Machine Learning and Deep Learning with Python

Machine Learning and Deep Learning with Python

Theory and Applications

Mohd Halim Mohd Noor

Contents

Preface

Vast amount of data is being generated in various fields and this big data can be leveraged to improve business decision and process. Machine learning is a subfield of artificial intelligence which enables computers to learn from data and make predictions without being explicitly programmed. It has revolutionized many aspects of our life and transformed the way we approach various problems, ranging from robotics and computer vision to natural language processing and personalized recommendations.

The book provides the fundamental concept and tools of machine learning and deep learning as well as the practices to implement programs capable of learning from data. The book covers mathematical background including linear algebra and optimization, basic supervised learning such as linear regression and logistic regression and more advanced learning algorithms such as ensemble learning and deep learning, as well as unsupervised learning and dimensionality reduction. The practical aspects introduce the production-ready Python frameworks, Scikit-Learn and TensorFlow.

This book is suitable for undergraduate students and graduate students in computer science, statistics and electrical engineering and anyone who has the appropriate mathematical background. The readers should already be familiar with basic linear algebra, calculus, probability and computer programming. Prior exposure to Python is helpful but not necessary.

I would especially like to thank my wife Zuraidah, for always being on my side and unconditionally supporting me, my children for being the joy of my life and my parents for their love and support. My sincere thanks to the students of my Machine Learning course for their valuable feedback on an earlier draft of the book. I gratefully acknowledge the support of the School of Computer Sciences, Universiti Sains Malaysia in the completion of this book.

Chapter 1

Introduction

1.1 Artificial Intelligence and Machine Learning

Artificial intelligence (AI) is a field of study in computer science that focuses on developing machines which can behave intelligently. There are various approaches in AI design such as statistical (including machine learning), symbolic and evolutionary computation. Machine learning is a subset of AI, which focuses on a computer's ability to learn from observations (data) of the world. Deep learning is a subset of machine learning, which takes the idea one step further whereby the computer learns the pattern in the data without human intervention.

Machine learning is about learning from data. For example, in a scenario where we are asked by a physician to build a system that can classify brain cancer as benign or malignant brain cancer tumors in CT scan images. The physician needs to provide a set of images from different patients which is called a dataset. The dataset should contain images of brain cancer and images of normal brains. In addition to providing the data, the physician needs to "label" the images since we are not the domain expert, and we would not know which images are brains with cancerous tumors and which images are not. With the dataset, we feed the images together with the labels to the computer with a machine learning algorithm to learn to detect brain cancer by "seeing" all the images as shown in Figure 1.1.

This process is repeated until the computer has learned from the data, and a "set of rules" will be produced which allows the computer to detect brain cancer in CT scan images. This set of rules is the predictive model. The predictive model can be used by the physician to assist him/her in analyzing new CT scan images and generate the prediction (cancer or no cancer). The process of learning from the data to produce the set of rules is known as "training". Using the predictive model to perform prediction on new data is known as "testing".

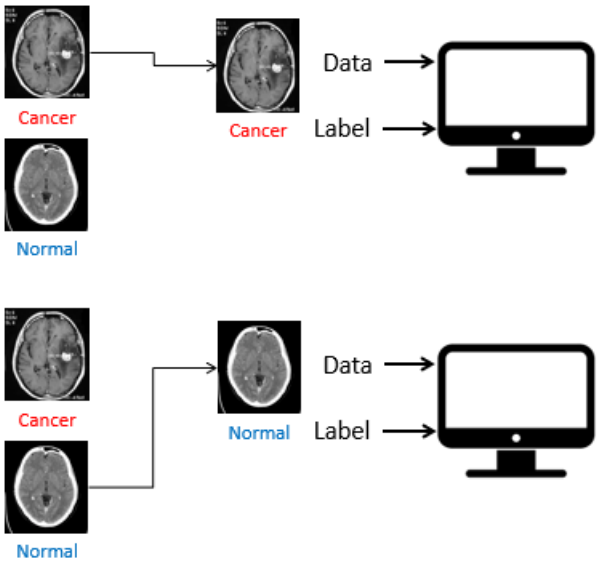


Figure 1.1 The images and the labels are fed together and the computer will learn to detect images with brain cancer

Machine learning was introduced way back in the 1950s. But the first formal definition (that I am aware of) was given by Tom Mitchell in his book *Machine Learning*, 1997 (Tom M. Mitchell 1997).

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Based on the definition, we know that machine learning is a field of study that concerns algorithms that allow computers to automatically improve through experience. Using the above scenario, the experience is the database of CT scan images (with labels) and the task is the classification of the images into cancer or normal (no cancer). In order to know if the computer is improving with experience, we have to evaluate the performance of the image classification. Therefore, performance measure allows us to quantify the ability of the computer to perform the classification.

Why machine learning is needed? Do we need machine learning in every prediction problem? Machine learning is not needed if the data can be easily analyzed and predicted. For example, if we are asked to estimate (predict) the resale value of a car into low, medium and high given two factors, the car's mileage and age. The rules to estimate the resale value could be as follows.

If mileage $<$ 100,000 km and age $\leq$ 10 years, then resale is high

If mileage $<$ 100,000 km and age $>$ 10 years, then resale is medium

If mileage $\geq$ 100,000 km and age $\leq$ 10 years, then resale is medium

If mileage $\geq$ 100,000 km and age $>$ 10 years, then resale is low

The rules could be designed because there are two factors only. However, solving the problem becomes infeasible if more factors are added to the dataset. For example, could we design a set of rules that can accurately estimate the resale value given the following factors?

1. Mileage
2. Age
3. Maker and model

4. Year of manufacture
5. Horsepower
6. Transmission mode
7. Fuel type
8. Service records
9. Condition of interior
10. Condition of exterior
11. Accident history
12. Modification

The task of designing a robust set of rules is almost impossible. Even if it is possible, the task will be time-consuming, expensive and the model will be difficult to maintain. Note that normally, real-world datasets are of high dimensions, complex and noisy. In some cases, the data is unstructured e.g. images, time series and texts which will increase the complexity of the system. For example, in the spam e-mail filter problem, the content of the e-mails needs to be analyzed and categorized into spam e-mail or non-spam e-mail. One way to categorize the e-mails is to identify the words that can be considered spam e.g. cash bonus, 100% free, extra income etc. in the e-mails. If any of the words are present, the e-mail is classified as spam e-mail. However, this method is not reliable and inaccurate because such e-mails may be spam to some users but others might find it interesting. Human face recognition using a camera is another example where it is infeasible to design the rules manually. To recognize human faces, facial features such as the shape and/or size of nose, eyes and mouth need to be identified and extracted, and the combination of the features can help us identify the person in the image. However, the number of feature combinations could be thousands or even tens of thousands. Even if we can define all the rules, the combination may not be exhaustive which may lead to subpar performance.

1.2 Types of Machine Learning

In general, machine learning can be divided into three types: supervised learning, unsupervised learning and reinforcement learning (Jordan and T. M. Mitchell 2015). In a supervised learning setting, the aim is to learn the mapping of the data (or the input) to the response of the data (or the output). For example, using the above scenario, given the CT scan images which are the data, we want to develop a mapping function (predictive model) that can perform the mapping of the image to the image's label which is either cancer or normal. Supervised learning works by searching for the optimal function among the many possible functions that best map between the inputs and output. This is done by making an assumption about the distribution of the data and fitting a function that best characterizes the data. Since the computer needs to learn the input-output mapping, each data in the dataset needs to be labeled. The labels allow the machine learning algorithm to guide its search for the optimal function.

In an unsupervised learning setting, the label of the data is unavailable. Since the dataset is unlabeled, it is more widely and easily applicable than supervised learning. Data labeling is time-consuming, expensive and requires human experts. The goal of unsupervised learning is to find interesting patterns in the data and group the data into clusters. Typically, the grouping is performed based on the similarity of the data to each other. For example, given a dataset about customers, the clustering analysis of the customers could be based on their shopping behavior such as the kind of products that have been purchased, the average expenses and the average quantity per order. The outcome of the clustering analysis could be used for a targeted marketing campaign or to improve customer service operations.

Reinforcement learning is a type of machine learning that is less commonly used. It is an approach that allows computers to learn based on occasional reward and punishment. For example, consider a robot learning to navigate through an environment to reach a destination. The robot explores the environment, and while exploring it might hit some obstacles and learn to avoid them. The process of learning is performed by rewarding the robot if it can avoid the obstacles and vice versa.

1.3 Python for Machine Learning

Python is an open source, high-level, interpreted and general purpose programming language, used by many data scientists for various data science and machine learning projects. Python is widely used in scientific and research communities due to its simple syntax and readability, making it accessible even to those without an engineering background. It has a lot of data science-related libraries which makes it more suited for quick prototyping.

This section introduces two essential libraries for machine learning: NumPy and Pandas. These libraries provide a computational foundation for general numerical data processing. NumPy provides fast and flexible multi-dimensional array, typically used for storing large datasets, while Pandas provides high-performance, easy-to-use data structures and operations for data manipulation and data analysis.

We can download Python from python.org. But for Windows operating system, it is recommended to install Anaconda distribution, which already includes Python and most of the libraries that we need to perform machine learning. Anaconda can be downloaded from anaconda.com. As of writing, the current version of Anaconda includes Python 3.9. Anaconda comes with the Jupyter Notebook, an interactive integrated development environment (IDE) that allows us to write, edit and share codes via web browser. It is recommended to create an isolated environment so that different projects can be developed without having conflicting library versions. We can create an isolated environment using the following command in the Anaconda Prompt (terminal). In this example, `my_env` is the name of the environment.

`conda create --name my_env` Once the environment is created, we activate the environment using the following command:

```
conda activate my_env
```

`my_env` is an empty environment and we need to install Python and the other libraries into the environment. Now, let us install Python into the created environment using the following command. The command will install the latest version. Note that conda will install the dependencies automatically. `conda install -c anaconda python`

We can install different versions of Python by specifying the version in the command. In this example, Python version 3.8 will be installed:

```
conda install -c anaconda python=3.8
```

Now, let us install Scikit-Learn and the other libraries that will be used in this book:

```
conda install -c conda-forge numpy pandas scikit-learn matplotlib
```

Scikit-Learn is introduced in the following chapter. Finally, we install Jupyter Notebook into the environment:

`conda install -c anaconda jupyter` The environment is now ready to be used. Launch Jupyter Notebook by clicking Jupyter Notebook's Launch button under Anaconda3 folder. We can also

launch Jupyter Notebook by typing the following command in the terminal:

```
jupyter notebook
```

We can specify the location of our working folder as follows:

```
jupyter notebook "C:\users\myuser"
```

This will launch a new browser window or a new tab showing the Notebook Dashboard as shown in Figure 1.2(a).

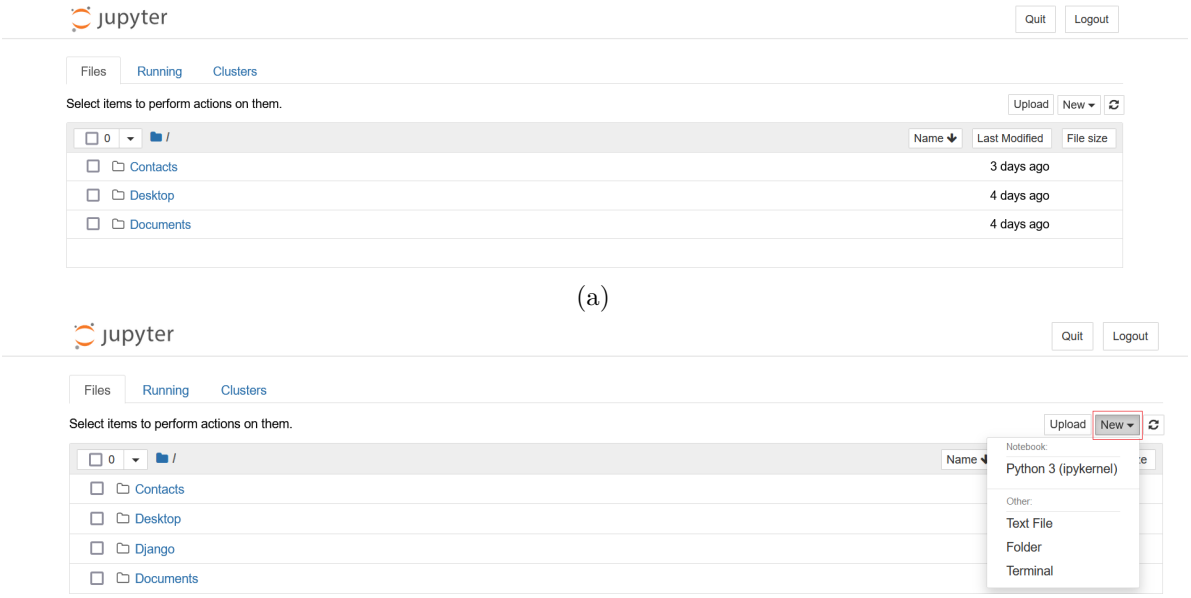


Figure 1.2 (a) Jupyter Notebook dashboard and (b) Dropdown menu in Jupyter Notebook dashboard

On the top right of the dashboard, there is a dropdown menu labeled "New" as shown in Figure 1.2(b). Click the Python version to create a new Notebook. Then, rename the Notebook either by clicking on the current name and editing it, or by choosing **Rename** under **File** in the top menu bar. For this example, we will name it "MyFirstNotebook".

1.3.1 NumPy Array

Now, let us import NumPy to create an array. In the first line of the Notebook, type:

```
import numpy as np

data1 = [2, 5, 4, 8, 0, 1]
arr1 = np.array(data1)
print(arr1)
```

```
[2 5 4 8 0 1]
```

First, we define a list of numbers. A list is simply a collection of data which in this case a sequence of integers. Then, we create an array using the list. We can create an array using `arange(a, b)`. It is a NumPy function that generates values within half-open interval `[a, b)`. This means the

interval includes `a` but excludes `b`. By default, the step size is 1. In this example, we generate a sequence of values from 0 to 19:

```
arr1 = np.arange(0, 20)
print(arr1)
```

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

We can specify a different step size as follows. In this example, the step size is set to 2:

```
arr1 = np.arange(0, 20, 2)
print(arr1)
```

```
[ 0  2  4  6  8 10 12 14 16 18]
```

Note that step size can be a floating point number such as 0.1. There are other functions for creating new arrays. `zeros` and `ones` create arrays of 0s and 1s respectively. In this example, we use `zeros` to create a two dimensional array by passing a tuple as the size of the array as follows:

```
arr2 = np.zeros((3,4))
print(arr2)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

In this example, we use `ones` to create a two dimensional array by passing a tuple as the size of the array as follows:

```
arr2 = np.ones((3,4))
print(arr2)
```

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

We can create a multidimensional array by manually specifying the elements. In this example, we create a two-dimensional array as follows:

```
data2 = [[3, 4.9, -2.5, 1.7], [-5.2, -3.1, 7.4, 3.6], [4.2, 6.8, -4.3, -5.7]]
arr2 = np.array(data2)
print(arr2)
```

```
[[ 3.   4.9 -2.5  1.7]
 [-5.2 -3.1  7.4  3.6]
 [ 4.2  6.8 -4.3 -5.7]]
```

The shape of the array will be inferred from the data. We can confirm the size of the array using the `ndim` and `shape` attributes. `ndim` provides the number of dimensions of the array while `shape`

provides the shape of the array as follows:

```
print(arr2.ndim)
print(arr2.shape)
```

```
2
(3, 4)
```

We can access the elements in the array using index and square bracket `[]`. In this example, we are accessing the third element in array `arr3` as follows:

```
data3 = [3, 6, 9, 12, 15, 18]
arr3 = np.array(data3)
print(arr3[2])
```

```
9
```

Note that, indexing in Python starts at 0. Therefore, the index of the third element is 2. The same index notation is used to access the elements in multidimensional array. But, for accessing elements in multidimensional array, we need to specify two indexes row,column. Figure 1.3 shows the index for each element.

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3

Figure 1.3 The index of the rows and columns of 3 × 4 array

In this example, we are accessing element at the first row and second column as follows:

```
data4 = [[1, 3, 5, 7], [2, 4, 6, 8], [3, 6, 9, 12]]
arr4 = np.array(data4)
print(arr4[0,1])
```

```
3
```

Slicing an array is similar to slicing a list. We need to specify `start:stop` in the square bracket `[]`. To slice multidimensional array, we need to specify two `start:stop`, each for row and column. In this example, we are selecting the first two rows and last three columns as shown in Figure 1.4:

```
print(arr4[0:2,1:4])
```

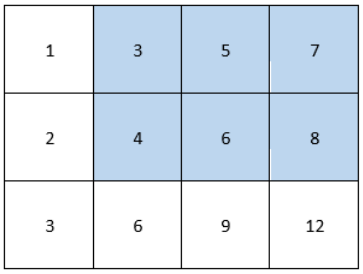


Figure 1.4 Retrieving elements of the first two rows and the last three columns

```
[[3 5 7]
 [4 6 8]]
```

NumPy enables batch operations on data or what is known as vectorization. Any arithmetic operations between equal-size arrays applies the operation element-wise. Let us create another array called `arr5`. We are going to use the `randn` function to generate some random normally distributed data as follows:

```
arr5 = np.random.randn(3,4)
print(arr5)
```

```
[[ 0.94194631  0.58996195 -0.03229058  0.25856654]
 [-0.27937617 -0.1477404   1.77243638 -1.92555553]
 [ 0.22847046 -1.16258011 -1.02645735  1.67708209]]
```

Now, let us perform addition operation between the `arr4` and `arr5` and store the result in `arr6`. We perform subtraction, multiplication and division using the following operators: `-`, `*` and `/`.

```
arr6 = arr4 + arr5
print(arr6)
```

```
[[ 1.94194631  3.58996195  4.96770942  7.25856654]
 [ 1.72062383  3.8522596   7.77243638  6.07444447]
 [ 3.22847046  4.83741989  7.97354265 13.67708209]]
```

Transposing an array is performed using the special `T` attribute as follows:

```
print(arr6.T)
```

```
[[ 1.94194631  1.72062383  3.22847046]
 [ 3.58996195  3.8522596   4.83741989]
 [ 4.96770942  7.77243638  7.97354265]
 [ 7.25856654  6.07444447 13.67708209]]
```

We can perform a matrix computation such as dot product using `dot` function. In this example, we perform dot product between `arr5` and `arr6`. We transpose `arr6` and perform the operation and store the result in `arr7` as follows:

```
arr7 = np.dot(arr6.T, arr5)
print(arr7)
```

```
[[ 2.08611801  -2.86188677  -0.32689754   2.60337559]
 [ 3.41052944  -4.07508157   1.7465579    1.62325451]
 [ 4.32960104  -7.4874254    5.4312373   -0.30948885]
 [ 8.26493422 -12.51586636  -3.5067584   13.11773168]]
```

NumPy also provides mathematical functions that compute statistics about an entire array or about the data along an axis such as summation (`sum`), average (`mean`), standard deviation (`std`) and variance (`var`). In this example, we calculate the average of the entire array as follows:

```
arr8 = np.random.randn(2,3)
print(np.mean(arr8))
```

```
-0.3457276111356699
```

We can perform mean of the data along an axis as follows:

```
print(np.mean(arr8, axis=0))
```

```
[-0.57580011 -0.00288407 -0.45849865]
```

```
print(np.mean(arr8, axis=1))
```

```
[-0.35447918 -0.33697604]
```

Here, `axis=0` means "compute mean across the row" while `axis=1` means "compute mean across the column". In many cases, we need to convert an array from one shape to another. NumPy uses row major order when reshaping arrays. In row major order, the consecutive elements of a row reside next to each other. In this example, we reshape `arr9` a one-dimensional array to two-dimensional array as shown in Figure 1.5.

```
arr9 =  np.arange(0,9)
print(arr9.reshape((3,3)))
```

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

We can let NumPy infers the dimension from the data. To do that, we specify -1 to infer the dimension of the column from the data as follows:

```
arr10 =  np.arange(0,12)
print(arr10.reshape((4,-1)))
```

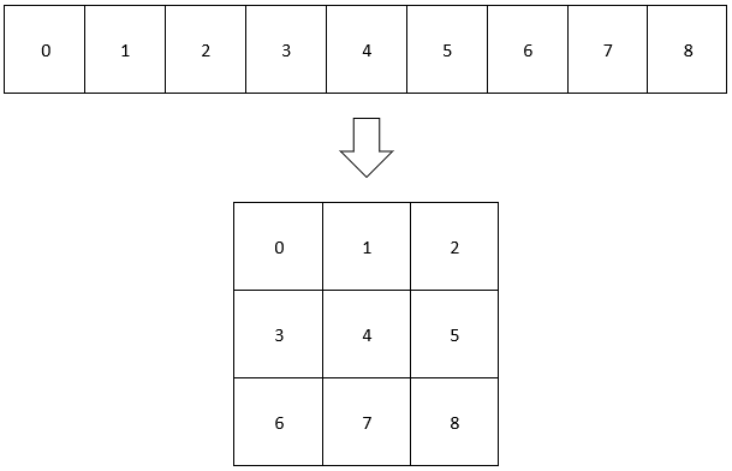


Figure 1.5 Reshaping 1d array to 2d array

```
[[0  1  2]
 [3  4  5]
 [6  7  8]
 [9 10 11]]
```

The opposite operation of reshaping is known as flattening. Here, we flatten `arr10` into a one-dimensional array as follows:

```
print(arr10.flatten())
```

```
[0  1  2  3  4  5  6  7  8  9 10 11]
```

1.3.2 Pandas DataFrame

Now, let us use Pandas to store and manipulate data. Pandas Series is a one-dimensional array-like object which is similar to NumPy array. However, unlike NumPy array, `Series` has an explicit index which can be changed to different values. Now, let us import Pandas module and create a `Series` from a list as follows:

```
import pandas as pd

data1 = [2, 5, 4, 8, 0, 1]
sr1 = pd.Series(data1)
print(sr1)
```

```
0    2
1    5
2    4
3    8
4    0
5    1
dtype: int64
```

Notice that the indexes are automatically generated and each element is associated with an index.

We can display the content of the `Series` without displaying the index as follows:

```
print(sr1.values)
```

```
[2 5 4 8 0 1]
```

We access the elements in a Series using index notation as follows:

```
print(sr1[2])
```

```
4
```

We can specify our own indexes for each element as follows:

```
sr2 = pd.Series([4, 7, 1, -2], index=['b', 'd', 'f', 'c'])
print(sr2)
```

```
b    4
d    7
f    1
c   -2
dtype: int64
```

Similar to NumPy array, we can perform element-wise arithmetic operations such as addition, subtraction, multiplication and division on a Series. The operation preserves the index-value link as follows:

```
print(sr2 * 2)
```

```
b     8
d    14
f     2
c    -4
dtype: int64
```

A `DataFrame` represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, etc.). The resulting `DataFrame` has its index assigned automatically as follows:

```
data = {'university': ['USM', 'UM', 'UPM', 'UKM', 'UTM', 'UiTM', 'UniMAP'],
        'year': [1969, 1905, 1931, 1970, 1972, 1956, 2001],
        'location': ['Pulau Pinang', 'Kuala Lumpur', 'Serdang', 'Bangi',
                     'Skudai', 'Shah Alam', 'Arau']}
fm1 = pd.DataFrame(data)
```

The head method can be used to select only the first five rows of the `DataFrame` as follows:

```
print(fm1.head())
```

```
  university  year           location
0        USM  1969       Pulau Pinang
1         UM  1905       Kuala Lumpur
2        UPM  1931  Serdang, Selangor
3        UKM  1970    Bangi, Selangor
4        UTM  1972      Skudai, Johor
```

A column can be retrieved by specifying the column name as follows:

```
print(fm1['university'])
```

```
0       USM
1        UM
2       UPM
3       UKM
4       UTM
5      UiTM
6    UniMAP
Name: university, dtype: object
```

Note that a column in a `DataFrame` is a `Series`. A row can be retrieved using a special `loc` attribute with the index. In this example, we are retrieving a row with index 3 as follows:

```
print(fm1.loc[3])
```

```
university                UKM
year                     1970
location      Bangi, Selangor
Name: 3, dtype: object
```

We can add a row to a `DataFrame` as follows:

```
fm1.loc[7] = {'year': 2000, 'university': 'UTeM', 'location': 'Durian Tunggal'}
```

We can drop a row from a `DataFrame` by specifying the index as follows:

```
fm1.drop(6)
```

Finally, we can save and load to and from disk in text format using Pandas. In this example, we save the content of `fm1` to a csv (text) file using `to_csv` method as follows:

```
fm1.to_csv('uni_in_Malaysia.csv', sep=';', header=True, index=False)
```

The first argument is the filename, the second argument is to specify the separator to separate the values, the third argument to write out the column names and the fourth argument is not to write the index. To load a csv file, we use `read_csv` method. In this example, we are reading the file that has been saved as follows:

```
fm3 = pd.read_csv('uni_in_Malaysia.csv', sep=';')
```

Exercises

1. Explain artificial intelligence and how machine learning is related to artificial intelligence.
2. Describe the types of artificial intelligence.
3. Give some real-world applications of artificial intelligence.
4. State the common programming languages used for machine learning.

Chapter 2

Supervised Learning

2.1 Introduction

Supervised learning is the most commonly used approach to machine learning. Some of the popular supervised learning applications are e-mail spam detection, optical character recognition and face recognition. In a supervised learning setting, we are given a labeled dataset or in other words, the dataset comes in pairs of $(\mathbf{x}^i, y^i)$, where $i = 1, 2, ..., N$ and N is the number of samples or instances in the dataset. Each input $\mathbf{x}^i$ is a d-dimensional vector, $\mathbf{x}^i = (x_1, x_2, \ldots, x_d)$ where each element, x_j is a feature describing the i-th input.

2.2 Feature Vector

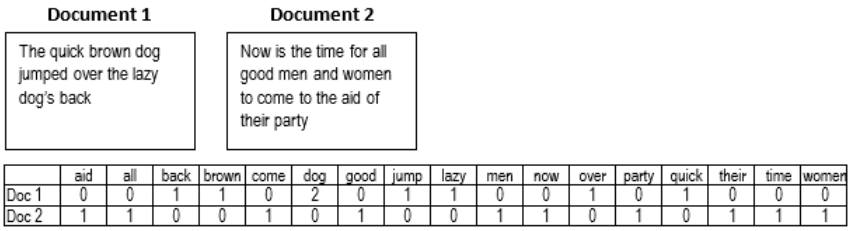
Let us consider a dataset consisting of N patient data as shown in Figure 2.1. The dataset is arranged in the form of a table whereby the columns are the features of the patients. A row represents a patient. This is known as structured data. The feature vector of i-th patient, $\mathbf{x}^i = (x_1, x_2, ..., x_d)$ where feature ${x_1}^i$ refers to i-th patient's name, ${x_2}^i$ refers to i-th patient's age, ${x_3}^i$ refers to i-th patient's gender, ${x_4}^i$ refers to i-th patient's height, etc. The features can be categorical or numerical.

Name	Age	Gender	Height (cm)	Weight (kg)	Fever	...
Patient 1 Name	43	Male	175.8	85.6	Yes	...
Patient 2 Name	38	Male	182.3	98.3	No	...
...	...		...	...	...	...
Patient N Name	50	Female	170.1	73.8	Yes	...

Figure 2.1 Patient data in table form

A dataset of N text documents such as e-mails, review etc. Text data is typically converted into a form which can be used as input to the machine learning models. One of the commonly used methods to represent text data is the bag-of-words. The bag-of-words converts the text data into a sequence of integers based on the occurrence of the words in the document as shown in Figure 2.2. A bag-of-words feature vector, $\mathbf{x}^i = (x_1, x_2, ..., x_d)$ where d is the number of words in the whole corpus, ${x_1}^i$ refers to the number of occurrences of word 1 in i-th document, ${x_2}^i$ refers to the number of occurrence of word 2 in i-th document, etc.

Time series data is a sequence of data that is ordered by time. For example, temperature sensors that are used to monitor the temperature of the environment generate measurements every 1 hour. The sequence of the values over time is important because it contains information such as the trend



	aid	all	back	brown	come	dog	good	jump	lazy	men	now	over	party	quick	their	time	women
Doc 1	0	0	1	1	0	2	0	1	1	0	0	1	0	1	0	0	0
Doc 2	1	1	0	0	1	0	1	0	0	1	1	0	1	0	1	1	1

Figure 2.2 Text data with its bag-of-words representation

and seasonality. A single sample (measurement) does not carry sufficient information to perform predictions. Therefore, for time series data, usually, we segment or group several samples, enough to contain the temporal information, and the segmentation is used for the prediction as shown in Figure 2.3. The subsequent segmentation may partially overlap the previous segmentation. The feature vector is a segmentation, $\mathbf{x}^i = (x_1, x_2, ..., x_d)$ where d is the size of segmentation, x^i_j refers to j-th sample in the i-th segmentation.

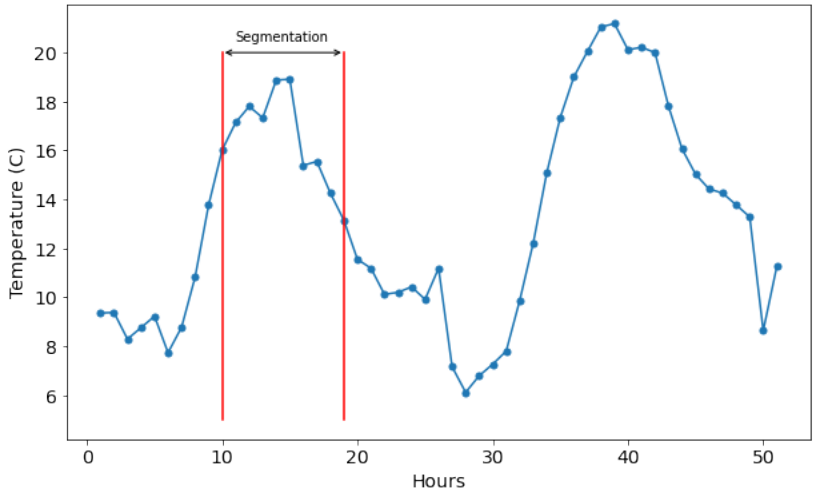


Figure 2.3 A segment of time series data is considered as an input for predictions

Finally, prediction with machine learning may involve image data. Image data is a collection of pixels arranged in rows and columns as shown in Figure 2.4. The arrangement of the pixels defines the visual information known as spatial structure. The image size determines the number of pixels. The image can be a grayscale image or a color image. For grayscale images, each pixel has one value representing the intensity of the pixel. The range of value is $[0, 255]$ whereby 0 represents black and 255 represents white. For color images, each pixel has three values for red, green and blue. The combination of the values defines the color of the pixel. The feature vector $\mathbf{x}^i = (x_{1,1}, x_{1,2}, x_{1,3}, ..., x_{d,1}, x_{d,2}, x_{d,3})$ where $x_{j,1}$, $x_{j,2}$ and $x_{j,3}$ refer to red, green and blue of j-th pixel and d is the number of pixels in the image.

2.3 Types of Supervised Learning

The output or response variable y^i can be either a categorical or a numerical variable. If the output is a categorical variable, y^i is a finite set, $y \in \{1, 2, ..., C\}$ where C denotes the number of labels (classes) such as low, medium and high. This problem is known as multiclass classification. In the case of $C = 2$, say yes and no, the problem is known as binary classification. If the output is a numerical value, $y \in \mathbb{R}$ such as house price and salary, the problem is known as regression. Table

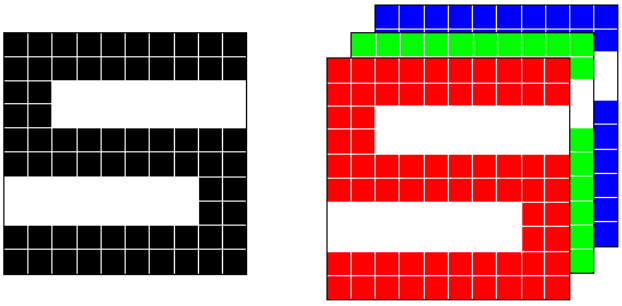


Figure 2.4 Each pixel value in a grayscale image is the intensity while each pixel value in a color image has three intensity values for red, green and blue

2.1 summarizes the types of supervised learning problems.

Table 2.1 The supervised learning problems depend on the type of target or response variable

Type	Output	Example
Binary classification	$y^i \in \{0, 1\}$ or $y^i \in \{-1, 1\}$	Detecting brain cancer (yes or no)
Multiclass classification	$y^i \in \{1, 2, ..., C\}$ where C is the number of classes and $C > 2$	Classifying vehicles (lorry, van, car and motorcycle)
Regression	$y^i \in \mathbb{R}$	Predicting (estimating) house price and salary

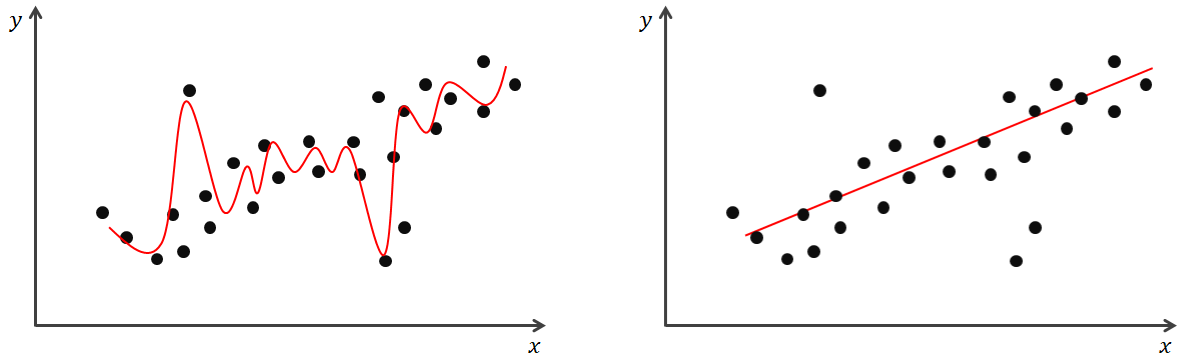
The data points are generated by an unknown function $y = f(\mathbf{x})$. The aim of supervised learning is to approximate (learn) the unknown function $\hat{f}$ given the dataset. With the approximated function, the output of a new data point $\mathbf{x}$ can be predicted $\hat{y} = \hat{f}(\mathbf{x})$. The approximated function is also called hypothesis or predictive model. Note that the two terms will be used interchangeably. To learn a function, we choose an algorithm such as decision tree, support vector machine or k-nearest neighbors. By choosing an algorithm, we are making an assumption about the data that we are trying to learn.

2.4 Overfitting and Underfitting

There are two problems that can be encountered when learning the hypothesis. The first problem is overfitting, whereby the hypothesis learns every minor variation in the data including the noise. This could easily happen when a highly flexible algorithm is used such as decision tree, support vector machine and neural network. Overfitting causes the hypothesis to become too specific, consequently failing to predict new data. Conversely, we do not want a hypothesis that fails to capture the variation in the data, which is known as underfitting. This could happen when a simple algorithm is used such as linear regression, logistic regression and linear discriminant analysis. The two problems are illustrated in Figure 2.5.

2.5 Loss Function

When approximating or learning the hypothesis $\hat{f}(\mathbf{x})$, we want a hypothesis that makes the least mistakes, which means the one with minimum incorrect classification or the best estimation. In machine learning, the mistakes or errors can be quantified using several loss functions. A loss



(a) (b)

Figure 2.5 (a) Overfitting is when every minor variation in the data is learned and (b) underfitting is when the pattern of the data is not learned

function tells how bad a hypothesis is in performing the prediction.

Zero-one loss is a function used to quantify incorrect classification. The loss function calculates the summation of every misclassification and normalizes the summation by the number of instances (Hastie et al. 2009). The zero-one loss is defined as follows:

$$J_{0-1}(\hat{f}) = \frac{1}{N}\sum_{i=1}^{N} I(y^i \neq \hat{f}(\mathbf{x})^i) \tag{2.1}$$

where y^i is the class label (ground truth) of instance i, $\hat{f}(\mathbf{x})^i$ is the predicted class of instance i and $I(c) = \begin{cases} 1 & \texttt{if } c \texttt{ is True} \\ 0 & \texttt{if } c \texttt{ is False} \end{cases}$

Another loss function for classification problems is the negative log-likelihood or also known as cross-entropy. The loss function measures the difference between the probability distribution of the response and the prediction (Bishop 2006). For binary classification, the loss function is defined as follows:

$$J_{nll}(\hat{f}) = -\frac{1}{N}\sum_{i=1}^{N}[y^i \log_e \hat{f}(\mathbf{x}^i) + (1-y^i)\log_e(1-\hat{f}(\mathbf{x}^i))] \tag{2.2}$$

where $y^i \in \{0,1\}$ is the class label or ground truth of instance i, $\hat{f}(\mathbf{x})^i$ is the predicted probability of instance i.

For multiclass classification whereby the number of classes $C > 2$, the loss for each class label c per instance i is calculated as follows:

$$J_{nll}(\hat{f}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_c^i \log_e \hat{f}_c(\mathbf{x}^i) \tag{2.3}$$

The squared loss function is typically used to calculate the errors in regression problems. The loss function calculates the squared difference between the response and the predicted value and normalizes it by the number of instances (Bickel and Doksum 2015). The squaring amplifies large prediction errors, thus encouraging the hypothesis not to make many mistakes. Conversely, if the prediction is close to being correct, the error will be very small, and little attention will be given to

further improving the prediction.

$$J_{sq}(\hat{f}) = \frac{1}{N}\sum_{i=1}^{N}(y^i - \hat{f}(\mathbf{x})^i)^2 \tag{2.4}$$

where y^i is the response (ground truth) of instance i, $\hat{f}(\mathbf{x})^i$ is the predicted value of instance i.

Often, we build a set of hypotheses using different algorithms. This is crucial because there is no single algorithm that will always give the best performance for every problem. This is the No Free Lunch Theorem. Given the loss function, we choose a hypothesis $\hat{f}$ from a set of hypotheses, H that minimizes the loss function.

$$\hat{f} = \arg\min_{\hat{f}\in H} J(\hat{f}) \tag{2.5}$$

2.6 Performance Metrics

Besides the error, specifically in classification problems, we can evaluate the performance of the hypothesis by analyzing the number of correct and incorrect classifications in a specific table form called confusion matrix. Table 2.2 shows a table called confusion matrix for a binary classification problem, where the two classes are Yes and No. The number of correctly classified for class Yes is represented by True Positive while the number of correctly classified for class No is represented by True Negative. The number of incorrect classifications is represented by False Negative and False Positive. False Negative is the number of instances belonging to Yes which are misclassified as No. False Positive is the number of instances belonging to No which are misclassified as Yes.

Table 2.2 A confusion matrix

	Yes	No
Yes	True Positive (TP)	False Negative (FN)
No	False Positive (FP)	True Negative (TN)

Several performance metrics can be derived from the confusion matrix. Here we look at the four commonly used metrics. The first metric is called Recall. Recall is the fraction of relevant instances that are correctly classified. It measures the ability of the hypothesis or the predictive model to predict positive instances. Recall is defined as follows:

$$\texttt{Recall} = \frac{TP}{TP + FN} \tag{2.6}$$

The second commonly used metric is Precision. Precision is the fraction of relevant instances among the classified instances. It measures the predictive model's accuracy in predicting positive instances. Precision is defined as follows:

$$\texttt{Precision} = \frac{TP}{TP + FP} \tag{2.7}$$

The F-score measure is derived from recall and precision. Specifically, F-score is the harmonic mean of precision and recall and is defined as follows:

$$\texttt{F-score} = 2 \cdot \frac{\texttt{Recall} \cdot \texttt{Precision}}{\texttt{Recall} + \texttt{Precision}} \tag{2.8}$$

Lastly, the accuracy measures the overall correctness of the predictive model. The metric is defined as follows:

$$\texttt{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.9}$$

For regression problems, the mean squared loss (MSE) is typically used to evaluate the hypothesis. In addition to MSE, mean absolute error (MAE) is also used in model evaluation. The metrics are defined as follows:

$$\texttt{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y^i - \hat{f}(\mathbf{x}^i))^2 \tag{2.10}$$

$$\texttt{MAE} = \frac{1}{N}\sum_{i=1}^{N}|y^i - \hat{f}(\mathbf{x}^i)| \tag{2.11}$$

The absolute difference between response and the predicted value is taken to avoid the canceling of positive and negative magnitude. The metric tells us how far the hypothesis prediction deviates from the response. Another commonly used performance metric is the R-squared ($\texttt{R}^2$). $\texttt{R}^2$ measures the goodness of fit of the hypothesis. It measures the ratio of the hypothesis error to the worst or baseline error which is the average of the response. $\texttt{R}^2$ can be written as follows:

$$\texttt{R}^2 = 1 - \frac{\texttt{SS}_{res}}{\texttt{SS}_{tot}} \tag{2.12}$$

where $\texttt{SS}_{res}$ is the sum of squared error of the hypothesis and $\texttt{SS}_{tot}$ is the sum of squared error of the baseline. The errors are defined as follows:

$$\texttt{SS}_{res} = \sum_{i=1}^{N}(y^i - \hat{f}(\mathbf{x}^i)) \tag{2.13}$$

$$\texttt{SS}_{tot} = \sum_{i=1}^{N}(y^i - \bar{y}) \tag{2.14}$$

where $\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y^i$

$\texttt{R}^2$ provides a measure in the range of 0 and 1. $\texttt{R}^2 = 1$ indicates a perfect hypothesis while $\texttt{R}^2 = 0$ indicates a worst possible hypothesis.

2.7 Training, Validation and Test Sets

While it is important for a hypothesis to perform well on training data, it must also be able to generalize to new or unseen data. The ability of the hypothesis to perform well on unseen data is known as generalization. To achieve generalization, the dataset is split into three subsets, namely training set, validation set and test set. This method is called hold-out method. A portion of the data is hold-out for validation and evaluation. The training set will have a larger portion of the instances, typically around 60%–80% of the dataset. The remaining instances are split into validation and test sets as shown in Figure 2.6. The dataset can also be split first into training and test sets. Then, the training set is further split into training and validation sets. The split ratio

varies from one application or problem to another. The commonly used split ratios are 6:2:2, 7:1:2 and 8:1:1.

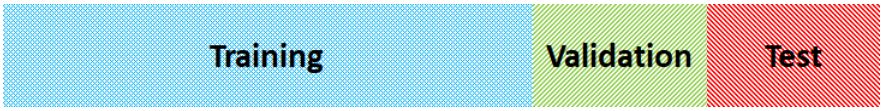


Figure 2.6 The dataset is split into training, validation and test sets

The training set is used to learn the hypothesis. The machine learning algorithms have several parameters that need to be set and fine-tuned to obtain an optimal hypothesis. To find the optimal parameters, the validation set is used to validate the hypothesis. If the loss is too large, the hypothesis is retrained with different parameters using the training set, and the newly trained hypothesis is validated using the validation set. Once an optimal hypothesis is obtained, the test set is used to evaluate the performance of the hypothesis. The test loss is the generalization errors.

Ideally, if we have sufficient data, a portion of instances can be set aside for validation. But data is often scarce and we do not have sufficient training and validation instances to make a reliable estimate of the error. A popular method to solve this problem is cross-validation where the dataset is split into training and test sets. Then, the cross-validation is employed to train the hypothesis using the training set. We split the data into K partitions of equal-sized, say $K = 5$ as shown in Figure 2.7.

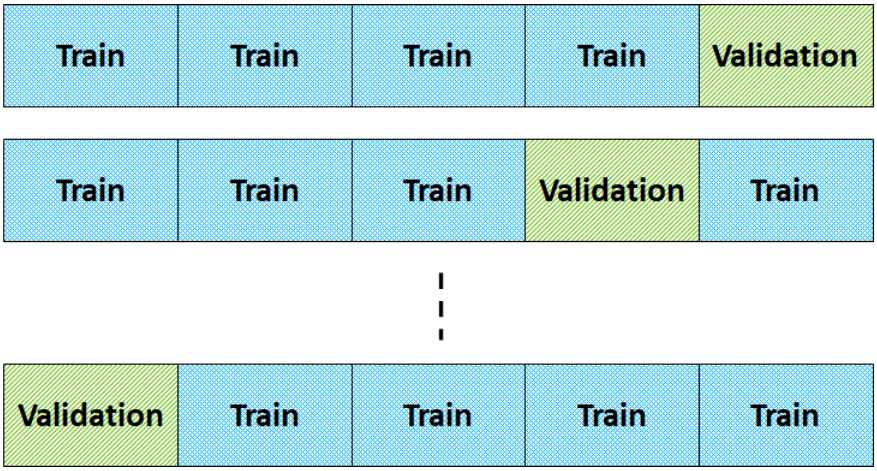


Figure 2.7 K-fold cross-validation, the training set is split into K partitions

The hypothesis is learned using $K-1$ partitions and the k-th partition is used as the validation set. We repeat this for $k = 1, 2, ..., K$ and the error estimate is the average error over K repetition. The common value of K is 3, 5 and 10. When $K = N$, this is known as leave-one-out cross-validation whereby all instances except the k-th instance are used to learn the hypothesis.

In summary, the dataset is split into training set, D_{tr}, validation set, D_{vd} and test set, D_{te}. A set of hypotheses, H are trained using the D_{tr}.

$$\text{Learning} : \hat{f} = \arg\min_{\hat{f} \in H} \frac{1}{|D_{tr}|} \sum_{i=1}^{|D_{tr}|} l(y^i, \hat{f}(\mathbf{x}^i)) \tag{2.15}$$

where l is the loss function to calculate the loss value or errors between the ground truth y and the prediction $\hat{f}(\mathbf{x}^i)$.

Choose a hypothesis that minimizes the validation loss.

$$\text{Validating} : \hat{f} = \arg\min_{\hat{f} \in H} \frac{1}{|D_{vd}|} \sum_{i=1}^{|D_{vd}|} l(y^i, \hat{f}(\mathbf{x}^i)) \tag{2.16}$$

Evaluate the chosen hypothesis, $\hat{f}^*$ on the test set and calculate the generalization error, ϵ.

$$\text{Test} : \epsilon = \frac{1}{|D_{te}|} \sum_{i=1}^{|D_{te}|} l(y^i, \hat{f}(\mathbf{x^i})) \tag{2.17}$$

2.8 Working with Real Data

In this section, we will work through a machine learning project end to end, starting with loading a dataset to evaluating a predictive model. It is best to work with real world data when learning about machine learning. There are many real world open datasets available on public repository, ranging across various domains such as natural language processing, education and healthcare. Here are some of the popular sources of machine learning datasets.

1. Kaggle datasets
2. UC Irvine Machine Learning Repository
3. WorldData.AI
4. VisualData
5. CMU Libraries

We will be using the prognosis of heart disease dataset which can be downloaded from UC Irvine Machine Learning Repository. The data was collected from 303 consecutive patients referred for coronary angiography at the Cleveland Clinic between May 1981 and September 1984. The dataset contains 76 attributes, but all we will be using a subset of 14 features. The objective is to predict the presence of heart disease or not. We can use our browser to download the dataset file. But, let us use Python to download the dataset using `requests` module. First, we import `requests` as follows:

```
import requests
```

Here are the URLs of the dataset. The first URL is the link to the data and the second URL is the documentation of the dataset.

```
data_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases
/heart-disease/processed.cleveland.data'
data_readme = 'https://archive.ics.uci.edu/ml/machine-learning-databases
/heart-disease/heart-disease.names'
```

To download the dataset using `requests`, we use the get method and specify the URL as the parameter. Then, we save the content to a file as follows:

```
r = requests.get(data_url, allow_redirects=True)
open('data_df.csv', 'wb').write(r.content)
```

We do the same step to download the readme file as follows:

```
r = requests.get(data_readme, allow_redirects=True)
open('readme.txt', 'wb').write(r.content)
```

The readme file describes the columns of the dataset. The first column is the output or response variable and the rest of the columns are the features. Let us create a list of the column names as follows:

```
columns = ['age', 'sex', 'cp', 'trestbps', 'chol', 'fbs', 'restecg', 'thalach',
           'exang', 'oldpeak', 'slope', 'ca', 'thal', 'heart_disease']
```

We will be using Pandas to read the csv file as follows:

```
import pandas as pd
data_df = pd.read_csv('data_df.csv', sep=',', header=None)
print(data_df.head())
```

```
     0    1    2      3      4    5    6      7    8    9   \
0  63.0  1.0  1.0  145.0  233.0  1.0  2.0  150.0  0.0  2.3
1  67.0  1.0  4.0  160.0  286.0  0.0  2.0  108.0  1.0  1.5
2  67.0  1.0  4.0  120.0  229.0  0.0  2.0  129.0  1.0  2.6
3  37.0  1.0  3.0  130.0  250.0  0.0  0.0  187.0  0.0  3.5
4  41.0  0.0  2.0  130.0  204.0  0.0  2.0  172.0  0.0  1.4

    10   11   12  13
0  3.0  0.0  6.0   0
1  2.0  3.0  3.0   2
2  2.0  2.0  7.0   1
3  3.0  0.0  3.0   0
4  1.0  0.0  3.0   0
```

We set `header` to `None` since the first row of the csv file is not the feature names but a patient record. Then the list of columns is assigned to `data_df` to rename the columns as follows:

```
data_df.columns = columns
print(data_df.head())
```

```
    age  sex   cp  trestbps   chol  fbs  restecg  thalach  \
0  63.0  1.0  1.0     145.0  233.0  1.0      2.0    150.0
1  67.0  1.0  4.0     160.0  286.0  0.0      2.0    108.0
2  67.0  1.0  4.0     120.0  229.0  0.0      2.0    129.0
3  37.0  1.0  3.0     130.0  250.0  0.0      0.0    187.0
4  41.0  0.0  2.0     130.0  204.0  0.0      2.0    172.0

   exang  oldpeak  slope   ca thal  heart_disease
0    0.0      2.3    3.0  0.0  6.0              0
1    1.0      1.5    2.0  3.0  3.0              2
2    1.0      2.6    2.0  2.0  7.0              1
3    0.0      3.5    3.0  0.0  3.0              0
4    0.0      1.4    1.0  0.0  3.0              0
```

Each row represents one patient. As we can see there are 10 columns including the response variable (`heart_disease`). Pandas uses NumPy data types (`dtypes`) for Series and columns of a DataFrame. NumPy supports `bool` (boolean), `float` (floating-point numbers), `int` (integer), `timedelta64` (time difference) and `datetime64` (date and time). Pandas stores string using `StringDtype` and `object`. `StringDtype` is a dedicated data type for string while `object` can hold any Python object, including strings. Pandas recommends using `StringDtype` to store text data.

```
print(data_df.dtypes)
```

```
age              float64
sex              float64
cp               float64
trestbps         float64
chol             float64
fbs              float64
restecg          float64
thalach          float64
exang            float64
oldpeak          float64
slope            float64
ca                object
thal              object
heart_disease      int64
dtype: object
```

Pandas automatically infer the data types of each column. As can be seen in the list, all columns are inferred as `float64` except column `ca`, verb|thal| and `heart_disease` which are inferred as `object` and `int64`. However, we know that according to the readme, column `sex`, `cp`, `fbs`, `restecg`, `exang`, `slope`, `ca`, `thal` and `heart_disease` are categorical while column `trestbps`, `chol`, `thalach` and `oldpeak` are floating-point features (numerical).

Now, let us fix the data type of the features as follows:

```
data_df['sex'] = data_df['sex'].astype('category')
data_df['cp'] = data_df['cp'].astype('category')
data_df['fbs'] = data_df['fbs'].astype('category')
data_df['restecg'] = data_df['restecg'].astype('category')
data_df['exang'] = data_df['exang'].astype('category')
data_df['slope'] = data_df['slope'].astype('category')
data_df['ca'] = data_df['ca'].astype('category')
data_df['thal'] = data_df['thal'].astype('category')
data_df['heart_disease'] = data_df['heart_disease'].astype('category')
```

```
print(data_df.dtypes)
```

```
age              float64
sex             category
cp              category
trestbps         float64
chol             float64
fbs             category
restecg         category
thalach          float64
exang           category
oldpeak          float64
slope           category
ca              category
thal            category
heart_disease   category
dtype: object
```

Let us generate the descriptive statistics of the dataset using describe() as follows:

```
print(data_df.describe())
```

```
              age    trestbps        chol     thalach     oldpeak
count  303.000000  303.000000  303.000000  303.000000  303.000000
mean    54.438944  131.689769  246.693069  149.607261    1.039604
std      9.038662   17.599748   51.776918   22.875003    1.161075
min     29.000000   94.000000  126.000000   71.000000    0.000000
25%     48.000000  120.000000  211.000000  133.500000    0.000000
50%     56.000000  130.000000  241.000000  153.000000    0.800000
75%     61.000000  140.000000  275.000000  166.000000    1.600000
max     77.000000  200.000000  564.000000  202.000000    6.200000
```

The method calculates the commonly used statistics such as mean, std, the lower and upper percentile, maximum and minimum. By default, the method displays columns with numerical data only. That is why categorical features (columns) are not shown in the descriptive statistics.

To select pandas categorical columns, set parameter `include` to category as follows:

```
print(data_df.describe(include='category'))
```

```
          sex     cp    fbs  restecg  exang  slope   ca thal  \
count   303.0  303.0  303.0    303.0  303.0  303.0  303  303
unique    2.0    4.0    2.0      3.0    2.0    3.0    5    4
top       1.0    4.0    0.0      0.0    0.0    1.0  0.0  3.0
freq    206.0  144.0  258.0    151.0  204.0  142.0  176  166

  heart_disease
            303
              5
              0
            164
```

It will show statistics such as `count`, `unique`, `top` and `freq`. The `top` is the most common value.

The `freq` is the most common value's frequency.

2.9 Using Scikit-Learn

In this section, we will introduce Scikit-Learn, an easy to use machine learning library for Python. Scikit-Learn provides end-to-end tools from data pre-processing to model evaluation and is designed to interoperate with NumPy and Pandas (Buitinck et al. 2013). In Scikit-Learn, all objects share a common three simple interfaces: estimator, predictor and transformer. The estimator interface is used for building a predictive model. An estimator is an object which has a method called `fit` that accepts training data for learning a model. Other machine learning tasks such as normalization, feature selection and dimensionality reduction are implemented as estimator.

The predictor interface is a method of an estimator object. The interface is used by estimators of machine learning models to make predictions. The method is called `predict`, accepts test data and produces the prediction based on the learned model. Finally, some estimators such as normalization and feature selector are capable of making data transformation. The interface is implemented as a method of the estimator called `transform`. The method accepts the data and returns the transformed version of the data.

2.9.1 Splitting Dataset

Now, let us use Scikit-Learn to split the dataset into training and test sets. The function to split a dataset is called `train_test_split`. Given a dataset, the function splits it into two partitions. First, we import the function as follows:

```
from sklearn.model_selection import train_test_split
```

We take the target column and then drop the target column. Using the helper function, we split the dataset with a ratio of 70%:30%. There are several parameters that we have to pass the function. The parameters are feature data (`X`), target (`y`) and test set size (`test_size`). The value of size should be between 0.0 and 1.0 which represents the proportion of test set. We can also define the size of the training set by specifying the value of parameter `train_size`. The random state is the parameter to control the random number generator used to split the dataset. We set its value to ensure the same split can be reproduced.

```
y = data_df['heart_disease']
X = data_df.drop(columns=['heart_disease'])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=50)
```

Let us display the size of each set as follows:

```
print(X_train.shape)
print(X_test.shape)
```

```
(212, 13)
(91, 13)
```

2.9.2 Data Visualization

We have seen the data to get a general understanding of the dataset. Now we want to analyze further to get more insight. We have put the test set aside and we will be only exploring the training set. Pandas provides a plotting function that uses the Matplotlib library to visualize the data. Let us use scatter plot to visualize `age` against `trestbps` as shown in Figure 2.8. The `figsize` is used to set the figure size.

```
X_train.plot(kind='scatter', x='age' , y='trestbps', figsize=(9,6))
```

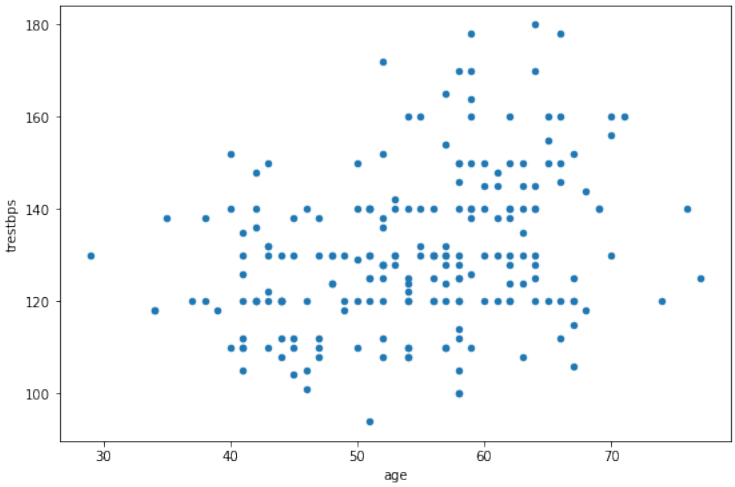


Figure 2.8 The output of plotting age against trestbps

We can set the alpha to see the regions with a high density of data points. Let us set it to 0.4 as follows:

```
X_train.plot(kind='scatter', x='age' , y='trestbps', alpha=0.4, figsize=(9,6))
```

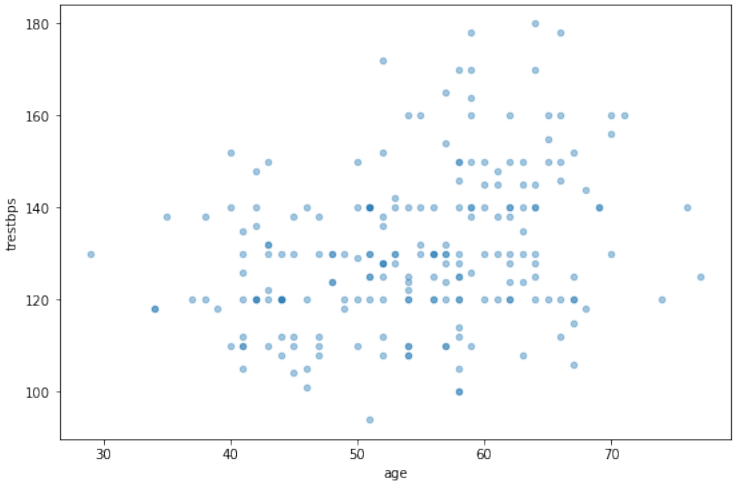


Figure 2.9 The high-density regions are highlighted

We can see from the plot as shown in Figure 2.9 that there are more patients aged between 45 to 65 years and their resting blood pressure is above 120 mmHg.

We can set the size of the data points based on one of the features in the training set. In Figure 2.10, we use `restecg` values to set parameter `s` which is the parameter to define the marker size. The marker size is the value of `restecg` plus a constant value and to the power of 3. The constant value is to avoid the marker size to be zero and the power of 3 is to increase the radius of the data points.

```
X_train.plot(kind='scatter', x='age' , y='trestbps',
             s=(X_train['restecg'].astype('int')+2)**3,
             alpha=0.4, figsize=(9,6))
```

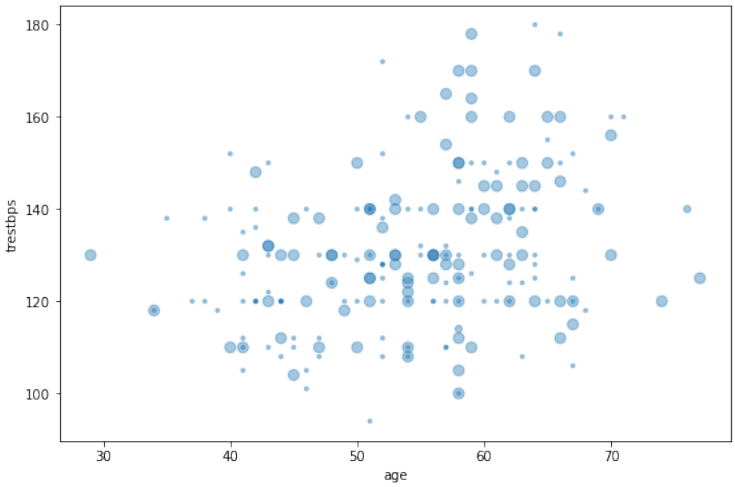


Figure 2.10 The size of the data points are defined by their corresponding restecg values

We can also set the color of the data points based on the target (`heart_disease`) by setting the parameter `c` as follows:

```
X_train.plot(kind='scatter', x='age' , y='trestbps',
             s=(X_train['restecg'].astype('int')+2)**3,
             alpha=0.4, c=y_train, colormap='bwr', figsize=(10,6))
```

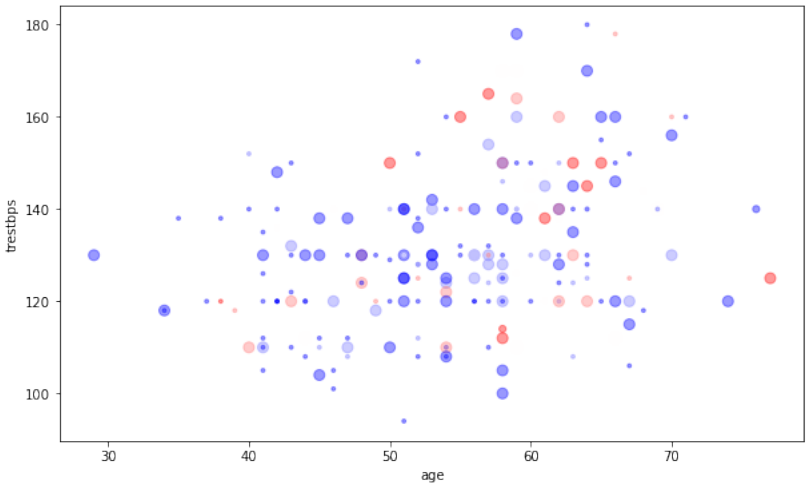


Figure 2.11 The data points are colored based on their target classes

We use a predefined colormap `bwr` which ranges from blue to red. Figure 2.11 shows the scatter plot.

The `info` method is useful to get a quick description of the data, in particular the total number of rows, each the data type of the columns (features) and the number of non-null values.

```
X_train.info()
```

```
Int64Index: 212 entries, 121 to 176
Data columns (total 13 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       212 non-null    float64
 1   sex       212 non-null    category
 2   cp        212 non-null    category
 3   trestbps  212 non-null    float64
 4   chol      212 non-null    float64
 5   fbs       212 non-null    category
 6   restecg   212 non-null    category
 7   thalach   212 non-null    float64
 8   exang     212 non-null    category
 9   oldpeak   212 non-null    float64
 10  slope     212 non-null    category
 11  ca        212 non-null    category
 12  thal      212 non-null    category
dtypes: category(8), float64(5)
memory usage: 12.6 KB
```

2.9.3 Handling Missing Values

There are 212 records or instances and non-null count for each column is 212. However, there are missing values as indicated by the readme. The missing values are denoted by `?`. We can use `unique` method to determine if there are `?` in the dataset. In this example, we use the method to check column `thal` as follows:

```
X_train['thal'].unique()
```

```
['7.0', '3.0', '6.0', '?']
Categories (4, object): ['7.0', '3.0', '6.0', '?']
```

As we can see, there are four labels for the column including `?`. In Python, a missing value is represented either by None or `np.nan`. Let us replace `?` with `np.nan`. First, we import `numpy`. Then, we use `replace` method to replace the missing values with `np.nan`. Note that we can specify `inplace = True` instead of using assignment (=) operator.

```
import numpy as np
X_train = X_train.replace('?', np.nan)
```

As we can see from the output of `info`, non-null count for column 12 is less than 169 indicating there is a missing value.

```
X_train.info()
```

```
Int64Index: 212 entries, 121 to 176
Data columns (total 13 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       212 non-null    float64
 1   sex       212 non-null    category
 2   cp        212 non-null    category
 3   trestbps  212 non-null    float64
 4   chol      212 non-null    float64
 5   fbs       212 non-null    category
 6   restecg   212 non-null    category
 7   thalach   212 non-null    float64
 8   exang     212 non-null    category
 9   oldpeak   212 non-null    float64
 10  slope     212 non-null    category
 11  ca        208 non-null    category
 12  thal      210 non-null    category
dtypes: category(8), float64(5)
memory usage: 12.6 KB
```

Let us use `fillna` to fill the missing value. In this example, we fill the missing value with the next valid value (after the missing value) as follows:

```
X_train['thal'] = X_train['thal'].fillna(method='backfill')
```

We can also fill the missing value with the previous valid value as follows:

```
X_train['ca'] = X_train['ca'].fillna(method='ffill')
```

As we can see, the number of non-null is 212 for all columns, indicating the missing values have been filled.

