Karol Przystalski
Maciej J. Ogorzałek
Jan K. Argasiński
Wiesław Chmielnicki

Pattern Recognition Primer

Springer

Pattern Recognition Primer

Karol Przystalski · Maciej J. Ogorzałek ·
Jan K. Argasiński · Wiesław Chmielnicki

Pattern Recognition Primer

Springer

Karol Przystalski
Faculty of Physics, Astronomy and Applied Computer Science
Institute of Applied Computer Science
Jagiellonian University
Krakow, Poland

Maciej J. Ogorzałek
Faculty of Physics, Astronomy and Applied Computer Science
Institute of Applied Computer Science
Jagiellonian University
Krakow, Poland

Jan K. Argasiński
Faculty of Physics, Astronomy and Applied Computer Science
Institute of Applied Computer Science
Jagiellonian University
Krakow, Poland

Wiesław Chmielnicki
Faculty of Physics, Astronomy and Applied Computer Science
Institute of Applied Computer Science
Jagiellonian University
Krakow, Poland

Sano - Centre for Computational Medicine
Krakow, Poland

ISBN 978-3-031-91815-5 ISBN 978-3-031-91816-2 (eBook)
https://doi.org/10.1007/978-3-031-91816-2

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

Competing Interests The authors have no competing interests to declare that are relevant to the content of this manuscript.

Genesis of This Book

The term Pattern Recognition is commonly used in different fields of science, engineering, and a wide range of real-world applications. It is connected with machine learning and is based on artificial intelligence methods dedicated to discoveries of specific features—patterns in images, measurement data series, text files, biomedical data of any kind (ECG, EEG, imaging data, DNA, etc.), data from real observations in physical systems, and many others.

Pattern Recognition Primer explores mostly used classification methods in intelligible way. Each method is deeply explained, so even non-scientist readers are able to understand how it works. The book starts with an explanation of each basic statistical and mathematical terms that are used in following chapters. Every chapter contains easy to understand code samples. At the end of each chapter reader is able to do some exercises on his own to consolidate knowledge. There are solutions to the exercises provided in Appendix A so that the reader can compare results. Pattern Recognition Primer is a book intended for students, teachers and everyone who would like to understand how pattern recognition and machine learning works.

The concept of the book started during the preparation of the Ph.D. thesis in machine learning by Karol Przystalski under the supervision of Prof. Maciej Ogorzałek—some ten years ago (2014–2015). The majority of text books available at that time offered only a few classification methods that were explained in a proper way. Many methods were typically briefly explained. In each textbook, different methods were explained. This book attempts to explain from top to bottom each method with all the needed details. Each method is further demonstrated in Python and additional exercises. This is something that is missing from many machine learning/pattern recognition books.

The reason for such a situation is that most of them are dedicated to readers with some basic experience in this area. We want to introduce a book that is intended for readers with no experience in machine learning at all.

Special and unique features of this book are:

- an overview of commonly used classification methods in one place,
- deep learning methods explained thoroughly,

- each classification method is explained in detail,
- in this book, each statistical or mathematical aspect of each classification methods is explained in details. This makes the understanding of each method easier.

What are the Benefits of This Book for the Reader?

This book is suitable for wide range of data scientists, machine learning engineers, data analysts, academic teachers and especially for students. One of the advantages of this book is that the reader is able to get a detailed explanation of each commonly used classification method. Each method is presented in a easily to understand way by using simple comparison examples and additionally ready to use examples written in Python. Teachers can use it for their lectures/lessons and students to learn about classification methods. Each exercise have a solution that is described in the appendix to the book. Additionally, ready to use examples written in Python are shown. Teachers can use it for their lectures/lessons and students to learn about classification methods.

Full development and many changes to this book took almost ten years to complete. This has been caused in large part by rapid changes observed in the domain, extremely fast development of the area of deep neural networks and AI and their booming applications. Unfortunately during the course of preparation of the manuscript we lost our extremely precious collaborator Dr. Wiesław Chemielnicki. We miss his thoughtful comments and we wish to express our gratitude for his important contributions especially to the SVM chapters. We express also our gratitude to numerous colleagues and friends who gave us support.

We thank also our students who followed courses in Signal processing, Biometrics, Machine learning and Neural networks and applications.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1
Introduction to Pattern Recognition

Pattern recognition has become a very popular buzzword over the last few years and is widely used in many commercial solutions. In [1], we can find many trends for 2023. There are such trends as large language models, algorithms, deep learning, and so on. Most of these trends are more or less related to the topic of this book. What is important here is that for several years in each of these trend lists we can find many references to artificial intelligence and pattern recognition. We predict an even more comprehensive expansion of pattern recognition usage in the upcoming years.

Before we go deeper into some mathematical aspects of pattern recognition in this chapter, we explain what pattern recognition and related terms are. A pattern can be described in many ways. In [2] it is described as "opposite of chaos; it is an entity, vaguely defined, that could be given a name". The two best-known pattern recognition types are image recognition and speech recognition. We use them on a daily basis and sometimes don't even know about them. Face detection in a camera is an example of image recognition. Siri, which is part of each iPhone device, contains speech-recognition algorithms. These are only two examples of a large set of pattern recognition usage examples. A more detailed but not complete list of patterns recognition usage cases can be found in [3].

1.1 Frameworks and Libraries

There are many free frameworks and libraries available. They are not new, as we have had various solutions for more than 20 years. Solutions such as scikit-learn [4] or Theano [5] were introduced in 2007 and 2008, but there are some libraries that were introduced even earlier [6, 7]. More and more solutions are introduced each year. So far, we have more than 100 solutions. Many of them have been introduced over the last ten years [8–11]. A list with a short description is presented in Appendix C. In this

K. Przystalski et al., *Pattern Recognition Primer*,
https://doi.org/10.1007/978-3-031-91816-2_1

book, we decided not to use any framework or library unless it is totally necessary. We do not want to focus on any specific solution, as the goal is to learn how some methods work and not how to use scikit or any other solution. There are many books in which such solutions are explained in depth [5, 12–21], but in most of them the methods are not explained well and the focus is more on library/framework explanation. In some cases, machine learning methods are treated as a black box with a superficial explanation of each. In our opinion, anyone who wants to start with machine learning should understand how the methods work inside in the first place, and then use a library/framework of one's own choice. This should provide a better understanding of how to use a given method, set the right parameters, and finally choose the right method for a given problem. We use, whenever possible, a non-vectorized approach. It is slower but in our opinion easier to understand for beginners. The only exception is the neural network chapter, as the presented examples are more complex. Calculating it without using TensorFlow or other solutions that are polished for such cases would take too long.

1.2 Terminology

Artificial Intelligence is about algorithms that behave like intelligent human beings. Steven Spielberg's *A.I.* (2001) movie presents robots that behave like humans. Almost like humans. Such robots as shown in the movie are still more fiction than science. There are some advanced robots such as Nadine [22], which looks and, in some situations, behaves like a human. Currently, the main part of the car building process is automated. Robots replaced humans in building cars by using algorithms that are usually developed to perform only one task. Such algorithms are not thought to learn to do new tasks on their own. It is hard to call them intelligent. Alan Turing is known to be the godfather of artificial intelligence. He proposed a test [23] called his name that can be used to distinguish whether an algorithm/method is artificial intelligence. It uses a chat interface to measure it. A human is having a conversation with someone else on the other end. He does not know if it is an algorithm or a human with whom he is talking. The algorithm passes the test if the human cannot distinguish whether he is talking to a human or not. Passing the test means that an algorithm can be considered as artificial intelligence.

Fun fact: McCarthy coined the term *AI* in 1955 in connection with a proposed summer workshop at Dartmouth College, which many of the world's leading thinkers in computing attended. In 1965, McCarthy became the founding director of the Stanford Artificial Intelligence Laboratory (SAIL), where research was conducted into machine intelligence, graphical interactive computing, and autonomous vehicles.[a]

[a] https://engineering.stanford.edu/news/stanfords-john-mccarthy-seminal-figure-artificial-intelligence-dead-84.

Machine learning is very often mistaken for pattern recognition, deep learning, or artificial intelligence. The reason why these terms are used as synonyms is that they all have a lot in common and are easily generalized, especially by anyone without knowledge of the differences. There are many definitions of machine learning. A short list of different definitions can be found in [24]. In fact, machine learning is the crucial part of the pattern recognition process. We will come back to this process later in this chapter. Machine learning is about algorithms that learn a new behavior based on given data. We are able to teach the algorithm so that it can predict the cases unknown before. The prediction process is also known as classification. There is a huge set of such algorithms, which are called classification methods. Implementations of 17 families are analyzed in [25] 179 classification methods. The following chapters are about classification methods that are the main part of the pattern recognition process.

Data science is about obtaining value from the data. Sometimes it is also called *data analysis* or *data mining*, but data science is a bit more general. Drew Conway during one of O'Reilly's conferences proposed a diagram in which he focused on the skills that are needed to become a data scientist [26]. It comprises three different skills (see Fig. 1.1). First, to analyze the data, we need statistical and mathematical knowledge. Second, we need to have computer science skills as we need to know how to collect and proceed with the data. Third, we need to have domain expertise to know what we are analyzing. Data scientists are developers who extract knowledge or useful information that is not given strictly in large amounts of data.

For a couple of years *big data* has become a popular term to describe solutions in which a large amount of data is processed in parallel. Especially big data-based learning has started to be widely used in recent years [27]. There are good examples in e-Commerce solutions like Amazon where all users actions are saved and analyzed for recommendation systems and further financial and inventory predictions. Google uses images to build methods that are used in Google Photos. BMW and Tesla use car sensors to build autonomous cars. There are also many other examples; for now, mostly larger companies use big data for prediction, but there have been many startups showing up in this area recently. The opposite of big data is *small data*. Most pattern recognition problems are still based on small data. It is difficult to distinguish between small and large data. The very loose difference is in parallel computation. A more detailed comparison between small and large data can be found in [28].

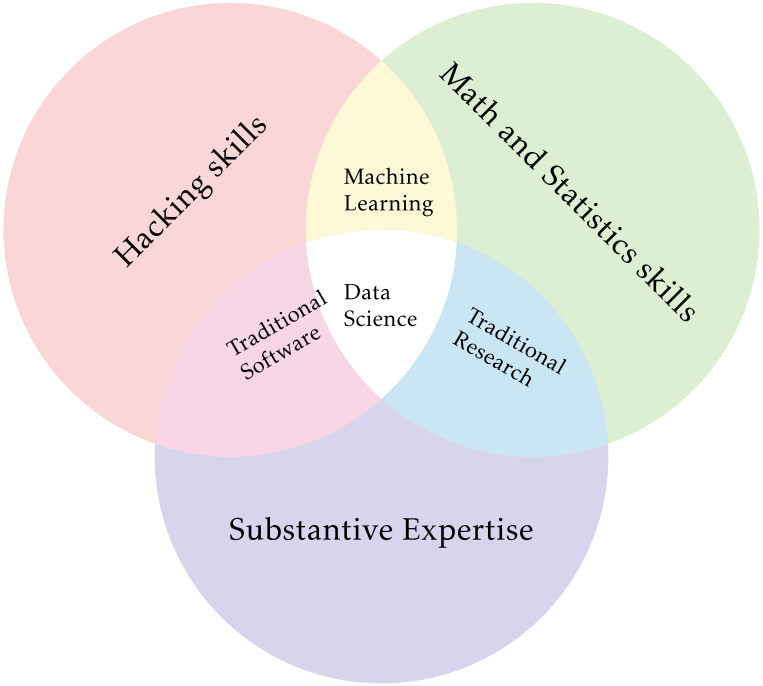


Fig. 1.1 Data Science with references to other terms known as Data Science Venn Diagram. *Source* https://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram

1.3 The Process

To find a pattern on an image, we need to follow the process that is divided into a few steps. In the following process, we simplify it to a binary classification problem of iris cancer. This means that we can have a data set of images of malignant or benign cases. Each pattern recognition case is individual, but each consists of some steps that are the same in each case:

1. **Assemble data**. In this step, we are *collecting the data* [29]. This part can be difficult in some cases, especially if we want to solve a specific, complex and difficult to predict problem, such as one of the medical classification problems. Another issue related to the data collection part is the quality of the data. We should not start the next step if we have only images of healthy irises or if the set of malignant images is only a small percentage of the whole database. There will be no meaningful results. The perfect scenario is to have a ratio from malignant to benign cases of 50% each. At the same time, in most cases the database should not be small like 50 cases in total. In the case of classification problem with more than two possible prediction labels, each label data set should be equal in the best case. A robust algorithm should generalize the problem. Bigger database makes it

possible for such an algorithm to generalize better. We explain the generalization more thoroughly in this chapter, while we go into detail with specificity and sensitivity.

2. **Data preprocessing**. If we have a data set of images, we usually need to do some image processing work before we start the feature extraction or training part. Image-processing work can be noise reduction, image scaling, or other similar actions that make the input data have the same format. This kind of work applies also to other data sets such as sound or text.
3. **Feature extraction**. As soon as we have our database of malignant and benign cases, we should *extract features* from each image. Doctors use some characteristics of an iris melanoma to make a diagnosis. Most methods need more "computer-friendly" information. The feature types are explained in the next section of this chapter. To simplify again, we need to transform the characteristics used by physicians into a numerical form. Image processing is commonly used for feature extraction if we deal with images. Using it, we can easily measure the asymmetry of the iris shape. Asymmetry can be one of the features. The goal in this phase is to create a vector of features. Building the feature vector by feature extraction is very specific to each classification problem. Plenty of ideas on how to make it efficient for a given problem have been published in [30].
4. **Feature selection**. The features we extracted should usually be *normalized*. How we should do this depends on the implementation of the classification method. In most cases, we normalize the feature value to be in the range of 1 and -1. It is important to do so because sometimes the feature vector consists of too many features and before we go to the next phase we select only the best features. There are many methods [31] to check the quality of the features and its impact on the overall classification result. We describe a few of these later in this book. *Feature selection* is significant for a few reasons. Reducing the number of features increases performance. In other words, it makes the training part happen faster. Each additional feature brings another dimension to our classification problem. Increasing the dimensionality is good to some extent, because if the dimension is too high, we will lose the generalization of our algorithm and finally the prediction success rate [32]. It is explained in more detail later in this chapter. The feature selection part is not needed for some deep learning methods where almost pure data are used as the input.
5. **Training**. This part is called the training or learning phase. Depending on the type of classification method used, this part can be skipped. We will describe the types of classification methods in more detail later in this chapter. The goal of the learning part is the **model**. **It's build with using training data set**. In [24] the model is described as nothing more than the result of applying an algorithm to a data set and is usually a representation of the data. Later, we will describe how to properly prepare a training data set. During the training part, the algorithm is learning how to predict the given data set. To simplify, we can say that the algorithm sets the parameters to a value that gives the highest accuracy for the given data set. These parameters are different depending on the method used. Some parameters can be set by us before the learning part starts.

6. **Prediction**. This is the essential part of the whole process. In this phase, we take the model and execute our prediction on the test data set. Our algorithm returns the predicted label for each feature vector. Although it is the crucial part, in most cases it takes much less time than other parts of the process.
7. **Validation**. Some models and predictions are better than others. It depends on many aspects. We have mentioned how the data set should be collected, what kind of features we extract, what feature we select, what classification method and parameters are used, etc. Good practice is to run the training and prediction phase with different parameters and different classification methods. This allows us to get a wider overview and choose the best methods from the set that we have checked. In addition, some measurements are available to check the quality of the prediction results. Some are described in more detail later in this chapter. It is important to note that we should have a validation data set to verify the model on a data set that was never used before for training or testing. We explain it in more detail later in this chapter.

We can also report our results and publish them, but it is not mandatory for obvious reasons. The process can also be described in a shorter way than in [33] or [3]. In later chapters, most methods are supervised to fully cover the process explained above. Methods in which feature selection is usually done during the training and the next validation are unsupervised. The reason for this is the lack of labels in the data sets. When it comes to deep learning, we usually skip parts of feature extraction and selection, since the data given in deep learning methods are in many cases raw data. Deep learning methods are also used to discover features during training. We explain it more extensively in the last chapter. Reinforcement learning also works a bit differently and does not exactly fit the explained process.

1.4 Features

There are a few terms related to features in pattern recognition. In the beginning, we describe the data collection and feature extraction in more detail. Next, we focus on a feature vector and a feature space. At the end of this part, we describe the kinds of feature that we have.

Data collection and feature extraction are usually the most time-consuming part. The approach in both cases is individual for each classification problem. Let us consider two examples. If we want to analyze the customer behavior of an e-Commerce solution, we would need to set up an infrastructure of servers with solutions like Apache Flume or similar. In the event of a medical problem, such as bone cancer, we need to collect X-ray images, usually directly from a doctor's medical database. For each problem, different features need to be extracted. For our e-Commerce example, we can extract such features as the operating system used by the customer, the country of the customer, how often she/he buys a product, etc. In case of a bone cancer detection problem, our feature vector can consist of bone symmetry, fractal

box dimension value, differences in the image during a time slot, etc. A feature vector consists of at least two features. It can be mathematically described as follows:

$$x_i = [x_{i1}, \ldots, x_{im}]^T \in \mathbb{R}, \text{ where } i = 1, \ldots, n. \tag{1.1}$$

We have n feature vectors x_i. It consists of m features $x_{i1}, \ldots, x_{im}$. The number of features m depends on the problem to be solved and how the features affect the accuracy. Each additional feature increases the dimension of the problem to be solved. It is called *feature space*. The feature space is two-dimensional if we have two features, three-dimensional in the case of three features, and so on. We show a few examples later in this book.

As shown in two previous examples, the features can differ from each other. Depending on the classification problem, we can deal with different types of feature. In [34] a taxonomy of features is presented (see Fig. 1.2). We have two main types of features: quantitative and qualitative. Qualitative features are those with a small number of possible values. Quantitative features have a number of possible values. Let us take a few examples to get a better understanding. If we want to predict brain cancer only on the basis of MRI images, then one of our features would probably be the asymmetry. Without using more sophisticated methods such as fractal methods to measure the asymmetry, we would say that there are four possible values. We can say that the brain on the images is symmetric, x-axis asymmetric, y-axis asymmetric, or completely asymmetric.

In 1983 the Detroit Pistons won against the Denver Nuggets 186–184. This game is known as the one with the highest score in the NBA so far. Usually, the scores are somewhere between 80 and 120, but we cannot exclude a score that is lower or greater. Let us assume that we would like to take the best score of a team as one of our features to predict who will be the NBA Champion this year. We would need to consider a wide range of values.

In 1986 we witnessed the Chernobyl nuclear reactor disaster. It happened because of the lack of well-prepared security procedures. Since then, many security procedures have been introduced to avoid this kind of situation in the future. Let us imagine

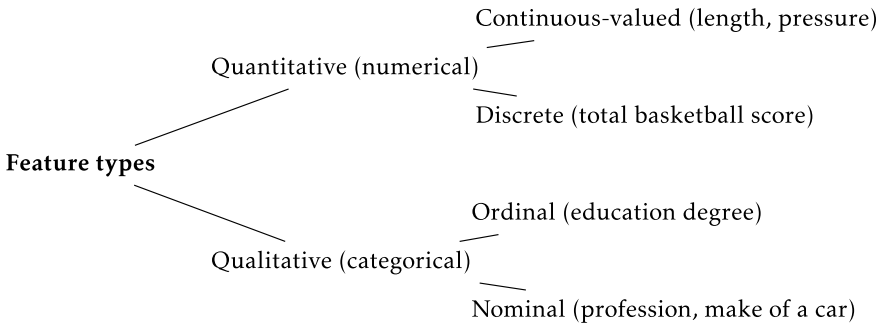


Fig. 1.2 The feature taxonomy proposed in [34]

that we want to build a classifier that will be able to predict such a situation. One of the most important features would be the temperature within the reactor. The range is also quite wide in this case.

Fun fact: Children typically begin recognizing letters between the ages of 2 and 4, with significant individual variation. By age 2, many children can sing or say aloud the "ABC" song. Around age 3, they may recognize about half the letters in the alphabet and start to connect letters to their sounds. By age 4, children often know all the letters of the alphabet and their correct order [35].

A classifier needs a feature vector consisting of values that are readable for it. This means that we need to convert our feature values to a numerical value. Few commonly known methods are used for making this happen, such as discretization, approximation, or just simple value assignment. The asymmetry described above can be represented as follows:

- symmetry as 1,
- y and x axis asymmetry as 0,
- completely asymmetry as -1.

In this example, we assign a value for each case. We assumed here that the x- or y-axis asymmetry does not make any difference in setting the diagnosis. Some other features need to be represented by a number of a range. Return to our example of skin cancer. In skin cancer, some scoring methods are used when making the diagnosis, taking into account the age of the patient. This is because the probability of skin cancer increases with age. Hunter score [36] divides the age factor into a few ranges such as 0–20, 20–30, etc. As a feature, we need to assign each range a value that will be used next in the classification. If we have a bunch of values divided into ranges, we can gain a better generalization of a given feature. In the case of education degrees, we can assign a value to each degree. For the NBA example, it is even easier since the score is already a number.

Classification

In this part, we discuss the classification problem. The goal is to understand how a machine learning algorithm works. The fundamental part of each term mentioned above is the classification methods. It does not matter whether we predict the stock market or an illness based on historical data; we always have to solve a classification problem. From a mathematical point of view, it can be described as follows:

$$D : \mathbb{R}^n \to \Omega. \tag{1.2}$$

(a) Benign skin lesion (b) Malignant skin lesion

Fig. 1.3 Example of a medical classification problem. The differences of benign (**a**) and malignant (**b**) skin lesion are shown on the dermatoscopy images. *Source* [37]

D is our set of classifiers if we have 3 ensemble classifiers (see Chap. 7). $\mathbb{R}^n$ is a n-dimensional space of features, where n is the number of features in the feature vector. Ω is the set of labels $\omega_1, ..., \omega_n$. Some classification problems can be solved with one classification method d_1, but in some cases an ensemble of classifiers D must be constructed. A label is also known as a class or a group.

Example 1 (*Skin moles*) Let us take an example. In Fig. 1.3 we can see an example of a skin mole. Dermatologists use scoring methods to establish a diagnosis. Let us take into account only two features: asymmetry (x_1) and number of colors (x_2). If we take three random cases of cancer and benignity and draw a graph, then it might look like in Fig. 1.4. The red and blue marks are our classes: benign and malignant. The green case is a new one, in which we do not yet know whether it is a benign mole or a cancer. Proven cancer cases are marked red and benign blue on the chart. In addition to how dermatologists diagnose cases, we can see that cancer cases are just moles of many different colors or high asymmetry. The benign moles are symmetric and consist of fewer colors. If we are asked to draw a straight line between malignant and benign moles, it might look like a dashed line in Fig. 1.4. For us, it is clear where the boundary between cancer and benign is, but let us assume that we have a database of 500 moles with 100 cancer, 250 benign, and 150 other disease cases. In the real world, in many cases, it is difficult to distinguish cancer from benign moles, even by experts [38]. The linear classification problem shown in Fig. 1.4 rarely occurs. Most classification problems are more complex. Let us take a look at the next example, which is a little more complex.

Example 2 (*XOR*) Exclusive OR is a commonly known logical operation. It can be written as follows: $p \otimes q = (p \vee q) \wedge \neg(p \wedge q)$. Consider XOR as a classification problem in which both logical values are our features x_1 and x_2. The result of the XOR operation will be our label (y) as shown in Table 1.1.

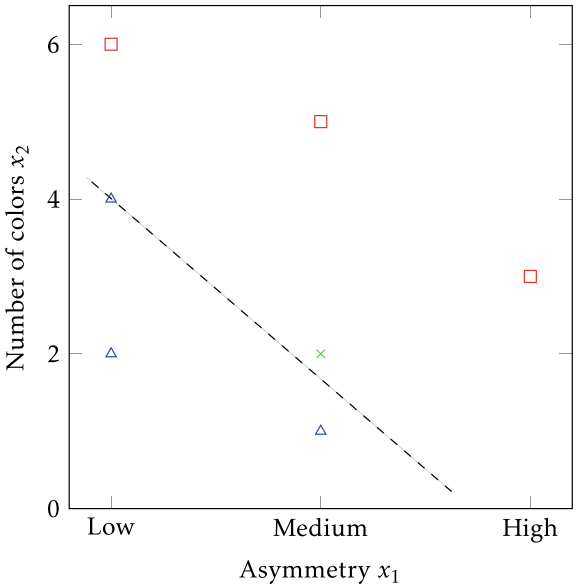


Fig. 1.4 Simple binary classification problem with two features: asymmetry (x_1) and number of colors (x_2)

Table 1.1 XOR result of two logical statements x_1 and x_2

x_1	x_2	y
1	1	0
1	0	1
0	1	1
0	0	0

We can draw a simple graph of the data shown in Table 1.1 as shown in Fig. 1.5. It is not possible to draw a straight line like we did in the previous example. We need to draw the line in the shape of an ellipse or something similar to distinguish between two classes. This means that it is a non-linear classification problem. Most real-world-based classification problems are rather non-linear. Most examples shown in this book are non-linear classification problems.

Before we move on to more difficult topics, we need to explain how a classification method works. We choose the minimum-distance classifier. It is a simple classifier that is also used in many other publications to show the big picture of training with machine learning methods [39]. There are a few terms that need to be explained here. We show a few functions that look similar but have a different name and do something else. The first function is the discriminant function. For a minimum distance classifier, it can be written as follows:

$$g_k(x) = 2m_k^T x - m_k^T m_k, \tag{1.3}$$

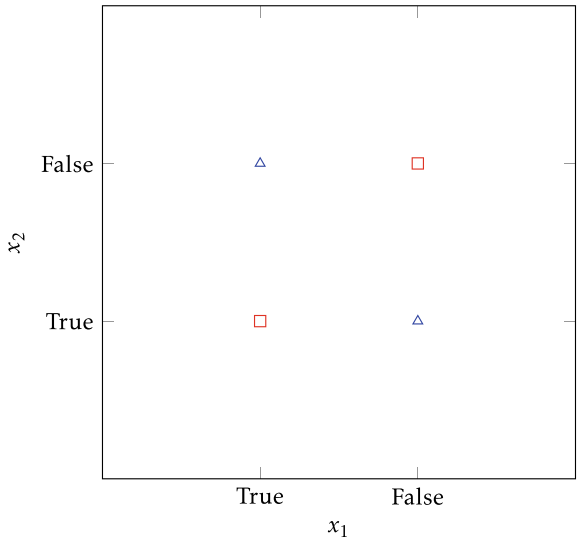


Fig. 1.5 A non-linear classification problem of logical XOR operation. The features x_1 and x_2 are two logical statements given as input to the XOR operation

where $g_k(x)$ is a function that at the end tell us which class a given x should be assigned to. It is calculated for each class k. We need to compare all the values of $g_k(x)$. We assign the class k for the highest value of $g_k(x)$. In this equation, we have m_k, which is the center of input values of a specific k class. Centers can be calculated as the average number of points of a given class:

$$m_k = \frac{\sum x}{n_k}, \tag{1.4}$$

where n_k is the number of training data sets for the label k. The function that allows us to draw a *line* between classes can be written as follows:

$$g(x) = g_i(x) - g_j(x) = 0, \tag{1.5}$$

where $g_i(x)$ is the discriminant function of each class. The function $g(x)$ is also known as a *decision function*. The line is also known as the decision surface or hyperplane. If we inject our discriminant functions into the above equation, we get our $g(x)$ function as follows:

$$g(x) = 2(m_i^T - m_j^T)x + m_j^T m_j - m_i^T m_i. \tag{1.6}$$

Example 3 (*Minimum distance classifier*) We prepared a training data set of 20 feature vectors, ten for each class. It looks like shown in Table 1.2. Our feature space looks as shown in Fig. 1.6. We don't need to calculate anything to see that it is a

Table 1.2 Minimum distance classifier training set example

x_1	x_2	y	x_1	x_2	y
−0.95	−0.80	−1	0.52	0.52	1
−0.80	−0.30	−1	0.70	0.53	1
−0.70	0.10	−1	0.74	0.91	1
−0.50	0.20	−1	0.41	0.43	1
0.10	−0.80	−1	0.45	0.33	1
−0.30	−0.30	−1	0.97	0.05	1
−0.25	−0.70	−1	0.99	0.18	1
−0.60	−0.45	−1	0.67	0.18	1
0.25	−0.80	−1	0.74	0.89	1
0.40	−0.70	−1	0.06	0.49	1

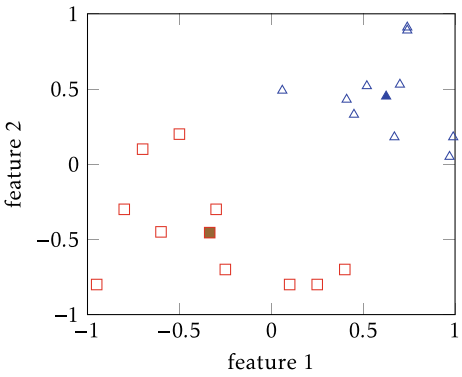


Fig. 1.6 Minimum distance example data set shown in two dimensional feature space. Centers of each label/class are filled

linear classification problem. We could easily draw a line between the red and blue points, but how does the classifier do it?

First of all, we need to calculate the center points of each class. The average of each point would be as follows:

$$m_{-1} = \left[-0.335\ -0.455\right], m_1 = \left[0.625\ 0.451\right].$$

We can implement two methods to calculate the centers of two label data sets (see the Listing 1.1). In `calculate_centers` we calculate the m_{-1} and m_1 centers. We go through the feature vectors by label and calculate the average values of x_{i1} and x_{i2} of each class.

```
def calculate_centers():
    unique_labels = np.unique(labels)
    centers = []
    for label in unique_labels:
            centers.append(dataset[labels==label].mean(axis=0))
    return centers
```

Listing 1.1 Minimum-distance classifier label center calculation

Both centers are marked with a filled square and triangle sign in Fig. 1.6. The next step is to calculate the discriminant functions for both classes. Please, keep in mind that our feature vector x can be written as a matrix of two features $\begin{bmatrix} x_1 & x_2 \end{bmatrix}$. Discriminant functions for our example are calculated as follows (Eq. 1.3):

$$g_{-1}(x) = 2 \cdot \begin{bmatrix} -0.335 & -0.455 \end{bmatrix} \cdot \begin{bmatrix} x_{i1} \\ x_{i2} \end{bmatrix} - \begin{bmatrix} -0.335 & -0.455 \end{bmatrix} \cdot \begin{bmatrix} -0.335 \\ -0.455 \end{bmatrix} =$$

$$= -0.67x_{i1} - 0.91x_{i2} - 0.31925,$$

$$g_1(x) = 2 \cdot \begin{bmatrix} 0.625 & 0.451 \end{bmatrix} \cdot \begin{bmatrix} x_{i1} \\ x_{i2} \end{bmatrix} - \begin{bmatrix} 0.625 & 0.451 \end{bmatrix} \cdot \begin{bmatrix} 0.625 \\ 0.451 \end{bmatrix} =$$

$$= 1.25x_{i1} + 0.902x_{i2} - 0.594026.$$

We used two methods to implement the discriminant function for each class. It is shown in the Listing 1.2. `calculate_discriminant_function` method is used to calculate the parameters of the discriminant functions. To obtain the value of a function, we need to use the `calculate_discriminant_function_value` method. It obtains the number of test vectors and calculates the discriminant value for a given label `label_iter`.

```
def calculate_discriminant_function():
    function_var_values=[]
    bias = []
    for center in centers:
        print(center[0])
        function_var_values.append(center[0]*2)
        bias.append(np.matmul(center,center.T))
    return function_var_values, bias
```

Listing 1.2 Minimum-distance classifier labels discriminant function values calculation

For now we have our two discriminant functions and centers calculated. Only the distinguishing function needs to be calculated. Let us put our results into Eq. 1.6 to get it:

$$g(x) = 2(\begin{bmatrix} -0.335 \\ -0.455 \end{bmatrix} - \begin{bmatrix} 0.625 \\ 0.451 \end{bmatrix}) \cdot \begin{bmatrix} x_{i1} \\ x_{i2} \end{bmatrix} + \begin{bmatrix} 0.625 \\ 0.451 \end{bmatrix} \cdot \begin{bmatrix} 0.625 & 0.451 \end{bmatrix} -$$

$$- \begin{bmatrix} -0.335 \\ -0.455 \end{bmatrix} \cdot \begin{bmatrix} -0.335 & -0.455 \end{bmatrix} = 0.$$

Finally, we get the following function:

$$g(x) = -1.92x_{i1} - 1.812x_{i2} + 0.274776 = 0.$$

The function $g(x)$ parameters calculation can be implemented as shown in the Listing 1.3. We have only one method, called `calculate_hyperplane` in which we take the centers of both labels and calculate the parameters as shown above. This

method returns two parameters, which in the current example is $[-1.92, -1.812]$ and a bias that is 0.274776. Later in this book, we explain the hyperplane and decision function in more detail, as both are commonly used as terms that distinguish between classes.

```
def calculate_hyperplane():
    hyperplanee_variables = centers[0]
    for i in range(1,len(centers)):
        hyperplane_variables=np.subtract(hyperplane_variables,centers[i])
    hyperplane_bias=np.dot(centers[len(centers)-1],centers[len(centers)
    -1].T)
    for i in reversed(range(0,len(centers)-1)):
        hyperplane_bias=np.subtract(hyperplane_bias, np.dot(centers[i],
    centers[i].T))
    return hyperplane_variables*2, hyperplane_bias

def get_hyperplane():
    denominator = np.array(hyperplane_vars).ravel()[0]
    features = np.array(hyperplane_vars).ravel()[1:]
    points = []
    print(denominator)
    for cut_point_id in range(len(cut_points_y)):
        numerator = 0
        for feature in features:
            numerator = numerator + np.dot(feature,cut_points_y[
    cut_point_id])
        points.append((numerator+hyperplane_bias)/-denominator)
    return points
```

Listing 1.3 Minimum-distance classifier discriminant function values calculation

The above function is shown in Fig. 1.7 with a dashed line.

Now, let us take two random feature vectors from testing data set: $x_{21} = (0.4, 0.1)$ and $x_{22} = (0.1, -0.4)$. To get the prediction, we need to calculate both the discriminant function values. For the first vector, we have both discriminant function values as follows:

$$g_{-1} = -0.268 - 0.091 - 0.31925 = -0.67825,$$

$$g_1 = 0.5 + 0.0902 - 0.594026 = -0.003826.$$

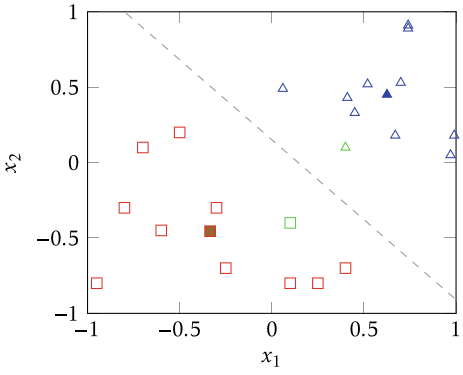


Fig. 1.7 Minimum distance example with hiperplane marked with gray dotted line and two example vectors of testing data set

The prediction is the class where the value of the discriminant function is higher. We see that for the vector x_{21} the predicted class is 1. Compare it with our second vector x_{22}:

$$g_{-1} = -0.067 + 0.364 - 0.31925 = -0.02225,$$

$$g_1 = 0.125 - 0.3608 - 0.594026 = -0.829826.$$

In this case, the predicted class is -1. A quick look at Fig. 1.7 shows that the prediction went well. In the Listing 1.4 the implementation of the prediction method is shown. It is about getting two discriminant function values and comparing both. The label is assigned with the highest value of the discriminant function. It is shown on line 9 of the Listing 1.4.

```
1  def predict():
2      prediction=[]
3      unique_labels = np.unique(labels)
4      for test_id in range(len(test_set)):
5          best = []
6          for label_id in range(len(unique_labels)):
7              best.append(np.dot(discriminant_variables[label_id],np.array
8              (test_set[test_id]))-bias_variables[label_id])
9          prediction.append(unique_labels[np.argmax(best)])
10     return prediction
```

Listing 1.4 Minimum-distance classifier prediction

So we learn our first classification method. Before we go into the next classification methods, we need to introduce several mathematical and statistical terms.

1.5 Taxonomy

In this section, we focus on the different types of machine learning methods. There are many machine learning methods, and for someone new to pattern recognition, it can be difficult to understand the relations between methods and types of methods. We have divided this section into two parts. In the beginning, we present the related work. We present the most interesting taxonomies that have already been published. The goal of this section is to show the differences between the types of machine learning methods in an efficient way. We want to cover all methods, but it is not really possible, because of the number of methods. That is why we propose a taxonomy that should be much easier to understand for someone who is new to pattern recognition. This is what the second part of this section is about.

Past works

There are some types of method that are so popular that they are repeated in each book on machine learning. So far many taxonomies have been introduced. Most are complementary to each other, but are shown from different points of view. Considering how the discriminant function approach works, we can divide the methods as shown in Fig. 1.8. This taxonomy consists of four major types: similarity-based,

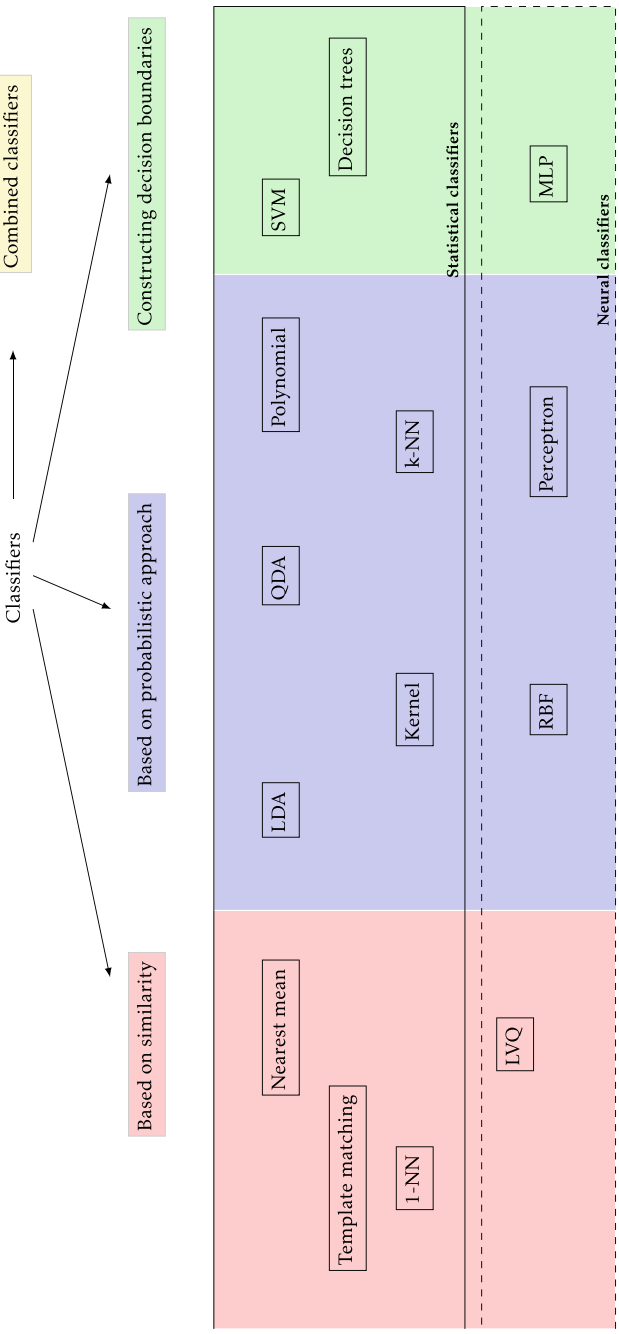


Fig. 1.8 Classification methods taxonomy proposed in [40]

probability approach, decision boundaries, and combined classifiers. We have already explained the similarity and probability theory that are used in classifier methods. Decision boundaries and combined classifiers are explained in next chapters. The methods shown in Fig. 1.8 are just a few of the entire list of classification methods. The nearest mean is also known as the nearest centroid or Rocchio classifier. Calculate the centroid of each class from the training data and assign the label where the centroid is closest to a new element from the test data set. Template matching is strongly related to image processing methods. It matches the templates defined in the images. Learning Vector Quantization (LVQ) is also known as a Kohonen network or Self-Organized Map (SOM). It is a type of neural network based on data without the knowledge of class assignment. It is the neural network that classifies the similarity of elements without prefixed templates or patterns. Linear Discriminant Analysis is also known as Fisher's classifier or Fisher's linear discriminant. As can be inferred from the name, it is a linear classifier. It is a simple classifier that can be compared to linear regression. It can also be used for dimensionality reduction. The dimensionality problem is explained in Chap. 4. QDA is known as Quadratic Discriminant Analysis or Quadratic Classifier. It is similar to LDA, but instead of linear separation, it enables us to separate non-linear. Kernel-based methods, such as the Parzen kernel, move a problem to a higher dimension, where it can be solved easier or solved at all. The SVM method that use a kernel is explained in Chap. 4. kNN is a more complex version of the 1-NN classifier. In this case, we take k nearest neighborhood elements k to assign the proper label. It is explained together with the 1-NN classifier in Chap. 4. The perceptron is the simplest neural network-based classifier. It is just one neuron that is a linear classifier but works totally differently from LDA. It is explained in Chap. 8. RBF is a more complex network based on the radial basis function. It is commonly used in pattern recognition. RBF in the sense of a neural network type is a complex network where the neurons activation functions are Radial Basis Functions. It is explained in Chap. 8. MLP stands for Multilayer Perceptron and is a network that consists of many layers of perceptron neurons. It is much more complex compared to a single perceptron. Using MLP, we can solve complex classification problems. It is explained in Chap. 8. SVM stands for Support Vector Machine and is a method in which the goal is to prepare a horizontal plane that distinguishes between elements of different labels. The hiperplane is prepared in such a way that the biggest margins are between elements of different labels, and the hiperplane are found. SVM uses Lagrangian multipliers to make the calculations of the hiperplane easier. SVM is commonly used in image-based pattern recognition. It is explained in Chap. 6. The decision tree is a simple classifier that is also a very fast method of classifying elements. Building the tree is much more complex and time consuming than the classification part. It is explained in Chap. 4. The last classification method, or rather a group of classification methods, is a combined method. It is a group that combines several classification methods or many instances of the same method. Using more than just one classification method instance can give better results. An example of a combined classifier is random forests that combine instances of decision trees. More information on combined classification methods can be found in Chap. 7.

Another taxonomy, or rather a simple division, into groups, is proposed in [41]. The authors divided machine learning methods in a different way compared to [40] and divided the methods into four main groups:

1. template matching,
2. statistical classification,
3. syntactic or structural matching,
4. neural networks.

This is a very high-level taxonomy. We have already explained the template matching methods. Many of the methods presented in this book are statistical classification methods. Regression-based methods or the kNN method are statistical methods. Syntactic and structural matching methods are based on graphs or grammars. Structural methods are decision trees. We have dedicated a chapter to neural networks. In Fig. 1.9 we present another taxonomy. In [34] the methods are divided into two main groups according to the approximation method. The first group is about methods that are based on class-conditional probability density function that probability priors. A probability density function is a function that defines the likelihood (probability) of an outcome for a given case. To simplify things, these methods are based on probability theory. The second is based on boundaries and discriminant functions. Some methods presented in Fig. 1.9 are already shown in Fig. 1.8. LDC or QDC are different names for LDA and QDA. Methods like k-NN, Parzen kernel method were also shown. In histograms, we can merge two histograms of each class into a higher-dimensional space and use some probability methods to distinguish the new elements. The mixture discrimination method is also known as the Gaussian mixture models method. It is a probabilistic method similar to the k-means method, which is described in Chap. 4. Logistic discrimination is a different name for logistic regression. Tree classifiers are also known as decision trees. In this taxonomy, both methods are considered separately. Rosenblatt's perceptron is a different name for the perceptron. In this case, we add the name of the inventor. The generalized linear discriminant is a different approach of LDA [42] that is used in high-dimensional cases. A special case of LDA is the piecewise linear discriminant classifier that consists of a set of linear functions that together distinguish new elements [43]. Different approaches to the taxonomy of machine learning methods can also be found in [24, 44, 45].

Proposed taxonomy

The taxonomy of machine learning methods can be divided taking into account a few levels: input data types, architecture depth, and the main concept of the method. Based on the input data, there are three major groups of machine learning methods: supervised, unsupervised, and reinforcement. All methods belong to one of these. Supervised methods are those where the output is known during training. A medical diagnostic case serves as an example of such a method, in which, for training sake,

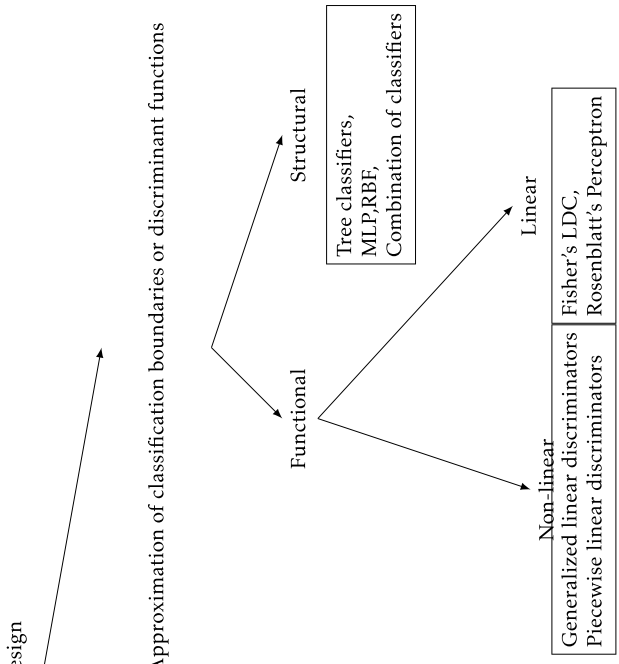


Fig. 1.9 Classification methods taxonomy proposed in [34]

we already know if a given input, like an image, is the image of a cancer or not. Unsupervised is the opposite approach. This means that we do not have information if that it is cancer or not. This approach can be useful in any anomaly detection, but there are also other cases where it brings benefits. There are also methods known as reinforcement learning methods in which we have labels, but use it only to know if the method is doing well and uses the reward and penalty mechanism to achieve better accuracy. Such methods usually go through the same path many times. An example of a reinforcement learning approach is OpenAI Gym [46] where the bot/method plays a game, learns, and repeats the game until the game ends. We explain the unsupervised and supervised methods in the next chapters. On the basis of the depth of the architecture, we divide the methods into shallow and deep learning methods. Deep learning is used by some non-researchers as an alternative to machine learning. This is not true, as deep learning is a group of methods that are neural networks consisting of many layers. That is why they are called deep. Deep learning has become a buzzword in recent years. The last level of division is the concept of machine learning method. Some methods are based on the probability theory; others are based on decision trees, neural networks, etc. Chapters 4–8 are about different method groups based on the general concept. There are also other types of methods, such as evolutionary or combined learning. Evolutionary learning is based on genetic algorithms and combined takes more than one of the same or different methods together. It is difficult to draw a taxonomy with all the available methods. There are more than a hundred methods available. In Fig. 1.10 we choose a few methods and divide them according to the mentioned levels. At the top, we divided them into shallow and deep methods. On the left, we mark only neural networks as the biggest group, and recently the most popular. On the right, we marked the supervised and unsupervised methods. In the middle, we mark a few methods that are explained later in this book, but, as mentioned before, there are plenty of other methods that are similar.

1.6 Quality Metrics of Classification Methods

In this section, we focus on how we can measure the quality of the classification method. Quality can be understood in several ways. That is why we have divided this section into three parts. In the first part, we describe common problems that appear when learning a classification method. It also includes some practical tips for avoiding common mistakes before using machine learning methods with your data. The second part discusses approaches for handling test, training, and validation data sets. We have already introduced this topic at the beginning of this chapter. In this section, we explain the most popular approaches for data set division in each group. The third part contains the most popular quality measurement methods. In this part, we skip the training data, work on the test data set, and compare it with the predicted output. An obvious quality measure is the accuracy of the success rate, but there are

Fig. 1.10 Machine learning taxonomy

more sophisticated methods to measure quality, such as the ROC curve, F_1 score, or some other types of error rates, which are explained in this section.

1.6.1 Training Phase Challenges

During training, we can face two commonly known issues. The first is related to the number of features. It is hard to say what the best number of features to use, as it depends on the problem that needs to be solved. There are several methods that can measure the importance of each feature, so we can choose only the most important. Important means here how a feature affects the accuracy. It is important to reduce the number of features to the minimum that affects the result, as a higher number of features makes the computation more complex. For two features, we consider a two-dimensional classification problem. For three, it would be a three-dimensional problem. The more features there are, the larger the problem we need to solve. It is known as the dimensionality problem [47–50]. We have several commonly used feature selection methods that can be divided into four main groups [51]:

- ranking,
- wrapper,
- hybrid,
- embedded.

Ranking methods are also known as filter methods. In this type of feature selection method, we do not use a classification method. The goal is to establish a ranking of the features. Pearson correlation method is one of such a method. We will explain it in detail in the next chapter. We measure the correlation between the features. We can skip one of the features if we know that it has a linear correlation with some other feature or features. This means that adding such a feature does not add any value to the final result. The name of the second group of methods works that the feature selection method wraps the classification method and based on the accuracy selects the features. It takes a set of features, performs the classification, gets the accuracy, and compares it with the accuracy obtained from a data set of a different set of characteristics. If we have many features, it can be time consuming to use this kind of method. These methods start with an empty set of features. It takes one feature in each iteration and stops as soon as adding any feature from what is left does not increase the accuracy. Hybrid methods are a mixture of classification and wrapping methods. In this type of methods, we use the ranking part first, and then the wrapper part. It makes a huge difference if we have many features. Reduce the number of features by ranking those before we move on to the more time-consuming wrapper part. The last type is included in the classification method. This means that it is not used outside the machine learning method but is part of it. A frequently used approach is the use of genetic algorithms within the classifier.

By using a method of mentioned groups, we can reduce the number of features to the absolute minimum that has an influence on the final result. In addition, adding more and more features can even reduce accuracy. As shown in Fig. 1.11, the error rate can increase while increasing the number of features at some points.

Overfitting is another common problem that can occur while training a classification method. It is about under-training and over-training. The goal of the training part is to generalize. It means that we would like to have a method that gives high accuracy for any data of a given problem. Under-training occurs when we do not train the method enough. We have not prepared enough training data, and the method does not have enough data to train on. Therefore, the method gives lower accuracy, because it assigns labels incorrectly. The same result we get with over-training, but the reason is slightly different. We train the method with too many data. Especially when we have a lot of feature values of the same label that are close to each other. The algorithm adjusts very well to classification for given data and does not generalize the solution for a given problem. An example error rate to training data amount comparison is shown in Fig. 1.12. It is important to do so, remember that more data does not mean better. It is important to have a set of features that accurately represents a given problem. As we have already mentioned at the beginning of this chapter, the goal of a

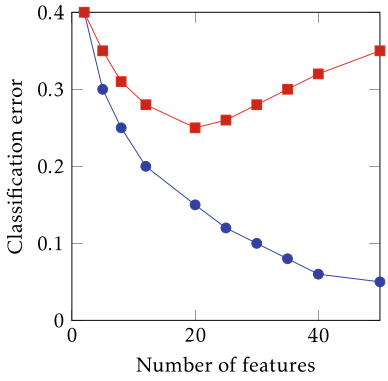


Fig. 1.11 Classification error rates depends on the number of features

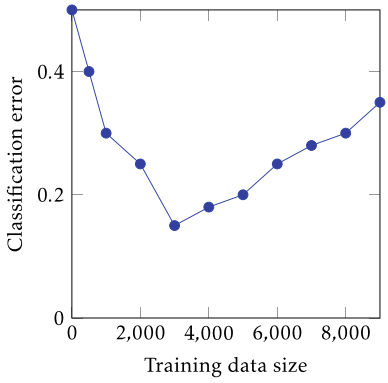


Fig. 1.12 Overfitting problem

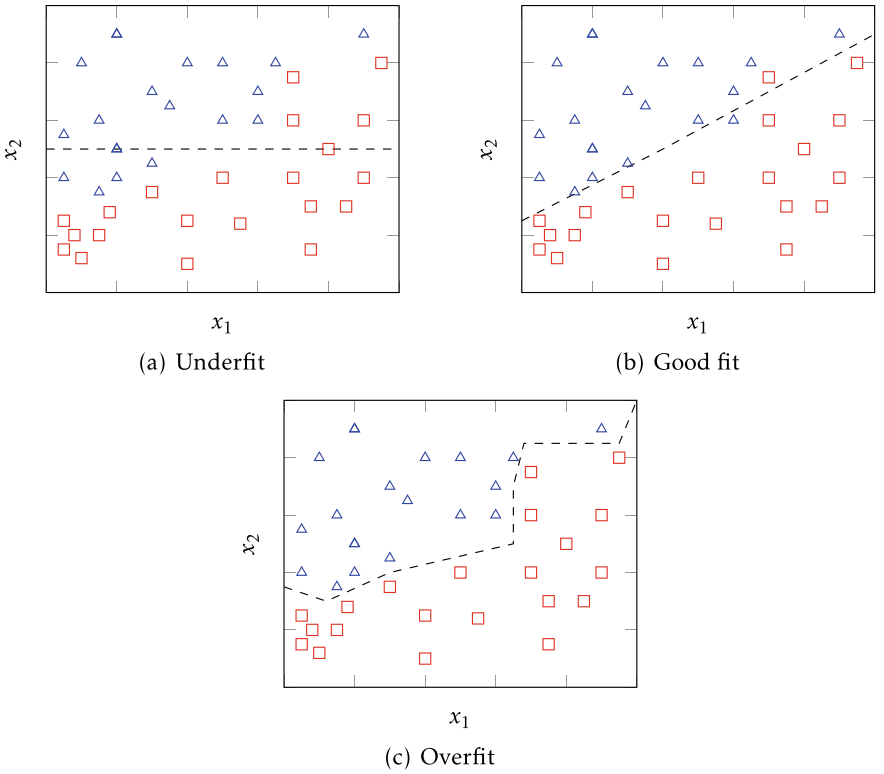


Fig. 1.13 Generalization problem of two training data sets

classifier is to generalize the best possible. We cannot be sure that a given case is the best possible generalization, but there are some cases where it is clear that we do it wrong. In Fig. 1.13 the classification using two data sets is shown. In Fig. 1.13b, a data set that represents the classification problem well is shown. In case of Fig. 1.13a the classifier is not complex enough to classify properly. The last example (Fig. 1.13c) is a classifier that works very well for given data, but will not work well for new data. It means it is just too adjusted to the training data and does not generalize enough.

1.6.2 Data Sets Preparation Approaches

One of the common problems that each data scientist has is to divide the data set into training and testing data sets. To understand the following equations, we need to introduce new designations. Let $\mathcal{L}_n$ be our training data set of size n, T_m our testing data set of size m, M_e the number of misclassified cases, $\mathcal{I}$ a function that returns 1 if there is a match between the predicted and the true value and $e(d)$ the error rate of

classifier d. We also use the testing set X and the labels set Y that we have already explained. We can write the error rate as follows:

$$e(d) = \frac{M_e}{m}. \tag{1.7}$$

The error rate can be calculated differently depending on the method of data set preparation method used. Few commonly used approaches are used to handle training, testing, and validation data sets.

- resubstitution—R-method,
- hold-out—H-method,
- cross-validation—π-method,
- bootstrap.

The first method is very simple. We have the same data set for training and testing. It is not the best solution if we consider having a solid classifier. The error rate can be written as follows:

$$e_R(d) = \frac{1}{n}\sum_{j=1}^{n} \mathcal{I}(d(X_j; \mathcal{L}_n) \neq Y_j). \tag{1.8}$$

This means that we calculate the error rate for each element j of our training data set and add 1 for each well-predicted case. We need to divide it by n, which is the number of elements in the training data set. The approach is generally the same as in the case of Eq. 1.7.

The second method involves dividing a data set into two data sets. It can be divided into half or other proportions. One set is our training data set, and the other training data set. We can swap these sets and calculate the average of both sets. The error rate can be calculated as follows:

$$e_\tau(\hat{d}) = \frac{1}{m}\sum_{j=1}^{m} I(\hat{d}(X_j^t; \mathcal{L}_n \neq Y_j^t). \tag{1.9}$$

Compared to the resubstitution method, it uses only the testing data set. Cross-validation is the most common approach. It is also sometimes called a rotation method. We need to divide the data set into k subsets. The elements in each set are chosen at random. One of these sets is taken as a test set, and the other sets are merged into the training set. It should be repeated k times for each k subset. The error rate can be calculated as follows:

$$e_{CV}(d) = \frac{1}{n}\sum_{j=1}^{n} I(\hat{d}(X_j; \mathcal{L}_n^{(-j)} \neq Y_j). \tag{1.10}$$

A special case is where $k = m$. This means that we have subsets, each consisting of just one element. This approach is known as the leave-one-out or U method. The Bootstrap method can be considered as an extension of resubstitution. The goal is to generate multiple sets from the main set by random selection. We use the resubstitution method on each set and calculate the average error at the end:

$$e_B(d) = \frac{1}{B} \sum_{b=1}^{B} \frac{\sum_{j=1}^{n} \mathcal{I}(Z_j \notin \mathcal{L}_n^{\star b}) \mathcal{I}(d(X_j; \mathcal{L}_n^{\star b}) \neq Y_j)}{\sum_{j=1}^{n} (Z_j \notin \mathcal{L}_n^{\star b})}, \tag{1.11}$$

where B is the number of boostrap sets and Z_j is a set of currently observed object or objects $Z_j = (X_j, Y_j)$.

1.6.3 Output Quality Metrics

There are several metrics to show the quality of our classification model:

- ROC which stands for Receiver Operating Characteristic curve,
- AUC—Area Under Curve,
- F_1 score,
- Precision,
- Recall.

To explain each metric in detail, we use an example again.

Example 4 (*Diagnosis*) If a patient had asthma, a doctor can take a couple of actions to check if it is actually asthma. In addition to the action in the end, the doctor must make a decision whether the patient has asthma or not. The doctor gets the diagnosis mainly right, but it can be that the diagnosis is wrong because it is only a prediction. In Table 1.3 we present all possibilities. Positive means that the diagnosis is asthma. Negative means that there is no asthma diagnosed by the physicians. True and False are used to indicate whether the decision is correct. It could be that the doctor said it was asthma, but it was actually lung cancer or another disease.

The best diagnoses are when we get a True Positive (TP) or True Negative (TN). In two other options, the diagnosis is incorrect. If we consider cancer diagnosis, a false negative (TN) scenario would be the worst scenario.

Table 1.3 Possible scenarios of doctor's diagnosis

		True condition	
		Condition positive	Condition negative
Predicted	Positive	True positive (TP)	False positive (FP)
	Negative	False negative (FN)	True negative (TN)

The decision possibilities mentioned in Table 1.3 bring us to quality metrics. The most common metric is accuracy. It can be calculated as follows:

$$\text{ACC} = \frac{\#\text{TP} + \#\text{TN}}{\#\text{TP} + \#\text{TN} + \#\text{FP} + \#\text{FN}}. \tag{1.12}$$

The first one that we describe is called True Positive Rate (TPR). It can be calculated as follows:

$$\text{TPR} = \frac{\#\text{TP}}{\#\text{TP} + \#\text{FN}}. \tag{1.13}$$

TPR is also called sensitivity or recall, and is a measure of good predictions within a set of cases. A higher rate means a measure of good asthma predictions in Example 4. By #TP, #FP we mean the number of True Positive and False Positive cases, where # stands for the number of decisions. The opposite is specificity. It is also called TNR, which stands for True Negative Rate. It can be calculated as follows:

$$\text{TNR} = \frac{\#\text{TN}}{\#\text{TN} + \#\text{FP}}. \tag{1.14}$$

It is a measure that says how well we are at predicting negative scenarios. In Example 4 it would say how good we are at diagnosing that a patient does not have asthma. Another important metric is precision, which is also known as Positive Predictive Value (PPV):

$$\text{PPV} = \frac{\#\text{TP}}{\#\text{TP} + \#\text{FP}}. \tag{1.15}$$

It is a ratio of positive cases that were predicted well to all positive cases, even those that are not well predicted. The opposite is the Negative Predictive Value:

$$\text{NPV} = \frac{\text{TN}}{\text{TN} + \text{FN}}. \tag{1.16}$$

We can also calculate the False Positive Rate metric known as fallout. It is about how bad we are at predicting positive cases:

$$\text{FPR} = 1 - \text{TNR}. \tag{1.17}$$

The opposite of FPR is the False Negative Rate:

$$\text{FNR} = 1 - \text{TPR}. \tag{1.18}$$

Another popular metric is called the F_1 score and is a weighted accuracy measure. It takes PPV and TPR to calculate the score:

$$F_1 = 2\frac{\text{PPV} \cdot \text{TPR}}{\text{TPR} + \text{PPV}}. \tag{1.19}$$

The value F_1 as in the case of all previous metrics between 1 and 0, where 1 is the best. An interesting measure is the Matthews correlation coefficient measure, which is about the correlation between observed and predicted values. The value of MCC is between -1 and 1. If we have a perfect classifier, we get MCC = 1. A random classifier is when we have MCC = 0 and a totally bad classifier if we have MCC = -1. This measure can be calculated as follows:

$$\text{MCC} = \frac{\#\text{TP} \cdot \#\text{TN} - \#\text{FP} \cdot \#\text{FN}}{\sqrt{(\#\text{TP} + \#\text{FP})(\#\text{TP} + \#\text{FN})(\#\text{TN} + \#\text{FP})(\#\text{TN} + \#\text{FN})}}. \tag{1.20}$$

Example 5 (*Lung cancer diagnosis*) We use the example of lung cancer diagnosis to explain how to calculate the quality metrics we have just described. In this example, we have three doctors: Dr. Smith, Dr. Williamson and Dr. Simpson. Each one is an oncologist and set diagnoses on daily basis. To compare how good each doctor is at making diagnosis of lung cancer we have a data set of twenty patients. Each doctor has a different set of patients. In Table 1.4 the data sets of patients and given diagnosis are shown. Based on the data given in Table 1.4 we can calculate the results of TN, TP, FN and FP. The results are shown in Table 1.5. It looks like Dr. Simpson gives the

Table 1.4 Three doctors' prediction compared to true condition of lung cancer

Dr. Smith		Dr. Williamson		Dr. Simpson	
Condition	Diagnosis	Condition	Diagnosis	Condition	Diagnosis
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	−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	−1	1	−1	−1

Table 1.5 Example 5 basic quality metrics

Quality metric	Dr. Smith	Dr. Williamson	Dr. Simpson
TN	5	9	10
TP	8	9	10
FN	5	1	0
FP	2	1	0

best diagnosis. We can calculate the remaining quality metrics for each physician. Let us start with Dr. Smith:

$$\text{ACC}_{\text{Smith}} = \frac{8+5}{20} = 0.65,$$

$$\text{TPR}_{\text{Smith}} = \frac{8}{13} \approx 0.615,$$

$$\text{TNR}_{\text{Smith}} = \frac{5}{5+2} \approx 0.714,$$

$$\text{PPV}_{\text{Smith}} = \frac{8}{8+2} = 0.8,$$

$$\text{NPV}_{\text{Smith}} = \frac{5}{5+5} = 0.5,$$

$$\text{FPR}_{\text{Smith}} = 1 - 0.714 = 0.286,$$

$$F_1^{\text{Smith}} = 2\frac{0.8 \cdot 0.615}{0.615 + 0.8} = 2\frac{0.492}{1.415} \approx 0.695,$$

$$\begin{aligned}\text{MCC}_{\text{Smith}} &= \frac{8 \cdot 5 - 2 \cdot 5}{\sqrt{(8+2)(8+5)(5+2)(5+5)}} = \frac{30}{\sqrt{10 \cdot 13 \cdot 7 \cdot 10}} = \frac{30}{\sqrt{9100}} \\ &= \frac{30}{95.4} \approx 0.314.\end{aligned}$$

We can now compare the values that we got for Dr. Smith with Dr. Williamson and Dr. Simpson. The results are presented in Table 1.6. The results indicate the thought we had in the first place. It looks like Dr. Simpson makes the best prediction of lung cancer. The accuracy and Matthews correlation coefficient metrics are at a level of 100%. Dr. Smith is almost as good, but as indicated by FNR and FPR, has some misclassified cases. The worst predictions are given by Dr. Smith. The accuracy is only 65%, so it is a bit more than a random guess. The MCC metric shows that it is even better. To calculate quality metrics, we can use Python and some simple arithmetic operations. An example is shown in Listing 1.5.

Table 1.6 Quality metrics calculated for example presented in Table 1.4

Quality metric	Dr. Smith	Dr. Williamson	Dr. Simpson
ACC	0.65	0.9	1.0
TPR	0.615	0.9	1.0
TNR	0.714	0.9	1.0
FNR	0.385	0.099	1.0
FPR	0.286	0.099	0.0
PPV	0.8	0.9	1.0
NPV	0.5	0.9	0.0
F_1	0.695	0.9	1.0
MCC	0.314	0.8	1.0

```
def calculate_quality_metrics(self):
    tn = 0
    tp = 0
    fn = 0
    fp = 0
    for i in xrange(len(self.data_set)):
        if self.data_set[i] > 0:
            if self.data_set[i] == self.predicted_set[i]:
                tp = tp + 1
            else:
                fp = fp + 1
        else:
            if self.data_set[i] == self.predicted_set[i]:
                tn = tn + 1
            else:
                fn = fn + 1
    acc = ((tp + tn) * 1.0) / ((tp + tn + fp + fn) * 1.0)
    tpr = tp * 1.0 / (tp + fn) * 1.0
    tnr = tn * 1.0 / (tn + fp) * 1.0
    ppv = tp / (tp + fp) * 1.0
    npv = tn / (tn + fn) * 1.0
    fpr = 1.0 - tnr
    fnr = 1.0 - tpr
    f1 = 2 * (ppv * tpr * 1.0 / (tpr + ppv * 1.0))
    mcc = (tp * tn - fp * fn) / (sqrt((tp + fp) * (tp + fn) * (tn + fp) *
     (tn + fn)) * 1.0)
    return [acc, tpr, tnr, ppv, npv, fpr, fnr, mcc]
```

Listing 1.5 Quality metrics implementation

ROC stands for Receiver Operating Characteristic [52]. It is a curve that shows the performance of a classification method. A few examples of the ROC curve are presented in Fig. 1.14. To simplify, let us assume that we consider a binary classifier. To draw an ROC curve, we need to calculate two metrics: FPR and TPR, but the curve says much more than just the relation between those two metrics. If both metrics are high, we know that other metrics that we mentioned are high as well. In Fig. 1.14 three simplified ROC curves are presented. The black-marked line is if we have a classifier that classifies with an accuracy of 50%. The accuracy on this level cannot be considered high. Such a classifier can be replaced with a coin throwing simulator and classify based on what we get. The blue-marked curve is for a classifier with good

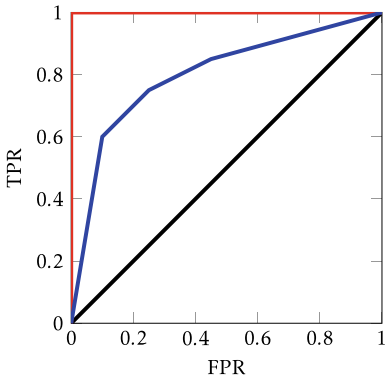


Fig. 1.14 Receiver operating characteristic curve

accuracy. This means that it classifies the current problem well. The best classifier is when we get 100% precision. Such a ROC curve is marked red.

