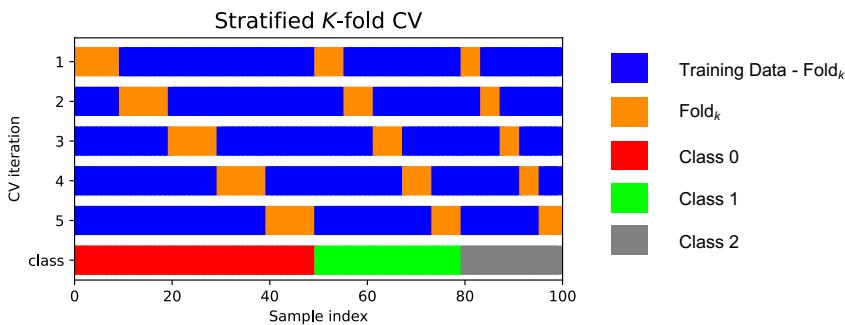


Fig. 9.3: The working mechanism of stratified K -fold CV where the data is divided into a set of consecutive folds such that the proportion of samples from each class is kept the same as the full data ($K = 5$).

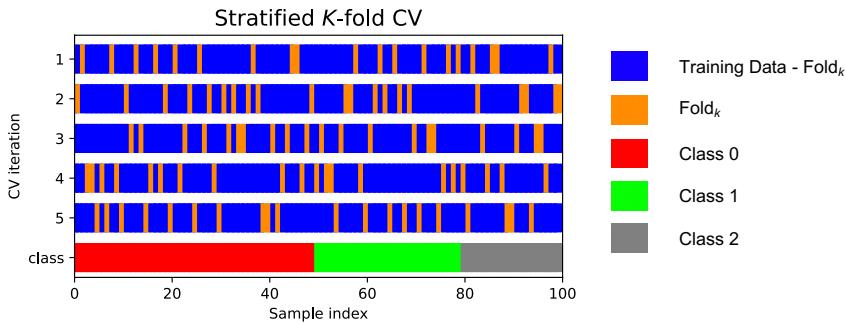


Fig. 9.4: The working mechanism of stratified K -fold CV with shuffling ($K = 5$).

Shuffle and Split (Random Permutation CV): Suppose we have a training dataset that is quite large in terms of sample size and, at the same time, we have a computationally intensive Ψ to evaluate. In such a setting, even training one single model using all the available dataset may take a relatively long time. Evaluating Ψ on this data using leave-one-out is certainly not a good option because we need to repeat the entire training process n times on almost the entire dataset. We may decide to use, for example, 5-fold CV. However, even that could not be efficient because for each iteration of 5-fold CV, the training data has a sample size of $n - n/5$, which is still large and could be an impediment in repeating the training process several times. In such a setting, we may still decide to train our final model on the entire training

data (to use all available data in training the classifier that will be finally used in practice), but for evaluation, we may randomly choose a subset of the entire dataset for training and testing, and repeat this procedure K times. This *shuffle and split* evaluation process (also known as *random permutation CV*) does not guarantee that an observation that is used once in a fold as a test observation does not appear in other folds. Fig. 9.5 shows a schematic representation of the splits for this estimator where we choose the size of training data at each iteration to be 50% of the entire dataset, the test data to be 20%, and the rest 30% are unused data (neither used for training nor testing).

One might view the performance estimate obtained this way as an overly pessimistic estimate of our final trained model; after all, the final model is generally trained on a much larger dataset, which means the final model has generally a better performance compared to any surrogate model trained as part of this estimator. However, we need to note that removing a portion of samples in large-sample settings has generally less impact on the performance of a learning algorithm than removing a similar proportion in small-sample situations. At the same time, shuffle-and-split is commonly used in large-sample settings, which to some extent limits the degree of its pessimism.

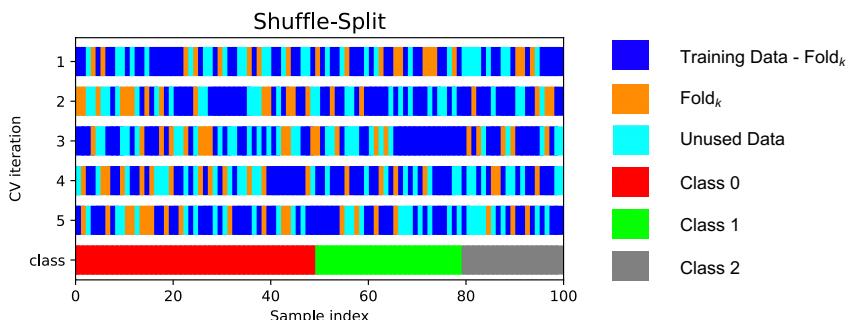


Fig. 9.5: The working mechanism of shuffle and split where the training and test data at each iteration is 50% and 20%, respectively. In contrast with other forms of cross-validation, here an observation could be used for testing multiple times across various CV iterations.

Scikit-learn implementation of various cross-validation schemes: The `KFold` class from `sklearn.model_selection` module creates the indices for Fold_k and $\mathbf{S}_{tr,n} - \text{Fold}_k$ for each iteration of K -fold CV. By default, it splits the data $\mathbf{S}_{tr,n}$ into 5 consecutive folds without applying any shuffling (similar to Fig. 9.1). However, using the following parameters we can change its default behaviour:

- 1) `n_splits` changes the number of folds;
- 2) setting `shuffle` to `True` leads to shuffling the data before splitting into `n_splits` folds; and

- 3) using `random_state` we can create reproducible results if `shuffle` is set to `True`.

The leave-one-out estimator is obtained using `LeaveOneOut` class from the same module. This is equivalent to using `KFold(n_splits=n)` where n is the sample size.

Let us now use this class to implement the K -fold CV in the Iris classification dataset. In this regard, we load the original Iris dataset before any preprocessing, as before we split the dataset into a training and test set (in case a test set is also desired), and we apply the cross-validation only on the training data. We create an object from `KFold` class and invoke the `split` method to generate the training ($\mathbf{S}_{tr,n} - \text{Fold}_k$) and test (Fold_k) sets for each iteration of the 3-fold CV. Furthermore, in iteration k , we train a classifier (here 5NN) on the training set ($\mathbf{S}_{tr,n} - \text{Fold}_k$) and test its accuracy on Fold_k , which is the test set for that iteration. Last but not least, we obtain the overall K -fold CV estimate of the accuracy by taking the mean of the accuracy estimates obtained at each iteration.

```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier as KNN

# load training data
iris = datasets.load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.
    target, random_state=100, test_size=0.2, stratify=iris.target)
print('X_train_shape: ' + str(X_train.shape) + '\nX_test_shape: ' +
    str(X_test.shape) +
    '\ny_train_shape: ' + str(y_train.shape) + '\ny_test_shape: ' +
    str(y_test.shape) + '\n')

# to set up the K-fold CV splits
K_fold = 3
kfold = KFold(n_splits=K_fold, shuffle=True, random_state=42) # ↵
# shuffling is used
cv_scores = np.zeros(K_fold)
knn = KNN()
for counter, (train_idx, test_idx) in enumerate(kfold.split(X_train, ↵
    y_train)):
    print("K-fold CV iteration " + str(counter+1))
    print("Train indices:", train_idx)
    print("Test indices:", test_idx, "\n")
    X_train_kfold = X_train[train_idx,]
    y_train_kfold = y_train[train_idx,]
    X_test_kfold = X_train[test_idx,]
    y_test_kfold = y_train[test_idx,]
```

```

cv_scores[counter] = knn.fit(X_train_kfold, y_train_kfold).
score(X_test_kfold, y_test_kfold)

print("the accuracy of folds are: ", cv_scores)
print("the overall 3-fold CV accuracy is: {:.3f}".format(cv_scores.
mean()))

```

X_train_shape: (120, 4)

X_test_shape: (30, 4)

y_train_shape: (120,)

y_test_shape: (30,)

K-fold CV iteration 1

Train indices: [1 2 3 5 6 7 8 13 14 16 17 19 20 21 23 25 27 28 29 32 33 34 35 37 38 39 41 43 46 48 49 50 51 52 54 57 58 59 60 61 63 66 67 68 69 71 72 74 75 77 79 80 81 82 83 84 85 86 87 90 92 93 94 95 98 99 100 101 102 103 105 106 108 111 112 113 115 116 117 119]

Test indices: [0 4 9 10 11 12 15 18 22 24 26 30 31 36 40 42 44 45 47 53 55 56 62 64 65 70 73 76 78 88 89 91 96 97 104 107 109 110 111 114 115 116 118 119]

K-fold CV iteration 2

Train indices: [0 1 2 4 9 10 11 12 14 15 18 20 21 22 23 24 26 29 30 31 32 36 37 40 41 42 44 45 47 48 51 52 53 55 56 57 58 59 60 61 62 63 64 65 70 71 73 74 75 76 78 79 81 82 86 87 88 89 91 92 93 96 97 99 101 102 103 104 105 106 107 108 109 110 112 114 115 116 118 119]

Test indices: [3 5 6 7 8 13 16 17 19 25 27 28 33 34 35 38 39 43 46 49 50 54 66 67 68 69 72 77 80 83 84 85 90 94 95 98 100 111 113 117]

K-fold CV iteration 3

Train indices: [0 3 4 5 6 7 8 9 10 11 12 13 15 16 17 18 19 22 24 25 26 27 28 30 31 33 34 35 36 38 39 40 42 43 44 45 46 47 49 50 53 54 55 56 62 64 65 66 67 68 69 70 72 73 76 77 78 80 83 84 85 88 89 90 91 94 95 96 97 98 100 104 107 109 110 111 113 114 117 118]

Test indices: [1 2 14 20 21 23 29 32 37 41 48 51 52 57 58 59 60 61 63 71 74 75 79 81 82 86 87 92 93 99 101 102 103 105 106 108 112 115 116 119]

the accuracy of folds are: [0.975 0.975 0.95]

the overall 3-fold CV accuracy is: 0.967

Note that here we did not use `np.load('data/iris_train_scaled.npz')` to load the Iris dataset that we preprocessed before. This is not a matter of taste but necessity to avoid data leakage. To understand the reason, it is important to realize that each Fold_k should be precisely treated as an independent test set. At the same time,

recall from Section 4.4-4.6 that when both splitting and preprocessing are desired, the data is first split into training and test sets and then preprocessing is applied to the training set only (and the test set is transformed by statistics obtained on training set). This practice ensures that no information from the test set is used in training via preprocessing. In CV, our training set after the “split” at each iteration is $S_{tr,n} - \text{Fold}_k$ and the test data is Fold_k . Therefore, if a preprocessing is desired, the preprocessing must be applied to $S_{tr,n} - \text{Fold}_k$ at each iteration separately and Fold_k should be transformed by statistics obtained on $S_{tr,n} - \text{Fold}_k$. In Chapter 11, we will see how this could be done in scikit-learn. In the meantime, we have to realize that if we apply the CV on the training data stored in `np.load('data/iris_train_scaled.npz')`, some information of Fold_k has already been used in $S_{tr,n} - \text{Fold}_k$ via preprocessing, which is not legitimate (see Section 4.6 for a discussion on why this is an illegitimate practice).

Rather than implementing a for loop as shown above to train and test surrogate classifiers at each iteration, scikit-learn offers a convenient way to obtain the CV scores using `cross_val_score` function from `sklearn.model_selection` module. Among several parameters of this class, the following parameters are specially useful:

- 1) `estimator` is to specify the estimator (e.g., classifier or regressor) used within CV;
- 2) `X` is the training data;
- 3) `y` is the values of target variable;
- 4) `cv` determines the CV strategy as explained here:
 - setting `cv` to an integer K in a classification problem uses stratified K -fold CV with no shuffling. To implement, the K -fold CV for classification, we can also set `cv` parameter to an object of `KFold` class and if shuffling is required, the `shuffle` parameter of `KFold` should be set to `True`;
 - setting `cv` to an integer K in a regression problem uses standard K -fold CV with no shuffling (note that stratified K -fold CV is not defined for regression). If shuffling is required, the `shuffle` parameter of `KFold` should be set to `True` and `cv` should be set to an object of `KFold`;
- 5) `n_jobs` is used to specify the number of CPU cores that can be used in parallel to compute the CV estimate. Setting `n_jobs=-1` uses all processors; and
- 6) `scoring`: the default `None` value of this parameter leads to using the default metric of the estimator `score` method; for example, in case of using the estimator `DecisionTreeRegressor`, \hat{R}^2 is the default scorer and in case of `DecisionTreeClassifier`, the accuracy estimator is the default scorer.

`cross_val_score` uses a `scoring` function (the higher, the better) rather than a `loss` function (the lower, the better). In this regard, a list of possible scoring functions are available at ([Scikit-metrics, 2023](#)). We will also discuss in detail some of these scoring functions later in this Chapter. Here we use `cross_val_score` to implement the standard K -fold CV. Note that the results are the same as what were obtained before.

```
from sklearn.model_selection import cross_val_score

#kfold = KFold(n_splits=K_fold, shuffle=True, random_state=42) # kfold
#    ↵is already instantiated from this line that we had before
knn = KNN()
cv_scores = cross_val_score(knn, X_train, y_train, cv=kfold)
print("the accuracy of folds are: ", cv_scores)
print("the overall 3-fold CV accuracy is: {:.3f}".format(cv_scores.
    ↵mean()))
```

the accuracy of folds are: [0.975 0.975 0.95]

the overall 3-fold CV accuracy is: 0.967

Next we use `cross_val_score` to implement the stratified K -fold CV. As we said before, setting `cv` parameter to an integer by default uses the stratified K -fold CV for classification.

```
cv_scores = cross_val_score(knn, X_train, y_train, cv=3)
print("the accuracy of folds are: ", cv_scores)
print("the overall 3-fold CV accuracy is: {:.3f}".format(cv_scores.
    ↵mean()))
```

the accuracy of folds are: [0.925 0.95 1.]

the overall 3-fold CV accuracy is: 0.958

Similar to `KFold` class that can be used to create the indices for the training and test at each CV iteration, we also have `StratifiedKFold` class from `sklearn.model_selection` module that can be used for creating indices. Therefore, another way that gives us more control over stratified K -fold CV is to set the `cv` parameter of `cross_val_score` to an object from `StratifiedKFold` class. I For example, here we shuffle the data before splitting (similar to Fig. 9.4) and set the `random_state` for reproducibility:

```
from sklearn.model_selection import StratifiedKFold
strkfold = StratifiedKFold(n_splits=K_fold, shuffle=True,
    ↵random_state=42)
cv_scores = cross_val_score(knn, X_train, y_train, cv=strkfold)
print("the accuracy of folds are: ", cv_scores)
print("the overall 3-fold CV accuracy is: {:.3f}".format(cv_scores.
    ↵mean()))
```

the accuracy of folds are: [0.925 1. 1.]

the overall 3-fold CV accuracy is: 0.975

To implement the shuffle and split method, we can use `ShuffleSplit` class from `sklearn.model_selection` module that can be used for creating indices and then set the `cv` parameter of `cross_val_score` to an object from `ShuffleSplit` class.

Two important parameters of this class are `test_size` and `train_size`, which could be either a floating point between 0.0 and 1.0 or an integer. In the former case, the value of the parameter is interpreted as the proportion with respect to the entire dataset, and in the latter case, it becomes the number of observations used.

9.1.2 Evaluation Metrics for Classifiers

Our aim here is to use an evaluation rule to estimate the performance of a trained classifier. In this section, we will cover different metrics to measure various aspects of a classifier performance. We will discuss both the probabilistic definition of these metrics as well as their empirical estimators. Although we distinguish the probabilistic definition of a metric from its empirical estimator, it is quite common in machine learning literature to refer to an empirical estimate as the metric itself. For example, the *sensitivity* of a classifier is a probabilistic concept. However, because in practice we almost always need to estimate it from a data at hand, it is common to refer to its empirical estimate *per se*, as the sensitivity.

Suppose a training set $\mathbf{S}_{tr,n}$ is available and used to train a binary classifier ψ . In particular, ψ maps realizations of random feature vector \mathbf{X} to realizations of a class variable Y that takes two values: “P” and “N”, which are short for “positive” and “negative”, respectively; that is, $\psi(\mathbf{X}) : \mathbb{R}^p \rightarrow \{P, N\}$ where p is the dimensionality of \mathbf{X} . Although assigning classes as P and N is generally arbitrary, the classification emphasis is often geared towards identifying class P instances; that is to say, we generally label the “atypical” label as positive and the “typical” one as negative. For example, people with a specific disease (in contrast with healthy people) or spam emails (in contrast with non-spam emails) are generally allocated as the positive class.

The mapping produced by $\psi(\mathbf{X})$ from \mathbb{R}^p to $\{P, N\}$ has an intermediary step. Suppose \mathbf{x} is a realization of \mathbf{X} . The classification is performed by first mapping $\mathbf{x} \in \mathbb{R}^p$ to $s(\mathbf{x})$, which is a *score* on a univariate continuum such as \mathbb{R} . This score could be, for example, the distance of an observation from the decision hyperplanes, or the estimated probability that an instance belongs to a specific class. The score $s(\mathbf{x})$ is then compared to t , which is a specific value of threshold T . We assume scores are oriented such that if $s(\mathbf{x})$ exceeds t , then \mathbf{x} is assigned to P and otherwise to N. Formally, we write

$$\psi(\mathbf{x}) = \begin{cases} P & \text{if } s(\mathbf{x}) > t, \\ N & \text{otherwise.} \end{cases} \quad (9.3)$$

In what follows, we define several probabilities that can help measure different aspects of a classifier. To discuss the empirical estimators of these probabilities, we assume the trained classifier ψ is applied to a sample of size m , $\mathbf{S}_{te,m} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_m, y_m)\}$ and observations $\mathbf{x}_1, \dots, \mathbf{x}_m$ are classified as

$\hat{y}_1, \dots, \hat{y}_m$, respectively. We can categorize the outcome of the classifier for each \mathbf{x}_i as a true positive (TP), a false positive (FP), a true negative (TN), or a false negative (FN), which mean:

- 1) TP: both \hat{y}_i and y_i are P;
- 2) FP: \hat{y}_i is P but y_i is N (so the positive label assigned by the classifier is false);
- 3) TN: both \hat{y}_i and y_i are N; and
- 4) FN: \hat{y}_i is N but y_i is P (so the negative label assigned by the classifier is false).

Furthermore, the number of true positives, false positives, true negatives, and false negatives among instances of $\mathbf{S}_{te,m}$ that are classified by ψ are denoted n_{TP} , n_{FP} , n_{TN} , and n_{FN} , respectively. The actual number of positives and negatives within $\mathbf{S}_{te,m}$ are denoted by n_P and n_N , respectively. Therefore, $n_P = n_{TP} + n_{FN}$, $n_N = n_{TN} + n_{FP}$, and $n_{TP} + n_{FP} + n_{TN} + n_{FN} = n_P + n_N = m$. We can summarize the counts of these states in a square matrix as follows, which is known as *Confusion Matrix* (see Fig. 9.6).

n_{TP}	n_{FN}
n_{FP}	n_{TN}

Fig. 9.6: Confusion matrix for a binary classification problem

True positive rate (*tpr*): the probability that a class P instance is correctly classified; that is, $tpr = P(s(\mathbf{X}) > t | \mathbf{X} \in P)$. It is also common to refer to *tpr* as *sensitivity* (as in clinical/medical applications) or *recall* (as in information retrieval literature). The empirical estimator of *tpr*, denoted \hat{tpr} , is obtained as

$$\hat{tpr} = \frac{n_{TP}}{n_P} = \frac{n_{TP}}{n_{TP} + n_{FN}}. \quad (9.4)$$

False positive rate (*fpr*): the probability that a class N instance is misclassified; that is, $fpr = P(s(\mathbf{X}) > t | \mathbf{X} \in N)$. Its empirical estimator, denoted \hat{fpr} , is

$$\hat{fpr} = \frac{n_{FP}}{n_N} = \frac{n_{FP}}{n_{TN} + n_{FP}}. \quad (9.5)$$

True negative rate (*tnr*): the probability that a class N instance is correctly classified; that is, $tpr = P(s(\mathbf{X}) \leq t | \mathbf{X} \in N)$. In clinical and medical applications, it is common to refer to *tpr* as *specificity*. Its empirical estimator is

$$\hat{tnr} = \frac{n_{TN}}{n_N} = \frac{n_{TN}}{n_{TN} + n_{FP}}. \quad (9.6)$$

False negative rate (f_{nr}): the probability that a class P instance is misclassified; that is, $f_{nr} = P(s(\mathbf{X}) \leq t | \mathbf{X} \in P)$. Its empirical estimator is

$$\hat{f}_{nr} = \frac{n_{FN}}{n_P} = \frac{n_{FN}}{n_{TP} + n_{FN}}. \quad (9.7)$$

Although here four metrics are defined, we have $tpr + f_{nr} = 1$ and $fpr + tnr = 1$; therefore, having tpr and fpr summarize all information in these four metrics.

Positive predictive value (ppv): this metric is defined by switching the role of $s(\mathbf{X}) > t$ and $\mathbf{X} \in P$ in the conditional probability used to define tpr ; that is to say, the probability that an instance predicted as class P is truly from that class (i.e., is correctly classified): $ppv = P(\mathbf{X} \in P | s(\mathbf{X}) > t)$. It is also common to refer to ppv as *precision*. The empirical estimator of ppv is

$$\hat{ppv} = \frac{n_{TP}}{n_{TP} + n_{FP}}. \quad (9.8)$$

False discovery rate (fdr)¹: this metric is defined as the probability that an instance predicted as class P is truly from class N (i.e., is misclassified): $fdr = P(\mathbf{X} \in N | s(\mathbf{X}) > t) = 1 - ppv$. The empirical estimator of fdr is

$$\hat{fdr} = \frac{n_{FP}}{n_{TP} + n_{FP}}. \quad (9.9)$$

F_1 score: to summarize the information of recall and precision using one metric, we can use F_1 measure (also known as F_1 score), which is given by the harmonic mean of recall and precision:

$$F_1 = \frac{2}{\frac{1}{\text{recall}} + \frac{1}{\text{precision}}} \frac{2 \text{ recall} \times \text{precision}}{\text{recall} + \text{precision}}. \quad (9.10)$$

As can be seen from (9.10), if either of precision and recall are low, F_1 will be low. The plug-in estimator of F_1 score is obtained by replacing recall and precision in (9.10) with their estimates obtained from (9.4) and (9.8), respectively.

Error rate: As seen in Section 4.9, the error rate, denoted ε , is defined as the probability of misclassification by the trained classifier; that is to say,

$$\varepsilon = P(\psi(\mathbf{X}) \neq Y). \quad (9.11)$$

¹ Inspired by (Soric, 1989), in a seminal paper (Benjamini and Hochberg, 1995), Benjamini and Hochberg coined the term “false discovery rate”, and they defined that as the expectation of (9.9). The use of fdr here can be interpreted similar to the definition of (tail area) false discovery rate by Efron in (Efron, 2007).

However, because misclassification can occur for both instances of P and N class, ε can be equivalently written as

$$\varepsilon = fpr \times P(Y = N) + fnr \times P(Y = P), \quad (9.12)$$

where $P(Y = i)$ is the prior probability of class $i \in \{P, N\}$. The empirical estimate of ε , denoted $\hat{\varepsilon}$, is obtained by using empirical estimates of fpr and fnr given in (9.5) and (9.7), respectively, as well as the empirical estimates of $P(Y = P)$ and $P(Y = N)$, which under a random sampling assumption, are $\frac{n_P}{m}$ and $\frac{n_N}{m}$, respectively; that is,

$$\hat{\varepsilon} = \frac{n_{FP}}{n_N} \times \frac{n_N}{m} + \frac{n_{FN}}{n_P} \times \frac{n_P}{m} = \frac{n_{FP} + n_{FN}}{m} \equiv \frac{n_{FP} + n_{FN}}{n_{FP} + n_{FN} + n_{TP} + n_{TN}}, \quad (9.13)$$

which is simply the proportion of misclassified observations (c.f. Section 4.9). An equivalent simple form to write $\hat{\varepsilon}$ directly as a function of y_j and \hat{y}_j is

$$\hat{\varepsilon} = \frac{1}{m} \sum_{j=1}^m I_{\{y_j \neq \hat{y}_j\}}, \quad (9.14)$$

where I_A is the indicator of event A.

Accuracy: This is the default metric used in the `score` method of classifiers in scikit-learn, and is defined as the probability of correct classification; that is to say,

$$acc = 1 - \varepsilon = P(\psi(\mathbf{X}) = Y). \quad (9.15)$$

Similarly, (9.15) can be written as

$$acc = tnr \times P(Y = N) + tpr \times P(Y = P). \quad (9.16)$$

Replacing tnr , tpr , $P(Y = N)$, and $P(Y = P)$ by their empirical estimates yields the accuracy estimate, denoted \hat{acc} , and is given by

$$\hat{acc} = \frac{n_{TN}}{n_N} \times \frac{n_N}{m} + \frac{n_{TP}}{n_P} \times \frac{n_P}{m} = \frac{n_{TN} + n_{TP}}{m} \equiv \frac{n_{TP} + n_{TN}}{n_{FP} + n_{FN} + n_{TP} + n_{TN}}, \quad (9.17)$$

which is simply the proportion of correctly classified observations. This can be equivalently written as

$$\hat{acc} = \frac{1}{m} \sum_{j=1}^m I_{\{y_j = \hat{y}_j\}}. \quad (9.18)$$

Although accuracy has an intuitive meaning, it could be misleading in cases where the data is imbalanced; that is, in datasets where the proportion of samples from various classes are different. For this reason, it is common to report other metrics

of performance along with the accuracy. The following example demonstrates the situation.

Example 9.1 Consider the following scenario in which we have three classifiers to classify whether the topic of a given news article is politics (P) or not (N). We train three classifiers, namely, Classifier 1, Classifier 2, and Classifier 3, and evaluate them on a test sample $S_{te,100}$ that contains 5 political and 95 non-political articles. The results of this evaluation for these classifiers are summarized as follows:

Classifier 1: this classifier classifies all observations to N class.

Classifier 2: $n_{TP} = 4$ and $n_{TN} = 91$.

Classifier 3: $n_{TP} = 1$ and $n_{TN} = 95$.

Which classifier is better to use?

To answer this question, we first look into the accuracy estimate. However, before doing so, we first obtain all elements of the confusion matrix for each classifier. Since $n_P = n_{TP} + n_{FN}$, $n_N = n_{TN} + n_{FP}$, from the given information we have:

Classifier 1: $n_{TP} = 0$, $n_{TN} = 95$, $n_{FP} = 0$, and $n_{FN} = 5$.

Classifier 2: $n_{TP} = 4$, $n_{TN} = 91$, $n_{FP} = 4$, and $n_{FN} = 1$.

Classifier 3: $n_{TP} = 1$, $n_{TN} = 95$, $n_{FP} = 0$, and $n_{FN} = 4$.

This means that

$$\begin{aligned}\hat{acc}_{\text{Classifier 1}} &= \frac{0 + 95}{0 + 95 + 0 + 5} = 0.95, \\ \hat{acc}_{\text{Classifier 2}} &= \frac{4 + 91}{4 + 91 + 4 + 1} = 0.95, \\ \hat{acc}_{\text{Classifier 3}} &= \frac{1 + 94}{1 + 94 + 1 + 4} = 0.95.\end{aligned}$$

In terms of accuracy, all classifiers show a high accuracy estimate of 95%. However, Classifier 1 seems quite worthless in practice as it classifies any news article to N class! On the other hand, Classifier 2 seems quite effective because it can retrieve four out of the five P instances in $S_{te,100}$, and in doing so it is fairly precise because among the eight articles that it labels as P, four of them is actually correct ($n_{TP} = 4$ and $n_{FP} = 4$). As a result, it is inappropriate to judge the effectiveness of these classifiers based on the accuracy alone. Next we examine the (empirical estimates of) recall (tpr) and precision (ppv) for these classifier:

$$\begin{aligned}\hat{tpr}_{\text{Classifier 1}} &= \frac{0}{0+5} = 0, \\ \hat{ppv}_{\text{Classifier 1}} &= \frac{0}{0+0}, \quad (\text{undefined}) \\ \hat{tpr}_{\text{Classifier 2}} &= \frac{4}{4+1} = 0.8, \\ \hat{ppv}_{\text{Classifier 2}} &= \frac{4}{4+4} = 0.5, \\ \hat{tpr}_{\text{Classifier 3}} &= \frac{1}{1+4} = 0.2, \\ \hat{ppv}_{\text{Classifier 3}} &= \frac{1}{1+0} = 1.\end{aligned}$$

As seen here, in estimating the ppv of Classifier 1, we encounter division by 0. In these situations, one may dismiss the worth of the classifier or, if desired, define the estimate as 0 in order to possibly calculate other metrics of performance that are based on this estimate (e.g., F_1 score). Nonetheless, here we perceive Classifier 1 as worthless and focus on comparing Classifier 2 and Classifier 3. As can be seen, Classifier 2 has a higher \hat{tpr} than Classifier 3 but its \hat{ppv} is lower. How can we decide which classifier to use in practice based on these two metrics?

Depending on the application, sometimes we care more about precision and sometimes we care more about the recall. For example, if we have a cancer diagnostic classification problem, we would prefer having a higher recall because it means having a classifier that can better detect individuals with cancer—the cost of not detecting a cancerous patient could be death. However, suppose we would like to train a classifier that classifies an email as spam (positive) or not (negative) and based on the assigned label it directs the email to the spam folder, which is not often seen, or to the inbox. In this application, it would be better to have a higher precision so that ideally none of our important emails is classified as spam and left unseen in the spam folder. If we have no preference between recall and precision, we can use the F_1 score estimate to summarize both metrics in one. The F_1 scores for Classifier 2 and Classifier 3 are

$$\begin{aligned}\hat{F}_1_{\text{Classifier 2}} &= 2 \times 0.8 \times 0.5 / (0.8 + 0.5) = 0.615, \\ \hat{F}_1_{\text{Classifier 3}} &= 2 \times 1 \times 0.2 / (1 + 0.2) = 0.333,\end{aligned}$$

which shows Classifier 2 is the preferred classifier. ■

Area Under the Receiver Operating Characteristic Curve (ROC AUC): All the aforementioned metrics of performance depend on t , which is the specific value of threshold used in a classifier. However, depending on the context that a trained classifier will be used in the future, we may decide to change this threshold at the final stage of classification, which is a fairly simple adjustment, for example, to balance recall and precision depending on the context. Then the question is whether we can evaluate a classifier over the entire range of the threshold used in its structure rather

than a specific value of the threshold. To do so we can evaluate the classifier by the Receiver Operating Characteristic (ROC) curve and the Area Under the ROC Curve (ROC AUC).

ROC curve is obtained by varying t over its entire range and plotting pairs of (fpr, tpr) where fpr and tpr are values on the horizontal and vertical axes for each t , respectively. Recall that given tpr and fpr , we also have fnr and tnr , which means ROC curve is in fact a concise way to summarize the information in these four metrics. A fundamental property of the ROC curve is that it is a monotonic increasing function from $(0, 0)$ to $(1, 1)$. This is because as $t \rightarrow \infty$, $P(s(\mathbf{X}) > t | \mathbf{X} \in P)$ and $P(s(\mathbf{X}) > t | \mathbf{X} \in N)$ approach zero. At the same time, these probabilities naturally increase for a decreasing t . In the extreme case, when $t \rightarrow -\infty$, both of these probabilities approach one.

Let $p(s(\mathbf{X}) | \mathbf{X} \in P)$ and $p(s(\mathbf{X}) | \mathbf{X} \in N)$ denote the probability density function (pdf) of scores given by a classifier to observations of class P and N, respectively. Therefore, tpr and fpr , which are indeed functions of t , can be written as

$$\begin{aligned} tpr &= \int_t^{\infty} p(s(\mathbf{X}) | \mathbf{X} \in P) ds(\mathbf{X}), \\ fpr &= \int_t^{\infty} p(s(\mathbf{X}) | \mathbf{X} \in N) ds(\mathbf{X}). \end{aligned}$$

Intuitively, a classification rule that produces larger scores for the P class than the N class is more effective than the one producing similar scores for both classes. In other words, the more $p(s(\mathbf{X}) | \mathbf{X} \in P)$ and $p(s(\mathbf{X}) | \mathbf{X} \in N)$ differ, the less likely in general the classifier misclassifies observations. In the extreme case where $p(s(\mathbf{X}) | \mathbf{X} \in P) = p(s(\mathbf{X}) | \mathbf{X} \in N)$, then $P(s(\mathbf{X}) > t | \mathbf{X} \in P) = P(s(\mathbf{X}) > t | \mathbf{X} \in N), \forall t$. This situation corresponds to random classification and the ROC curve becomes the straight line connecting $(0, 0)$ to $(1, 1)$ (the solid line in Fig. 9.7). In this case, the area under the ROC curve (ROC AUC) is 0.5.

Now suppose there is a t_0 such that $P(s(\mathbf{X}) > t_0 | \mathbf{X} \in P) = 1$ and $P(s(\mathbf{X}) > t_0 | \mathbf{X} \in N) = 0$. This means that the entire masses of $p(s(\mathbf{X}) | \mathbf{X} \in P)$ and $p(s(\mathbf{X}) | \mathbf{X} \in N)$ pdfs are on (t_0, ∞) and $(-\infty, t_0]$ intervals, respectively. This can also be interpreted as perfect classification. This situation leads to the point $(0, 1)$ for the ROC curve. At the same time, it is readily seen that as t increases in (t_0, ∞) , fpr remains 0 but tpr decreases until it approaches 0. On the other hand, for decreasing t in the range $(-\infty, t_0]$, tpr remains 1 while fpr increases until it approaches 1. This is the dashed ROC curve in Fig. 9.7. In this case, the ROC AUC is 1.

To estimate the ROC curve, we can first rank observations in a sample based on their scores given by the classifier, and then changing the threshold of classifier from ∞ to $-\infty$, and plot a curve of tpr against fpr . Once the ROC curve is generated, the ROC AUC is basically the area under the ROC curve. It can be shown that the ROC AUC is equivalent to the probability that a randomly chosen member of the positive class is ranked higher (the score is larger) than a randomly chosen member of the negative class. This probabilistic interpretation of ROC AUC leads to an efficient way to estimate the ROC AUC (denoted as \hat{auc}) as follows (Hand and Till, 2001):

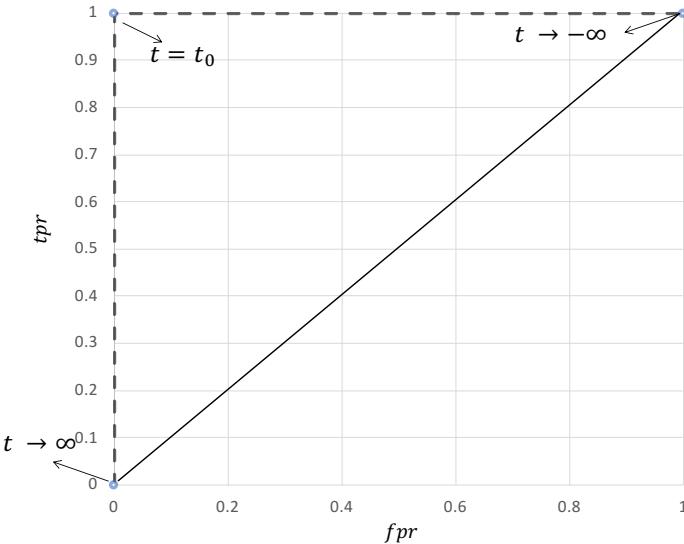


Fig. 9.7: The ROC curve for random (solid) and perfect (dashed) classification.

$$\hat{auc} = \frac{R_N - 0.5(n_N(n_N + 1))}{n_N n_P}, \quad (9.19)$$

where R_N is the sum of ranks of observations from the N class. Given n_P and n_N , the maximum of \hat{auc} is achieved for maximum R_N , which happens when the maximum score given to any observation from the N class is lower than the minimum score given to any observation from the P class. In other words, there exists a threshold t that “perfectly” discriminates instances of class P from class N. In this case, $R_N = \sum_{i=1}^{n_N} (n_P + i) = n_P n_N + 0.5(n_N(n_N + 1))$, which leads to $\hat{auc} = 1$.

Example 9.2 The following table shows the scores given to 10 observations by a classifier. The table also shows the true class of each observations and their ranks based on the given scores. We would like to generate the ROC curve and also find the \hat{auc} .

rank	true class	score
1	P	0.95
2	P	0.80
3	P	0.75
4	N	0.65
5	N	0.60
6	P	0.40
7	N	0.30
8	P	0.25
9	N	0.20
10	N	0.10

The ROC curve is shown in Fig. 9.8. As we lower down the threshold, we mark some of the thresholds that lead to the points on the curve.

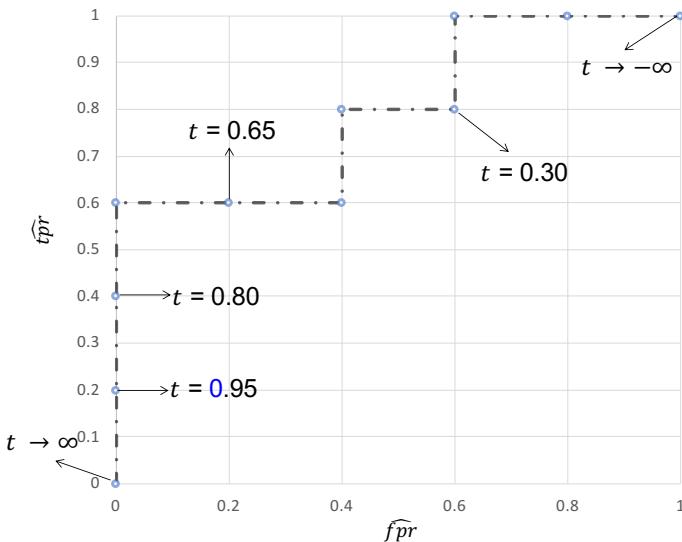


Fig. 9.8: The ROC curve for Example 9.2.

The area under the ROC curve is $auc = 0.6 \times 0.4 + 0.8 \times 0.2 + 1 \times 0.4 = 0.8$. This can also be obtained from (9.19) without plotting the ROC curve. From the table, we have $R_N = 4 + 5 + 7 + 9 + 10 = 35$, $n_N = 5$, and $n_P = 5$. Therefore, $auc = \frac{35 - 0.5(5 \times 6)}{25} = 0.8$. ■

Extension to Multiclass Classification: Extending accuracy or error rate to multiclass classification is straightforward. They can be defined as before by the probability of correct or incorrect classification. As their definition is symmetric across all classes, their empirical estimates can also be obtained by counting the number

of correctly/incorrectly classified observations divided by the total number of observations. Other metrics presented for binary classification can be also extended to multiclass classification ($c > 2$ classes) by some sort of averaging over a series of *one-versus-rest* (*OvR*), also known one-versus-all (*OvA*), classification problems. In this regard, one successively perceives each class as P and all the rest as N, obtain the metric of performance for the binary classification problem, and then obtain the average metric across all binary problems. Here, we present the averaging rules in connection with the empirical estimate of tpr . Other metrics can be obtained similarly.

Micro averaging: In this strategy, TP, FP, TN, and FN are first counted over for all c binary problems. Let n_{TP_i} , n_{FP_i} , n_{TN_i} , and n_{FN_i} denote each of these counts for a binary classification problem in which class $i = 0, 1, \dots, c - 1$ is considered P. Then,

$$\hat{tpr}_{\text{micro}} = \frac{\sum_{i=0}^{c-1} n_{TP_i}}{\sum_{i=0}^{c-1} m_i} = \frac{\sum_{i=0}^{c-1} n_{TP_i}}{\sum_{i=0}^{c-1} (n_{TP_i} + n_{FN_i})}. \quad (9.20)$$

where m_i is the number of observations in class i .

Macro averaging: In this strategy, the metric is calculated for each binary classification problem and then combined by

$$\hat{tpr}_{\text{macro}} = \frac{1}{c} \sum_{i=0}^{c-1} \hat{tpr}_i, \quad (9.21)$$

where \hat{tpr}_i is the \hat{tpr} for the binary classification problem where class $i = 0, 1, \dots, c - 1$ is considered P. As can be seen, in this averaging classes are given the same weight regardless of the class-specific sample size.

Weighted averaging: Here, each class-specific metric is weighted by the class sample size. Therefore,

$$\hat{tpr}_{\text{weighted}} = \sum_{i=0}^{c-1} \frac{m_i}{\sum_{i=0}^{c-1} m_i} \hat{tpr}_i = \sum_{i=0}^{c-1} \frac{n_{TP_i} + n_{FN_i}}{\sum_{i=0}^{c-1} (n_{TP_i} + n_{FN_i})} \hat{tpr}_i. \quad (9.22)$$

Scikit-learn implementation of evaluation metrics of classifiers: In scikit-learn, metrics are implemented as different classes in `sklearn.metrics` module. A metric such as accuracy and confusion matrix can be calculated using the actual and the predicted labels for the sample. Therefore, to implement them, we only need to import their classes and pass these two sets of labels as arguments:

- For accuracy: use `accuracy_score(y_true, y_pred)`
- For confusion matrix: use `confusion_matrix(y_true, y_pred)`

Other metrics such as recall, precision, and F_1 score are defined for the positive class (in case of multiclass, as mentioned before, OvR can be used to successively treat each class as positive). Therefore, to use their scikit-learn classes, we should pass actual labels and predicted labels, and can pass the label of the positive class by `pos_label` parameter (in case of binary classification, `pos_label=1` by default):

- For recall: use `recall_score(y_true, y_pred)`
- For precision: use `precision_score(y_true, y_pred)`
- For F_1 score: use `f1_score(y_true, y_pred)`

For multiclass classification, `recall_score`, `precision_score`, and `f1_score` support micro, macro, and weighted averaging. To choose a specific averaging, the `average` parameter can be set to either '`'micro'`', '`'macro'`', or '`'weighted'`'.

Last but not least, to compute the ROC AUC, we need the actual labels and the given scores. Classifiers in scikit-learn have either `decision_function` method, `predict_proba`, or both. The `decision_function` returns scores given to observations; for example, for linear classifiers a score for an observation is proportional to the signed distance of that observation from the decision boundary, which is a hyperplane. As the result, the output is just a 1-dimensional array. On the other hand, `predict_proba` method is available when the classifier can estimate the probability of a given observation belonging to a class. Therefore, the output is a 2-dimensional array of size, sample size \times number of classes, where sum of each row over columns becomes naturally 1. For ROC AUC calculation, we can use:

- `roc_auc_score(y_true, y_score)`

`y_score`: for binary classification, this could be the output of `decision_function` or `predict_proba`. In the multiclass case, it should be a probability array of shape sample size \times number of classes. Therefore, it could be output of `predict_proba` only.

For multiclass classification, `roc_auc_score` supports macro, weighted, and micro averaging (the latter in scikit-learn version 1.2.0). This is done by setting the `average` to either '`'macro'`', '`'weighted'`', or '`'micro'`'.

Example 9.3 Suppose we have the following actual and predicted labels identified by `y_true` and `y_pred`, respectively,

```
y_true = [2, 3, 3, 3, 2, 2, 2]
y_pred = [3, 2, 2, 3, 2, 2, 2]
```

where label 2 is the positive class. In that case, $n_{TP} = 3$, $n_{FP} = 2$ and from (9.8), $\text{precision}=0.6$. We can compute this metrics as follows (note that we explicitly set `pos_label=2`; otherwise, as this is a binary classification problem by default it uses `pos_label=1`, which is not a valid label and leads to error):

```
sklearn.metrics import precision_score
y_true = [2, 3, 3, 3, 2, 2, 2]
```

```
y_pred = [3, 2, 2, 3, 2, 2, 2]
precision_score(y_true, y_pred, pos_label=2)
```

0.6

Example 9.4 In the following example, we estimate several metrics in the Iris classification problem both for the macro averaging and for each class individually (i.e., when each class is considered as the positive class):

```
import numpy as np
from sklearn import datasets
from sklearn.metrics import accuracy_score, confusion_matrix,
    recall_score, precision_score, f1_score, roc_auc_score,
    classification_report
from sklearn.linear_model import LogisticRegression as LRR

iris = datasets.load_iris()
X_train = iris.data
y_train = iris.target
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.
    target, random_state=100, test_size=0.2, stratify=iris.target)
print('X_train_shape: ' + str(X_train.shape) + '\nX_test_shape: ' +
    str(X_test.shape) +
    '\ny_train_shape: ' + str(y_train.shape) + '\ny_test_shape: ' +
    str(y_test.shape) + '\n')

lrr = LRR(C=0.1)
y_test_pred = lrr.fit(X_train, y_train).predict(X_test)

print("Accuracy = {:.3f}".format(accuracy_score(y_test, y_test_pred)))
print("Confusion Matrix is\n {}".format(confusion_matrix(y_test,
    y_test_pred)))
print("Macro Average Recall = {:.3f}".format(recall_score(y_test,
    y_test_pred, average='macro')))
print("Macro Average Precision = {:.3f}".format(precision_score(y_test,
    y_test_pred, average='macro')))
print("Macro Average F1 score = {:.3f}".format(f1_score(y_test,
    y_test_pred, average='macro')))
print("Macro Average ROC AUC = {:.3f}".format(roc_auc_score(y_test, lrr.
    predict_proba(X_test), multi_class='ovr', average='macro')))

# set "average" to "None" to print these metrics for each class
# individually (when the class is considered positive)
np.set_printoptions(precision=3)
print("Class-specific Recall = {}".format(recall_score(y_test,
    y_test_pred, average=None)))
print("Class-specific Precision = {}".format(precision_score(y_test,
    y_test_pred, average=None)))
```

```

print("Class-specific F1 score = {}".format(f1_score(y_test,_
    ↵y_test_pred, average=None)))
print(classification_report(y_test, y_test_pred)) #_
    ↵classification_report is used to print them out nicely

```

```

X_train_shape: (120, 4)
X_test_shape: (30, 4)
y_train_shape: (120,)
y_test_shape: (30,)

Accuracy = 0.967
Confusion Matrix is
[[10  0  0]
 [ 0 10  0]
 [ 0  1  9]]
Macro Average Recall = 0.967
Macro Average Precision = 0.970
Macro Average F1 score = 0.967
Macro Average ROC AUC = 0.997
Class-specific Recall = [1.  1.  0.9]
Class-specific Precision = [1.      0.909 1.      ]
Class-specific F1 score = [1.      0.952 0.947]
precision      recall      f1-score      support
          0        1.00      1.00      1.00      10
          1        0.91      1.00      0.95      10
          2        1.00      0.90      0.95      10
accuracy                          0.97      30
macro avg                      0.97      0.97      0.97      30
weighted avg                    0.97      0.97      0.97      30

```

■ In order to use one of the aforementioned metrics along with cross-validation, we can set the `scoring` parameter of `cross_val_score` to a string that represents the metric. In this regard, legitimate strings are listed at ([Scikit-metrics, 2023](#)) and can be seen as follows as well:

```

from sklearn.metrics import SCORERS
SCORERS.keys()

```

```
dict_keys(['explained_variance', 'r2', 'max_error',
        'neg_median_absolute_error', 'neg_mean_absolute_error',
        'neg_mean_absolute_percentage_error', 'neg_mean_squared_error',
        'neg_mean_squared_log_error', 'neg_root_mean_squared_error',
        'neg_mean_poisson_deviance', 'neg_mean_gamma_deviance', 'accuracy',
        'top_k_accuracy', 'roc_auc', 'roc_auc_ovr', 'roc_auc_ovo',
        'roc_auc_ovr_weighted', 'roc_auc_ovo_weighted', 'balanced_accuracy',
        'average_precision', 'neg_log_loss', 'neg_brier_score',
        'adjusted_rand_score', 'rand_score', 'homogeneity_score',
        'completeness_score', 'v_measure_score', 'mutual_info_score',
        'adjusted_mutual_info_score', 'normalized_mutual_info_score',
        'fowlkes_mallows_score', 'precision', 'precision_macro',
        'precision_micro', 'precision_samples', 'precision_weighted',
        'recall', 'recall_macro', 'recall_micro', 'recall_samples',
        'recall_weighted', 'f1', 'f1_macro', 'f1_micro', 'f1_samples',
        'f1_weighted', 'jaccard', 'jaccard_macro', 'jaccard_micro',
        'jaccard_samples', 'jaccard_weighted'])
```

For example, here we use '`roc_auc_ovr`' to calculate stratified 3-fold cross-validation (with shuffling) ROC AUC in a OvR fashion for Iris classification dataset:

```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import StratifiedKFold
K_fold = 3
strkfolds = StratifiedKFold(n_splits=K_fold, shuffle=True,
                           random_state=42)
lrr = LRR(C=0.1)
cv_scores = cross_val_score(lrr, X_train, y_train, cv=strkfolds,
                           scoring='roc_auc_ovr')
cv_scores
```

```
array([0.98324515, 0.99905033, 0.99810066])
```

Scikit-learn implementation to monitor multiple evaluation metrics in cross-validation: In evaluating a predictive model, it is often desired to monitor multiple evaluation metrics. Insofar as cross-validation rule is concerned, `cross_val_score` does not allow us to monitor multiple metrics through its `scoring` parameter. To monitor several scoring metrics, we can use `cross_validate` class from `sklearn.model_selection` module. Then one way to compute various metrics is to set the `scoring` parameter of `cross_validate` to a list of strings that represents metrics of interest, for example,

```
cross_validate(..., scoring=['accuracy', 'roc_auc'])
```

However, a more flexible way is to define a function with signature

```
arbitrary_name(estimator, X, y)
```

that returns a dictionary containing multiple scores and pass that through the `scoring` parameter of `cross_validate` (see the following example).

Example 9.5 Here we would like to use cross-validation to evaluate and record the performance of logistic regression in classifying two classes in Iris classification dataset. As the performance metrics, we use the accuracy, the ROC AUC, and the confusion matrix.

```

import pandas as pd
import numpy as np
from sklearn import datasets
from sklearn.metrics import roc_auc_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import cross_validate
from sklearn.model_selection import StratifiedKFold
from sklearn.linear_model import LogisticRegression as LRR

def accuracy_roc_confusion_matrix(clf, X, y):
    y_pred = clf.predict(X)
    acc = accuracy_score(y, y_pred)
    mat = confusion_matrix(y, y_pred)
    y_score = clf.decision_function(X)
    auc = roc_auc_score(y, y_score)
    return {'accuracy': acc,
            'roc_auc': auc,
            'tn': mat[0, 0],
            'fp': mat[0, 1],
            'fn': mat[1, 0],
            'tp': mat[1, 1]}

iris = datasets.load_iris()
X_train = iris.data[iris.target!=0,:]
y_train = iris.target[iris.target!=0]
print('X_train_shape: ' + str(X_train.shape) + '\ny_train_shape: ' +
      str(y_train.shape) + '\n')
lrr = LRR()
strkfolds = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
cv_scores = cross_validate(lrr, X_train, y_train, cv=strkfolds,
                           scoring=accuracy_roc_confusion_matrix)
df = pd.DataFrame(cv_scores)
display(df)
print("the overall 5-fold CV accuracy is: {:.3f}".
      format(cv_scores['test_accuracy'].mean()))
print("the overall 5-fold CV roc auc is: {:.3f}".
      format(cv_scores['test_roc_auc'].mean()))
print("the aggregated confusion matrix is:\n {}".format(np.array(df.
      sum(axis=0)[4:], dtype='int_').reshape(2,2)))

```

```
X_train_shape: (100, 4)
y_train_shape: (100,)
```

	fit_time	score_time	test_accuracy	test_roc_auc	test_tn	test_fp	\
0	0.014002	0.004842	0.90	1.00	8	2	
1	0.006391	0.001709	1.00	1.00	10	0	
2	0.008805	0.002197	0.90	0.96	9	1	
3	0.010306	0.001830	0.95	1.00	10	0	
4	0.008560	0.002558	1.00	1.00	10	0	
	test_fn	test_tp					
0	0	10					
1	0	10					
2	1	9					
3	1	9					
4	0	10					

the overall 5-fold CV accuracy is: 0.950

the overall 5-fold CV roc auc is: 0.992

the aggregated confusion matrix is:

```
[[47  3]
 [ 2 48]]
```

9.1.3 Evaluation Metrics for Regressors

Let $f(\mathbf{X})$ be a trained regressor, which estimates Y for a given realization of \mathbf{X} . To discuss the empirical estimates of evaluation metrics of $f(\mathbf{X})$, we assume the trained regressor is applied to a sample of size m , $\mathbf{S}_{te,m} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_m, y_m)\}$ and estimates the target values of $\mathbf{x}_1, \dots, \mathbf{x}_m$ as $\hat{y}_1, \dots, \hat{y}_m$, respectively. Our aim is to estimate the predictive ability of the regressor. We discuss three metrics and their estimators, namely, MSE, MAE, and R^2 , also known as coefficient of determination.

Mean Square Error (MSE): The MSE of $f(\mathbf{X})$ is defined as

$$\text{MSE} = E_{Y,\mathbf{X}} \left[(Y - f(\mathbf{X}))^2 \right], \quad (9.23)$$

where the expectation is with respect to the joint distribution of Y and \mathbf{X} . The empirical estimate of MSE, denoted $\hat{\text{MSE}}$, is given by

$$\hat{\text{MSE}} = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2. \quad (9.24)$$

The summation is also known as the residual sum of squares (RSS), which we have seen previously in Section 5.2.2; that is,

$$\text{RSS} = \sum_{i=1}^m (y_i - \hat{y}_i)^2. \quad (9.25)$$

In other words, $\hat{\text{MSE}}$ is the normalized RSS with respect to the sample size.

Mean Absolute Error (MAE): To define the MAE, the squared loss in MSE is replaced with the absolute loss; that is to say,

$$\text{MAE} = E_{Y,\mathbf{X}}[|Y - f(\mathbf{X})|]. \quad (9.26)$$

The estimate of MAE, denoted $\hat{\text{MAE}}$, is obtained as

$$\hat{\text{MAE}} = \frac{1}{m} \sum_{i=1}^m |y_i - \hat{y}_i|. \quad (9.27)$$

R^2 (coefficient of determination): R^2 is perhaps the most common metric used for regressor evaluation to the extent that is used as the metric of choice in `score` method of any regressor in scikit-learn. Formally, (for the squared loss) it is given by (Dougherty et al., 2000):

$$R^2 = \frac{E[(Y - E[Y])^2] - E_{Y,\mathbf{X}}[(Y - f(\mathbf{X}))^2]}{E[(Y - E[Y])^2]}. \quad (9.28)$$

As $E[Y]$ is the optimal constant estimator of Y , R^2 measured the relative decrease in error by using regressor $f(\mathbf{X})$ instead of using the trivial constant regressor $E[Y]$. Plug-in estimator of R^2 , denoted \hat{R}^2 , is obtained by replacing $E_{Y,\mathbf{X}}[(Y - f(\mathbf{X}))^2]$ and $E[(Y - E[Y])^2]$ with their empirical estimates, which yields,

$$\hat{R}^2 = 1 - \frac{\text{RSS}}{\text{TSS}}, \quad (9.29)$$

where RSS is given in (9.25) and

$$\text{TSS} = \sum_{i=1}^m (y_i - \bar{y})^2, \quad (9.30)$$

where $\bar{y} = \frac{1}{m} \sum_{i=1}^m y_i$. Some points to remember:

- R^2 is a scoring function (the higher, the better), while MSE and MAE are loss functions (the lower, the better).
- R^2 is a relative measure, while MSE and MAE are absolute measures; that is, given a value of MSE, it is not possible to judge whether the model has a good

performance or not unless extra information on the scale of the target values is given. This is also seen from (9.24). If we change the scale of target values and their estimates by multiplying y_i and \hat{y}_i by a constant c , MSE is multiplied by c^2 . In contrast, this change of scale does not have an impact on the \hat{R}^2 .

Scikit-learn implementation of evaluation metrics of regressors: Similar to classification metrics, regression metrics are also defined in `sklearn.metrics` module:

- For \hat{R}^2 : use `r2_score(y_true, y_pred)`
- For MSE : use `mean_squared_error(y_true, y_pred)`
- For MAE : use `mean_absolute_error(y_true, y_pred)`

Similar to classification metrics, we can estimate these metrics with cross-validation by setting the `scoring` parameter of `cross_val_score` to the corresponding string that can be found from `SCORERS.keys()` (also listed at ([Scikit-metrics, 2023](#))). In this regard, an important point to remember is that cross-validation implementation in scikit-learn accepts a scoring function, and not a loss function. However, as we said, MSE and MAE are loss functions. That is the reason that legitimate “scorers” retrieved by `SCORERS.keys()` include, for example, `'neg_mean_squared_error'` and `'neg_mean_absolute_error'`. Setting the `scoring` parameter of `cross_val_score` to `'neg_mean_squared_error'` internally multiplies the MSE by -1 to obtain a score (the higher, the better). Therefore, if needed, we can use `'neg_mean_squared_error'` and then multiply the returned cross-validation score by -1 to obtain the MSE loss, which is non-negative.

9.2 Model Selection

The model selection stage is a crucial step towards constructing successful predictive models. Oftentimes, practitioners are given a dataset and are asked to construct a classifier or regressor. In such a situation, the practitioner has the option to choose from a plethora of models and algorithms. The practices that the practitioner follows to *select* one final trained model that can be used in practice from such a vast number of candidates are categorized under *model selection*.

From the perspective of the three-stage design process discussed in Section [1.1.6](#), the model selection stage includes the initial candidate set selection (Step 1) and selecting the final mapping (Step 3). For example, it is a common practice to initially narrow down the number of plausible models based on

- practitioner’s prior experience with different models and their sample size-dimensionality requirements;
- the nature of data at hand (for example, having time-series, images, or other types); and
- if available, based on prior knowledge about the field of study from which the data has been collected. This stage *ipso facto* is part of model selection, albeit based on prior knowledge.