```
X_train.info()
```

```
Int64Index: 212 entries, 121 to 176
Data columns (total 13 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   age       212 non-null    float64
 1   sex       212 non-null    category
 2   cp        212 non-null    category
 3   trestbps  212 non-null    float64
 4   chol      212 non-null    float64
 5   fbs       212 non-null    category
 6   restecg   212 non-null    category
 7   thalach   212 non-null    float64
 8   exang     212 non-null    category
 9   oldpeak   212 non-null    float64
 10  slope     212 non-null    category
 11  ca        212 non-null    category
 12  thal      212 non-null    category
dtypes: category(8), float64(5)
memory usage: 12.6 KB
```

2.9.4 Normalization and Standardization

We need to scale the data since the range of the features varies widely. For example, the range of `trestbps` is 94 to 200 while the range of `oldpeak` is 0 to 6.2. Machine learning algorithms do not perform well when the range of the data heavily differs. There are two commonly used feature scaling methods: normalization and standardization. Normalization rescales the values to be in the range of 0 to 1. Standardization shifts the values to have a mean of 0 and standard deviation of 1. Scikit-Learn provides modules called `MinMaxScaler` and `StandardScaler` for normalization and standardization respectively.

Let us import `MinMaxScaler` for normalizing the features as follows:

```
from sklearn.preprocessing import MinMaxScaler
```

First, we create an object of `MinMaxScaler`. Then, we fit the scaler with the training data to estimate the minimum and maximum values of the features. Finally, we apply the scaler to normalize the training data by calling transform function as follows:

```
scaler = MinMaxScaler()
scaler.fit(X_train)
X_train_norm = scaler.transform(X_train)
```

By default, `MinMaxScaler` rescales the features into the range of [0,1]. Notice that the function returns a NumPy array. A different range can be specified by passing a tuple to `feature_range` parameter when creating the object of `MinMaxScaler`.

```
scaler = MinMaxScaler(feature_range=(0,5))
```

There is no missing value in test set. Thus, we apply the scaler to normalize the test data as follows:

```
X_test_norm = scaler.transform(X_test)
```

2.9.5 Training and Testing

Now that everything is prepared, we are ready to select and train a machine learning model. For this example, we will be using a dummy model. The model always predicts the most frequent class label in `y_train`. Let us import `DummyClassifier` as follows:

```
from sklearn.dummy import DummyClassifier
```

First, we create an estimator object of `DummyClassifier`. Then, the `fit` method of the estimator is used to train the predictive model. We pass the training features and target to `fit` method as follows:

```
model = DummyClassifier(strategy='stratified')
model.fit(X_train_norm, y_train)
```

Then, let us use the trained model to predict the test set. To produce the prediction, we use `predict` method with the test set as follows:

```
y_pred = model.predict(X_test_norm)
```

Now that we have the predicted labels, we can calculate the loss value and the accuracy of the model. Scikit-Learn provides the functions to compute the loss and the accuracy given the true labels and predicted labels.

```
from sklearn.metrics import zero_one_loss, accuracy_score
```

The functions are used by passing the true values and the predicted values as parameters. The loss value indicates that 63.7% of the instances are misclassified.

```
loss_val = zero_one_loss(y_test, y_pred)
print(loss_val)
```

```
0.6593406593406593
```

The accuracy of the prediction is given below.

```
accuracy = accuracy_score(y_test, y_pred)
print(accuracy)
```

```
0.34065934065934067
```

Scikit-Learn provides a function to print out the report of the classification whereby it shows the recall, precision and F-score for each class. Please note that the results in the book could be different from yours due to different parameter settings.

```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.54      0.54      0.54        46
           1       0.08      0.05      0.06        20
           2       0.20      0.30      0.24        10
           3       0.18      0.15      0.17        13
           4       0.00      0.00      0.00         2

    accuracy                           0.34        91
   macro avg       0.20      0.21      0.20        91
weighted avg       0.34      0.34      0.34        91
```

The classification report shows the recall, precision and F-score measures for each class. The number of instances for each class is indicated by support. The bottom part of the report shows the accuracy of the model and the average of the precision, recall and F-score. Classification report provides two types of average value: macro average (averaging the unweighted mean per class label) and weighted average (averaging the support-weighted mean per class label). Macro average is the average value of the metric without considering the weights (the proportion of the instances) of each class while weighted average is the weighted average of the metric based on the proportion of the instances of each class. Finally, the classification report provides the model accuracy which indicates the overall performance of the model.

Exercises

1. Describe the differences between supervised learning and unsupervised learning.
2. Give examples of real-world applications of supervised learning.
3. Give examples of real-world applications of unsupervised learning.
4. Describe cross-validation.
5. Explain the trade-off between bias and variance.
6. Imagine you work at a real estate company and are tasked with estimating house prices. You decide to address this problem in a data-driven manner and develop a model that predicts adequate market prices from houses' properties.
   (a) Characterize the task of prediction: supervised or unsupervised? Regression or classification? Justify your answers.
   (b) How would you set up your data? State potential features along with their respective data type and state the target variable.
   (c) State the loss function for evaluating the predictive model.

Chapter 3

Linear Models for Regression

3.1 Introduction

A regression problem is when the response variable is continuous, $y = \mathbb{R}$ such as salary, height, price etc. Consider a regression problem of predicting the price of a house given its built-up area. The data is given in Figure 3.1. The built-up area is the input feature, x and the price is the response variable, y which can take values ranging from RM100,000 to RM1,300,000.

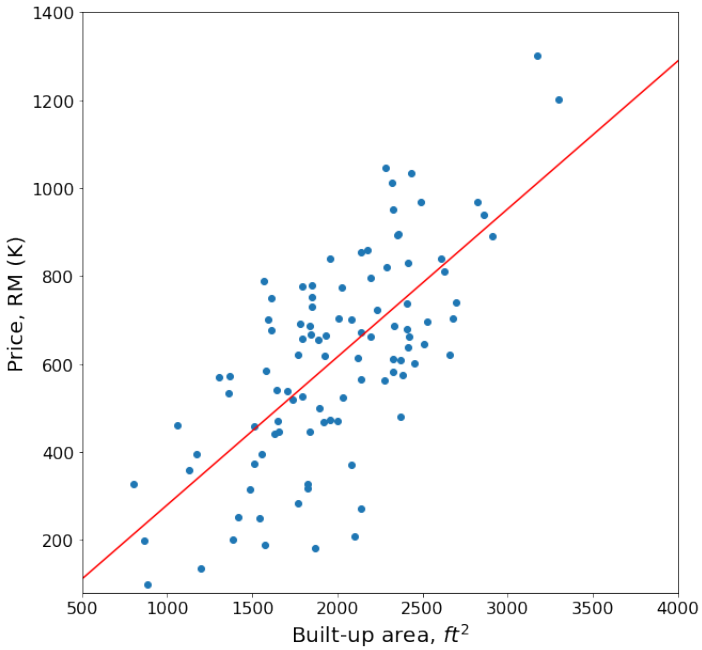


Figure 3.1 House prices against their built-up area in square feet

Linear regression is one of the most widely used algorithms for solving regression problems such as predicting house price, employee salary and sales revenue. The algorithm assumes the response has a linear relationship with the input feature (Seber 2009). Thus, the linear regression model is

expressed as follows:

$$\hat{f}(\mathbf{x}) = w_0 + \sum_{j=1}^{d} w_j x_j \tag{3.1}$$

where w_0 is the intercept and $w_j x_j$ is the inner product between the model's parameters (or weights) and the input features. Suppose the input is 1 dimensional (there is only one feature as above), the linear regression model is represented as follows:

$$\hat{f}(\mathbf{x}) = w_0 + w_1 x_1 \tag{3.2}$$

where the w_0 and w_1 are the intercept and the slope respectively. This is known as univariate (simple) linear regression. A univariate linear regression model is fitted to those data points. The fitted line is indicated by the red line.

The univariate linear regression can be generalized to multivariate linear regression where the input is d dimensional data points. Consider the problem of predicting house price, in addition to the built-up area, the inputs can be the number of rooms, the number of floors, location etc. The model is known as multivariate linear regression. A common notational trick is to introduce a constant input, x_0 whose value is always 1. Thus, the intercept term can be combined with the other terms as follows:

$$\hat{f}(\mathbf{x}) = \sum_{j=0}^{d} w_j x_j = \mathbf{w}^\top \mathbf{x} \tag{3.3}$$

where $\mathbf{w}$ is the weight vector and $\mathbf{x}$ is the input vector.

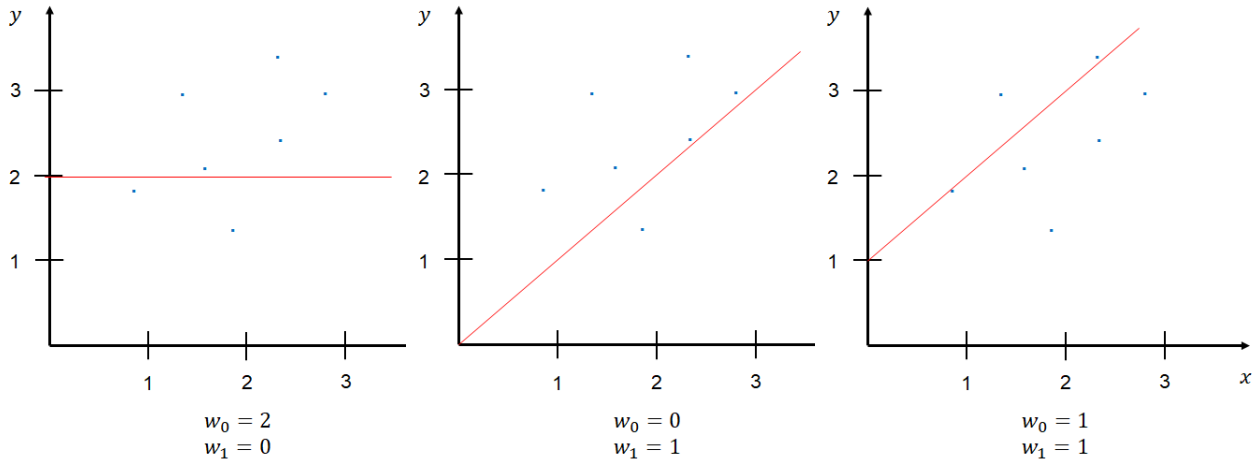
3.2 Model Parameters

The role of weights is to define the direction of the fitted line. Let's review the role of intercept and slope in univariate linear regression. Figure 3.2 illustrates three linear models with different parameters. Linear model 1, in Figure 3.2(a), the weights are $w_0 = 2$, $w_1 = 0$. Since the slope is 0 and the intercept is 2, the linear model is a horizontal line crossing 2 at the y-axis. Linear model 2, in Figure 3.2(b), the weights are $w_0 = 0$, $w_1 = 1$. The slope of the model is 1 and the intercept is 0. Hence the linear model has an upward trend, and the line is going through the origin. Finally, linear model 3, in Figure 3.2(c), the weights are $w_0 = 1$, $w_1 = 1$. The linear model has a similar trend with linear model 2. However, the line is crossing 1 at the y-axis due to its intercept being 1.

Now, how do we find the parameters that will give us an optimal linear regression model? Essentially, the idea is we want to choose weights such that the linear regression model will give us prediction $\hat{f}(\mathbf{x})$ that is close to the response, y. This is formulated by choosing a set of weights, $\mathbf{w}$ that minimizes the sum of squared loss (the difference between the response and the predicted value) over all the training instances. We ignore the fraction $\frac{1}{N}$ to simplify the explanation. Ignoring the fraction will not affect the parameter estimation since it is a constant. The squared loss is defined as follows:

$$J(\mathbf{w}) = \sum_{i=1}^{N} (y^i - \hat{f}(\mathbf{x}^i))^2 \tag{3.4}$$

where y^i is the response of instance i and $\hat{f}(\mathbf{x}^i) = \mathbf{w}^\top \mathbf{x}$ is the predicted value of instance i.



(a) (b) (c)

Figure 3.2 Parameters (intercept and slope) define the direction of the fitted lines

3.3 Least Squares

The common method to estimate the weights that will minimize the loss function is the least squares (Rossi 2018). The method is illustrated in Figure 3.3. The data points are indicated by the blue dots and the red line is the linear regression model. The predicted value for each training instance is shown as blue crosses. The blue line from a data point to the blue cross is the prediction error which is the difference between the response and the predicted value. The aim is to find a set of weights that will minimize the sum of squared errors.

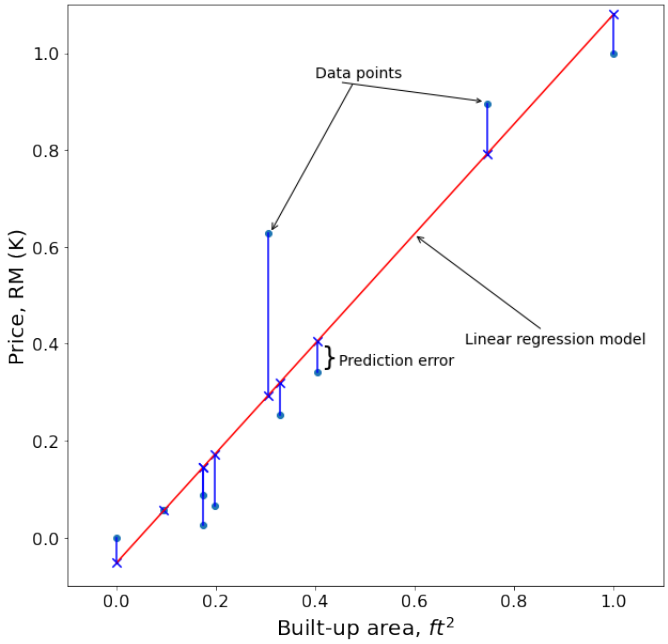


Figure 3.3 Least squares find the weights that will minimize the sum of squared errors (the length of the blue lines)

Let's express the loss function using matrix notation.

$$J(\mathbf{w}) = \sum_{i=1}^{N}(y^i - \hat{f}(\mathbf{x}^i))^2 = (\mathbf{y} - \mathbf{w}^\top \mathbf{x})^2 \tag{3.5}$$

where $\mathbf{y}$ is the output (response) vector, $\mathbf{w}$ is the weight vector and $\mathbf{x}$ is the input vector.

Differentiating J with respect to $\mathbf{w}$.

$$\frac{\partial J}{\partial \mathbf{w}} = -2\mathbf{x}(\mathbf{y} - \mathbf{w}^\top \mathbf{x}) \tag{3.6}$$

Set the derivative to zero to solve for $\mathbf{w}$.

$$-2\mathbf{x}(\mathbf{y} - \mathbf{w}^\top \mathbf{x}) = 0 \tag{3.7}$$

$$-\mathbf{x}^\top \mathbf{y} + \mathbf{w}\mathbf{x}^\top \mathbf{x} = 0 \tag{3.8}$$

$$\mathbf{w}\mathbf{x}^\top \mathbf{x} = \mathbf{x}^\top \mathbf{y} \tag{3.9}$$

The optimal weights of the linear regression model is given by

$$\mathbf{w} = (\mathbf{x}^\top \mathbf{x})^{-1}\mathbf{x}^\top \mathbf{y} \tag{3.10}$$

The least squares might fail to provide the optimal weights. This occurs when the input features are (perfectly) correlated e.g. $x_1 = 2x_2$. Then, the $\mathbf{x}^\top \mathbf{x}$ is singular (determinant is zero) and the inverse of the matrix does not exist.

Numerical Example: Given the following dataset, find $\mathbf{w}$ using least square and gradient descent.

$$\mathbf{x} = \begin{bmatrix} 1 & 3.1 \\ 1 & 2.9 \\ 1 & 3.5 \\ 1 & 4.3 \\ 1 & 3.6 \end{bmatrix}$$

$$\mathbf{y} = \begin{bmatrix} 2.0 \\ 2.6 \\ 2.2 \\ 3.2 \\ 3.0 \end{bmatrix}$$

Calculate $\mathbf{w} = (\mathbf{x}^\top \mathbf{x})^{-1}\mathbf{x}^\top \mathbf{y}$

$$(\mathbf{x}^\top \mathbf{x})^{-1} = \begin{bmatrix} 10.5685 & -2.9795 \\ -2.9795 & 0.8562 \end{bmatrix}$$

$$\mathbf{x}^\top \mathbf{y} = \begin{bmatrix} 13 \\ 46 \end{bmatrix}$$

$$\mathbf{w} = \begin{bmatrix} 0.334 \\ 0.652 \end{bmatrix}$$

3.4 Gradient Descent

The weights of the linear regression model can be estimated using an optimization algorithm known as gradient descent. Gradient descent is an optimization algorithm for finding a local minimum of a differentiable function. Note that, gradient descent is a generic algorithm that is used not only in linear regression but it is applied in various places in machine learning. The algorithm iteratively updates the parameters of the function by computing the gradient of the function. Consider a differentiable function, $f(x) = x^2 - 10x$ as shown in Figure 3.4. We know that the function is minimum when $x = 5$. Assuming that $x = 12$, to get to the minimum, we need to move to the "left". Specifically, we need to update x by subtracting a value from it. To determine the value that we need to subtract from x, we compute the gradient at $x = 12$ which is the slope of the tangent line at the given point. In other words, the gradient is the rate of change of the function at a given point.

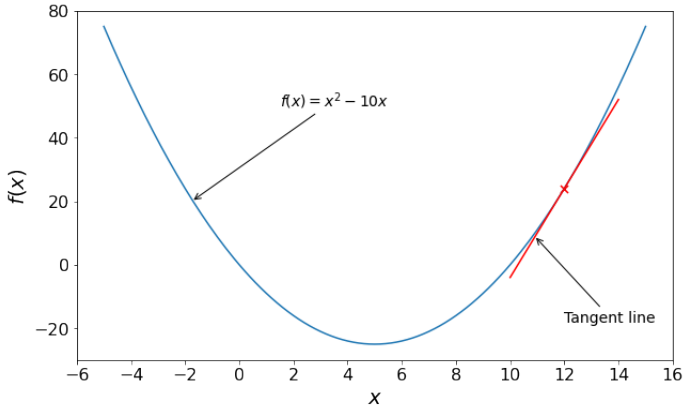


Figure 3.4 The gradient is the slope of the tangent line at a given point

Given a function, the gradient is computed by taking the derivative of the function with respect to its parameter.

$$\frac{\partial f}{\partial x} = 2x - 10 \tag{3.11}$$

Then, the parameter is updated iteratively until optimal solution is obtained as shown in Algorithm 1:

Algorithm 1 Gradient Descent Algorithm

1: **repeat**
2: $\quad x \leftarrow x - \alpha \frac{\partial f}{\partial x}$
3: **until** converge

α is known as the step size or learning rate, where $0 \leq \alpha \leq 1$, and it controls the amount of the update to be applied to the parameter x. A small α corresponds to a little step which means the optimization will take longer to complete while a large α may cause the gradient descent to overshoot (miss the minimum) and fail to converge.

Let's analyze the parameter update equation. Consider $x = 12$ as shown in Figure 3.4. For now, assuming $\alpha = 1$. We know that $\partial f/\partial x > 0$ (positive value). Based on the update equation, x will be reduced by the value of $\partial f/\partial x$. If x is to the left of the minimum as shown in Figure 3.5. We know $\partial f/\partial x < 0$ (negative value). Based on the update equation, x will be increased by the value of $\partial f/\partial x$.

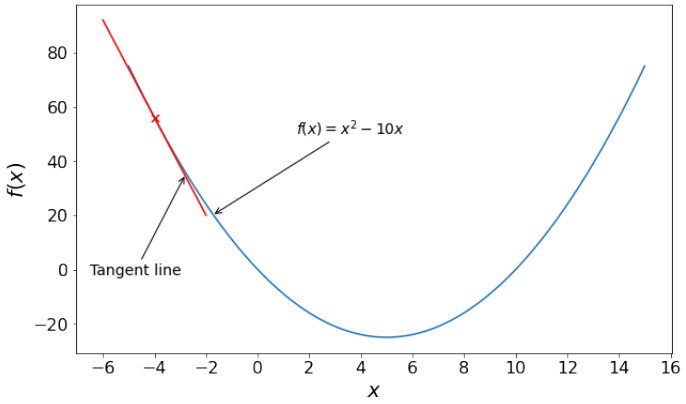


Figure 3.5 The gradient is the slope of the tangent line at a given point

In the context of linear regression, the squared loss is a differentiable function, thus gradient descent can be used to find the weights that will minimize the loss function. The weights are iteratively updated based on the derivative of the loss function. For iterative k, the weights are updated as follows:

$$\mathbf{w}_k = \mathbf{w}_{k-1} - \alpha \mathbf{g}_k \tag{3.12}$$

where α is the learning rate which controls the amount of update to be applied to the weights, $\mathbf{g}_k$ is the derivative of the loss function with respect to each weight. Given a squared loss

$$J(\mathbf{w}) = \frac{1}{2N}\sum_{i=1}^{N}(y^i - \hat{f}(\mathbf{x}^i))^2, \tag{3.13}$$

the derivative of the loss function with respect to a single weight w_j, which corresponds to the scalar component $g_{k,j}$ of the gradient vector $\mathbf{g}_k$ is

$$g_{k,j} = \frac{\partial J}{\partial w_j} = -\sum_{i=1}^{N} x_j^i (y^i - \hat{f}(\mathbf{x}^i)) \tag{3.14}$$

where N is the number of instances. $\frac{1}{2}$ is a mathematical convenience to cancel out 2 when the derivative is performed on the loss function.

Numerical Example: Given the following dataset, find $\mathbf{w}$ using least square and gradient descent.

$$\mathbf{x} = \begin{bmatrix} 1 & 3.1 \\ 1 & 2.9 \\ 1 & 3.5 \\ 1 & 4.3 \\ 1 & 3.6 \end{bmatrix}$$

$$\mathbf{y} = \begin{bmatrix} 2.0 \\ 2.6 \\ 2.2 \\ 3.2 \\ 3.0 \end{bmatrix}$$

Randomly initialize $\mathbf{w}$.

$$\mathbf{w} = \begin{bmatrix} 0.45 \\ 0.75 \end{bmatrix}$$

Calculate $\hat{\mathbf{y}}$

$$\hat{\mathbf{y}} = \begin{bmatrix} 2.775 \\ 2.626 \\ 3.075 \\ 3.675 \\ 3.150 \end{bmatrix}$$

Calculate the loss using mean squared error.

$J_{mse} = \frac{1}{2N} \sum_{i=1}^{N} (y - \hat{y})^2 = 0.1615$

Calculate the gradient with respect to $\mathbf{w}$

$$\begin{aligned} \frac{\partial J}{\partial w_0} &= -\frac{1}{N} \sum_{i=1}^{N} x_0 (y^i - \hat{y}^i)) \\ &= \frac{(0.775 + 0.025 + 0.875 + 0.475 + 0.15)}{5} \\ &= 0.46 \end{aligned}$$

$$\begin{aligned} \frac{\partial J}{\partial w_1} &= -\frac{1}{N} \sum_{i=1}^{N} x_1 (y^i - \hat{y}^i)) \\ &= \frac{(2.405 + 0.0725 + 3.063 + 2.043 + 0.540)}{5} \\ &= 1.624 \end{aligned}$$

Suppose $\alpha = 0.01$. Update $\mathbf{w}$.

$$\begin{aligned} w_0 &:= w_0 - \alpha \frac{\partial J}{\partial w_0} \\ &:= 0.45 - 0.01(0.46) \\ &:= 0.445 \end{aligned}$$

$$\begin{aligned} w_1 &:= w_1 - \alpha \frac{\partial J}{\partial w_1} \\ &:= 0.75 - 0.01(1.624) \\ &:= 0.734 \end{aligned}$$

Using the updated $\mathbf{w}$, calculate (new) prediction and loss.

$J_{mse} = \frac{1}{2N} \sum_{i=1}^{N} (y - \hat{y})^2 = 0.135$

Notice that the loss is decreased after the weight update, from 0.1615 to 0.1350. The steps are repeated until the error is converged.

3.5 Exchangeability

Machine learning algorithms cannot understand or do not model the cause and effect relationships which is necessary to explain why something happens. In real-world scenarios, there are causal relationships between actions and consequences such as when a patient has a fever and headache

because of dengue, the dengue is the cause, and the effects are fever and headache. So, how do machine learning models able to predict the output given an input? A machine learning model gets its predictive power by eliminating the differences in data, ignoring the outliers, reducing the errors and finding the mapping between the input features and the output. Since machine learning algorithms are not able to attribute predictions to causes, they rely on a concept called exchangeability to ensure we can build useful predictive models.

What is exchangeability? When we train a predictive model for a particular problem, we use a dataset that contains samples of the whole population. We assume the samples in the dataset are sufficiently representative. Specifically, the size of the dataset should not only be large enough, but the data should also be representative of the actual environment. Therefore, the predictive models that we build using the dataset can be applied to predict unknown new data. This assumption is made on the basis that the unknown future data is exchangeable with the training data. This concept of data exchangeability assumes that the training data and test data are independent and identically distributed known as IID.

The assumption that data are independent and identically distributed is essential in machine learning. Independent means each instance is unrelated to every other instance in the dataset. Identically distributed means the instances are sampled from the same probability distribution. Consider a dataset on hospital patients. The dataset is independent and identically distributed if each patient's data is independent or is unrelated to other patients' data and they have the same probability distribution, or in other words, the law that generates the data is the same from one patient to another.

3.6 Bias-Variance Decomposition

The main goal of machine learning is to find a function that is a good approximation $\hat{f}(\mathbf{x}; D_{tr})$ of the true function based on the training set D_{tr}. In a regression setting, a good function is learned by quantifying the difference between the response variable y and the prediction $\hat{y} = \hat{f}(\mathbf{x}; D_{tr})$ using the squared loss. Specifically, we want the squared loss to be minimal for both $D_{tr} = (x^1, y^1), (x^2, y^2), ..., (x^N, y^N)$ and data points not in the training set. Ideally, we want the learned function to be perfect (minimal prediction error), however, such a function is not possible due to the limitation of the machine learning algorithm and noise in the data.

The prediction error can be decomposed into irreducible error and reducible error (Kohavi and D. Wolpert 1996). The irreducible error is due to the fact that the data that we have are not able to completely describe the target being predicted. There are other features not within the dataset that have some effect on the target. Consider the problem of predicting the sales of a food truck given its location and number of tourists. The features might be able to give a prediction, but they will not explain everything because there are other factors that affect the sales such as the weather, the raw material shortage and other issues intrinsic to the problems that are being modeled.

The reducible error is the error that is associated with the predictive model which can be decomposed into bias and variance errors. Bias error refers to the amount of difference between the target and the prediction of the model. Every machine learning algorithm comes with a set of assumptions that form the inductive bias of the predictive model. For example, the linear regression algorithm assumes a linear relationship between the features and the target, hence the model is described in the form of a linear equation. However, if the assumption does not hold, it may lead to a high inductive bias error. We call this underfitting model. Figure 3.6 illustrates a linear model indicated by the red line which does not approximate well the true function indicated by the blue curve.

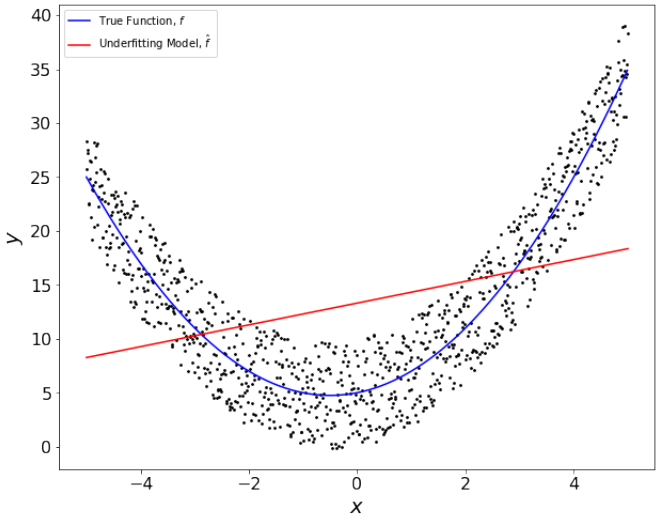


Figure 3.6 A predictive model with inductive bias due to its simplistic assumption about the data

On the other hand, some powerful machine learning algorithms such as decision tree and support vector machine make strong assumptions about the data. The algorithms are discussed in the later chapters. Due to their strong learning ability, the algorithms can learn every variation including the randomness in the data. Thus, the predictive models have low inductive bias error, but high variance error whereby the models are sensitive to small fluctuations in the data. Variance error also reflects the variability in the model prediction. It shows the inconsistency of the model across different datasets. This phenomenon is known as overfitting whereby the models do not generalize well on new data. Figure 3.7 illustrates an overfitting model in which the model fits the data very closely as indicated by the green line.

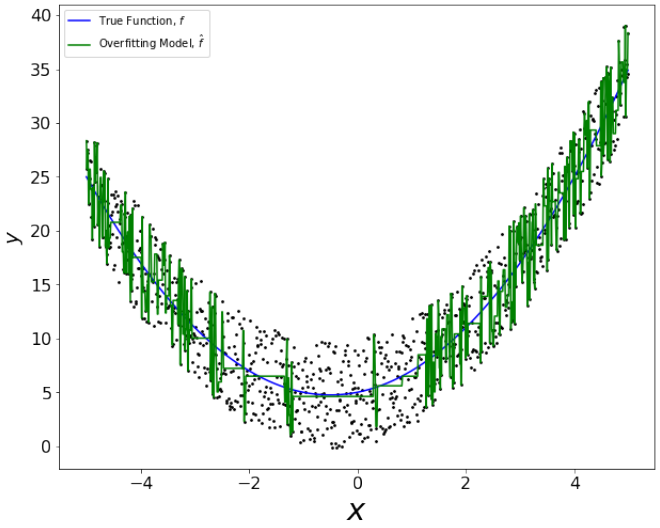


Figure 3.7 An overfitting predictive model learns the variation of the data very closely

Suppose the data is generated by a function (model) $y = f(\mathbf{x}) + \epsilon$, with $E(\epsilon) = 0$ and $\texttt{var}(\epsilon) = \sigma^2$. The derivation of the bias-variance decomposition of the squared loss is as follows:

$$\begin{aligned} E[(y - \hat{f})^2] &= E[(f + \epsilon - \hat{f})^2] \\ &= E[(f - E(\hat{f}) + E(\hat{f}) - \hat{f} + \epsilon)^2] \end{aligned} \tag{3.15}$$

Using $(a+b+c)^2 = a^2+b^2+c^2+2ab+2ac+2bc$ and assuming $a = f - E(\hat{f})$, $b = E(\hat{f}) - \hat{f}$ and $c = \epsilon$, we can expand the equation as follows:

$$\begin{aligned} E[(y-\hat{f})^2] =& E[(f-E[\hat{f}])^2] + E[(E[\hat{f}]-\hat{f})^2] + E[\epsilon^2] \\ &+ 2E[\epsilon(E[\hat{f}]-\hat{f})] + 2E[\epsilon(f-E[\hat{f}])] \\ &+ 2E[(f-E[\hat{f}])(E[\hat{f}]-\hat{f})] \end{aligned} \tag{3.16}$$

$$\begin{aligned} E[(y-\hat{f})^2] =& (f-E[\hat{f}])^2 + E[(E[\hat{f}]-\hat{f})^2] + E[\epsilon^2] \\ &+ 2E[\epsilon](E[\hat{f}]-\hat{f}) + 2E[\epsilon]E[(f-E[\hat{f}])] \\ &+ 2E[E[\hat{f}]-\hat{f}](f-E[\hat{f}]) \end{aligned} \tag{3.17}$$

We know that $E[E[\hat{f}]-\hat{f}] = E[\hat{f}] - E[\hat{f}] = 0$. Thus, the equation is reduced to

$$\begin{aligned} E[(y-\hat{f})^2] &= (f-E[\hat{f}])^2 + E[(E[\hat{f}]-\hat{f})^2] + E[\epsilon^2] \\ &= (f-E[\hat{f}])^2 + E[(E[\hat{f}]-\hat{f})^2] + \texttt{var}(\epsilon) \\ &= \texttt{bias}(\hat{f})^2 + \texttt{var}(\hat{f}) + \sigma^2 \end{aligned} \tag{3.18}$$

where `bias` and `var` are the bias and variance errors and σ^2 is the irreducible error.

Ideally, we want to build a model that has both small bias and variance errors, a model that learns the patterns of the data accurately and generalizes well to unseen data. However, in practice, we must make a trade-off between increasing the model complexity to capture the variation in the data and simplifying the model to avoid overfitting as shown in Figure 3.8. Simple machine learning algorithms such as linear regression make fewer assumptions about the data. Thus, the algorithm often produces predictive models with high bias error but they are consistent across datasets and are not sensitive to changes in the data. More complex machine learning algorithms such as decision tree and support vector machine have more parameters that can be manipulated to learn the underlying structure of the data. Thus, the algorithms may produce models with low bias error. But, the models tend to have higher variance and there is a risk of overfitting the data. Hence, we need to find the optimal balance between bias and variance.

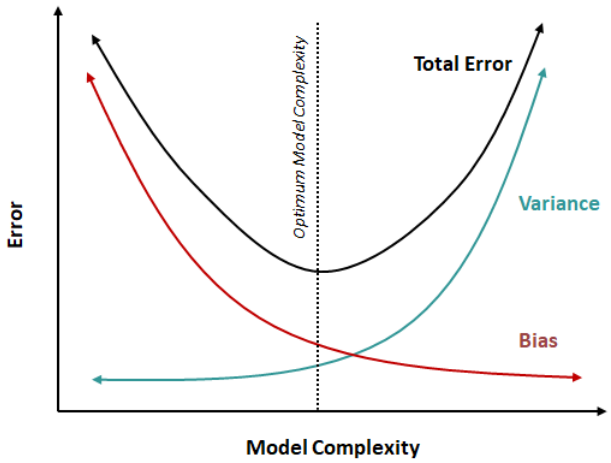


Figure 3.8 A trade-off between bias and variance, as model complexity increases, bias error is decreasing but variance error is increasing

3.7 Implementing Linear Regression

In this section, we will be implementing Linear Regression using Scikit-Learn. We will be using California Housing dataset that contains information about the housing values in California district. The data is based on the 1990 California census. This dataset can be fetched from Internet using Scikit-Learn. This dataset is obtained from the StatLib repository. The features are information such as the average number of rooms, house age and house location. The target variable is the median house value for California districts, expressed in hundreds of thousands of dollars ($100,000).

Let us import the function to download the dataset.

```
from sklearn.datasets import fetch_california_housing
```

Now, we can use the function to download the dataset. The function returns a dictionary-like object. It contains the data, the feature names and the dataset description. The parameter `as_frame` is set to `True` to acquire the data in the form of Pandas DataFrame.

```
california_housing = fetch_california_housing(as_frame=True)
```

Once it is loaded, we can get the DataFrame as follows:

```
data_df = california_housing.frame
```

Let us display the first five rows.

```
print(data_df.head())
```

```
   MedInc  HouseAge  AveRooms  AveBedrms  Population  AveOccup  Latitude  \
0  8.3252      41.0  6.984127   1.023810       322.0  2.555556     37.88
1  8.3014      21.0  6.238137   0.971880      2401.0  2.109842     37.86
2  7.2574      52.0  8.288136   1.073446       496.0  2.802260     37.85
3  5.6431      52.0  5.817352   1.073059       558.0  2.547945     37.85
4  3.8462      52.0  6.281853   1.081081       565.0  2.181467     37.85

   Longitude  MedHouseVal
0    -122.23        4.526
1    -122.22        3.585
2    -122.24        3.521
3    -122.25        3.413
4    -122.25        3.422
```

We can see that the dataset contains 9 columns whereby the first 8 columns are the features and the last column is the target. All features are numerical features encoded as floating number and there are no missing values. Now, let us inspect the data types of each column.

```
data_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 9 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   MedInc       20640 non-null  float64
 1   HouseAge     20640 non-null  float64
 2   AveRooms     20640 non-null  float64
 3   AveBedrms    20640 non-null  float64
 4   Population   20640 non-null  float64
 5   AveOccup     20640 non-null  float64
 6   Latitude     20640 non-null  float64
 7   Longitude    20640 non-null  float64
 8   MedHouseVal  20640 non-null  float64
dtypes: float64(9)
memory usage: 1.4 MB
```

As we can see, the data type is correctly inferred as float. Now, let us prepare the dataset for linear regression.

```
y = data_df['MedHouseVal']
X = data_df.drop(columns=['MedHouseVal'])
```

We split the dataset into training and testing sets with the ratio of 7:3 using `train_test_split`. Then, we display the size of each set.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=20)
print(X_train.shape)
print(X_test.shape)
```

```
(14448, 8)
(6192, 8)
```

Then, we normalize both the training and test sets using `MinMaxScaler`.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train_norm = scaler.fit_transform(X_train)
X_test_norm = scaler.transform(X_test)
```

3.7.1 Least Squares Linear Regression

Now the training set is prepared, we can build the least squares linear regression model. We are going to import `LinearRegression` from Scikit-Learn, and then create an object of `LinearRegression`.

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
```

Once we have created the object, we can call the fit method to learn the train data.

```
model.fit(X_train_norm, y_train)
```

We can print out the intercept and the estimated coefficients of the model.

```
print('Estimated intercept coefficients: ', model.intercept_)
print('Number of coefficients: ', len(model.coef_))
print('Coefficients: ', model.coef_)
```

```
Estimated intercept coefficients:  3.717387600133579
Number of coefficients:  8
Coefficients:  [  6.28646321   0.48047174 -14.81633581  21.05575561
  -0.2146071  -4.05042218  -3.89136493  -4.20295128]
```

Now, let us use the model to predict the test set.

```
y_pred = model.predict(X_test_norm)
```

Scikit-Learn provides functions to compute regression loss such as mean squared error, mean absolute error and R^2. We import the functions and use them to evaluate the performance of the model. Please note that the results in the book could be different from yours due to different parameter settings.

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
print('Mean Absolute Error: ', mean_absolute_error(y_test, y_pred))
print('Mean Squared Error: ', mean_squared_error(y_test, y_pred))
print('R-squared: ', r2_score(y_test, y_pred))
```

```
Mean Absolute Error:  0.5349006568619926
Mean Squared Error:  0.5245761338156759
R-squared:  0.6149202506648622
```

The metrics show that the least squares linear regression model has a mean absolute and mean squared errors of 0.5349 and 0.5246 respectively. Based on the R^2 value, the model is able to explain about 61.49% (0.6149) of the data.

3.7.2 Gradient Descent Regression

Scikit-Learn provides a module called `SGDRegressor` for implementing gradient descent-based linear regression. The module has several parameters that need to be set to implement linear regression as follows:

- `loss` is the loss function to be used
- `eta0` is the value of the learning rate
- `learning_rate` is the learning rate schedule, either constant or changing
- `tol` is the stopping criteria of the training
- `max_iter` is the training maximum number of epochs

Let us import the module and create the object by specifying the parameters.

```
from sklearn.linear_model import SGDRegressor
model_gd = SGDRegressor(loss='squared_error', max_iter=300, eta0=0.02,
                  learning_rate='constant', tol=None, verbose=1, random_state=1)
```

As shown above, we use the squared error loss, set the maximum iteration and initial learning rate value (`eta0`) to 300 and 0.02 respectively. Parameter `learning_rate` allows the learning rate value to be reduced for each iteration step. We set the parameter to `'constant'` to keep the value of learning rate constant. The parameter can be set to `optimal`, `invscaling` or `adaptive`. These parameter values change the learning rate for every epoch. Both `optimal` and `invscaling` reduce the learning rate based on a predefined equation. The equations are documented in Scikit-Learn's full user guide. `adaptive` reduces the learning rate if the training loss is not decreased for a certain number of epochs. Parameter `tol` is used to set the tolerance value for the early stopping algorithm which will be explained in the later chapter. For now, we set the parameter to `None`. Parameter `verbose` is set to 1 to display the output of the model training.

We call `fit` by passing the training set. The training progress is shown for each iteration. The most important value that should be observed is the average loss (`Avg. loss`). The value indicates the average training loss of the model for each iteration.

```
model_gd.fit(X_train_norm, y_train)
```

```
-- Epoch 1
Norm: 6.74, NNZs: 8, Bias: 2.070237, T: 14448, Avg. loss: 0.336953
Total training time: 0.00 seconds.
-- Epoch 2
Norm: 7.17, NNZs: 8, Bias: 2.983634, T: 28896, Avg. loss: 0.282832
Total training time: 0.01 seconds.
-- Epoch 3
Norm: 7.54, NNZs: 8, Bias: 3.443707, T: 43344, Avg. loss: 0.277963
Total training time: 0.01 seconds.

...

-- Epoch 298
Norm: 9.38, NNZs: 8, Bias: 3.912330, T: 4305504, Avg. loss: 0.272324
Total training time: 0.27 seconds.
-- Epoch 299
Norm: 9.31, NNZs: 8, Bias: 4.118629, T: 4319952, Avg. loss: 0.272640
Total training time: 0.27 seconds.
-- Epoch 300
Norm: 9.31, NNZs: 8, Bias: 3.988856, T: 4334400, Avg. loss: 0.272158
Total training time: 0.27 seconds.
```

We can print out the parameters of the model as follows:

```
print('Estimated intercept coefficients: ', model_gd.intercept_)
print('Number of coefficients: ', len(model_gd.coef_))
print('Coefficients: ', model_gd.coef_)
```

Now, let us use the model to predict the test set.

```
y_pred_gd = model_gd.predict(X_test_norm)
```

Using the same functions above, we compute regression loss such as mean squared error, mean absolute error and R^2.

```
print(mean_absolute_error(y_test, y_pred_gd))
print(mean_squared_error(y_test, y_pred_gd))
print(r2_score(y_test, y_pred_gd))
```

```
0.5461181694006848
0.5344161657879705
0.607696900609306
```

The metrics show that the gradient descent-based linear regression is slightly worse than the least squares linear regression. The gradient descent-based linear regression has slightly higher mean absolute and mean squared errors of 0.5461 and 0.5344 respectively. The R^2 value is also slightly lower whereby the model is able to explain about 60.77% (0.6077) of the data.

Exercises

1. Describe dependent and independent variables.
2. Describe the assumption of linear regression.
3. Describe the difference between simple linear and multiple linear regressions.
4. Find the least square regression line for the following dataset:

$$\mathbf{x} = \begin{bmatrix} -1 \\ 0 \\ 1 \\ 2 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} 0 \\ 2 \\ 4 \\ 5 \end{bmatrix}$$

5. Describe the difference between the gradient descent and least squares methods.
6. Explain the effect of learning rate in gradient descent algorithm.

Chapter 4

Linear Models for Classification

4.1 Introduction

This chapter focuses on linear models for classification problems. Classification is a task in machine learning where the aim is to predict the class labels of a given set of inputs. We will explore the concepts and methods behind linear models such as perceptron and logistic regression. We will discuss the assumptions and limitations of these models.

4.2 Perceptron

Perceptron is a parametric algorithm for solving binary classification problems whereby the label, $y \in \{-1, +1\}$ (Rosenblatt 1957). The assumption of perceptron is that data points are linearly separable. Specifically, we can define a hyperplane that can divide the feature space into regions and each region is labeled according to the classification. Although the assumption sounds simplistic, if data points belonging to a particular class have similar characteristics and are clustered together, the hyperplane can be defined between the clusters of the two classes to classify the data points.

Essentially, given an input vector, $\mathbf{x} = (x_1, x_2, \ldots, x_d)$, the perceptron is a linear model that predicts the labels via

$$\hat{f}(\mathbf{x}) = \text{sign}(w_0 + \sum_{j=1}^{d} w_j x_j) \tag{4.1}$$

where w_0 is the intercept also known as bias and $w_j x_j$ is the inner product between the model's coefficient (or weight) vector and the input vector also known as the linear combination of the inputs. We can introduce a constant input, x_0 whose value is always 1. Then, the expression can be simplified as follows:

$$\hat{f}(x) = \text{sign}(\sum_{j=0}^{d} w_j x_j) = \mathbf{w}^\top \mathbf{x} \tag{4.2}$$

where $\mathbf{w}^\top \mathbf{x}$ is the inner product in vector form.

sign(z) is the sign function which returns $+1$ if z is a positive value and returns -1 if z is a negative

value. Thus, given a test input, $\acute{x}$, the classification of the input is given as follows:

$$\hat{f}(\acute{x}) = \begin{cases} +1 & \mathbf{w}^\top \mathbf{x} > 0 \\ -1 & \mathbf{w}^\top \mathbf{x} < 0 \end{cases} \tag{4.3}$$

Consider a binary classification problem in Figure 4.1. The labels of the data points are yellow $(+1)$ and purple (-1). A perceptron model is fitted to those data points. The decision boundary is defined by $\mathbf{w}^\top \mathbf{x}$ as indicated by the red line. The input space classified as $+1$ is denoted by the orange shaded region while the purple shaded region denotes the input space classified as -1.

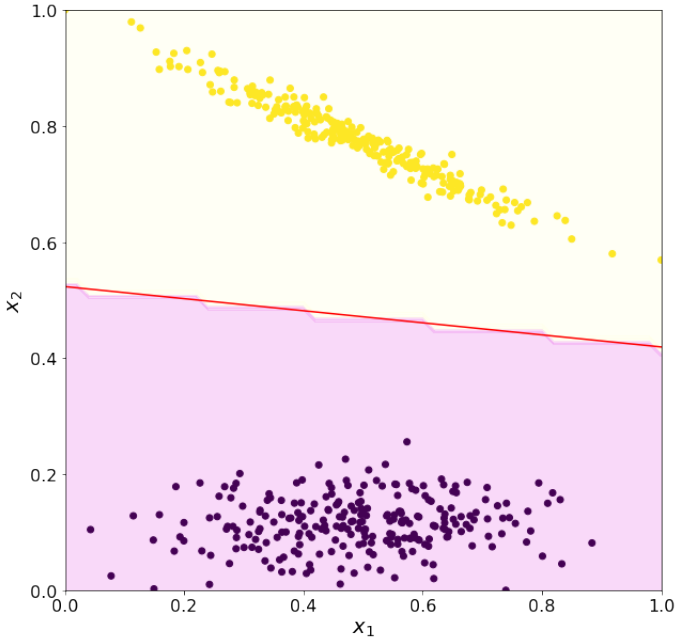


Figure 4.1 The decision boundary and regions of the perceptron are indicated by the red line and the shaded regions respectively

Numerical Example: Consider the perceptron model $\hat{f}(\mathbf{x}) = -4 + x_1 + x_2$. Predict the labels of the following inputs:

$$\mathbf{x} = \begin{bmatrix} 1 & 1 \\ 1.5 & 1 \\ 3 & 3 \\ 3 & 4 \end{bmatrix}$$

$\hat{f}(\mathbf{x}^1) = -4 + 1 + 1 = -2.\ \hat{y}^1 = -1$

$\hat{f}(\mathbf{x}^2) = -4 + 1.5 + 1 = -1.5.\ \hat{y}^2 = -1$

$\hat{f}(\mathbf{x}^3) = -4 + 3 + 3 = 2.\ \hat{y}^3 = +1$

$\hat{f}(\mathbf{x}^4) = -4 + 3.5 + 3 = 3.\ \hat{y}^4 = +1$

As shown above, the weights, $\mathbf{w}$ define the hyperplane that separates the data points. The algorithm to find the optimal weights is given in Algorithm 2. First, the weights are initialized randomly. The algorithm keeps training until all instances are correctly classified. Thus, if the data is not linearly separable, the training will loop forever. For each iteration, error, e is initialized to zero, and the whole instances of the training set are classified and compared with their corresponding labels one

by one. If the classification is incorrect, increment e, and the weights are updated with values of element-wise multiplication $\odot$ between the instance and the label. Otherwise, the weights are not updated.

Algorithm 2 Perceptron Algorithm

```
initialize weights, w
while e ≠ 0 do
    e ← 0
    for (x,y) in training set do
        f̂(x) ← w^⊤ x
        if f̂(x) ≠ y then
            w ← w + y ⊙ x
            e ← e + 1
        end if
    end for
end while
```

4.3 Logistic Regression

Logistic regression is a parametric algorithm for solving classification problems whereby the label, $y \in \{0, 1\}$ (Hosmer Jr, Lemeshow, and Sturdivant 2013). Logistic regression and perceptron are similar whereby a hyperplane is defined to divide the feature space into regions labeled according to the classification. The hyperplane is defined by the inner product between parameters and the input vectors (summation of the weighted inputs), $\mathbf{w}^\top \mathbf{x}$. Then, the resultant of the inner product is passed to a function that ensures the predicted value to be $0 \leq \hat{f}(\mathbf{x}) \leq 1$ via

$$\hat{f}(\mathbf{x}) = \text{sigm}(\mathbf{w}^\top \mathbf{x}) \tag{4.4}$$

where $\text{sigm}(\cdot)$ denotes the sigmoid function, also known as logistic function which is defined as:

$$\text{sigm}(z) = \frac{1}{(1 + e^{-z})} \tag{4.5}$$

Figure 4.2 shows the graph of the sigmoid function. The sigmoid function has an S-shaped curve, which maps its input to a value within the range of 0 and 1. Thus, logistic regression has a probabilistic connotation. The curve has two horizontal asymptotes, one end is approaching 1 and the other end is approaching 0. Due to this property, the sigmoid function is differentiable at every point.

As mentioned above, a logistic regression model outputs values within the range of 0 and 1. If we threshold the output at 0.5, a classification rule can be defined as follows:

$$\hat{f}(\mathbf{x}) = \begin{cases} 1 & \text{sigm}(z) > 0.5 \\ 0 & \text{sigm}(z) \leq 0.5 \end{cases} \tag{4.6}$$

Figure 4.3 shows the output probabilities of 50 simulated inputs to the sigmoid function. The input values vary from -17 to $+22$ and their predicted values are indicated by the red circles. Using a threshold value of 0.5, the classification of the data is indicated by the green and blue markers. As can be seen in the figure, output probabilities above the threshold value are classified as 1. Otherwise, they are classified as 0.

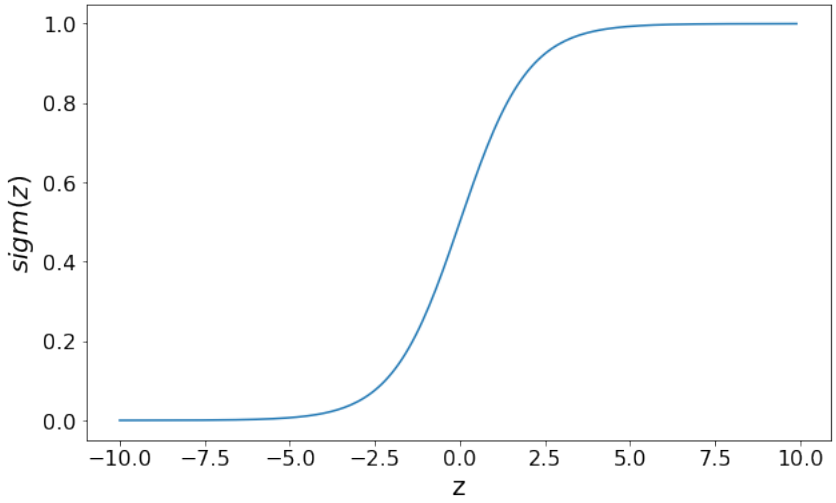


Figure 4.2 The sigmoid function is over −10 to +10

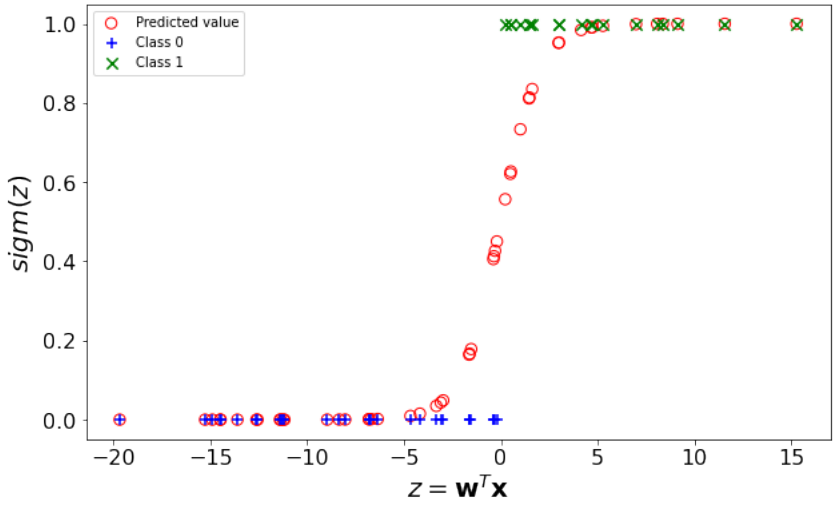


Figure 4.3 The output probabilities of the sigmoid function and their classification using a threshold value of 0.5

Numerical Example: Consider the logistic regression model $\hat{f}(\mathbf{x}) = g(z)$ where $z = -4 + x_1 + x_2$ and $g(z)$ is the sigmoid function. Predict the labels of the following inputs:

$$\mathbf{x} = \begin{bmatrix} 1 & 1 \\ 1.5 & 1 \\ 3 & 3 \\ 3 & 4 \end{bmatrix}$$

$z^1 = -4 + 1 + 1 = -2$

$g(z^1) = \frac{1}{1+e^{-(-2)}} = 0.119$

$\hat{y}^1 = 0$

$z^2 = -4 + 1.5 + 1 = -1.5$

$g(z^2) = \frac{1}{1+e^{-(-1.5)}} = 0.182$

$\hat{y}^2 = 0$

$z^3 = -4 + 3 + 3 = 2$

$g(z^3) = \frac{1}{1+e^{-2}} = 0.881$

$\hat{y}^3 = 1$

$z^4 = -4 + 3.5 + 3 = 3$

$g(z^4) = \frac{1}{1+e^{-3}} = 0.953$

$\hat{y}^4 = 1$

4.3.1 Multiclass Classification Setting

So far, we have seen logistic regression for binary classification. Real-world problems may involve more than two classes. For multiclass classification, we need to train multiple logistic regression models, one for each of the C classes. For example, consider a classification problem as illustrated in Figure 4.4. The data points belong to three classes, red, blue and green. We build three logistic regression models for each of the three classes. First, we treat red instances as class 1, and all the other instances as class 0. Then, we build the logistic regression model to distinguish red instances from the rest. Next, we treat blue instances as class 1 and all the other instances as class 0, and build the logistic regression model to distinguish blue instances from the rest. We perform the same method for green instances. Eventually, we have three logistic regression models as shown in Figure 4.5.

Given a test point, $\acute{\mathbf{x}}$, each model $\hat{f}_c(\acute{\mathbf{x}})$ performs the prediction on the test point and produces C predicted values.

$$\hat{f}_c(\acute{\mathbf{x}}) = \mathbf{w}_c^\top \acute{\mathbf{x}} \tag{4.7}$$

where $\mathbf{w}$ is the weight vector, $\acute{\mathbf{x}}$ is the input vector. Then, the predicted values are normalized using the softmax function. The softmax function turns the predicted values into probabilities. Each predicted value will be in the range of 0 and 1 and the sum of the values equal to 1.

$$\hat{f}_c(\acute{\mathbf{x}}) = \frac{e^{\hat{f}_c(\acute{\mathbf{x}})}}{\sum_{k=1}^{C} e^{\hat{f}_k(\acute{\mathbf{x}})}} \tag{4.8}$$

The test point is assigned to the class with the largest probability.

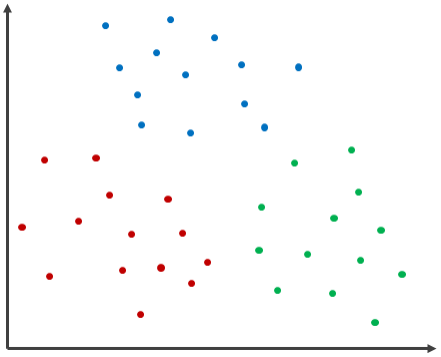
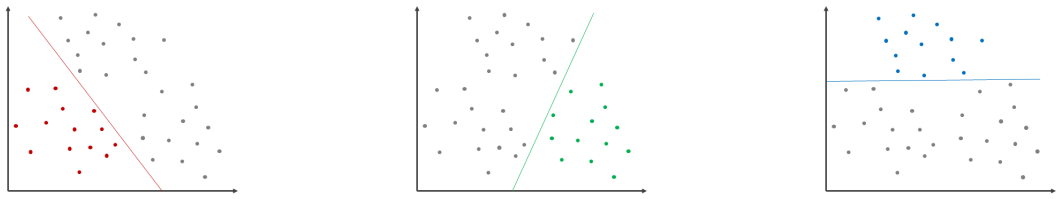


Figure 4.4 Data points belong to three classes: red, blue and green



(a) (b) (c)

Figure 4.5 Logistic regression in a multiclass classification setting (a) the model distinguishes red data points from the rest, (b) the model distinguishes green data points from the rest, and (c) the model distinguishes blue data points from the rest

4.3.2 Parameter Estimation

Now, how do we estimate the parameters of the logistic regression? We will discuss the estimation of the parameters for two-class logistic regression to simplify the explanation. The loss function of logistic regression is the negative log-likelihood or cross entropy as follows:

$$J(\mathbf{w}) = -\sum_{i=1}^{N} y^i \log \hat{y}^i + (1 - y^i) \log (1 - \hat{y}^i) \tag{4.9}$$

where $\hat{y}^i = \hat{f}(\mathbf{x}^i) = \sigma(z^i)$, σ denotes the sigmoid function and $z^i = \mathbf{w}^\top \mathbf{x}^i$

We take the derivative of the loss function with respect to the parameters and set it to zero. Using chain rule, the derivative of the loss function is

$$\frac{\partial J}{\partial w_j} = \sum_{i=1}^{N} \frac{\partial J}{\partial \hat{y}^i} \cdot \frac{\partial \hat{y}^i}{\partial z^i} \cdot \frac{\partial z^i}{\partial w_j} \tag{4.10}$$

$$\frac{\partial J}{\partial w_j} = \sum_{i=1}^{N} \frac{\hat{y}^i - y^i}{\hat{y}^i(1 - \hat{y}^i)} \hat{y}^i(1 - \hat{y}^i) x_j^i = \sum_{i=1}^{N} (\hat{y}^i - y^i) x_j^i \tag{4.11}$$

$$\sum_{i=1}^{N} (\hat{y}^i - y^i) x_j^i = 0 \tag{4.12}$$

However, the maximum likelihood estimation of the parameters is not in closed form. Thus, the equation needs to be solved using an optimization algorithm such as gradient descent, Newton's method etc. Using gradient descent, the parameters are estimated as follows:

$$\mathbf{w} := \mathbf{w} - \alpha \mathbf{g} \tag{4.13}$$

where α is the learning rate and $\mathbf{g} = \frac{\partial J}{\partial \mathbf{w}}$ is the gradient or the derivative of the loss function with respect to the parameters.

Using Newton's method, the parameters are estimated as follows:

$$\mathbf{w} := \mathbf{w} - \alpha \mathbf{H}^{-1} \mathbf{g} \tag{4.14}$$

where $\mathbf{H}^{-1}$ is the Hessian matrix or the second-order partial derivatives of the loss function with respect to the parameters.

4.4 Probability Theory

We have seen how probability plays a key role in machine learning. In this chapter, we are going to discuss further how probability theory can be used to make predictions. Data is generated from a process that is not completely known. Since we do not have complete knowledge about the process, probability theory is used to model the uncertainty within the process. In this chapter, we will discuss how probability theory can be used to represent uncertainty and make predictions.

A random process by which we observe something uncertain is called a random experiment. A random experiment is any situation where the process can lead to more than one possible outcome. Consider a random process of tossing a coin which may result in two mutually exclusive outcomes, either head or tail turns up. The outcomes of this process cannot be predicted with certainty. We can only say how likely each outcome is. The chance of a particular event happening is called probability. Another important concept that is associated with probability is the random variable. A random variable is a numerical description of the outcome of a random experiment.

Consider the process of tossing a coin with two mutually exclusive outcomes, which are either head or tail. We define a random variable X that takes one of the two values. Let $X = 1$ be the value for heads and $X = 0$ denotes the outcome of tails. The probability of an event $X = x$ is denoted by $p(X = x)$ or $P(x)$ and its value satisfies $0 \leq p(x) \leq 1$ and $\sum_{x \in X} p(x) = 1$. A higher probability value indicates a higher chance of an event occurring. Thus, using probability value, we can predict the outcome of the next toss. For example, the prediction will be head if $p(X = 1) > 0.5$ and tails otherwise.

If $p(X)$ is not known, we can estimate the probability by repeatedly tossing the coin and observing the outcomes. Given a sample of outcomes of past N tosses, the estimated probability of heads is

$$p(X = 1) = \frac{N_h}{N} \tag{4.15}$$

where N_h is the number of tosses with outcome heads. For example, if the number of tosses is 10 and the sample $\mathcal{X} = \{1, 1, 0, 1, 0, 0, 1, 1, 0, 0\}$, the estimated probability is $p(X = 1) = \frac{5}{10}$

Given two independent events, A and B, such as getting heads from tossing two coins. Note that independent events mean each event is not affected by other events. We can calculate the probability of the two events happening as follows:

$$p(A \cap B) = p(A, B) = p(A) \times p(B) \tag{4.16}$$

Some events are dependent on other events. Consider taking two balls from a jar containing 5 red and 5 blue balls. Initially, both red and blue balls have an equal chance of being taken. If the first ball was red, the second ball is less likely to be red as there are 4 red balls left in the jar. The probability that the second ball is chosen is called conditional probability. Let A and B be the event of selecting the second and first balls respectively. The conditional probability of event A given that event B is given as follows:

$$p(A \mid B) = \frac{p(A, B)}{p(B)} \tag{4.17}$$

The conditional probability can be rewritten as follows:

$$p(A \mid B) = \frac{p(B \mid A)p(A)}{p(B)} \tag{4.18}$$

where $p(B \mid A)$ is the conditional probability of event B given A, also known as the likelihood and $p(A \mid B)$ is the conditional probability of event A and B, also known as the posterior probability. $p(A)$ and $p(B)$ are the probabilities of observing A and B respectively which can be interpreted as the prior probability and evidence. Using the law of total probability, $p(B) = \sum_j p(B \mid A_j)p(A_j)$. The above definition is known as Bayes' rule (Davies 1988).

The following example illustrates the use of Bayes' rule in medical testing. Suppose that for every 100 women, 25 are indeed pregnant.

$p(y = 1) = 0.10$

The sensitivity of the pregnancy test is 98% which means if a woman is pregnant, the test will be positive with a probability of 0.98.

$p(x = 1 \mid y = 1) = 0.98$

Additionally, the false positive rate of the test kit is 1.5%.

$p(x = 1 \mid y = 0) = 0.015$

Suppose a woman takes a pregnancy test and the outcome of the test is positive. What is the probability the woman is pregnant?

$$\begin{aligned} p(y = 1 \mid x = 1) &= \frac{p(x = 1 \mid y = 1)p(y = 1)}{p(x = 1 \mid y = 1)p(y = 1) + p(x = 1 \mid y = 0)p(y = 0)} \\ &= \frac{0.98 \times 0.20}{0.98 \times 0.20 + 0.015 \times 0.80} \\ &= 0.942 \end{aligned}$$

Since the probability is 0.942, we can conclude that the woman is pregnant.

4.5 Naive Bayes

We have seen how we can perform classification when uncertainty is modeled using probabilities i.e. the likelihood, prior and evidence. But, in many cases, this information is not available to us. In the absence of the information, we can estimate the probabilities from past observations (training data). Then, using the estimated probabilities, we can classify the feature vector $\mathbf{x}$ by applying them to the Bayes' rule (Bayes 1763) as follows:

$$p(y = c \mid \mathbf{x}) = \frac{p(\mathbf{x} \mid y = c)p(y = c)}{p(\mathbf{x})} \tag{4.19}$$

The key to using this approach is to estimate the class-conditional probabilities of the features or the likelihood. Let d be the number of features, the likelihood is

$$\begin{aligned} p(x_1, x_2, ..., x_d \mid y = c) = p(x_1 \mid x_2, ..., x_d, y = c)p(x_2 \mid ..., x_d, y = c) \\ ...p(x_d \mid y = c) \end{aligned} \tag{4.20}$$

However, calculating the likelihood is cumbersome especially when the number of features is large. Naive Bayes is a simple classifier that is based on applying the Bayes' rule with a strong independence assumption between the features (Hand and Yu 2001; Kononenko 1990). Due to this assumption,

the calculation of likelihood is simplified to

$$p(\mathbf{x} \mid y = c) = \prod_{j=1}^{d} p(x_j \mid y = c) \tag{4.21}$$

Thus, the posterior probability becomes

$$p(y = c \mid \mathbf{x}) = \frac{p(y = c) \prod_{j=1}^{d} p(x_j \mid y = c)}{p(x_1, x_2, ..., x_d)} \tag{4.22}$$

Since $p(x_1, x_2, ..., x_d)$ is a constant, and the prediction is made by comparing the posterior probabilities and choosing the class with the largest probability. Thus, the denominator can be omitted and the classification rule becomes

$$p(y = c \mid \mathbf{x}) \propto p(y = c) \prod_{j=1}^{d} p(x_j \mid y = c) \tag{4.23}$$

The prior probability $p(y = c)$ can be estimated by assuming equiprobable classes or by calculating the number of instances in the dataset which belong to class c. The estimation of the conditional probability or feature's distribution $p(x_j \mid y = c)$ is done by assuming the distribution of the features (George H John and Langley 2013). The assumption is done based on the type of the feature which is described in the following sections.

4.5.1 Categorical Naive Bayes

First, we discuss the case where the features are categorical data. Assuming there is a set of features and each feature j has its own categorical distribution. For each feature j in the training set, the categorical distribution is estimated conditioned on class c. The conditional probability of label k in feature j given class c is estimated as

$$p(x_j = k \mid y = c) = \frac{N_{kjc}}{N_c} \tag{4.24}$$

where N_{kjc} is the number of times label k appears in feature j which belongs to class c and N_c is the number of instances with class c.

4.5.2 Bernoulli Naive Bayes

Bernoulli distribution is a discrete distribution with only two possible values for the random variable. Bernoulli naive Bayes is typically used for document or text classification in which the features are the binary word occurrence $x_j \in 0, 1$, where x_j is a boolean feature expressing the presence (occurrence) or absence of the j-th word in the document. If the probability of a word's occurrence is p, then the probability of absence is $1 - p$. Thus, the likelihood of document $\mathbf{x}$ given class c is given by

$$p(\mathbf{x} \mid y = c) = \prod_{j=1}^{d} p(x_j \mid y = c) \tag{4.25}$$

which is the product of the conditional probability of each word's occurrence x_j given class c. $p(x_j \mid y = c)$ is the probability of j-th word occurring in class c. This probability can be estimated by counting the number of times j-th word occurs in the documents labeled with class c.

4.5.3 Multinomial Naive Bayes

Multinomial distribution is a generalized form of the binomial distribution, a discrete distribution of a random variable with more than two outcomes. Using multinomial distribution, in the case of document classification, rather than word occurrence vectors, we can use the frequency of a word as a feature. The likelihood of a document given class c is given by

$$p(\mathbf{x} \mid y = c) = \frac{x_j!}{\prod_{j=1}^{d} x_j!} \prod_{j=1}^{d} p(x_j \mid y = c) \tag{4.26}$$

and $p(x_j \mid y = c)$ can be estimated as follows:

$$p(x_j \mid y = c) = \frac{N_j c}{N_c} \tag{4.27}$$

where $N_j c$ is the number of times word j appears in the document with class c.

Numerical Example: Consider the following dataset. Given $\acute{\mathbf{x}} = \{x_1 = \texttt{Sunny}, x_2 = \texttt{Mild}, x_3 = \texttt{Normal}\}$, predict the class of $\acute{\mathbf{x}}$.

Outlook	Temperature	Humidity	Play
Sunny	Hot	High	No
Cloudy	Hot	High	Yes
Rain	Mild	High	Yes
Rain	Cool	Normal	No
Cloudy	Mild	High	Yes
Sunny	Cool	Normal	Yes
Rain	Mild	High	No
Sunny	Mild	High	No

The number of instances is $N = 8$

Prior probability

$p(y = \texttt{Yes}) = \frac{4}{8} = \frac{1}{2}$, $p(y = \texttt{No}) = \frac{4}{8} = \frac{1}{2}$

Conditional probability for x_1

$p(x_1 = \texttt{Sunny} \mid y = \texttt{Yes}) = \frac{1}{4}$, $p(x_1 = \texttt{Sunny} \mid y = \texttt{No}) = \frac{2}{4}$

$p(x_1 = \texttt{Cloudy} \mid y = \texttt{Yes}) = \frac{2}{4}$, $p(x_1 = \texttt{Cloudy} \mid y = \texttt{No}) = \frac{0}{4}$

$p(x_1 = \texttt{Rain} \mid y = \texttt{Yes}) = \frac{1}{4}$, $p(x_1 = \texttt{Rain} \mid y = \texttt{No}) = \frac{2}{4}$

Conditional probability for x_2

$p(x_2 = \texttt{Hot} \mid y = \texttt{Yes}) = \frac{1}{4}$, $p(x_2 = \texttt{Hot} \mid y = \texttt{No}) = \frac{1}{4}$

$p(x_2 = \texttt{Mild} \mid y = \texttt{Yes}) = \frac{2}{4}$, $p(x_2 = \texttt{Mild} \mid y = \texttt{No}) = \frac{2}{4}$

$p(x_2 = \texttt{Cool} \mid y = \texttt{Yes}) = \frac{1}{4}$, $p(x_2 = \texttt{Cool} \mid y = \texttt{No}) = \frac{1}{4}$

Conditional probability for x_3

$p(x_2 = \texttt{High} \mid y = \texttt{Yes}) = \frac{3}{4}$, $p(x_2 = \texttt{High} \mid y = \texttt{No}) = \frac{3}{4}$

$p(x_2 = \texttt{Normal} \mid y = \texttt{Yes}) = \frac{1}{4}$, $p(x_2 = \texttt{Normal} \mid y = \texttt{No}) = \frac{1}{4}$

Posterior probability

$$p(y = \texttt{Yes} \mid \acute{\mathbf{x}}) \propto (\frac{1}{4} \cdot \frac{2}{4} \cdot \frac{1}{4} \cdot \frac{1}{2}) \propto 0.015625$$

$$p(y = \texttt{No} \mid \acute{\mathbf{x}}) \propto (\frac{2}{4} \cdot \frac{2}{4} \cdot \frac{1}{4} \cdot \frac{1}{2}) \propto 0.03125$$

Since $p(y = \texttt{No} \mid \acute{\mathbf{x}}) > p(y = \texttt{Yes} \mid \acute{\mathbf{x}})$, the prediction is No.

4.5.4 Gaussian Naive Bayes

In the case of real-valued features, we typically assume the features are distributed according to the Gaussian distribution. The likelihood of feature x_j given class c is given by

$$p(x_j \mid y = c) = \frac{1}{\sqrt{2\pi\sigma_{jc}^2}} e^{-\frac{(x_j - \mu_{jc})^2}{2\sigma_{jc}^2}}$$

where μ_{jc} is the mean of feature x_j with class c and σ_{jc} is the variance of feature x_j with class c.

4.5.5 Log Trick

Multiplying the conditional probabilities may cause computational instability or known as numerical underflow. This is because the value of $p(\mathbf{x} \mid y = c)$ is often very small especially if $\mathbf{x}$ is a high dimensional vector. Thus, to avoid the numerical underflow, we take the logarithm of the conditional probabilities when applying Bayes rule. Note that the multiplication operation becomes an addition in the logarithm space.

$$\log p(y = c \mid \mathbf{x}) \propto \log(\prod_{j=1}^{d} p(x_j \mid y = c) \cdot p(y = c)) \propto \sum_{j=1}^{d} \log p(x_j \mid y = c) + \log p(y = c)$$

Numerical Example: Given the following dataset where 0 is female and 1 is male. Given $\acute{\mathbf{x}} = \{\acute{x}_1 = 1.73, \acute{x}_2 = 66\}$, predict the gender of $\acute{\mathbf{x}}$.

Height (m)	Weight (kg)	Gender
1.72	68	1
1.83	79	1
1.71	65	0
1.95	83	1
1.68	60	0
1.81	76	1
1.75	69	0
1.67	57	0

The number of instances is $N = 8$

Prior probability

$p(y = \texttt{Male}) = \frac{4}{8} = \frac{1}{2}$, $p(y = \texttt{Female}) = \frac{4}{8} = \frac{1}{2}$

μ and σ^2 for x_1

$\mu_{1,0} = 1.7025$, $\sigma^2_{1,0} = 0.0013$

$\mu_{1,1} = 1.8275$, $\sigma^2_{1,1} = 0.0090$

μ and σ^2 for x_2

$\mu_{2,0} = 62.75$, $\sigma^2_{2,0} = 28.2500$

$\mu_{2,1} = 76.50$, $\sigma^2_{2,1} = 40.3333$

Probability density function of $\acute{\mathbf{x}}$ given class 0.

$$p(\acute{x}_1 \mid y = 0) = \frac{1}{\sigma_{1,0}\sqrt{2\pi}} e^{-\frac{(\acute{x}_1 - \mu_{1,0})^2}{2\sigma^2_{1,0}}}$$
$$= 8.2721$$

$$p(\acute{x}_2 \mid y = 0) = \frac{1}{\sigma_{2,0}\sqrt{2\pi}} e^{-\frac{(\acute{x}_2 - \mu_{2,0})^2}{2\sigma^2_{2,0}}}$$
$$= 0.0622$$

Posterior probability of class 0.

$$p(y = 0 \mid \acute{\mathbf{x}}) \propto \log(p(\acute{x}_1 \mid y = 0) \cdot p(\acute{x}_2 \mid y = 0) \cdot p(y = 0))$$
$$\propto -1.356$$

Probability density function of $\acute{\mathbf{x}}$ given class 1.

$$p(\acute{x}_1 \mid y = 1) = \frac{1}{\sigma_{1,1}\sqrt{2\pi}} e^{-\frac{(\acute{x}_1 - \mu_{1,1})^2}{2\sigma^2_{1,1}}}$$
$$= 2.4799$$

$$p(\acute{x}_2 \mid y = 1) = \frac{1}{\sigma_{2,1}\sqrt{2\pi}} e^{-\frac{(\acute{x}_2 - \mu_{2,1})^2}{2\sigma^2_{2,1}}}$$
$$= 0.0160$$

Posterior probability of class 1.

$$p(y = 1 \mid \acute{\mathbf{x}}) \propto \log(p(\acute{x}_1 \mid y = 1) \cdot p(\acute{x}_2 \mid y = 1) \cdot p(y = 1))$$
$$\propto -3.920$$

Since $p(y = 0 \mid \acute{\mathbf{x}}) > p(y = 1 \mid \acute{\mathbf{x}})$, the prediction is class 0 (female).

4.6 Naive Bayes is a Linear Classifier

In many cases, naive Bayes leads to a linear decision boundary (Rennie et al. 2003). Given an input $\mathbf{x}$, the prediction is 1 if

$$p(y = 1 \mid \mathbf{x}) > p(y = 0 \mid \mathbf{x}) \tag{4.28}$$

Taking a log of both sides and using Bayes rule, we have

$$\log p(\mathbf{x} \mid y=1) + \log p(y=1) - \log p(\mathbf{x} \mid y=0) + \log p(y=0) > 0 \tag{4.29}$$

Suppose the features are Gaussian.

$$\log p(\mathbf{x} \mid y=c) = \mathcal{N}(\mu_c, \sum_c) \tag{4.30}$$

where μ_c and $\sum_c$ are the mean and variance of the distribution. Note that $\sum_c$ is diagonal and symmetric.

$$(\mathbf{x}-\mu_1)^\top \sum_1^{-1}(\mathbf{x}-\mu_1) + \log p(y=1) - (\mathbf{x}-\mu_0)^\top \sum_1^{-1}(\mathbf{x}-\mu_0) + \log p(y=0) > 0 \tag{4.31}$$

Expanding the expression, we have

$$\mathbf{x}^\top \sum_1^{-1}\mathbf{x} - 2\mathbf{x}^\top \sum_1^{-1}\mu_1 + \mu_1^\top \sum_1^{-1}\mu_1 - \mathbf{x}^\top \sum_0^{-1}\mathbf{x} + 2\mathbf{x}^\top \sum_0^{-1}\mu_0 - \mu_0^\top \sum_0^{-1}\mu_0 + \log p(y=1) + \log p(y=0) > 0 \tag{4.32}$$

Assuming both classes share the same covariance matrix, $\sum_1 = \sum_0 = \sum$, the quadratic terms are canceled. We have

$$\mathbf{x}^\top \sum^{-1}(\mu_0 - \mu_1) + \mu_1^\top \sum^{-1}\mu_1 - \mu_0^\top \sum^{-1}\mu_0 + \log p(y=1) + \log p(y=0) > 0 \tag{4.33}$$

where $\mu_1^\top \sum^{-1} \mu_1 - \mu_0^\top \sum^{-1} \mu_0 + \log p(y=1) + \log p(y=0)$ can be considered as a constant C.

Thus, we have

$$\mathbf{x}^\top \sum^{-1}(\mu_0 - \mu_1) + C > 0 \tag{4.34}$$

which is a linear form in $\mathbf{x}$.

4.7 Implementing Linear Classifiers

In this section, we will be implementing perceptron and logistic regression using Scikit-Learn. We will be using the popular iris dataset that contains three different types of irises' (Setosa, Versicolor and Virginica) petal and sepal length features. Thus, the prediction task is a multiclass classification problem which is to classify the data into Setosa, Versicolor or Virginica.

Let us import the function to download the dataset. The name of the function is `load_iris`.