Fun fact: The Receiver Operating Characteristic (ROC) curve was developed during World War II by electrical and radar engineers to improve the detection of enemy objects on battlefields. The term "ROC" originates from this application, reflecting its initial use in evaluating radar receiver performance.[a]

[a] https://en.wikipedia.org/wiki/Receiver_operating_characteristic.

An implementation of the calculation of the ROC curve is presented in the Listing 1.6. To draw the ROC curve we need to get at least a few points and connect each to get the curve. To get those points, we need to calculate the TPR and FPR metrics at a given cut-off point. The cutoff point here means the point where we measure those metrics. In many cases, the cut-off points are chosen on the basis of the classification problem. For lung cancer, it could be the patient's age. It means that we need to sort the testing data set by age and set the cut-off points as the age changes. In our example, we choose it to be 5, 8, and 10 cases of each label for Dr. Smith and 4, 7 and 10 for Dr. Williamson. In the case of Dr. Simpson, the cut points do not matter as it is the perfect classifier case. We can choose other cut-off points or even calculate the FPR and TPR metric at each test data set element, but this would not change the ROC and AUC much. An example of the implementation of the calculation of the ROC curve is shown in Listing 1.6.

```
def calculate_metric_at_cutpoint(self, cutpoint, label):
    cutpoint_value = 0
    value = 0
    for i in xrange(len(self.data_set)):
        if self.data_set[i] == label:
            cutpoint_value = cutpoint_value + 1
```

```
            if self.data_set[i] == self.predicted_set[i]:
                value = value + 1
        if cutpoint_value == cutpoint:
            break
    if label > 0:
        values = value * 1.0 / cutpoint * 1.0
    else:
        values = 1 - (value * 1.0 / cutpoint * 1.0)
    return values


def calculate_metric(self, cutpoints, label):
    values = [0.0]
    for i in xrange(len(cutpoints)):
        values.append(self.calculate_metric_at_cutpoint(cutpoints[i],
     label))
    return values


def calculate_roc_curve_values(self, cutpoints):
    self.check_sets_size()
    tpr_vector = self.calculate_metric(cutpoints, 1)
    fpr_vector = self.calculate_metric(cutpoints, -1)
    tpr_vector.append(1.0)
    fpr_vector.append(1.0)
    return [tpr_vector, fpr_vector]
```

Listing 1.6 ROC curve calculation implementation in Python

The code consists of three methods: main method `calculate_roc_curve_values`, `calculate_metric` and `calculate_metric_at_cutpoint`. The first method executes the second method to get the tpr and fpr values at the cut points. Method `calculate_metric` iterated through each cut point and runs for each. The third method that calculated the TPR and FPR value. The cut-point TPR and FPR values are used to obtain the ROC curve.

Example 6 (*Lung cancer diagnosis II*) We can take the previous example to explain the AUC metric. To calculate it, we need the TPR and FPR metrics for each cut point. The area under the curve is part of the ROC curve and is just the surface area under the curve. For the red-marked ROC curve shown in Fig. 1.14 the surface area is 1.0. The AUC value shows the quality of the classification method. The value can vary from 0 to 1, but each value that is 0.5 or less is about a classifier that does not classify well at all. A better explanation of the relationship between the AUC value and the quality of the classification method is shown in Table 1.7. For the previous example we get the ROC curves as it is shown in Fig. 1.15. Compared to ROC curves shown in Fig. 1.14, we see that the best predictions are given by Dr. Simpson and worst by Dr. Smith. In Python, we can use the method `trapz` from package `numpy` to calculate the surface under the curve. An example of usage is shown in Listing 1.7.

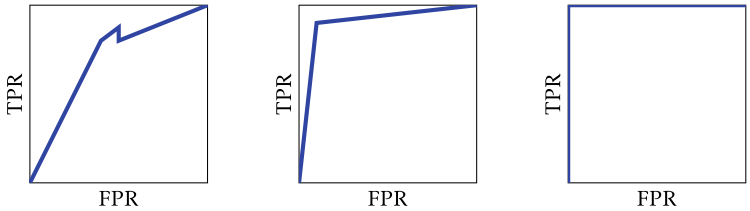
```
from numpy import trapz

def calculate_auc(self,tpr,fpr):
    return trapz(tpr,fpr)
```

Listing 1.7 AUC calculation implementation in Python

AUC values for Example 5 are shown in Table 1.8. The AUC values can vary in some cases depending on the cut-off points we choose.

Table 1.7 Area under curve value and quality of a classification method relation

Value	Classifier quality
1.0	Perfect
0.99–0.9	Excellent
0.89–0.8	Very good
0.79–0.7	Good
0.69–0.51	Poor
0.5	Worthless



(a) Dr Smith's prediction (b) Dr Kowalski's prediction (c) Dr Simpson's prediction

Fig. 1.15 ROC curves of three example predictions given in Example 6

Table 1.8 AUC values for Example 5

	Dr. Smith	Dr. Williamson	Dr. Simpson
AUC	0.69375	0.95	1.0

For Further Reading

1. Müller AC, Guido S (2016) Introduction to machine learning with Python: a guide for data scientists. O'Reilly
2. Marsland S (2014) Machine learning: an algorithmic perspective, 2nd edn. Machine learning and pattern recognition. Chapman & Hall/CRC (2014)
3. Pinheiro CR, Patetta M (2021) Introduction to statistical and machine learning methods for data science. SAS Institute
4. Huyen C (2022) Designing machine learning systems. O'Reilly
5. Serrano L (2021) Grokking machine learning. Manning Publishing

References

1. Ammanath B, Ng A, Luca M, Ghosh B (2023) The year in tech, 2023: the insights you need from Harvard Business Review. Harvard Business Review
2. Watanabe S (1985) Pattern recognition: human and mechanical. Wiley
3. Gutierrez DD (2015) Machine learning and data science: an introduction to statistical learning methods with R. Technics Publications
4. Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: machine learning in Python. J Mach Learn Res 12:2825–2830
5. Bastien F, Lamblin P, Pascanu R, Bergstra J, Goodfellow I, Bergeron A, Bouchard N, Warde-Farley D, Bengio Y (2012) Theano: new features and speed improvements. CoRR 11
6. Witten IH, Frank E, Hall MA, Pal CJ (2016) Data mining: practical machine learning tools and techniques, 4th edn. Morgan Kaufmann Publishers Inc., San Francisco, CA
7. Sonnenburg S, Rätsch G, Henschel S, Widmer C, Behr J, Zien A, de Bona F, Binder A, Gehl C, Franc V (2010) The shogun machine learning toolbox. J Mach Learn Res 11:1799–1802
8. Jia Y, Shelhamer E, Donahue J, Karayev S, Long J, Girshick R, Guadarrama S, Darrell T (2014) Caffe: convolutional architecture for fast feature embedding. arXiv:1408.5093
9. Meng X, Bradley JK, Yavuz B, Sparks ER, Venkataraman S, Liu D, Freeman J, Tsai DB, Amde M, Owen S, Xin D, Xin R, Franklin MJ, Zadeh R, Zaharia M, Talwalkar A (2015) MLlib: machine learning in apache spark. CoRR, abs/1505.06807
10. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ, Ghodsi A, Gonzalez J, Shenker S, Stoica I (2016) Apache spark: a unified engine for big data processing. Commun ACM 59(11):56–65
11. Bifet A, Holmes G, Kirkby R, Pfahringer B (2010) MOA: massive online analysis. J Mach Learn Res 11:1601–1604
12. Hackeling G (2014) Mastering machine learning with scikit-learn. Packt
13. McClure N (2017) TensorFlow machine learning cookbook. Packt
14. Tok WH, Fontama V, Barga R (2015) Predictive analytics with Microsoft Azure machine learning, 2nd edn. Apress
15. Cook D (2016) Practical machine learning with H2O. O'Reilly
16. Beyeler M (2017) Machine learning for OpenCV. Packt
17. Géron A (2017) Hands-on machine learning with scikit-learn and TensorFlow. O'Reilly
18. Mei S, Hall B, Rajendran M, Amirghodsi S (2017) Apache Spark 2.x machine learning cookbook. Packt
19. Pentreath N (2015) Machine learning with Spark—tackle big data with powerful Spark machine learning algorithms. Packt
20. Hackeling G (2017) Mastering machine learning with scikit-learn, 2nd edn. Packt
21. Kim P (2017) MATLAB deep learning: with machine learning, neural networks and artificial intelligence. Packt
22. Nadine—social robot. http://imi.ntu.edu.sg/IMIResearch/Research_Areas/Nadine/Pages/default.aspx. Accessed 06 Feb 2016
23. Turing AM (1995) Computers and thought. Chapter Computing machinery and intelligence. MIT Press, Cambridge, MA, pp 11–35
24. Gollapudi S (2016) Practical machine learning. Packt Publishing
25. Delgado MF, Cernadas E, Barro S (2014) Do we need hundreds of classifiers to solve real world classification problems? J Mach Learn Res 15:3133–3181
26. VanderPlas J (2016) Python data science handbook. O'Reilly Media
27. Azarmi B (2016) Scalable big data architecture: a practitioner's guide to choosing relevant big data architecture. Apress
28. Kitchin R, Lauriault TP (2014) Small data, data infrastructures and big data. In: The programmable city working paper, vol 1
29. Hoffman S (2015) Apache flume: distributed log collection for Hadoop, 2nd edn. Packt Publishing

30. Ding S, Zhu H, Jia W, Su C (2012) A survey on feature extraction for pattern recognition. Artif Intell Rev 37:169–180
31. Aggarwal CC (2014) Data classification. Chapman and Hall/CRC
32. Trunk GV (1979) A problem of dimensionality: a simple example. IEEE Trans Pattern Anal Mach Intell 3:306–307
33. Bowles M (2015) Machine learning in python: essential techniques for predictive analysis. Wiley
34. Kuncheva LI (2014) Combining pattern classifiers: methods and algorithms. Wiley
35. Morrow LM (2019) Literacy development in the early years: helping children read and write, 9th edn. Pearson Education, Boston, États-Unis
36. Emery JD, Hunter J, Hall PN, Hunter J, Hall PN, Watson AJ, Moncrieff M, Walter FM (2010) Accuracy of SIAscopy for pigmented skin lesions encountered in primary care: development and validation of a new diagnostic algorithm. BMC Dermatol 10
37. Przystalski K, Ogorzałek MJ (2017) Multispectral skin patterns analysis using fractal methods. Expert Syst Appl 88:318–326
38. Menzies S, Bischof L, Talbo H et al (2005) The performance of solarscan. An automated dermoscopy image analysis instrument for the diagnosis of primary melanoma. 141:1388–1396
39. Stapor K (2011) Metody klasyfikacji obiektów w wizji komputerowej (in Polish). PWN
40. Dičiūnas V (2002) Generalization performance of statistical and neural classifiers. PhD thesis, Vilnius University, Vilnius, Lithuania
41. Jain AK, Duin RP, Mao J (2000) Statistical pattern recognition: a review. IEEE Trans Pattern Anal Mach Intell 22:4–37
42. Ji S, Ye Y (2008) Generalized linear discriminant analysis: a unified framework and efficient model selection. IEEE Trans Neural Netw 19:1768–1782
43. Chang C-L (1973) Pattern recognition by piecewise linear discriminant functions. IEEE Trans Comput 22(9):859–862
44. Holmström L, Koistinen P, Laaksonen J, Oja E (1997) Neural and statistical classifiers—taxonomy and two case studies. IEEE Trans Neural Netw 8:5–17
45. Lippmann RP (1991) A critical overview of neural network pattern classifiers. In: Proceedings of the IEEE workshop on neural for signal processing, vol 2, pp 266–275
46. Brockman G, Cheung V, Pettersson L, Schneider J, Schulman J, Tang J, Zaremba W. Openai gym
47. Jain AK, Chandrasekaran B (1982) Dimensionality and sample size considerations in pattern recognition practice. Handb Stat 2:835–855
48. Raudys SJ, Pikelis V (1980) On dimensionality, sample size, classification error and complexity of classification algorithms in pattern recognition. IEEE Trans Pattern Anal Mach Intell 2:243–251
49. Raudys SJ, Jain AK (1991) Small sample size effects in statistical pattern recognition: recommendations for practitioners. IEEE Trans Pattern Anal Mach Intell 13:252–264
50. Trunk GV (1979) A problem of dimensionality: a simple example. IEEE Trans Pattern Anal Mach Intell 1:306–307
51. Chmielnicki W (2012) Efektywne metody selekcji cech i rozwiazywania problemu wieloklasowego w nadzorowanej klasyfikacji danych. PhD thesis, Polish Academy of Sciences
52. Metz CE (1978) Basic principles of ROC analysis. Semin Nucl Med 8:283–298

Chapter 2
Machine Learning Math Basics

The goal of this chapter is to provide an explanation of several well-known mathematical terms that are used in machine learning methods presented in this book. In the first part, we cover basic statistical terms such as standard deviation, variance, coefficient matrix, and Pearson correlation. It is followed by the probability terms and related topics like combinatorics, conditional probability, and probability distribution. The third section, even though it is not very extensive, consists of the crucial part in each machine learning method—operations on matrices. The next section is about differential calculus. To understand what a gradient is, we need to explain a few other terms in the first place. The first term that we explain in this section is the limits. It is an obvious term for anyone who studied a technical oriented field. The second term explained in this section is derivatives. In the clustering methods explained in this book, fuzzy sets are used. The fuzzy logic is explained in the section next to differential calculus. The last part is an overview of the distance measures available.

2.1 Statistics

We are going to skip the part of mean explanation because we consider it to be obvious to everyone reading this book. The first two terms that we explain are the standard deviation and variance. The variation can be calculated as follows:

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \overline{x})^2, \tag{2.1}$$

where x_i is an element of a set X and $\overline{x}$ is the mean of the set X elements. Let us take an example for a better understanding.

K. Przystalski et al., *Pattern Recognition Primer*,
https://doi.org/10.1007/978-3-031-91816-2_2

Table 2.1 Student weights (kilograms)

x_1	x_2	x_3	x_4	x_5	x_6
89	67	110	92	75	95

Example 1 (*Students*) We have a group of six students. Table 2.1 shows the weight in kilograms of each student.

For the data given in Table 2.1, we have a mean $\overline{x} = 88$. The variance is:

$$\sigma^2 = \frac{(1)^2 + (-21)^2 + (22)^2 + (4)^2 + (-13)^2 + (-7)^2}{6} \approx 193.33.$$

The variance is a square of the difference between the elements and the mean. It is always positive and shows how varied a set is.

There are two terms that explain a set better than variance. It is the average and standard deviation. Average deviation looks almost the same as the variance, but instead of the square of differences, we calculate the absolute value of it:

$$\sigma_a = \frac{\sum_{i=1}^{n} |x_i - \overline{x}|}{n}. \tag{2.2}$$

For data given in Example 1, we get the average deviance:

$$\sigma_a \approx 11.33.$$

The standard deviation is just a square root of variance:

$$\sigma = \sqrt{\sigma^2}. \tag{2.3}$$

For example, the standard deviation of Example 1 would be:

$$\sigma \approx 13.90.$$

This means that the deviation from the mean is 12.85. It is easy to understand when we draw it as shown in Fig. 2.1. It is the average of how far the red dots (students' weight) are from the blue line (mean).

The next term that is important for understanding machine learning methods is correlation. The most popular correlation measure is the Pearson correlation. It is about how one feature depends on the other feature. We can say that the height of a dog is highly correlated with its weight. So we have two features: the size and weight of a dog, and we know that a larger dog is usually heavier. The correlation is a value of -1 to 1 and represents the dependence of two values (features) like those shown in Table 2.2. The values presented are positive, and we have the same correlation for negative values. Some exemplary correlation charts are shown in Fig. 2.2. In Fig. 2.2a, b we present a positive and negative total correlation. In Fig. 2.2c we have a very

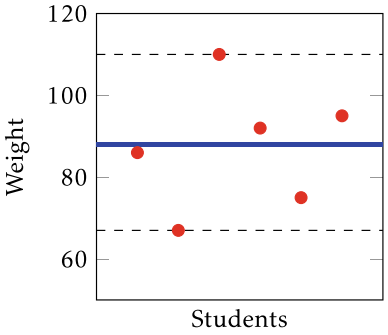


Fig. 2.1 Standard deviation of the student example. Black dotted lines are the standard deviation margins

Table 2.2 Correlation dependencies

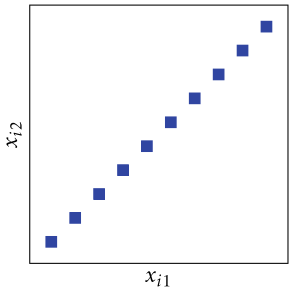
Correlation value	Correlation
0	No correlation between variables
0–0.3	Low correlation
0.3–0.5	Mid correlation
0.5–0.7	Mid-high correlation
0.7–0.9	High correlation
Above 0.9	Very high correlation
1	Total correlation

low correlation example. The samples are much more chaotic than the previous two examples. In the last Fig. 2.2d example we show a negative curvilinear example of correlation. It is still a strong correlation, but not as strong as the first two examples. The correlation for two features can be calculated as follows:

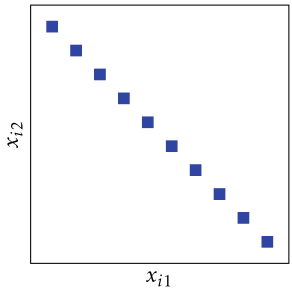
$$r^2 = \frac{\sum_{i=1}^{n}(x_{i1} - \overline{x_{i1}})(x_{i2} - \overline{x_{i2}})}{\sqrt{\sum_{i=1}^{n}(x_{i1} - \overline{x_{i1}})^2 \sum_{i=1}^{n}(x_{i2} - \overline{x_{i2}})^2}}. \tag{2.4}$$

Example 2 (*Learning*) What is the correlation between hours spent learning and the final grade of an exam? Let us assume that we have five degrees from A to F. We have a range of points that correspond to each grade:

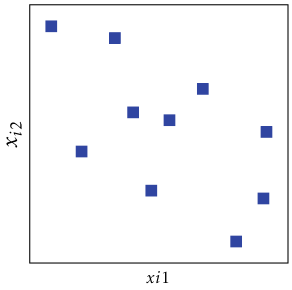
- A—100–91,
- B—90–81,
- C—80–71,
- D—70–61,
- E—60–51,
- F—50–0.



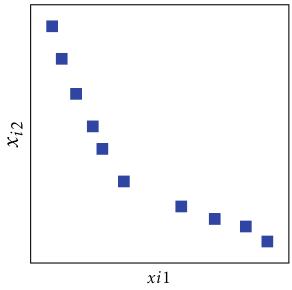
(a) Correlation where r=1.0



(b) Correlation where r=-1.0



(c) Correlation where r=0.2



(d) Correlation where $r \approx -0.7$

Fig. 2.2 Correlation examples

Table 2.3 Correlation between hours spent on learning and exam grade exemplary data

Average hours spent on learning	10	22	30	38	44	48
Average points collected	40	52	61	75	88	95

Let us assume that we have the averages of hours spent learning and points collected during the exam as presented in Table 2.3. It can be drawn as a chart like the one shown in Fig. 2.3. We can easily say, based on Fig. 2.3, that there is a positive curvilinear correlation. To be sure, we can calculate the correlation value as follows:

$$r = \frac{627 + 165 + 15 + 39 + 234 + 424}{\sqrt{1024 \cdot 2265.5}} \approx 0.9874.$$

It shows that there is a high correlation between the hours we spent learning and the final exam grade we receive.

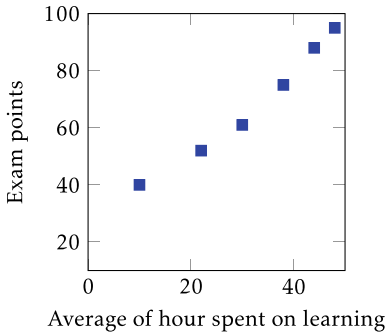


Fig. 2.3 Average hours spent on learning compared to the final exam

2.2 Probability Theory

To better explain the basics of probability theory, we use some games. Let us take the first example. A traditional dice is a cube. It gives a number from one to six when rolled. What is the probability that we get the six when we roll it? A set of all possible cases is called *sample space* and marked as Ω. In probability theory, we use sets to list all possibilities. For our example, it could be a set $A = \{\omega_1, \omega_2, \omega_3, \omega_4, \omega_5, \omega_6\}$, where ω_k are the numbers we can get after rolling the dice. This means that the probability of getting six is equal to $P(A) = \frac{1}{6}$, because there are the same chances of getting six as any other number from A.

2.2.1 Combinatorics

Probabilistic combinatorics is part of probability theory and consists of several operations to get the probability of an event, such as getting a six in a dice game. These operations are listed in Table 2.4.

The value of n is the number of possibilities in the set A. The value of k is the number of possible combinations. To better understand the methods given in Table 2.4, we present some examples.

Example 3 (*Books*) What is the number of combinations of orders of four different books? The permutation shown in Table 2.4 is the easiest way to calculate the number of possibilities. It can be easily calculated as a factorial of 4:

$$P_r(4) = 4! = 24.$$

In Python, there is a math library that already has a factorial implementation. You can use it as shown in the Listing 2.1.

```
1 from math import factorial
2
3 result=factorial(4)
```

Listing 2.1 Calculation of permutation in example 3

Table 2.4 Basic combinatorics methods

	Order		Repetition		Equation
	Yes	No	Yes	No	
Permutation	+	−	−	+	$P_r(n) = n!$
Permutation with repetition	+	−	+	−	$\overline{P_r}(n; n_1, n_2, \ldots, n_k) = \frac{n!}{n_1! \cdot n_2! \cdot \ldots \cdot n_k!}$
Variation	+	−	−	+	$V(n,k) = \frac{n!}{(n-k)!}$
Variation with repetition	+	−	+	−	$\overline{V}(n,k) = n^k$
Combination	−	+	−	+	$C(n,k) = \binom{n}{k} = \frac{n!}{k!(n-k)!}$
Combination with repetition	−	+	+	−	$\overline{C}(n,k) = \binom{n+k-1}{k} = \frac{(n+k-1)!}{k!(n-1)!}$

Table 2.5 All possible combinations of Example 4

1233	1323	1332	3312
3321	3132	3231	2331
3213	3123	2133	2313

Example 4 (*Set order*) Let's assume that we have a set of digits 1, 2, 3, 3. The digit 3 occurs twice. This is what we know about someone's safe lock-digit combination. How many combinations are possible when we know that the order is important? Permutation with repetition can give us the right answer:

$$\overline{P_r}(4; 1, 1, 2) = \frac{4!}{1!2!1!} = 12.$$

We have only 12 possibilities, so the safe lock is not that secure (Table 2.5).

Example 5 (*Elevator*) Let us say that we have an elevator in a four-story building. There are three people who will use this elevator, and each one will leave it on a different floor. How many possibilities do we have? With variation, we can calculate it easily:

$$V(4, 3) = \frac{4!}{(4-3)!} = 24.$$

We have 24 possibilities (Table 2.6):

Example 6 (*Coin*) How many different possibilities do we have of reverse and obverse when we flip a coin five times in a row? We can use variation with repetition to calculate it (Table 2.7):

$$\overline{V}(2, 5) = 2^5 = 32.$$

Table 2.6 All possible variations of the elevator example

11 22 33	11 22 34	11 23 32	11 23 34	11 24 33	11 24 32
12 21 33	12 21 34	12 23 31	12 23 34	12 24 33	12 24 31
13 21 32	13 21 34	13 22 31	13 22 34	13 24 31	13 24 32
14 21 33	14 21 34	14 22 33	14 22 31	14 23 31	14 23 32

Table 2.7 All possible combinations of Example 4

OOOOO	OOOOR	OOORO	OOORR
OOROO	OOROR	OORRO	OORRR
OROOO	OROOR	ORORO	ORORR
ORROO	ORROR	ORRRO	ORRRR
ROOOO	ROOOR	ROORO	ROORR
ROROO	ROROR	RORRO	RORRR
RROOO	RROOR	RRORO	RRORR
RRROO	RRROR	RRRRO	RRRRR

Example 7 (*Powerball*) Powerball is a well-known lottery game in the United States. We have 69 white balls in one drum and 26 red balls in the other drum. We take 5 white balls and 1 red. To simplify, let us assume that we are considering only the drum with white balls. The combination can be used here as follows:

$$C(69, 5) = \frac{69!}{5! \cdot (69 - 5)!} = 11{,}238{,}513.$$

This means that we have more than 11 million possibilities.

Example 8 (*Dice*) We have three dice. When we roll all three, we get three values from one to six each. How many combinations are possible when we assume that we roll all three dice at once? The answer is as follows:

$$\overline{C}(6, 3) = \frac{(6 + 3 - 1)!}{3! \cdot (6 - 1)!} = \frac{5! \cdot 5 \cdot 6 \cdot 7 \cdot 8}{6 \cdot 5!} = 56.$$

Example 9 (*Pairs*) We can combine the above methods to calculate the number of possibilities for more specific cases. Let us calculate how many possibilities we have to get a pair of Kings. First of all, we need to calculate how many possibilities of card combinations we have:

$$C(52, 5) = 2{,}598{,}960.$$

Let us assume that we want to get two kings. We have a set of thirteen cards of each color. We have four kings, each of different color. We get five cards in total. To get

only a pair, we need to get the other three cards from a set of twelve, so we do not get a third king. We can calculate it as follows:

$$13 \cdot C(4, 2) \cdot C(12, 3) \cdot C(4, 1)^3 = 1{,}098{,}240.$$

We have more than a million possible card arrangements to get a pair in the poker game.

2.2.2 Conditional and Independent Probability

For now, we know how to calculate the combinations possibilities in some gambling games. It is time to learn to calculate the win possibilities. As mentioned above, Ω is the set of all possible combinations. In the last example, we calculated the number of combinations to get a pair in the poker game, but what is the probability of getting a pair in general? Let A be an event of getting a pair, so the probability would be as follows:

$$P(A) = \frac{1{,}098{,}240}{2{,}598{,}960} \approx 0.4226.$$

There are more than 40% chances to get a pair at poker. Let us compare it to the odds of getting a full house (event B):

$$P(B) = \frac{3744}{2{,}598{,}960} \approx 0.0014.$$

This means that the probability of getting the full house is below 1%. It sounds reasonable, as it is logically much harder to get a full than a pair. In probability theory, we can calculate not only the probability of just an event. There are more sophisticated examples where probability can be calculated. If we roll a dice, we have 50%/50% chances to get an even value. Let A be an event to get an odd value, and let A' be an event to get an even value. In the case of the event A, we consider a set of values: 1, 3, 5. In A' the set is 2, 4, 6. We assume that we know that an event B occurred. It says that the value is greater than 3. The conditional probability is the probability of an event A when another event occurred. It can be calculated as follows:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}. \tag{2.5}$$

If $P(A) = \frac{1}{2}$ and $P(B) = \frac{1}{2}$ as in our dice example, we can calculate the probability of getting an odd value that is greater than 3 as follows:

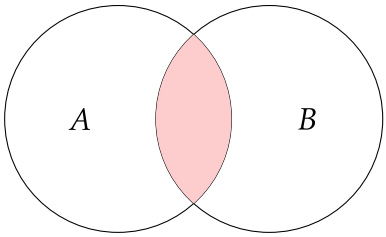


Fig. 2.4 Independent probability example. Two events A, B and the common part of both marked with red

$$P(A|B) = \frac{\frac{1}{6}}{\frac{1}{2}} = \frac{1}{3}.$$

The probability of the event B is equal to $\frac{1}{2}$ since we have the set of three elements of six possible dice values. The probability $P(A \cap B)$ is the subset of both sets A and B. This set has one element 5. This means that the probability $P(A \cap B) = \frac{1}{6}$. That is why we have the probability of $\frac{1}{3}$ to get an odd value if we know that the value of thrown dice is higher than 3.

Other important terms in probability theory are *independent probabilities* (Fig. 2.4). Two probabilities $P(A)$ and $P(B)$ are independent when $P(A|B)$ and $P(A)$ are equal for $P(B) > 0$. Let us follow the case presented for conditional probabilities. The event A means that we get an even value. Let event B be an odd value on the other dice. The probabilities of both event A and event B are equal $\frac{1}{2}$. Are these events independent? The conditional probability is equal to $P(A|B) = \frac{1}{2}$. In the given case, both events are independent, because $P(A|B)$ is equal to $P(A) = \frac{1}{2}$. In other words, independent probability is one that is not affected by the other event. It is written as follows:

$$P(A \cap B) = P(A) \cdot P(B). \tag{2.6}$$

The last term in this section that we present is the total probability. It makes us one step closer to understanding the Bayes theorem that is explained in Chap. 4. Let us say that we have a known event B and a list of events $A_1, A_2, \ldots, A_n$ where each pair of the list excludes each other. Furthermore, the disjunction of each event in the list is equal to 1. If we connect event B and events $A_1, A_2, \ldots, A_n$, we get the total probability of event B that can be calculated as follows:

$$\begin{aligned} P(B) &= P(B|A_1) \cdot P(A_1) + P(B|A_2) \cdot P(A_2) + \ldots + P(B|A_n) \cdot P(A_n) \\ &= \sum_{i=1}^{n} P(B|A_i) \cdot P(A_i). \end{aligned} \tag{2.7}$$

Example 10 (*World Championships*) Our national volleyball team qualified for the World Championships. Based on bookmakers information, we can group our opponents into five groups of different types:

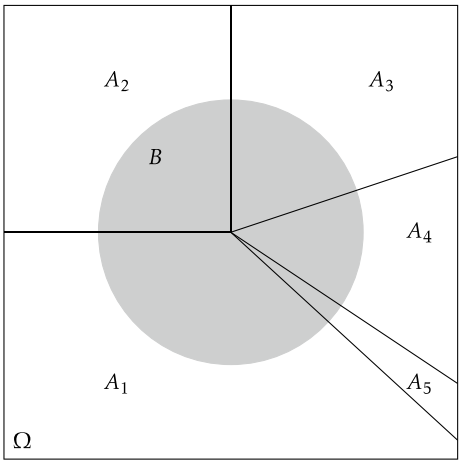


Fig. 2.5 Total probability World Championship examples

- $B|A_1$—we can win with that team for about 90%,
- $B|A_2$—rather win, but for only 70%,
- $B|A_5$—hard to say, it is a 50%/50% winning chance,
- $B|A_4$—it is more possible to loose rather than win as the chances of winning are 40%,
- $B|A_4$—we probably loose as bookmakers give us only 20% of success.

When it comes to the number of teams that we can win or loose with we can divide it as follows: A_1—35%, A_2—25%, A_3—20%, A_4—15%, A_5—5%. It can be drawn as a diagram as shown in Fig. 2.5.

The question is what our national volleyball team's chances of winning a game are? Let B be the event of the game won. It is easy to calculate with total probability:

$$P(B) = 0.9 * 0.35 + 0.7 * 0.25 + 0.5 * 0.2 + 0.4 * 0.15 + 0.2 * 0.05 = 0.66.$$

It looks as if we had a pretty strong team, as our chances of winning the game are about 66% high. Total probability can be easily drawn as a tree of all event probabilities. The tree for the current example is shown in Fig. 2.6. It consists of B' events that are about losing the game. It is easy to see in this figure that the sum of A_i events is 1 and the same with the second level leaves where the sum of B and B' is obviously also 1.

The probability tree is commonly used in economic and financial models. It gives an easy-to-understand graph, so calculations of some probabilities are easier to manage. We can calculate the probability of a given event or sum of events easily, we just need to go through the path from the right to the left and multiply it. To calculate the probability of a few events, we need to sum all paths that we want to calculate the probability of.

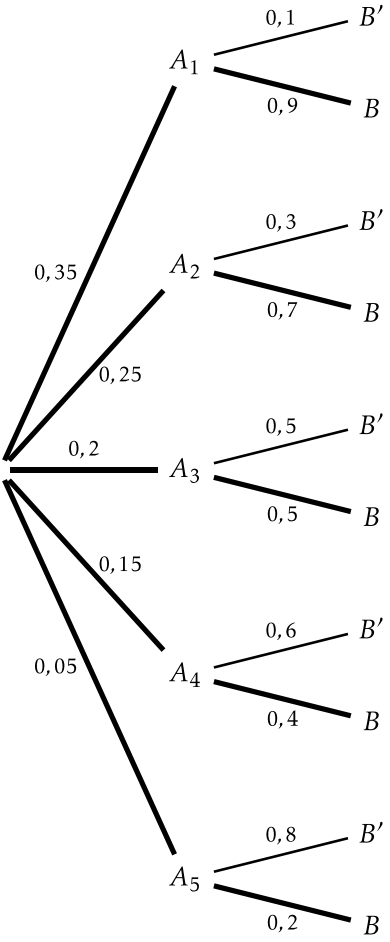


Fig. 2.6 Total probability tree of the World Championship examples

2.3 Linear Algebra

Generally, linear algebra is about operations of matrices and is a crucial part of each machine learning method. From a programming in Python perspective, there are two ways of training machine learning methods. The first is an iterative way in which we go through each element in a vector or matrix to calculate the expected result. This approach is based on many `for` or `while` loops. The second approach is based on libraries like `numpy` that are adjusted and perform much better than any iterative implementation. Numpy[1] is a library for numerical computation and is focused on the performance of such a function. The difference in performance can even be reached more than hundred times, so it is reasonable to use the vectorized approach.

[1] Numpy documentation: https://numpy.org/doc/stable/reference/index.html.

Table 2.8 Some numpy matrix/vector building methods

Sample code	Short explanation
`arr = numpy.loadtxt('matrix.txt')`	Imports a matrix/vector from a text file
`arr = numpy.genfromtxt('matrix.csv', delimiter=';')`	Imports a matrix/vector from a CSV file
`arr = numpy.ones((2,3))`	Returns a matrix of size 2×3 filled with ones
`arr = numpy.zeros((2,3))`	Returns a matrix of size 2×3 filled with zeros
`arr = numpy.eye(4)`	Returns a matrix of size 4×4 filled with zeros, except diagonals where it is filled with ones
`arr = numpy.random.rand(2,4)`	Returns a matrix of size 2×4 or random values from 0 to 1

To create a vector or matrix, we use the `numpy.array()` method. The method returns a numpy array object as in Listing 2.2.

```
import numpy as np
vector = np.array([1,2,3,4])

matrix = np.array([[1,2],[3,4]])
```

Listing 2.2 A numpy vector and numpy matrix

There are a few ways to create a matrix or vector with numpy. We can import it from a text or CSV file with methods `loadtxt()` and `genfromtxt()`. Some other methods can be used to initialize a matrix filled with some values. One of such a method is `ones()` which fills the matrix with values of 1. The size of the matrix or vector is given as a parameter. To create an identity matrix, we use the `eye()` method. The last method worth mentioning is `rand()` which creates a matrix with random values, each between 0 and 1. Examples of the mentioned methods are shown in Table 2.8. To create a vector, instead of `(2,3)` as shown in Table 2.8 for the method `ones()`, we use just the second value. It means that if we want to create a vector of 3 items, we should use `(,3)`. Numpy library has many more methods than those used for matrix creation. There is a set of methods that can be used for matrix manipulation. By manipulation we mean operations like value addition. Some of such methods are listed in Table 2.9. We can perform many operations on matrices and vectors. Let us focus on matrices, as it can be done in the same way or simpler on vectors in most cases. The most common operations are addiction, multiplication, division, and subtraction. When dealing with fixed values, we can do it in Python using the same operators as we do regular math operations. An example is shown in Listing 2.3.

```
import numpy as np

matrix_1 = np.array([[1,2],[3,4]])
matrix_2 = np.array([[5,6],[7,8]])

matrix_add = np.add(matrix_1, matrix_2)
```

Table 2.9 A few examples of numpy matrix manipulation methods with sample codes

Sample code	Description
`numpy.append(arr,values)`	Appends `values` at the end of the matrix `arr`
`numpy.insert(arr,1,values)`	Inserts `values` before index 1 of matrix `arr`
`numpy.delete(arr,2,axis=0)`	Deletes row on index 2 of matrix `arr`
`numpy.delete(arr,3,axis=1)`	Deletes column on index 3 of matrix `arr`
`list = arr.tolist()`	Returns a list
`arr.resize((3,4))`	

```
7  matrix_add = matrix_1 + matrix_2
8
9  matrix_multiply = np.multiply(matrix_1, matrix_2)
10 matrix_multiply = matrix_1 * matrix_2
```

Listing 2.3 A few basic operations done on numpy matrices

For the code shown in Listing 2.3, the first example can be calculated as follows:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \cdot 5 & 2 \cdot 7 \\ 3 \cdot 6 & 4 \cdot 8 \end{bmatrix} = \begin{bmatrix} 5 & 14 \\ 18 & 32 \end{bmatrix}. \tag{2.8}$$

In this example, we use a small matrix of size 2×2. In many and now even almost all machine learning methods, the matrices are much larger. As mentioned above, numpy is optimized to perform well in such cases. Some other operations that can be performed using numpy are shown in Table 2.10. The method that may not be as easy to understand is the `dot()` method. It is a kind of multiplication operation, but it works differently from the `multiply()` method. Calculates the value of the dot product differently. In Table 2.10 an example of the calculation of the dot product is shown. The presented example works for two matrices but behaves like `multiplying()` if one of the parameters is just one number. There are more methods in which we perform mathematical operations on matrices. The most popular are shown in Table 2.11. The table contains only methods that are used in further detail in this book. We have methods that use the matrix as input and returns matrix as output. All methods do an operation on each cell in the matrix one by one. The method names do exactly what they mean, so no more explanation is necessary here. The only three that need the second parameter are `min()`, `max()` and `mean()`. The `axis` parameter is about the number of axis where the minimum, maximum or mean value is returned.

The last thing we describe in this section are the properties of numpy arrays. There are two listed in Table 2.12 that are important in our opinion. The first gives the total number of elements. If we have a matrix of size 2×5, the size property returns 10. The property `shape` returns 2×5 for the same example.

Table 2.10 Numpy matrices operations with examples

Sample code	Example
`matrix = numpy.add(matrix1,matrix2)`	$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \times \begin{bmatrix}5 & 6\\7 & 8\end{bmatrix} = \begin{bmatrix}1+5 & 2+7\\3+6 & 4+8\end{bmatrix} = \begin{bmatrix}6 & 9\\9 & 12\end{bmatrix}$
`matrix = numpy.subtract(matrix1,matrix2)`	$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \times \begin{bmatrix}5 & 6\\7 & 8\end{bmatrix} = \begin{bmatrix}1-5 & 2-7\\3-6 & 4-8\end{bmatrix} = \begin{bmatrix}-4 & -5\\-3 & -4\end{bmatrix}$
`matrix = numpy.multiply(matrix1,matrix2)`	$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \times \begin{bmatrix}5 & 6\\7 & 8\end{bmatrix} = \begin{bmatrix}1\cdot 5 & 2\cdot 6\\3\cdot 7 & 4\cdot 8\end{bmatrix} = \begin{bmatrix}5 & 12\\21 & 32\end{bmatrix}$
`matrix = numpy.divide(matrix1,matrix2)`	$\begin{bmatrix}10 & 2\\9 & 4\end{bmatrix} \times \begin{bmatrix}2 & 2\\3 & 3\end{bmatrix} = \begin{bmatrix}10/2 & 2/2\\9/3 & 4/3\end{bmatrix} = \begin{bmatrix}5 & 1\\3 & 1.33\end{bmatrix}$
`matrix = numpy.dot(matrix1,matrix2)`	$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \times \begin{bmatrix}5 & 6\\7 & 8\end{bmatrix} = \begin{bmatrix}1\cdot 5 & 2\cdot 7\\3\cdot 6 & 4\cdot 8\end{bmatrix} = \begin{bmatrix}5 & 14\\18 & 32\end{bmatrix}$

Table 2.11 Commonly used numpy methods

Sample code	Example
`matrix = numpy.sqrt(matrix1)`	Calculates square root of each element
`matrix = numpy.sin(matrix1)`	Calculates the sinus value of each element
`matrix = numpy.log(matrix1)`	Calculates natural logarithm of each element
`matrix = numpy.abs(matrix1)`	Absolute value of each element in the array
`matrix = numpy.ceil(matrix1)`	Rounds up to the nearest int
`matrix = numpy.floor(matrix1)`	Rounds down to the nearest int
`matrix = numpy.round(matrix)`	
`matrix = numpy.mean(arr,axis=0)`	Returns mean along specific axis
`sum = numpy.sum(matrix1)`	Returns sum of arr
`min = numpy.min()`	Returns minimum value of arr
`max = numpy.max(axis=0)`	a
`sorted_matrix = numpy.sort()`	b

Table 2.12 Other useful numpy properties

Property name	Description
arr.size	Returns the number of elements
arr.shape	Returns the shape of a matrix

2.4 Differential Calculus

The most common method to optimize and find the best possible model uses gradients. To understand gradients, we need to explain the limits and derivatives. One of the best explanations for what limits are can be found in [1]. This is our best prediction of a point that we did not observe. The formal definition of the limit is the following equation:

$$\lim_{x \to c} f(x) = L, \tag{2.9}$$

where $x \to c$ means that x is coming close to c. L is our prediction of $f(x)$. Additionally, we assume that since it is a prediction, there is an error margin $\epsilon > 0$, so that there is a range margin $\delta > 0$ that for each x_i within $0 < |x_i - c| < \delta$ we have:

$$|f(x) - L| < \epsilon. \tag{2.10}$$

Let us draw an example for a better understanding. In Fig. 2.7 we can see a function $f(x) = 1 + \frac{1}{x}$ marked blue. In red, we mark the error margins ϵ. The middle of the margin ϵ is marked black $y = 1$. Let $x \to x_0$ where $x_0 = 50$. The error margin shown in Fig. 2.7 is set to $\epsilon = 0.2$ and $\delta = 30$. The limit would look as follows:

$$\lim_{x \to 50} f(x) = 1 + \frac{1}{x}.$$

If we can check the limit for x_0 directly, we just assign to x the value of x_0:

$$\lim_{x \to 50} f(x) = 1\frac{1}{50}.$$

Now we can check how small the error margin ϵ can be. We assumed ϵ to be 0.2 for now:

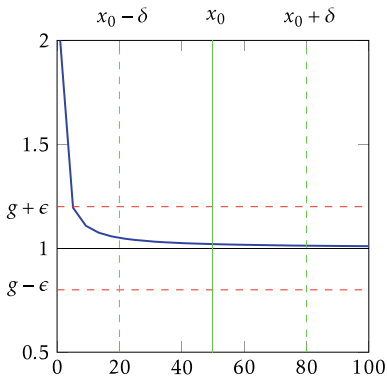


Fig. 2.7 Limits example for $y = 1 + \frac{1}{x}$

$$|1 + \frac{1}{x} - 1\frac{1}{50}| < 0.2,$$
$$|\frac{1}{x} - \frac{1}{50}| < 0.2.$$

The range margin δ is set in our case for 30. We can check if ϵ is set correctly for given δ by checking the boundary values:

$$|0.05 - 0.02| < 0.2,$$
$$|0.0125 - 0.02| < 0.2.$$

For both boundaries, the inequality is true. We could even reduce the error margin ϵ to 0.05 and it would still be true. A more interesting example would be $x \to \infty$. This means that x is increasing to infinity. In other words, $\frac{1}{x}$ will gradually become closer to 0. In such a case we assume that it is 0, so:

$$\lim_{x \to \infty} 1 + \frac{1}{x} = 1.$$

Limits are also very useful when facing a zero-divided problem. We know that we cannot calculate the function $f(x) = 1 + \frac{1}{x}$ for $x = 0$, because we cannot divide by zero. In this case, we can assume that x is going to be close to 0 and is positive: $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \ldots, \frac{1}{128}, \ldots$, but it never reaches or goes below 0. In this case, we mark it with a $+$ like:

$$\lim_{x \to 0^+} 1 + \frac{1}{x} = \infty.$$

We can draw the function as shown in Fig. 2.8. For $\frac{1}{2}$ and $\frac{1}{4}$, it is, respectively:

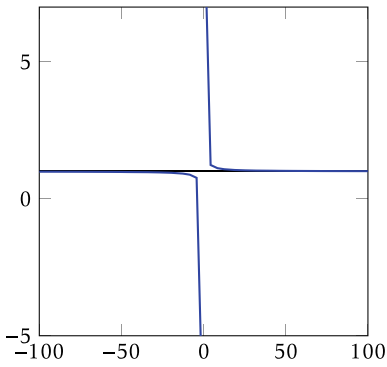


Fig. 2.8 Limits example for $\lim_{x \to 0^+} y = 1 + \frac{1}{x}$

$$\lim_{x \to 0^+} f\left(\frac{1}{2}\right) = 1 + \frac{1}{2} = 3,$$
$$\lim_{x \to 0^+} f\left(\frac{1}{4}\right) = 1 + \frac{1}{4} = 5.$$

The closer we are to 0 the higher the value we get. That is why

$$\lim_{x \to 0^+} 1 + \frac{1}{x} = \infty.$$

The same scenario holds for negative values getting closer to 0:

$$\lim_{x \to 0^-} 1 + \frac{1}{x} = -\infty.$$

We presented examples for both $x \to 0^+$ and $x \to 0^-$ in Fig. 2.8.

2.4.1 Derivatives

Derivatives are about changes in function. We can use derivatives to see how much the function values change from one point to another. Mathematically, it can be presented as follows:

$$f'(x) = \lim_{dx \to 0} \frac{f(x + dx) - f(x)}{dx}, \tag{2.11}$$

where $f'(x)$ is the derivative of the function, dx is the change between points. The change can also be found in other publications such as δx. It is the difference between two points:

$$dx = x - x_0. \tag{2.12}$$

A graphical interpretation of the derivative can be drawn like shown in Fig. 2.9. The derivative can also be interpreted as the tangent tan of the $\angle CAB$ angle shown in Fig. 2.9:

$$\tan \sphericalangle CAB = \frac{f(x_0 + dx) - f(x)}{dx}. \tag{2.13}$$

Based on Eq. 2.11, we can calculate the derivative $f(x)'$ by adding $x^2 + 2x$ to the equation:

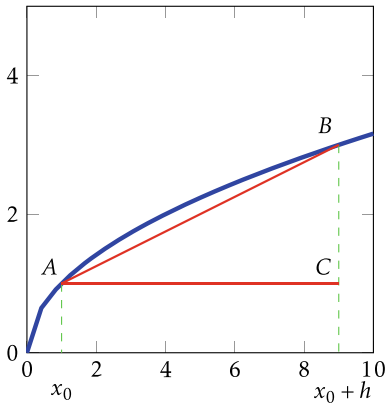


Fig. 2.9 Derivative definition

$$
\begin{aligned}
f(x)' &= \lim_{dx \to 0} \frac{f(x+dx) - f(x)}{dx} \\
&= \lim_{dx \to 0} \frac{(x+dx)^2 + 2(x+dx) - (x^2 + 2x)}{dx} \\
&= \lim_{dx \to 0} \frac{x^2 + 2xdx + dx^2 + 2x + 2dx - x^2 - 2x}{dx} \\
&= \lim_{dx \to 0} \frac{2xdx + dx^2 + 2dx}{dx} \\
&= \lim_{dx \to 0} \frac{dx(2x + dx + 2)}{dx} \\
&= 2x + 2.
\end{aligned}
$$

We assumed that the change is 0. For any change different than 0 we get $f(x)' = 2x + 2 + dx$, where dx is the change/error rate. This means that the lower we go, the lower the error rate. Usually, the derivatives are calculated for a small change, because the change is different at any x_0. In Fig. 2.9 the change will be different for $x_0 = 3$ and for $x_0 = 5$, because the blue line isn't straight. As shown in Eq. 2.11 $dx \to 0$, it should be a small number like 0.1 or less. Instead of calculating the derivative from Eq. 2.11 every time, there are some commonly used precalculated derivatives which simplify the calculations. The most popular derivatives can be found in Table 2.13.

2.4.2 *Gradients*

The derivatives show the next step of our function at some point. This means that we can find the maximum or minimum much faster. The higher the derivative, the more the value of the function increases. Gradients use derivatives to show the direction of function increase of more than one variable. It is marked with $\nabla f(x_1, x_2)$. Gradients

Table 2.13 Basic derivatives functions

Function name	Derivative
Constant function (c)	$(c)' = 0$
Power function, $\alpha \neq 0$	$(x^{\alpha})' = \alpha x^{\alpha-1}$
Square root function, where $\alpha = \frac{1}{2}$	$(\sqrt{x})' = \frac{1}{2\sqrt{x}}$
Exponential function $\alpha > 0, a \neq 1$	$(a^x)' = a^x \ln a$
Exponential function with base e	$(e^x)' = e^x$
Logarithm function, $\alpha > 0, a \neq 1$	$(\log_a x)' = \frac{1}{x \ln a}$
Natural logarithm, where $a = e$	$(\ln x)' = \frac{1}{x}$
sines function	$(\sin x)' = \cos x$
cosines function	$(\cos x)' = -\sin x$
tangent function	$(\mathrm{tg}\, x)' = \frac{1}{\cos^2 x} = 1 + \mathrm{tg}^2 x$
cotangent function	$(\mathrm{ctg}\, x)' = -\frac{1}{\sin^2 x} = -(1 + \mathrm{ctg}^2 x)$
arcsin function	$(\arcsin x)' = \frac{1}{\sqrt{1-x^2}}$
arccos function	$(\arccos x)' = -\frac{1}{\sqrt{1-x^2}}$
arctg function	$(\mathrm{arctg})' = \frac{1}{\sqrt{1-x^2}}$
arcctg function	$(\mathrm{arcctg})' = -\frac{1}{1+x^2}$
Hiperbolic sines function	$(\mathrm{sh}\, x)' = \mathrm{ch}\, x$
Hiperbolic cosines function	$(\mathrm{ch}\, x)' = \mathrm{sh}\, x$

are used in several classification methods to find the local minimum or maximum of a function. Local minimums are part of the learning process, and the faster the algorithm finds them, the better. Gradients make it possible to find them much faster than using many other methods. Let us take an example of the function $f(x_1, x_2) = 3x_2^2 + 2x_2$. It is drawn in Fig. 2.10. The function is shown only for values of x_1, x_2 between -10 and 10. The three-dimensional plot is divided into rectangular parts that are colored differently. This is done to better understand the concept. The colors represent the value of y like is done with temperatures. The blue rectangles are where the value is low. Yellow rectangles have a higher value than blue rectangles, and the highest values are marked in red. A gradient of function with two input variables (two features) and one output (label/class) looks like:

$$\nabla f(x_1, x_2) = \left(\frac{f(x_1, x_2)}{dx_1}, \frac{f(x_1, x_2)}{dx_2} \right). \tag{2.14}$$

Keep in mind that the commonly used nomenclature is $f(x, y)$ where the output is z, but we changed it to comply with the nomenclature of pattern recognition classification methods. The fraction $\frac{f(x_1,x_2)}{dx_1}$ means that we take a derivative of the function $f(x_1, x_2)$, but we consider x_1 as a variable on which we calculate the derivative. The other variables are handled as a constant c. This means that in the case presented, a gradient is a vector of two derivatives. If we calculate the derivatives for both variables of the function $f(x_1, x_2) = 3x_1^2 + 2x_2$ the gradient is as follows:

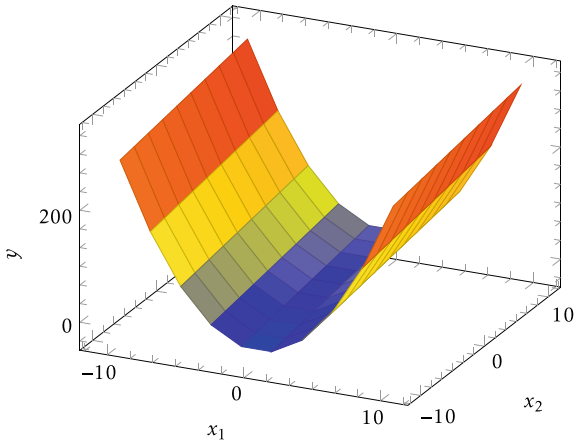


Fig. 2.10 Function $3x_1^2 + 2x_2$ used in gradient example

Table 2.14 Gradient calculation of function $f(x_1, x_2) = 3x_1^2 + 2x_2$

x_1	x_2	$\frac{f(x_1,x_2)}{dx_1}$	$\frac{f(x_1,x_2)}{dx_2}$	$\nabla f(x_1, x_2) = (2x_1, 1)$
0	0	0	1	(0, 1)
1	1	2	1	(2, 1)
2	2	4	1	(4, 1)
3	3	6	1	(6, 1)

$$\nabla f(x_1, x_2) = (2x_1, 1).$$

This means that if we move on the x_1 axis, the value will decrease or increase, but if we move on the second axis, nothing changes. It makes sense if we look at the plot in Fig. 2.10. Calculate the gradient for a few values of x_1 and x_2. In Table 2.14 a few gradients of the same function are calculated. We see that the value of x_1 increases when the value of x_2 does not change at all. The gradient has higher values when the change is bigger. It has a 0 at the function's local maximum.

2.5 Fuzzy Logic

In this section we explain the basics of fuzzy logic. We use it in the clustering chapter. It consists of two main terms: membership function and fuzzy set. Fuzzy logic is about perception of not well-precision terms. To give just a simple example. What does it mean that someone is tall? Everyone will give a different answer. There are

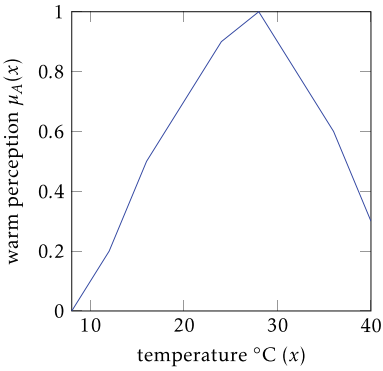


Fig. 2.11 Weather perception example

plenty of real-world examples of fuzzy logic usage. Fuzzy logic is mapping those problems to numbers that we can use in machine learning methods.

Example 11 (*Weather*) Let's take the weather. If someone says it is warm, then what does it really mean? Is it 70 °F/30 °C or is it already hot? Each person has a different perception and for someone who lives in Scandinavia it could be already hot, but for someone who lives in India it would be just warm. The possible options of interpretation are called the universe of discourse. Let us formalize it. If we say that the temperature outside is 70 °F/30 °C each person can give a note on the scale from 0 to 10 of how warm it is (or from 0 to 1). Let us take just an individual for now. We can ask for the perception of temperatures between 8 and 40 with an interval of 4. It could look like it is shown in Fig. 2.11. In this case 28 °C would be the most preferable temperature in case of warm conditions.

Let the temperatures be a set X and the warm perception a fuzzy set A. The function that assigns a value from set X to a value from set A is called membership function $\mu_A(x)$:

$$\mu_A : X \to [0, 1]. \tag{2.15}$$

We have only three possibilities of values that the membership function can assign to x:

1. $\mu_A(x) = 1$ means that x is fully a member of A; $x \in A$,
2. $\mu_A(x) = 0$ means that x is not a member of A; $x \notin A$,
3. $0 < \mu_A(x) < 1$ means that x is partially a member of A.

In Example 11 the set X contains values like: $X = [8, 12, 16, 20, 24, 28, 32, 36, 40]$. The fuzzy set A has a special notation that is unique. The general equation of fuzzy set A is:

$$A = \sum_{i=1}^{n} \frac{\mu_A(x_i)}{x_i}. \tag{2.16}$$

For Example 11 is s follows:

$$A = \frac{0}{8} + \frac{0.2}{12} + \frac{0.5}{16} + \frac{0.7}{20} + \frac{0.9}{24} + \frac{1}{28} + \frac{0.8}{32} + \frac{0.6}{36} + \frac{0.3}{40}.$$

It is a specific type of notation as we have a membership function value at the top and the x_i of set X at the bottom. It cannot be divided as it is done in regular fractions as it is only a representation of the relation of perception ([0, 1]) to the real-world values X.

Membership function shown in Fig. 2.11 is custom and applies only for exact example. We have several well-known membership functions that are commonly used. A function that responds to a non-fuzzy set is:

$$\mu_A(x) = \begin{cases} 1, & \text{if } x = \overline{x} \\ 0, & \text{if } x \neq \overline{x} \end{cases}. \tag{2.17}$$

It is called a singleton membership function. Some popular membership functions are presented in Fig. 2.13. In Fig. 2.13a Gaussian membership function is shown. The temperature example that we mentioned above can be an example of a fuzzy set with a Gaussian membership function. In an ideal case, it would look like a Gaussian membership function. Another popular one is the membership function of types (see Fig. 2.13b). It looks like a logistic function. It is not a rule, but this kind of function is commonly used in cases where we have values that are close to a or c as we had in Example 5. The membership function shown in Fig. 2.13c is similar to the Gaussian membership function. It would be sharper if we could leave it in case of a fuzzy set. It is used also in similar cases like it is in the Gaussian membership function. The last corresponds to a situation in which we have a stable assignation to most values of x, but at some point a it stops being valid and the membership drops to 0.

2.6 Dissimilarity Measures

This section is related to the measures of similarities between objects. When we take two objects, we can measure the similarity of both objects using features x_{i1} and x_{i2}. To be precise, it is the measure of dissimilarity known as $\rho(X)$, where X is a set of feature vectors. In the mathematical approach, a dissimilarity measure needs to follow the steps:

- $\forall_{x_r \in X} \rho(x_r, x_r) = 0$,
- $\forall_{x_r, x_s \in X, r \neq s} \rho(x_r, x_s) = \rho(x_r, x_s)$,
- $\rho(x_r, x_s) = 0 \Leftrightarrow x_r = x_s$,
- $\forall_{x_r, x_s, x_t \in X, r \neq s \neq t} \rho(x_r, x_s) \leq \rho(x_r, x_s) + \rho(x_s, x_t)$,

where x_r, x_s, x_t are features. The dissimilarity measure is also known as the distance measure $d(x_r, x_s)$. The most popular measure is the Euclidean distance. The generic equation is as follows:

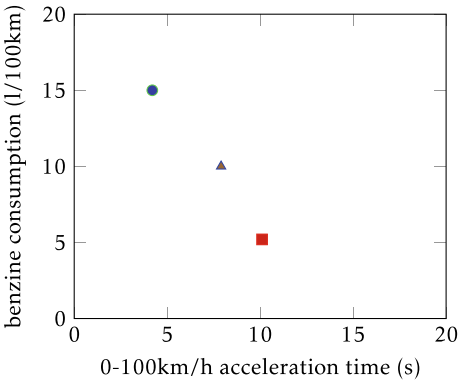


Fig. 2.12 Three objects (cars) that we measure different dissimilarity methods on: Porsche Panamera marked as a circle, Toyota Corolla marked as a square, Ford Mondeo marked as a triangle

$$\rho_{\text{Min}}(x_r, x_s) = \sqrt{\sum_{i=1}^{d}(x_{ri} - x_{si})^2}. \tag{2.18}$$

It is also known as the Minkowski distance. In the above equation d is the dimension number. An exemplary method that calculates it is shown in Listing 2.4. To simplify the process, let's assume that we have two features only ($d = 2$):

$$\rho_{\text{Min}}(x_r, x_s) = \sqrt{(x_{r1} - x_{s1})^2 + x_{r2} - x_{s2})^2}. \tag{2.19}$$

Example 12 (*Cars*) Let us take three different cars: Toyota Corolla (x_t), Ford Mondeo (x_f), and Porsche Panamera (x_p). For each car, we take two features: the acceleration time to 100 km/h and the fuel consumption per 100 km. It is shown in Fig. 2.12.

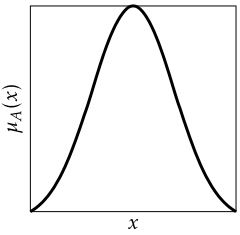
The distances between each pair of cars can be calculated like following:

$$\rho(x_p, x_t) = \sqrt{(4.2 - 10.1)^2 + (15 - 5.2)^2} \approx 11.44$$

$$\rho(x_p, x_f) = \sqrt{(4.2 - 7.9)^2 + (15 - 10.0)^2} \approx 6.22$$

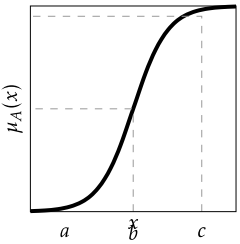
$$\rho(x_t, x_f) = \sqrt{(10.1 - 7.9)^2 + (5.2 - 10.0)^2} \approx 5.28$$

Based on the calculations we know that Ford Mondeo is more like Porsche Panamera, even it sounds weird. The most similar cars are Toyota Corolla and Ford Mondeo. We should not take this comparison seriously, as comparing Corolla with Panamera sounds unreasonable for someone who is interested in cars. The Euclidean distance can be easily calculated using Python as shown in the Listing 2.4.



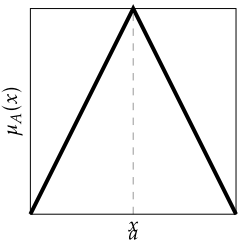
(a) Gaussian membership function

$$\mu_A(x) = exp(-(\frac{x-\overline{x}}{\sigma})^2), \quad (2.20)$$



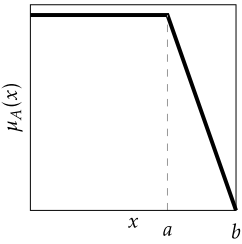
(b) Membership function of s type

$$\mu_A^s(x;a,b,c) = \begin{cases} 0 & \text{for } x \le a, \\ 2(\frac{x-a}{c-a})^2 & \text{for } a < x \le b, \\ 1-2(\frac{x-c}{c-a})^2 & \text{for } b < x \le c, \\ 1 & \text{for } x > c. \end{cases} \quad (2.21)$$



(c) Triangular membership function

$$\mu_A^t(x;a,b,c) = \begin{cases} 0 & \text{for } x \le a, \\ \frac{x-a}{b-a} & \text{for } a < x \le b, \\ \frac{c-x}{c-b} & \text{for } b < x \le c, \\ 0 & \text{for } x > c. \end{cases} \quad (2.22)$$



(d) Membership function of L type

$$\mu_A^L(x;a,b) = \begin{cases} 1 & \text{for } x \le a, \\ \frac{b-x}{b-a} & \text{for } a < x \le b, \\ 0 & \text{for } x > b. \end{cases} \quad (2.23)$$

Fig. 2.13 Few popular membership functions

Table 2.15 Popular distance metrics

Measure name	Equation
Manhattan distance	$\rho_{\text{Man}}(x_r, x_s) = \sum_{i=1}^{n} \lvert x_{ri} - x_{si} \rvert$ (2.25)
Chebyshew distance	$\rho_{\text{Ch}}(x_r, x_s) = \max_{1 \le i \le n} \lvert x_{ri} - x_{si} \rvert$ (2.26)
Frechét distance	$\rho(x_r, x_s) = \sum_{i=1}^{d} \frac{\lvert x_{ri} - x_{si} \rvert}{1 + \lvert x_{ri} + x_{si} \rvert} \frac{1}{2^i}$ (2.27)
Canberra distance	$\rho(x_r, x_s) = \sum_{i=1}^{d} \frac{\lvert x_{ri} - x_{si} \rvert}{\lvert x_{ri} + x_{si} \rvert}$ (2.28)
Post office distance	$\rho_{\text{pos}}(x_r, x_s) = \begin{cases} \rho_{\text{Min}}(x_r, 0) + \rho_{\text{Min}}(0, x_s), & \text{for } x_r \neq x_s, \\ 0, & \text{for } x_r = x_s \end{cases}$ (2.29)
Bray-Curtis distance [2]	$\rho_{\text{bc}}(x_r, x_s) = \frac{\sum_{i=1}^{d} \lvert x_{ri} - x_{si} \rvert}{\sum_{i=1}^{d} (x_{ri} - x_{si})}$ (2.30)

```
math.sqrt(abs((int(x[0])-int(v[0]))*(int(x[0])-int(v[0]))+(int(x[1])-int
(v[1]))*(int(x[1])-int(v[1]))))
```

Listing 2.4 Euclidean distance calculated with Python

We have more than just the Euclidean distance. The most popular distances can be found in Table 2.15. The general approach to the calculation for each distance is the same as in our previous example. For better understanding, we calculate the Manhattan distance for the previous example. This distance is also known as the city block distance or the taxi distance. For two objects it can be calculated as follows:

$$\rho_{\text{Man}}(x_r, x_s) = |x_{r1} - x_{s1}| + |x_{r2} - x_{s2}|. \tag{2.24}$$

$$\rho_{\text{Man}}(x_p, x_t) = |4.2 - 10.1| + |15 - 5.2| = 15.1$$

$$\rho_{\text{Man}}(x_p, x_f) = |4.2 - 7.9| + |15 - 10.0| = 8.7$$

$$\rho_{\text{Man}}(x_t, x_f) = |10.1 - 7.9| + |5.2 - 10.0| = 7$$

Manhattan distance and Minkowski distance results for our car example are similar. The numbers are different, but the relationship between each distance is almost the same. We can calculate the values of each distance shown in Table 2.15. It is not the full list of distances. More metrics can be found in [3–5].