After this stage, the practitioner is usually faced with one or a few learning algorithms. However, each of these learning algorithms generally has one or a few hyperparameters that should be estimated. We have already seen hyperparameters in the previous sections. These are parameters that are not estimated/given by the learning algorithm using which a classifier/regressor is trained but, at the same time, can significantly change the performance of the predictive model. For example, k in k NN (Chapter 5), C or ν in various forms of regularized logistic regression (Section 6.2.2), minimum sample per leaf, maximum depth, or even the choice of impurity criterion in CART (Section 7.2.2 and 7.2.4), or the number of estimators or the choice of base estimators and all its hyperparameters in using AdaBoost (Section 8.6.1). Training a final model involves using a learning algorithm as well as estimating its hyperparameters—a process, which is known as hyperparameter tuning and is part of model selection. Two common data-driven techniques for model selection are *grid search* and *random search*.

9.2.1 Grid Search

In grid search, we first define a space of hyperparameters for a learning algorithm Ψ and conduct an exhaustive search within this space to determine the “best” values of hyperparameters. In this regard,

- 1) for continuous-valued hyperparameters we assume and discretize a range depending on our prior experiences/knowledge and/or computational capabilities and resources;
- 2) assume a range for discrete/categorical hyperparameters;
- 3) define the discrete hyperparameter search space (denoted by \mathcal{H}_Ψ^D) as the Cartesian product of sets of values defined for each hyperparameter; and
- 4) for each point in \mathcal{H}_Ψ^D , estimate the performance of the corresponding model and pick the hyperparameter values that lead to the best performance.

The aforementioned procedure can be extended to as many as learning algorithms to determine the best combination of learning algorithm-hyperparameter values.

Example 9.6 Consider AdaBoost.SAMME algorithm presented in Section 8.6.1. Suppose based on prior knowledge and experience, we decide to merely use CART as the base classifier. In that case, one possible choice is to treat η (the learning rate), M (the maximum number of base classifiers), the minimum samples per leaf (denoted s), and the choice of impurity criterion as hyperparameters. Based on our prior experience we decide:

$\eta \in \{0.1, 1, 10\}$, $M \in \{10, 50\}$, $s \in \{5, 10\}$, criterion $\in \{\text{gini, entropy}\}$
and use the default values of scikit-learn as for the values of other hyperparameters.
Determine \mathcal{H}_Ψ and its cardinality.

$$\begin{aligned}\mathcal{H}_{\Psi}^D &= \{0.1, 1, 10\} \times \{10, 50\} \times \{5, 10\} \times \{\text{gini, entropy}\} = \\ &\{(0.1, 10, 5, \text{gini}), (1, 10, 5, \text{gini}), (10, 10, 5, \text{gini}), \dots, (10, 50, 10, \text{entropy})\}.\end{aligned}$$

Therefore, $|\mathcal{H}_{\Psi}^D| = 3 \times 2 \times 2 \times 2 = 24$, where $|.|$ denotes the cardinality. ■

Suppose after initial inspection (Step 1 of the design process), T learning algorithms, denoted $\Psi_i, i = 1, \dots, T$, have been chosen. For each Ψ_i , we specify a discrete hyperparameter space $\mathcal{H}_{\Psi_i}^D$. Depending on the estimation rules that are used to estimate the metric of performance, there are two types of grid search that are commonly used in practice:

- grid search using validation set
- grid search using cross-validation

Hereafter, we denote a predictive model (classifier/regressor) constructed by applying the learning algorithm Ψ_i with a specific set of hyperparameters $\theta \in \mathcal{H}_{\Psi_i}^D$ on a training sample \mathbf{S}_1 by $\psi_{\theta, i}(\mathbf{S}_1)$. Furthermore, let $\eta_{\psi_{\theta, i}(\mathbf{S}_1)}(\mathbf{S}_2)$ denote the value of a scoring (the higher, the better) metric η obtained by evaluating $\psi_{\theta, i}(\mathbf{S}_1)$ on a sample \mathbf{S}_2 .

Grid search using validation set: In our discussions in the previous chapters, we have generally split a given sample into a training and a test set to train and evaluate an estimator, respectively. To take into account the model selection stage, a common strategy is to split the data into three sets known as training, validation, and test sets. The extra set, namely, the *validation set*, is then used for model selection. In this regard, one can first split the entire data into two smaller sets, for example, with a size of 75% and 25% of the given sample, and use the smaller set as test set. The larger set (i.e., the set of size 75%) is then split into a “training set” and a “validation set” (we can use again the same ratio of 75% and 25% to generate the training and validation sets).

Let \mathbf{S}_{tr} , \mathbf{S}_{va} , and \mathbf{S}_{te} denote the training, validation, and test sets, respectively. Furthermore, let $\Psi_{\theta, va}^*$ denote the optimal learning algorithm-hyperparameter values estimated using the grid search on validation set. We have

$$\Psi_{\theta, va}^* = \underset{\Psi_i, i=1,2,\dots,T}{\operatorname{argmax}} \eta_{\psi_{\theta^*, i}(\mathbf{S}_{tr})}(\mathbf{S}_{va}), \quad (9.31)$$

where

$$\theta_i^* = \underset{\theta \in \mathcal{H}_{\Psi_i}^D}{\operatorname{argmax}} \eta_{\psi_{\theta, i}(\mathbf{S}_{tr})}(\mathbf{S}_{va}), \quad i = 1, 2, \dots, T. \quad (9.32)$$

Although (9.31) and (9.32) can be combined in one equation, we have divided them into two equations to better describe the underlying selection process. Let us elaborate a bit on these equations. In (9.32), θ_i^* , which is the estimate of the optimal values of hyperparameter for each learning algorithm $\Psi_i, i = 1, 2, \dots, T$, is obtained. In this regard, for each $\theta \in \mathcal{H}_{\Psi_i}^D$, Ψ_i is trained on \mathbf{S}_{tr} to construct the classifier $\psi_{\theta, i}(\mathbf{S}_{tr})$. Then this model is evaluated on the validation set \mathbf{S}_{va} to obtain

$\eta_{\psi_{\theta,i}(\mathbf{S}_{tr})}(\mathbf{S}_{va})$. To estimate the optimal values of hyperparameters for Ψ_i , we then compare all scores obtained for each $\theta \in \mathcal{H}_{\Psi_i}^D$. At the end of (9.32), we have T sets of hyperparameter values θ_i^* (one for each Ψ_i). The optimal learning algorithm-hyperparameter values is then obtained by (9.31) where the best scores of different learning algorithms are compared and the one with maximum score is chosen. There is no need to compute any new score in (9.31) because all required scores used in the comparison have been already computed in (9.32).

Once $\Psi_{\theta,va}^*$ is determined from (9.31), we can apply it to \mathbf{S}_{tr} to train our “final” model. Nevertheless, this model has already been trained and used in (9.31), which means that we can just pick it if it has been stored. Alternatively, we can combine \mathbf{S}_{tr} and \mathbf{S}_{va} to build a larger set ($\mathbf{S}_{tr} \cup \mathbf{S}_{va}$) that is used in training one final model. In this approach, the final model is trained by applying the estimated optimal learning algorithm-hyperparameter values $\Psi_{\theta,va}^*$ to $\mathbf{S}_{tr} \cup \mathbf{S}_{va}$. There is nothing wrong with this strategy (as a matter of fact this approach more closely resembles the grid search using cross-validation that we will see next). We just try to utilize the full power of available data in training. With a “sufficient” sample size and well-behaved learning algorithms, we should not generally see a worse performance.

Once the final model is selected/trained, it can be applied on \mathbf{S}_{te} for evaluation. Some remarks about this grid search on validation set:

- \mathbf{S}_{va} simulates the effect of having test data within $\mathbf{S}_{tr} \cup \mathbf{S}_{va}$. This means that if in the process of model selection a preprocessing stage is desired, it should be applied to \mathbf{S}_{tr} , and \mathbf{S}_{va} (similar to \mathbf{S}_{te}) should be normalized using statistics found based on \mathbf{S}_{tr} . If we want to train the final model on $\mathbf{S}_{tr} \cup \mathbf{S}_{va}$, another normalization should be applied to $\mathbf{S}_{tr} \cup \mathbf{S}_{va}$ and then \mathbf{S}_{te} should be normalized bases on statistics found on $\mathbf{S}_{tr} \cup \mathbf{S}_{va}$.
- One potential problem associated with grid search using validation set is that the estimate of the optimal learning algorithm-hyperparameter values obtained from (9.31) could be biased to the single sample chosen for validation set. Therefore, if major probabilistic differences exist between the single validation set from the test data, the final constructed model may not generalize well and shows poor performance on unseen data. This problem is naturally more pronounced in small sample where the model selection could become completely biased towards the specific observations that are in validation set. Therefore, this method of model selection is generally used in large sample.

Grid search using cross-validation: In grid search using cross-validation, we look for the combination of the learning algorithm-hyperparameter values that leads to the highest score evaluated using cross-validation on \mathbf{S}_{tr} . For a K -fold CV, this is formally represented as:

$$\Psi_{\theta, cv}^* = \underset{\Psi_i, i=1,2,\dots,T}{\operatorname{argmax}} \eta_{\psi_{\theta_i^*, i}}^{K-fold}, \quad (9.33)$$

where

$$\theta_i^* = \operatorname{argmax}_{\theta \in \mathcal{H}_{\Psi_i}^D} \eta_{\psi_{\theta, i}}^{K-fold}, \quad i = 1, 2, \dots, T, \quad (9.34)$$

and where

$$\eta_{\psi_{\theta, i}}^{K-fold} = \frac{1}{K} \sum_{k=1}^K \eta_{\psi_{\theta, i}(\mathbf{S}_{tr}-\text{Fold}_k)}(\text{Fold}_k), \quad (9.35)$$

in which Fold_k denote the set of observations that are in the k^{th} fold of CV. Once $\Psi_{\theta, cv}^*$ is determined, a final model is then trained using the estimated optimal learning algorithm-hyperparameter values $\Psi_{\theta, cv}^*$ on \mathbf{S}_{tr} . Once this final model is trained, it can be applied on \mathbf{S}_{te} for evaluation.

Scikit-learn implementation of grid search model selection: To implement the grid search using cross-validation, we can naturally write a code that goes over the search space and use `cross_val_score` to compute the score for each point in the grid. Nevertheless, scikit-learn has a convenient way to implement the grid search. For this purpose, we can use `GridSearchCV` class from `sklearn.model_selection` module. A few important parameters of this class are:

- 1) `estimator`: this is the object from the estimator class;
- 2) `param_grid`: this could be either a dictionary to define a grid or a list of dictionaries to define multiple grids. Either way, the keys in a dictionary should be parameters of the estimator that are used in model selection. The possible values of each key used in the search are passed as a list of values for that key. For example, the following code defines two grids to search for hyperparameter tuning in logistic regression:

```
grids = [{{'penalty': ['l2'], 'C': [0.01, 0.1]},  
         {'penalty': ['elasticnet'], 'C': [0.01, 0.1], 'l1_ratio':[0.2, 0.8]},  
         {'solver':['saga']}]}]
```

- 3) `cv`: determines the CV strategy (similar to the same parameter used in `cross_val_score`; See Section 9.1.1 for more details)
- 4) `scoring`: in its most common use case, we only pass the name of a scoring function (a string). However, as discussed before for `cross_validate`, multiple metrics can also be used by a list of strings or a callable function returning a dictionary. If multiple metrics are used here, then the `refit` method should be set to the metric that is used to find the best hyperparameter values for training the final estimator (see next).
- 5) `refit`: by default this parameter is `True`. This means once all scores are determined, then it trains (“refit”) an estimator using the best found combination of hyperparameters (led to the highest CV score) on the entire dataset. If multiple

metrics are used in `scoring`, this parameter should be a string that identifies the metric used for refitting.

Once an instance of class `GridSearchCV` is created, we can treat it as a regular classifier/regressor as it implements `fit`, `predict`, `score` and several other methods. Here `fit` trains an estimator *for each given combination of hyperparameters*. The `predict` and `score` methods can be used for prediction and evaluation using *the best found estimator*, respectively. Nevertheless, an object from `GridSearchCV` class has also some important attributes such as:

- 1) `cv_results_`: it is a dictionary containing the scores obtained on each fold for each hyperparameter combination as well as some other information regarding training (e.g., training time);
- 2) `best_params_`: the combination of hyperparameters that led to the highest score;
- 3) `best_score_`: the highest score obtained on the given grids; and
- 4) `best_estimator_`: not available if `refit=False`; otherwise, this attribute gives access to the estimator trained on the entire dataset using the best combination of hyperparameters—`predict` and `score` methods of `GridSearchCV` use this estimator as well.

Example 9.7 Let us consider the Iris classification dataset again. We wish to use `GridSearchCV` class to tune hyperparameters of logistic regression over a hyperparameter space defined using the same grids as specified in the previous page. We monitor both the accuracy and the weighted ROC AUC during the cross validation but we use the accuracy for hyperparameter tuning. Once the hyperparameter tuning is completed, we also find the test accuracy (accuracy on the test data) for the best model trained on the entire training data using the hyperparameter combination that led to the highest CV accuracy. As for the cross-validation, we use stratified 3-fold CV with shuffling data before splitting (all these tuning, training, and testing steps can be implemented in a few lines of codes using scikit-learn!)

```
import pandas as pd
from sklearn import datasets
from sklearn.linear_model import LogisticRegression as LRR
from sklearn.model_selection import GridSearchCV, StratifiedKFold,
    train_test_split

iris = datasets.load_iris()
X_train = iris.data
y_train = iris.target
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.
    target, random_state=100, test_size=0.2, stratify=iris.target)
print('X_train_shape: ' + str(X_train.shape) + '\nX_test_shape: ' +
    str(X_test.shape) +
    '\ny_train_shape: ' + str(y_train.shape) + '\ny_test_shape: ' +
    str(y_test.shape) + '\n')
lrr = LRR(max_iter=2000)
```

```

grids = [{{'penalty': ['l2'], 'C': [0.01, 0.1]},  

          {'penalty': ['elasticnet'], 'C': [0.01, 0.1], 'l1_ratio':[0.2, 0.8], 'solver':['saga']}]}  

strkfold = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)  

gscv = GridSearchCV(lrr, grids, cv=strkfold, n_jobs=-1, scoring = ['accuracy', 'roc_auc_ovr_weighted'], refit='accuracy')  

score_best_estimator=gscv.fit(X_train, y_train).score(X_test, y_test)  

print('the highest score is: {:.3f}'.format(gscv.best_score_))  

print('the best hyperparameter combination is: {}'.format(gscv.  

    best_params_))  

print('the accuracy of the best estimator on the test data is: {:.3f}'.  

    format(score_best_estimator))  

df = pd.DataFrame(gscv.cv_results_)  

df

```

X_train_shape: (120, 4)

X_test_shape: (30, 4)

y_train_shape: (120,)

y_test_shape: (30,)

the highest score is: 0.950

the best hyperparameter combination is: {'C': 0.1, 'l1_ratio': 0.2, 'penalty':

'elasticnet', 'solver': 'saga'}

the accuracy of the best estimator on the test data is: 0.967

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C
0	0.418733	0.000479	0.003294	0.000478	0.01
1	0.428889	0.001791	0.003241	0.000199	0.1
2	0.003677	0.000962	0.004429	0.000792	0.01
3	0.001299	0.000221	0.002921	0.000350	0.01
4	0.016373	0.001285	0.003531	0.001409	0.1
5	0.014243	0.003561	0.005710	0.000220	0.1

	param_penalty	param_l1_ratio	param_solver
0	12	NaN	NaN
1	12	NaN	NaN
2	elasticnet	0.2	saga
3	elasticnet	0.8	saga
4	elasticnet	0.2	saga
5	elasticnet	0.8	saga

[the rest of the output is omitted for brevity]



To implement the grid-search using validation set, we can first use the function `train_test_split` from `sklearn.model_selection` to set aside the test set,

if desired, and then use it once again to set aside the validation set. The indices of the training and validation sets can be then created by using some cross validation generator such as `PredefinedSplit` class from `sklearn.model_selection` and used as `cv` parameter of `GridSearchCV`, which accepts generators. To use `PredefinedSplit`, we can set indices of validation set to an integer such as 0 but the indices of the training set should be set to -1 to be excluded from validation set (see the following example).

Example 9.8 In this example, we use the same data, classifier, and hyperparameter search space as in Example 9.7; however, instead of grid search cross-validation, we use validation set.

```
import pandas as pd
from sklearn import datasets
from sklearn.linear_model import LogisticRegression as LRR
from sklearn.model_selection import GridSearchCV, StratifiedKFold,
    train_test_split, PredefinedSplit

iris = datasets.load_iris()
X, X_test, y, y_test = train_test_split(iris.data, iris.target,
    random_state=100, stratify=iris.target)
indecis = np.arange(len(X))
X_train, X_val, y_train, y_val, indecis_train, indecis_val =
    train_test_split(X, y, indecis, random_state=100, stratify=y)
ps_ind = np.zeros(len(indecis), dtype='int_')
ps_ind[indecis_train] = -1 # training set indices are set to -1 and
# validation set indices are left as 0
print('indices used in CV splitter:\n', ps_ind)
pds = PredefinedSplit(ps_ind)
print('the split used as training and validation sets:\n', *pds.split())
lrr = LRR(max_iter=2000)
grids = [{ 'penalty': ['l2'], 'C': [0.01, 0.1]},
          { 'penalty': ['elasticnet'], 'C': [0.01, 0.1], 'l1_ratio':[0.2,
          0.8], 'solver':['saga']}]
gscv = GridSearchCV(lrr, grids, cv=pds, n_jobs=-1, scoring =
    ['accuracy', 'roc_auc_ovr_weighted'], refit='accuracy')
score_best_estimator=gscv.fit(X, y).score(X_test, y_test) #gscv is fit
#on X = X_train+X_val
print('the highest score is: {:.3f}'.format(gscv.best_score_))
print('the best hyperparameter combination is: {}'.format(gscv.
    best_params_))
print('the accuracy of the best estimator on the test data is: {:.3f}'.
    format(score_best_estimator))
df = pd.DataFrame(gscv.cv_results_)
df
```

indices used in CV splitter:

```
[ -1 -1 -1 -1 -1 -1  0  0  0 -1 -1  0 -1 -1 -1 -1  0 -1 -1 -1  0 -1 -1
-1 -1 -1 -1 -1  0 -1 -1 -1 -1 -1  0  0 -1 -1  0 -1 -1  0 -1 -1 -1 -1]
```

```
0 -1 0 -1 0 -1 -1 -1 0 -1 -1 -1 0 -1 -1 0 -1 -1 -1 0 -1 0 -1
0 -1 0 0 -1 -1 -1 0 -1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 0 -1 -1
-1 -1 -1 -1 -1 -1 0 -1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1]
```

the split used as training and validation sets:

```
(array([ 0,  1,  2,  3,  4,  5,  6, 10, 11, 13, 14, 15, 16,
       18, 19, 20, 22, 23, 24, 25, 26, 27, 28, 30, 31, 32,
       33, 34, 35, 38, 39, 41, 43, 45, 46, 47, 49, 51, 53,
       54, 55, 57, 58, 59, 61, 62, 64, 65, 66, 67, 69, 71,
       73, 76, 77, 78, 80, 82, 83, 84, 85, 86, 87, 88, 89,
       90, 91, 92, 94, 95, 96, 97, 98, 99, 100, 101, 103, 105,
       106, 107, 108, 109, 110, 111]), array([
    7,  8,  9, 12, 17, 21, 29,
   36, 37, 40, 42, 44, 48,
   50, 52, 56, 60, 63, 68, 70, 72, 74, 75, 79, 81, 93,
  102, 104]))
```

the highest score is: 0.893

the best hyperparameter combination is: {'C': 0.1, 'l1_ratio': 0.8, 'penalty':

'elasticnet', 'solver': 'saga'}

the accuracy of the best estimator on the test data is: 0.974

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_C
0	0.223788	0.0	0.002445	0.0	0.01
1	0.015926	0.0	0.002789	0.0	0.1
2	0.002857	0.0	0.002804	0.0	0.01
3	0.001344	0.0	0.003534	0.0	0.01
4	0.016076	0.0	0.002317	0.0	0.1
5	0.009496	0.0	0.002699	0.0	0.1

	param_penalty	param_l1_ratio	param_solver
0	12	NaN	NaN
1	12	NaN	NaN
2	elasticnet	0.2	saga
3	elasticnet	0.8	saga
4	elasticnet	0.2	saga
5	elasticnet	0.8	saga

[the rest of the output is omitted for brevity]

■

9.2.2 Random Search

Naturally, for a specific Ψ , if, 1) \mathcal{H}_Ψ^D that is used in the grid search contains the optimal set of hyperparameter values; 2) the estimation rule can accurately estimate

the actual performance; and 3) the computational requirement is affordable, it is quite likely that the grid search can lead to the optimal values for hyperparameters. However, in practice, none of these is guaranteed: 1) we are generally blind to the optimal values of hyperparameters for a specific application; 2) depending on the estimation rule and available sample size for evaluation, we constantly deal with a bias-variance-computation trilemma for the estimation rules; and 3) for large $|\mathcal{H}_\Psi^D|$ (i.e., a large number of hyperparameter values to evaluate), the computational requirements for grid search could become a burden. As a result, an alternative common technique for hyperparameter tuning is *random search*. In this strategy, one randomly chooses and examines part of \mathcal{H}_Ψ^D or, alternatively, randomly samples points for evaluation from a predefined subregion of the entire hyperparameter space. It has been shown that this strategy can lead to models that are as good or even better than the grid search (Bergstra and Bengio, 2012).

Scikit-learn implementation of random search model selection: For this purpose, we can use `RandomizedSearchCV` class from `sklearn.model_selection` module. Many parameters in `RandomizedSearchCV` behave similarly to those of `GridSearchCV`. That being said, rather than `param_grid` in `GridSearchCV`, in `RandomizedSearchCV` we have `param_distributions`, which similar to `param_grid` can be either a dictionary or a list of dictionaries. Nonetheless, possible values of a key in each dictionary here can even include distributions. Distributions that can be used in `RandomizedSearchCV` are arbitrary as long as they provide an `rvs()` method for sampling such as those found in `scipy.stats` (scipy, 2023). Furthermore, the number of times that `param_distributions` is sampled, is determined by `n_iter` parameter.

Two distributions that often come in handy for hyperparameter tuning are uniform and log-uniform (also known as reciprocal) distributions implemented by `uniform` and `loguniform` from `scipy.stats`, respectively. While a random variable is uniformly distributed in a range $[a, b]$ if its probability density function is constant in that range, and 0 otherwise, a random variable is log-uniformly distributed if its logarithm is uniformly distributed in $[\log(a), \log(b)]$. This is useful because sometimes we know the lower limit of a parameter but we do not have an upper bound for the parameter (e.g., C in logistic regression). In these situations, if a discrete grid is desired, it is common to take values that grow exponentially; for example, $0.01, 0.1, 1, 10, 100, \dots$. This is because a fixed change in a small values of hyperparameter could affect the results more than the same change for large values. In the same situation, if rather than a discrete grid, a range is desired, the use of log-uniform distribution ensures the sampling is uniform across intervals such as $[0.01, 0.1], [0.1, 1], [1, 10], \dots$

Example 9.9 In this example, we use the same data, classifier, and hyperparameter search space as in Example 9.7; however, instead of `GridSearchCV`, we use `RandomizedSearchCV`. As in Example 9.7, we consider both l_1 and l_2 penalty. However, C is assumed to be log-uniformly distributed between 0.01 and 100, and ν (defined in Section 6.2.2) is assumed to be uniformly distributed between 0 and 1.

```

import pandas as pd
from sklearn import datasets
from sklearn.linear_model import LogisticRegression as LRR
from sklearn.model_selection import RandomizedSearchCV,
    StratifiedKFold, train_test_split
from scipy.stats import loguniform, uniform

iris = datasets.load_iris()
X_train = iris.data
y_train = iris.target
X_train, X_test, y_train, y_test= train_test_split(iris.data, iris.
    target, random_state=100, test_size=0.2, stratify=iris.target)
print('X_train_shape: ' + str(X_train.shape) + '\nX_test_shape: ' +
    str(X_test.shape) +
    '\ny_train_shape: ' + str(y_train.shape) + '\ny_test_shape: ' +
    str(y_test.shape) + '\n')
lrr = LRR(max_iter=2000)
distrs = [{ 'penalty': ['l2'], 'C': loguniform(0.01, 100)},
    { 'penalty': ['elasticnet'], 'C': loguniform(0.01, 100),
    'l1_ratio':uniform(0, 1), 'solver':['saga']}]
strkfolds = StratifiedKFold(n_splits=3, shuffle=True, random_state=42)
gscv = RandomizedSearchCV(lrr, distrs, cv=strkfolds, n_jobs=-1, scoring=
    ['accuracy', 'roc_auc_ovr_weighted'], refit='accuracy', n_iter=10,
    random_state=42)
score_best_estimator=gscv.fit(X_train, y_train).score(X_test, y_test)
print('the highest score is: {:.3f}'.format(gscv.best_score_))
print('the best hyperparameter combination is: {}'.format(gscv.
    best_params_))
print('the accuracy of the best estimator on the test data is: {:.3f}'.
    format(score_best_estimator))
df = pd.DataFrame(gscv.cv_results_)
df.iloc[:,4:9]

```

X_train_shape: (120, 4)
X_test_shape: (30, 4)
y_train_shape: (120,)
y_test_shape: (30,)

the highest score is: 0.983
the best hyperparameter combination is: {'C': 2.7964859516062437,
 'l1_ratio':
 0.007066305219717406, 'penalty': 'elasticnet', 'solver': 'saga'}
the accuracy of the best estimator on the test data is: 0.967

	param_C	param_penalty	param_l1_ratio	param_solver	\
0	15.352247	12	NaN	NaN	
1	8.471801	12	NaN	NaN	

```

2  2.440061           12      NaN      NaN
3  0.042071  elasticnet  0.058084  saga
4  0.216189  elasticnet  0.142867  saga
5  0.012088           12      NaN      NaN
6  7.726718  elasticnet  0.938553  saga
7  0.05337  elasticnet  0.183405  saga
8  2.796486  elasticnet  0.007066  saga
9  0.14619           12      NaN      NaN

                                params
0      {'C': 15.352246941973478, 'penalty': 'l2'}
1      {'C': 8.471801418819974, 'penalty': 'l2'}
2      {'C': 2.440060709081752, 'penalty': 'l2'}
3  {'C': 0.04207053950287936, 'l1_ratio': 0.05808...}
4  {'C': 0.21618942406574426, 'l1_ratio': 0.14286...}
5  {'C': 0.012087541473056955, 'penalty': 'l2'}
6  {'C': 7.726718477963427, 'l1_ratio': 0.9385527...}
7  {'C': 0.05337032762603955, 'l1_ratio': 0.18340...}
8  {'C': 2.7964859516062437, 'l1_ratio': 0.007066...}
9  {'C': 0.14618962793704957, 'penalty': 'l2'}

```

Exercises:

Exercise 1: Load the California Housing dataset, which is part of `sklearn.datasets` module. We wish to use all features to predict the target `MedHouseVal` and to do so, we train and evaluate CART and AdaBoost (with CART as the estimator) using the standard 5-fold cross-validation with shuffling. To create a CART regressor anywhere that is needed, only set the minimum number of samples required at each leaf to 10 and leave all other hyperparameters as default. For the AdaBoost, set the number of regressors to 50 (`n_estimators`). In the cross-validation (`KFold`), CART, and AdaBoost, set `random_state=42`. Compute and report the estimates of the MAE and the R^2 using CV for the CART and the AdaBoost. Which case is showing the best performance in terms of \hat{R}^2 ?

Exercise 2: Use the same dataset as in Exercise 1 but this time randomly split the dataset to 80% training and 20% test. Use cross-validation grid search to find the best set of hyperparameters for random forest to predict the target based on all features. To define the space of hyperparameters of random forest assume:

1. `max_features` is 1/3;
2. `min_samples_leaf` $\in \{2, 10\}$;
3. `n_estimators` $\in \{10, 50, 100\}$

In the grid search, use 10-fold CV with shuffling and compute and report both the estimates of both the MAE and R^2 . Then evaluate the model that has the highest \hat{R}^2 determined from CV on the held-out test data and record its test \hat{R}^2 .

Exercise 3: In this exercise, use the same data, classifier, and hyperparameter search space as in Example 9.9; however, instead of grid search cross-validation, use validation set (for reproducibility, set the `random_state` in `RandomizedSearchCV`, `StratifiedKFold`, and `train_test_split` to 42). Monitor and report both the accuracy and the weighted ROC AUC on the validation set but use the accuracy for hyperparameter tuning. Report the best estimated set of hyperparameters, its score on validation-set, and the accuracy of the corresponding model on the test set.

Exercise 4: Suppose in a sample of 10 observations, there are 5 observations from the P class and 5 from the N class. We have two classifiers, namely, classifier A and classifier B, that classify an observation to P if the classifier score given to the observation is equal or greater than a threshold. We apply these classifiers to our sample and based on the scores given to observations, we rank them from 1 (having the highest score) to 10 (having the lowest score). Suppose ranks of observations that belong to class P are:

For classifier A: 1, 2, 5, 9, and 10.

For classifier B: 1, 3, 5, 7, and 9.

Let \hat{auc}_A and \hat{auc}_B denote the ROC AUC estimate for classifiers A and B, respectively. What is $\hat{auc}_A - \hat{auc}_B$?

- A) -0.08 B) -0.06 C) 0 D) 0.06 E) 0.08

Exercise 5: Which of the following best describes the use of resubstitution estimation rule in a regression problem to estimate R^2 ?

- A) We can not use resubstitution estimator for a regressor
- B) We can use resubstitution estimator for a regressor but it is not a recommended estimation rule
- C) We can use resubstitution estimator for a regressor and, as a matter of fact, it is the best estimation rule

Exercise 6: Which of the following evaluation rules can not be used for regression?

- A) stratified 5-fold cross-validation
- B) bootstrap
- C) resubstitution
- D) shuffle and split
- E) standard 5-fold cross-validation
- F) leave-one-out

Exercise 7: Suppose we have the following training data. What is the leave-one-out error estimate of the standard 3NN classification rule based on this data?

class	A	A	A	B	B	B
observation	0.45	0.2	0.5	0.35	0.45	0.4

- A) $\frac{2}{6}$ B) $\frac{3}{6}$ C) $\frac{4}{6}$ D) $\frac{5}{6}$ E) $\frac{6}{6}$

Exercise 8: Suppose we have a classifier that assigns to class positive (P) if the score is larger than a threshold and to negative (N) otherwise. Applying this classifier to some test data results in the scores shown in the following table where “class” refers to the actual class of an observation. What is the estimated AUC of this classifier based on this result?

observation #	class	score	observation #	class	score
1	P	0.25	6	P	0.35
2	P	0.45	7	N	0.38
3	N	0.48	8	N	0.3
4	N	0.4	9	P	0.5
5	N	0.33	10	N	0.2

- A) 0.585 B) 0.625 C) 0.665 D) 0.755 E) 0.865

⊕ **Exercise 9:**

Suppose in a binary classification problem, the class conditional densities are multivariate normal distributions with a common covariance matrix $\mathcal{N}(\mu_P, \Sigma)$ and $\mathcal{N}(\mu_N, \Sigma)$, and prior probability of classes π_P and π_N . Show that for any linear classifier in the form of

$$\psi(\mathbf{x}) = \begin{cases} P & \text{if } \mathbf{a}^T \mathbf{x} + b > T \\ N & \text{otherwise,} \end{cases} \quad (9.36)$$

where \mathbf{a} and b determine the discriminant function, the optimal threshold, denoted, T_{opt} , that minimizes the error rate

$$\varepsilon = \pi_N \times fpr + \pi_P \times fnr, \quad (9.37)$$

is

$$T_{opt} = \frac{h(\boldsymbol{\mu}_N) + h(\boldsymbol{\mu}_P)}{2} - \frac{g(\boldsymbol{\Sigma}) \ln\left(\frac{\pi_N}{\pi_P}\right)}{h(\boldsymbol{\mu}_N) - h(\boldsymbol{\mu}_P)}. \quad (9.38)$$

where

$$\begin{aligned} h(\boldsymbol{\mu}) &= \mathbf{a}^T \boldsymbol{\mu} + b, \\ g(\boldsymbol{\Sigma}) &= \mathbf{a}^T \boldsymbol{\Sigma} \mathbf{a}. \end{aligned} \quad (9.39)$$

Exercise 10: Suppose in a binary classification problem, the estimate of the ROC AUC for a specific classifier on a test sample \mathbf{S}_{te} is 1. Let n_{FP} and n_{FN} denote the number of false positives and false negatives that are produced by applying this classifier to \mathbf{S}_{te} , respectively, and \hat{acc} denote its accuracy estimate based on \mathbf{S}_{te} . Which of the following cases is impossible?

- A) $\hat{acc} = 1$
- B) $\hat{acc} < 1$
- C) $n_{FP} > 0$ and $n_{FN} = 0$
- D) $n_{FP} > 0$ and $n_{FN} > 0$

Exercise 11: Use scikit-learn to produce the ROC curve shown in Example 9.2.

Hint: one way is to use the true labels (`y_true`) and the given scores (`y_pred`) as inputs to `sklearn.metrics.RocCurveDisplay.from_predictions(y_true, y_pred)`. Another way is first use the true labels and the given scores as inputs to `sklearn.metrics.roc_curve` to compute the fpr , tpr (we can also view the thresholds at which they are computed). Then use the computed fpr and tpr as the inputs to `sklearn.metrics.RocCurveDisplay` to create the roc curve display object, and plot the curve by its `plot()` method.



Chapter 10

Feature Selection

Feature selection is a class of dimensionality reduction in which an “important” subset of features from a larger set of features is selected. A broader class of dimensionality reduction is *feature extraction* in which a set of candidate features are transformed to a lower cardinality set of new features using some linear or non-linear mathematical transformations. Feature selection can be seen as a special type of feature extraction in which the physical nature of features are kept by the selection process. Consider the EEG classification example that we have already seen in the first chapter. There, feature selection (based on prior knowledge) was used to select the C1 channel among the 64 available channels in the dataset. However, later we used a signal modeling technique to extract informative features including frequencies and amplitudes of some sinusoids from EEG recordings—this practice was feature extraction. In contrast with the widely used context-free search strategies for feature selection, feature extraction is generally domain dependent; that is to say, the success of commonly used transformations for feature extraction highly depends on the form of data and the context. For example, common transformations used in classification of time-series is generally different from those used for tabular datasets. Or even within time-series, some transformations are more common in a particular application than others—for example, Mel-frequency cepstral coefficients are some types of features more commonly used for speech processing rather than EEG analysis. As a result, in this chapter, we focus on feature selection search strategies that are context-free.

10.1 Dimensionality Reduction: Feature Selection and Extraction

The number of features in a given dataset is generally referred to as the *dimensionality* of the feature vectors or the dataset. Generally, when we have a dataset with a relatively large dimensionality, it is helpful to construct the final predictive model in a reduced dimensionality space. The methods used to reduce the dimensionality

of the original feature space are known as *dimensionality reduction* methods. There are three main reasons why one should reduce dimensionality:

- *interpretability*: it is often helpful to be able to see the relationships between the inputs (feature vector) and the outputs in a given problem. For example, what biomarkers (features) can possibly classify a patient with bad prognosis from another patient with good prognosis? There are two layers into interpretability of a final constructed model. One layer is the internal structure of the predictive model. For example, in Section 7.4, we observed that CART has a glass-box structure that helps shed light on the input-output relationship. Another layer is the input to the model *per se*. For example, if the original dataset has a dimensionality of 20,000, using all features as the input to the CART would impair interpretability regardless of its glass-box structure. In these situations, it is helpful to focus on “important” features that contribute to the prediction;
- *performance*: in pattern recognition and machine learning literature *peaking phenomenon* is often cited as a possible reason for dimensionality reduction. For example, consider the following comments from some pioneers in the field: “For training samples of finite size, the performance of a given discriminant rule in a frequentist framework does not keep on improving as the number p of feature variables is increased. Rather, its overall unconditional error rate will stop decreasing and start to increase as p is increased beyond a certain threshold, depending on the particular situation” (McLachlan, 2004, p. 391). A more thorough picture of this phenomenon is given in (Zollanvari et al., 2020); and
- *computational limitations*: given a fixed sample size, a larger data dimensionality would imply a larger number of floating point operations. Depending on the available hardware, and storage/processing models, this could become the bottleneck of the machine learning design process. These limitations along with the peaking phenomenon are collectively known as the *curse of dimensionality*: “Just recall the curse of dimensionality that we often faced to get good error rates, the number of training samples should be exponentially large in the number of components. Also, computational and storage limitations may prohibit us from working with many components” (Devroye et al., 1996, p. 561).

Here we create an experiment using synthetic data to demonstrate the effect of peaking phenomenon.

Example 10.1 Here we use `make_classification` from `sklearn.datasets` to generate synthetic datasets. In this regard, a dataset containing 200 observations from two 100-dimensional Gaussian distributions with independent features (100 observations from each class) is generated. Among the 100 features in the dataset, the first 10 features are informative and all other features are just random noise. For each dataset, we train a CART and LDA by adding one feature at a time to the classifier, starting from informative features going into non-informative ones. For each trained classifier, we record its accuracy on a test dataset of size 800

observations. This entire process is repeated 20 times to find the average accuracy of each classifier as a function of feature size.

```

import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier as CART
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

np.random.seed(42)
n_features = 100
mc_no = 20 # number of Monte Carlo simulation
acc_test = np.zeros((2, n_features)) # to record test accuracies for both LDA and CART
cart = CART()
lda = LDA()
for j in np.arange(mc_no):
    # by setting shuffle=False in make_classification, the informative features are put first
    X, y = make_classification(n_samples=1000, n_features=n_features, n_informative=10, n_redundant=0, n_repeated=0, n_classes=2, n_clusters_per_class=1, class_sep=1, shuffle=False)

    X_train_full, X_test_full, y_train, y_test = train_test_split(X, y, stratify=y, train_size=0.2)

    acc_test[0] += np.fromiter((cart.fit(X_train_full[:, :i], y_train).score(X_test_full[:, :i], y_test) for i in np.arange(1, n_features+1)), float)
    acc_test[1] += np.fromiter((lda.fit(X_train_full[:, :i], y_train).score(X_test_full[:, :i], y_test) for i in np.arange(1, n_features+1)), float)

for i, j in enumerate(['CART', 'LDA']):
    plt.figure(figsize=(4, 2.5), dpi = 150)
    plt.plot(np.arange(1, n_features+1), acc_test[i]/mc_no, 'k')
    plt.ylabel('test accuracy', fontsize='small')
    plt.xlabel('number of selected features', fontsize='small')
    plt.title('peaking phenomenon: ' + j, fontsize='small')
    plt.yticks(fontsize='small')
    plt.xticks(np.arange(0, n_features+1, 10), fontsize='small')
    plt.grid(True)

```

As we can see in Fig. 10.1, after all informative features are added, adding any additional feature would be harmful to the average classifier performance, which is a manifestation of the peaking phenomenon. ■

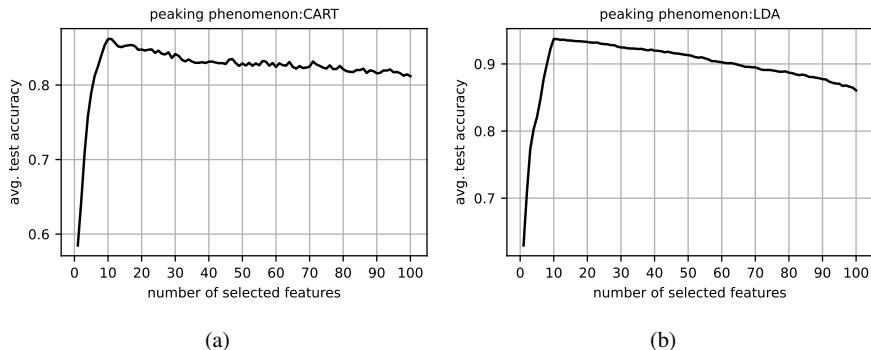


Fig. 10.1: Average test accuracy as a function of feature size: (a) CART; and (b) LDA.

10.2 Feature Selection Techniques

As mentioned earlier, the goal in feature selection is to choose a subset of candidate features that is important for prediction. The importance of a feature or a feature subset could be judged either by prior knowledge or could be data-driven. Although selecting features using prior knowledge is itself a legitimate feature selection strategy, which naturally depends on the quality of the prior knowledge and is domain dependent, we focus on objective criteria and methods that are data-driven. In this regard, there are three classes of methods:

- Filters
- Wrappers
- Embedded

10.2.1 Filter Methods

These methods select features based on *general characteristics of data measured independently of a learning algorithm*. They are called filters because the process filters out the original feature set to create the most “promising” subset before a predictive model is trained. The advantages of using filter methods include their efficiency (fast execution) and their generality to various learning algorithms. The latter property is attributed to the fact that filter methods select features based on certain characteristics of data rather than based on the interaction between a specific learning algorithm and the data. As an example, in Section 5.2.2, we have already seen the application of Pearson’s correlation coefficients in selecting features in a regression problem. Other criteria include statistical tests, information-theoretic measures, interclass distance, estimate of the Bayes error, and correlation coefficients,

to name just a few criteria. Detailed descriptions of all aforementioned criteria are beyond our scope. As a result, here we briefly discuss the main idea behind filter methods using statistical tests—and even there our discussion is not meant to be a complete guide on various aspects of statistical tests.

Statistical Tests: Statistical tests in feature selection are used to examine the “importance” of a feature by testing some properties of populations based on available data—for example, by testing the hypothesis that a single feature is unimportant in distinguishing two or multiple classes from each other. To put it another way, the definition of importance here depends on the type of hypothesis that we test. For example, we may designate a feature as important if based on the given sample, the hypothesis that there is no difference between the mean of class conditional densities is rejected. If based on the statistic (i.e., a quantity derived from data) that is used in the test (test statistic), we observe a compelling evidence against the null hypothesis, denoted \mathcal{H}_0 , we reject \mathcal{H}_0 in the favor of the *alternative hypothesis* \mathcal{H}_a , and we conclude the feature is potentially important; otherwise, we continue to believe in \mathcal{H}_0 . Therefore, in feature selection applications, \mathcal{H}_0 is generally the assertion that a feature is unimportant and the alternative hypothesis \mathcal{H}_a bears the burden of proof.

Formally, a test rejects a null hypothesis if its *P-value* (also known as the observed significance level) is less than a *significance level* (usually denoted by α), which is generally set to 0.05. For example, we reject \mathcal{H}_0 if the *P-value* is 0.02, which is less than $\alpha = 0.05$. The significance level of a test is the probability of a false positive occurring, which for our purpose means claiming a feature is important (i.e., rejecting \mathcal{H}_0) if \mathcal{H}_0 is true (i.e., the feature is not actually important). The *probability* of a false positive is defined based on the assumption of repeating the test on many samples of the same size collected from the population of interest; that is to say, if \mathcal{H}_0 is true and the test procedure is repeated infinitely many times on different samples of the same size from the population, the proportion of rejected \mathcal{H}_0 is α . A detailed treatment of *P-value* and how it is obtained in a test is beyond the scope here (readers can consult statistical textbooks such as ([Devore et al., 2013](#))), but the lower the *P-value*, the greater the strength of observed test statistic against the null hypothesis (more compelling evidence against the hypothesis that the feature is unimportant).

Common statistical tests that can be used for identifying important features include *t*-test, which is for binary classification, and ANOVA (short for analysis of variance), which is for both binary and multiclass classification. In ANOVA, a significant *P*-value (i.e., a *P*-value that is less than α) means the feature would be important in classifying at least two of the classes. In ANOVA, a statistic is computed that under certain assumptions follows *F*-distribution ([Devore et al., 2013](#)) (therefore, the statistic is known as *F*-statistic and the statistical test is known as *F*-test). Whether the computed *F*-statistic is significant or not is based on having the *P*-value less than α . In regression, the most common statistical test to check the importance of a feature in predicting the target is based on a test that determines whether the Pearson’s correlation coefficient is statistically significant. For this purpose, it also conducts an *F*-test.

It is worthwhile to remember that many common statistical tests only take into account the effect of one feature at a time regardless of its multivariate relationship with other features. This is itself a major shortcoming of these tests because in many situations features work collectively to result in an observed target value. Nonetheless, insofar as statistical tests are concerned, it can be shown that when we have many features, and thereof many statistical tests to examine the importance of each feature at a time, the probability of having at least one false positive among all tests, also known as family-wise error rate (FWER), is more than α where α is the significance level for each test (see Exercise 1). A straightforward approach to control FWER is the well-known Bonferroni correction. Let m denote the number of tests performed. In this approach, rather than rejecting a null hypothesis when $P\text{-value} \leq \alpha$ as it is performed in testing a single hypothesis, Bonferroni correction involves rejecting the null hypotheses by comparing P -values to α/m (Exercise 2). Despite the ease of use, Bonferroni correction is a conservative approach; that is to say, the strict threshold to reject a null hypothesis results in increasing the probability of not rejecting the null hypothesis when it is false (i.e., increases the false negative rate). There are more powerful approaches than Bonferroni. All these approaches that are essentially used to control a metric of interest such as FWER, fpr or fdr in multiple testing problem are collectively known as *multiple testing corrections*. It would be instructive to see the detail algorithm (and later implementation) of at least one of these corrections. As a result, without any proof, we describe a well-known multiple testing correction due to Benjamini and Hochberg ([Benjamini and Hochberg, 1995](#)).

Benjamini–Hochberg (BH) false discovery rate procedure: This procedure is for controlling the expected (estimate of the) false discovery rate in multiple tests; that is, to control (see Section 9.1.2 for the definition of fdr):

$$E[\hat{fdr}] = E\left[\frac{n_{FP}}{n_{TP} + n_{FP}}\right]. \quad (10.1)$$

Suppose we have p features and, therefore, p null hypothesis denoted by $H_{0,1}, H_{0,2}, \dots, H_{0,p}$ (each null hypothesis asserts that the corresponding feature is unimportant). Therefore, we conduct p statistical tests to identify the null hypotheses that are rejected in the favor of their alternative hypotheses. The BH fdr procedure for multiple test correction is presented in the following algorithm.

Algorithm Benjamini–Hochberg False Discovery Rate Procedure

- 1) Set a desired maximum level, denoted α , for the expectation of $f\hat{d}r$ as defined in (10.1); for example, we set $\alpha = 0.01$.
- 2) Sort P -values obtained from tests and denote them by $P_{(1)} \leq P_{(2)} \leq \dots \leq P_{(p)}$.
- 3) Let j denote the largest i such that $P_{(i)} \leq \frac{i}{p}\alpha$. To find j , start from the largest P -value and check whether $P_{(p)} \leq \frac{p}{p}\alpha$; if not, then check whether $P_{(p-1)} \leq \frac{p-1}{p}\alpha$; if not, then check $P_{(p-2)} \leq \frac{p-2}{p}\alpha$, and continue. Stop as soon as the inequity is met— j is the index of the $P_{(i)}$ that meets this condition for the first time.
- 4) Reject the null hypothesis $H_{0,(1)}, H_{0,(2)}, \dots, H_{0,(j)}$. This means all $P_{(i)}, i \leq j$ (P -values smaller than $P_{(j)}$) are also significant even if $P_{(i)} \leq \frac{i}{p}\alpha$ is not met for some of them.

Example 10.2 Suppose we have 10 features and run ANOVA test for each to identify important features. The following P -values are obtained where P_i denotes the P -value for feature $i = 1, 2, \dots, 10$. Apply BH procedure to determine important features by an $\alpha = 0.05$.

$P_1 = 0.001, P_2 = 0.1, P_3 = 0.02, P_4 = 0.5, P_5 = 0.002, P_6 = 0.005, P_7 = 0.01, P_8 = 0.04, P_9 = 0.06, P_{10} = 0.0001$

Step 1: $\alpha = 0.05$, which is already given in the problem.

Step 2: Sort P -values: $0.0001 \leq 0.001 \leq 0.002 \leq 0.005 \leq 0.01 \leq 0.02 \leq 0.04 \leq 0.06 \leq 0.1 \leq 0.5$.

Step 3:

- is $0.5 \leq \frac{10}{10}0.05$? No.
- is $0.1 \leq \frac{9}{10}0.05$? No.
- is $0.06 \leq \frac{8}{10}0.05$? No.
- is $0.04 \leq \frac{7}{10}0.05 = 0.035$? No.
- is $0.02 \leq \frac{6}{10}0.05 = 0.03$? Yes.

Step 4: Reject any null hypothesis with a P -value less or equal to 0.02. That is, null hypothesis corresponding to 0.0001, 0.001, 0.002, 0.005, and 0.01. This means the indices of important features are 10, 1, 5, 6, 7, and 3. ■

Using multiple tests correction, we select a number of features that are important based on the P -values and what we try to control using the correction procedure. This way, we generally do not know in advance how many features will be selected. In other words, given an α for the correction, which is a measure of the expected

\hat{fdr} or FWER, we end up with a number of “important” features that is unknown in advance. Although leaving the number of selected features to the algorithm would be desirable in many situations, there is another way with which we can impose the number of features we wish to select for the final model training. In this approach, we rank the P -values for all tests in an ascending order, and simply pick k features with the least P -values (the most significant P -values); that is to say, we pick the k best individual features. The obvious drawback of this approach is that the number of selected features are subjective. That being said, even the value of α used in multiple testing correction is subjective but values such as $\alpha = 0.01$ or $\alpha = 0.05$ are typically acceptable and common to use.

Scikit-learn implementation of statistical tests for filter feature selection: Feature selection using statistical tests is implemented by different classes in `sklearn.feature_selection` module. For example, to check the significance of F -statistic computed as part of ANOVA in classification, we can use `f_classif` function. The F -test to check significance of F -statistic computed from Pearson’s correlation coefficient in regression is implemented in `f_regression` function. The aforementioned feature selection filter based on Benjamini–Hochberg procedure for the “`fdr`” correction is implemented using `SelectFdr` class. Filters based on other corrections such as “`fpr`” (false positive rate) or FWER are implemented using `SelectFpr` and `SelectFwe`, respectively. Here, we present some examples to further clarify the utility of these classes. By default `SelectFdr`, `SelectFpr`, and `SelectFwe` use ANOVA test for classification (i.e., `f_classif`), but this could be changed so that these corrections can be used with other statistical tests for classification or regression problems. For example, to use F -test for regression, we can use `SelectFdr(score_func=f_regression)`. To explicitly specify a number of features to pick, `SelectKBest` class can be used. We can also use `SelectPercentile` to keep a percentage of features. Again, by default, these classes use `f_classif`, which implements ANOVA, and that could be changed depending on the required tests.

Example 10.3 Here we wish to conduct ANOVA and then use BH procedure to correct for multiple testing. For this purpose we use the Genomic dataset `GenomicData_orig.csv` that was used in Exercise 1, Chapter 6. The data had 21050 features, and thereof that many statistical tests to determine the importance of each feature based on F -statistic. For this purpose, we first apply ANOVA (using `f_classif`) and “manually” implement the procedure outlined above for the BH correction. Then we will see that using `SelectFdr`, we can easily use ANOVA, correct for multiple testing based on BH procedure, and at the same time, implement the filter (i.e., select our important features). Last but not least, `SelectKBest` is used to pick the best 50 individual features based on ANOVA.

```
from sklearn.feature_selection import SelectFdr, f_classif, SelectKBest
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
```

```
from sklearn.impute import SimpleImputer # to impute
from sklearn.preprocessing import StandardScaler # to scale

df = pd.read_csv('data/GenomicData_orig.csv')
df.head()
df = df.drop(df.columns[[0,1]], axis=1)

# preprocessing including imputation and standardization
np.random.seed(42)
y = df.ClassType
X = df.drop('ClassType', axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y)
print('X shape = {}'.format(X_train.shape) + '\ny shape = {}'.format(y_train.shape))
print('X shape = {}'.format(X_test.shape) + '\ny shape = {}'.format(y_test.shape))

imp = SimpleImputer(strategy='median').fit(X_train)
X_train = imp.transform(X_train)
X_test = imp.transform(X_test)
scaler = StandardScaler().fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

anova_res = f_classif(X_train, y_train) # f_classif returns one pair of F statistic and P-value for each feature
pvalues = anova_res[1]
# check the number of P-values that are less than 0.01 significance
level
alpha=0.01
print("The number of important features based on F statistic (no multiple hypothesis correction):", sum(anova_res[1]<alpha))
pvalues_taken_before_BH = pvalues[pvalues < alpha]
print("The maximum p-value considered significant before BH correction is: {:.4f}".format(pvalues_taken_before_BH.max()))

# to implement BH procedure for the P-values obtained from ANOVA
pvalues_sorted = np.sort(anova_res[1])
pvalues_taken_using_BH = pvalues_sorted[pvalues_sorted <= np.arange(1, X_train.shape[1]+1)*alpha/X_train.shape[1]]
print("The number of important features based on F statistic (after multiple hypothesis correction):", len(pvalues_taken_using_BH))
print("The maximum p-value considered significant after BH correction is: {:.4f}".format(pvalues_taken_using_BH.max()))

# to use SelectFdr to use both the BH procedure and filter features
filter_fdr = SelectFdr(alpha=0.01).fit(X_train, y_train) # filter_fdr is a transformer using which we can transform the data
```

```

print("Let's look at the p-values for the first 20 features\n",_
      ↵filter_fdr.pvalues_[0:20]) # using pvalues_ attributes to access p_
      ↵values
X_train_filtered_BH = filter_fdr.transform(X_train) #
      ↵X_train_filtered_BH is our filtered data based on ANOVA and fdr_
      ↵correction
print("The number of important features based on F statistic (after_
      ↵multiple hypothesis correction):", X_train_filtered_BH.shape[1])
X_test_filtered_BH = filter_fdr.transform(X_test)
print("The shape of test data (the number of features should match with_
      ↵training): ", X_test_filtered_BH.shape)

# to use SelectKBest
filter_KBest = SelectKBest(k=50).fit(X_train, y_train)
X_train_filtered_KBest = filter_KBest.transform(X_train)
print("The number of important features based on F statistic (using_
      ↵K=50 best individual feature):", X_train_filtered_KBest.shape[1])
X_test_filtered_KBest = filter_KBest.transform(X_test)
print("The shape of test data (the number of features should match with_
      ↵training): ", X_test_filtered_KBest.shape)

```

```

X shape = (129, 21050)
y shape = (129,)
X shape = (43, 21050)
y shape = (43,)
The number of important features based on F statistic (no multiple_
      ↵hypothesis
correction): 7009
The maximum p-value considered significant before BH correction is: 0.0100
The number of important features based on F statistic (after multiple_
      ↵hypothesis
correction): 5716
The maximum p-value considered significant after BH correction is: : 0.
      ↵0027
Let's look at the p-values for the first 20 features
[1.26230819e-06 2.57999668e-04 7.20697614e-01 4.61019523e-01
 1.44329139e-04 7.73450170e-07 7.88695806e-01 5.38203614e-01
 6.12687071e-01 6.08016364e-01 7.42496319e-10 7.22040641e-01
 6.38454543e-01 9.08196717e-03 5.42029503e-17 8.64539548e-02
 8.26923725e-09 3.88193129e-06 8.46980459e-02 3.57175908e-01]
The number of important features based on F statistic (after multiple_
      ↵hypothesis
correction): 5716
The shape of test data (the number of features should match with_
      ↵training):
(43, 5716)
The number of important features based on F statistic (using K=50 best
individual feature): 50
The shape of test data (the number of features should match with_
      ↵training):
(43, 50)

```

10.2.2 Wrapper Methods

These methods wrap around a specific learning algorithm and evaluate the importance of a feature subset using the predictive performance of the learner trained on the feature subset. The learning algorithm that is used for wrapper feature selection does not necessarily need to be the same as the learning algorithm that is used to construct the final predictive model; for example, one may use an efficient algorithm such as decision trees or Naive Bayes for feature selection but a more sophisticated algorithm to construct the final predictive model based on selected features.

Commonly used wrapper methods include a search strategy using which the space of feature subsets is explored. Each time a candidate feature subset is selected for evaluation based on the search strategy, a model is trained and evaluated using an evaluation rule along with an evaluation metric. In this regard, cross-validation is the most popular evaluation rules and either the accuracy, the ROC AUC, or the F_1 score would be common metrics to use. At the end of the search process, the feature subset with highest score is returned as the chosen feature subset. Although there are many search strategies to use in this regard, here we discuss three approaches that are straightforward to implement:

- Exhaustive Search
- Sequential Forward Search (SFS)
- Sequential Backward Search (SBS)

Exhaustive Search: Suppose we wish to select k out of p candidate features that are “optimal” based on a score such as the accuracy estimate. In this regard, one can exhaustively search the space of all possible feature subsets of size k out of p . This means that there are $\binom{p}{k} = \frac{p!}{k!(p-k)!}$ feature subset to choose from. However, this could be extremely large even for relatively modest values of p and k . For example, for $\binom{50}{20} > 10^{13}$. In reality the situation is exacerbated because we do not even know how many features should be selected. To put it another way, there is generally no *a priori* evidence about any specific k . In other words, an exhaustive search should be conducted for all possible values of $k = 1, 2, \dots, p$. This means that to find the optimal feature subset, the number subset evaluation by an exhaustive search is $\sum_{k=1}^p \binom{p}{k} = 2^p - 1$. Therefore, the number of feature subset to evaluate grows exponentially and becomes quickly impractical as p grows—for example for $p = 50$, 10^{15} feature subsets should be evaluated! Although exhaustive search is the optimal search strategy, the exponential complexity of the search space is the main impediment in its implementation. As a result, we generally rely on suboptimal search strategies as discussed next.