```
from sklearn.datasets import load_iris
iris = load_iris()
```

The object iris is a dictionary (data structure), thus we can explore it using its keys. Here, we acquire the features and create a DataFrame using Pandas. We also assign the column names of the DataFrame which are acquired from the dictionary.

```
import pandas as pd
data_df = pd.DataFrame(iris.data, columns=iris.feature_names)
print(data_df.head())
```

The target is acquired from the dictionary as follows:

```
target = iris.target
```

As usual, we split the dataset into training and test sets. We put aside 30% of the data as the test set.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(data_df, target, test_size=0.3,
    random_state=0)
```

We use standardization to rescale the features.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_std = scaler.fit_transform(X_train)
```

4.7.1 Perceptron

Scikit-Learn provides a module called perceptron to implement perceptron classifier. We set the maximum iteration of the training process to 300. Then, we call `fit` and pass the training set to train the model.

```
from sklearn.linear_model import Perceptron
model_pt = Perceptron(max_iter=300)
model_pt.fit(X_train_std, y_train)
```

We use the model to predict the standardized test set as follows:

```
X_test_std = scaler.transform(X_test)
y_pred = model_pt.predict(X_test_std)
```

Scikit-Learn provides a function to create a confusion matrix of the classification. The confusion matrix allows us to analyze the performance of the classification of each class.

```
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
print(accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
0.9555555555555556
[[16  0  0]
 [ 0 17  1]
 [ 0  1 10]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        16
           1       0.94      0.94      0.94        18
           2       0.91      0.91      0.91        11

    accuracy                           0.96        45
   macro avg       0.95      0.95      0.95        45
weighted avg       0.96      0.96      0.96        45
```

The precision, recall and F-score measures for each class are above 0.90. This shows that model performed well in classifying the classes. The model achieved a high accuracy of 0.96.

4.7.2 Using Pipeline

Scikit-Learn provides a transformer interface called `pipeline` to combine data preprocessing and data modeling steps. `pipeline` is used to chain multiple estimators into one and it is useful since normally we have a fixed sequence of steps in processing the data such as standardization followed by modeling and classification.

`make_pipeline` is a function for creating a `pipeline`. Let us import the function and create the pipeline to standardize the data and build the logistic regression classifier.

```
from sklearn.pipeline import make_pipeline
model_pt_pipeline = make_pipeline(StandardScaler(), Perceptron(max_iter=300))
model_pt_pipeline.fit(X_train, y_train)
```

For convenience, we create a function to print out the accuracy score, confusion matrix and classification report.

```
def print_evaluation(y_test, y_pred):
    print(accuracy_score(y_test, y_pred))
    print(confusion_matrix(y_test, y_pred))
    print(classification_report(y_test, y_pred))
```

Using the pipeline that we created above, we pass the test set to perform the predictions. Then, we call the `print_evaluation` to display the results. Notice that the standardization is internally done in the pipeline before it is used for prediction.

```
y_pred = model_pt_pipeline.predict(X_test)
print_evaluation(y_test, y_pred)
```

```
0.9555555555555556
[[16  0  0]
 [ 0 17  1]
 [ 0  1 10]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        16
           1       0.94      0.94      0.94        18
           2       0.91      0.91      0.91        11

    accuracy                           0.96        45
   macro avg       0.95      0.95      0.95        45
weighted avg       0.96      0.96      0.96        45
```

A perceptron classifier can also be built by using `SGDClassifier` module. Using the module, we need to specify the following parameters:

- `loss` is the loss function to be used
- `eta0` is the value of the learning rate
- `learning_rate` is the learning rate schedule, either constant or changing
- `tol` is the stopping criteria of the training
- `penalty` is the regularization term, either '`l1`' or '`l2`'. By default, this parameter is set to '`l2`'. However, regularization is not available in perceptron. Therefore, this parameter will be set to `None`.
- `max_iter` is the training maximum number of epochs

Let us import the module and create the object by specifying the parameters. We set `loss` to '`perceptron`', `eta0` to 1, `learning_rate` to '`constant`', `penalty` and `tol` to `None` and `max_iter` to 300.

```
model_sgd_pipeline = make_pipeline(StandardScaler(), SGDClassifier(loss='perceptron',
    eta0=1, learning_rate='constant', penalty=None, tol=None, max_iter=300))
model_sgd_pipeline.fit(X_train, y_train)
y_pred = model_sgd_pipeline.predict(X_test)
print_evaluation(y_test, y_pred)
```

```
0.9555555555555556
[[16  0  0]
 [ 1 16  1]
 [ 0  0 11]]
              precision    recall  f1-score   support

           0       0.94      1.00      0.97        16
           1       1.00      0.89      0.94        18
           2       0.92      1.00      0.96        11

    accuracy                           0.96        45
   macro avg       0.95      0.96      0.96        45
weighted avg       0.96      0.96      0.96        45
```

4.7.3 Logistic Regression

Now, let us build a logistic regression classifier to classify the dataset. Scikit-Learn provides a module called `LogisticRegression` to implement logistic regression classifier. Similarly, we use `pipeline` to chain the standardization and modeling steps.

```
from sklearn.linear_model import LogisticRegression
model_lg_pipeline = make_pipeline(StandardScaler(), LogisticRegression())
model_lg_pipeline.fit(X_train, y_train)
```

Here, we evaluate the performance of the model as follows:

```
y_pred = model_lg_pipeline.predict(X_test)
print_evaluation(y_test, y_pred)
```

```
0.9777777777777777
[[16  0  0]
 [ 0 17  1]
 [ 0  0 11]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        16
           1       1.00      0.94      0.97        18
           2       0.92      1.00      0.96        11

    accuracy                           0.98        45
   macro avg       0.97      0.98      0.98        45
weighted avg       0.98      0.98      0.98        45
```

Logistic regression achieved a better result compared to perceptron. The precision, recall and F-score measures for each class show a higher value, and the model's accuracy is improved by 0.02.

A logistic regression classifier can be built using the same `SGDClassifier` module. We need to specify the following parameters:

- loss function is set to '`log_loss`'
- `max_iter` is the maximum number of epochs

```
from sklearn.linear_model import SGDClassifier
SGDClassifier(loss='log_loss', max_iter=300)
```

4.7.4 Gaussian Naive Bayes

Now, let us build a naive Bayes classifier to classify the dataset. Scikit-Learn provides a module called `GaussianNB` to implement Gaussian naive Bayes classifier. Besides `GaussianNB`, Scikit-Learn provides `BernoulliNB`, `MultinomialNB` and `CategoricalNB` for modeling categorical data and text data. Similarly, we use `pipeline` to chain the standardization and modeling steps.

```
from sklearn.naive_bayes import GaussianNB
model_nb_pipeline = make_pipeline(StandardScaler(), GaussianNB())
model_nb_pipeline.fit(X_train, y_train)
```

We evaluate the performance of the model using the test set as follows:

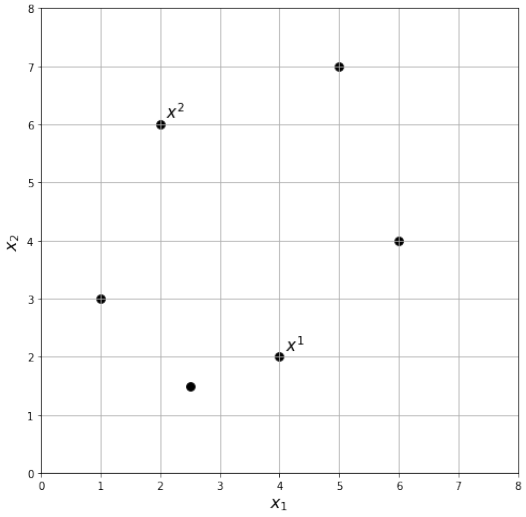
```
y_pred = model_nb_pipeline.predict(X_test)
print_evaluation(y_test, y_pred)
```

```
1.0
[[16  0  0]
 [ 0 18  0]
 [ 0  0 11]]
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        16
           1       1.00      1.00      1.00        18
           2       1.00      1.00      1.00        11

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

Naive Bayes achieved a better result compared to perceptron and logistic regression. Naive Bayes achieved a perfect prediction for each class with the model's accuracy 1.0.

Exercises

1. Describe the difference between perceptron and logistic regression.
2. State the mathematical representation for logistic regression.
3. Describe the difference between softmax and sigmoid function.
4. Given a hyperplane $h(x) = -8.0 + x_1 + 1.6x_2$ and a set of data points as shown in the following plot:



   (a) Given a perceptron with hyperplane h, compute the predicted value of data point x^1.

   (b) Given a logistic regression with hyperplane h, compute the predicted value of data point x^2.

5. Consider the following training data with two classes 0 and 1.

x_1	x_2	y
yes	mid	0
no	large	1
no	mid	1
yes	large	1
no	small	0

(a) Calculate the prior probability of class 0 and class 1.

(b) Calculate the conditional probability of the features.

(c) Given a new data $x = x_1 = \texttt{yes}, x_2 = \texttt{small}$, predict the class of the data.

Chapter 5

k-Nearest Neighbors

5.1 Introduction

Machine learning algorithms can be categorized into parametric and non-parametric. A parametric model refers to a model with a specific functional form that is defined by a fixed number of parameters. The machine learning algorithms that we have seen in the previous chapters, linear regression and logistic regression are examples of parametric models whereby the models are in linear form. The advantages of parametric models are simple, fast to compute and require less training data due to their strong assumption about the data. However, the assumption may not hold which may lead to large errors. Non-parametric model refers to models that whose number of parameters grows with the amount of training data. Unlike parametric models, non-parametric models do not assume any specific functional form. The models do not make strong assumptions about the data; hence they are more flexible and powerful. However, the complexity of the models grows with the size of the training set and can be computationally infeasible for very large datasets.

k-nearest neighbors is a non-parametric algorithm that can be used for classification and regression. The algorithm assumes that instances that belong to the same class tend to be close to each other (Cover and Hart 1967). In other words, data points are not randomly distributed across the feature space, but rather, spatially clustered as shown in Figure 5.1. As shown in the figure, the data points (instances) belong to three classes, with the label y coded as 0 for blue, 1 for orange and 2 for green. Data points belonging to blue are distributed in the upper left region of the feature space. The green and orange data points are distributed in the lower right of the feature space with the green data points occupying the upper part of the region while orange data points occupying the lower part of the region.

On the basis of this assumption, if we are given a new data point, the label of the data point can be determined based on the neighboring data points. Consider a new data point indicated by "x" marker in red color. Based on its location, we can somehow deduce that the data point belongs to orange. This is because there are more orange data points neighboring the new data point.

Specifically, k-nearest neighbors algorithm uses k data points in the training set that are nearest to the test input (new data point) to decide the label of the test point. Let $\acute{x}$ denotes a test input and $S_n \subseteq D_{tr}$ denotes the set of k nearest neighbors of $\acute{x}$. The k nearest neighbors are data points with distances smaller than the distances of other data points. Let input-output pairs in the training set

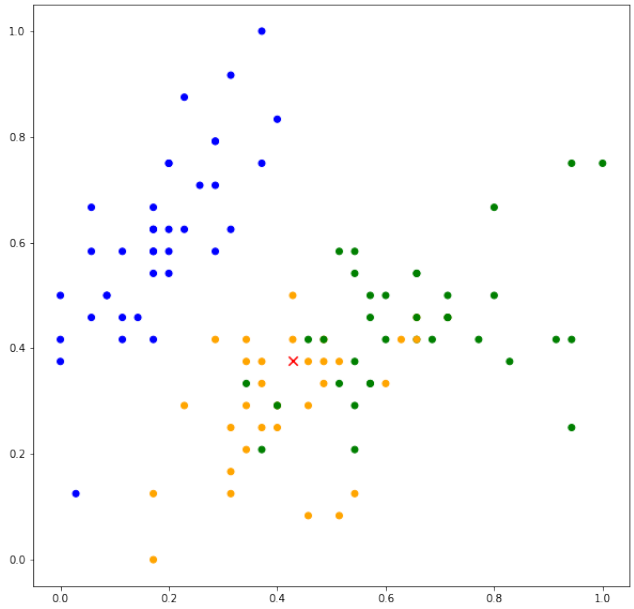


Figure 5.1 Data points appear in clusters of three classes

but not in S_n be $\forall(x, y) \in D_{tr} \setminus S_n$. The distance of nearest neighbors, x_n is

$$\max(\text{dist}(x_n, \acute{x})) \leq \text{dist}(x, \acute{x}) \tag{5.1}$$

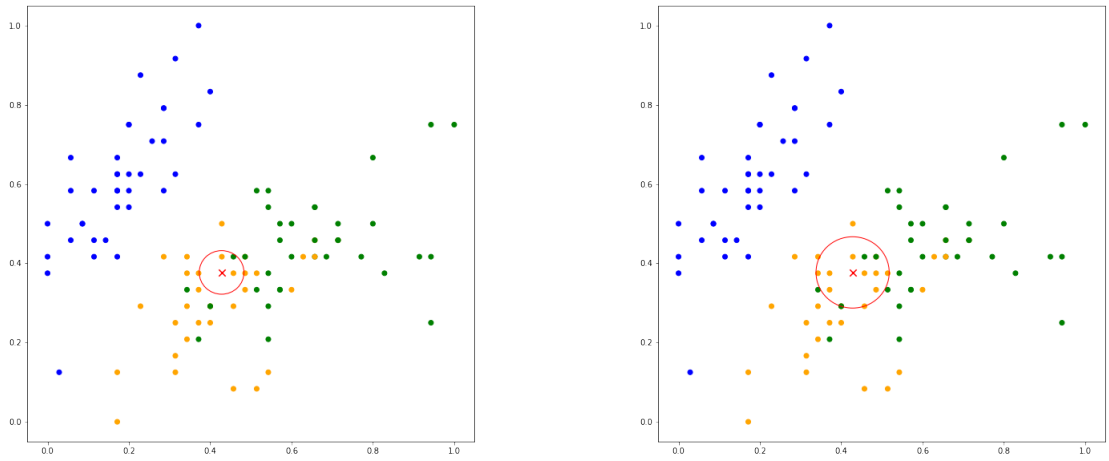
The prediction of k-nearest neighbor classifier is defined as follows:

$$p(y = c|\acute{x}, k) = \frac{1}{k} \sum_{(x_n, y_n) \in S_n} I(y_n = c) \tag{5.2}$$

where $I(e) = \begin{cases} 1 & \text{if } e \text{ is true} \\ 0 & \text{if } e \text{ is false} \end{cases}$ k-nearest neighbors is an example of memory-based or instance-based learning. The predictive model is the set of training instances and k is the parameter of the algorithm (needs to be set), which determines the number of neighbors that will be considered in the prediction. Consider $k = 3$, the neighbors consist of two data points belonging to orange and one data point belonging to green as shown in Figure 5.2(a). The predicted values are $p(y = 0, \mathbf{x}) = 0.0$, $p(y = 1, \mathbf{x}) = 0.667$, $p(y = 2, \mathbf{x}) = 0.333$.

Thus, the test point is classified as orange. Now when $k = 10$, more neighboring data points are considered in the classification. As can be seen in Figure 5.2(b), out of ten data points, eight data points belong to orange while two belong to green. The predicted values are $p(y = 0, \mathbf{x}) = 0.0$, $p(y = 1, \mathbf{x}) = 0.8$, $p(y = 2, \mathbf{x}) = 0.2$.

Note the increase in probability estimate of $p(y = 1, \mathbf{x})$. In general, the prediction is more susceptible to noise when the value of k is small. A large value of k reduces the effect of noise on the prediction but is more computationally expensive. However, the optimal value of k depends on the data and should be empirically selected using the validation set. Furthermore, k must be a positive and non-zero value. It is also advisable to choose an odd value to avoid equal scores for binary classification.



(a) (b)

Figure 5.2 The k value defines the number of neighboring data points to be checked to determine the prediction

5.2 Distance Functions

k-nearest neighbor algorithm relies on a distance function to determine the neighbors of the test input. The most commonly used distance function is the Euclidean distance. Euclidean distance measures the straight line distance between two data points (Cohen, Lee, and Sklar 2016). The distance function is defined as follows:

$$\texttt{dist}(x_n, \acute{x}) = \sqrt{\sum_{i=1}^{|S_n|}(x_n^i - \acute{x^i})^2} \tag{5.3}$$

An alternative to Euclidean distance is the Manhattan distance. Manhattan distance is the distance between two points measured at the right angle along the vertical and horizontal axes (Cohen, Lee, and Sklar 2016).

$$\texttt{dist}(x_n, \acute{x}) = \sum_{i=1}^{|S_n|}(|x_n^i - \acute{x^i}|) \tag{5.4}$$

Hamming distance is used for categorical variables whereby the distance is measured as the number of different symbols between the two instances (B. Waggener and W. N. Waggener 1995). For example, the Hamming distance between two instances consists of three features as follows is 2.

$\mathbf{x}^1 = (\text{Yes}, \text{Pass}, \text{High})$

$\mathbf{x}^2 = (\text{No}, \text{Fail}, \text{High})$

5.3 Implementing k-Nearest Neighbors

In this section, we will be implementing k-nearest neighbors using Scikit-Learn. First, we demonstrate k-nearest neighbors for classification problem, then k-nearest neighbors for regression problems. We will be using the breast cancer dataset which is available from Scikit-Learn. The prediction task is a binary classification problem which is to classify the data into malignant and benign.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

dataset = load_breast_cancer()
train_data, X_test, train_target, y_test = train_test_split(dataset.data,
    dataset.target, test_size=0.3, random_state=10)
```

We need to set aside a part of the training set for validation purposes. Later, the validation set will be used to find the optimal number of nearest neighbors.

```
X_train, X_vald, y_train, y_vald = train_test_split(train_data, train_target,
    test_size=0.2, random_state=10)
```

We create a function to print out the accuracy score, confusion matrix and classification report.

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

def print_clf_evaluation(y_test, y_pred):
    print(accuracy_score(y_test, y_pred))
    print(confusion_matrix(y_test, y_pred))
    print(classification_report(y_test, y_pred))
```

5.3.1 k-Nearest Neighbors Classifier

k-nearest neighbors classifier can be built by importing `KNeighborsClassifier` module. We set the following parameters:

- `n_neighbors` is the number of neighbors, k to use. The default value is 5.
- metric is the distance metric to use for the tree. The default metric is '`minkowski`', and with `p` = 2 is equivalent to the standard Euclidean distance. The other metric values are '`manhattan`' and '`cosine`'.
- `p` is the power parameter for the minkowski metric. The default value is 2. If `p` is 1, this is equivalent to using Manhattan distance.

Let us use the default value for those parameters. Later, we will perform an experiment to find the optimal `n_neighbors` value.

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline

knn_clf_pipeline = make_pipeline(StandardScaler(), KNeighborsClassifier())
knn_clf_pipeline.fit(X_train, y_train)
```

We evaluate the model on the test set.

```
y_pred = knn_clf_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.9707602339181286
[[ 57   2]
 [  3 109]]
              precision    recall  f1-score   support

           0       0.95      0.97      0.96        59
           1       0.98      0.97      0.98       112

    accuracy                           0.97       171
   macro avg       0.97      0.97      0.97       171
weighted avg       0.97      0.97      0.97       171
```

Now, let us find the optimal value of `n_neighbors`. For this example, we will be using `matplotlib`, a plotting library to produce charts and graphs. `pyplot` is an interface for creating various types of plots. For example, to create a line chart, we use plot and pass the coordinates of `x` and `y` as shown in Figure 5.3.

```
from matplotlib import pyplot as plt
import numpy as np

x = np.array([0, 20])
y = np.array([0, 10])
plt.plot(x, y)
```

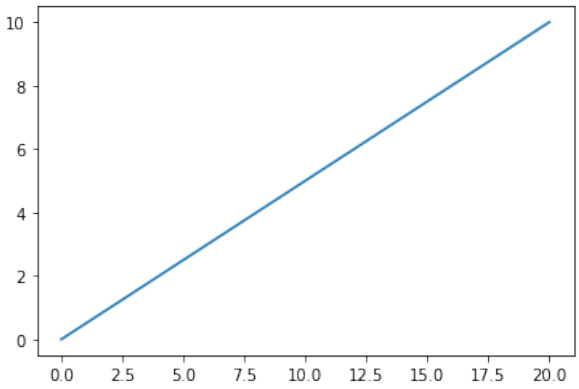


Figure 5.3 A line chart

One way to find the optimal value of `n_neighbors` is to implement a loop that iterates over a range of `n_neighbors` e.g. 1 to 20. For each iteration, we train the model with the corresponding `n_neighbors` value and evaluate the model on the validation set. The validation score (accuracy) is used to determine the optimal value of `n_neighbors`. We keep the score of every iteration in list `scores`.

```
scores = []
max_score = 0
best_k = 0
for k in range(1,20):
    knn_clf_pipeline = make_pipeline(StandardScaler(),
                                     KNeighborsClassifier(n_neighbors=k))
    knn_clf_pipeline.fit(X_train, y_train)
    score = knn_clf_pipeline.score(X_vald, y_vald)
    if score > max_score:
        max_score = score
        best_k = k
    scores.append(score)
```

We plot the score against `n_neighbors`. The optimal `n_neighbors` is 4 as shown in Figure 5.4.

```
plt.plot(np.arange(1,20), scores)
plt.xticks(np.arange(1,20))
plt.grid()
plt.show()
print('Best k: ', best_k)
```

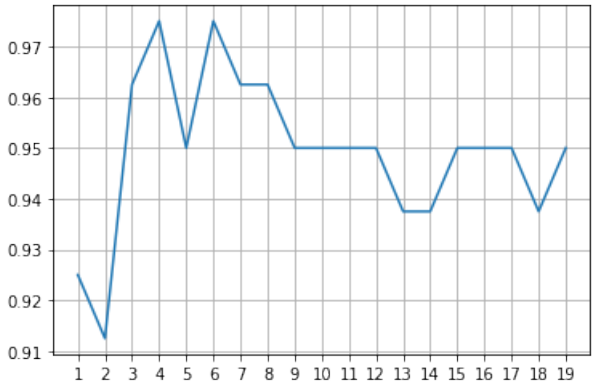


Figure 5.4 Plot of model's score for every k value

We rebuild the model using the optimal value of `n_neighbors`. Then, we evaluate the model on the test set.

```
knn_clf_pipeline = make_pipeline(StandardScaler(),
                                 KNeighborsClassifier(n_neighbors=best_k))
knn_clf_pipeline.fit(X_train, y_train)
y_pred = knn_clf_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.9824561403508771
[[ 59   0]
 [  3 109]]
              precision    recall  f1-score   support

           0       0.95      1.00      0.98        59
           1       1.00      0.97      0.99       112

    accuracy                           0.98       171
   macro avg       0.98      0.99      0.98       171
weighted avg       0.98      0.98      0.98       171
```

5.3.2 Using Cross-Validation

So far, we use the hold-out method to split the dataset. It would be better to use the cross-validation to find the optimal value of `n_neighbors` especially if the dataset is limited in the number of samples. Scikit-Learn provides a function called `cross_val_score` to implement cross-validation whereby the function needs to be called by passing the model, the training set and the number of folds (optional).

Now, let us modify the for loop by replacing the line where we fit the model with the function. We define 3 folds for the cross-validation. The function returns 3 accuracy scores, thus, we compute the mean score and store it in list `cv_scores`.

```
from sklearn.model_selection import cross_val_score

cv_scores  = []
max_score = 0
best_k = 0
for k in range(1,20):
    knn_clf_pipeline = make_pipeline(StandardScaler(),
    KNeighborsClassifier(n_neighbors=k))
    scores = cross_val_score(knn_clf_pipeline, train_data, train_target, cv=3)
    mean_score = scores.mean()
    if mean_score > max_score:
        max_score = mean_score
        best_k = k
    cv_scores.append(mean_score)
```

We plot the model's scores for every k as shown in Figure 5.5.

```
plt.plot(np.arange(1,20), cv_scores)
plt.xticks(np.arange(1,20))
plt.grid()
plt.show()
print('Best k: ', best_k)
```

Then, we build the model using the best k, and evaluate the model on the test set as follows:

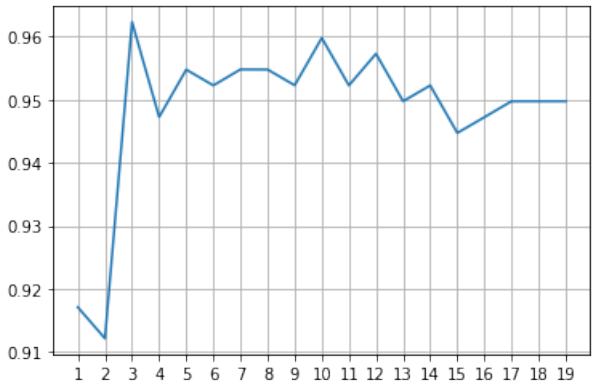


Figure 5.5 Plot of model's score for every k value using cross-validation

```
knn_clf_pipeline = make_pipeline(StandardScaler(),
    KNeighborsClassifier(n_neighbors=best_k))
knn_clf_pipeline.fit(train_data, train_target)
y_pred = knn_clf_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.9824561403508771
[[ 57   2]
 [  1 111]]
              precision    recall  f1-score   support

           0       0.98      0.97      0.97        59
           1       0.98      0.99      0.99       112

    accuracy                           0.98       171
   macro avg       0.98      0.98      0.98       171
weighted avg       0.98      0.98      0.98       171
```

5.3.3 k-Nearest Neighbors Regressor

In this section, we will be implementing k-nearest neighbors for a regression problem. We will be using California Housing dataset. First, we load the dataset and split it into training and test sets.

```
from sklearn.datasets import fetch_california_housing
dataset = fetch_california_housing()
X_train, X_test, y_train, y_test = train_test_split(dataset.data, dataset.target,
    test_size=0.3, random_state=10)
```

k-nearest neighbors classifier can be built by importing `KNeighborsRegressor` module. The parameters to be set are similar to `KNeighborsRegressor` which are `n_neighbors`, `metric` and `p`.

```
knn_reg_pipeline = make_pipeline(StandardScaler(), KNeighborsRegressor(n_neighbors=3,
    metric='manhattan'))
knn_reg_pipeline.fit(X_train, y_train)
```

We create a function to calculate the regression loss.

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
def print_reg_evaluation(y_test, y_pred):
    print('Mean Absolute Error: ', mean_absolute_error(y_test, y_pred))
    print('Mean Squared Error: ', mean_squared_error(y_test, y_pred))
    print('R-squared: ', r2_score(y_test, y_pred))
```

```
y_pred = knn_reg_pipeline.predict(X_test)
print_reg_evaluation(y_test, y_pred)
```

```
Mean Absolute Error:  0.4195496936907838
Mean Squared Error:  0.39638406038650403
R-squared:  0.7076040506450141
```

Exercises

1. Explain parameter k in k-nearest neighbors algorithm.
2. Explain how does the k-nearest neighbors algorithm make the predictions on the unseen data.
3. Explain why k-nearest neighbors algorithm is not recommended for large datasets.
4. Explain how to choose the optimal value of k in k-nearest neighbors algorithm.
5. Explain k in the context of bias-variance tradeoff.

Chapter 6

Decision Tree

6.1 Introduction

Decision tree is one of the most widely used machine learning algorithms in practice due to its being easy to understand and implement. More importantly, its predictions are understandable, making the decision process easy to understand. Decision tree is a non-parametric algorithm that can be used for both classification and regression problems (Breiman et al. 2017). The assumption of the decision tree algorithm is similar to k-nearest neighbors whereby data points are not randomly distributed in the feature space. Instead, data points belonging to a class tend to be close to each other, thus, appearing in clusters. K-nearest neighbors algorithm simply stores the training instances to perform predictions. Although the k-nearest neighbors algorithm is simple and can be powerful, it is memory intensive and computationally inefficient especially when the training set is large. Furthermore, what we are interested in is only the prediction. Consider a classification problem of yes and no. If a test point falls in a cluster of yes, we will be able to immediately know that the test point belongs to Yes before we calculate the distances. Therefore, the distance calculation is not really needed if the regions of Yes and No have been defined and we know in which region the test points fall.

This idea is implemented in the decision tree algorithm. The decision tree algorithm does not store the training data. Instead, the training data is used to build a tree structure consisting of decision nodes and leaf nodes that divides that feature space into regions of similar labels. Consider a tree structure and its classification regions in Figure 6.1. The first node in the tree structure is the root node, which represents the whole training set which consists of two input features, temperature, x_1 and humidity, x_2 and two labels, yes and no. This root node represents the first split whereby the training set is split along one dimension (temperature) by defining a split-point or a threshold at 10^o. The split results in two sets of instances, one with all instances with $x_1 \leq 10^o$ and the others with x_2 at 50%. Then, the region $x_1 \leq 10^o$ is split along x_2 at 50% and the region $x_1 > 10^o$ is split along x_2 at 30%.

The decision tree has three decision nodes including the root node, which represent the condition for each feature split. The result of this process produces two regions representing the two labels indicated by the last four nodes known as the leaf nodes. Thus, in decision tree, we just need to store the conditions of the split and the labels. Given a test point, a decision tree performs the prediction by evaluating the test point against the condition of each node and traversing through the true branches until it reaches a leaf node. For example, given a test input $\mathbf{x} = (45, 15)$. The

condition of the root node is false ($x_2 > 10$). Thus we take the right branch and the following condition is false ($x_1 > 30$). The right branch is taken and the input is predicted as Yes.

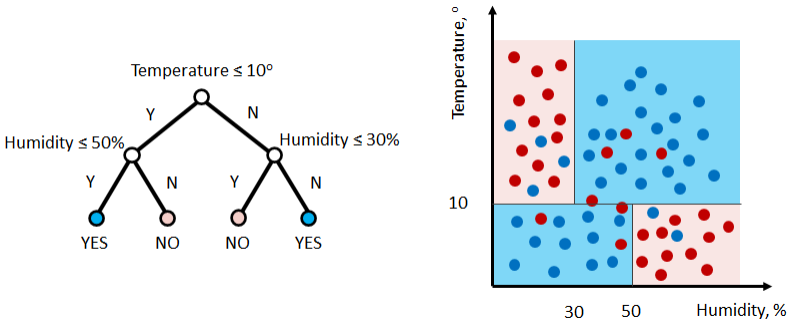


Figure 6.1 Decision tree and the classification regions

Decision tree algorithm can be used for regression problems. Since the predicted value is a numerical value, the leaf nodes represent the mean of the responses of the instances in the regions. Consider a tree structure and its regression regions in Figure 6.2. The predicted values are estimated by the mean of the responses in the regression regions.

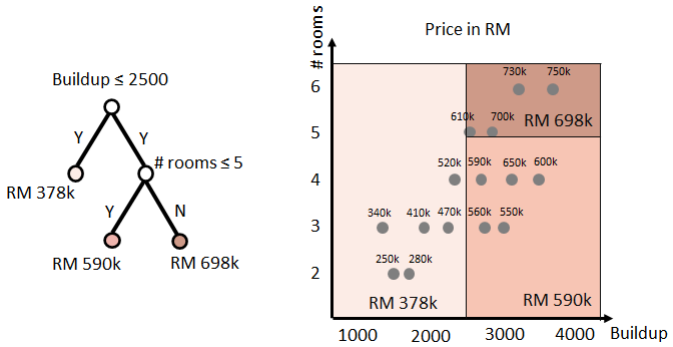


Figure 6.2 Decision tree and its regression (estimated) values

6.2 Building a Tree

The accuracy of the prediction depends on where the split-point is defined for each decision node. To find the optimal split, we measure the reduction in randomness or uncertainty after the feature is split (Quinlan 1986). The reduction in randomness is known as information gain. The information gain of splitting the set or node t along feature, x_j at a split-point, s is given as follows:

$$G(t, x_j, s) = H(t) - (\frac{|t_l|}{|t|}H(t_l) + \frac{|t_r|}{|t|}H(t_r)) \tag{6.1}$$

where t_l and t_r are the nodes (sets) of the left region (branch) and right region (branch) respectively after splitting t at split-point s. $\frac{|t_\eta|}{|t|}$ is the fraction of instances in branch t_η. $H(t_\eta)$ is the impurity function which measures the quality of split or the homogeneity of the labels in set t_η.

In the classification setting, there are two commonly used impurity functions which are entropy and

Gini. Entropy measures the uncertainty of the feature set. Entropy function is defined as follows:

$$H(t) = -\sum_{c=1}^{C} p_c \log_2 p_c \tag{6.2}$$

where C is the number of class labels, p_c is the probability of selecting an instance of class c in node t.

Gini measures the probability of a randomly chosen instance being wrongly labeled. Gini function is defined as follows:

$$H(t) = \sum_{c=1}^{C} p_c(1 - p_c) = \sum_{c=1}^{C} p_c - \sum_{c=1}^{C} p_c^2 = 1 - \sum_{c=1}^{C} p_c^2 \tag{6.3}$$

where C is the number of class labels, p_c is the probability of selecting an instance of class c in node t. For two classes, the node impurity is maximum when the subsets have an equal number of instances in which the probability of both classes is 0.5 as shown in Figure 6.3.

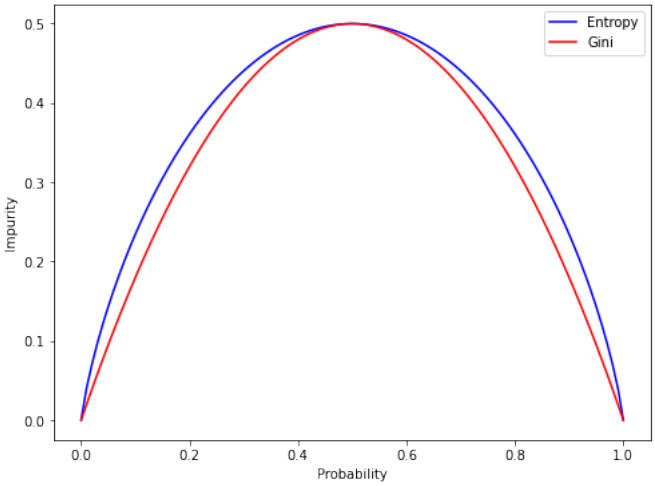


Figure 6.3 Impurity measures for binary classification. Entropy has been scaled to a range of [0,0.5]

Numerical Example: Consider the following split as shown in Figure 6.4. Calculate the entropy and Gini measures and the information gain.

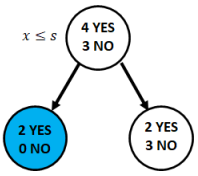


Figure 6.4 A decision node

Information gain of the split with entropy measure:

$H(t) = -(\frac{4}{7}\log_2\frac{4}{7} + \frac{3}{7}\log_2\frac{3}{7}) = 0.9852$

$H(t_l) = -(\frac{2}{2}\log_2\frac{2}{2} + \frac{0}{2}\log_2\frac{0}{2}) = 0.0$

$H(t_r) = -(\frac{2}{5}\log_2\frac{2}{5} + \frac{3}{5}\log_2\frac{3}{5}) = 0.9710$

$G(t, x_1, s) = 0.9852 - \frac{2}{7}(0) - \frac{5}{7}(0.9710) = 0.2916$

Information gain of the split with Gini measure:

$H(t) = 1 - [(\frac{4}{7})^2 + (\frac{3}{7})^2] = 0.4898$

$H(t_l) = 1 - [(\frac{2}{2})^2 + (\frac{0}{2})^2] = 0.0$

$H(t_r) = 1 - [(\frac{2}{5})^2 + (\frac{3}{5})^2] = 0.4800$

$G(t, x_1, s) = 0.4898 - \frac{2}{7}(0) - \frac{5}{7}(0.4800) = 0.1469$

In the regression setting, the impurity is measured by the sum of squared errors. The sum of squared errors is defined as follows:

$$E(t) = \sum_{i=1}^{|t|}(y^i - \hat{y})^2 \tag{6.4}$$

where y^i is the response of i-th instance and $\hat{y} = \frac{1}{|t|}\sum_{i=1}^{|t|} y^i$ is the mean response.

Numerical Example: Consider the following data. Calculate the information gain if the x is split at 2010 ($x \leq 2010$)

$$x = \begin{bmatrix} 2010 \\ 2015 \\ 2012 \\ 2000 \\ 2018 \\ 2014 \\ 2008 \\ 2011 \end{bmatrix}$$

$$y = \begin{bmatrix} 0.20 \\ 0.35 \\ 0.25 \\ 0.15 \\ 0.40 \\ 0.27 \\ 0.45 \\ 0.26 \end{bmatrix}$$

$E(t) = \sum_{i=1}^{|t|}(y^i - \hat{y})^2 = 0.0719$

$E(t_l) = \sum_{i=1}^{|t_l|}(y^i - \hat{y})^2 = 0.05167$

$E(t_r) = \sum_{i=1}^{|t_r|}(y^i - \hat{y})^2 = 0.01732$

$G(t, x, s) = 0.0719 - \frac{3}{8}(0.05167) - \frac{5}{8}(0.01732) = 0.0416$

Based on the information gain equation above, the optimal split can be achieved when the sum of impurity of both branches is minimum (minimum uncertainty in both branches). Therefore, the problem of finding the optimal split can be viewed as finding the best feature and the best split-point for that feature which maximizes the information gain. Given a node t, find the best feature and the best split-point can be expressed as follows:

$$(x_j, s) = \arg \max_{j \in \{1,2,\ldots,d\}, s \in S_{x_j}} G(t, x_j, s) \tag{6.5}$$

where d is the number of input features, S_{x_j} is the set of possible split-points for feature x_j. For numerical features, the set of possible split-points can be obtained by sorting the unique values of x_j^i. For categorical features, the set of possible thresholds is replaced with the set of feature labels. The algorithm to build a decision tree is given below. The algorithm builds a decision tree with leaf nodes composed of one class or pure (instances in the node belong to the same class) as shown in Algorithm 3 (Utgoff 1989).

Algorithm 3 Decision Tree Algorithm

1: **repeat**
2: (x_j, t) = compute $G(t, x_j, s)$ where $j \in \{1, 2, ..., d\}, s \in S_{x_j}$
3: (t_l, t_r) = split node t at s
4: **until** all leaf nodes are pure

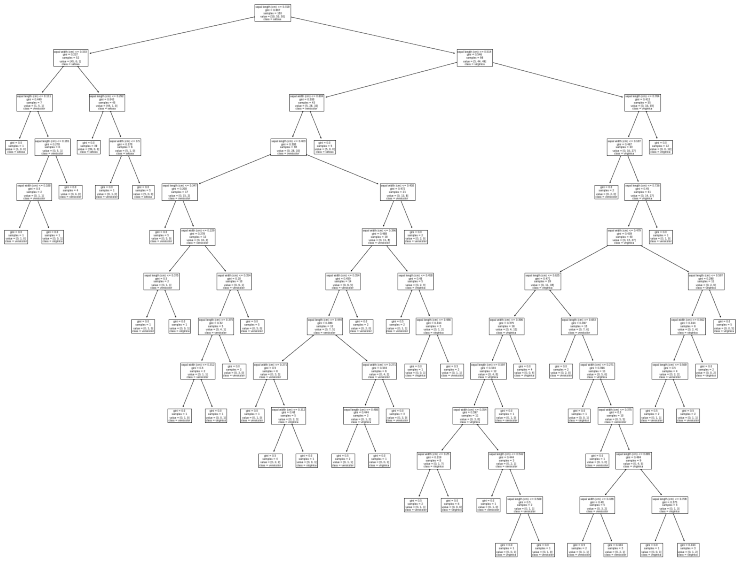
6.3 Pruning a Tree

The algorithm builds a decision tree with zero error (misclassification) because it will keep on splitting until all the nodes are composed of one class (pure leaf nodes). This may cause overfitting (the model to overfit the data) (Breslow and Aha 1997). Figure 6.5 illustrates a decision tree classifier fitted with the iris dataset. Iris dataset contains data about three different iris species: iris setosa (Setosa), iris virginica (Virginica) and iris versicolor (Versicolor). Figure 6.5(a) shows the structure of the tree whereby the tree grows to its full depth. Note the complexity of the tree due to the large number of decision nodes. Figure 6.5(b) shows the decision surface of the tree. The figure shows a few blue regions with a single data point which indicates overfitting. These decision boundaries are induced due to the algorithm continues splitting until all leaf nodes are pure. Consider a test point indicated by the black cross in Figure 6.5(b). The test point will be classified as Virginica although the test point falls in the region where the majority of the data points belong to Versicolor. To prevent overfitting when building the tree, we can limit the number of splits in the tree structure which is known as pruning. In general, there are two approaches to tree pruning: pre-pruning and post-pruning.

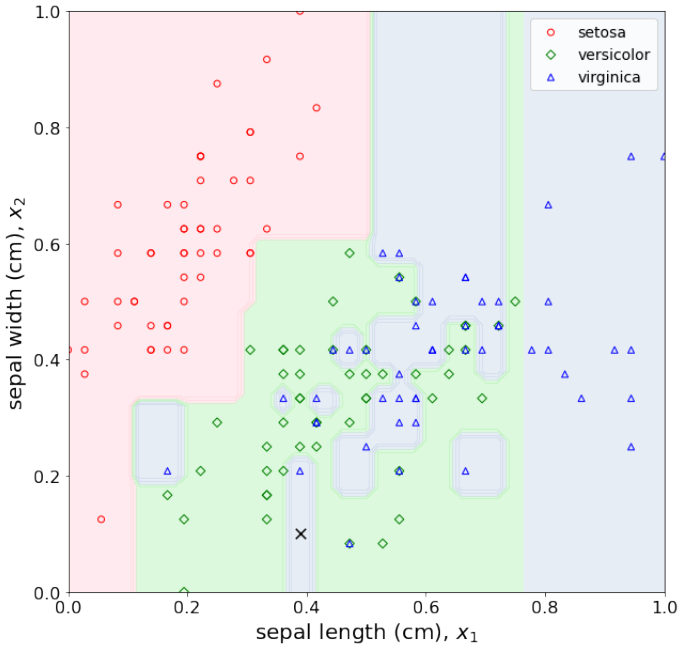
6.3.1 Pre-pruning

In pre-pruning, we stop the tree-building process if the information gain of the split does not exceed a certain threshold. The threshold value is defined to justify the splitting of the node. In this way, the tree will not grow too large. We can also specify the maximum depth of the tree structure. The depth of the tree is defined by the maximum distance from the root node to a leaf node. Other common parameters that can be defined are the minimum number of instances required to split the node, the minimum number of instances to be a leaf and the maximum number of leaf nodes. Figure 6.6(a) shows a pruned decision tree by specifying the maximum depth to 4. The distance is counted by the number of paths from the root node to a leaf node. The decision boundaries of the tree are shown in Figure 6.6(b). We can see that the blue regions now belong to green. Therefore, using the pruned decision tree, the black cross will be classified as Versicolor. Although pre-pruning can prevent overfitting, fine-tuning the parameters can be difficult and sometimes the obtained model is not optimal.

Since the algorithm of building the tree is stopped before the misclassification rate is zero, some of the leaf nodes will not be pure. How do we assign which class to a leaf node? The leaf nodes are assigned to the class with the majority instances. For example, if leaf node s is composed of 5 instances belonging to positive class and 2 instances belonging to negative class. Node t will be

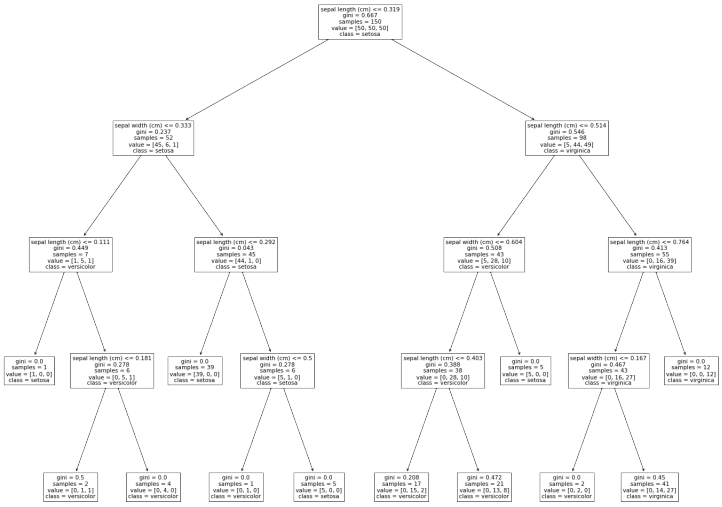


(a)

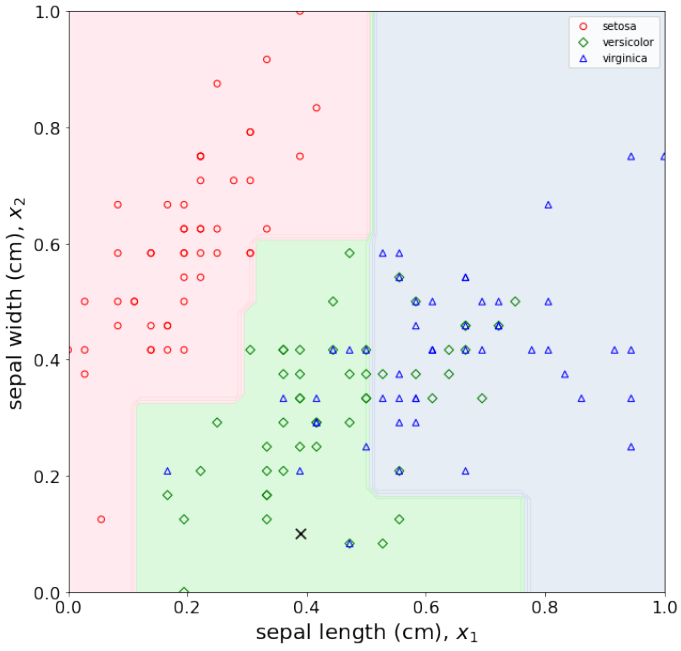


(b)

Figure 6.5 An overfitting decision tree fitted with iris dataset (a) the structure of the decision tree and (b) the decision boundary of the tree



(a)



(b)

Figure 6.6 The pruned decision tree (a) the structure of the tree with maximum depth of 4 and (b) the decision boundary of the tree

assigned to positive class. The classification of a test point is given as follows:

$$c = \arg\max_c p(y = c|t) \tag{6.6}$$

where $p(y = c|\hat{x}, t)$ is the predicted probability of class c given the test point which is based on the proportion of instances belong to class c in node t. The probability of misclassification of leaf node t is

$$r(t) = 1 - \max_c p(y = c|t) \tag{6.7}$$

6.3.2 Post-pruning

The post-pruning is to let the tree grow until the error rate is zero (full tree). Then, the size of the tree is reduced by removing the insignificant branches. This is known as cost-complexity pruning. In this method, we prune the tree by considering the error rate of the tree and the complexity of the tree. The tree complexity is defined as the number of leaf nodes. The higher the number of leaf nodes, the higher the complexity because the tree has more possibilities in partitioning the training set.

Specifically, the pruning method is governed by the cost-complexity measure. The function is to be minimized while pruning the tree. The function is given as follows:

$$R_\alpha(T) = R(T) + \alpha|\tilde{T}| \tag{6.8}$$

where $|\tilde{T}|$ is the tree complexity, α is the complexity parameter. $R(T)$ is the error rate of tree T which is defined as follows:

$$R(T) = r(t)p(t) \tag{6.9}$$

where $r(t)$ is the probability of misclassification of leaf node t. $p(t) = \frac{N(t)}{N}$ where $N(t)$ is the number of instances in node t and N is the total number of instances.

$R(T)$ favors a larger tree because a large tree minimizes the error rate while $|\tilde{T}|$ favors a smaller tree because a small tree minimizes the complexity of that tree.

The complexity parameter α governs the trade-off between the tree's error rate and complexity. A large value of α results in a smaller tree while a small value of α results in a larger tree. How do we determine which nodes are insignificant? Consider a tree structure in Figure 6.7. We look at any decision node and its two leaf nodes e.g. node t_6 and its two descendants, t_8 and t_9. If the reduction in error when splitting t_6 is not significant, t_8 and t_9 are pruned, and t_6 becomes the leaf node.

Now, how do we evaluate the reduction in error? Based on the cost-complexity function, we can define the cost-complexity of a node, t as follows:

$$R_\alpha(t) = R(t) + \alpha \tag{6.10}$$

For any node t, in general, the error rate of its branch T_t is less than or equal to its node.

$$R(t) \geq R(T_t) \tag{6.11}$$

where $R(T_t) = R(t_R) + R(t_L)$. For example, consider node t_6, the sum of error rate of t_8 and t_9 is less than or equal to the error of t_6. Notice that the expression is parameterized by α. We know

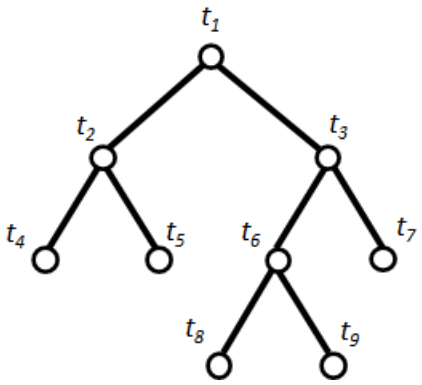


Figure 6.7 A decision tree with four decision nodes and five leaf nodes

that when α is sufficiently small, $R(t) \geq R(T_t)$ is true. Therefore, if we choose a higher α, we will be able to know which decision node is the weakest link.

We can rewrite $R(t) \geq R(T_t)$ as follows:

$$R(t) + \alpha \geq R(T) + \alpha|\tilde{T}| \tag{6.12}$$

Solving for α yields

$$\alpha \leq \frac{R(T) - R(t)}{1 - |\tilde{T}|} \tag{6.13}$$

The weakest link in a tree is

$$\alpha(t) = \frac{R(T) - R(t)}{1 - |\tilde{T}|} \tag{6.14}$$

where $R(T)$ is the error rate of tree T, $R(t)$ is the error rate of node t and $|\tilde{T}|$ is the number of leaf nodes in T.

Therefore, a decision node with the lowest α is the weakest link and its branch should be pruned. Then, the process is repeated on the pruned tree, until there is only the root node. By the end of this procedure, we have many subtrees. The optimal tree is selected among the subtrees using a validation set or cross-validation.

Numerical Example: Consider a tree structure in Figure 6.8. The training set consists of 10 instances belonging to yes and 10 instances belonging to no.

Iteration 1

Node t_1

$R(t_1) = \frac{10}{20} \cdot \frac{20}{20} = \frac{1}{2}$

$R(T_{t_1}) = 0$ (all leaf nodes are pure)

$\alpha(t_1) = \frac{\frac{1}{2} - 0}{5-1} = 0.125$

Node t_2

$R(t_2) = \frac{2}{7} \cdot \frac{7}{20} = \frac{1}{10}$

$R(T_{t_2}) = 0$ (all leaf nodes are pure)

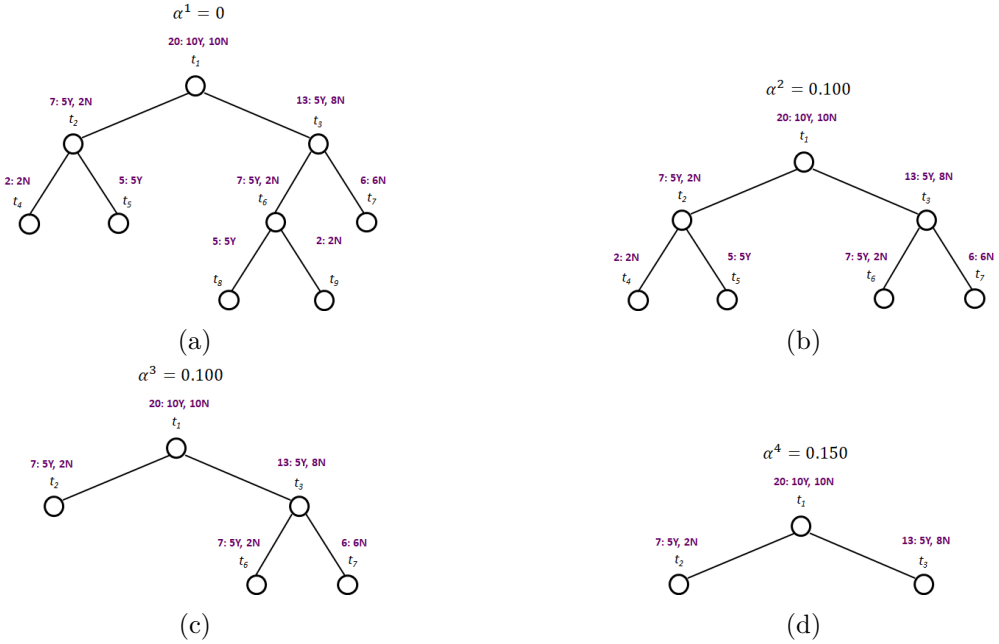


Figure 6.8 The prune tree for each iteration using post-pruning method

$\alpha(t_2) = \frac{\frac{1}{10} - 0}{2-1} = 0.100$

Node t_3

$R(t_3) = \frac{5}{13} \cdot \frac{13}{20} = \frac{1}{4}$

$R(T_{t_3}) = 0$ (all leaf nodes are pure)

$\alpha(t_3) = \frac{\frac{1}{4} - 0}{3-1} = 0.125$

Node t_6

$R(t_6) = \frac{2}{7} \cdot \frac{7}{20} = \frac{1}{10}$

$R(T_{t_6}) = 0$ (all leaf nodes are pure)

$\alpha(t_6) = \frac{\frac{1}{10} - 0}{2-1} = 0.100$

Cost-complexity measure for each node:

$\alpha(t_1) = 0.125$

$\alpha(t_2) = 0.100$

$\alpha(t_3) = 0.125$

$\alpha(t_6) = 0.100$

$\alpha(t_6) = 0.100$ is minimum, hence we prune t_8 and t_9 (t_6 has fewer instances than t_2). The subtree T_1 is given in Figure 6.8(a).