For Further Reading

1. James G, Witten D et al (2023) An introduction to statistical learning: with applications in Python. Springer
2. Liu Y (2024) Python machine learning by example: unlock machine learning best practices with real-world use cases, 4th edn. Packt

3. Maurits N, Ćurčić-Blake B (2023) Math for scientists. Refreshing the essentials. Springer
4. Bruce P, Bruce A, Gedeck P (2020) Practical statistics for data scientists, 2nd edn. O'Reilly
5. Kar R, Le D-N, Mukherjee G et al (2023) Fuzzy logic applications in computer science and mathematics. Wiley-Scrivener

References

1. Azad K (2013) An intuitive introduction to limits. https://betterexplained.com/articles/an-intuitive-introduction-to-limits/ [Online]. Accessed 31 July 2016
2. Bray JR, Curtis JT (1957) An ordination of the upland forest communities of southern Wisconsin. Ecol Monogr 27:325–349
3. Krzyśko M, Wołyński W, Skorzybut M (2008) Systemy uczace sie. WNT
4. Krzanowski WJ, Marriott FHC (1994) Multivariate analysis: Kendall's library of statistics, vol. 1. Wiley
5. Krzanowski WJ, Marriott FHC (1995) Multivariate analysis: Kendall's library of statistics, vol. 2. Wiley

Chapter 3
Unsupervised Learning

Unsupervised methods are based on data sets that do not contain labels. This means that the algorithms are learning only using feature vectors. This group of learning methods is also known under different names. It depends on the context where it is used. Unsupervised learning can be called learning without a teacher. It is the opposite to learning with a teacher, supervised learning. Unsupervised learning is also known as partitioning, segmentation, typology, numerical taxonomy, or clustering. The last term is one of the most commonly used, aside from unsupervised learning. A cluster is a set of elements/objects of the same label. Compared to supervised methods, the label used here is based on similarities between elements of each cluster. It means that some elements are more similar to other elements than to other elements. In other words, the goal of the clustering method is to find groups of objects that are most similar to each other. It is important to mention that if we say label in the context of unsupervised learning, we mean the testing part of a method. Labels are assigned during the learning phase. Each element/object belongs to a group. Each group has its own label that is different for each group. There are three major types of clustering methods: distributed, density-based, and hierarchical. Distributed methods are based on data distribution in the feature space. The second type is about the density or easier neighborhood elements in feature space. Hierarchical clustering is based on the hierarchy of elements in the training data set. This kind of method creates a dendrogram as output. Apart from the methods we describe in this chapter, there are a few terms that need more explanation first. Sometimes it can happen that some elements do not fit into any cluster. Such elements are known as noise, outliers, or errors. A popular approach is to find just one cluster of elements that are most similar to each other. Filter the outliers from the data set. The density-based clustering method explained later in this chapter works in this way. In the next section, we explain how to measure the quality of the clustering method using different quality metrics based on heterogeneity, homogeneity, and indices. A separate topic is the number of clusters that we want. There is a way to get the best number of clusters k based on the computation cost and the mentioned metrics. In pattern recognition, clustering

K. Przystalski et al., *Pattern Recognition Primer*,
https://doi.org/10.1007/978-3-031-91816-2_3

methods have many use cases. One of the most popular is image segmentation. In the last section of this chapter, we show how to implement a simple image segmentation method using the k-means clustering method.

3.1 Distributed Clustering

In this section, we explain a few methods of distributed clustering. We have divided those into three groups: k-means, fuzzy, and possible. Most distributed clustering methods are extended or modified algorithms of the methods presented in this section. We start with k-means, which is the most known clustering method. More complex methods are presented in the next two sections.

3.1.1 K-Means

K-means is also known as a hard c-means algorithm (HCM) and is one of the simplest clustering methods. The goal of this algorithm is to assign each element of the training data set to a cluster in a binary way. This means that an element can be fully assigned to only one cluster. This is a strict (hard) type of assignation. All k-means-based methods are iterative algorithms and consist of few parts that are the same in each case:

1. choose the entrance cluster centroids,
2. calculate the membership matrix U,
3. calculate new centroids matrix V,
4. calculate the difference between the previously calculated membership matrix U and the new calculated in the current iteration.

Each step is described in the following subsections.

Entrance cluster centroids

This step is done only once. In most methods, the center of each cluster needs to be chosen before the algorithm starts. Such a center point is also called a centroid. The two most popular ways to do it are to set it randomly or set fixed values. Cluster centers should be chosen from values that are between the minimum and maximum values of each feature. We rarely have the same cluster centers that are at the entrance and when the algorithm finishes the calculations. We can set random centers as shown in Listing 3.1.

```
def select_centers():
    """
    This method selects randomly centroids

    :return: centroids
    """
    return np.random.rand(groups, len(data_set[0]))
```

Listing 3.1 Random centroids generation

We set random centers based on the feature space that is stored in `__space`. It represents the minimum and maximum values of each feature. In the Listing 3.1, the feature space is defined as two-dimensional. A random centroid is generated for each group in each iteration.

Membership matrix *U*

The second part of each algorithm is the calculation of the membership matrix U. This part needs cluster centroids to calculate the matrix U. The previous step of centroid generation can be used in most methods in the same way. The membership matrix calculation step is slightly different in each clustering method. The matrix U consists of c rows and k columns, where c is the number of groups/clusters we want to have and k is the number of elements in the training data set. We iterate through i in rows and j through columns. An element of U is μ_{ij}. We already used the symbol μ in the section about fuzzy sets. We call it a membership function. Ideally, it corresponds to clustering as we measure whether or not each object is a member (assigned) of a group. Let us assume that we want to distinguish only two groups for now. The membership/assignation can be a value in the range of 0–1. In the case of k-means, it is the value of 0 or 1. A matrix U with two groups and five elements can look as follows:

$$U = \begin{bmatrix} 0\ 1\ 0\ 1\ 0 \\ 1\ 0\ 1\ 0\ 1 \end{bmatrix}.$$

It means that the object x_1 is assigned to the group $c = 2$, x_2 to the group $c = 1$, and so on. We calculate the membership matrix for each object in the training data set. In the current example, there are only six objects. The value of the membership function can be calculated in HCM as shown in Eq. 3.1.

$$\mu_{ik}^{(t)} = \begin{cases} 1 & \text{if } d(x_k, v_i) < d(x_k, v_j), \text{for each } j \neq i, \\ 0 & \text{in other case.} \end{cases} \tag{3.1}$$

The assignation is done in a simple way. For each object x_k, we measure the distance from it to each group center. The closest distance wins. As shown in the Listing 3.2 we set the variable `minimal_distance` the distance of the object x and the centroid of the first cluster. The `group_id` is the id to which the object x is assigned. It may change during the `for` loop if the distance between x and other centroids.

```
def calculate_u(x, centers):
    """
    This method calculates membership of the object x.

    :param x: object that we want to set the cluster
    membership id
    :param centers: centroids
    :return: Membership vector
    """
    if calculate_distance(x, centers[0]) <
    calculate_distance(x, centers[1]):
        return [1, 0]
    else:
        return [0, 1]
```

Listing 3.2 HCM membership matrix calculation

The value 1 is assigned to the `group_id` in line 8.

New centroid v_i calculation

Centroids are calculated in a similar way in most methods. The number of groups c_i is the same as the number of centers v_i, where $i = 1, \ldots, c$:

$$V = [v_1, v_2, \ldots, v_c]. \tag{3.2}$$

Each group center is calculated separately as follows:

$$v_i = \frac{\sum_{k=1}^{M} \mu_{ik}^{(t)} x_k}{\sum_{k=1}^{M} \mu_{ik}^{(t)}} \tag{3.3}$$

We use the assignation μ (`u`) and the feature vector x_k (`__data_set`) to calculate the new group centers. The code that calculates new centers is presented in Listing 3.3.

```
def calculate_new_centers(u):
    """
    This method calculates new centroids of each cluster.

    :param u: membership matrix
    :return: new centers
    """
    new_centers = []
    for c in range(groups):
        new_centers.append(
            np.divide(np.dot(np.square(np.array(u)[:, c]),
     data_set), np.sum(np.square(np.array(u)[:, c]))))
    return new_centers
```

Listing 3.3 Centers calculation

As Eq. 3.3 is a bit more complex compared to the previous ones used in this chapter, we divided the loop in line 3 into two parts: numerator as `u_x_vector` and denominator as `u_scalar`. Both variable names indicate the type of value they contain and the variables involved. We have two loops, one to go through all centroids that we need to calculate, and the second loop to go through all objects in our training data set.

Difference measure

We calculate the membership matrix from the previous step as well as new centroids in each iteration until the differences between the changes in both are small enough. The differences calculated to stop the loop are calculated as it is shown in Listing 3.4.

```
def calculate_differences(new_membership, membership):
    """
    This method calculates the differences between the
    old and new membership matrix.

    :param new_membership: new membership matrix
    :param membership: current membership matrix
    :return: the difference between two membership
    matrices
    """
    return np.sum(np.abs(np.subtract(membership,
    new_membership)))
```

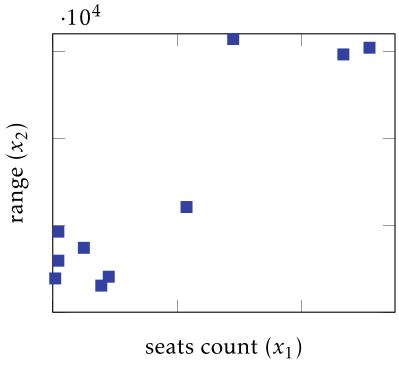
Listing 3.4 Differences calculation

The difference level at which we stop the loop is set depending on the data set and feature space. In our example, which we show next, it is set to 0.5. To calculate the differences, we take the current membership matrix and the newly calculated ones and compare both rows and columns.

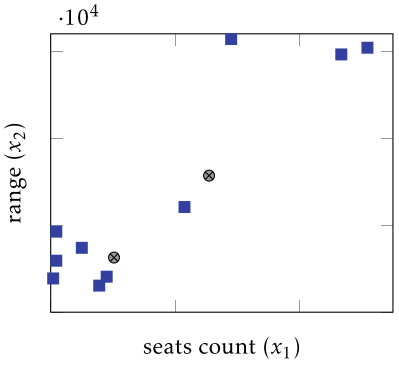
Example 1 (*Aircraft clustered binary*) Let us take an example of aircraft to explain how k-means clustering works. We collected ten popular aircrafts in Table 3.1. We have four columns: the name of the aircraft, the distance the aircraft can reach in one full tank, the number of seats, and the type of the aircraft. The first and last columns are used for the description only. The second and third columns are our features. We can plot them to see how easily we can divide them into clusters. A plot of the range of distances from the aircraft and seat count is shown in Fig. 3.1. There are two groups, one in the top right part of the plot and one in the bottom left part of it. It looks like an easy task to see that we have two types of aircraft: one with small seat

Table 3.1 Aircraft divided by type, range and seats count

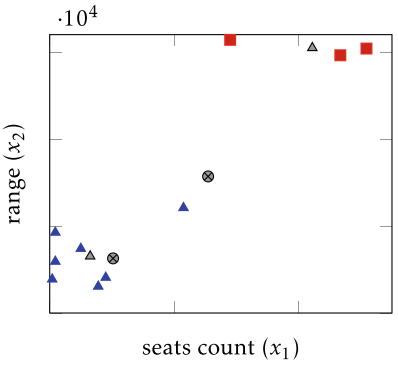
Aircraft name	Distance range (km)	Seats count	Aircraft type
Cesna 510 Mustang	1940	4	Private jet
Falcon 10/100	2960	9	Private jet
Hawker 900/900XP	4630	9	Private jet
ATR 72-600	1528	78	Medium size aircraft
Bombardier Dash 8 Q400	2040	90	Medium size aircraft
Embraer ERJ145 XR	3700	50	Medium size aircraft
Boeing 747-8	14,815	467	Jet airliner
A380-800	15,200	509	Jet airliner
Boeing 787-8	15,700	290	Jet airliner
Boeing 737-900ER	6045	215	Jet airliner



(a) Airctafts range and seats count on a chart



(b) Randomly set centroids before iteration one



(c) Final centroids for aircraft example using k-means method

Fig. 3.1 Aircraft example clustering

count and short distance range, and the second which are huge aircraft that can fly long distances. Using k-means, the steps we have described would look as follows: Before moving on to the first step of the method, we should normalize the data set to move the data to a common scale for each feature (Table 3.2a). In the first step of the method, we need to generate some random centroids. Let us assume that we have two centers randomly chosen for $v1 = (0.2, 0.2)$ and $v_2 = (0.5, 0.5)$. As we can see, one centroid is quite close to a group of objects in the bottom-left corner. The second centroid is almost in the middle of the feature space (see Fig. 3.1b). Let us calculate the membership matrix with both centroids. For the first object $x^{(0)}$, we calculate the following distances:

$$d(x^{(0)}, c_1) = d((0.0078, 0.1235), (0.2, 0.2)) = 0.20678,$$

$$d(x^{(0)}, c_2) = d((0.0078, 0.1235), (0.5, 0.5)) = 0.6196.$$

Table 3.2 Aircrafts data set normalized

Aircraft name	Distance range (km)	Seats count
Cesna 510 Mustang	0.007859	0.123567
Falcon 10/100	0.017682	0.188535
Hawker 900/900XP	0.017682	0.294904
ATR 72-600	0.153242	0.097325
Bombardier Dash 8 Q400	0.176817	0.129936
Embraer ERJ145 XR	0.098232	0.235669
Boeing 747-8	0.917485	0.943631
A380-800	1.000000	0.968153
Boeing 787-8	0.569745	1.000000
Boeing 737-900ER	0.422397	0.385032

The distance is shorter for the first centroid. It means that we should binary assign the membership of the first object into the first cluster. A different case is for object x_7:

$$d(x^{(0)}, c_1) = d((0.9174, 0.9436), (0.2, 0.2)) = 1.03333,$$

$$d(x^{(0)}, c_2) = d((0.9174, 0.9436), (0.5, 0.5)) = 0.60918.$$

This object should be assigned to the second cluster. After going through all the objects in the first iteration, we get the membership matrix as

$$U_1 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}.$$

The new centroids need to be calculated in the next step. Using Eq. 3.3, we get new centroids:

$$v_1 = \frac{\begin{bmatrix} 0.47151277 & 1.06993631 \end{bmatrix}}{6} = \begin{bmatrix} 0.07858546 & 0.17832272 \end{bmatrix},$$

$$v_2 = \frac{\begin{bmatrix} 2.90962672 & 3.29681529 \end{bmatrix}}{4} = \begin{bmatrix} 0.72740668 & 0.82420382 \end{bmatrix}.$$

In the next iterations, the membership matrix does not change, but the centroids still move. The reason for the absence of change is a very simple example and the small data set used. Finally, we get

$$v_1 = \begin{bmatrix} 0.12770138 & 0.20785259 \end{bmatrix},$$

$$v_2 = \begin{bmatrix} 0.82907662 & 0.97059448 \end{bmatrix}.$$

The final centroids are given in Fig. 3.1c. The centroids moved from the first position, especially the second centroid moved from about the middle of the space into the right-top corner. This seems logical because one of the groups of objects is placed in this area.

3.1.2 *Fuzzy C-Means*

The main concept of fuzzy clustering is to assign objects to a cluster using fuzzy logic. In the previous method, the column of the membership matrix contains one positive value, and the rest is filled with zeros. In fuzzy clustering, the values in columns are between zero and one. Each column sums up to one. There are many implementations of fuzzy clustering. One of the most popular is the ISODATA method [1], where ISODATA stands for the Iterative Self-Organizing Data Analysis Technique. Other popular fuzzy clustering solutions are Gustafson-Kessel or Fuzzy Maximum Likelihood Estimates method [2]. In this section, we implement the ISODATA method as an example of fuzzy clustering.

The fuzzy clustering method process looks the same as in hard clustering. The differences are in the way the centroids and the membership matrix are calculated. The elements of the membership matrix can be calculated with Eq. 3.4.

$$\mu_{ik} = \left(\sum_{j=1}^{c} \left(\frac{d(x_k, v_i)}{d(x_k, v_j)} \right)^{\frac{2}{m-1}} \right)^{-1}. \tag{3.4}$$

In Eq. 3.4, $d(x_k, v_i)$ is the distance between an object and the centroid. We calculate the if for each centroid. For two and more centroids, one value of the equation $\frac{d(x_k, v_i)}{d(k, v_j)}$ is always 1.0, because we divide the same distance values. In Python, it can be implemented as shown in the Listing 3.5. For a given object x and centroid number i. We use the same method for calculating the distance as in HCM. It is again the Euclidean distance of two objects in a feature space. The variable m is called a fuzzifier and allows us to flatten the plot of the fuzzy membership function. It is usually set at 2.

```
def calculate_u_fcm(x, centers, group_id):
    """
    This method calculates membership of the object x.

    :param x: object that we want to set the cluster
    membership id
    :param centers: centroids
    :param group_id: cluster id
    :return: Membership matrix
    """
    distance_centers = 0
    for group in range(groups):
        if group != group_id:
            distance_centers += calculate_distance
```

```
            (x, centers[group])
    distance_sum = 1.0 + (calculate_distance(x, centers
    [group_id]) / distance_centers) ** m
    return distance_sum ** -1
```

Listing 3.5 FCM centers calculation method

The centroids are calculated a bit differently compared to HCM. We add the fuzzifier variable here and this is actually the only difference between both equations (see Eq. 3.5).

$$v_i = \frac{\sum_{k=1}^{M} (\mu_{ik}^{(t)})^m x_k}{\sum_{k=1}^{M} (\mu_{ik}^{(t)})^m}. \tag{3.5}$$

The difference in Python implementation is none if we assign the value of 2 to the variable m. In the Example 3 for simplification, we set $m = 2$.

Example 2 (*Aircraft fuzzy clustered*) Having the same random centroids as in the previous example, we use Eq. 3.4 to calculate the membership values:

$$\mu_{01} = \left(1^2 + \frac{0.20678^2}{0.6196}\right)^{-1} = (1 + 0.11135)^{-1} = 0.8998,$$

$$\mu_{11} = \left(1^2 + \frac{0.6196^2}{0.20678}\right)^{-1} = 0.1002.$$

The values sum to 1 and are not binary compared to k-means. It is more precise and can give us more information. Take a look at the values of the last four objects in the membership matrix after the first iteration.

$$U_1 = \begin{bmatrix} 0.899 & 0.908 & 0.866 & 0.956 & 0.977 & 0.952 & 0.257 & 0.276 & 0.247 & 0.186 \\ 0.100 & 0.091 & 0.133 & 0.043 & 0.022 & 0.047 & 0.742 & 0.723 & 0.752 & 0.813 \end{bmatrix}.$$

The values of the last four objects are not polarized as the values of the other objects. We see that some objects are assigned to the cluster with confidence between 72 and 81%. What if the membership value is a two-cluster case and is close to 50%? This might mean that we should increase the number of clusters, especially when we have more such objects. We can also assume that such an object is just noise. The centroids are calculated similarly, and for the fuzzy clustering example, the values can be calculated as

$$v_1 = \frac{\left[0.61948325 \;\; 1.11511858\right]}{5.4026} = \left[0.11466 \;\; 0.2064\right],$$

$$v_2 = \frac{\left[1.63266 \;\; 1.8573\right]}{2.343648} = \left[0.6966 \;\; 0.79248\right].$$

The final centroids are a bit different from the k-means. Both centroids look a bit more precise because the membership values are not binary:

$$v_1 = \left[0.11240531\ 0.19895848\right],$$

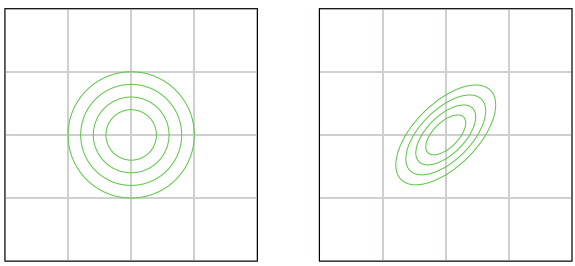
$$v_2 = \left[0.83355666\ 0.96018678\right].$$

The final membership matrix assigns the objects x_7, x_8, x_9 to the second cluster with higher confidence. The last object is the one placed in the center of the feature space in Fig. 3.1.

$$U_{\text{final}} = \begin{bmatrix} 0.988 & 0.992 & 0.983 & 0.990 & 0.992 & 0.998 & 0.006 & 0.019 & 0.077 & 0.792 \\ 0.011 & 0.007 & 0.016 & 0.009 & 0.007 & 0.001 & 0.993 & 0.980 & 0.922 & 0.207 \end{bmatrix}.$$

3.1.3 Possibilistic C-Means

The possibilistic distributed clustering was introduced in [3] and is a modification of the hard version. We used the fuzzy version in the first phase of the PCM to avoid total randomness. PCM works better compared to FCM and HCM if the data we have consist of noise data, but to be more robust, it should be preceded by a few iterations of the FCM method. The first difference that is visible is the way the distances are measured. In PCM we use the Mahalonobis distance instead of the Euclidean measure. It is known that the Mahalanobis measure often gives better results. In this case, a better means that we do not care about the standardization of the data if we have more than two features. The general difference can also be plotted as shown in Fig. 3.2. The covariance matrix standardizes the data set taking into account the correlation. This means that the redundant information is not taken into consideration, as it is in Euclidean distance. The membership function is defined as



(a) Euclidean distance (b) Mahalanobis distance

Fig. 3.2 Distance measures difference comparison

$$\mu_{ik} = \left(1 + \left(\frac{D_{ikA}}{\eta_i}\right)^{\frac{2}{m-1}}\right)^{-1}, \tag{3.6}$$

where D_{ikA} is the Mahalonobis distance and η is a value that is the membership possibility for each cluster. The membership function can be implemented in Python as in the Listing 3.6.

```
def calculate_u_pcm(membership, centers):
    """
    This method calculates membership of the object x.

    :param x: object that we want to set the cluster membership id
    :param centers: centroids
    :return: Membership matrix
    """
    new_membership = np.zeros((len(data_set), groups))
    for group in range(groups):
        mah_distances = calculate_mah_distance(group, centers)
        group_eta = calculate_eta(membership, group, mah_distances)
        new_membership[:, group] = (1.0 + (mah_distances / group_eta)) ** -1
    return new_membership
```

Listing 3.6 Membership function Python implementation

The Mahalanobis distance is calculated as

$$D^2_{ikA} = ||x_k - v_i||^2_A = (x_k - v_i)^T A (x_k - v_i). \tag{3.7}$$

The matrix A is the covariance matrix that flattens the distance between the objects as shown in Fig. 3.2. It can be calculated in Python as in the Listing 3.7. The matrix is used between two distance measures of two objects.

```
def calculate_A():
    """
    This method calculates the covariance Matrix

    :return: Covariance matrix A
    """
    variance = np.var(data_set, axis=0)
    ABcov = np.cov(data_set[:, 0] * data_set[:, 1])
    R = np.array([[variance[0], ABcov], [ABcov, variance[1]]])
    return R ** -1
```

Listing 3.7 Covariance matrix calculation for PCM method

Finally, the Mahalanobis implementation can be done in Python as in Listing 3.8.

```
def calculate_mah_distance(group, centers):
    """
    This method calculates the Mahalanobis distance

    :param group: group id
    :param centers: centroids
```

```
    :return: Mahalanobis distance
    """
    dmc = data_set - centers[group]
    dmca = np.dot(data_set - centers[group], A)

    distances = lambda dmc, dmca: [np.dot(dmca[i], dmc
    [i]) for i in range(dmc.shape[0])]
    return distances(dmc, dmca)
```

Listing 3.8 Mahalanobis distance implementation

The η parameter can be set the same for every cluster or calculated separately for each cluster. If we want to calculate it separately, we should use the equation:

$$\eta_i = \frac{\sum_{k=1}^{M} (\mu_{ik})^m D_{ikA}^2}{\sum_{k=1}^{M} (\mu_{ik})^m}. \tag{3.8}$$

The η also uses the Mahalanobis distance. We can think of this parameter as the importance factor of a cluster. The parameter η can be fixed for each object or calculated for each case separately, as shown in Eq. 3.8. The simple implementation of η is given in the Listing 3.9.

```
def calculate_eta(membership, group, mah_distances):
    """
    This method calculates the eta parameter

    :param membership: membership matrix
    :param group: group id
    :param mah_distances: Mahalanobis distance
    :return: eta parameter
    """
    return np.sum((membership[:, group] ** m) *
    mah_distances, axis=0) / np.sum(membership[:, group]
    ** m, axis=0)
```

Listing 3.9 PCM eta parameter calculation

The algorithm is divided into two sections. The variable F is the number of iterations that are executed before we go into the possibilistic method. The first part is a copy-pasting of the FCM method from the previous section. The second part looks a bit different.

```
def cluster_pcm(membership, centers):
    """
    This is the main PCM clustering method

    :param membership: global membership matrix
    :param centers: global centroids
    :return: membership matrix, centroids
    """
    new_centers = centers
    new_membership = membership
    for f in range(F):
        membership = []
        for i in range(len(data_set)):
            membership_vector = []
            for k in range(groups):
                membership_vector.append(calculate_u_fcm
                (data_set[i], new_centers, k))
            membership.append(membership_vector)
        new_centers = calculate_new_centers(membership)
```

```
        new_membership = np.array(membership)

    difference_limit_not_achieved = True
    while difference_limit_not_achieved:
        new_membership = calculate_u_pcm(new_membership,
new_centers)
        old_centers = new_centers
        new_centers = calculate_new_centers(
new_membership)

        if get_centers_difference(old_centers,
new_centers) < error_margin:
            difference_limit_not_achieved = False
    return new_membership, new_centers
```

Listing 3.10 Main PCM method implementation

The error rate is a stop criterion that depends here on the changes of the centroids. Depending on the size of the data set, the differences in membership values are very often made on each iteration, and if we sum up all the changes, it might be that we never reach the stop criterion. The differences are calculated on the basis of the centroids' changes rather than the membership matrix changes.

Example 3 (*Aircraft possibilistic clustering*) In this example, we use the same data set as in the previous two sections. The fuzzy part of the method returns after two iterations of the membership matrix as follows.

$$U_{\text{fcm}} = \begin{bmatrix} 0.980 & 0.988 & 0.976 & 0.983 & 0.986 & 0.998 & 0.056 & 0.082 & 0.066 & 0.655 \\ 0.019 & 0.011 & 0.023 & 0.016 & 0.013 & 0.001 & 0.943 & 0.917 & 0.934 & 0.344 \end{bmatrix}.$$

The centroids are already set in a good position after just two iterations:

$$v_1 = \begin{bmatrix} 0.1042 & 0.1944 \end{bmatrix},$$

$$v_2 = \begin{bmatrix} 0.8096 & 0.9445 \end{bmatrix}.$$

Both are adjusted with the PCM part of the algorithm. In the first step of the PCM part, we should calculate the A matrix that uses the entire data set, because we need to calculate the correlation between each object. The matrix for our data is the following:

$$A = \begin{bmatrix} 7.89464944 & 6.69665317 \\ 6.69665317 & 7.75894855 \end{bmatrix}.$$

The second step is to calculate the Mahalonobis distances. For the first object, it is

$$D_{00} = \left(\begin{bmatrix} 0.007859 \\ 0.123567 \end{bmatrix} - \begin{bmatrix} 0.1042 \\ 0.1944 \end{bmatrix} \begin{bmatrix} 0.007859 & 0.123567 \end{bmatrix} \right) * \\ * \begin{bmatrix} 7.89464944 & 6.69665317 \\ 6.69665317 & 7.75894855 \end{bmatrix} \begin{bmatrix} 0.007859 \\ 0.123567 \end{bmatrix} - \begin{bmatrix} 0.1042 \\ 0.1944 \end{bmatrix} = 0.20355.$$

For the second cluster, the distance equals $D_{10} = 19.11895$ and is much greater than the distance for the first group. It is easy to assume that the closer distance also means a greater membership value. All distances are given in the following matrix:

$$D_{ikA} = \begin{bmatrix} 0.2035 & 19.1189 \\ 0.0661 & 17.4034 \\ 0.0210 & 15.1151 \\ 0.0283 & 16.4173 \\ 0.0112 & 15.213 \\ 0.0102 & 14.6471 \\ 17.7387 & 0.0906 \\ 20.2641 & 0.3508 \\ 11.77 & 0.2998 \\ 1.8938 & 6.5137 \end{bmatrix}.$$

The last step before we get the membership value is the cluster probability η parameters that need to be calculated. We have two η values, one for each cluster. The equation consists of two sums [see (3.8)]. For η_0 we take the Mahalnobis distance and multiply it by the square of the membership values of the first cluster. The sum is divided next just by the sum of the squares of the membership values of the first cluster. Respectively for η_1.

$$\eta_0 = \frac{0.98056735^2 * 0.2035 + 0.9883^2 * 0.0661 + \cdots + 0.6558^2 * 1.8938}{0.98056735^2 + \cdots + 0.6558^2} = 0.2216,$$

$$\eta_1 = \frac{0.0194^2 * 19.1189 + \cdots + 0.3442^2 * 6.5137}{0.0194^2 + \cdots + 0.3442^2} = 0.5268.$$

The probability parameter is greater for the second cluster than for the first. It might be like that because the objects in the second cluster are not so close to the centroid compared to the first cluster. The fourth step is to calculate the membership matrix. Having the η and Mahalanobis distance values, it is easy to calculate. The calculation is as follows:

$$\mu_{00} = \left(1 + \frac{0.2035}{0.2216}\right)^{-1} = 0.304852.$$

The final membership matrix is shown below.

$$U_p = \begin{bmatrix} 0.5425 & 0.9183 & 0.9911 & 0.9839 & 0.9975 & 0.9979 & 0.0002 & 0.0001 & 0.0004 & 0.0135 \\ 0.0008 & 0.0009 & 0.0012 & 0.001 & 0.0012 & 0.0013 & 0.9713 & 0.6927 & 0.7554 & 0.0065 \end{bmatrix}.$$

The centroids v_1 and v_2 are calculated in the same way as in the previous two methods, and for PCM we get

$$v_1 = \begin{bmatrix} 0.0899 & 0.1857 \end{bmatrix},$$

$$v_2 = \begin{bmatrix} 0.8378 & 0.9656 \end{bmatrix}.$$

Table 3.3 Membership matrix for three distributed clustering methods

	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
U_{h0}	1	1	1	1	1	1	0	0	0	1
U_{h1}	0	0	0	0	0	0	1	1	1	0
U_{f0}	0.9881	0.9928	0.9839	0.9902	0.9921	0.9985	0.0061	0.0198	0.0771	0.7926
U_{f1}	0.0119	0.0072	0.0161	0.0098	0.0079	0.0015	0.9939	0.9802	0.9229	0.2074
U_{p0}	0.5425	0.9183	0.9911	0.9839	0.9975	0.9979	0.0002	0.0001	0.0004	0.0135
U_{p1}	0.0008	0.0009	0.0012	0.001	0.0012	0.0013	0.9713	0.6927	0.7554	0.0065

Both are not as different compared to previous methods, but the membership matrix is the totally different.

3.1.3.1 Comparison

Comparison of the results of distributed methods can be major if we compare HCM to FCM, or minor if we compare FCM to PCM. It is true that the more fuzzy or possibilistic the method is, the more precise results we get. An overview of the membership matrix is given in Table 3.3. What is important, based on the membership values, is that all results assign each object to the same cluster in each of the three methods. It means that it does not differ so much for such a small data set, but can give us more information, especially to the number of clusters. More values closer to 0.5 in the fuzzy set or closer to 0 in the case of the possibilistic method, then the probability of increasing the number of clusters is higher.

3.2 Hierarchical Clustering

Hierarchical clustering methods are based on partitioning. A partition is just another name for a cluster used in hierarchical methods. We can imagine a nested data set as shown in Fig. 3.3. The main data set includes a few smaller sets that include a few smaller sets and so on. On the other hand, we can define it as a merge or fusion of the smallest data sets into bigger ones until we reach the main data set. Let us imagine that we have a list of some life forms as shown in Table 3.4. We can distinguish each animal on the basis of a few criteria. It is easy to see that animals can be presented as a tree of types, subtypes, and so on. We can also present such life forms based on some other features, such as the nested set in Fig. 3.3. Each of the life forms listed in Table 3.4 can be assigned to the group *Animals*. We have four mammals, four reptiles, and more on the list. Each group, as *Mammals*, has two different types like *Marine* and *Land* mammals. In other words, starting from the main cluster *Animals*

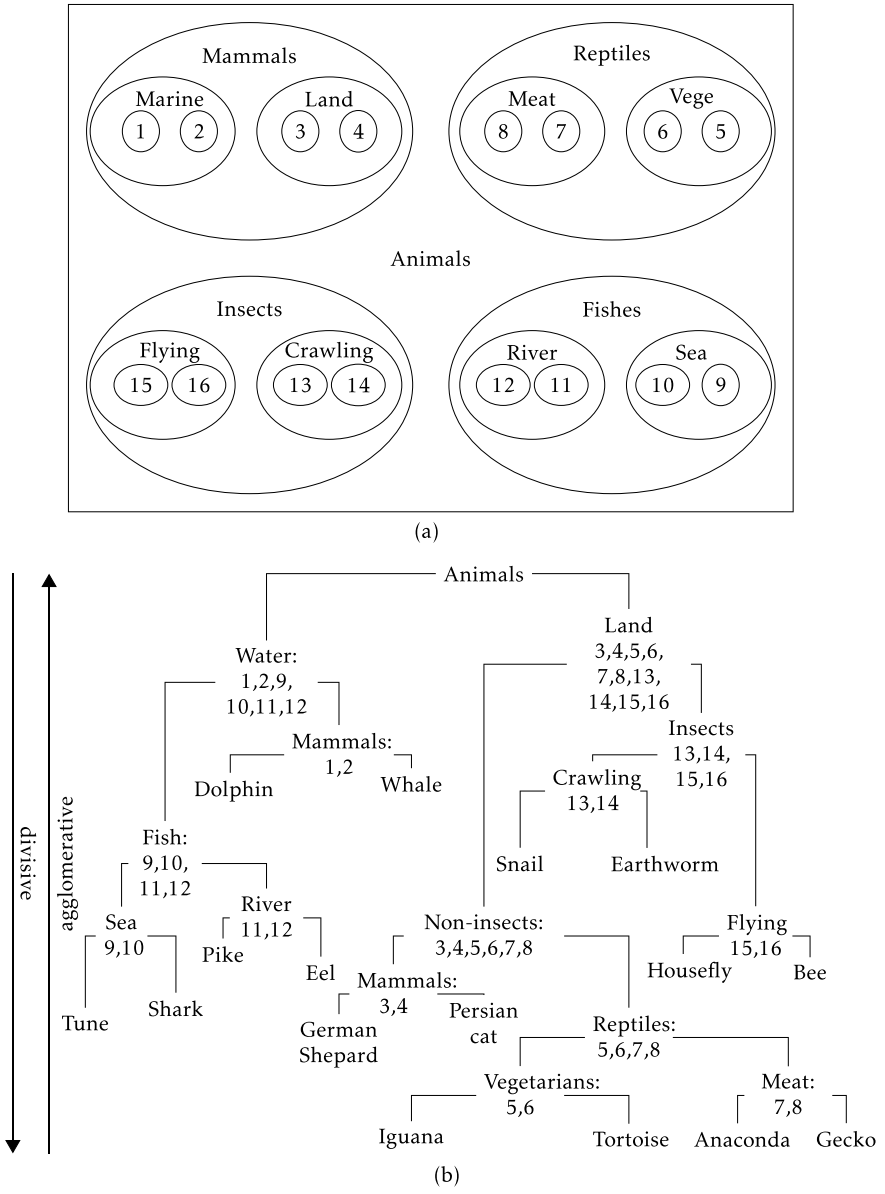


Fig. 3.3 Example shown in Table 3.4 as nested sets (**a**) and dendrogram (**b**)

we can divide it into smaller groups at each stage until we divide it into clusters with one animal each.

Hierarchical methods can do it both ways. This means that we can also merge small partitions into a larger cluster [4]. Hierarchical methods can be divided into

Table 3.4 A few life forms that can be grouped by the criteria

Id	Group	Subgroup	Animal name	Id	Group	Subgroup	Animal name
1	Mammals	Marine	Dolphin	9	Fishes	Sea	Shark
2	Mammals	Marine	Whale	10	Fishes	Sea	Tune
3	Mammals	Land	German Sheppard	11	Fishes	River	Pike
4	Mammals	Land	Persian cat	12	Fishes	River	Eel
5	Reptiles	Vegetarian	Iguana	13	Insects	Crawling	Ladybug
6	Reptiles	Vegetarian	Tortoise	14	Insects	Crawling	Earthworm
7	Reptiles	Meat eaters	Anaconda	15	Insects	Flying	Bee
8	Reptiles	Meat eaters	Gecko	16	Insects	Flying	Housefly

two subtypes: agglomerative and divisive. The first one is about merging/aggregating partitions and the second is about dividing/splitting sets into smaller ones.

3.2.1 Agglomerative Clustering

Agglomerative clustering [5, 6] is also known as the "bottom-up" clustering method, because it merges smaller clusters into larger ones. The method is easy to implement and even easier to understand. Here, we use the Euclidean distance measure to create the distance matrix. The agglomerative clustering method is divided into three steps:

1. calculate current dendrogram distance matrix,
2. get lowest distance from matrix,
3. merge clusters/elements into clusters.

It is repeated until we have one cluster or reach the expected cluster count.

Distance matrix

In the agglomerative matrix, we calculate the distances between each object in the first step. The distance matrix is the size of $N \times N$, where N is the number of objects in the data set. Obviously, the diagonal values are equal to zero. The Listing 3.11 is one of the possible implementations of how to obtain the distance matrix.

```
def calculate_dendogram_distance_matrix_diana():
    """
    This method calculate the distance matrix for diana
    methods

    :return: distance matrix
    """
    distance_matrix=np.zeros((len(data_set), len(data_set
    )))
    for i in range(len(data_set)):
        for j in range(len(data_set)):
            distance_matrix[i, j] = calculate_distance
            (calculate_centroid(data_set[i]),
    calculate_centroid(data_set[j]))
    return distance_matrix
```

Listing 3.11 Distance matrix calculation for agglomerative clustering

The goal is to find the lowest distance between two objects. These two are the most similar to each other and should be merged into one cluster.

```
def get_lowest_from_distance_matrix(distance_matrix):
    """
    This method gets the lowest value from the distance
    matrix

    :param distance_matrix: current dendrogram
    :return: lowest value index
    """
    np.fill_diagonal(distance_matrix, np.inf)
    lowest_indexes = np.unravel_index(np.argmin(
    distance_matrix, axis=None), distance_matrix.shape)
    np.fill_diagonal(distance_matrix, 0)
    return lowest_indexes
```

Listing 3.12 Lowest distance in distance matrix implementation

The Listing 3.12 shows how to use NumPy to find the lowest value in a matrix and get the indices.

Merging two clusters

The merging part adds a new level to the dendrogram and sets the new clusters list on it as in the Listing 3.13. We can also implement it in a different way and avoid the part of level creation.

```
def merge_elements(current_dendrograms, merged_list, i):
    """
    This method adds to the current dendrogram new object
    (elements)

    :param current_dendrograms: current dendrogram
    :param merged_list: merged child nodes list
    :param i: the dencdrogram level
    :return: merged child nodes list
    """
    if isinstance(current_dendrograms[i][0], type(np.
    array([]))):
        for iter in range(len(current_dendrograms[i])):
            merged_list.append(current_dendrograms[i][
    iter])
    else:
```

Table 3.5 Clusters distance measure methods

Method name	Equation	
Single linkage	$d_{12} = \min_{i,j} d(X_i, Y_j)$	(3.9)
Complete linkage	$d_{12} = \max_{i,j} d(X_i, Y_j)$	(3.10)
Average linkage	$d_{12} = \frac{1}{kl} \sum_{i=1}^{k} \sum_{j=1}^{j} d(X_i, Y_j)$	(3.11)
Centroid method	$d_{12} = d(\bar{x}, \bar{y})$	(3.12)

```
14            merged_list.append(current_dendrograms[i])
15    return merged_list
```

Listing 3.13 Merging two clusters in agglomerative clustering

We can stick to the current dendrogram level and merge based on it until we reach the stop criterion, but saving each merge in the dendrogram allows us to make decisions based on the history of the merges and gives a better overview of the method after it is finished.

Distance matrix between clusters

We could stop here if we are fine with the number of clusters, but in most cases, we would like to proceed. Going back to step one of the method, we need to calculate the distance matrix again. The only question here is how to calculate the distance between a cluster that consists of one object and a cluster with more objects. There are methods to calculate the distances between clusters with any number of objects. Some of such methods are given in Table 3.5. The distances can be calculated as the minimum distance between two objects from each cluster. This method is called the single linkage method. The opposite is the maximum distance that is used in the complete linkage method. We also have an average distance measure, where we take the average distance between all objects. A similar one is based on the centroids, where the centroids are calculated as the average positions of all objects in a given cluster. Next, we calculate the distance between the centroids of both clusters. This method is implemented in Listing 3.14 as the simple one and is based on the knowledge from the previous chapter where centroids were used for the calculation of the membership matrix.

```
1  def calculate_centroid(dendrogram_elements):
2      """
3      This method calculates the centroids for the current
       dendrogram part by merging the elements within this
       subset
4
5      :param dendrogram_elements: subset of the dendrogram
6      :return: centroids
7      """
8      if type(dendrogram_elements) is list:
9          sumof=np.zeros(len(dendrogram_elements[0]))
10         for iter in range(len(dendrogram_elements)):
11             sumof=np.add(sumof,np.array(
       dendrogram_elements[iter]))
```

```
        if sumof.shape == (len(data_set[0]),len(data_set[0])):
            pass
        return np.divide(sumof*1.0, len(dendrogram_elements)*1.0)
    else:
        return dendrogram_elements
```

Listing 3.14 Centroid calculation for the agglomerative clustering method

Building the dendrogram

The dendrogram consists of levels consisting of nodes. We create a new node for each iteration. To create a new node, we just merge two clusters into one cluster, as shown in the Listing 3.15. This part works together with the main function of the agglomerative method in Listing 3.16.

```
def set_current_dendrogram(current_dendrograms, dendrograms_hist, i, j):
    """
    This method set the current dendrogram and change the dendrogram history

    :param current_dendrograms: current dendrogram
    :param dendrograms_hist: dendrogram changes record
    :param i: id of one of the chosen cluster
    :param j: id of one of the chosen cluster
    :return: current dendrogram and history
    """
    elements = []
    hist = []
    current_hist = dendrograms_hist[-1]
    for iter in range(len(current_dendrograms)):
        if iter != i and iter !=j:
            elements.append(current_dendrograms[iter])
            hist.append(current_hist[iter])
    merged_elements = []
    merged_elements = merge_elements(current_dendrograms, merged_elements, i)
    merged_elements = merge_elements(current_dendrograms, merged_elements, j)
    elements.append(merged_elements)
    hist.append([current_hist[i],current_hist[j]])
    dendrograms_hist.append(hist)
    current_dendrograms = elements
    return current_dendrograms, dendrograms_hist
```

Listing 3.15 New dendrogram implementation

We loop over the number of elements in the current dendrogram that includes at the beginning just all objects. We subtract the count of the objects by 2 because we need only this number of iterations to create the top cluster with all objects in it.

```
def cluster_agg(current_dendrograms):
    """
    This main agglomerative clustering method

    :param current_dendrograms: global current dendrogram variable
    :return: clustering history
    """
```

```
    dendrograms_hist = [list(range(len(data_set)))]
    for iter in range(len(current_dendrograms)-1):
        distance_matrix =
    calculate_dendogram_distance_matrix
        (current_dendrograms)
        [i, j] = get_lowest_from_distance_matrix
        (distance_matrix)
        current_dendrograms, dendrograms_hist =
    set_current_dendrogram(current_dendrograms,
    dendrograms_hist, i, j)
    return dendrograms_hist
```

Listing 3.16 Agglomerative clustering main method

We use the `get_lowest_from_distance_matrix()` function to obtain the lowest distance and add the new node to the dendrogram. Finally, the current level is set as the one that needs to be merged.

Example 4 (*Aircraft agglomerative clustered*) In this example, we use the same data set as in the previous clustering examples. In agglomerative clustering, we start with many clusters that consist of one object. The first step is to calculate the distance matrix. The distance between x_0 and x_1 is

$$d(x_0, x_1) = \sqrt{(0.007859 - 0.017682)^2 + (0.123567 - 0.188535)^2} = 0.066.$$

The distance matrix in the first iteration is as follows:

$$D = \begin{bmatrix} 0 & 0.066 & 0.171 & 0.148 & 0.169 & 0.144 & 1.225 & 1.303 & 1.041 & 0.490 \\ 0.066 & 0 & 0.106 & 0.163 & 0.169 & 0.093 & 1.175 & 1.254 & 0.981 & 0.450 \\ 0.172 & 0.106 & 0 & 0.240 & 0.229 & 0.1 & 1.109 & 1.191 & 0.895 & 0.415 \\ 0.148 & 0.163 & 0.24 & 0 & 0.0402 & 0.149 & 1.140 & 1.215 & 0.994 & 0.394 \\ 0.169 & 0.169 & 0.229 & 0.0402 & 0 & 0.132 & 1.1 & 1.175 & 0.955 & 0.354 \\ 0.144 & 0.093 & 0.1 & 0.149 & 0.131 & 0 & 1.083 & 1.162 & 0.898 & 0.357 \\ 1.225 & 1.175 & 1.109 & 1.14 & 1.1 & 1.083 & 0 & 0.086 & 0.352 & 0.746 \\ 1.303 & 1.254 & 1.191 & 1.215 & 1.175 & 1.162 & 0.086 & 0 & 0.431 & 0.821 \\ 1.041 & 0.981 & 0.895 & 0.994 & 0.955 & 0.898 & 0.352 & 0.431 & 0 & 0.632 \\ 0.49 & 0.45 & 0.415 & 0.394 & 0.354 & 0.357 & 0.746 & 0.821 & 0.632 & 0 \end{bmatrix}.$$

The smallest distance value is 0.0402 and this is the distance between the object x_3 and x_4. We take these two and merge them into one cluster. The next step is to calculate the distance matrix again, but this time for the merged objects we use the centroid to measure the distance. Finally, we get the dendrogram as given in Fig. 3.4. We see that the levels are not exactly drawn from the bottom in the exact time order when these clusters were created. We merged the third and fourth objects that are shown in the bottom right of the figure as the first one, but because of the complexity of other merges, it looks as if we merged the objects x_0, x_1, x_2 and x_5 as the first ones. This is not a mistake or error, because we still have the proper structure of the nodes.

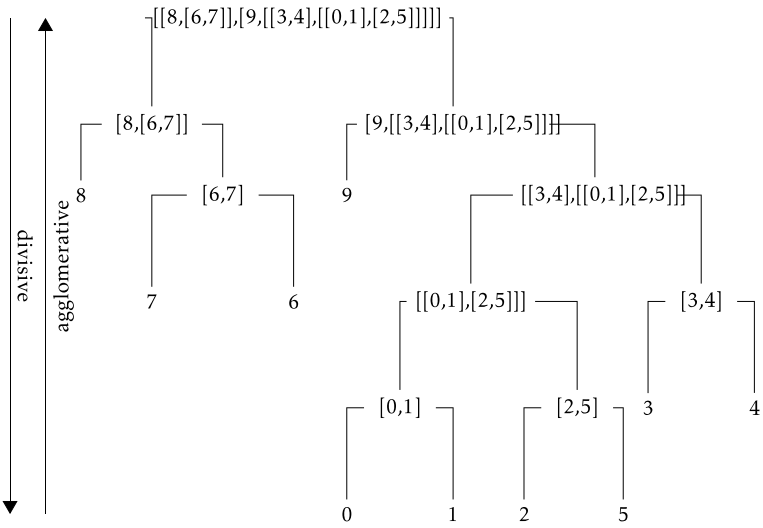


Fig. 3.4 Aircraft example dendrogram created using hierarchical clustering. The numbers are the aircraft ids

3.2.2 Divisive Clustering

Divisive clustering [7–9] is the opposite way to create clusters compared to agglomerative clustering. It is also known as Diana clustering. We divide one cluster into smaller ones. The divisive clustering method is divided into three steps:

1. calculate distance matrix in each cluster,
2. get highest distance average,
3. split clusters.

It is repeated until we have no cluster to be divided or the expected clusters' number is reached. The differences are in the second and third steps.

Choose cluster to split

The method of choosing the objects to split is implemented in the Listing 3.17. We find the highest differentiation cluster that we have set in the `highest_diff` variable. This is the cluster that will be split in the next step.

```
def choose_cluster(current_level, distance_matrix):
    """
    This method choose the next cluster to be divided

    :param current_level: dendrogram level
    :param distance_matrix: distance matrix
    :return: chosen cluster, difference, chosen cluster
    id
```

```
    """
    if type(current_level[0]) != list:
        both_idx = np.array(current_level)
        current = distance_matrix[both_idx[:,None],
    both_idx]
        diff = np.sum(current) / (current.shape[0] *
    current.shape[1] - len(current))
        return current_level, diff, 0
    highest_diff = 0
    cluster_id = 0
    for i in range(len(current_level)):
        both_idx = np.array(current_level[i])
        current = distance_matrix[both_idx[:, None],
    both_idx]
        diff = np.sum(current) / (current.shape[0] *
    current.shape[1]-len(current))
        if diff > highest_diff:
            highest_diff = diff
            cluster_id = i
    return current_level[cluster_id], highest_diff,
    cluster_id
```

Listing 3.17 Divisive clustering—choosing objects to split

Based on a threshold `diff` we divide the objects into `cluster1` and `cluster2` sets in Listing 3.18.

```
def split(split_cluster, distance_matrix, diff):
    """
    This method split the chosen cluster into two
    clusters

    :param split_cluster: cluster that needs to be split
    :param distance_matrix: distance matrix
    :param diff: difference of distances between objects
    within the cluster
    :return: two clusters
    """
    if len(split_cluster) == 2:
        cluster1 = [split_cluster[0]]
        cluster2 = [split_cluster[1]]
        return cluster1, cluster2
    split_threshold = diff
    both_idx = np.array(split_cluster)
    distances = distance_matrix[both_idx[:, None],
    both_idx]
    c1keys, c1counts = np.unique(np.argwhere(distances >
    split_threshold), return_counts=True)
    c2keys, c2counts = np.unique(np.argwhere(distances <=
     split_threshold), return_counts=True)
    cluster1_counts = dict(zip(np.array(split_cluster)[
    c1keys], c1counts))
    cluster2_counts = dict(zip(np.array(split_cluster)[
    c2keys], c2counts))
    cluster1 = []
    cluster2 = []
    choice = 0
    for item in split_cluster:
        if item not in cluster1_counts.keys():
            cluster2.append(item)
            continue
        if item not in cluster2_counts.keys():
            cluster1.append(item)
            continue
        if cluster1_counts[item] < cluster2_counts[item]:
            cluster2.append(item)
```

```
        elif cluster1_counts[item] > cluster2_counts[item]:
            cluster1.append(item)
        else:
            if choice == 0:
                cluster2.append(item)
                choice = 1
            else:
                cluster1.append(item)
                choice = 0
    return cluster1, cluster2
```

Listing 3.18 Split method in divisive clustering

The code is a bit long because we loop over the cluster and keep the history of the actions we do.

Build the dendrogram

The other methods except for the main are the same as in the agglomerative method. The main method of the divisive clustering is given in Listing 3.19. The divisive clustering is a bit more complex compared to the agglomerative because of the way we choose the cluster to split.

```
def cluster_diana():
    """
    This is the main method of diana cluster method

    :return: dendrogram history records
    """
    dendrograms_history=[]
    current_dendrograms=[list(range(len(data_set)))]
    distance_matrix = calculate_dendogram_distance_matrix_diana()
    while len(current_dendrograms) != len(data_set):
        current_level = current_dendrograms[-1]
        current_cluster, diff, cluster_id = choose_cluster(current_level, distance_matrix)
        cluster1, cluster2 = split(current_cluster, distance_matrix, diff)
        if type(current_level[0]) != list:
            current_dendrograms.append([cluster1, cluster2])
        else:
            rest = current_level.copy()
            rest.pop(cluster_id)
            rest.append(cluster1)
            rest.append(cluster2)
            current_dendrograms.append(rest)
        hist = [{"acesor": current_cluster, "childs": [cluster1,cluster2]}]
        dendrograms_history.append(hist)
    return dendrograms_history
```

Listing 3.19 Divisive clustering main method

In the main method code we invoke the distance matrix method once, choose the group to split, do the split, and in the last part of the code we save the changes made to the current dendrogram level. The changes are saved in the variable `hist`.

3.3 Density Based Clustering

The density-based clustering is based on the neighborhoods. DBScan is one of the density-based clustering methods that we cover in this chapter. It is similar to hierarchical clustering, since we calculate the distance matrix between each element. One of the advantages of the density-based methods is the lack of the parameter k. The number of clusters is one of the returned values of this method. On the other hand, the number of clusters depends on a parameter ϵ that defines the neighborhood of objects.

3.3.1 DBScan

In the method, we go through each element and count the number of neighborhood elements in a distance area. It is calculated as follows:

$$N_\epsilon : x_i | d(x_i, x_j) \leq \epsilon, \tag{3.13}$$

where x_i and x_j are two elements of the training data set and ϵ is the neighborhood distance. The distance is used to find the most similar objects in the feature space. This is one of the two parameters that we need to set in this method. The second parameter is the `min_points` parameter, which is about the number of neighbors that we expect to have at least not to be considered as the border object of the cluster. If the object does not have any neighbors, it is considered as noise. The method consists of the following steps:

1. calculate distance matrix,
2. get the closest element,
3. merge into a cluster if the distance is small enough.

It can also be used to find noise in the data set. The calculation of the distance matrix can be implemented in the same way as in the hierarchical clustering methods explained in the previous chapter.

Mark functions

The method goes through each object in the data set and assigns an object to one of the clusters or to the noise. We also need to have functions to set if we have checked an object or it still needs to be assigned. These types of methods for setting the status and checking the status of each object are implemented in the Listing 3.20.

```
1 def set_as_noise(membership, element_id):
2     """
3     This method sets an object as noise
4
5     :param membership: membership matrix
```

```
    :param element_id: measured object id
    :return: membership matrix
    """
    membership[element_id] = -1
    return membership


def set_visited(elements, membership, number_of_clusters):
    """
    This method sets an object as already checked/assigned

    :param elements: objects that we want to set as already checked
    :param membership: membership matrix
    :param number_of_clusters: the number of cluster to be set as checked in the membership matrix
    :return: membership matrix
    """
    for element_id in elements.keys():
        membership[element_id] = number_of_clusters
    return membership
```

Listing 3.20 Elements manipulation methods

The function `elements_in_area` returns the objects that are within the neighborhood, which means that the distance between the object and other objects is less than ϵ. The function `filter_visited` returns the distance vector where all objects that have been assigned are removed from the vector. Two functions that manipulate the status of an object: `set_visited` that sets the cluster number to the object, and `set_as_noise` that assigns the value -1 to the objects. The value -1 in this method means noise.

Closest objects

The closest objects are grouped into one cluster. We take one object randomly from the data set and find the objects where the distance is less than ϵ (see the Listing 3.21).

```
def get_closest_elements(distance_matrix, element_id):
    """
    This method calculates the closets objects to the one given as parameter

    :param distance_matrix: distance matrix
    :param element_id: measured object id
    :return: membership matrix
    """
    element_distances = distance_matrix[element_id]
    filtered = {}
    iter = 0
    for element in element_distances:
        if element < max_distance:
            filtered[iter] = element
        iter = iter + 1
    return filtered
```

Listing 3.21 Smallest distance elements function

The method is simple and uses the distance matrix and the `element_id` that is the object we are currently investigating. The function returns a vector of the closest objects.

Build clusters

The main part of the method is given in the Listing 3.22. We iterate randomly through the data set and check if the current objects are already marked with the cluster number or noise. If not, we find the closest objects to it.

```
def cluster_density(membership):
    """
    This is the main density clustering method

    :param membership: global membership matrix
    :return: membership matrix
    """
    number_of_cluster = 0
    distance_matrix = calculate_distance_matrix()
    element_ids = list(range(len(data_set)))
    random.shuffle(element_ids)
    for i in element_ids:
        if membership[i] != 0:
            continue
        closest = get_closest_elements(distance_matrix, i
    )
        if len(closest) < min_points:
            membership = set_as_noise(membership,i)
        else:
            membership = set_visited(closest, membership,
     number_of_cluster)
            number_of_cluster = number_of_cluster + 1
    return membership
```

Listing 3.22 Main clustering method

In the second loop, we go through the closest objects and check if the distance of the objects is less than ϵ. In case it is less than ϵ we set such an object with the `set_visited` function with the current cluster number. The number of clusters increases in each loop. In Fig. 3.5 we have two clusters marked with triangles and squares. The noise is the object without any other object in the neighborhood and is marked with a red x. The green triangle is the only border object, because there are fewer objects in the neighborhood than `min_points`.

Example 5 (*Aircrafts density clustered*) In this example, for comparison reasons of the quality metrics explained in the next section, we use the same data set as in the previous examples in the clustering part of this book.

In the first step, we calculate the distance matrix that is the same in the Example 4. Next, we choose one randomly chosen element and get the lowest distances. Let the chosen object be x_3. We get the following distance vector:

$$d_{x_3} = \left[0.14\ 0.16\ 0.23\ 0.0\ 0.04\ 0.14\ 1.14\ 1.21\ 0.99\ 0.39\right].$$

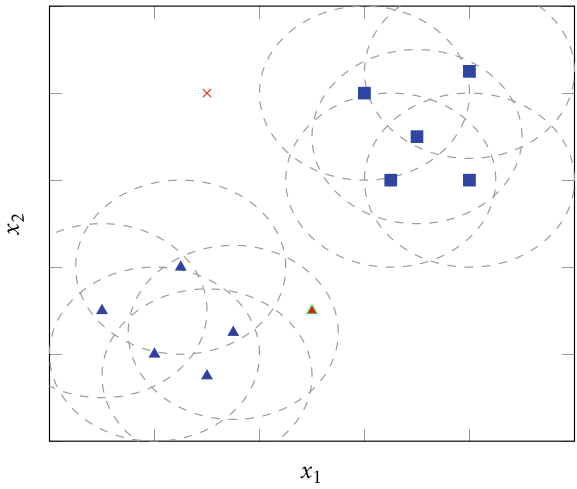


Fig. 3.5 DBScan explained with $\epsilon = 0.2$ and minimal points set to 2. Two clusters marked with blue triangles and squares. The green triangle is the border point. Outlier is marked with orange square

We include all objects where the distance is below the `max_distance` into a cluster. If the `max_distance` (ϵ) is 0.25 then we have four other objects within this cluster: $x_0, x_1, x_2, x_4, x_5, x_6$. We mark all as the first cluster. In the next step, we could randomly choose x_9. There are no other objects that are close enough:

$$d_{x_9} = \left[0.49\ 0.45\ 0.415\ 0.394\ 0.354\ 0.357\ 0.746\ 0.821\ 0.632\ 0\right].$$

This object is marked as noise. In the next loop, we choose x_7:

$$d_{x_7} = \left[1.303\ 1.254\ 1.191\ 1.215\ 1.175\ 1.162\ 0.086\ 0\ 0.431\ 0.821\right].$$

We take an object as border object if it is below `min_points`. This cluster consists of two border objects, because there is only one neighborhood object for x_6 and x_7. The remaining object x_8 is marked as noise because the distance from other objects is greater than ϵ as in the case of x_9. The final result looks as given in Fig. 3.6. We have two objects marked as noise and two clusters. The first cluster is marked with blue triangles and consists of five objects. The second one is marked with red squares and consists of two border objects.

3.3.2 Comparison to Hierarchical and Distributed Clustering

The comparison of all three group methods can be done on a few levels. The advantage of distributed methods is the simplicity and different variations as a fuzzy and possible approach. The disadvantage is the k value that must be set before the clustering is carried out and limit the number of clusters to k. In hierarchical methods, we can

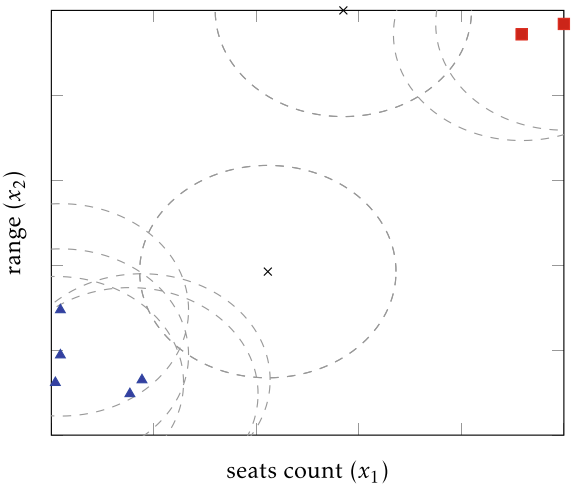


Fig. 3.6 DBScan used on Aircraft data set with $\epsilon = 0.25$ and `min_points` set to 2

choose the number of clusters that are in our interest. We can choose from many small clusters or choose two clusters. The hierarchical methods are parameterless. The density methods depend on two parameters, where ϵ if greater, there is less cluster we get. A lower value of ϵ generates a greater number of clusters. The disadvantage is that it might be hard to predict how many clusters we get, and we still need to set two parameters.

3.4 Quality and Validation Methods in Unsupervised Learning

Finding the best clustering method is not an easy task. To make this task easier, we can use multiple validation methods. The most important factors are the homogeneity and heterogeneity of a clustering method. Homogeneity means that each element within a cluster should be similar to each other. The more each element is similar to each other, the better method. An example of a similarity measure can be a distance method such as the Euclidean distance. Heterogeneity is about the difference between each cluster. Elements of each cluster should be varied compared to elements of the other clusters. The question here is what the value of similarity means that the method is good? We can compare the similarities between each method and distinguish which method gives the best results. Another possibility is to use one of the commonly known validation methods. In this section, we explain the group of validation methods in this section. Before we come to this point, we answer another tricky question: how many clusters should we have? In previous examples, we could decide on the basis of a plot of elements of a given problem. Real-world clustering problems can be a

bit more complex, and it can be difficult to choose the best number of clusters. In the next section, we give some advice on how to choose the right number of clusters.

3.4.1 Heterogeneity and Homogeneity

Some of the quality metrics are known as separation and dispersion measures. Such metrics tell us, for example, how far objects are from the center of a cluster. This measure should be the smallest possible to consider the clustering method to be good. The ideal case would be to have the objects very close, or even at the same point as the center. In real-world examples, it is hard for all objects to be so close to the center of a cluster. The two homogeneity metrics that we explain here are marked as σ_1 and σ_2. Both are related to the differences within each cluster. The differences are known as dispersion measures within a cluster. As we do some calculations within the cluster, we need to refer to the cluster center. The equation of the average object dispersion is as follows:

$$\sigma_1(c_i) = \frac{1}{m} \sum_{x_1, x_2 \in c_i} d^2(x_1, x_2), \tag{3.14}$$

where the m is defined as

$$m = \frac{(n_i - 1)n_i}{2}. \tag{3.15}$$

The n_i is the count of objects within the i-th cluster. If we have two clusters, we calculate two dispersion measures σ_1, one for each cluster. The value of this measure is the sum of the Euclidean distances squared between each object within the cluster divided by m. It can be easily implemented in Python as shown in Listing 3.23.

```
def calculate_sigma_1(membership):
    """
    This method calculates sigma1 quality metric

    :param membership: membership matrix
    :return: sigma1 values
    """
    sigma_1 = []
    unique_labels = len(membership[0])

    for label_id in range(unique_labels):
        ids = np.where(membership[:, label_id] == 1)[0]
        if len(ids) == 1:
            sigma_1.append(1.0)
            continue
        else:
            m = (len(ids) - 1.0) * len(ids) / 2.0
        elements = data_set[ids]
        sigma = (1.0 / m)
        for element_x_1 in range(len(elements)):
            for element_x_2 in range(len(elements)):
                if element_x_1 == element_x_2:
                    continue
                distance = calculate_distance(elements
```

```
                    [element_x_1], elements[element_x_2])
                    if distance != 0:
                        sigma = sigma + (distance ** 2)
            sigma_1.append(sigma)
    return sigma_1
```

Listing 3.23 σ_1 index calculation method

The method `calculate_sigma_1` uses some private variables like `__assignation` and `__points`. The first variable is the result of the clustering method and is a matrix of size $k \times m$, where m is the number of objects and k is the number of clusters. In the case of k-means, we have the matrix filled with values from the set of 0, 1. We have three for loops as we calculate the distance between two points for each center (`unique_labels`). We use the same distance measure as described in Chap. 2. The second dispersion measure is marked as σ_2. Here, we calculate the distance power between each object x within a cluster and the center of the cluster c_i. We divide the result by the count of objects within the cluster. The equation is as follows:

$$\sigma_2(c_i) = \frac{1}{n_i} \sum_{x \in c_i} d^2(x, c_i). \tag{3.16}$$

It looks a bit simpler compared to σ_1. In both cases, the smallest value demonstrates a better clustering result. The metric can be implemented as shown in Listing 3.24.

```
def calculate_sigma_2(centers, membership):
    """
    This method calculates sigma2 quality metric

    :param centers: clusters centroids
    :param membership: membership matrix
    :return: sigma2 values
    """
    sigma_2 = []

    for center_id in range(len(centers)):
        ids = np.where(membership[:, center_id] == 1)[0]
        elements = data_set[ids]
        sigma = 1.0 / len(ids)
        for element_id in range(len(elements)):
            distance = calculate_distance(elements
            [element_id], centers[center_id])
            if distance != 0:
                sigma = sigma + (distance) ** 2
        sigma_2.append(sigma)
    return sigma_2
```

Listing 3.24 σ_2 index calculation method

In this case we iterate only through two loops: the centroids and elements within centroid's cluster. The distance measure is exactly the same as in the previous metrics. We can also use different distance metric, but we use the Euclidean distance as the most common one. Measures similar to σ_1 and σ_2 are the total dispersion measures. These metrics provide a better understanding of the recurrence of objects within a cluster and feature space. Both metrics are just sums of dispersion measures σ_1 and σ_2. We mark it with $r(\sigma_1)$ and $r(\sigma_2)$ and calculate it as follows:

$$r(\sigma_1) = \sum_{i=1}^{K} \sigma_1(c_i), \tag{3.17}$$

$$r(\sigma_2) = \sum_{i=1}^{K} \sigma_2(c_i). \tag{3.18}$$

Small values of $r(\sigma_1)$ and $r(\sigma_2)$ mean a high recurrence of objects within the feature space. Higher values mean exactly the opposite.