Sequential Forward Search (SFS): This is a *greedy* search strategy in which at each iteration a single feature, which in combination with the current selected feature subset leads to the highest score, is identified and added to the selected feature subset. This type of search is known as greedy because the search does not revise the decision made in previous iterations (i.e., does not revise the list of selected feature subsets from previous iterations). Let $f_i, i = 1, \dots, p$, and F_i denote the i^{th} feature in a p -

dimensional feature vector and a subset of these features that is selected at iteration t , respectively. Formally, the algorithm is implemented in Algorithm “Sequential Forward Search”.

Algorithm Sequential Forward Search

1. Set $t = 0$ and F_0 to empty set. Choose a learning algorithm for model training, a scoring metric (e.g., accuracy), an evaluation rule (e.g., cross-validation), and a stopping criterion:
 - 1.1. One stopping criterion could be the size of feature subsets to pick; for example, one may decide to select $k < p$ features.
 - 1.2. Another stopping criterion that does not require specifying a predetermined number of features to pick is to continue the search as long as there is a tangible improvement in the score by adding one or a few features sequentially. For example, one may decide to stop the search if after adding 3 features the score is not improved more than a prespecified constant.
 2. Find the $f_i \notin F_t$ that maximizes the estimated scoring metric of the model trained using $F_t \cup f_i$. Denote that feature by f_i^{best} . In other words, f_i^{best} is the best univariate feature that along with features F_t leads to the highest score.
 3. update $F_t \leftarrow F_t \cup f_i^{\text{best}}$ and $t \leftarrow t + 1$.
 4. Repeat steps 2 and 3 until the stopping criterion is met (e.g., based on the first stopping criterion, if $|F_t| = k$). Once the search is stopped, the best feature subset is F_t .
-

Sequential Backward Search (SBS): This is another greedy search strategy in which at each iteration a single feature is removed from the current selected feature subset. The feature that is removed is the one that its removal leads to the highest score. Formally, the algorithm is implemented in Algorithm “Sequential Backward Search”.

Remark: The computational complexity of SBS is sometimes an impediment in its utility for feature selection. Suppose the set of candidate feature sets contains p features and we wish to select $p/2$ features using either SFS or SBS. In this case, it can be shown that the number of subsets that are evaluated in both SFS and SBS are the same (Exercise 3). Other things equal, it can be shown that in this setting, the computational complexity of SBS is generally much larger than SFS because the models in SBS are trained in higher-dimensional spaces compared with models trained in SFS.

Algorithm Sequential Backward Search

1. Set $t = 0$ and $F_0 = \{f_1, f_2, \dots, f_p\}$. Choose a learning algorithm for model training, a scoring metric, an evaluation rule, and a stopping criterion similar to SFS (i.e., a predetermined number of features or removing features as long as there is a tangible improvement in estimated performance score).
2. Find the $f_i \in F_t$ that maximizes the estimated scoring metric of the model trained using $F_t - f_i$. Denote that feature by f_i^{worst} . In other words, f_i^{worst} is the worst univariate feature that its removal from the current selected feature subset leads to the highest score.
3. Update $F_t \leftarrow F_t - f_i^{\text{worst}}$ and $t \leftarrow t + 1$.
4. Repeat steps 2 and 3 until the stopping criterion is met. Once the search is stopped, the best feature subset is F_t .

Scikit-learn implementation of SBS and SFS: Scikit-learn implements SFS and SBS with cross-validation through `SequentialFeatureSelector` class from `sklearn.feature_selection`. To use SFS and SBS, the `direction` parameter is set to '`forward`' and '`backward`', respectively.

The `SequentialFeatureSelector` class Scikit-learn (as of version 1.2.1) implements SBS and SFS with two stopping criteria, which can be chosen via parameter `n_features_to_select`. In this regard, this parameter can be set to: 1) either the absolute size (an integer) or proportion (a float) of candidate features; and 2) '`auto`' that stops the search if the improvement between two consecutive iteration is not more than the value of `tol` parameter. Similar to the `cv` parameter used `cross_val_score` or `GridSearchCV` that we saw in Chapter 9, we can set the `cv` parameter in `SequentialFeatureSelector` to have more control over the type of desired cross-validation to use.

Example 10.4 In this example, we use a similar approach as in Example 10.1 to generate a single set of synthetic data. The dataset contains 10 informative features and 90 uninformative noisy features (see Example 10.1 for data description). We then use SFS and SBS to choose 10 features and see how many of them are among the informative features. The use of this synthetic data here is insightful as we already know the number of informative and uninformative features so we can have a better picture of the performance of the feature selection mechanisms on the dataset.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.model_selection import StratifiedKFold, train_test_split
from sklearn.feature_selection import SequentialFeatureSelector
```



```

False True False True False False False False False False False
False False False False False False False True False False False
False False False False False False False False False False False
False False False False]

```

The number of truly informative features chosen by SBS is: 6
 $50.1 \text{ s} \pm 0 \text{ ns}$ per loop (mean \pm std. dev. of 1 run, 1 loop each)

In this example, SBS picks more of the informative features than SFS but it takes almost 9 times more than SFS to stop. ■

10.2.3 Embedded Methods

These are algorithms that once the predictive model is constructed, the structure of the model can be used for feature selection. In other words, the feature selection stage is embedded into the model training. In this regard, once the model is trained, the importance of features are judged by the properties of learners. For example, the magnitudes of coefficients in a linear model trained on normalized data could be viewed as the influence of their corresponding features on the target variable. Now if in the same scenario a feature coefficient is zero, it means no influence from that feature on the target variable in the given data. There are two types of embedded methods: 1) non-iterative; and 2) iterative.

Non-iterative Embedded Methods: Non-iterative embedded methods do not require an iterative search strategy to select features. Here, by “iteration”, we do not refer to the iterative optimization method that is used to construct the underlying predictive models given a fixed set of features. Instead, we refer to an iterative procedure for selecting features. An example of a non-iterative embedded method for feature selection is logistic regression based on l_1 regularization (lasso). As seen in Section 6.2.2, this classifier had the property of selecting features in a high-dimensional setting as it sets the coefficients of many features to zero. Some other models such as decision trees and random forest have also an internal mechanism to define “importance” for a feature. A common metric to measure importance in random forests is the Mean Decrease Impurity (MDI) (Louppe et al., 2013). The MDI for a feature k (f_k), denoted MDI_{f_k} , is defined as the normalized sum of impurity drops across all nodes and all trees that use f_k for splitting the nodes. Using similar notations as in Section 7.2.2, for a trained random forest containing n_{Tree} trees, this is given by

$$\text{MDI}_{f_k} = \frac{1}{n_{Tree}} \sum_{Tree} \sum_{m \in Tree: \text{if } \xi = (k, .)} \frac{n_{S_m}}{n} \Delta_{m, \xi}, \quad (10.2)$$

where $Tree$ is a variable indicating a tree in the forest, $\frac{n_{S_m}}{n}$ shows the proportion of samples reaching node m , $\xi = (k, .)$ means if the split was based on f_k with some threshold, and

$$\Delta_{m,\xi} = i(\mathbf{S}_m) - \frac{n_{\mathbf{S}_{m,\xi}^{\text{Yes}}}}{n_{\mathbf{S}_m}} i(\mathbf{S}_{m,\xi}^{\text{Yes}}) - \frac{n_{\mathbf{S}_{m,\xi}^{\text{No}}}}{n_{\mathbf{S}_m}} i(\mathbf{S}_{m,\xi}^{\text{No}}). \quad (10.3)$$

With this definition, a higher value of MDI_{f_k} is perceived as a greater feature importance. If we use Gini index as impurity measure, then MDI is known as Gini importance. The feature importance for a decision tree is obtained by applying (10.2) to that one tree.

Iterative Embedded Methods: In contrast with wrappers that compute a number of performance scores for adding or removing feature subsets, iterative embedded methods guide the search by some criteria related to the learning algorithm or the structure of the trained predictive model. For example, in *grafting* (Perkins et al., 2003), a forward selection procedure is proposed for models that are trained using gradient decent to optimize the log loss. At each iteration, the approach identifies and adds the feature that has the fastest rate in decreasing the loss function (i.e., has highest magnitude of the loss function gradient with respect to the weight of the feature). One of the most popular iterative embedded feature selection method is *recursive feature elimination* (RFE) (Guyon et al., 2002). In this method, a feature ranking criterion related to the structure of the trained predictive model is used to recursively remove lowest-ranked features—for example, as we said before, in linear models and random forests, the magnitude of coefficients and the MDI can be used for feature ranking, respectively. Formally RFE is implemented as in the next algorithm.

Although one of the stopping criteria in the presented algorithm for RFE is based on estimated performances, the efficiency of RFE when compared with SBS (will be discussed shortly) plays down the role of a “stopping” criterion. That is to say, rather than stopping the search, it is more common to use cross-validation to estimate the performance score for *all* possible candidate feature subsets exposed by RFE and then select the one with the highest estimated score. We refer to this procedure as “RFE-CV”. As an example, for p candidate features and $s = 1$, RFE-CV computes p scores estimated by CV.

By using RFE one can achieve significant computational efficiency with respect to SBS because:

- 1) at each iteration, the feature ranking criterion is an efficient way to identify features to remove. In comparison with SBS, if $F_t = l$, then SBS needs assessing l features using a model evaluation rule. For example, using K -fold CV, it should train and evaluate $l \times K$ surrogate models to remove one feature. However, in RFE, training one model on F_t would give us the information we need to remove one feature (or even a subset of features). Once the features are removed, in RFE-CV, K surrogate models are trained and evaluated to assign a score estimated by CV to the new feature subset; and
- 2) in SBS, features are removed one at a time. Although there is generalized form of SBS (called Generalized SBS) in which at iteration t all combinations of feature subsets of a specific size of F_t are examined for removal, the computational complexity of that could be even higher than SBS. However, removing multiple

Algorithm Recursive Feature Elimination

-
1. Set $t = 0$ and $F_0 = \{f_1, f_2, \dots, f_p\}$. Choose a learning algorithm that internally implements a ranking criterion. Choose a step size s , which determines the number of features to remove at each iteration. Pick a stopping criterion:
 - 1.1. One stopping criterion could be the size of feature subsets to pick; for example, one may decide to select $k < p$ features.
 - 1.2. Similar to SBS, another stopping criterion that does not require specifying a predetermined number of features to select could be removing features as long as there is a tangible improvement in the estimated performance score.
 2. Use features in F_t to train a model. Identify the s features that are ranked lowest and call this subset F_t^{lowest} .
 3. Update $F_t \leftarrow F_t - F_t^{\text{lowest}}$. In case of having a stopping criterion based on estimated performances, compute and record the estimated performance of the classifier trained using F_t . Set $t \leftarrow t + 1$.
 4. Repeat steps 2 and 3 until the stopping criterion is met. Once the search is stopped, the best feature subset is F_t .
-

features in RFE is quite straightforward with no additional computational cost and it is indeed the recommended approach for high-dimensional data.

Scikit-learn implementation of RFE: Scikit-learn implements two classes for RFE: RFE and RFECV, which are both in `sklearn.feature_selection` module. In RFE one needs to set `n_features_to_select` parameter, which indicates the number of features selected by the RFE. Currently, default value of this parameter, which is `None`, will return half of the features. RFECV, on the other hand, will pick the number of features from cross-validation. For example, for `step=1`, will compute p (number of features) CV scores and selects the feature subsets that led to the highest score. Another convenient and useful feature of RFE and RFECV classes is that they both implement `predict` and `score` methods using which they reduce the features in a provided test data to the selected number of features and perform prediction and compute the score.

Example 10.5 This is the same dataset used in Example 10.5. Here we use RFE to select 10 features and then use RFE-CV to choose a number of features determined from cross-validation.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
from sklearn.feature_selection import RFE
```

```

from sklearn.feature_selection import RFECV
from sklearn.model_selection import StratifiedKFold, train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as
    LDA

np.random.seed(42)
n_features = 100
n_informative = 10
lda = LDA()

X, y = make_classification(n_samples=1000, n_features=n_features, n_informative=10, n_redundant=0, n_repeated=0, n_classes=2, n_clusters_per_class=1, class_sep=1, shuffle=False)
X_train_full, X_test_full, y_train, y_test = train_test_split(X, y, stratify=y, train_size=0.2)
strkfolds = StratifiedKFold(n_splits=5, shuffle=True)

```

```

import time
start = time.time()
rfe = RFE(lda, n_features_to_select=10)
rfe.fit(X_train_full, y_train)
print(rfe.get_support())
print("The number of truly informative features chosen by RFE is:", sum(rfe.get_support()[:10]))
end = time.time()
print("The RFE execution time (in seconds): {:.4f}".format(end - start))
lda_score = rfe.score(X_test_full, y_test)
print("The score of the lda trained using selected features on training data is: {:.4f}".format(lda_score))

```

[True True True False True False True True True True False
 False False False False False False False False False False False
 False False False False False False False False False False False
 False False False False False False False False False False False
 False False False False False False False False False False False
 False False False False False False False False False False False
 False False False False False False False False False False False
 False False False False False False False False False False False
 False False False False False False False False False False False
 False False True False False]
 The number of truly informative features chosen by RFE is: 8
 The RFE execution time (in seconds): 0.2335
 The score of the lda trained using selected features on training data is: 0.9500

```

import time
start = time.time()
rfecv = RFECV(lda, cv=strkfolds, n_jobs=-1)
rfecv.fit(X, y)
print(rfecv.get_support())

```

```
print("The number of features selected by RFE CV is:", rfecv.  
      n_features_ )  
print("The number of truly informative features chosen by RFE is:",  
      sum(rfecv.get_support()[:10]))  
end = time.time()  
print("The RFE CV execution time (in seconds): {:.4f}".format(end -  
      start))  
lda_score = rfecv.score(X_test_full, y_test)  
print("The score of the lda trained using selected features on training  
      data is: {:.4f}".format(lda_score))
```

The number of features selected by RFE CV is: 12

The number of truly informative features chosen by RFE is: 9

The RFE CV execution time (in seconds): 1.3327

The score of the lda trained using selected features on training data is:
↳ 0.9563

Efficiency of RFE with respect to SBS is an extremely useful property. The computational burden incurred by SBS is a major impediment in the utility of SBS for high-dimensional data. In examples 10.4 and 10.5 we use a relatively modest dimensionality and we see that RFECV is about $50.1/1.33 \approx 38$ times faster to compute than SBS (the exact ratio depends on the runs and the machine). This ratio, which is an indicator of efficiency of RFECV with respect to SBS, becomes larger if we increase the number of dimensions and/or training sample size (see exercises 4 and 5).

Exercises:

Exercise 1: Suppose in a dataset we have m features and for each feature we test a null hypothesis \mathcal{H}_0 against an alternative hypothesis \mathcal{H}_a to determine whether the feature is important/significant (i.e., \mathcal{H}_0 is rejected in the favor of \mathcal{H}_a). Assuming each test has a significance level of $\alpha = 0.05$, what is FWER (the probability of having at least one false positive)?

⊕ Exercise 2: Suppose a study involves m hypothesis tests. Show that if each null hypothesis is rejected when the P -value for the corresponding test is less than or equal to α/m (Bonferroni correction), where α is the significance level of each test,

then $\text{FWER} \leq \alpha$. To prove use the following well-known results: 1) P -values have uniform distribution between 0 and 1 when the null hypothesis is true (Rice, 2006, p. 366); and 2) for events $A_i, i = 1, \dots, m$, $P\left(\bigcup_{i=1}^m A_i\right) \leq \sum_{i=1}^m P(A_i)$, which is known as the union bound.

Exercise 3: Assume we would like to choose the $p/2$ suboptimal feature subset out of p (even) number of features using both SFS and SBS.

1. write an expression in terms of p that shows the number of feature subsets that must be evaluated using SFS (simplify your answer as much as possible).
2. write an expression in terms of p that shows the number of feature subsets that must be evaluated using SBS (simplify your answer as much as possible).

Exercise 4: A) Run examples 10.4 and 10.5 on your machine and find the ratio of execution time of RFE-CV to that of SBS.

B) Rerun part (A) but this time by increasing the number of candidate features from 100 to 150 (i.e., setting `n_features` in `make_classification` to 150). How does this change affect the execution time ratio found previously in part (A)?

C) As in part (A), use `n_features = 100` but this time, change the size of training set from 200 to 1000 and run the examples. How does this change affect the execution time ratio found previously in part (A)?

Exercise 5: Suppose SBS is used to pick l out of p candidate features in a classification problem. At the same time, RFE-CV is used to pick an unknown number of features (because the number of selected features depends on the scores estimated by CV). Assume that in both SBS and RFE-CV, K -fold CV is used. Show that if $p \gg l$ and $K \gg 1$, the number of classifiers trained in the process of SBS is approximately $p/2$ more than the number of classifiers trained in RFE-CV.

Exercise 6: Suppose we have a dataset of 100 features and we wish to select 50 features using a wrapper method. We use 5-fold cross-validation with LDA classifier inside wrapper in all the following techniques. Which of the following statements is true?

- A) Using exhaustive search is likely to terminate faster than both sequential backward search and sequential forward search
- B) Using sequential backward search is likely to terminate faster than sequential forward search
- C) Using sequential forward search is likely to terminate faster than sequential backward search



Chapter 11

Assembling Various Learning Steps

So far we have seen that in order to achieve a final predictive model, several steps are often required. These steps include (but not limited to) normalization, imputation, feature selection/extraction, model selection and training, and evaluation. The misuse of some of these steps in connection with others have been so common in various fields of study that have occasionally triggered warnings and criticism from the machine learning community. This state of affairs can be attributed in large part to the misuse of resampling evaluation rules such as cross-validation and bootstrap for model selection and/or evaluation in connection with a combination of other steps to which we refer as a composite process (e.g., combination of normalization, feature extraction, and feature selection). In this chapter we focus primarily on the proper implementation of such composite processes in connection with resampling evaluation rules. To keep discussion succinct, we use feature selection and cross-validation as typical representatives of the composite process and a resampling evaluation rule, respectively. We then describe appropriate implementation of

1. Feature Selection and Model Evaluation Using Cross-Validation
2. Feature and Model Selection Using Cross-Validation
3. Nested Cross-Validation for Feature and Model Selection, and Evaluation

11.1 Using Cross-Validation Along with Other Steps in a Nutshell

Cross-validation is effective for model evaluation because it uses data at hand to simulates the effect of unseen future data for evaluating a trained model. Therefore, the held-out fold in each iteration of cross-validation, which serves as a test set for evaluating the surrogate model, must be truly treated as unseen data. In other words, the held-out fold should not be used in steps such as normalization and/or feature selection/extraction. After all, “unseen” data can naturally not be used in these steps.

To formalize the idea, let $\Xi : \mathbf{I} \rightarrow \mathbf{O}$ denote a composite process that transforms an input sample space \mathbf{I} to an output space \mathbf{O} , which depending on the application includes one or more processing steps that are data-dependent—steps such as

segmentation (e.g., if the segment duration is determined from data), imputation, normalization, and feature selection/extraction. To put it another way, Ξ is a composite *transformer*, which means it is a composite estimator that transforms data. Furthermore, similar to Section 9.1.1, let Fold_k and $\mathbf{S}_{tr,n} - \text{Fold}_k$ denote the held-out fold and training folds at iteration k in K -fold CV such that $\bigcup_{k=1}^K \text{Fold}_k = \mathbf{S}_{tr,n}$. Hereafter, we show transformation performed by Ξ on an element of the input space $\mathbf{S} \in \mathbf{I}$ by $\Xi(\mathbf{S})$. Our goal is to use cross-validation to evaluate the performance of a classifier trained on the output of the composite transformer Ξ when applied to training data. This is formalized as follows:

Goal:

1. learn Ξ from $\mathbf{S}_{tr,n}$ —learning Ξ from a given data $\mathbf{S}_{tr,n}$ means estimating all transformations that are part of Ξ from $\mathbf{S}_{tr,n}$. Let Ξ_{tr} denote the learned composite process;
2. use Ξ_{tr} to transform $\mathbf{S}_{tr,n}$ into a new training data $\Xi_{tr}(\mathbf{S}_{tr,n})$;
3. use the transformed training data $\Xi_{tr}(\mathbf{S}_{tr,n})$ along with a classification rule to train a classifier $\psi_{tr}(\mathbf{x})$; and
4. use K -fold CV to evaluate the trained classifier $\psi_{tr}(\mathbf{x})$.

In a nutshell, the correct way of using K -fold CV to evaluate $\psi_{tr}(\mathbf{x})$ is to apply K -fold CV *external* to the composite process Ξ . In fact implementation of the external CV is not new—it is merely the *proper* way of implementing the standard K -fold CV algorithm that was presented in Section 9.1.1 by realizing that the “learning algorithm” Ψ presented there is a combination of a composite estimator Ξ that transforms the data plus a classifier/regressor construction algorithm. Nevertheless, we present a separate algorithm, namely, “External” K -fold CV ([Ambroise and McLachlan, 2002](#)), to describe the implementation in more details. Furthermore, hereafter we refer to Ψ as a “composite estimator” because it is the combination of a composite transformer and an estimator (classifier/regressor).

Note that in the “External” K -fold CV algorithm, although the general form of processes used within $\Xi_1, \Xi_2, \dots, \Xi_K$ are the same, the precise form of each transformation is different because each Ξ_k is learned from a different data $\mathbf{S}_{tr,n} - \text{Fold}_k$, $k = 1, \dots, K$. At the same time, we observe that throughout the algorithm, Fold_k is not used anyhow in learning Ξ_k or training $\psi_k(\mathbf{x})$. Therefore, Fold_k is truly treated as an unseen data for the classifier trained on iteration-specific training set $\mathbf{S}_{tr,n} - \text{Fold}_k$. To better grasp and appreciate the essence of the “External” K -fold CV algorithm in achieving the “Goal”, in the next few sections in this chapter we use feature selection as a typical processing step used in Ξ ; after all, it was mainly the common misuse of feature selection and cross-validation that triggered several warnings ([Dupuy and Simon \(2007\)](#); [Ambroise and McLachlan \(2002\)](#); [Michiels et al. \(2005\)](#)). That being said, the entire discussion remains valid if we augment Ξ with other transformations discussed before.

Algorithm “External” K -fold CV

1. Split the dataset $\mathbf{S}_{tr,n}$ into K folds, denoted Fold_k .
 2. For k from 1 to K
 - 1.1. Learn Ξ from $\mathbf{S}_{tr,n} - \text{Fold}_k$; Let $\Xi_{tr,k}$ denote the learned composite process in this iteration.
 - 1.2. Use $\Xi_{tr,k}$ to transform $\mathbf{S}_{tr,n} - \text{Fold}_k$ into a new data $\Xi_{tr,k}(\mathbf{S}_{tr,n} - \text{Fold}_k)$.
 - 1.3. Use the transformed training data $\Xi_{tr,k}(\mathbf{S}_{tr,n} - \text{Fold}_k)$ to train a surrogate classifier $\psi_k(\mathbf{x})$.
 - 1.4 Classify observations in the transformed held-out fold $\Xi_{tr,k}(\text{Fold}_k)$ using $\psi_k(\mathbf{x})$.
 3. Compute the K -fold CV estimate of performance metric using misclassified/correctly classified observations obtained in step 1.4. This gives us the estimate of performance metric for $\psi_{tr}(\mathbf{x})$ (the “Goal”).
-

11.2 A Common Mistake

Insofar as feature selection and cross-validation are concerned, the most common mistake is to apply feature selection on the full data at hand, construct a classifier using selected features, and then use cross-validation to evaluate the performance of the classifier. This practice leads to *selection bias*, which is caused because the classifier is evaluated based on samples that were used in the first place to select features that are part of the classifier [Ambroise and McLachlan \(2002\)](#). This mistake may stem from the discrepancy between the perception of the “Goal” and the actual practical steps required to reach it. Here is a common perception of the Goal:

We wish to perform feature selection, construct a classifier on the selected features, and evaluate the performance of the constructed classifier using cross-validation.

This is an incorrect approach (the common mistake) to achieve the goal:

We need to perform feature selection first, construct a classifier on the selected features, and evaluate the performance of the constructed classifier using cross-validation applied on the data with selected features.

Unfortunately, here intuition could lead us astray. The following example illustrates the issue with this common mistake.

Example 11.1 Here we create a synthetic training data that has the same size as the genomic dataset (172×21050) used in Example 10.3. However, this synthetic dataset is a *null* data where all observations for all features across both classes are random samples of the standard Gaussian distribution. Naturally, all features are equally irrelevant to distinguish between the two classes because samples from both classes are generated from the same distribution. By keeping the sample size across both classes equal, we should expect a 50% accuracy for any learning algorithm applied and evaluated on this null dataset.

We wish to use this null data to examine the effect of the aforementioned common mistake. In this regard, once the training data is generated, we apply ANOVA (see Chapter 10) on the training set and pick the best 10 features, and then apply cross-validation on the training data to evaluate the performance of a CART classifier constructed using the 10 selected features. At the same time, because this is a synthetic model to generate data, we set the size of test set almost two times larger than the training set. In particular, we choose 172 (training set size) + 328 (test set size) = 500 . Furthermore, in order to remove any bias towards a specific realization of the generated datasets or selected folds used in CV, we repeat this entire process 50 times (`mc_no = 50` in the following code) and report the average 5-fold CV and the average test accuracies at the end of this process.

```

import numpy as np
from sklearn.datasets import make_classification
from sklearn.feature_selection import SelectKBest
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier as CART

np.random.seed(42)
n_features = 21050
n_sample = 500 # size of training + test sets
ratio_train2full = 172/500 # the proportion of training data to n_sample
mc_no = 50 # number of Monte Carlo repetitions
kfold = 5 # K-fold CV
cv_scores_matrix = np.zeros((mc_no, kfold))
test_scores = np.zeros(mc_no)
for j in np.arange(mc_no):
    X = np.random.normal(0, 1, size=(n_sample, n_features))
    y = np.concatenate((np.ones(n_sample//2, dtype='int_'), np.
    zeros(n_sample//2, dtype='int_')))
    X_train, X_test, y_train, y_test = train_test_split(X, y,
    stratify=y, train_size=ratio_train2full)
    filter_KBest = SelectKBest(k=10).fit(X_train, y_train)
    X_train_filtered = filter_KBest.transform(X_train)
    strkfolds = StratifiedKFold(n_splits=kfold, shuffle=True)
    cart = CART()

```

```

cv_scores_matrix[j,:] = cross_val_score(cart, X_train_filtered, y_train, cv=StrKFold)
X_test_filtered = filter_KBest.transform(X_test)
final_cart = cart.fit(X_train_filtered, y_train)
test_scores[j] = final_cart.score(X_test_filtered, y_test)

print("the average K-fold CV accuracy is: {:.3f}".
      .format(cv_scores_matrix.mean()))
print("the average test accuracy is: {:.3f}" .format(test_scores.mean()))

```

the average K-fold CV accuracy is: 0.649

the average test accuracy is: 0.493

As we can see, there is a considerable difference of $\sim 15\%$ between the average CV and the average test-set estimates of the accuracy. The test accuracy, as expected, is about 50%; however, the CV estimate is overly optimistic. This reason for this is solely because we used X_{train} to select features and then used the selected features within cross-validation to evaluate the classifier. This means that Fold $_k$ in CV was already seen in training CART through the feature selection process. In other words, Fold $_k$ is not an independent test set used for evaluating a surrogate CART trained on $S_{tr,n} - \text{Fold}_k$. ■

As we saw in the previous section, the correct approach to evaluate $\psi_{tr}(\mathbf{x})$ is to apply K -fold CV external to feature selection. In other words, to evaluate the classifier trained based on features that are selected by applying the feature selection on full training data, external K -fold CV implies:

We need to successively hold out one fold as part of cross-validation, apply feature selection on the remaining data to select features, construct a classifier using these selected features, and evaluate the performance of the constructed classifier on the held-out fold (and naturally take the average score at the end of cross-validation).

From a practical perspective, cross-validation can be used for both model evaluation, model selection, and both at the same time. As a result, in what follows we discuss implementations of all these cases separately.

11.3 Feature Selection and Model Evaluation Using Cross-Validation

Let $\mathbf{S}_{tr,n,p}$ denote a given training data with n observations of p dimensionality. Let $\psi_{tr,d}(\mathbf{x})$ denote a classifier trained on $d \leq p$ selected features using all n observations in the training data. Assuming there is no test data, we wish to estimate the performance of $\psi_{tr,d}(\mathbf{x})$ using cross-validation applied on $\mathbf{S}_{tr,n,p}$. To do so we can just use the “External” K -fold CV algorithm presented in Section 11.1 by

taking the composite process Ξ as the feature selection. This can also be seen as the application of the standard K -fold CV presented in Section 9.1.1 by taking the composite estimator Ψ as *feature selection* (Ξ) + *classifier/regressor learning algorithm*.

Scikit-Learn Pipeline class and implementation of feature selection and CV model evaluation: As we know from Chapter 9, `cross_val_score`, for example, expects a single estimator as the argument but here our estimator should be a composite estimator ($\Psi \equiv \text{feature selection} + \text{classifier construction rule}$). Can we create such a composite estimator Ψ and treat it as a single estimator in scikit-learn? The answer to this question is affirmative. In scikit-learn we can make a single estimator out of a feature selection procedure (the object from the class implementing the procedure) and a classifier (the object from the class implementing the classifier). This is doable by the `Pipeline` class from `sklearn.pipeline` module.

The common syntax for using the `Pipeline` class is `Pipeline(steps)` where `steps` is a list of tuples `(arbitrary_name, transformer)` with the last one being a tuple of `(arbitrary_name, estimator)`. In other words, all estimators that are chained together in a `Pipeline` except for the last one must implement the `transform` method so that the output of one is transformed and used as the input to the next one (these transformers create the composite transformer Ξ). The last estimator can be used to add the classification/regression learning algorithm to the `Pipeline` (to create the composite estimator Ψ). Below is an example where two transformers `PCA()` and `SelectKBest()`, and one estimator `CART()` are chained together to create a composite estimator named `pipe`.

```
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.tree import DecisionTreeClassifier as CART
from sklearn.feature_selection import SelectKBest
pipe = Pipeline(steps=[("dimen_reduc", PCA(n_components=10)), 
    ("feature_select", SelectKBest(k=5)), ("classifier", CART())])
```

Example 11.2 Here we aim to apply the external K -fold CV under the same settings as in Example 11.1. The external K -fold CV is used here to correct for the selection bias that occurred previously in Example 11.1. As mentioned before, we should create one composite estimator to implement *feature selection + classifier learning algorithm* and use that estimator within each iteration of CV. As in Example 11.1, at the end of the process, the average performance estimate obtained using cross-validation and test set are compared.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.feature_selection import SelectKBest
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
```

```

from sklearn.tree import DecisionTreeClassifier as CART
from sklearn.pipeline import Pipeline

np.random.seed(42)
n_features = 21050
n_sample = 500 # size of training + test set
ratio_train2full = 172/500
mc_no = 50 # number of Monte Carlo repetitions
kfold = 5 # K-fold CV
cv_scores_matrix = np.zeros((mc_no, kfold))
test_scores = np.zeros(mc_no)
strkfold = StratifiedKFold(n_splits=kfold, shuffle=True)
pipe = Pipeline(steps=[('kbest', SelectKBest(k=10)), ('clf', CART())])
# kbest and clf are arbitrary names
for j in np.arange(mc_no):
    X = np.random.normal(0, 1, size=(n_sample, n_features))
    y = np.concatenate((np.ones(n_sample//2, dtype='int_'), np.
    zeros(n_sample//2, dtype='int_')))
    X_train, X_test, y_train, y_test = train_test_split(X, y,
    stratify=y, train_size=ratio_train2full)
    cv_scores_matrix[j,:] = cross_val_score(pipe, X_train, y_train,
    cv=strkfold)
    final_cart = pipe.fit(X_train, y_train)
    test_scores[j] = final_cart.score(X_test, y_test)

print("the average K-fold CV accuracy is: {:.3f}".
    format(cv_scores_matrix.mean()))
print("the average test accuracy is: {:.3f}".format(test_scores.mean()))

```

the average K-fold CV accuracy is: 0.506

the average test accuracy is: 0.493

As we can see here, the average CV estimate of the accuracy is very close to that of test-set estimate and both are about 50% as they should (recall that we had a null dataset). Here by `cross_val_score(pipe, X_train, y_train, cv=strkfold)`, we are training 5 CART surrogate classifiers where each is trained using (possibly) different set of features of size 10 (because $S_{tr,n,p}$ – Fold $_k$ is different in each iteration of CV), and evaluate each on its corresponding held-out fold (Fold $_k$). This is a legitimate performance estimate of the outcome of applying the composite estimator Ψ to $S_{tr,n,p}$; that is to say, the performance estimate of our final constructed classifier, which itself is different from all surrogate classifiers. This is the classifier constructed using `final_cart = pipe.fit(X_train, y_train)`—we name it “final_cart” because this is what we use for predicting unseen observations in the future. Here as we could easily generate test data from our synthetic model, we evaluated and reported the performance of this classifier on test data using `test_scores[j] = final_cart.score(X_test,`

`y_test`). However, in the absence of test data, the CV estimate itself obtained by `cross_val_score(pipe, X_train, y_train, cv=strkfold)` is a reasonable estimate of the performance of `final_cart`—as observed the average difference between CV estimate and the hold-out estimate is less than 1%. ■

11.4 Feature and Model Selection Using Cross-Validation

As discussed in Section 9.2.1, a common approach for model selection is the grid search cross-validation. However, once feature selection is desired, the search for the best model should be conducted by taking into account the joint relationship between the search and the feature selection; that is to say, the best feature-model is the one that leads to the highest CV score across all examined combinations of features and models (i.e., a joint feature-model selection).

Scikit-Learn implementation of feature selection and CV model selection: As discussed in Section 9.2.1, the grid search cross-validation is conveniently implemented using `sklearn.model_selection.GridSearchCV` class. Therefore, a convenient way to integrate feature selection as part of model selection is using an object of `Pipeline` in `GridSearchCV`. However, as `Pipeline` creates a composite estimator out of several other estimators, there is a special syntax that we need to use to instruct `param_grid` used in `GridSearchCV` about the specific estimators over which we wish to conduct tuning (and of course the candidate parameter values for those estimator). Furthermore, we need to instantiate the `Pipeline` using one possible choice of transformer/estimator and then use `param_grid` of `GridSearchCV` to populate the choices used in `steps` of `Pipeline`. To illustrate the idea as well as introduce the special aforementioned syntax, we look into the following instructive example.

Example 11.3 Here we use `sklearn.datasets.make_classification` to generate a data of 1000 features with only 10 informative features (`n_informative=10`) and a train/test sample size of 200/800. We wish to use `GridSearchCV` for joint feature-model selection. For this purpose, we use $k = 5, 10, 50$ as candidate number of best features to select using `SelectKBest`. As for the classification algorithms, we use: 1) logistic regression with l_2 penalty and possible values of C being 0.1, 1, and 10; 2) CART and random forest (even though random forest has an internal feature ranking) with minimum samples per leaf being either 1 or 5; and 3) logistic regression with lasso penalty (l_1 regularization) but in doing so, k best individual features (`SelectKBest`) will not be used because l_1 regularization performs feature selection internally (Section 10.2.3). The following code implements this joint feature-model selection procedure.

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

```

from sklearn.datasets import make_classification
from sklearn.feature_selection import SelectKBest
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier as CART
from sklearn.ensemble import RandomForestClassifier as RF
from sklearn.linear_model import LogisticRegression as LRR
from sklearn.pipeline import Pipeline
from sklearn.model_selection import StratifiedKFold, GridSearchCV

np.random.seed(42)
n_features = 1000
acc_test = np.zeros(n_features)
kfold = 5 # K-fold CV
X, y = make_classification(n_samples=1000, n_features=n_features,
                           n_informative=10, n_redundant=0, n_repeated=0, n_classes=2,
                           n_clusters_per_class=1, class_sep=1, shuffle=False)
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
                                                    train_size=0.2)
pipe = Pipeline(steps=[('kbest', SelectKBest()), ('clf', CART())]) #_
# kbest and clf are arbitrary names
strkfolds = StratifiedKFold(n_splits=kfold, shuffle=True)

param_grid = [
    {'clf': [LRR()], 'clf__penalty': ['l2'], 'clf__C': [0.1, 1, 10],
     'kbest': [SelectKBest()], 'kbest__k': [5, 10, 50]},
    {'clf': [LRR()], 'clf__penalty': ['l1'], 'clf__C': [0.1, 1, 10],
     'clf__solver': ['liblinear'], 'kbest': ['passthrough']},
    {'clf': [CART(), RF()], 'clf__min_samples_leaf': [1, 5],
     'kbest': [SelectKBest()], 'kbest__k': [5, 10, 50]}
]

gscv = GridSearchCV(pipe, param_grid, cv=strkfolds)
score_best_estimator=gscv.fit(X_train, y_train).score(X_test, y_test)

print('the highest score is: {:.3f}'.format(gscv.best_score_))
print('the best composite estimator is: {}'.format(gscv.
    best_estimator_))
print('the best combination is: {}'.format(gscv.best_params_))
print('the accuracy of the best estimator on the test data is: {:.3f}'.
    format(score_best_estimator))
df = pd.DataFrame(gscv.cv_results_)
#df # uncomment this to see all 24 combinations examined in this example

```

the highest score is: 0.950
 the best composite estimator is: Pipeline(steps=[('kbest', SelectKBest()), ('clf', RandomForestClassifier())])
 the best combination is: {'clf': RandomForestClassifier(),

```
'clf__min_samples_leaf': 1, 'kbest': SelectKBest(), 'kbest__k': 10}
the accuracy of the best estimator on the test data is: 0.929
```

Several important points in the implementation of this example:

- here the choice of `SelectKBest()` and `CART()` as the estimators used along with '`kbest`' and '`clf`' in `steps=[('kbest', SelectKBest()), ('clf', CART())]` is arbitrary. They are replaced by specific choices that are indicated in `param_grid` used for `GridSearchCV`. For example, we can instantiate the `Pipeline` class as

```
Pipeline(steps=[('kbest', SelectKBest()),
               ('clf', KNeighborsClassifier())])
```

but because `KNeighborsClassifier()` is not used in `param_grid`, the `GridSearchCV` does not use `KNeighborsClassifier()` anyhow;

- to instruct `sklearn` about the specific choices of steps that should be used within `GridSearchCV`, we need to look into the names used with `steps` (of `Pipeline`) and then use each step name as one key in dictionaries used in `param_grid` (of `GridSearchCV`) and specify its values. For example, here '`clf`' is the (arbitrary) name given to the last step of the pipe. In defining our grids (using `param_grid`), we use this name as a key and then use a list of possible estimators [`CART()`, `RF()`] as its value (e.g., '`clf`': [`CART()`, `RF()`]) in the above code);
- the special `Pipeline` syntax: to set the parameters of each step, we use the name of the step along with a double underscore `_` in keys within each dictionary used in `param_grid` and use a list of possible values of that parameter as the value of that key. For example, '`clf__penalty`': ['12'] or '`kbest__k`': [5, 10, 50] that were used in the previous code; and
- in cases where multiple estimators are tuned on the same parameters with the same candidate values (of course if the parameters of interest exist in all those estimators), we can integrate them in one dictionary by adding them to the list of values used for the key pointing to that step. This is why in the above code, we used '`clf`': [`CART()`, `RF()`] in `{'clf': [CART(), RF()], 'clf__min_samples_leaf': [1, 5], 'kbest': [SelectKBest()], 'kbest__k': [5, 10, 50]}`.

In Example 11.3, we used the grid search cross-validation for model selection. Nevertheless, as we have already performed the cross-validation, it seems tempting to also use/report the CV estimate of the performance of the best classifier (outcome of the joint feature-model selection) as a legitimate CV estimate of that classifier. However, unfortunately, this is not a legitimate practice. To better understand the issue, let us focus on a single specific split of CV for iteration k : Fold_k and $\mathcal{S}_{tr,n} - \text{Fold}_k$; that is to say, we replace CV model selection with hold-out estimation used for model selection. At the same time, we restrict the model selection procedure to hyperparameter tuning for a specific classifier.

Recall that to have a legitimate hold-out estimate of performance for a classifier $\psi_k(\mathbf{x})$ that is trained on $\mathbf{S}_{tr,n} - \text{Fold}_k$, Fold_k should have not been used in training $\psi_k(\mathbf{x})$. In the process of model selection, we use Fold_k in estimating the performance of a series of classifiers where each is trained using a different combination of hyperparameter values. We can correctly argue that the performance estimate obtained by applying each classifier on the held-out fold (Fold_k) is a legitimate performance estimate of *that* classifier; however, *the moment we select one of these classifiers (i.e., the one to which we refer as the “best classifier”) over others based on the performance estimates that were obtained on Fold_k , we are indeed using Fold_k in the training process leading to that best classifier (through our “selection”), and that is not a legitimate hold-out estimate anymore!*

There is indeed a counterintuitive argument here: our estimate obtained on Fold_k for the best classifier is still the same as *before* designating that classifier as the best one, and we said previously that this estimate was a legitimate hold-out estimate of performance for that classifier before designating that as the best classifier. So what happens that after “selecting” that classifier over others (now that is called the best classifier), this is not a legitimate performance estimate even though the estimate is not changed?

To understand this, we need to put these practices in the context of repeated sampling of the underlying populations. Let θ^* denote the best combination of hyperparameter values (that is used to train the best classifier). A legitimate hold-out estimator must have a held-out fold that is independent of the training process. This is because the hold-out estimator (the same for CV) is designed to be an almost unbiased estimator in estimating the true error of a classification rule (slightly pessimistic because we use fewer samples in training). In broad terms, an unbiased estimator is one that on *average* (over many samples of the same population) behaves similarly to what is estimating. Let $E[\epsilon_{\theta}]$ denote the mean of true error rate for a classification algorithm (also known as *unconditional* error (Braga-Neto, 2020)) given a specific set of hyperparameters θ . Now suppose we have additional training datasets denoted as $\mathbf{S}'_{tr,n}, \mathbf{S}''_{tr,n}, \dots$, which are sampled from the same populations, and splitting each leads to $\mathbf{S}'_{tr,n} - \text{Fold}'_k$ and Fold'_k split, $\mathbf{S}''_{tr,n} - \text{Fold}''_k$ and Fold''_k split, \dots . We consider two cases:

Case 1. Were we going to train a classifier with a specific combination of hyperparameter which happen to be θ^* on $\mathbf{S}'_{tr,n} - \text{Fold}'_k, \mathbf{S}''_{tr,n} - \text{Fold}''_k, \dots$, and take the average of their performance estimates on $\text{Fold}'_k, \text{Fold}''_k, \dots$, then that average would have been very close to $E[\epsilon_{\theta^*}]$ —here each performance estimate is a *legitimate* estimate. This is what is meant by “before selecting the best classifier”: we treat θ^* as any other combination of hyperparameter values; that is, the same argument applies to any other combination of hyperparameter values because all classifiers are trained using the same combination.

Case 2. Similar to what we did in *finding θ^** , let us train a number of classifiers on $\mathbf{S}'_{tr,n} - \text{Fold}'_k$ where each is using a unique combination of hyperparameter values. Then by comparing their performance estimates on Fold'_k , we select $\theta^{*\prime}$, which is not necessarily the same as θ^* . Doing the same on $\mathbf{S}''_{tr,n} - \text{Fold}''_k$ and Fold''_k leads to $\theta^{*\prime\prime}, \dots$. Here to find each best classifier, we use each $\text{Fold}_k, \text{Fold}'_k, \text{Fold}''_k, \dots$.

in finding θ^* , $\theta^{*'}$, $\theta^{**'}$, In this case, the average of the performance estimates of each best classifier that is obtained on Fold'_ k , Fold''_ k , ..., is not necessarily close to any of $E[\varepsilon_{\theta^*}]$, $E[\varepsilon_{\theta^{*'}}]$, $E[\varepsilon_{\theta^{**'}}]$, The problem is caused because held-out folds are seen in training each base classifier (via selecting the best classifier), and each time we end up with a different classifier characterized by a unique θ . As a result, the performance estimates obtained are neither hold-out error estimates nor CV estimates if repeated on different splits.

Therefore, it is not legitimate to use the performance estimates obtained as part of the grid search CV for evaluating the performance of the best classifier. The implication of violating this is not purely philosophical as it could have serious implications in practice. To see that, in the following example we investigate the implication of such a malpractice in Example 11.2. In particular, we examine the impact of perceiving the value of CV obtained as part of the grid search as the performance estimate of the best classifier.

Example 11.4 Here we apply the grid search CV using the same hyperparameter space defined in Example 11.3 on the set of null datasets generated in Example 11.2. We examine whether the average CV accuracy estimates of the best classifiers selected as part of the grid search is close to 50%, which should be the average true accuracy of any classifier on these null datasets.

```
import numpy as np
from sklearn.pipeline import Pipeline
from sklearn.datasets import make_classification
from sklearn.model_selection import GridSearchCV
from sklearn.feature_selection import SelectKBest
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier as CART
from sklearn.ensemble import RandomForestClassifier as RF
from sklearn.linear_model import LogisticRegression as LRR

np.random.seed(42)
n_features = 21050
n_sample = 500
ratio_train2test = 172/500
mc_no = 20 # number of MC repetitions
kfold = 5 # K-fold CV
cv_scores = np.zeros(mc_no)
test_scores = np.zeros(mc_no)
pipe = Pipeline(steps=[('kbest', SelectKBest()), ('clf', CART())]) # ↵
# kbest and clf are arbitrary names
strkfolds = StratifiedKFold(n_splits=kfold, shuffle=True)

grids = [
    {'clf': [LRR()], 'clf__penalty': ['l2'], 'clf__C': [0.1, 1, 10]},
```

```

    'kbest': [SelectKBest()], 'kbest__k': [5, 10, 50]},
    {'clf': [LRR()], 'clf__penalty': ['l1'], 'clf__C': [0.1, 1, 10],
     'clf__solver': ['liblinear'], 'kbest': ['passthrough']},
    {'clf': [CART(), RF()], 'clf__min_samples_leaf': [1, 5],
     'kbest': [SelectKBest()], 'kbest__k': [5, 10, 50]}
]

for j in np.arange(mc_no):
    X = np.random.normal(0, 1, size=(n_sample, n_features))
    y = np.concatenate((np.ones(n_sample//2, dtype='int_'), np.
    zeros(n_sample//2, dtype='int_')))
    X_train, X_test, y_train, y_test = train_test_split(X, y,
    stratify=y, train_size=ratio_train2test)
    gscv = GridSearchCV(pipe, grids, cv=strkfold, n_jobs=-1)
    test_scores[j]=gscv.fit(X_train, y_train).score(X_test, y_test)
    cv_scores[j] = gscv.best_score_

print("the average K-fold CV accuracy is: {:.3f}".format(cv_scores.
mean()))
print("the average test accuracy is: {:.3f}".format(test_scores.mean()))

```

the average K-fold CV accuracy is: 0.568

the average test accuracy is: 0.502

As we can see there is a difference of more than 6% between the average accuracy estimates of the best 20 classifiers (`mc_no = 20`) obtained as part of the grid search CV and their average true accuracy (estimated on independent hold-out sets). The 56.8% average estimate obtained on this balanced null dataset is an indicator of some predictability of the class labels using given data, which is wrong as these are null datasets. ■

11.5 Nested Cross-Validation for Feature and Model Selection, and Evaluation

As we saw in the previous section, the performance estimates obtained as part of the grid search cross-validation can not be used for evaluating the performance of the best classifier. The question is then how to possibly estimate its performance? We have used test sets in Example 11.4 for evaluating the best classifier. In the absence of an independent test set, one can split the given data to training and test sets and use hold-out estimator. However, as discussed in Section 9.1.1, in situations where the sample size is not large, cross-validation estimator is generally preferred with respect to hold-out estimator. But can we use cross-validation to evaluate the performance of the classifier trained using the best combination of hyperparameter values obtained by the grid search CV?

The answer to this question is positive. To do so, we need to implement another cross-validation that itself is external to the entire learning process; that is, external to *feature selection + model selection* (and remember that we use *feature selection* as a typical step in Ξ). As this external cross-validation for model evaluation (or, equivalently, learning process evaluation) is wrapped around the entire process of *feature selection + model selection*, which can be implemented by grid search CV, this practice is commonly referred to as *nested cross-validation*.

At this point it is important to distinguish between three types of cross-validation that could be inside each other (although computationally it could be very expensive):

- (1) a possible cross-validation that could be used in wrapper feature selection (inner to all);
- (2) a cross-validation used for grid-search model selection (external to CV used in (1) but internal to the one used in (3)). In the context of nested cross-validation though, this cross-validation is known as the *internal CV*; and
- (3) a cross-validation used for evaluation (external to all). In the context of nested cross-validation, this cross-validation is known as the *external CV*.

Nevertheless, for the sake of clarity, we could refer to the CVs used in (1), (2), and (3) by their functionality; that is, “feature selection CV”, “model selection CV”, and “model evaluation CV”, respectively.

At the same time, it is important to realize that by using nested cross-validation, we are not changing the “best classifier”; that is, the classifier that was chosen by the procedure described in Section 11.4 remains the same. What we are doing by nested cross-validation is the way we evaluate that classifier (or, equivalently, the learning process leading to that classifier). The scikit-learn implementation of nested CV is left as an exercise.

Exercises:

Exercise 1: Here we would like to use the same data generating model and hyper-parameter space that were used in Example 11.4. Keep all the following and other parameters the same