Iteration 2

Node t_1

$R(t_1) = \frac{10}{20} \cdot \frac{20}{20} = \frac{1}{2}$

$R(T_{t_1}) = \frac{2}{20}$ (all leaf nodes are pure)

$\alpha(t_1) = \frac{\frac{1}{2} - \frac{2}{20}}{4-1} = 0.133$

Node t_2

$R(t_2) = \frac{2}{7} \cdot \frac{7}{20} = \frac{1}{10}$

$R(T_{t_2}) = 0$ (all leaf nodes are pure)

$\alpha(t_2) = \frac{\frac{1}{10} - 0}{2-1} = 0.100$

Node t_3

$R(t_3) = \frac{5}{13} \cdot \frac{13}{20} = \frac{1}{4}$

$R(T_{t_3}) = \frac{2}{20}$ (all leaf nodes are pure)

$\alpha(t_3) = \frac{\frac{1}{4} - \frac{2}{20}}{2-1} = 0.150$

Cost-complexity measure for each node:

$\alpha(t_1) = 0.125$

$\alpha(t_2) = 0.100$

$\alpha(t_3) = 0.125$

$\alpha(t_2) = 0.100$ is minimum, hence we prune t_4 and t_5. The subtree T_2 is given in Figure 6.8(b).

Iteration 3

Repeat the computation for each node.

Cost-complexity measure for each node:

$\alpha(t_1) = 0.150$

$\alpha(t_3) = 0.150$

$\alpha(t_3) = 0.150$ is minimum, hence we prune t_6 and t_7 (t_3 has fewer instances than t_1). The subtree T_3 is given in Figure 6.8(c).

Iteration 4

Repeat the computation for each node.

Cost-complexity measure for each node:

$\alpha(t_1) = 0.150$

We prune t_2 and t_3 (there is only one node) as shown in Figure 6.8(d).

For each subtree, we evaluate their performance using a validation set or cross-validation. The optimal tree is the one with the highest accuracy or lowest error.

6.4 Implementing Decision Tree

In this section, we will be implementing decision tree using Scikit-Learn. We will be using the same breast cancer dataset which is available from Scikit-Learn. The prediction task is to classify the data into malignant and benign.

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
dataset = load_breast_cancer()
X_train, X_test, y_train, y_test = train_test_split(dataset.data, dataset.target,
    test_size=0.3, random_state=10)
```

We create a function to print out the accuracy score, confusion matrix and classification report.

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

def print_clf_evaluation(y_test, y_pred):
    print(accuracy_score(y_test, y_pred))
    print(confusion_matrix(y_test, y_pred))
    print(classification_report(y_test, y_pred))
```

6.4.1 Decision Tree Classifier

Decision tree classifier can be built by importing DecisionTreeClassifier module from sklearn.tree. These are some of the parameters that can be set:

- `criterion` the criteria to measure the quality of a split. The possible values are `'gini'` (default) or `'entropy'`.
- `max_depth` is the maximum depth of the tree. This parameter accepts integer value or `None`. By default, the value is `None` which means the tree will be built until all leaves contain less than `min_samples_split samples`.
- `min_samples_split` is the minimum number of samples required to split the node. This parameter accepts an integer value or a float value such as 5 or 0.1. A float value refers to the fraction of the total instances.
- `min_samples_leaf` is the minimum number of samples required to be at a leaf node. This parameter accepts an integer value or a float value such as 5 or 0.1. A float value refers to the fraction of the total instances.
- `ccp_alpha` is complexity parameter used for Cost-Complexity Pruning. By default, the value is 0 which means no pruning is performed.

First, let us use the default value for those parameters.

```
from sklearn.tree import DecisionTreeClassifier
dt_clf = DecisionTreeClassifier(random_state=0)
dt_clf.fit(X_train, y_train)
```

We evaluate the model on the test set as follows:

```
y_pred = dt_clf.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.9181286549707602
[[ 56   3]
 [ 11 101]]
              precision    recall  f1-score   support

           0       0.84      0.95      0.89        59
           1       0.97      0.90      0.94       112

    accuracy                           0.92       171
   macro avg       0.90      0.93      0.91       171
weighted avg       0.92      0.92      0.92       171
```

6.4.2 Using GridSearchCV

Now, let us fine-tune the parameters of the classifier. Notice that we have more than one parameter and we can implement a nested loop to test every combination of the parameters. Fortunately, Scikit-Learn provides a module called `GridSearchCV` which can perform exhaustive search over specified parameter values for a model. By default, the function uses accuracy for the parameter search.

First, we need to define the parameters to be passed to `GridSearchCV`. To do that, we create a grid of parameters and their range of values using python's dictionary. For example, here, we define three parameters `criterion`, `max_depth` and `min_samples_split` and their range of values are given in the list as follows:

```
param_grid = {'criterion': ['gini', 'entropy'],
              'max_depth': [3, 4, 5, 6, 7],
              'min_samples_split': [3, 5, 7, 9, 11]}
```

Then, we create an object of the classifier and pass the object together with the grid of parameters to `GridSearchCV`. To run the parameter searching, we call `fit` method and pass the training set. Conducting an exhaustive search of all the parameters is incredibly time-consuming. In this example, there are 2 values for `criterion`, 5 values for `max_depth` and `min_samples_split`. This will produce 50 different combinations of parameters. In addition to that, the module performs cross-validation to generate a better result which by default 5-fold cross-validation. This will bring the total number of jobs to 250.

```
from sklearn.model_selection import GridSearchCV
dt_clf = DecisionTreeClassifier(random_state=0)
dt_grid_search = GridSearchCV(dt_clf, param_grid)
dt_grid_search.fit(X_train, y_train)
```

To get the optimal parameters, we use attribute `best_params_` as follows:

```
print(dt_grid_search.best_params_)
```

```
{'criterion': 'gini', 'max_depth': 5, 'min_samples_split': 3}
```

The mean score of the model with the optimal parameters is

```
print(dt_grid_search.best_score_)
```

```
0.9446518987341772
```

We rebuild the decision tree classifier using the optimal parameters. Then, the classifier is evaluated on the test set as follows:

```
dt_clf = DecisionTreeClassifier(criterion='gini', max_depth=5,
                                min_samples_split=3, random_state=0)
dt_clf.fit(X_train, y_train)
y_pred = dt_clf.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.9239766081871345
[[ 53   6]
 [  7 105]]
              precision    recall  f1-score   support

           0       0.88      0.90      0.89        59
           1       0.95      0.94      0.94       112

    accuracy                           0.92       171
   macro avg       0.91      0.92      0.92       171
weighted avg       0.92      0.92      0.92       171
```

Different scoring functions can be used by specifying parameter `scoring`. Various metrics are available such as '`f`' for F1 score, '`recall`' for recall and '`precision`' for precision. The following changes the scoring function of parameter searching to F1 score.

```
GridSearchCV(dt_clf, param_grid, scoring='f1')
```

Let us see how the decision tree classifier was constructed. We can visualize the tree using text representation or plot method. Here, we display a text report of the decision tree rules. The function to produce the text report is `export_test`.

```
from sklearn.tree import export_text
print(export_text(dt_clf, feature_names=list(dataset.feature_names)))
```

```
|--- worst perimeter <= 115.35
|   |--- worst concave points <= 0.16
|   |   |--- worst perimeter <= 102.05
|   |   |   |--- area error <= 48.98
|   |   |   |   |--- worst smoothness <= 0.19
|   |   |   |   |   |--- class: 1
|   |   |   |   |--- worst smoothness >  0.19
|   |   |   |   |   |--- class: 0
|   |   |   |--- area error >  48.98
|   |   |   |   |--- worst perimeter <= 98.62
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- worst perimeter >  98.62
|   |   |   |   |   |--- class: 1
|   |   |--- worst perimeter >  102.05
|   |   |   |--- worst texture <= 29.18
|   |   |   |   |--- worst smoothness <= 0.14
|   |   |   |   |   |--- class: 1
|   |   |   |   |--- worst smoothness >  0.14
|   |   |   |   |   |--- class: 0
|   |   |   |--- worst texture >  29.18
|   |   |   |   |--- mean texture <= 24.84
|   |   |   |   |   |--- class: 0
|   |   |   |   |--- mean texture >  24.84
|   |   |   |   |   |--- class: 1
|   |--- worst concave points >  0.16
|   |   |--- class: 0
|--- worst perimeter >  115.35
|   |--- worst concavity <= 0.18
|   |   |--- class: 1
|   |--- worst concavity >  0.18
|   |   |--- perimeter error <= 1.75
|   |   |   |--- mean texture <= 17.12
|   |   |   |   |--- class: 1
|   |   |   |--- mean texture >  17.12
|   |   |   |   |--- class: 0
|   |   |--- perimeter error >  1.75
|   |   |   |--- class: 0
```

We can use `plot_tree` to plot (visualize) the decision tree. The function needs to be used with Matplotlib library. The visualization of the decision tree is given in Figure 6.9.

```
from sklearn.tree import plot_tree
from matplotlib import pyplot as plt
plt.figure(figsize=(12,12))
plot_tree(dt_clf, feature_names=list(dataset.feature_names))
plt.show()
```

6.4.3 Using Post-pruning

Now, let us implement post-pruning to build an optimal decision tree classifier. Scikit-learn provides a function called `cost_complexity_pruning_path` which returns the alphas and the corresponding total leaf impurities (error rate) at each step of the pruning process.

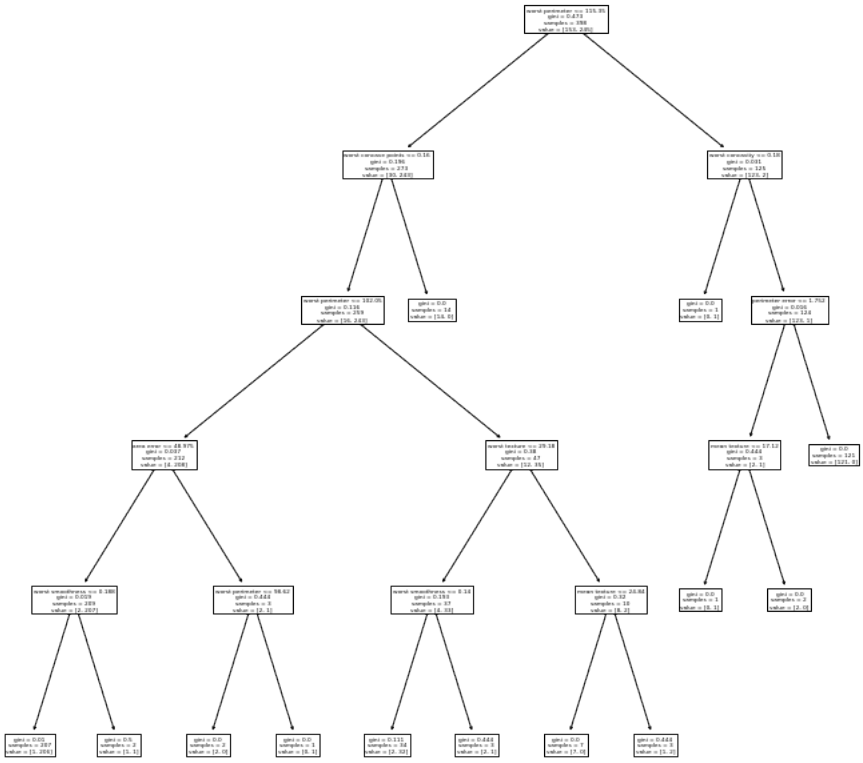


Figure 6.9 Visualizing the decision tree classifier

```
dt_clf = DecisionTreeClassifier(random_state=0)
path = dt_clf.cost_complexity_pruning_path(X_train, y_train)
ccp_alphas, impurities = path.ccp_alphas, path.impurities
```

We plot the impurities against the alphas. The maximum alpha value is removed, because it is the trivial tree with only one node. The plot is given in Figure 6.10.

```
plt.plot(ccp_alphas[:-1], impurities[:-1], marker="o", drawstyle="steps-post")
plt.xlabel("Alpha")
plt.ylabel("Total impurity of leaves")
plt.title("Total impurity vs alpha for training set")
```

Next, we implement a loop to iterate over the alphas. At each iteration, a decision tree classifier is trained using the alpha and cross-validation is performed to evaluate the classifier.

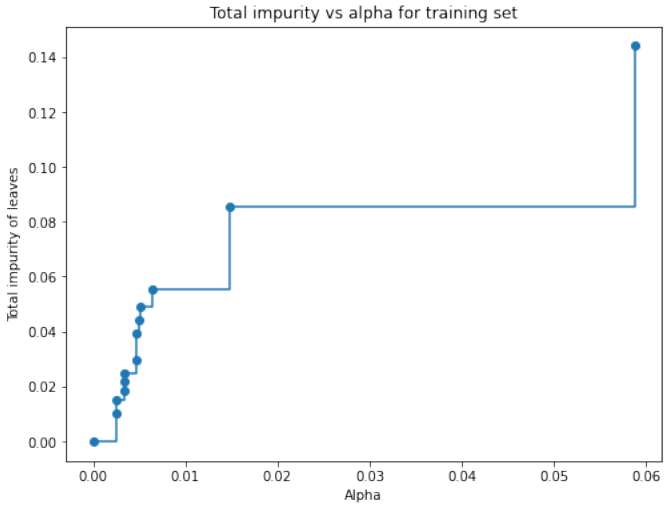


Figure 6.10 A plot of impurities (error rate) against the alphas for the training set

```
from sklearn.model_selection import cross_val_score
mean_scores = []
best_score = 0
best_alpha = 0
for ccp_alpha in ccp_alphas:
    dt_clf = DecisionTreeClassifier(random_state=0, ccp_alpha=ccp_alpha)
    score = cross_val_score(dt_clf, X_train, y_train, cv=3)
    mean_score = score.mean()
    mean_scores.append(mean_score)
    if mean_score > best_score:
        best_score = mean_score
        best_alpha = ccp_alpha
```

The best alpha is 0.0108 with a validation accuracy of 0.9246.

```
print(best_alpha)
print(best_score)
```

```
0.014873083271680028
0.9171033644717855
```

We plot the impurities of the tree against the alphas for the cross-validation. The plot is given in Figure 6.11

```
plt.plot(ccp_alphas, mean_scores, marker="o", drawstyle="steps-post")
```

We build the optimal decision tree classifier using the best alpha and evaluate it on the test set.

```
dt_clf = DecisionTreeClassifier(random_state=0, ccp_alpha=best_alpha)
dt_clf.fit(X_train, y_train)
y_pred = dt_clf.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

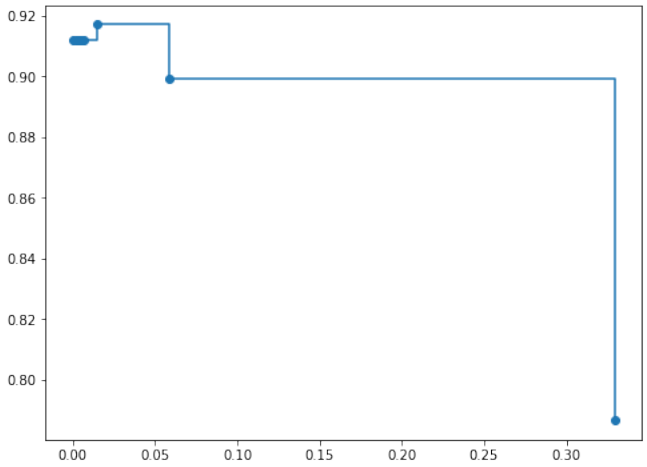


Figure 6.11 A plot of impurities (error rate) against the alphas for the cross-validation

```
0.935672514619883
[[ 52   7]
 [  4 108]]
              precision    recall  f1-score   support

           0       0.93      0.88      0.90        59
           1       0.94      0.96      0.95       112

    accuracy                           0.94       171
   macro avg       0.93      0.92      0.93       171
weighted avg       0.94      0.94      0.94       171
```

6.4.4 Decision Tree Regressor

In this section, we will be implementing decision tree for a regression problem. We will be using California Housing dataset. We load the dataset and split it into training and test sets.

```
from sklearn.datasets import fetch_california_housing
dataset = fetch_california_housing()
X_train, X_test, y_train, y_test = train_test_split(dataset.data, dataset.target,
    test_size=0.3, random_state=10)
```

Decision tree regressor can be built by importing `DecisionTreeRegressor` module. The parameters to be set are similar to `DecisionTreeClassifier` which are `criterion`, `max_depth`, `min_samples_split`, `min_samples_leaf` and `ccp_alpha`. However, for `criterion`, the possible values are `'squared_error'`, `'friedman_mse'`, `'absolute_error'` and `'poisson'`. The default criterion is `'squared_error'`.

Let us build a decision tree regressor with `'friedman_mse'`criteria and `max_depth` of 4.

```
from sklearn.tree import DecisionTreeRegressor
dt_reg = DecisionTreeRegressor(criterion='friedman_mse', max_depth=4)
```

We create a function to calculate the regression loss as follows:

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
def print_reg_evaluation(y_test, y_pred):
    print('Mean Absolute Error: ', mean_absolute_error(y_test, y_pred))
    print('Mean Squared Error: ', mean_squared_error(y_test, y_pred))
    print('R-squared: ', r2_score(y_test, y_pred))
```

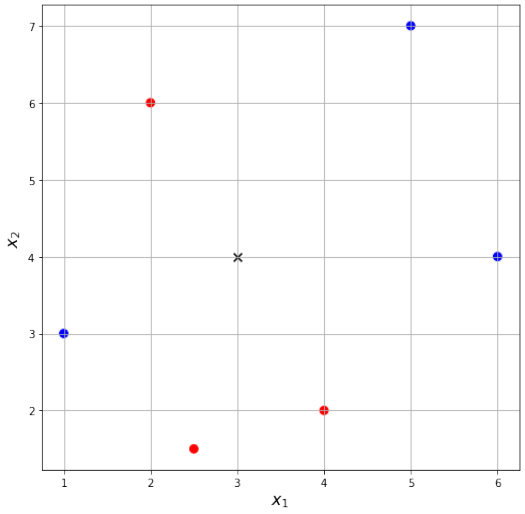
We evaluate the regressor on the test set.

```
y_pred = dt_reg.predict(X_test)
print_reg_evaluation(y_test, y_pred)
```

```
Mean Absolute Error:  0.5521514432172915
Mean Squared Error:  0.5692648479340473
R-squared:  0.5800771214569134
```

Exercises

1. Explain the properties of Gini and entropy impurity.
2. Describe which should be preferred among Gini impurity and entropy impurity.
3. Explain the mathematical formulation of information gain.
4. Explain parameters that can be used in pre-pruning to avoid overfitting.
5. Given a set of data points as shown in the following figure. Assuming a decision tree is built to classify the dataset into red and blue classes.



   (a) Calculate the information gain if the dataset is split at 4.5 along dimension x_1.

   (b) Assuming the decision tree consists of one decision node only which is the split at 4.5 along dimension x_1, state the probability of data point x is predicted as class red.

Chapter 7

Support Vector Machine

7.1 Introduction

Perceptron seeks to find a hyperplane that can linearly separate the data points into their classes. However, perceptron does not guarantee the hyperplane is the optimal solution. Figure 7.1 illustrates three possible solutions among an almost infinite number of possible hyperplanes. Although the hyperplanes perfectly separate the data points, misclassification could easily occur. For example, it is very likely for test points belonging to negative class to be above hyperplane h_3. Similarly, test points belonging to positive class could appear below hyperplane h_1. As for h_2, although the hyperplane appears to be a "better" hyperplane, the decision boundary is closer to positive class. Thus, the misclassification rate would increase especially for positive class.

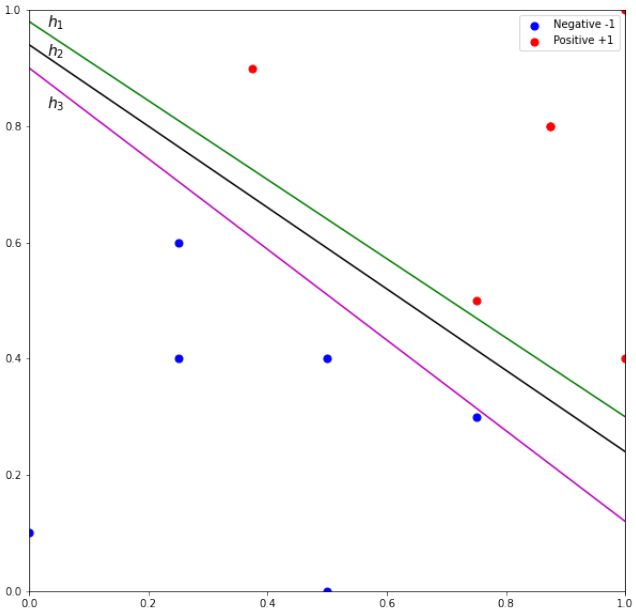


Figure 7.1 Three possible hyperplanes can linearly separate the data points perfectly

Support vector machine (SVM) finds an optimal hyperplane that best separates the data points. This optimal solution is the hyperplane that is far away from the data points of both classes. To this end, SVM introduces the notion of margin which is illustrated in Figure 7.2. The margin is defined by the distance from the parallel dotted lines that are crossing the closest data points of

both classes. The hyperplane is in the middle indicated by the blue line. The optimal hyperplane is the one that maximizes the margin of the training instances. The idea is if the margin is maximized, the hyperplane will be far away from the data points of both sides and reduce the chance of misclassification.

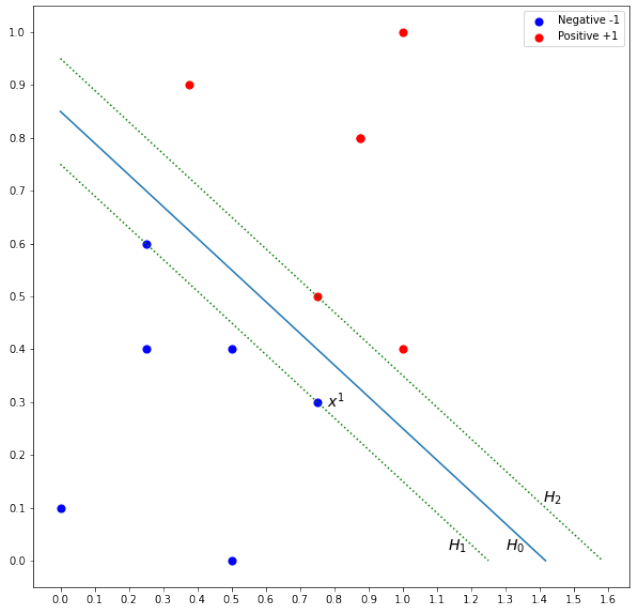


Figure 7.2 An optimal hyperplane that separates the two classes.

How do we find the margin? Given hyperplane H_0 that separates the two classes, we know that H_0 is equidistant from hyperplane H_1 and H_2. H_0 is defined by

$$\mathbf{w}^\top \mathbf{x} + b = 0 \tag{7.1}$$

where $\mathbf{x}$ is the feature vector and $\mathbf{w}$ and b are the parameters of the hyperplane. For each instance x^i, we know that

$$\mathbf{w}^\top \mathbf{x}^i + b \geq +1 \quad \text{for all} \quad \mathbf{x}^i \quad \text{belong to class} \quad +1 \tag{7.2}$$

$$\mathbf{w}^\top \mathbf{x}^i + b \leq -1 \quad \text{for all} \quad \mathbf{x}^i \quad \text{belong to class} \quad -1 \tag{7.3}$$

These are the constraints that need to be respected to select the optimal hyperplane. The first constraint states that all instances belonging to class $+1$ must be above H_2 and the second constraint states that all instances belonging to class -1 must be below H_1. By not violating these constraints, we are selecting two hyperplanes (that define the margin) with no data points in between them. Let's simplify the constraints. We multiply the first constraint on both sides with y^i.

$$y^i(\mathbf{w}^\top \mathbf{x}^i + b) \geq y^i(+1) \tag{7.4}$$

We know that for all x^i belonging to class $+1$, y^i is $+1$. Thus, the constraint becomes

$$y^i(\mathbf{w}^\top \mathbf{x}^i + b) \geq +1 \tag{7.5}$$

We do the same for the second constraint.

$$y^i(\mathbf{w}^\top \mathbf{x}^i + b) \leq y^i(-1) \tag{7.6}$$

We know that for all x^i belonging to class -1, y^i is -1. Thus, the constraint becomes

$$y^i(\mathbf{w}^\top \mathbf{x}^i + b) \geq +1 \tag{7.7}$$

Therefore, we can combine the two constraints as follows:

$$y^i(\mathbf{w}^\top \mathbf{x}^i + b) \geq +1 \texttt{ for all } \mathbf{x}^i \tag{7.8}$$

m is the margin as shown in Figure 7.3. We know that $\mathbf{w}$ is a vector that is perpendicular to the hyperplanes. If the unit vector of $\mathbf{w}$ is multiplied with m, we get a vector $\mathbf{v}$, where it has the same direction as $\mathbf{w}$ and with a distance equal to m. Let the unit vector be $\mathbf{u} = \frac{\mathbf{w}}{\|w\|}$, thus $\mathbf{v} = m\mathbf{u}$.

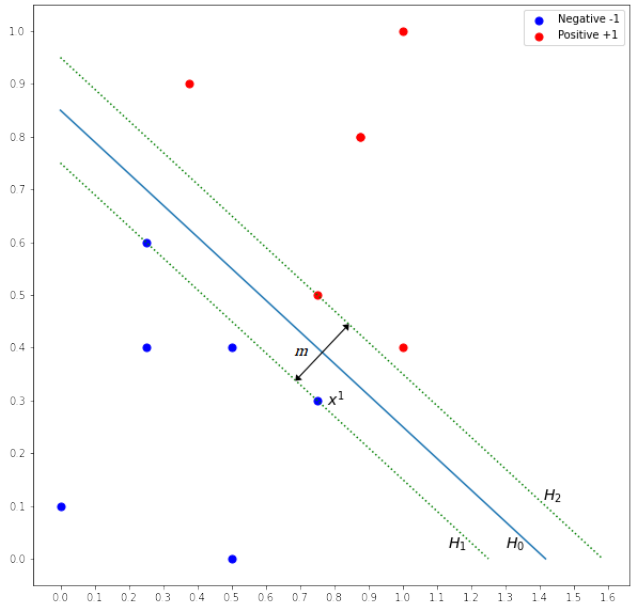


Figure 7.3 The margin of the two hyperplanes is denoted by m

We know that H_1 and H_2 are

$$H_1 : \mathbf{w}^\top \mathbf{x} + b = -1 \tag{7.9}$$

$$H_2 : \mathbf{w}^\top \mathbf{x} + b = +1 \tag{7.10}$$

Let $\mathbf{x}^1$ is a data point on H_1. If we add $\mathbf{x}^1$ with $\mathbf{v}$, the resultant is a point on H_2.

$$\mathbf{x}^2 = \mathbf{x}^1 + \mathbf{v} \tag{7.11}$$

We can write H_2 as follows:

$$\mathbf{w}^\top(\mathbf{x}^1 + \mathbf{v}) + b = 1 \tag{7.12}$$

$$\mathbf{w}^\top(\mathbf{x}^1 + m\frac{\mathbf{w}}{\|w\|}) + b = 1 \tag{7.13}$$

$$\mathbf{w}^\top\mathbf{x}^1 + b + m\frac{\|w\|^2}{\|w\|} = 1 \tag{7.14}$$

$$\mathbf{w}^\top\mathbf{x}^1 + b + m\|w\| = 1 \tag{7.15}$$

We know that $\mathbf{w}^\top\mathbf{x}^1 + b = -1$. We can rewrite the equation and solve for m.

$$m\|w\| = 2 \tag{7.16}$$

$$m = \frac{2}{\|w\|} \tag{7.17}$$

Therefore, margin m is maximized when $\|w\|$ is minimized. For mathematical convenience, we change from minimizing $\|w\|$ to minimizing $\frac{1}{2}\|w\|^2$.

$$\min \frac{1}{2}\|w\|^2 \tag{7.18}$$

subject to $y^i(\mathbf{w}^\top\mathbf{x}^i + b) \geq 1$ for $i = 1, 2, ..., N$.

You may skip the derivation of the solution and go to parameter solving.

We can solve the constrained optimization problem using quadratic programming (Nocedal and Wright 1999). Before that, we restate the optimization problem by applying Lagrange multiplier. Lagrange multiplier states that to find the maximum or minimum of a function $f(x)$ subjected to an equality constraint $g(x)$, form the following expression L, set the gradient of L to zero and find the Lagrange multiplier (parameter) λ.

$$L(x) = f(x) - \lambda g(x) \tag{7.19}$$

Substitute $f(x)$ and $g(x)$.

$$L_p = \frac{1}{2}\|w\|^2 - \lambda^i(y^i(\mathbf{w}^\top\mathbf{x}^i + b) - 1) \texttt{ for all } i \tag{7.20}$$

$$L_p = \frac{1}{2}\|w\|^2 - \sum_{i=1}^{N}\lambda^i y^i(\mathbf{w}^\top\mathbf{x}^i + b) + \sum_{i=1}^{N}\lambda^i \tag{7.21}$$

Differentiate L_p with respect to w and b, and set it to zero, $\frac{\partial L_p}{\partial w} = 0$ and $\frac{\partial L_p}{\partial b} = 0$.

$$w = \sum_{i=1}^{N} \lambda^i y^i \mathbf{x}^i \tag{7.22}$$

$$\sum_{i=1}^{N} \lambda^i y^i = 0 \tag{7.23}$$

Substitute w and $\sum_{i=1}^{N} \lambda^i y^i = 0$ into L_p and simplify it.

$$L_d = \frac{1}{2}(\sum_{i=1}^{N} \lambda^i y^i \mathbf{x}^i)^2 - \sum_{i=1}^{N}\sum_{j=1}^{N} \lambda^i \lambda^j y^i y^j (\mathbf{x}^i)^\top \mathbf{x}^j + \sum_{i=1}^{N} \lambda^i \tag{7.24}$$

$$L_d = \sum_{i=1}^{N} \lambda^i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \lambda^i \lambda^j y^i y^j (\mathbf{x}^i)^\top \mathbf{x}^j \tag{7.25}$$

Thus, the optimization problem becomes

$$\max_{\lambda}[\sum_{i=1}^{N} \lambda^i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \lambda^i \lambda^j y^i y^j (\mathbf{x}^i)^\top \mathbf{x}^j] \tag{7.26}$$

subject to $\lambda^i \geq 0$ and $\sum_{i=1}^{N} \lambda^i y^i = 0$.

As we can see above, the objective function now depends only on the Lagrange multipliers which is easier to be solved. Upon solving the optimization problem, most Lagrange multipliers ($\lambda^i = 0$) will be zero, and only a few of them are non-zero. The data points with $\lambda > 0$ are called the support vectors (Cortes and V. Vapnik 1995). These support vectors lie on the edge of the margin which defines the margin and the optimal hyperplane. From the above, we see the solution for $\mathbf{w}$ has the form

$$\mathbf{w} = \sum_{i=1}^{N} \lambda^i y^i \mathbf{x}^i \tag{7.27}$$

We can use any data point on the edge of the margin to solve b. However, typically we use an average of all the solutions for numerical stability.

$$b = \frac{1}{|S|}\sum_{s \in S} y^s - \mathbf{w}^\top \mathbf{x}^s \tag{7.28}$$

where S is the set of support vector indices. Given a test point $\hat{\mathbf{x}}$, the predicted value is

$$\hat{f}(\mathbf{x}) = \text{sign}(\mathbf{w}^\top \hat{\mathbf{x}} + b) \tag{7.29}$$

7.2 Soft Margin

Real-world data are typically noisy and not linearly separable. When the data are not separable, the algorithm above will not be able to produce the solution to the problem. To handle non-separable data, we relax the constraints by allowing data points within the margin. Consider the margin

in Figure 7.4. Three data points are allowed to be within the margin. These "errors" need to be quantified as indicated by ξ^i and included in the constraints. The variable ξ^i is known as slack variable. ξ^i represents the amount of which the data points on the wrong side of the margin. If $\xi^i = 0$, the data point is on the right side. If $0 \leq \xi \leq 1$, the data point is in the margin. If $\xi > 1$, the data point is on the wrong side of the hyperplane.

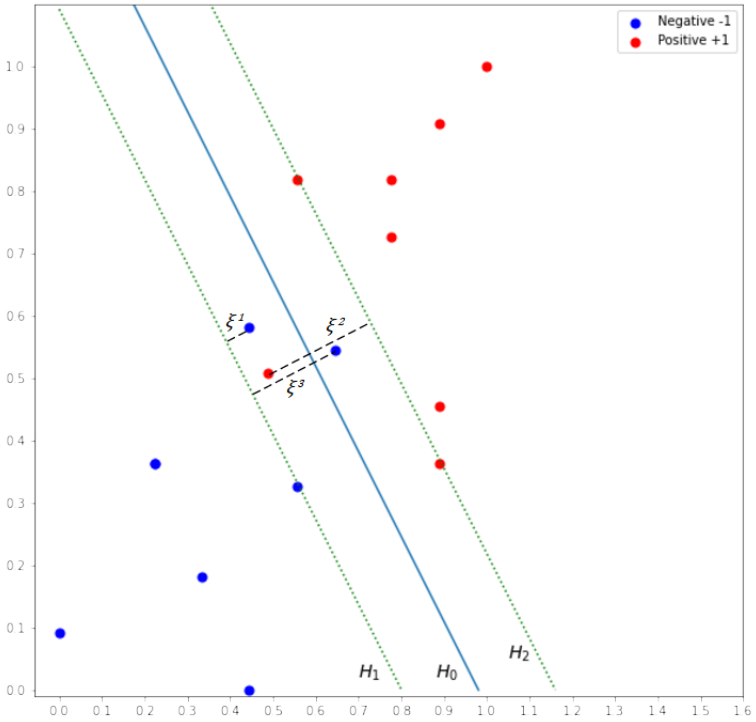


Figure 7.4 Data points in the margin are errors that are included in the constraints

The combined constraint is now defined as follows:

$$y^i(\mathbf{w}^\top \mathbf{x}^i + b) \geq 1 - \xi^i \tag{7.30}$$

where $\xi^i \geq 0$ and $i = 1, 2, ..., N$. The equation (constraint) implies that we define an error if the prediction is on the wrong side of the margin boundary. This is called the hinge loss. Given the prediction $\hat{y} = (\mathbf{w}^\top \mathbf{x} + b)$, the loss function is defined as

$$J_{hinge}(y, \hat{y}) = \max(0, 1 - y\hat{y}) \tag{7.31}$$

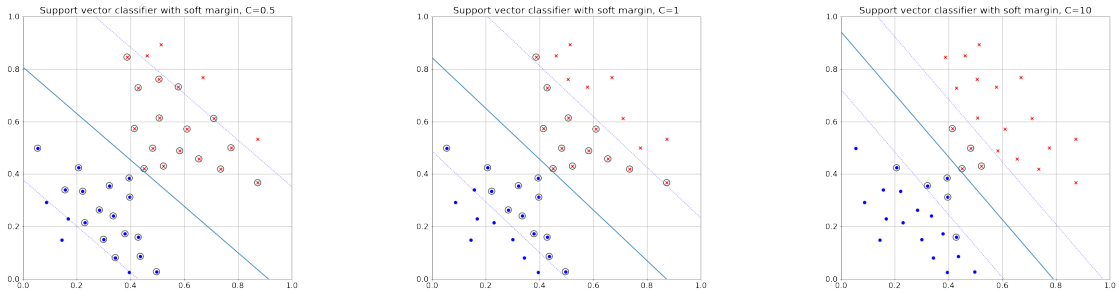
By introducing the slack variables in the optimization problem, it is still possible to satisfy the constraint, albeit with some data points within or on the wrong side of the hyperplane. We add a term to the objective function which represents the total amount of which the data points on the wrong side of their margin.

$$\min \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N} \xi^i \tag{7.32}$$

subject to $y^i(\mathbf{w}^\top \mathbf{x}^i + b) - 1 + \xi \geq 0$ and $\xi^i \geq 0$ for $i = 1, 2, ..., N$

where C is the regularization parameter to control the trade-off between maximizing the margin and minimizing the error. Figure 7.5 shows the effect of C on the hyperplanes. A small C gives

less importance to the error term, hence more mistakes are allowed, resulting in a large margin as shown in Figure 7.5(a). Conversely, a larger C allows fewer mistakes resulting in a smaller margin as shown in Figures 7.5(b) and 7.5(c)



(a) (b) (c)

Figure 7.5 The effect of C on the hyperplanes

You may skip the derivation of the solution and go to parameter solving.

Using Lagrange multiplier to formulate the optimization problem with respect to the Lagrange parameters:

$$L_p = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\xi^i - \sum_{i=1}^{N}\lambda^i(y^i(\mathbf{w}^\top\mathbf{x}^i + b) - 1 + \xi^i) - \sum_{i=1}^{N}\mu^i\xi^i \tag{7.33}$$

where μ^i is the Lagrange multiplier for constraint $\xi^i \geq 0$. Differentiate L_p with respect to w, b and ξ, and set it to zero, $\frac{\partial L_p}{\partial w} = 0$, $\frac{\partial L_p}{\partial b} = 0$ and $\frac{\partial L_p}{\partial \xi} = 0$.

$$w = \sum_{i=1}^{N}\lambda^i y^i \mathbf{x}^i \tag{7.34}$$

$$\sum_{i=1}^{N}\lambda^i y^i = 0 \tag{7.35}$$

$$C - \lambda^i - \mu^i = 0 \tag{7.36}$$

Since $\mu^i \geq 0$, this implies $0 \leq \lambda^i \leq C$. Substitute the derivatives into L_p and simplify it.

$$L_d = \sum_{i=1}^{N}\lambda^i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}\lambda^i\lambda^j y^i y^j (\mathbf{x}^i)^\top \mathbf{x}^j \tag{7.37}$$

subject to $\sum_{i=1}^{N}\lambda^i y^i = 0$ and $0 \leq \lambda \leq C$.

Upon solving the optimization problem using quadratic programming, data points that lie on the correct side of the hyperplane, have their corresponding $\lambda^i = 0$, and the others have their $\lambda^i > 0$. Of these data points, the ones that lie on the edge of the margin have their $\lambda^i < C$, and the rest lie within the margin with $\lambda^i = C$.

$\mathbf{w}$ is computed as follows:

$$\mathbf{w} = \sum_{i=1}^{N} \lambda^i y^i \mathbf{x}^i \tag{7.38}$$

As before, we can use any data point on the edge of the margin to solve b. However, typically we use an average of all the solutions for numerical stability.

$$b = \frac{1}{|S|} \sum_{s \in S} y^s - \mathbf{w}^\top \mathbf{x}^s \tag{7.39}$$

where S is the set of support vector indices. Given a test point $\hat{\mathbf{x}}$, the predicted value is

$$\hat{f}(\mathbf{x}) = \text{sign}(\mathbf{w}^\top \hat{\mathbf{x}} + b) \tag{7.40}$$

Numerical Example: Consider the following dataset. Suppose $C = 30$, a soft margin SVM is built by solving the optimization problem using quadratic programming. The Lagrange multipliers λ are given as follows. Calculate parameter $\mathbf{w}$ and b.

$$\mathbf{x} = \begin{bmatrix} 0.444 & 0.446 \\ 0.276 & 0.508 \\ 0.295 & 0.286 \\ 0.116 & 0.302 \\ 0.208 & 0.080 \\ 0.488 & 0.729 \\ 0.468 & 0.515 \\ 0.634 & 0.631 \\ 0.785 & 0.821 \\ 0.612 & 0.490 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} -1 \\ -1 \\ -1 \\ -1 \\ -1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \lambda = \begin{bmatrix} 30.000 \\ 18.359 \\ 0.000 \\ 0.000 \\ 0.000 \\ 6.138 \\ 30.000 \\ 0.000 \\ 0.000 \\ 12.211 \end{bmatrix}$$

$\mathbf{w} = \sum_{i=1}^{10} \lambda^i y^i \mathbf{x}^i = \begin{bmatrix} 6.121 & 3.201 \end{bmatrix}$

$b = \frac{1}{|S|} \sum_{s \in S} y^s - \mathbf{w}^\top \mathbf{x}^s = -4.316$

where $S = 1, 2, 6, 7, 10$

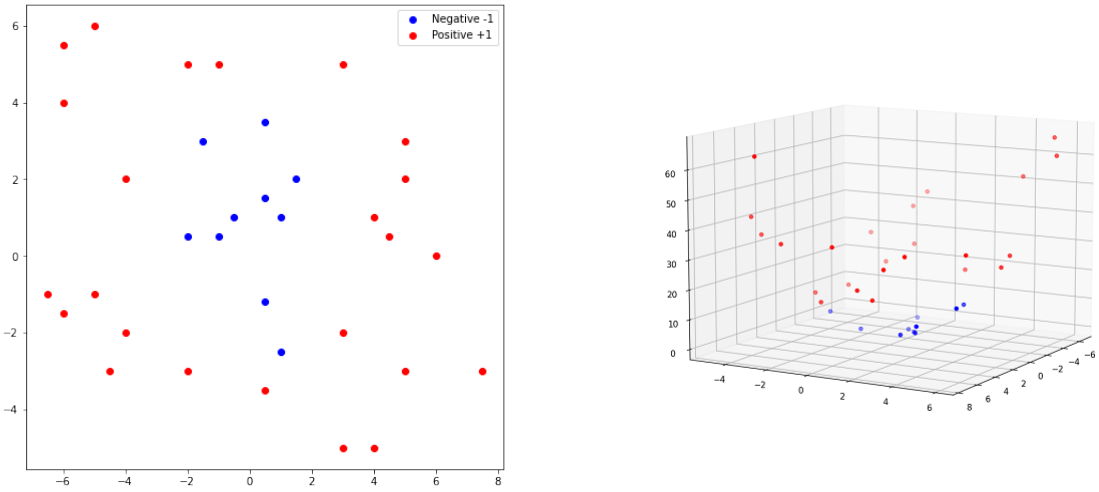
Given a test point $\hat{\mathbf{x}} = (0.515, 0.627)$, predict the test point using the SVM model.

$$\begin{aligned} \hat{y} &= \text{sign}(\mathbf{w}^\top \hat{\mathbf{x}} + b) \\ &= \text{sign}(0.836) \\ &= +1 \end{aligned}$$

7.3 Non-linear SVM with Kernel Trick

The algorithm we described above finds the linear model that divides the feature space. However, some real-world data is non-linear and a linear model will not be able to effectively classify the data as shown in Figure 7.6(a). As shown in the figure, no matter how we try to fit the linear model, the decision boundary will give a bad classification result. One way to deal with this kind of data is to extend the feature dimension by performing data transformation using basis functions such as polynomial and spline (Boser, I. M. Guyon, and V. N. Vapnik 1992). An example of non-linear transformation is given in Figure 7.6(b) whereby the polynomial basis function is used to map the data points from 2-dimension to 3-dimension. After the transformation, we can easily fit a linear

model to separate the data points.



(a) (b)

Figure 7.6 Non-linear data and the mapping of the data to a new feature space

We select a basis function, $\phi(\mathbf{x}^i)$ to transform each instance $\mathbf{x}^i$ from d-dimension to m-dimension where m is generally much larger than d. Then, the transformed dataset $\phi(\mathbf{x}^i) = (\phi_1(\mathbf{x}^i), \phi_2(\mathbf{x}^i), ...\phi_m(\mathbf{x}^i))$ is used to fit the linear model $\hat{f}(\phi(\mathbf{x}^i)) = \mathbf{w}^\top \phi(\mathbf{x}^i) + b$. The optimization problem of finding the optimal hyperplane has the same form except the constraints are defined in the new feature space.

$$\min \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N} \xi^i \tag{7.41}$$

subject to $y^i(\mathbf{w}^\top \phi(\mathbf{x}^i) + b) - 1 + \xi \geq 0$ and $\xi^i \geq 0$ for $i = 1, 2, ..., N$

Reformulate the problem using Lagrange multiplier:

$$L_d = \sum_{i=1}^{N} \lambda^i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \lambda^i \lambda^j y^i y^j \phi(\mathbf{x}^i)^\top \phi(\mathbf{x}^j) \tag{7.42}$$

subject to $\sum_{i=1}^{N} \lambda^i y^i = 0$ and $0 \leq \lambda^i \leq C$.

Although the mapping of the features to a high dimensional feature space allows the data to be linearly separable, computing the transformation is expensive especially when the dimension is very high. This is because we have to apply the transformation on each sample followed by the inner product. The idea of the kernel trick is that the inner product between the transformed data is not needed. Instead, we require only the kernel function that allows us to operate in the high-dimensional space without computing the data transformation. Therefore, we just need to apply the kernel function, k directly in the original feature space to achieve the same effect.

$$k(\mathbf{x}^i, \mathbf{x}^j) = \phi(\mathbf{x}^i)^\top \phi(\mathbf{x}^j) \tag{7.43}$$

Therefore, L_d is

$$L_d = \sum_{i=1}^{N} \lambda^i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \lambda^i \lambda^j y^i y^j k(\mathbf{x}^i, \mathbf{x}^j) \tag{7.44}$$

subject to $\sum_{i=1}^{N} \lambda^i y^i = 0$ and $0 \leq \lambda^i \leq C$. Given the set of support vector indices S, b can be solved as follows:

$$b^s = y^s - \sum_{m \in S} \lambda^m y^m k(\mathbf{x}^m, \mathbf{x}^s) \tag{7.45}$$

We use an average of all the solutions for numerical stability. Given a test point $\hat{\mathbf{x}}$, the predicted value is calculated as follows:

$$\hat{f}(\mathbf{x}) = \text{sign}(\sum_{s \in S} \lambda^s y^s k(\hat{\mathbf{x}}, \mathbf{x}^s) + b) \tag{7.46}$$

7.4 Kernels

We discuss some of the commonly used kernel functions in the literature. Polynomial kernel function is defined as follows:

$$k(\mathbf{x}, \mathbf{z}) = (1 + \mathbf{x}^\top \mathbf{z})^d \tag{7.47}$$

where d is the degree parameter. Consider a dataset with two input features x_1 and x_2 and a polynomial kernel with a degree of 2.

$$\begin{aligned} k(\mathbf{x}, \mathbf{z}) &= (1 + \mathbf{x}^\top \mathbf{z})^2 \\ &= (1 + x_1 z_1 + x_2 z_2)^2 \\ &= (1 + 2x_1 z_1 + 2x_2 z_2 + 2x_1 x_2 z_1 z_2 + x_1^2 z_1^2 + x_2^2 z_2^2) \end{aligned} \tag{7.48}$$

We can see from the above that the dimension of the dataset is extended from 2 to 6.

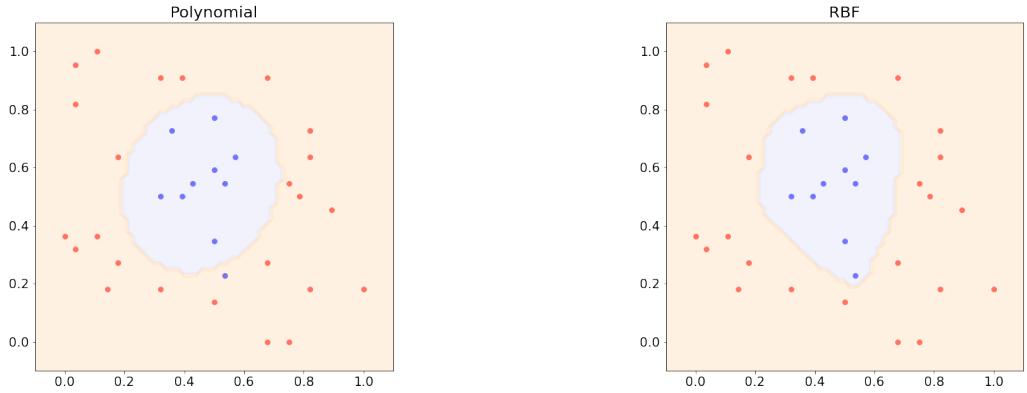
$$\phi(\mathbf{x}) = \begin{bmatrix} 1 \\ \sqrt{2}x_1 \\ \sqrt{2}x_2 \\ 2x_1 x_2 \\ x_1^2 \\ x_2^2 \end{bmatrix} \tag{7.49}$$

Radial basis function (RBF) kernel function is similar to the pdf of the normal distribution. The kernel function is defined as follows:

$$k(\mathbf{x}, \mathbf{z}) = e^{-\gamma \|\mathbf{x} - \mathbf{z}\|^2} \tag{7.50}$$

where $\|\mathbf{x} - \mathbf{z}\|^2$ is the Euclidean distance between the two data points. Figure 7.7 shows the decision boundaries of SVM with polynomial and RBF kernels. As shown in the figure, the data points are not linearly separable. However, with the kernels, SVM is able to classify the data points into their classes.

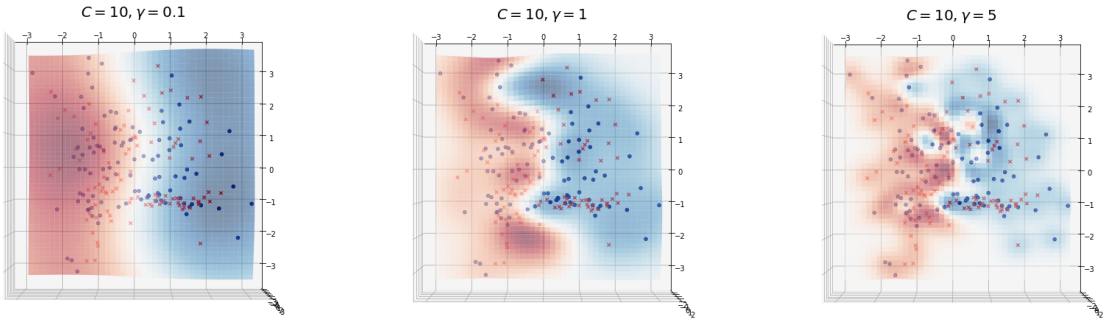
RBF kernel has a parameter γ that defines the spread of the kernel or the decision region of the



(a) (b)

Figure 7.7 Decision boundaries of non-linear SVM with (a) polynomial and (b) RBF kernels

support vectors. In other words, it determines how far the influence of a support vector in the prediction. A small value of γ means a large decision region area while a large value of γ means small decision region area. The effect of γ is illustrated in Figure 7.8. As can be seen in the figure, when γ is small, the model fails to capture the complexity of the data and the model behaves similarly to a linear model, separating the data points on the right from the left. When γ is set to a larger value, the model becomes more non-linear and able to learn the complexity of the data as shown in Figure 7.8(b) and Figure 7.8(c).



(a) (b) (c)

Figure 7.8 The effect of gamma on the decision boundaries

Numerical Example: Consider the following dataset. Suppose a non-linear SVM with RBF kernel and $\gamma = 0.8$ is built by solving the optimization problem using quadratic programming. The Lagrange multipliers λ are given as follows:

$$\mathbf{x} = \begin{bmatrix} 0.481 & 0.326 \\ 0.75 & 0.282 \\ 0.771 & 0.484 \\ 0.856 & 0.745 \\ 0.818 & 0.211 \\ 0.331 & 0.242 \end{bmatrix} \quad \mathbf{y} = \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \lambda = \begin{bmatrix} 47.777 \\ 37.501 \\ 50.000 \\ 35.278 \\ 50.000 \\ 50.000 \end{bmatrix}$$

All data points are support vectors since all $\lambda^i > 0$. Thus, support vector indices $S = 1, 2, 3, 4, 5, 6$. Thus, b can be solved as follows:

$b^s = y^s - \sum_{m \in S} \lambda^m y^m k(\mathbf{x}^m, \mathbf{x}^s)$

$b^1 = 11.643$, $b^2 = 11.370$, $b^3 = 15.719$, $b^4 = 1.449$, $b^5 = 4.832$, $b^6 = -0.683$

$b = \frac{1}{|S|}\sum_{i=1}^{|S|} b^i = 7.388$

Given a test point $\hat{\mathbf{x}} = (0.614, 0.258)$, predict the test point using the SVM model.

$$\begin{aligned}\hat{f}(\hat{\mathbf{x}}) &= \text{sign}(\sum_{s\in S} \lambda^s y^s k(\hat{\mathbf{x}}, \mathbf{x}^s) + b) \\ &= \text{sign}(-3.717) \\ &= -1\end{aligned}$$

7.5 Support Vector Regression

Support vector machine can be generalized for regression problems (Drucker et al. 1996). Similar to other regression models, support vector regression aims to fit the best line or hyperplane through the data points. However, in support vector regression, the best-fitted hyperplane is not a function that minimizes the error between the real and predicted value. Instead, support vector regression finds a hyperplane such that the margin has the most data points within it while fitting the error to be below a certain threshold ϵ. In other words, the best-fitted line is the one that has at most ϵ deviation from the target of the training set.

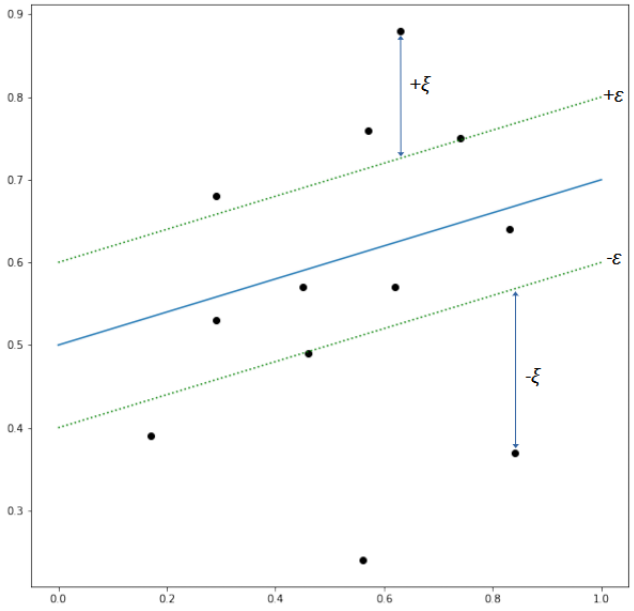


Figure 7.9 The prameters for the support vector regression

Figure 7.9 illustrates a support vector regression. The fitted line is indicated by the blue line and the boundary lines of the margin (also known as ϵ-tube) are indicated by the green dash lines. The error of the data points within the tube is tolerated (zero) while for data points that are outside the tube, the loss is the distance between the data points and the boundary lines. Thus, the loss function is defined as follows:

$$L = \begin{cases} 0, & \texttt{if} |y^i - \mathbf{w}^\top \mathbf{x}^i + b| < \epsilon \\ |y^i - \mathbf{w}^\top \mathbf{x}^i + b| - \epsilon, & \texttt{otherwise} \end{cases} \tag{7.51}$$

Similar to the soft margin hyperplane in support vector classification, the slack variables are

introduced to represent the error of data points that are out of the ϵ-tube. Thus, the optimization problem is given as follows:

$$\min \frac{1}{2}\|w\|^2 \tag{7.52}$$

$$\text{subject to } \begin{cases} |y^i - \mathbf{w}^\top \mathbf{x}^i + b| \leq \epsilon + \xi_+^i \\ |\mathbf{w}^\top \mathbf{x}^i + b - y^i| \leq \epsilon + \xi_-^i \\ \xi_+^i, \xi_-^i \geq 0 \end{cases} \quad \text{for } i = 1, 2, \ldots, N$$

Notice that there are two types of slack variables, for positive and negative errors to keep them positive. We restate the optimization problem by applying Lagrange multiplier:

$$\begin{aligned} L_p = & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}(\xi_+^i + \xi_-^i) \\ & - \sum_{i=1}^{N}\lambda_+^i(\epsilon + \xi_+^i - y^i + (\mathbf{w}^\top \mathbf{x}^i + b)) \\ & - \sum_{i=1}^{N}\lambda_-^i(\epsilon + \xi_-^i + y^i - (\mathbf{w}^\top \mathbf{x}^i + b)) \\ & - \sum_{i=1}^{N}\mu_+^i\xi_+^i - \sum_{i=1}^{N}\mu_-^i\xi_-^i \end{aligned} \tag{7.53}$$

$$\begin{aligned} L_p = & \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}(\xi_+^i + \xi_-^i) \\ & - \sum_{i=1}^{N}\lambda_+^i\epsilon - \sum_{i=1}^{N}\lambda_+^i\xi_+^i + \sum_{i=1}^{N}\lambda_+^i y^i - \sum_{i=1}^{N}\lambda_+^i(\mathbf{w}^\top \mathbf{x}^i + b) \\ & - \sum_{i=1}^{N}\lambda_-^i\epsilon - \sum_{i=1}^{N}\lambda_-^i\xi_+^i - \sum_{i=1}^{N}\lambda_-^i y^i + \sum_{i=1}^{N}\lambda_-^i(\mathbf{w}^\top \mathbf{x}^i + b) \\ & - \sum_{i=1}^{N}\mu_+^i\xi_+^i - \sum_{i=1}^{N}\mu_-^i\xi_-^i \end{aligned} \tag{7.54}$$

Differentiate L_p with respect to w, b and ξ, and set it to zero, $\frac{\partial L_p}{\partial w} = 0$, $\frac{\partial L_p}{\partial b} = 0$ and $\frac{\partial L_p}{\partial \xi_+} = 0$.

$$w = \sum_{i=1}^{N}(\lambda_+^i - \lambda_-^i)\mathbf{x}^i \tag{7.55}$$

$$\sum_{i=1}^{N}(\lambda_-^i - \lambda_+^i) = 0 \tag{7.56}$$

$$C - \sum_{i=1}^{N}(\lambda_+^i + \mu_+^i) = 0 \tag{7.57}$$

Substitute the derivatives into L_p and simplify it as follows:

$$\begin{aligned} L_p =& \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}(\mathbf{x}^i)^\top \mathbf{x}^j(\lambda_+^i - \lambda_-^i)(\lambda_+^j - \lambda_-^j) \\ &+ \sum_{i=1}^{N}(\lambda_+^i + \mu_+^i)\sum_{i=1}^{N}(\xi_+^i + \xi_-^i) \\ &+ \sum_{i=1}^{N} y^i(\lambda_+^i - \lambda_-^i) - \sum_{i=1}^{N}\epsilon(\lambda_+^i + \lambda_-^i) \\ &- \sum_{i=1}^{N}\lambda_+^i\xi_+^i - \sum_{i=1}^{N}\lambda_-^i\xi_-^i - \sum_{i=1}^{N}\mu_+^i\xi_+^i - \sum_{i=1}^{N}\mu_-^i\xi_-^i \end{aligned} \tag{7.58}$$

$$\begin{aligned} L_p =& \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}(\mathbf{x}^i)^\top \mathbf{x}^j(\lambda_+^i - \lambda_-^i)(\lambda_+^j - \lambda_-^j) \\ &+ \sum_{i=1}^{N} y^i(\lambda_+^i - \lambda_-^i) - \sum_{i=1}^{N}\epsilon(\lambda_+^i + \lambda_-^i) \\ &+ \sum_{i=1}^{N}\lambda_+^i\xi_+^i + \sum_{i=1}^{N}\lambda_+^i\xi_-^i + \sum_{i=1}^{N}\mu_+^i\xi_+^i + \sum_{i=1}^{N}\mu_+^i\xi_-^i \\ &- \sum_{i=1}^{N}\lambda_+^i\xi_+^i - \sum_{i=1}^{N}\lambda_-^i\xi_-^i - \sum_{i=1}^{N}\mu_+^i\xi_+^i - \sum_{i=1}^{N}\mu_-^i\xi_-^i \end{aligned} \tag{7.59}$$

$$\begin{aligned} L_p =& \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}(\mathbf{x}^i)^\top \mathbf{x}^j(\lambda_+^i - \lambda_-^i)(\lambda_+^j - \lambda_-^j) \\ &+ \sum_{i=1}^{N} y^i(\lambda_+^i - \lambda_-^i) - \sum_{i=1}^{N}\epsilon(\lambda_+^i + \lambda_-^i) \end{aligned} \tag{7.60}$$

$$\text{subject to } \begin{cases} \sum_{i=1}^{N}(\lambda_-^i - \lambda_+^i) = 0 \\ 0 \le \lambda_+^i \le C \\ 0 \le \lambda_-^i \le C \end{cases}.$$

Upon solving the optimization problem using quadratic programming, data points that lie within ϵ-tube have $\lambda_+^i = \lambda_-^i = 0$. The support vectors have $\lambda_+^i > 0$ or $\lambda_-^i > 0$, and we use them to calculate b. The dot product $(\mathbf{x}^i)^\top \mathbf{x}^j$ can be replaced with a kernel $k(\mathbf{x}^i, \mathbf{x}^j)$ and we can have a non-linear support vector regression.

7.6 Implementing Support Vector Machine

In this section, we will be implementing support vector machine using Scikit-Learn. We will be using the raisin dataset which can be downloaded from UC Irvine Machine Learning Repository. The data was collected from 900 raisins, including 450 from Kecimen and Besni raisin varieties grown in Turkey. The dataset contains seven morphological features which were extracted from the raisin images. The prediction task is a binary classification problem which is to classify the data into Kecimen and Besni raisin. The dataset is stored in zip file format. We will be using `wget`, a

utility function to download remote files from Internet. We also need to install `ZipFile` library to extract the zip file. Use conda or pip to install `wget` and `ZipFile`.

```
import pandas as pd
import wget
from zipfile import ZipFile
```

We use `wget` to download the file using the URL of the dataset.

```
data_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/
00617/Raisin_Dataset.zip'
wget.download(data_url)
```

Let us extract the zip file. The extracted file is contained in a folder name Raisin_Dataset.

```
with ZipFile('Raisin_Dataset.zip') as zObject:
    zObject.extractall()
```

The dataset file is available in two different formats, xlsx and arff. Let us read the xlsx file using Pandas. To read xlsx file, we need to install `OpenPyXL` library.

```
dataset = pd.read_excel('Raisin_Dataset/Raisin_Dataset.xlsx')
```

Prepare the dataset for splitting into training and test sets.

```
y = dataset['Class']
X = dataset.drop(columns=['Class'])
```

We split the dataset into training and test sets.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=10)
```

We create a function to print out the accuracy score, confusion matrix and classification report.

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

def print_clf_evaluation(y_test, y_pred):
    print(accuracy_score(y_test, y_pred))
    print(confusion_matrix(y_test, y_pred))
    print(classification_report(y_test, y_pred))
```

Support vector machine classifier can be built by importing SVC module from sklearn.svm. These are some of the parameters that can be set:

- `C` is the regularization parameter. This parameter accepts a positive integer value only. The default value is 1.
- `kernel` specifies the kernel type to be used in the algorithm. The possible values are `'linear'`, `'poly'`, `'rbf'`, `'sigmoid'` and `'precomputed'`. The default kernel is `'rbf'`.
- `degree` is the degree of the polynomial kernel function (`'poly'`). This parameter will be ignored by all other kernels. This parameter accepts integer value e.g. 2.

- `gamma` is kernel coefficient for `'rbf'`, `'poly'` and `'sigmoid'`. The possible values are `'scale'` (default), `'auto'` or float value. If `'scale'` is specified, gamma is $\frac{1}{d \cdot \sigma^2}$ where d is the number of features and σ^2 is the variance of the feature data. If `'auto'` is specified, `gamma` is $\frac{1}{d}$.
- `coef0` is an independent term in kernel function and it is only used in `'poly'` and `'sigmoid'`. This parameter accepts float value.

7.6.1 Linear SVM

First, let us build a linear SVM classifier.

```
from sklearn.svm import SVC
from sklearn.preprocessing import MinMaxScaler
from sklearn.pipeline import make_pipeline

svc_pipeline = make_pipeline(MinMaxScaler(), SVC(kernel='linear', C=1))
svc_pipeline.fit(X_train, y_train)
```

We evaluate the model on the test set.

```
y_pred = svc_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.8777777777777778
[[109  17]
 [ 16 128]]
              precision    recall  f1-score   support

       Besni       0.87      0.87      0.87       126
     Kecimen       0.88      0.89      0.89       144

    accuracy                           0.88       270
   macro avg       0.88      0.88      0.88       270
weighted avg       0.88      0.88      0.88       270
```

In the previous chapter, we demonstrated the use of `make_pipeline` to sequentially apply a list of steps such as standardization followed by model training. When using `make_pipeline`, each step is assigned a name which is generated automatically.

`Pipeline` module is used when the step names need to be manually specified. The purpose of manually specifying the step names is to assemble several steps while searching for different parameters using `GridSearchCV`. With `GridSearchCV`, we need to define a parameter grid for various steps in the pipeline. For this, we need to specify the step name and the parameter name separated by `__`.