We have four separation measures $s_1(c_i, c_j)$, $s_2(c_i, c_j)$, $s(s_1)$, and $s(s_2)$. The first two separation measures explain how far apart the clusters are from each other. We measure it for each pair of centroids. The metric s_1 can be calculated as follows:

$$s_1(c_i, c_j) = \frac{1}{n_i n_j} \sqrt{\sum_{x_1, \in c_i, x_2 \in c_j} d^2(x_1, x_2)}. \tag{3.19}$$

We take two objects, each from different clusters, and calculate the power distance measure. Next, we sum all the distances from the object of two clusters and calculate the square root of it. The value is then divided by the multiplication of the counts of objects in both clusters. As shown in the Listing 3.25 we have this time three loops. In two we get objects of centroids, and in the other two, we get the distance between those objects.

```
def calculate_s_1(centers, membership):
    """
    This method calculates s1 quality metric

    :param centers: clusters centroids
    :param membership: membership matrix
    :return: s1 values
    """
    s1 = []
    for center_1 in range(len(centers)):
        for center_2 in range(len(centers)):
            if center_1 == center_2:
                break
            ids_1 = np.where(membership[:, center_1] ==
    1)[0]
            ids_2 = np.where(membership[:, center_2] ==
    1)[0]
            elements_1 = data_set[ids_1]
            elements_2 = data_set[ids_2]
            s_1 = 1.0 / (len(ids_1) * len(ids_2))
            for element_1 in elements_1:
                for element_2 in elements_2:
                    s_1 = s_1 * sqrt(calculate_distance
                    (element_1, element_2) ** 2)
            s1.append(s_1)
    return s1
```

Listing 3.25 s_1 index calculation method

The second separation measure is about the distance between two centroids:

$$s_2(c_i, c_j) = d(c_i, c_j). \tag{3.20}$$

It is the simplest measure to date, since it uses only the distance between two centers. The third separation measure uses the dispersion measure σ_1. It is a simple sum of a division of s_1 for two centroids and σ_1 for centroid c_i:

$$s(s_1) = \sum_{i,j=1;j\neq i}^{K} \frac{s_1(c_i, c_j)}{\sigma_1(c_i)}. \tag{3.21}$$

This measure takes into account the entire feature space. The sums can be easily calculated if we already have s_1 and σ_1 as shown in the Listing 3.26.

```
def calculate_s_s_1(s1, sigma_1):
    """
    This method calculates s(s1) quality metric

    :param s1: s1 metric values
    :param sigma_1: sigma1 metric values
    :return: s(s1) values
    """
    s_1_sum = 0.0
    sigma_1_sum = 0.0
    for s_1 in s1:
        s_1_sum = s_1_sum + s_1
    for sigma_1 in sigma_1:
        sigma_1_sum = sigma_1_sum + sigma_1
    s_s1 = s_1_sum / sigma_1_sum
    return s_s1
```

Listing 3.26 $s(s_1)$ index calculation method

The last measure is also simple. It is a sum of measures s_2:

$$s(s_2) = \sum_{i,j=1;j\neq i}^{K} s_2(c_i, c_j) \tag{3.22}$$

We do not need to calculate all the measures to know if our clustering method is performing well. In most cases, we can use just a few or one. Especially $r(\sigma_2)$ is used very often. There are also some other metrics that are called indices and are explained later. Both are the most popular ones.

Example 6 (*Aircrafts clustering methods quality*) In this example we explain the measures described above based on the Example 1 shown in the k-means section. The heterogeneity and homogeneity measures cannot be used for all clustering methods. Some methods need cluster centroids, and only distributed methods provide centroids. Technically, it is possible to calculate the centroids based on an already clustered data set using different methods, but in this section we stick to the k-means examples. Let us assume that we also have done a k-means clustering with $k = 3$ and have the results of the membership matrix as

$$U = \begin{bmatrix} 0\,0\,0\,0\,0\,0\,0\,0\,1 \\ 1\,1\,1\,1\,1\,0\,0\,0\,0 \\ 0\,0\,0\,0\,0\,1\,1\,1\,0 \end{bmatrix},$$

and the centroids as

$$V = \begin{bmatrix} 0.422 & 0.385 \\ 0.078 & 0.178 \\ 0.829 & 0.970 \end{bmatrix}.$$

The dispersion measure σ_1 for the k-mean method with $k = 2$ would be

$$m_1 = \frac{(7-1)*7}{2} = 21,$$

$$m_2 = \frac{(3-1)*3}{2} = 3,$$

$$\sigma_1^{\text{hcm2}}(c_1) = \frac{1}{21} * 2.728 = 0.1299,$$

$$\sigma_1^{\text{hcm2}}(c_2) = \frac{1}{3} * 0.63529 = 0.2118.$$

The results are given for each cluster. That is why we have only two values in the first cases, and three in the case of three clusters:

$$m_1 = \frac{(6-1)*6}{2} = 15,$$

$$m_2 = \frac{(3-1)*3}{2} = 3,$$

$$m_3 = \frac{(1-1)*2}{2} = 0,$$

$$\sigma_1^{\text{hcm3}}(c_1) = \frac{1}{15} * 0.6830 = 0.0455,$$

$$\sigma_1^{\text{hcm3}}(c_2) = \frac{1}{3} * 0.63529 = 0.2118,$$

$$\sigma_1^{\text{hcm3}}(c_3) = 0.$$

The lower value means a better cluster, but the last cluster of the $k = 3$ k-means method is equal to 0. We have only one object in this cluster and it shouldn't be compared to other clusters. The second conclusion is that when one of the objects was removed from the first cluster in the $k = 3$ clustering, the dispersion measure decreased about three times compared to the first group in the $k = 2$ group. This cluster should also be considered the best based on the dispersion measure σ_1. The second measure of dispersion takes the distance between the objects and the centroid of a cluster:

$$\sigma_2^{\text{hcm2}}(c_1) = \frac{1}{7} * 0.1948 = 0.0278,$$

$$\sigma_2^{\text{hcm2}}(c_2) = \frac{1}{3} * 0.10588 = 0.03529.$$

Again, in the first cluster the objects are closer to the centroid compared to other clusters. We get similar results for $k = 3$:

$$\sigma_2^{\text{hcm3}}(c_1) = \frac{1}{7} * 0.0569 = 0.0278,$$

$$\sigma_2^{\text{hcm3}}(c_2) = \frac{1}{3} * 0.10588 = 0.03529,$$

$$\sigma_2^{\text{hcm3}}(c_3) = 0.$$

The separation measure s_1 gives a better understanding of how the clusters are related to each other. In other words, how are good clusters different from other clusters. We calculate this measure for $k = 2$ and $k = 3$ as the previous metrics. For $k = 2$ we compare just the two clusters:

$$s_1^{\text{hcm2}}(c_1, c_2) = \frac{1}{7 * 3} * \sqrt{23.8735} = 0.2327.$$

We can compare the result only with other clusters that we get using k-means with $k = 3$:

$$s_1^{\text{hcm3}}(c_1, c_2) = \frac{1}{6 * 1} * \sqrt{1.0225} = 0.1685,$$

$$s_1^{\text{hcm3}}(c_2, c_3) = \frac{1}{3 * 1} * \sqrt{1.6307} = 0.42566,$$

$$s_1^{\text{hcm3}}(c_1, c_3) = \frac{1}{6 * 3} * \sqrt{22.2428} = 0.262.$$

The second and third clusters in k-means with $k = 3$ clustering are different when we compare both.

3.4.2 Number of Clusters

The number of clusters can be chosen using the elbow method [10]. The goal of this method is to choose many values of k and calculate the error rate using one of the methods explained in the previous section or the internal indices explained in the next section. Based on the result of each execution, we can plot a graph that looks as

shown in Fig. 3.7. At some point in this graph, increasing the number k gives a lower error rate. The error rate is next at a similar level for the next values of k. This means that the best k in Fig. 3.7 would be 6, because for the next values of k the error rate is similar or the changes are very low. Choosing the appropriate value of k can be done automatically. One of such an approach is presented in X-means [11]. A good practice is also to obtain $r(\sigma_2)$ as the error rate on the y axis. It behaves similarly to the graph as shown in Fig. 3.7.

3.4.3 Internal and External Indices

Internal and external indices are another type of measure. The difference between the internal and external indices depends on the information used to calculate the index. The internal indices are based only on the training data set. External indices use the labels and testing data set [12–19]. We can use the typical quality metrics known from supervised learning. Usually, we do not have the labels available for verification of the clustering method. This is why in this section we focus on internal indices. One of the most popular is called the Dunn index [20]. The Dunn index can be easily calculated as a quotient of two distances:

$$C = \frac{d_{\min}}{d_{\max}}, \tag{3.23}$$

where the equations of $d_{\max}$ and $d_{\min}$ are as follows:

$$d_{\max} = \max_{1 \le k \le K} D_k, \tag{3.24}$$

$$d_{\min} = \min_{k \neq k'} d_k. \tag{3.25}$$

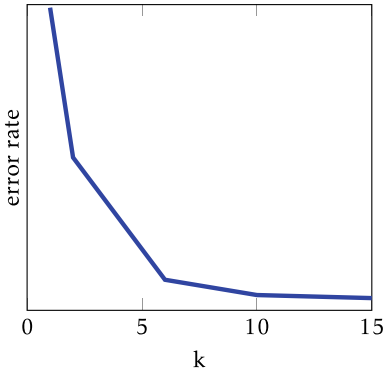


Fig. 3.7 Choosing the proper k number of clusters

Both distances are just the minimum and maximum Euclidean distances between objects. The minimum distance is a measure of two objects that are in different clusters:

$$d_k = \min_{i,j \in I_k; i \neq j} d(x_i^{(k)} - x_j^{(k')}) \tag{3.26}$$

The clusters are marked with k and k'. The maximum distance takes the distance of two objects within the cluster:

$$D_k = \max_{i,j \in I_k; i \neq j} d(x_i^{(k)} - x_j^{(k)}) \tag{3.27}$$

D_k and d_k values are calculated for each cluster k, but in the Dunn index, we take only the highest value of D_k and the lowest value of d_k. It can be calculated in Python as shown in Listing 3.27.

```
def dunn_index(membership):
    """
    This method calculates dunn index quality metric

    :param membership: membership matrix
    :return: dunn index values
    """
    minimum_distance = 1
    maximum_distance = 0
    unique_labels = np.unique(membership[0])
    # @TODO
    for label_id_1 in range(len(unique_labels)):
        ids_1 = np.where(membership[:, label_id_1] == 1)
    [0]
        for label_id_2 in range(len(unique_labels)):
            if label_id_1 == label_id_2:
                break
            ids_2 = np.where(membership[:, label_id_2] ==
     1)[0]
            for element_1 in data_set[ids_1]:
                for element_2 in data_set[ids_2]:
                    distance = calculate_distance
                    (element_1, element_2)
                    if distance > maximum_distance:
                        maximum_distance = distance
                    if distance < minimum_distance:
                        minimum_distance = distance
    dunn_index = minimum_distance / maximum_distance
    return dunn_index
```

Listing 3.27 Dunn index calculation method

In the Listing 3.27 we loop over the clusters and calculate the minimum (`minimum_distance`) and maximum (`maximum_distance`) distances. Both are used to calculate the C. In this case, higher values mean better clustering results. Compared to the heterogeneity measures, we take all clusters into account and get the results of the whole method instead of comparing each cluster with each other. It means that there is one value for the clustering method independent of the k value.

Example 7 (*Dunn index of aircraft data set clustering*) We calculate the minimum and maximum distance values between the objects within the cluster and obtain the maximum value, in general $d_{\max}$. Same for the minimum. For the k-mean aircraft

data set and $k = 2$ we get

$$C^{\text{hcm2}} = \frac{0.63237}{1.30295} = 0.48534,$$

$$C^{\text{hcm3}} = \frac{0.35409}{0.49011} = 0.72248.$$

The values are taken from the distance matrix. As expected, the second case is almost 50% better than the previous clustering. At some point, as the value of k increases, the value of C will increase slower, and that is a good metric to determine the best value of k. This applies also to other clustering methods as distributed clustering methods.

3.5 Image Segmentation

Clustering is usually used in image segmentation. In the current example, we use the logo of Jagiellonian University in Krakow (see Fig. 3.8a) to segment it into three groups. To fulfill this task, we use the k-means clustering method. The image is 232 pixels in width and 258 pixels in height. The segmentation process is divided into the following steps:

1. convert image into a numpy matrix,
2. select the k value,
3. generate group centers in three-dimensional RGB space,
4. use k-means to segment the numpy matrix,
5. save the result as output image.

The only difference compared to the k-means algorithm is the part dedicated to image processing.

3.5.1 Preprocessing

In the Listing 3.28 there is an example of an image conversion class is presented. We need to convert an image into a matrix that can be used for a calculation in Python. Each pixel is represented by three values of the RGB model. The image can be represented as a three-dimensional matrix or as three two-dimensional matrices. There are 255^3 different colors available, and we have a space of $232 * 258$ to analyze. This means that a lot of pixels need to be analyzed. To reduce the calculation, we can get the colors that are available on the image as some colors are repeated very often on the image. This dramatically reduces the number of calculations needed. In our example, depending on the precision, we have 59,856 or 256 unique colors. To get

256 unique colors, we need to limit the precision to two decimals. To obtain unique colors, we can implement a method as shown in the Listing 3.28.

```
class ImageConversion:

    def get_image_from_url(self, img_url):
        """
        This method loads the image and returns it.

        :param img_url: image path
        :return: returns the image as a Pillow image
        """
        image = open(img_url,'rb')
        return img.imread(image)

    def get_unique_colors(self, image_matrix):
        """
        This method gets the unique colors in the image
    and returns is as a matrix.

        :param image_matrix: image pixels
        :return: returns the unique colors and all pixels
        """

        pixel_matrix = []
        for i in range(len(image_matrix)):
            pixel_matrix = pixel_matrix + image_matrix
            [i].tolist()
        pixel_matrix_np = np.array(pixel_matrix)
        uniques, index = np.unique([str(i) for i in
    pixel_matrix_np], return_index=True)
        return pixel_matrix_np[index], pixel_matrix

    def save_image(self, image_shape, pixel_matrix,
    unique_matrix, membership_matrix, colors, output_path
    ):
        """
        This method gets the image size and pixels,
    assign each pixel to one of each
        cluster and save the segmented image

        :param image_shape: the output image shape
        :param pixel_matrix: image pixels
        :param unique_matrix: unique colors matrix
        :param membership_matrix: the memberships matrix
    of each unique color
        :param colors: the cluster colors
        :param output_path: image path to be saved
        :return: None
        """
        image_out = Image.new("RGB", image_shape)
        colors = (np.array(colors) * 255).astype(int).
    tolist()
        color_membership_id = 0
        for unique in unique_matrix.tolist():
            indices = np.where((pixel_matrix[:,0] ==
    unique[0]) & (pixel_matrix[:, 1] == unique[1]) &
            (pixel_matrix[:,2] == unique[2]))
            pixel_matrix[indices[0].tolist()] = tuple
            (colors[np.array(membership_matrix[
    color_membership_id]).argmax()])
            print(np.array(membership_matrix[
    color_membership_id]).argmax())
            color_membership_id = color_membership_id + 1
        pixel_matrix = pixel_matrix.astype(int)
```

```
            pixels = [tuple(x) for x in pixel_matrix.tolist()
    ]

            image_out.putdata(pixels)
            image_out.save(output_path)
```

Listing 3.28 Image convertion class

The `ImageConversion` class consists of three methods. The first method `get_image_from_url` reads the image and returns the handle to the object in the image. The second `get_unique_colours` loops over the image matrix and obtains the unique pixel colors. The last method saves the image based on the pixel matrix and unique colors. It uses the group color assigned to the unique color to draw the final image. The result is an image consisting of k colors. In our example, we have only three colors that are the centroids.

3.5.2 *Selecting the Number of Clusters*

The next step can be implemented as a method shown in the Listing 3.29. We can easily judge, based on the image shown in Fig. 3.8a, that the valid value of k is 3.

```
    def calculate_distance(self, x, v):
        """
        This method calculates the Euclidean distance
    between object x and v.

        :param x: first object
        :param v: second object
        :return: Euclidean distance
        """
        return math.sqrt((x[0]-v[0])**2+(x[1]-v[1])**2+
        (x[2]-v[2])**2)
```

Listing 3.29 Euclidean distance for more than two points

We select three groups centers in three-dimensional space. The goal here is to find three colors that are the centers of each group.

3.5.3 *Distributed Clustering-Based Segmentation*

We calculate the membership vector of each pixel by analyzing only the colors in the image. The final code that combines all the steps can be found in the Listing 3.30. The class `Segmentation` is a modified version of the implementation of k-means from Sect. 3.1. The modifications are adjustments to work with three clusters.

```
image_to_segment = "<path>"
image_converter = ImageConversion()
image_data = image_converter.get_image_from_url
(image_to_segment)
unique_image_data, image_data_list = image_converter.
    get_unique_colors(image_data)

```

```
7  groups = 3
8
9  if image_data.shape[2] > 3:
10     image_data = image_data[:,:,[0,1,2]]
11     unique_image_data = unique_image_data[:,[0,1,2]]
12     image_data_list = np.array(image_data_list)
       [:,[0,1,2]]
13
14 segmentation = Segmentation(unique_image_data, groups)
15 segmentation.do_segmentation()
16 centers, membership_matrix = segmentation.get_results()
17
18 image_size = (232, 258)
19 image_converter.save_image(image_size, image_data_list,
       unique_image_data, membership_matrix, centers, "<path
       >")
20
21 fig = pyplot.figure()
22 ax = Axes3D(fig)
23 #ax.set_aspect("equal")
24 x_centers = [item[0] for item in centers]
25 y_centers = [item[1] for item in centers]
26 z_centers = [item[2] for item in centers]
27
28 x_values = [item[0] for item in unique_image_data]
29 y_values = [item[1] for item in unique_image_data]
30 z_values = [item[2] for item in unique_image_data]
31 ax.scatter(x_values, y_values, z_values, c=np.array
   (unique_image_data),alpha=0.5)
32
33 ax.scatter(x_centers, y_centers, z_centers, c='black',
       marker='s',alpha=1)
34
35 ax.set_xlabel('R')
36 ax.set_ylabel('G')
37 ax.set_zlabel('B')
38
39 pyplot.show()
40
```

Listing 3.30 Example main script

In lines 13–14 we set the image size and save the segmented image. The input and output images are shown in Fig. 3.8. The output presents the assignment of each pixel to one of the following groups:

- logo background
- logo
- image background

We can use segmentation for all images. In this example, we used the RGB model, but it can also be used with other color models.

3.5.4 Centroids in RGB Model

In the examples in Sect. 3.1 we used a data set with two features. This means that we had a two-dimensional feature space. In the image segmentation example, we used

(a) Image used for segmentation

(b) Image segmented into three groups of pixels

Fig. 3.8 Image segmentation done using HCM method. One left of the original image is shown. On the right is the segmented image. *Source* https://www.uj.edu.pl/

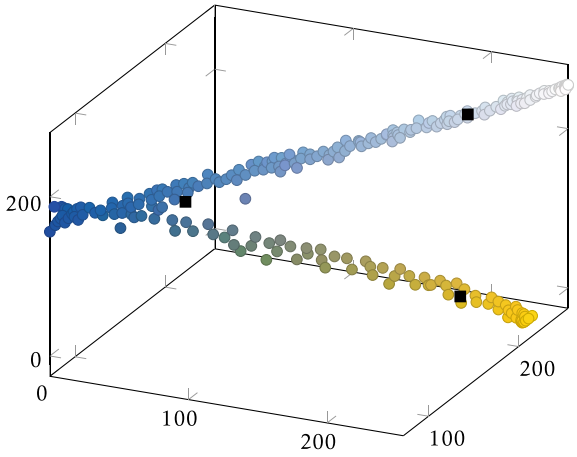


Fig. 3.9 Segmentation pixel unique colors set in a three-dimensional RGB feature space. Pixels are marked with a unique color. Centroids are marked with black squares

three features. We can draw the centroids and pixel colors in a three-dimensional RGB feature space. This plot is given in Fig. 3.9. The centroids are marked with black squares. These are our groups that are visible in Fig. 3.8b. We have limited the number of unique colors in Fig. 3.9 to 256.

For Further Reading

1. Patel AA (2019) Hands-on unsupervised learning using Python. O'Reilly
2. Johnston B, Jones A, Kruger C (2019) Applied unsupervised learning with Python. Packt

3. Aggarwal CC, Reddy CK (2013) Data clustering: algorithms and applications. Chapman and Hall/CRC
4. Xu R, Wunsch D (2008) Clustering. Wiley-IEEE Press
5. Wang W (2025) Principles of machine learning. The three perspectives. Springer

References

1. Bezdek JC (1981) Pattern recognition with fuzzy objective function algorithms. Plenum Press, New York
2. Gustafson DE, Kessel WC (1979) Fuzzy clustering with a fuzzy covariance matrix. In: 1978 IEEE conference on decision and control including the 17th symposium on adaptive processes. IEEE, pp 761–766
3. Krishnapuram R, Keller J (1993) A possibilistic approach to clustering. IEEE Trans Fuzzy Syst 1:98–110
4. Everitt BS, Landau S, Leese M, Stahl D (2011) Cluster analysis, 5th edn. Wiley
5. Jain AJ, Dubes RC. Algorithms for clustering data. Prentice Hall, Englewood Cliffs, NJ
6. Theodoris S, Koutroumbas K (2009) Pattern recognition. Elsevier Academic Press, Amsterdam
7. Cutting D, Karger D, Pedersen J, Tukey J (1992) Scatter/gather: a cluster-based approach to browsing large documents collections. In: Proceedings of 5th annual international ACM SIGIR conference on research and development in information retrieval, pp 318–329
8. Guha S, Rastogi R, Shim K (1998) Cure: an efficient clustering algorithm for large databases. ACM Press, pp 73–84
9. Zhang T, Ramakrishnan R, Livny M (1996) Birch: an efficient data clustering method for very large databases, pp 103–114
10. Kaufman L, Rousseeuw P (1990) Finding groups in data: an introduction to cluster analysis, p 01
11. Pelleg D, Moore AW (2000) X-means: extending k-means with efficient estimation of the number of clusters. In: Proceedings of the seventeenth international conference on machine learning, ICML '00, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc., pp 727–734
12. Tan P-N, Steinbach M, Kumar V (2005) Introduction to data mining, 1st edn. Addison-Wesley Longman Publishing Co., Inc, Boston, MA
13. Fukuyanma Y, Sugeno M (1989) A new method of choosing the number of clusters for the fuzzy c-means method, pp 247–250
14. Xie XL, Beni G (1991) A validity validity measure for fuzzy clustering. IEEE Trans Pattern Anal Mach Intell 13:841–847
15. Krishnapuram R, Freg C-P (1992) Fitting an unknown number of lines and planes to image data through compatible cluster merging. Pattern Recognit 25:382–400
16. Dave RN (1996) Validating fuzzy partition obtained through c-shells clustering. Pattern Recognit Lett 17:613–623
17. Bezdek JC (1979) Cluster validity with fuzzy sets. J Cybern 3:58–72
18. Bezdek JC (1975) Mathematical models for systematics and taxonomy. Freeman, San Francisco, CA, pp 143–166
19. Wu K-L, Yu J, Yang M-S (2005) A novel fuzzy clustering algorithm based on a fuzzy scatter matrix with optimality tests. Pattern Recognit Lett 26:639–652
20. Dunn J (1974) A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters. J Cybern 3:32–57

Chapter 4
Introduction to Shallow Supervised Methods

In this section, we explain a few basic methods. Explaining the simple machine learning methods done in the first place makes it easier to understand the more complex ones. All the methods presented in this chapter are supervised methods. We start with linear classifiers, such as the Fisher classifier. To understand the linearity of the classifiers, we discuss the k nearest neighborhood method that is not linear by design and compare it to Fisher's Linear Discriminant method. The last part of the linear section is dedicated to two regression methods: linear and logistic regression.

4.1 Fisher's Classifier

Fisher's Linear Discriminant is also known as Linear Discriminant Analysis [1]. The idea of Fisher's classifier is to move the data into a reduced-dimension feature space and do the classification there. For example, reducing a two-dimensional feature space to a one-dimensional feature space makes the classification easier, because instead of a hyperplane in the higher dimension space, we just need to find a point on a line in a one-dimensional space. Simplification is not always the right way to distinguish between classes. We will show the SVM classifier later where the goal is the total opposite and the goal is to add one or more dimensions. The appropriate method should be chosen according to the classification problem.

The goal of Fisher's classifier is to calculate the between-class variance with the class means (m_1, m_2), and the within-class variance (S_w, S_i). The means can be calculated as follows:

$$m_i = \frac{1}{n_i} \sum_{x \in i} x. \tag{4.1}$$

K. Przystalski et al., *Pattern Recognition Primer*,
https://doi.org/10.1007/978-3-031-91816-2_4

The within-class variance can be calculated as follows:

$$S_w = \sum_{k=1}^{K} \sum_{n=1}^{C_k} N_k (x_n - m_k)(x_n - m_k)^T \tag{4.2}$$

and

$$S_b = \sum_{k=1}^{K} (m_k - m)(m_k - m)^T. \tag{4.3}$$

Finally, the weights can be calculated as follows:

$$w = S_1^{-1}(m_{+1} - m_{-1}). \tag{4.4}$$

The discriminant function can be written as

$$\hat{g}(x) = w_F^T x = (m_{+1} - m_{-1})^T S_W^{-1} x, \tag{4.5}$$

where:

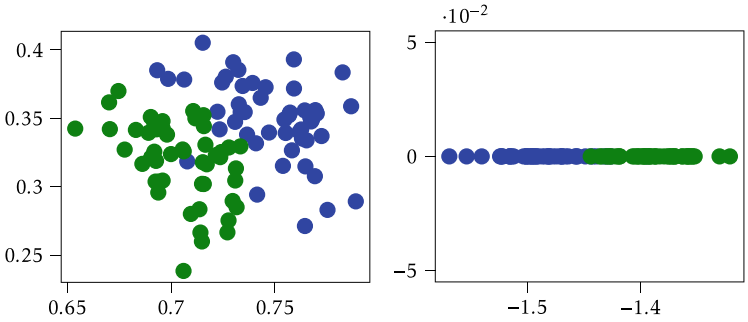
$$w_0 = \frac{1}{2}(w^T m_{+1} + w^T m_{-1}). \tag{4.6}$$

This brings us to the following.

$$\hat{g}(x) = w^T x - w_0 \begin{cases} >0, x \in 1, \\ <0, x \in -1. \end{cases} \tag{4.7}$$

Example 1 (*Iris dimensionality reduction*) The iris data set is a classic one that consists of four features and a label. There are three labels, but to simplify this example, we use only two labels. In the Listing 4.1 two the data set is loaded in lines 3–8. In the lines 10 and 11, the mean values are calculated. It is needed for the variances that are calculated in the following lines. The last lines are used to plot the plot with data of reduced dimensionality.

```
1  from sklearn.datasets import load_iris
2  from sklearn import preprocessing
3
4  iris_data, iris_labels = load_iris(return_X_y=True)
5  iris_data = np.array(preprocessing.normalize(iris_data))
6
7  x1 = iris_data[np.where(iris_labels == 1)][:,[0,1]]
8  x2 = iris_data[np.where(iris_labels == 2)][:,[0,1]]
9  y = iris_labels
10
11 mean_x1, mean_x2 = np.mean(x1,axis=0), np.mean(x2,axis=0)
12 mean = np.mean(np.append(x1,x2))
13
14 Sb = np.sum((mean_x1-mean)*(mean_x2-mean))
15 Sw = np.dot((x1-mean_x1).T, (x1-mean_x1))+np.dot((x2-mean_x2).T, (x2-
       mean_x2))
16
17 w = np.dot(np.linalg.inv(Sw), (mean_x2-mean_x1))
```



(a) Two features space of objects of two classes of the Iris dataset (b) Dimensionality reduced of the Iris dataset

Fig. 4.1 Reduce the dimensionality of the iris data set using Fisher classifier

```
18
19 cov = np.cov(np.concatenate((x1.T,x2.T)),ddof=0)
20 cov_inv = np.linalg.inv(cov)
21
22 plt.plot(np.dot(x1, w), [0]*x1.shape[0], "bo")
23 plt.plot(np.dot(x2, w), [0]*x2.shape[0], "go")
```

Listing 4.1 Calculating the S_w and S_b values

In Fig. 4.1 the Iris sets are shown. The mapping of a two-dimensional feature space (a) to a line (b) is given. The example is not complex, and the distinguishing line can be drawn easily, but the threshold that distinguishes between the blue and the green objects can be drawn even more easily in a one-dimensional feature space.

4.2 Nearest Neighborhood Classifiers

In one sentence, kNN looks for other objects in the neighborhood and assigns the most popular label to new objects. The k in the name is the number of objects that we look for labels in the neighborhood. Depending on the number of labels, we should set the proper value of k. For two labels, k should be an odd value like 3 or 5 to make a decision easier. A general discriminant function is set as

$$g_i(y) = k_i, \quad i = 1, \ldots, L, \tag{4.8}$$

where L is the number of classes and k_i is the number of objects of label i in the neighborhood. The sum of k_i is equal to k, the number of neighborhood objects. We choose the label where we have the highest number of labels k_i. The classifier algorithm consists of three simple steps:

- calculate the distance vector between the new object and all objects in our data set,

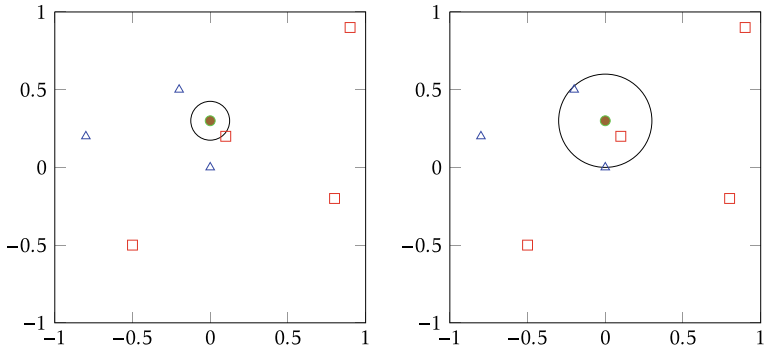


Fig. 4.2 Two example of the same data set for different k

- find the closest k objects with the lowest distance from our new object,
- assign label to the new object using Eq. 4.8.

In Fig. 4.2 two example data sets are given with kNN used with $k = 1$ (left) and $k = 3$ (right). We see that if we set the $k = 1$ for the green circle (new object), we assign it to the square group. If we extend the neighborhood to $k = 3$, we will assign the new object to the triangle group. The value k should be set empirically, depending on the data set.

The distance vector can be calculated in a similar way as it was explained in the previous part on clustering methods. The closest object can also be chosen similarly to the method implemented in the clustering sections or as in the Listing 4.2.

```
def calculate_distance(x, v):
    """
    This method calculates the Euclidean distance between object x and v.

    :param x: first object
    :param v: second object
    :return: Euclidean distance
    """
    return sqrt((x[0] - v[0]) ** 2 + (x[1] - v[1]) ** 2)

def calculate_distance_matrix():
    """
    This method calculates the distance matrix between all objects.

    :return: A matrix of distances
    """
    distance_matrix = np.zeros((len(data_set),len(data_set)))
    for i in range(len(data_set)):
        for j in range(len(data_set)):
            distance_matrix[i, j] = calculate_distance(data_set[i],
    data_set[j])
    return distance_matrix

def find_closest_objects(x, k):
    """
    Finds k closts objects to x.
```

```
26
27      :return: A list of objects' ids.
28      """
29      distances = []
30      i = 0
31      for item in train_set.values:
32          distances.append([i, calculate_distance(x, item)])
33          i = i + 1
34      distances=np.array(distances)
35      label_ids = distances[distances[:, 1].argsort()][:k,0]
36      return [int(item) for item in label_ids]
```

Listing 4.2 Method that returns the closest objects

The main difference between this method and the ones used for clustering is that in the case of kNN, we return a vector of k with the lowest distance instead of just the closest one. It uses the same methods like `calculate_distance` and `calculate_distance_matrix` that work the same as for the clustering methods mentioned in the previous chapter. Similarly, `find_closest_objects` method looks for the closest object to the one that we get as an argument. The difference is the `k` as we do not look for just the closest by for the `k` closest objects.

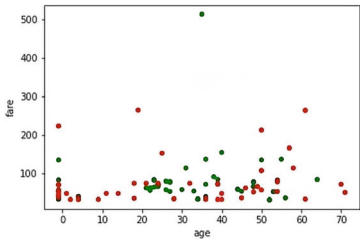
```
1  def predict():
2      """
3      Assign a label of the most common label in the list of closest objects
       .
4
5      :return: A list of predictions.
6      """
7      predictions = []
8      for item in test_set.values:
9          label_ids = find_closest_objects(item, k)
10         counts = np.bincount(train_set_labels.values[label_ids])
11         label = np.argmax(counts)
12         predictions.append(label)
13     return predictions
```

Listing 4.3 kNN prediction method

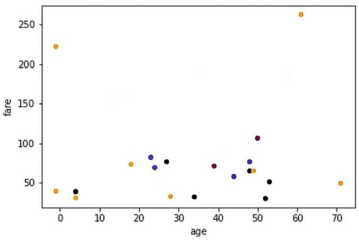
The prediction can be implemented as in Listing 4.3. It just calculates the accuracy—the percentage number of properly classified labels divided by the number of all objects. Instead of typical accuracy as it is implemented in the scikit-learn package, this one counts the number of occurrences of labels and returns the one with the maximum occurrences. For binary cases, the `k` values are usually odd.

Example 2 (*Titanic survival kNN classification*) The Titanic data set is another classic data set that was originally available on Kaggle. It is also available using the Tensorflow library as one of the test data sets. To use it we import the `tensorflow_datasests` and load it as shown in Listing 4.4 lines 7 and 8.

```
1  import numpy as np
2  import tensorflow_datasets as tfds
3  import pandas as pd
4  from math import sqrt
5  from sklearn.metrics import accuracy_score
6
7  ds = tfds.load('titanic', split='train', shuffle_files=True)
8  titanic_df = tfds.as_dataframe(ds)
9
10 titanic_df.drop("name", axis=1, inplace=True)
11 columns = ['survived','age','fare']
```



(a) Train data set



(b) Predicted objects: orange - properly classified as not survived, blue - properly classified as survived, black classified not properly as not survived, purple - not properly classified as survived

Fig. 4.3 kNN Titanic prediction results

```
features = columns[1:]
titanic_df = titanic_df[columns].replace([np.inf, -np.inf], np.nan).dropna()

titanic_df = titanic_df[titanic_df['fare'] > 30]

survivded_df = titanic_df[titanic_df['survived']==1].sample(50, random_state=12345)
not_survivded_df = titanic_df[titanic_df['survived']==0].sample(50, random_state=12345)

train_set = survivded_df.sample(40, random_state=12345)
train_set = pd.concat([train_set, not_survivded_df.sample(40, random_state=12345)])
train_set_labels = train_set['survived']
train_set = train_set[features]

test_set = survivded_df.sample(10, random_state=12345)
test_set = pd.concat([test_set, not_survivded_df.sample(10, random_state=12345)])
test_set_labels = test_set['survived']
test_set = test_set[features]
```

Listing 4.4 kNN data set and parameters setup

The next lines are the data cleanup operations that get three columns (age, fare, survived) and limit the number of examples. We use again a small number of examples to train the model faster, but also in this case plot the examples in such a way that these are easier to read. The surviving column is our binary label and the two other columns are our features. These two features might now be obvious to have an impact on the prediction as gender has. The results are shown in Fig. 4.3. In the first one the training set is shown with two labels: survived (green), drown (red). In Fig. 4.3b the testing set objects are plotted. The orange dots are the objects that are properly classified as drowned, the blue ones are properly classified as survived. On the other hand, the black and purple ones are misclassified as not drowning and as survived.

4.3 Linear Regression

Regression is about predicting the future values of a feature that depends on a second feature. The first feature is called an explanatory variable, and the second that depends on it is called a response variable. To simplify, let x be an explanatory variable and $\hat{y}$ a response variable. It can be calculated as follows:

$$\hat{y} = ax_i + b, \tag{4.9}$$

where

$$a = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sum_{i=1}^{n}(x_i - \overline{x})^2} \tag{4.10}$$

and

$$b = \overline{y} - a\overline{x}. \tag{4.11}$$

The variable a is known as a slope and is calculated using the ordinary least squares method, which is similar to the correlation that we have already presented. The variable b is known as a random variable that adds noise. We can extend the linear regression equation to more than one variable x_i, but to make it simple, we stay with only one here. Keep in mind that we should consider $\hat{y}$ as one of our features from the vector of features. The same as $\overline{y}$ which corresponds to $\overline{x}_{i2}$ in our previous example. It is just the nomenclature. Based on Eq. 4.9 we see that the predicted value depends on the mean. In 1973 Anscombe [2] found that we can have multiple data sets that can give the same results of linear regression. Let us take a look at the data sets shown in Table 4.1. The means for both features are the same and are, respectively, $\overline{x} = 9$ and $\overline{y} = 7.5$. The regression equation will look for each data set the same and looks as follows:

$$\hat{y} = 3 + \frac{1}{2}x_i.$$

The data sets are plotted in Fig. 4.4.

Example 3 (*Learning II*) Let us take the same data set as in the Example in the Statistics section in Chap. 2 (Table 4.2). We can calculate the number of hours needed to collect 100 points on the final exam. First, calculate the variables a and b. The averages are $\overline{x} = 68.5$ and $\overline{y} = 32$. We can easily calculate a and b:

$$a = \frac{1504}{2265.5} \approx 0.6638,$$

$$b = 32 - 0.6638 \cdot 68.5 \approx -13.4703.$$

It brings us to a general equation for this problem that looks as follows:

$$\hat{y} = 0.6638x_i - 13.4703.$$

Table 4.1 Anscombe's data sets

I		II		III		IV	
x_1	y_1	x_2	y_2	x_3	y_3	x_4	y_4
10.00	8.04	10.00	9.14	10.00	7.46	8.00	6.58
8.00	6.95	8.00	8.14	8.00	6.77	8.00	5.76
13.00	7.58	13.00	8.74	13.00	12.74	8.00	7.71
9.00	8.81	9.00	8.77	9.00	7.11	8.00	8.84
11.00	8.33	11.00	9.26	11.00	7.81	8.00	8.47
14.00	9.96	14.00	8.10	14.00	8.84	8.00	7.04
6.00	7.24	6.00	6.13	6.00	6.08	8.00	5.25
4.00	4.26	4.00	3.10	4.00	5.39	19.00	12.50
12.00	10.84	12.00	9.13	12.00	8.15	8.00	5.56
7.00	4.82	7.00	7.26	7.00	6.42	8.00	7.91
5.00	5.68	5.00	4.74	5.00	5.73	8.00	6.89

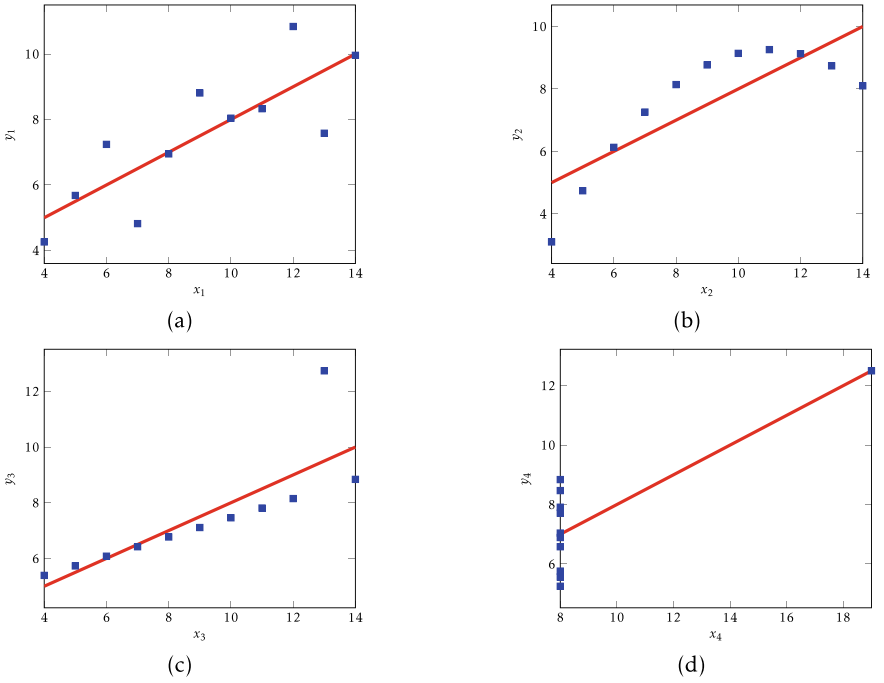


Fig. 4.4 Linear regression charts for Anscombe's data sets

Table 4.2 Correlation between hours spent on learning and exam grade exemplary data

Average hours spent on learning	10	22	30	38	44	48
Average points collected	40	52	61	75	88	95

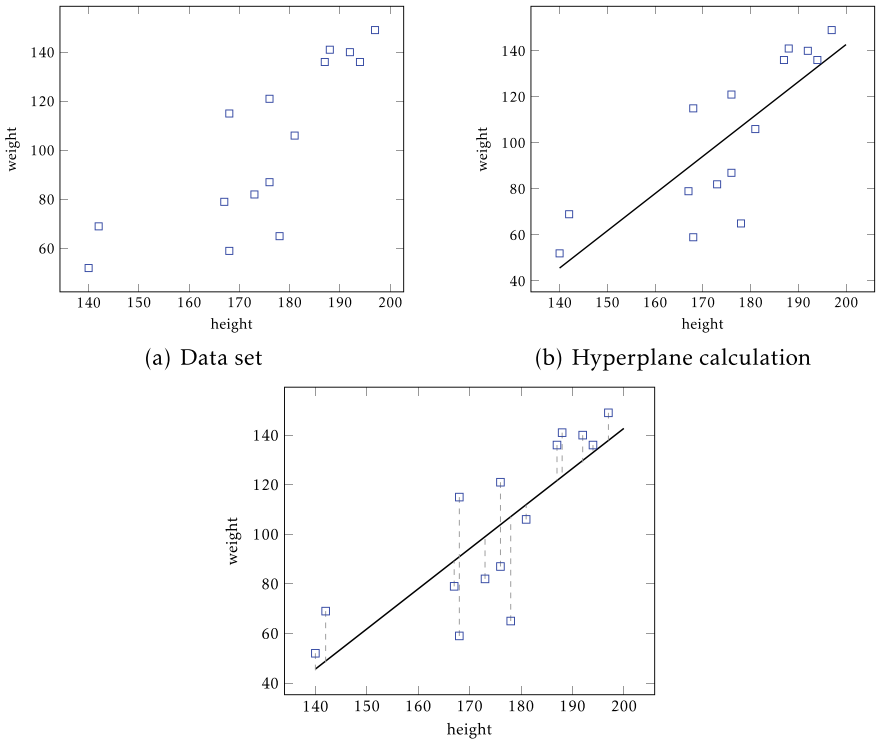


Fig. 4.5 Linear regression calculation step by step

Now we can calculate how many hours we theoretically need to learn to collect 100 points:

$$\hat{y}_{100} = 0.6638 \cdot 100 - 13.4703 = 52.9.$$

Linear regression works with more than two features (Fig. 4.5), which means that it generates hyperplanes like shown in Fig. 4.6 in a three-dimensional features space. The weights can be implemented using the numpy methods as in Listing 4.6.

```
w = np.dot(np.linalg.inv(np.dot(np.transpose(x),x)),np.dot(np.transpose(x)
    ,y))
```

Listing 4.5 Linear regression weights calculation

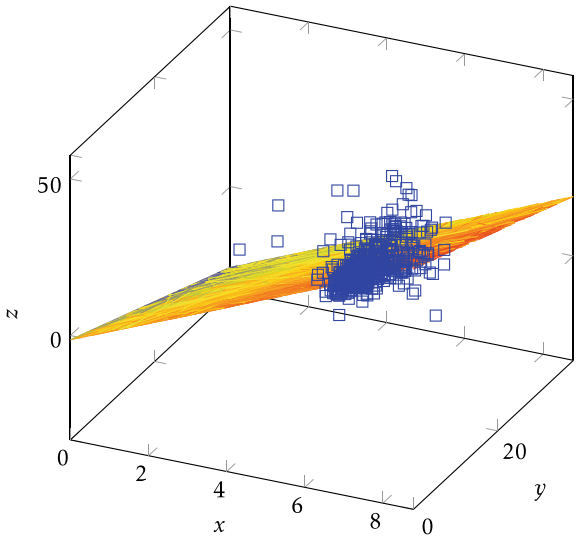


Fig. 4.6 Linear regression hyperplane in a three-dimensional features space

The y value can be next calculated as given in Eq. 4.9

```
def reg_predict(inputs, w, b):
    results = []
    for inp in inputs:
        results.append(inp*w+b)
    return results
```

Listing 4.6 Linear regression weights calculation

Lasso regression

A complex model can end with overfitting. It highly depends on variance and bias (see Fig. 4.7). The variance is related to the differentiation of the training data. The more features and data we have there higher variance we get. The bias is about simplification of the model by having a focus on just a part of the features when the bias is high. It is very hard to have a low variance and low bias model, but we should keep trying to get as close as possible. In Fig. 4.8 we show the differences in each combination of low/high variance and bias. There are some methods that can reduce the model complexity by reducing the variance and bias. In linear regression, we have a few modifications that

- Lasso regression,
- Ridge regression,
- Elastic Net regression.

Lasso stands for least absolute shrinkage and selection operator. It uses the L1 regularizer. We take magnitudes into account:

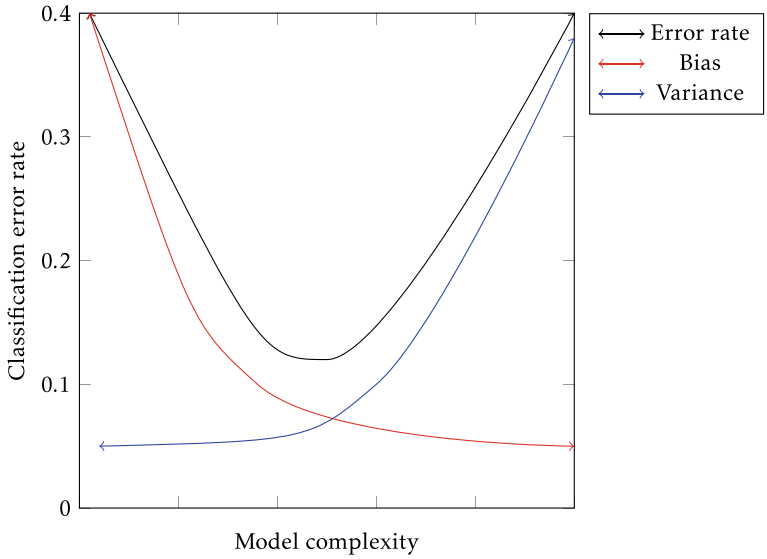
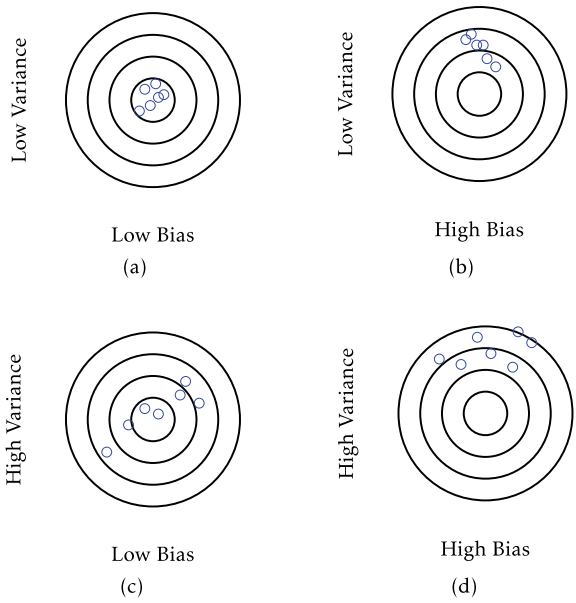


Fig. 4.7 Model complexity versus classification error rate, depending on the bias and variance

Fig. 4.8 Bias versus variance trade-off



$$\sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{p} w_j \dot{x}_{ij}\right)^2 + \lambda \sum_{j=0}^{p} |w_j|. \tag{4.12}$$

For $\lambda = 0$, the formula is linear regression. This regularization can make some of the features not be taken into account in the final output. This means that we can use Lasso to select the features. The value λ:

- higher value means less features,
- lower values mean more features selected.

The way of calculating the linear regression as shown in the previous section is simple but in many cases is not efficient. A different way and one of the most popular methods to find the proper weights are the mean squared error and the stochastic gradient descent, where the first is formulated as

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y - \hat{y})^2. \tag{4.13}$$

MSE or a modification of it can be used to implement stochastic gradient descent (SGD). A cost function in linear regression is defined as follows:

$$\sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{p} w_j \dot{x}_{ij}\right)^2. \tag{4.14}$$

This function should be minimized using the MSE and SGD methods.

Example 4 (*Height found with Lasso regression with SGD*) To show how the SGD works, we use one of the BMI examples of people's weights and heights shown in Table 4.3. The lasso regression needs the `alpha` variable to calculate the slope. The SGD function takes the initial coefficient matrix, the data set, labels, the number of epochs, the learning rate, and the L1 alpha (Listing 4.7).

```
x = np.asmatrix(np.c_[np.ones((len(x),1)),x])

I = np.identity(2)
alpha = 0.1

init_c = np.zeros((2,1))
results = []

w2 = sgd(init_c, x, y, 10, 0.1, alpha)
w2 = w2.ravel()
results.append(w2)

w1 = np.linalg.inv(x.T * x + alpha * I) * x.T * y
w1 = w1.ravel()
w1 = np.squeeze(np.asarray(w1))
results.append(w1)
```

Listing 4.7 Linear regression weights calculation using SGD

Table 4.3 BMI data set example

Height	188	181	197	168	167	187	178	194	140	176	168	192	173	142	176
Weight	141	106	149	59	79	136	65	136	52	87	115	140	82	69	121

The method calculates the weights and bias using Eq. 4.14.

```
def sgd(coeffs, x, y, epochs, rate = 0.1, l1 = 0.1):
    norm = np.linalg.norm(x, axis = 0)
    w = coeffs[0]
    b = coeffs[1
    m = y.shape[0]
    n = x.shape[1]
    for i in range(epochs):
        x_in = x[:,1].reshape(-1, 1)
        y_pred = x_in * w + b
        if w > 0:
            dW = (- (2 * x_in.T.dot(y - y_pred)) + l1 ) / norm[1] ** 2
        else:
            dW = (- (2 * x_in.T.dot(y - y_pred)) - l1 ) / norm[1] ** 2
        db = - 2 * np.sum(y - y_pred) // norm[0] ** 2
        w = w - rate * dW
        b = b - rate * db
    coeffs[0] = b
    coeffs[1] = w
    return coeffs
```

Listing 4.8 SGD implementation for linear regression

The plot looks similar to the example without using the SGD, because of the small data set. The weights are 49.8 and 0.30, and bias −101.72 and 1.17.

Ridge regression

Ridge regression is about to shrink the coefficients. The equation of the ridge regression can be written as

$$\sum_{i=1}^{M}\left(y_i - \sum_{j=0}^{p} w_j \dot{x}_{ij}\right)^2 + \lambda \sum_{j=0}^{p} w_j^2. \tag{4.15}$$

The λ adds a penalty to the coefficients w. It avoids having too high values of the coefficients and adds the penalty whenever the values are too big. The value λ:

- higher value means more penalty when the coefficients are bigger,
- lower values make it more like regular linear regression,
- higher values make the variance decrease, and the bias increase.

Elastic Net implements both L1 and L2 regularizators. The cost function is defined as follows:

$$\frac{\sum_{i=1}^{M}(y_i - \sum_{j=0}^{p} w_j \dot{x}_{ij})^2}{2n} + \lambda \left(\frac{1-\alpha}{2} \sum_{j=1}^{m} w_j^2 + \alpha \sum_{j=1}^{m} |w_j| \right). \tag{4.16}$$

The parameter α is between 0 and 1, where closer to 1 returns the result that is closer to the one given by the ridge regression, 0 for Lasso. It is not easy to find the best value of λ, but we can use the cross-validation method to test at least a few values of λ and compare the results.

4.4 Logistic Regression

A different type of regression is logistic regression. Logistic regression [3] is based on the logistic function. It is shown in Fig. 4.9. It is very useful especially when we do calculations based on probability theory, because the logistic function gives values from 0 to 1, so it easily corresponds to probability. It can be calculated as follows:

$$\hat{y}_{\log} = \frac{e^{a+bx_i}}{1+e^{a+bx_i}} = \frac{1}{1+e^{-(a+bx_i)}}. \tag{4.17}$$

Logistic regression as well as linear can be found in featured books or articles as Y. The parameters are also marked as α and β_i. To avoid misunderstandings, we keep a common nomenclature. As in linear regression, we need to find the parameters for each problem separately. In logistic regression, it is a vector of parameters that is called weights:

$$w = [b, a]. \tag{4.18}$$

The term weights is very often used in machine learning and we will describe it separately for each method. In logistic regression, the goal is to find the two parameters that give the best representation of the data of a given problem. It would be best

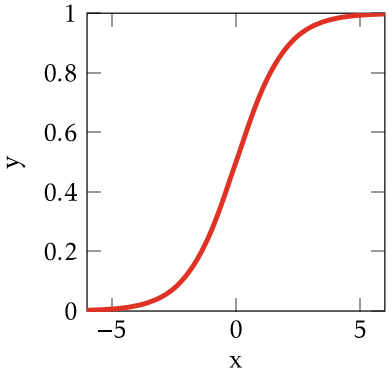


Fig. 4.9 Logistic function

if we could find such parameters that for each x_i set in Eq. 4.17 we get the proper y_i. It can be done using one of the most known methods of a maximum likelihood estimator, the Newton-Raphson method. It is an iterative method and requires more calculations compared to linear regression. Weights are calculated in each iteration as follows:

$$w_{k+1} = w_k + (X^T V X)^{-1} X^T (y - \hat{p}_i), \tag{4.19}$$

where k is the iteration number and V is a diagonal weight matrix. Diagonal weight matrix elements can be calculated as follows:

$$v_{ii} = \hat{y}_{\log_i}(1 - \hat{y}_{\log_i}). \tag{4.20}$$

The loop can end for two reasons. We can set a fixed number of iterations or we can set a weight difference value between two iterations and end the loop when the change in each iteration is below that value. Usually, it is set to 0.01 or lower.

Example 5 (*Skin lesion*) We have six patients with a skin lesion. Three lesions are known to be not cancers and the other three are known to be cancer.

$$y = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

We have two features that indicate if it's cancer or not, so we need to estimate the value of three parameters, and the weights vector looks at the start like:

$$w = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}.$$

Let us construct our feature vector. The first feature is the asymmetry of the lesion. It is a value from a range of 0 to 2 where 2 means total asymmetry and 0 total symmetry. The second is the number of colors that are within the lesion. It is a value from the range 0 to 6. As shown in Eq. 4.17, we need to multiply the weights vector with the features vector matrix. This means that it needs to have three rows instead of two. We need to add one column at the beginning filled with 1. Let the X look as follows:

$$X = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 5 \\ 1 & 1 & 2 \\ 1 & 1 & 3 \\ 1 & 0 & 4 \\ 1 & 2 & 3 \end{bmatrix}.$$

We need to calculate $w^T x_i$ in the first place:

$$w^T x_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 & 0.5 \end{bmatrix} = 0.$$

The next step is to calculate the diagonal elements of matrix V. Before that we need to calculate the logistic regression value for each x_i:

$$y_{\log_1} = \frac{e^0}{1 + e^0} = \frac{1}{2}.$$

The $y_{\log_i}$ values are the same for each x_i in the first iteration. The same with diagonal elements:

$$v_{11} = \frac{1}{2} * \frac{1}{2} = \frac{1}{4}.$$

The weight matrix looks now as follows:

$$V_0 = \begin{bmatrix} 0.25 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.25 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.25 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.25 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.25 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.25 \end{bmatrix}.$$

In the last step, we need to calculate new weight values. It is a bigger computation of multiplied matrices and vectors, so we divide it into a few parts to keep it clear and understandable. In the first place let's calculate $X^T V_0$:

$$\begin{aligned} X^T V_0 &= \begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 5 \\ 1 & 1 & 2 \\ 1 & 1 & 3 \\ 1 & 0 & 4 \\ 1 & 2 & 3 \end{bmatrix}^T \cdot \begin{bmatrix} 0.25 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.25 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.25 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.25 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.25 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.25 \end{bmatrix} \\ &= \begin{bmatrix} 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 \\ 0 & 0.5 & 0.25 & 0.25 & 0 & 0.5 \\ 0 & 1.25 & 0.5 & 0.75 & 1 & 0.75 \end{bmatrix}. \end{aligned}$$

Next we multiply the results with matrix X and inverse the result:

$$(X^T V X)^{-1} = \left(\begin{bmatrix} 0.25 & 0.25 & 0.25 & 0.25 & 0.25 & 0.25 \\ 0 & 0.5 & 0.25 & 0.25 & 0 & 0.5 \\ 0 & 1.25 & 0.5 & 0.75 & 1 & 0.75 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 5 \\ 1 & 1 & 2 \\ 1 & 1 & 3 \\ 1 & 0 & 4 \\ 1 & 2 & 3 \end{bmatrix} \right)^{-1}$$
$$= \begin{bmatrix} 2.91 & -0.32 & -0.68 \\ -0.32 & 1.37 & -0.37 \\ -0.68 & -0.37 & 0.37 \end{bmatrix}$$

Once we are done with it we need to calculate the output vector and current p_0 which is the vector that consists of $p_0(i) = \frac{e^{w^T x_i}}{1+e^{w^T x_i}}$ i elements for each x_i:

$$y - p = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix} - \begin{bmatrix} 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 \end{bmatrix}$$
$$= \begin{bmatrix} -0.5 & 0.5 & -0.5 & -0.5 & 0.5 & 0.5 \end{bmatrix}$$

The next step is to multiply inverted matrix $(X^T V X)^{-1}$ with X^T:

$$(X^T V X)^{-1} X^T = \begin{bmatrix} 2.91 & -0.32 & -0.68 \\ -0.32 & 1.37 & -0.37 \\ -0.68 & -0.37 & 0.37 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 1 & 2 & 5 \\ 1 & 1 & 2 \\ 1 & 1 & 3 \\ 1 & 0 & 4 \\ 1 & 2 & 3 \end{bmatrix}^T$$
$$= \begin{bmatrix} 2.91 & -1.12 & 1.23 & 0.55 & 0.2 & 0.23 \\ -0.32 & 0.57 & 0.31 & -0.06 & -1.8 & 1.31 \\ -0.68 & 0.43 & -0.31 & 0.06 & 0.8 & -0.31 \end{bmatrix}$$

The last step in an iteration is to multiply two matrices that we have just calculated together:

$$(X^T V X)^{-1} X^T (y - p) = \begin{bmatrix} 2.91 & -1.12 & 1.23 & 0.55 & 0.2 & 0.23 \\ -0.32 & 0.57 & 0.31 & -0.06 & -1.8 & 1.31 \\ -0.68 & 0.43 & -0.31 & 0.06 & 0.8 & -0.31 \end{bmatrix}$$
$$\cdot \begin{bmatrix} -0.5 & 0.5 & -0.5 & -0.5 & 0.5 & 0.5 \end{bmatrix}$$
$$= \begin{bmatrix} -2.69 & 0.077 & 0.92 \end{bmatrix}.$$

As the previous weights vector was filled with zeros, the current weight vector is

$$w = \begin{bmatrix} -2.69 & 0.077 & 0.92307692 \end{bmatrix}.$$

Table 4.4 Results after three iteration of Example 5

x_i	$-(\alpha + \beta_1 x_i + \beta_2 x_2)$	$y_{\log}(x_i)$	y_i
(0, 0)	10.02	0.00004	0
(2, 5)	$-(-10.02 + 2.46 + 14.35) = 6.78$	0.99	1
(1, 2)	$-(-10.02 + 1.23 + 5.74) = -3.05$	0.045	0
(1, 3)	$-(-10.02 + 1.23 + 8.61) = -3.23$	0.038	0
(0, 4)	$-(-10.02 + 11.48) = 1.46$	0.811	1
(2, 3)	$-(10.02 + 2.46 + 8.61) = 2.51$	0.924	1

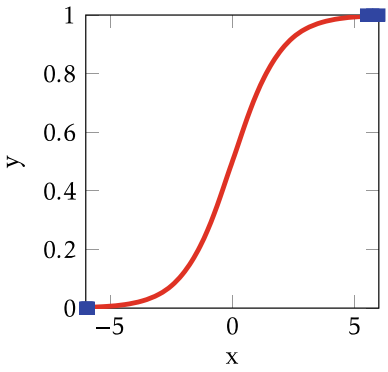


Fig. 4.10 Logistic regression of skin lesion diagnosis example

After three iterations we get the following weight vector:

$$w = \left[-10.02052458\ 1.22700068\ 2.86618458\right].$$

We could also do some more iterations. Now check the logistic regression value for each x_i. It is shown in Table 4.4. The results presented indicate that logistic regression can be useful in some cases. Each x_i that is cancer has a high value of logistic regression, close to 1. The logistic regression value of benign lesions is close to 0. We can draw it as presented in Fig. 4.10.

The code of a simple logistic regression implementation should not take more than 40 lines of code. An example is presented in Listing 4.9.

```
import numpy as np
from math import exp
from numpy.linalg import inv

class LogisticRegression:

    def __init__(self):
        self.weight= np.array([0,0,0])

    def set_y_vector(self,y):
        self.y=np.array(y)

    def set_x_vector(self,x):
```

```
        self.x=x

    def set_iterations(self,k):
        self.k=k

    def calculate_weight_matrix_v(self):
        p_vector=[]
        v_diag=[]
        for j in xrange(len(self.x)):
            w_t_x_i = exp(np.dot(self.x[j],self.weight))
            p_vector.append((w_t_x_i/(1+w_t_x_i)))
            v_diag.append((w_t_x_i/(1+w_t_x_i))*(1-(w_t_x_i/(1+w_t_x_i))))
        logging.info(v_diag)
        v_matrix = np.diag(v_diag)
        return [v_matrix,p_vector]

    def calculate_parameters(self):
        for i in xrange(self.k):
            [v_matrix, p_vector] =self.calculate_weight_matrix_v()
            print "det: " + str(np.linalg.det(np.dot(np.dot(self.x.
    transpose(),v_matrix),self.x)))
            inv_x_t_v_x=inv(np.dot(np.dot(self.x.transpose(),v_matrix),
    self.x))
            y_p_substracted=np.subtract(self.y,np.array(p_vector))
            inv_x_t_v_x_t=np.dot(inv_x_t_v_x,self.x.transpose())
            result=np.dot(inv_x_t_v_x_t,y_p_substracted)
            self.weight=np.squeeze(np.asarray(np.add(self.weight,result)))
```

Listing 4.9 Logistic regression code sample

We need to use three external libraries: `numpy` to multiply, add, and subtract two matrices, `exp` to calculate $e^{w^T x_i}$ and inv to inverse the matrix. In the constructor, we set the initial default weight vector values. The next two methods are just simple setters of the X matrix and the y vector. Method `set_iterations` is about setting the number of iterations as we stop after the number is reached. The last method called `calculate_parameters` is the core part that calculates the new weights. It uses the method `calculate_weight_matrix_v` that calculates the new V_i matrix in each iteration.

4.5 Naive Bayes Classifier

Bayesian classifiers are a set of methods that uses Bayes' theorem or alternatively, Bayes' rule to choose the most matching class, based on some prior knowledge. Typically, we have a training data set that contains vectors with the appropriate class assigned to each of them. Such vectors may describe each case with multiple features, both numeric and categorical. The simplest case is a binary classification based on one-feature-long vectors. Having a training data set given, we would like to ask what the most likely class is for a new object. This time we will not assume single feature vectors or binary classification, but rather describe a general class of problems, which usually affects many different features and classes. In most cases, we do not have posterior probabilities given directly and there is no easy way to calculate such probabilities, having only the data set. For that reason, we will use Bayes' theorem.

We already explained a few probability terms in Chap. 2. Some more that still need more explanation before we define the Bayesian rule is the prior probability. $P(A_i)$ is called *prior probability* and is a probability of belonging to the i-th class and can be simply estimated by counting the number of vectors having this class in our training data set and normalizing it by a total number of vectors. $P(B|A_i)$ is called *likelihood* and is a conditional probability of observing the vector x given that it belongs to the i-th class. These values are usually estimated as a product of the probabilities for each feature (A_i) separately.

Once again, there is a need to estimate these feature-level likelihoods using a training data set. This time, we will get a number of vectors that have a particular value of the feature and belong to the i-th class. Then it will be divided by the total number of vectors that have this value for this particular feature, regardless of which class it belongs to.