```
n_features = 21050
n_sample = 500
ratio_train2test = 172/500
mc_no = 20
```

Apply nested cross-validation on `X_train` and `y_train` and report the average estimate of accuracy obtained by that (recall that the average is over the number of Monet-Carlo simulations: `mc_no = 20`). For both CVs used as part of the model selection and the model evaluation, use stratified 5-fold CV. What is the average CV estimate obtained using nested cross-validation? Is it closer to the 50% compared to the what was achieved in Example 11.4?

Hint: in Example 11.4 after `GridSearchCV` is instantiated, use the object as estimator used in `cross_val_score`.

Exercise 2: Suppose we wish to use `SelectKBest` from scikit-learn in selecting a set of 50 features, denoted \mathbf{f} , in a dataset \mathbf{S}_{tr} including 10,000 features, and train a classifier ψ using \mathbf{S}_{tr} with selected features. In training the classifier, we use a pre-specified combination of hyperparameter values; that is, no hyperparameter tuning is used. We apply the 3-fold CV external to feature selection to properly evaluate the trained classifier. In the process of 3-fold CV, we obtained 3 feature sets denoted \mathbf{f}_1 , \mathbf{f}_2 , and \mathbf{f}_3 . Which of the following is correct about the relationship between \mathbf{f} and \mathbf{f}_k , $k = 1, 2, 3$ (two sets \mathbf{A} and \mathbf{B} are *equal*, denoted $\mathbf{A} = \mathbf{B}$, if and only if they contain the same elements)?

- A) $\mathbf{f} = \mathbf{f}_1 = \mathbf{f}_2 = \mathbf{f}_3$
- B) $\mathbf{f} \neq \mathbf{f}_1 = \mathbf{f}_2 = \mathbf{f}_3$
- C) $\mathbf{f} \neq \mathbf{f}_1 \neq \mathbf{f}_2 \neq \mathbf{f}_3$
- D) $\mathbf{f} = \mathbf{f}_1 \cup \mathbf{f}_2 \cup \mathbf{f}_3$ and $\mathbf{f}_1 = \mathbf{f}_2 = \mathbf{f}_3$
- E) $\mathbf{f} = \mathbf{f}_1 \cap \mathbf{f}_2 \cap \mathbf{f}_3$ and $\mathbf{f}_1 = \mathbf{f}_2 = \mathbf{f}_3$
- F) $\mathbf{f} = \mathbf{f}_1 \cup \mathbf{f}_2 \cup \mathbf{f}_3$ and $\mathbf{f}_1 \neq \mathbf{f}_2 \neq \mathbf{f}_3$
- G) $\mathbf{f} = \mathbf{f}_1 \cap \mathbf{f}_2 \cap \mathbf{f}_3$ and $\mathbf{f}_1 \neq \mathbf{f}_2 \neq \mathbf{f}_3$

Exercise 3: In an application, “our goal” is to perform feature selection, construct a classifier on the selected features, and estimate the error rate of the constructed classifier using cross-validation. To do so, we take the following “action”: perform feature selection first, construct a classifier on the selected features, reduce the dimensionality of the data with the selected features (i.e., create a “reduced data” with selected features), and estimate the error rate of the constructed classifier using cross-validation (CV) applied on the reduced data. Which of the following statement can best describe the legitimacy of this practice? Explain your rationale.

- A) Both the goal and the action taken to achieve the goal are legitimate.
- B) The goal is legitimate but the action taken to achieve the goal is not because the use of CV in this way would make it even more pessimistically biased.
- C) The goal is legitimate but the action taken to achieve the goal is not because the use of CV in this way could make it even optimistically biased.



Chapter 12

Clustering

Clustering is an important machine learning task that aims to discover “sensible” or “natural” groups (clusters) within a given data. In contrast with “classification” that was covered extensively in previous chapters, clustering is based on the assumption that there is no information about class labels to guide the process of grouping. In other words, clustering is an unsupervised classification of observations. Broadly speaking, “exclusive” (also known as “hard”) clustering techniques can be divided into *partitional* and *hierarchical*. Partitional techniques produce one single partition of data whereas hierarchical techniques create a nested sequence of partitions. The term exclusive refers to the fact that in each partition, each data item belongs to exactly one subset of that partition. This is in contrast with non-exclusive (fuzzy) clustering methods where each data item can have a degree of membership to each cluster. In this chapter, we will focus on some of the most popular techniques for partitional and hierarchical clustering and leave discussion of fuzzy clustering entirely to other resources.

12.1 Partitional Clustering

Let $\mathbf{S}_n = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ denote a given set of observations (training data) where $\mathbf{x}_i \in \mathbb{R}^p, i = 1, \dots, n$, represents a vector including the values of p feature variables, and n is the number of observations. Partitional clustering breaks \mathbf{S}_n into K non-empty subsets C_1, \dots, C_K such that

- $C_i \cap C_j = \emptyset, i \neq j, i, j = 1, \dots, K,$
- $\cup_{i=1}^K C_i = \mathbf{S}_n,$

where \cap and \cup denote intersection and union operators, respectively. The value of K may or may not be specified. The set $\mathcal{P} = \{C_1, \dots, C_K\}$ is known as a *clustering* and each C_i is a *cluster*. A partitional *clustering algorithm* produces a clustering when presented with data, and this is regardless of whether data actually contains any cluster or not. Therefore, it is the responsibility of the user to examine whether

there is any merit to clustering before performing the analysis. Furthermore, the lack of information about class labels makes clustering essentially a subjective process in nature. The same data items (observations) can be clustered differently for different purposes. For example, consider the following set of four data items: $S_4 = \{\text{Cherry, Mint, Mango, Chive}\}$. Depending on the definition of *clustering criterion*, below are some possible clusterings of S_4 :

clustering criterion	clustering
word initial	$\{\text{Cherry, Chive}\}, \{\text{Mint, Mango}\}$
word length	$\{\text{Mint}\}, \{\text{Chive, Mango}\}, \{\text{Cherry}\}$
botanical distinction	$\{\text{Cherry, Mango}\}, \{\text{Chive, Mint}\}$
color	$\{\text{Cherry}\}, \{\text{Mint, Chive}\}, \{\text{Mango}\}$

This shows the importance of incorporating domain knowledge in clustering. Such a knowledge can be encoded in the form of: 1) feature vectors that are used to represent each data item; 2) measures of similarity or dissimilarity between feature vectors; 3) grouping schemes (clustering algorithms) that are used to produce clusters from feature vectors; and 4) ultimate interpretation, confirmation, or rejection of the results obtained by a clustering algorithm.

Example 12.1 List all possible clusterings of a given data $S_3 = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$ by a partitional clustering algorithm.

All possible partitions include

$$\begin{aligned} \mathcal{P}_1 &= \{ \{\mathbf{x}_1\}, \{\mathbf{x}_2\}, \{\mathbf{x}_3\} \}, \\ \mathcal{P}_2 &= \{ \{\mathbf{x}_1, \mathbf{x}_2\}, \{\mathbf{x}_3\} \}, \\ \mathcal{P}_3 &= \{ \{\mathbf{x}_1, \mathbf{x}_3\}, \{\mathbf{x}_2\} \}, \\ \mathcal{P}_4 &= \{ \{\mathbf{x}_2, \mathbf{x}_3\}, \{\mathbf{x}_1\} \}, \\ \mathcal{P}_5 &= \{ \{\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_1\} \}. \end{aligned}$$



Example 12.2 Suppose in Example 12.1, we receive an observation \mathbf{x}_4 and would like to use some of the existing clusterings $\mathcal{P}_i, i = 1, \dots, 5$, to produce all possible clusterings of $S_4 = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}$ into 3 clusters. How can we do this?

There are two ways:

1) to add \mathbf{x}_4 as a singleton cluster (i.e., a cluster with only one data item) to those clusterings $\mathcal{P}_i, i = 1, \dots, 5$, that have two clusters; that is to say, finding the union of $\{\{\mathbf{x}_4\}\}$ with $\mathcal{P}_2, \mathcal{P}_3$, and \mathcal{P}_4 :

$$\begin{aligned} &\{ \{\mathbf{x}_1, \mathbf{x}_2\}, \{\mathbf{x}_3\}, \{\mathbf{x}_4\} \}, \\ &\{ \{\mathbf{x}_1, \mathbf{x}_3\}, \{\mathbf{x}_2\}, \{\mathbf{x}_4\} \}, \\ &\{ \{\mathbf{x}_2, \mathbf{x}_3\}, \{\mathbf{x}_1\}, \{\mathbf{x}_4\} \}. \end{aligned}$$

2) to add \mathbf{x}_4 to each cluster member of any clustering that already has three clusters; that is to say,

$$\begin{aligned} &\{\{\mathbf{x}_1, \mathbf{x}_4\}, \{\mathbf{x}_2\}, \{\mathbf{x}_3\}\}, \\ &\{\{\mathbf{x}_1\}, \{\mathbf{x}_2, \mathbf{x}_4\}, \{\mathbf{x}_3\}\}, \\ &\{\{\mathbf{x}_1\}, \{\mathbf{x}_2\}, \{\mathbf{x}_3, \mathbf{x}_4\}\}. \end{aligned}$$

Therefore, there are in total six ways to partition 4 observations into 3 clusters. ■

Example 12.2 sheds light on a systematic way to count the number of all possible clusterings of n observations into K clusters, denoted $S(n, K)$. Suppose we list all possible clusterings of \mathbf{S}_{n-1} , which is a set containing $n - 1$ observations. There are two ways to form K clusters of $\mathbf{S}_n = \mathbf{S}_{n-1} \cup \{\mathbf{x}_n\}$:

1. to add \mathbf{x}_n as a singleton cluster to each clustering of \mathbf{S}_{n-1} with $K - 1$ clusters—there are $S(n - 1, K - 1)$ of such clusterings;
2. to add \mathbf{x}_n to each cluster member of any clustering that already has K clusters—there are $S(n - 1, K)$ of such clusterings, and each has K clusters.

Therefore,

$$S(n, K) = S(n - 1, K - 1) + KS(n - 1, K). \quad (12.1)$$

The solution to this difference equation is Stirling number of the second kind (see (Jain and Dubes, 1988) and references therein):

$$S(n, K) = \frac{1}{K!} \sum_{i=0}^K (-1)^{K-i} \binom{K}{i} i^n. \quad (12.2)$$

This number grows quickly as a function of n and K . For example, $S(30, 5) > 10^{18}$. As a result, even if K is fixed *a priori*, it is impractical to devise a clustering algorithm that enumerates all possible clusterings to find the most sensible one. Therefore, clustering algorithms only evaluate a small “reasonable” fraction of all possible clusterings to achieve at a sensible solution. One of the simplest and most popular partitional clustering techniques is K -means (Lloyd, 1982; Forgy, 1965; MacQueen, 1967), which is described next.

12.1.1 K -Means

The standard form of K -means clustering algorithm, also known as Lloyd’s algorithm (Lloyd, 1982), is implemented in the algorithm presented next.

In (Selim and Ismail, 1984), it is shown that the K -means algorithm can be formulated as a nonconvex mathematical programming problem where a local minimum need not be the global minimum. That being said, it is shown that the algorithm converges to a local minimum for the criterion defined in (12.3); however, if $d_E^2[\mathbf{x}_{i,j}, \boldsymbol{\mu}_j]$

Algorithm K-Means

1. Set iteration counter $t = 1$, fix K , and randomly designate K data points as the K cluster centroids (means).
2. Assign each data point to the cluster with the nearest centroid.
3. Compute the *squared error* (also known as *within-cluster sum of squares* or *inertia*) criterion. In particular, given \mathbf{S}_n and a clustering \mathcal{P} , the squared error, denoted $e(\mathbf{S}_n, \mathcal{P})$, is given by

$$e(\mathbf{S}_n, \mathcal{P}) = \sum_{j=1}^K \sum_{i=1}^{n_j} d_E^2[\mathbf{x}_{i,j}, \boldsymbol{\mu}_j], \quad (12.3)$$

where $\mathbf{x}_{i,j}$ is the i^{th} data point belonging to cluster j having current mean $\boldsymbol{\mu}_j$ and n_j data points, and $d_E^2[\mathbf{x}, \mathbf{y}]$ is the square of the Euclidean distance between two vectors \mathbf{x} and \mathbf{y} , which is a *dissimilarity* metric (see Section 5.1.3 for the definition of Euclidean distance).

4. Recompute the centroid for each cluster using the current data points belonging to that cluster, and set $t \leftarrow t + 1$.
5. Repeat steps 2 to 4 until a stopping criterion is met. In particular, the algorithms stops if:
 - no data points change clusters; or
 - a maximum number of iterations is reached; or
 - there is no significant decrease in $e(\mathbf{S}_n, \mathcal{P})$; that is, the change in $e(\mathbf{S}_n, \mathcal{P})$ across two consecutive iterations is less than a prespecified value.

is replaced by Minkowski metric then the algorithm may converge to a Kuhn-Tucker point that need not be a local minimum either. Nonetheless, even for the criterion (12.3), a major drawback is the sensitivity of K -means to initial selection of centroids. In particular, for different initial selections, the algorithm may converge to different local minima, which leads to different clusterings. As a result, it is typical to run the algorithm a few times and take the solution that corresponds to the least value for the criterion.

Another approach that can possibly help with initialization of centroids is to keep them sufficiently apart. In this regard, in (Arthur and Vassilvitskii, 2007) an algorithm, namely, K -means++, is proposed that initially designate a data point as a centroid with a probability that is proportional to its distance from the nearest centroid that is already chosen. In particular, K -means++ is implemented in the following algorithm.

Algorithm K-Means++

1. Fix K and randomly designate a data point as the first cluster centroid.
2. Designate $\mathbf{x}_i \in \mathbf{S}_n, i = 1, \dots, n$, as the next cluster centroid with the probability $\frac{d_E^2[\mathbf{x}_i]}{\sum_{i=1}^n d_E^2[\mathbf{x}_i]}$ where $d_E[\mathbf{x}_i]$ denotes the Euclidean distance of \mathbf{x}_i to its nearest centroid that is already chosen.
3. Repeat step 2 until K centroids are chosen.
4. proceed with standard K -means.

Example 12.3 Consider six data points that are identified in Fig. 12.1 with \bullet points. Apply K -means clustering with $K = 2$ and take data points at $[0, 1]^T$ and $[0, 0]^T$ as the initial centroids of the two clusters. Stop the algorithm either once no data points change clusters or there is no change in the squared error.

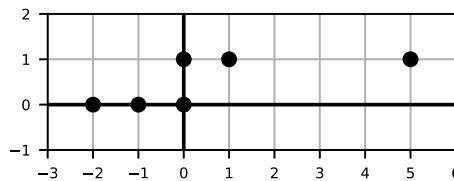


Fig. 12.1: Six data points used in Example 12.3.

Let $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_5$, and \mathbf{x}_6 represent data points at $[0, 1]^T, [1, 1]^T, [5, 1]^T, [0, 0]^T, [-1, 0]^T$, and $[-2, 0]^T$, respectively.

For $t = 1$:

- step 2: $\mathbf{x}_1, \mathbf{x}_2$, and \mathbf{x}_3 are assigned to cluster 1, denoted C_1 , with the centroid $\mu_1 = \mathbf{x}_1$, and $\mathbf{x}_4, \mathbf{x}_5$, and \mathbf{x}_6 are assigned to cluster 2, denoted C_2 , with the centroid $\mu_2 = \mathbf{x}_4$ (see Fig. 12.2a);
- step 3: $e(\mathbf{S}_n, \mathcal{P}) = 25 + 1 + 1 + 4 = 31$;
- step 4: recomputing μ_1 and μ_2 yields $\mu_1 = [2, 1]^T$ and $\mu_2 = [-1, 0]^T$.

For $t = 2$:

- step 2: \mathbf{x}_2 and \mathbf{x}_3 are assigned to C_1 , and $\mathbf{x}_1, \mathbf{x}_4, \mathbf{x}_5$, and \mathbf{x}_6 are assigned to C_2 (see Fig. 12.2b);
- step 3: $e(\mathbf{S}_n, \mathcal{P}) = 9 + 1 + (\sqrt{2})^2 + 1 + 1 = 14$;
- step 4: recomputing μ_1 and μ_2 yields $\mu_1 = [3, 1]^T$ and $\mu_2 = [-3/4, 1/4]^T$.

For $t = 3$:

- step 2: \mathbf{x}_3 is assigned to C_1 , and all other points are assigned to C_2 (see Fig. 12.2c);
- step 3:

$$\begin{aligned} e(\mathbf{S}_n, \mathcal{P}) &= 4(\text{distance of } \mathbf{x}_3 \text{ from } \boldsymbol{\mu}_1) + [0.75^2 + 0.75^2](\text{distance of } \mathbf{x}_1 \text{ from } \boldsymbol{\mu}_2) \\ &\quad + [1.75^2 + 0.75^2](\text{distance of } \mathbf{x}_2 \text{ from } \boldsymbol{\mu}_2) + [0.75^2 + 0.25^2](\text{distance of } \mathbf{x}_4 \text{ from } \boldsymbol{\mu}_2) \\ &\quad + [0.25^2 + 0.25^2](\text{distance of } \mathbf{x}_5 \text{ from } \boldsymbol{\mu}_2) + [1.25^2 + 0.25^2](\text{distance of } \mathbf{x}_6 \text{ from } \boldsymbol{\mu}_2) \\ &= 11.125; \end{aligned}$$

- step 4: recomputing $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ yields $\boldsymbol{\mu}_1 = [-0.4, 0.4]^T$ and $\boldsymbol{\mu}_2 = [5, 1]$.

For $t = 4$:

- step 2: \mathbf{x}_3 is assigned to C_1 , and all other points are assigned to C_2 (see Fig. 12.2d); Here we stop the search because no data point change clusters. We compute $e(\mathbf{S}_n, \mathcal{P}) = 6.4$, which is the squared error of the final clustering.
- step 4: recomputing $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ does not change .

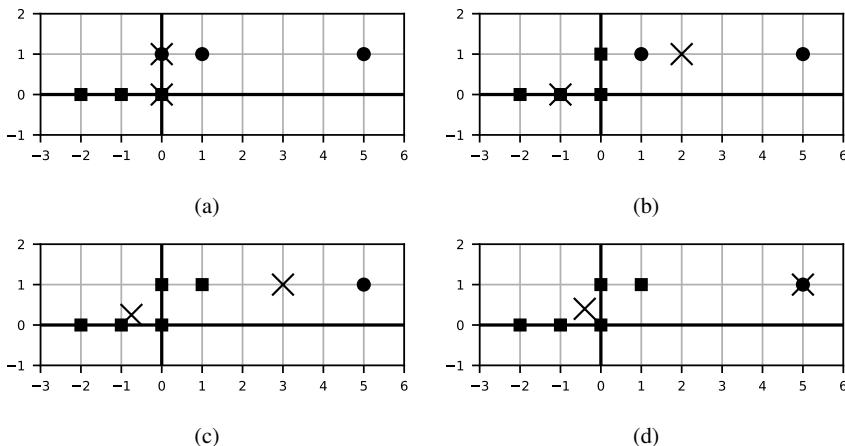


Fig. 12.2: The clustering in Example 3 produced by K -means at: (a) $t = 1$; (b) $t = 2$; (c) $t = 3$; and (d) $t = 4$; \bullet and \blacksquare represent points belonging to C_1 and C_2 , respectively. A \times identifies a cluster centroid.

Therefore, the identified partition is $\{ \{ \mathbf{x}_3 \}, \{ \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_4, \mathbf{x}_5, \mathbf{x}_6 \} \}$. ■

Scikit-learn implementation of K -means: K -means is implemented by `KMeans` class from `sklearn.cluster` module. By default, scikit-learn sets the number of clusters, K , to 8 but this choice can be changed by setting the value of `n_clusters`

parameter. The `init` parameter can be used to set the centroids initialization to '`k-means++`' (default), '`random`' (for random assignments of data points to centroids), or specific points (a specific array) specified by the user. By default (as of version 1.2.2), the algorithm will run 10 times (can be changed by `n_init`) and the best clustering in terms of having the lowest within-cluster sum of squares is used. Because a `KMeans` object is an estimator, similar to other estimators has a `fit()` method that can be used to construct the K -means clusterer. At the same time, `KMeans` implements the `predict()` method to assign data points to clusters. To do so, an observation is assigned to the cluster with the nearest centroid. Nevertheless, the clustering of training data (assigned cluster labels from 0 to $K - 1$) can be retrieved by the `labels_` attribute of a trained `KMeans` object and the centroids are obtained by `cluster_centers_` attribute. Last but not least, `KMeans` implements the `score(X)` method, which returns the negative of squared error computed for a given data \mathbf{X} and the identified centroids of clusters that are already part of a trained `KMeans` object; that is to say, given a data \mathbf{X} , it computes $-e(\mathbf{X}, \mathcal{P})$ defined in (12.3) where \mathcal{P} is already determined from the clustering algorithm. Here, the negative is used to have a score (the higher, the better) as compared with loss. Nonetheless, because (12.3) is not normalized with respect to the number of data points in \mathbf{X} (similar to the current implementation of the `score(X)` method), for a fixed \mathcal{P} and two datasets \mathbf{X}_1 and \mathbf{X}_2 where $\mathbf{X}_1 \subset \mathbf{X}_2$, $-e(\mathbf{X}_1, \mathcal{P}) > -e(\mathbf{X}_2, \mathcal{P})$.

Remarks on K -Means Functionality: The use of squared error makes large-scale features to dominate others. As a result, it is quite common to normalize data before applying K -means (see Section 4.6 for scaling techniques). At the same time, the use of the centroids to represent clusters naturally works well when the actual underlying clusters of data, if any, are isolated or compactly isotropic. As a result, when the actual clusters are elongated and rather mixed, K -means algorithm may not identify clusters properly. The situation is illustrated in the following example.

Example 12.4 In this example, we wish to train a K -means clusterer by treating the Iris training dataset that was prepared (normalized) in Section 4.6 as an unlabeled training data. Similar to Example 6.1, we only consider two features: sepal width and petal length. As we already know the actual “natural” clusters in this data, which are setosa, versicolor, and virginica Iris flowers, we set $K = 3$. Furthermore, we compare the clustering of unlabeled data with the actual groupings to have a sense of designated clusters.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.cluster import KMeans

# loading the scaled data and selecting the two features
arrays = np.load('data/iris_train_scaled.npz')
X_train = arrays['X']
y_train = arrays['y']
```

```

print('X shape = {}'.format(X_train.shape))
X_train = X_train[:,[1,2]] # selecting the two features

# necessities for plotting cluster boundaries
color = ('bisque', 'aquamarine', 'lightgrey')
cmap = ListedColormap(color)
mins = X_train.min(axis=0) - 0.1
maxs = X_train.max(axis=0) + 0.1
x = np.arange(mins[0], maxs[0], 0.01)
y = np.arange(mins[1], maxs[1], 0.01)
X, Y = np.meshgrid(x, y)
coordinates = np.array([X.ravel(), Y.ravel()]).T
fig, axs = plt.subplots(1, 2, figsize=(6, 2), dpi = 150, sharex=True, ↵
    ↵sharey=True)
fig.tight_layout()

# training the clusterer
K = 3
kmeans = KMeans(n_clusters=K, random_state=42)
kmeans.fit(X_train)

# plotting the cluster boundaries and the scatter plot of training data
Z = kmeans.predict(coordinates)
Z = Z.reshape(X.shape)
axs[0].tick_params(axis='both', labelsize=6)
axs[0].pcolormesh(X, Y, Z, cmap = cmap, shading='nearest')
axs[0].contour(X ,Y, Z, colors='black', linewidths=0.5)
axs[0].plot(X_train[:, 0], X_train[:, 1], 'k.', markersize=4)
axs[0].set_title('$K$=' + str(K), fontsize=8)
axs[0].set_ylabel('petal length (normalized)', fontsize=7)
axs[0].set_xlabel('sepal width (normalized)', fontsize=7)

axs[1].tick_params(axis='both', labelsize=6)
axs[1].pcolormesh(X, Y, Z, cmap = cmap, shading='nearest')
axs[1].contour(X ,Y, Z, colors='black', linewidths=0.5)
axs[1].plot(X_train[y_train==0, 0], X_train[y_train==0, 1], 'g.', ↵
    ↵markersize=4)
axs[1].plot(X_train[y_train==1, 0], X_train[y_train==1, 1], 'r.', ↵
    ↵markersize=4)
axs[1].plot(X_train[y_train==2, 0], X_train[y_train==2, 1], 'k.', ↵
    ↵markersize=4)
axs[1].set_title('$K$=' + str(K), fontsize=8)
axs[1].set_xlabel('sepal width (normalized)', fontsize=7)

print('The score for K={} is {:.3f}'.format(K, kmeans.score(X_train)))

```

X shape = (120, 4)
 The score for K=3 is -57.201

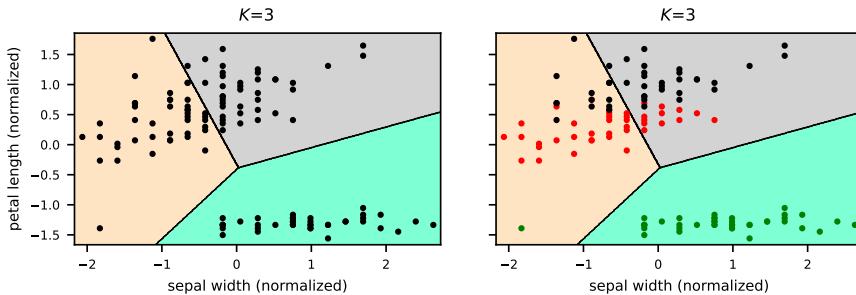


Fig. 12.3: The scatter plot of normalized Iris dataset (sepal width and petal length features) and cluster boundaries determined with K -means for $K = 3$. The plot on the left shows unlabeled data. The plot on the right shows the actual grouping of data points, which is known for the Iris dataset: green (setosa), red (versicolor), and black (virginica). Comparing plots on the left and the right shows that the K -means has been able to cluster green data points pretty well but the other two groups are quite mixed within identified clusters.

As we can see in Fig. 12.3, despite the elongated spread of green points, they are quite isolated from others and are identified as one cluster by K -means. However, the other two groups are quite mixed within identified clusters. For the sake of comparison, in Fig. 12.4 we plot the cluster boundaries obtained by two other combination of features, namely, petal width and petal length combination. Here all clustered are compactly isotropic and K -means works pretty well in clustering data points from all actual groups. ■

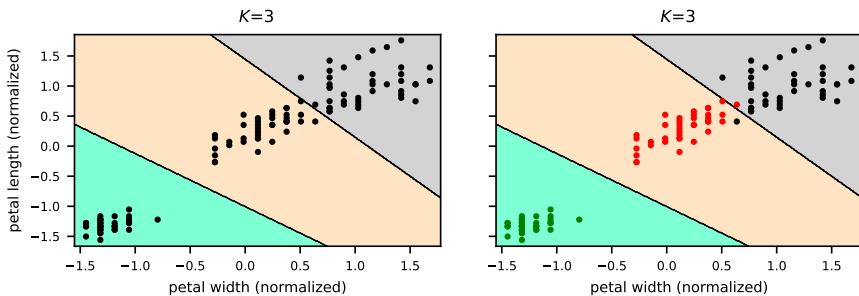


Fig. 12.4: The scatter plot of normalized Iris dataset (petal width and petal length) and cluster boundaries determined with K -means for $K = 3$. The plot on the left shows unlabeled data. The plot on the right shows the actual grouping of data points, which is known for the Iris dataset: green (setosa), red (versicolor), and black (virginica). Comparing plots on the left and the right shows that the K -means has performed well in clustering data points from all actual groups.

12.1.2 Estimating the Number of Clusters

So far we assumed the number of clusters, K , is known. However, in many situations, this is not known in advance. At the same time, the quality of clusters determined by K -means, similar to many other clustering algorithms, heavily depends on this parameter. Consider Example 12.4 in which K was set to 3 because we already knew that there are three types of Iris flowers in the data. However, in the absence of prior knowledge on the number of clusters, we should estimate K from data. Fig. 12.5 shows the identified clusters (for the case of sepal width and petal length features) if we set $K = 2$ and $K = 4$. As we can see in this figure, when $K = 2$ (underestimating K), one of the clusters completely mixes data points for Iris versicolor and virginica types. At the same time, for $K = 4$ (overestimating K), two clusters are created for Iris virginica flowers, which (in the absence of any biological justification) does not make sense. In practice, under- and over-estimating K render interpretation of results challenging. Two popular data-driven techniques to estimate K are based on the *elbow phenomenon* and the *silhouette coefficient*.

Elbow Phenomenon: The elbow phenomenon can be traced back to the work of Thorndike (Thorndike, 1953) and later Kruskal (Kruskal, 1964). Consider the K -means clustering that minimizes the squared error. There is clearly a trade-off between large and small number of clusters. For $K = 1$, there is a large squared error, which is proportional to the trace of the sample covariance matrix. On the other hand, setting K to the number of data points results in 0 squared error. In general, plotting squared error as a function of K results in a monotonically decreasing curve between these two extremes. The elbow phenomenon asserts that sometimes the curve of squared error (in general the clustering loss function) as a function of K

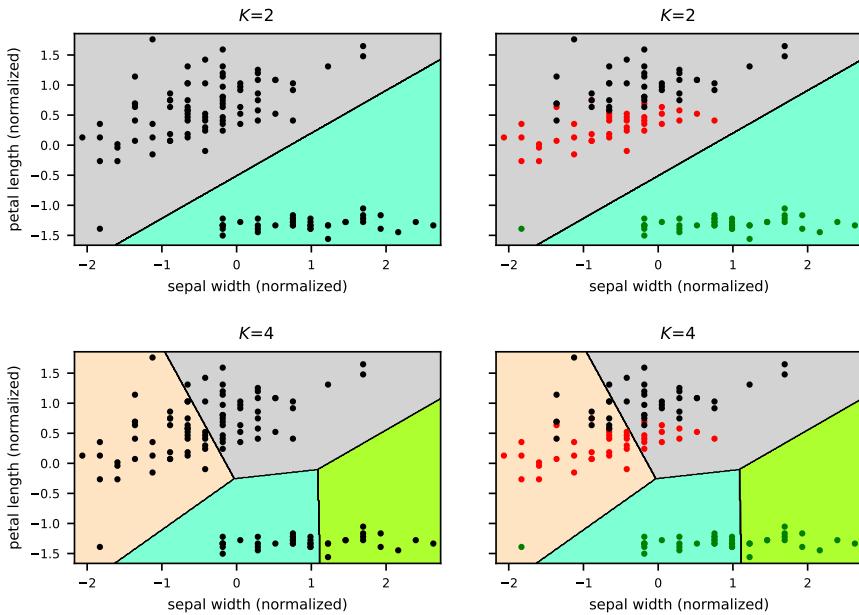


Fig. 12.5: The scatter plot of normalized Iris dataset (sepal width and petal length features) and cluster boundaries determined with K -means for: $K = 2$ (top row) and $K = 4$ (bottom row). The left panel shows unlabeled data and the right panel shows shows the actual grouping of data points, which is known for the Iris dataset: green (setosa), red (versicolor), and black (virginica).

does not change slope from some K onwards (thus creating the shape of an “elbow” in the curve). Such a point in the curve might be an indicator of the optimal number of clusters because the following small decreases are simply due to decrease in partition radius rather than data configuration. In the method based on the elbow phenomenon (“elbow method”), we plot the squared error for a range of K , which is determined *a priori*. Then the method designates the K after which the curve does not considerably change slope as the “appropriate” number of clusters. Naturally, such a K is acceptable as long as it is not widely in odds with the domain knowledge. We examine the method in the following example.

Example 12.5 Here we examine the elbow method for the K -means and the two feature combinations in the Iris flower datasets that were considered in Example 12.4: Case 1) sepal width and petal length; and Case 2) petal width and petal length. To plot the curve of squared error as a function of K , we simply run K -means algorithm for a range of K and plot `-kmeans.score(X_train)` (see Example 4).

In Case 1 (left plot in Fig. 12.6), elbow method points to $K = 2$. This is the points after which the slope of the curve does not change markedly. This choice would become more sensible by looking back more closely at Figs. 12.3 and 12.5. As we

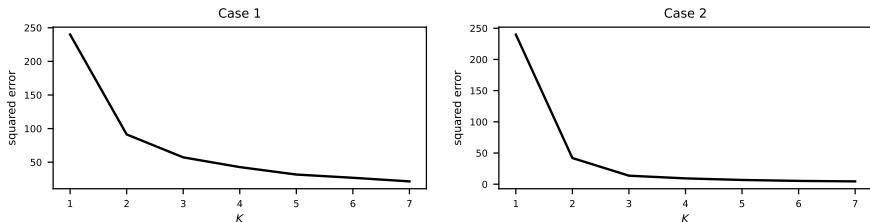


Fig. 12.6: Curve of squared error of K -means as a function of K for (unlabeled) Iris flower datasets: Case 1) sepal width and petal length features (left); and Case 2) petal width and petal length (right).

observed in Fig. 12.5, for $K = 2$ data points are grouped into two distinct clusters by K -means. It happens that one cluster (at the bottom of the figure) is populated by Iris setosa flowers (green points), and the other (on top of the figure) is mainly a mixture of Iris versicolor and virginica. However, when $K = 3$ in Fig. 12.3, one of the clusters is still quite isolated but the other two seem neither compactly isotropic nor isolated. Interestingly, looking into the types of Iris flowers within these two clusters also show that these two clusters are still quite mixed between Iris versicolor and virginica flowers.

In Case 2 (right plot in Fig. 12.6), however, it seems more reasonable to consider $K = 3$ as the right number of clusters because the slope of the curve does not change considerably after that point. In this case, when $K = 2$, there are still two distinct clusters similar to Case 1 (figure not shown): one cluster at the bottom left corner and one on the top right corner. However, for $K = 3$, the newly created clusters on top right corner are quite distinct in the left panel of Fig. 12.4: one cluster is characterized by petal width and petal length in the range of $[-0.27, 0.76]$ and $[-0.26, 0.74]$, respectively, whereas the other cluster has a relatively higher range for these two features. Here, it happens that we can associate some botanical factors to these clusters—each cluster is distinctly populated by one type of Iris flower. ■

Silhouette Coefficient: Silhouettes were proposed by Rousseeuw (Rousseeuw, 1987) to judge the relative isolation and compactness of clusters. The theory is used to define the *overall average silhouette width* and *silhouette coefficient* that can be used for *cluster validity analysis* and estimating an “appropriate” number of clusters. In this section, we elaborate on these concepts.

Similar to the quantitative evaluation of a classification rule, cluster validity is a quantitative evaluation of a clustering algorithm. In this regard, sometimes it is helpful to treat a labeled data as an unlabeled one and compare performance of clustering algorithms with respect to the actual clustering within the data. However, in many real scenarios, the major challenge for cluster validity is the lack of ground truth; that is to say, there is no information on the actual groupings of the given data; after all, if this ground truth was known, there was no need to perform clustering. As a result, to define a measure of cluster validity that can still operate in these

common situations, we have to rely on heuristic reasoning. For example, we may intuitively agree that data points that are part of a *valid* cluster should be more similar to each other than to data points from other clusters. This has been an assumption behind some of the popular cluster validity measures such as the *silhouette coefficient* (Kaufman and Rousseeuw, 1990).

As before, let $\mathbf{x}_{i,j}$ denote the i^{th} data point belonging to cluster j , denoted C_j , that has n_j data points. For each $\mathbf{x}_{i,j}$, we can define a *silhouette width* $s(\mathbf{x}_{i,j})$ such that

$$s(\mathbf{x}_{i,j}) = \frac{b(\mathbf{x}_{i,j}) - a(\mathbf{x}_{i,j})}{\max\{a(\mathbf{x}_{i,j}), b(\mathbf{x}_{i,j})\}}, \quad (12.4)$$

where

$$a(\mathbf{x}_{i,j}) = \frac{1}{n_j - 1} \sum_{\mathbf{x}_{l,j} \in C_j - \{\mathbf{x}_{i,j}\}} d[\mathbf{x}_{i,j}, \mathbf{x}_{l,j}], \quad (12.5)$$

$$b(\mathbf{x}_{i,j}) = \min_{r \neq j} \left\{ \frac{1}{n_r} \sum_{\mathbf{x}_{l,r} \in C_r} d[\mathbf{x}_{i,j}, \mathbf{x}_{l,r}] \right\}, \quad (12.6)$$

and where $d[\mathbf{x}_{i,j}, \mathbf{x}_{l,j}]$ is a dissimilarity measure between vectors $\mathbf{x}_{i,j}$ and $\mathbf{x}_{l,j}$ (e.g., Euclidean distance between two vectors). For a singleton cluster C_j , $s(\mathbf{x}_{i,j})$ is defined 0 (Rousseeuw, 1987). Assuming $n_j > 1, \forall j$, then the value of $a(\mathbf{x}_{i,j})$ shows the average dissimilarity of $\mathbf{x}_{i,j}$ from all other members of the cluster to which it is assigned. In Fig. 12.7, this is the average length of all lines from the blue point in cluster C_1 to all other points in the same cluster (average of all solid lines). To find $b(\mathbf{x}_{i,j})$ in (12.6), we successively compute the average dissimilarities of $\mathbf{x}_{i,j}$ to all objects from other clusters and find the minimum. In Fig. 12.7, this means we find the average length of all lines from the blue point in C_1 to all members of cluster C_2 (i.e., the average of all dotted lines), and then the average length of all lines from the blue point to all members of cluster C_3 (i.e., the average of all dashed lines). Because the former average is less than the latter, $b(\mathbf{x}_{i,j})$ (for the blue point) becomes the average of all dotted lines. This means that $b(\mathbf{x}_{i,j})$ is a measure of dissimilarity between $\mathbf{x}_{i,j}$ and all members of the second-best cluster (after C_j) for $\mathbf{x}_{i,j}$. This also implies that to compute $b(\mathbf{x}_{i,j})$, we need to assume availability of two clusters at least.

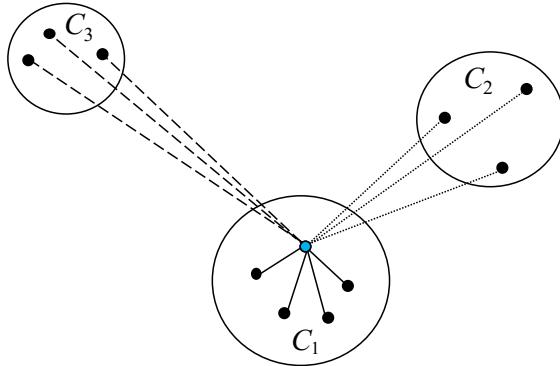


Fig. 12.7: Assuming $\mathbf{x}_{i,1}$ is the blue point in C_1 , then: 1) $a(\mathbf{x}_{i,1})$ is the average length of all lines from the blue point to all other points in C_1 (i.e., the average of all solid lines); and 2) $b(\mathbf{x}_{i,1})$ is the average length of all lines from the blue point to all members of cluster C_2 , which is the second-best (“neighboring”) cluster for $\mathbf{x}_{i,1}$ (i.e., the average of all dotted lines).

From the definition of $s(\mathbf{x}_{i,j})$ in (12.4), it is clear that $s(\mathbf{x}_{i,j})$ lies between -1 and 1. We can consider three cases, which help understand the meaning of $s(\mathbf{x}_{i,j})$:

- $b(\mathbf{x}_{i,j}) >> a(\mathbf{x}_{i,j})$: it means the average dissimilarity of $\mathbf{x}_{i,j}$ to other data points in its own cluster is much smaller than its average dissimilarity to objects in the second-best cluster for $\mathbf{x}_{i,j}$. This implies that cluster C_j is the very right cluster for $\mathbf{x}_{i,j}$. In this case, from (12.4), $s(\mathbf{x}_{i,j})$ is close to 1.
- $b(\mathbf{x}_{i,j}) << a(\mathbf{x}_{i,j})$: it means the average dissimilarity of $\mathbf{x}_{i,j}$ to other data points in its own cluster is much larger than its average dissimilarity to objects in the second-best cluster for $\mathbf{x}_{i,j}$. This implies that cluster C_j is not the right cluster for $\mathbf{x}_{i,j}$. In this case, from (12.4), $s(\mathbf{x}_{i,j})$ is close to -1.
- $b(\mathbf{x}_{i,j}) \approx a(\mathbf{x}_{i,j})$: this is the case when in terms of average dissimilarities, assigning $\mathbf{x}_{i,j}$ to C_j or its second-best cluster does not make much difference. In this case, from (12.4), $s(\mathbf{x}_{i,j})$ is close to 0.

The values $s(\mathbf{x}_{i,j})$ for data points in cluster C_j are used to define cluster-specific *average silhouette width*, denoted $\bar{s}(C_j)$, $j = 1, \dots, K$, which is given by

$$\bar{s}(C_j) = \frac{1}{n_j} \sum_{\mathbf{x}_{i,j} \in C_j} s(\mathbf{x}_{i,j}). \quad (12.7)$$

A value of $\bar{s}(C_j)$ close to 1 implies having a distinct cluster C_j . Finally, the *overall average silhouette width*, denoted $\bar{s}(K)$, is defined as the average of all values $s(\mathbf{x}_{i,j})$ across all clusters, which is equivalent to:

$$\bar{s}(K) = \frac{1}{\sum_{j=1}^K n_j} \sum_{j=1}^K n_j \bar{s}(C_j). \quad (12.8)$$

The value of $\bar{s}(K)$ can be seen as the average “within” and “between” dissimilarities of data points in a given clustering (partition). At the same time, one way to select an “appropriate” number of clusters is to find the K that maximizes $\bar{s}(K)$. This maximum value is known as *silhouette coefficient* (Kaufman and Rousseeuw, 1990, p. 87), denoted SC, and is given by

$$SC = \max_K \bar{s}(K). \quad (12.9)$$

In (Kaufman and Rousseeuw, 1990, p. 87), authors state that SC is a “measure of the amount of clustering structure that has been discovered” by the clustering algorithm. They also propose a subjective interpretation of SCs, which is summarized in Table 12.1 (see (Kaufman and Rousseeuw, 1990, p. 88)):

Table 12.1: Subjective interpretation of silhouette coefficient

SC range	Subjective Interpretation
0.71 - 1.00	A ‘strong’ clustering structure has been discovered
0.51 - 0.70	A ‘reasonable’ clustering structure has been discovered
0.26 - 0.50	A ‘weak’ clustering structure has been discovered
≤ 0.25	‘No substantial’ clustering structure has been discovered



As pointed out by Rousseeuw (Rousseeuw, 1987), silhouette width has the advantage that it only depends on the *outcome* of a clustering algorithm (i.e., partition of data points) and for its computation we do not even need to know the algorithm that has used to achieve the partitions. As a result, silhouette width could also be used to “modify” the result of a given clustering—for example, by moving a data point with negative silhouette width to its neighboring cluster (see Exercise 6).

Scikit-learn implementation of the overall average silhouette width: The silhouette width $s(\mathbf{x}_{i,j})$, which is defined for each data point in a training set, is obtained by the `silhouette_samples` function from the `sklearn.metrics` module. By default, scikit-learn sets the metric to 'euclidean' but, if desired, this can be changed by setting `metric` parameter, for example, to other choices listed at (Scikit-silhouette, 2023). The overall average silhouette width is obtained by `sklearn.metrics.silhouette_score` function. Both classes require a partition of training data, which is given by `labels` parameter—this could be, for example, the `labels_` attribute of a trained KMeans object.



Currently, there is a discrepancy between the terminology put forward by Kaufman and Rousseeuw in (Kaufman and Rousseeuw, 1990) and the scikit-learn documentation. For example, the description of `sklearn.metrics.silhouette_score` function currently says “Compute the mean Silhouette Coefficient of all samples”. In the terminology put forward by Kaufman and Rousseeuw (Kaufman and Rousseeuw, 1990, p. 86–87), this is the overall average silhouette width defined in (12.8), whereas the silhouette coefficient is defined over all K as given in (12.9).

Example 12.6 Here we consider the K -means and the two feature combinations in the Iris flower datasets that were used in Example 12.5: Case 1) sepal width and petal length; and Case 2) petal width and petal length. We find an appropriate value for K from overall average silhouette width.

```

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

# loading the scaled data, and setting feature indices and range of K
arrays = np.load('data/iris_train_scaled.npz')
X_train_full = arrays['X']
feature_indices = [[1,2], [3,2]]
K_val = range(2,8)

for j, f in enumerate(feature_indices):
    X_train = X_train_full[:,f] # selecting the two features

    # necessities for plotting cluster boundaries
    fig, ax = plt.subplots(1, 1, figsize=(4, 2), dpi = 150)
    fig.tight_layout()

    # training the clusterer and computing the overall average
    # silhouette width for each K
    silh = np.zeros(len(K_val))
    for i, K in enumerate(K_val):
        kmeans = KMeans(n_clusters=K, random_state=42)
        kmeans.fit(X_train)
        silh[i] = silhouette_score(X_train, kmeans.labels_)

    # plotting the the overall average silhouette width as a function
    # of K
    ax.tick_params(axis='both', labelsize=6)

```

```

ax.plot(K_val, silh, 'k-', markersize=4)
ax.set_ylabel(r'$\bar{s}(K)$', fontsize=7)
ax.set_xlabel('$K$', fontsize=7)
ax.set_title('Case ' + str(j+1), fontsize=8)

```

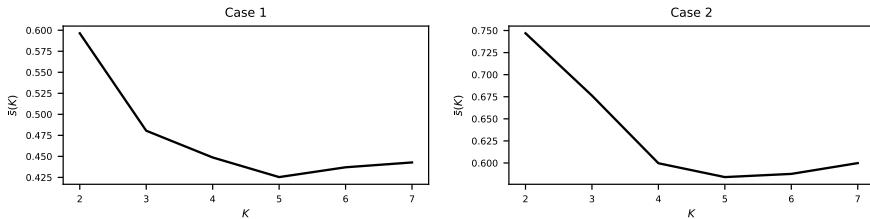


Fig. 12.8: The overall average silhouette width as a function of K for clusters obtained using K -means in the (unlabeled) Iris flower datasets: Case 1) sepal width and petal length features (left); and Case 2) petal width and petal length (right).

Here the use of overall average silhouette width points to $K = 2$ as the appropriate number of clusters in both cases. Furthermore, for Case 1 and Case 2, we have $SC = 0.59$ and $SC = 0.74$, respectively. Based on Table 12.1, this means for Case 1 and Case 2, ‘reasonable’ and ‘strong’ clustering structures have been discovered, respectively. ■

12.2 Hierarchical Clustering

In contrast with partitional clustering algorithms that create a partition of data points, hierarchical clustering algorithms create a nested sequence of partitions. Given two clusterings \mathcal{P}_1 and \mathcal{P}_2 of the same data items with K_1 and K_2 clusters, respectively, where $K_2 < K_1$, we say \mathcal{P}_1 is *nested* in \mathcal{P}_2 , denoted $\mathcal{P}_1 \sqsubset \mathcal{P}_2$, if each cluster in \mathcal{P}_1 is a subset of a cluster in \mathcal{P}_2 . This also implies that each cluster in \mathcal{P}_2 is obtained by merging some clusters in \mathcal{P}_1 .

Example 12.7 Which of the following clusterings are nested in partition \mathcal{P} where

$$\mathcal{P} = \{ \{\mathbf{x}_1, \mathbf{x}_4, \mathbf{x}_5\}, \{\mathbf{x}_2, \mathbf{x}_3\}, \{\mathbf{x}_6\} \} ?$$

$$\begin{aligned}\mathcal{P}_1 &= \{ \{\mathbf{x}_1, \mathbf{x}_4\}, \{\mathbf{x}_5\}, \{\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_6\} \}, \\ \mathcal{P}_2 &= \{ \{\mathbf{x}_1, \mathbf{x}_4, \mathbf{x}_6\}, \{\mathbf{x}_2, \mathbf{x}_3\} \}, \\ \mathcal{P}_3 &= \{ \{\mathbf{x}_1\}, \{\mathbf{x}_2\}, \{\mathbf{x}_3\}, \{\mathbf{x}_4\}, \{\mathbf{x}_5\}, \{\mathbf{x}_6\} \}, \\ \mathcal{P}_4 &= \{ \{\mathbf{x}_1, \mathbf{x}_2\}, \{\mathbf{x}_3\}, \{\mathbf{x}_4\}, \{\mathbf{x}_5\}, \{\mathbf{x}_6\} \}, \\ \mathcal{P}_5 &= \{ \{\mathbf{x}_1, \mathbf{x}_5\}, \{\mathbf{x}_2, \mathbf{x}_3\}, \{\mathbf{x}_4\}, \{\mathbf{x}_6\} \} .\end{aligned}$$

From the definition of nested clusterings, only \mathcal{P}_3 and \mathcal{P}_5 are nested in \mathcal{P} . ■

Hierarchical clustering algorithms are mainly classified into *agglomerative* and *divisive* methods. Agglomerative methods start with assigning each data item to a singleton cluster, and follow by successively merging clusters until a stopping criterion is met. In this regard, in each iteration, the *least dissimilar pair of clusters* from the previous iteration are merged to form a new cluster. Let \mathcal{P}_t denote the clustering obtained in iteration t . Clearly, an agglomerative method creates a sequence of clusterings such that $\mathcal{P}_1 \sqsubset \mathcal{P}_2 \sqsubset \mathcal{P}_3 \sqsubset \dots$. The stopping criterion can be a prespecified number of clusters to achieve.

Divisive methods, on the other hand, start with assigning all data items to a single cluster, and follow by successively splitting clusters until a stopping criterion is met. In this regard, in each iteration, the cluster that has the *highest dissimilar pair of clusters* in its two-subset partition is replaced by that partition. As a result, a divisive method creates a sequence of clusterings such that $\dots \sqsubset \mathcal{P}_3 \sqsubset \mathcal{P}_2 \sqsubset \mathcal{P}_1$.

Although it seems like the difference between agglomerative and divisive methods is as simple as changing the search direction, it turns out that divisive methods are computationally much more demanding than agglomerative methods. To see this, consider the transition from the first iteration to the second in both methods. The most straightforward way to merge the two most similar singleton clusters of an agglomerative method, is to compute the dissimilarity of every pairs of these clusters and merge those with the lowest dissimilarity. This requires computing and comparing $\binom{n}{2}$ pairwise dissimilarity functions between clusters where n is the number of data points. For example, when $n = 30$, we have $\binom{30}{2} = 435$. However, to replace the single cluster of a divisive method with its best two-subset partition, we need to compute and compare $2^{n-1} - 1$ pairwise dissimilarity functions between clusters (this can be seen from (12.2) by setting $K = 2$). To compare, when $n = 30$, we have $2^{30-1} - 1 = 536,870,911$! For this reason, agglomerative methods are considerably more popular than divisive methods and in what follows we focus on these methods.

Developments of agglomerative methods hinges upon several factors that are discussed next:

1. definition of pairwise cluster dissimilarity; and
2. efficiently updating dissimilarities.

12.2.1 Definition of Pairwise Cluster Dissimilarity

Dissimilarity between clusters C_i and C_j , denoted $d[C_i, C_j]$, is defined based on the measure of dissimilarity between data points (feature vectors) belonging to them. Let $d[\mathbf{x}, \mathbf{y}]$ denote a dissimilarity measure between two p -dimensional vectors \mathbf{x} and \mathbf{y} . Several *dissimilarity function*¹ between clusters can be defined. In particular, for two given clusters C_i and C_j with n_i and n_j data points, respectively, we have

- *min dissimilarity function*:

$$d_{\min}[C_i, C_j] = \min_{\mathbf{x} \in C_i, \mathbf{y} \in C_j} d[\mathbf{x}, \mathbf{y}]; \quad (12.10)$$

- *max dissimilarity function*:

$$d_{\max}[C_i, C_j] = \max_{\mathbf{x} \in C_i, \mathbf{y} \in C_j} d[\mathbf{x}, \mathbf{y}]; \quad (12.11)$$

- *average dissimilarity function*:

$$d_{\text{avg}}[C_i, C_j] = \frac{1}{n_i n_j} \sum_{\mathbf{x} \in C_i} \sum_{\mathbf{y} \in C_j} d[\mathbf{x}, \mathbf{y}]; \quad (12.12)$$

- *Ward (minimum variance) dissimilarity function*:

$$d_{\text{Ward}}[C_i, C_j] = \frac{n_i n_j}{n_i + n_j} d_E^2[\boldsymbol{\mu}_i, \boldsymbol{\mu}_j], \quad (12.13)$$

where $d_E^2[\mathbf{x}, \mathbf{y}]$ is the square of the Euclidean distance between two vectors \mathbf{x} and \mathbf{y} , and $\boldsymbol{\mu}_i$ and $\boldsymbol{\mu}_j$ are centroids of C_i and C_j , respectively.

12.2.2 Efficiently Updating Dissimilarities

Another important factor in development of an agglomerative method is to be able to efficiently update cluster dissimilarities in each iteration. In this regard, efficiency has two aspects:

1. at each iteration, we should avoid recomputing cluster dissimilarity function for clusters that have not been affected in the current iteration. For example, suppose in an iteration, a partition \mathcal{P} includes four clusters: C_1 , C_2 , C_3 , and C_4 , and the dissimilarities between all pairs of these clusters are available. At the same time,

¹ They are called *functions* because strictly speaking, there are cases that even for a *metric* $d[\mathbf{x}, \mathbf{y}]$, the dissimilarity of sets is neither a metric nor a *measure* (see (Theodoridis and Koutroumbas, 2009, pp. 620-621))

we decide to merge C_1 and C_4 to form a cluster, $C_1 \cup C_4$, because they are the least dissimilar. In the next iteration, that we have C_2 , C_3 , and $C_1 \cup C_4$, we should not recompute $d[C_2, C_3]$ because it remains the same as in the previous iteration; and

2. at each iteration, we should be able to compute the dissimilarity between a newly formed cluster with all other clusters, from the dissimilarity between the two merged clusters (that form the new cluster) and their dissimilarities with other clusters. In the above example, this means that $d[C_1 \cup C_4, C_2]$ should be computed from $d[C_1, C_2]$, $d[C_4, C_2]$, and $d[C_1, C_4]$, which are already available. In general, assuming C_i and C_j are merged, for $i, j \neq k$ we should have

$$d[C_i \cup C_j, C_k] = f(d[C_i, C_k], d[C_j, C_k], d[C_i, C_j]), \quad (12.14)$$

where $f(., .)$ is a function (update rule) that depends on the form of dissimilarity functions defined in (12.10)-(12.13). The choice of this update rule, which is indeed dictated by the choice of pairwise cluster dissimilarity function, leads to different agglomerative algorithms. Although there are a number of different update rules (thus, algorithms), the four most popular update rules are created by the use of the four dissimilarity functions introduced in (12.10)-(12.13).

Single linkage: Suppose we take the min dissimilarity function defined in (12.10) as the pairwise cluster dissimilarity. In that case, the dissimilarity update rule (12.14) becomes

$$d_{\min}[C_i \cup C_j, C_k] = \min\{d_{\min}[C_i, C_k], d_{\min}[C_j, C_k]\}. \quad (12.15)$$

It is straightforward to show (12.15) from (12.10). Suppose $C_i = \{\mathbf{x}_{1,i}, \dots, \mathbf{x}_{n_i,i}\}$, $C_j = \{\mathbf{x}_{1,j}, \dots, \mathbf{x}_{n_j,j}\}$, and $C_k = \{\mathbf{x}_{1,k}, \dots, \mathbf{x}_{n_k,k}\}$. We assume $d_{\min}[C_i, C_k]$ is obtained for pairs $\mathbf{x}_{r,i}$ and $\mathbf{x}_{s,k}$; that is, $d_{\min}[C_i, C_k] = d_{\min}[\mathbf{x}_{r,i}, \mathbf{x}_{s,k}]$. Furthermore, $d_{\min}[C_j, C_k]$ is obtained for pairs $\mathbf{x}_{t,j}$ and $\mathbf{x}_{u,k}$; that is, $d_{\min}[C_j, C_k] = d_{\min}[\mathbf{x}_{t,j}, \mathbf{x}_{u,k}]$. Taking $C_i \cup C_j$ does not alter any elements in C_i and C_j . Therefore, $d_{\min}[C_i \cup C_j, C_k] = \min\{d_{\min}[\mathbf{x}_{r,i}, \mathbf{x}_{s,k}], d_{\min}[\mathbf{x}_{t,j}, \mathbf{x}_{u,k}]\}$, which is the same as (12.15). An algorithm that uses the dissimilarity update rule (12.15) is known as *minimum* algorithms (Johnson, 1967), *nearest-neighbor* methods, or more famously as *single linkage* methods (Hastie et al., 2001; Jain and Dubes, 1988; Theodoridis and Koutroumbas, 2009).

Complete linkage: Suppose we take the max dissimilarity function defined in (12.11) as the pairwise cluster dissimilarity. It is straightforward to show that the dissimilarity update rule (12.14) becomes

$$d_{\max}[C_i \cup C_j, C_k] = \max\{d_{\max}[C_i, C_k], d_{\max}[C_j, C_k]\}. \quad (12.16)$$

An algorithms that uses the dissimilarity update rule (12.15) is known as *maximum* algorithms (Johnson, 1967), *furthest-neighbor* methods, or more famously as *complete linkage* methods (Hastie et al., 2001; Jain and Dubes, 1988; Theodoridis and Koutroumbas, 2009).

Group average linkage (“unweighted pair group method using arithmetic averages”): Suppose we take the average dissimilarity function defined in (12.12) as the pairwise cluster dissimilarity. In that case, the dissimilarity update rule (12.14) becomes

$$d_{\text{avg}}[C_i \cup C_j, C_k] = \frac{n_i}{n_i + n_j} d_{\text{avg}}[C_i, C_k] + \frac{n_j}{n_i + n_j} d_{\text{avg}}[C_j, C_k]. \quad (12.17)$$

This is because

$$\begin{aligned} d_{\text{avg}}[C_i \cup C_j, C_k] &= \frac{1}{(n_i + n_j)n_k} \sum_{x \in C_i \cup C_j} \sum_{y \in C_k} d_{\text{avg}}[\mathbf{x}, \mathbf{y}] \\ &= \frac{1}{(n_i + n_j)n_k} \left[\sum_{x \in C_i} \sum_{y \in C_k} d_{\text{avg}}[\mathbf{x}, \mathbf{y}] + \sum_{x \in C_j} \sum_{y \in C_k} d_{\text{avg}}[\mathbf{x}, \mathbf{y}] \right] \\ &= \frac{1}{(n_i + n_j)n_k} \left[n_i n_k d_{\text{avg}}[C_i, C_k] + n_j n_k d_{\text{avg}}[C_j, C_k] \right], \end{aligned} \quad (12.18)$$

which proves (12.17). An algorithm that uses the dissimilarity update rule (12.17) is known as *unweighted pair group method using arithmetic averages* (UPGMA). The UPGMA (also known as *group average linkage*) is a specific type of *average linkage* methods. Other average linkage methods are defined based on various forms of “averaging”. A detailed representation of these algorithms are provided in (Sneath and Sokal, 1973; Jain and Dubes, 1988). Nevertheless, due to popularity of UPGMA, sometimes in clustering literature “average linkage” and UPGMA are used interchangeably.

Ward’s (minimum variance) linkage: Suppose we take the Ward dissimilarity function defined in (12.13) as the pairwise cluster dissimilarity. Then the dissimilarity update rule (12.14) becomes (see Exercise 5)

$$\begin{aligned} d_{\text{Ward}}[C_i \cup C_j, C_k] &= \frac{n_i + n_k}{n_i + n_j + n_k} d_{\text{Ward}}[C_i, C_k] \\ &\quad + \frac{n_j + n_k}{n_i + n_j + n_k} d_{\text{Ward}}[C_j, C_k] - \frac{n_k}{n_i + n_j + n_k} d_{\text{Ward}}[C_i, C_j]. \end{aligned} \quad (12.19)$$

The algorithm that uses the dissimilarity update rule (12.19) is known as Ward’s linkage method (Ward, 1963).

In (Lance and Williams, 1967), it was shown that all the aforementioned update rules (as well as several others that are not covered here) are special forms of the following general update rule:

$$d[C_i \cup C_j, C_k] = \alpha_i d[C_i, C_k] + \alpha_j d[C_j, C_k] + \beta d[C_i, C_j] + \gamma |d[C_i, C_k] - d[C_j, C_k]|, \quad (12.20)$$

where coefficients α_i , α_j , β , and γ depend on the linkage, and $d[C_i \cup C_j, C_k]$ is the corresponding dissimilarity function (see Table 12.2).

linkage	α_i	α_j	β	γ
single	$\frac{1}{2}$	$\frac{1}{2}$	0	$\frac{-1}{2}$
complete	$\frac{1}{2}$	$\frac{1}{2}$	0	$\frac{1}{2}$
group average	$\frac{n_i}{n_i+n_j}$	$\frac{n_j}{n_i+n_j}$	0	0
Ward	$\frac{n_i+n_k}{n_i+n_j+n_k}$	$\frac{n_j+n_k}{n_i+n_j+n_k}$	$\frac{-n_k}{n_i+n_j+n_k}$	0

Table 12.2: Coefficients of the general update rule (12.20) for various linkage methods.

One way to consider both the aforementioned aspects of efficiency is to build a *dissimilarity matrix* and update the matrix at each iteration. At each iteration $t = 1, 2, \dots$, the dissimilarity matrix, denoted \mathbf{D}_t , is (in general²) a symmetric $(n - t + 1) \times (n - t + 1)$ matrix that includes the dissimilarities between all cluster pairs that are available in that iteration; that is, the element on the i^{th} row and j^{th} column of \mathbf{D}_t is $d_t[C_i, C_j]$, $1 \leq i, j \leq (n - t + 1)$, and $d_t[C_i, C_i] = 0$. Once the two least (distinct) dissimilar clusters are identified and merged, the two rows and columns corresponding to the two merged clusters are removed. Instead, one row and column corresponding to the dissimilarity of the newly formed cluster with all other clusters is added to the matrix to form \mathbf{D}_{t+1} . In what follows, we refer to this procedure as “updating \mathbf{D}_t ” (to transit from iteration t to $t + 1$). Methods that use such an updating scheme for dissimilarity matrix are known as *matrix updating algorithms* (MUA). Naturally, they can be used with any of the aforementioned linkage methods. Implementation of the MUA is presented in the next algorithm.

12.2.3 Representing the Results of Hierarchical Clustering

Interpreting the results of hierarchical clustering is much easier by a graphical representation of the nested sequence of clusterings. In this regard, the results are generally structured in a special type of tree known as *dendrogram*. A dendrogram contains a node at level t that represents a cluster created at iteration t of the algorithm. Lines that connect nodes in this tree represent nested clusterings. At the same time, it is common to draw a dendrogram in the dissimilarity scale used to create the clusters. Fig. 12.9 shows a dendrogram for five data points: $S_5 = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_5\}$. The figure shows that the dissimilarity of clusters $\{\{\mathbf{x}_2\}\}$ and $\{\{\mathbf{x}_3\}\}$ have been the least and

² see ([Vicari, 2014](#)) and references therein for some cases of asymmetric dissimilarity matrices

Algorithm Matrix Updating Algorithm

1. Set $t = 1$ and create the first partition \mathcal{P}_1 by assigning each data point to a cluster; that is, $\mathcal{P}_1 = \{\{\mathbf{x}_1\}, \{\mathbf{x}_2\}, \dots, \{\mathbf{x}_n\}\}$. Construct the dissimilarity matrix \mathbf{D}_t , which at this state is a $n \times n$ matrix, choose a “linkage” (one of the dissimilarity updating rules (12.15), (12.16), (12.17), or (12.19)), and a stopping criterion.
2. Identify the pair of clusters with the least dissimilarity; that is, the tuple (i, j) such that (here we assume there is no tie to identify the minimum in (12.21); otherwise, a tie breaking strategy should be used and that, in general, can affect the outcome):

$$(i, j) = \underset{\substack{1 \leq m, l \leq n-t+1 \\ m \neq l}}{\operatorname{argmin}} d_t[C_m, C_l]. \quad (12.21)$$

3. Merge clusters C_i and C_j , update the partition: $\mathcal{P}_t \leftarrow (\mathcal{P}_t - \{C_i, C_j\}) \cup \{C_i \cup C_j\}$, update \mathbf{D}_t , and set $t \leftarrow t + 1$.
4. Repeat steps 2 and 3 until the stopping criterion is met—a common stopping criterion is to merge all data points into a single cluster. Once the search is stopped, the sequence of obtained clusterings are $\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_t$, which satisfy $\mathcal{P}_1 \sqsubset \mathcal{P}_2 \sqsubset \dots \sqsubset \mathcal{P}_t$.

they were merged at level (iteration) 1. “Cutting” the dendrogram at (right above) level 1, leads to the clustering $\{\{\mathbf{x}_2, \mathbf{x}_3\}, \{\mathbf{x}_1\}, \{\mathbf{x}_4\}, \{\mathbf{x}_5\}\}$. This cut is shown by the horizontal dashed line right above 0.4. In the second iteration, the least dissimilarity was due to clusters $\{\{\mathbf{x}_2, \mathbf{x}_3\}\}$ and $\{\{\mathbf{x}_4\}\}$ and, as the result, they were merged at level 2. Cutting the dendrogram at this level leads to $\{\{\mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}, \{\mathbf{x}_1\}, \{\mathbf{x}_5\}\}$. This is identified by the horizontal dashed line right above 0.8.

Example 12.8 Suppose $\mathbf{S}_8 = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_8\}$ where $\mathbf{x}_1 = [1.5, 1]^T$, $\mathbf{x}_2 = [1.5, 2]^T$, $\mathbf{x}_3 = [1.5, -1]^T$, $\mathbf{x}_4 = [1.5, -2]^T$, $\mathbf{x}_5 = -\mathbf{x}_1$, $\mathbf{x}_6 = -\mathbf{x}_2$, $\mathbf{x}_7 = -\mathbf{x}_3$, and $\mathbf{x}_8 = -\mathbf{x}_4$. Apply the matrix updating algorithm with both the single linkage and complete linkage until there are two clusters. Use Euclidean distance metric as the measures of dissimilarity between data points.