For example, the pipeline consists of `MinMaxScaler` and `SVC` with linear kernel. We assign a name for each step such as `'scaler'` and `'svc'`. Then, we create an object of `Pipeline` by passing the pipeline's steps.

```
from sklearn.pipeline import Pipeline
scaler = MinMaxScaler()
svc = SVC(kernel='linear')
pipeline_steps = [('scaler', scaler), ('svc', svc)]
svc_pipeline = Pipeline(steps=pipeline_steps)
```

7.6.2 Using GridSearchCV

We define the parameter grid for `GridSearchCV`. There is no parameter for `MinMaxScaler`. We need to define the range of values for parameter `C` for `SVC`. To do that, we define the step name and parameter name separated by `__` as follows:

```
param_grid = {'svc__C': [0.01, 0.1, 0.5, 1, 3, 5, 10]}
```

Create the object of `GridSearchCV` and call `fit` to search for optimal `C` value.

```
from sklearn.model_selection import GridSearchCV
svc_grid_search = GridSearchCV(svc_pipeline, param_grid, cv=10)
svc_grid_search.fit(X_train, y_train)
```

The optimal value of `C` is

```
print(svc_grid_search.best_params_)
```

```
{'svc__C': 5}
```

We rebuild the SVM classifier with the optimal `C` value and evaluate the classifier.

```
svc_pipeline = make_pipeline(MinMaxScaler(), SVC(kernel='linear', C=5))
svc_pipeline.fit(X_train, y_train)
y_pred = svc_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.8851851851851852
[[109  17]
 [ 14 130]]
              precision    recall  f1-score   support

       Besni       0.89      0.87      0.88       126
     Kecimen       0.88      0.90      0.89       144

    accuracy                           0.89       270
   macro avg       0.89      0.88      0.88       270
weighted avg       0.89      0.89      0.89       270
```

7.6.3 Non-Linear SVM

Let us build a non-linear SVM classifier using RBF kernel.

```
scaler = MinMaxScaler()
svc = SVC(kernel='rbf')
pipeline_steps = [('scaler', scaler), ('svc', svc)]
svc_pipeline = Pipeline(steps=pipeline_steps)
```

We define the parameter grid for `GridSearchCV` to determine the optimal values and run the parameter searching.

```
param_grid = {'svc__C': [0.01, 0.1, 0.5, 1, 3, 5, 10],
              'svc__gamma': ['scale', 'auto']}
svc_grid_search = GridSearchCV(svc_pipeline, param_grid, cv=10)
svc_grid_search.fit(X_train, y_train)
```

The optimal parameters are

```
svc_grid_search.best_params_
```

```
{'svc__C': 5, 'svc__gamma': 'scale'}
```

We build the non-linear SVM classifier using the optimal parameters.

```
svc_pipeline = make_pipeline(MinMaxScaler(), SVC(kernel='rbf',
                                                 C=5, gamma='scale'))
svc_pipeline.fit(X_train, y_train)
y_pred = svc_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.8888888888888888
[[107  19]
 [ 11 133]]
              precision    recall  f1-score   support

       Besni       0.91      0.85      0.88       126
     Kecimen       0.88      0.92      0.90       144

    accuracy                           0.89       270
   macro avg       0.89      0.89      0.89       270
weighted avg       0.89      0.89      0.89       270
```

In this section, we will be implementing support vector machine for a regression problem. We will be using California Housing dataset. We load the dataset and split it into training and test sets.

```
from sklearn.datasets import fetch_california_housing
dataset = fetch_california_housing()
X_train, X_test, y_train, y_test = train_test_split(dataset.data, dataset.target,
    test_size=0.3, random_state=10)
```

Support vector machine regressor can be built by importing `SVR` module. The parameters to be set are similar to `SVC` which are `C`, `kernel`, `degree`, `gamma`, `coef0` and `epsilon`. `epsilon` is an additional parameter which determines the width of the tube around the estimated hyperplane.

The default value of `epsilon` is 0.1.

```
from sklearn.svm import SVR
svr = SVR(C=0.5, epsilon=0.2)
svr_pipeline = make_pipeline(MinMaxScaler(), svr)
svr_pipeline.fit(X_train, y_train)
```

We create a function to calculate the regression loss.

```
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
def print_reg_evaluation(y_test, y_pred):
    print('Mean Absolute Error: ', mean_absolute_error(y_test, y_pred))
    print('Mean Squared Error: ', mean_squared_error(y_test, y_pred))
    print('R-squared: ', r2_score(y_test, y_pred))
```

We evaluate the model on the test set.

```
y_pred = svr_pipeline.predict(X_test)
print_reg_evaluation(y_test, y_pred)
```

```
Mean Absolute Error:  0.4618549114230757
Mean Squared Error:  0.4732671164943599
R-squared:  0.6508906344747241
```

Exercises

1. Explain the main principle of SVM.
2. Explain the properties of Hinge loss.
3. Explain the role of C and how it affects the bias-variance trade-off.
4. Explain the concept of kernel trick.
5. Explain the effect of γ in support vector classifier with RBF kernel.
6. Explain the effect of using different ξ on bias and variance in support vector regression.

Chapter 8

Clustering

8.1 Introduction

In many cases, we are given a dataset but the instances have no labels associated with it. Specifically, we are given only $\mathbf{x}^i$ and not the label y^i. Thus, we do not know how many labels are there in the data, and which $\mathbf{x}^i$ comes from which y^i. This dataset is known as unlabeled dataset $D = \mathbf{x}^1, \mathbf{x}^2, ..., \mathbf{x}^N$. Since the labels are not known, we need to estimate the labels. Then, given the labels, we estimate the labels of the instances based on certain assumptions as shown in Figure 8.1. This problem is known as clustering.

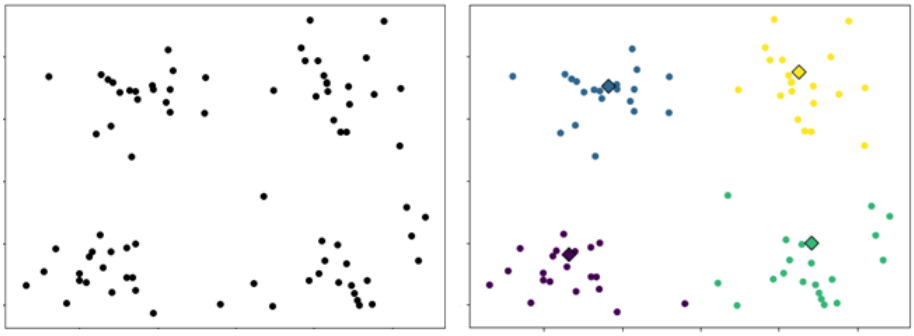


Figure 8.1 Clustering is partitioning the instances into a number of groups (clusters) such that instances belonging to a group are close to each other

8.2 k-Means

Suppose we have a dataset consisting of N instances. We want to partition the instances into k clusters. We can view a cluster as a group of data points that are close to each other which means the distances between the data points within the cluster are smaller than the distances between the data points and the data points outside the cluster. Thus, in k-means, the clusters are represented by the center of clusters, also known as the centroid (Forgy 1965; Lloyd 1982). The k-means clustering process can be divided into two repeating steps: cluster assignment and centroid update. The assignment step assigns data points to one of the k clusters based on the distance between the data points and the centroids. Let μ_k denote the centroid k. Data point $\mathbf{x}^i$ is assigned

to cluster μ_k whereby the distance between the data point and the centroid is minimum.

$$r^i = \arg\min_k \|\mathbf{x}^i - \mu_k\|^2 \tag{8.1}$$

where r^i is the index of cluster $k = 1, 2, ..., K$ to which $\mathbf{x}^i$ is assigned to and $\|\mathbf{x}^i - \mu_k\|^2$ is the euclidean distance.

After the assignment step, each centroid is updated by calculating the mean of the data points that are assigned to it. Let N_k denote the number of data points that are assigned to cluster k. The centroid update is given as follows:

$$\mu_k = \frac{1}{N_k} \sum_{i:r^i=k} \mathbf{x}^i \tag{8.2}$$

The goal is to find the assignment of the data points such that the distances between the data points of the clusters and their centroids are minimum. Thus, the two steps are repeated to minimize the following objective function.

$$(r^1, ..., r^N, \mu, ..., \mu^K) = \arg \min_{(r^1,...,r^N,\mu^1,...,\mu^K)} \sum_{i=1}^{N} \|\mathbf{x}^i - \mu_{r^i}\|^2 \tag{8.3}$$

k-means algorithm is given in Algorithm 4. We begin by specifying the number of clusters K. Each centroid is randomly initialized. In the next section, we will discuss an improved initialization method for the centroids. Then, repeatedly assign each data point to its nearest centroid and update each centroid by calculating the mean of all data points assigned to it. The algorithm is terminated when the assignments no longer change. Figure 8.2 illustrates the k-means clustering process.

Algorithm 4 k-Means Algorithm

1: initialize $\mu_1, \mu_2, ..., \mu_K$
2: **repeat**
3: Assignment: $r^i \leftarrow \arg\min_k \|\mathbf{x}^i - \mu_k\|^2$
4: Update: $\mu_k \leftarrow \frac{1}{N_k} \sum_{i:r^i=k} \mathbf{x}^i$
5: **until** converged

Numerical Example: Consider the following set of data points.

$$\mathbf{x} = \begin{bmatrix} 4 & 3 \\ 5 & 4 \\ 1 & 1 \\ 2 & 1 \end{bmatrix}$$

Suppose $k = 2$. Initialize the centroids.

$$\mu_k = \begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix}$$

For each data point, calculate its distance to the centroids.

For example, the distance between x^1 and μ_1 and μ_2 are

$d_1^1 = \sqrt{(4-1)^2 + (3-1)^2} = 3.61$

$d_2^1 = \sqrt{(4-2)^2 + (3-1)^2} = 2.83$

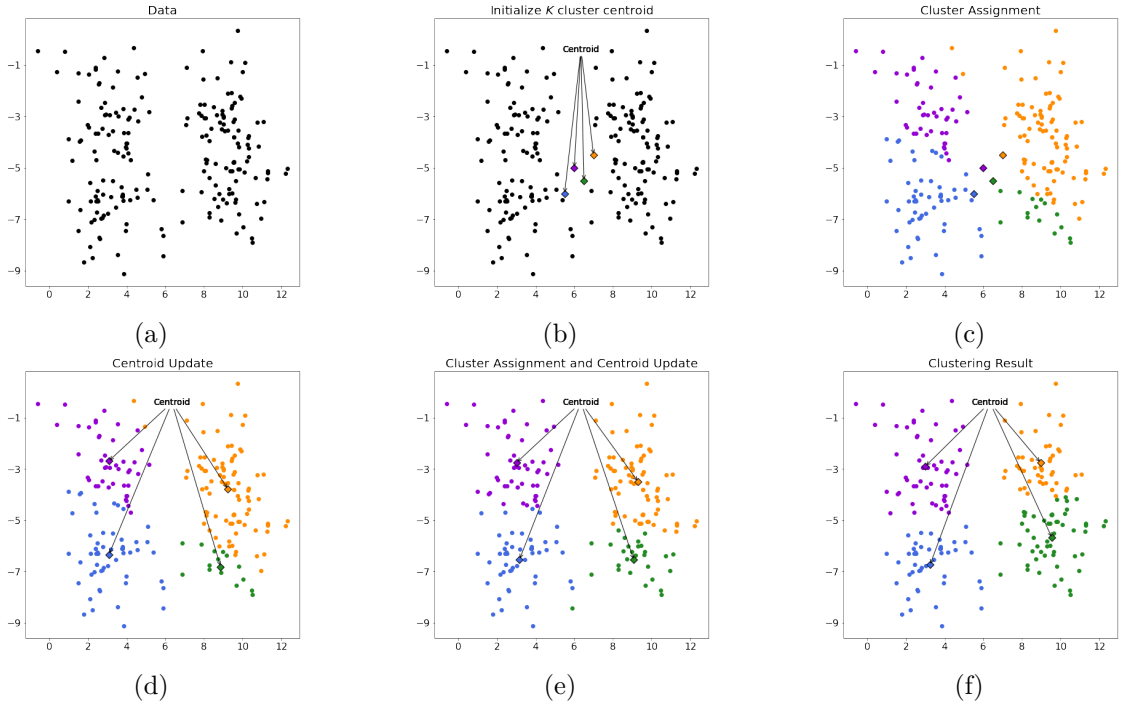


Figure 8.2 k-means steps are performed repeatedly until converged

$$d_1 = \begin{bmatrix} 3.61 \\ 5 \\ 0 \\ 1 \end{bmatrix}$$

$$d_2 = \begin{bmatrix} 2.83 \\ 4.24 \\ 1 \\ 0 \end{bmatrix}$$

For each data point, assign it to the closest centroid.

$$r = \begin{bmatrix} 2 \\ 2 \\ 1 \\ 2 \end{bmatrix}$$

Update the centroids.

$\mu_1 = (1, 1)$

$\mu_2 = (\frac{4+5+2}{3}, \frac{3+4+1}{3}) = (\frac{11}{3}, \frac{8}{3})$

The computation is repeated until the centroids no longer change.

8.2.1 k-Means++

Random initialization could lead to inconsistency and sub-optimal clustering. Consider a set of data points as shown in Figure 8.3. Assuming the number of clusters is 2 and the centroids are initialized randomly and happen to be at the location indicated by "x" marker in green and red colors. The data points at the top region are closer to the green marker while the data points at the

bottom region are closer to the red marker. The k-means algorithm will converge with the data points at the bottom region clustered together and the data points at the top region forming the second cluster. This is a sub-optimal clustering because the distance between the data points on the left and right is larger than the distance of the data points from the top to the bottom.

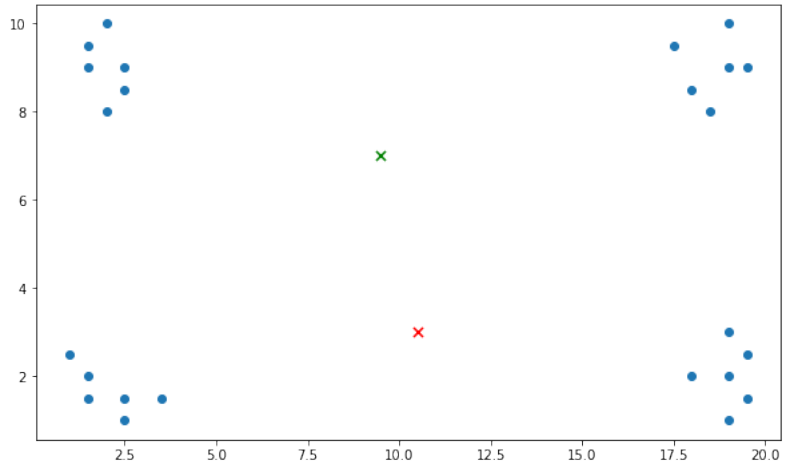


Figure 8.3 k-means algorithm produces a sub-optimal clustering due to the location of the initial centroids

k-means++ is an initialization method for k-means clustering algorithm (Arthur and Vassilvitskii 2007). Rather than just randomly initialize the centroids, k-means++ randomly chooses one of the data points to be the first centroid. Then, the subsequent centroids are chosen from the remaining data points with the probability proportional to the distance of the data points to the previously chosen centroid. As a result, the centroids will be far away from each other. The algorithm of k-means++ is given in Algorithm 5:

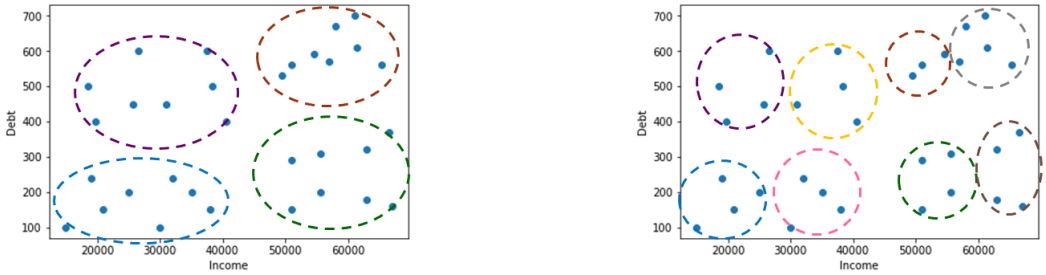
Algorithm 5 k-means++ Algorithm

1: Randomly pick one data point as the first centroid μ_1
2: **for** $k = 1$ to K **do**
3: Compute the distance of each data point from μ_k, $d(x^i) = \|x^i - \mu_k\|^2$
4: Compute the probability for each data point, $p(x^i)\frac{d(x^i)}{\sum_{j=1}^{N} d(x^j)}$
5: Choose one data point with the largest probability as μ_{k+1}
6: **end for**

8.2.2 Choosing the Value of k

As discussed, the number of clusters affects the clustering result. Thus, it is important to carefully choose the value of k. Unfortunately, there is no standard method to choose the value of k. It is common for the value of k to be chosen manually through visualization or based on domain knowledge. Typically, the value of k is estimated based on some later or downstream purpose. For example, consider a clustering problem in banking as shown in Figure 8.4(a). The features are income and debt. The data points can be grouped into four clusters. The clusters at the lower regions may represent customers with low debt, one with low income indicated by the blue oval and the second cluster with high income indicated by the green oval. The clusters at the top region represent customer with high debt and low income (purple oval) and high income (brown oval). Instead of four clusters, the data points can be grouped into eight clusters in which the clusters represent customers with low debt and high debt, but with different income scales as shown in

Figure 8.4(b).



(a) (b)

Figure 8.4 The number of clusters is defined based on a downstream purpose

Another way of defining the number of clusters is to determine the value based on the objective function. The objective function which is the sum of squared distances between the data points and their centroids indicates the compactness of the clusters (Blömer et al. 2016). As the value of k increases, the sum of squared distances decreases because each cluster will have fewer data points and the data points will be closer to their clusters. However, at some point, the rate of decrease of the sum of squared distances will start to decline due to overfitting. The value of k at which the decrease rate starts to decline is known as the elbow. Figure 8.5 shows a plot of sum of squared distances against different values of k. A sharp decrease in the sum of squared distances can be observed for the number of clusters equal to 2 to 4. Also, it is observed that the sum of squared distances starts to stagnant for the following number of clusters. Thus, the value of k is chosen to be 4.

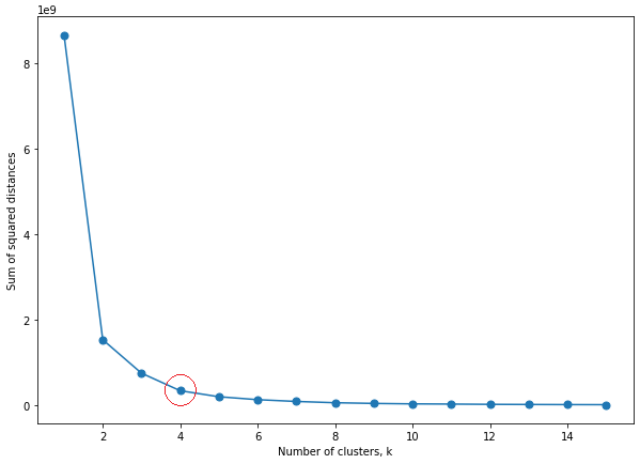


Figure 8.5 A plot of sum of squared distances against number of clusters. The elbow is indicated by the red circle

8.3 Hierarchical Agglomerative

The clustering results of k-means algorithm depend on the value of k and the results may vary from one clustering to another. Unlike k-means, hierarchical agglomerative clustering is a clustering algorithm that does not require a pre-specified number of clusters (Nielsen 2016). The algorithm begins with every data point representing a singleton cluster, in other words, there will be N number of clusters. Then, the algorithm recursively merges a pair of clusters with the closest distance until

all data points are merged into a single cluster. The iterative process is illustrated in Figure 8.6. There are five data points whereby each representing a singleton cluster. At each iteration, two closest clusters are merged into a single cluster, resulting in one less cluster. For example, in Figure 8.6(a), cluster 1 and cluster 2 are merged since those two are the closest pair. This step produces one less cluster (cluster 1-2, cluster 3, cluster 4 and cluster 5). In the next iteration, cluster 4 and cluster 5 are the closest pair, hence the clusters are merged as shown in Figure 8.6(b). This process is repeated until all clusters are progressively merged, as shown in Figures 8.6(c) and 8.6(d).

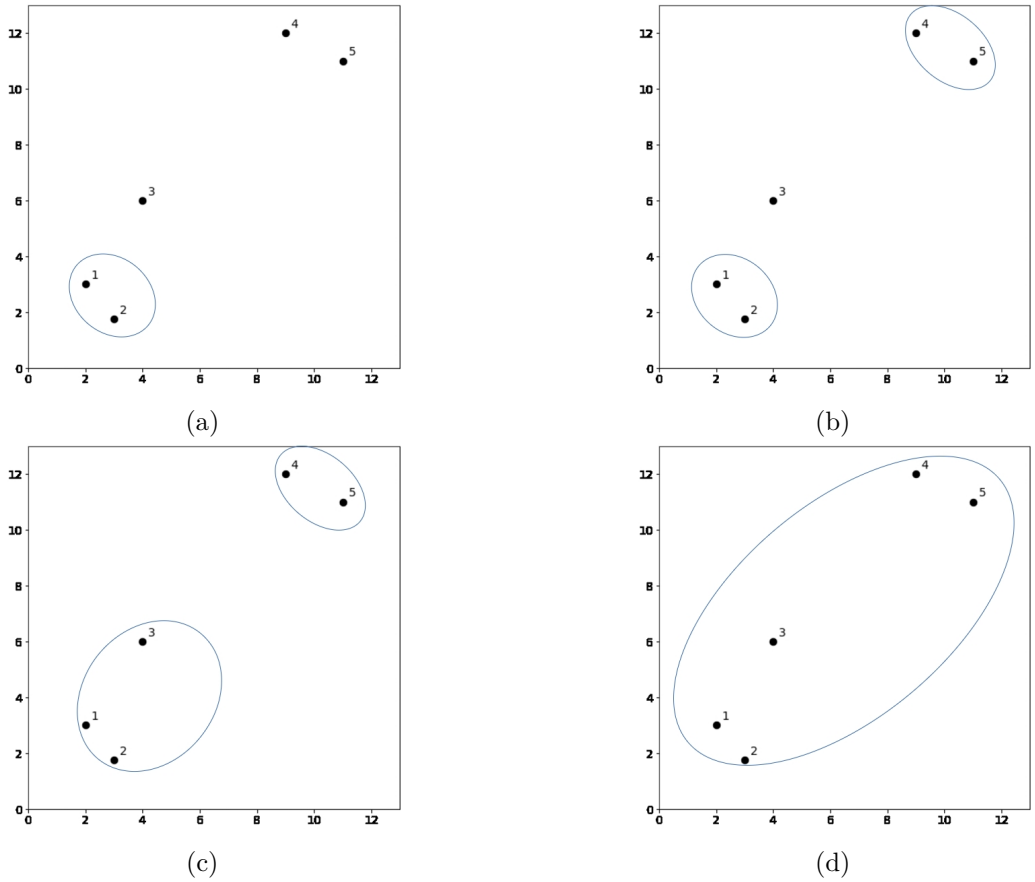
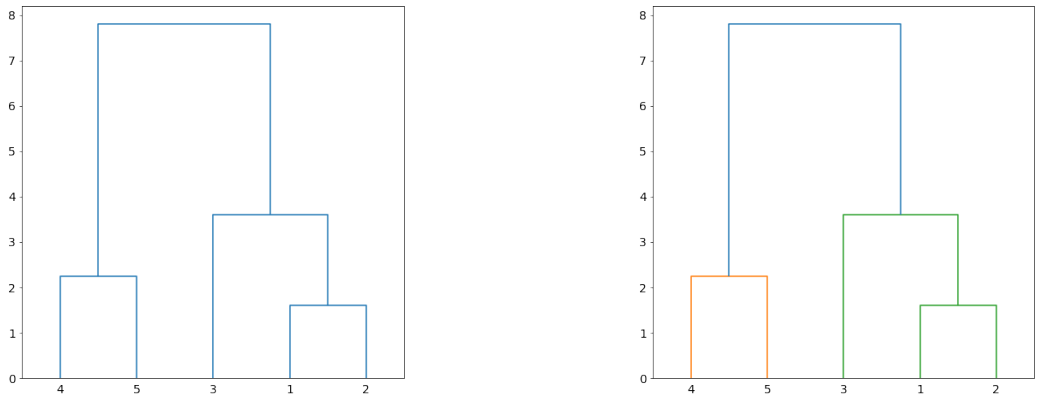


Figure 8.6 Data points are iteratively merged based on the distance between the clusters

8.3.1 Dendrogram

The iterative process produces a hierarchical representation in which the clusters at each level of the hierarchy are created. The hierarchical representation is called dendrogram. Figure 8.7(a) shows the dendrogram of clustering the data points in Figure 8.6. A dendrogram is a tree-like structure representing the order of the cluster merging process. The leaf nodes (numbers at the bottom of the figure) are the initial clusters. The horizontal lines connecting the clusters indicate the merging of two pairs of clusters. Furthermore, the lines show the sequence at which the clusters are merged at every level (iteration) with the connected lines at the bottom indicates the first in the sequence while the connected lines at the top is the last pair of clusters to be merged. The height of lines is proportional to the distance between the clusters. Clustering of a given size is induced if we cut the dendrogram at any given height. For example, if we cut at 5.0, two clusters will be induced as shown in Figure 8.7(b) on the right.



(a) (b)

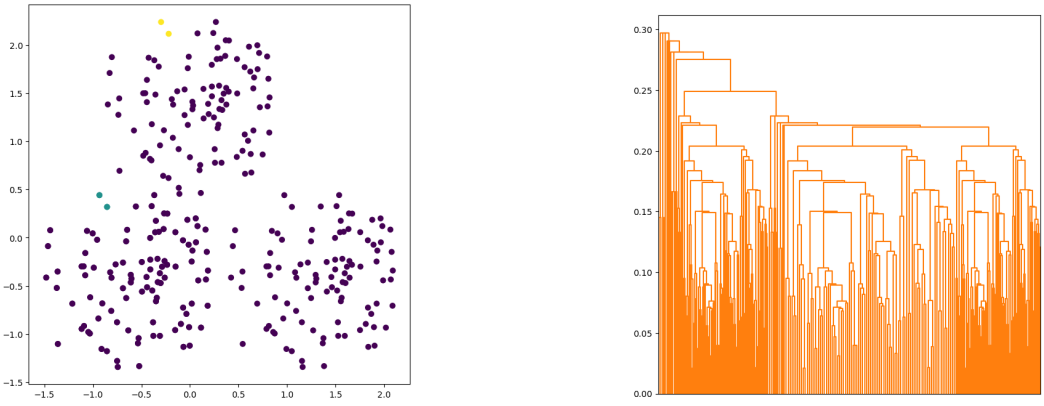
Figure 8.7 (a) The resulting dendrogram and (b) cutting the dendrogram at 5.0 induces two clusters indicated by orange and green lines

8.3.2 Linkage

Linkage specifies the dissimilarity or distance between two clusters (Murtagh and Contreras 2012). The distances between the clusters in Figure 8.6 are computed using single linkage or also known as nearest neighbor linkage. Single linkage defines the distance between two clusters G and H as the minimum distance between two data points in G and H.

$$d_{SL}(G, H) = \min_{x \in G, z \in H} d(x, z) \tag{8.4}$$

where $d(x, z)$ is the Euclidean distance. Single linkage defines the distance by considering only a pair of closest data points from two clusters regardless of the similarity of the other data points of the clusters. This has the effect of "chaining" whereby the algorithm tends to merge two clusters if two data points from these clusters are close enough regardless of the other data points in the clusters. As a result, the clusters that will be formed are not compact or have high within-cluster variation as shown in Figure 8.8.



(a) (b)

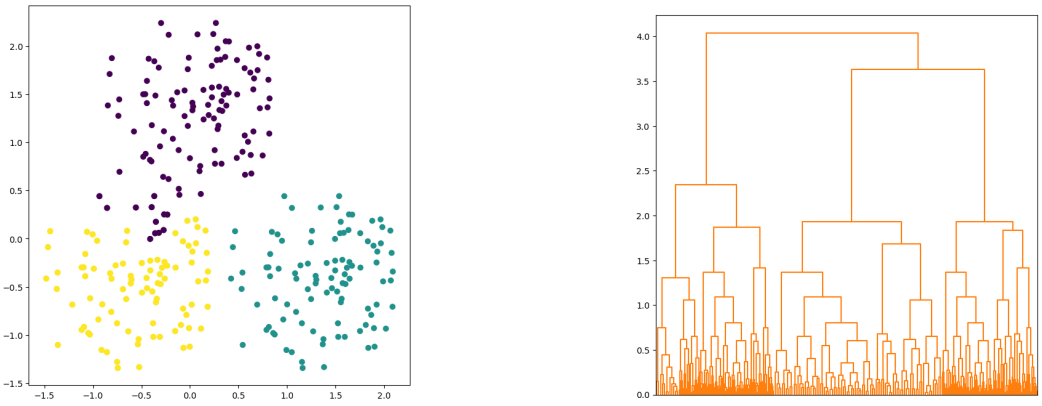
Figure 8.8 Clustering the data using single linkage agglomerative clustering: (a) the clustering result and (b) the dendrogram

Complete linkage is the opposite of single linkage whereby the distance between two clusters is

determined by the maximum distance between two points in G and H.

$$d_{CL}(G,H) = \max_{x \in G, z \in H} d(x,z) \tag{8.5}$$

Since the linkage is defined by the maximum distance, two clusters are considered close (or merged) if the data points in the clusters are relatively similar. Thus, the algorithm tends to produce more compact clusters. However, data points assigned to a cluster could be closer to data points of other clusters than they are to some data points of their own cluster. As shown in Figure 8.9, some data points that are assigned to purple cluster are closer to yellow cluster.



(a) (b)

Figure 8.9 Clustering the data using complete linkage agglomerative clustering: (a) the clustering result and (b) the dendrogram

Average linkage defines the distance between two clusters by calculating the average distance between all data points in the clusters. The linkage is defined as follows:

$$d_{AL}(G,H) = \frac{1}{N_G \dot{N}_H} \sum_{x \in G, z \in H} d(x,z) \tag{8.6}$$

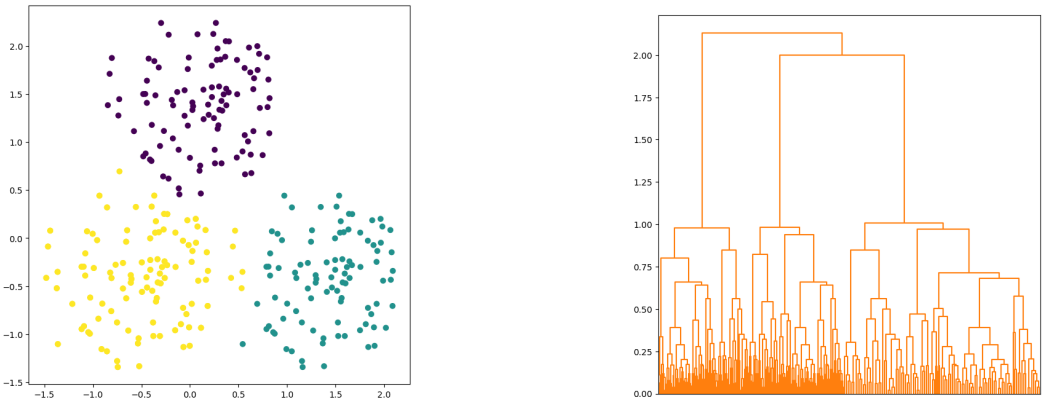
where N_G and N_H are the number of data points in cluster G and H. Average linkage tries to strike a balance whereby all data points in the clusters contribute equally in determining the distance between the clusters. Hence, the clusters tend to be more compact and far apart as shown in Figure 8.10.

Ward linkage is another linkage that produces a good clustering result. The linkage defines the distance between two clusters by considering the increase in variance when the clusters are merged. Ward linkage is defined as follows:

$$\begin{aligned} d_{WL}(G,H) &= \sum_{x \in G \cup H} d(x, \mu_{G \cup H}) - \sum_{x \in G} d(x, \mu_G) - \sum_{x \in H} d(x, \mu_H) \\ &= \frac{N_G N_H}{N_G + N_H} d(\mu_G, \mu_H) \end{aligned} \tag{8.7}$$

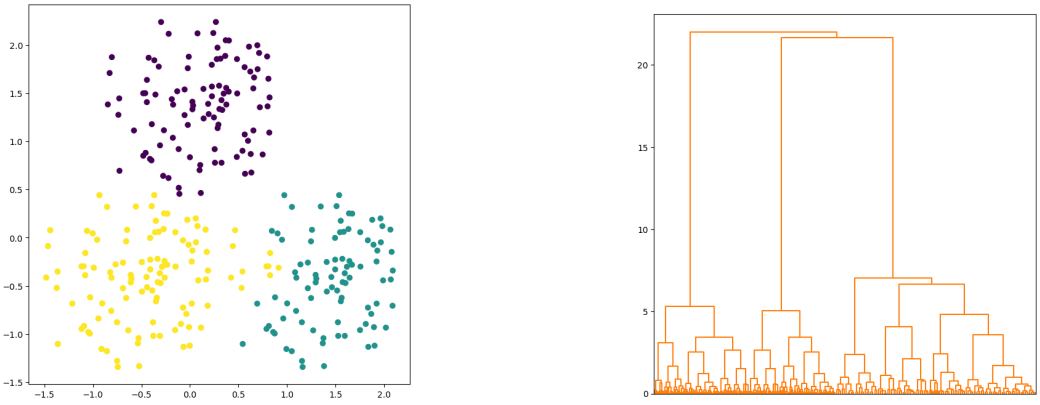
where N_G and N_H are the number of data points in cluster G and H. Instead of calculating the distance between the data points, ward linkage calculates the distance between the data points and the centroid of the clusters. Hence, the clusters are compact and far apart as shown in Figure 8.11.

Numerical Example: Consider the following set of data points.



(a) (b)

Figure 8.10 Clustering the data using complete linkage agglomerative clustering: (a) the clustering result and (b) the dendrogram



(a) (b)

Figure 8.11 Clustering the data using complete linkage agglomerative clustering: (a) the clustering result and (b) the dendrogram

$$\mathbf{x} = \begin{bmatrix} 1.0 & 1.0 \\ 1.5 & 1.5 \\ 5.0 & 5.0 \\ 3.0 & 4.0 \\ 4.0 & 4.0 \\ 3.0 & 3.5 \end{bmatrix}$$

Suppose we use complete linkage. Calculate the distance between the data points. The distance matrix is given as follows:

	x^1	x^2	x^3	x^4	x^5	x^6
x^1	0.00	0.71	5.66	3.61	4.24	3.20
x^2	0.71	0.00	4.95	2.92	3.54	2.50
x^3	5.66	4.95	0.00	2.24	1.41	2.50
x^4	3.61	2.92	2.24	0.00	1.00	0.50
x^5	4.24	3.54	1.41	1.00	0.00	1.12
x^6	3.20	2.50	2.50	0.50	1.12	0.00

where the first row lists the distances between x_1 and other data points, second row lists the distances between x_2 and other data points, etc.

Merge two closest clusters. We merge x_4 and x_6 since their distance is minimum (closest). Then, complete linkage is used to determine the distance between x_{4-6} and other data points. The distance matrix is given as follows:

	x^1	x^2	x^3	x^{4-6}	x^5
x^1	0.00	0.71	5.66	3.61	4.24
x^2	0.71	0.00	4.95	2.92	3.54
x^3	5.66	4.95	0.00	2.50	1.41
x^{4-6}	3.61	2.92	2.50	0.00	1.12
x^5	4.24	3.54	1.41	1.12	0.00

We repeat the process. Merge two closest clusters. We merge x_1 and x_2 since their distance is minimum (closest). Then, complete linkage is used to determine the distance between x_{1-2} and other data points. The distance matrix is given as follows:

	x^{1-2}	x^3	x^{4-6}	x^5
x^{1-2}	0.00	5.66	3.61	4.24
x^3	5.66	0.00	2.50	1.41
x^{4-6}	3.61	2.50	0.00	1.12
x^5	4.24	1.41	1.12	0.00

We repeat the process. Merge two closest clusters. We merge x_{4-6} and x_5 since their distance is minimum (closest). Then, complete linkage is used to determine the distance between x_{4-5-6} and other data points. The distance matrix is given as follows:

	x^{1-2}	x^3	x^{4-5-6}
x^{1-2}	0.00	5.66	4.24
x^3	5.66	0.00	2.50
x^{4-5-6}	4.24	2.50	0.00

We repeat the process. Merge two closest clusters. We merge x_3 and x_{4-5-6} since their distance is minimum (closest). Then, complete linkage is used to determine the distance between $x_{3-4-5-6}$ and x_{1-2}. The distance matrix is given as follows:

	x^{1-2}	$x^{3-4-5-6}$
x^{1-2}	0.00	5.66
$x^{3-4-6-6}$	5.66	0.00

We repeat the process. Merge x_{1-2} and $x_{3-4-5-6}$ since there are two clusters only. The dendrogram of the clustering is given in Figure 8.12.

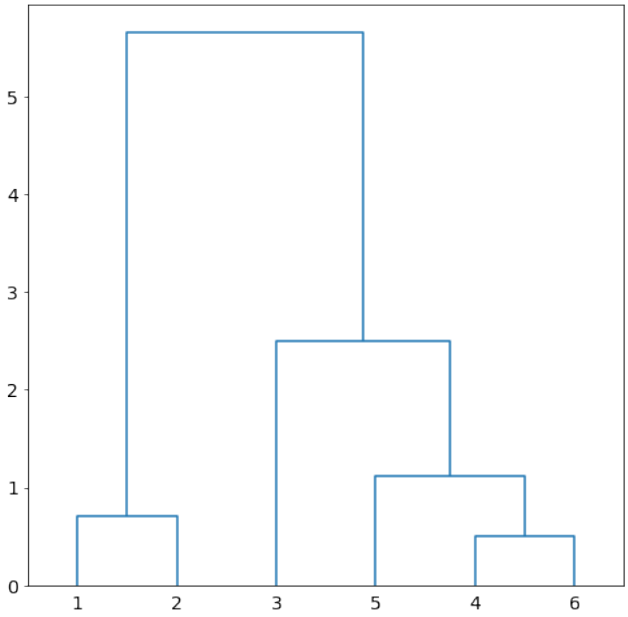


Figure 8.12 The dendrogram of complete linkage agglomerative clustering

8.4 Implementing Clustering Methods

In this section, we will be implementing the clustering methods using Scikit-Learn. We will generate synthetic data and then model it using the clustering methods. Scikit-Learn provides a function called `make_blobs` to generate isotropic Gaussian blobs. Let us import the function.

```
from sklearn.datasets import make_blobs
import numpy as np
```

`make_blobs` has several parameters which can be specified to control the synthetic data generation.

- `n_samples` is the total number of data points of the clusters. This parameter accepts an integer or an array. If the value is integer, it is the total number of data points equally divided among clusters. If an array is specified, each element of the array indicates the number of data points per cluster. The default value is 100.
- `centers` is the number of cluster centers to generate. This parameter accepts an integer value or None. This parameter is `None` by default and the number of clusters depends on parameter `n_samples`.
- `n_features` is the number of features for each instance. This parameter accepts an integer value and by default, the value is 2.
- `cluster_std` is the standard deviation of the clusters. By default, the value is 1.0.

Let us generate 4 clusters with 1.5 standard deviation and each cluster consists of 250 data points. The number of features is 2.

```
X, y = make_blobs(n_samples=500, n_features=2, centers=4, cluster_std=1.7)
```

Let us plot the data points using scatter plot. We set the size of the data points by passing a value to parameter `s`. The scatter plot is given in Figure 8.13.

```
from matplotlib import pyplot as plt
plt.figure(figsize=(8,6))
plt.scatter(X[:,0], X[:,1], s=3)
```

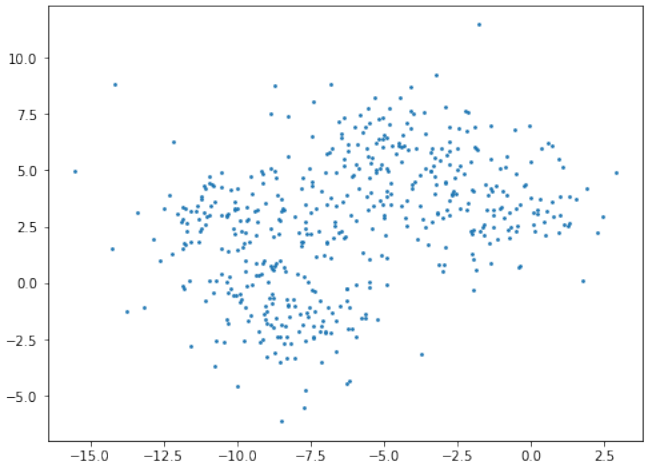


Figure 8.13 The generated data points

8.4.1 *k*-means Clustering

k-means clustering model can be built by importing `KMeans` module from sklearn.cluster. The module has several parameters as follows:

- `n_clusters` is the number of clusters. By default, the value is 8.
- `init` is the method of centroid initialization. We can either specify `'k-means++'`, `'random'` or an array. If `'k-means++'` is specified, the centroid will be initialized using k-means++ algorithm. If `'random'` is specified, the centroid will be initialized randomly. An array is passed if we want to initialize the location of the centroids manually. The default value is `'k-means++'`.
- `max_iter` is the maximum number of iterations of the k-means algorithm.

Now let us import `KMeans` from Scikit-Learn to model the data. We create an object of `KMeans`. We set the number of clusters equal to 4.

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=4)
```

We call `fit` to train the clustering model. Notice that we pass only the data only.

```
kmeans.fit(X)
```

We can display the centroid values the algorithm has generated for the final clusters using `cluster_centers_`

```
centroids = kmeans.cluster_centers_
print(centroids)
```

```
[[ -5.08321972   5.50372717]
 [-10.26313999   2.77233473]
 [ -8.01549434  -1.22373694]
 [ -1.06491259   3.39548429]]
```

We can display the summation of squared distance of the data points to their closest cluster using `inertia_`.

```
print(kmeans.inertia_)
```

```
2403.3542836498605
```

Let us display the clusters using the scatter plot. Function `subplots` is used to create a figure and a set of subplots. In this example, we create only one subplot. The function returns the figure and the axes. Axes is the region of the image with the data space. We use the axes object to display and annotate the scatter plot. The color of the centroids is set to red by passing `'r'` to parameter `c`. The scatter plot is given in Figure 8.14.

```
fig, ax = plt.subplots(figsize=(8,6))
ax.scatter(X[:,0], X[:,1], s=3)
ax.scatter(centroids[:,0], centroids[:,1], s=10, c='r')
ax.annotate('Cluster 1', xy=(centroids[0,0],centroids[0,1]))
ax.annotate('Cluster 2', xy=(centroids[1,0],centroids[1,1]))
ax.annotate('Cluster 3', xy=(centroids[2,0],centroids[2,1]))
ax.annotate('Cluster 4', xy=(centroids[3,0],centroids[3,1]))
```

Figure 8.14 The centroids are indicated by the red marker

Now, let us classify the data points using the k-means model.

```
y_pred = kmeans.predict(X)
```

We display the clusters using scatter plot. But now, we colorize the data points based on the prediction of the cluster model. Note the parameter `c` is set to `y_pred`. The scatter plot is given in Figure 8.15.

```
fig, ax = plt.subplots(figsize=(8,6))
ax.scatter(X[:,0], X[:,1], s=3, c=y_pred)
ax.scatter(centroids[:,0], centroids[:,1], s=10, c='r')
ax.annotate('Cluster 1', xy=(centroids[0,0],centroids[0,1]))
ax.annotate('Cluster 2', xy=(centroids[1,0],centroids[1,1]))
ax.annotate('Cluster 3', xy=(centroids[2,0],centroids[2,1]))
ax.annotate('Cluster 4', xy=(centroids[3,0],centroids[3,1]))
```

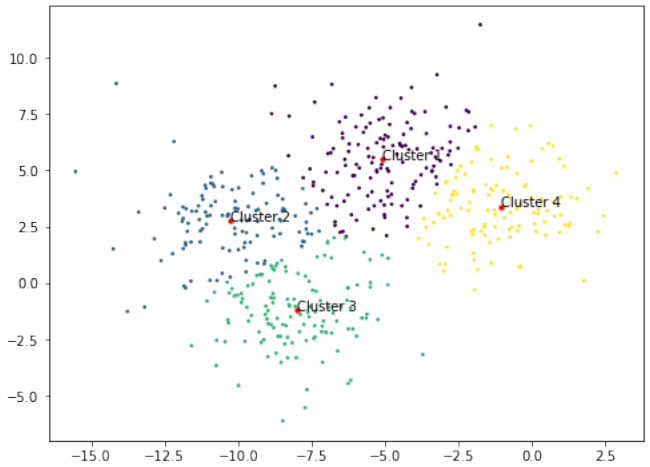


Figure 8.15 The clustering result of k-means

To manually define the initial values of the centroids, create an array containing the centroid values.

```
init_centroids = np.array([[0.01,0.02],[0.01,0.01],[0.02,0.01],[0.02,0.02]])
```

Then, pass the array to parameter `init` when the object is created.

```
kmeans = KMeans(n_clusters=4, init=init_centroids)
```

Let us display a plot of summation of squared distance against the number of clusters. We implement a loop to iterate over a set of number of clusters from 1 to 10. For each iteration, we create a model with the corresponding number of cluster, fit the model and get the `inertia_` value.

```
inertia = []
m = 10
for k in range(1,m):
    km = KMeans(n_clusters=k)
    km.fit(X)
    inertia.append(km.inertia_)
```

Plot the sum of squared distance against the number of clusters. As we can see, after k=4, the decrease in sum of squared distance is not significant. The plot is given in Figure 8.16.

```
plt.figure(figsize=(8,6))
num_clusters = [k for k in range(1,10)]
plt.plot(num_clusters, inertia, marker='.', markersize=10)
plt.xlabel("Number of clusters, k")
plt.ylabel("Inertia (Sum of squared distance)")
```

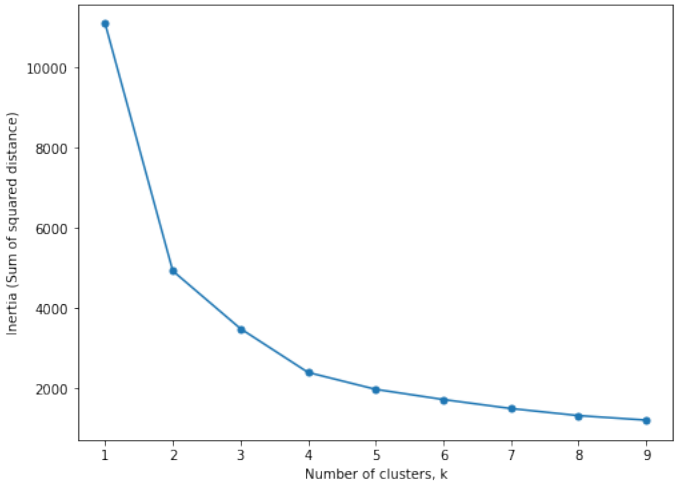


Figure 8.16 The sum of squared distance against number of clusters

8.4.2 Agglomerative Clustering

Now, let us implement Agglomerative Clustering. First, we create the dendrogram of the dataset. Note that this step is optional and can be skipped to the implementation of Agglomerative Clustering using Scikit-Learn. To create a dendrogram, we will be using the SciPY library. SciPY is a Python library used for scientific and technical computing. Two functions called `linkage` and `dendrogram` are needed to create a dendrogram.

`linkage` is used to define the type of linkage to create the dendrogram. The parameters to be set are as follows:

- `method` is the linkage method. The value is either '`single`', '`complete`', '`average`' or '`ward`'. The default value is '`single`'.
- `metric` is the metric to calculate the distance. The value is either '`euclidean`', '`cityblock`', '`cosine`', '`hamming`'. The default value is '`euclidean`'.

`dendrogram` method is used to create the dendrogram. Some of the parameters to be set are as follows:

- `Z` is the linkage method
- `color_threshold` is for brevity. Let t be the threshold, colors all links below t.
- `orientation` is the direction to plot the dendrogram, which can be either '`top`', '`bottom`', '`left`' or '`right`'. By default, the value is '`top`'.

First, we need to define the linkage. Here, we use complete linkage.

```
from scipy.cluster.hierarchy import linkage
linked = linkage(X, method='ward')
```

We create the dendrogram by passing the linkage. The dendrogram is given in Figure 8.17.

```
from scipy.cluster.hierarchy import dendrogram
plt.figure(figsize=(8,6))
dendrogram(linked, orientation='top')
plt.show()
```

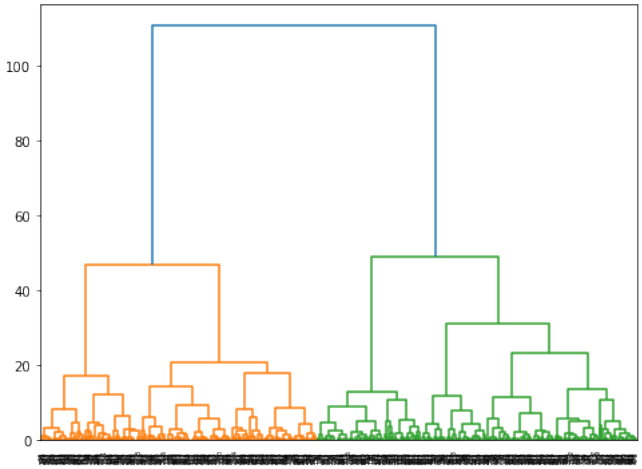


Figure 8.17 The created dendrogram using complete linkage

We have created a dendrogram using complete linkage. As we can see, dendrogram partitions the data into two clusters, indicated by orange and green. We can specify the cut threshold so that data will be partitioned into the desired clusters. To specify the cut threshold, we pass a value to parameter `color_threshold`. All links connecting nodes with distances greater than or equal to the threshold will be colored blue. Let us specify 35 as the cut threshold. Figure 8.18 shows the dendrogram with the specified threshold.

```
plt.figure(figsize=(8,6))
dendrogram(linked, color_threshold=35)
plt.show()
```

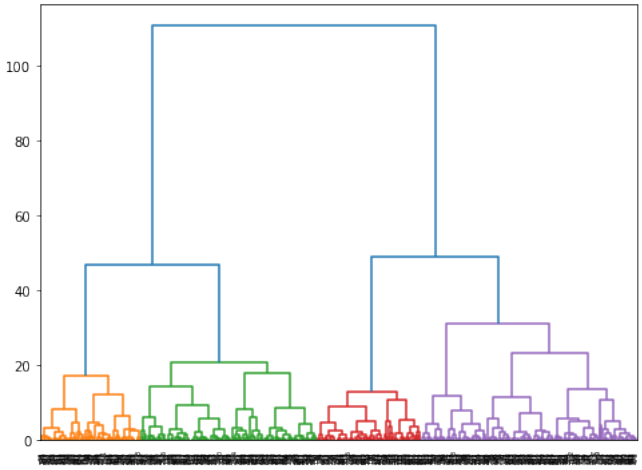


Figure 8.18 The dendrogram with cut threshold equal to 35

As we can see the links with distances greater than 35 are colored blue. Four clusters are created indicated by the orange, green, red and purple. Try using different linkages to see how the data will be clustered.

Let us cluster the data using Agglomerative Clustering. We have to import the module from Scikit-Learn. Agglomerative clustering model can be built by importing `AgglomerativeClustering` module from `sklearn.cluster`. The module has several parameters as follows:

- `n_clusters` is the number of clusters. The value is either an integer or `None`. The value must be `None`, if the `distance_threshold` is not `None`. By default, the value is 2.
- `affinity` is the metric used to compute the linkage. The value is either `'euclidean'`, `'l1'`, `'l2'`, `'manhattan'` or `'cosine'`.
- `linkage` is the maximum number of iterations of the k-means algorithm. The value is either `'ward'`, `'complete'`, `'average'` or `'single'`. The default value is `'ward'`.
- `distance_threshold` is the linkage distance threshold above which, clusters will not be merged. This parameter accepts a float or `None`. The default value is `None`.

Now, let us import `AgglomerativeClustering` from `sklearn.cluster` to model the data.

```
from sklearn.cluster import AgglomerativeClustering
agg = AgglomerativeClustering(n_clusters=4)
y_pred = agg.fit_predict(X)
```

We plot the clustering result using scatter plot. The scatter plot is given in Figure 8.19.

```
fig, ax = plt.subplots(figsize=(8,6))
ax.scatter(X[:,0], X[:,1], s=3, c=y_pred)
ax.scatter(centroids[:,0], centroids[:,1], s=10, c='r')
ax.annotate('Cluster 1', xy=(centroids[0,0],centroids[0,1]))
ax.annotate('Cluster 2', xy=(centroids[1,0],centroids[1,1]))
ax.annotate('Cluster 3', xy=(centroids[2,0],centroids[2,1]))
ax.annotate('Cluster 4', xy=(centroids[3,0],centroids[3,1]))
```

Figure 8.19 The clustering result of agglomerative clustering

We can perform clustering by specifying the `distance_threshold` instead of specifying the number of clusters. But `n_clusters` must be None.

```
agg = AgglomerativeClustering(n_clusters=None, distance_threshold=35)
y_pred = agg.fit_predict(X)
```

We plot the clustering result using scatter plot. The scatter plot is given in Figure 8.20.