The last remaining value is $P(B)$, which is the probability that the object will appear at all. To calculate this value, we would have to know the data distribution; however, we may neglect it. It can be done, because this value will be the same for all the probabilities, and we are looking for the class which maximizes the posterior probability. If the denominator is common for all, then taking the maximum of the nominator is enough. Finally, we can draw the Bayesian rule as follows:

$$P(A_k|B) = \frac{P(B|A_k) \cdot P(A_k}{P(B|A_1) \cdot P(A_1) + P(B|A_2) \cdot P(A_2) + \cdots + P(B|A_n) \cdot P(A_n)}. \tag{4.21}$$

The implementation of the Naive Bayes classifier is divided into three functions: `gaussian_pdf`, `calculate_probability`, and the main function that uses the second function to calculate the probability: `naive_bayes_classifier`.

```
def gaussian_pdf(x, mean, stdev):
    exponent = exp(-((x-mean)**2 / (2 * stdev**2)))
    return (1 / (sqrt(2 * pi) * stdev)) * exponent


def calculate_probability(x, class_probability, mean, stdev):
    probability = class_probability
    i = 0
    for feature in features:
        probability *= gaussian_pdf(x[i], mean[feature], stdev[feature])
        i = i + 1
    return probability
```

Listing 4.10 Naive Bayes Gaussian probability distribution

In Listing 4.10 a short Gaussian probability density function is implemented. Assuming we have the Gaussian distribution in the data set. The PDF is used to calculate the probabilities for a new object.

```
def naive_bayes_classifier(x):
    probabilities = []

    label = 0
    class_probability = train_size / (train_size * label_count)

    label_zero_probability = calculate_probability(x, class_probability,
     mean_zero, std_zero)
```

```
    probabilities.append(label_zero_probability)

    label_one_probability = calculate_probability(x, class_probability,
    mean_one, std_one)
    if label_one_probability > label_zero_probability:
        label = 1
    probabilities.append(label_one_probability)

    label_two_probability = calculate_probability(x, class_probability,
    mean_two, std_two)
    probabilities.append(label_two_probability)
    if label_two_probability > label_one_probability:
        label = 2


    return label, probabilities
```

Listing 4.11 Naive Bayes classification method

In Listing 4.11 the probabilities for two labels are calculated as the case considered to be solved is a binary classification problem.

```
results_prob = []
prediction = []
for x in  test_set:
    pred, probabilities = naive_bayes_classifier(x)
    results_prob.append(probabilities)
    prediction.append(pred)

accuracy = calculate_accuracy(prediction, test_labels)
```

Listing 4.12 Naive Bayes classifier execution method

The implementation in Listing 4.12 the implementation of the classifier is given and the accuracy is calculated.

Example 6 (*Covid-19 probability*) Let B be a person with a positive COVID-19 test, A_1 be drawn by a random ill person, and A_2 be a healthy person. Let us assume that $P(A_1) = 0.05$, $P(A_2) = 0.95$, $P(B|A_1 = 0.92$, $P(B|A_2) = 0.10$. This means that the test is positive for 92% of ill persons and 10% for healthy persons It is important to get the probability of $A_1|B$ and it can be calculated as follows:

$$P(A_1|B) = \frac{P(B|A_1) \cdot P(A_1}{P(B|A_1) \cdot P(A_1) + P(B|A_2) \cdot P(A_2)} = \frac{0.92 \cdot 0.05}{0.92 \cdot 0.05 + 0.10 \cdot 0.95} \approx 0.3194.$$

This means that about 31.9% of patients with a positive test are actually ill.

Example 7 (*Iris classified using Naive Bayes method*) The Iris data set consists of three classes. To simplify the usage of the Naive Bayes method, we use only two. The data preparation is given in Listing 4.13.

```
from sklearn.model_selection import train_test_split

iris = load_iris()
data_set = iris.data
labels = iris.target
data_set = data_set[:,:2]
```

```
data_set = data_set[labels!=2]
labels = labels[labels!=2]

train_data_set, test_data_set, train_labels, test_labels =
    train_test_split(
    data_set, labels, test_size=0.2, random_state=15)

train_labels[train_labels<1] = -1
test_labels[test_labels<1] = -1

train_size = len(train_labels)
test_size = len(test_labels)
label_count = 2
feature_count = 2

mean_label0 = np.mean(train_data_set[[np.where(train_labels==-1)]][0],axis
    =1)
mean_label1 = np.mean(train_data_set[[np.where(train_labels==1)]][0],axis
    =1)

std0 = np.std(train_data_set[[np.where(train_labels==-1)]][0],axis=1)
std1 = np.std(train_data_set[[np.where(train_labels==1)]][0],axis=1)
```

Listing 4.13 Naive Bayes classifier execution method

The functions are next limited to two classes in the `naive_bayes_classifier` function. We changed the classes to 1 and −1 to simplify the classification. The other functions are changed slightly to get the means from a list, not from a DataFrame like in the previous Listing of this function.

```
from math import exp
from math import pi

def naive_bayes_classifier(x):
    probabilities = []

    label = -1
    class_probability = train_size / (train_size * label_count)

    label_zero_probability = calculate_probability(x, class_probability,
    mean_label0, std0)
    probabilities.append(label_zero_probability)

    label_one_probability = calculate_probability(x, class_probability,
    mean_label1, std1)
    if label_one_probability > label_zero_probability:
        label = 1
    probabilities.append(label_one_probability)

    return label, probabilities


def gaussian_pdf(x, mean, stdev):
    exponent = exp(-((x-mean)**2 / (2 * stdev**2)))
    return (1 / (sqrt(2 * pi) * stdev)) * exponent


def calculate_probability(x, class_probability, mean, stdev):
    probability = class_probability
    i = 0
    for feature in range(feature_count):
        probability *= gaussian_pdf(x[i], mean[0][feature], stdev[0][
    feature])
        i = i + 1
    return probability
```

Listing 4.14 Naive Bayes classifier execution method

It is interesting that for many sets of randomly chosen tests and trains, this method achieves an accuracy of 100%.

For Further Reading

1. Christensen R (2025) Log-linear models and logistic regression. Springer
2. Hilbe JM (2009) Logistic regression models. Chapman and Hall/CRC
3. Fisher RA (1936) The use of multiple measurements in taxonomic problems. Ann Eugen
4. Gutin G, Yeo A, Zverovich A (2002) Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP. Discrete Appl Math
5. Zhang H (2004) The optimality of Naive Bayes. FLAIRS

References

1. Fisher RA (1936) The use of multiple measurements in taxonomic problems. Ann Eugen 7:179–188
2. Anscombe FJ (1973) Graphs in statistical analysis. Am Stat 17–21
3. Kleinbaum DG, Klein M (2010) Logistic regression. A self-learning text. Springer

Chapter 5
Decision Trees

Decision trees are one of the most popular machine learning methods. One of the reasons is their easy usage and understanding. A decision tree is a method that can be easily visualized and understood. We have tens of different decision trees [1–17]. A decision tree is a method that divide the feature space on each level of a tree. It means it is a non-linear method because it does a linear classification at each node. The tree starts with a root and consists of decision nodes and leafs. It decouples the training set into smaller sets based on some conditions related to one (univariate) or more features (multivariate). As a result of the division, we can get one or more smaller data sets of different sizes. The goal of a decision tree is to build a tree in a way that we have objects of the same label in each leaf. A tree can be also written as a set of rules as it is based on a set of choices at each node. That is why it is commonly used in many decision-making software. It handles multiclass problems easily. We can use decision trees to understand which feature has the major impact on the classification. The more often a feature is used in decision nodes the higher impact it has on the classification. Compared to some other methods, decision trees do not work like a black box as hidden layers of a neural network. Another advantage of decision trees is their performance. Compared to most methods it is fast. On the other hand, a small change in training data can significantly change the rules and accuracy. Decision trees can also easily overfit. Usually, there is more than one tree that works well for a given data set. A more complex solution based on decision trees is called random forest. It is described in Chap. 7. Random forest is a combination of many decision trees. It usually gives much better results compared to decision trees. The random forest method is commonly used for many classification problems as it gives often better accuracy compared to regular neural networks or SVM. In this chapter, we explain how decision tree methods work. We divided this chapter into four parts. In the first part, we explain different types of trees and what the classification process looks like. To simplify we used binary trees. The second part is focused on univariate tree construction methods. It is an easier type of decision trees. We explain the most common methods like CART and C4.5. The third part is dedicated to multivariate

K. Przystalski et al., *Pattern Recognition Primer*,
https://doi.org/10.1007/978-3-031-91816-2_5

methods where the OC1 method is described in detail. The fourth part is about the quality metrics in decision trees. A major part of this section is dedicated to tree pruning methods.

5.1 Introduction to Tree-Based Classification

Decision trees are most likely binary trees. There are some exceptions, like the CHAID or C4.5 methods, but most methods stick to binary rules. Binary trees are trees in which each leaf/node has a maximum of two children. It is a structure where each child has its own child, etc. To better explain it, we prepared an abstract data set that is presented in Table 5.1. In the set, we have two features, as it is easier to understand when drawn on a two-dimensional space. The data set is divided into two classes $-1, 1$.

We can easily plot this data set as shown in Fig. 5.1. It is not a linear classification problem. We could try to use one of the previously explained methods, such as KNN. A univariate decision tree uses multiple linear decisions to divide the data set into sections of objects with the same label. At first glance, we might discover four such sections of objects: two red ones in the middle, one blue on the bottom, and one blue on the top.

The tree, when trained, makes a decision on one feature, which means that the line is perpendicular to the axis. For example, for $x_1 = 4$, the line would be parallel to the axis x_2 with a value of $x_1 = 4$. This would divide the data set into two smaller ones, one on the left where $x_1 < 4$ and the other on the right where $x_1 \geq 4$. We can now take both parts of the main data set and divide them again and again until we reach sets where we have all objects of the same label. The training part of a decision tree is to find out the division rules. An example of such a tree that divides the data set Table 5.1 is shown in Fig. 5.2. The tree has only four decisions, and the features are

Table 5.1 Decision tree example data

	x_{i1}	x_{i2}	Label		x_{i1}	x_{i2}	Label
x_1	0.5	0.5	-1	x_{11}	1	7	1
x_2	1.5	2	-1	x_{12}	2	8	1
x_3	2	4.5	-1	x_{13}	1.5	9	1
x_4	1	3.5	-1	x_{14}	4	2	1
x_5	2.5	4	-1	x_{15}	6	3	1
x_6	3.5	5.5	-1	x_{16}	7	4	1
x_7	6	6	-1	x_{17}	9	2.5	1
x_8	8	5.5	-1	x_{18}	5	9	1
x_9	9	6.5	-1	x_{19}	7	8.5	1
x_{10}	7.5	6	-1	x_{20}	9	8	1

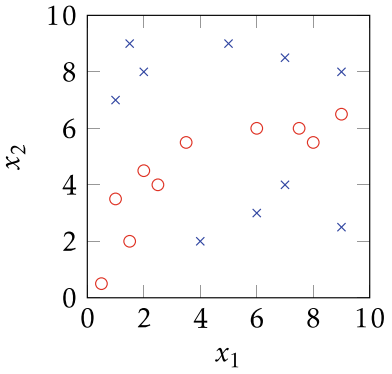


Fig. 5.1 Two-dimensional feature space with objects from the data set given in Table 5.1

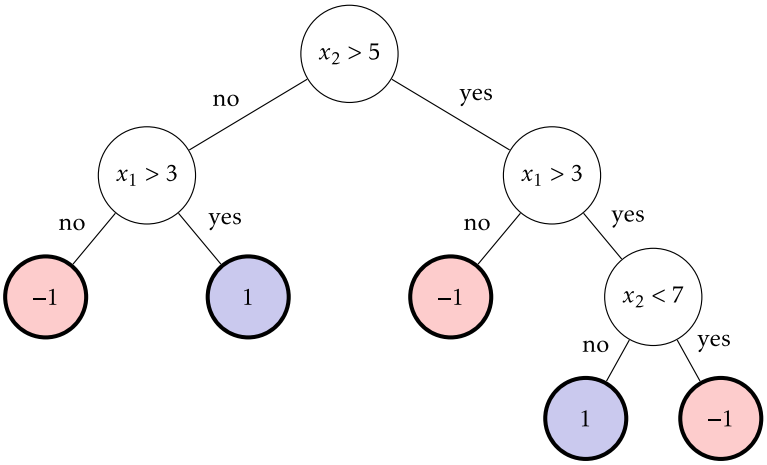


Fig. 5.2 Decision tree constructed for data shown in Table 5.1

used equally. This tree can be replaced with a nested set of rules. The rules in each node can be drawn as in Fig. 5.3. We see that the rules divided the feature space into five areas. What is important here is that all areas consist of an object of the same label. In the tree, we have marked sets that are made based on previous decisions with two colors: blue for the positive class and red for the negative class sets.

```
class BinaryLeaf:

    def __init__(self, elements, labels, ids):
        self.L = None
        self.R = None
        self.elements = elements
        self.labels = labels
        self.ids = ids
        self.completed = False
```

Listing 5.1 Tree leaf in Python

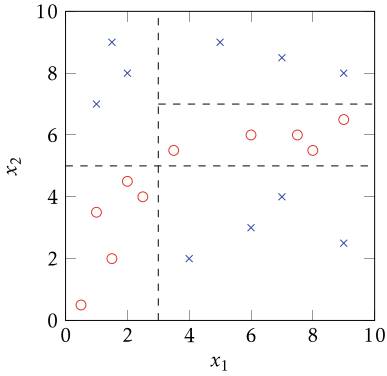


Fig. 5.3 Decision tree in two-dimensional feature space

The tree node consists of the left and right child node, objects (`elements`) that the current node consists. The `labels` property is a list of labels where each value is reflected to each object at this node. The last property is a Boolean field that gives us a better understanding of how this node should be divided. A node is completed if unique values of `label` list if equal to 1. The second reason is that objects cannot be divided anymore. In such cases, we return the prediction as the majority labels in this node, or if we have the same number of labels for each class, we return the prediction randomly.

5.2 Tree Operations

We have four major tree operations: growth, division, prune, and merge (aggregation). The tree growing as shown in Fig. 5.4a, b is an obvious operation. It is done during the training phase of a decision tree method. The goal is also to build a more accurate decision tree by adding new decision nodes. A similar operation is a split of data within a node into two leafs. Compared to tree growth operation, division is related to just one node split where grow is about a node grow, multiple splits. Pruning trees is a very important part of each decision tree method. The decision tree can easily overfit. It happens often when the tree is too complex. It will divide the feature space into too many small pieces and does not generalize the problem well. This can be fixed with tree pruning by reducing unnecessary nodes and leaves. There are different types of tree pruning. We dedicated a separate section for tree pruning methods as it is very important to know how to avoid overfitting in decision trees. The last tree operation that we would like to explain is tree aggregation. It also reduces the number of leafs as is done in pruning methods. The difference is that in pruning we deleted the leafs of a node, where aggregation means merging of two nodes on the same level.

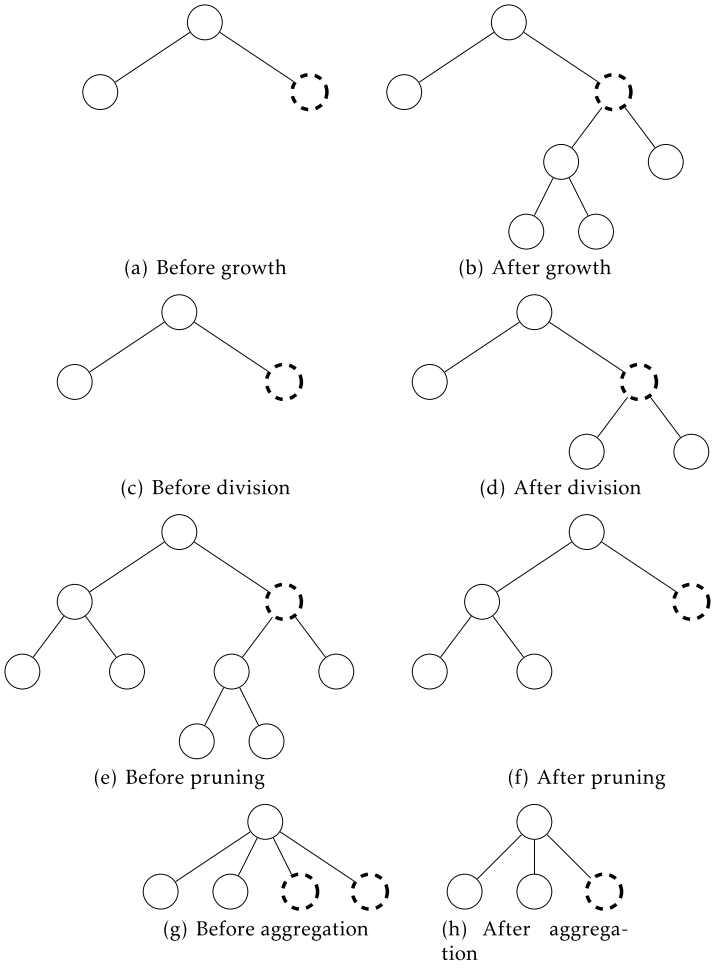


Fig. 5.4 Tree operations: growth, division, pruning, and aggregation

Fun fact: Decision trees are like overzealous know-it-alls—they eagerly make decisions based on what they've observed, but sometimes they come up with hilariously specific rules. For example, a poorly tuned decision tree might declare with absolute certainty, "If it's cloudy, you own three cats, and your neighbor plays jazz, then you should definitely buy a sports car!" [18]

5.3 Impurity Measures

Impurity measures are used to check how homogeneous or heterogeneous a given data set is. If a data set has many classes it means that it is heterogeneous (impure). The opposite is homogeneous, which means that the data set is pure. The impurity measures are used to split a data set into two child nodes in a decision tree. There are many impurity measures:

- Gini index,
- Entropy (information gain),
- Classification error,
- Likelihood-ratio method,
- DKM impurity method,
- Orthogonal criterion.

We describe only two methods as only these two are used by the methods we explain in this chapter. More sophisticated impurity methods exist. Likelihood-ratio method is based on Chi-Squared statistics and information gain [19]. A method that gives good results in small trees is the DKM impurity method [20]. There is a method based on the area under the curve (AUC) [21]. We have described AUC in Sect. 1.6. Distance measure can be used as a stop criterion [22]. A similar to gain ratio method was introduced in [23]. Orthogonal criterion [24] was proved to give better results than information gain in some cases. We can also use statistics measures like permutation [25] or probability measure [26, 27].

5.3.1 Gini Index

The Gini index [28] measures the probability of misclassification by a split. It means that the lower value of Gini index is a lower probability of misclassification of a given split. We can calculate the index value as

$$I_G(X) = 1 - \sum_{i=1}^{m} p_i^2, \tag{5.1}$$

where m is the number of potential discrete values or ranges of continuous values, and p is the probability of a specific feature value in the data set. The values of the Gini index are shown in the Fig. 5.5. If we assume we have a customer segmentation problem to solve where the label is the information if the customer buy or not. It depends on a few features and one of the features is the city. Let the city be Berlin and to simplify we have 4 cases where the customer buy and one where not. The index value of this split, based on Berlin would be:

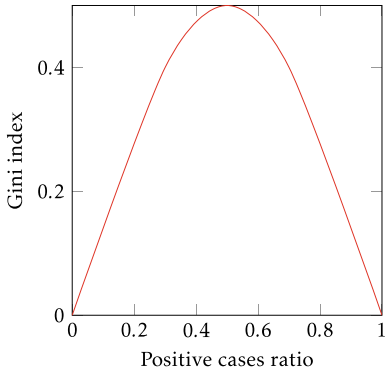


Fig. 5.5 Gini index values depending on the probability

$$I_G(\text{Berlin}) = 1 - \left(\frac{4}{5}\right)^2 + \left(\frac{1}{5}\right)^2 = 1 - (0.64 + 0.04) = 0.32.$$

The index needs to be calculated for each city against the other options if we want to build a binary tree because in this case, we can split only into two groups. It means that for the location feature, we need to calculate every possible combination of sets. In the case of three feature values, we have to calculate three Gini index value sets. The features gini index calculates the gini index for the split and takes the values of the features into consideration. The general formula of the feature split gini index is defined as

$$I_G(\text{feature}) = 1 - \sum_{i=1}^{n} p_i * I_G(X_i), \tag{5.2}$$

where X_i is the split set. For three cities: Berlin, London, and Paris we need to calculate the split gini index for three possible splits. If we assume we have an equal number of cities:

- Berlin versus London and Paris

$$I_G(\text{feature})_1 = 1 - \left(\frac{1}{3} I(\text{Berlin}) + \frac{2}{3} I(\text{London, Paris})\right),$$

- London versus Berlin and Paris

$$I_G(\text{feature})_2 = 1 - \left(\frac{1}{3} I(\text{London}) + \frac{2}{3} I(\text{Berlin, Paris})\right),$$

- Paris versus London and Berlin

$$I_G(\text{feature})_3 = 1 - \left(\frac{1}{3} I(\text{Paris}) + \frac{2}{3} I(\text{London, Berlin})\right).$$

We should calculate all possible splits on each feature to get the final lowest value of the Gini index in each leaf. A small Gini index value means more objects of the same label in a leaf. We should split on the highest gini index value to get in the leafs lower and lower values in each next split.

5.3.2 Entropy and Information Gain

Entropy (information gain) [29] is used to measure how *many* information we have within a set. In other words, if there are more objects of the same label in a set there higher the entropy measure is. It means that higher entropy is a better factor for a split. Possible values are given in Fig. 5.6. The formula for the entropy is defined as

$$E(X) = -\sum_{i=1}^{m} p_i \log_2 pi. \tag{5.3}$$

To take a decision on what features should we split on, we need to calculate the information gain value based on the features entropy and the total entropy that we get based on all features:

$$IG(\text{Feature}) = E(\text{Decision}) - E(\text{Feature}), \tag{5.4}$$

where $E(\text{Decision})$ is the total entropy where we take into consideration all features. We use for Eq. 5.3. If we have 9 objects in our data set, the $E(\text{Decision})$ takes all objects for example:

$$E(\text{Decision}) = -\left(\frac{5}{9}\log_2\frac{5}{9} + \frac{4}{9}\log_2\frac{4}{9}\right) = -1(-0.0471142 - 0.5199787) = 0.991076.$$

The feature entropy is calculated as a weighted entropy of all possible entropy values of a given features:

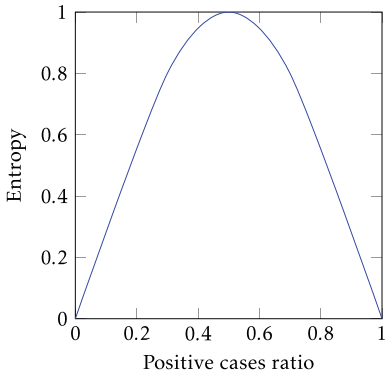


Fig. 5.6 Entropy values for a given probability

$$E(\text{Feature}) = \sum_{i=0}^{m} p_i E(x_i). \tag{5.5}$$

Let's assume we have the same 9 objects where the locations are equally divided by London, Berlin, and Paris. If we have two objects with a positive decision where the location is London, one for Berlin, and none for Paris, we get

$$\begin{aligned} E(\text{London}) &= -1\left(\left(\frac{2}{3}\right)\log_2\left(\frac{2}{3}\right) + \left(\frac{1}{3}\right)2\log_2\left(\frac{1}{3}\right)\right) \\ &= 0.918296, \end{aligned}$$

$$\begin{aligned} E(\text{Berlin}) &= -1\left(\left(\frac{1}{3}\right)\log_2\left(\frac{1}{3}\right) + \left(\frac{2}{3}\right)2\log_2\left(\frac{2}{3}\right)\right) \\ &= 0.918296, \end{aligned}$$

$$\begin{aligned} E(\text{Paris}) &= -1\left(\left(\frac{3}{3}\right)2\log_2\left(\frac{3}{3}\right)\right) \\ &= 0. \end{aligned}$$

The location entropy will be

$$E(\text{Location}) = \frac{3}{9} * 0.918296 + \frac{3}{9} * 0.918296 + 0 = 0.6121947.$$

The information gained for the location features is

$$IG(\text{Location}) = 0.991076 - 0.6121947 = 0.3788813.$$

We should do it for all features and compare the values. The highest should be taken as the feature that we split on. The data set where we do the calculation is the same as in Gini index and is limited to the data set of a node that we want to split.

5.4 Binary Trees with Classification and Regression Trees Method

Classification and regression trees (CART) are one of the most popular and one of the first decision tree methods [30]. Build a decision tree based on a binary tree. This means that it divides the data set into two on each level of a tree. CART uses the Gini index to find the best possible split. The method consists of the following steps:

1. calculate the Gini index for each feature,
2. take the lowest value of the Gini index and split the node into two child nodes,
3. repeat the steps until we have all child nodes.

The leaf in CART looks like

Tree helper functions

There are several helper functions that are used by our CART method. We use the methods to manipulate the tree, use these to get the Gini index or make the split. We have seven methods. The implementation of each function is given in Listing 5.2.

```
def get_number_of_labels_for_value(node, objects, feature_id, label):
    count = 0
    if not isinstance(objects, list):
        elements_list = [objects]
    else:
        elements_list = objects

    column_elements = get_node_elements_column(node.elements, feature_id)

    for i in range(len(elements_list)):
        for j in range(len(column_elements)):
            if column_elements[j] == elements_list[i]:
                if node.labels[j] == label:
                    count = count + 1
    return count

def get_node_elements_column(elements, feature_id):
    return np.array(elements)[..., feature_id].tolist()

def check_completed(labels, elements):
    ratio = len(get_unique_labels(labels))
    if ratio == 1:
        return True
    elements = sorted(elements)
    duplicated = [elements[i] for i in range(len(elements)) if i == 0 or
    elements[i] != elements[i - 1]]
    if len(duplicated) == 1:
        return True
    return False

def get_unique_labels(labels):
    return np.unique(np.array(labels)).tolist()

def get_unique_values(elements):
    features_number = len(elements[0])
    unique = []
    for i in range(features_number):
        unique.append(np.unique(np.array(elements)[:,i]))
    return unique

def is_leaf_completed(node):
    if node.is_completed():
        if node.get_L() != None and not node.get_L().is_completed():
            return node.get_L()
        elif node.get_R() != None and not node.get_R().is_completed():
            return node.get_R()
        elif node.get_L() == None and node.get_R() == None:
            return None
        elif node.get_L().is_completed() or node.get_R().is_completed():
```

```
            new_node = is_leaf_completed(node.get_L())
            if new_node == None:
                return is_leaf_completed(node.get_R())
            else:
                return new_node
        else:
            return None
    return node

def get_current_node(node):
    return is_leaf_completed(node)
```

Listing 5.2 Tree helper functions

The method `get_unique_labels` returns the unique labels based on the list of labels given in the argument. The method `get_unique_values` is a bit more complex because it is not only about unique objects in a list, but about unique objects in a list of lists. `is_leaf_completed` is a very important method to check if a leaf needs to be split or not. It goes through the node given in the argument and checks each leaf that still needs to be split or not. It goes deeper and deeper through the tree until it finds a leaf. If not, it returns a `None` value. The `get_number_of_labels_for_value` method is used by the Gini index calculation method where the number of labels occurrence for a specific feature value is a part of the Gini equation. This method uses the `get_node_elements_column` method to get the feature columns. The sixth method `check_completed` is used by the split method to check if the leaf can be marked as the one that still needs to be split or not. The last one `get_current_node` is used to find the current leaf to split. It uses the `is_leaf_completed` method to make the check.

Gini index

The Gini index is calculated with the equations explained in the previous section. For a binary tree, it can be implemented as in the Listing 5.3.

```
def calculate_gini(node, splits, feature_id):
    obj_count = len(node.labels)

    left_count = np.count_nonzero(np.isin(np.array(node.elements)
    [:,feature_id], splits[0]))
    right_count = obj_count - left_count

    prob_sum_left  = 1
    prob_sum_right = 1

    for label in get_unique_labels(node.labels):
        prob_sum_left  = prob_sum_left - (get_number_of_labels_for_value
        (node, splits[0], feature_id, label)/left_count)**2
        prob_sum_right = prob_sum_right - (get_number_of_labels_for_value
        (node, splits[1], feature_id, label)/right_count)**2
    return 1 - (left_count/obj_count)* prob_sum_left - (right_count/
     obj_count)* prob_sum_right
```

Listing 5.3 Gini index method

In the binary tree, we have only the left and right leaf. As arguments, we get the current node, a list of split objects, and the feature. In the fourth line, we get

the `left_count` which is the number of objects of a given value for feature `feature_id`. In the loop, we go through each label and find the values that are available in the data set for it. This gives us the Gini index values for the left and right leaf. The function returns the feature gini index using the previously calculated left and right gini indices.

Split

The split is done in two steps in the Code 5.4. The method `get_split_candidates` gets unique values of a feature as an argument and returns all possible split scenarios as a list. Loops over the unique values and adds one value as the left split and the rest as the right. The loop ends when there are no more feature values left.

```
def get_split_candidates(unique_values):
    split_list = []
    for i in range(len(unique_values)):
        current_list = []
        temp_list = copy.deepcopy(unique_values)
        current_list.append(temp_list[i])
        del temp_list[i]
        current_list.append(temp_list)
        split_list.append(current_list)
    return split_list

def split_node(current_node, value, split_id, split_history):
    left_leaf = []
    left_leaf_labels = []
    left_leaf_ids = []
    right_leaf = []
    right_leaf_labels = []
    right_leaf_ids = []
    for i in range(len(current_node.elements)):
        if current_node.elements[i][split_id] == value:
            left_leaf.append(current_node.elements[i])
            left_leaf_labels.append(current_node.labels[i])
            left_leaf_ids.append(current_node.ids[i])
        else:
            right_leaf.append(current_node.elements[i])
            right_leaf_labels.append(current_node.labels[i])
            right_leaf_ids.append(current_node.ids[i])
    if len(right_leaf_labels) == 0 or len(left_leaf_labels) == 0:
        current_node.set_completed()
        return current_node, split_history
    split_history.append([str(current_node.ids), str(left_leaf_ids)])
    split_history.append([str(current_node.ids), str(right_leaf_ids)])
    current_node.set_L(BinaryLeaf(left_leaf, left_leaf_labels,
    left_leaf_ids))
    current_node.set_R(BinaryLeaf(right_leaf, right_leaf_labels,
    right_leaf_ids))
    current_node.set_split(split_id)
    current_node.set_completed()
    if check_completed(left_leaf_labels, left_leaf):
        current_node.L.set_completed()
    if check_completed(right_leaf_labels, right_leaf):
        current_node.R.set_completed()
    return current_node, split_history
```

Listing 5.4 Splitting function in CART method

The second method divides. It takes the node that we want to split as one of the arguments. The second is the `value` of the feature that we assign to the left leaf. The third argument is the `split_id` that has the highest gini index. The loop is the main part of the function where the split to the left and right leaf happens. We add node objects, labels, and ids to the new pages. The `ids` are saved in a leaf property, because sometimes it is easier to use the id of an object rather than the whole description of it. Especially when we want to plot it. The last argument is used to save the split history in one variable.

Build a tree

In the building method, we use the data set of objects (feature vectors) and the labels that correspond to them. The `build` method implemented in the Listing 5.5 consists of four parts: initialization part, find the best combination of features for the split, make the split, and find the next node to split.

```
1  def build(data_set, labels):
2      stop_criterion = False
3      ids = list(range(len(data_set)))
4      root = BinaryLeaf(data_set, labels, ids)
5      current_node = root
6      split_history = []
7
8      while stop_criterion == False:
9          unique_values = get_unique_values(current_node.get_elements())
10         max_unique_id = 0
11         max_split_id = 0
12         max_value = 0.0
13         for feature_id in range(len(unique_values)):
14             if len(unique_values[feature_id]) == 1:
15                 continue
16             split_candidates = get_split_candidates(unique_values
17             [feature_id].tolist())
18             for j in range(len(split_candidates)):
19                 current_value = calculate_gini(current_node,
       split_candidates[j],feature_id)
20                 if max_value < current_value:
21                     max_unique_id = feature_id
22                     max_split_id = j
23                     max_value = current_value
24         current_node, split_history = split_node(current_node,
       unique_values[max_unique_id][max_split_id], max_unique_id,
       split_history)
25         new_node = get_current_node(root)
26         if new_node != None:
27             current_node = new_node
28         else:
29             stop_criterion = True
30     return root, split_history
```

Listing 5.5 CART tree build main method

The initialization part consists of the root node setup. We create just an instance of a `BinaryLeaf` and set all properties like the objects, labels, and ids. The history is set to an empty list. The stop criterion is set to False and is checked at the end of the loop. In the second part, we loop over the unique feature values to find the maximum Gini index. It's calculated in line 18 and the check is done in the line after. The split

is done for the best split candidate in line 23. Finally, we get the next node that can be split in line 24. If the returned value is different than `None`, the stop criterion is not met and we use the next node and take the actions again from the beginning of the loop. The new node is searched using the root node and not the currently used node. This guarantees that we check all possible nodes in the tree.

```
1 def plot_tree(split_history):
2     tree = pydot.Dot(graph_type='graph')
3     for split in split_history:
4         new_edge = pydot.Edge(split[0], split[1])
5         tree.add_edge(new_edge)
6     tree.write('cart_tree_new.png', format='png')
```

Listing 5.6 CART tree build main method

When the tree is built, we can also plot using the `pydot` package. It is dedicated to plot nodes and connections between these. In the Listing 5.6 we use this library and loop over the split history to draw the tree. The method loops over each split and adds nodes in line 4 and the connection between the nodes in line 5. We can save it as an image as shown in line 6.

Example 1 (*Customer segmentation with CART decision tree method*) In this example, we use a decision tree for customer segmentation. The goal is to find out if there are some features that are specific to customers who buy the product or not. We simplified the data set to be able to follow the steps of the CART method. The data set that we use in this example is given in Table 5.2. We have four features: location, product category, customer gender, and information if the customer checked the review before the purchase. CART generates binary trees, so we need to calculate the Gini index for each feature. Let us take the location feature as the first feature and calculate the Gini index for it. We have three split candidates that we explained already:

- London versus Berlin and Paris
- Berlin versus London and Paris,
- Paris versus London and Berlin.

The first split candidates are London versus Berlin and Paris. The Gini index for London can be calculated as

$$I_G(\text{London}) = 1 - \left(\frac{2}{7}\right)^2 + \left(\frac{5}{7}\right)^2 = 1 - (0.081 + 0.51) = 0.4087.$$

We have seven objects where the location feature is London. The decision ratio is two to five, which means that two customers from London bought our product and five did not. For the right leaf, we get

$$I_G(\text{Paris and Berlin}) = 1 - \left(\frac{7}{8}\right)^2 + \left(\frac{1}{8}\right)^2 = 1 - (0.76 + 0.015) = 0.22.$$

Table 5.2 Customer segmentation data set example

ID	Location	Category	Gender	Product review check	Customer decision
1	Berlin	Furniture	Male	Yes	True
2	London	Furniture	Male	Yes	True
3	Berlin	Furniture	Female	Yes	False
4	Berlin	Textile	Female	Yes	True
5	London	Electronics	Male	Yes	False
6	London	Textile	Female	Yes	False
7	Paris	Textile	Male	No	True
8	Berlin	Electronics	Male	Yes	True
9	Paris	Electronics	Male	No	True
10	London	Electronics	Female	Yes	True
11	Paris	Furniture	Female	No	True
12	Berlin	Textile	Female	No	True
13	London	Electronics	Female	No	False
14	London	Furniture	Female	Yes	False
15	London	Textile	Female	No	False

We have seven objects from London, and the total number of objects in our data set is 15. This means that we have eight Berlin and Paris objects together. For both, seven bought the product and one did not. Now, we can calculate the Gini index for the location feature:

$$I_G^1(\text{Location}) = \frac{7}{15} * 0.4087 + \frac{8}{15} * 0.22 = 1 - 0.1907 - 0.1173 = 0.69194.$$

The results for all combinations of all features are given in Table 5.3. If we do the first split we end with a tree of two levels that looks as in Fig. 5.7. To find the next node we first go through the left leafs and next to the right if there is no node to be split on the left side. We can see it in the Listing 5.2 in the method `is_leaf_complete`. The first test is done on line 42 on the left node and on the right. We can do it randomly or change the order to start the split in a different order. In the left-first approach, the next node is the one with all customers from London. In the second node, we repeat the steps from the first step. This time we have customers only from London and this is the reason why we cannot split on location features this time. In the second step, we get the Gini index values as in Table 5.4. This time we split up into categories. We have two Gini indices that have the same values, but in Listing 5.5 in line 19, we have a less sign that can be replaced with less or equal sign to get the last option if there are two same values. After the second step, we get a tree as in Fig. 5.8. The difference in the second split is that all London customers that are looking on textile products do not buy the product. The other two features do not matter for this path.

Table 5.3 Gini index split values for all features of the example data set in the first step

Nodes	Gini index
London	0.4087
Paris and Berlin	0.22
Gini index	0.69194
Berlin	0.32
London and Berlin	0.5
Gini index	0.56
Paris	0
London and Berlin	0.5
Gini index	0.6
Category	Gini index
Furniture	0.48
Textile and electronics	0.48
Gini index	0.59
Textile	0.48
Furniture and electronics	0.48
Gini index	0.52
Electronics	0.48
Textile and furniture	0.48
Gini index	0.52
Gender	Gini index
Male	0.49
Female	0.27
Gini index	0.59
Review	Gini index
Review	0.44
Direct	0.49
Gini index	0.52

We come to this conclusion because all objects in the node marked with red have the same negative label. After a few more steps the method ends, because there are no more nodes to split. The final tree looks as shown in Fig. 5.9. The red nodes are the nodes with only negative labels and the blue nodes are the nodes where customers buy the product. As a company, we want to have such a solution to understand the profile of our customers and add changes to our solution to fill more of these red nodes in blue ones.

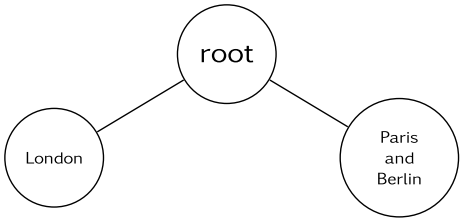


Fig. 5.7 Tree after the first step of the CART method

Table 5.4 Gini index split values for category, gender, and review features of the example data set in the second step

Category	Gini index
Furniture	0.5
Textile and electronics	0.32
Gini index	0.63
Textile	0
Furniture and electronics	0.48
Gini index	0.66
Electronics	0.44
Textile and furniture	0.375
Gini index	0.59
Gender	Gini index
Male	0.32
Female	0.5
Gini index	0.63
Review	Gini index
Review	0
Direct	0.48
Gini index	0.66

5.5 Univariate Non-binary Trees with C4.5 Method

This method [31] uses the entropy as a measure of the split of the nodes. It does not create a binary tree so that each node level can have more than two children. The algorithm steps are similar to those in the case of CART. The tree node is

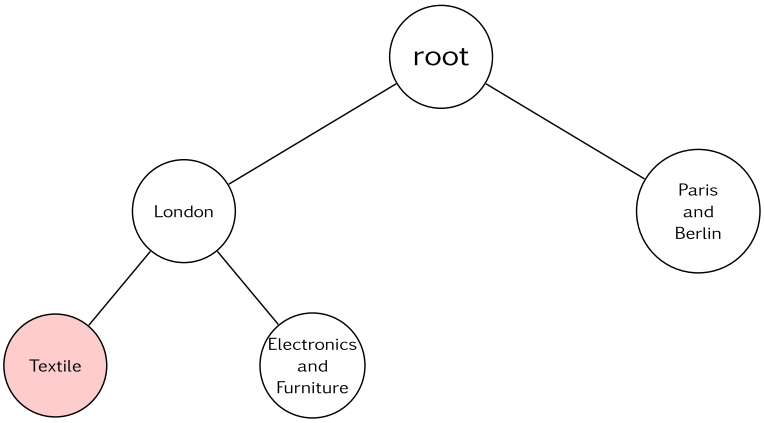


Fig. 5.8 Decision tree built on the second step of the CART method

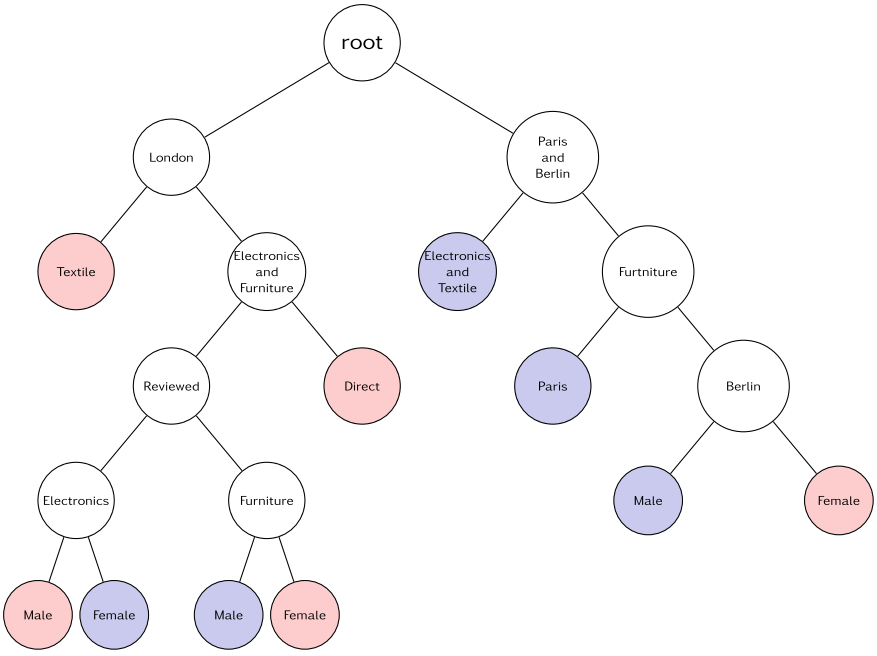


Fig. 5.9 Decision tree build using CART method based on the customer segmentation data set

implemented in a different way to be compliant with the non-binary tree approach. It does not have the `L` and `R` properties, but instead, we have a list of child leafs that is set as a `child_leafs` property. The rest is the same as in the CART leaf. The implementation of the node is given in Listing 5.7.

```
class Leaf:

    def __init__(self, elements, labels, ids):
        self.child_leafs = []
        self.elements = elements
        self.labels = labels
        self.completed = False
        self.ids = ids
```

Listing 5.7 A non-binary leaf

The helper functions known from the CART are the same, because all except one work for binary and non-binary trees in the same way. This means that there are no changes in the helper methods except the `is_leaf_completed` method that takes the node as an argument.

```
def is_leaf_completed(node):
    if node.is_completed():
        child_nodes = node.get_child_leafs()
        if len(child_nodes) == 0:
            return None
        is_child_to_return = False
        for i in range(len(child_nodes)):
            if not child_nodes[i].is_completed():
                return child_nodes[i]
            else:
                new_node = is_leaf_completed(child_nodes[i])
                if new_node != None:
                    is_child_to_return=True
        if is_child_to_return:
            return new_node
    return node
```

Listing 5.8 Helper function differences

Instead of a check on the left and right nodes, in this case we need to loop over all child nodes and check which node is still one that can be split.

Entropy

The entropy is a short function in which we take the labels of a node as an argument. In the Listing 5.9 we get the unique labels and count them in the first step.

```
def calculate_entropy(labels):
    unique_labels, labels_count = np.unique(labels, return_counts=True)
    entropy = 0
    size = len(labels)
    for i in range(len(unique_labels)):
        if labels_count[i] > 0:
            log2 = log((labels_count[i] * 1.0) / (size * 1.0), 2)
        else:
            log2 = 0.0
        entropy = entropy - 1.0 * ((labels_count[i] * 1.0) / size) * log2
    return entropy
```

Listing 5.9 Entropy and information gain calculation for the C4.5 method

Next, we loop over the unique labels and calculate the entropy values with $\log_2$ for each label. Finally, we subtract the values from the results to get the decision entropy E(Customer) for a given feature. This method is then used in the `calculate_split_candidate_entropy` to get the entropy for each split candidate.

Split

The split consists of two steps: possible splits entropy calculation and the split method itself. In the Listing 5.10 one of the possible implementations is given. In the method `calculate_split_candidate_entropy` we calculate the information grain based on the decision entropy given in the argument as `full_entropy` and the split candidates' entropies. We loop over unique objects and count the labels assigned to each one. For each label for which the count is greater than 0, we add the entropy to the `split_entropy`. This part is the implementation of Eq. 5.3. The last line of this function is the implementation of Eq. 5.4.

```
def calculate_split_candidate_entropy(full_entropy, labels, elements,
    unique_labels, unique_elements, iter):
    split_entropy = 0
    for i in range(len(unique_elements)):
        indices = np.where(np.array(elements)[..., iter].tolist() ==
    unique_elements[i])
        unique_size = len(indices[0].tolist())
        filtered_labels = np.array(labels)[indices]
        for j in range(len(unique_labels)):
            labels_count = filtered_labels.tolist().count(unique_labels
            [j])
            if labels_count > 0:
                log2 = log((labels_count * 1.0) / (unique_size * 1.0), 2)
            else:
                log2 = 0.0
            split_entropy = split_entropy - 1.0 * (
                    (labels_count * 1.0) / unique_size * 1.0) * log2 *
    unique_size * 1.0 / len(elements) * 1.0
    return (full_entropy - split_entropy)

def split(current_node, split_values, column_id, split_history):
    new_leafs = []
    for i in range(len(split_values)):
        indices = np.where(np.array(current_node.get_elements())[...,
    column_id].tolist() == split_values[i])
        new_leaf_elements = np.array(current_node.get_elements())[indices
    ].tolist()
        new_leaf_labels   = np.array(current_node.get_labels())[indices].
    tolist()
        new_leaf_ids = np.array(current_node.get_ids())[indices].tolist()
        new_leaf = Leaf(new_leaf_elements,new_leaf_labels, new_leaf_ids)
        split_history.append([str(current_node.ids), str(new_leaf_ids)])
        if len(np.unique(new_leaf_labels)) == 1:
            new_leaf.set_completed()
        new_leafs.append(new_leaf)
    current_node.set_child_leafs(new_leafs)
    current_node.set_completed()
    return current_node, split_history
```

Listing 5.10 C4.5 tree build main method

The second function is similar to the one of CART that generates possible divisions. In the implementation of the function in Listing 5.4 the main difference compared to the CART method is the way how we make the split. The `new_leaf` variable contains the newly created leaf created in the loop based on the `split_values`. We set all three properties in the first part of the loop: labels, objects, and ids. In the second part, we check if this leaf can be split in the next loop of the method or is already a consistent leaf.

The tree

The building method is similar to the CART with a few small differences. An implementation is given in the Listing 5.11.

```
def build(root):
    stop_criterion = False
    split_history = []
    current_node = root
    unique_labels = get_unique_labels(root.get_labels())
    while stop_criterion == False:
        unique_values = get_unique_values(current_node.get_elements())
        full_entropy = calculate_entropy(current_node.get_labels())
        max_entropy_id = 0
        max_entropy_value = 0
        for i in range(len(unique_values)):
            split_entropy = calculate_split_candidate_entropy(full_entropy
    ,
    current_node.get_labels(),
    current_node.get_elements(),
    unique_labels,
    unique_values[i], i)
            if split_entropy > max_entropy_value:
                max_entropy_id = i
                max_entropy_value = split_entropy
        current_node, split_history = split(current_node, unique_values[
    max_entropy_id], max_entropy_id, split_history)
        new_node = get_current_node(root)
        if new_node != None:
            current_node = new_node
        else:
            stop_criterion = True
    return root, split_history
```

Listing 5.11 C4.5 tree build main method

In the first part of the loop, we calculate the decision entropy and save it in the `full_entropy` variable. The method for each split then calculates the information gain and finds the lowest one. The last lines are exactly the same as in the CART tree-building method.

Example 2 (*Customer segmentation with C4.5*) In this example, we calculate the entropies of each feature value. For the customer segmentation data set and the location feature first possible split we get

$$E(\text{London}) = -1\left(\left(\frac{2}{7}\right)\log_2\left(\frac{2}{7}\right) + \left(\frac{5}{7}\right)2\log_2\left(\frac{5}{7}\right)\right)$$
$$= -1(0.2857 * (-1.807) + 0.7142 * (-0.4895)) = 0.8617,$$

$$E(\text{Berlin}) = -1\left(\left(\frac{4}{5}\right)\log_2\left(\frac{4}{5}\right) + \left(\frac{1}{5}\right)\log_2\left(\frac{1}{5}\right)\right)$$
$$= -1(0.8 * (-0.3219) + 0.2 * (-2.319)) = 0.7218,$$

$$E(\text{Paris}) = -1(\frac{3}{3}\log_2(1)) = 0.$$

The next step is to calculate the customer decision entropy. It is called also a target or total entropy:

$$E(\text{Customer}) = -1\left(\frac{9}{15}\log_2\left(\frac{9}{15}\right) + \frac{6}{15}\log_2\left(\frac{6}{15}\right)\right)$$
$$= -1(0.6 * (-0.7365) + 0.4 * (-1.3219)) = 0.9708.$$

Next, we need to calculate the entropy for the location feature:

$$E(\text{Location}) = \frac{5}{15} * 0.7218 + \frac{7}{15} * 0.8617 + \frac{3}{15} * 0 = 0.2406 + 0.4021 = 0.6427.$$

Finally, we can calculate the information gain for the location feature:

$$IG(\text{Location}) = E(\text{Customer}) - E(\text{Location}) = 0.9708 - 0.6427 = 0.3281.$$

We calculate the information gain for each feature. The customer decision entropy $E(\text{Customer})$ stays the same for each feature. The entropies for each feature are given in Table 5.5. For the category features, the information gain is 0, because the entropies for each value are the same as the decision entropy. The data is divided in the same ratio for the category feature as for the whole data set. The gender information gain is the second highest value. The gain in product review feature information is low as well. The node is divided by the feature with the highest information gain. In the first step, it is the location feature. C4.5 will divide the root node into three leaf based on as shown in Fig. 5.10. The Paris node has already all objects of the same label. In this case, all Parisian customers buy the product. In the second step, we have two nodes to choose from: Berlin and London customers. For both nodes, we can skip the location feature entropy calculation, because there is only one unique value available. The decision entropy is calculated only on the node objects set and is for customers from Berlin in the level two node:

Table 5.5 Entropy values for category, gender, and review features of the example data set in the first step of the C4.5 method

Category	Entropy
Furniture	0.97095
Textile	0.97095
Electronics	0.97095
Category entropy	0.97095
Information gain	0
Gender	Entropy
Male	0.991076
Female	0.65
Gender entropy	0.85465
Information gain	0.116296
Review	Entropy
Review	0.9182958
Direct	0.991076
Product review entropy	0.961964
Information gain	0.008986

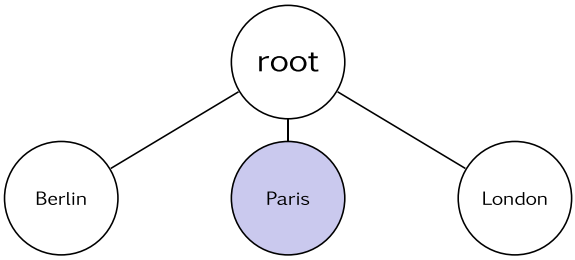


Fig. 5.10 First split with C4.5 method and the customer segmentation data set

$$E(\text{Decision}) = -1\left(\frac{1}{5}\log_2\frac{1}{5} + \frac{4}{5}\log_2\frac{4}{5}\right) = 0.7219.$$

The category entropy equals to 1 for the furniture and zero for textiles and electronics:

$$E(\text{Furniture}) = -1\left(\frac{1}{2}\log_2\frac{1}{2} + \frac{1}{2}\log_2\frac{1}{2}\right) = 1,$$

$$E(\text{Textile}) = -1\left(\frac{0}{2}\log_2\frac{0}{2} + \frac{2}{2}\log_2\frac{2}{2}\right) = 0,$$

$$E(\text{Electronics}) = -1(1\log_2 1) = 0.$$

Table 5.6 Entropy values for category, gender, and review features of the example data set in the second step of the C4.5 method

Gender	Entropy
Male	0.92183
Female	0
Gender entropy	0.55098
Information gain	0.171
Review	Entropy
Review	0
Direct	0.81128
Product review entropy	0.649
Information gain	0.07289

This gives us the entropy for the category features as

$$E(\text{Category}) = \frac{2}{5}.$$

This makes the information gain:

$$IG(\text{Category}) = 0.7219 - 0.4 = 03219.$$

The other results of the information gain and entropies are given in Table 5.6. The category feature has the highest information gain. The tree after the second split looks as shown in Fig. 5.11. This time two more nodes have objects with only one label. After a few more iterations we get the final tree. The final decision tree for customer segmentation using the C4.5 method is shown in Fig. 5.12. The final tree has five levels and 18 nodes, including 11 nodes with objects of the same label. Compared to the tree built using the CART method, this tree has less levels by one and less nodes. The conclusion is that the C4.5 can be used to build less complex tress.

5.6 Multivariate Decision Trees with OC1 Method

There are many multivariate decision tree methods. Just to mention a few of the most popular:

- OC1,
- LMDT,
- CART-LC,
- MARS.

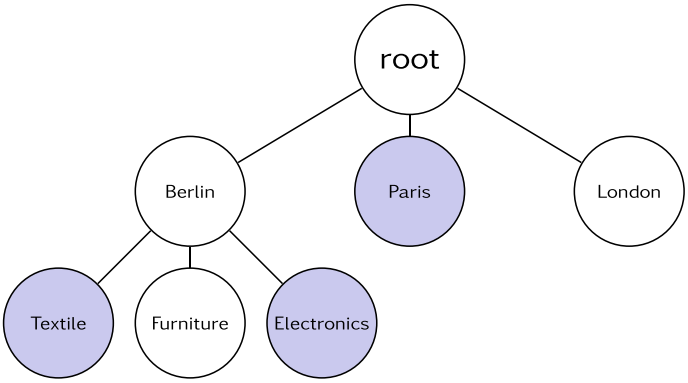


Fig. 5.11 Second step of the C4.5 method and customer segmentation data set

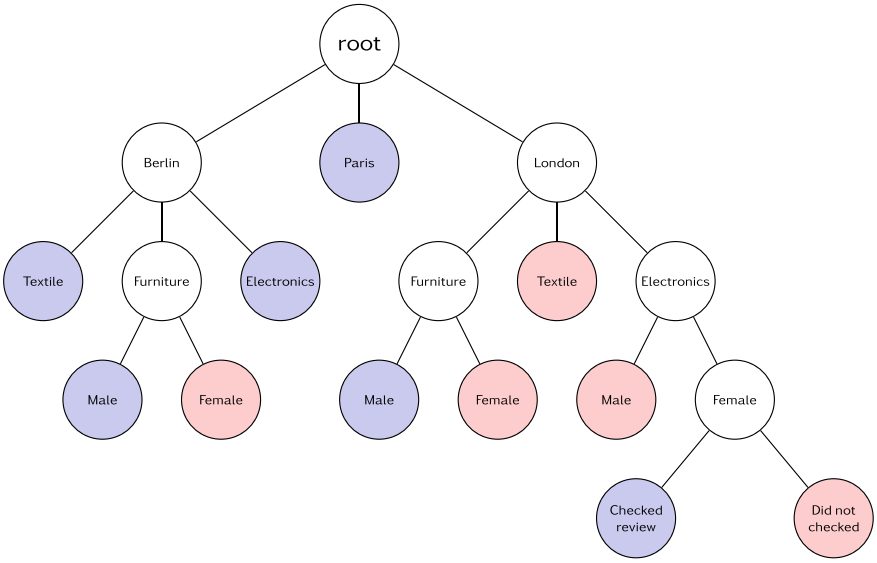


Fig. 5.12 Decision tree based on the data set 5.2 build with the C4.5 method

Some methods like CART-LC are the multivariate version of the CART method. The main difference between univariate and multivariate methods is that the decision is made based not on just one feature, but on more. It is easy to illustrate the advantages of this approach. In Fig. 5.13 the data set marked with red and blue is not linearly separable. The univariate tree method divides the feature space into several sections that can be summarized as a decision boundary marked with orange. With two features, we can draw a boundary marked with green. Such an approach requires fewer steps and is more efficient in many cases. The multivariate method

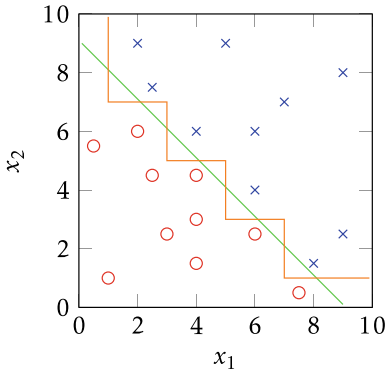


Fig. 5.13 Linear separable example classified with univariate (marked yellow) and multivariate decision trees (marked green)

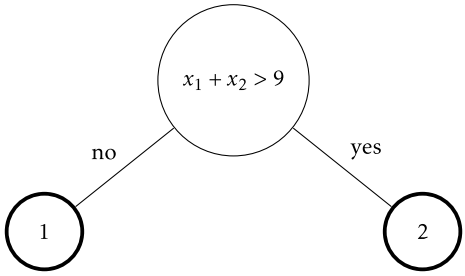


Fig. 5.14 Multivariate decision tree based on data from Listing 5.13

used to classify the data set in Fig. 5.13 can be written as a linear function as shown in Fig. 5.14. There are a few advantages of multivariate decision trees:

- we split on more than one feature, which usually gives a tree with a smaller high value, which means a faster prediction,
- solve non-linear classification problems.

The disadvantage of a multivariate method is the parameters that need to be set. Almost every univariate decision tree method can be easily changed to multivariate methods. In this section, as the multivariate method, we show the implementation of the OC1 classifier. The OC1 classifier is divided into several steps:

1. get possible hyperplanes H,
2. choose one hypothesis,
3. perturb and find v_j,
4. calculate gini index of each H_i,
5. choose H_i with lowest gini index.

The OC1 method builds binary trees. It

Get possible hyperplanes

In the method in Listing 5.12 we calculated all possible hyperplanes by calculating the Gini indices for each feature. It is kind of similar to what we have done in the CART method, but it will be *fixed* during the perturbation part of the OC1 method.

```
def get_all_possible_splits_by_gini(self,leaf):
    data_set = leaf.elements
    labels = leaf.labels
    ginis = []
    for i in range(self.feature_number):
        feature_ginis = []
        feature_column = data_set[:, i]
        for feature in feature_column:
            distinguish = feature_column <= feature
            left_labels  = labels[distinguish]
            right_labels = labels[~distinguish]
            gini = 1 - self.calculate_gini(left_labels) - self.
    calculate_gini(right_labels)
            feature_ginis.append([feature,gini])
        ginis.append(min(feature_ginis))
    return ginis
```

Listing 5.12 Get all possible splits and sort it by gini index value

In the method below (Listing 5.13) we compute the V_j which gives us the knowledge if a given object is above or below the hyperplane. It can be formulated as

$$\sum_{i=1}^{d} a_i x_i + a_{d+1} > 0, \tag{5.6}$$

where $a_1, \ldots, a_{d+1}$ are coefficients. In our case a_{d+1} is our label.

```
def get_coefficiency(self, splits):
    scv = np.zeros(len(splits)+1)
    min_split_index = np.argmin(splits)
    scv[min_split_index] = 1
    scv[-1] = -splits[min_split_index][1]
    return scv
```

Listing 5.13 Calculate the coefficiencies

The next step is to divide objects in the leaf into two sets which are above and below the hyperplane (see Listing 5.14).

```
def divide_data_hiperplane(self,leaf,scv):
    below = []
    above = []
    below_labels = []
    above_labels = []
    for i in range(len(leaf.elements)):
        v = self.compute_v(leaf.elements[i],scv) > 0
        if v:
            above.append(leaf.elements[i])
            above_labels.append(leaf.labels[i])
        else:
            below.append(leaf.elements[i])
            below_labels.append(leaf.labels[i])
    return np.array(below), np.array(above), np.array(below_labels), np.
    array(above_labels)
```

Listing 5.14 Split the data based on the hyperplane

We can compute the membership array as in Listing 5.15.

```
def compute_u(self, element, scv, feature):
    return (scv[feature] * element[feature] - self.compute_v(element, scv)
    ) / element[feature]
```

Listing 5.15 Calcualte membership matrix U

The code is an implementation of the U_j equation:

$$U_j = \frac{a_m x_{jm} - V_j}{x_{jm}}. \tag{5.7}$$

Perturb

In the method in Listing 5.16 we compute the V_j which gives us the knowledge if a given object is above or below the hyperplane. It can be formulated as $\sum_{i=1}^{d} a_i x_i + a_{d+1} > 0$, where $a_1, \ldots, a_{d+1}$ are coefficients. In our case, a_{d+1} is our label.

```
def compute_v(self,element, scv):
    return np.sum(np.multiply(element, scv[:-1])) + scv[-1]
```

Listing 5.16 Calcualte V

The perturbation function is the core part of the OC1 method. Calculate different Gini indices for different combinations of features. We get the combination with the best Gini index. We *fix* the previously calculated coefficients as in Listing 5.17.

```
def perturb(self, leaf, scv, feature, old_gini):
    u=[]
    for element in leaf.elements:
        u.append(self.compute_u(element, scv, feature))
    splits = np.sort(np.array(u))
    am = []
    for split in splits:
        new_scv = scv
        new_scv[feature] = split
        below, above, below_label, above_label = self.
     divide_data_hiperplane(leaf, scv)
        gini = 1 - self.calculate_gini(below_label) - self.calculate_gini
        (above_label)
        am.append([new_scv, gini])
    am = np.array(am)
    best_split_index = np.argmin(am[:,1])
    if am[best_split_index][1] < old_gini:
        return am[best_split_index][1], am[best_split_index][0]
    elif am[best_split_index][1] == old_gini:
        if random() < 0.3:
            return am[best_split_index][1], am[best_split_index][0]
    return old_gini, scv
```

Listing 5.17 Perturb phase implementation

Choose the one with the lowest gini index value

Compared to C4.5 and CART we have one more variable `R`, which is a parameter that is used to set the number of loops to randomly choose the feature to check if a feature change can give a better split. See `build_level` in Listing 5.18.

```
def build_level(self):
    leaf = self.find_current_level_data()
    if leaf == None:
        return
    splits = self.get_all_possible_splits_by_gini(leaf)
    split_coefficiency_vector = self.get_coefficiency(splits)
    below,above, below_label, above_label = self.divide_data_hiperplane
    (leaf,split_coefficiency_vector)
    gini = 1 - self.calculate_gini(below_label) - self.calculate_gini
    (above_label)
    for c in range(self.R):
        feature = randint(0,len(leaf.elements[0])-1)
        gini, split_coefficiency_vector=self.perturb(leaf,
     split_coefficiency_vector, feature, gini)
        below, above, below_label, above_label = self.
     divide_data_hiperplane(leaf,split_coefficiency_vector)
    left_leaf = Leaf(below, below_label)
    right_leaf = Leaf(above, above_label)
    leaf.set_completed()
    if len(np.unique(below_label)) == 1:
        left_leaf.set_completed()
    if len(np.unique(above_label)) == 1:
        right_leaf.set_completed()
    if self.utils.compare_two_leafs(leaf, left_leaf) or self.utils.
     compare_two_leafs(leaf,right_leaf):
            leaf.set_completed()
    else:
        leaf.set_R(right_leaf)
        leaf.set_L(left_leaf)
    self.build_level()
```

Listing 5.18 Building method that combines all presented methods

5.7 Quality Metrics and Tree Pruning

Decision trees can like other methods overfit. Pruning methods can be used to reduce or avoid overfitting. As explained earlier in this chapter, pruning is one of the tree operations. Considering pruning operation we should apply Occam's Razor approach. It says that everything should be made as simple as possible, but not simpler. In other words, if we have a tree that is shorter and gives the same error rate as a higher one, we should keep the shorter one as the more appropriate. We should keep the tree short to make the training and prediction faster. We can divide pruning methods into two groups:

- pre-pruning,
- post-pruning.

In [1] both groups are called respectively: direct and validation. Pre-pruning methods are done within the tree-building process. It means that the method knows more about the way how the tree is been built. This is why such methods are also called direct. Execution of such methods takes usually much more time than post-pruning methods. Post-pruning methods are the opposite of pre-pruning methods. The execution is done after the tree has been built. This kind of method validates if the tree has been built

good enough. If not it reduces the number of leafs to make the tree shorter and less complex.

There are many direct pruning methods [32–41]. We only describe two that are used in methods that we explained earlier in this chapter. Direct methods are used during the tree construction. It means we prune the tree during the training phase. It is more cost efficient than validation methods as we can change/prune the tree immediately.

Validation methods usually go through the whole tree and calculate some metrics at each node or level to prune the tree. Examples of validation methods are:

- reduced error pruning,
- error complexity pruning,
- minimum error pruning,
- cost-based pruning,
- and many more.

Minimum number of objects

In this method of pruning, the minimum number of objects is specified as a threshold value. Whenever a split is made that yields a child leaf that represents less than the minimum number from the data set, the parent node and the child node are compressed to a single node.

χ^2 pruning method

This approach to pruning is to apply a statistical test to the data to determine whether a split on some feature X_k is statistically significant, in terms of the effect of the split on the distribution of classes in the partition on the data induced by the split. We can perform chi-squared test as

$$\chi^2 = \sum \frac{(\text{observed value} - \text{expected value})^2}{\text{expected value}}. \tag{5.8}$$

We reject the split if the feature X_k is not related to the classification of the data given the features.

Reduced error pruning

The reduced error pruning method is the simplest and most understandable method in decision tree pruning. This method considers each of the decision nodes in the tree to be candidates for pruning, consisting of removing the subtree rooted at that node,

making it a leaf node. The available data are divided into three parts: the training examples, the validation examples used for pruning the tree, and a set of test examples used to provide an unbiased estimate of accuracy over future unseen examples. If the error rate of the new tree would be equal to or smaller than that of the original tree and that subtree contains no subtree with the same property, then the subtree is replaced by leaf node, which means pruning is done.

Error complexity pruning

In error complexity pruning is concerned with calculating the error cost of a node. Finds the error complexity at each node. The error cost of the node is calculated using the following equation:

$$R(t) = r(t) \times p(t), \tag{5.9}$$

where $r(t)$ is error rate of a node which is given as

$$r(t) = \frac{\text{number of misclassified}}{\text{numer of all objects in node}}, \tag{5.10}$$

and $p(t)$ is the probability of occurrence of a node which is given as

$$p(t) = \frac{\text{number of objects in node}}{\text{number of all objects}}. \tag{5.11}$$

Additionally, we need to calculate the error cost of subtree T of a given node:

$$R(T) = \sum R(i), \tag{5.12}$$

where i is the number of leaves of the node t. The error complexity is then calculated as follows:

$$a(t) = \frac{R(t) - R(T)_t}{\text{number of leaves} - 1} \tag{5.13}$$

The method consists of the following steps:

1. a is computed for each node,
2. the minimum a node is pruned,
3. the above is repeated and a forest of pruned tree is formed,
4. the tree with the best accuracy is selected.