Fig. 12.10a shows the scatter plot of these data points. We first consider the single linkage. In this regard, using Euclidean distance metric, \mathbf{D}_1 becomes (because it is symmetric we only show the upper triangular elements):

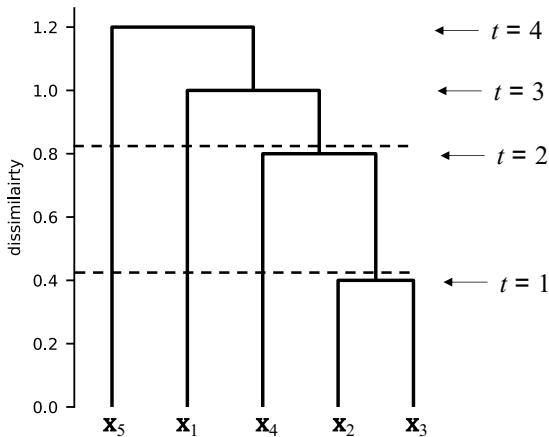


Fig. 12.9: A dendrogram for an agglomerative clustering of five data points.

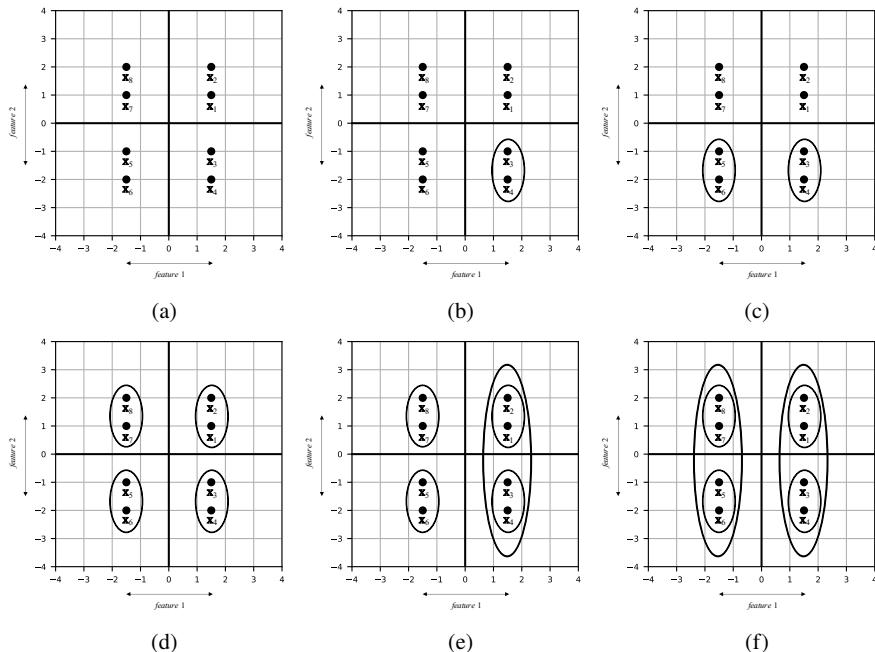


Fig. 12.10: The scatter plot of data points in Example 12.8, and merged clusters at iteration: (b) 1; (c) 2; (d) 4; (e) 5; and (f) 6

$$\mathbf{D}_1 = \begin{bmatrix} 0 & 1.00 & 2.00 & 3.00 & 3.60 & 4.24 & 3.00 & 3.16 \\ . & 0 & 3.00 & 4.00 & 4.24 & 5.00 & 3.16 & 3.00 \\ . & . & 0 & 1.00 & 3.00 & 3.16 & 3.60 & 4.24 \\ . & . & . & 0 & 3.16 & 3.00 & 4.24 & 5.00 \\ . & . & . & . & 0 & 1.00 & 2.00 & 3.00 \\ . & . & . & . & . & 0 & 3.00 & 4.00 \\ . & . & . & . & . & . & 0 & 1.00 \\ . & . & . & . & . & . & . & 0 \end{bmatrix}. \quad (12.22)$$

There are multiple pairs of clusters that minimize (12.21). Although we break the ties randomly, here due to symmetry of the problem, the outcome at the stopping point will remain the same no matter how the ties are broken.

For $t = 1$, the (selected) pair of clusters that minimizes (12.21) is $(3, 4)$ —at this stage they are simply $\{\{\mathbf{x}_3\}\}$ and $\{\{\mathbf{x}_4\}\}$. Therefore, these two clusters are merged to form a new cluster (see Fig. 12.10b), and \mathbf{D}_1 is updated to obtain \mathbf{D}_2 (at $t = 2$):

$$\mathbf{D}_2 = \begin{bmatrix} 0 & 1.00 & 2.00 & 3.60 & 4.24 & 3.00 & 3.16 \\ . & 0 & 3.00 & 4.24 & 5.00 & 3.16 & 3.00 \\ . & . & 0 & 3.00 & 3.00 & 3.60 & 4.24 \\ . & . & . & 0 & 1.00 & 2.00 & 3.00 \\ . & . & . & . & 0 & 3.00 & 4.00 \\ . & . & . & . & . & 0 & 1.00 \\ . & . & . & . & . & . & 0 \end{bmatrix}. \quad (12.23)$$

For $t = 2$, the (selected) pair of clusters that minimizes (12.21) is $(5, 6)$ —at this stage they are simply $\{\{\mathbf{x}_5\}\}$ and $\{\{\mathbf{x}_6\}\}$. Therefore, these two clusters are merged to form a new cluster (see Fig. 12.10c), and \mathbf{D}_2 is updated to obtain \mathbf{D}_3 :

$$\mathbf{D}_3 = \begin{bmatrix} 0 & 1.00 & 2.00 & 3.60 & 3.00 & 3.16 \\ . & 0 & 3.00 & 4.24 & 3.16 & 3.00 \\ . & . & 0 & 3.00 & 3.60 & 4.24 \\ . & . & . & 0 & 2.00 & 3.00 \\ . & . & . & . & 0 & 1.00 \\ . & . & . & . & . & 0 \end{bmatrix}. \quad (12.24)$$

Similarly, at $t = 3$ and $t = 4$, clusters $\{\{\mathbf{x}_1\}\}$ and $\{\{\mathbf{x}_2\}\}$, and $\{\{\mathbf{x}_7\}\}$ and $\{\{\mathbf{x}_8\}\}$ are merged, respectively (see Fig. 12.10d). As the result, the dissimilarity at iteration 5 is

$$\mathbf{D}_5 = \begin{bmatrix} 0 & 2.00 & 3.60 & 3.00 \\ . & 0 & 3.00 & 3.60 \\ . & . & 0 & 2.00 \\ . & . & . & 0 \end{bmatrix}. \quad (12.25)$$

For $t = 5$, the (selected) pair of clusters that minimizes (12.21) are the two clusters on the right part of Fig. 12.10e that are merged. \mathbf{D}_5 is updated to obtain \mathbf{D}_6 :

$$\mathbf{D}_6 = \begin{bmatrix} 0 & 3.00 & 3.00 \\ . & 0 & 2.00 \\ . & . & 0 \end{bmatrix}. \quad (12.26)$$

For $t = 6$, the pair of clusters that minimizes (12.21) are the two clusters on the left part of Fig. 12.10f that are merged. \mathbf{D}_6 is updated to obtain \mathbf{D}_7 :

$$\mathbf{D}_7 = \begin{bmatrix} 0 & 3.00 \\ . & 0 \end{bmatrix}. \quad (12.27)$$

At this stage, we have two clusters and the stopping criterion is met. Thus, we stop the algorithm. Fig. 12.10f shows the nested sequence of clusterings obtained by the algorithm and Fig. 12.11 shows its corresponding dendrogram.

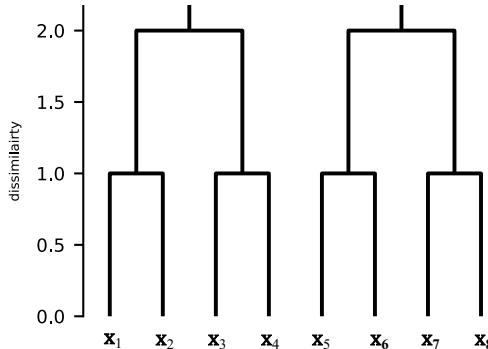


Fig. 12.11: The dendrogram for the single linkage clustering implemented in Example 12.8.

Repeating the aforementioned steps for the complete linkage results in Fig. 12.12 (steps are omitted and are left as an exercise [Exercise 4]). ■

Single linkage vs. complete linkage vs. Ward's linkage vs. group average linkage: In the single linkage clustering, two clusters are merged if a “single” dissimilarity (between their data points) is small regardless of dissimilarity of other observations within the two clusters. This causes grouping very dissimilar data points at early stages (low dissimilarities in the dendrogram) if there is some *chain of pairwise rather similar data points* between them (Hansen and Delattre, 1978). This is known as *chaining effect* and causes clusters produced by single linkage to be long and “straggly” (Sneath and Sokal, 1973; Jain and Dubes, 1988). This is in contrast with the complete linkage and the Ward's linkage that create clusters that are

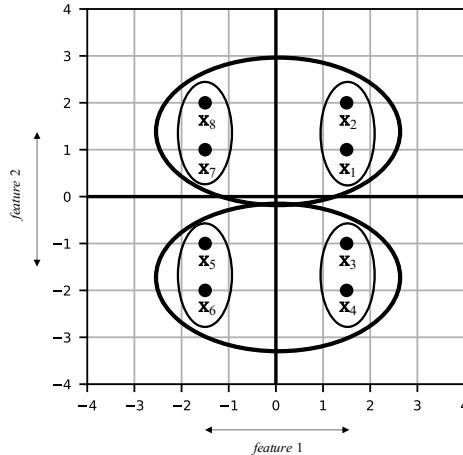


Fig. 12.12: Complete linkage clustering of data points used in Example 12.8.

relatively compact and hyperspherical ([Jain and Dubes, 1988](#)). In the case of complete linkage clustering, this is an immediate consequence of the max dissimilarity function used in the complete linkage: two clusters are merged if *all pairs of data points* that are part of them are “relatively similar”. This causes these observations be generally grouped at higher dissimilarities in the dendrogram. The group average linkage seems to have an intermediate effect between the single linkage and the complete linkage clusterings.

There are several Monte Carlo studies to compare the rate at which these methods recover some known clusters in synthetically generated data. In this regard, the results of ([Blashfield, 1976](#)) obtained by multivariate normally distributed data indicated that the Ward’s linkage method appeared to be the best followed by the complete linkage, then the group average linkage, and finally, by the single linkage. The results also show that in all cases, the “recovery rate” was negatively affected by the ellipticity of clusters. The results of ([Kuiper and Fisher, 1975](#)) using multivariate normal distributions with identity covariance matrices generally corroborate the results of ([Blashfield, 1976](#)) by placing the Ward’s linkage and the complete linkage as the best and runner up methods, and the single linkage as the poorest. Another study generated null data sets using univariate uniform and normal distributions and examined the performance of the methods (excluding the Ward’s method) in terms of not discovering unnecessary clusters ([Jain et al., 1986](#)). The results of this study placed the complete linkage as the best method, followed by the single linkage, and then by the group average. In contrast with the previous methods using normally distributed data, another work studied the recovery rate of these methods for synthetically generated data sets that satisfy the ultrametric inequality ([Milligan and Isaac, 1980](#)). The results placed the group average linkage first, followed by the complete linkage, then the Ward’s method, and finally the single linkage as

the poorest. There are many other studies that are not covered here. Despite all these comparative analyses and recommendations, the consensus conclusion is that there is no single best clustering approach. Different methods are good for different applications (Jain and Dubes, 1988). For example, if there is *a priori* information that there are elongated clusters in the data, the single linkage chaining “defect” can be viewed as an advantage to discover such cluster structures.

Python implementation of agglomerative clustering: There are two common ways to implement agglomerative clustering in Python. One is based on the scikit-learn `AgglomerativeClustering` class from `sklearn.cluster` module, and the other is based on the `linkage` function from `scipy.cluster.hierarchy` module. In the first approach, similar to any other scikit-learn estimator, we can instantiate the class and use the `fit()` method to train the estimator. Two stopping criteria that can be used are `n_clusters` (number of clusters to be found [at the level that the algorithm stops]) and `distance_threshold` (the dissimilarity at or above which no merging occurs). The type of linkage and the distance metric between data points can be set by the `linkage` (possible choices: 'ward', 'complete', 'average', 'single') and the `metric` (e.g., 'euclidean') parameters, respectively. Cluster labels can be found by the `labels_` attribute of a trained `AgglomerativeClustering` object. In the following code, we use this approach for both the single and the complete linkage clusterings of data points presented in Example 12.8.

```
import numpy as np
from sklearn.cluster import AgglomerativeClustering

X1 = np.array([[1.5, 1], [1.5, 2], [1.5, -1], [1.5, -2]])
X2=-X1
X = np.concatenate((X1, X2))
clustering = AgglomerativeClustering(n_clusters=2, linkage='single')
clustering.fit(X)
print('clustering for single linkage: ' + str(clustering.labels_))
clustering = AgglomerativeClustering(n_clusters=2, linkage='complete')
clustering.fit(X)
print('clustering for complete linkage: ' + str(clustering.labels_))
```

```
clustering for single linkage: [1 1 1 1 0 0 0 0]
clustering for complete linkage: [1 1 0 0 0 0 1 1]
```

One way to use the `linkage` function from `scipy.cluster.hierarchy` module for agglomerative clustering is to directly use the training data as the input to this function. The type of linkage and the distance metric between data points can be set by the `method` (with choices such as 'single', 'complete', 'average', 'ward') and the `metric` parameters, respectively. The output of `linkage` is in the form of a numpy array, known as “linkage matrix”, that in each “row” shows: 1) the two merged clusters (the first and second “columns” of the matrix); 2) the

dissimilarity for merged clusters (the third column); and 3) and the number of data points in the newly formed cluster (the fourth column). It is common to use this numpy array as the input to `scipy.cluster.hierarchy.dendrogram` function to plot the clustering dendrogram. In the following code, we plot the dendrogram for single linkage clustering in Example 12.8.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import dendrogram, linkage

X1 = np.array([[1.5, 1], [1.5, 2], [1.5, -1], [1.5, -2]])
X2=-X1
X = np.concatenate((X1, X2))
plt.figure(figsize=(4, 4), dpi = 150)
Z = linkage(X, method='single', metric='euclidean')
dendrogram(Z, above_threshold_color='k')
plt.ylabel('dissimilarity', fontsize=7)
plt.tick_params(axis='both', labelsize=7)
print("linkage matrix=\n" + str(Z))
```

```
linkage matrix=
[[ 0.  1.  1.  2.]
 [ 2.  3.  1.  2.]
 [ 4.  5.  1.  2.]
 [ 6.  7.  1.  2.]
 [ 8.  9.  2.  4.]
 [10. 11.  2.  4.]
 [12. 13.  3.  8.]]
```

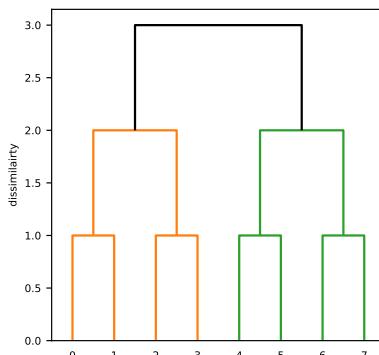


Fig. 12.13: The dendrogram for the single linkage clustering of data points used in Example 12.8 (the data points indices are presented next to the leaf nodes in the tree).

Exercises:

Exercise 1: Generate six sets of training data such that each training data is produced by setting a constant μ to one of the values $\{3, 1.5, 1.2, 1.1, 0.9, 0.0\}$ and then randomly sampling four bivariate Gaussian distributions with an identical identity covariance matrix and means: $\mu_0 = [\mu, \mu]^T$, $\mu_1 = [-\mu, \mu]^T$, $\mu_2 = [-\mu, -\mu]^T$, and $\mu_3 = [\mu, -\mu]^T$. Generate 1000 observations from each distribution, and then concatenate them to create each training data (thus each training data has a size of 4000×2). The training data that is created in this way has naturally four distinct clusters when $\mu = 3$ and becomes a “null” data (all data points from a single cluster) when $\mu = 0$.

- (A) Show the scatter plots of each training data, and plot the curve of overall average silhouette width as a function of K for $K = 2, 3, \dots, 40$. Describe your observations about the behavior of the tail of these curves as μ decreases.
- (B) What is the estimated optimal number of clusters based on the silhouette coefficient in each case?
- (C) What is the silhouette coefficients for the case where $\mu = 3$? What is the interpretation of this value based on Table 12.1?

Exercise 2: Apply the single linkage and the complete linkage clustering methods to the Iris training dataset that was used in Example 12.4 in this chapter. By merely looking at the dendrogram, which method, in your opinion, has produced “healthier” clusters?

Exercise 3: Use scikit-learn to apply K -means with $K = 2$ to data points used in Example 12.3. Use the same initialization as Example 12.3, obtain the squared error

at each iteration, and observe that the achieved squared errors (inertia) are the same as the hand calculations obtained in Example 12.3. (Hint: set `verbose` parameter to 1 to observe inertia).

Exercise 4: Use hand calculations to apply the matrix updating algorithm with the complete linkage to the data presented in Example 12.8.

⊕ **Exercise 5:** Prove the dissimilarity update rule presented in (12.19) for Ward's dissimilarity function.

Exercise 6: Suppose K -means with $K = 2$ is applied to a dataset and two clusters, namely, C_1 and C_2 , are obtained. In this setting, the value of the silhouette width for an observation that belongs to C_1 is s . What will be the silhouette width if we decide to consider this observation as part of C_2 instead of C_1 ?

- A) $s/2$ B) s C) $-s$ D) 0

Exercise 7: A clustering algorithm has produced the two clusters depicted in Fig. 12.14 where points belonging to cluster 1 and cluster 2 (a singleton cluster) are identified by • and ×, respectively. Using Euclidean distance as the measure of dissimilarity between data points, what is the overall average silhouette width for this clustering (use hand calculations)?

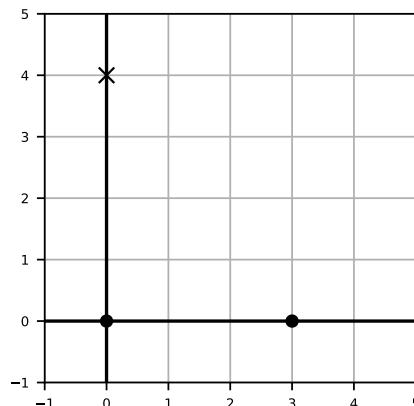


Fig. 12.14: Data points for Exercise 7

- A) $\frac{13}{10}$ B) $\frac{13}{20}$ C) $\frac{13}{30}$ D) $\frac{13}{40}$ E) $\frac{13}{60}$



Chapter 13

Deep Learning with Keras-TensorFlow

Deep learning is a subfield of machine learning and has been applied in various tasks such as supervised, unsupervised, semi-supervised, and reinforcement learning. Among all family of predictive models that are used in machine learning, by deep learning we exclusively refer to a particular class of models known as multilayered Artificial Neural Network (ANN), which are partially inspired by our understanding of biological neural circuits. Generally, an ANN is considered *shallow* if it has one or two layers; otherwise, it is considered a *deep* network. Each layer in an ANN performs some type of transformation of the input to the layer. As a result, multiple successive layers in an ANN could potentially capture and mathematically represent complex input-output relationship in a given data. In this chapter, we introduce some of the key principles and practices used in learning deep neural networks. In this regard, we use multi-layer perceptrons as a typical ANN and postpone other architectures to later chapters. In terms of software, we switch to Keras with TensorFlow backend as they are well-optimized for training and tuning various forms of ANN and support various forms of hardware including CPU, GPU, or TPU.

13.1 Artificial Neural Network, Deep Learning, and Multilayer Perceptron

ANNs use a set of linear and nonlinear operations to implement a meaningful mapping between a set of features and target variables. The number of layers of successive transformations used in a network is referred to as its depth; thus, the more layers, the deeper the network. The goal in deep learning is to use ANNs of multiple layers that can extract underlying patterns of data at increasingly more levels of abstraction. As a result, deep neural networks do not generally require a separate feature extraction stage. Although integrating feature extraction, which is generally domain-dependent, into the model construction phase is a considerable advantage of deep learning, the price is generally paid in computationally intensive model selection required for training deep neural networks.

Although the term “deep learning” was coined for the first time by Rina Dechter in 1986 ([Dechter, 1986](#)), the history of its development goes well beyond that point. The first attempt in mathematical representation of nervous system as a network of (artificial) neurons was based on seminal work of McCulloch and Pitts in 1943 ([McCulloch and Pitts, 1943](#)). They showed that under certain conditions, any arbitrary Boolean function can be represented by a network of triggering units with a threshold (neurons). Nevertheless, the strength of interconnections between neurons (synaptic weight) were not “learned” in their networks and should have adjusted manually. Rosenblatt studied similar network of threshold units in 1962 under the name *perceptrons* ([Rosenblatt, 1962](#)). He proposed an algorithm to learn the weights of perceptron. The first attempts in successfully implementing and learning a deep neural network can be found in studies conducted by Ivakhnenko and his colleagues in mid 60s and early 70s ([Ivakhnenko and Lapa, 1965; Ivakhnenko, 1971](#)). The work of Ivakhnenko *et al.* are essentially implementation of a multi-layer perceptron-type network. A few years later, some other architectures, namely, Cognitron ([Fukushima, 1975](#)) and Neocognitron ([Fukushima, 1980](#)), were introduced and became the foundation of modern *convolutional neural network*, which is itself one of the major driving force behind the current momentum that we observe today in deep learning.

In this section, we introduce *multilayer perceptron* (MLP), which is one of the basic forms of ANN with a long history and strong track record of success in many applications. The structure of MLP is closely related to Kolmogorov–Arnold representation theorem (KA theorem). The KA theorem states that we can write any continuous function f of p variables $f : [0, 1]^p \rightarrow \mathbb{R}$ as a finite sum of composition of univariate functions; that is to say, for f there exists univariate functions h_i and g_{ij} such that

$$f(\mathbf{x}) = \sum_{i=1}^{2p+1} h_i \left(\sum_{j=1}^p g_{ij}(x_j) \right), \quad (13.1)$$

where $\mathbf{x} = [x_1, x_2, \dots, x_p]^T$. However, KA theorem does not provide a recipe for the form of h_i and g_{ij} . Suppose we make choices as follows:

- $g_{ij}(x_j)$ takes the following form:

$$g_{ij}(x_j) = a_{ij}x_j + \frac{a_{i0}}{p}, \quad j = 1, 2, \dots, p, \quad (13.2)$$

where a_{i0} (known as “biases”) and a_{ij} are some unknown constants.

- $h_i(t)$ takes the following form:

$$h_i(t) = b_i\phi(t) + \frac{b_0}{k}, \quad i = 1, 2, \dots, k, \quad (13.3)$$

where b_0 (bias) and b_i are some unknown constants and in contrast with (13.1), rather than taking i from 1 to $2p + 1$, we take it from 1 to k , which is an arbitrary

integer number. $\phi(t)$ is known as *activation function* and in contrast with h_i in (13.1), it does not depend on i . Common choices of $\phi(t)$ are described next.

- An important and common family of $\phi(t)$ is *sigmoid*. A sigmoid is a nondecreasing function, having the property that $\lim_{t \rightarrow \infty} \sigma(t) = 1$ and $\lim_{t \rightarrow -\infty} \sigma(t)$ is either 0 or -1. We have already seen logistic sigmoid in Section 6.2.2 in connection with logistic regression—a subscript with the name is used to better differentiate various sigmoid functions introduced next:

$$\sigma_{\text{logistic}}(t) = \frac{1}{1 + e^{-t}}. \quad (13.4)$$

Another commonly used form of sigmoid function in the context of neural network is hyperbolic tangent, which is given by

$$\sigma_{\tanh}(t) = \tanh(t) = \frac{\sinh(t)}{\cosh(t)} = \frac{e^t - e^{-t}}{e^t + e^{-t}} = \frac{1 - e^{-2t}}{1 + e^{-2t}} = 2\sigma_{\text{logistic}}(2t) - 1. \quad (13.5)$$

Although less common compared with the previous two sigmoids, tangent inverse is another example of sigmoid functions:

$$\sigma_{\text{arctanh}}(t) = \frac{2}{\pi} \text{arctanh}(t). \quad (13.6)$$

An important non-sigmoid activation function is ReLU (short for Rectified Linear Unit) given by

$$\phi_{\text{ReLU}}(t) = \max(0, t). \quad (13.7)$$

Replacing (13.2) and (13.3) in (13.1) yields:

$$f(\mathbf{x}) \approx b_0 + \sum_{i=1}^k b_i \phi \left(a_{i0} + \sum_{j=1}^p a_{ij} x_j \right), \quad (13.8)$$

which can equivalently be written as

$$f(\mathbf{x}) \approx b_0 + \sum_{i=1}^k b_i u_i, \quad (13.9)$$

$$u_i = \phi \left(a_{i0} + \sum_{j=1}^p a_{ij} x_j \right). \quad (13.10)$$

In a supervised learning problem in which our goal is to estimate y , which is the target value for a given \mathbf{x} , we can take $f(\mathbf{x})$ in (13.8) as the estimate of y . This is indeed the mathematical machinery that maps an input to its output in a MLP with one *hidden* layer. A graphical representation of this mapping is depicted in Fig. 13.1. The hidden layer is part of the network that is neither directly connected to the inputs nor to the output (its connection is through some weights). In other words, the “hidden” layer, as the name suggests, is hidden from outside world in which we only see the input and the output.

To better resemble the similarity between a biological neuron and an artificial neuron, a biological neuron is overlaid on top of the network depicted in Fig. 13.1. Although we do this to draw the analogy between ANN and the biological network of neurons, it would be naive to think of ANNs as *in silico* implementation of biological neural networks!

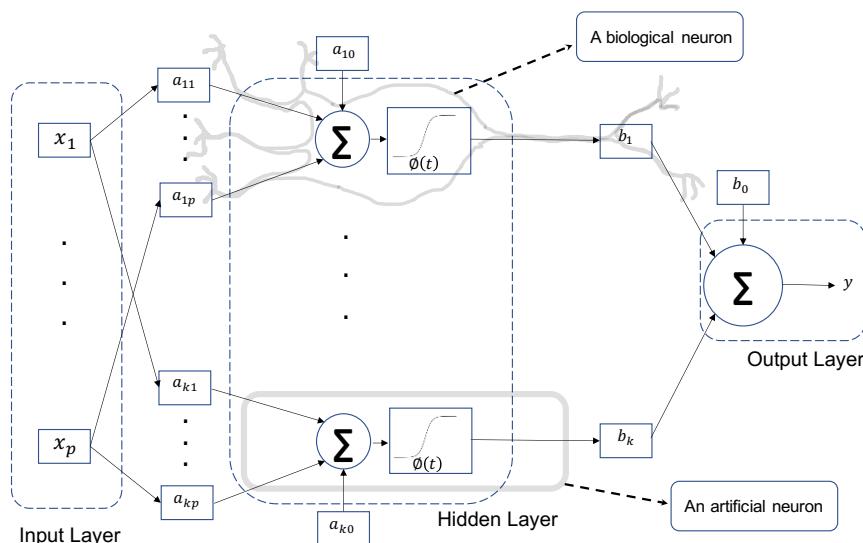


Fig. 13.1: MLP with one hidden layer and k neurons. This is known as a three-layer network with one input layer, one hidden layer, and one output layer.

As observed in Fig. 13.1 (also see (13.8)), each input feature is multiplied by a coefficient and becomes part of the input to each of the k neurons in the hidden layer. The output of these neurons are also multiplied by some weights, which, in turn, are used as inputs to the output layer. Although in Fig. 13.1, we take y as the sum of inputs to the single neuron in that layer, we may add a sigmoid and/or a threshold function used after the summation if classification is desired. Furthermore, the output layer itself can have a collection of artificial neurons, for example, if multiple target variables should be estimated for a given input.

The network in Fig. 13.1 is also known as a three-layer MLP because of the three layers used in its structure: 1) the input layer; 2) the hidden layer; and 3) the output layer. A special case of this network is where there is no hidden layer and each input feature is multiplied by a weight and is used as the input to the neuron in the output layer. That leads to the structure depicted in Fig 13.2, which is known as perceptron.

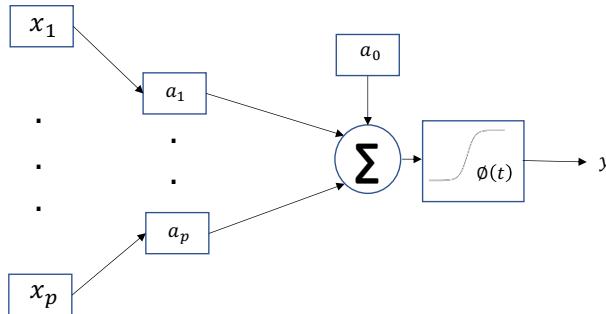


Fig. 13.2: The structure of a perceptron

There are two important differences between (13.8) and (13.1) that are highlighted here:

1. In (13.1), the choices of h_i and g_{ij} depend on the function $f(\mathbf{x})$; however, in (13.8), they are *our* choices presented in (13.2) and (13.3); and
2. As we impose our choices of h_i and g_{ij} presented in (13.2) and (13.3), the expression in (13.1) does not hold exactly. This is the reason we use \approx in (13.8). However, in 1989, in a seminal work, Cybenko showed that using (13.8), “arbitrary decision regions can be arbitrarily well approximated” (Cybenko, 1989). This is known as the universal expressive power of a three-layer MLP.

The input-output mapping presented above through one hidden layer can be extended to as many layers as desired. For example, the input-output mapping in an MLP with 2 hidden layers (a four-layer network) is presented by the following set of equations:

$$f(\mathbf{x}) \approx c_0 + \sum_{l=1}^{k_2} c_l u_l, \quad (13.11)$$

$$u_l = \phi(b_{l0} + \sum_{i=1}^{k_1} b_{li} v_i) \quad 1 \leq l \leq k_2, \quad (13.12)$$

$$v_i = \phi(a_{i0} + \sum_{j=1}^p a_{ij} x_j) \quad 1 \leq i \leq k_1. \quad (13.13)$$

Fig. 13.3 presents a graphical representation of a four-layer MLP characterized by (13.11)-(13.13).

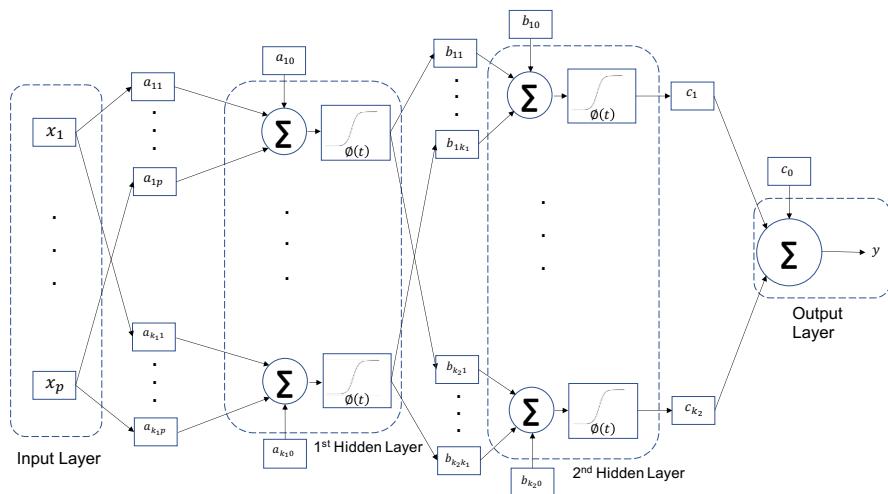


Fig. 13.3: MLP with two hidden layers and k_1 and k_2 neurons in the first and second hidden layers, respectively.

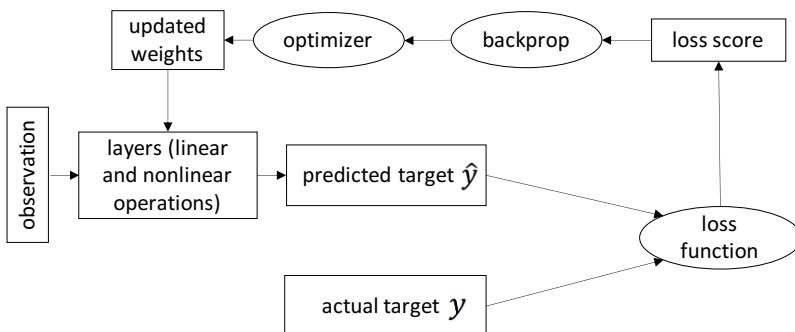


Fig. 13.4: The general machinery of training an ANN.

13.2 Backpropagation, Optimizer, Batch Size, and Epoch

So far we did not specify how the unknown weights in an MLP are adjusted. As in many other learning algorithms, these weights are estimated from training data in the training phase. In this regard, we use the iterative gradient descent process that uses steepest descent direction to update the weights. Fig. 13.4 presents a schematic description of this training process. In particular, the current weights of the network, which are initially assigned to some random values, result in a predicted target. The distance between the predicted target from the actual target is measured using a loss function and this distance is used as a feedback signal to adjust the weights.

To find the steepest descent direction, we need the gradient of the loss function with respect to weights. This is done efficiently using a two-stage algorithm known as *backpropagation* (also referred to as backprop), which is essentially based on chain rules of calculus. The two stages in backprop include some computations from the first to the last layer of the network (the first stage: forward computations) and some computations from the last layer to the first (the second stage: backward computations). Once the gradient is found using backprop, the adjustment to weights in the direction of steepest descent is performed using an *optimizer*; that is to say, the optimizer is the update rule for the weight estimates. There are two ways that we can update the weights in the aforementioned process:

1. We can compute the value of the gradient of loss function across all training data, and then update the weights. This is known as *batch* gradient descent. However, for a single update, all training data should be used in the backprop and the optimizer, and then take one single step in the direction of steepest descent. At the same time, a single presentation of the *entire* training set to the backprop is called an *epoch*. Therefore, in the batch gradient descent, one update for the weights occurs at the end of each epoch. In other words, the number of epochs is an indicator of the amount of times training data is presented to the network, which is perceived as the amount of training the ANN. The number of epochs is generally considered as a hyperparameter and should be tuned in model selection.
2. We can compute the value of the gradient of loss function for a *mini* training data (a small subset of training data), and then update the weights. This is known as *mini-batch* gradient descent. Nevertheless, the concept of epoch is still the same: a single presentation of the *entire* training set to the backprop. This means that if we partition a training data of size n into mini-batches of size $K \ll n$, then in an epoch we will have $\lceil n/K \rceil$ updates for weights. In this context, the size of each mini-batch is called *batch size*. A good choice, for example, is 32 but in general it is a hyperparameter and should be tuned in model selection.

13.3 Why Keras?

Although we can use scikit-learn to train certain types of ANNs, we will not use it for that purpose because:

1. training “deep” neural networks requires estimating and adjusting many parameters and hyperparameters, which is a computationally expensive process. As a result successful training various forms of neural networks highly rely on parallel computations, which are realized using Graphical Processing Units (GPUs) or Tensor Processing Units (TPUs). However, scikit-learn currently does not support using GPU or TPU.
2. scikit-learn does not currently implement some popular forms of neural networks such as convolutional neural networks or recurrent neural networks.

As a result, we switch to Keras with TensorFlow backend as they are particularly designed for training and tuning various forms of ANN and support various forms of hardware including CPU, GPU, or TPU.

As previously mentioned in Chapter 1, simplicity and readability along with a number of third-party packages have made Python as the most popular programming language for data science. When it comes to deep learning, Keras library has become such a popular API to the extent that its inventor, Francois Chollet, refers to Keras as “the Python of deep learning” ([Chollet, 2021](#)). In a recent survey conducted by Kaggle in 2021 on “State of Data Science and Machine Learning” ([Kaggle-survey, 2021](#)), Keras usage among data scientists was ranked fourth (47.3%) only preceded by xgboost library (47.9%), TensorFlow (52.6%), and scikit-learn (82.3%). But what is Keras?

Keras is a deep learning API written in Python. It provides high-level “building blocks” for efficiently training, tuning, and evaluating deep neural networks. However, Keras relies on a backend engine, which does all the low-level operations such as tensor products, differentiation, and convolution. Keras was originally released to support Theano backend, which was developed at Université de Montréal. Several months later when TensorFlow was released by google, Keras was refractored to support both Theano and TensorFlow. In 2017, Keras was even supporting some additional backends such as CNTK and MXNet. Later in 2018, TensorFlow adopted Keras as its high-level API. In September 2019, TensorFlow 2.0 was released and Keras 2.3 became its last multi-backend release and it was announced that: “Going forward, we recommend that users consider switching their Keras code to `tf.keras` in TensorFlow 2.0. It implements the same Keras 2.3.0 API (so switching should be as easy as changing the Keras import statements)” ([Keras-team, 2019](#)).

If Anaconda was installed as explained in Section 2.1, TensorFlow can easily be installed using conda by executing the following command line in a terminal ([Tensorflow, 2023](#)): `conda install -c conda-forge tensorflow`. We can also check their version as follows:

```
import tensorflow as tf
from tensorflow import keras
```

```
print(tf.__version__)
print(keras.__version__)
```

13.4 Google Colaboratory (Colab)

As mentioned earlier, for training deep neural networks it is recommended to use GPU or TPU cards. However, one may not have access to such hardware locally. In such a case, we can work on Google colaboratory (Colab), which gives access to a limited free GPU and TPU runtime. Colab is like a Jupyter notebook service stored on google drive that connects a notebook to cloud-based run time. In other words, Colab notebook is the cloud-based google implementation of Jupyter notebook. Therefore, we should be able to run any code that runs locally in Jupyter notebook on Colab, if all packages are installed. At the same time, Colab currently comes with TensorFlow, Keras, Scikit-Learn, numpy, pandas, and many more, which means we have already access to a variety of useful Python packages and there is no need to install them separately. However, for installing any package, if needed, we can run: `!pip install PACKAGE_NAME` in a code cell in Colab (“!” is used because it is a command line rather than Python code).

To open a Colab notebook, we can go to Google Drive and open a notebook as in Fig. 13.5. Once a Colab notebook is opened, we should see an empty notebook as in Fig. 13.6. This is where codes are written and executed. As seen on top of Fig. 13.6, similar to a Jupyter notebook, a Colab notebook has two types of cells (Code and Text) as well.

As of this writing, the version of TensorFlow, Keras, and Scikit-Learn that are already pre-installed in Colab are (the code snippet is run on Colab):

```
import tensorflow as tf
from tensorflow import keras
import sklearn
print(tf.__version__)
print(keras.__version__)
print(sklearn.__version__)
```

2.9.2

2.9.0

1.0.2

By default, running codes in Colab is done over CPUs provided by Google servers. However, to utilize GPU or even TPU, we can go to “Runtime → Change runtime type” and choose GPU/TPU as the Hardware accelerator (see Fig. 13.7). However, if we want to train ANN models on TPU, it requires some extra work, which is not needed when GPU is used. As a result of that, and as for our purpose here (and

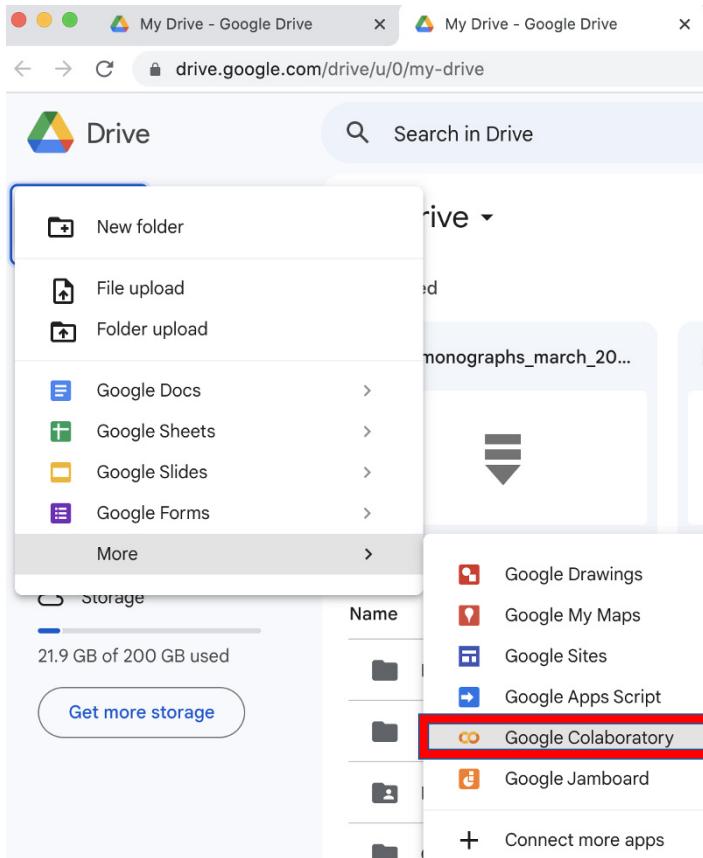


Fig. 13.5: Opening a Colab notebook from Google Drive.

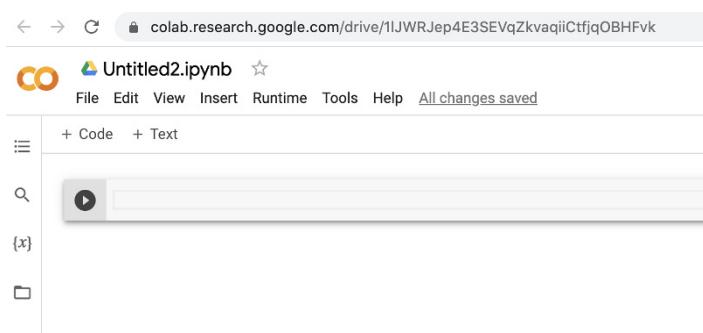


Fig. 13.6: An empty Colab notebook

in many other applications) the use of GPU suffice the need, we skip those details (details for that purpose are found at ([Tensorflow-tpu, 2023](#))).

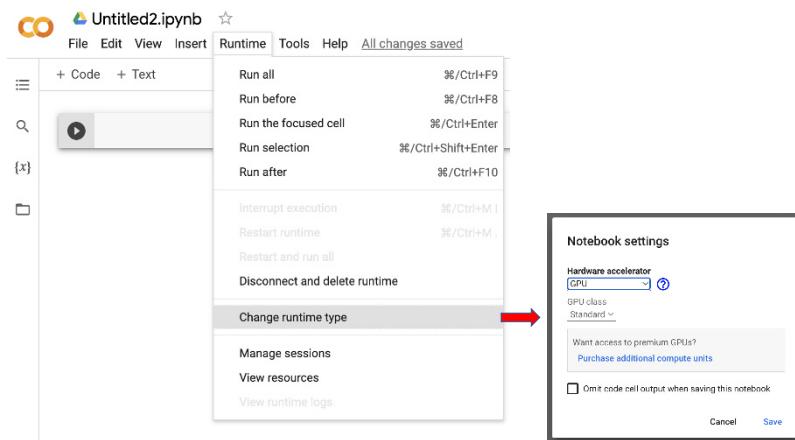


Fig. 13.7: Using a hardware accelerator

Once a GPU is selected as the Hardware accelerator, we can ensure the GPU is used as follows:

```
import tensorflow as tf
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
```

```
Num GPUs Available:  1
```

13.5 The First Application Using Keras

In this section, we discuss building, compiling, training, and evaluating ANNs in Keras. To better clarify the concepts, we use a well-known dataset known as MNIST.

13.5.1 Classification of Handwritten Digits: MNIST Dataset

The MNIST dataset has a training set of 60,000 grayscale 28×28 images of handwritten digits, and an additional test set of size 10,000. It is part of `keras.dataset` module and can be loaded as four numpy arrays as follows (in the code below and to have reproducible results as we proceed with this application, we also set the random seeds for Python, NumPy, and TensorFlow):

```
from tensorflow.keras.datasets import mnist

seed_value= 42
# set the seed for Python built-in pseudo-random generator
import random
random.seed(seed_value)

# set the seed for numpy pseudo-random generator
import numpy as np
np.random.seed(seed_value)

# set the seed for tensorflow pseudo-random generator
import tensorflow as tf
tf.random.set_seed(seed_value)

(X_train_val, y_train_val), (X_test, y_test) = mnist.load_data()

print(type(X_train_val))
print(X_train_val.dtype)
print(X_train_val.shape)
print(X_train_val.ndim)
```

```
<class 'numpy.ndarray'>
uint8
(60000, 28, 28)
3
```

As can be seen the data type is `unit8`, which is unsigned integers from 0 to 255. Before proceeding any further, we first scale down the data to [0, 1] by dividing each entry by 255—here rescaling is not data-driven and we use prior knowledge in doing so. The default data type of such a division would be double precision. As the memory itself is an important factor in training deep ANNs, we also change the data type to `float32`.

```
X_train_val, X_test = X_train_val.astype('float32')/255, X_test.  
    ↪astype('float32')/255  
print(X_train_val.dtype)
```

float32

Also notice the shape of `X_train_val` as a three-dimensional numpy array, which is also known as a rank-3 tensor. To train a multilayer perceptron, the model works on a feature vector (not an image), which is similar to models that we have seen previously in scikit-learn. This means that the model expects a rank-2 tensor, which is an array of (sample size, feature size). To do so, we reshape the arrays:

```
X_train_val, X_test = X_train_val.reshape(60000, 28*28), X_test.  
    ↪reshape(10000, 28*28)  
X_train_val[0].shape
```

```
(784,)
```

For the model selection purposes, we randomly split `X_train_val` into training and validation sets:

```
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(X_train_val, ▾
    y_train_val, stratify=y_train_val, test_size=0.25)
X_train.shape
```

```
(45000, 784)
```

13.5.2 Building Model Structure in Keras

All models in Keras are objects of `tf.keras.Model` class. There are three ways to build object of this class (to which we simply refer as models): 1) Sequential API; 2) Functional API; and 3) Subclassing API. Sequential models are used when stack of standard layers is desired where each layer has one input tensor and one output tensor. This is what we are going to use hereafter. The functional and subclassing are used for cases when multiple inputs/outputs (for example to locate and classify an object in an image) or more complex uncommon architectures are desired. There are two ways to use Sequential models:

- Method 1: pass a list of layers to `keras.Sequential` class constructor to instantiate.
- Method 2: instantiate `keras.Sequential` first with no layer and then use `add()` method to add layers one after another.

There are many layers available in `keras.layers` API. Each layer consists of a recipe for mapping the input tensor to the output tensor. At the same time, many layers have state, which means some weights learned using a given data by the learning algorithm. However, some layers are stateless—they are only used for reshaping an input tensor. A complete list of layers in Keras can be found at ([Keras-layers, 2023](#)). Next we examine the aforementioned two ways to build a Sequential model.

Method 1:

```
from tensorflow import keras
from tensorflow.keras import layers

mnist_model = keras.Sequential([
    layers.Dense(128, activation="sigmoid"),
    layers.Dense(64, activation="sigmoid"),
    layers.Dense(10, activation="softmax")
])
```

```
# mnist_model.weights # if we uncomment this code and try to run it at
# this stage, an error is raised because the weights are not set (the
# model is not actually built yet)
```

Method 2:

```
from tensorflow import keras
from tensorflow.keras import layers

mnist_model = keras.Sequential()
mnist_model.add(layers.Dense(128, activation="sigmoid"))
mnist_model.add(layers.Dense(64, activation="sigmoid"))
layers.Dense(10, activation="softmax")
```

<tensorflow.python.keras.core.Dense at 0x7fab7e4ec730>

Let us elaborate on this code. A `Dense` layer, also known as *fully connected layer*, is indeed the same hidden layers that we have already seen in Figs. 13.1 and 13.3 for MLP. They are called fully connected layers because every input to a layer is connected to every neuron in the layer. In other words, the inputs to each layer, which could be the outputs of neurons in the previous layer, go to every neuron in the layer.

While the use of activation functions (here logistic sigmoid) in both hidden layers is based on (13.12) and (13.13), the use of `softmax` in the last layer is due to the multiclass (single label) nature of this classification problem. To understand this, first note that the number of neurons in the output layer is the same as the number of digits (classes). This means that for each input feature vector \mathbf{x} , the model will output a c -dimensional vector (here $c = 10$). The softmax function takes this c -dimensional vector, say $\mathbf{a} = [a_0, a_1, \dots, a_{(c-1)}]^T$, and normalized that into a probability vector $\text{softmax}(\mathbf{a}) = \mathbf{p} = [p_0, p_1, \dots, p_{(c-1)}]^T$ (which should add up to 1) according to:

$$p_i = \frac{e^{a_i}}{\sum_i e^{a_i}}, \quad i = 0, \dots, c - 1. \quad (13.14)$$

Then an input is classified to the class with the highest probability.

The codes presented in Method 1 or Method 2 do not build the model in the strict sense. This is because “building” a model means setting the weights of layers, but the number of weights depends on the input size, which is not given in the above codes. In that case, the model will be built as soon as it is called on some data (for example, using `fit()` method). Nevertheless, we can also “build” the model using both Method 1 and 2 if we just specify the shape of each input using `input_shape` parameter:

```
from tensorflow import keras
from tensorflow.keras import layers

mnist_model = keras.Sequential([
    layers.Flatten(input_shape=(28, 28)),
    layers.Dense(128, activation="relu"),
    layers.Dense(64, activation="relu"),
    layers.Dense(10, activation="softmax")])
```

```

        layers.Dense(128, activation="sigmoid"),
input_shape = X_train[0].shape),
        layers.Dense(64, activation="sigmoid"),
        layers.Dense(10, activation="softmax")
    ])

```

```
mnist_model.weights # the weights are initialized (the biases to 0 and
˓→the rest randomly)
```

```
[<tf.Variable 'dense_237/kernel:0' shape=(784, 128) dtype=float32, numpy=
array([[ 0.0266955 , -0.00956997, -0.02386561, ..., -0.0407474 ,
       0.01480078, -0.0003909 ],
       [ 0.0645309 , -0.06136192, -0.05915052, ...,  0.00383252,
       0.05406239,  0.0454815 ],
       ...,
       ...,
       [-1.50822967e-01,  6.51585162e-02,  1.69166803e-01,
       -1.23789206e-01,  1.93499446e-01,  2.29302317e-01,
       2.81352669e-01, -2.16220707e-01, -5.38927913e-02,
       -2.35425845e-01]], dtype=float32)>,
<tf.Variable 'dense_239/bias:0' shape=(10,) dtype=float32,_
˓→numpy=array([0., 0.,
0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)>]
```

We can use `summary()` method that tabulates the shape of output and the total number of parameters that are/should be estimated in each layer.