```
fig, ax = plt.subplots(figsize=(8,6))
ax.scatter(X[:,0], X[:,1], s=3, c=y_pred)
ax.scatter(centroids[:,0], centroids[:,1], s=10, c='r')
ax.annotate('Cluster 1', xy=(centroids[0,0],centroids[0,1]))
ax.annotate('Cluster 2', xy=(centroids[1,0],centroids[1,1]))
ax.annotate('Cluster 3', xy=(centroids[2,0],centroids[2,1]))
ax.annotate('Cluster 4', xy=(centroids[3,0],centroids[3,1]))
```

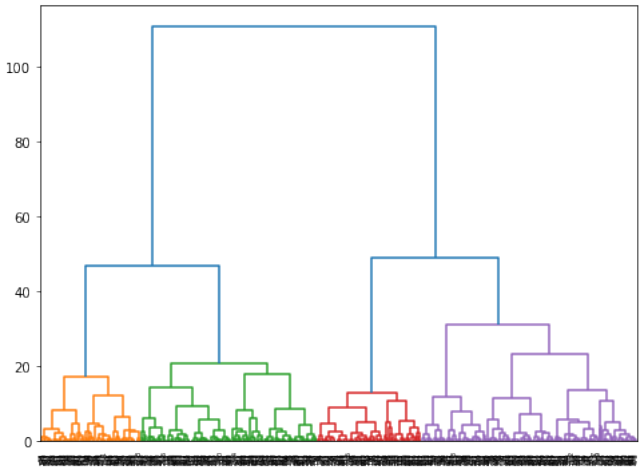


Figure 8.20 The result of clustering by specifying the linkage distance threshold

Exercises

1. Explain the steps of k-means clustering algorithm.
2. Explain the the main difference between k-means and k-nearest neighbors.
3. Explain how to determine k using the elbow method.
4. Explain dendrogram in hierarchical agglomerative clustering algorithm.
5. Explain the different parts of a dendrogram in the hierarchical agglomerative clustering algorithm.
6. Explain the different linkage methods used in the hierarchical agglomerative clustering algorithm.

Chapter 9

Dimensionality Reduction

9.1 Introduction

In some problems, the datasets that we are analyzing and modeling are of high dimensions. The high dimension refers to a large number of features that exist in the dataset which could be in the order of hundreds, thousounds or even tens of thousands. We might think that having an abundance of features is a good strategy because more feature means information for solving the problem. Unfortunately, that is not the case due to several reasons.

First, visualizing and analyzing a high-dimensional dataset to identify patterns and gain insights is difficult. For example, it is easier to visualize and analyze two-dimensional data than three-dimensional data. Beyond three dimensions, the analysis is more difficult and it is not possible to visualize the data. Second, some datasets may contain redundant features and features that are not relevant to solving the problem. These features should be removed because they do not add additional information to the prediction. Moreover, the features will incur unnecessary storage consumption and computation to the model training.

Finally, the complexity of the problem and the machine learning models depends on the number of input dimensions as well as the number of instances. Although additional features may provide more information to the prediction, the volume of the feature space grows exponentially, thus the data becomes sparse. Figure 9.1 illustrates the issues of data sparsity. As we move from one dimension to three dimensions, there is more unoccupied space than the data points. As the number of dimensions grows, the data points appear to be equidistant and it becomes more difficult to cluster. Thus, given a high-dimensional dataset, the dataset should be analyzed to reduce it to a representation with lower dimensions.

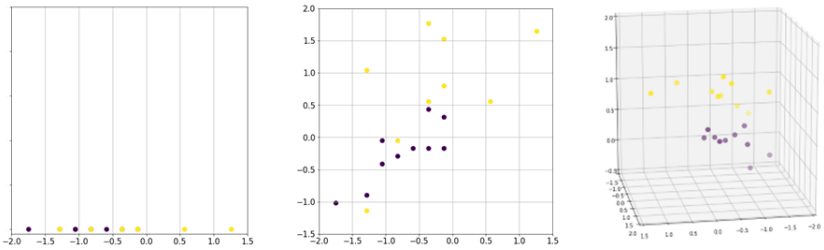


Figure 9.1 Data density reduces exponentially as we move from one dimension to three dimensions. Data density refers to the amount of data points that can be stored per unit area

9.2 Principal Component Analysis

Principal component analysis (PCA) is one of the most commonly used dimensionality reduction method. In PCA, the data in the original d-dimensional space is projected (or transformed) to k-dimensional space, where $k < d$. This is achieved by finding a hyperplane onto which to project the data. For simplicity, consider the data points in 2-d feature space as shown in Figure 9.2. The aim is to reduce the number of dimensions by projecting the data into 1-d feature space. The figure shows two possible projections directions, one is onto a line in green color and the other one is onto a line in red color. The projected data points are indicated by blue circles. It can be seen that projecting data onto the green line results in overlapping data points while the red line results in data points that are scattered far away from each other. In other words, PCA seeks to find a direction in which to project the data that maximizes the variance of the data (Pearson 1901).

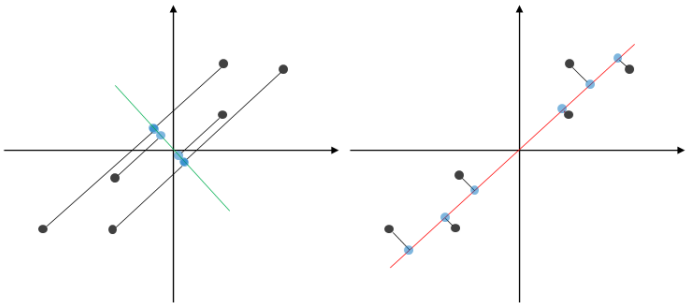


Figure 9.2 Projection direction on the left produces overlapping data points while projection direction on the right maximizes the variance of the data

Let us consider the best solution for projecting 2-d data to 1-d data. The vector that represents the best solution (line) that the data is to be projected onto is $\mathbf{w}_1$. The vector is known as the first principal component. Thus, the projection of data $\mathbf{x}$ is defined by

$$\mathbf{z} = \mathbf{w}_1^\top \mathbf{x} \tag{9.1}$$

We want the projected data to have a maximum variance. Thus, we seek to find $\mathbf{w}_1$ such that $\text{var}(\mathbf{w}_1^\top \mathbf{x})$ is maximized subject to a constraint $\|\mathbf{w}_1\| = 1$, to ensure we obtain a unique solution. We rewrite the equation as follows:

$$\text{var}(\mathbf{w}_1^\top \mathbf{x}) = \mathbf{w}_1^\top \sum \mathbf{w}_1 \tag{9.2}$$

where $\sum$ is the covariance of $\mathbf{x}$. We restate the optimization problem by applying Lagrange multipliers.

$$\max \mathbf{w}_1^\top \sum \mathbf{w}_1 - \lambda(\mathbf{w}_1^\top \mathbf{w}_1 - 1) \tag{9.3}$$

Taking the derivative with respect to $\mathbf{w}_1$ and setting it equal to zero.

$$2\mathbf{w}_1 \sum -2\lambda \mathbf{w}_1 = 0 \tag{9.4}$$

Thus, $\sum \mathbf{w}_1 = \lambda \mathbf{w}_1$. We can see that $\mathbf{w}_1$ is the eigenvector of $\sum$ and λ is the corresponding eigenvalue. Because we want to maximize $\mathbf{w}_1 \sum \mathbf{w}_1$, we choose the eigenvector with the largest eigenvalue for the maximum variance.

We can find the second vector $\mathbf{w}_2$ which is known as the second principal component. If the second

component is solved, the 2-d dataset is not reduced, but transformed into a new 2-d representation. Note that the number of components that can be solved is equal to the number of dimensions of the data.

The second principal component should also maximize the variance, subject to $\|\mathbf{w}_2\| = 1$ and $\mathbf{w}_2^\top \mathbf{w}_1 = 0$. We restate the problem by applying Lagrange multiplier.$\max \mathbf{w}_2^\top \sum \mathbf{w}_2 - \lambda(\mathbf{w}_2^\top \mathbf{w}_2 - 1) - \beta(\mathbf{w}_2^\top \mathbf{w}_1 - 0)$

Taking the derivative with respect to $\mathbf{w}_2$ and setting it equal to zero.

$$2\mathbf{w}_2 \sum -2\lambda \mathbf{w}_2 - \beta \mathbf{w}_1 = 0 \tag{9.5}$$

If we multiply the expression with $\mathbf{w}_1$

$$2\mathbf{w}_2^\top \sum \mathbf{w}_1 - 2\lambda \mathbf{w}_2^\top \mathbf{w}_1 - \beta \mathbf{w}_1^\top \mathbf{w}_1 = 0 \tag{9.6}$$

We know that $\mathbf{w}_1^\top \mathbf{w}_2 = 0$ and $\mathbf{w}_1^\top \mathbf{w}_1 = 1$. Then, the expression becomes $\beta \cdot 1 = 0$. Thus, we know $\beta = 0$.

We can reduce $2\mathbf{w}_2 \sum -2\lambda \mathbf{w}_2 - \beta \mathbf{w}_1 = 0$ to

$$\sum \mathbf{w}_2 = \lambda \mathbf{w}_2 \tag{9.7}$$

where $\mathbf{w}_2$ is the eigenvector and λ is the corresponding eigenvalue. Note that PCA is an unsupervised method in which it does not use the labels of the data. Thus, the resulting low dimesional representation may not be optimal for classification.

Solving for Eigenvectors and Eigenvalues

Suppose that

$$\sum \mathbf{w} = \lambda \mathbf{w} \tag{9.8}$$

where covariance matrix $\sum$ of data points $\mathbf{x}$ is defined by

$$\sum = \frac{B^\top B}{N-1} \tag{9.9}$$

where $B = \mathbf{x} - JJ^\top \mathbf{x}$. J is a $N \times 1$ vector of ones.

Rewrite the above expression as follows:

$$\mathbf{w}(\sum -\lambda I) = 0 \tag{9.10}$$

where I is an identity matrix

Solve $\det(\sum -\lambda I) = 0$ to obtain λ.

Substitute λ into $\mathbf{w}(\sum -\lambda I) = 0$ to obtain $\mathbf{w}$.

How do we choose the number of components k? Total variance of the data is defined as follows:

$$\sigma_T^2 = \sum_j^d \sigma_j^2 \tag{9.11}$$

where $\sigma_j^2 = \frac{1}{N-1} \sum_{i=1}^N (z_j^i - \bar{z}_j)^2$ is the variance of component j. Explained variance is the ratio of

variance of a principal component to the total variance, $\frac{\sigma_j^2}{\sigma_T^2}$. We choose k where the cumulative explained variance is between 0.95 and 0.99.

Numerical Example: Consider the following dataset:

$$\mathbf{x} = \begin{bmatrix} 4 & 2 \\ 5 & 5 \\ 2 & 3 \\ 8 & 7 \\ 11 & 5 \\ 10 & 8 \end{bmatrix}$$

The mean of $\mathbf{x}$ is

$$\mu = \begin{bmatrix} 6.67 & 5 \end{bmatrix}$$

Mean normalize $\mathbf{x} - \mu$

$$\tilde{\mathbf{x}} = \begin{bmatrix} -2.67 & -3 \\ -1.67 & 0 \\ -4.67 & -2 \\ 1.33 & 2 \\ 4.33 & 0 \\ 3.33 & 3 \end{bmatrix}$$

Calculate $\sum = \frac{B^\top B}{N-1}$ where $B = \tilde{\mathbf{x}} - JJ^\top \tilde{\mathbf{x}}$.

Note that

$$B = \begin{bmatrix} -2.67 & -3 \\ -1.67 & 0 \\ -4.67 & -2 \\ 1.33 & 2 \\ 4.33 & 0 \\ 3.33 & 3 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}^\top \begin{bmatrix} -2.67 & -3 \\ -1.67 & 0 \\ -4.67 & -2 \\ 1.33 & 2 \\ 4.33 & 0 \\ 3.33 & 3 \end{bmatrix} = \begin{bmatrix} -2.67 & -3 \\ -1.67 & 0 \\ -4.67 & -2 \\ 1.33 & 2 \\ 4.33 & 0 \\ 3.33 & 3 \end{bmatrix}$$

$$\sum = \frac{B^\top B}{6-1} = \begin{bmatrix} 12.67 & 6 \\ 6 & 5.2 \end{bmatrix}$$

Find λ by solving $\mathbf{w}(\sum - \lambda I) = 0$ where

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\mathbf{w}(\begin{bmatrix} 12.67 & 6 \\ 6 & 5.2 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}) = 0$$

$$\det(\begin{bmatrix} 12.67 - \lambda & 6 \\ 6 & 5.2 - \lambda \end{bmatrix}) = 0$$

$$(12.67 - \lambda)(5.2 - \lambda) - 6^2 = 0$$

$$\lambda^2 - 17.87\lambda + 29.88 = 0$$

$$\lambda_1 = 16.0 \text{ and } \lambda_2 = 1.867$$

λ_1 is the largest, hence the first principal component.

Substitute $\lambda_1 = 16.0$

$$\begin{bmatrix} 12.67 & 6 \\ 6 & 5.2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = 16 \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$$

$12.67w_1 + 6w_2 = 16w_1$

$6w_1 + 5.2w_2 = 16w_2$

$w_1 = \frac{6}{3.33} w_2$

Let $w_2 = 1.0$, thus $w_1 = \frac{6}{3.33} = 1.8$

$$\mathbf{w} = \begin{bmatrix} 1.8 \\ 1 \end{bmatrix}$$

The projected data $\mathbf{z} = \mathbf{w}^\top \tilde{\mathbf{x}}$

9.3 Linear Discriminant Analysis

Principal component analysis projects the data in the directions of maximum variance. However, in some cases, maximizing the variance of the data may not achieve highly discriminative features. Fisher's linear discriminant analysis (LDA) is also a projection-based method that seeks to find a vector that defines the projection direction of the data (Fisher 1936). Unlike principal component analysis which maximizes the variance of the data (Martinez and Kak 2001), linear discriminant analysis uses the labels of the data and projects the data in the direction that maximizes the separation of the classes as shown in Figure 9.3.

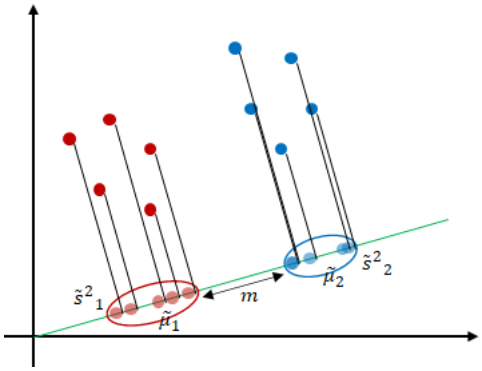


Figure 9.3 Projection direction maximizes the maximum class separation

How do we find the projection that maximizes the class separation of the data? LDA states that maximum separation is achieved when the distance between the two clusters is maximum (or far away from each other). We consider the classification of two classes first, then generalize to more than two classes.

9.3.1 Binary Class LDA

Let $\tilde{\mu}_1$ and $\tilde{\mu}_2$ denote the means of data after projection which belong to class Red and Blue respectively.

$$\tilde{\mu}_1 = \mathbf{w}^\top \mu_{\mathbf{1}} \tag{9.12}$$

$$\tilde{\mu}_2 = \mathbf{w}^\top \mu_{\mathbf{2}} \tag{9.13}$$

Furthermore, let $\tilde{s}_1^2$ and $\tilde{s}_2^2$ denote the scatters (variances) of data after projection which belong to class red and blue respectively.

$$\tilde{s}_1^2 = \sum_{x \in c_1} (\mathbf{w}^\top \mathbf{x} - \mathbf{w}^\top \mu_{\mathbf{1}})^2 \tag{9.14}$$

$$\tilde{s}_2^2 = \sum_{x \in c_2} (\mathbf{w}^\top \mathbf{x} - \mathbf{w}^\top \mu_{\mathbf{2}})^2 \tag{9.15}$$

To ensure maximum class separation, we want the means to be far away from each other and the projected data in the clusters to be close to each other. In other words, we want the mean difference $|\tilde{\mu}_1 - \tilde{\mu}_2|$ to be large and the summation of the scatter $\tilde{s}_1^2 + \tilde{s}_2^2$ to be small. Therefore, the objective is to maximize

$$J(\mathbf{w}) = \frac{|\tilde{\mu}_1 - \tilde{\mu}_2|^2}{\tilde{s}_1^2 + \tilde{s}_2^2} \tag{9.16}$$

The numerator can be expressed as follows:

$$\begin{aligned} |\tilde{\mu}_1 - \tilde{\mu}_2|^2 &= |\mathbf{w}^\top \mu_{\mathbf{1}} - \mathbf{w}^\top \mu_{\mathbf{2}}|^2 \\ &= \mathbf{w}^\top (\mu_{\mathbf{1}} - \mu_{\mathbf{2}})(\mu_{\mathbf{1}} - \mu_{\mathbf{2}})^\top \mathbf{w} \\ &= \mathbf{w}^\top s_B \mathbf{w} \end{aligned} \tag{9.17}$$

where $s_B = (\mu_{\mathbf{1}} - \mu_{\mathbf{2}})(\mu_{\mathbf{1}} - \mu_{\mathbf{2}})^\top$ is between-class scatter matrix.

The denominator can be expressed as follows:

$$\begin{aligned} \tilde{s}_k^2 &= \sum_{x \in c_k} (\mathbf{w}^\top \mathbf{x} - \mathbf{w}^\top \mu_{\mathbf{k}})^2 \\ &= \sum_{x \in c_k} (\mathbf{w}^\top \mathbf{x} - \mathbf{w}^\top \mu_{\mathbf{k}})(\mathbf{w}^\top \mathbf{x} - \mathbf{w}^\top \mu_{\mathbf{k}}) \\ &= \sum_{x \in c_k} \mathbf{w}^\top (\mathbf{x} - \mu_{\mathbf{k}})(\mathbf{x} - \mu_{\mathbf{k}})^\top \mathbf{w} \\ &= \mathbf{w}^\top \sum_{x \in c_k} (\mathbf{x} - \mu_{\mathbf{k}})(\mathbf{x} - \mu_{\mathbf{k}})^\top \mathbf{w} \\ &= \mathbf{w}^\top s_W \mathbf{w} \end{aligned} \tag{9.18}$$

where $s_W = \sum_{x \in c_k} (\mathbf{x} - \mu_{\mathbf{k}})(\mathbf{x} - \mu_{\mathbf{k}})^\top$ is within-class scatter matrix.

Therefore, the objective function is

$$J(\mathbf{w}) = \frac{\mathbf{w}^\top s_B \mathbf{w}}{\mathbf{w}^\top s_W \mathbf{w}} \tag{9.19}$$

Taking the derivative and setting it equal to zero.

$$\begin{aligned}\frac{(\mathbf{w}^\top s_W \mathbf{w})2s_B\mathbf{w} - (\mathbf{w}^\top s_B \mathbf{w})2s_W\mathbf{w}}{2\mathbf{w}s_W\mathbf{w}} &= 0 \\ s_B\mathbf{w} - \frac{\mathbf{w}^\top s_B\mathbf{w}}{\mathbf{w}^\top s_W\mathbf{w}} s_W\mathbf{w} &= 0 \\ s_W^{-1}s_B\mathbf{w} &= J(\mathbf{w})\mathbf{w}\end{aligned} \tag{9.20}$$

Given that $J(\mathbf{w})$ is a scalar, the expression becomes

$$s_W^{-1}s_B\mathbf{w} = \lambda\mathbf{w} \tag{9.21}$$

where $\mathbf{w}$ is the eigenvector of $s_W^{-1}s_B$ and λ is the corresponding eigenvalue.

9.3.2 Multi-class LDA

When there are more than two classes, the between-class scatter matrix is defined as follows:

$$s_B = \sum_{k=1}^{C} N_k(\mu_k - \mu)(\mu_k - \mu) \tag{9.22}$$

where $\mu_k = \frac{1}{N_k}\sum_{\mathbf{x}\in c_k}\mathbf{x}$ and $\mu = \frac{1}{N}\sum_{\forall\mathbf{x}}\mathbf{x}$ is the global (overall) mean.

The within-class scatter matrix is defined as follows:

$$s_W = \sum_{k=1}^{C} s_k^2 \tag{9.23}$$

where $s_k^2 = \sum_{x\in c_k}(\mathbf{x} - \mu_k)(\mathbf{x} - \mu_k)$. The number of components is defined as follows:

$$k = \min(C - 1, d) \tag{9.24}$$

where C is the number of classes.

Solving for Eigenvectors and Eigenvalues

Compute s_B and s_W.

Compute s_W^{-1}.

$\mathbf{w}(s_W^{-1}s_B - \lambda I) = 0$ where I is an identity matrix.

Solve $\det(s_W^{-1}s_B - \lambda I) = 0$ to obtain λ.

Substitute λ into $\mathbf{w}(s_W^{-1}s_B - \lambda I) = 0$ to obtain $\mathbf{w}$.

Numerical Example: Consider the following dataset:

$$\mathbf{x} = \begin{bmatrix} 5 & 2 \\ 6 & 5 \\ 7 & 3 \\ 3 & 9 \\ 5 & 11 \\ 6 & 9 \end{bmatrix}$$

$\mu_1 = \begin{bmatrix} 6 & 3 \end{bmatrix}$

$\mu_2 = \begin{bmatrix} 4.667 & 9.667 \end{bmatrix}$

$s_1^2 = \begin{bmatrix} 1 & 0.5 \\ 0.5 & 2.333 \end{bmatrix}$

$s_2^2 = \begin{bmatrix} 2.333 & 0.333 \\ 0.333 & 1.333 \end{bmatrix}$

$s_B = (\mu_1 - mu_2)^\top (mu_1 - mu_2) = \begin{bmatrix} 1.778 & -8.444 \\ -8.444 & 40.111 \end{bmatrix}$

$s_W = s_1^2 + s_2^2 = \begin{bmatrix} 3.333 & 0.8333 \\ 0.8333 & 3.666 \end{bmatrix}$

$s_W^{-1} = \begin{bmatrix} 0.318 & 0.072 \\ 0.072 & 0.289 \end{bmatrix}$

$s_W^{-1} s_B = \begin{bmatrix} 1.176 & -5.586 \\ -2.570 & 12.209 \end{bmatrix}$

Find λ by solving $\mathbf{w}(s_W^{-1} s_B - \lambda I) = 0$ where I is an identity matrix.

$\mathbf{w}(\begin{bmatrix} 1.176 & -5.586 \\ -2.570 & 12.209 \end{bmatrix} - \lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix})$

$\det(\begin{bmatrix} 1.176 - \lambda & -5.586 \\ -2.570 & 12.209 - \lambda \end{bmatrix}) = 0$

$(1.176 - \lambda)(12.209 - \lambda) - 14.356 = 0$

$\lambda^2 - 13.385\lambda + 0.002 = 0$

$\lambda_1 = 13.384$ and $\lambda_2 = 0.0015$

Substitute $\lambda_1 = 13.384$

$\begin{bmatrix} 1.176 & -5.586 \\ -2.570 & 12.209 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = 13.384 \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$

$1.176w_1 - 5.586w_2 = 13.384w_1$

$-2.57w_1 + 12.209w_2 = 13.384w_2$

$w1 = -\frac{5.586}{12.208} w2$

Let $w2 = 1$, thus $w1 = -\frac{5.586}{12.208} = -0.456$

$\mathbf{w} = \begin{bmatrix} -0.456 \\ 1 \end{bmatrix}$

9.4 Feature Subset Selection

Subset selection is an approach of selecting the set of features that is most relevant to predicting the target variable. Specifically, given a dataset with d dimensions (features), the aim is to find k

features that provide the most information to the prediction and discard the other $(d-k)$ features. There are 2^d possible subsets of d features and ideally, we want to evaluate all subsets to determine the optimal one. However, it is computationally infeasible especially when d is large. Thus, heuristic methods are employed to obtain a satisfactory solution in an acceptable time. The approach is categorized into two based on the choice of evaluation metrics: filter method and wrapper method.

9.4.1 Wrapper Method

Wrapper methods utilize a machine learning model to evaluate the feature subsets (Kohavi and George H. John 1997). Each subset is used to train the model which is then evaluated on a hold-out set. The misclassification rate of the hold-out set represents the score of the subset. Wrapper method is computationally intensive because a new model is trained for each subset. However, the selected subset contains the most relevant features for that particular machine learning model and problem. Wrapper method can be classified into two types: sequential forward selection and sequential backward selection.

In sequential forward selection, we start with an empty set of features (no features), and add one feature into the subset one by one. The feature to be added is the feature with the minimum error. The process of adding a feature into the subset stops when the error rate is not decreasing or the decrease is very minimal. The algorithm of sequential forward selection is given in Algorithm 6.

Algorithm 6 Sequential Forward Selection Algorithm

Training set with d dimensions, $\mathbf{x} = x_1, x_2, ..., x_d$
$F = \emptyset$
k is the number of selected features
Split training set into training D_{tr} and validation subsets D_{vd}
for 1 to k **do**
    for x_j in $\mathbf{x} \in D_{tr}$ **do**
        Train a predictive model $\hat{f}$ using $F \cup x_j$
        Compute the error rate $\epsilon_{\hat{f}}$ on D_{vd}
    end for
    Choose x_j with minimum $\epsilon_{\hat{f}}(F \cup x_j)$
    add x_j into F
end for

In sequential backward selection, we start with all features, and remove one feature from the subset one by one. The feature to be removed is the feature that results in minimum error rate upon removal. The process of removing a feature from the subset stops when the error rate is not decreasing or the decrease is very minimal. The algorithm of sequential backward selection is given in Algorithm 7.

9.4.2 Filter Method

Filter methods utilize various statistical tests to determine the relevance of the features instead of the error rate of a machine learning model (I. Guyon and Elisseeff 2003). Some of the common statistical measures are analysis of variance (ANOVA), mutual information and chi-square. Filter methods rank the features according to their relevance. The top k features are chosen via cross-validation. Because filter methods do not rely on a machine learning model, filter methods are less computationally intensive. However, the selected set of features is generally less optimal and gives lower prediction performance. Thus, filter methods are preferable for uncovering the relationship

Algorithm 7 Sequential Backward Selection Algorithm

Training set with d dimensions, $\mathbf{x} = x_1, x_2, ..., x_d$
$F = \mathbf{x}$
k is the number of selected features
Split training set into training D_{tr} and validation subsets D_{vd}
for 1 to $(d - k)$ **do**
    for x_j in $\mathbf{x} \in D_{tr}$ **do**
        Train a predictive model $\hat{f}$ using $F - x_j$
        Compute the error rate $\epsilon_{\hat{f}}$ on D_{vd}
    end for
    Choose x_j with minimum $\epsilon_{\hat{f}}(F - x_j)$
    x_j from F
end for

between the features and the target variable, and can be used as preprocessing step for wrapper methods.

9.5 Implementing Feature Reduction

In this section, we will be implementing PCA and LDA using Scikit-learn. First, we use the simple iris dataset as shown in Figure 9.4. Then, we will use a more challenging dataset.

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_iris
dataset = load_iris()
X = dataset.data
y = dataset.target
```

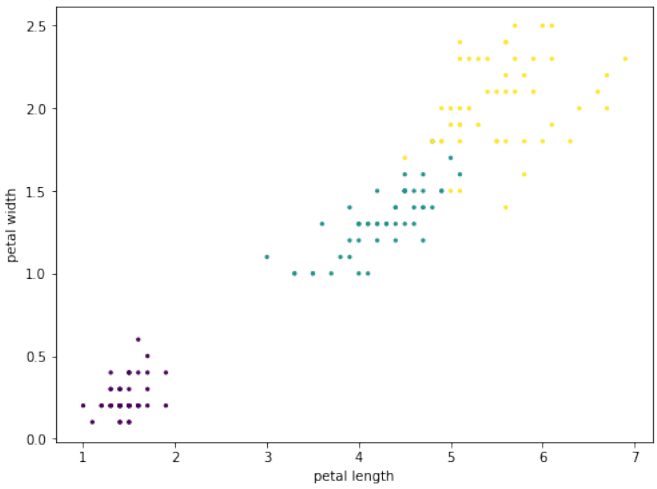


Figure 9.4 The scatter plot of iris dataset

Now, let us reduce the number of dimension using PCA. We are going to import `PCA` module from Scikit-Learn and then create an object of `PCA` to transform the data. We will be using attribute petal length and petal width only, which means we are reducing from 2-d to 1-d. We import `PCA` from `sklearn.decomposition`. We set parameter `n_components` to 1. We can specify `None` to keep all components.

```
from sklearn.decomposition import PCA
pca = PCA(n_components=1)
X_pca = pca.fit_transform(X[:,2:4])
```

We create a dummy data (y axis) for plotting the transformed 1-d data. As we can see, the reduced data is easier to analyze as shown in Figure 9.5

```
import numpy as np
y_dummy = np.zeros(len(X_pca))
plt.scatter(X_pca, y_dummy, c=y)
plt.xlabel('PCA')
```

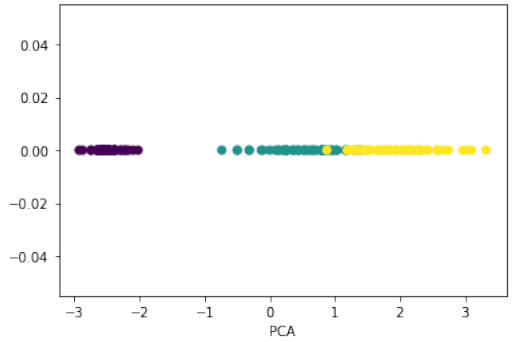


Figure 9.5 The reduced 1-d dataset

Now, let us use all the attributes in the PCA transformation. The transformed dataset is shown in Figure 9.6.

```
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)
plt.scatter(X_pca[:,0], X_pca[:,1], c=y)
plt.xlabel('PCA1')
plt.ylabel('PCA2')
```

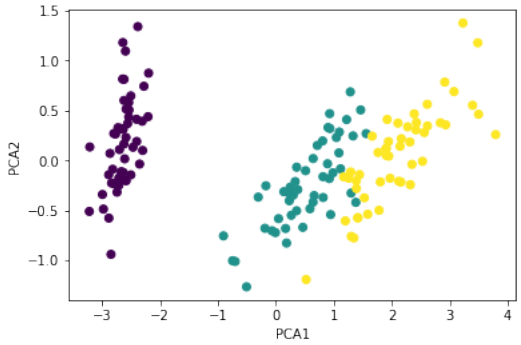


Figure 9.6 The dataset after transformation using PCA

As you can see, the data transformation is not better than the one in Figure 9.5. Thus, using all the attributes in feature reduction will not necessarily lead to a better transformation. The second PCA component is perpendicular to the first axis. Note that the second axis is in the direction that has less amount of variation.

Now, let us use a bigger dataset, the Quantitative Structure Activity Relationship (QSAR) biodegradation dataset. The dataset has been used to develop QSAR models for the study of the relationships between chemical structure and biodegradation of molecules. The dataset contains biodegradation experimental values of 1055 chemicals whereby 356 are discriminate ready and 699 are not ready biodegradable molecules. Thus, the prediction task is to classify the data into ready biodegradable and not ready biodegradable.

```
import pandas as pd
import wget
from zipfile import ZipFile

data_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/
00254/biodeg.csv'
wget.download(data_url)
```

Load the dataset file.

```
data_df = pd.read_csv('biodeg.csv', sep=';', header=None)
dataset = data_df.values
y = dataset[:,-1]
X = dataset[:,:-1]
print(X.shape)
```

We split the dataset into training and test sets.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

We create a function to print out the accuracy score, confusion matrix and the classification report.

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

def print_clf_evaluation(y_test, y_pred):
    print(accuracy_score(y_test, y_pred))
    print(confusion_matrix(y_test, y_pred))
    print(classification_report(y_test, y_pred))
```

The dataset contains 41 features. We use PCA to reduce to 2 dimensions, model the transformed data using SVM algorithm and evaluate the classifier.

```
from sklearn.pipeline import make_pipeline
from sklearn.svm import SVC
svm_pipeline = make_pipeline(PCA(n_components=2), SVC(C=1, kernel='linear'))
svm_pipeline.fit(X_train, y_train)
```

We evaluate the classifier on the test set.

```
y_pred = svm_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.7066246056782335
[[224   0]
 [ 93   0]]
              precision    recall  f1-score   support

         NRB       0.71      1.00      0.83       224
          RB       0.00      0.00      0.00        93

    accuracy                           0.71       317
   macro avg       0.35      0.50      0.41       317
weighted avg       0.50      0.71      0.59       317
```

As we can see the classifier performed badly whereby class `RB` has zero precision, recall and F-score. This shows that reducing the number of dimensions to 2 results in the classifier does not have sufficient information to correctly classify the data.

Let us determine the optimal number of components. We use the explained variance which provides us the amount of information or variance each principal component holds after projecting the data to a lower dimensional subspace. This can be determined by examining the cumulative explained variance ratio as a function of the number of components.

```
pca = PCA()
X_train_pca = pca.fit_transform(X_train)
```

The explained variance ratio can be obtained using `explained_variance_ratio_`. The value is for each PCA component and sum to 1.

```
print(pca.explained_variance_ratio_)
```

```
[4.62673531e-01 2.70480635e-01 1.07755746e-01 4.36559974e-02
 3.02220543e-02 2.47064864e-02 1.45692737e-02 1.02310273e-02
 7.95011626e-03 6.45758082e-03 3.99561145e-03 3.21950035e-03
 2.55745329e-03 1.96021711e-03 1.81857554e-03 1.47945721e-03
 1.33585287e-03 1.12085852e-03 7.74831761e-04 4.82960444e-04
 4.40409529e-04 4.06638441e-04 3.00948628e-04 2.84178322e-04
 2.29227663e-04 2.00620276e-04 1.50112112e-04 1.12861429e-04
 8.34773811e-05 8.05686901e-05 7.65556758e-05 6.41364716e-05
 4.98202136e-05 2.83239031e-05 1.60472404e-05 1.18525775e-05
 8.79044658e-06 4.80627240e-06 2.27209676e-06 4.78156095e-07
 1.07217038e-07]
```

Plot the explained variance ratio against PCA components. The plot is given in Figure 9.7.

```
x_axes = np.arange(1,42,1)
y_axes = np.cumsum(pca.explained_variance_ratio_)
plt.figure(figsize=(8,6))
plt.plot(x_axes, y_axes, marker='.')
plt.xticks(np.arange(1,X_train_pca.shape[1],2))
plt.show()
```

This curve quantifies how much of the total 30-dimensional variance is contained within the first d

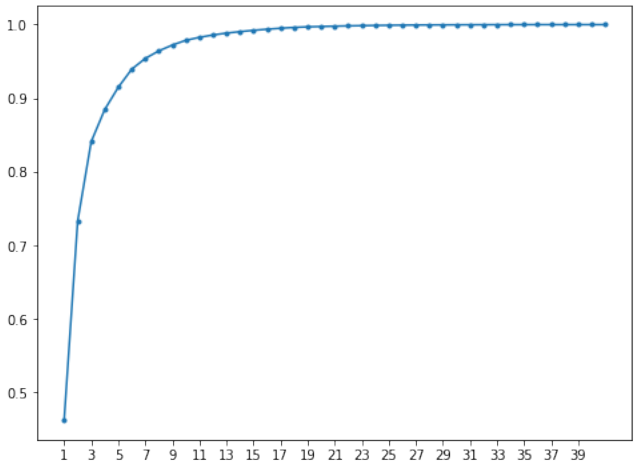


Figure 9.7 The graph of explained variance ratio against PCA components

components. For example, we see that the first 15 components contain approximately 99.0% of the variance. Let us calculate the sum of the first 15 explained variance ratio. We can see that the sum equals 0.992 (99.2%).

```
print(np.sum(pca.explained_variance_ratio_[:15]))
```

```
0.9922538064547924
```

We set `n_components` to 15, rebuild the classifier and evaluate the classifier.

```
svm_pipeline = make_pipeline(PCA(n_components=15), SVC(C=1, kernel='linear'))
svm_pipeline.fit(X_train, y_train)

y_pred = svm_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.8359621451104101
[[194  30]
 [ 22  71]]
              precision    recall  f1-score   support

         NRB       0.90      0.87      0.88       224
          RB       0.70      0.76      0.73        93

    accuracy                           0.84       317
   macro avg       0.80      0.81      0.81       317
weighted avg       0.84      0.84      0.84       317
```

The performance evaluation shows a significant increase in precision, recall and F-score for class `RB`. The classification accuracy increases from 0.71 to 0.84.

Now, let us implement LDA to reduce the dataset. We import `LinearDiscriminantAnalysis` module from Scikit-Learn. By default, parameter `n_components` is `None` which means the number of components will be determined automatically based on the number of classes and features.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
svm_pipeline = make_pipeline(LinearDiscriminantAnalysis(),
                             SVC(C=1, kernel='linear'))
svm_pipeline.fit(X_train, y_train)
```

We evaluate the classifier on the test set.

```
y_pred = svm_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.8422712933753943
[[196  28]
 [ 22  71]]
              precision    recall  f1-score   support

         NRB       0.90      0.88      0.89       224
          RB       0.72      0.76      0.74        93

    accuracy                           0.84       317
   macro avg       0.81      0.82      0.81       317
weighted avg       0.85      0.84      0.84       317
```

The classification report shows a similar performance to PCA. It can be observed that LDA achieved a slight increase in F-score measure of 0.01 for class `NRB`. The accuracy of classifier is 0.84.

9.6 Implementing Feature Selection

In this section, we will be implementing the filter and wrapper-based feature selection methods using Scikit-Learn. The filter method is implemented using a module called `SelectKBest`. `SelectKBest` takes as input a scoring function that returns univariate scores. Several scoring functions are provided as follows. We can choose either '`chi2`' (chi-squared), '`f_classif`' (ANOVA) or '`mutual_info_classif`' (mutual information) for classification problem. For regression, we choose either '`f_regression`' (ANOVA) or '`mutual_info_regression`' (mutual information).

Let us import the module and perform feature selection on the QSAR biodegradation dataset. `SelectKBest` has two parameters. The first parameter is `score_func` which is used to define scoring function. The default value is '`f_classif`'. The second parameter is `k` which is the number of top features to select, and by default, the value is 10. We can specify '`all`' to bypass selection.

```
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2, f_classif, mutual_info_classif
from sklearn.pipeline import Pipeline
scaler = StandardScaler()
kBest = SelectKBest(k='all')
steps = [('scaler', scaler), ('kbest', kBest)]
pipeline = Pipeline(steps=steps)
pipeline.fit(X_train, y_train)
```

To display the scores for each feature, use `scores_` attribute. In this case, the scores are the F value of the features.

```
feat_scores = pipeline['kbest'].scores_
print(feat_scores)
```

```
[1.30991715e+02 3.89631092e-02 7.10330087e+01 9.70062048e+00
 4.22370118e+01 2.33678971e+01 1.00033571e+02 3.06790064e+01
 2.61849128e+00 3.73984059e+01 4.51887934e+01 1.30413811e+01
 8.56742854e+01 6.84483016e+01 9.11356266e+01 5.92147912e-01
 4.83909175e+00 1.14360265e+01 2.79029003e+00 2.14710565e+01
 8.09777954e+00 1.21620052e+02 7.75173614e+00 9.84029627e+00
 5.33862419e+01 7.97725101e+00 1.28908859e+02 4.31420826e-02
 4.85715857e+00 5.46205592e+00 2.76568765e+01 1.22794490e+01
 8.73141888e+01 5.84585505e+01 4.12664615e-03 6.74488465e+01
 1.30664824e+01 5.62535568e+01 1.09495022e+02 2.21362891e+01
 3.76767712e+01]
```

We use the F value to select the number of features. In this example, we select features whose F value greater than 50. We can also use their p value to determine the relevant features by using `pvalues_` attribute.

```
feat_scores_sort = np.sort(feat_scores)
feat_scores_sort > 50
```

```
array([False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
        True,  True,  True,  True,  True,  True,  True,  True,  True,
        True,  True,  True,  True,  True])
```

There are 14 features. We rerun `SelectKBest` with `k` equal to 14.

```
kBest = SelectKBest(k=14)
steps = [('scaler', scaler), ('kbest', kBest)]
pipeline = Pipeline(steps=steps)
pipeline.fit(X_train, y_train)
```

We get the selected features using `get_support`.

```
pipeline['kbest'].get_support(indices=True)
```

```
[ 0  2  6 12 13 14 21 24 26 32 33 35 37 38]
```

Let us create a pipeline that select the features and create the classifier.

```
svm_pipeline = make_pipeline(StandardScaler(), SelectKBest(k=14),
                             SVC(C=1, kernel='linear'))
svm_pipeline.fit(X_train, y_train)
```

We evaluate the classifier on the test set.

```
y_pred = svm_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.8675078864353313
[[204  20]
 [ 22  71]]
              precision    recall  f1-score   support

         NRB       0.90      0.91      0.91       224
          RB       0.78      0.76      0.77        93

    accuracy                           0.87       317
   macro avg       0.84      0.84      0.84       317
weighted avg       0.87      0.87      0.87       317
```

Let us implement the wrapper-based feature selection method. The wrapper-based feature selection method is implemented using a module called `SequentialFeatureSelector`. The parameters of this module are:

- `estimator` is the machine learning model to use for feature selection.
- `n_features_to_select` is the number of features to select. This parameter accepts an integer value or a float value. If float, it is the fraction of the total number of features.
- `direction` is for specifying '`forward`' (forward selection) or '`backward`' (backward selection). The default is '`forward`'.
- `cv` is the number of folds for the cross-validation. This parameter accepts integer value or `None`. If `None`, cv equal to 5. The default is `None`.
- `n_jobs` is number of jobs to run in parallel. The default is `None`, which means `n_jobs` equal to 1.

First, we need to create a machine learning model. Then, the model is passed to `SequentialFeatureSelector`. We select 14 features and 3 folds cross-validation. `n_jobs` is set to 2 to speed up the computation.

```
from sklearn.feature_selection import SequentialFeatureSelector
scaler = StandardScaler()
svm = SVC(C=1, kernel='linear')
sfs = SequentialFeatureSelector(svm, n_features_to_select=14,
                                direction='backward', n_jobs=2)
steps = [('scaler', scaler), ('sfs', sfs)]
pipeline = Pipeline(steps=steps)
pipeline.fit(X_train, y_train)
```

We call `get_support` to get the selected features.

```
print(pipeline['sfs'].get_support(indices=True))
```

```
[ 1,  2,  3,  5,  6,  7, 12, 13, 19, 29, 31, 34, 36, 37]
```

We create a pipeline to select the features and train the model.

```
svm_pipeline = make_pipeline(scaler, sfs, svm)
svm_pipeline.fit(X_train, y_train)
```

We evaluate the classifier on the test set.

```
y_pred = svm_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.8391167192429022
[[198  26]
 [ 25  68]]
              precision    recall  f1-score   support

         NRB       0.89      0.88      0.89       224
          RB       0.72      0.73      0.73        93

    accuracy                           0.84       317
   macro avg       0.81      0.81      0.81       317
weighted avg       0.84      0.84      0.84       317
```

Exercises

1. Explain how the first principal component axis is selected in PCA.
2. Explain how the maximum separation is determined in LDA.
3. Explain the difference between PCA and LDA.
4. Explain how LDA is used in multi-class classification settings.
5. Explain the difference between forward feature selection and backward feature selection.
6. Given a two-dimensional dataset, we want to represent the data in only one dimension. There are two methods available: PCA and LDA. State which method to be used to reduce the dataset. Justify your answers.

$$\mathbf{x} = \begin{bmatrix} 3.5 & 3 \\ 5.5 & 4 \\ 4 & 4 \\ 5 & 5 \\ 6. & 4.5 \\ 5 & 3.5 \end{bmatrix}$$

Chapter 10

Ensemble Learning

10.1 Introduction

Given a problem, the usual approach to solving the problem is to build several predictive models using different machine learning algorithms. When building the predictive models, we carefully choose the parameters of the models using the validation set and then evaluate and compare their performance on the test set to determine the best model. This is due to the fact that there is no single algorithm that will always give us the best model for every problem. Building an optimal predictive model with finite data is a challenging task. Machine learning algorithms come with different sets of assumptions that induce certain models. If the assumption about the data is wrong, it leads to underfitting and high bias error.

One can fine-tune the parameters to get a model with the highest validation accuracy. However, finding the optimal parameters is an arduous task. In many cases, even after considerable fine-tuning, the model is still not good enough and fails on certain instances. Furthermore, in some applications such as medical and healthcare, datasets are not easy to obtain and the number of instances is typically limited. This further exacerbates the problem and the task of developing a reliable predictive model becomes somewhat infeasible. Evidently, a single model is not sufficient and there is a need for algorithms that can combine multiple models to produce better predictive models. Ensemble learning is an approach to combining the decisions of multiple models when predicting new instances (Opitz and Maclin 1999). The idea is that, if a model incorrectly predicts an instance, there may be another model that is able to get it right. Essentially, we want to leverage the strength of multiple models with different characteristics such that the models induce different outcomes so that they can complement each other.

10.2 Ensemble Diversity

An ensemble model supposes to perform better than any single model, that is, it should have a lower classification or regression error. We might think that a good ensemble model can be simply obtained by combining several accurate individual models. However, model accuracy is not the only consideration when building an ensemble model. More importantly, when we are building an ensemble model, the individual models or known as base models are diverse in the sense that they are making different predictions (Sollich and Krogh 1995). Otherwise, there would be no performance improvement even if the ensemble model is a combination of a large number of accurate models. Building ensemble models is not an easy task. One simple approach to ensemble learning

is to train multiple base models and assign a weightage to each model to indicate its importance. But this approach normally does not work because the models are trained for the same objective. The task becomes even more challenging when the same training set is used to train the models which could make them highly correlated. Furthermore, at the same time, we need to ensure the performance of the models must not very poor. Otherwise, the ensemble model's performance would not improve or worsen.

Ensemble diversity refers to the difference among the individual models that make up the ensemble model. Several approaches have been used to achieve the goal of generating diverse models. The first approach is to use different learning algorithms to train the base models. Different learning algorithms have different inductive biases. If a base model fails to predict certain instances, other models might get them right due to their different assumptions about the data, which improves the ensemble performance. For example, one base model may be parametric (e.g. logistic regression) and others are non-parametric models (e.g. decision tree and non-linear support vector machine).

We can also build diverse base models using the same learning algorithm but with different parameters. For example, we can build different base models by varying the kernel functions in support vector machine, the number of nearest neighbors in k-nearest neighbors the tree depth and the minimum number of instances in a leaf node in decision tree (Liu, Ting, and Fan 2005). If the models are trained using optimization techniques such as gradient descent in linear regression and neural network (this algorithm is discussed in a later chapter), the initial weights, learning rate and batch size are hyperparameters that can be varied to build diverse models (Kolen and Pollack 1990). Another approach to building diverse base models is sampling manipulation whereby the base models are trained using different training subsets. The subsets can be created using bootstrapping (Efron and Tibshirani 1994), a sampling method with replacement whereby instances are randomly drawn from the training set. Base models that are trained with different subsets are usually diverse. This ensemble learning is known as bagging which will be discussed later. The base models can also be trained sequentially whereby for each iteration (training), instances that are incorrectly predicted are given more emphasis in training subsequent base models. This ensemble learning is known as boosting.

10.3 Combination Methods

Given a set of base models, there are numerous methods to combine the predictions of the models into a single prediction. The commonly used methods are voting, linear combination and stacking.

Voting is used to combine base models that output class labels, thus it is applicable for classification ensembles. Voting combines the classification of base models by majority voting. Each base model outputs a particular class label and the class label with the most votes is chosen as the ensemble output. The voting ensemble is defined as follows:

$$\hat{y} = \texttt{mode}(h_1(\mathbf{x}), h_1(\mathbf{x}), ..., h_M(\mathbf{x})) \tag{10.1}$$

where $h_m(\mathbf{x}) \in H$ is the set of base models. Linear combination is used to combine real-valued outputs such as class probabilities, thus it is applicable to both classification and regression ensembles. In linear combination method, each base model is assigned a weightage which implies the importance of the model. The combined probability of class c of the models is defined as follows:

$$\hat{p}(y = c|\mathbf{x}) = \sum_{m=1}^{M} w_m p_m(y = c|\mathbf{x}) \tag{10.2}$$

where w_m is the weightage of m-th base model and $w_m \geq 0$ and $\sum_{m=1}^{M} w_m = 1$. If the weights are equal $w_m = \frac{1}{M}$, this is a simple averaging. However, in practice, the base models would have different performances, hence the base models would have different weights, and finding the optimal weights manually is difficult if not infeasible (Zhou 2012). One possible way to solve for w is to assess the accuracies of the base models on the validation set.

Stacking or staked generalization is a method that involves the use of machine learning algorithm to train a predictive model to combine the predictions of the base models (D. H. Wolpert 1992). The predictive model is referred to as the meta model. Specifically, the predictions of the base models serve as inputs to the meta model which is trained to optimally combine the predictions. In other words, the meta model learns the errors and corrects the biases of the base models. Figure 10.1 shows the stacking combination method whereby a meta model h accepts the predictions of M base models to produce the prediction. The meta model can be built using any machine learning algorithm. But in practice, logistic regression and linear regression are often used as the meta model for classification and regression problems respectively.

The training is done in two phases. First, the training set is divided into two subsets, one is for training the base models and the other one is used to train the meta model. Using the first subset, the base models are trained and the base models should be as different as possible so that they will make different errors. The second training phase uses the second subset to train the meta model. The second subset is fed to the base model to produce a set of predictions which will be used as the training input for the meta model. Then, the ensemble stacking model is evaluated on the test set by feeding the data to the base models, collecting the predictions of the base models and feeding the predictions to the meta model to get the final predictions.

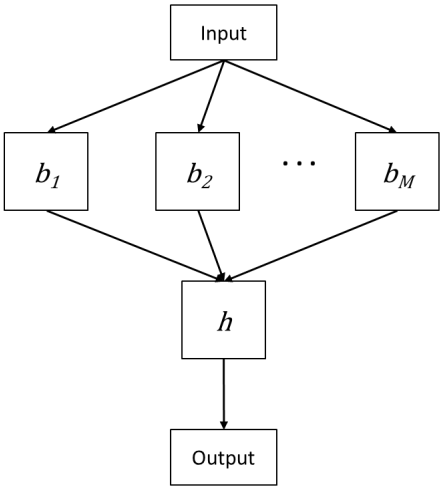


Figure 10.1 The meta model combines the predictions of the base models

10.4 Bagging

Bagging, short for bootstrapping aggregating, is a method to build a set of diverse base models by training them on different training subsets (Breiman 1996). As the name implies, the training subsets are created by bootstrapping. Bootstrapping is a sampling method with replacement whereby instances are randomly drawn from the training set. When sampling is performed with replacement, each subset is independent of other subsets. Also, it allows some instances to appear more than once in the subsets and others may not appear at all. Thus, the subsets are different from one

another, although they are drawn from the same training set. Building M base models, we need M subsets. For each subset, given a training set of size N, we randomly draw N_s instances from the training set with replacement. It is worth noting that the size of the subsets may be smaller ($N_s < N$) if the training set is large. Then, the base models are trained with these M subsets. The final prediction is obtained by majority voting or simple averaging.

In the case of regression, the average of all the predictions is taken as the final prediction. Figure 10.2 shows how bootstrapping is used to build a bagging model. As shown in the figure, an instance may appear more than one time in the subset due to sampling with replacement. For example, instance 2 and 7 appear twice in subset 1. The M base models are trained on the bootstrap samples, and their predictions are aggregated by averaging or voting. Note that bagging is a method that is independent of any machine learning algorithm. In other words, we can use any algorithm to build the M base models. The bagging algorithm is given Algorithm 8.

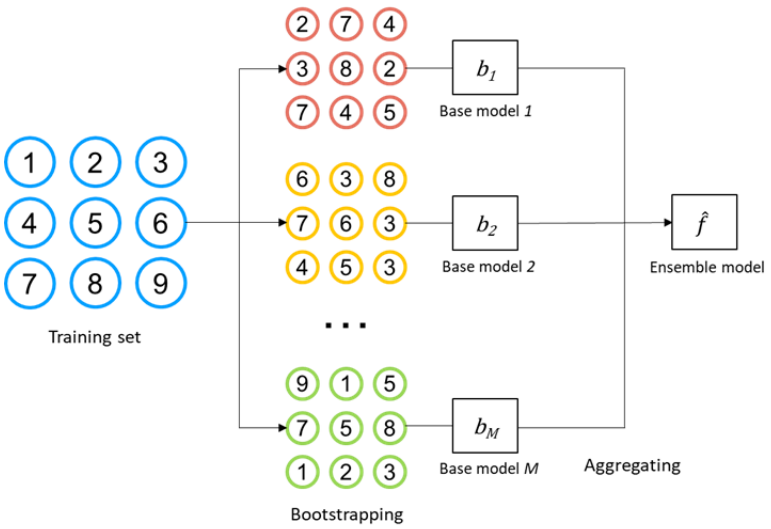


Figure 10.2 The bootstrap samples are used to build the base models which are then aggregated to produce the final prediction

Algorithm 8 Bagging Algorithm

1: **for** $m = 1$ to M **do**
2: Randomly sample training subset D_m from D_{tr} (bootstrapping)
3: Train a base model b_m using D_m
4: **end for**

Bagging exploits many base models to outperform a single model and those base models are built by training on overlapping subsets. This has the effect of reducing the variance while retaining the bias. Let us see why variance error can be reduced with bagging. The bias-variance decomposition shows that the prediction can be decomposed into bias, variance and irreducible errors where the variance error is defined as follows:

$$\texttt{var} = E[(E[\hat{f}(\mathbf{x})] - \hat{f}(\mathbf{x}))^2] \tag{10.3}$$

From equation 10.3, we can see that variance error will be large if the outputs of the predictive models are far away from the expected predicted value. The final prediction of a bagging model

which consists of M base models is defined as follows:

$$\hat{f}(\mathbf{x}) = \frac{1}{M}\sum_{m=1}^{M}\hat{f}_m(\mathbf{x}) \tag{10.4}$$

Since the final prediction is the average predicted value of M base models and if the number of base models is large ($M \to \infty$), then the output of the bagging model will be close to the expected predicted value ($\hat{f}(\mathbf{x}) \sim E[\hat{f}(\mathbf{x})]$)) which in turn reduces the variance error. It is worth noting that a good reduction of variance error can be achieved if the base models are uncorrelated (diverse base models) or in other words, the base models make different errors. However, in practice, the base models could be correlated due to the generated training subsets are not significantly different. Hence, the base models will make the same errors and the variance error is not reduced.

Another advantage of bagging is that it provides an estimate of the test error without performing cross-validation or holding out a test set. When bootstrapping is performed, two independent sets are created whereby one set consists of instances that are chosen by sampling with replacement. The other set is all the instances that are not chosen in the sampling process which is known as out-of-bag set. Since this out-of-bag instances are not used to train the base model, the instances can be used to obtain the test error of the model. If we do the same for all base models and compute their test errors, we obtain an estimate of the test error of the ensemble model. Formally, let S_m denotes the out-of-bag set of base model m. The out-of-bag error of the model is

$$\epsilon_{OOB} = \frac{1}{|S_m|}\sum_{(\mathbf{x},y)\in S_m} l(y, h(\mathbf{x})) \tag{10.5}$$

where l is a loss function such as zero-one loss. The out-of-bag error is an estimate of test error since the models were not trained on the instances. If M is large enough, the out-of-bag error will stabilize and converge to the cross-validation error.

10.5 Random Forest

Random forest is an extension to the bagging which combines bagging and random feature selection to build a collection of diverse decision trees (Breiman 2001). Since random forest is based on bagging, each decision tree is trained on a randomly sampled subset (Ho 1995). Thus, it has all the advantages of bagging method such as the out-of-bag error and reduction of variance. But unlike bagging, during the tree building phase, only a subset of features is considered when choosing the best split-point for each decision node. The idea is to build diverse and uncorrelated to ensure optimal variance reduction can be achieved. Due to the feature randomness, the trees will be different and make different errors which will be averaged out during aggregation.

The algorithm of random forest is given as follows. For each tree, a subset is randomly sampled using bootstrapping. When building the tree using the subset, only a subset of features is used to determine the best split-point to define the decision nodes. Specifically, k features are randomly chosen out of d features ($k < d$). Each selected feature is evaluated to determine the best-split point and the best one is chosen to define the decision node. These steps are recursively repeated until the predefined criteria are met e.g. maximum depth of the tree and the maximum number of leaf nodes.

Random forest has a built-in feature importance which can be obtained from the random forest construction. For each decision node in a tree, the feature with the best split-point is chosen based on some impurity functions. The impurity function can be gini or entropy for classification and sum

of squared error for regression. The feature with the highest decrease of impurity is chosen for the decision node. The feature importance can be calculated as the average of impurity decrease over all decision trees in the ensemble random forest model. The algorithm of random forest is given by Algorithm 9.

Algorithm 9 Random Forest Algorithm

1: **for** $m = 1$ to M **do**
2: Randomly sample training subset D_m from D_{tr}
3: Build decision tree h_m by repeating the following steps for each decision node
4: Randomly choose k features from d features
5: Choose the best split-point among k features
6: Define the decision node using the best split-point
7: **end for**

10.6 Boosting

Boosting is an ensemble learning algorithm that combines many weak base models to produce a strong model. A weak model is a model that performs slightly better than random guessing. Thus, a weak model has an error rate that is slightly below 50%. The model could be any machine learning model such as a decision tree with a single split which is also known as decision stump. Unlike bagging models which consist of independent base models, boosting method sequentially builds the base models in which the models are trained based on the predictions of the previous models. We describe boosting in general first. The following sections present two popular boosting algorithms, adaptive boosting and gradient boosting.

Generally, a boosting model takes the form

$$\hat{f}(\mathbf{x}) = \sum_{m=1}^{M} \alpha_m b(\mathbf{x}; \gamma_m) \tag{10.6}$$

where α_m is the weight of base model m. The ensemble model is built in an iterative manner whereby at iteration m, we add base model $\alpha_m b(\mathbf{x}; \gamma_m)$ to the ensemble model. The final prediction is obtained by evaluating all the base models and taking the weighted sum.

The training is an iterative process whereby base models are added sequentially to the ensemble boosting model. At each iteration m, the base model and its corresponding optimal weight are solved and added to the ensemble, and this process is repeated for M base models.

$$\hat{f}_m(\mathbf{x}) = \hat{f}_{m-1}(\mathbf{x}) + \alpha_m b(\mathbf{x}; \gamma_m) \tag{10.7}$$

Typically, the optimal base model of each iteration is trained by minimizing a loss function.

$$\min_{\alpha_m, \gamma_m} \sum_{i=1}^{N} L(y^i, \alpha_m b(\mathbf{x}; \gamma_m)) \tag{10.8}$$

where L is the loss function such as the squared loss and negative log-likelihood.

10.7 Adaptive Boosting

Adaptive boosting, or simply AdaBoost (Freund and Schapire 1997) sequentially trains the base models on weighted training instances in which the weights are also sequentially modified to indicate the importance of the instances in the training process. Initially, the instances have equal weights. In the following iteration, the weights of the instances are individually updated based on the predictions of the base model. Those instances that were incorrectly predicted have their weights increased, while the instances that were correctly predicted have their weights decreased. Thus, the weights of the instances that are difficult to predict correctly will be ever-increased as iterations proceed, forcing the subsequent base models to focus on those instances.

Figure 10.3 illustrates the training process of AdaBoost using a simple training set. As shown in the figure, the number of iterations is 3, hence the training process builds three base models. Initially, equal weight is assigned to each instance and the base model is trained in the usual manner. Then, the weights of the instances are adjusted so that the misclassified instances are given more importance in the next iteration. In iteration $m = 1$, the three misclassified positive instances as indicated by the circle have their weights increased. This is shown by the size of those instances in iteration $m = 2$. The same procedure is followed for iteration $m = 3$.

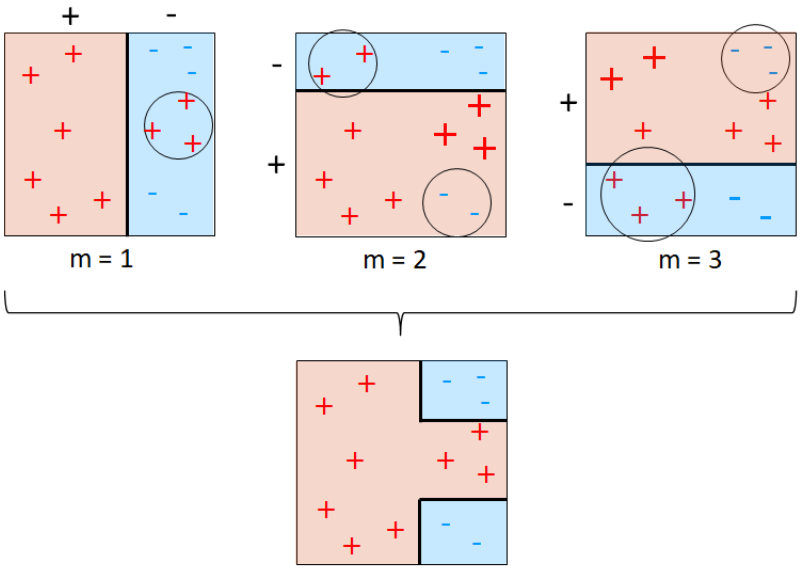


Figure 10.3 A series of base models are trained on weighted instances that are individually updated for every iteration

We describe the formulation of AdaBoost for binary classification. Although AdaBoost was initially introduced for binary classification, it has been extended to multi-class classification and regression. In AdaBoost, the optimal weak model is solved using the exponential loss which is defined as follows:

$$L(\hat{f}) = e^{-y\hat{f}(\mathbf{x})} \tag{10.9}$$

Consider a dataset $(\mathbf{x}^1, y^1), (\mathbf{x}^2, y^2), ..., (\mathbf{x}^N, y^N)$ where each target $y^i \in -1, +1$. At iteration m, we solve for the optimal weak model $b(\mathbf{x})$ and the corresponding weight α_m which to be added to the

ensemble model using the exponential loss.

$$\begin{aligned}\alpha_m, b_m &= \arg\min_{\alpha,b} \sum_{i=1}^{N} e^{-y^i(\hat{f}_{m-1}(\mathbf{x}^i)+\alpha b(\mathbf{x}^i))} \\ &= \arg\min_{\alpha,b} \sum_{i=1}^{N} e^{-y^i \hat{f}_{m-1}(\mathbf{x}^i)} e^{-\alpha y^i b(\mathbf{x}^i)}\end{aligned} \tag{10.10}$$

Let $w_m^i = e^{-y^i \hat{f}_{m-1}(\mathbf{x}^i)}$. Note that the term does not depend on α and b, but depends on the ensemble model $\hat{f}_{m-1}$. Thus, we can assume w_m^i as the weight of instance i at iteration m.

$$\alpha_m, b_m = \arg\min_{\alpha,b} \sum_{i=1}^{N} w_m^i e^{-\alpha y^i b(\mathbf{x}^i)} \tag{10.11}$$

The expression can be written based on two cases: misclassification ($y \neq b(\mathbf{x})$) and correct classification ($y = b(\mathbf{x})$).

$$\alpha_m, b_m = \arg\min_{\alpha,b} \sum_{i|y=b(\mathbf{x})} w_m^i e^{-\alpha} + \sum_{i|y\neq b(\mathbf{x})} w_m^i e^{\alpha} \tag{10.12}$$

We can write $\sum_{i|y=b(\mathbf{x})} w_m^i e^{-\alpha} = \sum_{i=1}^{N} w_m^i e^{-\alpha} - \sum_{i|y\neq b(\mathbf{x})} w_m^i e^{-\alpha}$. Therefore, equation 10.12 can be written as follows:

$$\begin{aligned}\alpha_m, b_m &= \arg\min_{\alpha,b} \sum_{i=1}^{N} w_m^i e^{-\alpha} - \sum_{i|y\neq b(\mathbf{x})} w_m^i e^{-\alpha} + \sum_{i|y\neq b(\mathbf{x})} w_m^i e^{\alpha} \\ &= \arg\min_{\alpha,b} e^{-\alpha} \sum_{i=1}^{N} w_m^i + \sum_{i|y\neq b(\mathbf{x})} w_m^i (e^{\alpha} - e^{-\alpha}) \\ &= \arg\min_{\alpha,b} (e^{\alpha} - e^{-\alpha}) \sum_{i=1}^{N} w_m^i I(y \neq b(\mathbf{x}^i)) + e^{-\alpha} \sum_{i=1}^{N} w_m^i\end{aligned} \tag{10.13}$$

Assuming the instance weights have been normalized such that $\sum_{i=1}^{N} w_m^i = 1$. The normalization factor is a constant factor, hence it will not affect the minimization operation.

$$\alpha_m, b_m = \arg\min_{\alpha,b} (e^{\alpha} - e^{-\alpha}) \sum_{i=1}^{N} w_m^i I(y \neq b(\mathbf{x}^i)) + e^{-\alpha} \tag{10.14}$$

From equation 10.14, since $e^{\alpha} - e^{-\alpha}$ is a positive value, we see that the weak base model b_m to be added is

$$b_m = \arg\min \sum_{i=1}^{N} w_m^i I(y \neq b(\mathbf{x}^i)) \tag{10.15}$$

We can see that this is actually the weighted error of the base model. Note that the base model does not have to be strong. It is sufficient that the weighted error is less than 0.5.

Now, we need to solve for α_m. Let us denote the weighted error as $\epsilon_m = \sum_{i=1}^{N} w_m^i I(y \neq b(\mathbf{x}^i))$. We

substitute ϵ_m into the expression.

$$\begin{aligned}\alpha_m &= \arg\min_{\alpha,b}(e^{\alpha} - e^{-\alpha})\epsilon_m + e^{-\alpha} \\ &= \arg\min_{\alpha,b} e^{\alpha}\epsilon_m - e^{-\alpha}\epsilon + e^{-\alpha}\end{aligned} \tag{10.16}$$

We take the derivative with respect to α, and then divide it by $e^{-\alpha}$.

$$\begin{aligned}\frac{\partial(e^{\alpha}\epsilon_m - e^{-\alpha}\epsilon + e^{-\alpha})}{\partial\alpha} &= e^{\alpha}\epsilon_m + e^{-\alpha}\epsilon_m - e^{-\alpha} \\ &= e^{2\alpha}\epsilon_m + \epsilon_m + 1\end{aligned} \tag{10.17}$$

Setting it to zero to solve for the minimum.

$$\begin{aligned}e^{2\alpha}\epsilon_m + \epsilon_m + 1 &= 0 \\ e^{2\alpha} &= \frac{1-\epsilon_m}{\epsilon_m} \\ 2\alpha &= \ln\frac{1-\epsilon_m}{\epsilon_m} \\ \alpha_m &= \frac{1}{2}\ln\frac{1-\epsilon_m}{\epsilon_m}\end{aligned} \tag{10.18}$$

Therefore, the ensemble model is then updated as follows:

$$\hat{f}_{m+1}(\mathbf{x}) = \hat{f}_m(\mathbf{x}) + \alpha_m b_m(\mathbf{x}) \tag{10.19}$$

Recall that $w_m^i = e^{-y^i \hat{f}_{m-1}(\mathbf{x}^i)}$. The weights of the next iteration is

$$\begin{aligned}w_{m+1}^i &= e^{-y^i \hat{f}_m(\mathbf{x}^i)} \\ &= e^{-y^i(\hat{f}_{m-1}(\mathbf{x}^i) + \alpha_m b_m(\mathbf{x}^i))} \\ &= e^{-y^i \hat{f}_{m-1}(\mathbf{x}^i)} \cdot e^{-y^i b_m(\mathbf{x}^i)\alpha_m} \\ &= w_m^i \cdot e^{-y^i b_m(\mathbf{x}^i)\alpha_m}\end{aligned} \tag{10.20}$$

We can rewrite $-y^i b_m(\mathbf{x}^i) = 2 \cdot I(y^i \neq b_m(\mathbf{x}^i)) - 1$. Thus, the weight update becomes

$$\begin{aligned}w_{m+1}^i &= w_m^i \cdot e^{(2\cdot I(y^i \neq b_m(\mathbf{x}^i)) - 1)\alpha_m} \\ &= w_m^i \cdot e^{\alpha_m 2\cdot I(y^i \neq b_m(\mathbf{x}^i)) - \alpha_m} \\ &= w_m^i \cdot e^{\alpha_m 2\cdot I(y^i \neq b_m(\mathbf{x}^i))} \cdot e^{-\alpha_m}\end{aligned} \tag{10.21}$$

Notice that the term $e^{-\alpha_m}$ multiplies all the weights by the same value and it will be canceled out in the normalization step. Thus, it has no effect on the weight update and can be dropped. Putting everything together, the algorithm of AdaBoost is given in Algorithm 10.

10.8 Gradient Boosting

Gradient boosting is a generalized boosting algorithm that is based on gradient descent. Recall that gradient descent is used to approximate the function (predictive model) by minimizing the loss function. This is done by computing the gradients of the loss function with respect to the weights

Algorithm 10 AdaBoost Algorithm

Initialize the instance weights $w_0^i = \frac{1}{N}$ where $i = 1, 2, ..., N$
for $m = 1$ to M **do**
    Train a base model b_m using the weighted training set
    $\epsilon_m = \frac{\sum_{i=1}^{N} w_m^i I(y \neq b(\mathbf{x}^i))}{\sum_{i=1}^{N} w_m^i}$
    if $\epsilon_m < \frac{1}{2}$ **then**
        Compute $\alpha_m = \frac{1}{2} \ln \frac{1-\epsilon_m}{\epsilon_m}$
        $\hat{f}_m(\mathbf{x}) = \hat{f}_{m-1}(\mathbf{x}) + \alpha_m b_m(\mathbf{x})$
        Set $w_{m+1}^i = \frac{w_m^i \cdot e^{\alpha_m 2 \cdot I(y^i \neq b_m(\mathbf{x}^i))}}{\sum_{i=1}^{N} w_{m+1}^i}$
    else
        return $\hat{f}_{m-1}(\mathbf{x})$
    end if
end for
$\hat{f}(\mathbf{x}) = \texttt{sign}[\sum_{m=1}^{M} \alpha_m b_m(\mathbf{x})]$

which are then used to update the weights. Using the same approach, we can search the same way over functions instead of the weights (Friedman 2001; Friedman 2002).

Consider a gradient boosting algorithm with M steps. The algorithm begins with a random guessed function as the initial weak base model, which could be the average function as indicated by the black line in Figure 10.4. For each subsequent m steps, we add a base model that will correct the error of its predecessor's predictions. This is done by calculating the difference between the predictions and the true values as indicated by the blue line. The prediction error which is also known as residual, is then used as the target to train the base model and add it to the ensemble model. The reasoning behind this is that if we can approximate the residuals, we can use it to correct wherever errors it had made. This is shown in the figure on the right, the residuals have been reduced when the average function is added with the newly built base model.

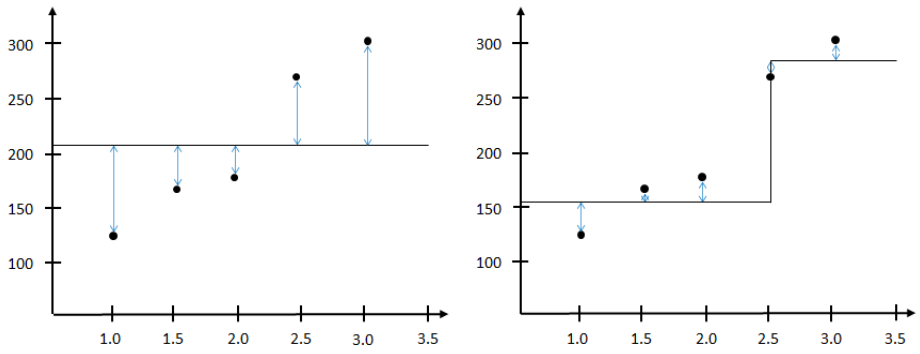


Figure 10.4 The base model is trained on the residuals to improve the prediction

We describe the formulation of gradient boosting for regression. Given a loss function

$$L(\hat{f}) = \sum_{i=1}^{N} L(y^i, \hat{f}(\mathbf{x}^i)) \tag{10.22}$$

where $L(y^i, \hat{f}(\mathbf{x}^i))$ is the regression loss such as squared loss function and $\hat{f}$ is the approximated function or the trained model. The aim is to minimize $L(\hat{f})$ with respect to $\hat{f}$. Recall that in

boosting, weak base models are sequentially added to produce an ensemble model. Thus, minimizing the loss function can be viewed as

$$\hat{\mathbf{f}} = \arg\min L(\hat{f}) \tag{10.23}$$

where $\hat{\mathbf{f}} = (\hat{f}(\mathbf{x}^1), \hat{f}(\mathbf{x}^2), ..., \hat{f}(\mathbf{x}^N))$. The optimization problem is solved using gradient descent. At step m, the gradient of $L(\hat{f})$ is

$$g_m^i = [\frac{\partial L(y^i, \hat{f}(\mathbf{x}^i))}{\partial \hat{f}(\mathbf{x}^i)}]_{\hat{f}=\hat{f}_{m-1}} = -(y^i - \hat{f}(\mathbf{x}^i)) \tag{10.24}$$

We would like to reduce the residuals of the previous step. Thus the negative gradient is taken

$$r_m^i = -g_m^i = (y^i - \hat{f}(\mathbf{x}^i)) \tag{10.25}$$

Train $b_m(\mathbf{x}, \gamma_m)$ using the $(\mathbf{x}^1, r^1), (\mathbf{x}^2, r^2), ..., (\mathbf{x}^N, r^N)$.

$$\gamma_m = \arg\min \sum_{i=1}^{N} (r_m^i - b(\mathbf{x}^i, \gamma))^2 \tag{10.26}$$

Finally, add b_m to the ensemble model.

$$\hat{f}_m = \hat{f}_{m-1} + \alpha b_m \tag{10.27}$$

where α is a constant value typically in the range of 0 and 1.

In practice, the base model is implemented a decision tree with a limited depth e.g. maximum depth of 3. Gradient boosting can be used for both regression and classification. For classification, the log loss (binary cross entropy) is used as the loss function. Putting everything together, the algorithm of gradient boosting is given in Algorithm 11.

Algorithm 11 Gradient Boosting Algorithm

1: **for** $m = 1$ to M **do**
2: Compute the gradient residual $r_m^i = -[\frac{\partial L(y^i, \hat{f}(\mathbf{x}^i))}{\partial \hat{f}(\mathbf{x}^i)}]_{\hat{f}=\hat{f}_{m-1}}$
3: Train a base model which minimize $\sum_{i=1}^{N}(r_m^i - b_m(\mathbf{x}^i, \gamma_m))^2$
4: Update $\hat{f}_m(\mathbf{x}) = \hat{f}_{m-1}(\mathbf{x}) + b_m(\mathbf{x}, \gamma_m)$
5: **end for**
6: Return $\hat{f}(\mathbf{x}) = \hat{f}_M(\mathbf{x})$

10.9 Implementing Ensemble Models

In this section, we will be implementing ensemble learning methods using Scikit-Learn. We will be using the QSAR biodegradation dataset. The prediction task is to classify the data into ready biodegradable and not ready biodegradable.

```
import pandas as pd
import wget
from zipfile import ZipFile
from sklearn.model_selection import train_test_split
data_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/
00254/biodeg.csv'
wget.download(data_url)
data_df = pd.read_csv('biodeg.csv', sep=';', header=None)
dataset = data_df.values
y = dataset[:,-1]
X = dataset[:,:-1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=10)
```

We create a function to print out the accuracy score, confusion matrix and classification report.

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

def print_clf_evaluation(y_test, y_pred):
    print(accuracy_score(y_test, y_pred))
    print(confusion_matrix(y_test, y_pred))
    print(classification_report(y_test, y_pred))
```

10.9.1 Stacking

Stacking ensemble classifier can be built by importing `StackingClassifier` from `sklearn.ensemble`. The parameters of the module are:

- `estimators` is the list of base models which will be stacked together.
- `final_estimator` is the classifier which will be used to combine the base models. If `final_estimator` is not defined, logistic regression will be used.
- `cv` is the number of folds for the cross-validation splitting strategy. If the value is `None`, `cv` equal to 5. The default value is `None`.
- `passthrough` accepts either `False` or `True`. If the value is `False`, only the predictions of estimators will be used as training data for `final_estimator`. If the value is `True`, the `final_estimator` is trained on the predictions as well as the original training data. By default, the value is `False`.

Let us build a stacking classifier using decision tree, k-nearest neighbors and logistic regression classifiers. First, we build the classifiers, and append the classifiers to a list.

```
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
dtc = DecisionTreeClassifier(random_state=0)
lgc = LogisticRegression()
knn = KNeighborsClassifier()

classifiers = []
classifiers.append(('c1', dtc))
classifiers.append(('c2', lgc))
classifiers.append(('c3', knn))
```

Now, we build the stacking classifier. We use support vector machine as `final_estimator`.

```
from sklearn.ensemble import StackingClassifier
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
svc = SVC(probability=True)
stk_clf = StackingClassifier(estimators=classifiers, final_estimator=svc)
stk_clf_pipeline = make_pipeline(StandardScaler(), stk_clf)
stk_clf_pipeline.fit(X_train, y_train)
```

We evaluate the classifier on the test set.

```
y_pred = stk_clf_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.886435331230284
[[193  19]
 [ 17  88]]
              precision    recall  f1-score   support

         NRB       0.92      0.91      0.91       212
          RB       0.82      0.84      0.83       105

    accuracy                           0.89       317
   macro avg       0.87      0.87      0.87       317
weighted avg       0.89      0.89      0.89       317
```

10.9.2 Voting Classifier

Voting ensemble classifier can be built by importing `VotingClassifier` from `sklearn.ensemble`. The parameters of the module are:

- `estimators` is the list of base models.
- `voting` is either '`hard`' or '`soft`'. If '`hard`', the output is the predicted class. If '`soft`', the output is the predicted probabilities.
- `weights` is the sequence of weights to weight the occurrences of predicted class labels for hard voting or class probabilities before averaging for soft voting.

Let us build a voting classifier using decision tree, SVM and logistic regression classifiers. First, we build the classifiers, and append the classifiers to a list.