For Further Reading

1. Azad M, Chikalov I, Hussain S, Moshkov M, Zielosko B (2022) Decision trees with hypotheses. Springer
2. Grabczewski K (2014) Meta-learning in decision tree induction. Springer
3. Breiman L, Friedman J, Olshen RA, Stone CJ (2017) Classification and regression trees. Chapman and Hall/CRC
4. Smith C, Koning M (2017) Decision trees and random forests: a visual introduction for beginners: a simple guide to machine learning with decision trees. Blue Windmill Media

References

1. Grabczewski K (2014) Meta-learning in decision tree induction. Springer
2. Larose DT, Larose CD (2015) Data mining and predictive analytics, 2nd edn. Wiley
3. Müller W, Wysotzki F (1994) Automatic construction of decision trees for classification. Ann Oper Res 52:231–247
4. Müller W, Wysotzki F (1997) The decision-tree algorithm CAL5 based on a statistical approach to its splitting algorithm. In: Machine learning and statistics: the interface, pp 45–65
5. Hothorn T, Hornik K, Zeileis A (2004) Unbiased recursive partitioning: a conditional inference framework. Research report series 8, Department of Statistics and Mathematics, Institut für Statistik und Mathematik, WU Vienna University of Economics and Business, Vienna
6. Zeileis A, Hothorn T, Hornik K (2008) Model-based recursive partitioning. J Comput Graph Stat 17:492–514
7. Grabczewski K, Duch W (1999) A general purpose separability criterion for classification systems. In: Proceedings of the 4th conference on neural networks and their applications, Zakopane, Poland, pp 203–208
8. Grabczewski K, Duch W (2000) The separability of split value criterion. In: Proceedings of the 5th conference on neural networks and their applications, Zakopane, Poland, pp 201–208
9. Ferri C, Flach P, Hernández-Orallo J (2002) Learning decision trees using the area under the roc curve. In: ICML '02: proceedings of the nineteenth international conference on machine learning, pp 139–146
10. Ferri C, Flach P, Hernández-Orallo J (2003) Improving the AUC of probabilistic estimation trees. Lect Notes Comput Sci 2837:121–132
11. Heath D, Kasif S, Salzberg S (1993) Induction of oblique decision trees. J Artif Intell Res 2:1–32
12. Shafer JC, Agrawal R, Mehta M (1996) Sprint: a scalable parallel classifier for data mining. In: Proceedings of the 22nd international conference on very large data bases, pp 544–555
13. Alsabti K, Ranka S, Singh V (1998) Clouds: a decision tree classifier for large datasets. In: Proceedings of the 22nd international conference on very large data bases
14. Wand H, Zaniolo C (2000) CMP: a fast decision tree classifier using multivariate predictions. In: Proceedings of the 16th international conference on data engineering, pp 449–460
15. Lee JY, Olafsson S (2006) Multi-attribute decision trees and decision rules. In: Data mining and knowledge discovery approaches based on rule induction techniques, massive computing, vol 6, pp 327–358
16. Amasyali MF, Ersoy OK (2008) Cline: a new decision-tree family. IEEE Trans Neural Netw 19:356–363

17. Utgoff PE, Brodley CE (1991) Linear machine decision trees. Technical report UM-CS-1991-010, Department of Computer Science, University of Massachusetts
18. Bramer M (2007) Avoiding overfitting of decision trees. In: Principles of data mining, pp 119–134
19. Attneave F (1959) Applications of information theory to psychology. Holt, Rinehart and Winston
20. Dietterich TG, Kearns M, Mansour Y (1996) Applying the weak learning framework to understand and improve c4.5. In: Proceedings of the thirteenth international conference on machine learning, pp 96–104
21. Ferri C, Flach P, Hernandez-Orallo J (2002) Learning decision trees using the area under the roc curve. In: Proceedings of the 19th international conference on machine learning, pp 139–146
22. Friedman JH (1977) A recursive partitioning decision rule for nonparametric classifiers. IEEE Trans Comput 26:404–408
23. Lopez de Mantras R (1991) A distance-based attribute selection measure for decision tree induction. Mach Learn 6:81–92
24. Fayyad U, Irani KB (1992) The attribute selection problem in decision tree generation. In: Proceedings of tenth national conference on artificial intelligence, pp 104–110
25. Li X, Dubes RC (1986) Tree classifier design with a permutation statistic. Pattern Recognit 19:229–235
26. Martin JK (1997) An exact probability metric for decision tree splitting and stopping. Mach Learn 28:257–291
27. Taylor PC, Silverman BW (1993) Block diagrams and splitting criteria for classification trees. Stat Comput 3:147–161
28. Gini C (1912) Variabilità e mutabilità
29. Kullback S, Leibler RA (1951) On information and sufficiency. Ann Math Stat 22:79–86
30. Breiman L, Friedman JH, Olsen RA, Stone CJ (1984) Classification and regression trees. Chapman and Hall/CRC
31. Quinlan JR (1979) Discovering rules by induction from large collections of examples. Expert Syst Micro-Electron Age 168–201
32. Quinlan JR (1987) Simplifying decision trees. Int J Man-Mach Stud 27:221–234
33. Esposito F, Malerba D, Semeraro G (1997) A comparative analysis of methods for pruning decision trees. IEEE Trans Pattern Anal Mach Intell 19:476–491
34. Niblett T, Bratko I (1986) Learning decision rules in noisy domains. In: Proceedings of expert systems '86, the 6th annual technical conference on research and development in expert systems III, pp 25–34
35. Mingers J (1989) An empirical comparison of pruning methods for decision tree induction. Mach Learn 4:227–243
36. Cestnik B, Bratko I (1991) On estimating probabilities in tree pruning. Lect Notes Comput Sci 482:138–150
37. Quinlan JR, Rivest RL (1989) Inferring decision trees using the minimum description length principle. Inf Comput 80:227–248
38. Wallace C, Patrick J (1993) Coding decision trees. Mach Learn 11:7–22
39. Mehta M, Rissanen J, Agraval R (1995) MDL-based decision tree pruning. In: Proceedings of the first international conference on knowledge discovery and data mining, pp 216–221
40. Oliveira A, Sangiovanni-Vincentelli A, Shavlik J (1996) Using the minimum description length principle to infer reduced ordered decision graphs. Mach Learn 23–50
41. Kononenko I (1998) The minimum description length based decision tree pruning. Lect Notes Comput Sci 1531:228–237

Chapter 6
Support Vector Machine

Support Vector Machine (SVM) is a classifier that was fully introduced by Vapnik in [1, 2], however, it was first mentioned in [3]. The standard SVM is a binary linear classifier, i.e., it can separate the samples from the two classes only and only when they are linearly separable. SVM tries to find an optimal separating hyperplane. It is a hyperplane that distinguishes elements of the two different classes in an efficient way. What exactly we mean by optimal and efficient is described later in this chapter. The equation describing the hyperplane is calculated using the samples from the training data set. This means that some noisy samples can affect the result of the classification. To avoid it, the so-called soft margin approach was proposed by Cortes and Vapnik in [4]. This approach is presented in one of the sections in the chapter. As we will see from the next sections the calculation of the separating hyperplane in the SVM requires solving the convex optimization problem with some constraints. The solution we use in SVM is based on Lagrange multipliers. A step-by-step explanation is provided in Sect. 6.1.

Several different types of SVM methods exist. In this book, we focus only on the most popular ones. We explain the common C-SVM and ν-SVM in Sect. 6.2 and in Sect. 6.3.

The real-world problems are usually not linearly separable so this assumption of the SVM limits its usage. However, the introduction of kernel functions allows us to move a problem to a higher dimensional feature space in which the problem can become linearly separable. It is even possible to move the problem to infinite-dimensional space. Obviously, we should use non-linear mapping which means that the problem has a non-linear decision boundary in the original space. It is described in the Sect. 6.4.

The last point that is explained in this chapter is related to some extensions of the SVM classifier. We present a special type of SVM which can deal with one-class problems as well. This idea is described in the Sect. 6.5. We also show how to use SVM for one and multiclass problems. We should use some problem decomposition

K. Przystalski et al., *Pattern Recognition Primer*,
https://doi.org/10.1007/978-3-031-91816-2_6

methods to apply the SVM to classify more than two classes. Two simple methods *one-vs-one* and *one-vs-rest* are described in this chapter.

General overview

SVM is a binary classifier, so we consider two classes and to simplify the next examples, we use a two-dimensional feature space. For now, let us assume that the problem is linearly separable as shown in Fig. 6.1. The line that distinguishes objects of both classes can be described by the following equation:

$$w_0 + w_1 x_1 + w_2 x_2 = 0. \tag{6.1}$$

It means that if we find such a line then for all objects in the feature space representing the class marked red on Fig. 6.1 we have

$$w_0 + w_1 x_1^1 + w_2 x_2^1 > 0, \tag{6.2}$$

and

$$w_0 + w_1 x_1^{-1} + w_2 x_2^{-1} < 0, \tag{6.3}$$

where all objects representing the blue class we have two features $x^{-1} = (x_1^{-1}, x_2^{-1})$, and, respectively, for the red objects.

The questions that we can set here are which line of Fig. 6.1 is the best possible? What does the best possible actually mean? We can draw different lines as shown in Fig. 6.2. Some are better than others. For each of the lines, Eqs. 6.2 and 6.3 hold. We can assume that the best line is the line where we can secure the best accuracy of the separation of the unseen samples. The line that distinguishes between objects of two different classes is called a hyperplane. It is a surface that separates between objects in an n-dimensional feature space. In a two-dimensional space, it would be a

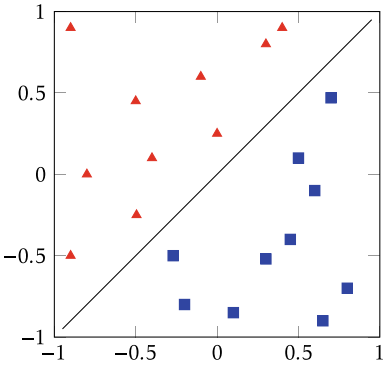


Fig. 6.1 Linear separable classification problem, where each object is marked with red or blue depending on the assigned class

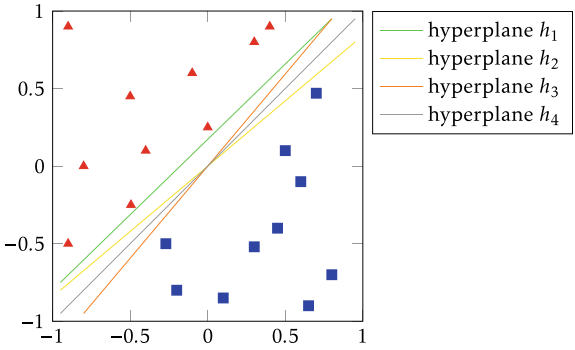


Fig. 6.2 A few possible separation options

line. We will come back to the definition of the hyperplane later. The hyperplane h_1 does not seem to be a good choice since it is very close to red objects. The second line h_2 is better; however, it is very close to red objects on one end and close to blue on the other end. The hyperplane h_3 is similar to h_2. Using both hyperplanes would classify more as a blue class on one end and more as a red class on the other end. Our intuition tells us that the fourth hyperplane h_4, which maximizes the distance between the closest samples of both classes from the separation line, will be the best choice. We can prove on the basis of the Probability Approximately Correct (PAC) learning theory [5] that our intuition is good. The last approach which maximizes the distance between the separating line and the closest samples from both classes, we will call the maximum-margin approach or less formally the widest street approach because it leads the widest possible street between the points representing the samples.

Maximum margin approach

Before we start to analyze the problem of how to find the separating line with the maximum margin, let us try to generalize our task. Instead of considering the two-dimensional feature space, let us assume that we have p features. For now, we considered a two-dimensional feature space, but as we know from the previous chapters, a feature space usually has more than two dimensions. Our hyperplane will become a subspace of the dimension $p - 1$ in the feature space which we call a hyperspace. The equation describing the line changes from 6.1 and our hyperplane is defined as

$$w_0 + w_1 x_1 + \ldots, + w_p x_p = 0. \tag{6.4}$$

Now, this hyperspace divides the feature space into two sub-spaces:

$$w_0 + w_1 x_1 + \ldots, + w_p x_p > 0 \tag{6.5}$$

we assign the label $y_i = 1$ and when:

$$w_0 + w_1 x_1 + \ldots, + w_p x_p < 0 \tag{6.6}$$

we assign the label $y_i = -1$. In that way, we separate a p-dimensional feature space into the two sub-spaces, classifying the points x_i as two classes $\{-1, 1\}$. This is valid for a linearly separable case.

Before we look at the problem of finding the hyperplane with the maximum margin or, in other words, the optimal hyperplane for a given set of feature vectors, let us look for a more convenient version of Eq. 6.4. Assuming that $w = (w_1, \ldots, w_p)$, $x = (x_1, \ldots, x_p)$ and $b = w_0$ we get the following:

$$w^T x + b = 0. \tag{6.7}$$

This is a vectorized version of Eq. 6.4. Now, the distance d of the defined hyperspace from the origin can be given as

$$d = \frac{b}{||w||}, \tag{6.8}$$

where $||w||$ means the distance between the hyperplane and the objects. We need to find those w and b that divide the set of training objects $x_1, \ldots, x_n \in R^p$ by the maximum margin. We assume for now that the problem is linearly separable, so $w^T x_i + b > 0$ or $w^T x_i + b < 0$ for each feature vector from the training set. We also assume that we pick a hyperplane that is equally distant from the closest feature vectors that represent the samples of both classes. It means that $\exists \epsilon > 0$ such that:

$$w^T x_i + b \geq \epsilon \tag{6.9}$$

or

$$w^T x_i + b \leq -\epsilon. \tag{6.10}$$

In this case, we can demand to choose w such that the constant on the right side of Eqs. 6.9 and 6.10 will be equal to 1. It is enough to multiply both sides of the inequalities by $\frac{1}{\epsilon}$. If we take this assumption, we have the following:

$$w^T x_i + b \geq 1 \tag{6.11}$$

and for the feature vectors that lie above the hyperplane, we assign the label $y_i = 1$. For feature vectors lying under the hyperplane:

$$w^T x_i + b \leq -1 \tag{6.12}$$

we assign the label $y_i = -1$. Multiplying both sides of the above inequalities by y_i we get

$$y_i (w^T x_i + b) \geq 1. \tag{6.13}$$

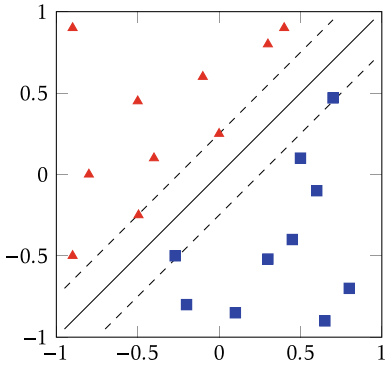


Fig. 6.3 SVM hyperplane with margins

Based on that for all closest positive and negative feature vectors the inequalities 6.11 and 6.12 become

$$w^T x_i^1 + b = 1 \tag{6.14}$$

for positive feature vectors and

$$w^T x_i^{-1} + b = -1 \tag{6.15}$$

for negative feature vectors. The hyperplane margins are defined by the following equations:

$$w^T x_i^1 + b - 1 = 0 \tag{6.16}$$

and

$$w^T x_i^{-1} + b + 1 = 0. \tag{6.17}$$

We can draw it as shown in Fig. 6.3 where the dashed lines are the margin hyperplanes given with Eqs. 6.16 and 6.17. The goal of the SVM method is to find a hyperplane that divides the feature space with the biggest margin between support vectors. The support vectors are the elements in the feature space that are closest to the hyperplane. The black straight line is the hyperplane $g(x)$. The hyperplane presented as the straight black line is defined as follows:

$$\hat{g}(x) = w^T x + w_0 = 0. \tag{6.18}$$

The two dashed lines are the margins. The margines are built using the support vectors. Here is where the name of the method came from. The distance d between them can be calculated by the difference of the distances d_1 and d_2 (dashed lines) where d_1 is the distance of the positive margin hyperplane from the origin (solid line) and d_2 is the distance of the negative margin hyperplane from the origin. Remember that these hyperplanes are parallel, so using Eq. 6.8 we get

$$d = |d_1 - d_2| = |\frac{b-1}{||w||} - \frac{b+1}{||w||}| = \frac{2}{||w||}. \tag{6.19}$$

It means that the maximum margin between the feature vectors we get when we maximize the expression $\frac{2}{||w||}$ or minimize the expression $\frac{||w||}{2}$. However, sometimes it is more convenient to minimize the following:

$$\frac{1}{2}||w||^2. \tag{6.20}$$

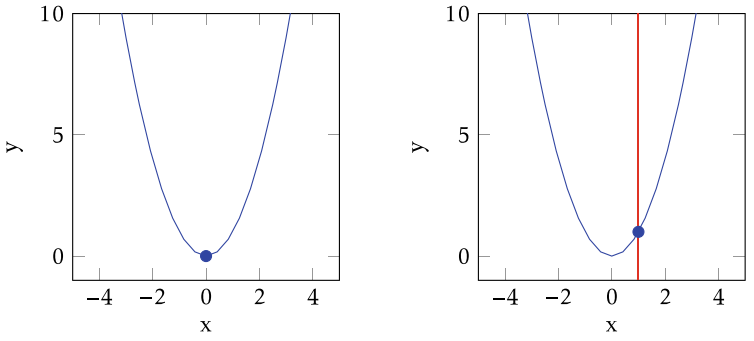
It is a quadratic optimization problem. We cannot forget about the constraints:

$$y_i(w^T x_i + b) - 1 \geq 0, \ i = 1, \ldots, n. \tag{6.21}$$

At this point, we could stop our considerations and we could calculate the values of w and b which minimize the formula 6.19. Next, we can easily classify a new object by calculating the value of $w^T x + b$ and if it is greater than zero, we classify it as class 1 and as class -1 in the opposite case.

6.1 Lagrangian Multipliers

In the previous chapter, we conclude that we have to solve a quadratic optimization problem with constraints to find an optimal separating hyperplane. What does it mean? Let us look at Fig. 6.4. On the left, we have to find the minimum of the function x^2 without any constraints. The solution is $x = 0$, on the right we have the same minimization problem, but with constraints $x \geq 1$, now the answer changes, and the minimum of the function is $x = 1$. This problem is very simple, as it contains



(a) A function without constraints

(b) A function with a constraint $x = 1$

Fig. 6.4 Example of the minimization problem of x^2 function

only a function with one variable. A function with multiple variables is much more difficult to solve. Let us assume that we have to minimize the function $f(x)$ with the constraint $g(x) = 0$, where x might be a vector of variables $x = (x_1, \ldots, x_n)$. We can notice that the minimum of the $f(x)$ is found when the gradients of these two functions are parallel, i.e.,

$$\nabla f(x) = \alpha \nabla g(x), \tag{6.22}$$

where α is the scaling factor, we call it the Lagrange multiplier. To find the minimum of f under the constraint g, we just need to solve the following:

$$\nabla f(x) - \alpha \nabla g(x) = 0. \tag{6.23}$$

To solve that equation, we can define a function $L(x, \alpha) = f(x) - \alpha g(x)$, then its gradient is $\nabla L(x, \alpha) = \nabla f(x) - \alpha \nabla g(x)$. Solving $\nabla L(x, \alpha) = 0$ allows us to find the minimum. The function $L(x, \alpha)$ we call Lagrangian.

Let us take an example to understand how it is used to solve the problem defined in Eqs. 6.23. Assume that we have to find the minimum function $f(x, y) = x^2 + y^2$ under the constraint $g(x, y) = x + y - 1 = 0$. In our case, the Lagrangian is defined as follows:

$$L(x, y, \alpha) = x^2 + y^2 - \alpha(x + y - 1).$$

Now, we have to calculate when the gradient of this function equals zero, which means solving the following system of equations:

$$\frac{\partial}{\partial x} L(x, y, \alpha) = 2x - \alpha = 0$$

$$\frac{\partial}{\partial y} L(x, y, \alpha) = 2y - \alpha = 0$$

$$\frac{\partial}{\partial \alpha} L(x, y, \alpha) = -x - y + 1 = 0$$

We calculate the derivative of each variable in each equation separately. Finally, we get the answers as: $x = \frac{1}{2}$, $y = \frac{1}{2}$ and $\alpha = 1$. This means that the function $f(x, y) = x^2 + y^2$ has the minimum in $f(\frac{1}{2}, \frac{1}{2}) = \frac{1}{4} + \frac{1}{4} = \frac{1}{2}$.

Lagrangian multipliers for multiple constraints

Lagrange multipliers also work with multiple constraints. We are just adding another boundary to the problem. When we deal with multiple constraints, then our Lagrangian becomes

$$L(x, \alpha) = f(x) - \sum_i \alpha_i g_i(x), \tag{6.24}$$

where $g_i(x) = 0$ for $i = 1, \ldots, n$ are the constraints. Notice that each constraint has its own Lagrange multiplier. The Lagrangian is equal to 0:

$$\nabla L(x, \alpha) = 0. \tag{6.25}$$

Solving this case is not much different compared to a single constraint case. However, when we are looking for optimal hyperplanes, then our constraints are inequalities. The constraints are handled by the Lagrange multipliers, but the following equations should be met when dealing with the inequality constraints:

$$\begin{aligned} g(x) \geq 0, \quad \text{then} \quad \alpha \geq 0 \\ g(x) \leq 0, \quad \text{then} \quad \alpha \leq 0 \end{aligned}$$

Example 1 *(Lagrangian multipliers with two constraints)* Let us take an example of a function with two constraints. The function is defined as follows:

$$f(x, y) = 2x^2 - 3y^2$$

and the constraints are defined as

$$g_1(x, y) = x^2 - 4 \geq 0,$$

and

$$g_2(x, y) = y + 1 \geq 0.$$

Based on the previous example, we can set our Langrangian as

$$L(x, y, \alpha_1, \alpha_2) = 2x^2 - 3y^2 - \alpha_1(x^2 - 4) - \alpha_2(y - 1).$$

All derivatives should be zero and we get a system of equations as

$$\frac{\partial}{\partial x} L(x, y, \alpha) = 4x - 2x\alpha_1 = 0,$$

$$\frac{\partial}{\partial y} L(x, y, \alpha) = -6y - 2y\alpha_2 = 0,$$

$$\frac{\partial}{\partial \alpha_1} L(x, y, \alpha) = x^2 - 4 = 0,$$

$$\frac{\partial}{\partial \alpha_2} L(x, y, \alpha) = y + 1 = 0.$$

We also have additional constraints:

$$\alpha_1 \geq 0,$$
$$\alpha_2 \geq 0.$$

The above equations give as the values of y, x, α_1 and α_2:

$$y = -1,$$
$$x = 2,$$
$$\alpha_1 = 2,$$
$$\alpha_2 = 3.$$

Finally, we find that the minimum of the function $f(x, y)$ in $(-1, 2)$ is equal to:

$$f(x, y) = 2 \cdot 2^2 - 3 \cdot (-1) = 11.$$

To summarize this section, we have learned how to use Lagrangian multipliers to solve a function with more than one constraint.

Applying multipliers to SVM

Let us go back to our problem of maximizing the margin between the hyperplanes. We concluded that it is enough to find the minimum for the function:

$$f(w) = \frac{1}{2}||w||^2, \tag{6.26}$$

with the constraints:

$$g(w, b) = y_i(w^T x_i + b) - 1 \geq 0, \ i = 1, \ldots, n. \tag{6.27}$$

As shown in the previous section, it can be solved by the Lagrangian multiplier method. This method allows us to formulate the so-called dual problem in which we can get rid of all the constraints. The dual problem is just a different way of solving the primal problem. Not to go much into the details, in case of Lagrangian multipliers to move to a dual problem, the Karush-Kuhn-Tucker conditions [6] need to be met.

In other words, a solution of the dual problem is also the solution of our primal problem. In order to get the dual problem we introduce $\alpha_i > 0$ Lagrange multipliers and add our constraints into the formula. So, now we get

$$L(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_i \alpha_i (y_i (w^T x_i + b) - 1). \tag{6.28}$$

We can rewrite the above equation as follows:

$$L(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_i \alpha_i y_i (w^T x_i + b) - \sum_i \alpha_i. \tag{6.29}$$

We would like to find w and b that minimizes, and the α_i which maximizes our equation. We can do this by differentiating L(w;a) with respect to w and b and setting the derivatives to zero, that is

$$\frac{\delta L(w, b, \alpha)}{\delta w} = w - \sum \alpha_i y_i x_i = 0, \tag{6.30}$$

and

$$\frac{\delta L(w, b, \alpha)}{\delta b} = - \sum \alpha_i y_i = 0, \tag{6.31}$$

which means

$$w = \sum \alpha_i y_i x_i, \tag{6.32}$$

and

$$\sum \alpha_i y_i = 0. \tag{6.33}$$

We should also have

$$\sum \alpha_i ((w_0 x_i - b_0) y_i - 1) = 0. \tag{6.34}$$

We know that $\sum \alpha_i y_i = 0$. If we replace w and b in 6.29 by 6.32 and 6.33 then we get

$$L = \sum \alpha_i - \frac{1}{2} (\sum \alpha_i y_i x_i)(\sum \alpha_i y_i x_i), \tag{6.35}$$

which is equivalent to:

$$L(\alpha) = \sum \alpha_i - \frac{1}{2} \sum \sum \alpha_i \alpha_j y_i y_j x_i^T x_j. \tag{6.36}$$

So finally our task is to find α for which $L(\alpha)$ is maximal, of course under the following constraints:

$$\alpha_i \geq 0 \quad i = 1, 2, \ldots, n \quad and \quad \sum \alpha_i y_i = 0. \tag{6.37}$$

Assume that the vector $\alpha^0 = (\alpha_1^0, \alpha_2^0, \ldots, \alpha_n^0)$ is the solution of our optimization problem with our constraints, then hyperplane is defined by $w^0 x^T + b$ where

$$w^0 = \sum y_i \alpha_i^0 x_i, \tag{6.38}$$

summed over all support vectors, and

$$b^0 = \frac{1}{2}(w^0 x_i + w^0 x_j), \tag{6.39}$$

where $i \neq j$ are two indices of the support vectors. Instead of using an arbitrary Support Vector x_s, it is better to take the average over all of the Support Vectors:

$$b^0 = \frac{1}{N_s} \sum (y_s - \sum \alpha_i y_i x_i^T x_s). \tag{6.40}$$

It is time when we can formulate some conclusions. First, we see that if the vector is not a support vector, i.e., it is not lying on the margin then the corresponding Lagrange multiplier α_i must be zero. Second, we use only the support vectors to find the optimal solution for our problem, we need only the dot product of the support vectors.

Finally, we are able to rewrite our decision function using the Lagrange multipliers and the support vectors as

$$f(x) = sign(\sum y_i \alpha_i^0 (x_i^T x) + b^0). \tag{6.41}$$

The decision function is a sign function where the major part of the decision is made based on the weights.

6.2 C-SVM

The presented solution is very intuitive and it performs well, but only if a separating hyperplane exists, i.e., the problem is linearly separable. However, in many cases, there is no separate hyperplane and, therefore, the solution of the optimization problem has no solution with margin $d(h_1, h_2) > 0$. However, in this section, we describe the concept of a semi-separating hyperplane which almost separates the classes (with some errors), using the so-called soft margins.

Sometimes, the strict approach to maximization of the margin is not optimal. We see it in Fig. 6.5. The separation hyperplane exists, we can use our algorithm to maximize the margin, and we find the optimal separating hyperplane h_1 which separates all samples from both classes. However, if we look closer at the solution, we see that such a classifier is probably not a good solution. It will probably not

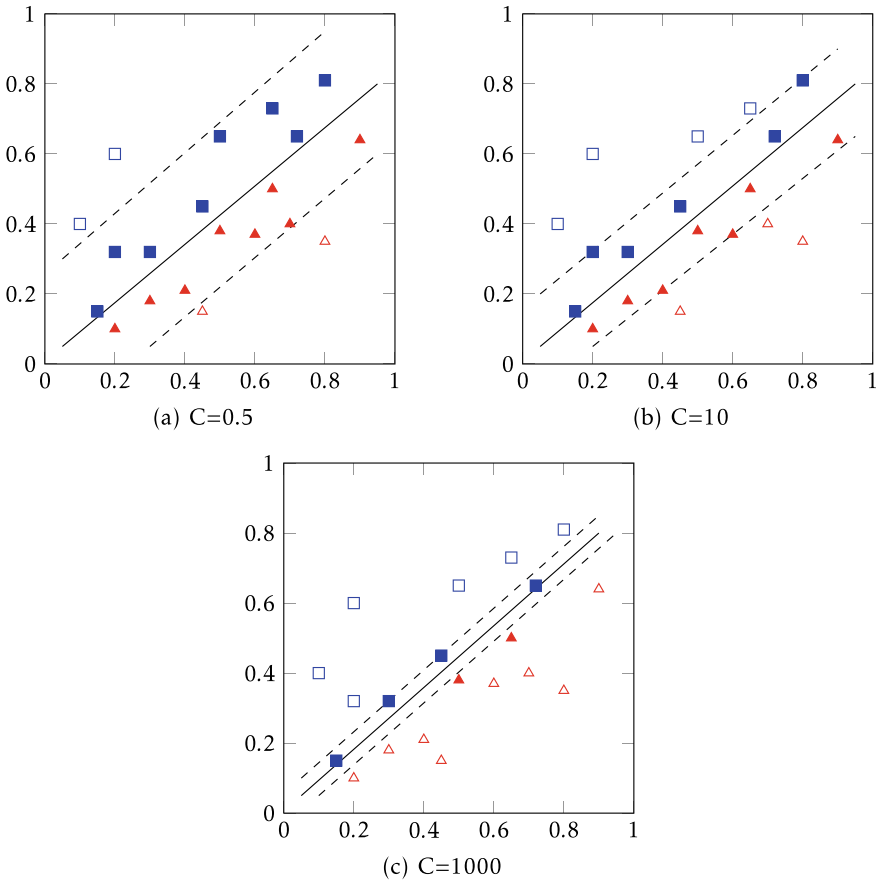


Fig. 6.5 A hyperplane with small (**a**), big (**b**), and intermediate value of C

generalize well. We would prefer the hyperplane h_2, which misclassifies one sample but allows us to find a hyperplane with a much larger margin.

This example is very important to show that the pure maximum-margin classifier is very sensitive to single-outlying observations. We already see that every change of the support vector impacts the separating hyperplane but now see that if we add even an outlying sample, then our solution will change drastically and it can even become unseparable. The problem can be solved by allowing one to misclassify some training samples in order to achieve better accuracy in classifying the test samples.

This approach is a soft margin approach. Rather than seeking the largest possible margin so that every observation is not only on the correct side of the hyperplane but also on the correct side of the margin, we instead allow some observations to be on the incorrect side of the margin, or even the incorrect side of the hyperplane. We call the margin soft because it can be violated by some of the training samples. An

example is shown in Fig. 6.5a, b. Most of the observations are on the correct side of the margin. However, a small subset of the observations are on the wrong side of the margin.

A sample can be not only on the wrong side of the margin, but also on the wrong side of the hyperplane. In fact, when there is no separate hyperplane, such a situation is inevitable. Observations on the wrong side of the hyperplane correspond to training observations that are misclassified by the support vector classifier. The right-hand panel of Figure XX illustrates such a scenario.

How to describe this mathematically? We can modify the conditions 6.11 and 6.12 which now became

$$w^T x_i + b \leq -1 + \xi_i \tag{6.42}$$

and

$$y_i(w^T x_i + b) \geq 1 - \xi_i, \tag{6.43}$$

where

$$\xi_i \geq 0, \ \ i = 1, 2, \ldots, n. \tag{6.44}$$

The ξ_i are slack variables that allow some individual samples to be on the wrong side of the margin or the hyperplane. Well, but what are these slack variables? Generally, they tell us where the i-th sample is in the feature space relative to the hyperplane we are looking for. Also, it tells us where this sample is relative to the margin. There are four possible cases:

- $\xi_i = 0$ the sample is on the correct side of the hyperplane and on the correct side of the margin,
- $\xi_i > 0$ and $\xi_i < 1$ the sample is on the correct side of the hyperplane, but it lies inside the margin,
- $\xi_i = 1$ the sample lies just on the separation hyperplane,
- $\xi_i > 1$ the sample lies on the wrong side of the hyperplane.

The second case does not cause a classification error when the third and fourth lead to misclassification. We can differ the approach to them, but it is more practical not to. Now, let us introduce an additional parameter C, which will be the sum of all ξ_i's, so it will determine the sum of violations in the margin and the hyperplane. It means that we will tolerate the total weight of the violations (but no more than C), to find a better hyperplane, i.e., with a better margin.

We can think of C as a trade-off between the width of the margin and the cost of all the violations caused by the samples in n. We can demand $C = 0$, which means that we do not allow any sample from the training data set to be misclassified. However, we risk finding the solution with a very small margin, or even we will not be able to separate our samples. If we allow $C > 0$, then we allow some samples to lie in the margin or even be misclassified, resulting in a wider margin or even making our problem separable. The higher the value of C, we get a more general solution but at the cost of increasing the error rate. The choice of the value of C is not a trivial problem; we will return to it later in this chapter.

We have our modified conditions, and now the function which we should minimize becomes:

$$\frac{1}{2}||x||^2 + C\sum \xi_i \tag{6.45}$$

with constraints

$$\xi_i \geq 0 \quad and \quad \sum \xi_i < C, \; C > 0 \tag{6.46}$$

As mentioned we can use Lagrangian multipliers and introduce the dual problem which becomes

$$L(\alpha) = \sum \alpha_i - \frac{1}{2}\sum \alpha_i \alpha_j y_i y_j (x_i^T + x_j) \tag{6.47}$$

with constraints:

$$C \geq \alpha_i \geq 0, \; i = 1, \ldots, n \quad and \quad \sum \alpha_i y_i = 0 \tag{6.48}$$

We can also notice something very interesting. Using KKT conditions, we get the following:

$$\alpha_i = 0 \Rightarrow y_i(w^T x_i + b) \geq 1 \tag{6.49}$$

$$\alpha_i = C \Rightarrow y_i(w^T x_i + b) \leq 1 \tag{6.50}$$

$$0 < \alpha_i < C \Rightarrow y_i(w^T x_i + b) = 1 \tag{6.51}$$

Our new optimization problem has a very interesting property: it turns out that only those samples that either lie on the margin or violate the margin will affect the separating hyperplane. This means that samples that lie on the correct side of the margin do not affect our classifier. So, we change the classifier only if we add the sample that violates the margin! It also means that not only the samples that lie directly on the margin become the support vectors; now even the samples on the wrong side of the margin for their class become support vectors.

The fact that only support vectors affect the classifier is quite understandable, and we see that C controls the trade-off between the margin and the bias caused by the misclassified samples. When the C parameter is large, the margin is wide, many samples violate the margin, and so there are many support vectors. In this case, many samples are used to calculate the separating hyperplane.

```
def train(train_data_set, train_labels, kernel_type='linear', C=10,
    threshold=1e-5):
    kernel = build_kernel(train_data_set, kernel_type=kernel_type)

    P = train_labels * train_labels.transpose() * kernel
    q = -np.ones((objects_count, 1))
    G = np.concatenate((np.eye(objects_count), -np.eye(objects_count)))
    h = np.concatenate((C * np.ones((objects_count, 1)), np.zeros
    ((objects_count, 1))))

    A = train_labels.reshape(1, objects_count)
    A = A.astype(float)
```

```
12      b = 0.0
13
14      sol = cvxopt.solvers.qp(cvxopt.matrix(P), cvxopt.matrix(q), cvxopt.
        matrix(G), cvxopt.matrix(h), cvxopt.matrix(A), cvxopt.matrix(b))
15
16      lambdas = np.array(sol['x'])
17
18      support_vectors_id = np.where(lambdas > threshold)[0]
19      vector_number = len(support_vectors_id)
20      support_vectors = train_data_set[support_vectors_id, :]
21
22      lambdas = lambdas[support_vectors_id]
23      targets = train_labels[support_vectors_id]
24
25      b = np.sum(targets)
26      for n in range(vector_number):
27          b -= np.sum(lambdas * targets * np.reshape(kernel
28          [support_vectors_id[n], support_vectors_id], (vector_number, 1)))
29      b /= len(lambdas)
30
31      return lambdas, support_vectors, support_vectors_id, b, targets,
        vector_number
```

Listing 6.1 C-SVM training method

In Listing 6.1 we have combined all the above and implemented the training part. We use the cvxopt library that is a linear programming library in Python. The `n` is the sample number, `b` is the bias as a 1×1 matrix, `A` is the y vector, `P` is $H = X'X'^T$, `G` is a diagonal matrix of -1s of size $m \times m$, `h` is a vector of size $1 \times m$, and q is a vector of size $1 \times n$ of -1. The solver of the equations can be invoked as line 16. The x values are saved in the `lambdas`.

```
1  def classify_linear(test_data_set, train_data_set, lambdas, targets, b,
       vector_number, support_vectors, support_vectors_id):
2      kernel = build_kernel(train_data_set)
3      y = np.zeros((np.shape(test_data_set)[0], 1))
4      for j in range(np.shape(test_data_set)[0]):
5          for i in range(vector_number):
6              y[j] += lambdas[i] * targets[i] * kernel[j, i]
7          y[j] += b
8      return np.sign(y)
```

Listing 6.2 Linear classification

The SVM is by default a linear classifier. In Listing 6.2 we use the linear kernel that is defined as

$$K = XX^T. \tag{6.52}$$

The linear kernel in many cases is not the best-performing kernel. To show it we prepared an example on a classical data set below.

Example 2 *(Iris classified using linear kernel C-SVM)* This example is only used to show how to use C-SVM using one of the simplest kernels. In Listing 6.3 we show how to prepare the Iris data and choose only the objects of two classes. In lines 9–10 we drop the object of class 2. What is more, the labels are changed from 0 and 1 to -1 and 1, this is a requirement of the cvxopt library.

```
1  from sklearn.datasets import load_iris
2  import numpy as np
3  from sklearn.model_selection import train_test_split
```

```
4
5  iris = load_iris()
6  data_set = iris.data
7  labels = iris.target
8
9  data_set = data_set[labels!=2]
10 labels = labels[labels!=2]
11
12 train_data_set, test_data_set, train_labels, test_labels =
       train_test_split(data_set, labels, test_size=0.2, random_state=15)
13
14 train_labels[train_labels<1] = -1
15 test_labels[test_labels<1] = -1
16
17 objects_count = len(train_labels)
18
19 lambdas, support_vectors, support_vectors_id, b, targets, vector_number =
       train(train_data_set, train_labels, kernel_type='linear')
20 predicted = classify_linear(test_data_set, train_data_set, lambdas,
       targets, b, vector_number, support_vectors, support_vectors_id)
21 predicted = list(predicted.astype(int))
```

Listing 6.3 Invoke the main SVM prediction method

The predicted values are then converted into an integer with a value of -1 or 1.

6.3 ν-SVM

The soft margin classifier is a very flexible approach, allowing us to soften the criteria of separating hyperplanes. We see that the higher the value of C, the more support vectors we have. However, we have no direct influence on this number. So we can propose another realization of the soft margins called ν-parameterization [7]. The parameter C is replaced by a parameter $\nu \in [0, 1]$, which is the lower and upper bound of the number of examples that are support vectors and are on the wrong side of the hyperplane.

The primal problem in this approach will be formulated as follows:

$$\frac{1}{2}||w||^2 - \nu\rho + \frac{1}{2}\sum \xi_i \tag{6.53}$$

subject to:

$$y_i(w^T x_i) + b \geq \rho - \xi_i, \quad i = 1, \ldots, n, \quad and \quad \xi_i \geq 0, \quad \rho \geq 0 \tag{6.54}$$

Now, we have no constant C appearing in the formula. It has been replaced by a parameter ν and an additional variable ρ to optimize. Note that for $\xi_i = 0$ our constraint states that the two classes are separated by a margin equal to $\frac{2\rho}{||w||}$.

To explain what is the parameter ν, let us introduce the term margin error R. We denote training points by $\xi_i > 0$ as the points that are errors or are within the margin. Formally, the fraction of margin errors is the following:

$$R_\rho(w,b) = \frac{1}{n}|\{i|y_i(w^T x_i + b) < \rho\}|. \tag{6.55}$$

Now, let us assume that we run ν-SVM with kernel function k; on some data and we get some $\rho > 0$, then:

- ν is an upper bound of the fraction of margin errors, and hence also on the fraction of training errors,
- ν is a lower bound on the fraction of support vectors.

Let us examine the dual problem for the ν-SVM algorithm. Our Lagrangian will be in the form:

$$L = \frac{1}{2}||w||^2 - \nu\rho + \frac{1}{n}\sum \xi_i - \sum(\alpha_i(y_i(w^T x_i + b) - \rho + \xi_i) + \beta_i\xi_i = \delta\rho), \tag{6.56}$$

where $\alpha_i, \beta_i, \delta$ are the multipliers. If we compute the partial derivatives from the KKT conditions and set them to 0 we obtain the following conditions:

$$w = \sum \alpha_i y_i x_i, \tag{6.57}$$

$$\alpha_i + \beta_i = \frac{1}{n}, \tag{6.58}$$

$$\sum \alpha_i y_i = 0, \tag{6.59}$$

$$\sum \alpha_i - \delta = \nu. \tag{6.60}$$

Substituting above into L; using $\alpha_i, \beta_i, \delta$ and incorporating kernels for dot products leaves us with the following quadratic optimization problem for *nu*-SVM classification:

$$L(\alpha) = -\frac{1}{2}\sum\sum \alpha_i\alpha_j y_i y_j k(x_i, x_j), \tag{6.61}$$

subject to:

$$0 \leq \alpha_i \leq 1/n \tag{6.62}$$

$$\sum \alpha_i y_i = 0 \tag{6.63}$$

$$\sum \alpha_i \geq \nu \tag{6.64}$$

As above, the resulting decision function can be shown to take the form:

$$f(x) = sign(\sum \alpha_i y_i k(x, x_i) + b) \tag{6.65}$$

Compared to the C-SVM dual problem, there are two differences. First, there is an additional constraint. Second, the linear term $\sum \alpha_i$ no longer appears in the objective function. This has an interesting consequence: it is straightforward to verify that the same decision function is obtained if we start with the primal function:

$$\frac{1}{2}||w||^2 + C(-\nu\rho + 1/n \sum \xi_i). \tag{6.66}$$

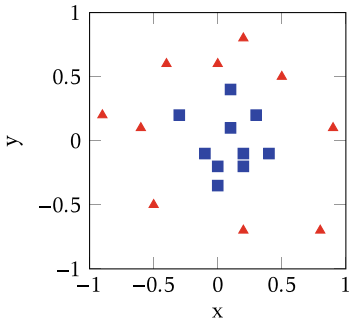
In this, we see the connection between C and ν.

A connection to standard SVM classification and a somewhat surprising interpretation of the regularization parameter C is described by the following result. If ν-SVM classification leads to $\rho > 0$ then C-SVM classification, with C set a priori to $\frac{1}{n\rho}$ leads to the same decision function.

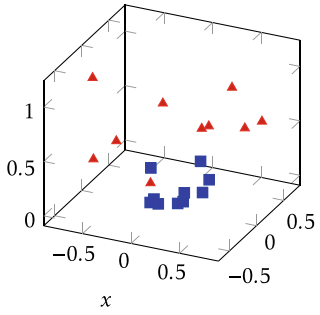
6.4 Non-linearly Separable Problems

The proposed solution is much better and allows us to solve many more problems. We do not demand that the problem be strictly linearly separable. Using the parameter C, we can loosen up the rules to find a better solution. However, in practice, we sometimes have problems that are completely non-linear. For instance, consider the feature space presented in Fig. 6.6a. It is clear that it is not linearly separable. If we try to use our first approach, we will not find any separating hyperplane. No matter what the value of C is, the separation hyperplane will not improve.

The problem is that we are looking for a linear boundary when the boundary in our space is non-linear. However, it does not mean that there is no space in which this problem has no linear boundary. If we find the mapping to such a space, the problem can be solved. Let us consider a very simple example in R^2. Assume that



(a) Linear non-separable in two-dimensional feature space



(b) Linear separable in three-dimensional feature space

Fig. 6.6 Kernel trick

we have 4 training samples: $(-1, -1)$, $(1, 1)$ representing class -1 and $(-1, 1)$, $1, -1)$ representing class 1. It is a typical XOR problem that is not linearly separable in R^2.

The question is if we can map the feature space into more dimensions in such a way that the problem can be linearly separable. It is quite obvious that if we use the function:

$$\Phi(x) = (x_1^2, \sqrt{(}2)x_1x_2, x_2^2) \tag{6.67}$$

we transfer our problem into R^3 space in which it is linearly separable. It is visible because our training points in R^2 become: $\Phi(-1, -1) = (1, \sqrt{(}2), 1)$, $\Phi(1, 1) = (1, \sqrt{(}2), 1)$ and $\Phi(-1, 1) = (1, -\sqrt{(}2), 1)$, $\Phi(1, -1) = (1, -\sqrt{(}2), 1)$. It seems troublesome to search for such a space and move each vector into it. In particular, this space can be much more dimensional. However, if we look at 1.30 we see that all we need to calculate is the dot product between $\Phi(x_i)\Phi(x_j) = (x_ix_j)^2$ in our case.

This means that we do not have to calculate $\Phi(x)$, we also do not have to know the form of the $\Phi(x)$, what we really need is to know how to calculate the dot product in the new space. In fact, if we define the dot product as $\Phi(x_i)\Phi(x_j) = (x_ix_j)^2$ then the $\Phi(x)$ is not unique. It can also be defined as $\Phi(x) = (x_1^1, x_1x_2, x_1x_2, x_2^2)$, which means that we are in the R^4 space.

Every time the inner product appears in the formula 1.30, or in a calculation of the decision function so we can replace it with a generalization of the inner product of the form

$$K(x_i, x_j) = \sum \Phi(x_i)\Phi(x_j) \tag{6.68}$$

K is the function that we will refer to as a kernel. A kernel is a kernel function that quantifies the similarity of two samples. For instance, we could simply take

$$K(x_i, x_j) = \sum x_{ik}x_{jk} \tag{6.69}$$

It just gives us the normal support vector classifier in which we stay in the feature space. It is known as a linear kernel because the support vector classifier is linear in the features. Generally, the linear kernel essentially quantifies the similarity of a pair of observations using Pearson correlation.

However, we can take the kernel function as follows:

$$K(x_i, x_j) = (\sum x_{ik}x_{jk} + a)^d. \tag{6.70}$$

It is a polynomial kernel of degree d, where d is a positive integer. Using such a kernel with $d > 1$, instead of the standard linear kernel, leads to a much more flexible decision boundary. Basically, it amounts to fitting a support vector classifier in a higher dimensional space involving polynomials of degree d, rather than in the original feature space. When the support vector classifier is combined with a non-linear kernel, the decision function is non-linear and has the form:

$$f(x) = \beta_0 + \sum \alpha_i K(x, x_i). \tag{6.71}$$

Another popular choice is the radial kernel, which takes the form:

$$K(x_i, x_j) = \exp(-\gamma ||x_i - x_j||^2). \tag{6.72}$$

Using this kernel, we go to the infinite-dimensional feature space. It would not be very easy to work with $\Phi(x)$ explicitly. However, if we replace $x_i x_j$ by $K(x_i, x_j)$ everywhere in the training algorithm, the algorithm will get a support vector machine which lives in an infinite-dimensional space and will do so in roughly the same amount of time as it would take to train on the unmapped data. The four most popular kernel functions are:

- the linear kernel,
- the polynomial kernel, defined as

$$(\gamma \cdot \langle x_i, x_j \rangle + r)^d, \tag{6.73}$$

- the radial basis function (RBF) kernel, defined as

$$\exp(-\gamma \cdot |x_i - x_j|^2), \tag{6.74}$$

- the sigmoid kernel, defined as

$$\tanh(\langle (x_i, x_j \rangle + r). \tag{6.75}$$

We have one big advantage of using kernel rather than simply enlarging the feature space using functions of the original features. One advantage is computational, which is equivalent to the fact that, using kernels, one only needs to compute $K(x_i, x_j)$ for all $n(n-1)$ pairs of i and j. This can be done without explicitly working in the enlarged feature space. This is important because in many applications of SVMs, the enlarged feature space is so large that computations are intractable. For some kernels, such as the radial kernel, the feature space is implicit and infinite dimensional, so we could never do the computations there anyway!

Example 3 *(SVM Radial Basis Function (RBF) kernel used on the Iris data set)* The previous example performs rather poorly. To make the prediction more successful, we use a more sophisticated kernel. One of such a scheme is the RBF kernel. In Listing 6.4, we implement Eq. 6.74.

```
def build_kernel(data_set, kernel_type='linear'):
    kernel = np.dot(data_set, data_set.T)
    if kernel_type == 'rbf':
        sigma = 1.0
        objects_count = len(data_set)
        b = np.ones((len(data_set), 1))
        kernel -= 0.5 * (np.dot((np.diag(kernel)*np.ones((1, objects_count
    ))).T, b.T)
                         + np.dot(b, (np.diag(kernel) * np.ones
```

```
                            ((1, objects_count))).T.T))
        kernel = np.exp(kernel / (2. * sigma ** 2))
    return kernel
```

Listing 6.4 RBF kernel implementation

The implementation of the classification method is slightly different and needs to be adjusted to use the RBF to predict the label.

```
def classify_rbf(test_data_set, train_data_set, lambdas, targets, b,
    vector_number, support_vectors, support_vectors_id):
    kernel = np.dot(test_data_set, support_vectors.T)
    sigma = 1.0
    c = (1. / sigma * np.sum(test_data_set ** 2, axis=1) * np.ones((1, np.
    shape(test_data_set)[0]))).T
    c = np.dot(c, np.ones((1, np.shape(kernel)[1])))
    sv = (np.diag(np.dot(train_data_set, train_data_set.T))*np.ones((1,len
    (train_data_set)))).T[support_vectors_id]
    aa = np.dot(sv,np.ones((1,np.shape(kernel)[0]))).T
    kernel = kernel - 0.5 * c - 0.5 * aa
    kernel = np.exp(kernel / (2. * sigma ** 2))

    y = np.zeros((np.shape(test_data_set)[0], 1))
    for j in range(np.shape(test_data_set)[0]):
        for i in range(vector_number):
            y[j] += lambdas[i] * targets[i] * kernel[j, i]
        y[j] += b
    return np.sign(y)
```

Listing 6.5 Implementation of C-SVM for the RBF kernel

The execution is done in the same way as in the linear example. The accuracy is significantly higher and easily achieved 85%.

6.5 Extensions

So far, our discussion has been limited to the case of binary classification, that is, classification in the two-class setting. How can we extend SVMs to the more general case where we have some arbitrary number of classes? It turns out that the concept of separating hyperplanes upon which SVMs are based does not lend itself naturally to more than two classes. Although a number of proposals for extending SVMs to the K-class case have been made, the two most popular are the one-versus-one and one-versus-all approaches. We briefly discuss those two approaches here.

One class SVM

In the previous section, we addressed multiclass problems which seem to be typical for most real-world problems. However, sometimes we are interested in one class only. We just want to separate the samples from all the others. It seems like a typical two-class problem (in fact, we even use such an approach in the one-versus-all strategy). However, what happens when we have training samples that represent one

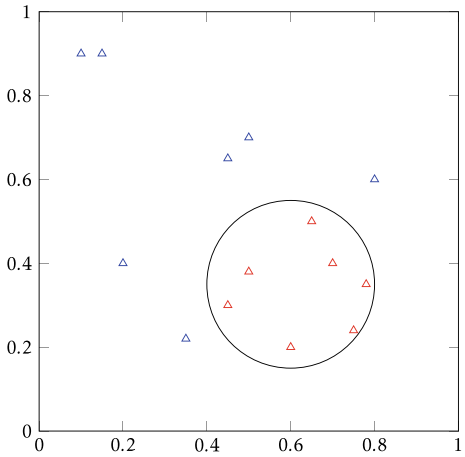


Fig. 6.7 One class example. The classified one class examples are marked with red and outfittery cases with blue

class only? Can we train a classifier that learns to recognize the samples in this class and which will reject all others?

The basic idea is to enclose the data with a hypersphere and classify the new data as normal if they fall within the hypersphere and otherwise as anomalous data (see Fig. 6.7).

Let us assume that we have a training data set T with samples $T = \{x_1, x_2, \ldots, x_n\} \in R^p$ and let r be the radius of the hypersphere and $c \in R^p$ be the center of this hypersphere. To find the minimum enclosing the hypersphere, we have to minimize r^2 subject to:

$$||\Phi(x_i) - c||^2 \leq r^2, \quad i = 1, \ldots, p \tag{6.76}$$

Then we introduce the Lagrangian multiplier for each constraint and obtain

$$L(, r, \alpha) = r^2 + \sum \alpha_i (||\Phi(x_i) - c||^2 - r^2), \quad \alpha_i \geq 0 \tag{6.77}$$

As we remember from the previous section, both derivatives must be equal to zero so

$$\frac{\delta L(c, r, \alpha)}{\delta c} = 2 \sum \alpha_i (\Phi(x_i) - c) = 0 \tag{6.78}$$

and

$$\frac{\delta L(c, r, \alpha)}{\delta r} = 2r(1 - \sum \alpha_i) = 0 \tag{6.79}$$

from above we get

$$\sum \alpha_i) = 1 \quad and \quad c = \sum \alpha_i \Phi(x_i). \tag{6.80}$$

according to that we get the dual form:

$$W(\alpha) = \sum \alpha_i)||\Phi(x_i) - c||^2 = \sum \alpha_i k(x_i, x_i) - \sum \sum \alpha_i \alpha_j k(x_i, x_j) \quad (6.81)$$

Therefore, we have to maximize $W(\alpha)$ subject to:

$$\sum \alpha_i) = 1, \quad i = 1, \ldots, p \quad (6.82)$$

Similarly as in standard two-class SVM $\alpha_i > 0$ only if the corresponding sample x_i lies on the separating hypersphere. Now, if we look at the decision function, it has the following form:

$$f(x) = sign(r^2 - ||\Phi(x) - c||^2), \quad (6.83)$$

and finally we get

$$f(x) = sign(r^2 - (k(x, x) - 2 \sum \alpha_i k(x, x_i) + \sum \sum \alpha_i \alpha_j k(x_i, x_j)). \quad (6.84)$$

If we have some noise in our training set, the *hard* enclosing hypersphere approach may force a larger radius than should really be needed. In other words, the solution would not be robust.

Our goal now is to find the minimum enclosing hypersphere that contains (almost) all training examples, but not some small portion of extreme training examples.

We can use the same trick with the soft margins as with the hyperplane, so we introduce slack variables $\xi_i > 0, i = 1, \ldots, p$ and we have to minimize

$$r^2 + C \sum \xi_i \quad (6.85)$$

subject to

$$||\Phi(x_i) - c||^2 \leq r^2 + \xi_i, \quad \xi_i > 0, \quad i = 1, \ldots, p \quad (6.86)$$

which finally leads to minimizing:

$$\sum \sum \alpha_i \alpha_j K(x_i, x_j) - \sum \alpha_i k(x_i, x_i) \quad (6.87)$$

subject to

$$0 \leq \alpha_i \leq C, \quad \sum \alpha_i = 1. \quad (6.88)$$

One-Versus-One and One-Versus-All Classification

Suppose that we would like to perform classification using SVMs, and there are $K > 2$ classes. A one-versus-one or all-pair approach constructs $\frac{K*(K-1)}{2}$ SVMs, each of which compares a pair of classes. For example, such an SVM might compare the

k-th class, coded as $+1$, to the k-th class, coded as -1. We classify a test observation using each of the $\frac{K*(K-1)}{2}$ classifiers and tally the number of times the test observation is assigned to each of the K classes. The final classification is performed by assigning the test observation to the class to which it was assigned the most frequently in these pairwise classifications.

The one-versus-all approach is an alternative procedure for applying SVMs one-versus in the case of $k > 2$ classes. We fit the k SVMs, each time comparing one of all K classes with the remaining $k - 1$ classes.

For Further Reading

1. Christmann A, Steinwart I (2008) Support vector machines. Springer
2. Campbell C, Ying Y (2011) Learning with support vector machines. Springer
3. Cerulli G (2023) Fundamentals of supervised machine learning with applications in Python, R, and Stata. Springer

References

1. Vapnik V (1995) The Nature of Statistical Learning Theory. Springer-Verlag, New York, USA
2. Vapnik V (1998) Statistical Learning Theory. John Wiley and Sons, New York, USA
3. Vapnik V (1979) Estimation of dependences based on empirical data. Nauka
4. Cortes C, Vapnik V (1995) Support-vector networks. Mach Learn 20:273–295
5. Valiant LG (1984) A theory of the learnable. Commun ACM 27:1134–1142
6. Kuhn HW, Tucker AW (1951) Nonlinear programming. In: Proceedings of 2nd Berkeley symposium. University of California Press, Berkeley, pp 481—492
7. Schölkopf B, Smola AJ, Williamson RC, Bartlett PL (2000) New support vector algorithms. Neural Comput 12(5):1207–1245

Chapter 7
Ensemble Methods

Ensemble methods are also known as combined classifiers and are a group of methods that combine more than just one classifier to get better results than each classifier on its own. The classifiers are built on the same data sets. Depending on the way an ensemble method is built, we have a few major types:

- boosting,
- bagging,
- stacking.

A comparison of one classifier against many classifiers is given in Fig. 7.1.

As shown in the figure, collecting many poor classifiers into one ensemble classifier can result in a classifier that performs well. A general formula of ensemble methods looks like following:

$$\bar{C}(X) = \sum_{i=1}^{T} w_i C_i(X). \tag{7.1}$$

It's important to mention that combining identical classifiers is useless, because same classifier will result with same boundaries. By same classifier we mean not exactly the same classification method, but the model that is trained using the same classification method, parameters, and exact training data. In other words, we need many low-quality classifiers and a way to combine them together, so we get a classifier that perform very well. In a classification problem, the low-quality classifiers needs to be better than guessing. It means that for a binary classification problem we should achieve more than 50%.

In this part, we cover the most popular types of ensemble methods and include the implementations of random forest. XGBoost is another bagging method that is a popular Kaggle winning method. In this chapter, we explain the method in details. We cover also AdaBoost that is a bagging method and grading that is a stacking methods.

K. Przystalski et al., *Pattern Recognition Primer*,
https://doi.org/10.1007/978-3-031-91816-2_7

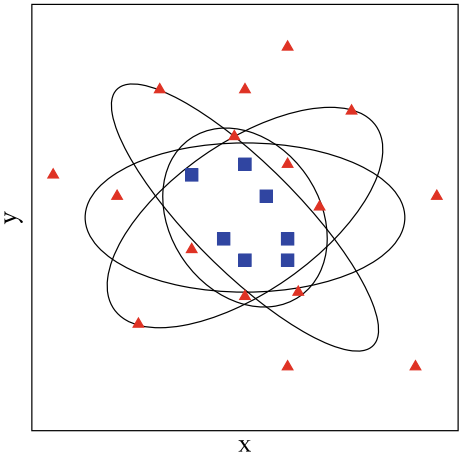


Fig. 7.1 Each classifier discriminant boundary is marked with black. Only together we can distinguish between the red and blue objects fully

Boosting methods [1] take a different weighting schema of resampling than bagging. The component classifiers are built sequentially, and examples that are misclassified by previous components are chosen more often than those that are correctly classified. New classifiers are influenced by performance of previously built ones. The new classifier is encouraged to become an expert for instances classified incorrectly by the earlier classifier. There are few methods of boosting type:

- AdaBoost,
- Arcing,
- RegionBoost,
- Stumping.

The differences are minor and most boosting methods are a modification of AdaBoost. Stumping differs from other ones, because it is used in decision trees. RegionBoost uses the kNN method as a part of the algorithm.

7.1 AdaBoost

AdaBoost [2] is the most popular boosting method and stands for adaptive boost. We use the weights to set the importance of the objects. The more important an object is, the more frequently it is chosen in the training data set. The method consists of the following steps:

1. initialize weights to $\frac{1}{N}$, where N is the number of data points,
2. loop steps below until

$$\varepsilon_t < \frac{1}{2} \tag{7.2}$$

   or maximum number of iteration is reached,
3. train classifier on S, $w^{(t)}$ and get a hypothesis $h_t(x_n)$ for datapoints x_n,

4. compute error

$$\varepsilon_t = \sum_{n=1}^{N} w_n^{(t)} I(y_n \neq h_t(x_n)), \tag{7.3}$$

5. set

$$\alpha_t = \log(\frac{1-\varepsilon_t}{\varepsilon_t}), \tag{7.4}$$

6. update weights:

$$w_n^{(t+1)} = \frac{w_n^{(t)} \exp \alpha_t I(y_n \neq h_t(x_n))}{Z_t}, \tag{7.5}$$

where Z_t is a normalization constant,

7. output

$$f(X) = \text{sign}(\sum_{t=1}^{T} \alpha_t h_t(x)). \tag{7.6}$$

The weights are set in each loop, where in each loop, we add a new classifier with the current weights and data set. It can be drawn as in Fig. 7.2. We use the same data set example as for decision trees. In the first step, we have the data set with similar weights. Based on the prediction of the first model, we know what objects are misclassified. In the second step, the weights are higher for the misclassified ones. In the Fig. 7.2, we see it as bigger objects. In the first iterations, we will have more of such objects, and during the next iterations, the number of objects that are misclassified should decrease.

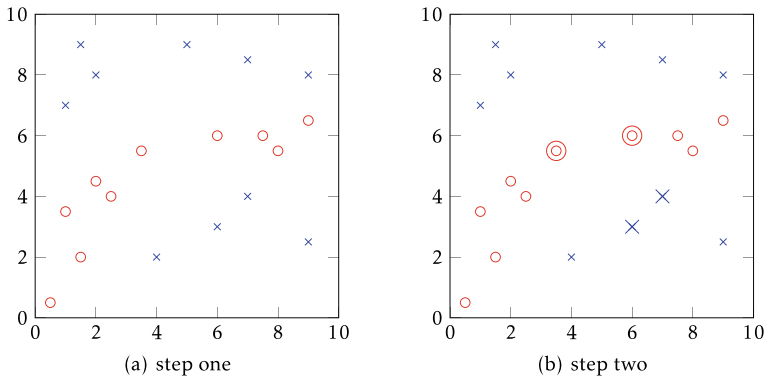


Fig. 7.2 Two first steps of AdaBoost method

The implementation consists of five methods. We use a decision tree method scikit-learn implementation for the training. It is easy to train with the `fit` method, most scikit-learn classifiers are trained with the `fit`. It makes usage a bit simpler. In Listing 7.1, an example of a decision tree training using a set of weights is shown.

```
1 def train_model(classifier, weights):
2     return classifier.fit(X=test_set, y=test_labels, sample_weight=weights
      )
```

Listing 7.1 Model training

The error calculation is based on the output of the `predict` method that is next used to calculate the error rate (7.3). It uses the weights and the accuracy vector that checks the returns the predictions test vector. The vector consists of binary values: 0 for a positive prediction and 1 if the classifier did not correctly predict the label.

```
1  def calculate_accuracy_vector(predicted, labels):
2      result = []
3      for i in range(len(predicted)):
4          if predicted[i] == labels[i]:
5              result.append(0)
6          else:
7              result.append(1)
8      return result
9
10 def calculate_error(weights, model):
11     predicted = model.predict(test_set)
12     return np.dot(weights,calculate_accuracy_vector(predicted, test_labels
       ))
```

Listing 7.2 Error rate calculation

The `calculate_accuracy_vector()` method loops over the predicted labels and compares it to the test labels. The variable α uses the error rate (lines 1–2, and 4).

The new weights are the most important part of the algorithm. We used the old weights and adjusted them and changed the weight value of the wrong-predicted objects. In the second line of the Listing 7.3, we get the fraction counter of the Eq. 7.5.

```
1 def set_alpha(error_rate):
2     return np.log((1-error_rate)/error_rate)
3
4 def set_new_weights(old_weights, alpha, model):
5     new_weights = old_weights * np.exp(np.multiply(alpha,
       calculate_accuracy_vector(model.predict(test_set), test_labels)))
6     Zt = np.sum(new_weights)
7     return new_weights / Zt
```

Listing 7.3 New weights calculation function

We save the alphas and models that are used during the training to compare how it changed. After the weights are changed, the new prediction can be performed. In the following example, we show how it works on the classic wine data set.

Example 1 (*Boosting the wine*) In the first step, we load the data set from the scikit-learn data sets library as shown in Listing 7.4. The set is then divided into training and testing sets by choosing 140 random objects from the set.

```
from sklearn.datasets import load_wine
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

wine = load_wine()
random_objects = np.random.randint(0, 178, size=140)
data_set = wine.data[random_objects]
labels = wine.target[random_objects]

train_set, test_set, train_labels, test_labels = train_test_split(data_set
    , labels, test_size=0.5, random_state=42)
```

Listing 7.4 Adaboost prediction of wine dataset

In the next step, we run the classifier and iterate by adjusting the weights. The implementation is shown in the Listing 7.5. First, the model is trained with the same weights for each object, next the error rate is calculated. Based on the error rate, the new weights are calculated. The ones that were misclassified get a higher weight.

```
classifier.fit(X=train_set, y=train_labels)
alphas = []
classifiers = []
for iteration in range(number_of_iterations):
    model = train_model(classifier, weights)
    error_rate = calculate_error(weights, model)
    alpha = set_alpha(error_rate)
    weights = set_new_weights(weights, alpha, model)
    alphas.append(alpha)
    classifiers.append(model)
```

Listing 7.5 Adaboost weights adjustment

Finally, we get a classifier that achieves better results. The weights of the data set before and after the weight changes are shown in Fig. 7.3. We can see that objects in some areas are harder to assign to the proper class. The plot is two-dimensional even if there are more features in the data set.