```
mnist_model.summary()
```

Model: "sequential_76"

Layer (type)	Output Shape	Param #
dense_237 (Dense)	(None, 128)	100480
dense_238 (Dense)	(None, 64)	8256
dense_239 (Dense)	(None, 10)	650

Total params: 109,386

Trainable params: 109,386

Non-trainable params: 0



In Section 13.5.1 we reshaped the rank-3 input tensor to make it in the form of (samples, features). To do so, we flattened each image. Another way is to add a flattning layer as the first layer of the network and use the

same rank-3 tensor as the input (without reshaping). In that case the model is defined as follows:

```
# mnist_model = keras.Sequential([
#                               layers.Flatten(),
#                               layers.Dense(128,
#                                         activation="sigmoid"),
#                               layers.Dense(64,
#                                         activation="sigmoid"),
#                               layers.Dense(10, activation="softmax")
#                           ]) # here we have not specified the
#                               #input_shape but we can do so by layers.
#                               Flatten(input_shape=(28, 28))
```

13.5.3 Compiling: optimizer, metrics, and loss

After building the model structure in Keras, we need to *compile* the model. Compiling in Keras is the process of *configuring the learning process* via `compile()` method of `keras.Model` class. In this regard, we set the choice of optimizer, loss function, and metrics to be computed through the learning process by setting the `optimizer`, `loss`, and `metrics` parameters of `compile()` method, respectively. Here we list some possible common choices for classification and regression:

1. `optimizer` can be set by replacing `name` in `optimizer = name`, for example, with one of the following strings where each refers to an optimizer known with the same name: `adam`, `rmsprop`, `sgd`, or `adagrad` (for a complete list of optimizers supported in Keras see ([Keras-optimizers, 2023](#))). Doing so uses default values of parameters in the class implementing each optimizer. If we want to have more control over the parameters of an optimizer, we need to set the `optimizer` parameter to an object of optimizer class; for example, `optimizer = keras.optimizers.Adam(learning_rate=0.01)` using which we are changing the 0.001 default value of `learning_rate` used in Adam optimizer to 0.01.
2. `metrics` can be set by replacing `name(s)` in `metrics = [name(s)]` with a list of strings where each refer to a metric with the same name; for example, `accuracy` (for classification), or `mean_squared_error` (for regression). A more flexible way is to use the class implementing the metric; for example, `metrics = ["mean_squared_error", "mean_absolute_error"]` is equivalent to
`metrics = [keras.metrics.MeanSquaredError(),`
 `keras.metrics.MeanAbsoluteError()]`

For a complete list of metrics refer to ([Keras-metrics, 2023](#)).

3. loss can be set by replacing name in loss = name with one of the following strings where each refer to a loss known with the same name:

- **binary_crossentropy** (for binary classification as well as multilabel classification, which more than one label is assigned to an instance);
- **categorical_crossentropy** (for multiclass classification when target values are *one-hot encoded*);
- **sparse_categorical_crossentropy** (for multiclass classification when target values are *integer encoded*),
- **mean_squared_error** and **mean_absolute_error** for regression.

For a complete list of losses supported in Keras see ([Keras-losses, 2023](#)). We can also pass an object of the class implementing the loss function; for example, loss = keras.losses.BinaryCrossentropy() is equivalent to loss = binary_crossentropy.

In case of binary classification (single-label) and assuming the labels are already encoded as integers 0 and 1, if we use **binary_crossentropy**, in the last layer we should use one neuron and **sigmoid** as the activation function. The output of activation function in the last layer in this case is the probability of class 1. Alternatively, it can also be treated as a multi-class classification and use two neurons in the last layer with a **softmax** activation and then use **sparse_categorical_crossentropy**. In this case, the output of the last layer is the probability of classes 0 and 1. Here are examples to see this equivalence:

```
true_classes = [0, 1, 1] # actual labels
class_probs_per_obs = [[0.9, 0.1], [0.2, 0.8], [0.4, 0.6]] # the ↵outputs of softmax for two neurons
scce = tf.keras.losses.SparseCategoricalCrossentropy()
scce(true_classes, class_probs_per_obs).numpy()
```

0.27977657

```
true_classes = [0, 1, 1]
class_1_prob_per_obs = [0.1, 0.8, 0.6] # the outputs of sigmoid for one ↵neuron
bce = tf.keras.losses.BinaryCrossentropy(from_logits=False)
bce(true_classes, class_1_prob_per_obs).numpy()
```

0.2797764



Q1. The choice of the loss function depends on encoding schemes. What is an encoding scheme?

Q2. What is the difference between integer encoding and one-hot encoding?

Q3. If we care about a metric such as accuracy to be monitored during training, why don't we use its corresponding cost/loss function error =

1 – accuracy in the backprop? Why do we use a loss function such as categorical cross-entropy to serve as a proxy for having a better predictive performance?

Answer to Q1: Categorical data in real-world applications are ubiquitous but, on the contrary, the algorithms at the core of various popular machine learning methods such as ANNs are based on the assumption of having numerical variables. As a result, various encoding schemes are created to transform categorical variables to their numerical counterparts. In case of classification, the target (class label) is generally a nominal variable (i.e., a categorical variable with no natural ordering between its values). As a result, the use of encoding along with the class label is commonplace (see Exercise 3, Chapter 4).

Answer to Q2: Integer encoding, also known as ordinal encoding, treats unordered categorical values as if they were numeric integers. For example, a class label, which could be either “Red”, “Blue”, and “Green” is treated as 0, 1, and 2. This type of encoding raises questions regarding the relative magnitudes assigned to categories after numeric transformations; after all, categorical features have no relative “magnitude” but numbers do. On the other hand, one-hot encoding replaces one categorical variable with N possible categories with standard basis vectors of N dimensional Euclidean vector space. For example, “Red”, “Blue”, and “Green” are replaced by $[1, 0, 0]^T$, $[0, 1, 0]^T$, and $[0, 0, 1]^T$, respectively. As a result, these orthogonal binary vectors are equidistant with no specific order, which compared with the outcomes of integer encoding, align better with the intuitive meaning of categorical variables. However, because each utilized standard basis vector lives in an N dimensional vector space, one-hot encoding can blow up the size of data matrix if many variables with many categories are present.

Answer to Q3: The answer to this question lies in the nature of backprop. As we said before, in backprop we efficiently evaluate the steepest descent direction for which we need the gradient of the loss function with respect to weights. However, using a loss such as 1 – accuracy is not suitable for this purpose because this is a piecewise constant function of weights, which means the gradient is either zero (the weights do not move) or undefined. Therefore, other loss functions such as `categorical_crossentropy` for classification is commonly used. The use of `mean_absolute_error` loss function for regression is justified by *defining* its derivative at zero as a constant such as zero.

We now compile the previously built `mnist_model`. Although the labels stored in `y_train`, `y_val`, and `y_test` are already integer encoded (and we can use `sparse_categorical_crossentropy` to compile the model), here we show the utility of `to_categorical()` for one-hot encoding and, therefore, we use `categorical_crossentropy` to compile the model:

```
mnist_model.compile(optimizer="adam", loss="categorical_crossentropy",
                     metrics=["accuracy"])
```

```
y_train_1_hot = keras.utils.to_categorical(y_train, num_classes = 10)
y_val_1_hot = keras.utils.to_categorical(y_val, num_classes = 10)
y_test_1_hot = keras.utils.to_categorical(y_test, num_classes = 10)
print(y_train.shape)
print(y_train_1_hot.shape)
```

```
(45000,)
(45000, 10)
```

⊕ **categorical_crossentropy** and **sparse_categorical_crossentropy**: As we said before, depending on whether we use one-hot encoded labels or integer encoded labels, we need to use either the **categorical_crossentropy** or the **sparse_categorical_crossentropy**. But what is the difference between them?

Suppose for a multiclass (single label) classification with c classes, a training data $\mathbf{S}_{tr} = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$ is available. Applying one-hot encoding on $y_j, j = 1, \dots, n$, leads to label vectors $\mathbf{y}_j = [y_{j0}, \dots, y_{j(c-1)}]$ where $y_{ji} \in \{0, 1\}, i = 0, \dots, c - 1$. Categorical cross-entropy (also known as cross-entropy) is defined as

$$e(\boldsymbol{\theta}) = -\frac{1}{n} \sum_{j=1}^n \sum_{i=0}^{c-1} y_{ji} \log(P(Y_j = i | \mathbf{x}_j; \boldsymbol{\theta})), \quad (13.15)$$

where $\boldsymbol{\theta}$ denotes ANN parameters (weights). Comparing this expression with the cross-entropy expression presented in Eq. (6.39) in Section 6.2.2 shows that (13.15) is indeed the extension of the expression presented there to multiclass classification with one-hot encoded labels. Furthermore, $P(Y_j = i | \mathbf{x}_j; \boldsymbol{\theta})$ denotes the probability that a given observation \mathbf{x}_j belongs to class i . In multiclass classification using ANN, we take this probability as the output of the softmax function when \mathbf{x}_j is used as the input to the network; that is to say, we estimate $P(Y_j = i | \mathbf{x}_j; \boldsymbol{\theta})$ by the value of p_i defined in (13.14) for the given observation. Therefore, we can rewrite (13.15) as:

$$e(\boldsymbol{\theta}) = -\frac{1}{n} \sum_{j=1}^n \mathbf{y}_j^T \log(\mathbf{p}_j), \quad (13.16)$$

where here $\log(\cdot)$ is applied element-wise and $\mathbf{p}_j = [p_{j0}, p_{j1}, \dots, p_{j(c-1)}]^T$ is the vector output of softmax for a given input \mathbf{x}_j . This is what is implemented by **categorical_crossentropy**. The **sparse_categorical_crossentropy** implements a similar function as in (13.16) except that rather than one-hot encoded labels, it expects integer encoding with labels y_j in the range of $0, 1, \dots, c - 1$. This way, we can write (sparse) cross-entropy as:

$$e(\theta) = -\frac{1}{n} \sum_{j=1}^n \log(p_{jy_j}). \quad (13.17)$$

Next we present an example to better understand the calculation of cross-entropy.

Example 13.1 Suppose we have a class label with possible values, Red, Blue, and Green. We use one-hot encoding and represent them as Red : $[1, 0, 0]^T$, Blue : $[0, 1, 0]^T$, and Green : $[0, 0, 1]^T$. Two observations \mathbf{x}_1 and \mathbf{x}_2 are presented to an ANN with some values for its weights. The true classes of \mathbf{x}_1 and \mathbf{x}_2 are Green and Blue, respectively. The output of the network after the softmax for \mathbf{x}_1 is $[0.1, 0.4, 0.5]^T$ and its output for \mathbf{x}_2 is $[0.05, 0.8, 0.15]^T$. What is the cross-entropy?

The classes of \mathbf{x}_1 and \mathbf{x}_2 are $[0, 0, 1]^T$ and $[0, 1, 0]^T$, respectively; that is, $\mathbf{y}_1 = [0, 0, 1]^T$ and $\mathbf{y}_2 = [0, 1, 0]^T$. At the same time, $\mathbf{p}_1 = [0.1, 0.4, 0.5]^T$ and $\mathbf{p}_2 = [0.05, 0.8, 0.15]^T$. Therefore, (13.16) yields

$$\begin{aligned} e(\theta) &= -\frac{1}{2}(\mathbf{y}_1^T \log(\mathbf{p}_1) + \mathbf{y}_2^T \log(\mathbf{p}_2)) \\ &= -\frac{1}{2}\left((0 \times \log(p_{10}) + 0 \times \log(p_{11}) + 1 \times \log(p_{12})) \right. \\ &\quad \left. + (0 \times \log(p_{20}) + 1 \times \log(p_{21}) + 0 \times \log(p_{22}))\right) = -\frac{1}{2}(\log(0.5) + \log(0.8)) = 0.458 \end{aligned} \quad (13.18)$$

We now use `keras.losses.CategoricalCrossentropy()` to compute this:

```
from tensorflow import keras
labels_one_hot = [[0, 0, 1], [0, 1, 0]]
class_probs_per_obs = [[0.1, 0.4, 0.5], [0.05, 0.8, 0.15]]
cce = keras.losses.CategoricalCrossentropy()
cce(labels_one_hot, class_probs_per_obs).numpy()
```

0.45814535

For the sparse categorical cross-entropy, the labels are expected to be integer encoded; that is to say, $y_1 = 2$ and $y_2 = 1$. Therefore, we can pick the corresponding probabilities at index 2 and 1 (assuming the first element in each probability vector has index 0) from \mathbf{p}_1 and \mathbf{p}_2 , respectively, to write

$$e(\theta) = -\frac{1}{2}(\log(0.5) + \log(0.8)) = 0.458 \quad (13.19)$$

We can also use `keras.losses.SparseCategoricalCrossentropy()` to compute this:

```
labels = [2, 1]
class_probs_per_obs = [[0.1, 0.4, 0.5], [0.05, 0.8, 0.15]]
```

```
scce = keras.losses.SparseCategoricalCrossentropy()  
scce(labels, class_probs_per_obs).numpy()
```

0.45814538

As we can see in both cases (using categorical cross-entropy with one-hot encoded labels and sparse categorical cross-entropy with integer encoded labels), the loss is the same.



13.5.4 Fitting

Once a model is compiled, training the model is performed by calling the `fit()` method of `keras.Model` class. There are several important parameters of the `fit()` method that warrant particular attention:

1. `x`: represents the training data matrix;
2. `y`: represents the training target values;
3. `batch_size`: can be used to set the batch size used in the mini-batch gradient descent (see Section 13.2);
4. `epochs`: can be used to set the number of epochs to train the model (see Section 13.2);
5. `validation_data`: can be used to specify the validation set and is generally in the form of `(x_val, y_val)`;
6. `steps_per_epoch`: this is used to set the number of batches processed per epoch. The default value is `None`, which for a batch size of K , sets the number of batches per epoch to the conventional value of $\lceil n/K \rceil$ where n is the sample size. This option would be helpful, for example, when n is such a large number that each epoch may take a long time if $\lceil n/K \rceil$ is used;
7. `callbacks_list`: a list of callback objects. A callback is an object that is called at various stages of training to take an action. There are several callbacks implemented in Keras but a combination of the following two is quite handy in many cases (each of these accepts multiple parameters [see their documentations] but here we present them along with few of their useful parameters):
 - `keras.callbacks.EarlyStopping(monitor="val_loss", patience=0)`

Using this callback, we stop training after a monitored metric (`monitor` parameter) does not improve after a certain number of epochs (`patience`). Why do we need to stop training? On one hand, training an ANN with many epochs may lead to overfitting on the training data. On the other hand, too few epochs could lead to underfitting, which is the situation where the model is not capable of

capturing the relationship between input and output even on training data. Using early stopping we can set a large number of epochs that could be potentially used for training the ANN but stop the training process when no improvement in the monitored metric is observed after a certain number of epochs.

- `keras.callbacks.ModelCheckpoint(filepath="file_path.keras", monitor="val_loss", save_best_only=False, save_freq="epoch")`

By default, this callback saves the model at the end of each epoch. This default behaviour could be changed by setting `save_freq` to an integer in which case the model is saved at the end of that many batches. Furthermore, we often desire to save the best model at the end of epoch if the monitored metric (`monitor`) is improved. This behaviour is achieved by `save_best_only=True` in which case the saved model will be overwritten only if the `monitor` metric is improved. We can also set `verbose` to 1 to see additional information about when the model is/is not saved.

Here we first prepare a list of callbacks and then train our previously compiled model:

```
import time
my_callbacks = [
    keras.callbacks.EarlyStopping(
        monitor="val_accuracy",
        patience=20),
    keras.callbacks.ModelCheckpoint(
        filepath="model/best_model.keras",
        monitor="val_loss",
        save_best_only=True,
        verbose=1)
]

start = time.time()
history = mnist_model.fit(x = X_train,
                           y = y_train_1_hot,
                           batch_size = 32,
                           epochs = 200,
                           validation_data = (X_val, y_val_1_hot),
                           callbacks = my_callbacks)

end = time.time()
training_duration = end - start
print("training duration = {:.3f}".format(training_duration))
```

```
Epoch 1/200
1407/1407 [=====] - 2s 1ms/step - loss: 0.9783 -
accuracy: 0.7587 - val_loss: 0.2579 - val_accuracy: 0.9225
```

```
Epoch 00001: val_loss improved from inf to 0.25790, saving model to
```

```
model/best_model.keras
Epoch 2/200
1407/1407 [=====] - 2s 1ms/step - loss: 0.2295 -
accuracy: 0.9330 - val_loss: 0.1773 - val_accuracy: 0.9485

Epoch 00002: val_loss improved from 0.25790 to 0.17731, saving model to
model/best_model.keras

...
Epoch 13/200
1407/1407 [=====] - 2s 1ms/step - loss: 0.0137 -
accuracy: 0.9971 - val_loss: 0.0888 - val_accuracy: 0.9753

Epoch 00013: val_loss improved from 0.09088 to 0.08877, saving model to
model/best_model.keras
Epoch 14/200
1407/1407 [=====] - 2s 1ms/step - loss: 0.0117 -
accuracy: 0.9974 - val_loss: 0.1019 - val_accuracy: 0.9722

Epoch 00014: val_loss did not improve from 0.08877
Epoch 46/200
1407/1407 [=====] - 2s 1ms/step - loss: 0.0050 -
accuracy: 0.9986 - val_loss: 0.1318 - val_accuracy: 0.9778

...
Epoch 00046: val_loss did not improve from 0.08877
Epoch 47/200
1407/1407 [=====] - 2s 1ms/step - loss: 1.
-1854e-04 -
accuracy: 1.0000 - val_loss: 0.1361 - val_accuracy: 0.9782

Epoch 00047: val_loss did not improve from 0.08877
Epoch 48/200
1407/1407 [=====] - 2s 1ms/step - loss: 8.
-4957e-05 -
accuracy: 1.0000 - val_loss: 0.1316 - val_accuracy: 0.9777

Epoch 00048: val_loss did not improve from 0.08877

...
Epoch 67/200
1407/1407 [=====] - 1s 1ms/step - loss: 0.0011 -
accuracy: 0.9997 - val_loss: 0.1728 - val_accuracy: 0.9756

Epoch 00067: val_loss did not improve from 0.08877
training duration = 122.778
```

```
import matplotlib.pyplot as plt
epoch_count = range(1, len(history.history['loss'])) + 1
plt.figure(figsize=(13,4))
```

```

plt.subplot(121)
plt.plot(epoch_count, history.history['loss'], 'b', label = 'training loss')
plt.plot(epoch_count, history.history['val_loss'], 'r', label = 'validation loss')
plt.legend()
plt.ylabel('loss')
plt.xlabel('epoch')
plt.subplot(122)
plt.plot(epoch_count, history.history['accuracy'], 'b', label = 'training accuracy')
plt.plot(epoch_count, history.history['val_accuracy'], 'r', label = 'validation accuracy')
plt.legend()
plt.ylabel('accuracy')
plt.xlabel('epoch')

```

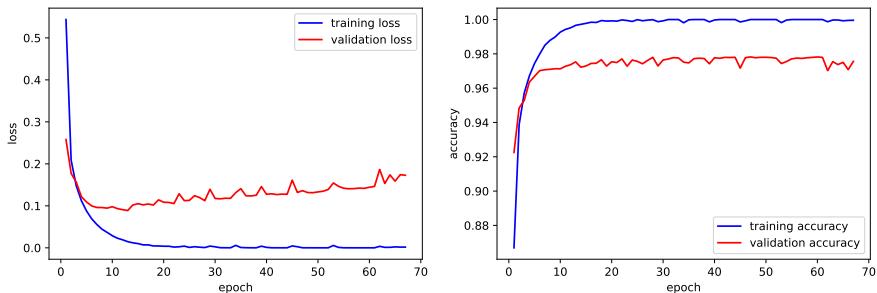


Fig. 13.8: The loss (left) and the accuracy (right) of `mnist_model` as functions of epoch on both the training set and the validation set.

Here we put together all the aforementioned codes for this MNIST classification problem in one place:

```

import time
from tensorflow import keras
import matplotlib.pyplot as plt
from tensorflow.keras import layers
from tensorflow.keras.datasets import mnist
from sklearn.model_selection import train_test_split

seed_value= 42

# set the seed for Python built-in pseudo-random generator
import random

```

```
random.seed(seed_value)

# set the seed for numpy pseudo-random generator
import numpy as np
np.random.seed(seed_value)

# set the seed for tensorflow pseudo-random generator
import tensorflow as tf
tf.random.set_seed(seed_value)

# data preprocessing
(X_train_val, y_train_val), (X_test, y_test) = mnist.load_data()
X_train_val, X_test = X_train_val.astype('float32')/255, X_test.
˓→astype('float32')/255
X_train_val, X_test = X_train_val.reshape(60000, 28*28), X_test.
˓→reshape(10000, 28*28)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, ↵
˓→y_train_val, stratify=y_train_val, test_size=0.25)
y_train_1_hot = keras.utils.to_categorical(y_train, num_classes = 10)
y_val_1_hot = keras.utils.to_categorical(y_val, num_classes = 10)
y_test_1_hot = keras.utils.to_categorical(y_test, num_classes = 10)

# building model
mnist_model = keras.Sequential([
    layers.Dense(128, activation="sigmoid", ↵
˓→input_shape = X_train[0].shape),
    layers.Dense(64, activation="sigmoid"),
    layers.Dense(10, activation="softmax")
])

# compiling model
mnist_model.compile(optimizer="adam", loss="categorical_crossentropy", ↵
˓→metrics=["accuracy"])

# training model
my_callbacks = [
    keras.callbacks.EarlyStopping(
        monitor="val_accuracy",
        patience=20),
    keras.callbacks.ModelCheckpoint(
        filepath="best_model.keras",
        monitor="val_loss",
        save_best_only=True,
        verbose=1)
]
```

```

start = time.time()
history = mnist_model.fit(x = X_train,
                           y = y_train_1_hot,
                           batch_size = 32,
                           epochs = 200,
                           validation_data = (X_val, y_val_1_hot),
                           callbacks=my_callbacks)

end = time.time()
training_duration = end - start
print("training duration = {:.3f}".format(training_duration))
print(history.history.keys())
print(mnist_model.summary())

# plotting the results
epoch_count = range(1, len(history.history['loss']) + 1)
plt.figure(figsize=(13,4))
plt.subplot(121)
plt.plot(epoch_count, history.history['loss'], 'b', label = 'training loss')
plt.plot(epoch_count, history.history['val_loss'], 'r', label = 'validation loss')
plt.legend()
plt.ylabel('loss')
plt.xlabel('epoch')
plt.subplot(122)
plt.plot(epoch_count, history.history['accuracy'], 'b', label = 'training accuracy')
plt.plot(epoch_count, history.history['val_accuracy'], 'r', label = 'validation accuracy')
plt.legend()
plt.ylabel('accuracy')
plt.xlabel('epoch')

```



Suppose in an application we use `callbacks.EarlyStopping(monitor="metric A", patience=K,...)` to perform early stopping based on a metric A and use `callbacks.ModelCheckpoint(monitor="metric B", save_best_only=True,...)` to save the best model once metric B is improved. The training stops at epoch M . Can we claim that the best model is the model saved at epoch $M - K$?

The claim is correct if metric A and metric B are the same; otherwise, it is not necessarily true. In the above application, for example, the best model is saved at epoch 13, while the simulations stopped at epoch 67. This

is because the metric used in `ModelCheckpoint` was `val_loss` and the metric used in `EarlyStopping` was `val_accuracy` and even though if one of them increases the other one “generally tends to decrease”, we can not claim an increase in one will necessarily decrease the other one or vice versa. This is apparent from the above example where epoch 13 led to the lowest `val_loss` (and therefore, the “best” model), but the simulations stopped at epoch=67 because `patience` was 20 and epoch=47 led to the highest `val_accuracy`.

13.5.5 Evaluating and Predicting

We can evaluate the performance of a trained network on a given test data using the `evaluate()` method of `keras.Model` class. It returns the loss and metric(s) that were specified in compiling the model. For efficiency purposes (and facilitating parallel computation on GPU), the computation occurs over batches, which by default have size 32 (can be changed).

Furthermore, when we talk about evaluating a model, we generally wish to use the “best” model; that is, the model that led to the best value of metric used in `callbacks.ModelCheckpoint(monitor=metric)`. Note that this is the model saved at the end of the training process if `ModelCheckpoint` is used with `save_best_only=True` and it is different from the model fitted at the end of training process. For example, in the MNIST classification that we have seen so far, we would want to use `best_model.keras`, and not `mnist_model`, which holds the weights obtained at the end of training process. Therefore, we need to load and use the saved best model for prediction:

```
# load and evaluate the "best" model
best_mnist_model = keras.models.load_model("best_model.keras")
loss, accuracy = best_mnist_model.evaluate(X_test, y_test_1_hot,_
    verbose=1)
print('Test accuracy of the model with lowest loss on validation set,_
    (the best model) = {:.3f}'.format(accuracy))
```

```
313/313 [=====] - 0s 681us/step - loss: 0.1037 -
accuracy: 0.9694
Test accuracy of the model with lowest loss on validation set (the best_
model) = 0.969
```

Similarly, we would like to use the “best” model for predicting targets for new observations (inference)—in the above code, once model is loaded, it is called `best_mnist_model`. In this regard, we have couple of options as discussed next:

1. we can use `predict()` method of `keras.Model` class to obtain probabilities of an observation belonging to a class. The computation is done over batches, which is useful for large-scale inference:

```

prob_y_test_pred = best_mnist_model.predict(X_test)
print("the size of predictions is (n_sample x n_classes):",_
      prob_y_test_pred.shape)
print("class probabilites for the first instance:\n",_
      prob_y_test_pred[0])
print("the assigned class for the first instance is: ",_
      prob_y_test_pred[0].argmax()) # the one with the highest probability
print("the actual class for the first instance is: ", y_test_1_hot[0]._
      argmax())
y_test_pred = prob_y_test_pred.argmax(axis=1)
print("predicted classes for the first 10 isntances:", y_test_pred[:10])

```

the size of predictions is (n_sample x n_classes): (10000, 10)
 class probabilites for the first instance:
 [2.2035998e-08 1.5811942e-06 1.0725732e-06 2.2704395e-05 1.6993125e-08
 2.1640132e-07 1.2613171e-12 9.9997342e-01 1.9071523e-07 8.7907057e-07]
 the assigned class for the first instance is: 7
 the actual class for the first instance is: 7
 classes fir the first 10 isntances: [7 2 1 0 4 1 4 9 5 9]

2. for small number of test data, we can use `model(x)`:

```

prob_y_test_pred = best_mnist_model(X_test).numpy() # calling numpy()_
      ↵on a tensorflow tensor creates numpy arrays
print("the size of predictions is (n_sample x n_classes):",_
      prob_y_test_pred.shape)
print("class probabilites for the first instance:\n",_
      prob_y_test_pred[0])
print("the assigned class is: ", prob_y_test_pred[0].argmax()) # the_
      ↵one with the highest probability
print("the actual class is: ", y_test_1_hot[0].argmax())
y_test_pred = prob_y_test_pred.argmax(axis=1)
print("predicted classes for the first 10 isntances:", y_test_pred[:10])

```

the size of predictions is (n_sample x n_classes): (10000, 10)
 class probabilites for the first instance:
 [2.2036041e-08 1.5811927e-06 1.0725712e-06 2.2704351e-05 1.6993125e-08
 2.1640091e-07 1.2613171e-12 9.9997342e-01 1.9071486e-07 8.7906972e-07]
 the assigned class is: 7
 the actual class is: 7
 predicted classes for the first 10 isntances: [7 2 1 0 4 1 4 9 5 9]

At this stage, as we can compute the class probabilites (scores) and/or classes themselves, we may want to use various classes of `sklearn.metrics` to evaluate the final model using different metrics:

```

from sklearn.metrics import accuracy_score, confusion_matrix,_
      ↵roc_auc_score

print("Accuracy = {:.4f}".format(accuracy_score(y_test, y_test_pred)))

```

```
print("Confusion Matrix is\n {}".format(confusion_matrix(y_test,_
->y_test_pred)))
print("Macro Average ROC AUC = {:.3f}".format(roc_auc_score(y_test,_
->prob_y_test_pred, multi_class='ovr', average='macro')))
```

```
Accuracy = 0.9779
Confusion Matrix is
[[ 971     0     1     0     1     1     1     3     1     1]
 [  0 1124     2     3     0     1     2     1     2     0]
 [  4   1 1005     8     2     0     2     3     7     0]
 [  0     0     4  989     0     5     0     6     4     2]
 [  1     0     2     0  950     0     6     3     3    17]
 [  3     0     0     6     1  873     4     0     4     1]
 [  8     3     1     0     2     4  938     0     2     0]
 [  2     4     9     4     0     0     0 1005     1     3]
 [  4     1     1     3     3     5     2     3  949     3]
 [  3     5     0     4     8     5     0     7     2  975]]
```

Macro Average ROC AUC = 1.000

13.5.6 CPU vs. GPU Performance

It is quite likely that executing the MNIST classification code on a typical modern laptop/desktop with no GPU hardware/software installed (i.e., using only CPUs) is more efficient (faster) than using GPUs (e.g., available through Google Colab). Why is that the case?

GPs are best suitable when there is a *large amount of computations that can be done in parallel*. At the same time, computations in training neural networks are *embarrassingly parallelizable*. However, we may not see the advantage of using GPUs with respect to CPUs if “small”-scale computations are desired (for example, training a small size neural networks as in this example). To examine the advantage of using GPU with respect to CPU, in Exercise 1 we scale up the size of the neural network used in this example and compare the execution times.

13.5.7 Overfitting and Dropout

Overfitting: Here is a comment about overfitting from renowned figures in the field of machine learning and pattern recognition ([Duda et al., 2000](#), p. 16):

While an overly complex system may allow perfect classification of the training samples, it is unlikely perform well on new patterns. This situation is known as overfitting’.

In other words, in machine learning overfitting could be merely judged by the performance of a trained model on training data and unseen data. We can claim a model is overfitted if the classifier/regressor has a perfect performance on training data but it exhibits a poor performance on unseen data. The question is whether we

can judge overfitting by the data at hand. Recall what we do in model selection when we try to tune a hyperparameter on a validation set (or using cross-validation if we desire to remove the bias to a specific validation set): we monitor the performance of the model on the validation set as a function of the hyperparameter and pick the value of the hyperparameter that leads to the best performance. The reason we use a validation set is that the performance on that serves as a proxy for the performance of the model on test set. Therefore, we can use it to judge the possibility of overfitting as well. In this regard, Duda *et al.* write ([Duda et al., 2000](#), p. 16):

For most problems, the training error decreases monotonically during training [as a function of hyperparameter], Typically, the error on the validation set decreases, but then increases, an indication that the classifier may be overfitting the training data. In validation, training or parameter adjustment is stopped at the first minimum of the validation error.

There are a few noteworthy points to consider:

1. To be able to judge whether a classifier is overfitting or not, we need two components: the performance of the classifier on training data and its performance on unseen data. There is no problem with finding the performance on training data; after all, it is already available and the apparent error shows how well the classifier performs on training data. However, the only means to judge the performance on unseen data are evaluation rules. Therefore, as stated by Duda *et al.*, a decrease and then increase in validation error (as a function of hyperparameter) is an “indication that the classifier may be overfitting”. Although generally such a behaviour is perceived and referred to as an indicator of overfitting, there is an implicit uncertainty in our ability to judge the overfitting.
2. We may use other performance metrics such as AUC, F_1 -score or even the loss function as indicators of overfitting; after all, they are all metrics of *performance* that measure different aspects of a predictions made by a trained model;
3. There are situations that the validation error (or loss) as a function of hyperparameter values becomes like a plateau, while the training error improves. We can not blindly perceive such a situation as an indicator of overfitting unless the training error is remarkably better than validation error. This is because this situation is generally due to: 1) the optimization algorithm (optimizer) is stuck in a local minima; and/or 2) the algorithm is actually resistant to overfitting such that validation error does not start increasing! For the second case, albeit in connection with AdaBoost, see the following comment from ([J. Friedman, 2000](#)):

In fact, Breiman (1996) (referring to a NIPS workshop) called AdaBoost with trees the “best off-the-shelf classifier in the world” [see also Breiman (1998b)]. Interestingly, in many examples the test error seems to consistently decrease and then level off as more classifiers are added, rather than ultimately increase. For some reason, it seems that AdaBoost is resistant to overfitting.

With this explanation about overfitting, let us examine the performance curves obtained before for the MNIST application. Looking into the loss curve as a function of epoch in Fig. 13.8 shows that after epoch 13, the validation loss increases, which

is an indicator that the classifier may be overfitting. There are various ways to prevent possible overfitting. A popular approach is the concept of *dropout* that we discuss next.

Dropout: Since its conception in (Srivastava et al., 2014), dropout has become a popular approach to guard against overfitting. The idea is to randomly omit each neuron in hidden layers or even each input feature with a pre-specified probability known as *dropout rate*. Although dropout rate is typically set between 0.2 to 0.5, it is itself a hyperparameter that should be ideally tuned in the process of model selection. However, typically the dropout rate for an input feature is lower (e.g., 0.2) than a neuron in a hidden layer (e.g., 0.5). By omitting a unit, we naturally omit any connection to or from that unit as well. Fig. 13.9 shows what is meant by omitting some neurons/features along with their incoming and outgoing connections. To facilitate implementation, it is generally assumed the same dropout rate for all units (neuron or an input feature) in the same layer.

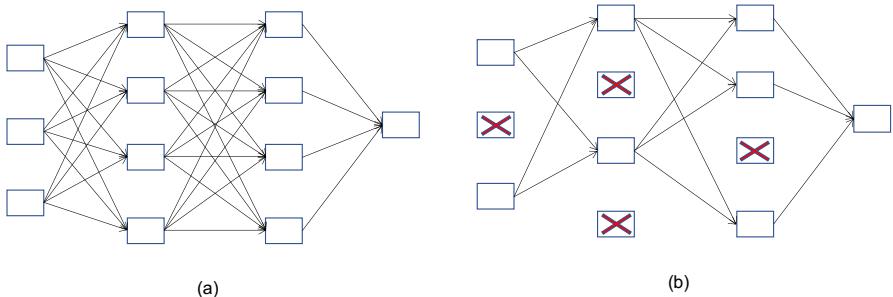


Fig. 13.9: (a) an MLP when no dropout is used; (b) an MLP when a dropout is used

The mini-batch gradient descent that was discussed before is used to train neural networks with dropout. The main difference lies in training where for each training instance in a mini-batch, a network is sampled by dropping out some units from the full architecture. Then the forward and backward computations that are part of the backprop algorithm are performed for that training instance in the mini-batch over the sampled network. The gradient for each weight that is used in optimizer to move the weight is averaged over the training instances in that mini-batch. In this regard, if for a training instance, its corresponding sampled network does not have a weight, the contribution of that training instance towards the gradient is set to 0. At the end of this process, we obtain estimates of weights for the full network (i.e., all units and their connections with no dropout). In testing stage, the full network is used (i.e., all connections with no dropout). However, the outgoing weights from a unit are multiplied by $(1 - p)$ where p is the dropout rate for that unit. This approximation accounts for the fact that the unit was active in training with probability of $(1 - p)$. In other words, by this simple approximation and using the full network in testing

we do not need to use many sampled networks with shared weights that were used in training.

In Keras, there is a specific layer, namely, `keras.layers.Dropout` class, that can be used to implement the dropout. In this regard, if a dropout for a specific layer is desired, a `Dropout` layer should be added right after that. Let us now reimplement the MNIST application (for brevity part of the output is omitted):

```
        layers.Dense(64, activation="sigmoid"),
        layers.Dropout(0.3),
        layers.Dense(10, activation="softmax")
    ])

# compiling model
mnist_model.compile(optimizer="adam", loss="categorical_crossentropy",
                     metrics=["accuracy"])

# training model
my_callbacks = [
    keras.callbacks.EarlyStopping(
        monitor="val_accuracy",
        patience=20),
    keras.callbacks.ModelCheckpoint(
        filepath="best_model.keras",
        monitor="val_loss",
        save_best_only=True,
        verbose=1)
]

start = time.time()
history = mnist_model.fit(x = X_train,
                           y = y_train_1_hot,
                           batch_size = 32,
                           epochs = 200,
                           validation_data = (X_val, y_val_1_hot),
                           callbacks=my_callbacks)

end = time.time()
training_duration = end - start
print("training duration = {:.3f}".format(training_duration))
print(history.history.keys())
print(mnist_model.summary())

# plotting the results
epoch_count = range(1, len(history.history['loss']) + 1)
plt.figure(figsize=(13,4))
plt.subplot(121)
plt.plot(epoch_count, history.history['loss'], 'b', label = 'training loss')
plt.plot(epoch_count, history.history['val_loss'], 'r', label = 'validation loss')
plt.legend()
plt.ylabel('loss')
plt.xlabel('epoch')
plt.subplot(122)
```

```

plt.plot(epoch_count, history.history['accuracy'], 'b', label =_
    'training accuracy')
plt.plot(epoch_count, history.history['val_accuracy'], 'r', label =_
    'validation accuracy')
plt.legend()
plt.ylabel('accuracy')
plt.xlabel('epoch')

```

Epoch 1/200

1407/1407 [=====] - 3s 2ms/step - loss: 1.2182 -
accuracy: 0.6240 - val_loss: 0.2984 - val_accuracy: 0.9124

Epoch 00001: val_loss improved from inf to 0.29836, saving model to
best_model.keras

...

Epoch 74/200

1407/1407 [=====] - 2s 1ms/step - loss: 0.0308 -
accuracy: 0.9904 - val_loss: 0.1008 - val_accuracy: 0.9779

Epoch 00074: val_loss did not improve from 0.08361

Epoch 75/200

1407/1407 [=====] - 2s 1ms/step - loss: 0.0295 -
accuracy: 0.9904 - val_loss: 0.1008 - val_accuracy: 0.9780

Epoch 00075: val_loss did not improve from 0.08361

training duration = 136.439

dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 128)	100480
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 64)	8256
dropout_1 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 10)	650

Total params: 109,386

Trainable params: 109,386

Non-trainable params: 0

None

```

# evaluate on test data
best_mnist_model = keras.models.load_model("best_model.keras")

```

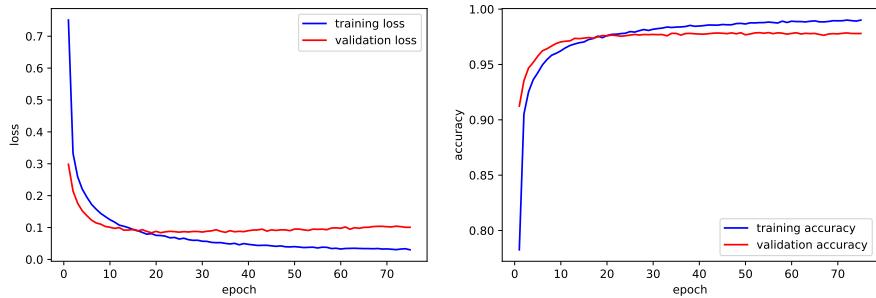


Fig. 13.10: The loss (left) and the accuracy (right) of `mnist_model` trained using dropout as functions of epoch on both the training set and the validation set. The figure shows that with dropout the performance curves over training and validation sets do not diverge as in Fig. 13.8.

```
loss, accuracy = best_mnist_model.evaluate(X_test, y_test_1_hot, ↵
                                         verbose=1)
print('Test accuracy of the dropout model with lowest loss (the best ↵
      model) = {:.3f}'.format(accuracy))
```

```
313/313 [=====] - 0s 735us/step - loss: 0.0800 - ↵
accuracy: 0.9772
Test accuracy of the dropout model with lowest loss (the best model) = 0. ↵
977
```

Here we observe that using dropout slightly improved the performance on the test data: 97.7% (with dropout) vs. 96.9% (without dropout). At the same time, comparing Fig. 13.10 with Fig. 13.8 shows that the performance curves over training and validation sets are closer, which indicates that the dropout is indeed guarding against overfitting.

13.5.8 Hyperparameter Tuning

As mentioned in Section 13.1, a major bottleneck in using deep neural networks is the large number of hyperparameters to tune. Although Keras-TensorFlow comes with several tuning algorithms, here we discuss the use of grid search cross-validation implemented by `sklearn.model_selection.GridSearchCV` that was discussed in Section 9.2.1. The entire procedure can be used with random search cross-validation discussed in Section 9.2.2 if `GridSearchCV` and `param_grid` are replaced with `RandomizedSearchCV` and `param_distributions`, respectively.

To treat a Keras classifier/regressor as a scikit-learn estimator and use it along with scikit-learn classes, we can use

- `keras.wrappers.scikit_learn.KerasClassifier`
- `keras.wrappers.scikit_learn.KerasRegressor`

Similar wrappers are also implemented by `scikeras` library ([Scikeras-wrappers, 2023](#)), which supports Keras functional and subclassing APIs as well. As the number of layers and neurons are (structural) hyperparameters to tune, we first write a function, called `construct_model`, which constructs and compiles a sequential Keras model depending on the number of layers and neurons in each layer. Then treating `construct_model` as the argument of `KerasClassifier`, we create the classifier that can be treated as a scikit-learn estimator; for example, to use it within `GridSearchCV` or, if needed, as the classifier used within `Pipeline` class to create a composite estimator along with other processing steps. In what follows, we also treat the dropout rate of the optimizer as well as the number of epochs as two additional hyperparameters to tune—in contrast with `EarlyStopping` callback, here we determine the epoch that jointly with other hyperparameters leads to the highest CV score. For that reason, and to reduce the computational complexity of the grid search, we assume candidate epoch values are in the set {10, 30, 50, 70}.

```
def construct_model(hidden_layers = 1, neurons=32, dropout_rate=0.25, learning_rate = 0.001):
    # building model
    model = keras.Sequential()
    for i in range(hidden_layers):
        model.add(layers.Dense(units=neurons, activation="sigmoid"))
        model.add(layers.Dropout(dropout_rate))

    model.add(layers.Dense(10, activation="softmax"))
    # compiling model
    model.compile(loss='categorical_crossentropy',
                  optimizer=keras.optimizers.Adam(learning_rate),
                  metrics=['acc'])

    return model
```

```
import time
import pandas as pd
from tensorflow import keras
import matplotlib.pyplot as plt
from tensorflow.keras import layers
from tensorflow.keras.datasets import mnist
from sklearn.model_selection import train_test_split
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.model_selection import StratifiedKFold, GridSearchCV

seed_value= 42

# set the seed for Python built-in pseudo-random generator
```

```
import random
random.seed(seed_value)

# set the seed for numpy pseudo-random generator
import numpy as np
np.random.seed(seed_value)

# set the seed for tensorflow pseudo-random generator
import tensorflow as tf
tf.random.set_seed(seed_value)

# data preprocessing
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train, X_test = X_train.astype('float32')/255, X_test.
    ↪astype('float32')/255
X_train, X_test = X_train.reshape(60000, 28*28), X_test.reshape(10000,
    ↪28*28)

strkfold = StratifiedKFold(n_splits=3, shuffle=True)
keras_est = KerasClassifier(build_fn=construct_model, verbose=1)
param_grid = {'hidden_layers': [1, 2], 'neurons': [32, 64], ↪
    ↪'dropout_rate': [0.2, 0.4], 'epochs': range(10,90,20)}
gscv = GridSearchCV(keras_est, param_grid, cv=strkfold)

# training model
score_best_estimator=gscv.fit(X_train, y_train).score(X_test, y_test)

#best model based on grid search cv
print('the highest CV score is: {:.3f}'.format(gscv.best_score_))
print('the best combination is: {}'.format(gscv.best_params_))
print('the accuracy of the best estimator on the test data is: {:.3f}'.
    ↪format(score_best_estimator))
```

```
Epoch 1/10
1250/1250 [=====] - 1s 690us/step - loss: 1.2428
    ↵ acc:
0.6646
Epoch 2/10
1250/1250 [=====] - 1s 674us/step - loss: 0.4253
    ↵ acc:
0.8824
...
the highest CV score is: 0.972
the best combination is: {'dropout_rate': 0.2, 'epochs': 70,
    ↵ 'hidden_layers': 2,
```

```
'neurons': 64}
the accuracy of the best estimator on the test data is: 0.978
```

A detailed view of cross-validation scores can be achieved by looking into `gscv.cv_results_` attribute. In the following code, for brevity we only present the fold-specific CV scores and the overall CV score for all 32 combinations of hyperparameter values. As we can see, the highest `mean_test_score` is 0.972, which is what we saw before by looking into `gscv.best_score_`:

```
pd.options.display.float_format = '{:,.3f}'.format
df = pd.DataFrame(gscv.cv_results_)
df
```

	split0_test_score	split1_test_score	split2_test_score	mean_test_score
0	0.947	0.946	0.945	0.946
1	0.962	0.962	0.959	0.961
2	0.944	0.947	0.943	0.945
3	0.963	0.964	0.960	0.962
4	0.961	0.956	0.953	0.957
5	0.970	0.971	0.966	0.969
6	0.957	0.959	0.954	0.957
7	0.971	0.970	0.967	0.969
8	0.958	0.960	0.957	0.958
9	0.971	0.970	0.968	0.970
10	0.957	0.958	0.953	0.956
11	0.972	0.973	0.969	0.971
12	0.960	0.961	0.958	0.960
13	0.973	0.972	0.969	0.971
14	0.958	0.960	0.954	0.958
15	0.975	0.971	0.969	0.972
16	0.939	0.941	0.940	0.940
17	0.957	0.958	0.953	0.956
18	0.924	0.925	0.922	0.924
19	0.950	0.950	0.948	0.949
20	0.949	0.951	0.948	0.949
21	0.968	0.966	0.963	0.965
22	0.940	0.940	0.935	0.939
23	0.964	0.963	0.960	0.962
24	0.952	0.953	0.949	0.951
25	0.969	0.969	0.966	0.968
26	0.942	0.943	0.940	0.942
27	0.966	0.966	0.963	0.965
28	0.954	0.955	0.952	0.954
29	0.969	0.970	0.966	0.968
30	0.947	0.947	0.944	0.946
31	0.966	0.964	0.963	0.964

Exercises:

Exercise 1: In the MNIST application presented in this chapter, use the same parameter for compiling and fitting except the following three changes:

1. replace the MLP with another MLP having 5 hidden layers with 2048 neurons in each layer.
 2. remove the early stopping callback and instead train the model for 10 epochs.
 3. modify the code to add a dropout layer with a rate of 0.2 to the input feature vector (flattened input). Do not use dropout in any other layer. Hint: Use `Flatten()` layer to flatten input images. This allows adding a dropout layer before the first hidden layer.
- (A) Execute the code only on CPUs available in a personal machine. What is the execution time?
- (B) Execute the code on GPUs provided by Colab. What is the execution time?
- (C) Suppose each weight in this network is stored in memory in 4 bytes (float32). What is approximately the minimum amount of memory (RAM) in mega byte (MB) required to store all parameters of the network? (the answer to questions of this type are specially insightful when, for example, we wish to deploy a trained network on memory-limited embedded devices).

Exercise 2: We use a multi-layer perceptron (MLP) in a regression problem to estimate the oil price (a scalar quantity) based on 10 variables (therefore, our input feature vectors are 10 dimensional and the output is univariate). Our MLP has two hidden layers with 50 neurons in the first hidden layer (the layer closer to the input) and 20 in the second hidden layer (the layer closer to the output). What is the total number of parameters that should be learned in this network?

Exercise 3: We have the perceptron depicted in Fig. 13.11 that takes binary inputs x_1 and x_2 ; that is to say, $x_i = 0, 1, i = 1, 2$ and lead to an output y . With choices of weights indicated below, the network leads to an approximate implementation of which logical gates (OR gate, AND gate, NAND gate, XOR gate)?

- A) with $a_1 = a_2 = 50$ and $a_0 = -75$ the network implements _____ gate
- B) with $a_1 = a_2 = -50$ and $a_0 = 75$ the network implements _____ gate

Exercise 4: Suppose the perceptron depicted in Fig. 13.12 is used in a binary classification problem. What activation function, what threshold T , and what loss function should be used to make the classifier $\psi(\mathbf{x})$ equivalent to logistic regression?

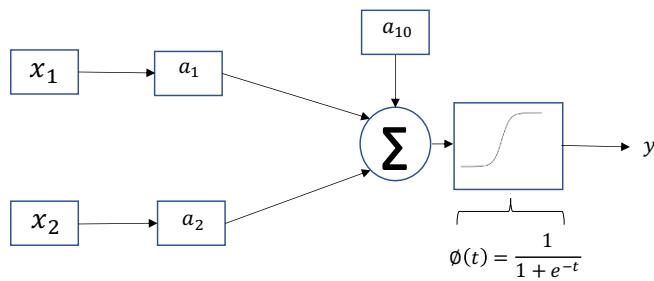


Fig. 13.11: he perceptron regressor used in Exercise 3.

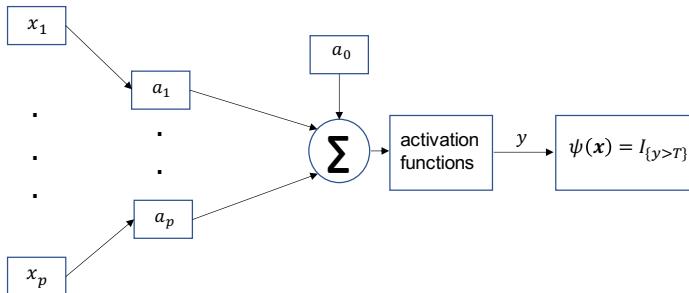


Fig. 13.12: The perceptron classifier used in Exercise 4.

Exercise 5: Suppose for training a neural network in an application with a training sample size of 10000, we use mini-batch gradient descent with a batch size of 20. How many weight updates occur in one epoch?

- A) 500 B) 50 C) 20 D) 1

Exercise 6: Suppose in training a neural network using Keras-TensorFlow, we set the epochs parameter of the `fit()` method of the `keras.Model` class to 500. Furthermore, we use `EarlyStopping` callback with the value of `patience` parameter being 50 and we set the `monitor` parameter of the callback to `val_accuracy`. The training stops at epoch 170. What is the epoch that had the highest “validation accuracy”?

- A) 170 B) 220 C) 120 D) 70 E) 500

Exercise 7: Suppose someone has trained a multi-layer perceptron in a regression problem to estimate the oil price (a scalar quantity) based on 100 variables (therefore, our input feature vectors are 100 dimensional and the output is univariate). The MLP has 50 hidden layers with 1000 neurons in each hidden layer. Suppose to store each parameter of the model we need 4 bytes. What is approximately the minimum

amount of memory (in MB) require to store all parameters of the model?

- A) 20 MB B) 44 MB C) 88 MB D) 176 MB E) 196 MB

Exercise 8: We use a multi-layer perceptron with two hidden layers to estimate a univariate output using a univariate input x . Suppose each hidden layer has 100 neurons and the ReLU activation function. The output layer has no activation function. Assuming all weights and biases in the network are 1, what is the output for $x = -2$?

- A) -1 B) 0 C) 1 D) 101 E) 10101



Chapter 14

Convolutional Neural Networks

A series of successful applications of Convolutional Neural Networks (CNNs) in various computer vision competitions in 2011 and 2012 were a major driving force behind the momentum of deep learning that we see today. As an example, consider the 2012 ImageNet competition where a deep convolutional neural network, later known as AlexNet, reported a top-5 test error rate of 15.3% for image classification. This error rate was remarkably better than the runner up error of 26.2% achieved by a combination of non-deep learning methods. Following the success of deep learning techniques in 2012, many groups in 2013 used deep CNNs in the competition. This state of affairs is in large part attributed to the ability of CNNs to integrate feature extraction into model construction stage. In other words, and in analogy with embedded feature selection methods discussed in Chapter 10, CNNs have an embedded feature extraction mechanism that can be very helpful when working with multidimensional signals such as time-series, digital images, or videos. There are even some efforts to leverage this important property of CNNs for tabular data types. In this chapter we describe the working mechanism of these networks, which is essential for proper implementation and use. Last but not least, the practical software implementation in Keras will be discussed.

14.1 CNN, Convolution, and Cross-Correlation

The embedded feature extraction mechanism along with training efficiency and the performance have empowered CNNs to achieve breakthrough results in a wide range of areas such as image classification (Krizhevsky et al., 2012; Simonyan and Zisserman, 2015), object detection (Redmon et al., 2016), electroencephalogram (EEG) classification (Lawhern et al., 2018), and speech recognition (Abdel-Hamid et al., 2014), to just name a few. Consider an image classification application. The conventional way to train a classifier is to use some carefully devised feature extraction methods such as HOG (Histogram of Oriented Gradients) (Dalal and Triggs, 2005), SIFT (Scale Invariant Feature Transform) (Lowe, 1999), or SURF (Speeded-Up Ro-

bust Feature) (Bay et al., 2006), and use the extracted features in some conventional classifiers such as logistic regression or kNN. This way, we are restricted to a handful of feature extraction algorithms that can heavily influence the performance of our classifiers. However, in training CNNs features are extracted in the process of model construction. A CNN is essentially an ANN that uses convolution operation in at least one of its layers. Therefore, to understand the working mechanism of CNN, it is important to have a grasp of the convolution operation.

Suppose $x[i]$ and $k[i]$ are two discrete 1-dimensional signals, where i is an arbitrary integer. In the context of convolutional neural networks, $x[i]$ is the input signal, and $k[i]$ is referred to as kernel, albeit as we will see later both have generally higher dimensions than one. The convolution of x and k , denoted $x * k$, is

$$y[m] \triangleq (x * k)[m] = \sum_{i=-\infty}^{\infty} x[i]k[m-i], \quad \forall m. \quad (14.1)$$

This can be seen as flipping the kernel, sum the products, and slide the kernel over the input. One can easily show that convolution operation is commutative; that is, $(x * k) = (k * x)$. This is due to the fact that in convolution definition, we are flipping one signal with respect to another. If we do not flip one, we will have the *cross-correlation* between x and k given by

$$y[m] \triangleq (x \star k)[m] = \sum_{i=-\infty}^{\infty} x[i]k[i-m], \quad \forall m. \quad (14.2)$$

Cross-correlation, however, is not commutative. However, flipping or not makes sense when $k[i]$ has known values in the first place (so that we can judge whether inside the summation is flipped with respect to $k[i]$ or not). In cases where $k[i]$ itself is unknown and is the objective of a learning process that uses (14.1) or (14.2), flipping does not really matter. For example, suppose a $k[i]$ is learned by using (14.2) in a learning process. One can assume that what has been learned is a flipped version of another signal called $k'[i]$. Because using $k'[i]$ in (14.1) is equivalent to using $k[i]$ in (14.2), we can conclude that $k[i]$ could be also learned by using (14.1) in the same learning process (to find $k'[i]$ and then flip to obtain $k[i]$). In learning CNNs, learning $k[i]$ is the objective. Therefore, many existing libraries use cross-correlation in the learning process of CNN but we still refer to these networks as “convolutional” neural networks.

14.2 Working Mechanism of 2D Convolution

Depending on the type of data, convolution operation is generally categorized as 1D, 2D, or 3D convolutions. In particular, 1D, 2D, and 3D convolutions are generally

used for processing multivariate time-series, images, and videos, respectively. The distinguishing factor between these types of convolution is the number of dimensions through which the kernel slides. For example, in 1D convolution, the kernel is slid in one dimension (through time steps), whereas in 2D convolution, the kernel is slid in two dimensions (e.g., in both height and width directions of an image). This also implies that, if desired, we can even use 2D convolution with sequential data such as multivariate time-series.

As the majority of applications are focused on the utility of 2D convolutions (this does not mean others are not useful), in what follows we elaborate in detail the working principles of 2D convolutions. The use of “2D” convolution, should not be confused with the dimensionality of input and kernel tensors used to hold the data and the kernel. For example, in the 2D convolution that is used in CNN, each input image is stored in a 3D tensor (rank-3 tensor: height, width, and depth) and the kernel is a 4D tensor (rank-4 tensor: height, width, input depth, and output depth). To better understand how the 2D convolution operates on a 3D tensor of input and 4D tensor of kernel, we first start with a simpler case, which is the convolution of a 2D input tensor with a 2D kernel tensor.

14.2.1 Convolution of a 2D Input Tensor with a 2D Kernel Tensor

Convolution and sparse connectivity: Suppose we have a 2D tensor $X[i, j]$ (hereafter referred to as an image or input) and a 2D kernel $K[i, j]$. Although in theory there is no restriction on the width or the height of X and K , in training CNN, kernels are generally taken to be much smaller than inputs. For example, regardless of the size of X , K is assumed to be matrices of size 3×3 or 5×5 . This property of CNNs is known as *sparse connectivity* (also referred to as *sparse interactions*) because regardless of the size of input, useful features are extracted by relatively small-size kernels. This property also leads to efficiency in training because fewer parameters are stored in memory. At the same time, because an objective in training CNNs is to *estimate* the parameters of kernels themselves (the process that leads to learning the feature extractor), sparse connectivity also implies fewer parameters to be estimated.

The result of convolving (with no flipping) X and K can be obtained by sliding K over X and each time sum all the element-wise products of weights of K and values in “patches” of X (of the same size as K). Fig. 14.1 shows an example of convolution operation. The dotted area in the middle of X shows elements of X at which K could be centered to have a legitimate convolution; otherwise (i.e., placing the center of K outside this area) leads to an illegitimate convolution, which means there will be no X element to multiply by a K element. For this reason the size of Y (the output of convolution) is smaller than X . In particular, if X has a height h_X and width w_X (i.e., size $h_X \times w_X$), and the kernel has a size of $h_K \times w_K$, the size of Y will be $(h_X - h_K + 1) \times (w_X - w_K + 1)$.

Padding: To enforce having an output with the same size as the input, we can add a sufficient number of rows and columns of zeros to the input—this process is known

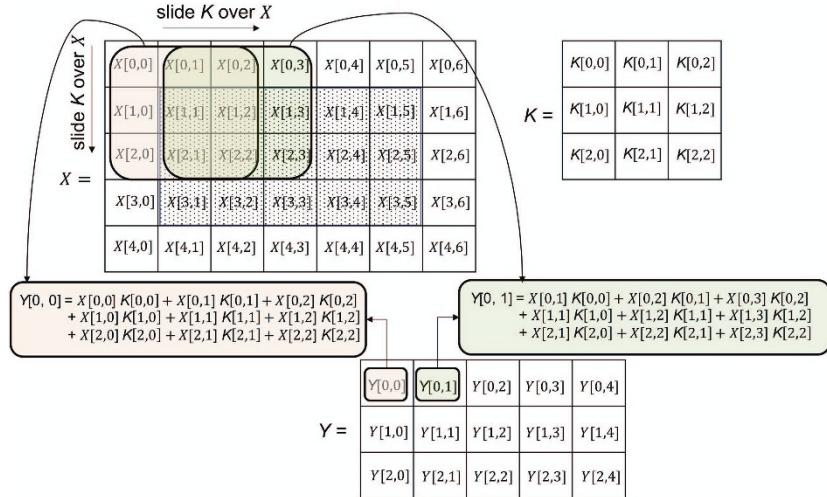


Fig. 14.1: The “convolution” operation for a 2D input signal X and a 2D kernel K to create the output Y . The dotted area in the middle of X shows elements of X at which K could be centered to have a legitimate convolution.

as *zero padding*. In this regard, we need to add $h_X - (h_X - h_K + 1) = h_K - 1$ and $w_K - 1$ rows and columns of zeros, respectively. It is common to perform this symmetrically; that is to say, adding $\frac{h_K-1}{2}$ rows of zeros on top and the same number of rows at the bottom of the input (and similarly, adding $\frac{w_K-1}{2}$ columns to the left and the same number of columns to right of the input). Naturally, this would make sense if h_K and w_K are odd numbers. Fig. 14.2 shows the convolution conducted in Fig. 14.1 except that zero padding is used to have an output with the same size as the input.

Neuron and feature map: Similar to the MLP that we saw in Chapter 13, a bias term is added to the summation used in convolution, and the result is then used as the input to some activation function. This resembles the artificial neuron that we saw earlier in Chapter 13, except that here the inputs to a neuron is determined by small patches of the image. The collection of the outputs of all neurons in the same layer produces what is known as the *feature map* (also referred to as activation map), which can itself be depicted as an image. Therefore, the output of one neuron produces one value in the feature map. Fig. 14.3 shows two artificial neurons used in a CNN layer (compare with MLP in Fig. 13.1) and two “pixels” in the output feature map.

Although sometimes “feature map” is used to simply point to the output of the convolution operation (e.g., (Goodfellow et al., 2016, p. 333)), in the context of CNN where each layer possesses an activation function, we refer to the output of neurons (after the activation function) as the feature map—this way we can say, for example,

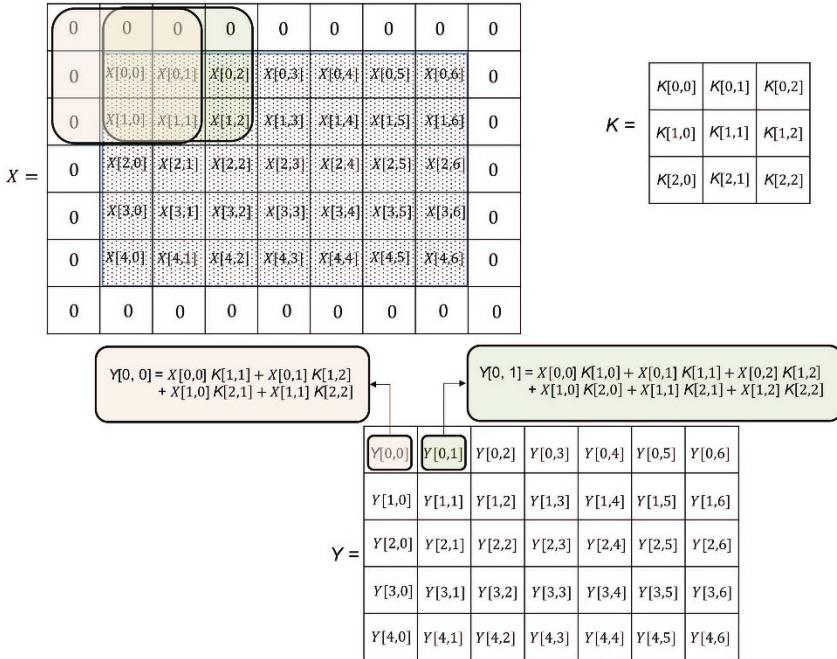


Fig. 14.2: The “convolution” with zero padding for a 2D input signal X and a 2D kernel K to create an output Y of the same size as X . The dotted area in the middle of X shows elements of X at which K could be centered to have a legitimate convolution.

the output feature map from one layer is used as the input to the next layer in a multi-layer CNN.

Parameter sharing: Parameter sharing means that all neurons in the same layer share the same weights (kernel). This is because the small-size kernel (due to sparse connectivity) slides over the entire input tensor, which, in turn, produces the inputs to each neuron by multiplying different patches of the input tensor with the same kernel. Beside the sparse connectivity, this property (i.e., parameter/kernel sharing between neurons in the same layer) is another factor that leads to a small number of parameters to estimate. At the same time, a kernel, when learned, serves as a feature detector. For example, if the learned kernel serves as a vertical edge detector, convolving that over an image and looking at the outputs of neurons (i.e., feature maps) tells us the areas of the image at which this feature is located. This is because we apply the same feature detector over the entire image.

Strides: Rather than the regular convolution that was described earlier in which the kernel is slid with a step (stride) of 1 in both direction, we can have strided convolution in which the stride is greater than 1. We can even have different stride values across height and width directions. Fig. 14.4 shows a strided convolution

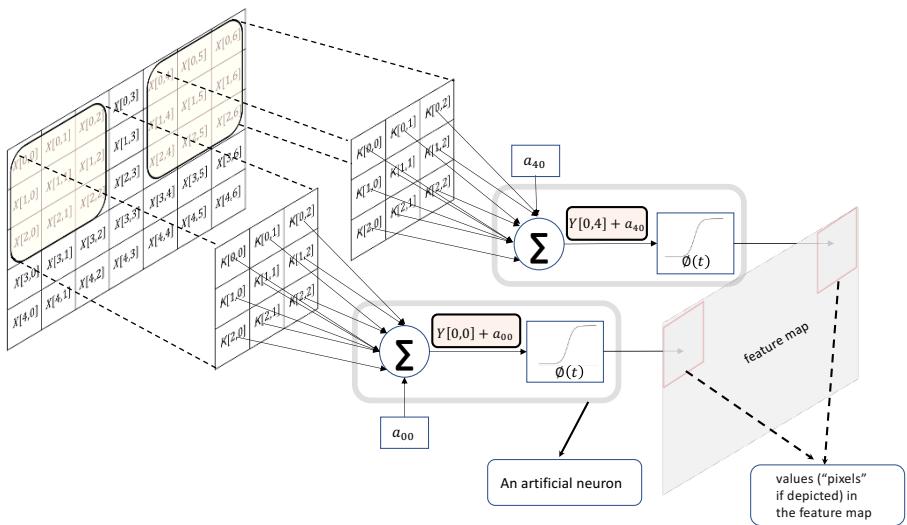


Fig. 14.3: Each artificial neuron in a CNN layer produces one “pixel” in the output feature map.

where the stride in the height and width directions are 2 and 3, respectively. If h_s and w_s denote strides in the direction of height and width, respectively, then the output of the strided convolution is $(\lfloor \frac{h_x-h_k}{h_s} \rfloor + 1) \times (\lfloor \frac{w_x-w_k}{w_s} \rfloor + 1)$. As we can see, the effect of a stride greater than 1 is downsampling the feature map by a rate equal to the stride value.

Max-pooling: In CNN, it is typical to perform *pooling* once output feature maps are generated. This is done by using a pooling layer right after the convolutional layer. Pooling replaces the values of the feature map over a small window with a summary statistic obtained over that window. There are various types of pooling that can be used; for example, *max-pooling* replaces the values of the feature map within the pooling window with one value, which is their maximum. *Average-pooling*, on the other hand, replaces these values with their average.

Regardless of the actual function used for pooling, it is typical to use windows of 2×2 that are shifted with a stride of 2—the concept of “stride” is the same as what we saw before for the convolution operation and the magnitude of strides used for pooling does not need to be the same as the convolutional strides. Similar to the use of convolutional stride that was discussed before, using pooling in this way leads to smaller feature maps; that is to say, feature maps are downsampled with a rate of 2. From a computational perspective, downsampling helps reduce the number of

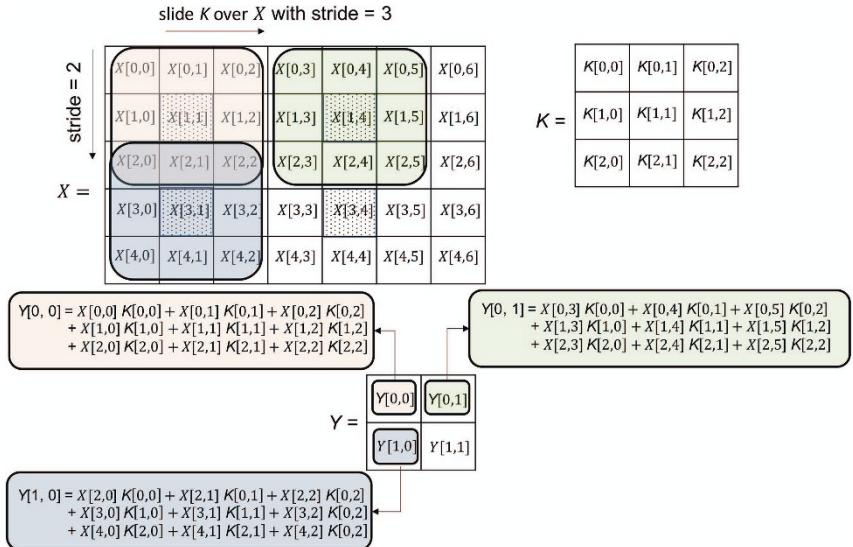


Fig. 14.4: Strided convolution. The effect of a stride greater than 1 is downsampling the feature map.

parameters that eventually should be estimated in a CNN. This is because typically the output feature maps of the last convolutional layer is flattened and use as an input to a dense layer. Without downsampling feature maps, here we end up with a huge number of dense layer parameters to estimate (recall that in dense layer every neuron is connected to all input variables).

Another way that max-pooling would help is in *local translation invariance*. This means that if we move the input by a small amount, the output of pooling does not change much because it is a summary of a local neighborhood of the feature map. In some applications this is a desired property because the presence of a feature is more important than where it is exactly located. Fig. 14.5 shows an example that illustrates the mechanism of pooling (in particular max-pooling) as well as the local translation invariance that is achieved by the max-pooling. In this example, we have two inputs X_1 and X_2 where X_2 is obtained by shifting X_1 to the left (plus an additional column on its right). As a result, $Y_2 = X_2 * K$ is similar to $Y_1 = X_1 * K$ except that it is shifted to the left (plus one additional value on its right). This is not surprising as convolution is *translation equivariant*. Nonetheless, if we compare Y_2 and Y_1 element-wise, all values of Y_1 is changed in Y_2 . However, this is not the case with the output of max-pooling operation as in both cases the outcomes are the same.

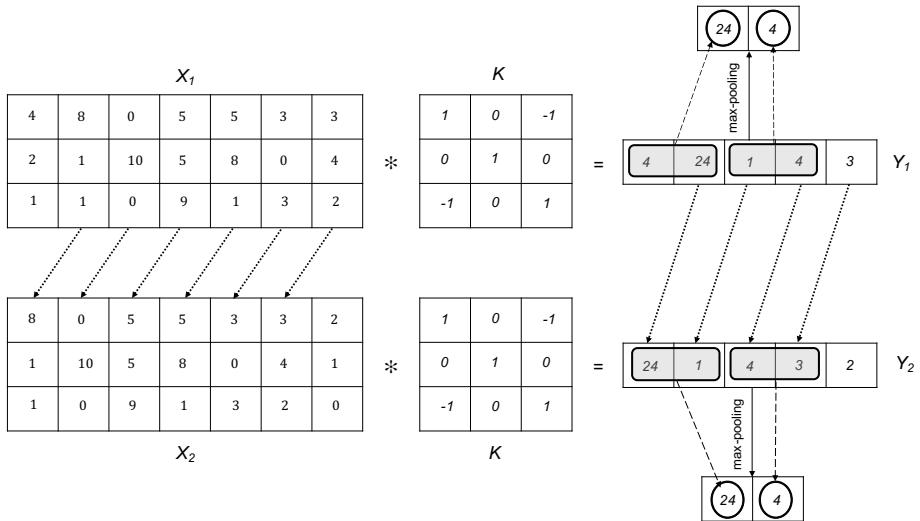


Fig. 14.5: Max-pooling and the local translation invariance.

14.2.2 Convolution of a 3D Input Tensor with a 4D Kernel Tensor

The actual “2D” convolution that occurs in a layer of CNN, is the convolution of a 3D input tensor and a 4D kernel tensor. To formalize the idea, let \mathbf{X} , \mathbf{K} , and \mathbf{Y} denote the 3D input tensor, 4D kernel tensor, and 3D output tensor, respectively. Let $\mathbf{X}[i, j, k]$ denote the element in row i , column j , and channel k of the input \mathbf{X} for $1 \leq i \leq h_{\mathbf{X}}$ (input height), $1 \leq j \leq w_{\mathbf{X}}$ (input width), and $1 \leq k \leq c_{\mathbf{X}}$ (the number of channels or depth). For an image encoded in RGB format, $c_{\mathbf{X}} = 3$ where each channel refers to one of the red, green, and blue channels. Furthermore, here we used the *channels-last* convention to represent the channels as the last dimension in \mathbf{X} . Alternatively, we could have used *channels-first* convention that uses the first dimension to refer to channels.

Furthermore, let $\mathbf{K}[i, j, k, l]$ denote the 4D kernel tensor where $1 \leq i \leq h_{\mathbf{K}}$ (kernel height), and $1 \leq j \leq w_{\mathbf{K}}$ (kernel width), $1 \leq k \leq c_{\mathbf{K}} = c_{\mathbf{X}}$ (input channels or input depth), and $1 \leq l \leq c_{\mathbf{Y}}$ (output channels or output depth). The sparse connectivity assumption stated before means that generally $h_{\mathbf{K}} \ll h_{\mathbf{X}}$ and $w_{\mathbf{K}} \ll w_{\mathbf{X}}$. Furthermore, note that $c_{\mathbf{K}}$ is the same as $c_{\mathbf{X}}$. The fourth dimension can be understood easily by assuming \mathbf{K} containing $c_{\mathbf{Y}}$ filters of size $h_{\mathbf{K}} \times w_{\mathbf{K}} \times c_{\mathbf{X}}$. The convolution (to be precise, cross-correlation) of \mathbf{X} and \mathbf{K} (with no padding of the input) is obtained by computing the 3D output *feature map* tensor \mathbf{Y} with elements $\mathbf{Y}[m, n, l]$ where $1 \leq m \leq h_{\mathbf{Y}} = h_{\mathbf{X}} - h_{\mathbf{K}} + 1$, $1 \leq n \leq w_{\mathbf{Y}} = w_{\mathbf{X}} - w_{\mathbf{K}} + 1$, and $1 \leq l \leq c_{\mathbf{Y}}$, as

$$\mathbf{Y}[m, n, l] = \sum_{i, j, k} \mathbf{X}[i, j, k] \mathbf{K}[m + i, n + j, k, l], \quad (14.3)$$

where the summation is over all valid indices. Fig. 14.6 shows the working mechanism of 2D convolution as shown in (14.3). Few remarks about the 2D convolution as shown here:

1. l in the third dimension of $\mathbf{Y}[m, n, l]$ refers to each feature map (matrix), and all feature maps have the same size $h_{\mathbf{Y}} \times w_{\mathbf{Y}}$;
2. assuming one bias term for each kernel, training a single layer in CNN means estimating $c_{\mathbf{Y}}(c_{\mathbf{Y}} h_{\mathbf{K}} w_{\mathbf{K}} + 1)$ parameters for the kernel using the back-propagation algorithm;
3. when there are multiple convolutional layers in one network, the output feature map of one layer serves as the input tensor to the next layer;
4. all channels of a kernel are shifted with the same stride (see Fig. 14.7); and
5. the pooling, if used, is applied to every feature map separately;

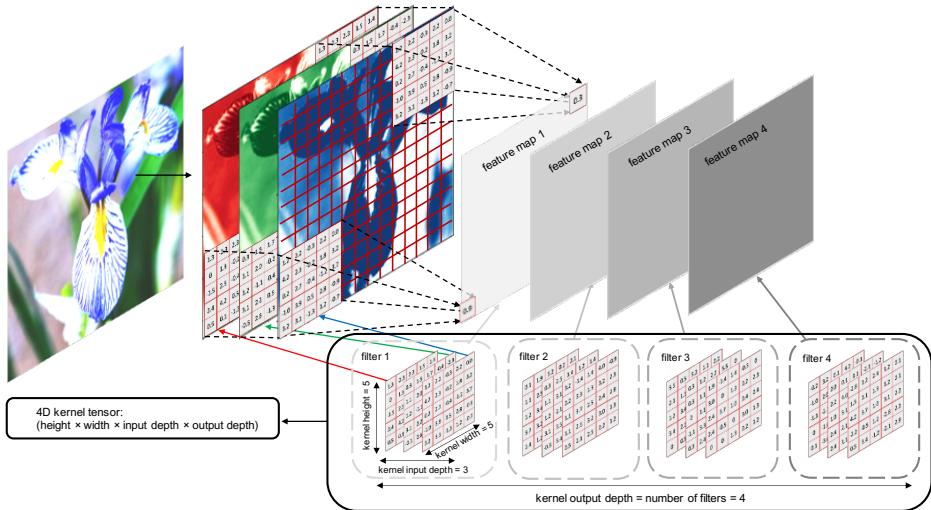


Fig. 14.6: The working mechanism of 2D convolution: the convolution of a 3D input tensor with a 4D kernel tensor.

14.3 Implementation in Keras: Classification of Handwritten Digits

Here we train a CNN with 2D convolution for classification of images of handwritten digits that are part of the MNIST dataset. Many concepts that we discussed

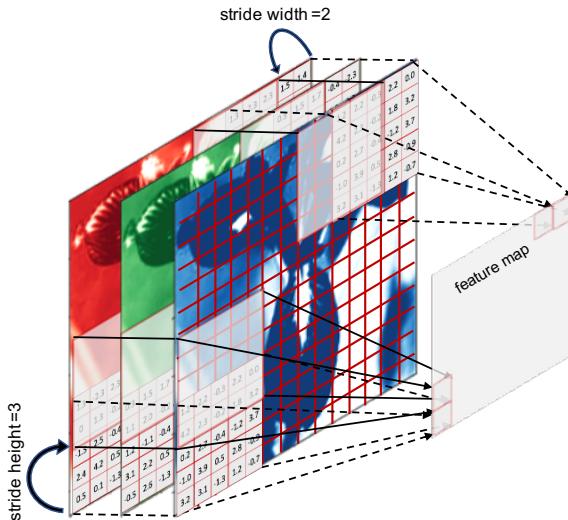


Fig. 14.7: Convolutional strides.

in Chapter 13 directly apply to training CNNs. As before, we need to build, compile, train, and evaluate the trained model using similar classes (e.g., `Sequential`) and methods (e.g., `compile()`, `fit()`, `evaluate()` and `predict()`). However, there are some differences that we list here:

1. the input data: in Section 13.5.1, we vectorized images to create feature vectors. This was done because MLP accepts feature vectors; however, in using CNN, we do not need to do this as CNNs can work with images directly. That being said, each input (image) to a 2D convolution should have the shape of $\text{height} \times \text{width} \times \text{channels}$ (depth). However, our raw images have shape of 28×28 . Therefore, we use `np.newaxis` to add the depth axis;
2. the layers of CNN: to add 2D convolutional layer in Keras, we can use `keras.layers.Conv2D` class. The number of filters (number of output feature maps), kernel size ($h_K \times w_K$), the strides (along the height and the width), and padding are set through the following parameters:

```
keras.layers.Conv2D(filters, kernel_size,
                    strides=(1, 1), padding="valid")
```

where `kernel_size` and `strides` can be either a tuple/list of 2 integers that is applied to height and width directions, respectively, or one integer that is applied to both directions. The default `padding="valid"` causes no padding. Changing that to `padding="same"` results in zero padding;

3. to add max-pooling, we can add

```
keras.layers.MaxPooling2D(pool_size=(2, 2),  
                         strides=(1, 1), padding="valid").
```

pool_size can be either a tuple/list of 2 integers that shows the height and width of the pooling window, or one integer that is then considered as the size in both dimensions;

4. although we can use any activation functions presented in Chapter 13, a recommended activation function in convolutional layers is ReLU;
5. the “classification” part: convolutional layers act like feature detectors (extractors). Regardless of whether we use one or more convolutional layers, we end up with a number of feature maps, which is determined by the number of filters in the last convolutional layer. The question is how to perform classification with these feature maps. There are two options:

- flattening the feature maps of the last convolutional layer using a flattening layer (`layers.Flatten`) and then use the results as feature vectors to train fully connected layers similar to what we have seen in Chapter 13. Although this practice is common, there are two potential problems. One is that due to large dimensionality of these feature vectors, the fully connected layer used here could potentially overfit the data. For this reason, it is recommended to use dropout in this part to guard against overfitting. Another problem is that the fully connected layers used in this part have their own hyperparameters and tuning them entails an increase in the computational needs;
- apply the *global average pooling* (Lin et al., 2014) after the last convolutional layer. This can be done using `keras.layers.GlobalAveragePooling2D`. At this stage, we have two options for the number of filters in the last convolutional layer: 1) we can set the number of filters in the last layer to the number of categories and then use a softmax layer (`keras.layers.Softmax()`) to convert the global average of each feature map as the probability of the corresponding class for the input; and 2) use the number of filters in the last layer as we generally use in CNNs (e.g., increasing the number of filters as we go deeper in the CNN), and then apply the global average pooling. As this practice leads to a larger vector than the number of classes (one value per feature map), we can then use that as the input to a fully connected layer that has a softmax activation and a number of neurons equal to the number of classes. In what follows, we will examine training a CNN classifier on MNIST dataset using these three approaches.

Case 1: Flattened feature maps of the last convolutional layer as the input to two fully connected layers in the “classification” part of the network trained for the MNIST application.

For faster execution, use GPU, for example, provided by the Colab. This case is implemented as follows (for brevity, part of the code output is omitted):

```
import time
import numpy as np
from tensorflow import keras
import matplotlib.pyplot as plt
from tensorflow.keras import layers
from tensorflow.keras.datasets import mnist
from sklearn.model_selection import train_test_split

seed_value= 42

# setting the seed for Python built-in pseudo-random generator
import random
random.seed(seed_value)

# setting the seed for numpy pseudo-random generator
import numpy as np
np.random.seed(seed_value)

# setting the seed for tensorflow pseudo-random generator
import tensorflow as tf
tf.random.set_seed(seed_value)

# scaling (using prior knowledge)
(X_train_val, y_train_val), (X_test, y_test) = mnist.load_data()
X_train_val, X_test = X_train_val.astype('float32')/255, X_test.
˓→astype('float32')/255

# to make each input image a 3D tensor
X_train_val = X_train_val[:, :, :, np.newaxis]
X_test = X_test[:, :, :, np.newaxis]

X_train, X_val, y_train, y_val = train_test_split(X_train_val,_
˓→y_train_val, stratify=y_train_val, test_size=0.25)
y_train_1_hot = keras.utils.to_categorical(y_train, num_classes = 10)
y_val_1_hot = keras.utils.to_categorical(y_val, num_classes = 10)
y_test_1_hot = keras.utils.to_categorical(y_test, num_classes = 10)

# building model
mnist_model = keras.Sequential([
    layers.Conv2D(16, 3, activation="relu",_
˓→input_shape = X_train[0].shape),
    layers.MaxPooling2D(),
    layers.Conv2D(64, 3, activation="relu"),
    layers.MaxPooling2D(),
    layers.Flatten(),
```

```
        layers.Dropout(0.25),
        layers.Dense(32, activation="relu"),
        layers.Dropout(0.25),
        layers.Dense(10, activation="softmax")
    ])

# compiling model
mnist_model.compile(optimizer="adam", loss="categorical_crossentropy",
    metrics=["accuracy"])

# training model
my_callbacks = [
    keras.callbacks.EarlyStopping(
        monitor="val_accuracy",
        patience=5),
    keras.callbacks.ModelCheckpoint(
        filepath="best_model.keras",
        monitor="val_loss",
        save_best_only=True,
        verbose=1)
]

start = time.time()
history = mnist_model.fit(x = X_train,
                           y = y_train_1_hot,
                           batch_size = 32,
                           epochs = 200,
                           validation_data = (X_val, y_val_1_hot),
                           callbacks=my_callbacks)

end = time.time()
training_duration = end - start
print("training duration = {:.3f}".format(training_duration))
print(history.history.keys())
print(mnist_model.summary())

# plotting the results
epoch_count = range(1, len(history.history['loss']) + 1)
plt.figure(figsize=(13,4), dpi=150)
plt.subplot(121)
plt.plot(epoch_count, history.history['loss'], 'b', label = 'training
    loss')
plt.plot(epoch_count, history.history['val_loss'], 'r', label =
    'validation loss')
plt.legend()
plt.ylabel('loss')
```

```

plt.xlabel('epoch')
plt.subplot(122)
plt.plot(epoch_count, history.history['accuracy'], 'b', label =_
    'training accuracy')
plt.plot(epoch_count, history.history['val_accuracy'], 'r', label =_
    'validation accuracy')
plt.legend()
plt.ylabel('accuracy')
plt.xlabel('epoch')

best_mnist_model = keras.models.load_model("best_model.keras")
loss, accuracy = best_mnist_model.evaluate(X_test, y_test_1_hot,_
    verbose=1)
print('Test accuracy of the model with the lowest loss (the best model)_
    == {:.3f}'.format(accuracy))

```

Epoch 1/200
1407/1407 [=====] - 13s 9ms/step - loss: 0.6160 -
accuracy: 0.8051 - val_loss: 0.0902 - val_accuracy: 0.9727

Epoch 00001: val_loss improved from inf to 0.09015, saving model to
best_model.keras
Epoch 2/200
1407/1407 [=====] - 11s 8ms/step - loss: 0.1444 -
accuracy: 0.9571 - val_loss: 0.0566 - val_accuracy: 0.9827

Epoch 00002: val_loss improved from 0.09015 to 0.05660, saving model to
best_model.keras

...

Epoch 24/200
1407/1407 [=====] - 12s 8ms/step - loss: 0.0218 -
accuracy: 0.9928 - val_loss: 0.0442 - val_accuracy: 0.9909

Epoch 00024: val_loss did not improve from 0.03510
training duration = 282.147
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 26, 26, 16)	160
max_pooling2d_2 (MaxPooling2D)	(None, 13, 13, 16)	0
conv2d_3 (Conv2D)	(None, 11, 11, 64)	9280
max_pooling2d_3 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten_1 (Flatten)	(None, 1600)	0

```

dropout_2 (Dropout)           (None, 1600)          0
=====
dense_2 (Dense)              (None, 32)            51232
=====
dropout_3 (Dropout)           (None, 32)            0
=====
dense_3 (Dense)              (None, 10)            330
=====

Total params: 61,002
Trainable params: 61,002
Non-trainable params: 0

None
313/313 [=====] - 1s 2ms/step - loss: 0.0289 -
accuracy: 0.9909
Test accuracy of the model with the lowest loss (the best model) = 0.991

```

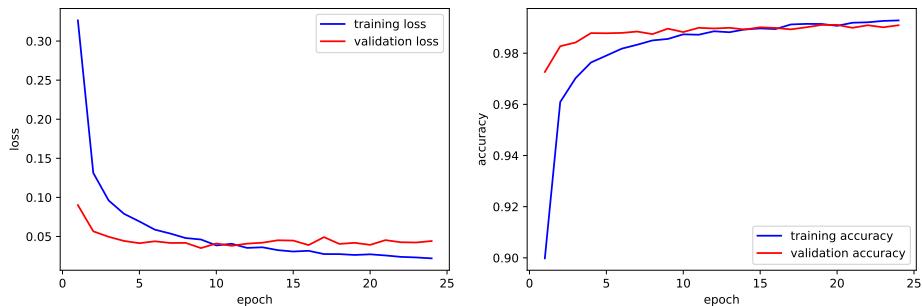


Fig. 14.8: The output of the code in Case 1: (left) loss; and (right) accuracy, as functions of epoch.

Case 2: Global average pooling with the number of filters in the last convolutional layer being equal to the number of classes in the MNIST application.

For this purpose, we replace the “building model” part in the previous code with the following code snippet:

```

# building model
mnist_model = keras.Sequential([
    layers.Conv2D(32, 3, activation="relu", ↴
    ↵input_shape = X_train[0].shape),
    layers.MaxPooling2D(),
    layers.Conv2D(10, 3, activation="relu"),
    layers.MaxPooling2D(),
    layers.GlobalAveragePooling2D(),
    keras.layers.Softmax()
])

```

```

Epoch 1/200
1407/1407 [=====] - ETA: 0s - loss: 1.6765 - accuracy: 0.4455
Epoch 1: val_loss improved from inf to 1.23908, saving model to
    ↵best_model.keras
1407/1407 [=====] - 16s 11ms/step - loss: 1.6764
accuracy: 0.4454 - val_loss: 1.2391 - val_accuracy: 0.6179

...
Epoch 40/200
1407/1407 [=====] - ETA: 0s - loss: 0.3431 - accuracy: 0.8950
Epoch 40: val_loss did not improve from 0.36904
1407/1407 [=====] - 15s 11ms/step - loss: 0.3432
accuracy: 0.8950 - val_loss: 0.3699 - val_accuracy: 0.8851
training duration = 630.068
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
Model: "sequential_2"

-----  

Layer (type)          Output Shape       Param #
=====  

conv2d_4 (Conv2D)     (None, 26, 26, 32)   320  

max_pooling2d_4 (MaxPooling 2D) (None, 13, 13, 32)   0  

conv2d_5 (Conv2D)     (None, 11, 11, 10)    2890  

max_pooling2d_5 (MaxPooling 2D) (None, 5, 5, 10)    0  

global_average_pooling2d_2 (GlobalAveragePooling2D) (None, 10)    0  

softmax_2 (Softmax)   (None, 10)        0  

=====  

Total params: 3,210
Trainable params: 3,210
Non-trainable params: 0

-----  

None
313/313 [=====] - 1s 2ms/step - loss: 0.3394 - accuracy: 0.9008
Test accuracy of the dropout model with lowest loss (the best model) = 0.901

```

Case 3: Global average pooling with a “regular” number of filters in the last convolutional layer and then a dense layer with the same number of neurons as the number of classes.

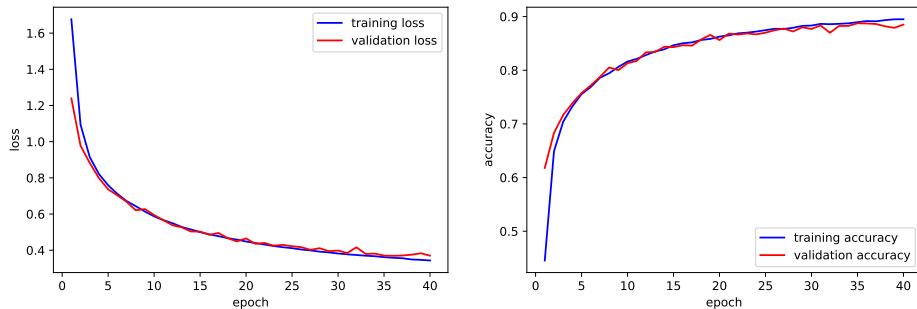


Fig. 14.9: The output of the code for Case 2: (left) loss; and (right) accuracy, as functions of epoch.

For this purpose, we replace the “building model” part in Case 1 with the following code snippet:

```
# building model
mnist_model = keras.Sequential([
    layers.Conv2D(16, 3, activation="relu",
    ↵input_shape = X_train[0].shape),
    layers.MaxPooling2D(),
    layers.Conv2D(64, 3, activation="relu"),
    layers.MaxPooling2D(),
    layers.GlobalAveragePooling2D(),
    layers.Dense(10, activation="softmax")
])
```

Epoch 1/200
1407/1407 [=====] - ETA: 0s - loss: 1.1278 -
 accuracy: 0.6446
Epoch 1: val_loss improved from inf to 0.63142, saving model to
 best_model.keras
1407/1407 [=====] - 13s 9ms/step - loss: 1.1265 -
accuracy: 0.6451 - val_loss: 0.6314 - val_accuracy: 0.8123
...
Epoch 26/200
1407/1407 [=====] - ETA: 0s - loss: 0.0924 -
 accuracy: 0.9721
Epoch 26: val_loss did not improve from 0.11053
1407/1407 [=====] - 16s 11ms/step - loss: 0.0924
accuracy: 0.9721 - val_loss: 0.1210 - val_accuracy: 0.9636
training duration = 350.196
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
Model: "sequential_3"

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 26, 26, 16)	160
max_pooling2d_6 (MaxPooling 2D)	(None, 13, 13, 16)	0
conv2d_7 (Conv2D)	(None, 11, 11, 64)	9280
max_pooling2d_7 (MaxPooling 2D)	(None, 5, 5, 64)	0
global_average_pooling2d_3 (GlobalAveragePooling2D)	(None, 64)	0
dense (Dense)	(None, 10)	650

Total params: 10,090
 Trainable params: 10,090
 Non-trainable params: 0

None
 313/313 [=====] - 1s 2ms/step - loss: 0.0775 -
 accuracy: 0.9767
 Test accuracy of the model with the lowest loss (the best model) = 0.977

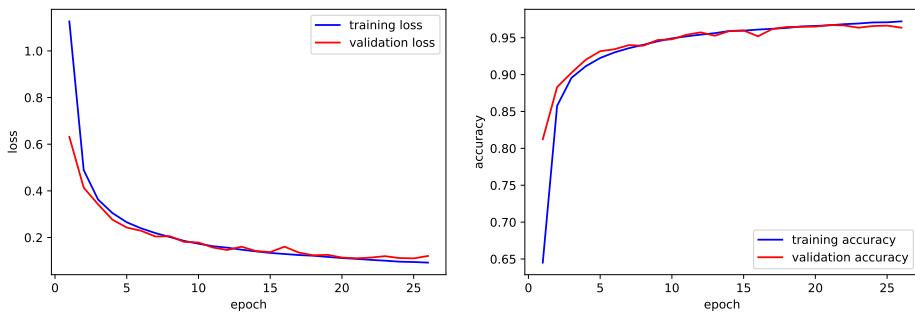


Fig. 14.10: Output of the code for Case 3: (left) loss; and (right) accuracy, as functions of epoch.

Exercises:

Exercise 1: Suppose in the MNIST application, we wish to perform hyperparameter tuning for the number of filters and layers in CNN. However, we would like to

choose an increasing pattern for the number of filters that depends on the depth of the convolutional base of the network. In particular, the number of filters to examine are:

1. for a CNN with one layer: 4 filters;
2. for a CNN with two layers: 4 and 8 filters in the first and second layers, respectively; and
3. for a CNN with three layers: 4, 8, and 16 filters in the first, second, and third layers, respectively;

In all cases: 1) the output of the (last) convolutional layer is flattened and used as the input to two fully connected layers with 32 neurons in the first layer; 2) there is a 2×2 max-pooling right after each convolutional layer and a dropout layer with a rate of 0.25 before each dense layer; and 3) all models are trained for 5 epochs.

- (A) write and execute a code that performs hyperparameter tuning using 3-fold stratified cross-validation with shuffling. For reproducibility purposes, set the seed for numpy, tensorflow, and Python built-in pseudo-random generators to 42. What is the highest CV score and the pattern achieving that score?
- (B) what is the estimated test accuracy of the selected (“best”) model trained on the full training data?
- (C) the following lines show parts of the code output:

```
Epoch 1/5  
1250/1250 [=====] - ...  
Epoch 2/5  
1250/1250 [=====] - ...  
Epoch 3/5  
1250/1250 [=====] - ...  
Epoch 4/5  
1250/1250 [=====] - ...  
Epoch 5/5  
1250/1250 [=====] - ...  
625/625 [=====] - ...
```

What do “1250” and “625” represent? Explain how these numbers are determined.

- (D) the following lines show parts of the code output:

```
Epoch 1/5  
1875/1875 [=====] - ...  
Epoch 2/5  
1875/1875 [=====] - ...  
Epoch 3/5  
1875/1875 [=====] - ...  
Epoch 4/5  
1875/1875 [=====] - ...  
Epoch 5/5
```

1875/1875 [=====] - ...
 313/313 [=====] - ...

What do “1875” and “313” represent?

Exercise 2: In a 2D convolutional layer, the input height, width, and depth are 10, 10, and 16, respectively. The kernel height, width, and the output depth (number of filters) are, 3, 3, and 32, respectively. How many parameters in total are learned in this layer?

- A) 51232 B) 51216 C) 4624 D) 4640

Exercise 3: In a 2D convolutional layer, the input height, width, and depth are 30, 30, and 32, respectively. We use a strided convolution (stride being 3 in all directions), no padding, and a kernel that has height, width, and the output depth of 5, 5, and 128, respectively. What is the shape of the output tensor?

- A) $9 \times 9 \times 128$ B) $10 \times 10 \times 128$ C) $9 \times 9 \times 32$ D) $10 \times 10 \times 32$

Exercise 4: Suppose in a regression problem with a univariate output, the input to a 2D convolutional layer is the following 4×4 (height) \times 4 (width) \times 1 (depth) tensor:

2	-1	-1	-1
-1	2	-1	-1
-1	-1	2	-1
-1	-1	-1	2

In the convolutional layer, we use a kernel that has height, width, and output depth of 3, 3, and 32, respectively, stride 1, and the ReLU activation function. This layer is followed by the global average pooling as the input to a fully connected layer to estimate the value of the output. Assuming all weights and biases used in the structure of the network are 1, what is the output for the given input (use hand calculations)?

- A) -17 B) -15 C) 0 D) 1 E) 15 F) 17

Exercise 5: Suppose in a regression problem with a univariate output, the input to a 1D convolutional layer is a time-series segment, denoted x_t , where $x_t = (-1)^t, t = 0, 1, \dots, 9$. In the convolutional layer, we use a kernel that has a size of 5 and the output depth of 16, stride is 2, and the ReLU activation function. This layer is followed by the global average pooling as the input to a fully connected layer to estimate the value of the output. Assuming all weights and biases used in the structure of the network are 1, what is the output for the given input (use hand calculations)?

- A) 1 B) 17 C) 33 D) 49 E) 65 F) 81

Exercise 6: Use Keras to build the models specified in Exercise 4 and 5 and compute the output for the given inputs.

Hints:

- Use the `kernel_initializer` and `bias_initializer` in the convolutional and dense layers to set the weights.
- Pick an appropriate initializer from ([keras-initializers, 2023](#)).
- The 1D convolutional layer that is needed for the model specified in Exercise 5 can be implemented by `keras.layers.Conv1D`. Similarly, the global average pooling for that model is `layers.GlobalAveragePooling1D`.
- To predict the output for the models specified in Exercise 4 and 5, the input should be 4D and 3D, respectively (the first dimension refers to “samples”, and here we have only one sample point as the input).



Chapter 15

Recurrent Neural Networks

The information flow in a classical ANN is from the input layer to hidden layers and then to the output layer. An ANN with such a flow of information is referred to as a feed-forward network. Such a network, however, has a major limitation in that it is not able to capture dependencies among sequential observations such as time-series. This is because the network has no memory and, as a result, observations are implicitly assumed to be independent.

Recurrent neural networks (RNNs) are fundamentally different from the feed-forward networks in that they operate not only based on input observations, but also on a set of internal states. The internal states capture the past information in sequences that have already been processed by the network. That is to say, the network includes a cycle that allows keeping past information for an amount of time that is not fixed *a-priori* and, instead, depends on its input observations and weights. Therefore, in contrast with other common architectures used in deep learning, RNN is capable of learning sequential dependencies extended over time. As a result, it has been extensively used for applications involving analyzing sequential data such as time-series, machine translation, and text and handwritten generation. In this chapter, we first cover fundamental concepts in RNNs including standard RNN, stacked RNN, long short-term memory (LSTM), gated recurrent unit (GRU), and vanishing and exploding gradient problems. We then present an application of RNNs for sentiment classification.

15.1 Standard RNN and Stacked RNN

We first formalize the dynamics of RNN. Let \mathbf{x}_t , \mathbf{h}_t , and \mathbf{y}_t denote the $p \times 1$, $l \times 1$, and $q \times 1$ input feature vector, hidden states, and output vector, respectively. Given an input sequence $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T)$, a standard RNN unit (also referred to as *vanilla* RNN), computes $(\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_T)$ and $(\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_T)$ by iterating through the following recursive equations for $t = 1, \dots, T$ (Graves et al., 2013):

$$\mathbf{h}_t = f_h(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h), \quad (15.1)$$

$$\mathbf{y}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y, \quad (15.2)$$

where \mathbf{W}_{xh} (input weight matrix), \mathbf{W}_{hh} (recurrent weight matrix), and \mathbf{W}_{hy} denote $l \times p$, $l \times l$, and $q \times l$ weight matrices, \mathbf{b}_h and \mathbf{b}_y denote bias terms, and $f_h(\cdot)$ denote an element-wise nonlinear hidden layer function (an activation function) such as logistic sigmoid or hyperbolic tangent. As in previous neural networks, depending on the application one may use an activation function in (15.2) to obtain \mathbf{y}_t ; for example, in a multiclass single label classification problem, $\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y$ is used as the input to a softmax to obtain \mathbf{y}_t . Fig. 15.1 depicts the working mechanism of the standard RNN unit characterized by the recursive utility of equations (15.1) and (15.2)—in Keras, this is implemented via `keras.layers.simpleRNN` class.

A basic RNN unit is already temporally deep because if it is unfolded in time, it becomes a composite of multiple nonlinear computational layers (Pascanu et al., 2014). Besides the depth of an RNN in a temporal sense, Pascanu *et al.* explored various other ways to define the concept of depth in RNN and to construct *deep* RNNs (Pascanu et al., 2014). A common way to define deep RNNs is to simply stack multiple recurrent hidden layers on top of each other (Pascanu et al., 2014). Nonetheless, various forms of stacking have been proposed. The standard stacked RNN is formed by using the hidden state of a layer as the input to the next layer. Assuming N hidden layers, the hidden states for layer j and the output of the network \mathbf{y}_t are iteratively computed for $j = 1, \dots, N$, and $t = 1, \dots, T$, as (Graves et al., 2013)

$$\mathbf{h}_t^j = f_h(\mathbf{W}_{h^{j-1}h^j}\mathbf{h}_t^{j-1} + \mathbf{W}_{h^jh^j}\mathbf{h}_{t-1}^j + \mathbf{b}_h^j), \quad (15.3)$$

$$\mathbf{y}_t = \mathbf{W}_{h^Ny}\mathbf{h}_t^N + \mathbf{b}_y, \quad (15.4)$$

where $\mathbf{h}_t^0 = \mathbf{x}_t$, and \mathbf{h}_t^j denotes the state vector for the j^{th} hidden layer, and where an identical hidden layer function $f_h(\cdot)$ is assumed across all layers. Fig. 15.2a depicts the working mechanism of the *stacked* RNN characterized by (15.3) and (15.4). The recurrence relation (15.3) was also used in (Hermans and Schrauwen, 2013) but the output is computed by all the hidden states. This form of stacked RNN is characterized by (Hermans and Schrauwen, 2013)

$$\mathbf{h}_t^j = f_h(\mathbf{W}_{h^{j-1}h^j}\mathbf{h}_t^{j-1} + \mathbf{W}_{h^jh^j}\mathbf{h}_{t-1}^j + \mathbf{b}_h^j), \quad (15.5)$$

$$\mathbf{y}_t = \sum_{j=1}^N \mathbf{W}_{h^jy}\mathbf{h}_t^j + \mathbf{b}_y, \quad (15.6)$$

where \mathbf{W}_{h^jy} are weight matrices learned to compute \mathbf{y}_t .

Another form of stacking is to determine the input of one layer by the hidden state of the previous layer as well as the input sequence. In this case, the hidden states for layer j and the output of the network \mathbf{y}_t are iteratively computed for $j = 1, \dots, N$, and $t = 1, \dots, T$, by (Graves, 2013) (see Fig. 15.2b)

$$\mathbf{h}_t^j = f_h \left(\mathbf{W}_{xh} \mathbf{x}_t + \mathbf{W}_{h^{j-1}h^j} \mathbf{h}_t^{j-1} + \mathbf{W}_{h^j h^j} \mathbf{h}_{t-1}^j + \mathbf{b}_h^j \right), \quad (15.7)$$

$$\mathbf{y}_t = \mathbf{W}_{h^N y} \mathbf{h}_t^N + \mathbf{b}_y, \quad (15.8)$$

where $\mathbf{h}_t^0 = \mathbf{0}$.

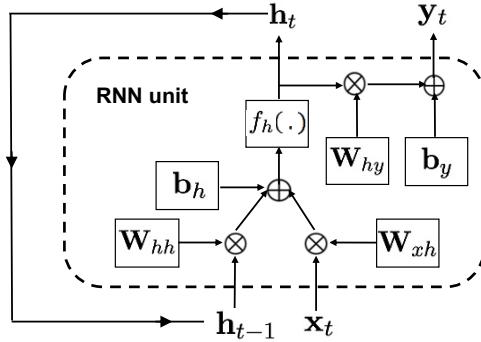


Fig. 15.1: A schematic diagram for the working mechanism of the standard RNN unit.

The models presented so far are known as *unidirectional* RNNs because to predict \mathbf{y}_t they only consider information from the present and the past context; that is to say, the flow of information is from past to present and future observations are not used to predict \mathbf{y}_t . Schuster *et al.* (Schuster and Paliwal, 1997) proposed the *bidirectional* RNN that can utilize information from both the past and the future in predicting \mathbf{y}_t . In this regard, two sets of hidden layers are defined: 1) *forward hidden states*, denoted \mathbf{h}_t^f , which similar to the standard RNN, are computed iteratively from $t = 1$ to T (i.e., using sequence $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T)$); and 2) *backward hidden states*, denoted \mathbf{h}_t^b , which are computed iteratively from $t = T$ to 1 (i.e., using sequence $(\mathbf{x}_T, \mathbf{x}_{T-1}, \dots, \mathbf{x}_1)$). Once hidden states are computed, the output sequence is obtained by a linear combination of forward and backward states. In particular, we have

$$\mathbf{h}_t^f = f_h \left(\mathbf{W}_{xh}^f \mathbf{x}_t + \mathbf{W}_{hh}^f \mathbf{h}_{t-1}^f + \mathbf{b}_h^f \right), \quad (15.9)$$

$$\mathbf{h}_t^b = f_h \left(\mathbf{W}_{xh}^b \mathbf{x}_t + \mathbf{W}_{hh}^b \mathbf{h}_{t+1}^b + \mathbf{b}_h^b \right), \quad (15.10)$$

$$\mathbf{y}_t = \mathbf{W}_{hy}^f \mathbf{h}_t^f + \mathbf{W}_{hy}^b \mathbf{h}_t^b + \mathbf{b}_y, \quad (15.11)$$

where \mathbf{W}_{xh}^f , \mathbf{W}_{hh}^f , \mathbf{W}_{xh}^b , \mathbf{W}_{hh}^b , \mathbf{W}_{hy}^f , \mathbf{W}_{hy}^b , \mathbf{b}_h^f , \mathbf{b}_h^b , and \mathbf{b}_y are weight matrices and bias terms of compatible size.

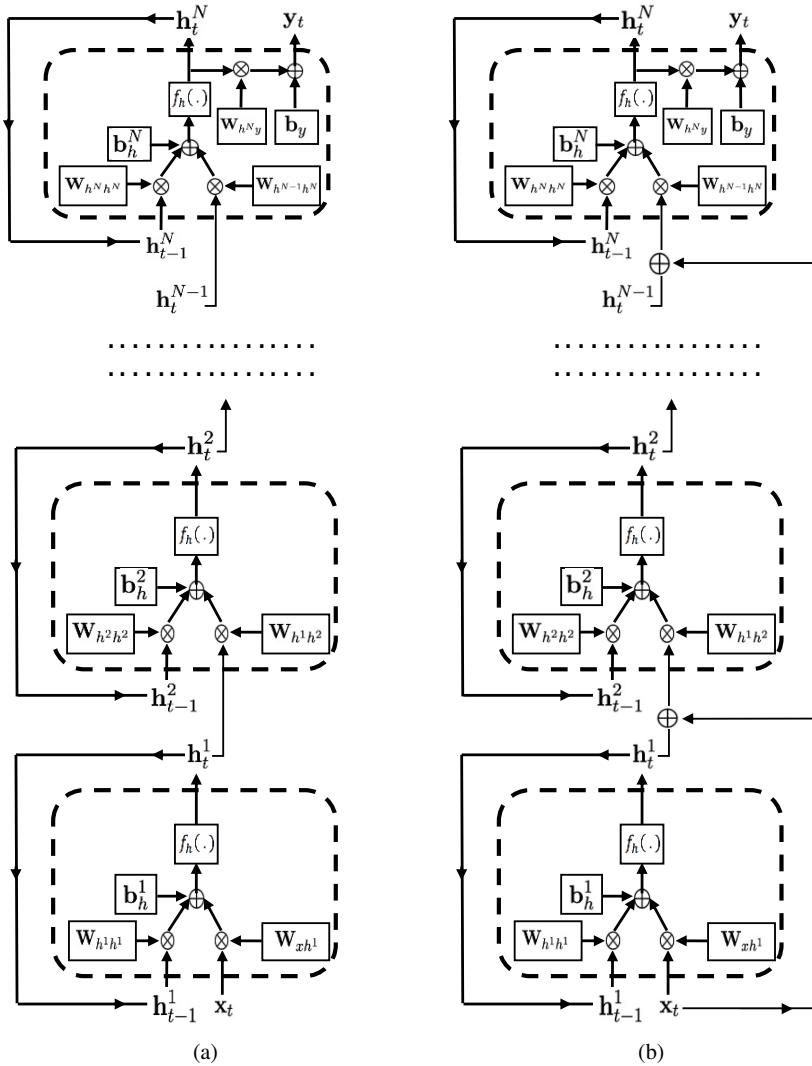


Fig. 15.2: A schematic diagram for the working mechanism of stacked RNN proposed in (Graves et al., 2013) (a), and (Graves, 2013) (b).

15.2 \oplus Vanishing and Exploding Gradient Problems

Despite the powerful working principles of standard RNN, it does not exhibit the capability of capturing long-term dependencies when trained in practice. This state of affairs is generally attributed to the vanishing gradient and the exploding gradient problems described in (Bengio et al., 1994; Hochreiter and Schmidhuber, 1997).

Vanishing gradient problem makes learning long-term dependencies difficult (because the influence of short-term dependencies dominate (Bengio et al., 1994)), whereas exploding gradient induces instability of the learning process. The root of the problem is discussed in details elsewhere (Bengio et al., 1994; Pascanu et al., 2013). Here we provide a simplistic view that helps shed light on the cause of the problem. Consider the following recurrent relations that characterize an RNN with univariate input, state, and $b_h = 0$:

$$h_t = f_h(w_{xh}x_t + w_{hh}h_{t-1}), \quad (15.12)$$

$$y_t = w_{hy}h_t + b_y. \quad (15.13)$$

To compute, say, y_5 , we need to compute h_5 . Assuming $h_0 = 0$, we have

$$\begin{aligned} h_5 &= f_h(w_{xh}x_5 + w_{hh}h_4), \\ h_4 &= f_h(w_{xh}x_4 + w_{hh}h_3), \\ h_3 &= f_h(w_{xh}x_3 + w_{hh}h_2), \\ h_2 &= f_h(w_{xh}x_2 + w_{hh}h_1), \\ h_1 &= f_h(w_{xh}x_1). \end{aligned} \quad (15.14)$$

or, equivalently,

$$h_5 = f_h\left(w_{xh}x_5 + w_{hh}f_h\left(w_{xh}x_4 + w_{hh}f_h\left(w_{xh}x_3 + w_{hh}f_h\left(w_{xh}x_2 + w_{hh}f_h(w_{xh}x_1)\right)\right)\right)\right). \quad (15.15)$$

As described in Section 13.2, in order to find the weights of an ANN, we need to find the gradient of the loss function with respect to weights (i.e., steepest descent direction). The loss function L in RNN is computed over the entire sequence length and the underlying algorithm for computing the gradients for RNN is called *backpropagation through time* (BPTT) (Werbos, 1990). In particular,

$$\frac{\partial L}{\partial w_{hh}} = \sum_{t=1}^T \frac{\partial L_t}{\partial w_{hh}}, \quad (15.16)$$

where L_t (the loss at a specific t) depends on y_t . Let us assume a specific time t_0 and consider the term $\frac{\partial L_{t_0}}{\partial w_{hh}}$. Because L_{t_0} depends on y_{t_0} , the chain rule of calculus yields

$$\frac{\partial L_{t_0}}{\partial w_{hh}} = \frac{\partial L_{t_0}}{\partial y_{t_0}} \frac{\partial y_{t_0}}{\partial h_{t_0}} \frac{\partial h_{t_0}}{\partial w_{hh}}. \quad (15.17)$$

Consider the term $\frac{\partial h_{t_0}}{\partial w_{hh}}$ in (15.17). From (15.12), using the regular chain rule once more, we have

$$\frac{\partial h_{j+1}}{\partial w_{hh}} = \left(h_j + w_{hh} \frac{\partial h_j}{\partial w_{hh}} \right) f'_j, \quad j = 1, \dots, t_0 - 1, \quad (15.18)$$

$$\frac{\partial h_1}{\partial w_{hh}} = 0, \quad (15.19)$$

where for the ease of notation we denoted derivative of a function g with respect to its input as g' and denoted $f'_h(w_{xh}x_{j+1} + w_{hh}h_j)$ by f'_j . Recursively computing (15.18) starting from (15.19) for $j = 1, \dots, t_0 - 1$, yields¹

$$\frac{\partial h_{t_0}}{\partial w_{hh}} = \sum_{k=1}^{t_0-1} w_{hh}^{t_0-1-k} h_k \prod_{j=k}^{t_0-1} f'_j \quad (15.20)$$

For instance,

$$\frac{\partial h_5}{\partial w_{hh}} = h_4 f'_4 + w_{hh} h_3 f'_4 f'_3 + w_{hh}^2 h_2 f'_4 f'_3 f'_2 + w_{hh}^3 h_1 f'_4 f'_3 f'_2 f'_1. \quad (15.21)$$

To analyze the effect of repetitive multiplications by w_{hh} , we assume a linear activation function $f_h(u) = u$. This could be perceived simply as an assumption (e.g., see (Pascanu et al., 2013)) or, alternatively, the first (scaled) term of Maclaurin series for some common activation functions (and therefore, an approximation around 0). Therefore, $f'_j = 1, j = 1, \dots, t_0 - 1$. Using this assumption to recursively compute h_j from (15.12) yields

$$h_j = w_{xh} \sum_{k=1}^j w_{hh}^{j-k} x_k, \quad j = 1, \dots, t_0 - 1. \quad (15.22)$$

Replacing (15.22) in (15.20) with some straightforward algebraic simplification yield (recall that $f'_j = 1$)

$$\frac{\partial h_{t_0}}{\partial w_{hh}} = w_{xh} \sum_{k=1}^{t_0-1} (t_0 - k) w_{hh}^{t_0-1-k} x_k. \quad (15.23)$$

Expression (15.23) shows the dominance of short-term dependencies with respect to long-term dependencies in the gradient. In particular, to capture the effect of a “far” input (e.g., x_1) on minimizing L_{t_0} (which affects moving w_{hh} during the gradient decent), we have a multiplication by a high power of w_{hh} (e.g., $w_{hh}^{t_0-2}$ for x_1). In contrast capturing the effect of “near” inputs (e.g. x_{t_0-1}) requires a multiplication by a low power of w_{hh} (e.g., w_{hh}^0 for x_{t_0-1}). In other words, capturing the effect of long terms dependencies requires multiplying w_{hh} by itself many times. Therefore,

¹ Here to write the expression (15.20) we have used the regular chain rule of calculus, which is well-known. We can also achieve at this relationship using the concept of *immediate partial derivatives*, which is introduced by Werbos (Werbos, 1990) and used, for example, in (Pascanu et al., 2013) to treat the vanishing and exploding gradient phenomena.

if $w_{hh} < 1$ (or if $w_{hh} > 1$), high powers of w_{hh} will vanish (or will explode), which causes the vanishing (or exploding) gradient problem.

15.3 LSTM and GRU

The problem of gradient exploding associated with the standard RNN can be resolved by the *gradient clipping*. For example, clipping the norm of the gradient when it goes over a threshold (e.g., the average norm over a number of updates) (Pascanu et al., 2013). However, to alleviate the vanishing gradient problem, more sophisticated recurrent network architectures namely, *long short-term memory* (LSTM) units (Hochreiter and Schmidhuber, 1997), and a more recent one, *gated recurrent units* (GRUs) (Cho et al., 2014), have been proposed. In both these networks, the mapping between pair of \mathbf{x}_t and \mathbf{h}_{t-1} to \mathbf{h}_t , which in standard RNN was based on a single activation function, is replaced by a more sophisticated activation function. In particular, LSTM implements the activation function $f_h(\cdot)$ in (15.1) by the following composite function:²

$$\mathbf{i}_t = \sigma_{\text{logistic}}(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i), \quad (15.24)$$

$$\mathbf{f}_t = \sigma_{\text{logistic}}(\mathbf{W}_{xf}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f), \quad (15.25)$$

$$\tilde{\mathbf{c}}_t = \sigma_{\tanh}(\mathbf{W}_{xc}\mathbf{x}_t + \mathbf{W}_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c), \quad (15.26)$$

$$\mathbf{c}_t = \mathbf{i}_t \odot \tilde{\mathbf{c}}_t + \mathbf{f}_t \odot \mathbf{c}_{t-1}, \quad (15.27)$$

$$\mathbf{o}_t = \sigma_{\text{logistic}}(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o), \quad (15.28)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \sigma_{\tanh}(\mathbf{c}_t), \quad (15.29)$$

where \odot denotes element-wise multiplication, \mathbf{W}_{xi} , \mathbf{W}_{xf} , \mathbf{W}_{xc} , and \mathbf{W}_{xo} denote $l \times p$ weight matrices, \mathbf{W}_{hi} , \mathbf{W}_{hf} , \mathbf{W}_{hc} , and \mathbf{W}_{ho} denote $l \times l$ weight matrices, \mathbf{b}_i , \mathbf{b}_f , \mathbf{b}_c , and \mathbf{b}_o denote $l \times 1$ bias terms, and \mathbf{i}_t , \mathbf{f}_t , \mathbf{c}_t , and \mathbf{o}_t are all some $l \times 1$ vectors known as the states of *input gate*, *forget gate*, *memory cell*, and *output gate*, respectively. Given \mathbf{h}_t from (15.29), the output sequence can be computed as presented earlier in (15.2). Therefore, based on (15.24)-(15.29), there are $4(lp + l^2 + l)$ parameters learned in one LSTM layer.

The central concept in LSTM is the use of “memory” cell and “gates” that control what should be kept in the memory at each time. Based on (15.27), the state of the memory cell at time t is obtained by partially forgetting the memory content \mathbf{c}_{t-1} (determined by the state of the “forget” gate at t , which is \mathbf{f}_t), and partially adding a new memory content $\tilde{\mathbf{c}}_t$ (determined by the state of the “input” gate at t , which is \mathbf{i}_t). Consider a single “hidden unit” with input and forget gate states being $i_t = 0$ and $f_t = 0$, respectively (by each hidden unit we refer to one dimension of the composite function (15.24)-(15.29)). Consider the following four cases that

² Comparing to the original form of LSTM (Hochreiter and Schmidhuber, 1997), this version is due to (Gers et al., 1999) and has an additional gate (the forget gate).

(approximately) occur depending on the limits of $\sigma_{\text{logistic}}(\cdot)$, which is used in both (15.24) and (15.25):

- $i_t = 0$ and $f_t = 0$: $c_t = 0 \rightarrow$ reset the memory content to 0;
- $i_t = 1$ and $f_t = 0$: $c_t = \tilde{c}_t \rightarrow$ overwrite the memory content by the new memory;
- $i_t = 0$ and $f_t = 1$: $c_t = c_{t-1} \rightarrow$ keep the memory content;
- $i_t = 1$ and $f_t = 1$: $c_t = \tilde{c}_t + c_{t-1} \rightarrow$ add the new memory to the previous memory content;

Although these cases are valid in the limits, we also note that $\sigma_{\text{logistic}}(\cdot)$ converges exponentially to 0 or 1 as its input argument decreases or increases, respectively. Last but not least, the output gate controls the amount of memory that should be exposed at each time. As we can see, not only LSTM can propagate information by its additive memory, it also *learns* when to forget or keep the memory, which is a powerful feature of “gated RNNs”. In Keras, LSTM is implemented via `keras.layers.LSTM` class.

GRU, on the other hand, is a simpler network architecture than LSTM, which makes it more efficient to train. At the same time, it has shown comparable performances to LSTM (Chung et al., 2014), which along with its simplicity makes it an attractive choice. Similar to LSTM, GRU has also gating units that controls the flow of information. In GRU, the activation function $f_h(\cdot)$ in (15.1) is implemented by following composite function:

$$\mathbf{z}_t = \sigma_{\text{logistic}}(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_z), \quad (15.30)$$

$$\mathbf{r}_t = \sigma_{\text{logistic}}(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_r), \quad (15.31)$$

$$\tilde{\mathbf{h}}_t = \sigma_{\tanh}(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_h(\mathbf{h}_{t-1} \odot \mathbf{r}_t) + \mathbf{b}_h), \quad (15.32)$$

$$\mathbf{h}_t = \mathbf{z}_t \odot \tilde{\mathbf{h}}_t + (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1}, \quad (15.33)$$

where \mathbf{W}_{xz} , \mathbf{W}_{xr} , \mathbf{W}_{xh} , and \mathbf{W}_h denote $l \times p$, $l \times p$, $l \times p$, and $l \times l$ weight matrices, respectively, \mathbf{b}_z , \mathbf{b}_r , \mathbf{b}_h denote $l \times 1$ bias terms, and \mathbf{z}_t and \mathbf{r}_t denote $l \times 1$ state vector of the *update gate* and the *reset gate*, respectively. In contrast with LSTM, GRU does not use the concept of “memory cell”. The hidden state is used directly to learn keeping or forgetting past information. Consider a single hidden unit. When the state of the update gate is close to 0, the hidden state content is kept; that is, we have $h_t = h_{t-1}$ in the corresponding unit. However, when the state of the update gate is close to 1, the amount of the past hidden state that is kept is determined by the reset gate via (15.32) and is stored in the candidate hidden state \tilde{h}_t , which is used to set h_t via (15.33). In that case, if the reset gate is 0, the hidden state is essentially set by the immediate input. Similar to standard RNN units, LSTM and GRU can be stacked on top of each other to construct a deeper network. In that case and similar to (15.3)-(15.8), expressions (15.24)-(15.29) and (15.30)-(15.33) for LSTM and GRU, respectively, can be extended to all layers depending on the specific form of stacking. Last but not least, the number of parameter learned in Keras for a GRU layer (via `keras.layers.GRU`) is slightly different from the number of parameters in (15.30)-(15.32). This is because in Keras, two separate bias vectors of size $l \times 1$

are learned in (15.30)-(15.32) ([keras-gru, 2023](#)). This makes the total number of parameters learned in a GRU layer $3(lp + l^2 + 2l)$.

15.4 Implementation in Keras: Sentiment Classification

In this application, we intend to use text data to train an RNN that can classify a movie review represented as a string of words to a “positive review” or “negative review”—this is an example of sentiment classification. In this regard, we use the “IMDB dataset”. This is a dataset, which contains 25000 positive and the same number of negative movie reviews collected from the IMDB website ([Maas et al., 2011](#)). Half of the entire dataset is designated for training and the other half for testing. Although the (encoded) dataset comes bundled with the Keras library and could be easily loaded by: `from tensorflow.keras.datasets import imdb` and then `imdb.load_data()`, we download it in its raw form from ([IMDB, 2023](#)) to illustrate some preprocessing steps that would be useful in the future when working raw text data.