```
dtc = DecisionTreeClassifier(random_state=0)
lgc = LogisticRegression()
svc = SVC(probability=True)
classifiers = []
classifiers.append(('c1', dtc))
classifiers.append(('c2', lgc))
classifiers.append(('c3', svc))
```

Now, we build the voting ensemble classifier. The voting scheme is set to '`soft`'. That means every

individual classifier provides a probability value of an example belonging to a particular target class.

```
from sklearn.ensemble import VotingClassifier
vot_clf = VotingClassifier(classifiers, voting='soft')
vot_clf_pipeline = make_pipeline(StandardScaler(), vot_clf)
vot_clf_pipeline.fit(X_train, y_train)
```

We evaluate the classifier on the test set.

```
y_pred = vot_clf_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.861198738170347
[[191  21]
 [ 23  82]]
              precision    recall  f1-score   support

         NRB       0.89      0.90      0.90       212
          RB       0.80      0.78      0.79       105

    accuracy                           0.86       317
   macro avg       0.84      0.84      0.84       317
weighted avg       0.86      0.86      0.86       317
```

10.9.3 Bagging Classifier

Bagging ensemble classifier can be built by import `BaggingClassifier` from `sklearn.ensemble`. The parameter

- `base_estimator` is the base model. If the parameter is not specified, decision tree will be used. The default value is `None`.
- `n_estimators` is the number of base models in the ensemble. By default, the value is 10.
- `max_samples` is the number of instances to draw from the training set to train each base model. This parameter accepts an integer or float value. If float, it is the fraction of the total number of instances. By default, the value is 1.0.
- `max_features` is he number of features to draw from the training set to train each base model. This parameter accepts an integer or float value. The default value is 1.0.
- `bootstrap` is for specifying whether samples are drawn with replacement. If `False`, sampling without replacement is performed. By default, the value is `True`.
- `oob_score` is for specifying whether to use out-of-bag samples to estimate the generalization error. This parameter is available if `bootstrap` value is `True`. By default, the value is `False`.
- `n_jobs` is the number of jobs to run in parallel for both training and testing. By default, the value is `None`, means 1.

Let us build a bagging classifier using decision tree as the base models. We set the number of base models equal to 120 and `oob_score` equal to `True`. Later we will display the out-of-bag score.

```
from sklearn.ensemble import BaggingClassifier
from sklearn.pipeline import Pipeline
dt = DecisionTreeClassifier(criterion='entropy', random_state=0)
bag_clf = BaggingClassifier(dt, n_estimators=120, oob_score=True,
                            n_jobs=2, random_state=0)
steps = [('scaler', StandardScaler()),('bag_clf', bag_clf)]
bag_clf_pipeline = Pipeline(steps=steps)
bag_clf_pipeline.fit(X_train, y_train)
```

We display the out-of-bag score by using `oob_score_` attribute.

```
print(bag_clf_pipeline['bag_clf'].oob_score_)
```

```
0.8658536585365854
```

We evaluate the classifiers on the test set. The test accuracy is similar to out-of-bag score.

```
y_pred = bag_clf_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.8675078864353313
[[201  11]
 [ 31  74]]
              precision    recall  f1-score   support

         NRB       0.87      0.95      0.91       212
          RB       0.87      0.70      0.78       105

    accuracy                           0.87       317
   macro avg       0.87      0.83      0.84       317
weighted avg       0.87      0.87      0.86       317
```

10.9.4 Random Forest Classifier

Now, let us build the Random Forest classifier. The classifier can be built by import `RandomForestClassifier` from `sklearn.ensemble`. The parameters of the module are:

- `n_estimators` is the number of base estimators in the ensemble. By default, the value is 100.
- `criterion` the criteria to measure the quality of a split. The possible values are `'gini'` (default) or `'entropy'`.
- `max_depth` is the maximum depth of the tree. This parameter accepts an integer value or `None`. By default, the value is `None` which means the tree will be built until all leaves contain less than `min_samples_split` samples.
- `min_samples_split` is the minimum number of samples required to split the node. This parameter accepts an integer value or a float value such as 5 or 0.1. A float value refers to the fraction of the total instances. By default, the value is 2.
- `min_samples_leaf` is the minimum number of samples required to be at a leaf node. This parameter accepts an integer value or a float value such as 5 or 0.1. A float value refers to the fraction of the total instances. By default, the value is 1.

- `max_features` The number of features to consider when looking for the best split. The possible values are `'sqrt'`, `'log2'` or `None`. By default, the value is `'sqrt'`.
- `max_leaf_nodes` is the maximum number of leaves. By default, the value is `None` which means an unlimited number of leaf nodes.
- `bootstrap` is for specifying whether samples are drawn with replacement. If `False`, sampling without replacement is performed. By default, the value is `True`.
- `oob_score` is for specifying whether to use out-of-bag samples to estimate the generalization error. This parameter is available if `bootstrap` value is `True`. By default, the value is `False`.
- `n_jobs` is the number of jobs to run in parallel for both training and testing. By default, the value is `None`, means 1.

We import the module to build a random forest classifier. The number of base models is set to 120, `oob_score` is set to `True` and `max_features` is set to `'sqrt'`.

```
from sklearn.ensemble import RandomForestClassifier
rf_clf = RandomForestClassifier(n_estimators=120, max_features='sqrt',
                                oob_score=True, n_jobs=2, random_state=0)
steps = [('scaler', StandardScaler()), ('rf_clf', rf_clf)]
rf_clf_pipeline = Pipeline(steps=steps)
rf_clf_pipeline.fit(X_train, y_train)
```

Similarly, we can display the out-of-bag score of the classifier.

```
print(rf_clf_pipeline['rf_clf'].oob_score_)
```

```
0.8685636856368564
```

We evaluate the classifiers on the test set.

```
y_pred = rf_clf_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.861198738170347
[[202  10]
 [ 34  71]]
              precision    recall  f1-score   support

         NRB       0.86      0.95      0.90       212
          RB       0.88      0.68      0.76       105

    accuracy                           0.86       317
   macro avg       0.87      0.81      0.83       317
weighted avg       0.86      0.86      0.86       317
```

10.9.5 AdaBoost Classifier

AdaBoost classifier can be built by importing `AdaboostClassifier` from `sklearn.ensemble`. The parameters of the module are:

- `base_estimator` is the base model from which the boosted ensemble is built. If

`base_estimator` is not defined, decision tree with a maximum depth equals to 1 will be used. By default, the value is `None`.

- `n_estimators` is the maximum number of base models at which boosting is terminated. By default, the value is 50.
- `learning_rate` is the weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier. There is a trade-off between the `learning_rate` and `n_estimators` parameters. The default value is 1.0.

We import the module and build AdaBoost classifier using decision tree with maximum depth equal to 1 as the base model. The number of base model is set to 300 and the learning rate is set to 0.05.

```
from sklearn.ensemble import AdaBoostClassifier
scaler = StandardScaler()
dt = DecisionTreeClassifier(max_depth=1, random_state=0)
ada_clf = AdaBoostClassifier(dt, n_estimators=300, learning_rate=0.05)
ada_clf_pipeline = make_pipeline(scaler, ada_clf)
ada_clf_pipeline.fit(X_train, y_train)
```

We evaluate the classifiers on the test set.

```
y_pred = ada_clf_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.8738170347003155
[[200  12]
 [ 28  77]]
              precision    recall  f1-score   support

         NRB       0.88      0.94      0.91       212
          RB       0.87      0.73      0.79       105

    accuracy                           0.87       317
   macro avg       0.87      0.84      0.85       317
weighted avg       0.87      0.87      0.87       317
```

10.9.6 Gradient Boosting Classifier

Finally, we implement gradient boosting. We import the module from `sklearn.ensemble`. Some of the parameters are

- `loss` is the loss function to be optimized. The possible values are '`log_loss`', '`deviance`', '`exponential`'. By default, the value is '`log_loss`'.
- `n_estimators` is the number of boosting stages to perform.
- `criterion` is the criterion of the decision tree. The possible values are '`friedman_mse`', '`squared_error`' and '`mse`'. The default value is '`friedman_mse`'.
- `learning_rate` is the weight of each tree. The default value is 0.1.
- `max_depth` is the maximum depth of the tree. This parameter accepts integer value. By default, the value is 3.
- `min_samples_split` is the minimum number of samples required to split the node. This parameter accepts an integer value or a float value such as 5 or 0.1. A float value refers to the

fraction of the total instances. By default, the value is 2.

- `min_samples_leaf` is the minimum number of samples required to be at a leaf node. This parameter accepts an integer value or a float value such as 5 or 0.1. A float value refers to the fraction of the total instances. By default, the value is 1.
- `max_features` The number of features to consider when looking for the best split. The possible values are `'sqrt'`, `'log2'` or `None`. By default, the value is `'sqrt'`.
- `max_leaf_nodes` is the maximum number of leaves. By default, the value is `None` which means an unlimited number of leaf nodes.

Let us build a gradient boosting classifier. We `n_estimator` equal to 500 and `learning_rate` equal to 0.05. The maximum depth of each base model is 1 and `max_features` is set to `'sqrt'`.

```
from sklearn.ensemble import GradientBoostingClassifier
scaler = StandardScaler()
gbt_clf = GradientBoostingClassifier(n_estimators=500, learning_rate=0.05,
                                     max_depth=1, max_features='sqrt',
                                     random_state=0)
gbt_clf_pipeline = make_pipeline(scaler, gbt_clf)
gbt_clf_pipeline.fit(X_train, y_train)
```

We evaluate the classifiers on the test set.

```
y_pred = gbt_clf_pipeline.predict(X_test)
print_clf_evaluation(y_test, y_pred)
```

```
0.8738170347003155
[[199  13]
 [ 27  78]]
              precision    recall  f1-score   support

         NRB       0.88      0.94      0.91       212
          RB       0.86      0.74      0.80       105

    accuracy                           0.87       317
   macro avg       0.87      0.84      0.85       317
weighted avg       0.87      0.87      0.87       317
```

Exercises

1. Explain weak models in the context of ensemble learning.
2. Explain the differences between bagging and boosting.
3. Explain the difference between voting and stacking using a linear perceptron as the combiner function.
4. Explain why is the training efficiency of random forest better than bagging.
5. Explain out-of-bag error.
6. Explain the differences between gradient boosting and adaptive boosting decision trees.
7. You are tasked to build a classification model for a prediction problem. The dataset is large

with a low number of noisy samples. However, it is found that a single model gets a very low performance. Thus, it is decided that the ensemble learning methods bagging or boosting is to be used to build a better classification model for the prediction problem. Choose a method and explain your reason for choosing the method over another method.

8. A data scientist builds an ensemble model using voting method for a binary classification problem. The ensemble model consists of three base models, h_1, h_2 and h_3. Assuming that given an input $\mathbf{x}$, the ensemble model misclassifies the data with the following probabilistic outputs, explain the shortcoming of the ensemble model and suggest an improvement that could be made to improve the accuracy of the ensemble model.

   $h_1(\mathbf{x}) = [0.25, 0.75]$

   $h_2(\mathbf{x}) = [0.12, 0.88]$

   $h_3(\mathbf{x}) = [0.19, 0.81]$

Chapter 11

Artificial Neural Network

11.1 Introduction

Artificial neural network or simply neural network is a machine learning technique that is modeled loosely after the human brain. It is a powerful, scalable and versatile technique that has been used to solve highly complex problems such as e-mail spam filtering, machine translation and pattern recognition. The fundamental building block of a neural network is the perceptron. Recall that a perceptron is a hyperplane that divides the feature space into positive and negative regions. The prediction of the hyperplane is defined by

$\hat{f}(\mathbf{x}) = \texttt{sign}(\sum_{j=0}^{d} w_j x_j) = \mathbf{w}^\intercal \mathbf{x}$

where $\mathbf{x}$ is the feature vector and $\mathbf{w}$ is the weight vector that defines the hyperplane. This computation can be represented by a network diagram as shown in Figure 11.1.

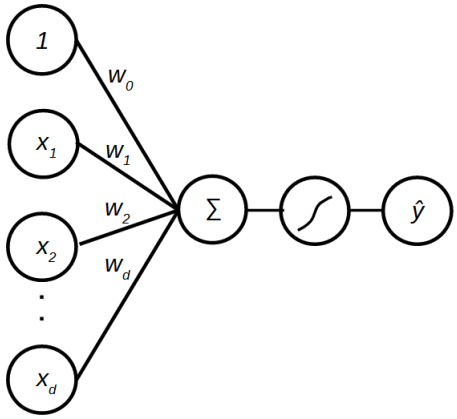


Figure 11.1 A network diagram of a perceptron

The perceptron accepts an arbitrary number of input features x_j, and each input is associated with a weight w_j. The linear combination of the inputs are then fed to the sign function to produce the prediction $\hat{y}$. Multiple perceptrons can be stacked on top of each other, forming a layer of output which is known as a fully connected layer. This layer of perceptrons can be connected to another layer of perceptrons as shown in Figure 11.2. This model is known as multilayer perceptron aka neural network. Note that an individual perceptron (later referred to as a unit) is simplified by representing the summation, sign function and prediction with a single node a_j. As can be seen

in the figure, the neural network has two layers of units, an intermediate layer (also known as a hidden layer) and an output layer. Thus, we have two sets of weights, one is for the hidden layer and the second set is for the output layer. The output layer is not directly connected to the input features. Instead, the output layer gets its inputs from the hidden layer which maps the inputs to a new feature space. We can think of this as deriving a new set of inputs for the prediction.

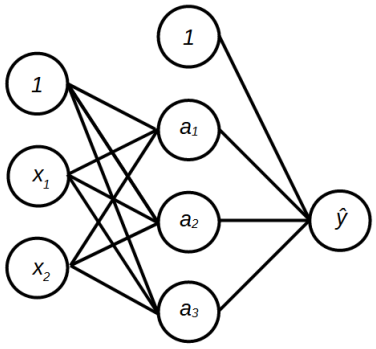


Figure 11.2 Multilayer of perceptrons known as neural network

In neural networks, the sigmoid function is used instead of the sign function. The sigmoid function which is also known as activation function is used to introduce non-linearity to the model. This will help the neural network to learn non-linear decision boundaries. Sigmoid function maps the input to any value range between $(0, 1)$ as shown in Figure 11.3(a). It has a nice property where it tends to bring the activations towards 0 or 1 due to the steep slope in the middle of the function. However, the gradient at either tail of 0 or 1 is very small or almost zero. This may cause the weights of the network not to be updated during training. This problem is known as vanishing gradient. Sigmoid function is defined as follows:

$$\mathbf{a}(\mathbf{z}) = \frac{1}{1 + e^{-\mathbf{z}}} \tag{11.1}$$

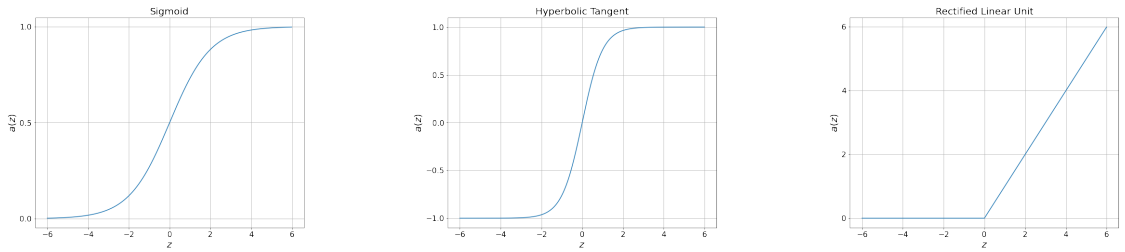
where $\mathbf{z} = \mathbf{w}^\intercal \mathbf{x}$ is the inner product between the weight vector $\mathbf{w}$ and the input vector $\mathbf{x}$. Besides sigmoid function, the commonly used activations are hyperbolic tangent (tanh) and rectified linear unit (ReLU). Tanh is similar to sigmoid function. Instead of mapping the input to any value in the range of $(0, 1)$, the output of the function is in the range of $(-1, 1)$. This is shown in Figure 11.3(b). It is closely related to sigmoid because the following holds.

$$\mathbf{a}(\mathbf{z}) = 2\sigma(2\mathbf{z}) - 1 \tag{11.2}$$

where σ is the sigmoid function. Although they have similar properties, tanh is preferred because it often converges faster than the sigmoid (Y. A. LeCun et al. 2012). ReLU is a piecewise linear function whereby it is linear for all positive values and zero for all negative values as shown in Figure 11.3(c). Compared to sigmoid and tanh, ReLU is simpler to compute and therefore takes less time to train and execute. Unlike sigmoid and tanh, ReLU does not suffer from saturation or vanishing gradient. However, ReLU may cause some units to always output zero due to the zero gradient at $z \leq 0$. As a result, the units are not taking part in the prediction (Lu et al. 2020). ReLU function is defined as follows:

$$\mathbf{a}(\mathbf{z}) = \max(0, \mathbf{z}) \tag{11.3}$$

Various variant of ReLU have been proposed such as Leaky ReLU and exponential liner unit (ELU).



(a) (b) (c)

Figure 11.3 The commonly used activation functions

11.2 Forward Propagation

The output of the neural network is obtained by propagating the inputs through the hidden layer to the output layer. Let L be the number of layers in a neural network excluding the input layer. Hence, for the neural network in Figure 11.2, $L = 2$ where $l = 1$ is the hidden layer and $l = 2$ is the output layer. The weights including the bias are denoted by $\mathbf{w}$ where $w_{ij}^{(l)}$ denotes the weight associated with the connection between unit i in layer $l-1$ and unit j in layer l. The $w_{0j}^{(l)}$ is the bias associated with unit j in layer l. We denote the i-th input as x_i. The forward propagation of input x_i to the units of layer l is given as follows:

$$z_1^{(1)} = w_{01}^{(1)} + w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 \tag{11.4}$$

$$z_2^{(1)} = w_{02}^{(1)} + w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 \tag{11.5}$$

$$z_3^{(1)} = w_{03}^{(1)} + w_{13}^{(1)} x_1 + w_{23}^{(1)} x_2 \tag{11.6}$$

$$a_1^{(1)} = g(z_1^{(1)}) \tag{11.7}$$

$$a_2^{(1)} = g(z_2^{(1)}) \tag{11.8}$$

$$a_3^{(1)} = g(z_3^{(1)}) \tag{11.9}$$

where $g(z_j^{(l)})$ is the activation function. The propagation of the signals to the output layer follows the same computation.

$$z^{(2)} = w_0^{(2)} + w_1^{(2)} a_1^{(1)} + w_2^{(2)} a_2^{(1)} + w_3^{(2)} a_3^{(1)} \tag{11.10}$$

$$\hat{y} = a^{(2)} = g(z^{(2)}) \tag{11.11}$$

We can generalize the forward propagation from layer l to layer $l+1$ as follows:

$$z_j^{(l+1)} = \sum_{i=0}^{s_l} w_{ij}^{(l+1)} a_i^{(l)} \tag{11.12}$$

$$a_j^{(l+1)} = g(z_j^{(l+1)}) \tag{11.13}$$

where s_l is the number of units in layer l.

11.3 Output Layer

Neural networks can be used for both classification and regression. For regression, the number of units is equal to the number of output responses. For example, if there are three responses, the number of units at the output layer is three. The activation function is typically linear $\mathbf{a}(\mathbf{x}) = z(\mathbf{x})$. ReLU or sigmoid function is used if the response is normalized. For binary classification, one unit is sufficient at the output layer to represent both classes. The sigmoid function is used to obtain an output value in the range of $(0, 1)$. This is similar to logistic regression. If the predicted value is above 0.5, assign the test input to class 1. Otherwise, assign it to class 0. For multiclass classification, the number of units is equal to the number of classes. The c-th unit represents the probability of class c whereby the sum of the probabilistic outputs is 1. Softmax function is used to ensure the output is probabilistic.

$$\hat{y}_j = \frac{e^{z_j}}{\sum_{c=1}^{C} e^{z_c}} \tag{11.14}$$

11.4 Backward Propagation Algorithm

The parameters of the neural networks needs to be estimated to solve the prediction problem. The estimation of the parameters (also known as training or learning) is a process of finding the weight and bias values that will produce the desired output at the output layer when a certain input is given to the network. How does a neural network learn? For a human to learn to do things, we will be told whether we are doing it right or wrong. For example, if we are to learn to play football, we watch how the ball flies as we kick the ball. The next time we kick the ball, we remember the mistake that we have done before and improvise the movement to kick the ball a bit better. Similarly, a neural network learns to solve a problem by comparing the output that is produced by the network with the output that was meant to produce. The difference between the neural network's output and the label (true output) is the amount of error that the neural network needs to reduce to improve its prediction. The error is calculated using the loss function. The choice of the loss function depends of the prediction problem. For regression problem, the mean squared error is used while the negative log-likelihood (cross entropy) is used for classification problem.

$$J_{sq}(\hat{f}) = \frac{1}{N}\sum_{i=1}^{N}(y^i - \hat{f}(\mathbf{x})^i)^2 \tag{11.15}$$

$$J_{nll}(\hat{f}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_c^i \log_e \hat{f}_c(\mathbf{x}^i) \tag{11.16}$$

$J(\hat{f})$ is a measure of how badly the network is performing. Thus, the loss function needs to be minimized in order for the neural network to produce the correct output. Minimizing the loss function involves adjusting the weights of the connections between two units in the network. The weights are adjusted in such a way that the output that will be produced by the network is close to the actual output. This involves propagating the error from the output layer to the hidden layer(s). Hence the process is known as backward propagation (or backpropagation) (Rumelhart, Geoffrey E. Hinton, and Williams 1986). The backpropagation is performed using the gradient descent. The backpropagation algorithm computes the partial derivatives (gradients) of the loss function with respect to each weight of the neural network. Consider the neural network in Figure 11.4.

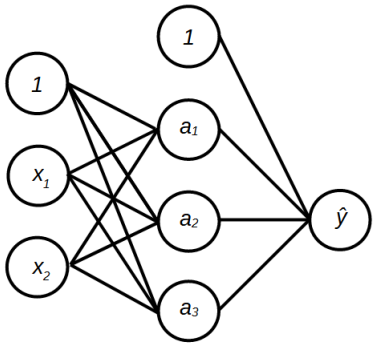


Figure 11.4 Multilayer of perceptrons known as neural network

The gradient with respect to weight $w_i^{(2)}$ connecting the hidden layer to the output layer is

$$\frac{\partial J}{\partial w_i^{(2)}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial w_i^{(2)}} \tag{11.17}$$

The gradient with respect to weight $w_{ij}^{(1)}$ connecting the input layer to the hidden layer is

$$\frac{\partial J}{\partial w_{ij}^{(1)}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z^{(2)}} \cdot \frac{\partial z^{(2)}}{\partial a_{ij}^{(1)}} \cdot \frac{\partial a_{ij}^{(1)}}{\partial z_{ij}^{(1)}} \cdot \frac{\partial z_{ij}^{(1)}}{\partial w_{ij}^{(1)}} \tag{11.18}$$

Then, the weights are updated using the computed gradients.

$$w_{ij}^{(l)} := w_{ij}^{(l)} - \alpha \frac{\partial J}{\partial w_{ij}^{(l)}} \tag{11.19}$$

where α is the step size or learning rate. Note that $:=$ is an assignment operator.

11.5 Variants of Gradient Descent

In batch gradient descent, all the training instances are fed to the network, the error for each training instance is calculated and the average is taken over all the errors, as shown in Figure 11.5. The average error is then used to update the weights of the neural network. Since the error is calculated over all training instances, the gradient is a good estimate of the true gradient and the training is stable. Also, the computation is less demanding since the weights are updated only once per epoch. However, the training could get stuck at the saddle point or suboptimal minimum because the calculated gradient is the same for every weight update.

In stochastic gradient descent, instead of the whole training set, a single training instance is used to calculate the error and update the weights as shown in Figure 11.6. Since the gradient is estimated on a single instance, the gradient is noisy and it is more difficult to find the optimal minimum. Thus, it will take longer to train the model. Also, stochastic gradient descent is computationally intensive because of the frequent weight update. However, due to the noisy gradient, training may escape saddle point and suboptimal minimum.

Mini-batch gradient descent is the commonly used technique for training neural networks. In mini-batch gradient descent, the training set is split into a number of small batches as shown in Figure 11.7. During training, one batch is fed to the network at a time, the error is calculated over

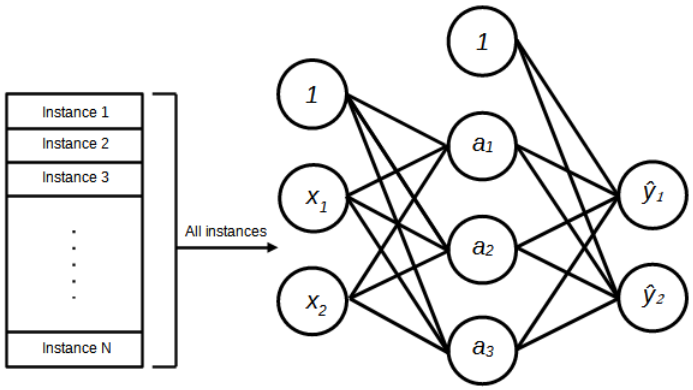


Figure 11.5 The entire training set is fed to the network to update the weights

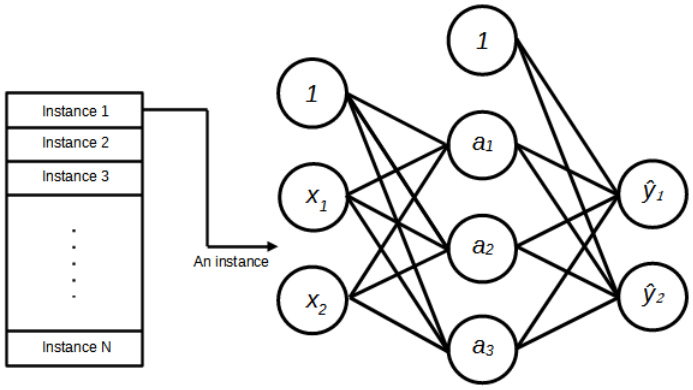


Figure 11.6 An instance is fed to the network to update the weights

that mini-batch and used to update the weights. The gradient is less noisy compared to stochastic gradient descent. Thus, the training is more stable and easier to find the optimal minimum and is more computationally efficient.

11.6 Introduction to TensorFlow

TensorFlow is a machine learning framework which can be used to design, build and train neural network models. TensorFlow can be used to perform numerical computations which other libraries can also do such as NumPy. The difference is that the numerical computations are done with data flow graphs. Before we can use TensorFlow, we need to install the library. The library and its documentation can be found at www.tensorflow.org. We can install TensorFlow either using conda or pip. The official website guides the installation using pip.

If we want to enable the GPU support in TensorFlow, we need to install CUDA and CuDNN. Otherwise, the following command can be skipped and proceed with TensorFlow installation.

```
conda install -c conda-forge cudatoolkit=11.2 cudnn=8.1.0
```

Install TensorFlow using pip.

```
pip install tensorflow
```

or install TensorFlow using conda.

```
conda install -c conda-forge tensorflow
```

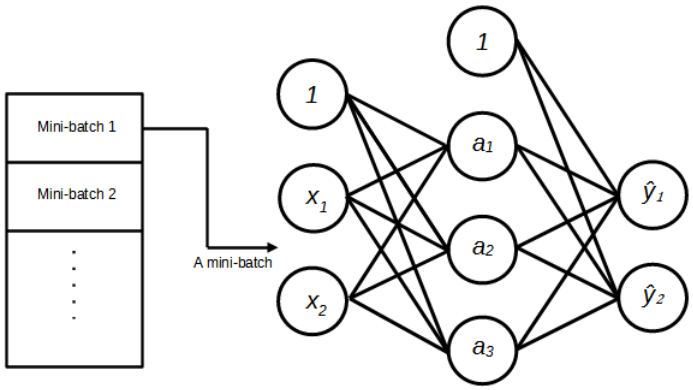


Figure 11.7 A batch is fed to the network to update the weights

To test the installation, import TensorFlow and print the version.

```
import tensorflow as tf
print(tf.__version__)
```

```
2.9.0
```

In TensorFlow, data is represented by tensors. Tensors are simply mathematical objects that can be used to describe scalars, vectors or matrices. Let's create some basic tensors. Here is a "scalar" or "rank-0" tensor . A scalar contains a single value, and no "axes".

```
x0 = tf.constant(4)
print(x0)
```

```
tf.Tensor(4, shape=(), dtype=int32)
```

Here is a "vector" or "rank-1" tensor is like a list of values. A vector has one dimension. Notice the shape of the tensor as displayed in the console.

```
x1 = tf.constant([1,2,3,4])
print(x1)
```

```
tf.Tensor([1 2 3 4], shape=(4,), dtype=int32)
```

Here, we are creating two matrices. A "matrix" or "rank-2" tensor has two dimensions.

```
x2 = tf.constant([[1,2,3],[4,5,6]])
x3 = tf.constant([[4,5,6],[7,8,9]])
```

Here we are creating a tensor with three dimensions. Tensors may have more axes e.g. three, four or more. Here, we are creating tensors with three dimensions. Note the shape of the tensor. The elements `[1,2,3]`, '1' is `x4[0,0,0]`, '2' is `x4[0,0,1]` and '3' is `x4[0,0,2]`.

```
x4 = tf.constant([[[1,2,3],[4,5,6]],
                  [[2,4,5],[6,7,8]]])
print(x4)
```

```
tf.Tensor(
[[[1 2 3]
  [4 5 6]]

 [[2 4 5]
  [6 7 8]]], shape=(2, 2, 3), dtype=int32)
```

11.6.1 Arithmetic Operations

Let us perform an addition involving scalar tensor.

```
result = tf.add(x0, 5)
print(result)
```

```
tf.Tensor(9, shape=(), dtype=int32)
```

Performing a multiplication on a vector tensor (x1) results in all the element will be multiplied with the specified value.

```
result = tf.multiply(x1, 5)
print(result)
```

```
tf.Tensor([ 5 10 15 20], shape=(4,), dtype=int32)
```

Here we perform arithmetic operations involving the matrix tensors. Here we are performing element-wise multiplication between x2 and x3.

```
result = tf.multiply(x2, x3)
print(result)
```

```
tf.Tensor(
[[ 4 10 18]
 [28 40 54]], shape=(2, 3), dtype=int32)
```

We can also use operators to perform arithmetic and matrix multiplication operations.

```
res1 = x2 + x3
print(res1)
res2 = x2 - x3
print(res2)
res3 = x2 * x3
print(res3)
res4 = x2 / x3
print(res4)
```

```
tf.Tensor(
[[ 5  7  9]
 [11 13 15]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[-3 -3 -3]
 [-3 -3 -3]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[ 4 10 18]
 [28 40 54]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[0.25       0.4        0.5       ]
 [0.57142857 0.625      0.66666667]], shape=(2, 3), dtype=float64)
```

To perform dot product, type the following statement. We need to transpose x2 so that the matrix size is compatible.

```
result = tf.matmul(tf.transpose(x2), x3)
print(result)
```

```
tf.Tensor(
[[32 37 42]
 [43 50 57]
 [54 63 72]], shape=(3, 3), dtype=int32)
```

Dot product can also be performed using `@` operator.

```
res5 = tf.transpose(x2) @ x3
print(res5)
```

```
tf.Tensor(
[[32 37 42]
 [43 50 57]
 [54 63 72]], shape=(3, 3), dtype=int32)
```

11.6.2 Indexing

TensorFlow follows standard Python indexing rules, similar to indexing a list or a string in Python, and the basic rules for NumPy indexing.

- indexes start at 0
- negative indices count backwards from the end

- colons (:), are used for slices: start:stop:step

Perform the following indexing and try to understand the indexing results. Here, we are retrieving the first row of x5.

```
x5 = tf.constant([[1,3,5],[2,4,6],[3,5,7]])
print(x5[0,:])
```

```
tf.Tensor([1 3 5], shape=(3,), dtype=int32)
```

We can select the last row or column by specifying −1. Here, we are retrieving the last column of x5.

```
print(x5[:,-1])
```

```
tf.Tensor([5 6 7], shape=(3,), dtype=int32)
```

Sometimes we need to reshape a tensor. To view the shape of a tensor, use shape property.

```
x6 = tf.constant([[1,2,3],[4,5,6],[7,8,9],[10,11,12]])
print(x6.shape)
```

```
(4, 3)
```

We can reshape a tensor into a new shape. To reshape a tensor use reshape method.

```
x6_reshape = tf.reshape(x6, [6, 2])
print(x6_reshape)
```

```
tf.Tensor(
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]], shape=(6, 2), dtype=int32)
```

Can we reshape a tensor into any shape? Yes, as long as the elements required for reshaping are equal in both shapes. We are allowed to have one unknown dimension by passing −1 as the value of `reshape`. Specifically, we do not have to specify an exact value for one of the dimensions in the `reshape` method.

```
x6_reshape = tf.reshape(x6, [-1, 2])
print(x6_reshape)
```

```
tf.Tensor(
[[ 1  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]
 [ 9 10]
 [11 12]], shape=(6, 2), dtype=int32)
```

An array can be converted to a tensor. Here, we create a NumPy array with one dimension.

```
import numpy as np
a1 = np.array([2,4,6,8])
print(a1)
```

```
[2 4 6 8]
```

```
x7 = tf.convert_to_tensor(a1)
print(x7)
```

```
tf.Tensor([2 4 6 8], shape=(4,), dtype=int32)
```

A tensor can be converted to a NumPy array. Here, the tensor is added with a value of 2.

```
x8 = tf.add(x7,2)
a2 = x8.numpy()
print(a2)
```

```
[ 4  6  8 10]
```

11.7 Implementing Artificial Neural Networks

Now, let us implement a neural network using TensorFlow. We will be using the same dataset as in the previous chapter, QSAR biodegradation dataset. The prediction task is to classify the data into ready biodegradable and not ready biodegradable. We load the dataset and then split it into training and test sets.

```
import pandas as pd
import wget
from zipfile import ZipFile
from sklearn.model_selection import train_test_split
data_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/
00254/biodeg.csv'
wget.download(data_url)
data_df = pd.read_csv('biodeg.csv', sep=';', header=None)
dataset = data_df.values
y = dataset[:,-1]
X = dataset[:,:-1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=10)
```

We import `MinMaxScaler` and normalize the data.

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train_norm = scaler.fit_transform(X_train)
X_test_norm = scaler.transform(X_test)
```

We need to encode the labels to one-hot encoding format. First we use `LabelEncoder` from `sklearn.preprocessing` to convert the string format to integer.

```
from sklearn.preprocessing import LabelEncoder
enc = LabelEncoder()
y_train = enc.fit_transform(y_train)
y_test = enc.transform(y_test)
```

Then, we use `one_hot` method to convert the integer labels to one-hot encoding format. Parameter depth defines the `depth` of the one-hot dimension.

```
y_train_1hot = tf.one_hot(y_train, depth=2)
y_test_1hot = tf.one_hot(y_test, depth=2)
```

Now, let us build the neural network. We will be using a module called `Sequential`. The module allows us to stack layers of fully connected units (neurons) to form a multilayer perceptron or neural network. The layers are defined as a list and the list is passed to the module. Here, we create the list and define the layers starting from the input followed by the hidden layer and the output layer. The input layer has 41 input units, the hidden layer has 20 units with ReLU activation function and the output layer has 1 unit with sigmoid activation function. The hidden and output layers are define using `Dense` module which is the regular fully connected layer. The module has a parameter called `use_bias` which can be set to `True` or `False`. By default, `use_bias` parameter is set to `True`.

```
layers = [tf.keras.layers.Input(shape=(41,)),
          tf.keras.layers.Dense(units=20, activation='relu'),
          tf.keras.layers.Dense(units=1, activation='sigmoid')]
```

We import the `Sequential` module and create the model.

```
from tensorflow.keras.models import Sequential
model = Sequential(layers=layers)
```

We can display the architecture of the model using `summary` method.

```
print(model.summary())
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 20)                840

 dense_1 (Dense)             (None, 1)                 21

=================================================================
Total params: 861
Trainable params: 861
Non-trainable params: 0
_________________________________________________________________
```

The summary reports the number of weights for each layer in the last column. The first column shows the name of each layer and the second column shows their output. We can also build a `Sequential` model by adding the layers one-by-one using `add` method as follows.

```
model = Sequential()
model.add(tf.keras.layers.Input(shape=(41,)))
model.add(tf.keras.layers.Dense(units=20, activation='relu'))
model.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
```

We can easily get the model's list of layers by its index.

```
output_layer = model.layers[1]
```

We can assess the weights using `get_weights` and `set_weights` methods. In the example, we get the weights including the bias and display them. Notice that the weights are randomly initialized and bias is initialized to zero.

```
weights, biases = output_layer.get_weights()
```

11.7.1 Loss Functions

After the model is created, we need to define the loss function to be used to train the model. TensorFlow provides various loss functions in `tf.keras.losses` such as (binary) cross entropy and mean squared error. Some of the loss functions are

- `BinaryCrossentropy` computes the cross entropy loss between labels and predicted labels.
- `CategoricalCrossentropy` computes the cross entropy loss between the labels and predicted labels.
- `MeanSquaredError` computes the mean of squared difference between targets and predicted

values.

Since the prediction task is binary classification, we will use the binary cross entropy.

```
loss_fn = tf.keras.losses.BinaryCrossentropy()
```

Let us predict the first five rows of the training set and use the loss function to compute the loss value. We pass the data to `predict` method of the model to produce the predictions. In this example, we pass first five rows of the training set.

```
y_pred = model.predict(X_train_norm[:5])
```

Then, we pass the true labels and the predicted labels to the loss function. `squeeze` function is to reduce the tensor's dimension to 1-d since the `y_train` is 1-d.

```
print(loss_fn(y_train[:5], tf.squeeze(y_pred)))
```

```
tf.Tensor(0.6442506, shape=(), dtype=float32)
```

11.7.2 Optimizers

Now, let us define the training algorithm or known as optimizer in TensorFlow. In addition to the gradient descent, TensorFlow provides the variants of gradient descent such as Adagrad and Adam algorithms. Some of the training algorithms are

- `SGD` is the gradient descent (with momentum) algorithm
- `Adadelta` is the optimizer that implements the Adadelta algorithm.
- `RMSprop` is the optimizer that implements the RMSprop algorithm.
- `Adam` is the optimizer that implements the Adam algorithm.

The optimizer has a parameter (`learning_rate`) to set the learning rate. By default, the value is 0.001. Optimizers such as `Adadelta`, `RMSprop` and `Adam` have additional parameters for adaptive learning rate. Please refer to the official documentation for more information. Let us use the Adam algorithm.

```
optimizer = tf.keras.optimizers.Adam(learning_rate=0.002)
```

The model needs to be compiled before it can be trained. A model is compiled by calling `compile` method to specify the loss function and the optimizer (training algorithm). We can specify the list of metrics to be evaluated by the model during training and testing. In this example, we use the accuracy metric since the prediction task is binary classification.

```
model.compile(optimizer=optimizer, loss=loss_fn, metrics=['accuracy'])
```

We can also specify the parameters using string such as

```
model.compile(optimizer='adam', loss='binary_crossentropy')
```

Now, the model is ready to be trained. To train the model, we simply call the `fit` method. We pass the training data, specify the maximum epochs, the batch size and the size of the validation

set. By default, the validation set is not used. Notice that `fit` method returns a `History` object which contains a record of the loss and accuracy values.

```
history = model.fit(x=X_train_norm, y=y_train, epochs=200,
                    batch_size=64, validation_split=0.2)
```

```
Epoch 1/200
10/10 [==============================] - 1s 26ms/step - loss: 0.6556
 - accuracy: 0.6254 - val_loss: 0.6094 - val_accuracy: 0.7230
Epoch 2/200
10/10 [==============================] - 0s 5ms/step - loss: 0.6230
 - accuracy: 0.6322 - val_loss: 0.5749 - val_accuracy: 0.7230
...
Epoch 200/200
10/10 [==============================] - 0s 6ms/step - loss: 0.3034
 - accuracy: 0.8695 - val_loss: 0.3326 - val_accuracy: 0.8581
```

`History` object has `history` attribute to access the values.

```
train_loss = history.history['loss']
vald_loss = history.history['val_loss']
train_acc = history.history['accuracy']
vald_acc = history.history['val_accuracy']
```

We plot the loss and accuracy against the epoch as shown in Figure 11.8.

```
from matplotlib import pyplot as plt
x = np.arange(1, 201, 1)
plt.figure(figsize=(8,6))
plt.plot(x, train_loss, label='Train Loss')
plt.plot(x, vald_loss, label='Validation Loss')
plt.plot(x, train_acc, label='Train Accuracy')
plt.plot(x, vald_acc, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.legend()
```

We can see that over the 200 epochs, the training loss and validation loss decrease from 0.65 to 0.30 while the training accuracy and validation accuracy increase from 0.62 to 0.85. This shows that the model has learned from the data. Also, we can see that the loss value does not reduce significantly after epoch 75 which indicates that the training has converged.

Let us evaluate the model on the test set. We round the predicted values to obtain the predicted labels.

```
y_pred = model.predict(X_test_norm)
y_pred = tf.round(y_pred)
```

We display the confusion matrix and the classification report.

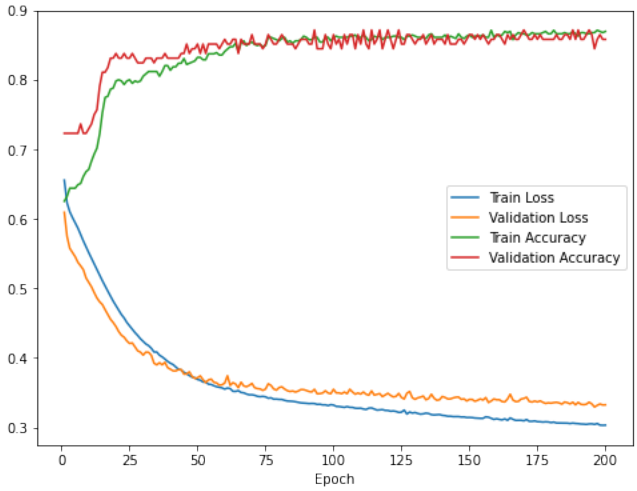


Figure 11.8 The training loss and accuracy against epoch.

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

print(accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
0.8675078864353313
[[195  17]
 [ 25  80]]
              precision    recall  f1-score   support

           0       0.89      0.92      0.90       212
           1       0.82      0.76      0.79       105

    accuracy                           0.87       317
   macro avg       0.86      0.84      0.85       317
weighted avg       0.87      0.87      0.87       317
```

11.7.3 Saving and Loading Models

There are different ways to save the models. We can save the weights of the model only or the entire model. To save the weights of the model, we use `save_weights` method. By default, the method uses the TensorFlow checkpoint format with a `.ckpt` extension. In this example, we save the weights in a folder `model_weights`.

```
model.save_weights('model_weights/')
```

To restore the weights, first, we need to define and compile the model and then use `load_weights` method to restore the weights.

```
model_restore = Sequential()
model_restore.add(tf.keras.layers.Input(shape=(41,)))
model_restore.add(tf.keras.layers.Dense(units=20, activation='relu'))
model_restore.add(tf.keras.layers.Dense(units=1, activation='sigmoid'))
model_restore.compile(optimizer=optimizer, loss=loss_fn, metrics=['accuracy'])
model_restore.load_weights('model_weights/')
```

We use evaluate method to evaluate the restored model.

```
loss, acc = model_restore.evaluate(X_test_norm, y_test)
```

```
10/10 [==============================] - 0s 1ms/step - loss: 0.3239
 - accuracy: 0.8675
```

To save the entire model, we use `save` method. By default, TensorFlow save the model in `SavedModel` format.

```
model.save('entire_model/')
```

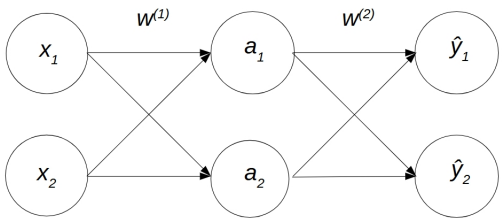
Use `load_model` method to load the saved model.

```
model_saved = tf.keras.models.load_model('entire_model/')
loss, acc = model_saved.evaluate(X_test_norm, y_test)
```

```
10/10 [==============================] - 0s 2ms/step - loss: 0.3239
 - accuracy: 0.8675
```

Exercises

1. Explain the role of the activation functions in neural networks.
2. Explain the use of the loss functions in neural networks.
3. Derive the update equations when the hidden units use tanh. Use the fact that the derivative of tanh is $(1 - \texttt{tanh}^2)$.
4. Explain the variants of gradient descent.
5. Consider a neural network with one hidden layer. The activation function of the hidden layer and output layer is rectified linear unit and sigmoid.



(a) Calculate the output of the network given weight $w^{(1)}, w^{(2)}$ and input data $\mathbf{x}$.

$$w^{(1)} = \begin{bmatrix} -1 & 3 \\ 2 & 1 \end{bmatrix}$$
$$w^{(2)} = \begin{bmatrix} 1 & -1 \\ -2 & 1 \end{bmatrix}$$
$$\mathbf{x} = \begin{bmatrix} 1 & -1 \end{bmatrix}$$

(b) Calculate the gradient if the ground truth (label) $y = \begin{bmatrix} 0 & 1 \end{bmatrix}$.

6. A neural network is created for predicting a dataset with 20 features and two classes.
   (a) State the number of units of input and output layers. Justify your answer.
   (b) State the activation function of the output layer.
   (c) State the loss function to train the neural network.
7. A neural network is created for predicting a dataset with 40 features and four classes.
   (a) State the number of units of input and output layers. Justify your answer.
   (b) State the activation function of the output layer.
   (c) State the loss function to train the neural network.

Chapter 12

Regularization for Neural Network

12.1 Introduction

The main problem in machine learning is how to ensure the predictive models perform well not only on the training set but also on new data. Numerous methods have been introduced to improve the model generalization. These methods are collectively known as regularization. This chapter explores various regularization methods for neural networks such as norm penalty, early stopping and dropout. We discuss the concepts of the regularization methods and how the methods can applied to neural networks.

12.2 Parameter Norm Penalty

Parameter norm penalty limits the capacity of the model by regularizing the weights of the neural network. The regularization is performed by adding a penalty term Ω to the loss function as follows:

$$J(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} L(y^i, \hat{y}^i) + \lambda\Omega(\mathbf{w}) \tag{12.1}$$

where $\lambda \in [0, \infty]$ is the weight of the penalty term (also known as shrinkage parameter), $L(y^i, \hat{y}^i)$ is the standard loss function e.g. mean squared error, negative log-likelihood or other loss functions. Setting λ to 0 results in no regularization and a large value of λ corresponds to more regularization. In general, the penalty term penalizes the weights of the neural network from going too large. This is because, when the training algorithm minimizes $J(\mathbf{w})$, it will minimize both the loss function and the weights of the neural network. This may cause some of the weights to be close to zero especially if the shrinkage parameter is set to a large value. As a result, units with weight values close to zero switch off. This reduces the chance of overfitting because the neural network capacity is reduced.

L^2 parameter norm penalty or known as weight decay is one of the most commonly used penalty terms. This regularization method adds penalty term $\Omega(\mathbf{w}) = \frac{1}{2}\sum_l \|\mathbf{w}^{(l)}\|_2^2$ to the objective function.

The derivative of objective function $J(\mathbf{w})$ is

$$\frac{\partial J(\mathbf{w})}{\partial w_{ij}^{(l)}} = \frac{\partial L(y^i, \hat{y}^i)}{\partial w_{ij}^{(l)}} + \lambda w_{ij}^{(l)} \tag{12.2}$$

The weight update is

$$\begin{aligned} w_{ij}^{(l)} &:= w_{ij}^{(l)} - \alpha(\frac{\partial L(y^i, \hat{y}^i)}{\partial w_{ij}^{(l)}} + \lambda w_{ij}^{(l)}) \\ &:= w_{ij}^{(l)} - \alpha\frac{\partial L(y^i, \hat{y}^i)}{\partial w_{ij}^{(l)}} - \alpha\lambda w_{ij}^{(l)} \\ &:= w_{ij}^{(l)}(1 - \alpha\lambda) - \lambda\frac{\partial L(y^i, \hat{y}^i)}{\partial w_{ij}^{(l)}} \end{aligned} \tag{12.3}$$

As we can see in equation 12.3, the term $w_{ij}^{(l)}(1 - \alpha\lambda)$ has the effect of decreassing $w_{ij}^{(l)}$ with every update because the weight is multiplied by a factor $(1 - \alpha\lambda)$ which is less than 1 (assuming $0 < \alpha\lambda < 1$). Another parameter norm penalty is L^1 parameter norm penalty. This regularization method adds penalty term $\Omega(\mathbf{w}) = \sum|\mathbf{w}|$ to the objective function.

The derivative of objective function $J(\mathbf{w})$ is

$$\frac{\partial J(\mathbf{w})}{\partial w_{ij}^{(l)}} = \frac{\partial L(y^i, \hat{y}^i)}{\partial w_{ij}^{(l)}} + \lambda\texttt{sign}(w_{ij}^{(l)}) \tag{12.4}$$

The weight update is

$$\begin{aligned} w_{ij}^{(l)} &:= w_{ij}^{(l)} - \alpha(\frac{\partial L(y^i, \hat{y}^i)}{\partial w_{ij}^{(l)}} + \lambda\texttt{sign}(w_{ij}^{(l)})) \\ &:= w_{ij}^{(l)} - \alpha\lambda\texttt{sign}(w_{ij}^{(l)}) - \alpha\frac{\partial L(y^i, \hat{y}^i)}{\partial w_{ij}^{(l)}} \end{aligned} \tag{12.5}$$

As we can see in equation 12.5, the effect of L^1 regularization is different from L^2 regularization. Specifically, the weights are updated based on gradients that are regularized by the constant factor with a sign equal to $\texttt{sign}(w_{ij}^{(l)})$.

12.3 Early Stopping

When training a neural network, the training loss and validation loss decrease over time. However, after some time, the validation loss begins to increase indicating the model is overfitting the data. Early stopping is a method to terminate the training algorithm before the overfitting occurs. The model is overfitting the data is detected based on the evaluation of the validation set. For each epoch, after the backpropagation, the validation set is used to evaluate the model performance by calculating the validation loss or validation accuracy. The performance metric is used to determine if the model's performance is improving or not. Every time the performance is improving, we store a copy of the model's parameters. When the model performance is not improving and the training algorithm is terminated, we return the stored parameters. The early stopping algorithm is given as follows. p is the number of epochs we will wait before the training is terminated if no improvement is observed. The parameter is known as "patience". Algorithm 12 shows the pseudocode of the early stopping.

Algorithm 12 Early Stopping Algorithm

```
E is maximum epoch
Initialize w
Initialize p
j ← 0
v ← ∞
w* ← w
for i ← 1 to E do
    update w by performing backpropagation
    v* ← ValidationSetError(w)
    if v* < v then
        j ← 0
        w* ← w
        v ← v*
    else
        j ← j + 1
    end if
    if j = p then
        terminate loop
    end if
end for
```

12.4 Dropout

One of the approaches to improve the accuracy of the prediction is to build several predictive neural networks and combine the predicted values via averaging. However, training a diverse set of neural networks is difficult and expensive due to the challenging task of finding the optimal hyperparameters. Furthermore, neural networks typically require a large number of training instances. Thus, training several neural networks may not be feasible because there may not be enough data to create different subsets to train the networks.

Dropout is a regularization method that approximately ensembles multiple neural networks with different architectures (Srivastava et al. 2014). Dropout works by randomly eliminating a number of units in a hidden layer during the training of the neural network. The method defines a probability of a unit being dropped in a layer. For example, if the probability is set to 0.5, all the units in the layer are dropped at a 0.5 rate. For each weight update, after the instances are loaded into the mini-batch, a set of units of each layer is randomly sampled to be dropped. The mini-batch is then loaded to the reduced neural network for forward and backpropagation. This can be seen as estimating the weights of different network architectures since different sets of units are eliminated as shown in Figure 12.1. Thus, the randomly dropping out units has the effect of training and evaluating exponentially many different neural networks. At test time, the resulting neural network is used without dropout.

Another advantage of dropout is preventing the co-adaptation of units in the neural network (Geoffrey E Hinton et al. 2012). Co-adaptation occurs when some units are highly dependent on other units which means when the independent units fail to produce the correct activations, the dependent units are affected as well, hence the neural network will fail to correctly predict the inputs. With dropout, no units can be too "powerful" due to the random elimination, thus forcing all units in the neural network to take part in the prediction.

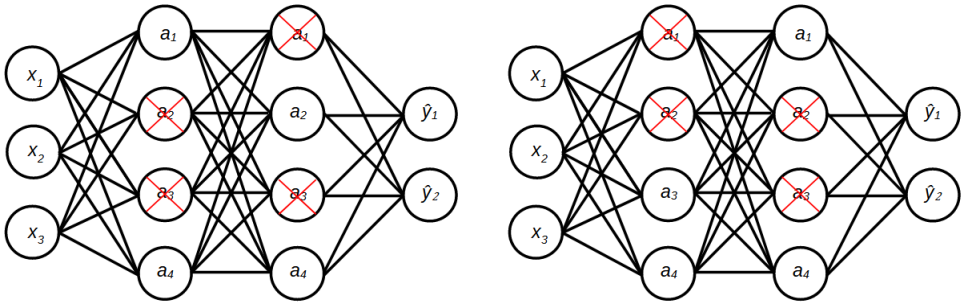


Figure 12.1 Different units are randomly dropped for each backpropagation

Let p be the probability of units to be retained. Therefore, the drop rate is $1-p$. $\mathbf{r}$ is the vector of Bernoulli whereby each element in the vector has the probability of p of being 1. Note that Bernoulli variable can take either 0 (failure) or 1 (success) with the probability of being success is p.

$$\mathbf{r} = \text{Bernoulli}(p) \tag{12.6}$$

During training, the forward propagation with dropout is implemented as follows:

$$\tilde{\mathbf{a}}^{(l)} = \frac{\mathbf{r}^{(l)} * \mathbf{a}^{(l)}}{p} \tag{12.7}$$

$$\mathbf{z}_i^{(l+1)} = \mathbf{w}_i^{(l)} \tilde{\mathbf{a}}^{(l)} + w_{i0}^{(l)} \tag{12.8}$$

$$\mathbf{a}_i^{(l+1)} = g(\mathbf{z}_i^{(l+1)}) \tag{12.9}$$

where $*$ is the element-wise multiplication, $\mathbf{a}^{(l)}$ is the activation of the hidden layer l and $\tilde{\mathbf{a}}^{(l)}$ is the reduced activation of the hidden layer. The activation of the layer is multiplied element-wise with the vector of Bernoulli. This causes some of the activations to be zero. The reduced activation is scaled with $\frac{1}{p}$ to ensure the expectation of the input to the next layer remains the same. The reduced activation is then used as input to the next layer. At test time, the forward propagation is implemented without dropout.

12.5 Batch Normalization

We commonly used mini-batch gradient descent to train neural networks. In mini-batch gradient descent, the weight update is performed for every mini-batch and each batch may contain data from different distributions. Furthermore, the distribution of the inputs to each hidden layer may change after each mini-batch when the weights are updated. This could slow down the training process because each layer needs to adapt to the new distribution.

Batch normalization is a regularization method that incorporates normalization into the model architecture (Ioffe and Szegedy 2015). Specifically, the method allows each layer to perform normalization on the layer's output for each mini-batch. The layer normalization has the effect of stabilizing and speed up the training process. Let z_i denotes the summation of the weighted inputs of unit i of a hidden layer. We calculate the mean of z_i over the mini-batch.

$$\bar{z}_i = \sum_{m=1}^{M} z_i^m \tag{12.10}$$

where M is the number of instances in the mini-batch. The variance of the mini-batch is

$$\sigma_i^2 = \frac{1}{M}\sum_{m=1}^{M}(z_i^m - \bar{z}_i) \tag{12.11}$$

The normalization is given as follows:

$$\acute{z}_i^m = \frac{z_i^m - \bar{z}_i}{\sqrt{\sigma_i^2 + \epsilon}} \tag{12.12}$$

$$\tilde{z}_i^m = \gamma_i \acute{z}_i^m + \beta_i \tag{12.13}$$

where ϵ is a small constant for computation stability, and γ and β are trainable parameters to provide flexibility to the layer to adjust the distribution of the activations. The parameters are estimated during training. Batch normalization can be applied before or after the non-linearity (activation function). Note that the authors applied the normalization before non-linearity.

12.6 Implementing Regularization for Neural Network

In this section, we will be implementing the regularization methods to improve the generalization of neural network. We will be using the same dataset as in the previous chapter, QSAR dataset. The prediction task is to classify the data into ready biodegradable and not ready biodegradable. Now, let us load a dataset, build and train the neural network.

```
import pandas as pd
import tensorflow as tf
import wget
from zipfile import ZipFile
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import LabelEncoder

data_url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/
00254/biodeg.csv'
wget.download(data_url)
data_df = pd.read_csv('biodeg.csv', sep=';', header=None)
dataset = data_df.values
y = dataset[:,-1]
X = dataset[:,:-1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=10)
scaler = MinMaxScaler()
X_train_norm = scaler.fit_transform(X_train)
X_test_norm = scaler.transform(X_test)

enc = LabelEncoder()
y_train = enc.fit_transform(y_train)
y_test = enc.transform(y_test)
y_train_1hot = tf.one_hot(y_train, depth=2)
y_test_1hot = tf.one_hot(y_test, depth=2)
```

In the previous chapter, the model is built using `Sequential` module. Although `Sequential` module

is convenient, the module cannot be used to build a neural network with complex architecture. TensorFlow provides the `functional` API to build models that are more flexible than the `Sequential` module. We will be building a neural network with an architecture given in Figure 12.2.

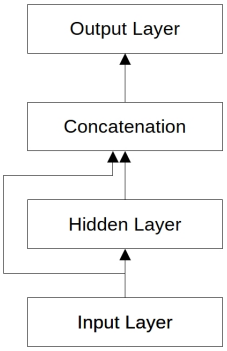


Figure 12.2 The model's architecture

Before we build the model, we describe the implementation of norm penalty regularization. The norm penalty is implemented by specifying parameter `kernel_regularizer` of Dense module. First, we import `L2` module from `tensorflow.keras.regularizers` and create the object of `L2`.

```
from tensorflow.keras.regularizers import L2
l2_regularizer = L2(l2=0.01)
```

Now let us build the neural network. First, we need to create an `Input` object for the input layer. Next we create a `Dense` object for the hidden layer with the `L2` object. Also, we apply batch normalization to the hidden layer. Then, we use `Concatenate` module to concatenate the input and the output of the hidden layer. Finally, we create a `Dense` object for the output layer.

```
from tensorflow.keras.layers import Input, Dense, Concatenate, BatchNormalization
inputs = Input(shape=(41,))
hidden = Dense(units=20, activation='relu',
               kernel_regularizer=l2_regularizer)(inputs)
batchnorm = BatchNormalization()(hidden)
concat = Concatenate()([inputs, batchnorm])
outputs = Dense(units=1, activation='sigmoid')(concat)
```

Dropout regularization can be implemented by importing `Dropout` from `tensorflow.keras.layers`. To implement dropout, we apply it after the layer. The following codes build a neural network with dropout regularization applies to the hidden layer.

```
from tensorflow.keras.layers import Dropout
...
hidden = Dense(units=20, activation='relu',
               kernel_regularizer=l2_regularizer)(inputs)
dropout = Dropout(rate=0.3)(hidden)
concat = Concatenate()([inputs, dropout])
...
```

Once the layers have been defined, we create the model by specifying which the inputs and the

outputs.

```
from tensorflow.keras.models import Model
model = Model(inputs=inputs, outputs=outputs)
model.compile(optimizer='adam', loss='binary_crossentropy',
              metrics=['accuracy'])
```

The `fit` method accepts a callbacks argument that lets us specify a list of actions to be performed at various stages of training i.e. at the start or end of each epoch. For example, the early stopping is implemented as a callback when the model is fitted with the training set. Furthermore, `ModelCheckpoint` callback is also implemented to save the weights at regular intervals during the training.

We describe some of the parameters of the callbacks. The complete description can be found at the official documentation. `EarlyStopping` callback has several parameters to be set e.g. `monitor` and `patience`. `monitor` is used to define the metric to be monitored during the training and `patience` is the number of epochs with no improvement after which training will be stopped.

The parameters of `ModelCheckpoint` callback are `filepath`, `monitor` and `save_weights_only`. `filepath` is the path to save the model, monitor is used to define the metric to be monitored during the training and `save_weights_only` is used to specify whether the model's weights will be saved or the full model is saved.

```
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
train_callbacks = [EarlyStopping(monitor='val_loss', patience=10),
                   ModelCheckpoint(filepath='checkpoints/', monitor='val_loss',
                                   save_weights_only=True)]
history = model.fit(x=X_train_norm, y=y_train, epochs=200, batch_size=64,
                    validation_split=0.2, callbacks=train_callbacks)
```

We get the training and accuracy values from `History`.

```
train_loss = history.history['loss']
vald_loss = history.history['val_loss']
train_acc = history.history['accuracy']
vald_acc = history.history['val_accuracy']
```

We plot the training and accuracy against epoch as shown in Figure 12.3.

```
from matplotlib import pyplot as plt
import numpy as np
x = np.arange(1, len(train_loss)+1, 1)
plt.figure(figsize=(8,6))
plt.plot(x, train_loss, label='Train Loss')
plt.plot(x, vald_loss, label='Validation Loss')
plt.plot(x, train_acc, label='Train Accuracy')
plt.plot(x, vald_acc, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.legend()
```

Load the model weights that are considered the best.

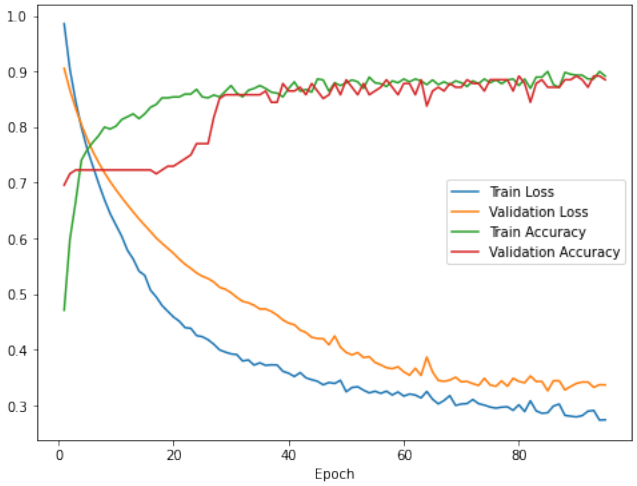


Figure 12.3 The training loss and accuracy against epoch

```
model.load_weights('checkpoints/')
```

We evaluate the model on the test set.

```
y_pred = model.predict(X_test_norm)
y_pred = tf.round(y_pred)
```

We display the confusion matrix and the classification report.

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
print(accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
0.8832807570977917
[[198  14]
 [ 23  82]]
              precision    recall  f1-score   support

           0       0.90      0.93      0.91       212
           1       0.85      0.78      0.82       105

    accuracy                           0.88       317
   macro avg       0.88      0.86      0.87       317
weighted avg       0.88      0.88      0.88       317
```

The model performed well in classifying the classes, achieving an F-score of above 0.80 for both classes. The accuracy of the model is 0.88.

12.7 Using TensorBoard for Visualization

TensorBoard is an interactive visualization tool which can be used to visualize the training metrics such as loss and accuracy and the model's computation graph. TensorBoard is automatically

installed when TensorFlow is installed. First, let us define the folder to store the training log data.

```
import datetime
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
```

Navigate to the folder where `logs` is located. Start TensorBoard using `tensorboard` command.

```
tensorboard --logdir logs/fit
```

Then, open localhost URL (http://localhost:6006/) in a web browser as shown in Figure 12.4. TensorBoard can be loaded in Jupyter Notebook. Load the TensorBoard notebook extension. Then, start TensorBoard within a notebook experience with `%tensorboard` command.

```
%load_ext tensorboard
%tensorboard --logdir logs/fit
```

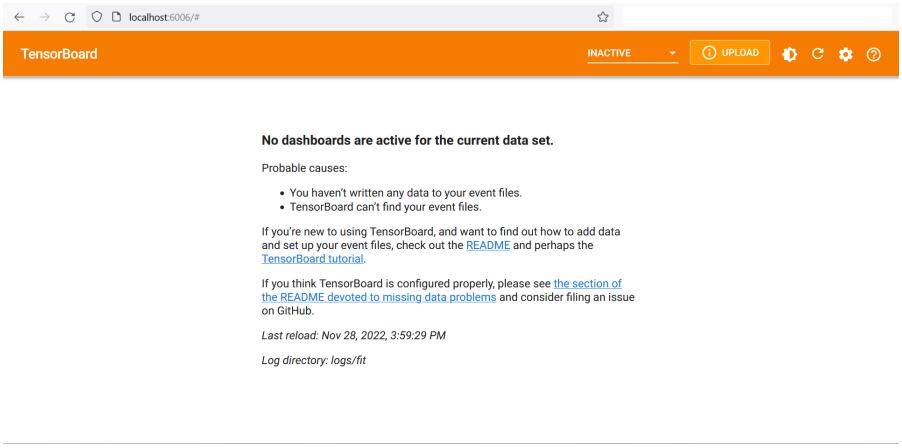


Figure 12.4 An example of TensorBoard dashboard

A brief overview of the dashboards is shown above. The `Scalars` dashboard shows how the loss and metrics change with every epoch. We can use Scalars to track training speed, learning rate and other scalar values. The `Graph` dashboard visualizes the model's computation graph. We can use `Graph` can be used to help to ensure the model is built correctly. The `Distributions` and `Histograms` dashboards show the distribution of a Tensor over time which can be used to visualize and verify weights and biases changing as expected. Let us define the folder to store the log data.