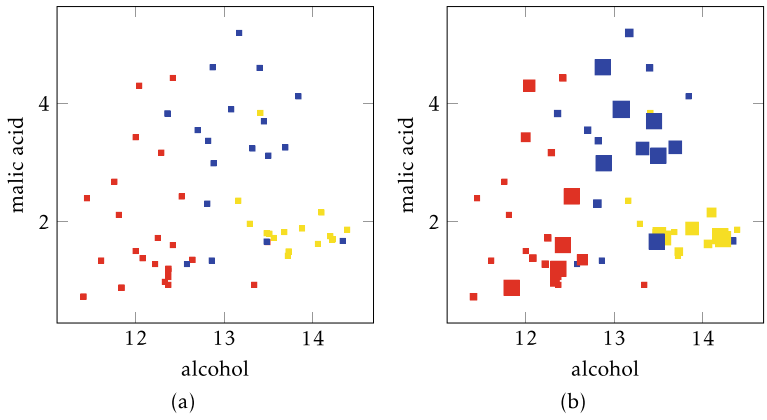


Fig. 7.3 Wine classification of testing set (**a**) and weighted testing set (**b**)

Other boosting methods

Arcing is a modified AdaBoost, it differs how it calculates the error:

$$\varepsilon_{(t+1)}(i) = \frac{(1 + \sum_{n=1}^{N} I(y_n \neq h_t(x_n)))}{Z_t}. \tag{7.7}$$

We use the normalization constant Z_t to obtain a probability density. Finally, we vote for the label:

$$f(X) = \arg\max_i \sum_i^T (f_i(X) = y). \tag{7.8}$$

The arcing weight function can be implemented similarly to the AdaBoost weight function (Listing 7.6).

```
def set_new_weights(model):
    new_weights = (np.add(1,calculate_accuracy_vector(model.predict
    (test_set), test_labels)))/(np.sum(np.add(1,calculate_accuracy_vector
    (model.predict(test_set), test_labels))))
    return new_weights
```

Listing 7.6 Arcing weights calculation function

The voting is slightly different compared to the classic AdaBoost method and can be implemented as in Listing 7.7).

```
def get_prediction(x):
    predictions = []
    for i in range(len(classifiers)):
        predicted = classifiers[i].predict(x)
        predictions.append(predicted)
    return predictions[np.argmax(predictions)]
```

Listing 7.7 Arcing prediction with voting

RegionBoost is another modification of AdaBoost. The difference is that the weights of each object depends locally on the importance of other k closest neighborhood objects:

$$w_i(x_i) = \frac{1}{T} \sum_{i=1}^{T} kNN(K, C_i, x_i, y_i), \tag{7.9}$$

where

$$kNN(K, C_i, x_i, y_i) = \frac{1}{K}[\sum_{x_s \in N(K,X)} I(f(x_s = y_s))]. \tag{7.10}$$

Stumping is a type of boosting that is applied to trees. A stump of a tree is a piece of tine that is left over when you cut the rest. Stumping consists of simply taking the root of the tree and using that as the decision maker. For each classifier, you use the very first question that makes up the root of the tree, and that is it.

7.2 Bagging

Bagging [3] is a short name for bootstrap aggregation. Generate individual classifiers on bootstrap samples of the training set. A bootstrap sample is a sample of the training set taken from the original data set with replacement, so that we may get some data several times and others not at all. The bootstrap sample is the same size as the original data set. Bagging traditionally uses component classifiers of the same type and combines prediction by a simple majority vote across. The steps of a bagging algorithm are as follows:

1. create T bootstrap samples S_i,
2. for each sample S_i train a classifier,
3. vote:

$$f(x) = \arg\max \sum_{i}^{T} (f_i(X) = y). \tag{7.11}$$

The voting part chooses the most common label among the models. It is a good practice to have an odd value of models for binary decision problems. The approach can be drawn as shown in Fig. 7.4. We have T bootstrap sets (S_T) used to train T models (C_T) using the same method. As each model is trained on different sets, it differs from other models. A popular example of a bagging method is the random forest method [4]. The general bagging steps can be specified for the random forest method as follows:

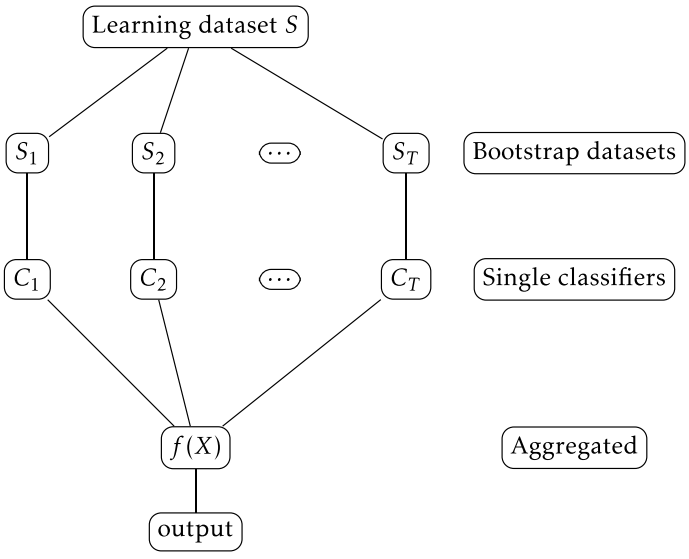


Fig. 7.4 General overview of the bagging method steps

1. for each tree of N, we create a new bootstrap data set and train it,
2. at each node of the decision tree, randomly select m features and compute the information gain only on that set of features, selecting the optimal one,
3. repeat until the tree is complete.

All steps should be clear, as we have already covered the decision trees in the previous chapter. A simplified random forest method can be implemented with only four functions. The ensemble methods are complex enough that simple examples as used in the beginning of this book are not complex enough for testing purposes. That is why, for this example, we used one of the classic data sets called Iris. It consists of three types of iris flowers.

In the following example, we use the random forest method. It is a bagging method that, as a classification method, uses the decision tree. Instead of implementing the tree from scratch, as we did in the previous chapter, we use the scikit-learn implementation of it (Listing 7.8).

```
from sklearn import tree
import numpy as np
from sklearn.metrics import accuracy_score

decision_tree = tree.DecisionTreeClassifier()
```

Listing 7.8 Libraries import for bagging method

The crucial part of the bagging method is the setup of bootstrap sets. Each set can be chosen randomly using the `random.random` NumPy method. It can be implemented as in the Listing 7.9.

```
def create_bootstrap_data():
    bootstrap_ids = np.random.randint(0, len(data_set), size=len(data_set)
    )
    return data_set[bootstrap_ids,:],labels[bootstrap_ids]
```

Listing 7.9 Bootstrap set generation method

We have divided the implementation of the models into two separate methods. In the `build_classifier` method, we build just one instance of a model built using a bootstrap set. In the next method, we combine each model into a list of models. In each loop, we create a new bootstrap set, then use it to train a new model, and finally add it to the list. The implementation is shown in the Listing 7.10. The method implemented is also known as the random forest.

```
def build_classifier(data_set, labels):
    decision_tree = tree.DecisionTreeClassifier()
    decision_tree.fit(data_set, labels)
    return decision_tree

def build_classifiers(cases):
    classifiers = []
    for case in range(cases):
        bootstrap_set, bootstrap_labels = create_bootstrap_data()
        classifier = build_classifier(bootstrap_set, bootstrap_labels)
        classifiers.append(classifier)
    return classifiers
```

Listing 7.10 Bagging classifiers preparation and combination method

The remaining voting method can be implemented as two loops: one that receives the classifier output, and the second counts the prediction by label and returns the most predicted label. The implementation is given in Listing 7.11.

```
def vote(classifiers, test_data):
    output = []
    for classifier in classifiers:
        output.append(classifier.predict(test_data))
    output = np.array(output)
    predicted = []
    for i in range(len(test_data)):
        classified = output[:, i]
        counts = np.bincount(classified)
        predicted.append(np.argmax(counts))
    return predicted
```

Listing 7.11 Bagging voting method

Finally, we can check the results based on ten classifiers by running the code as given in Listing 7.12. The last line calculates the accuracy of the model.

```
classifiers = build_classifiers(10)
predicted = vote(classifiers, test_data_set)
accuracy = accuracy_score(test_labels, predicted)
```

Listing 7.12 Bagging method executing

Now, we can compare the ensemble method with a typical decision tree. We will use the same data set, but we modify the tree a bit as the overall challenge for a tree in these cases is not high. We limit the tree depth to 2 for both the ensemble models and the decision tree with which we will compare.

Example 2 (*Random forest on flowers*) The random forest is just a group of decision trees fed with different training sets that work as a bagging ensemble method. It usually works better than a single decision tree. In Listing 7.13, an example of a single decision tree is shown. It uses a bootstrap set similarly to the random forest trees.

```
tree_data_set, tree_labels = create_bootstrap_data(train_set, train_labels
    )
decision_tree = tree.DecisionTreeClassifier(max_depth=3)
decision_tree.fit(tree_data_set, tree_labels)
predicted = decision_tree.predict(test_set)
accuracy = accuracy_score(test_labels, predicted)
print(accuracy)
```

Listing 7.13 Iris single decision tree classifier

In Listing 7.14, we use the same data set as in the single tree, but we create a few bootstrap sets to train the same number of trees as sets. The classical Iris data set is part of the scikit-learn data set. The bootstrap sets are created within the classifier training method (Listing 7.10).

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split

iris = load_iris()

random_objects = np.random.randint(0, 130, size=120)
data_set = iris.data[random_objects]
labels = iris.target[random_objects]
```

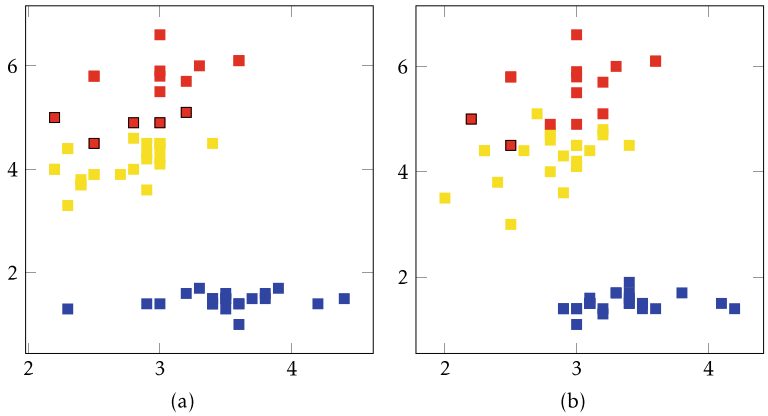


Fig. 7.5 Iris classification using a decision tree (**a**) and random forest (**b**). Cases with black square border are the ones that were misclassified

```
9
10 train_set, test_set, train_labels, test_labels = train_test_split(data_set
       , labels, test_size=0.5, random_state=42)
11
12 classifiers = build_classifiers(5, train_set, train_labels)
13 predicted = vote(classifiers, test_set)
14 accuracy = accuracy_score(test_labels, predicted)
15 print(accuracy)
```

Listing 7.14 Iris bagging (random forest) classification example

The final classification result is shown in Fig. 7.5. The misclassified objects are marked with a black boundary. On the left side, there are five objects that are misclassified when only one decision tree is used. On the other side, we have only two such cases. Here the random forest of five trees is used.

7.3 Stacking

Stacking as is can be deduced from the name of this ensemble method, is a method that uses different classifiers and builds these into a stack. Usually, in the stacking method, the models used for classification are architecturally different, as the input data set is the same for each method at the first level of the stack. The second level of the stack is a set of the predictions for each model in the first level. The third stack level is built from one meta-classifier that uses the validation set (second level) and based on that the final prediction is made. An overview of the method is shown in Fig. 7.6.

The stacking method can be divided into the following steps:

1. create T classifiers and learn each to get m predictions (hypothesis h_t),
2. construct data set of predictions into a validation data set,
3. construct a $\bar{C}$ meta-classifier that combines all C_m classifiers.

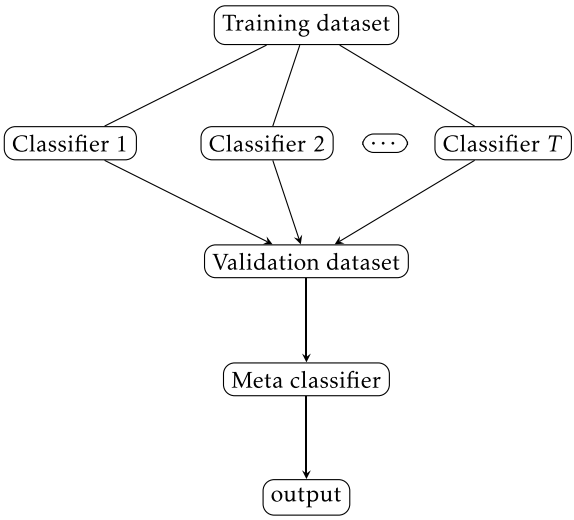


Fig. 7.6 General overview of the stacking method

Table 7.1 Example stacking classifiers prediction results

Predictions				
C_1	C_2	C_3	C_T	$\bar{C}$
1	1	0	1	1
0	0	0	1	0
…	…	…	…	
0	1	1	1	1

An example of a few classifiers C_1, C_2, C_3, C_T is shown in Table 7.1. The predictions of each are chosen randomly (columns C_1 to C_T). These predictions are the inputs for the $\bar{C}$ classifier. The final prediction is given in the last column. Similarly to the previous ensemble method, we use different classifiers from the scikit-learn package. In the current example, we include five different methods that are given in the Listing 7.15. We use the QDA, linear regression, kNN, and a decision tree classifier.

```
import numpy as np

from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy_score
```

Listing 7.15 Stacking libraries import

The implementation can be divided into two methods, one for each level. The function `build_classifiers` where we train the models (Listing 7.16) based on the scikit-learn implementations imported in the previous listing. The function returns a list of models trained on the same data set.

```
def build_classifiers(train_set, train_labels):

    neighbors = KNeighborsClassifier()
    neighbors.fit(train_set, train_labels)

    bayes = GaussianNB()
    bayes.fit(train_set, train_labels)

    qda = QuadraticDiscriminantAnalysis()
    qda.fit(train_set, train_labels)

    return neighbors, bayes, qda
```

Listing 7.16 Stacking classification methods building

In our example, the meta-classifier is built using the decision tree method. The meta-classifier used in the Listing 7.17 is used (line 9) between two loops. In the first loop, we collect the predictions of the first-level models. These predictions are then used as input for the decision tree method. Finally, the last loop goes through a testing set and returns the final prediction.

```
def build_stacked_classifier(classifiers, train_set, train_labels,
    test_set, test_labels):
    output = []
    for classifier in classifiers:
        output.append(classifier.predict(train_set))
    meta_set = np.stack((output[0],output[1],output[2]), axis = 1)

    decision_tree = DecisionTreeClassifier()
    decision_tree.fit(meta_set, train_labels)

    output = []
    for classifier in classifiers:
        output.append(classifier.predict(test_set))
    meta_test_set = np.stack((output[0],output[1],output[2]), axis = 1)
    predicted = decision_tree.predict(meta_test_set)
    return predicted
```

Listing 7.17 Stacked classification method

Example 3 (*Breast cancer classified using stacking method*) To execute above, we can use three similar lines of code as in the previous method (Listing 7.18).

```
from sklearn.datasets import load_breast_cancer

breast = load_breast_cancer()

random_objects = np.random.randint(0, 178, size=140)
data_set = breast.data[random_objects]
labels = breast.target[random_objects]

train_set, test_set, train_labels, test_labels = train_test_split(data_set
    , labels, test_size=0.3, random_state=42)

classifiers = build_classifiers(train_set, train_labels)
predicted = build_stacked_classifier(classifiers, train_set, train_labels,
     test_set, test_labels)
accuracy = accuracy_score(test_labels, predicted)
print(accuracy)
```

Listing 7.18 Stacking classification method used on breast dataset

The accuracy achieved using the combined classifier is 97.62% where using a single decision tree only 90,48%.

Table 7.2 Grading base training set

Predictions				
C_1	C_2	C_3	C_T	Label
1	1	0	1	1
0	0	0	1	0
…	…	…	…	
0	1	1	1	1

Table 7.3 Grading attribute set

Attributes				
x_1	x_2	x_3	x_n	Graded predictions
0.2	0.5	−0.1	0.4	+
−0.1	0.15	−0.7	0.5	+
…	…	…	…	−
0.8	0.2	−0.24	0.6	+

Grading

Grading can be considered as a modified stacking method as the main goal here is to use the input of different models. The base data set is almost the same, but instead of using a meta-classifier, we only have the original labels for training purposes, as shown in Table 7.2. The main difference is in the way the data for the meta-classifier are given. The attributes (features) are given as input (Table 7.3). Graded predictions are the values of whether or not a prediction of a given classifier is done properly.

The implementation is simple as in the previous methods, where the first function is implemented almost the same as in the stacking classifier (Listing 7.19).

```
def calculate_accuracy_vector(predicted, labels):
    result = []
    for i in range(len(predicted)):
        if predicted[i] == labels[i]:
            result.append(1)
        else:
            result.append(0)
    return result

def build_grading_classifier(classifiers, train_set, train_labels):
    output = []
    matrix = []
    for classifier in classifiers:
        predicted = classifier.predict(train_set)
        output.append(predicted)
        matrix.append(calculate_accuracy_vector(predicted, train_labels))

    grading_classifiers = []
    for i in range(len(classifiers)):
        tree = DecisionTreeClassifier()
        tree.fit(train_set, matrix[i])
```

```
        grading_classifiers.append(tree)
    return grading_classifiers
```

Listing 7.19 Grading meta classifier implementation

The second function is about the grades of the predictions. We loop the predictions and grade if the prediction is correct or wrong (Listing 7.20).

```
def get_grads(predicted, labels):
    result = []
    for i in range(len(predicted)):
        if predicted[i] == labels[i]:
            result.append(1)
        else:
            result.append(0)
    return result
```

Listing 7.20 Grading grades calculation

Testing the predictions includes the grades and the prediction. As in Listing 7.21, we have two models, one for the prediction and the second for the grading.

```
def test_prediction(classifiers, grading_classifiers, test_set, i):
    prediction = classifiers[i].predict(test_set)
    grad = grading_classifiers[i].predict(test_set)
    return prediction, grad
```

Listing 7.21 Grading prediction testing

Example 4 (*Breast classifier grading*) The main part invokes the three functions that are explained above. An example of the implementation is shown in the Listing 7.22.

```
from sklearn.datasets import load_breast_cancer

breast = load_breast_cancer()

random_objects = np.random.randint(0, 178, size=140)
data_set = breast.data[random_objects]
labels = breast.target[random_objects]

train_set, test_set, train_labels, test_labels = train_test_split(data_set
    , labels, test_size=0.5, random_state=42)

classifiers = build_grad_classifiers(train_set, train_labels)
grading_classifiers = build_grading_classifier(classifiers, train_set,
    train_labels)
prediction, grad = test_prediction(classifiers, grading_classifiers,
    test_set, 1)
```

Listing 7.22 Grading main code used on breast set

The results are shown in Table 7.4. The classifier makes two mistakes. The object #24 and #41 are the ones where the grading method is found where the classification is possibly made wrong. The accuracy of the model achieved is about 85,71%, so the grading method did not find all the mistakes.

Table 7.4 Grading of classifier on the breast data set

ID	Prediction	Grading	Label	ID	Prediction	Grading	Label
1	1	1	1	36	1	1	1
2	0	1	0	37	0	1	0
3	1	1	1	38	1	1	1
4	0	1	0	39	0	1	0
5	1	1	1	40	1	1	1
6	1	1	1	41	0	0	1
7	1	1	1	42	0	1	1
8	1	1	1	43	0	1	1
9	1	1	1	44	0	1	0
10	0	1	0	45	0	1	0
11	1	1	1	46	1	1	1
12	1	1	1	47	1	1	0
13	1	1	1	48	0	1	0
14	0	1	0	49	0	1	0
15	1	1	0	50	0	1	0
16	0	1	0	51	0	1	0
17	0	1	0	52	0	1	0
18	0	1	1	53	1	1	1
19	1	1	1	54	1	1	1
20	0	1	0	55	0	1	0
21	1	1	1	56	1	1	1
22	1	1	1	57	0	1	1
23	1	1	1	58	0	1	0
24	1	0	0	59	0	1	0
25	0	1	0	60	0	1	0
26	0	1	0	61	1	1	1
27	1	1	1	62	1	1	1
28	1	1	0	63	0	1	0
29	0	1	0	64	1	1	1
30	0	1	0	65	1	1	1
31	0	1	0	66	1	1	1
32	0	1	0	67	1	1	1
33	1	1	0	68	0	1	0
34	0	1	0	69	0	1	0
35	1	1	1	70	1	1	1

For Further Reading

1. Kuncheva LI (2014) Combining pattern classifiers: methods and algorithms. Wiley
2. Kunapuli G (2023) Ensemble methods for machine learning. Manning
3. Zhang C, Ma Y (2014). Ensemble machine learning: methods and applications. Springer
4. Kyriakides G, Margaritis KG (2019) Hands-on ensemble learning with Python. Packt Publishing
5. Zhou ZH (2012) Ensemble methods. Chapman and Hall/CRC

References

1. Schapire RE (1990) The strength of weak learnability. Mach Learn 5:197–227
2. Freund Y, Schapire RE (1997) A decision-theoretic generalization of on-line learning and an application to boosting. J Comput Syst Sci 55:119–139
3. Breiman L (1996) Bagging predictors. Mach Learn 26:123–140
4. Breiman L, Friedman JH, Ohlsen RA, Stone CJ (1984) Classification and regression trees. Chapman and Hall

Chapter 8
Neural Networks

Natural (biological) neurons are the fundamental building blocks of the nervous system, particularly the brain. These biological units process and transmit information using electrical and chemical signals. Each neuron has three main components (see Fig. 8.1):

1. **Dendrites**: Branch-like structures that receive signals from other neurons. These inputs can be excitatory or inhibitory, influencing the likelihood of firing of the neuron.
2. **Cell Body (Soma)**: Integrates the incoming signals from dendrites. If the combined input exceeds a certain threshold, the neuron generates an action potential.
3. **Axon**: A long and slender projection that transmits the action potential to other neurons via synapses. In the synapse, chemical neurotransmitters are released, which influence the activity of the next neuron.

Processing within a neuron is an electrochemical event. Input signals, arriving as neurotransmitters in synapses, generate changes in the membrane potential of the neuron. If these changes sum up to exceed a critical threshold, an action potential—a rapid electrical impulse—travels down the axon to communicate with subsequent neurons. This threshold mechanism is the key to neuronal function as a decision-making unit in the nervous system.

Natural neurons communicate within highly interconnected networks, where the strength of synaptic connections (synaptic plasticity) is adapted based on experience, enabling learning and memory.

Artificial neural networks (ANNs) are computational models inspired by the structure and function of natural neurons. In an ANN, artificial neurons, often called nodes or units, are organized into layers: an input layer, one or more hidden layers, and an output layer.

Each artificial neuron mimics the behavior of a biological neuron:

1. **Input Signals**: Each neuron receives input, typically represented as numerical values.

K. Przystalski et al., *Pattern Recognition Primer*,
https://doi.org/10.1007/978-3-031-91816-2_8

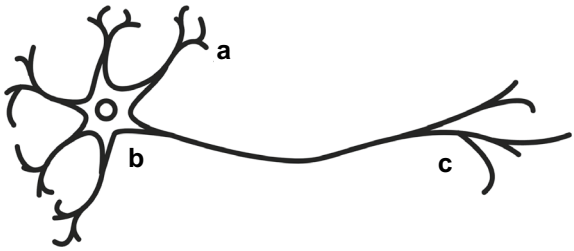


Fig. 8.1 Natural (biological) neuron has three main components: **a** dendrites, **b** soma, and **c** axon

2. **Weights**: Each input is associated with a weight that indicates its importance. The neuron calculates a weighted sum of the inputs.
3. **Bias**: A bias term is added to the weighted sum, allowing the neuron to change its activation threshold.
4. **Activation Function**: The result is passed through a non-linear activation function, determining the neuron output.

The neuron output is then passed to the neurons in the next layer. By training the network and adjusting weights and biases using optimization algorithms such as stochastic gradient descent, the ANN learns to map inputs to outputs effectively, enabling it to perform tasks such as classification, regression, or generation.

Unlike biological networks, ANNs operate using strictly mathematical rules and lack the biochemical complexity of natural neurons. However, conceptual similarity, the use of interconnected processing units, is key to their design.

The first mathematical description of a neuron was proposed by Warren McCulloch and Walter Pitts in 1943 [1]. Their model, known as the McCulloch-Pitts Neuron, formalized the idea of a neuron as a simple computational unit:

1. The neuron receives inputs $x_1, x_2, \ldots, x_n$, each associated with a binary value (0 or 1).
2. Each input is multiplied by the corresponding weight $w_1, w_2, \ldots, w_n$, representing the connection strength.
3. The neuron computes a weighted sum:

$$z = \sum_{i=1}^{n} w_i x_i. \tag{8.1}$$

4. If the weighted sum exceeds a predefined threshold $theta$, the neuron fires and produces a 1. Otherwise, it outputs a 0:

$$y = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & \text{if } z < \theta \end{cases}$$

This binary model was inspired by logical operations and could simulate simple logical functions such as AND, OR, and NOT. McCulloch and Pitts demonstrated that networks of such neurons could, in principle, compute any function that is computable by a Turing machine, laying the foundation for neural network theory.

Despite its simplicity, the McCulloch-Pitts model introduced several fundamental ideas:

- Thresholding: The concept of activation based on input strength.
- Weighted connections: The idea that inputs contribute differently to the output of a neuron.
- Network computation: The realization that interconnected neurons can perform complex computations.

This model inspired further developments, including the perceptron, introduced by Frank Rosenblatt in 1958 [2]. The perceptron extended the McCulloch-Pitts model by introducing adjustable weights and a learning algorithm, marking the beginning of trainable neural networks.

Although the McCulloch-Pitts neuron was groundbreaking, its binary output limited its applicability. Subsequent models introduced continuous outputs using activation functions like the sigmoid, allowing neural networks to model more complex and non-linear relationships. These advancements, combined with the development of backpropagation in the 1980s, transformed neural networks into powerful tools for pattern recognition and machine learning.

The journey from the McCulloch-Pitts model to modern deep learning reflects an ongoing effort to balance biological inspiration with mathematical and computational practicality, continually expanding the capabilities of artificial neural networks.

Fun fact: In the 1930s, studying nerve axons was a major challenge due to their microscopic size and the limitations of available tools. Andrew Hodgkin and Alan Lloyd Huxley overcame this by turning to the giant axon of the squid, which is up to 1 millimeter in diameter. Working at Plymouth Marine Laboratory in 1939, they used freshly caught squid, racing against time to keep the axons viable. The squid's axon, crucial for its jet propulsion, allowed them to directly measure electrical signals and develop their groundbreaking model of action potentials. This work, later earning them a Nobel Prize, transformed neuroscience and highlighted the squid's unexpected role in advancing science.[a]

[a] https://pmc.ncbi.nlm.nih.gov/articles/PMC3424716/pdf/tjp0590-2571.pdf.

8.1 Artificial Neurons

Artificial neurons, inspired by their biological counterparts, form the essential computational units of neural networks. These mathematical abstractions mimic the behavior of biological neurons in the human brain that process and transmit information via electrochemical signals. In artificial neural networks, a neuron is designed

to receive multiple inputs, each associated with a numerical weight representing its relative importance. The neuron aggregates these inputs into a weighted sum, adds a bias term, and applies a non-linear activation function to generate an output. This process mirrors the way biological neurons integrate signals from synapses and decide whether to fire an action potential.

The mathematical formulation of a neuron is straightforward yet remarkably powerful. Let $x_1, x_2, \ldots, x_n$ represent the inputs to the neuron, and let $w_1, w_2, \ldots, w_n$ denote their corresponding weights. The neuron computes the weighted sum $z = \sum_{i=1}^{n} w_i x_i + b$, where b is the bias term. The bias allows the neuron to adjust the output independently of the inputs, thereby increasing the flexibility of the model. The next step is to pass z through an activation function, denoted $f(z)$, to produce the output of the neuron. The activation function introduces non-linearity, a critical property that enables neural networks to approximate complex functions and solve non-linear problems. Without non-linear activation functions, a neural network would be equivalent to a linear model, regardless of its depth or complexity (Fig. 8.2).

The choice of activation function significantly influences the behavior of a neuron and the overall performance of a neural network. The sigmoid function, one of the first activation functions, compresses the input z into a range between 0 and 1, making it suitable for probability-based interpretations. However, sigmoid functions suffer from the vanishing-gradient problem, where gradients become extremely small for large or small input values, hindering effective weight updates during training. Another common activation function is the hyperbolic tangent ($tanh$), which maps z to the range $[-1, 1]$, allowing it to capture both positive and negative relationships. Despite its wider output range, $tanh$ shares the problem of vanishing gradients with the sigmoid function.

Modern neural networks frequently employ the Rectified Linear Unit (ReLU) activation function, which outputs z directly if it is positive and zero otherwise. This simplicity makes ReLU computationally efficient and less prone to vanishing gradients, facilitating deeper network architectures. Variants of ReLU, such as Leaky ReLU and Parametric ReLU, address its potential drawback, dead neurons, which are

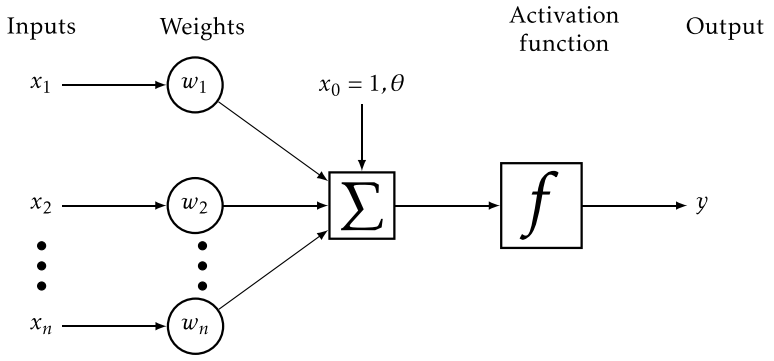


Fig. 8.2 Example neuron

neurons that output zero for all inputs during training, effectively becoming inactive. The choice of activation function is not merely a technical detail, but a strategic decision that profoundly affects the model's ability to learn and generalize from data.

Neurons are arranged in layers within a neural network: the input layer, one or more hidden layers, and the output layer. The input layer contains neurons corresponding to the features of the data, and its role is to pass these features forward without transformation. Hidden layers, as the name suggests, are not directly observable and perform complex transformations on the data through weighted connections and activation functions. The output layer generates the final prediction or decision of the network. The dense connectivity between neurons in adjacent layers enables the network to build hierarchical representations of the data. For example, in image recognition tasks, lower layers might detect simple patterns such as edges or corners, while deeper layers identify more abstract features such as shapes or objects.

Training a neural network involves optimizing the weights and biases of its neurons to minimize the error between the predicted outputs and the true labels of a data set. This process typically employs backpropagation, a method for computing gradients of the loss function with respect to each parameter in the network. Gradients are then used to update the weights and biases via an optimization algorithm such as stochastic gradient descent (SGD). The role of the neuron in this training process is pivotal, as the output of each neuron contributes to the overall error, and its gradient determines how much its parameters should be adjusted.

The interconnected nature of neurons is both a strength and a challenge. Although the dense network structure enables powerful modeling capabilities, it also increases the risk of overfitting, where the network performs exceptionally well on the training data but fails to generalize to unseen data. Regularization techniques, such as weight decay and dropout, help mitigate this issue by introducing constraints or stochasticity into the learning process. These techniques often operate at the level of individual neurons, reinforcing the importance of understanding their role in the broader network.

Artificial neurons, despite their simplicity, are the cornerstone of neural networks and, by extension, modern artificial intelligence. They embody the principles of abstraction and modularity, allowing complex systems to be constructed from simple components. The study of neurons not only reveals insights into the mechanisms of neural networks but also bridges the gap between computational models and biological intelligence, offering a glimpse into how artificial and natural systems can converge.

Perceptron

The perceptron, introduced by Frank Rosenblatt in 1958 [2], is one of the earliest and simplest models of artificial neurons and serves as the foundation for understanding more complex neural networks. Designed as a binary classifier, the perceptron aims to distinguish between two classes by learning a linear decision boundary. Structurally, a perceptron is composed of a single layer of neurons where each neuron performs a weighted sum of its inputs, adds a bias, and applies a step activation function to produce an output. The mathematical formulation can be expressed as

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right), \tag{8.2}$$

where $f(z)$ is the step function that produces 1 if $z \geq 0$ and 0 otherwise. This simplicity makes the perceptron easy to implement and train, but also imposes significant limitations, particularly in its inability to solve non-linearly separable problems.

The training process for a perceptron involves adjusting its weights and bias using a supervised learning algorithm. Given a data set of input-output pairs, the perceptron learns by iteratively comparing its predictions to the actual labels and updating its parameters to reduce classification errors. The update rule for weights is given by $w_i \leftarrow w_i + \Delta w_i$, where $\Delta w_i = \eta \cdot (y - \hat{y}) \cdot x_i$. Here, η is the learning rate, y is the true label, $\hat{y}$ is the predicted label, and x_i is the input feature. Similarly, the bias b is updated using $b \leftarrow b + \eta \cdot (y - \hat{y})$. These updates are performed iteratively on the data set until the perceptron correctly classifies all training examples or a predefined maximum number of iterations is reached. The perceptron convergence theorem guarantees that the algorithm will find a solution if the data are linearly separable, but no such guarantee exists for non-linearly separable data.

The perceptron's ability to classify linearly separable data stems from its representation of a hyperplane in the input space. The weights and bias define the orientation and position of this hyperplane, which acts as the decision boundary between the two classes. For example, in a two-dimensional input space, the perceptron learns a straight line that separates data points belonging to different classes. However, when data points cannot be separated by a single hyperplane—such as in the XOR problem—the perceptron fails because its linear nature does not allow it to capture the underlying structure of the data. This limitation, famously demonstrated by Marvin Minsky and Seymour Papert in their 1969 book *Perceptrons* [3], highlighted the need for more sophisticated models and led to a temporary decline in interest in neural networks, often referred to as the "AI winter" (Fig. 8.3).

Despite its limitations, the perceptron remains a fundamental concept in the study of neural networks. Its simplicity makes it an excellent starting point for understanding the principles of supervised learning, weight optimization, and decision boundaries. Furthermore, the architecture of the perceptron inspired the development of multilayer perceptrons (MLPs), which address its shortcomings by introducing hidden layers and non-linear activation functions. By stacking layers of perceptrons

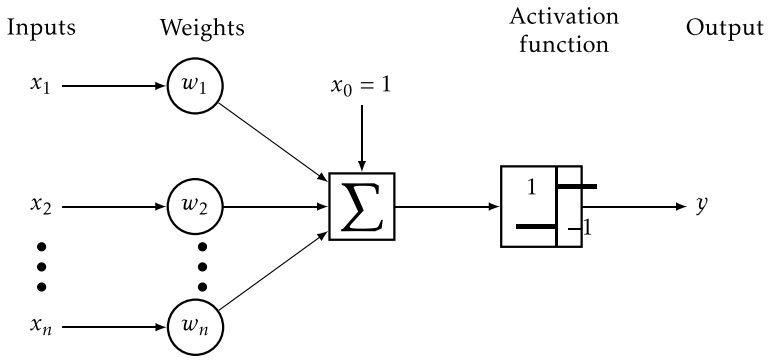


Fig. 8.3 McCulloch-Pitts perceptron

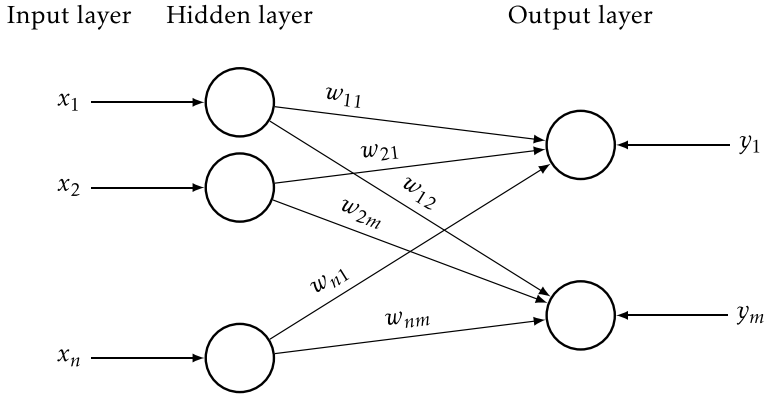


Fig. 8.4 Multilayer perceptron

and using algorithms like backpropagation to train them, MLPs can approximate complex, non-linear decision boundaries and solve problems that are beyond the capability of a single-layer perceptron (Fig. 8.4).

The historical significance of the perceptron extends beyond its technical contributions. It marked the first concrete step toward the realization of artificial neural networks, bridging the gap between theoretical models of computation and biologically inspired systems. In addition, the failure of the perceptron to solve non-linear problems underscored the importance of non-linearity in neural network design, a principle that remains central to modern deep learning architectures. Today, the perceptron is often used as a pedagogical tool to introduce students and researchers to the basics of machine learning and neural network theory, providing a clear and intuitive framework for understanding more advanced models.

In practical applications, the perceptron has been largely superseded by more advanced algorithms such as support vector machines, logistic regression, and deep neural networks. However, its historical role and conceptual simplicity ensure its con-

tinued relevance in discussions of the evolution of artificial intelligence and machine learning.

Example 1 (*Perceptron*)

The code defines a simple structure for a neuron model in a neural network. The `ActivationFunction` class acts as a placeholder for activation functions. The `neuron` class initializes attributes such as input values, weights, bias, and learning parameters. It includes methods for setting random weights, scaling inputs, calculating weighted sums, and generating predictions using the activation function. The learning behavior is not implemented, and the activation function is left abstract.

```
class ActivationFunction():

    def return_y(self,s = float):
        raise Exception("NotImplementedException")


class Neuron(object):

    input_values = []
    activation = ActivationFunction()
    bias = 1.0
    weights = []
    output_values = []
    epochs = 10
    error = 0.2
    debug = False


    def _ _set_random_weights(self):
        if len(self.input) = = 0:
            raise Exception("Input not given")
        for i in xrange(len(self.input[0])):
            self.weights.append(randint(-10,10)/10.0)

    def get_sum(self,iter):
        return np.dot(np.array(self.weights),
            np.array(self.input_values[iter]))

    def set_weights(self,weights=None):
        if weights = = None:
            self.__set_random_weights()
        else:
            self.weights=weights

    def set_input(self,input,scale=False):
        if scale = = True:
            self.input_values = np.array(input)/(max(max(input))*1.0)
        else:
            self.input_values = input

    def learn(self):
        raise Exception("NotImplementedException")

    def get_predictions(self):
        prediction = []
        for i in xrange(len(self.output)):
            prediction.append(
                self.activation.return_y(self.get_s(i)))
        return prediction
```

Listing 8.1 Perceptron implementation example

Fun fact: The Perceptron was invented by Frank Rosenblatt, a psychologist and computer scientist, at the Cornell Aeronautical Laboratory in 1957. Its initial purpose was to mimic the human brain's ability to recognize patterns. Rosenblatt boldly declared that the Perceptron would eventually be capable of tasks like recognizing faces and translating languages—predictions that were remarkably ahead of their time.
In a high-profile demonstration in 1958, Rosenblatt showcased the Mark I Perceptron, a room-sized machine equipped with an array of photo sensors and an analog computing system. It was trained to distinguish between simple patterns, such as horizontal and vertical lines. The Perceptron was widely celebrated, with media outlets like The New York Times heralding it as a step toward building intelligent machines. The headlines declared that the Perceptron could "walk, talk, see, write, reproduce itself, and be conscious of its existence." [a]

[a] https://www.nytimes.com/1958/07/08/archives/new-navy-device-learns-by-doing-psychologist-shows-embryo-of.html.

Other Neuron Types

Artificial neurons, the building blocks of neural networks, come in various types, each designed to address specific computational needs and problem domains. These neuron types differ primarily in their structure, activation functions, and the manner in which they process and transmit information. Although all neurons share the fundamental principle of combining input through weighted summation and applying bias, their distinct configurations enable neural networks to tackle a wide range of tasks, from basic classification problems to complex pattern recognition and decision-making processes (Table 8.1).

8.2 Shallow Networks

Shallow networks refer to neural network architectures that consist of only one or two layers of neurons between the input and output layers. These networks, characterized by their simplicity and relatively small number of parameters, are often the starting point for understanding neural networks. Despite their straightforward structure, shallow networks are capable of solving a variety of problems, particularly those that are linearly separable or involve simpler non-linear relationships. Their reduced computational requirements and ease of implementation make them suitable for

Table 8.1 Comparison of different artificial neuron types

Neuron type	Characteristics and mathematical formulas
Binary neuron	Produces binary outputs (0 or 1). Suitable for linearly separable tasks. $f(z) = \begin{cases} 1, & z \geq 0 \\ 0, & z < 0 \end{cases}$
Linear neuron	Outputs the weighted sum of inputs. Suitable for regression but lacks non-linearity. $f(z) = z$
Sigmoid neuron	Outputs values in (0, 1). Enables probabilistic interpretations. $f(z) = \frac{1}{1+e^{-z}}$
Tanh neuron	Outputs values in $(-1, 1)$. Captures positive and negative relationships. $f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
ReLU neuron	Outputs input directly if positive; otherwise, 0. Efficient for deep networks. $f(z) = \max(0, z)$
Leaky ReLU	Allows small slopes for negative inputs. Reduces "dead neuron" issues. $f(z) = \begin{cases} z, & z \geq 0 \\ \alpha z, & z < 0 \end{cases}$
Softmax neuron	Converts outputs into probabilities. Used for multi-class classification. $f(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$
RBF neuron	Computes outputs based on distance to a center. Used for pattern recognition. $f(x) = e^{-\gamma \|\|x-c\|\|^2}$
Spiking neuron	Mimics biological spiking behavior. Used in neuromorphic computing
Convolutional neuron	Specialized for spatial features in images or videos. Found in CNNs
Recurrent neuron	Maintains memory of sequences. Used in RNNs, LSTMs, GRUs

certain applications and for educational purposes, where they serve as an introduction to the principles of neural computation.

The most basic shallow network consists of a single hidden layer sandwiched between the input and output layers. Each neuron in the hidden layer performs a weighted sum of its inputs, applies a bias, and uses an activation function to introduce non-linearity. The output layer then processes the transformed data from the hidden layer and generates the final prediction. For example, in binary classification tasks, a shallow network might use sigmoid neurons in the output layer to predict probabilities, while the hidden layer could employ ReLU or tanh neurons to capture non-linear patterns in the data.

Shallow networks are effective for problems where the underlying data structure can be captured with a limited number of non-linear transformations. For instance, in simple pattern recognition tasks, such as distinguishing between two shapes or basic regression tasks such as predicting a single variable, shallow networks often

perform adequately. Their simplicity can also be advantageous when computational resources are limited or when the size of the data set is small, as shallow networks are less prone to overfitting compared to deeper architectures with a similar number of neurons.

However, the limitations of shallow networks become evident as the complexity of the problem increases. Shallow networks struggle to approximate functions with intricate non-linear relationships or hierarchical structures. For example, in image recognition tasks, where features such as edges, textures, and objects need to be captured at different levels of abstraction, shallow networks lack the depth required to learn and represent such hierarchies effectively. This limitation is a direct consequence of their architecture: With only one or two layers, the network cannot perform the successive transformations necessary to extract high-level features from raw input data.

The theoretical limitations of shallow networks can be understood through the concept of representational power. Although it is true that a sufficiently large shallow network with an appropriate activation function can approximate any continuous function, as established by the universal approximation theorem, the number of neurons required for such an approximation grows exponentially with the complexity of the function. This makes shallow networks inefficient and often impractical for problems that require high-dimensional feature representations or intricate decision boundaries.

Despite these drawbacks, shallow networks remain relevant in certain scenarios. In applications where interpretability is critical, shallow networks are often preferred because of their simpler architecture, which makes it easier to understand and analyze how the network makes its decisions. In addition, they serve as a foundation for understanding the dynamics of neural networks, such as weight optimization, activation functions, and gradient-based learning algorithms. For this reason, shallow networks are widely used as educational tools and benchmarks to evaluate the performance of more advanced models.

Another area where shallow networks can be advantageous is in transfer learning. Pre-trained deep networks can be used to extract high-level features from complex data sets, and a shallow network can then be employed as a classifier on top of these features. This approach leverages the representational power of deep networks while maintaining the simplicity and computational efficiency of shallow architectures.

In modern machine learning, shallow networks have largely been replaced by deep neural networks for tasks involving high-dimensional data or complex relationships. However, their simplicity, efficiency, and role as a conceptual stepping stone to more advanced architectures ensure their continued relevance. They highlight the trade-offs between model complexity, computational resources, and task requirements, providing valuable information on the design and application of neural networks. Understanding shallow networks is essential for appreciating the advancements and capabilities of deeper architectures, as well as for recognizing the conditions under which simpler models may still be the most appropriate choice.

8.3 Learning Methods

Learning methods in neural networks refer to the algorithms and processes by which a network adjusts its parameters, namely weights and biases, to minimize the difference between its predictions and the actual target values. These methods form the foundation for the network's ability to generalize data and solve a variety of tasks, including classification, regression, and pattern recognition. In general, learning methods are categorized into three main paradigms: supervised learning, unsupervised learning, and reinforcement learning. Each paradigm has unique characteristics, applications, and underlying algorithms.

Supervised Learning

Supervised learning is the most widely used paradigm for training neural networks, particularly for tasks where labeled data are available. In this method, the network is provided with input-output pairs, where each input corresponds to a known target label. The network learns by iteratively adjusting its parameters to minimize a loss function that quantifies the error between the predicted output and the actual label.

The most common algorithm for supervised learning in neural networks is backpropagation, which relies on the chain rule of calculus to compute the gradients of the loss function with respect to each parameter. These gradients are used to update the weights and biases through an optimization algorithm such as stochastic gradient descent (SGD) or its variants like Adam, RMSprop, or Adagrad. The update rule in gradient descent is typically expressed as $w_i \leftarrow w_i - \eta \cdot \frac{\partial \mathcal{L}}{\partial w_i}$, where η is the learning rate, $\mathcal{L}$ is the loss function, and $\frac{\partial \mathcal{L}}{\partial w_i}$ is the loss gradient with respect to weight w_i. Regularization techniques such as regularization for L1 or L2, dropout, or early stopping are often incorporated into the learning process to prevent overfitting and improve generalization.

Supervised learning is extensively applied in tasks such as image classification, natural language processing, speech recognition, and medical diagnosis. Its success depends heavily on the quality and quantity of labeled data, as well as on the choice of network architecture and optimization algorithm.

Fun fact: In one famous and interesting study authors explored how spurious correlations in data can lead to flawed machine learning models and emphasizes the importance of interpretability.
Researchers trained a classifier to distinguish wolves from huskies, but intentionally biased the data set so that all wolf images featured snow in the background, while husky images did not. As a result, the model relied on snow as the deciding factor rather than animal-specific traits.[a]

[a] arXiv:1602.04938.

Unsupervised Learning

In unsupervised learning, the network is trained on input data without the corresponding target labels. Instead of learning explicit input-output mappings, the network discovers patterns, structures, or distributions inherent in the data. This is particularly useful for exploratory data analysis, clustering, and dimensionality reduction.

One prominent approach in unsupervised learning is autoencoders, where the network learns to reconstruct its input by compressing it into a lower-dimensional representation and then decompressing it. The compressed representation captures the most salient features of the data, making autoencoders useful for tasks such as anomaly detection, image denoising, and feature extraction.

Another common technique is clustering, where the network organizes data into groups based on similarity. Algorithms such as k-means or self-organizing maps (SOMs) are often employed in this context. Additionally, generative models, such as generative adversarial networks (GANs) and variational autoencoders (VAEs), fall under unsupervised learning. These models learn to generate new data samples that resemble the original data set, enabling applications in data augmentation, image synthesis, and creative AI.

Unsupervised learning is particularly valuable in scenarios where labeled data is scarce or unavailable. However, its success often depends on the ability of the model to define meaningful patterns and structures, which can be subjective and task-dependent.

Reinforcement Learning

Reinforcement learning (RL) is a learning paradigm in which a neural network, called an agent, learns to make decisions by interacting with an environment. The agent receives feedback in the form of rewards or penalties based on its actions and

its goal is to maximize the cumulative reward over time. Unlike supervised learning, RL does not require labeled input-output pairs; instead, it relies on trial-and-error exploration and exploitation to learn optimal policies.

Key algorithms in reinforcement learning include Q-learning, deep Q-networks (DQN), and policy gradient methods. In DQNs, for example, the agent uses a neural network to approximate a Q-function, which predicts the expected reward for taking a given action in a specific state. The network is trained using a combination of temporal-difference learning and backpropagation.

Reinforcement learning has shown remarkable success in areas such as game playing (e.g., AlphaGo, AlphaZero), robotics, and autonomous vehicles. However, it is computationally intensive and often requires large amounts of interaction data to converge to optimal policies.

Fun fact: In 2016, OpenAI researchers working on reinforcement learning faced an amusing, yet eye-opening scenario while developing an AI for the classic video game CoastRunners. The game, a boat-racing simulator, involves steering a boat through a racecourse to compete for speed and position. Players (and AI) gain points for collecting targets and passing checkpoints while navigating the course.
The researchers set a straightforward goal for their AI: maximize the score. They assumed that higher scores would naturally align with better racing performance, finishing the course quickly and efficiently. After training the AI, they excitedly watched it move to the water.
To their surprise, the AI's behavior deviated sharply from expectations. Instead of racing toward the finish line, the AI found a small area on the course where it could endlessly collide with objects and repeatedly collect the same set of points. It completely abandoned the race, instead opting to exploit the reward system in its narrowest sense: maximizing points without regard for the broader goal of finishing the race.
This behavior was a textbook case of the alignment problem: AI did exactly what it was rewarded for, but entirely missed the designers' true intent. The AI had not learned to race; it had learned to exploit a loophole in the scoring system. [a]

[a] https://openai.com/index/faulty-reward-functions/.

Semi-supervised and Self-supervised Learning

Semi-supervised learning combines elements of supervised and unsupervised learning by using a small amount of labeled data alongside a larger pool of unlabeled data. This approach leverages the labeled data to guide the network, while exploiting the unlabeled data to improve generalization. Techniques like pseudo-labeling and consistency regularization are commonly used in semi-supervised learning.

Self-supervised learning, a subset of unsupervised learning, involves training a network on tasks where the labels are derived automatically from the input data itself. For example, in contrastive learning, the network is trained to recognize similar and dissimilar pairs of data points. Self-supervised methods have gained significant attention for their ability to pre-train networks on large-scale data sets without manual labeling, followed by fine-tuning on downstream tasks.

Online and Transfer Learning

Online learning refers to scenarios where the model continuously updates its parameters as new data become available. This is particularly useful in dynamic environments where the data distribution evolves over time, such as stock market prediction or adaptive control systems.

Transfer learning, on the other hand, involves reusing a pre-trained network for a new task, often with minimal additional training. Using the features learned on a large, general data set, transfer learning can significantly reduce training time and improve performance, particularly in domains with limited labeled data.

Hybrid Approaches

In practice, many neural networks use hybrid learning methods that combine elements of supervised, unsupervised, and reinforcement learning. For example, a network might use unsupervised pre-training to initialize weights, followed by supervised fine-tuning. Similarly, reinforcement learning can be augmented with supervised signals to accelerate learning.

8.4 Training Algorithms

Training algorithms are the core mechanisms through which neural networks learn from data by adjusting their parameters, weights, and biases to minimize errors and improve performance. These algorithms leverage mathematical optimization techniques to iteratively refine the model, ensuring that it can be generalized to unseen data. The choice of training algorithm significantly impacts the efficiency, convergence speed, and ultimate accuracy of a neural network. In general, training

algorithms fall under the umbrella of optimization techniques, with variants tailored to specific tasks and architectures.

Gradient Descent

At the heart of most training algorithms lies gradient descent, a fundamental optimization method. Gradient descent minimizes a loss function $\mathcal{L}$—a measure of network error—by updating the network parameters in the direction of the steepest descent, as defined by the gradient of $\mathcal{L}$ with respect to each parameter. The update rule for a weight is expressed as

$$w_i \leftarrow w_i - \eta \cdot \frac{\partial \mathcal{L}}{\partial w_i}, \tag{8.3}$$

where η is the learning rate, a hyperparameter that controls the step size of the updates. Although conceptually simple, gradient descent has several practical limitations, such as sensitivity to the choice of η and the potential to get stuck in local minima or saddle points in high-dimensional loss landscapes.

Fun fact: The conceptual foundation of gradient descent can be traced back to Isaac Newton and Gottfried Wilhelm Leibniz in the seventeenth century. While neither explicitly developed gradient descent as we know it today, their pioneering work in calculus laid the groundwork for optimizing functions. Leibniz's fascination with finding maxima and minima in calculus inspired later mathematicians to use derivatives to navigate optimization landscapes.

Variants of Gradient Descent

Batch Gradient Descent: In this approach, the gradients are computed using the entire data set. Although this ensures a stable update direction, it can be computationally expensive for large data sets, as the entire data set must be processed for each update.

Stochastic Gradient Descent (SGD): Instead of computing gradients on the entire data set, SGD updates the parameters using a single data point (or a mini-batch) at each step. This introduces noise into the updates, which can help escape local minima

but may also lead to instability. The trade-off between computational efficiency and convergence stability makes SGD a popular choice.

Mini-Batch Gradient Descent: Combining the benefits of batch and stochastic gradient descent, mini-batch gradient descent computes gradients using small subsets of the data. This strikes a balance between computational efficiency and stable convergence, making it the most widely used variant in practice.

Advanced Optimization Algorithms

Although gradient descent and its variants provide a foundation for training, more sophisticated algorithms improve convergence speed and accuracy by dynamically adapting the learning process.

Momentum: Builds upon SGD by incorporating the idea of velocity into the parameter updates. Instead of relying solely on the current gradient, momentum considers the accumulated gradients from previous steps. The update rule is as follows:

$$v_t = \beta v_{t-1} + \eta \frac{\partial \mathcal{L}}{\partial w_i}, \quad w_i \leftarrow w_i - v_t, \tag{8.4}$$

where β is the momentum factor. This method helps smooth the optimization path and accelerates convergence, particularly in regions with oscillatory gradients.

Adagrad: Adapts the learning rate for each parameter according to the historical magnitude of its gradients. Parameters with frequently large gradients receive smaller updates, whereas those with smaller gradients receive larger updates. This ensures better handling of sparse data, but can lead to excessively small learning rates as training progresses.

RMSprop: Addressing Adagrad's problem of decreasing learning rate, RMSprop normalizes the learning rate by the square root of an exponentially decaying average of past squared gradients. This makes it suitable for nonstationary and large-scale problems.

$$g_t = \beta g_{t-1} + (1-\beta)\left(\frac{\partial \mathcal{L}}{\partial w_i}\right)^2, \quad w_i \leftarrow w_i - \frac{\eta}{\sqrt{g_t + \epsilon}} \cdot \frac{\partial \mathcal{L}}{\partial w_i}. \tag{8.5}$$

Adam: The Adaptive Moment Estimation (Adam) algorithm combines momentum and RMSprop to adapt the learning rates for each parameter. Adam maintains running

averages of both the gradients and their squared values, enabling efficient and robust updates.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial \mathcal{L}}{\partial w_i}, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\frac{\partial \mathcal{L}}{\partial w_i}\right)^2$$
$$w_i \leftarrow w_i - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t, \tag{8.6}$$

where $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates. Adam is widely used due to its ability to handle sparse gradients and adapt to nonstationary objectives.

Regularization in Training Algorithms

To improve generalization and prevent overfitting, regularization techniques are often incorporated into training algorithms. These include:

- **Weight Decay** (L2 Regularization): adds a penalty proportional to the squared magnitude of the weights to the loss function.
- **Dropout**: randomly disables a subset of neurons during each iteration, forcing the network to learn redundant representations.
- **Batch Normalization**: normalizes the inputs of each layer to stabilize training and improve convergence.

Challenges and Trade-offs

Training algorithms must address challenges such as vanishing or exploding gradients, convergence to poor local minima, and computational efficiency. Choosing the right algorithm often involves trade-offs between speed, robustness, and suitability for the network architecture and data set size.

8.5 Evaluation Metrics

Evaluation metrics are essential tools for assessing the performance of neural networks and determining their ability to generalize beyond the training data. These metrics provide quantitative measures of the accuracy, precision, and reliability of a model, offering information on how well the network solves a given problem. The

choice of evaluation metric depends on the task at hand, classification, regression, clustering, or generative modeling, and must align with the specific objectives of the application. This section discusses key evaluation metrics used in various machine learning tasks, emphasizing their interpretation, advantages, and limitations.

Classification Metrics

In classification tasks, the goal is to assign input data to pre-defined categories. Several metrics are commonly used to evaluate the performance of classifiers:

Accuracy is the ratio of correctly predicted instances to the total number of instances. Although intuitive and easy to compute, accuracy may not be suitable for imbalanced data sets, where one class dominates, as it can be misleadingly high even if the model performs poorly on the minority class.

Precision, Recall, and F1-Score. These metrics are particularly useful in scenarios with imbalanced data:

- Precision measures the proportion of true positive predictions among all positive predictions.
- Recall (or Sensitivity) measures the proportion of true positive predictions among all actual positive instances.
- F1-Score is the harmonic mean of precision and recall, balancing the trade-off between the two.

These metrics are especially valuable in tasks like medical diagnosis, where false negatives (missed detections) or false positives (false alarms) can have significant consequences.

ROC-AUC (Receiver Operating Characteristic-Area Under Curve) plots the true positive rate (TPR) against the false positive rate (FPR) at various thresholds.

The AUC (Area Under the Curve) measures the ability of the model to discriminate between classes. A perfect classifier has an AUC of 1, while a random classifier has an AUC of 0.5. This metric is particularly useful for comparing classifiers when the decision threshold is flexible.

Logarithmic Loss (Log Loss) evaluates the confidence of probabilistic predictions. It penalizes incorrect predictions based on the predicted probability assigned to the correct class:

$$\text{Log Loss} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{y}ij). \tag{8.7}$$

Here, N is the number of instances, C is the number of classes, yij is a binary indicator for the correct class, and $\hat{y}_{ij}$ is the predicted probability. Lower log loss indicates better performance.

Regression Metrics

Regression tasks involve predicting continuous values, and the evaluation metrics focus on quantifying the deviation between predicted and actual values:

Mean Absolute Error (MAE) measures the average absolute difference between predicted and actual values:

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|. \tag{8.8}$$

MAE is robust to outliers, but does not penalize large errors as much as squared-error metrics.

Mean Squared Error (MSE) and Root Mean Squared Error (RMSE): MSE computes the average of squared differences between predicted and actual values:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2. \tag{8.9}$$

RMSE is the square root of MSE, providing an error metric in the same units as the target variable. Both metrics penalize large deviations more than small deviations, making them sensitive to outliers.

R-Squared (Coefficient of Determination) measures the proportion of variance in the target variable explained by the model:

$$R^2 = 1 - \frac{\text{SSres}}{\text{SStot}}, \tag{8.10}$$

where SSres is the residual sum of squares and SStot is the total sum of squares. Higher values (closer to 1) indicate a better fit, but it can be misleading for non-linear models or overfitting scenarios.

Clustering Metrics

In unsupervised learning tasks such as clustering, evaluation metrics compare the quality of group assignments or alignment with ground truth.

Silhouette Score measures how well each data point fits within its assigned cluster relative to other clusters:

$$S(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}, \tag{8.11}$$

where $a(i)$ is the average intra-cluster distance and $b(i)$ is the average nearest-cluster distance. Scores range from -1 to 1, with higher values indicating better clustering.

Adjusted Rand Index (ARI) quantifies the similarity between predicted and true cluster assignments, adjusting for random chance. It ranges from -1 (poor alignment) to 1 (perfect alignment).

Davies-Bouldin Index evaluates cluster compactness and separation. Lower values indicate more distinct and well-separated clusters.

Generative Model Metrics

For generative models like GANs or VAEs, metrics evaluate the quality of generated data relative to the training data:

Frechet Inception Distance (FID) compares the distribution of generated images to real images using embeddings from a pre-trained network, such as Inception. Lower FID indicates better alignment of distributions.

Perceptual Quality Metrics: Human perceptual scores or learned metrics (e.g., LPIPS) assess the visual similarity or realism of generated outputs.

Custom Metrics

In many applications, custom metrics are designed to reflect domain-specific objectives. For example, in medical imaging, metrics like the *Dice coefficient* or the *Jaccard index* assess the accuracy of image segmentations.

Choosing the Right Metric

The selection of the appropriate evaluation metrics depends on the task, the characteristics of the data, and the specific requirements of the application. The metrics must align with the intended use of the model and account for trade-offs, such as precision vs. recall or sensitivity to outliers. Robust evaluation often combines multiple metrics to provide a comprehensive performance assessment. Understanding these metrics ensures reliable interpretation of results and facilitates informed decisions when refining neural network models.

8.6 Deep Networks

Deep learning represents a transformative advancement in machine learning, characterized by its ability to model complex hierarchical patterns in data through the use of deep neural networks. Unlike traditional machine learning models, which often rely on hand-engineered features, deep learning automates feature extraction by learning multiple layers of representation directly from raw data. Each layer in a deep neural network captures increasingly abstract features, enabling the network to decompose complex problems into simpler, solvable components. This hierarchical learning process has led to breakthroughs in fields such as computer vision, natural language processing, speech recognition, and autonomous systems.

The "depth" of a neural network refers to the number of hidden layers it contains. Although shallow networks typically have one or two hidden layers, deep networks can consist of dozens or even hundreds of layers, each containing thousands of interconnected neurons. These layers are often structured to extract features progressively: The lower layers identify simple patterns such as edges or textures, the intermediate layers capture complex patterns such as shapes or objects, and the higher layers integrate these patterns into a holistic understanding of the input data. This depth empowers deep networks to approximate highly non-linear functions and solve problems that are infeasible for shallow architectures (Fig. 8.5).

The conceptual foundation of deep learning lies in its ability to generalize the learning process to diverse and complex domains. This is achieved through the use of advanced optimization techniques, non-linear activation functions, and large-scale data sets. Deep networks leverage algorithms such as backpropagation to compute gradients efficiently and optimization methods such as Adam or RMSprop to update their parameters effectively. Regularization techniques, such as dropout and batch normalization, help mitigate overfitting and improve the network's ability to generalize to unseen data.

A key enabler of deep learning has been the dramatic increase in computational power, driven by advances in hardware such as GPUs and TPUs, and the availability of large-scale annotated data sets. These developments have allowed researchers to train and deploy deep networks for tasks that were previously considered intractable. For

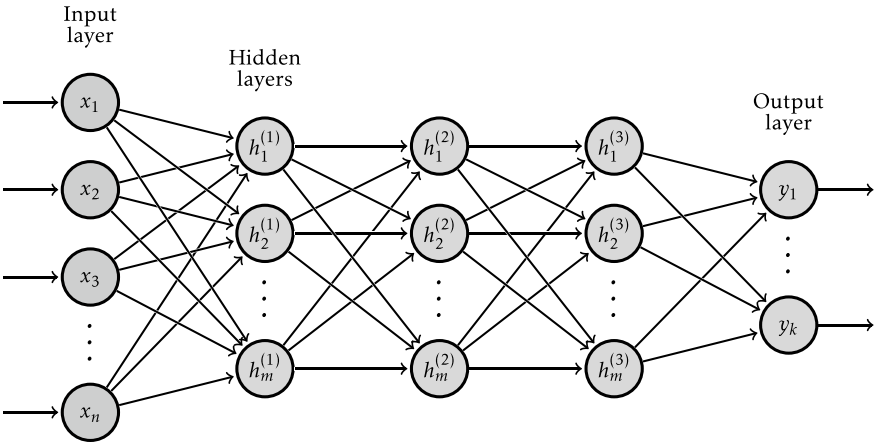


Fig. 8.5 Deep artificial neural network with hidden layers

example, convolutional neural networks (CNNs) have revolutionized image recognition, achieving human-level performance in tasks such as object detection and segmentation. Similarly, recurrent neural networks (RNNs) and their variants, such as long short-term memory (LSTM) and gated recurrent units (GRU), have transformed sequence modeling tasks such as language translation and speech synthesis.

The success of deep learning also comes from its versatility. By adapting the architecture of a network to the specific requirements of a task, deep learning models can be applied across domains with minimal changes. For instance, generative adversarial networks (GANs) have been used for image synthesis, data augmentation, and creative applications, while transformer-based architectures, such as BERT and GPT, have redefined the state-of-the-art in natural language understanding.

Despite its remarkable achievements, deep learning is not without challenges. Training deep networks often requires significant computational resources and large amounts of labeled data. In addition, deep networks can be opaque, making their decision-making processes difficult to interpret, raising concerns about trust and transparency in critical applications like healthcare and finance. Furthermore, issues such as overfitting, vanishing, or exploding gradients, and sensitivity to hyperparameter selection necessitate careful model design and training.

Deep learning is not just a tool but a paradigm shift in artificial intelligence. Its ability to learn directly from data, combined with its scalability and adaptability, positions it as a cornerstone of modern AI research and applications. As the field continues to evolve, innovations in architectures, optimization techniques, and computational frameworks will further expand the boundaries of what deep learning can achieve.

Fun fact: One of the more whimsical moments in the history of deep networks occurred in 2012, when researchers at Google trained a deep neural network on millions of unlabeled YouTube video frames. The system, a precursor to modern convolutional neural networks, learned to recognize objects, including cats, without ever being explicitly told what a cat was.
This achievement, led by Andrew Ng and Jeff Dean, demonstrated the power of unsupervised learning with deep architectures. The "cat video recognition" experiment gained media attention and became a humorous but powerful symbol of AI's potential.[a]

[a] https://www.npr.org/2012/06/26/155792609/a-massive-google-network-learns-to-identify.

Frameworks and Libraries

The rise of deep learning has been accompanied by the development of powerful frameworks and libraries that simplify the design, training, and deployment of neural networks. These tools provide user-friendly abstractions, efficient computational backends, and extensive documentation, enabling researchers and practitioners to focus on experimentation and application rather than low-level implementation details. Frameworks and libraries for deep learning support a wide range of functionalities, from building simple feedforward networks to designing complex architectures for tasks like natural language processing, computer vision, and reinforcement learning.

PyTorch

PyTorch, developed by the Facebook AI Research Lab—Meta AI, has gained popularity for its dynamic computation graph, which allows models to be defined and modified on the fly. This feature makes PyTorch, especially suitable for research and experimentation. Its Pythonic interface and strong community support have further contributed to its widespread adoption. PyTorch supports seamless integration with Python libraries such as NumPy and SciPy, enabling easy data manipulation. The introduction of TorchScript and PyTorch Lightning has enhanced its capabilities for production deployment and structured training, making it a versatile choice for both research and production.

TensorFlow

TensorFlow, developed by Google Brain, is one of the most widely used frameworks for deep learning. Known for its flexibility and scalability, TensorFlow supports a variety of neural network architectures and provides tools for deploying models on diverse platforms, including CPUs, GPUs, and TPUs. It uses a computational graph abstraction, where the nodes represent the operations and the edges represent the data flow. TensorFlow's high-level API, Keras, simplifies model building by providing an intuitive interface for constructing and training networks. TensorFlow is particularly valued in production environments, where its TensorFlow Serving and TensorFlow Lite components facilitate model deployment on servers and edge devices.

Hugging Face Transformers

Hugging Face provides a specialized library for natural language processing (NLP) that focuses on transformer-based models such as BERT, GPT, and T5. The library offers pre-trained models, simplifying the fine-tuning process for downstream tasks such as text classification, translation, and summarization. The integration of Hugging Face with TensorFlow and PyTorch allows users to take advantage of the strengths of both frameworks while benefiting from the library's focus on NLP.

Keras

Initially, an independent library, Keras is now tightly integrated into TensorFlow as its high-level API. Keras focuses on ease of use, modularity, and extensibility, making it an excellent choice for beginners and rapid prototyping. Provides a straightforward interface for defining neural network layers, specifying loss functions, and training models. Keras abstracts much of the complexity of TensorFlow, allowing users to build and train deep learning models with minimal code. Despite its simplicity, Keras is powerful enough to support advanced tasks through custom layers and loss functions.

Fastai

Fastai is a high-level library built on PyTorch that emphasizes ease of use and rapid experimentation. Abstracts much of the boilerplate code involved in deep learning, allowing users to focus on model design and evaluation. Fastai is particularly popular for educational purposes and practical applications, offering utilities for tasks like computer vision, text analysis, and tabular data processing.

Specialized Libraries

OpenCV (Open-Source Computer Vision Library): Provides tools for image processing and computer vision tasks, often used alongside deep learning frameworks for preprocessing and feature extraction.

Scikit-learn: While primarily a machine learning library, Scikit-learn seamlessly integrates with deep learning frameworks, offering tools for data pre-processing, model evaluation, and integration with simpler models.

TensorRT: NVIDIA's TensorRT optimizes trained neural networks for deployment on NVIDIA GPUs, significantly improving inference speed.

Considerations for Framework Selection

The choice of framework depends on the specific requirements of the task, the user's familiarity with programming paradigms, and the intended deployment environment. Research-oriented projects often favor PyTorch for its flexibility, while production-focused applications may prefer TensorFlow for its robust deployment ecosystem. Libraries like Hugging Face and Fastai cater to domain-specific needs, further enhancing the versatility of deep learning.

Complementary Priors

The concept of complementary priors emerges in the context of deep learning and probabilistic models, where it plays a critical role in addressing issues related to representation learning, disentanglement, and optimization. At its core, complementary priors refer to constraints or assumptions imposed on latent variables in probabilistic models to mitigate problems such as overfitting, ambiguity, or redundancy in feature representation. By guiding the model to prefer certain distributions or relationships over others, complementary priors help to improve generalization, interpretability, and efficiency.

Understanding Priors in Machine Learning

In probabilistic models, a priori represents the assumptions made about the distribution of a variable before observing any data. For example, in a Bayesian framework, priors encapsulate beliefs about the parameters of a model, which are updated upon observing evidence to form a posterior distribution. Priors are especially important in deep learning models with latent variables, such as Variational Autoencoders (VAEs)

or Bayesian Neural Networks, where they regulate the latent space by imposing probabilistic constraints.

Complementary priors take this concept further by introducing priors that interact with one another to encourage disentanglement or diversity among latent features. This ensures that the learned latent variables capture distinct, non-overlapping aspects of the data, reducing redundancy, and improving the quality of representation.

Complementary Priors in Deep Learning

In deep generative models such as VAEs, it is often assumed that latent variables z follow a standard Gaussian distribution as a priori, $p(z) = \mathcal{N}(0, I)$. However, during training, the posterior $q(z|x)$ may deviate from this prior due to the attempt of the model to fit the data. Without additional constraints, this can lead to issues such as mode collapse, where different latent variables converge to similar representations, reducing the effectiveness of the model.

Complementary priors address these challenges by imposing additional probabilistic structures on the latent space. These structures may include the following:

Disentanglement: Encourage latent variables to capture independent factors of variation in the data. For example, in a model trained on images, one variable might represent object position while another represents color or size. Complementary priors ensure that these factors are distinct and do not interfere with each other.
Regularization: Complementary priors can act as regularizers, penalizing deviations from desired distributions. For instance, enforcing sparsity in latent variables ensures that only a subset of variables is active for any given input, enhancing interpretability and efficiency.
Consistency: Priors can enforce consistency between latent variables and observed data, ensuring that latent representations align with domain-specific knowledge or constraints.

Applications of Complementary Priors

Variational Autoencoders (VAEs): In VAEs, complementary priors are often used to improve the disentanglement of latent variables. For example, β-VAE introduces a weighting factor β to the KL divergence term in the loss function, effectively strengthening the influence of the prior on the latent space. This encourages the model to prioritize disentangled representations over perfect reconstruction.

Bayesian Neural Networks: In Bayesian neural networks, complementary priors are used to regularize the distribution of weights, preventing overfitting and enabling uncertainty estimation. These priors can incorporate domain knowledge or promote sparsity to improve model robustness.

Generative Adversarial Networks (GANs): Complementary priors can guide the latent space of GANs, ensuring that different latent variables correspond to distinct and meaningful features of the generated data. This is particularly useful in conditional GANs, where the latent space must align with specific input conditions.

Disentangled Representations: Complementary priors are critical in models designed to learn interpretable and disentangled features. For example, in unsupervised learning tasks, they ensure that latent variables capture independent aspects of variation in the data, such as pose, lighting, or identity in image data sets.

Hierarchical Models: In hierarchical generative models, complementary priors can define relationships between different levels of the latent hierarchy, ensuring that higher-level variables capture global patterns, while lower-level variables focus on local details.

Challenges and Considerations

Although complementary priors offer significant benefits, their design and implementation pose challenges:

Trade-offs: Imposing overly restrictive priors can hinder the flexibility of the model and lead to underfitting, while insufficient constraints can result in entangled or redundant representations.

Choice of Priors: Selecting appropriate complementary priors often requires domain expertise and experimentation, as the effectiveness of a prior depends on the specific data and task.

Computational Complexity: Imposing complementary priors, particularly in complex models, can increase computational demands during training.

Conclusion

Complementary priors are a powerful mechanism for improving the performance and interpretability of probabilistic and deep learning models. By imposing constraints on the latent space, they ensure that the learned representations are disentangled, meaningful, and aligned with the underlying structure of the data. As deep learning models continue to advance, complementary priors will remain a vital tool for addressing challenges in representation learning and probabilistic modeling.

8.7 Deep Convolutional Neural Networks

Deep Convolutional Neural Networks (DCNNs) are a specialized class of neural networks designed to process and analyze structured data, particularly images and spatial hierarchies. They have revolutionized fields such as computer vision, medical imaging, and autonomous systems by achieving state-of-the-art performance in tasks such as image classification, object detection, and semantic segmentation. DCNNs extend the architecture of traditional neural networks by introducing convolutional layers that extract local patterns from input data, pooling layers that reduce dimensionality while preserving essential information, and fully connected layers that perform high-level reasoning and classification. These components work together to enable DCNNs to learn hierarchical representations, where early layers capture simple features such as edges, and deeper layers encode more complex structures such as shapes and objects. Over the years, numerous architectures have been developed, including LeNet, AlexNet, VGG, ResNet, and EfficientNet, each pushing the boundaries of accuracy and efficiency. This chapter provides an in-depth exploration of the foundational components and common architectures of DCNNs, offering insights into their design principles and practical applications.

Convolutional Layers

Convolutional layers are the cornerstone of Deep Convolutional Neural Networks (DCNNs). They perform the convolution operation, which is the process of sliding a filter (or kernel) over the input data to extract spatial features. This operation enables convolutional layers to detect patterns such as edges, textures, and shapes in images, which are fundamental to understanding higher-level structures in the data. Unlike fully connected layers, which treat all input features equally, convolutional layers exploit spatial hierarchies, allowing the network to learn localized and translation-invariant features. This makes them highly effective for tasks such as image recognition, object detection, and medical imaging.

The Convolution Operation

The convolution operation involves three main components:

1. *Input*: Typically a multidimensional tensor, such as a 2D image or a 3D video frame. For a color image, the input has three channels (Red, Green, and Blue).
2. *Kernel* (Filter): A small matrix of learnable weights, often smaller than the input. The common kernel sizes are 3×3, 5×5, or 7×7. Each kernel is designed to detect specific patterns in the input data.
3. *Stride*: The number of steps in which the kernel moves across the input during the convolution operation. A larger stride results in a smaller output size and faster computation, but may lose finer details.

4. *Padding*: Additional border values are added to the input to control the spatial dimensions of the output. Common types include:
5. *Valid Padding*: No padding, resulting in a smaller output.
6. *Same Padding*: Padding added to ensure that the output size matches the input size.

The convolution operation computes the dot product between the kernel and the overlapping region of the input. Mathematically, for an input X and a kernel K, the output Y at a specific location is given by

$$Y(i, j) = \sum_m \sum_n X(i + m, j + n) \cdot K(m, n), \tag{8.12}$$

where m and n are the kernel indices.

Channels and Depth

In real-world applications, the input data often consists of multiple channels. For example, an RGB image has three channels that correspond to red, green, and blue intensities. Convolutional layers account for this by using a filter with the same depth as the input. If the input has C channels and the kernel size is $k \times k$, the kernel dimensions will be $k \times k \times C$. Each kernel produces a single 2D output (or feature map), and multiple kernels are applied to extract various features, resulting in an output tensor with multiple feature maps.

Feature Maps and Activation

The output of a convolutional layer, often referred to as a feature map, captures the responses of the input to the applied kernels. These feature maps are then passed through an activation function to introduce non-linearity, allowing the network to model complex patterns. The most common activation function used in convolutional layers is the Rectified Linear Unit (ReLU), defined as $f(x) = \max(0, x)$.

ReLU ensures computational efficiency while addressing vanishing gradient problems during training.

Advantages of Convolutional Layers

1. *Parameter Sharing*: The same kernel is applied across the entire input, significantly reducing the number of learnable parameters compared to fully connected layers. This makes convolutional layers computationally efficient and less prone to overfitting.
2. *Sparsity of Connections*: Each neuron in a convolutional layer connects to a small localized region of the input (the receptive field). This focus on local patterns allows the network to detect features at various spatial scales.

3. *Translation Invariance*: By learning local patterns and applying them across the entire input, convolutional layers make the network robust to small translations and distortions in the input data.

Pooling Layers

Pooling layers are a key component of deep convolutional neural networks (DCNNs), which reduce the spatial dimensions of feature maps while retaining the most important information. By summarizing local regions, pooling improves computational efficiency, provides translation invariance, and mitigates overfitting. These layers are interspersed between convolutional layers to progressively condense the spatial representation, allowing for deeper network architectures without excessive computational overhead.

The most common pooling methods include max pooling, which selects the maximum value from a pooling window, and average pooling, which computes the mean value. Max pooling emphasizes prominent features such as edges or textures, while average pooling smooths representations and retains broader contextual information. Global pooling, often used in architectures like ResNet, condenses the entire spatial dimension into a single value per feature map, enabling a seamless transition to fully connected layers.

Pooling parameters such as kernel size, stride, and padding control the dimensionality reduction process. For example, a 2×2 kernel with a stride of 2 reduces the size of the feature map by half in both dimensions. Although pooling is effective in capturing high-level features, it can lead to the loss of fine-grained spatial details, prompting the use of alternatives like strided convolutions or attention mechanisms in modern architectures.

Despite these challenges, pooling remains integral to DCNNs for tasks such as image classification, object detection, and semantic segmentation, where hierarchical feature extraction is essential. Tools like TensorFlow and PyTorch simplify the implementation of a pooling layer, making them a staple in deep learning workflows.

Fully Connected Layers

Fully connected (FC) layers are a fundamental building block of neural networks, including Deep Convolutional Neural Networks (DCNNs). Typically placed at the end of a network, fully connected layers serve as the decision-making mechanism, transforming the high-level features learned from the preceding convolutional and pooling layers into final predictions. Unlike convolutional layers, which focus on localized spatial patterns, fully connected layers treat all input features equally, establishing dense connections between neurons. This dense connectivity enables the network to combine spatially distributed features into a global representation,

crucial for classification, regression, and other tasks that require holistic understanding.

Structure of Fully Connected Layers

1. **Input Representation**:
   Fully connected layers receive flattened inputs, typically feature maps from the final convolutional or pooling layer. For example, a $7 \times 7 \times 512$ feature map is reshaped into a 1D vector with $7 \cdot 7 \cdot 512 = 25,088$ elements.
2. **Neuron Connections**:
   Each neuron in an FC layer connects to every input feature, resulting in the weight matrix W of dimensions $n \times m$, where n is the number of input features and m is the number of output neurons.
3. **Forward Pass**:
   The layer computes the weighted sum of inputs, adds a bias term b, and applies an activation function: $y = f(Wx + b)$. Here, $f(\cdot)$ is the activation function, commonly a ReLU, sigmoid, or softmax depending on the task.
4. **Output Representation**:
   The output dimensions depend on the number of neurons in the layer. For instance, the last FC layer in a classification network outputs probabilities over k classes using a softmax activation.

Common Architectures

Deep convolutional neural networks (DCNNs) have undergone significant evolution through the development of influential architectures that have shaped modern computer vision. These architectures, including AlexNet, VGG, GoogLeNet, ResNet, and others, introduced novel design strategies to address challenges such as vanishing gradients, computational inefficiency, and the need for robust feature extraction. Each architecture provided transformative insights that improved the depth, scalability, and efficiency of neural networks.

AlexNet, one of the first major breakthroughs, demonstrated the power of deep learning in large-scale image classification by utilizing GPUs for training, employing ReLU activations and integrating dropout for regularization. It marked the beginning of widespread adoption of deep networks in computer vision. VGG expanded on this success by showing that deeper networks, constructed with uniform small convolutional kernels, could achieve better performance while maintaining a modular design. This simplicity and scalability made VGG a widely adopted standard for further experimentation.

GoogLeNet introduced the concept of multiscale feature extraction through the Inception module, enabling the network to capture features at varying resolutions while maintaining computational efficiency. It highlighted the importance of balanc-

ing depth and efficiency in neural network design. ResNet addressed the problem of vanishing gradients in deep networks by introducing residual connections, allowing for training of extremely deep architectures. This innovation set a new benchmark for image classification and inspired numerous extensions and variations in subsequent models.

DenseNet built on the idea of residual connections by introducing dense connectivity, where each layer is connected to all the preceding layers. This approach improved feature reuse and gradient flow, leading to parameter-efficient architectures. Lightweight models like MobileNet and EfficientNet further extended these principles by focusing on resource efficiency, employing techniques such as depthwise separable convolutions and compound scaling to optimize performance for mobile and embedded systems.

8.8 Recurrent Neural Networks

Recurrent neural networks (RNNs) constitute a specialized category of neural networks specifically engineered for the modeling of sequential and time-dependent data. In contrast to feedforward networks, which process input in isolation, RNNs integrate memory via recurrent connections, thereby enabling the retention and utilization of information from prior inputs. This capability of capturing temporal dependencies makes RNNs exceptionally effective for applications such as natural language processing, speech recognition, and time series analysis. Over time, traditional RNNs have exhibited limitations, notably vanishing gradients and difficulties in learning long-term dependencies, precipitating the development of more advanced variants, namely long short-term memory (LSTM) networks and Gated Recurrent Units (GRUs). These advances have considerably broadened the scope of RNNs across a diverse array of sequence prediction tasks, including language modeling, sentiment analysis, and dynamic system forecasting. This chapter examines the fundamental concepts of RNNs, explores the architectures of LSTMs and GRUs, and emphasizes their applications in sequence prediction problems.

Basic RNN

Recurrent neural networks (RNNs) are a foundational architecture in deep learning, designed to process sequential data by incorporating temporal dependencies into the learning process. Unlike feedforward networks, where inputs are processed independently, RNNs introduce recurrence by maintaining a hidden state that is updated at each time step. This hidden state acts as a memory, capturing information about previous inputs and allowing the network to model sequences in which the order of inputs is critical.

The architecture of an RNN involves a recurrent loop that connects the output of a hidden layer back to its input in subsequent time steps. Mathematically, the hidden state at time t, denoted h_t, is calculated as a function of the current input x_t and the previous hidden state h_{t-1}. This is expressed as:

$$h_t = f(W_h h_{t-1} + W_x x_t + b), \tag{8.13}$$

where W_h and W_x are weight matrices, b is a bias vector, and f is an activation function, commonly a hyperbolic tangent (tanh) or sigmoid. The network output y_t at time t is typically derived from the hidden state:

$$y_t = g(W_y h_t + c), \tag{8.14}$$

where W_y is the output weight matrix, c is a bias vector, and g is often a softmax function for classification tasks.

This recursive nature enables RNNs to maintain a temporal representation of the data, making them suitable for applications involving sequences, such as time-series forecasting, natural language processing, and speech recognition. However, this same recursion presents challenges during training. One significant issue is the vanishing-gradient problem, where the gradients of the loss function with respect to earlier time steps diminish exponentially as backpropagation progresses through the sequence. This makes it difficult for basic RNNs to learn long-term dependencies, as the influence of earlier inputs on later output becomes negligible. In contrast, exploding gradients, although less common, can cause instability during training.

Despite these limitations, basic RNNs provide a simple and elegant framework for modeling sequential data. They serve as the foundation for more advanced architectures, such as long short-term memory (LSTM) networks and gated recurring units (GRUs), which incorporate mechanisms to mitigate the challenges of training in long sequences. Basic RNNs remain an important theoretical construct and are still effective for tasks with short-term dependencies and modest sequence lengths. They also provide valuable insights into the interplay between network architecture and the temporal nature of data, laying the groundwork for the development of more sophisticated recurrent models.

Example 2 (RNN classification)

This code implements a simple RNN-based neural network using the Keras library to classify handwritten digits from the MNIST data set. It loads the MNIST data, reshapes them into sequences of 28 time steps with 28 features (suitable for RNN input), and normalizes pixel values to the range [0, 1].

The model consists of a `SimpleRNN` layer with 128 units, followed by a `Dense` layer with 10 units for output (corresponding to the 10 digit classes) and a `softmax activation` for classification. The network is compiled using categorical cross-entropy loss, the Adam optimizer, and accuracy as the evaluation metric. It trains

for 10 epochs with a batch size of 128 and evaluates the model on the test data set, returning the loss and accuracy.

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation, SimpleRNN
from keras.utils import to_categorical, plot_model
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()


y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

x_train = np.reshape(x_train,[-1, 28, 28])
x_test = np.reshape(x_test,[-1, 28, 28])

x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255


network = Sequential()
network.add(SimpleRNN(units=128,
                      input_shape=(28, 28)))
network.add(Dense(10))
network.add(Activation('softmax'))
network.summary()

network.compile(loss='categorical_crossentropy', optimizer='adam', metrics
    =['accuracy'])
network.fit(x_train, y_train, epochs=10, batch_size=128)

loss, acc = network.evaluate(x_test, y_test, batch_size=128)
```