```
import time
import numpy as np
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
from tensorflow.keras import layers
#from tensorflow.keras.datasets import imdb
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical
seed_value= 42

# set the seed for Python built-in pseudo-random generator
import random
random.seed(seed_value)

# set the seed for numpy pseudo-random generator
import numpy as np
np.random.seed(seed_value)

# set the seed for tensorflow pseudo-random generator
tf.random.set_seed(seed_value)
```

Once the dataset is downloaded, each of the “aclImdb/train” and “aclImdb/test” folders contains two folders, namely “pos” and “neg”, which include the positive and negative reviewers, respectively. Here we first load the downloaded dataset using `keras.utils.text_dataset_from_directory`. This method returns a Tensor-

Flow Dataset object. We then use its `concatenate` method to add the negative reviews to positive reviews.

```
# load the training (+ validation) data for positive and negative reviews, concatenate them, and generate the labels
directory = ".../aclImdb/train/pos" # replace ... by the path to "aclImdb", which contains the downloaded data
X_train_val_pos = keras.utils.text_dataset_from_directory(directory=directory, batch_size = None, label_mode=None, shuffle=False)
directory = ".../aclImdb/train/neg"
X_train_val_neg = keras.utils.text_dataset_from_directory(directory=directory, batch_size = None, label_mode=None, shuffle=False)
X_train_val = X_train_val_pos.concatenate(X_train_val_neg)
y_train_val = np.array([0]*len(X_train_val_pos) + [1]*len(X_train_val_neg))
```

Found 12500 files belonging to 1 classes.

Found 12500 files belonging to 1 classes.

```
# load the test data for positive and negative reviews, concatenate them, and generate the labels
directory = ".../aclImdb/test/pos"
X_test_pos = keras.utils.text_dataset_from_directory(directory=directory, batch_size = None, label_mode=None, shuffle=False)
directory = ".../aclImdb/test/neg"
X_test_neg = keras.utils.text_dataset_from_directory(directory=directory, batch_size = None, label_mode=None, shuffle=False)
X_test = X_test_pos.concatenate(X_test_neg)
y_test = np.array([0]*len(X_test_pos) + [1]*len(X_test_neg))
```

Found 12500 files belonging to 1 classes.

Found 12500 files belonging to 1 classes.

Next we use `TextVectorization`, which is a versatile and efficient method that can be used for both: 1) *text standardization*, which in its most standard form means replacing punctuations with empty strings and converting all letters to lower case; and 2) *text tokenization*, which in this regard, by default it splits the input text based on whitespace (i.e., identifies words as tokens). The use of `output_mode='int'` in `TextVectorization` assigns one integer to each identified word in the vocabulary. We choose a vocabulary of size 5000 words, which means the most (5000-2) frequent words in `X_train_val` (to which the `TextVectorization` is “adapted”) will be part of the vocabulary; therefore, we will not have any integer greater than 5000 to represent a word. The first two integers, 0 and 1, will be assigned to “no word” and “unknown word” [UNK], respectively.

```
# use TextVectorization to learn (by "adapt()" method) the vocabulary
from tensorflow.keras.layers import TextVectorization as TV
vocab_size = 5000
tv = TV(max_tokens=vocab_size, output_mode='int')
tv.adapt(X_train_val)
print("Total number of words in the dictionary: ", len(tv.
    get_vocabulary()))
```

Total number of words in the dictionary: 5000

Next we use the learned vocabulary to transform each sequence of words that are part of `X_train_val` into corresponding sequence of integers.

```
# encode each sequence ("seq") that is part of the X_train_val Dataset
# object using the learned vocabulary
X_train_val_int_encoded_var_len = []
for seq in X_train_val:
    seq_encoded = tv([seq]).numpy()[0]
    X_train_val_int_encoded_var_len.append(seq_encoded)
X_train_val_int_encoded_var_len[0]

array([ 1, 322, 7, 4, 1078, 220, 9, 2085, 31, 2, 167,
       62, 15, 47, 81, 1, 43, 400, 119, 136, 15, 4894,
      56, 1, 148, 8, 2, 4946, 1, 480, 70, 6, 256,
      12, 1, 1, 1972, 7, 73, 2363, 6, 641, 71, 7,
     4894, 2, 1, 6, 2031, 1, 2, 1, 1422, 37, 69,
      68, 205, 141, 65, 1216, 4894, 1, 2, 1, 5, 2,
     219, 904, 32, 2929, 70, 5, 2, 4711, 10, 672, 3,
      65, 1422, 51, 10, 208, 2, 383, 8, 60, 4, 1474,
    3622, 775, 6, 3580, 187, 2, 400, 10, 1192, 1, 31,
      322, 4, 350, 363, 2971, 142, 132, 6, 1, 29, 5,
     123, 4894, 1474, 2409, 6, 1, 322, 10, 517, 12, 106,
    1471, 5, 56, 582, 103, 12, 1, 322, 7, 234, 1,
      49, 4, 2292, 12, 9, 207])
```

If desired, we can convert back the sequence of integers into words.

```
# decode an encoded sequence back to words
inverse_vocabulary = dict(enumerate(tv.get_vocabulary()))
decoded_sentence = " ".join(inverse_vocabulary[i] for i in
    X_train_val_int_encoded_var_len[0])
decoded_sentence
```

'[UNK] high is a cartoon comedy it ran at the same time as some other.
 ↵[UNK] about school life such as teachers my [UNK] years in the
 ↵teaching [UNK] lead me to believe that [UNK] [UNK] satire is much
 ↵closer to reality than is teachers the [UNK] to survive [UNK] the
 ↵[UNK] students who can see right through their pathetic teachers
 ↵[UNK] the [UNK] of the whole situation all remind me of the schools i
 ↵knew and their students when i saw the episode in which a student
 ↵repeatedly tried to burn down the school i immediately [UNK] at high
 ↵a classic line inspector im here to [UNK] one of your teachers
 ↵student welcome to [UNK] high i expect that many adults of my age
 ↵think that [UNK] high is far [UNK] what a pity that it isn't'

```
# encode each sequence ("seq") that is part of the X_test Dataset
↪object using the learned vocabulary
```

```
X_test_int_encoded_var_len = []
for seq in X_test:
    seq_encoded = tv([seq]).numpy()[0]
    X_test_int_encoded_var_len.append(seq_encoded)
X_test_int_encoded_var_len[0]
```

```
array([ 10,  418,     3,  208,    11,    18,   226,   311,   101,   107,
       1,    6,    33,     4,  164,   340,     5,  1946,   527,   934,    12,
      10,   14,    1,     6,   68,     9,   80,    36,    49,    10,   672,
       5,   1,    1,   27,   14,   61,   491,     6,   82,   220,   10,
     14,  359,    1,  251,     2,  109,     5,  3216,     1,    53,   74,
       3,  1785,    1,  251,  1001,     1,    17,   136,     1,     2,
  1890,     5,   4,    50,   18,     7,   12,     9,    69,  3006,    17,
  254,  1400,  11,   29,   117,   593,    12,     2,   417,   740,
  60,   14,  2940,  46,   14,  3000,   33,  2162,   299,     2,
   86,   357,     5,     2,   18,    3,   66,  1614,     6,
  1670,   299,     2,   326,   357,   131,   1,    2,   740,
   10,   22,    61,   208,   106,   362,     8,  1670,
  19,   106,   371,  2269,   346,   15,    74,   258,  2660,
   22,     6,  372,   253,   68,   98,  2570,   11,   18,   14,
   85,     3,   10,  1405,   12,   23,   138,   68,     9,   156,
   23,  1856])
```

Until now the encoded sequence of reviews have different length. However, for training, we need sequences of the same length. Here we choose the first 200 words (that are of course part of the learned vocabulary) of each sequence. In this regard, we pad the sequences. Those that are shorter will be (by default) pre-padded by zeros to become 200 long.

```
# unify the length of each encoded review to a sequence_len. If the
↪review is longer, it is cropped and if it is shorter it will padded
↪by zeros (by default)
sequence_len = 200

X_train_val_padded_fixed_len = keras.utils.
↪pad_sequences(X_train_val_int_encoded_var_len, maxlen=sequence_len)
X_test_padded_fixed_len = keras.utils.
↪pad_sequences(X_test_int_encoded_var_len, maxlen=sequence_len)
X_train_val_padded_fixed_len[0:2]
```

```
array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0,  0,  0,  0,  0,  0,  0,  0,  0,  1,  322,  7,  4,
       1078,  220,   9, 2085,   31,   2,  167,   62,   15,  47,  81,
        1,   43,  400,  119,  136,   15, 4894,   56,   1, 148,   8,
        2, 4946,   1,  480,   70,   6,  256,   12,   1,   1, 1972,
        7,   73, 2363,   6,  641,   71,   7, 4894,   2,   1,   6,
      2031,   1,   2,   1, 1422,   37,   69,   68,  205, 141,  65,
      1216, 4894,   1,   2,   1,   5,   2,  219,  904,  32, 2929,
        70,   5,   2, 4711,   10,  672,   3,   65, 1422,  51,  10,
      208,   2,  383,   8,   60,   4, 1474, 3622,  775,   6, 3580,
      187,   2,  400,   10, 1192,   1,   31,  322,   4,  350,  363,
      2971,  142, 132,   6,   1,   29,   5, 123, 4894, 1474, 2409,
        6,   1,  322,   10,  517,   12, 106, 1471,   5,  56,  582,
      103,   12,   1,  322,   7,  234,   1,   49,   4, 2292,  12,
        9, 207],
      [ 1, 228, 335,   2,   1,   1,   33,   4,   1, 101,  30,
      420,  21,  25,   1, 113,   1,  879,   81, 102, 571,   4,
      246,  33,   2, 398,   5, 4676,   1, 2039, 3841,  34,   1,
        37, 183, 4415, 156, 2236,   40,  345,   3,  40,   1,   1,
      2151, 4635,   3,   1,   1, 2587,   37,   24,  449, 334,   6,
        2, 1889, 493, 4164,   1, 207,  228,   22,  334,   6, 4463,
        1,   1,  39,   27, 272, 117,   51, 107, 1004, 113,  30,
      537,  42, 2805, 502,  42,   28,   1,  13, 131,   2, 116,
      1961, 193, 4676,   3,   1, 268, 1659,   6, 114, 10, 249,
        119, 4495,   6,   28,   29,   5, 3691, 2834,   1,  95, 113,
      2545,   6, 107,   4, 220,   9,  265,   4, 4244, 506, 1054,
        6,   25, 2602, 161, 136,   15,   1,   1, 182,   1,  42,
        1, 16,   2, 549,   6, 120,   49,  30,   39, 252, 140,
      4428, 156, 2236,   9,   2, 367,  262,   42,   21,   2,  81,
      545, 245,   4, 375, 2081,   39,   32, 1004,   83,  82,  51,
        35,  90, 118,   49,   6,  82,   17,   65, 276, 270,  35,
      139, 192,   9,   6,   2, 3230, 297,   5, 747,   9,  39,
        1,   1, 13,  42, 270, 11,  20,  77,   1,  23,   6,
      328, 377]], dtype=int32)
```

Now we are in the position that we can start building and training the model. However, for applications related to natural language processing, there is an important layers that can significantly boost the performance and, at the same time, speed up the training. The core principle behind using this layer, which is known as *word embedding layer*, is to represent each word by a relatively low-dimensional vector, which is learned during the training process. The idea of representing a word by a vector is not really new. One-hot encoding was an example of that. Suppose we have a vocabulary of 15 words. We are given the following sequence of four words and all these words belong to our vocabulary: “hi how are you”. One way to use one-hot encoding to represent the sequence is similar to Fig. 15.3. However, using this approach we lose information about the order of words within the sequence. As a result, this is not suitable for models such as RNN and CNN that

are used with sequential data. Another way to keep the order of words using one-hot encoding is depicted in Fig. 15.4. However, this approach leads to considerably sparse high-dimensional matrices. In word embedding, which is implemented by `keras.layers.Embedding`, we represent each word with a low-dimensional vector (the dimension is a hyperparameter) where each element of this vector is estimated through the learning algorithm (see Fig. 15.5). Given a batch of word sequences of size “samples × sequence length”, the embedding “layer” in Keras returns a tensor of size “samples × sequence length × embedding dimensionality”. Next, we use an embedding 16-dimensional layer, a GRU with 32 hidden units, and early stopping call back with a patience parameter of 3 to train our classifier. The classifier shows an accuracy of 86.4% on the test set.

	we	why	when	what	hey	were	hi	are	bye	how	they	you	is	not	yes
seq	0	0	0	0	0	0	1	1	0	1	0	1	0	0	0

Fig. 15.3: The use of one-hot encoding without preserving the order of words within sequence identified by “seq”: hi how are you.

	we	why	when	what	hey	were	hi	are	bye	how	they	you	is	not	yes
hi	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
how	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
are	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
you	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

Fig. 15.4: The use of one-hot encoding to preserve the order of words within sequence.

	dim1	dim2	dim3	dim4	dim5
hi	0.2	-0.1	-0.3	-0.4	0.1
how	0.1	0.2	0.1	-0.2	0.2
are	-0.3	0.1	-0.3	-0.1	0.2
you	-0.2	0.15	-0.1	-0.2	0.1

Fig. 15.5: An embedding of 5 dimension (in practice the dimension is usually set between 8 to 1024).

```
# split the training+validation data into training set and validation set
X_train, X_val, y_train, y_val = train_test_split(X_train_val_padded_fixed_len, y_train_val,
stratify=y_train_val, test_size=0.25)

# build the model
imdb_model = keras.Sequential([
    layers.Embedding(vocab_size, 16),
    layers.GRU(32, input_shape=(None, 1)),
    layers.Dense(1, activation='sigmoid')
])

# compiling model
imdb_model.compile(optimizer="adam", loss="binary_crossentropy",
metrics=["accuracy"])

# training model
my_callbacks = [
    keras.callbacks.EarlyStopping(
        monitor="val_accuracy",
        patience=3),
    keras.callbacks.ModelCheckpoint(
        filepath="best_model.keras",
        monitor="val_accuracy",
        save_best_only=True,
        verbose=1)
]

history = imdb_model.fit(x = X_train,
    y = y_train,
    batch_size = 32,
    epochs = 100,
    validation_data = (X_val, y_val),
    callbacks=my_callbacks)
```

```
Epoch 1/100
586/586 [=====] - ETA: 0s - loss: 0.4868 - accuracy: 0.7461
Epoch 1: val_accuracy improved from -inf to 0.83408, saving model to best_model.keras
586/586 [=====] - 32s 52ms/step - loss: 0.4868 - accuracy: 0.7461 - val_loss: 0.3796 - val_accuracy: 0.8341
Epoch 2/100
586/586 [=====] - ETA: 0s - loss: 0.2927 - accuracy: 0.8811
Epoch 2: val_accuracy improved from 0.83408 to 0.86656, saving model to
```

```

best_model.keras
586/586 [=====] - 30s 50ms/step - loss: 0.2927 -
accuracy: 0.8811 - val_loss: 0.3194 - val_accuracy: 0.8666
Epoch 3/100
586/586 [=====] - ETA: 0s - loss: 0.2374 - accuracy:
0.9073
Epoch 3: val_accuracy did not improve from 0.86656
586/586 [=====] - 30s 50ms/step - loss: 0.2374 -
accuracy: 0.9073 - val_loss: 0.3407 - val_accuracy: 0.8578
Epoch 4/100
586/586 [=====] - ETA: 0s - loss: 0.2070 - accuracy:
0.9202
Epoch 4: val_accuracy did not improve from 0.86656
586/586 [=====] - 31s 53ms/step - loss: 0.2070 -
accuracy: 0.9202 - val_loss: 0.3395 - val_accuracy: 0.8558
Epoch 5/100
586/586 [=====] - ETA: 0s - loss: 0.1751 - accuracy:
0.9348
Epoch 5: val_accuracy did not improve from 0.86656
586/586 [=====] - 32s 55ms/step - loss: 0.1751 -
accuracy: 0.9348 - val_loss: 0.3844 - val_accuracy: 0.8664

```

```

print(history.history.keys())
print(imdb_model.summary())

# plotting the results
epoch_count = range(1, len(history.history['loss']) + 1)
plt.figure(figsize=(13,4), dpi=150)
plt.subplot(121)
plt.plot(epoch_count, history.history['loss'], 'b', label = 'training loss')
plt.plot(epoch_count, history.history['val_loss'], 'r', label = 'validation loss')
plt.legend()
plt.ylabel('loss')
plt.xlabel('epoch')
plt.subplot(122)
plt.plot(epoch_count, history.history['accuracy'], 'b', label = 'training accuracy')
plt.plot(epoch_count, history.history['val_accuracy'], 'r', label = 'validation accuracy')
plt.legend()
plt.ylabel('accuracy')
plt.xlabel('epoch')

best_imdb_model = keras.models.load_model("best_model.keras")

```

```

loss, accuracy = best_imdb_model.evaluate(X_test_padded_fixed_len, y_test, verbose=1)
print('Test accuracy = {:.3f}'.format(accuracy))

```

```

dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
Model: "sequential_2"

```

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, None, 16)	80000
gru_1 (GRU)	(None, 32)	4800
dense_2 (Dense)	(None, 1)	33

```

Total params: 84,833
Trainable params: 84,833
Non-trainable params: 0

```

```

None
782/782 [=====] - 8s 9ms/step - loss: 0.3210 -
accuracy: 0.8637
Test accuracy = 0.864

```

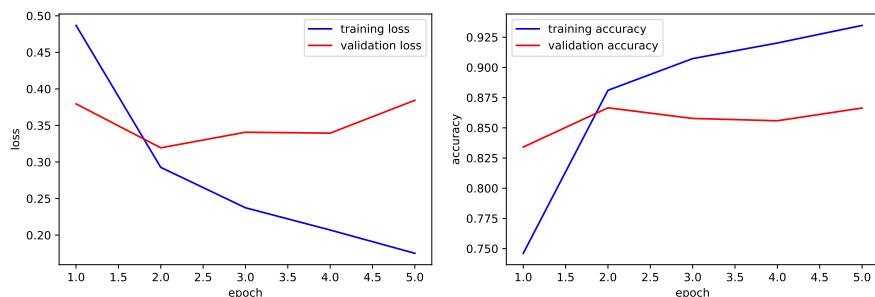


Fig. 15.6: The loss (left) and the accuracy (right) of the GRU classifier with a word embedding layer as functions of epoch for both the training and the validation sets.

Exercises:

Exercise 1: In the sentiment classification application using IMDB dataset, consider the same layers and datasets that were used in this chapter.

- (A) Remove the embedding layer. Train the network and estimate the accuracy on the test set. Comment on the results.

- (B) Replace the GRU layer with an LSTM layer with the same number of hidden units. What is the accuracy on the test set (make sure to use the embedding layer as in the original network)?
- (C) In this part, we wish to show the utility of CNNs for analyzing sequential data. In this regard, replace the GRU layer with a 1D convolutional layer (`layers.Conv1D`) with 32 filters and kernel of size 7. Use a one-dimensional max-pooling (`layers.MaxPooling1D`) with a pooling size of 2 (default), followed by a (one-dimensional) global average pooling layer (`GlobalAveragePooling1D`). What is the accuracy on the test set (use the embedding layer as in the original network)?

Exercise 2: In a regression problem, suppose we have an input sequence of (x_1, \dots, x_T) where $x_t = t - 2$ for $t = 1, \dots, T$ and $T \geq 2$. We would like to use the standard RNN with a “tanh” activation function and a 2-dimensional hidden state \mathbf{h}_t for $t = 1, \dots, T$ to estimate an output variable at t , denoted y_t , where $t = 1, \dots, T$. Assume $\mathbf{h}_0 = [0, 0]^T$ and all weight matrices/vectors and biases used in the structure of the RNN are weights/vectors/scalers of appropriate dimensions with all elements being 1. What is y_2 (i.e., y_t for $t = 2$)?

- A) 1 B) 2.99 C) 2.92 D) 2.52 E) 3.19

⊕ **Exercise 3:** Describe in words the main cause of the vanishing and exploding gradient problems in a standard RNN with a linear activation function.

Exercise 4: Consider a GRU with two layers that is constructed by the standard RNN stacking approach. In each layer we assume we have only one hidden unit. Let h_t^j denote the hidden state at time t for layer $j = 1, 2$. Suppose $x_1 = 0.5$, $h_0^j = 0, \forall j$, and all weight coefficients are 1 and biases -0.5. What is h_1^2 ?

- A) $\sigma_{\text{logistic}}(-0.5)\sigma_{\tanh}(-0.5)$
 B) $\sigma_{\text{logistic}}(0.5)\sigma_{\tanh}(-0.5)$
 C) $\sigma_{\text{logistic}}(-0.5)\sigma_{\tanh}(0.5)$
 D) $\sigma_{\text{logistic}}(0.5)\sigma_{\tanh}(0.5)$

Exercise 5: In a regression problem with a univariate output variable, we use Keras to train a GRU layer with 32 hidden units, followed by a dense layer. The network predicts the output variable based on bivariate input sequences of length 200. How many parameters in total are learned for this network?

- A) 41697 B) 48264 C) 3489 D) 3297 E) 3456

Exercise 6: In a sentiment classification application with binary output, we use Keras to train a network that includes a word embedding layer for a vocabulary size of 1000 and embedding dimensionality of 8, followed by a standard RNN layer with 64 hidden units, and then a dense layer with a logistic sigmoid activation function.

We use this network to classify word sequences of length 200. How many parameters in total are learned for this network?

- A) 12737
- B) 6272
- C) 6337
- D) 4737

References

- Abdel-Hamid, O., Mohamed, A.-r., Jiang, H., Deng, L., Penn, G., and Yu, D. (2014). Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(10):1533–1545.
- Abibullaev, B. and Zollanvari, A. (2019). Learning discriminative spatirospectral features of erps for accurate brain-computer interfaces. *IEEE Journal of Biomedical and Health Informatics*, 23(5):2009–2020.
- Adam-Bourdarios, C., Cowan, G., Germain, C., Guyon, I., Kégl, B., and Rousseau, D. (2015). The Higgs boson machine learning challenge. In Cowan, G., Germain, C., Guyon, I., Kégl, B., and Rousseau, D., editors, *Proceedings of the NIPS 2014 Workshop on High-energy Physics and Machine Learning*, volume 42 of *Proceedings of Machine Learning Research*, pages 19–55.
- Ambroise, C. and McLachlan, G. J. (2002). Selection bias in gene extraction on the basis of microarray gene-expression data. *Proc. Natl. Acad. Sci. USA*, 99:6562–6566.
- Anaconda (2023). <https://www.anaconda.com/products/individual>, Last accessed on 2023-02-15.
- Anderson, T. (1951). Classification by multivariate analysis. *Psychometrika*, 16:31–50.
- Arthur, D. and Vassilvitskii, S. (2007). k-means++: the advantages of careful seeding. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics.
- Bache, K. and Lichman, M. (2013). UCI machine learning repository. *University of California, Irvine, School of Information and Computer*.
- Bay, H., Tuytelaars, T., and Van Gool, L. (2006). Surf: Speeded up robust features. In Leonardis, A., Bischof, H., and Pinz, A., editors, *Computer Vision – ECCV 2006*, pages 404–417, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

- Benjamini, Y. and Hochberg, Y. (1995). Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57:289–300.
- Bergstra, J. and Bengio, Y. (2012). Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305.
- Bishop, C. (2006). *Pattern Recognition and Machine Learning*. Springer-Verlag.
- Blashfield, R. K. (1976). Mixture model tests of cluster analysis: Accuracy of four agglomerative hierarchical methods. *Psychological Bulletin*, 83:377–388.
- Braga-Neto, U. M. (2020). *Fundamentals of Pattern Recognition and Machine Learning*. Springer.
- Braga-Neto, U. M., Zollanvari, A., and Dougherty, E. R. (2014). Cross-validation under separate sampling: strong bias and how to correct it. *Bioinformatics*, 30(23):3349–3355.
- Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140.
- Breiman, L. (1999). Pasting small votes for classification in large databases and on-line. *Machine Learning*, 36(1-2):85–103.
- Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.
- Breiman, L. (2004). Population theory for boosting ensembles. *The Annals of Statistics*, 32(1):1–11.
- Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Chapman and Hall/CRC.
- Chatgpt (2022). <https://openai.com/blog/chatgpt>, Last accessed on 2023-02-15.
- Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.
- Chen, T. and He, T. (2015). Higgs Boson Discovery with Boosted Trees. In Cowan, G., Germain, C., Guyon, I., Kégl, B., and Rousseau, D., editors, *Proceedings of the NIPS 2014 Workshop on High-energy Physics and Machine Learning*, volume 42 of *Proceedings of Machine Learning Research*, pages 69–80.
- Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734.
- Chollet, F. (2021). *Deep Learning with Python*. Manning, Shelter Island, NY, second edition.
- Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling.
- Clemmensen, L., Witten, D., Hastie, T., and Ersbøll, B. (2011). Sparse discriminant analysis. *Technometrics*, 53(4):406–413.
- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Math. Control Signal Systems*, 2:303–314.

- Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893.
- Dechter, R. (1986). Learning while searching in constraint-satisfaction-problems. In *Proceedings of AAAI*.
- Devore, J. L., Farnum, N. R., and Doi, J. A. (2013). *Applied Statistics for Engineers and Scientists*. Cengage Learning, third edition.
- Devroye, L., Gyorfi, L., and Lugosi, G. (1996). *A Probabilistic Theory of Pattern Recognition*. Springer, New York.
- Dougherty, E. R., Kim, S., and Chen, Y. (2000). Coefficient of determination in nonlinear signal processing. *Signal Processing*, 80(10):2219–2235.
- Drucker, H. (1997). Improving regressors using boosting techniques. In *International Conference on Machine Learning*.
- Duda, R. O., Hart, P. E., and Stork, D. G. (2000). *Pattern Classification*. Wiley.
- Dudani, S. A. (1976). The distance-weighted k-nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(4):325–327.
- Dupuy, A. and Simon, R. M. (2007). Critical review of published microarray studies for cancer outcome and guidelines on statistical analysis and reporting. *J Natl. Cancer Inst.*, 99:147–157.
- Efron, B. (1983). Estimating the error rate of a prediction rule: Improvement on cross-validation. *Journal of the American Statistical Association*, 78(382):316–331.
- Efron, B. (2007). Size, power and false discovery rates. *The Annals of Statistics*, 35(4):1351–1377.
- f2py (2023). <https://www.numfys.net/howto/F2PY/>, Last accessed on 2023-02-15.
- Fisher, R. (1936). The use of multiple measurements in taxonomic problems. *Ann. Eugen.*, 7:179–188.
- Fisher, R. (1940). The precision of discriminant function. *Ann. Eugen.*, 10:422–429.
- Fix, E. and Hodges, J. (1951). Discriminatory analysis, nonparametric discrimination: consistency properties. *Technical report. Randolph Field, Texas: USAF School of Aviation Medicine*.
- Forgy, E. (1965). Cluster analysis of multivariate data: Efficiency versus interpretability of classification. *Biometrics*, 21(3):768–769.
- Freund, Y. and Schapire, R. E. (1997). A decision-theoretic generalization of online learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139.
- Friedman, J., Hastie, T., and Tibshirani, R. (2000). Additive Logistic Regression: a Statistical View of Boosting. *The Annals of Statistics*, 38(2).
- Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. *Annals of Statistics*, 29:1189–1232.
- Friedman, J. H. (2002). Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38:367–378.
- Fukushima, K. (1975). Cognitron: A self-organizing multilayered neural network. *Biological Cybernetics*, 20:121–136.

- Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202.
- Gers, F., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: continual prediction with lstm. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, volume 2, pages 850–855.
- Goodfellow, I. J., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press, Cambridge, MA, USA. <http://www.deeplearningbook.org>.
- Graves, A. (2013). Generating sequences with recurrent neural networks. *CoRR*.
- Graves, A., Mohamed, A., and Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649.
- Guardian-chatgpt (2023). <https://www.theguardian.com/technology/2023/feb/02/chatgpt-100-million-users-open-ai-fastest-growing-app>, Last accessed on 2023-02-15.
- Guyon, I., Weston, J., Barnhill, S. D., and Vapnik, V. N. (2002). Gene selection for cancer classification using support vector machines. *Machine Learning*, 46:389–422.
- Hand, D. and Till, R. J. (2001). A simple generalisation of the area under the roc curve for multiple class classification problems. *Machine Learning*, 45(2):171–186.
- Hansen, P. and Delattre, M. (1978). Complete-link cluster analysis by graph coloring. *Journal of the American Statistical Association*, 73(362):397–403.
- Hastie, T., Tibshirani, R., and Friedman, J. (2001). *The Elements of Statistical Learning*. Springer, New York, NY, USA.
- Hermans, M. and Schrauwen, B. (2013). Training and analysing deep recurrent neural networks. In Burges, C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K., editors, *Advances in Neural Information Processing Systems*, volume 26.
- Hochreiter, S. and Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780.
- IMDB (2023). <https://ai.stanford.edu/~amaas/data/sentiment/>, Last accessed on 2023-02-15.
- Ingber, L. (1998). Statistical mechanics of neocortical interactions: Training and testing canonical momenta indicators of eeg. *Mathematical and Computer Modelling*, 27(3):33–64.
- Ivakhnenko, A. G. (1971). Polynomial theory of complex systems. *IEEE Transactions on Systems, Man and Cybernetics*, 4:364–378.
- Ivakhnenko, A. G. and Lapa, V. G. (1965). *Cybernetic Predicting Devices*. CCM Information Corporation, New York.
- J. Friedman, T. Hastie, R. T. (2000). Additive logistic regression: A statistical view of boosting. *Annals of Statistics*, 28:337–407.
- Jain, A. and Dubes, R. (1988). *Algorithms for Clustering Data*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

- Jain, N. C., Indrayan, A., and Goel, L. R. (1986). Monte carlo comparison of six hierarchical clustering methods on random data. *Pattern Recognition*, 19(1):95–99.
- Johnson, S. C. (1967). Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254.
- Kaggle-survey (2021). <https://www.kaggle.com/kaggle-survey-2021>, Last accessed on 2023-02-15.
- Kass, G. V. (1980). An exploratory technique for investigating large quantities of categorical data. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 29(2):119–127.
- Kaufman, L. and Rousseeuw, P. J. (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley.
- Kay, S. M. (2013). *Fundamentals of Statistical Signal Processing, Volume III, Practical Algorithm Development*. Prentice Hall.
- Kelley Pace, R. and Barry, R. (1997). Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3):291–297.
- keras-gru (2023). <https://github.com/tensorflow/tensorflow/blob/e706a0bf1f3c0c666d31d6935853cfbea7c2e64e/tensorflow/python/keras/layers/recurrent.py#L1598>, Last accessed on 2023-02-15.
- keras-initializers (2023). <https://keras.io/api/layers/initializers/>, Last accessed on 2023-02-15.
- Keras-layers (2023). <https://keras.io/api/layers/>, Last accessed on 2023-02-15.
- Keras-losses (2023). <https://keras.io/api/losses/>, Last accessed on 2023-02-15.
- Keras-metrics (2023). <https://keras.io/api/metrics/>, Last accessed on 2023-02-15.
- Keras-optimizers (2023). <https://keras.io/api/optimizers/>, Last accessed on 2023-02-15.
- Keras-team (2019). <https://github.com/keras-team/keras/releases/tag/2.3.0>, Last accessed on 2023-02-15.
- Kim, H. and Loh, W.-Y. (2001). Classification trees with unbiased multiway splits. *Journal of the American Statistical Association*, 96(454):589–604.
- Kittler, J. and DeVijver, P. (1982). Statistical properties of error estimators in performance assessment of recognition systems. *IEEE Trans. Pattern Anal. Machine Intell.*, 4:215–220.
- Krijthe, J. H. and Loog, M. (2014). Implicitly constrained semi-supervised linear discriminant analysis. In *Proceedings of the 22nd International Conference on Pattern Recognition*, pages 3762–3767.
- Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105.
- Kruskal, J. (1964). Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. 29:1–27.

- Kuiper, F. K. and Fisher, L. (1975). 391: A monte carlo comparison of six clustering procedures. *Biometrics*, 31(3):777–783.
- Lachenbruch, P. A. and Mickey, M. R. (1968). Estimation of error rates in discriminant analysis. *Technometrics*, 10:1–11.
- Lance, G. N. and Williams, W. T. (1967). A general theory of classificatory sorting strategies 1. hierarchical systems. 9(4):373–380.
- Lawhern, V. J., Solon, A. J., Waytowich, N. R., Gordon, S. M., Hung, C. P., and Lance, B. J. (2018). Eegnet: a compact convolutional neural network for eeg-based brain-computer interfaces. *Journal of Neural Engineering*, 15(5):056013.
- Lemoine, B. (2022). <https://cajundiscordian.medium.com/is-lamda-sentient-an-interview-ea64d916d917>, Last accessed on 2023-02-15.
- Lin, M., Chen, Q., and Yan, S. (2014). Network in network. In *International Conference on Learning Representations (ICLR)*.
- Lloyd, S. P. (1982). Least squares quantization in pcm. *IEEE Trans. Inf. Theory*, 28(2):129–136.
- Loh, W.-Y. and Shih, Y.-S. (1997). Split selection methods for classification trees. *Statistica Sinica*, 7(4):815–840.
- Loog, M. (2014). Semi-supervised linear discriminant analysis through moment-constraint parameter estimation. *Pattern Recognition Letters*, 37:24–31.
- Loog, M. (2016). Contrastive pessimistic likelihood estimation for semi-supervised classification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 38(3):462–475.
- Louppe, G., Wehenkel, L., Sutera, A., and Geurts, P. (2013). Understanding variable importances in forests of randomized trees. In Burges, C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K., editors, *Advances in Neural Information Processing Systems*, volume 26.
- Lowe, D. (1999). Object recognition from local scale-invariant features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157 vol.2.
- Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 142–150, Portland, Oregon, USA. Association for Computational Linguistics.
- MacQueen, J. B. (1967). Some methods for classification and analysis of multivariate observations. In Cam, L. M. L. and Neyman, J., editors, *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press.
- matplotlib-colors (2023). https://matplotlib.org/stable/gallery/color/named_colors.html, Last accessed on 2023-02-15.
- matplotlib-imshow (2023). https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.imshow.html, Last accessed on 2023-02-15.
- matplotlib-markers (2023). https://matplotlib.org/stable/api/markers_api.html#module-matplotlib.markers, Last accessed on 2023-02-15.

- McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133.
- McLachlan, G. J. (1975). Iterative reclassification procedure for constructing an asymptotically optimal rule of allocation in discriminant analysis. *Journal of the American Statistical Association*, 70(350):365–369.
- McLachlan, G. J. (2004). *Discriminant Analysis and Statistical Pattern Recognition*. Wiley, New Jersey.
- Michiels, S., Koscielny, S., and Hill, C. (2005). Prediction of cancer outcome with microarrays: a multiple random validation strategy. *Lancet*, 365:488–92.
- Miller, T. W. (2015). *Marketing Data Science, modeling Techniques in Predictive Analytics with R and Python*. Pearson FT Press.
- Milligan, G. W. and Isaac, P. D. (1980). The validation of four ultrametric clustering algorithms. *Pattern Recognition*, 12(2):41–50.
- Müller, A. and Guido, S. (2017). *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media, Incorporated.
- NumPy-array (2023). <https://numpy.org/doc/stable/reference/routines.array-creation.html>, Last accessed on 2023-02-15.
- NumPy-arrsum (2023). <https://numpy.org/doc/stable/reference/generated/numpy.ndarray.sum.html>, Last accessed on 2023-02-15.
- NumPy-sum (2023). <https://numpy.org/doc/stable/reference/generated/numpy.sum.html>, Last accessed on 2023-02-15.
- Pandas-dataframe (2023). <https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html#pandas.DataFrame>, Last accessed on 2023-02-15.
- Pandas-io (2023). https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html, Last accessed on 2023-02-15.
- Pandas-readhtml (2023). https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_html.html, Last accessed on 2023-02-15.
- Pascanu, R., Gulcehre, C., Cho, K., and Bengio, Y. (2014). How to construct deep recurrent neural networks. In *Proc. of ICLR*.
- Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on Machine Learning*, volume 28, pages 1310–1318.
- Perkins, S., Lacker, K., and Theiler, J. (2003). Grafting: Fast, incremental feature selection by gradient descent in function space. *J. Mach. Learn. Res.*, 3:1333–1356.
- Pikelis, V. S. (1973). The error of a linear classifier with independent measurements when the learning sample size is small. *Statist. Problems of Control*, 5:69–101. in Russian.
- Pillo, P. J. D. (1976). The application of bias to discriminant analysis. *Communications in Statistics - Theory and Methods*, 5:843–854.
- pyplot (2023). https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html, Last accessed on 2023-02-15.

- Python-copy (2023). <https://docs.python.org/3/library/copy.html>, Last accessed on 2023-02-15.
- Python-repeat (2023). <https://docs.python.org/3/library/itertools.html#itertools.repeat>, Last accessed on 2023-02-15.
- Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1:81–106.
- Quinlan, J. R. (1993). *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- Ramalho, L. (2015). *Fluent Python*. O'Reilly.
- Raudys, S. and Pikelis, V. (1980). On dimensionality, sample size, classification error, and complexity of classification algorithm in pattern recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 2:242–252.
- Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788.
- Rice, J. A. (2006). *Mathematical Statistics and Data Analysis*. Belmont, CA: Duxbury Press., third edition.
- Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, Washington, D.C.
- Rousseeuw, P. (1987). Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53–65.
- Schuster, M. and Paliwal, K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681.
- Scikeras-wrappers (2023). <https://www.adriangb.com/scikeras/stable/generated/scikeras.wrappers.KerasClassifier.htmls>, Last accessed on 2023-02-15.
- Scikit-digits (2023). https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_digits.html, Last accessed on 2023-02-15.
- Scikit-eval (2023). https://scikit-learn.org/stable/modules/model_evaluation.html, Last accessed on 2023-02-15.
- Scikit-kNN-C (2023). <https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>, Last accessed on 2023-02-15.
- Scikit-metrics (2023). https://scikit-learn.org/stable/modules/model_selection.html, Last accessed on 2023-02-15.
- Scikit-silhouette (2023). https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_samples.html#sklearn.metrics.silhouette_samples, Last accessed on 2023-02-15.
- Scikit-split (2023). https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html, Last accessed on 2023-02-15.
- scipy (2023). <https://docs.scipy.org/doc/scipy/reference/stats.html>, Last accessed on 2023-02-15.

- scipy-normal (2023). https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.multivariate_normal.html, Last accessed on 2023-02-15.
- Selim, S. Z. and Ismail, M. A. (1984). K-means-type algorithms: A generalized convergence theorem and characterization of local optimality. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(1):81–87.
- Shahrokh Esfahani, M. and Dougherty, E. R. (2013). Effect of separate sampling on classification accuracy. *Bioinformatics*, 30(2):242–250.
- Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In Bengio, Y. and LeCun, Y., editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.
- Smith, C. (1947). Some examples of discrimination. *Annals of Eugenics*, 18:272–282.
- Sneath, P. H. A. and Sokal, R. R. (1973). *Numerical Taxonomy: The Principles and Practice of Numerical Classification*. W H Freeman & Co (Sd).
- Sorić, B. (1989). Statistical discoveries and effect-size estimation. *Journal of the American Statistical Association*, 84:608–610.
- Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.
- Tensorflow (2023). <https://anaconda.org/conda-forge/tensorflow>, Last accessed on 2023-02-15.
- Tensorflow-tpu (2023). <https://www.tensorflow.org/guide/tpu>, Last accessed on 2023-02-15.
- Theodoridis, S. and Koutroumbas, K. (2009). *Pattern Recognition, Fourth Edition*. Academic Press.
- Thoppilan, R., Freitas, D. D., Hall, J., Shazeer, N., Kulshreshtha, A., and et al. (2022). Lamda: Language models for dialog applications. *CoRR*.
- Thorndike, R. (1953). Who belongs in the family? *Psychometrika*, 18(4):267–276.
- Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288.
- UCI-EEG (2023). <https://archive.ics.uci.edu/ml/datasets/EEG+Database>, Last accessed on 2023-02-15.
- Vicari, D. (2014). Classification of asymmetric proximity data. *Journal of Classification*, 31:386–420.
- Wald, A. (1944). On a statistical problem arising in the classification of an individual into one of two groups. *Ann. Math. Statist.*, 15:145–162.
- Ward, J. H. (1963). Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244.
- Werbos, P. J. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.
- Wolpert, D. H. (1992). Stacked generalization. *Neural Networks*, 5(2):241–259.

- Yarowsky, D. (1995). Unsupervised word sense disambiguation rivaling supervised methods. In *In Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, pages 189–196.
- Zhu, J., Zou, H., Rosset, S., and Hastie, T. J. (2009). Multi-class adaboost. *Statistics and Its Interface*, 2:349–360.
- Zollanvari, A. and Dougherty, E. R. (2015). Generalized consistent error estimator of linear discriminant analysis. *IEEE Trans. Sig. Proc.*, 63:2804–2814.
- Zollanvari, A. and Dougherty, E. R. (2019). Optimal bayesian classification with vector autoregressive data dependency. *IEEE Transactions on Signal Processing*, 67(12):3073–3086.
- Zollanvari, A., James, A. P., and Sameni, R. (2020). A theoretical analysis of the peaking phenomenon in classification. *Journal of Classification*, 37:421–434.

Index

K-means, 322, 324
K-means++, 323, 324
P-value, 287
F-statistic, 287, 290
F-test, 287
F₁ score, 252
l₁ regularization, 167, 177
l₂ regularization, 167, 177
adagrad, 366
adam, 366
binary_crossentropy, 367
categorical_crossentropy, 367, 369
keras.
 Model.compile, 366, 402
 Model.evaluate, 377, 402
 Model.fit, 371, 402
 Model.predict, 377, 402
 Model, 363
 Sequential, 363
 callbacks.EarlyStopping, 371
 callbacks.ModelCheckpoint, 372
 layers.Conv1D, 413, 432
 layers.Conv2D, 402
 layers.Dense, 364
 layers.Dropout, 382
 layers.Embedding, 428
 layers.Flatten, 403
 layers.GRU, 422
 layers.LSTM, 422
 layers.TextVectorization, 424
 layers.simpleRNN, 416
 wrappers.scikit_learn, 386
 keras, 17, 358
 mean_absolute_error, 367
 mean_squared_error, 367
 rmsprop, 366
 scikeras, 386
 scipy.cluster.hierarchy.dendrogram,
 347
 scipy.cluster.hierarchy.linkage,
 346
 scipy.stats.multivariate_normal,
 99
 scipy.stats, 99, 276
 sgd, 366
 sklearn.
 cluster.AgglomerativeClustering,
 346
 cluster.KMeans, 324
 datasets.make_classification,
 284
 ensemble.AdaBoostClassifier,
 222
 ensemble.AdaBoostRegressor, 222
 ensemble.BaggingClassifier, 217,
 219
 ensemble.BaggingRegressor, 217,
 219
 ensemble.GradientBoostingClassifier,
 229

ensemble.GradientBoostingRegressor,
 229
 ensemble.RandomForestClassifier,
 218
 ensemble.RandomForestRegressor,
 218
 feature_selection.RFECV, 299
 feature_selection.RFE, 299
 feature_selection.SelectFdr,
 290
 feature_selection.SelectFpr,
 290
 feature_selection.SelectFwe,
 290
 feature_selection.SelectKBest,
 290
 feature_selection.f_classif,
 290
 feature_selection.f_regression,
 290
 feature_selection.
 SequentialFeatureSelector, 295
 impute.SimpleImputer, 181
 linear_model.ElasticNet, 180
 linear_model.Lasso, 180
 linear_model.LinearRegression,
 179
 linear_model.Ridge, 179
 metrics.RocCurveDisplay, 281
 metrics.accuracy_score, 259
 metrics.confusion_matrix, 259
 metrics.f1_score, 260
 metrics.precision_score, 260
 metrics.recall_score, 260
 metrics.roc_auc_score, 260
 metrics.roc_curve, 281
 metrics.silhouette_samples, 333
 metrics.silhouette_score, 333
 model_selection.GridSearchCV,
 271, 274
 model_selection.KFold, 245
 model_selection.LeaveOneOut,
 246
 model_selection.PredefinedSplit,
 274
 model_selection.RandomizedSearchCV,
 276
 model_selection.ShuffleSplit,
 249
 model_selection.StratifiedKFold,
 249
 model_selection.cross_val_score,
 248, 262, 263
 model_selection.cross_validate,
 263
 model_selection.train_test_split,
 116
 neighbors.KNeighborsClassifier,
 123
 neighbors.KNeighborsRegressor,
 144
 pipeline.Pipeline, 308, 312
 preprocessing.LabelEncoder, 130
 preprocessing.MinMaxScaler, 122
 preprocessing.OrdinalEncoder,
 130
 preprocessing.StandardScaler,
 122
 tree.DecisionTreeClassifier,
 199
 tree.DecisionTreeRegressor, 199
 utils.Bunch, 113
 sklearn, *see* scikit-learn
 sparse_categorical_crossentropy,
 367, 369
 tensorflow, 17, 358
 xgboost, 17, 234

accuracy, 127, 253
 AdaBoost, 220
 AdaBoost.R2, 222, 223
 AdaBoost.SAMME, 220, 222
 agent, 5
 agglomerative clustering, 336, 346
 alternative hypothesis, 287
 Anaconda, 21
 analysis of variance, 287, 290
 ANOVA, *see* analysis of variance
 apparent error, 239

- area under the receiver operating characteristic curve, 255
artificial intelligence, 7
attribute, 10
average-pooling, 398
- backpropagation, 357
backpropagation through time, 419
backward hidden states, 417
bagging, 216
batch size, 357
Benjamini–Hochberg false discovery rate procedure, 289, 290
bidirectional RNN, 417
binary decision tree, 187
Bonferroni correction, 288
boosting, 220
bootstrap estimator, 240
- C4.5, 191
CART classifier, *see* classification and regression tree classifier
CART regressor, *see* classification and regression tree regressor
CHAID, 191
chaining effect, 344
classification, 3
classification and regression tree classifier, 191, 199
classification and regression tree regressor, 197, 199
classifier, 3
cluster, 319
cluster dissimilarity function, pairwise, 337
cluster validity analysis, 330
clustering, 3, 319
clustering, agglomerative, *see* agglomerative clustering
clustering, complete linkage, *see* linkage, complete
clustering, divisive, *see* divisive clustering
clustering, group average linkage, *see* linkage, group average
clustering, hierarchical, *see* hierarchical clustering
clustering, partitional, *see* partitional clustering
clustering, single linkage, *see* linkage, single
clustering, Ward’s linkage, *see* linkage, Ward
CNN, *see* convolutional neural network
coefficient of determination, *see* R-squared
Colab, 359
conda, 21, 22
confusion matrix, 251
convolution, 395
convolutional neural network, 393, 402, 413
cross-correlation, 394
cross-validation, 240
cross-validation, external, 305, 316
cross-validation, feature selection, 316
cross-validation, internal, 316
cross-validation, model evaluation, 316
cross-validation, model selection, 316
cross-validation, nested, 316
cross-validation, random permutation, 244, 249
cross-validation, standard K -fold, 241, 248
cross-validation, stratified K -fold, 243, 249
CRUISE, 191
curse of dimensionality, 284
- data, 8
data leakage, 121
decision stump, 222
decision tree, 188
deep RNN, 416
dendrogram, 340
dense layer, *see* fully connected layer

- density estimation, 3
- design process, 6
- dimensionality, 10, 283
- dimensionality reduction, 3, 284
- dissimilarity function, average, 337
- dissimilarity function, max, 337
- dissimilarity function, min, 337
- dissimilarity function, minimum variance, *see* dissimilarity function, Ward
- dissimilarity function, Ward, 337
- distance-weighted kNN classifier, 136
- distance-weighted kNN regressor, 146
- divisive clustering, 336
- dropout, 381
- dropout rate, 381
- E0 estimator, 240
- early stopping, 222, 372
- elastic-net, 167, 177
- elbow phenomenon, 328
- embedded methods, 297
- embedding layer, *see* word embedding layer
- encoding, 115, 130, 367, 428
- ensemble learning, 9
- epoch, 357
- error estimation, 16, 237
- error rate, 126, 252
- exclusive clustering, 319
- exhaustive search, 293
- exploding gradient problem, 418, 421
- false discovery rate, 252, 288
- false negative, 251
- false negative rate, 252
- false positive, 251
- false positive rate, 251
- family-wise error rate, 288, 290, 301
- fdr, *see* false discovery rate
- feature, 10
- feature extraction, 10, 283
- feature map, 396
- feature selection, 10, 283, 286
- feature vector, 10
- filter methods, 286
- flattening, 403
- forward hidden states, 417
- fully connected layer, 364
- FWER, *see* family-wise error rate
- gated recurrent unit, 421
- GBRT classifier, *see* gradient boosting regression tree classifier
- GBRT regressor, *see* gradient boosting regression tree regressor
- global average pooling, 403, 413
- gradient boosting, 222
- gradient boosting regression tree classifier, 228, 229
- gradient boosting regression tree regressor, 227, 229
- gradient clipping, 421
- grafting, 298
- graphical processing unit, 359, 379
- greedy search, 293, 294
- grid search, 268
- grid search using cross-validation, 270
- grid search using validation set, 269, 273
- GRU, *see* gated recurrent unit
- hierarchical clustering, 335
- hold-out estimator, 238
- hyperparameter, 6
- ID3, 191
- impurity drop, 192
- intelligence, 7
- interpretability, 284
- iterative embedded methods, 298
- Jupyter notebook, 23
- k-nearest neighbors classifier, 123, 133

- k-nearest neighbors regressor, 138, 144
Kaggle, 17
kNN classifier, *see* k-nearest neighbors classifier
kNN regressor, *see* k-nearest neighbors regressor

labeled data, 5
lasso, 167, 177, 180
LDA, *see* linear discriminant analysis
leaf nodes, 187, 193
learning, 8
least squares, 176
leave-one-out, 242
linear discriminant analysis, 155, 157, 161
linear regression, 176, 179
linkage, complete, 338, 344, 346
linkage, group average, 339, 344, 346
linkage, minimum variance, *see* linkage, Ward
linkage, single, 338, 344, 346
linkage, Ward, 339, 344, 346
Lloyd's algorithm, 321
local translation invariance, 399
logistic classification, *see* logistic regression
logistic regression, 162, 166, 167
long short-term memory, 421
loss, 357, 367
LSTM, *see* long short-term memory

machine learning, 2
macro averaging, 259
MAE, *see* mean absolute error
marketing data science, 130
Markov decision process, 5
matrix updating algorithm, 340, 341
max-pooling, 398
MDI, *see* mean decrease impurity
mean absolute error, 266
mean decrease impurity, 297
mean square error, 265
micro averaging, 259
mini-batch, 357

model evaluation, 16
model selection, 6, 237, 267
MSE, *see* mean square error
MUA, *see* matrix updating algorithm
multilayer perceptron, 352
multinomial logistic regression, 166
multiple linear regression, 176
multiple testing corrections, 288

non-iterative embedded methods, 297
normal equation, 177
null hypothesis, 287

observed significance level, *see* P-value
odds, 170
odds ratio, 171
one-versus-rest, 259
optimizer, 357, 366
ordinary least squares, 176
overfitting, 379

padding, 395
parameter sharing, 397
partitional clustering, 319
pasting, 219
peaking phenomenon, 284
Pearson's correlation coefficient, 141
perceptron, 352, 355
pip, 21
positive predictive value, 252
precision, *see* positive predictive value
pseudo labels, 5
Python programming, 17, 23
 StopIteration, 52
 __init__.py, 51, 64
 __next__() method, 52, 55
 def keyword, 47
 enumerate function, 44
 for loop, 41
 help, 32
 if-elif-else statement, 47
 iter function, 52
 matplotlib, 94

Axes.plot, 95
%matplotlib, 94
pyplot.figure, 96
pyplot.pcolormesh, 102
pyplot.plot, 95
pyplot.scatter, 99
pyplot.subplots, 97
pyplot.subplot, 96
pyplot style, 94
backend, 94
frontend, 94
object-oriented style, 95
next function, 52
numpy, 63
 add function, 76
 arange function, 68
 argmax function, 76
 argmin function, 76
 argsort function, 76
 array function, 64
 astype method, 65
 copy method, 71
 divide function, 76
 dtype attribute, 65
 eye function, 68
 floor_divide function, 76
 full function, 68
 itemsize attribute, 67
 log10 function, 76
 log function, 76
 max function, 76
 mean function, 76
 meshgrid function, 102
 min function, 76
 mod function, 76
 multiply function, 76
 ndarray, 63
 ndim attribute, 66
 newaxis object, 74
 ones function, 68
 power function, 76
 prod function, 76
 random.normal function, 69
 ravel method, 74
 reshape method, 73
shape attribute, 66
size attribute, 66
sort function, 76
sqrt function, 76
std function, 76
subtract function, 76
sum function, 76
var function, 76
zeros function, 67
array indexing, 69
array slicing, 70
array view, 70
broadcasting, 77
fancy indexing, 71
ufuncs, *see* Python
 programming, numpy,
 universal functions
 universal functions, 75
 vectorized operation, 75, 77
pandas, 81
 DataFrame, 86
 DataFrame, columns attribute,
 92
 DataFrame, drop method, 105
 DataFrame,fillna method,
 105
 DataFrame, iloc attribute, 91
 DataFrame, index attribute, 92
 DataFrame, loc attribute, 91
 DataFrame, explicit index, 90
 DataFrame, from a dictionary of
 Series, 88
 DataFrame, from a list of
 dictionaries, 89
 DataFrame, from arrays or lists,
 88
 DataFrame, from dictionary of
 arrays or lists, 88
 DataFrame, implicit index, 90
 DataFrame, masking, 91
 Series, 81
 Series, iloc attribute, 85
 Series, index attribute, 82
 Series, loc attribute, 85
 Series, values attribute, 82

- Series**, explicit index, 83
- Series**, implicit index, 83
 - read and write data, 93
- range** constructor, 43
- yield** keyword, 54
- zip** function, 45
- alias, 51
- arithmetic operator, 26
- built-in data structures, 27
- dictionary**, 36
 - dictionary**, `dict` constructor, 38
 - dictionary**, `items` method, 43
 - dictionary**, `keys` method, 37
 - dictionary**, `values` method, 38
 - dictionary**, key, 36
 - dictionary**, value, 36
- dynamically-typed, 24
- extended iterable unpacking, 39, 40
- generator expression, 57
- generator function, 53
- genexprs, *see* Python programming, generator expression
- idioms, 43
- indentation, 41
- iterable, 41
- iterator, 41, 52
- list, 27
 - list comprehension, 46
 - list, `append` method, 30
 - list, `copy` method, 31
 - list, `insert` method, 30
 - list, `list` constructor, 31
 - list, `pop` method, 30
 - list, `remove` method, 30
 - list, `sort` method, 30
 - list, deep copy, 31
 - list, indexing, 28
 - list, modifying elements, 30
 - list, shallow copy, 31
 - list, slicing, 29, 31
- listcomps, *see* Python programming, list comprehension
 - logical operator, 26
 - magic function, 76
 - membership operator, 27
 - module, 49
 - package, 50
 - relational operator, 26
 - rounding errors, 26
 - sequence packing, 35
 - sequence unpacking, 34, 39
 - set, 38
 - set, difference, 39
 - set, intersection, 39
 - set, symmetric difference, 39
 - set, union, 39
 - string, 25
 - tuple, 33
 - variables, 24
- QDA, *see* quadratic discriminant analysis
- quadratic discriminant analysis, 182
- QUEST, 191
- R-squared, 145, 266
- random forest, 217
- random search, 275
- RCO, *see* relative change in odds
- recall, *see* true positive rate
- receiver operating characteristic curve, 256, 281
- recurrent neural network, 415, 416, 422
- recursive feature elimination, 298, 299
 - recursive feature elimination with cross-validation, 298
- regression, 3
- regressor, 3
- regularization, 166
- reinforcement learning, 5
- relative change in odds, 172
- residual sum of squares, 145, 176
- resubstitution estimator, 239
- RFE, *see* recursive feature elimination

- RFE-CV, *see* recursive feature elimination with cross-validation
- ridge, 167, 177
- RNN, *see* recurrent neural network
- ROC, *see* receiver operating characteristic curve
- ROC AUC, *see* area under the receiver operating characteristic curve
- sample size, 10
- SBS, *see* sequential backward search
- scikit-learn, 17, 112
- scikit-learn, estimator, 112
- scikit-learn, predictor, 113
- scikit-learn, transformer, 112
- segmentation, 304
- self-training, 5
- semi-supervised learning, 5
- sensitivity, *see* true positive rate
- sequential backward search, 295
- sequential forward search, 293
- SFS, *see* sequential forward search
- shrinkage, 166
- shuffle and split, *see* cross-validation, random permutation
- signal modeling, 10
- signal segmentation, 12
- significance level, 287
- silhouette coefficient, 330, 333
- silhouette width, 331
- silhouette width, average, 332
- silhouette width, overall average, 332
- simple linear regression, 176
- softmax, 364
- sparse connectivity, 395
- specificity, *see* true negative rate
- splits in CART, 191
- splitter, *see* splitting strategy in CART
- splitting strategy in CART, 192
- stacked RNN, 416
- stacking, 215
- statistical test, 287
- strides, 397, 398
- supervised learning, 2
- surrogate classifier, 241
- tensor processing unit, 359
- test data, 16
- test set, 116
- test statistic, 287
- test-set error estimator, 127
- text standardization, 424
- text tokenization, 424
- total sum of squares, 145
- training, 6
- training data, 6
- translation equivariant, 399
- true negative, 251
- true negative rate, 251
- true positive, 251
- true positive rate, 251
- underfitting, 193
- unidirectional RNN, 417
- unlabeled data, 5
- unsupervised learning, 3
- UPGMA, *see* linkage, group average
- vanishing gradient problem, 418, 421
- weighted averaging, 259
- word embedding layer, 427
- wrapper methods, 293
- XGBoost, 230, 234