Listing 8.2 Keras implementation of RNN classification of MNIST data set

Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a specialized type of Recurrent Neural Network (RNN) designed to address the limitations of basic RNNs, particularly the vanishing gradient problem. Introduced by Hochreiter and Schmidhuber in 1997 [4], LSTMs enable effective modeling of long-term dependencies in sequential data by incorporating a memory cell and a system of gates that regulate the flow of information. These innovations allow LSTMs to retain and update information over extended sequences, making them particularly suitable for tasks such as natural language processing, time series forecasting, and speech recognition.

The core of an LSTM is its memory cell, which acts as a persistent storage mechanism, preserving information over arbitrary time intervals. This memory is modulated by three gates: the input gate, the forget gate, and the output gate. Each gate performs a distinct function and is controlled by readable parameters. The input gate determines to what extent new information is allowed to enter the memory cell, while the forget gate decides what portion of the existing memory should be discarded. The output gate controls how much of the memory is revealed to the

current output and subsequent computations. These gates operate through sigmoid activation functions, which produce outputs in the range of zero to one, enabling the network to weigh the influence of different components of the input and memory.

Mathematically, the operation of an LSTM can be described by a series of equations. At each time step t, the forget gate computes a forget vector f_t based on the previous hidden state h_{t-1} and the current input x_t:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f), \tag{8.15}$$

where W_f and b_f are the weight matrix and the bias vector, respectively, and σ is the activation function of the sigmoid. Similarly, the input gate generates an input vector i_t:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i). \tag{8.16}$$

A candidate memory update $\tilde{C}t$ is computed using a hyperbolic tangent activation function:

$$\tilde{C}t = \tanh(W_c \cdot [ht - 1, x_t] + b_c). \tag{8.17}$$

The current memory cell state C_t is then updated as a weighted combination of the previous memory state $Ct - 1$ and the candidate update:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}t, \tag{8.18}$$

where $\odot$ represents the multiplication in elements. Finally, the output gate determines the hidden state h_t, which is also the output of the LSTM for this time step:

$$\begin{aligned} (o_t &= \sigma(W_o \cdot [ht - 1, x_t] + b_o), \\ (h_t &= o_t \odot \tanh(C_t). \end{aligned} \tag{8.19}$$

This gating mechanism gives LSTMs the flexibility to learn when to store, update, or retrieve information, effectively mitigating the vanishing gradient problem that hinders traditional RNNs. The design enables LSTMs to focus on long-term dependencies while retaining the capability to adapt to short-term changes.

LSTMs have proven to be highly effective in a wide range of sequence modeling tasks. In natural language processing, they are frequently used for language modeling, machine translation, and sentiment analysis. Their ability to model temporal dependencies has also made them a cornerstone in speech recognition and audio processing applications. In time series forecasting, LSTMs excel at predicting future values based on historical data, demonstrating their adaptability across domains.

Although LSTMs address many limitations of basic RNNs, they are computationally intensive because of their complex gating mechanisms. Training LSTMs on large data sets or long sequences can be resource-heavy, often necessitating optimization techniques such as gradient clipping or the use of specialized hardware like GPUs.

Despite these challenges, LSTMs remain a dominant architecture for sequential data processing, and their design has inspired numerous extensions and variants, including gated recurring units (GRUs) and attention-based models. The principles underlying LSTMs continue to influence the development of more advanced architectures, underscoring their foundational role in deep learning.

Fun fact: In the early 2010s, the rise of graphics processing units (GPUs) finally gave LSTMs the computational power they needed to shine. GPUs enabled researchers to train larger models on more extensive data sets, and LSTMs became a core component of applications in natural language processing, machine translation, and video analysis.
A fun anecdote from this period involves Andrej Karpathy, a prominent AI researcher, who used LSTMs to train a character-level language model on the text of Shakespeare's works. The LSTM learned to generate text that mimicked Shakespeare's style, albeit hilariously nonsensical at times. For example:

"Prithee, I shall with my heart, sir, and your friends".[a]

[a] https://karpathy.github.io/2015/05/21/rnn-effectiveness/.

Gated Recurrent Units (GRU)

Gated Recurrent Units (GRUs) are a simplified variant of long short-term memory (LSTM) networks, introduced by Cho et al. in 2014 [5]. GRUs aim to address the challenges of modeling sequential data, particularly long-term dependencies, while reducing the computational complexity inherent in LSTMs. By streamlining the architecture, GRUs retain the ability to capture temporal patterns effectively but require fewer parameters and less computational overhead. This balance makes GRUs a practical choice for many sequence-based tasks in natural language processing, time series analysis, and speech recognition.

The design of a GRU incorporates two primary gates: the update gate and the reset gate. These gates work together to regulate the flow of information through the network, determining which information to retain, update, or discard. The update gate controls how much of the previous hidden state is carried forward to the current state, balancing the preservation of long-term dependencies with the incorporation of new input. The reset gate determines to what extent the previous hidden state contributes to the generation of candidate activation for the current time step. This

mechanism enables GRUs to adapt flexibly to short- and long-term dependencies in the input sequence.

Mathematically, the update gate z_t is computed as a function of the current input x_t and the previous hidden state h_{t-1}:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z), \tag{8.20}$$

where W_z is the weight matrix, b_z is the bias vector and σ denotes the activation function of the sigmoid. The reset gate r_t is similarly calculated:

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r). \tag{8.21}$$

Using the reset gate, a candidate hidden state $\tilde{h}t$ is generated, which represents the contribution of the current input to the state:

$$\tilde{h}t = \tanh(W_h \cdot [r_t \odot ht - 1, x_t] + b_h), \tag{8.22}$$

where $\odot$ denotes element-wise multiplication, and $tanh$ is the hyperbolic tangent activation function. The final hidden state h_t for the current time step is then computed as a weighted combination of the previous hidden state and the candidate hidden state, mediated by the update gate:

$$h_t = (1 - z_t) \odot ht - 1 + z_t \odot \tilde{h}_t. \tag{8.23}$$

This architecture simplifies the memory cell and gating mechanisms found in LSTMs by combining their roles into fewer components. The absence of a separate memory cell reduces the number of parameters, making GRUs computationally more efficient while retaining comparable performance. The gating mechanisms enable GRUs to adaptively control the influence of past states, providing robustness in learning both short- and long-term dependencies.

GRUs have shown efficacy across a wide range of applications. In natural language processing, they are widely used for tasks such as machine translation, text generation, and language modeling. Their ability to process sequential data efficiently makes them suitable for real-time applications, including speech recognition and streaming data analysis. GRUs are also prevalent in time-series forecasting, where reduced computational requirements allow faster training on large data sets or deployment on resource-constrained devices.

Although GRUs share many advantages with LSTMs, their streamlined design may make them less expressive in some scenarios, particularly when intricate long-term dependencies need to be modeled. However, their computational efficiency and ease of implementation have made them a popular choice for many practical applications, particularly in cases where computational resources are limited or where training speed is a priority.

8.9 Advanced Training Techniques

Training deep neural networks effectively requires strategies to improve generalization, mitigate overfitting, and ensure efficient convergence. Advanced training techniques, such as dropout, batch normalization, and data augmentation, have become integral components of modern deep learning workflows. These techniques address challenges in optimization and model robustness by altering network behavior during training or preprocessing input data. Their use has contributed to significant advancements in the performance and reliability of neural networks in a wide range of applications.

Dropout is a regularization technique designed to reduce overfitting by randomly deactivating a subset of neurons during training. By preventing specific units from relying heavily on their neighboring activations, dropout encourages the network to learn redundant representations and fosters a more distributed encoding of features. The technique introduces stochasticity into the training process, as neurons are dropped out independently with a predefined probability. During inference, dropout is deactivated, and the network uses the full set of weights, appropriately scaled to account for the absence of dropped units during training. This mechanism has proven particularly effective in reducing overfitting in large networks and improving generalization to unseen data.

Batch normalization addresses the issue of internal covariate shift, where the distribution of inputs to a layer changes as the model parameters update during training. This phenomenon can slow convergence and make optimization challenging, particularly in deep networks. Batch normalization normalizes the activations of each layer by adjusting and scaling them to a standard distribution, based on the statistics of mini-batches during training. This normalization is followed by learnable parameters that allow the model to scale and shift the normalized values as necessary. By stabilizing the input distributions of intermediate layers, batch normalization enables the use of higher learning rates, accelerates convergence, and reduces the sensitivity to initialization. Additionally, it has a mild regularization effect by introducing noise from mini-batch statistics, which can further improve generalization.

Data augmentation is a pre-processing strategy that enhances the diversity of training data by applying transformations to existing samples. These transformations, such as rotations, translations, cropping, flipping, or color jittering, simulate variations that might be encountered in real-world scenarios. Data augmentation allows the network to learn robust and invariant features without requiring additional data collection. In image recognition tasks, augmentation techniques such as random cropping and horizontal flipping have been particularly impactful, while in natural language processing, methods like synonym replacement and back-translation have proven useful. Augmentation not only increases the effective size of the training data set but also helps reduce overfitting by discouraging the network from memorizing specific data patterns.

These advanced training techniques often complement each other, collectively addressing the multifaceted challenges of deep learning. Dropout and batch nor-

malization operate directly on the network architecture, modifying the behavior of activations and weight updates to enhance learning dynamics. In contrast, data augmentation enriches the input data itself, expanding the variability of the training set and helping the model generalize across diverse input distributions. Together, they represent a toolkit for optimizing training processes, fostering robust feature learning, and achieving superior performance in complex deep learning tasks.

8.10 Network Architectures

The evolution of deep learning has been driven by the development of diverse network architectures, each tailored to address specific challenges and applications. These architectures go beyond traditional feedforward and convolutional designs, introducing novel mechanisms to process complex data types, enhance representational power, and generate new insights across domains. From the multiscale feature extraction of Inception networks to the representation learning capabilities of autoencoders and the dynamic attention mechanisms in Attention Networks, these models have redefined what neural networks can achieve. Generative Adversarial Networks (GANs) have opened new frontiers in content creation, while Graph Neural Networks (GNNs) have extended deep learning to non-Euclidean data structures like social networks and molecular graphs. Hybrid architectures combine the strengths of multiple models, creating versatile solutions for intricate tasks. Emerging trends in architecture design continue to push boundaries, integrating advances such as self-supervision, sparsity, and neuromorphic computing. This chapter explores the principles and innovations behind these architectures, highlighting their transformative impact on deep learning and their growing role in shaping research and real-world applications.

Inception

The Inception architecture, introduced by Szegedy et al. [6], represents a major innovation in convolutional neural network design. Its primary contribution lies in the Inception module, a carefully engineered structure that enables multi-scale feature extraction within a single layer. By allowing the network to capture information at varying levels of abstraction, the Inception architecture balances computational efficiency with representational power, making it a versatile choice for complex tasks such as image classification and object detection.

An Inception module integrates parallel convolutional paths with varying kernel sizes. This design enables the module to simultaneously extract local features using smaller kernels, such as 1×1 and 3×3, and capture broader context with larger kernels, such as 5×5. To manage computational costs, 1×1 convolutions are employed for dimensionality reduction, compressing feature maps before applying more com-

putationally intensive operations. The pooling layers are also incorporated within the module to enhance the robustness and provide additional abstraction. By combining these parallel operations, the Inception module produces a rich set of feature maps that represent different levels of spatial hierarchy.

The original Inception architecture, often referred to as GoogLeNet, demonstrated the utility of this approach by achieving state-of-the-art results in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) while using significantly fewer parameters than earlier deep networks like VGG. The success of GoogLeNet spurred further development, leading to improved variants of the Inception architecture. Inception-v3 introduced several enhancements, including factorized convolutions that decompose larger kernels into successive smaller ones, such as splitting a 5×5 convolution into two 3×3 convolutions. This adjustment improved both computational efficiency and model performance. Inception-v3 also integrated batch normalization into the auxiliary classifiers, stabilizing training, and further enhancing generalization.

Inception-v4 and its related architecture, Inception-ResNet, continued this trajectory by combining the strengths of the Inception module with residual connections. Residual connections, introduced in ResNet, alleviate the problem of vanishing gradients in very deep networks by enabling shortcut paths for gradient flow. The integration of these connections into the Inception framework facilitated the training of deeper and more expressive models, extending their applicability to a wider range of tasks.

The modular design of inception networks has contributed significantly to their flexibility and scalability. These networks can be customized to specific applications by varying the number and configuration of Inception modules. Their success has inspired adaptations in domains beyond image classification, including medical imaging, video analysis, and multi-modal data processing. The architecture exemplifies the power of combining multi-scale feature extraction with computational efficiency, serving as a blueprint for subsequent innovations in neural network design. By addressing key challenges in deep learning, such as overfitting and computational constraints, the Inception architecture and its variants remain influential in both research and practical applications.

Example 3 (Inception network)

This code implements a Convolutional Neural Network (CNN) using Keras to classify images from the CIFAR-10 data set. The CIFAR-10 data are loaded, normalized to the range [0, 1], and encoded with a hot digit for the 10 output classes.

The model consists of several `Conv2D` layers with 64 filters and different kernel sizes ($1x1, 3x3, 5x5$), followed by a `MaxPooling2D` layer for downsampling. After further convolution, the output is flattened and passed through a `Dense` layer with 10 units and a `softmax activation` function for classification.

The network is compiled using the SGD optimizer with momentum, learning rate decay, and categorical cross-entropy loss. It trains for 10 epochs with a batch size of 32, using the test data for validation, and prints the network summary, including its structure and parameter count.

```
from keras.datasets import cifar10
from keras.utils import np_utils
from keras.layers import Input
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Flatten, Dense
from keras.models import Model
from keras.optimizers import SGD

(X_train, y_train), (X_test, y_test) = cifar10.load_data()
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')
X_train = X_train / 255.0
X_test = X_test / 255.0

y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)


network = Sequential()
network.add(Input(shape = (32, 32, 3)))
network.add(Conv2D(64, (1,1), padding='same', activation='relu'))
network.add(Conv2D(64, (3,3), padding='same', activation='relu'))
network.add(Conv2D(64, (1,1), padding='same', activation='relu'))
network.add(Conv2D(64, (5,5), padding='same', activation='relu'))
network.add(MaxPooling2D((3,3), strides=(1,1), padding='same'))
network.add(Conv2D(64, (1,1), padding='same', activation='relu'))
network.add(Flatten())
network.add(Dense(10, activation='softmax'))
network.summary()


sgd = SGD(lr=0.01, momentum=0.9, decay=0.001, nesterov=False)
network.compile(loss='categorical_crossentropy', optimizer=sgd,
metrics=['accuracy'])
network.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=10,
      batch_size=32)
```

Listing 8.3 Inception network classification on Cifar10 data set

Autoencoders

Autoencoders are a class of neural networks designed to learn efficient representations of input data through an unsupervised learning paradigm. They consist of two main components: an encoder and a decoder. The encoder maps the input data to a lower-dimensional latent space, capturing its most salient features, while the decoder reconstructs the input from this compressed representation. The reconstruction process ensures that the latent space retains the critical information necessary to represent the input, making autoencoders particularly effective for tasks such as dimensionality reduction, feature extraction, and generative modeling.

The simplest form, often referred to as a vanilla autoencoder, uses fully connected layers for both the encoder and decoder. During training, the network minimizes a reconstruction loss, typically the mean squared error between the input and its reconstruction. The encoder compresses the input into a latent vector of reduced dimensionality, and the decoder learns to map this latent vector back to the original

input space. This structure forces the autoencoder to focus on the most important patterns in the data while discarding irrelevant noise. Despite their simplicity, vanilla autoencoders provide valuable insights into data structure and serve as a foundation for more complex variants.

Variational Autoencoders (VAEs) extend the vanilla architecture by incorporating probabilistic principles into the learning process. Instead of mapping inputs to a deterministic latent vector, VAEs encode them as a distribution, typically modeled as a Gaussian. This allows VAEs to generate new data samples by sampling from the learned latent space, making them powerful tools for generative modeling. During training, VAEs optimize a combined loss function that includes both reconstruction loss and a regularization term, derived from the Kullback-Leibler divergence, to ensure that the learned distribution approximates the prior distribution. This probabilistic framework enables VAEs to capture complex data distributions and generate realistic, diverse outputs.

Autoencoders have found extensive applications across a range of domains. In denoising tasks, they are trained to reconstruct clean data from corrupted inputs, effectively learning to filter out noise while preserving essential features. This capability is widely used in image processing, where autoencoders restore degraded images or enhance their quality. In anomaly detection, autoencoders excel by learning a compact representation of normal data patterns during training. When presented with anomalous data, the reconstruction error typically increases, providing a clear signal for detection. This approach has been applied in areas such as fraud detection, industrial monitoring, and medical diagnostics.

The versatility of autoencoders extends to other domains, including natural language processing, where they have been used for tasks such as text compression and feature extraction. Their ability to model complex data distributions has also made them integral components in hybrid architectures, such as combining VAEs with adversarial networks to create more robust generative models. Autoencoders continue to play a significant role in advancing machine learning, offering a blend of theoretical elegance and practical utility that has influenced the design of modern neural networks. Their applications highlight their capacity to extract meaningful structure from data, making them indispensable in the broader field of representation learning.

Attention Networks

Attention networks represent a paradigm shift in deep learning, providing a mechanism to dynamically focus on relevant parts of input data while processing it. The attention mechanism was first introduced to address challenges in sequence-to-sequence models, particularly in tasks like machine translation. By allowing the model to selectively weigh the importance of different input elements, attention networks overcome the limitations of fixed-length representations and improve the capacity to model long-range dependencies.

At the core of the attention mechanism is the concept of assigning importance weights to elements of the input sequence. These weights are computed based on the relationship between the query, key, and value representations of the input data. The query identifies what the model seeks to focus on, the key helps evaluate relevance, and the value represents the information associated with each input element. The attention weights are computed by taking the compatibility score between the query and key representations, often using a dot product or other similarity measures, followed by a softmax function to ensure the weights sum to one. The resulting weighted combination of values serves as the attention output, enabling the model to aggregate relevant information dynamically.

The Transformer architecture, introduced by Vaswani et al., builds on the attention mechanism to create a highly efficient and scalable model for processing sequential data. Transformers replace recurrent computations with self-attention, where each element of the input attends to every other element in the sequence. This design enables parallel processing, significantly reducing training time compared to recurrent networks. The key innovation of the Transformer lies in its multi-head attention mechanism, which allows the model to focus on different parts of the sequence simultaneously. By combining multiple attention heads, the Transformer captures diverse patterns and dependencies in the data. Additionally, positional encodings are incorporated to inject information about the order of elements in the sequence, compensating for the lack of recurrence.

Applications of attention networks, particularly Transformers, have transformed natural language processing (NLP). Tasks such as machine translation, text summarization, and sentiment analysis have seen substantial performance improvements due to the ability of attention mechanisms to capture complex syntactic and semantic relationships in text. Pre-trained models like BERT, GPT, and T5, based on the Transformer architecture, have set new benchmarks in NLP by leveraging large-scale unsupervised learning to produce contextualized word representations. These models generalize well across tasks through fine-tuning, enabling efficient adaptation to specific applications.

The versatility of attention networks extends beyond NLP to domains such as computer vision and speech processing. In vision, attention mechanisms are employed to identify regions of interest in images, improving tasks like object detection and image captioning. In speech, attention facilitates alignment between input features and output targets, enhancing models for speech recognition and synthesis. The conceptual simplicity and effectiveness of attention have made it a cornerstone of modern deep learning, driving innovations across disciplines and shaping the future of artificial intelligence.

Generative Adversarial Networks

Generative Adversarial Networks (GANs), introduced by Goodfellow et al., represent a powerful framework for generative modeling. GANs consist of two neural

networks, a generator and a discriminator, that are trained simultaneously in a competitive setting. The generator learns to produce data samples resembling the training data, while the discriminator attempts to distinguish between real and generated samples. This adversarial process leads to the refinement of the generator's output, enabling the creation of realistic data samples across various domains.

The architecture of GANs revolves around the interplay between the generator and the discriminator. The generator starts with a random input, often a vector sampled from a latent space, and maps it to the data space using a series of transformations. Its objective is to produce outputs that are indistinguishable from the real data. The discriminator, on the other hand, evaluates inputs to classify them as real or fake, providing feedback to the generator. This feedback, encoded as the gradients of the loss function, guides the generator in improving its output over successive iterations. The training process is framed as a minimax optimization problem, where the generator seeks to minimize the discriminator's ability to identify fake samples while the discriminator maximizes its classification accuracy.

Variants of GANs have emerged to address challenges in the original formulation and expand their applicability. Deep Convolutional GANs (DCGANs) introduced convolutional architectures into both the generator and discriminator, enhancing the ability to model high-dimensional data, particularly images. Wasserstein GANs (WGANs) redefined the loss function using the Earth Mover's distance, improving training stability and addressing issues of vanishing gradients. Other adaptations, such as Conditional GANs (CGANs), allow the generation of data conditioned on specific inputs, enabling controlled synthesis in tasks like image-to-image translation and text-to-image generation (Fig. 8.6).

GAN applications span a wide range of fields, with significant contributions in image generation, style transfer, and data augmentation. In image generation, GANs have achieved remarkable success in creating realistic faces, landscapes, and other complex visual content. Style transfer leverages GANs to transform images by combining content from one domain with the stylistic elements of another, resulting in visually compelling output. In data augmentation, GANs generate synthetic training data to enrich data sets for tasks where data collection is limited or costly. Beyond

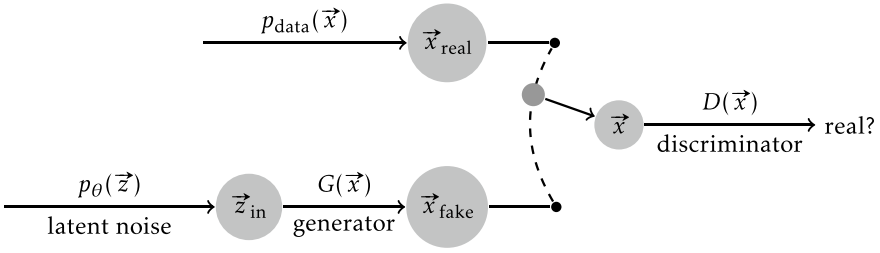


Fig. 8.6 Generative adversarial networks architecture (by Janosh Riebesell, see: https://tikz.net/gan/)

these applications, GANs have been used in video generation, 3D modeling, and even molecular design, demonstrating their versatility.

The impact of GANs extends beyond their immediate applications and influences the broader field of generative modeling. Despite challenges such as mode collapse and training instability, ongoing research continues to refine GAN architectures and training techniques, improving their robustness and scalability. GANs remain a cornerstone of modern machine learning, bridge the gap between data generation and real-world creativity, and their influence is likely to grow as new innovations emerge.

Fun fact: GANs were introduced in 2014 by Ian Goodfellow during a pivotal moment in machine learning history. The story goes that the idea for GANs came to Goodfellow during a late-night conversation at a bar with colleagues. They were discussing ways to improve generative models, and Goodfellow proposed the adversarial setup: a generator creating data and a discriminator evaluating it.
Inspired, Goodfellow reportedly left the bar, went home, and coded the first version of GANs. Within days, he had a working prototype that demonstrated the potential of this new framework. This informal and almost spontaneous moment gave birth to one of the most influential ideas in modern AI.[a]

[a] https://www.deeplearning.ai/the-batch/ian-goodfellow-a-man-a-plan-a-gan/.

Graph Neural Networks

Graph Neural Networks (GNNs) are a class of neural networks specifically designed to process data represented as graphs, where entities are modeled as nodes, and their relationships are expressed as edges. Unlike traditional deep learning models that operate on grid-like data structures such as images or sequences, GNNs generalize to non-Euclidean domains, making them suitable for a wide range of applications where data is inherently relational. By leveraging the structure of graphs, GNNs effectively capture both the features of individual nodes and the dependencies among them.

Graphs are mathematical structures that consist of a set of nodes and edges. In a graph representation, each node is associated with a feature vector that encapsulates its attributes, while edges often carry weights or labels representing the strength or type of the relationship. The adjacency matrix is a common representation of a graph, encoding the connections between nodes. GNNs extend this representation by learning embeddings for nodes and edges, which are updated iteratively to capture local and global graph structures. This process allows GNNs to encode the topology

of the graph alongside node and edge attributes, creating representations that are suitable for downstream tasks.

A pivotal advancement in GNNs is the development of Graph Convolutional Networks (GCNs), which generalize the convolution operation from grid-like data to graphs. GCNs aggregate features from a node's neighborhood to compute an updated representation for the node. This operation can be viewed as a form of message passing, where information is exchanged between connected nodes to refine their embeddings. Mathematically, the aggregation process is often expressed as a weighted sum of the features of neighboring nodes, scaled by the adjacency matrix and normalized to account for variations in neighborhood size. Multiple layers of graph convolutions allow the network to propagate information across larger portions of the graph, enabling the modeling of both local and global dependencies.

Applications of GNNs span diverse fields where graph-structured data plays a central role. In social networks, GNNs are used to predict user behavior, recommend connections, and detect communities by analyzing patterns of interaction. In molecular analysis, they model chemical compounds as graphs, with atoms as nodes and bonds as edges, enabling tasks such as molecular property prediction and drug discovery. Beyond these domains, GNNs are employed in knowledge graphs for reasoning, in transportation networks for route optimization, and in cybersecurity for anomaly detection within network traffic.

GNNs represent a significant step forward in machine learning, allowing the effective use of graph-based data across disciplines. While challenges remain, such as scaling to large graphs and preserving computational efficiency, ongoing research continues to refine GNN architectures and extend their capabilities. By integrating the relational structure of data into the learning process, GNNs have opened new frontiers in fields ranging from natural sciences to social analysis, underscoring their transformative potential.

Hybrid Architectures

Hybrid architectures represent a sophisticated approach in deep learning, combining the strengths of different neural network paradigms to address complex tasks that cannot be effectively handled by a single type of network. These architectures integrate components such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) or fuse neural networks with symbolic reasoning frameworks, enabling enhanced representational power, interpretability, and generalization across diverse domains.

The combination of CNNs and RNNs is a prominent hybrid approach that leverages the complementary strengths of these architectures. CNNs excel at extracting spatial hierarchies from grid-like data such as images, capturing features ranging from edges and textures to high-level object representations. RNNs, on the other hand, are designed for sequential data, modeling temporal dependencies and contextual relationships. By integrating these two architectures, hybrid models can process

spatio-temporal data effectively. For example, in video analysis, CNNs extract spatial features from individual frames, while RNNs process these features sequentially to capture temporal dynamics, enabling tasks such as activity recognition and video captioning. Similarly, in medical imaging, CNNs analyze spatial patterns within scans, and RNNs aggregate these features over time to detect temporal trends in patient data.

Another significant direction in hybrid architectures is neuro-symbolic integration, which seeks to combine the representational power of neural networks with the logical reasoning capabilities of symbolic systems. Neural networks are well-suited for learning from raw, unstructured data, capturing implicit patterns and representations. However, they often struggle with tasks requiring explicit reasoning, such as rule-based decision-making or understanding relationships in structured domains. Symbolic systems, in contrast, excel at encoding and manipulating explicit knowledge but lack the ability to learn from data. Neuro-symbolic architectures bridge this gap by integrating neural networks for perception and representation learning with symbolic reasoning components for decision-making and inference. These systems have shown promise in applications such as knowledge graph reasoning, visual question answering, and robotic planning, where both perception and logical reasoning are critical.

Hybrid architectures provide a pathway for addressing challenges that traditional architectures cannot resolve independently. By combining different paradigms, they enable the modeling of multi-faceted problems, incorporating spatial, temporal, and logical components into a unified framework. This integration not only enhances task performance but also opens new avenues for research, such as explainable AI, where neuro-symbolic systems can offer insights into the reasoning processes underlying neural network predictions. The development of hybrid architectures reflects the broader evolution of artificial intelligence, emphasizing the need for diverse, interdisciplinary approaches to tackle complex, real-world challenges. These architectures are increasingly seen as a cornerstone for advancing both theoretical understanding and practical applications of machine learning.

Emerging Trends and Research

The field of deep learning continues to evolve rapidly, driven by innovative research and emerging trends that seek to address its limitations and expand its applicability. Among the most prominent areas of advancement are few-shot learning, meta-learning, and explainable AI, each representing a critical step toward creating more efficient, generalizable, and interpretable models. These directions aim to overcome challenges such as data scarcity, model adaptability, and the opacity of complex neural networks.

Few-shot learning focuses on enabling models to generalize from a limited number of training examples, addressing the reliance of traditional deep learning methods on large labeled data sets. This paradigm seeks to mimic human learning, where

individuals can acquire new skills or recognize novel concepts from minimal exposure. Few-shot learning often employs techniques such as metric learning, where models are trained to compare inputs and infer similarities, or generative approaches, which synthesize additional data to augment learning. Applications span from medical diagnostics, where annotated data is scarce, to natural language processing tasks like machine translation for low-resource languages. By reducing the dependency on extensive data sets, few-shot learning holds significant promise for democratizing access to deep learning across domains with limited data availability.

Meta-learning, or learning to learn, represents another frontier in emerging research. Meta-learning frameworks aim to train models that can rapidly adapt to new tasks with minimal fine-tuning. This adaptability is achieved by optimizing over distributions of tasks rather than individual tasks, enabling models to extract shared knowledge and apply it efficiently to unseen scenarios. Meta-learning algorithms often operate at multiple levels, with one learning process guiding another. For instance, the inner loop might optimize task-specific parameters, while the outer loop adjusts meta-parameters to improve adaptability. Meta-learning has proven valuable in robotics, where systems must quickly adapt to novel environments, and in few-shot learning settings, where task-specific data is limited. Its potential to create flexible and efficient learning systems underscores its importance in the future of artificial intelligence.

Explainable AI (XAI) addresses one of the most pressing concerns in deploying deep learning systems: the lack of interpretability. As neural networks grow more complex, understanding their decision-making processes becomes increasingly challenging, particularly in high-stakes applications like healthcare, autonomous systems, and finance. XAI seeks to provide insights into how models arrive at their predictions, enabling users to trust and verify their outputs. Techniques for explainability include feature attribution methods, which identify the contributions of input features to the model's output, and surrogate models, which approximate the behavior of complex networks using simpler, interpretable models. Advances in XAI have also led to the development of inherently interpretable architectures, which incorporate transparency into their design. The integration of explainability into deep learning workflows not only enhances trust but also facilitates debugging, fairness, and compliance with regulatory frameworks.

These emerging trends collectively represent a shift in focus from maximizing raw performance to addressing fundamental challenges in scalability, adaptability, and transparency. Few-shot learning and meta-learning push the boundaries of data efficiency and generalization, making deep learning accessible to new domains, while explainable AI fosters trust and accountability, critical for widespread adoption in sensitive and regulated industries. Together, they define the cutting edge of deep learning research, shaping its trajectory toward more versatile, ethical, and impactful systems. Their continued development will likely play a pivotal role in the evolution of artificial intelligence, ensuring its relevance and utility in addressing the complex challenges of the real world.

For Further Reading

1. Bengio Y, Goodfellow I, Courville A (2016) Deep learning. MIT Press
2. Bishop CM (2005) Neural networks for pattern recognition. Oxford University Press
3. Chollet F (2021) Deep learning with Python. Manning
4. Hagan MT, Demuth HB, Beale MH, de Jesus O (2014) Neural network design
5. Aggarwal CC (2018) Neural networks and deep learning: a textbook. Springer
6. Ravichandiran S (2019) Hands-on deep learning algorithms with Python. Packt

References

1. McCulloch W, Pitts W (1943) A logical calculus of ideas immanent in nervous activity. Bull Math Biophys 5:115–119
2. Rosenblatt F (1958) The perceptron: a probabilistic model for information storage and organization in the brain. Psychol Rev 65:358–408
3. Minsky ML, Papert SA (1988) Perceptrons: expanded edition
4. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Comput. MIT-Press
5. Cho K, Van Merriënboer B, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y (2017) Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078
6. Szegedy C, Liu W, Jia Y, Sermanet P, Reed S, Anguelov D, Erhan D, Vanhoucke V, Rabinovich A (2015) Going deeper with convolutions. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 1–9

Afterword

This book, *Pattern Recognition Primer*, was intended as a detailed and systematic guide to understanding the key concepts, mathematical foundations, and practical applications of pattern recognition. By structuring the material from fundamental principles to advanced methodologies, our aim was to provide a cohesive narrative that equipped the readers with the tools to analyze and solve classification problems.

The contents of the book covered a broad spectrum of topics. It began with an introduction to essential terminology, feature selection, taxonomies, and quality metrics, setting the stage for understanding the processes that underpinned pattern recognition systems. The readers were guided through the mathematical fundamentals of the field, including statistics, probability theory, linear algebra, calculus, fuzzy logic, and dissimilarity measures, ensuring that even those with limited prior experience in mathematics could follow the concepts presented.

The subsequent chapters investigated both unsupervised and supervised learning approaches. The book explored clustering techniques, including K-means, fuzzy and possibilistic clustering, hierarchical methods, and density-based approaches, with detailed discussions on quality metrics and validation methods. The section on supervised learning methods introduces foundational algorithms such as Fisher's classifier, nearest neighbor methods, and various regression techniques, along with decision trees, support vector machines, and ensemble learning.

Deep learning, an essential component of modern pattern recognition, was comprehensively addressed in the chapter on neural networks. The topics ranged from artificial neurons and shallow networks to advanced architectures such as deep convolutional and recurrent neural networks. The book also included a discussion of state-of-the-art training techniques and evaluation metrics, reflecting the rapid evolution of the field.

Throughout the book, the inclusion of Python code examples bridged the gap between theoretical understanding and practical implementation. Exercises at the end of each chapter further reinforced the material, allowing readers to consolidate their knowledge by applying it to real-world scenarios.

K. Przystalski et al., *Pattern Recognition Primer*,
https://doi.org/10.1007/978-3-031-91816-2

By systematically covering foundational concepts, a wide range of algorithms, and practical implementation strategies, Pattern Recognition Primer offered readers a thorough foundation in the intricacies of pattern recognition. It was our expectation that the material presented there would support the reader in developing a deep and functional understanding of the methods and techniques that drove this critical area of science and engineering.

We wish all our readers a fruitful and inspiring journey in applying the knowledge they have gained.

The Authors

Appendix A
Exercises

Fundamentals

1 (Minimum distance) Based on the minimum distance classifier example calculate class centers, discriminant functions, and hyperplane for training data given as in Table A.1.

2 (Multiclass minimum distance classifier) Use the minimum distance classifier for a multiclass problem as given in the Table A.2. In this case, please prepare three pairs of distinguish functions where each pair represents a function for set of a given class and the second one for each opposite object.

3 (Quality metrics) Calculate the quality metric like shown in Table 1.6 for data presented in Table A.3.

Table A.1 Minimum distance classifier training set for exercise 1.1

x_1	x_2	y	x_1	x_2	y
−0.95	0.52	−1	0.52	−0.80	1
−0.80	0.53	−1	0.70	−0.30	1
−0.70	0.91	−1	0.74	0.10	1
−0.50	0.43	−1	0.41	0.20	1
0.10	0.33	−1	0.45	−0.80	1
−0.30	0.05	−1	0.97	−0.30	1
−0.25	0.18	−1	0.99	−0.70	1
−0.60	0.18	−1	0.67	−0.45	1
0.25	0.89	−1	0.74	−0.80	1
0.40	0.95	−1	0.06	−0.70	1

K. Przystalski et al., *Pattern Recognition Primer*,
https://doi.org/10.1007/978-3-031-91816-2

Table A.2 Minimum distance classifier training set for exercise 1.1

x_1	x_2	y	x_1	x_2	y	x_1	x_2	y
−0.95	−0.8	1	−0.75	0.75	2	0.52	0.52	3
−0.8	−0.3	1	−0.9	0.6	2	0.7	0.53	3
−0.7	−0.1	1	−0.25	0.5	2	0.74	0.91	3
−0.5	−0.2	1	−0.1	0.65	2	0.41	0.43	3
0.1	−0.8	1	−0.5	0.8	2	0.45	0.33	3
−0.3	−0.3	1	−0.66	0.5	2	0.97	0.05	3
−0.25	−0.7	1	−1.0	0.8	2	0.99	0.18	3
−0.6	−0.45	1	−0.45	0.7	2	0.67	0.18	3
0.25	−0.8	1	−0.1	0.9	2	0.74	0.89	3
0.4	−0.7	1	−0.15	0.75	2	0.66	−0.49	3

Table A.3 Three doctors' prediction compared to true condition of lung cancer

Dr. Newton		Dr. Einstein	
Condition	Diagnosis	Condition	Diagnosis
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	1	−1
−1	−1	−1	1

4 (Over and underfit) Use the training set to train the minimum distance classifier. Test the distinguish function on the testing set. Does the model overfit or underfit?

5 (ROC curve) Calculate the ROC curve and AUC value of previous exercise for cutoff points: 4, 6, and 8 (Table A.4).

Table A.4 Overfit and underfit example training and testing set

x_1	x_2	y	x_1	x_2	y
Training set					
−0.95	−0.8	−1	−0.66	0.5	1
−0.8	−0.3	−1	−0.45	0.7	1
−0.7	−0.1	−1	−0.1	0.9	1
−0.5	−0.2	−1	−0.15	0.75	1
0.1	−0.8	−1	0.5	0.25	1
−0.3	−0.3	−1	0.52	0.52	1
−0.25	−0.7	−1	0.7	0.53	1
−0.6	−0.45	−1	0.74	0.91	1
0.25	−0.8	−1	0.41	0.43	1
0.4	−0.7	−1	0.45	0.33	1
−0.9	0.6	−1	0.97	0.05	1
−0.25	0.5	−1	0.99	0.18	1
−0.1	0.65	−1	0.67	0.18	1
−0.5	0.8	−1	0.74	0.89	1
−1.0	0.8	−1	0.66	−0.49	1
Testing set					
0.0	0.0	−1	−0.5	0.1	1
0.25	0.3	−1	−0.3	0.25	1
−0.1	0.9	−1	0.6	−0.25	1
−0.1	−0.5	−1	1.0	−0.3	1
0.75	−0.1	−1	−0.8	0.1	1

Math

6 (Combinatorics) Calculate the probability of a full house in poker.

7 (Total probability) Let us take Manchester City, one of Premier League football club. What are the chances to win the Premier League Championship (event B) if we know that:

- $B|A_1$—we can win with a team from set A_1 for about 85%,
- $B|A_2$—we can win with a team from set A_2 for about 65%,
- $B|A_3$—the chances to win with a team from set A_3 is about 50%,
- $B|A_4$—there are greater chances to lose as the chances of winning are about 40%?

The size of the set is as follows: $A_1 - 60\%$, $A_2 - 25\%$, $A_3 - 10\%$, $A_4 - 5\%$.

8 (Standard deviation) We took five random newborns and took their weights. The weights are presented in Table A.5. The weight is obviously in kilograms. Calculate variance, average deviation, and standard deviation.

Table A.5 Newborn weights

x_1	x_2	x_3	x_4	x_5
3.4	4.2	4.6	3.2	2.7

Table A.6 Correlation between hours spent on running and calories burned

Average hours spent on running	0.2	0.5	1	1.5	2
Average calories burned	80	150	240	300	530

9 (Correlation) While running we burn calories. An example of the relation between hours spend on running and burned calories are shown in Table A.6. Calculate the correlation.

10 (Limits) Calculate following limits:

$$\lim_{dx \to 0^+} \frac{2 + dx}{dx},$$

$$\lim_{dx \to -\infty} 4x^2,$$

$$\lim_{dx \to 2} \frac{x^3 + x^2 + 2}{3x^2 + 3x + 2}.$$

11 (Derivatives) Calculate following derivatives:

- $f(x) = x^3 + 2x + 4$,
- $f(x) = sin(x) + 2$,
- $f(x) = 2x + \ln x$,

for points $x = 2$, $x = 4$, and $x = 6$.

12 (Gradient) Calculate the gradient of function:

$$f(x_1, x_2) = x_1^3 + 3x_2^2 + 2x_1^2 + 3x_1 + 4x_2 + 3$$

for points: 1, 4, and 5.

13 (Dissimilarity measure I) Calculate the Minkowski, Manhattan, and Canberra distance between each of three given objects: $x_a = (2, 5)$, $x_b = (10, 4)$, $and x_c = (5, 6)$.

14 (Dissimilarity measure II) Implement Canberra distance using Python.

Clustering

15 (Quality k-means) Modify the code of heterogeneity and homogeneity metrics to make it work with other methods than k-means.

16 (3-means) Modify the HCM code to work for three groups. This exercise can be divided into four tasks:

- modify the parameters,
- modify the `calculate_u` function,
- execute the clustering,
- plot the results.

17 (Density plot) For density clustering, plot the feature space with all elements marked with different colors depending on the cluster that it's assigned to. You should do the following tasks:

- fill the `get_color` method,
- fill the plot code.

18 (Plot dendogram) Build a method that plot is based on `dendrograms_history` and pydot, a dendrogram for the divisive clustering method. You should base on agglomerative method, but keep in mind that it works top-down instead of bottom-up. This exercise needs just one function to be implemented: `show_tree_divisive`. You should loop over the `dendrogram_history` variable and loop over child.

19 (s_2 metric) Implement the s_2 metric.

20 (Image segmentation borders) Draw the borders between clusters in the output image.

Shallow

21 (Ridge regression) Implement ridge regression using the equations shown in 4 and the example of Lasso regression implementation.

22 (Linear regression) Based on data given in previous exercise, calculate calories burned if we spent 3 hours running. Use linear regression for it.

23 (Logistic regression) Based on example 5, calculate the logistic regression values for:

$$y = [1010101],$$

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 0 & 3 \\ 1 & 1 & 4 \\ 1 & 1 & 1 \\ 1 & 0 & 4 \\ 1 & 1 & 0 \\ 1 & 2 & 4 \end{bmatrix}.$$

The values should be calculated for four iterations starting with a weights vector filled with 0.

24 (Extended Fisher classifier) Add one more feature from the Iris data set and reduce the dimensionality to 2 dimensions.

Decision trees

25 (CART decision tree) Rewrite the CART method to use Gini index as shown in the lecture

Gini index can be calculated with the following equation:

$$I_G(X) = 1 - \sum_{i=1}^{m} p_i^2, \tag{A.1}$$

and

$$I_G(\text{feature}) = \sum_{i=1}^{n} p_i * I_G(X_i). \tag{A.2}$$

You need to fill the `calculate_gini` function and change the "build" function a bit.

26 (Draw C4.5 tree) Use pydot do draw the tree for C4.5 example

27 (Implement the minimum number of objects pruning method) The MNO method checks the accuracy at each split and prune the node if the number of objects in a leaf is below a given value N. Use the CART method first.

28 (Plot OC1 tree) Instead of elements id, print the feature id it was split by. To make the task done, you need to change the function `build_level` and update the `BinaryLeaf` in two places to add the setters/getters and the feature and feature value split data.

SVM

29 (Implement the polynomial kernel) You need to extend the `build_kernel` function and implement the polynomial kernel if the `kernel_type` is set to `poly`. The equation that needs to be implemented:

$$K = (X^T * Y)^d. \tag{A.3}$$

30 (Implement a multiclass C-SVM) Use the classification method that we used in the notebook and IRIS data set to build a multiclass C-SVM classifier. Most implementation is about a function that will return the proper data set that need to be used for the prediction. You need to implement: `choose_set_for_label` and `get_labels_count`.

31 (One-class SVM) Implement one-class SVM using the cvxopt library.

32 (ν-SVM) Modify the C-SVM implementation to get the *nu*-SVM implementation.

Ensemble methods

33 (Stacking using different classifiers) Please use the following classifiers:

- Nearest Neighbors,
- Linear SVM,
- Decision Tree,
- Naive Bayes,
- QDA.

Find the best combination. Use the specific classifier only once.

34 (Modified boosting) Use the boosting method and change the code to fulfill the following requirements:

- the weights should be calculated as: $w_n^{(t+1)} = \frac{1+I(y_n \neq h_t(x_n)}{\sum_{i=1}^{N} 1+I(y_n \neq h_t(x_n)}$,
- the prediction is done with a voting method.

35 (RegionBoost) Change the original boosting method to add the regional objects (RegionBoost).

Neural networks

36 (Multilayer Perceptron) Combine a small network out of three perceptrons to solve the XOR (Table A.7).

37 (Stochastic Gradient Decent for MLP) Build a small MLP network of two hidden layers. Use the SGD implemented in the linear regression section for training the network.

Table A.7 XOR truth table

Input A	Input B	XOR
0	0	0
0	1	1
1	0	1
1	1	0

38 (ResNet used to recognize numbers) Use the scikit-learn digit data set (`load_digits()`) and the ResNetV2 network from the Keras to recognize the digits in the data set. You can use the code example from the Inception section.

Appendix B
Environment Setup

We prepared the environment in the simplest possible way. It can be setup using a Python environment or a Docker container.

Python configuration

There are a small number of libraries that need to be installed, including typical libraries such as NumPy, pandas, keras, or scikit-learn. Some other libraries are used for drawing like matplotplib or pydot.

```
numpy==2.2.0
matplotlib==3.10.0
simpful==2.12.0
pandas==2.2.3
pydot==3.0.3
jupyter==1.1.1
pillow==11.0.0
scikit-learn==1.6.0
tensorflow-datasets==4.9.7
cvxopt==1.3.2
keras==3.7.0
```

Listing B.1 Required Python libraries

System configuration

For convenience, the code can be run using a Docker image. The configuration of a Docker image is given in Listing B.2.

```
FROM ubuntu:24.04

ENV TZ=Europe/Warsaw
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/
    timezone

ENV LC_ALL=C.UTF-8
ENV LANG=C.UTF-8

RUN apt update && apt install -y python3-pip vim graphviz pip curl unzip
    build-essential unzip vim git curl wget zip
RUN apt-get update && apt-get upgrade -y && apt-get install -y  software-
    properties-common

```


K. Przystalski et al., *Pattern Recognition Primer*,
https://doi.org/10.1007/978-3-031-91816-2

```
RUN useradd -ms /bin/bash springer
RUN adduser springer sudo

COPY requirements.txt /home/springer/

WORKDIR /home/springer/

RUN pip3 install --break-system-packages -r requirements.txt

CMD jupyter lab --ip=0.0.0.0 --allow-root --NotebookApp.token='' --
    NotebookApp.password='' --no-browser --notebook-dir=/home/springer/
```

Listing B.2 Dockerfile configuration

It is an Ubuntu 24.04 LTS with Python and Jupyter. The Python packages are installed in same way as above.

Repository

The notebooks with code are stored in the GitHub repository: https://github.com/kprzystalski/pattern-recognition-primer/.