To use TensorBoard, we add `TensorBoard` callback to create and store the training logs.

```
from tensorflow.keras.callbacks import TensorBoard
train_callbacks = [EarlyStopping(monitor='val_loss', patience=10),
                   ModelCheckpoint(filepath='checkpoints/', monitor='val_loss',
                                   save_weights_only=True),
                   TensorBoard(log_dir=log_dir, histogram_freq=1)]
```

Train the model on the training data.

```
inputs = Input(shape=(41,))
hidden = Dense(units=20, activation='relu',
               kernel_regularizer=l2_regularizer)(inputs)
batchnorm = BatchNormalization()(hidden)
concat = Concatenate()([inputs, batchnorm])
outputs = Dense(units=1, activation='sigmoid')(concat)
model = Model(inputs=inputs, outputs=outputs)
model.compile(optimizer='adam', loss='binary_crossentropy',
              metrics=['accuracy'])
history = model.fit(x=X_train_norm, y=y_train, epochs=200, batch_size=64,
                    validation_split=0.2, callbacks=train_callbacks)
```

On the TensorBoard dashboard, click the refresh button to reload the dashboard. Choose `Scalar` from the dropdown menu or click `Scalar` button to show the training and validation loss and accuracy. Click `Graph` to show the model's computation graph. Figures 12.5 and 12.6 show `Scalar` and `Graph` dashboards.

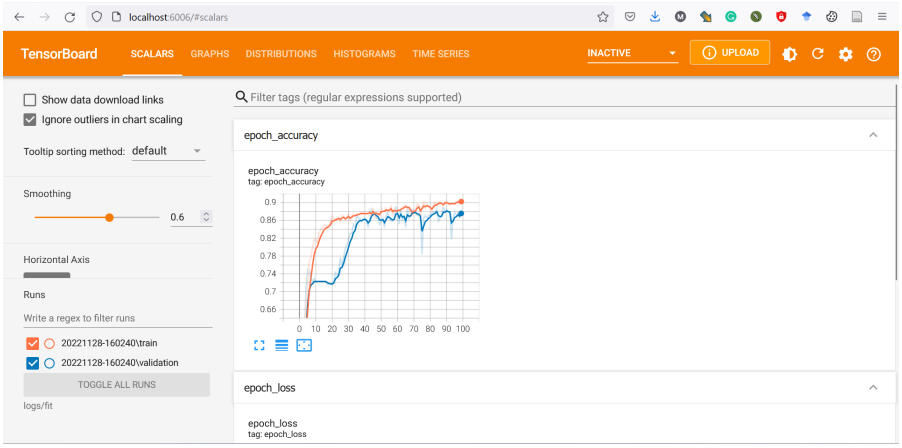


Figure 12.5 Scalar dashboard shows the training and validation loss and accuracy

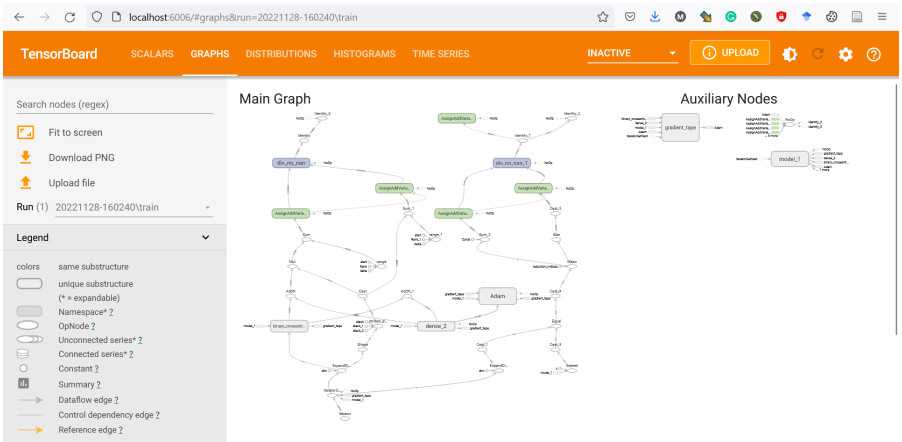


Figure 12.6 Graphs dashboard shows the model's computation graph

Exercises

1. Explain parameter norm penalty regularization.

2. Explain the role of λ in parameter norm penalty regularization.
3. Explain how does early stopping work as a form of regularization.
4. Explain the role of patience parameter in early stopping.
5. Explain dropout as a regularization technique in neural networks.
6. Explain concept behind batch normalization as a regularization strategy.

Chapter 13

Convolutional Neural Network

13.1 Introduction

Unstructured data such as images and time series contain rich information both in spatial and temporal domains. Take image data as an example, the spatial structure of the pixels defines the visual information of the data. Figure 13.1 illustrates an image with 7 rows and 6 columns of pixels. The black and white pixels are arranged to form a digit of 5. As has been described in section 2.2, to predict image data using machine learning algorithms, the 2-d input is transformed into a 1-d feature vector consisting of the pixel values. Specifically, a single pixel represents an input to the predictive model, which means the number of inputs is equal to the number of pixels.

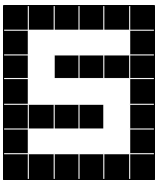


Figure 13.1 An image with 7 rows and 6 columns of black and white pixels

Although this approach seems feasible, the input data is noisy and high-dimensional, and increases the model complexity. For example, in a fully connected neural network, each unit in a layer is connected to each unit in the subsequent layer. By defining such a neural network, the number of weights will be tremendously high because we need a different weight for every single neural connection. Consider a black and white image of 100×100 being classified by a neural network as shown in Figure 13.2. The 2-d image data is flattened into a feature vector of size 10000. If there are 10000 units in the hidden layer, there will be 100 million neural connections between the input and hidden layers whereby each connection is having different weight parameter. Furthermore, since the 2-d input data is flattened into a 1-d vector, all spatial information is completely lost and cannot be leveraged by the neural network.

As we mentioned above, image data has rich spatial information which could be used in the prediction task. How do we leverage this information in the neural network? To leverage the spatial information, instead of taking all the pixels for processing, we define a small region of the image and process the pixels in that region. Consider an image as a 2-d array of pixel values where each pixel value is an input unit of the neural network as shown in Figure 13.3. A small region or called filter is defined, in this case, 3×3 pixels as indicated by the blue square. We connect the input pixels in

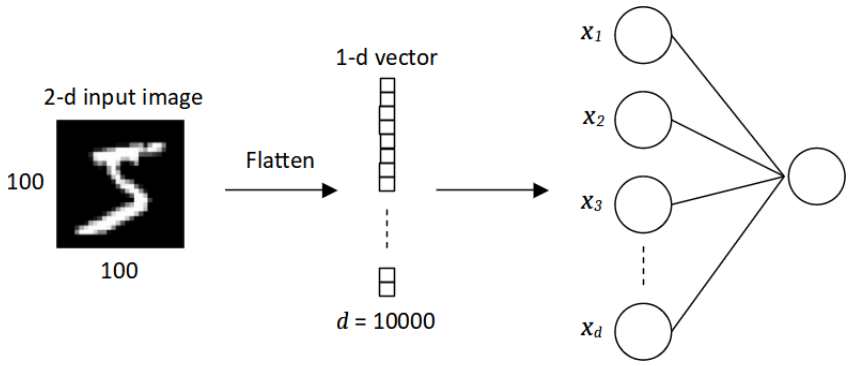


Figure 13.2 The spatial information is lost due to flattening and the number of input units of the neural network is equal to the number of pixels

the region to a single unit of the hidden layer. Similarly, each connection has a weight that defines the strength of the connection. Thus, essentially, the filter is a set of weights that are learned using the backpropagation algorithm.

We define the same neural connections across the whole input pixels as shown in the figure. To do this, the window is slid across the whole input image from the left edge to the right edge, followed by sliding the window downward until it reaches the bottom right corner of the image. In this way, the units of the hidden layer only see a particular region of the image which allows the network to extract features or details from the spatially closed pixels. Furthermore, the number of weights is reduced significantly since the units of the hidden layer are connected to only a fraction of the input pixels. Also, since the weights are shared across the input image, features that matter in one part of the image can be extracted by the filter in another part of the image. This filter operation is called convolution.

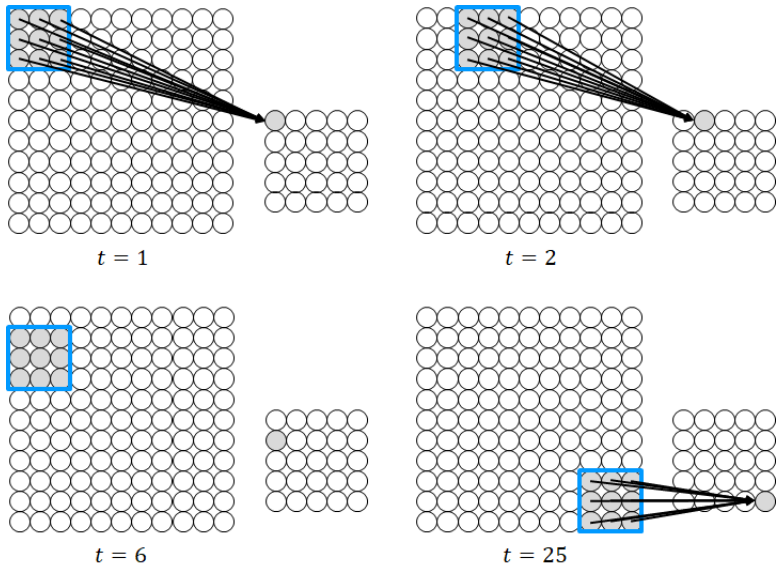


Figure 13.3 Patches of input pixels are connected to the units of the hidden layer

13.2 Feature Extraction and Convolution

Figure 13.4 demonstrates the motivation of feature extraction and how the features can be extracted by the convolution operation. The figure shows two images of a digit of five. The image on the

right contains noises, hence the digit is a bit distorted or deformed. If we are to classify the image by performing a pixel-by-pixel comparison, the right image will be misclassified. A better way to classify the images is to identify the features of a digit of five. A feature is a specific part of the image with a certain pattern that characterizes a particular object, which can be used to identify an object. For example, the features of a digit of five are indicated by the red boxes. Feature 1 and feature 2 characterize the corners while feature 3 characterizes the horizontal line of the digit. If these features can be detected on the deformed image roughly at the same location, we can conclude that the image is indeed a digit of five.

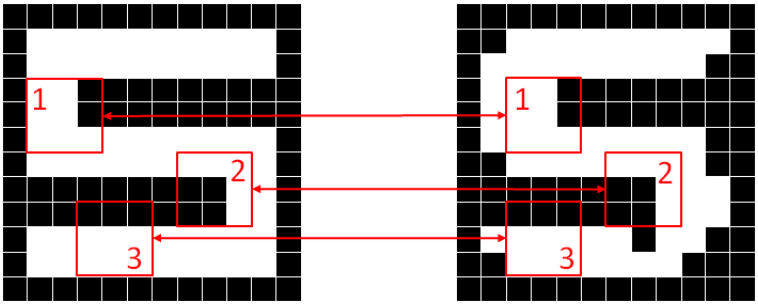


Figure 13.4 The features of the image are indicated by the red regions

Figure 13.5 illustrates an example of applying convolution on a 2-d input image. Convolution operation involves applying element-wise multiplication between the values of the filter and the pixel values of the small region, followed by taking the summation of the multiplications. As shown in the figure, the red boxes indicates the first convolution operation involving the filter and the region containing pixel a, b, e and f. The convolution operation on the input image produces an output of 2 × 3. The output of the convolution is known as feature map.

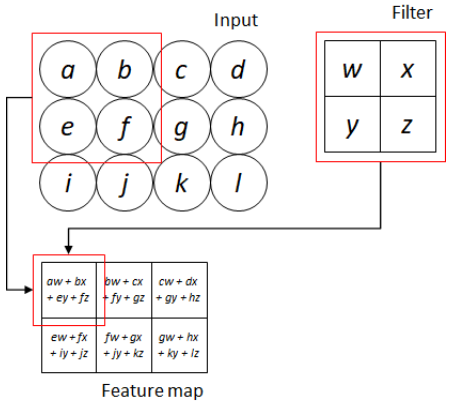
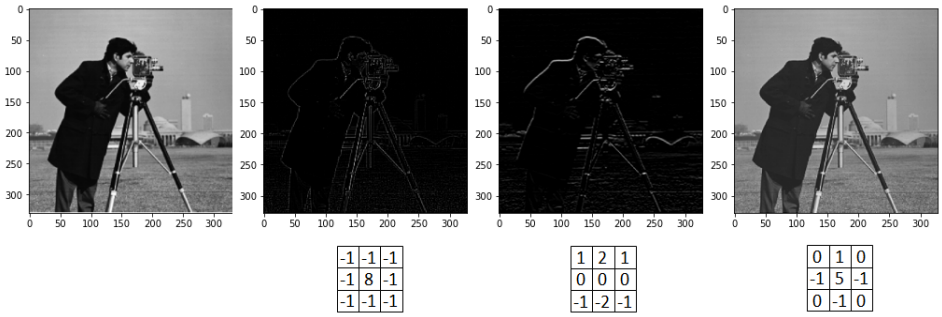


Figure 13.5 Convolution operation involving a filter of 2 × 2 and an image of 3 × 4. The output of the feature map indicated by the red box is produced by applying the filter to the corresponding top left region of the input image.

The feature map indicates wherein the input image, the feature has been detected by the filter i.e. high values of the feature map represent greater activation or detected features. By simply changing the weights of the filter, different feature maps can be produced as shown in Figure 13.6. As can be seen in the figure, three different filters have been applied to the original image on the left generating three different outputs.



-1	-1	-1
-1	8	-1
-1	-1	-1

1	2	1
0	0	0
-1	-2	-1

0	1	0
-1	5	-1
0	-1	0

Figure 13.6 The original image on the left is applied with three different filters. The first two images are applied with two different edge detection filters and the last image is sharpened with the sharpen filter.

13.3 Convolutional Neural Network

We have seen how convolution operation capitalizes spatial structure that is inherent in image data to detect and extract local features. This concept of preserving the spatial structure and feature extraction is at the core of convolutional neural networks (CNN). A CNN minimally consists of a convolutional layer, a pooling layer and a fully connected layer (Skansi 2018). The convolutional layer aims to extract discriminative local features directly from the image data. To extract the optimal features for the prediction, the weights of the filters are optimized during the backpropagation phase.

The purpose of the pooling layer is to downsample the size of the feature map to reduce the computational cost. Furthermore, a pooling layer suppresses the irrelevant information in the feature map to improve the performance of the prediction. We will discuss pooling layers in the following section. Lastly, the fully connected layer is the traditional neural network layer where each unit is connected to the units of the subsequent layer. The multi-dimensional feature maps are flattened into a 1-d feature vector which will be used as the input layer to the fully connected layer(s) for classification or regression.

Typically, a CNN has a series of alternating convolutional and pooling layers before the fully connected layers. The subsequent convolutional and pooling layers take the feature map previously generated by the convolutional and pooling layer as input and perform convolution operations to produce a feature map which in turn is reduced by the pooling operations. This allows the network to learn the features in a hierarchical manner, producing more salient features (Y. LeCun, Bengio, and G. Hinton 2015). Figure 13.7 illustrates an example of a CNN network consisting of two convolutional layers and two pooling layers in between, followed by two fully connected layers.

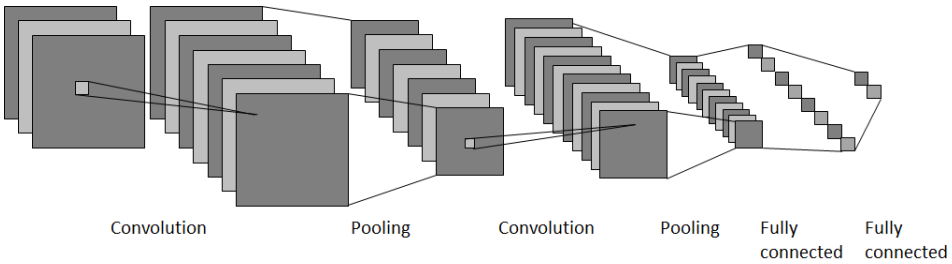


Figure 13.7 A CNN consists of convolutional, pooling and fully connected layers

13.4 Convolutional Layer

A CNN consists of convolutional, pooling and fully connected layers. Let l be the l-th layer where $l = 1$ is the first layer and L is the last layer. A convolutional layer consists of a set of different filters to extract different types of features as shown in Figure 13.8. Let $K \in \mathbb{R}^{k_1 \times k_2 \times C \times D}$ denotes the set of filters where k_1 and k_2 represent the height and width of the filters, C is the number of input channels and D is the number of filters or the depth of the feature maps (number of feature maps).

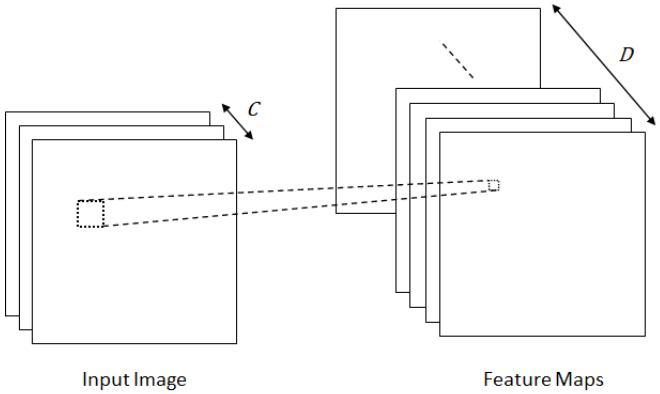


Figure 13.8 A convolutional layer with different filters to produce different feature maps

Given an input image with $C = 1$ channel (grayscale image) and a filter w of size $k_1 \times k_2$, the convolution operation produces feature map d as follows:

$$z_{i,j,d}^{(l+1)} = \sum_{m=0}^{k_1} \sum_{n=0}^{k_2} w_{m,n,d}^{(l)} \cdot a_{i+m,j+n,d}^{(l)} \tag{13.1}$$

$$a_{i,j,d}^{(l+1)} = g(z_{i,j,d}^{(l+1)}) \tag{13.2}$$

$$\tag{13.3}$$

where b is the bias, $z_{i,j,d}$ is the output of feature map d with i-th row and j-th column, g is an activation function applied to each output of the feature map.

In the case of color images with $C = 3$ channels, each channel has a filter of the same size but with a different set of weights as shown in Figure 13.9. The convolution operation is applied independently to each of the channels, yielding three sets of outputs which are summed to produce a single feature map. The convolution operation is defined as follows:

$$z_{i,j,d}^{(l+1)} = \sum_{m=1}^{k_1} \sum_{n=1}^{k_2} \sum_{c=1}^{C} w_{m,n,c,d}^{(l)} \cdot a_{i+m,j+n,c}^{(l)} + b \tag{13.4}$$

Based on the above definition, the size of the feature maps is determined by the filter size. Another parameter that affects the size of the feature map is the stride which is the number of pixels that the filter is shifted (slid) over the input along the width and height. Typically, the stride is set to 1. However, we can set stride to a larger value which results in a feature map with a smaller size. Consider a 9×9 input image convolved with 3×3 filter. The convolution operation with stride

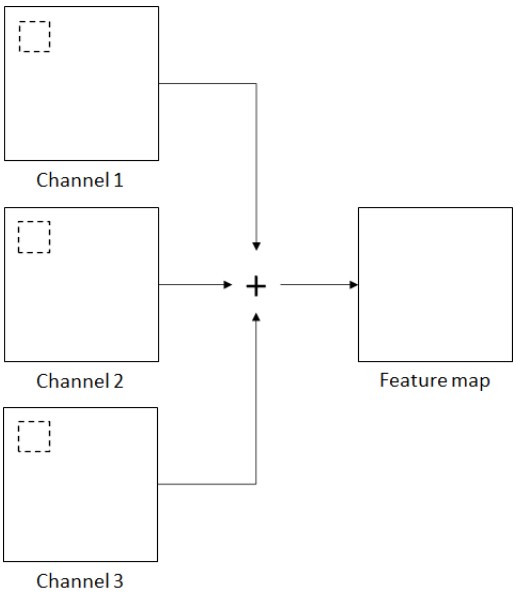


Figure 13.9 Applying convolution on multi-channel input produces a feature map

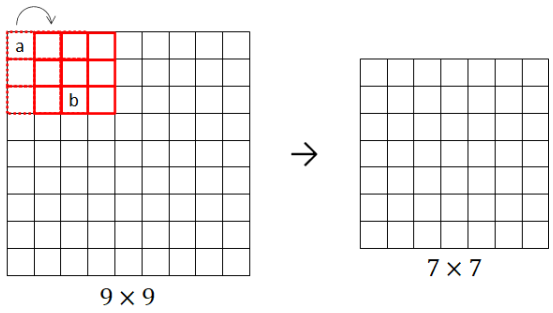
values of 1 and 2 produces feature maps of size 7 × 7 and 4 × 4, respectively, as shown in Figures 13.10(a) and 13.10(b).

We see that applying convolution operation with 3 × 3 filter and stride equals 1 to the 9 × 9 input image results in a feature map with the size of 7 × 7, a reduction of about 40% of the total input pixels. The sliding window from the top left corner to the bottom right corner results in not only a reduced feature map but also the corner and edge pixels of the input image having a minimal contribution to the output of the convolution operation. This is because the pixels are processed much less than those in the middle. As shown in Figure 13.10(a), pixel 'a' is used only in one convolution operation while pixel 'b' is used in nine convolution operations. Evidently, the information at the edges is not preserved as well as the information in the middle of the input image.

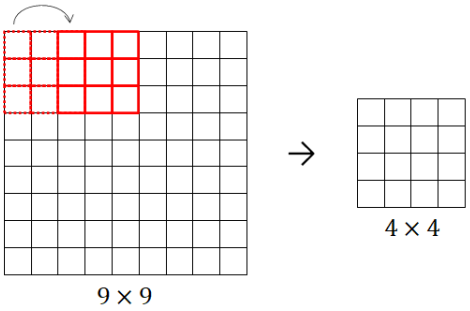
A solution to the problem is to pad the input image with extra zero pixels around its boundary. The padding allows the corner and edge pixels to be the middle pixels, thus increasing their contribution to the output of the convolution operation and consequently preserving the information at the edges of the input image. Furthermore, it produces a feature map with the same size as the input image (before the zero padding). Consider a 9 × 9 input image is padded with zero pixels as shown in Figure 13.11. As shown in the figure, there is one-pixel border which means the amount of zero padding is one. A filter of 3 × 3 with a stride equal to 1 is applied to the input image resulting in a feature map with a size of 9 × 9.

13.5 Pooling Layer

While the role of a convolutional layer is to extract features from the input data, the role of a pooling layer is to reduce the size of the feature maps by mapping patches of activations to a single activation value. The pooling operation works by defining a window that is similar to a filter and applying it to the feature map along both width and height. For each window, the pooling operation summarizes the presence of the feature values in the window. There are pooling operations which are computing the maximum values and averaging the values in the window. The maximum pooling returns the most prominent activation while the average pooling returns the average of the activations in the



(a)



(b)

Figure 13.10 Convolution with stride equal to 1 and 2, producing feature maps of (a) 7 × 7 and (b) 4 × 4 respectively

window. In general, maximum pooling (or max-pooling) is the most commonly used in CNN.

Typically, the window size or pool size is 2 × 2 with a stride equals 2 which means the window is not overlapping when it is slid from left to right and from top to bottom, reducing 75% of the activations of the feature map. Unlike convolutional layers, pooling layers operate independently on every slice of the feature maps which means a convolutional layer only spatially resizes while keeping the depth of the feature maps. Figure 13.12 shows a 4 × 4 feature map is reduced with a pool size of 2 × 2 and a stride of 2 using maximum and average pooling operations, producing a 2 × 2 feature map.

The use of pooling layers has several advantages. First, the dimension of the feature representation can be reduced, consequently reducing the computation time and the weights of the neural network. Secondly, a feature map contains the detected features and their positions in the input image. A slight

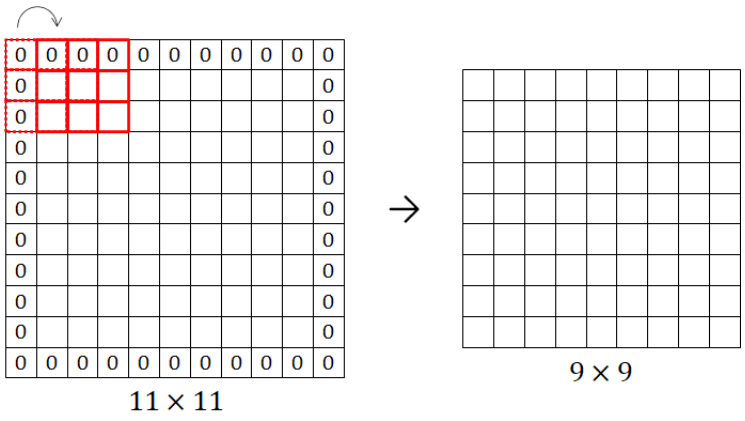


Figure 13.11 The boundary of the input image is padded with extra zero pixels

change to the positions of the features in the input image results in a different feature map. Pooling layers discard the positions of the detected features and keep only the prominent features, thereby producing translation invariance features that are more robust and reliable (Goodfellow, Bengio, and Courville 2017). In CNN, the convolutional layer and pooling layer are stacked alternately to extract and downsample features in a hierarchical manner. However, in a very deep neural network, the number of pooling layers is relatively lesser than the number of convolutional layers because the pooling operation can drastically reduce the size of the feature map. Thus, fewer pooling layers are required to reduce the feature map to the desired size.

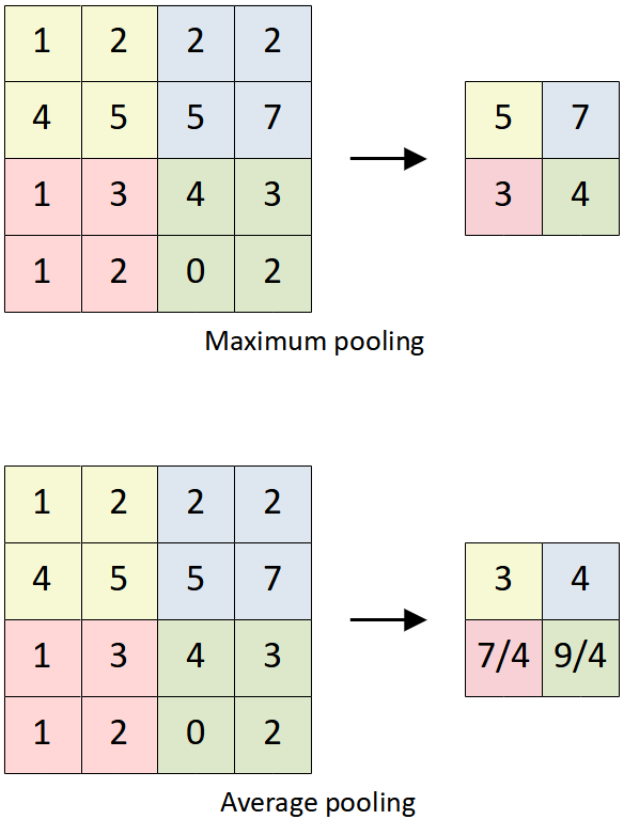


Figure 13.12 Maximum pooling returns the maximum activations while average pooling returns the average of the activations in the window

13.6 Fully Connected Layer

In a CNN, the series of convolutional and pooling layers is known as feature learning pipeline, which is used to learn the feature representation. The feature learning pipeline is followed by a series of fully connected layers for prediction. Typically, the last feature maps that are produced by the feature learning pipeline are flattened to a 1-d feature vector. The feature vector is then used as input to the fully connected layers by connecting each feature (activation) to each unit of the first hidden layer. These fully connected layers works exactly the same way as the traditional neural networks. One might apply the ReLU, sigmoid or tanh after each hidden layer. As for the output layer, the number of units and activation function depends on the specific requirements of the application. It is noteworthy that the fully connected layers are densely connected, thus the majority of the weights of the network lie in these layers. Although there is no recommended size

for the last feature maps, one might want to go as low as possible to reduce the overall number of weights.

13.7 Implementing Convolutional Neural Networks

In this section, we will be implementing CNN using TensorFlow. We will be using the flowers dataset. The flowers dataset consists of images of flowers with 5 possible class labels. Thus, the prediction task is a multiclass classification whereby to classify into Dandelion, Daisy, Tulips, Sunflowers and Roses. Now, let us download the dataset. We download the dataset using a utility function called `get_file`. The function needs three parameters `origin`, `fname` and `untar`. `origin` is used to specify the URL of the dataset, `fname` is used to specify the folder of the dataset and `untar` is used to extract the downloaded file. We import `pathlib` a module that can be used to represent the filesystem paths.

```
import tensorflow as tf
import pathlib
dataset_url = "https://storage.googleapis.com/download.tensorflow.org/
example_images/flower_photos.tgz"
data_dir = tf.keras.utils.get_file(origin=dataset_url,
                                   fname='flower_photos',
                                   untar=True)
data_dir = pathlib.Path(data_dir)
```

The flowers dataset contains five sub-folders, one per class.

```
flowers_photos/
|_daisy/
|_dandelion/
|_roses/
|_sunflowers/
|_tulips/
```

The dataset is located in the following folder.

```
print(data_dir)
```

```
C:\Users\trainer\.keras\datasets\flower_photos
```

TensorFlow provide `image_dataset_from_directory` to create the Dataset. The function has several parameters which can be set.

- `directory` is directory where the data is located.
- `labels` is either '`inferred`', `None` or a list/tuple of integer labels of the same size as the number of image files found in the directory. If '`inferred`', labels are generated from the directory structure. If `None`, there are no labels. The default value is '`inferred`'.
- `label_mode` describes the encoding of labels. The value is either '`int`', '`categorical`', '`binary`' or `None`. '`int`'means that the labels are encoded as integers, '`categorical`' means that the labels are encoded as a categorical vector, '`binary`' means that the labels are either 0 or 1 and `None` means no labels. By default the value is '`int`'.
- `batch_size` is the batch size of the data. By default, the value is 32.

- `image_size` is the size to resize images to after they are read from disk. The default size is (256, 256).
- `validation_split` is the fraction of data to reserve for validation. By default, the value is `None`.
- `subset` is the subset of the data to return. The value is either '`training`', '`validation`' or '`both`'. Only used if `validation_split` is set.
- `shuffle` is either `True` or `False`. The default is `True` which is to shuffle the data.

We use the function to split the dataset into training and validation sets. We define the parameters as follows:

```
batch_size = 32
img_height = 108
img_width = 150
img_size = (img_height, img_width)
vald_split = 0.3
```

First, we create the training set.

```
train_data = tf.keras.utils.image_dataset_from_directory(
    directory=data_dir, batch_size=batch_size, image_size=img_size,
    validation_split=vald_split, subset='training', shuffle=True, seed=0
)
```

```
Found 3670 files belonging to 5 classes.
Using 2569 files for training.
```

Then, we create the validation set.

```
vald_data = tf.keras.utils.image_dataset_from_directory(
    directory=data_dir, batch_size=batch_size, image_size=img_size,
    validation_split=vald_split, subset='validation', shuffle=True, seed=0
)
```

```
Found 3670 files belonging to 5 classes.
Using 1101 files for validation.
```

The dataset has five different classes.

```
class_names = train_data.class_names
print(class_names)
```

```
['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```

Let us visualize some of the images in the training set. The output is shown in Figure 13.13

```
from matplotlib import pyplot as plt
for images, labels in train_data.take(1):
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title(class_names[labels[i]])
        plt.axis("off")
```

Figure 13.13 Some of the images in flowers dataset

Now, let us build the CNN. We will be using the `Subclassing` API to build the model. Model `Subclassing` is fully-customizable and enables us to implement custom network architecture and forward pass of the model. To use `Subclassing`, we need to subclass the `Model` class and implement the constructor (`__init__`) and `call` method. In the constructor, we define the layers that we use to perform the computations in `call` method. Notice that we do not need to create the input layer. Unlike `Functional` API, we can implement prety much anything in `call` method such as looping, if else and any low-level Tensor operations. This allows us to create a complex non-linear model to experiment with.

```
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Rescaling, Dense
from tensorflow.keras.layers import Conv2D, MaxPool2D, Flatten

class CNN_Model(Model):
    def __init__(self):
        super(CNN_Model, self).__init__()
        self.rescale = Rescaling(1./255)
        self.conv1 = Conv2D(filters=8, kernel_size=(3,3),
                            padding='same', activation='relu')
        self.pool1 = MaxPool2D(pool_size=(2,2))
        self.conv2 = Conv2D(filters=16, kernel_size=(3,3),
                            padding='same', activation='relu')
        self.pool2 = MaxPool2D(pool_size=(2,2))
        self.conv3 = Conv2D(filters=32, kernel_size=(3,3),
                            padding='same', activation='relu')
        self.pool3 = MaxPool2D(pool_size=(2,2))
        self.conv4 = Conv2D(filters=64, kernel_size=(3,3),
                            padding='same', activation='relu')
        self.pool4 = MaxPool2D(pool_size=(2,2))
        self.conv5 = Conv2D(filters=128, kernel_size=(3,3),
                            padding='same', activation='relu')
        self.pool5 = MaxPool2D(pool_size=(2,2))
        self.flatten = Flatten()
        self.dense1 = Dense(units=128, activation='relu')
        self.dense2 = Dense(units=64, activation='relu')
        self.dense3 = Dense(units=5, activation='softmax')

    def call(self, x):
        x = self.rescale(x)
        x = self.conv1(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = self.pool2(x)
        x = self.conv3(x)
        x = self.pool3(x)
        x = self.conv4(x)
        x = self.pool4(x)
        x = self.conv5(x)
        x = self.pool5(x)
        x = self.flatten(x)
        x = self.dense1(x)
        x = self.dense2(x)
        out = self.dense3(x)
        return out
```

We create an object of `CNN_Model` and use `build` method to specify the shape of the input. The input shape is `(None, img_height, img_width, channel)`.

```
model = CNN_Model()
model.build(input_shape=(None,) + img_size + (3,))
model.summary()
```

```
Model: "cnn__model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 rescaling (Rescaling)       multiple                  0

 conv2d (Conv2D)             multiple                  224

 max_pooling2d (MaxPooling2D)  multiple                0

 conv2d_1 (Conv2D)           multiple                  1168

 max_pooling2d_1 (MaxPooling2D)  multiple              0

 conv2d_2 (Conv2D)           multiple                  4640

 max_pooling2d_2 (MaxPooling2D  multiple               0

 conv2d_3 (Conv2D)           multiple                  18496

 max_pooling2d_3 (MaxPooling2D  multiple               0

 conv2d_4 (Conv2D)           multiple                  73856

 max_pooling2d_4 (MaxPooling2D  multiple               0

 flatten (Flatten)           multiple                  0

 dense (Dense)               multiple                  196736

 dense_1 (Dense)             multiple                  8256

 dense_2 (Dense)             multiple                  325

=================================================================
Total params: 303,701
Trainable params: 303,701
Non-trainable params: 0im}
```

Notice that summary shows the layers without the shape of the output. This is because the forward pass is hidden in `call` method. Thus, it will be more difficult to analyze the model architecture especially if the network is large and complex.

TensorFlow provides two cross entropy loss function for multiclass classification: `CategoricalCrossentropy` and `SparseCategoricalCrossentropy`. `CategoricalCrossentropy` expects the labels in a one-hot representation. If the labels are provided as integers, use `SparseCategoricalCrossentropy`. Since our labels are integers, we use `SparseCategoricalCrossentropy`. Then, we compile the model by specifying the optimizer, the loss function and the metrics.

```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.Adam()
model.compile(optimizer=optimizer, loss=loss_fn, metrics=['accuracy'])
```

We train the model using the training set.

```
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
train_callbacks = [EarlyStopping(patience=10, monitor='val_loss'),
                   ModelCheckpoint(filepath='checkpoints/', monitor='val_loss',
    save_weights_only=True)]
model.fit(train_data, validation_data=vald_data, epochs=100, callbacks=train_callbacks)
```

13.8 Creating Custom Layers

TensorFlow provides many common layers for building a deep learning model. But in some cases, we might need to define our own application-specific layers or a block of layers in sequence. `Layer` module give us the flexibility to build a custom layer which can be used in `Model`.

For example, let us build a custom layer consisting of two convolution layers followed by a max-pooling layer. We need to import `Layer` module and subclass the module.

```
from tensorflow.keras.layers import Layer
```

Let us define the layer. The constructor accepts nine parameters which are the number of filters, kernel size, strides, padding of the convolutional layers and the pooling size of the max-pooling layer.

```
class DoubleConv(Layer):
    def __init__(self, n_filters1, n_filters2, k1, k2, s1, s2, pad1, pad2, pool_size):
        super(DoubleConv, self).__init__()
        self.conv1 = Conv2D(filters=n_filters1, kernel_size=(k1,k1),
                            strides=(s1,s1), padding=pad1,
                            activation='relu')
        self.conv2 = Conv2D(filters=n_filters2, kernel_size=(k2,k2),
                            strides=(s2,s2), padding=pad2,
                            activation='relu')
        self.max_pool = MaxPool2D(pool_size=(pool_size,pool_size))

    def call(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.max_pool(x)
        return x
```

Now, let us use the custom layer in the CNN model.

```
class CNN_Model_2(Model):
    def __init__(self):
        super(CNN_Model_2, self).__init__()
        self.rescale = Rescaling(1./255)

        self.double_conv1 = DoubleConv(n_filters1=8, n_filters2=8, k1=1, k2=3,
                                       s1=1, s2=1, pad1='same', pad2='same',
                                       pool_size=2)
        self.double_conv2 = DoubleConv(n_filters1=16, n_filters2=16, k1=1, k2=3,
                                       s1=1, s2=1, pad1='same', pad2='same',
                                       pool_size=2)
        self.double_conv3 = DoubleConv(n_filters1=32, n_filters2=32, k1=1, k2=3,
                                       s1=1, s2=1, pad1='same', pad2='same',
                                       pool_size=2)
        self.double_conv4 = DoubleConv(n_filters1=64, n_filters2=64, k1=1, k2=3,
                                       s1=1, s2=1, pad1='same', pad2='same',
                                       pool_size=2)
        self.double_conv5 = DoubleConv(n_filters1=128, n_filters2=128, k1=1, k2=3,
                                       s1=1, s2=1, pad1='same', pad2='same',
                                       pool_size=2)
        self.flatten = Flatten()
        self.dense1 = Dense(units=128, activation='relu')
        self.dense2 = Dense(units=64, activation='relu')
        self.dense3 = Dense(units=5, activation='softmax')

    def call(self, x):
        x = self.rescale(x)
        x = self.double_conv1(x)
        x = self.double_conv2(x)
        x = self.double_conv3(x)
        x = self.double_conv4(x)
        x = self.double_conv5(x)
        x = self.flatten(x)
        x = self.dense1(x)
        x = self.dense2(x)
        out = self.dense3(x)
        return out
```

13.9 Creating Input Pipelines

`image_dataset_from_directory` allows the dataset to be split into training and validation sets. However, there is no option to create the test set. To create the test set, we need to manually create a separate folder for test and load the data using the function. TensorFlow provides a module `tensorflow.data.Dataset` which allows for finer-grain control over the input pipeline. First, we get all the path to the image files. `glob` method looks for files matching the given pattern.

```
data_path = list(data_dir.glob('*/*.jpg'))
```

Let us import `Dataset` from `tensorflow.data`. We create `Dataset` object and specify the source of the data i.e. the path of the data folder. `Dataset` object contains all the paths to the images.

```
from tensorflow.data import Dataset
list_ds = tf.data.Dataset.list_files(str(data_dir/'*/*'), shuffle=True, seed=0)
```

We can take five paths and display them. The output shows the path to the files.

```
for f in list_ds.take(5):
    print(f)
```

The dataset is structured as shown in Figure 13.14.

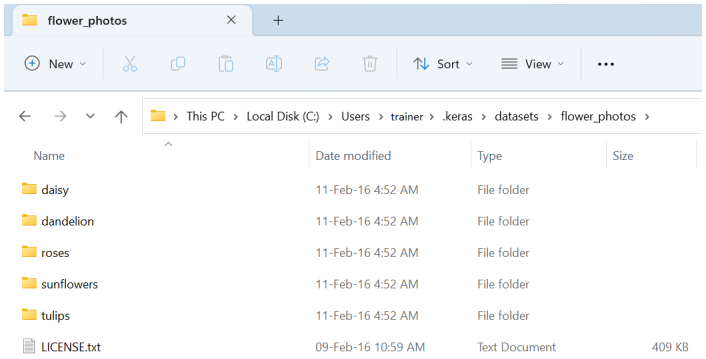


Figure 13.14 The folder structure of the dataset

We get the class names from the tree structure of the files.

```
import numpy as np
class_names = []
folder_path = data_dir.glob('*')
for path in folder_path:
    if path.is_dir():
        class_names.append(path.name)
print(class_names)
```

```
['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```

We split the dataset into training and validation sets. The dataset has 3670 images.

```
data_count = len(data_path)
print(data_count)
```

```
3670
```

We put aside 10% and 20% of the dataset as the validation set and test set respectively.

```
vald_size = int(data_count * 0.1)
test_size = int(data_count * 0.2)
train_size = data_count - (vald_size + test_size)
print(train_size)
print(vald_size)
print(test_size)
```

```
2569
367
734
```

We split the dataset using `skip` and `take` methods. `skip` method creates a subset that skips N elements of the dataset. `take` method creates a subset by retrieving N elements from the dataset. First, we create the test set by skipping `data_count-test_size` elements of the dataset. The skipped elements are allocated for training and validation sets.

```
test_ds = list_ds.skip(data_count-test_size)
train_vald_ds = list_ds.take(data_count-test_size)
```

Using `train_vald_set`, we create the validation set by skipping `train_size` elements and skipped elements are for training set.

```
vald_ds = train_vald_ds.skip(train_size)
train_ds = train_vald_ds.take(train_size)
print(len(train_ds))
print(len(vald_ds))
print(len(test_ds))
```

```
2569
367
734
```

Next, we create functions to convert file path (in dataset) to `(image, label)` pair. The first function is to get the label from the file path. The first line splits the path into path components. The second line takes the directory name (class name) and converts it to one-hot form. The next function reads the image file, decodes the image and resizes it to the desired size. The last function uses `get_label` and `decode_img` functions to create the `(image, label)` pair.

```
import os
def get_label(file_path):
    parts = tf.strings.split(file_path, os.path.sep)
    one_hot = parts[-2] == class_names
    return tf.argmax(one_hot)

def decode_img(file_path):
    img = tf.io.read_file(file_path)
    img = tf.io.decode_jpeg(img, channels=3)
    return tf.image.resize(img, [img_height, img_width])

def process_path(file_path):
    label = get_label(file_path)
    img = decode_img(file_path)
    return img, label
```

We use `map` method of `Dataset` to create a dataset of (image,label) pairs.

```
train_ds = train_ds.map(process_path, num_parallel_calls=tf.data.AUTOTUNE)
vald_ds = vald_ds.map(process_path, num_parallel_calls=tf.data.AUTOTUNE)
test_ds = test_ds.map(process_path, num_parallel_calls=tf.data.AUTOTUNE)
```

Now, if we take sample(s) from the dataset, we will get the data and its label (X,y).

```
for image, label in train_ds.take(1):
    print("Image shape: ", image.numpy().shape)
    print("Label: ", label.numpy())
```

```
Image shape:  (108, 150, 3)
Label:  2
```

We configure the dataset i.e. shuffle the samples and set the batch size.

```
def configure_dataset(ds, batch_size):
    ds = ds.cache()
    ds = ds.shuffle(buffer_size=30)
    ds = ds.batch(batch_size, drop_remainder=True)
    ds = ds.prefetch(buffer_size=tf.data.AUTOTUNE)
    return ds

train_ds = configure_dataset(train_ds, 32)
vald_ds = configure_dataset(vald_ds, 32)
test_ds = configure_dataset(test_ds, 32)
```

We build the CNN model.

```
model = CNN_Model()
model.build(input_shape=(None,) + img_size + (3,))
```

We define the optimizer and the loss function and compile the model.

```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()
optimizer = tf.keras.optimizers.Adam()
model.compile(optimizer=optimizer, loss=loss_fn, metrics=['accuracy'])
```

We train the model using the training set.

```
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
train_callbacks = [EarlyStopping(patience=10, monitor='val_loss'),
                   ModelCheckpoint(filepath='checkpoints/', monitor='val_loss',
                                   save_weights_only=True)]
model.fit(train_ds, validation_data=vald_ds, epochs=100, callbacks=train_callbacks)
```

We use `evaluate` method to evaluate the model using the test set.

```
loss, acc = model.evaluate(test_ds)
```

```
22/22 [==============================] - 1s 43ms/step - loss: 0.6501
 - accuracy: 0.7386
```

If we want to get the predicted values, we need to implement a loop and use `predict` method.

```
y_test = np.empty((0,))
y_pred = np.empty((0,))
for image_batch, label_batch in test_ds:
    pred = model.predict(image_batch)
    pred = np.argmax(pred, axis=1)
    y_pred = np.append(y_pred, pred)
    y_test = np.append(y_test, label_batch.numpy())
```

Then, we can display the confusion matrix and classification by passing the predicted labels.

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report
print(accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
0.9076704545454546
[[107   3   1   2   1]
 [  7 160   5   3   2]
 [  4   4  98   3  11]
 [  0   4   0 135   0]
 [  2   0  12   1 139]]
              precision    recall  f1-score   support

         0.0       0.89      0.94      0.91       114
         1.0       0.94      0.90      0.92       177
         2.0       0.84      0.82      0.83       120
         3.0       0.94      0.97      0.95       139
         4.0       0.91      0.90      0.91       154

    accuracy                           0.91       704
   macro avg       0.90      0.91      0.90       704
weighted avg       0.91      0.91      0.91       704
```

The classification report shows that the model performed well in classifying the data with F-score above 0.90 for all classes except one. The accuracy of the model is 0.91.

Exercises

1. Explain how does CNN differ from fully connected neural networks.
2. Explain the different layers in CNN.
3. Explain the purpose of using zero padding in CNN.
4. Explain the effect of high stride on the feature map.
5. Consider an input image of size 12×12. Determine the size of the feature maps if a filter of size 3×3 with a stride of 1 is applied to the input image.
6. Explain the purpose of a pooling layer in CNN.
7. Explain the different types of pooling layer and their characteristics.
8. Explain the role of the fully-connected layer in CNN.
9. Calculate the convolution operation between the following input image and kernel.

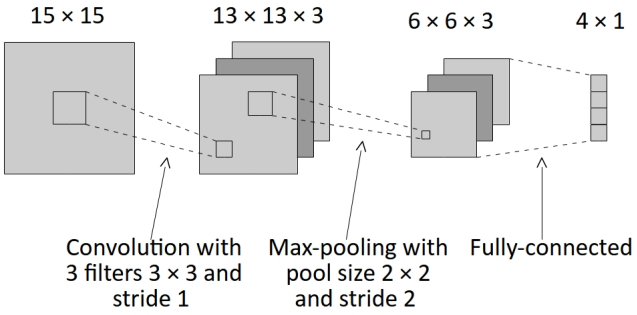
2	0	1	0
0	1	0	1
0	2	1	0
2	1	0	0

Image

0	1	0
1	1	2
0	2	0

Kernel

10. Consider a small CNN that predict a 15×15 image into 4 classes. The network has a convolutional layer, a max-pooling layer followed by a fully-connected layer. For this network a bias is used for the convolutional and fully-connected layer.



(a) Determine the number of weights in the convolutional layer.

(b) Determine the feature vector.

(c) Determine the number of weights of the entire network.

Chapter 14

Recurrent Neural Network

14.1 Introduction

The artificial neural network architectures that have been discussed in the previous chapters are designed for 1-d and 2-d input data such as vectors and images. However, certain data types such as text and those generated by sensors contain sequential dependencies over time. A data point is dependent on data points that come before and after it. This can be seen in text data for any language, a sentence is constrained by the grammar of the language. The sequence of the words in a sentence exhibits a significant sequential correlation. For example, "*I like playing football*", the word that comes after playing should be a sport, game or other similar nouns and it is very unlikely the following word is another verb. Thus, it is important that this sequential pattern or temporal structure is explored to improve the accuracy of the prediction. In the following section, we will discuss a specialized type of artificial neural network for processing a sequence of data. The neural network is called a recurrent neural network (Rumelhart, Geoffrey E. Hinton, and Williams 1986).

14.2 Neural Network with Feedback Loops

First, let us recall the artificial neural network. A neural network accepts an arbitrary number of inputs whereby each input is multiplied by a weight and the resultants are summed and passed to a non-linear activation function to produce an internal state of the neural network. The internal state is then fed to the subsequent layers to produce the predictive output. Now, consider a sequential dataset as shown in Table 14.1. The dataset has four input features, a target and the first column indicates the time the instances are collected which is every hour. Considering this is a sequential data, the target at a later time is mostly dependent on the inputs of the prior time.

Table 14.1 A sequential dataset

t, hour	x_1	x_2	x_3	x_4	y
8.00	0.62	0.13	0.35	0.82	0
9.00	0.58	0.15	0.39	0.79	0
...	...	...	...	...	...
23.00	0.49	0.74	0.63	0.36	1

To predict the target is a straightforward task for an artificial neural network. We feed the input features to the neural network and the signals are propagated through the hidden layers to the output layer. Since, the data is time series, we can repeat the same steps for the subsequent time

steps (rows) to predict the output. Figure 14.1 illustrates this approach whereby three input time steps are fed to the neural network. Note that the same neural network is used for each time step. At each time step, the input is given to the neural network and the predictive output is generated at that particular time step.

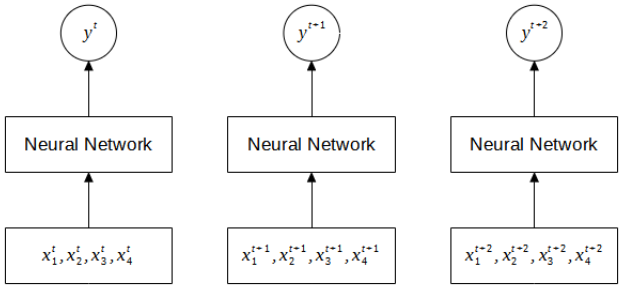


Figure 14.1 The sequence of input time steps is fed to the neural networks separately

Although the neural network is able to learn the mapping between the features and the target, this approach assumes the data is non-sequential. This is because the neural network treats the instances as an individual isolated input, and consequently the relationship between the instances that is inherent in the sequence is not learned. To address this limitation, a neural network at time t should be able to pass its computation (state) to the later time steps allowing the information to be used to generate the predictive outputs as shown in Figure 14.2. As we can see in the figure, each predictive output is a function of both the current input and the state from the previous time step. We can describe this model as a neural network with a recurrent connection as shown in Figure 14.2 (on the right).

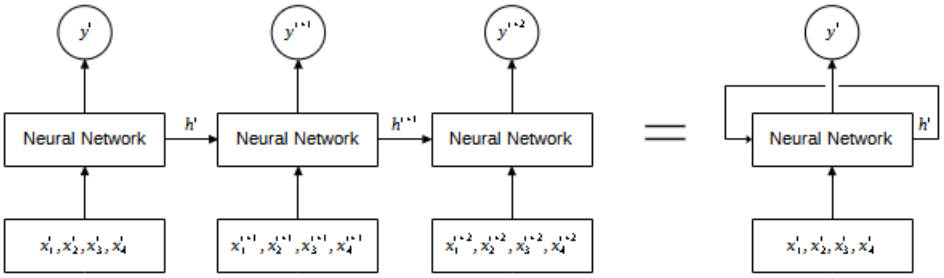


Figure 14.2 The neural networks are linked to the subsequent neural networks allowing information to be passed from one time step to the next time step

The recurrent neural network (RNN) is characterized by the recurrent connection (loop) which passes the information from one time step to another time step. The recurrent connection defines how the internal state of the neural network at each time step is updated. Specifically, the internal state is parameterized by a set of weights which will be learned as the sequences of data is processed over the course of training. Note that the weights are shared across the sequence which means the same weights are to be applied to each individual time step. Figure 14.3 illustrates the internal structure of a RNN.

Now, let us look at the forward propagation of the RNN. Given a sequence of data $\mathbf{x}^t$, the output

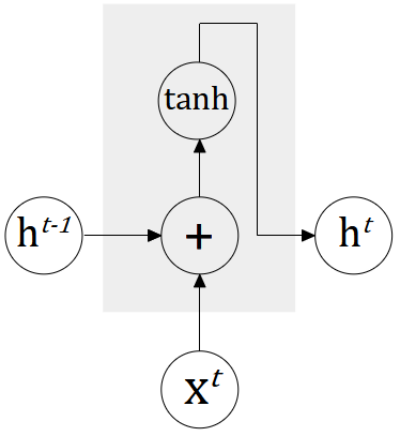


Figure 14.3 The internal structure of a RNN

for each time step $t = 1$ to $t = T$ is given as follows:

$$\hat{y}^t = g(\mathbf{o}^t) \tag{14.1}$$

$$\mathbf{o}^t = \mathbf{w}^\top \mathbf{h}^t + \mathbf{c} \tag{14.2}$$

$$\mathbf{h}^t = f(\mathbf{z}^t) \tag{14.3}$$

$$\mathbf{z}^t = \mathbf{u}^\top \mathbf{x}^t + \mathbf{b}_x + \mathbf{v}^\top \mathbf{h}^{t-1} + \mathbf{b} \tag{14.4}$$

where $\mathbf{h}^t$ is the hidden state of the RNN at time t, $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$ are the weight matrices for input-to-hidden, hidden-to-hidden and hidden-to-output connections, $\mathbf{b}$ and $\mathbf{c}$ are the biases and f is typically tanh activation function.

The above example is a RNN that maps an input sequence to an output sequence of the same length. This type of sequence modeling is known as many-to-many and a common application is machine translation. However, RNN are flexible as the length of input and output can be configured to many-to-one and one-to-many as shown in Figure 14.4. Many-to-one takes a sequence of input and produces a single output. This type of sequence modeling is common in sentiment analysis where we want to determine the emotional tone of a document or a collection of sentences. One-to-many takes a single input and produces a sequence of output and a typical example of this modeling is image captioning. Given an input image, we want to generate a textual description of the image.

The training of RNN is considerably more complex and difficult compared to standard neural networks. The total loss for many-to-one modeling is calculated based on the single output prediction. In the case of many-to-many and one-to-many, the total loss for a given input or sequence of input paired with the sequence of output is the sum of the losses over all the time steps. Computing the gradient of the loss function with respect to the weights is computationally expensive. The gradient computation begins with the forward propagation of the input sequence from left to right, followed by the backward propagation from right to left through the network. This learning algorithm is called backpropagation through time (Rumelhart and McClelland 1987; Werbos 1988). A major challenge in the training of RNN is that the propagation of the gradients over many time steps may vanish (Hochreiter, Bengio, et al. 2001).

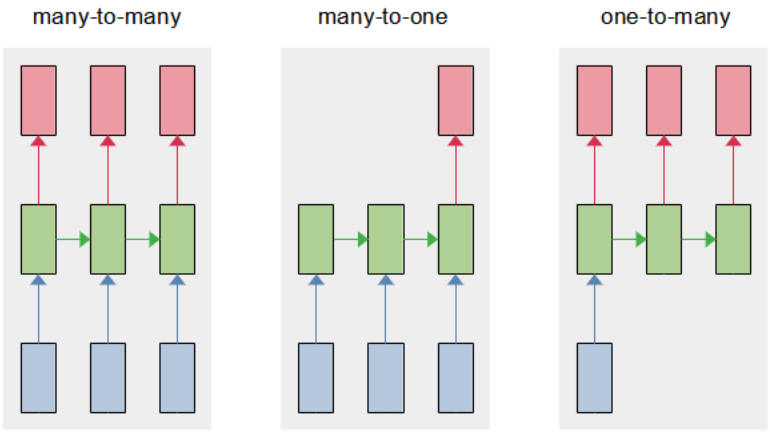


Figure 14.4 Types of sequence modeling

14.3 Long Short-term Memory

Long short-term memory (LSTM) is a variant of RNN that was designed to overcome the problem of vanishing gradient when learning long-term dependencies (Hochreiter and Schmidhuber 1997). LSTM is arguably the most widely used variant and it is applicable in various applications such as image captioning, machine translation, sentiment analysis and healthcare (Van Houdt, Mosquera, and Nápoles 2020). Similar to the standard RNN, LSTMs have the form of a chain of repeating units. An LSTM unit consists of several gates that control the passing of information from one time step to another time step: forget gate, input gate and output gate. Figure 14.5 illustrates the flow of information and the interconnection of the gates. The key element in an LSTM unit is the cell $\mathbf{c}^t$ which remembers the information (state) at each time step. The cell is indicated by the horizontal line running through the top of the block. The cell state is updated by controlling the flow of information through the gates.

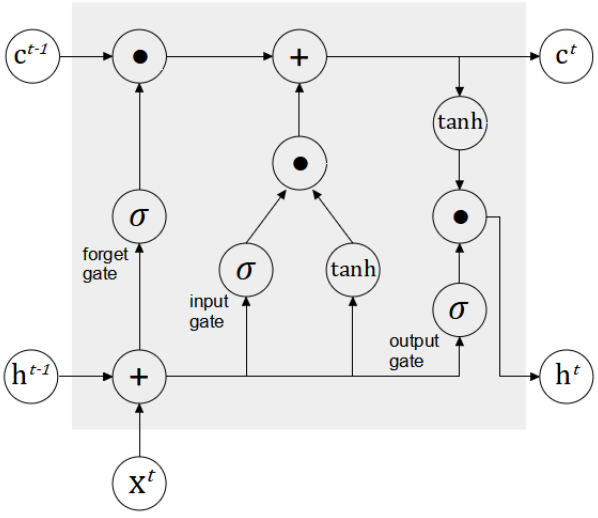


Figure 14.5 The architecture of an LSTM unit

The forget gate accepts the input $\mathbf{x}^t$ and the previous hidden state $\mathbf{h}^{t-1}$, and determines which information should be filtered out from the previous cell state $\mathbf{c}^{t-1}$. This operation is performed by

a sigmoid activation function, which outputs the value in the range of 0 (filter out) and 1 (keep).

$$\mathbf{f}^t = \sigma(\mathbf{u}_f^\top \mathbf{x}^t + \mathbf{b}_{xf} + \mathbf{v}_f^\top \mathbf{h}^{t-1} + \mathbf{b}_{hf}) \tag{14.5}$$

where $\mathbf{u}_f$ and $\mathbf{v}_f$ are weight matrices that are learned.

The input gate is responsible for storing information in the current cell state $\mathbf{c}^t$ based on the current input $\mathbf{x}^t$ and the output of the previous time step $\mathbf{h}^{t-1}$. The operation involves activation functions, sigmoid and tanh. The sigmoid activation function decides which relevant information to be retrieved and the tanh activation function creates the candidate vector which will be used to update the cell state.

$$\mathbf{i}^t = \sigma(\mathbf{u}_i^\top \mathbf{x}^t + \mathbf{b}_{xi} + \mathbf{v}_i^\top \mathbf{h}^{t-1} + \mathbf{b}_{hi}) \tag{14.6}$$

$$\mathbf{g}^t = \texttt{tanh}(\mathbf{u}_g^\top \mathbf{x}^t + \mathbf{b}_{xg} + \mathbf{v}_g^\top \mathbf{h}^{t-1} + \mathbf{b}_{hg}) \tag{14.7}$$

The cell state is updated via the following computation.

$$\mathbf{c}^t = \mathbf{f}^t \odot \mathbf{c}^{t-1} + \mathbf{i}^t \odot \mathbf{g}^t \tag{14.8}$$

where $\odot$ is element-wise multiplication.

The output gate accepts the current input $\mathbf{x}^t$ and the output of the previous time step $\mathbf{h}^{t-1}$ and combines them with the cell state $\mathbf{c}^t$. The computation of the output $\mathbf{h}^t$ is given as follows:

$$\mathbf{o}^t = \sigma(\mathbf{u}_o^\top \mathbf{x}^t + \mathbf{b}_{xo} + \mathbf{v}_o^\top \mathbf{h}^{t-1} + \mathbf{b}_{ho}) \tag{14.9}$$

$$\mathbf{h}^t = \texttt{tanh}(\mathbf{c}^t) \odot \mathbf{o}^t \tag{14.10}$$

14.4 Gated Recurrent Unit

Gated recurrent unit (GRU) is another popular variant of RNN. It is similar to LSTM whereby gating mechanism are used to control the flow of information from one time step to another time step. However, GRU has a relatively simpler architecture whereby it has two gates and does not store the state of each time step. Hence, the network has fewer parameters and is faster in terms of computation and training (Chung et al. 2014). A GRU unit has two gates: reset gate and update gate. Figure 14.6 illustrates the interconnection of the gates.

The reset gate takes the previous hidden state $\mathbf{h}^{t-1}$ and decides which information to ignore and process with the current input $\mathbf{x}^t$ to update the hidden state.

$$\mathbf{r}^t = \sigma(\mathbf{u}_r^\top \mathbf{x}^t + \mathbf{b}_{xr} + \mathbf{v}_r^\top \mathbf{h}^{t-1} + \mathbf{b}_{hr}) \tag{14.11}$$

The update gate is computed similarly to the reset gate. However, it has a different usage whereby it controls how much information will be carried over from the previous hidden state to the current hidden state during the hidden state update.

$$\mathbf{z}^t = \sigma(\mathbf{u}_z^\top \mathbf{x}^t + \mathbf{b}_{xz} + \mathbf{v}_z^\top \mathbf{h}^{t-1} + \mathbf{b}_{hz}) \tag{14.12}$$

Then, the hidden state at time t is updated as follows:

$$\mathbf{h}^t = \mathbf{z}^t \odot \mathbf{n}^t + (1 - \mathbf{z}^t) \odot \mathbf{h}^{t-1} \tag{14.13}$$

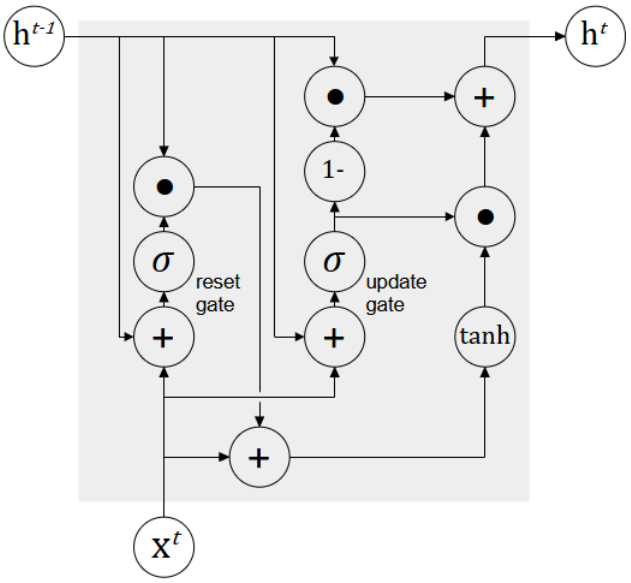


Figure 14.6 The architecture of a GRU

where $\mathbf{n}^t$ is given by

$$\mathbf{n}^t = \texttt{tanh}(\mathbf{u}_n^\top \mathbf{x}^t + \mathbf{v}_n^\top (\mathbf{r}^t \odot \mathbf{h}^{t-1}) + \mathbf{b}_n) \tag{14.14}$$

14.5 Similarities and Differences Between LSTM and GRU

Gated RNN such as LSTM and GRU are capable of learning long dependencies due to the additive component to update the hidden state from $t-1$ to t. Unlike RNN which replaces the hidden state with a new value computed from the current input and previous hidden state, LSTM and GRU update the current hidden state by adding the existing value with a new value. The additive component comes with several advantages. First, any important features that are extracted by the gating mechanism are not overwritten due to the addition operation. Thus, it is easy for each LSTM or GRU unit to store important features in the long sequence of inputs. Secondly, the additive component creates a direct connection between temporal units. This allows the error to be backpropagated easily to the earlier temporal units, without passing through the non-linear activation functions which may cause the vanishing gradient.

Despite their similarities, LSTM and GRU have a few differences. Both LSTM and GRU units control the amount of information flowing from the previous time step. However, the GRU unit does not independently control the new candidate activation while the LSTM unit directly controls the new candidate activation being added to the cell state. LSTM unit controls the amount of information to be passed to the next unit by the output gate. GRU unit fully exposes the hidden state to the next unit without any control.

14.6 Using TensorFlow Datasets

In this section, we will be creating the input pipeline for the RNN which we will be implementing in the next section. We will be using a dataset from TensorFlow Datasets (TFDS) library. TFDS provides a collection of ready-to-use datasets for use with TensorFlow. The list of datasets can be

found at `https://www.tensorflow.org/datasets/catalog/overview`.

Let us install TFDS library. Type the following command to install using `pip`.

```
pip install tensorflow-datasets
```

Type the following command to install using `conda`.

```
conda install -c conda-forge tensorflow-datasets
```

Now, import `tensorflow_datasets` as `tfds`.

```
import tensorflow_datasets as tfds
```

We will be using Sentiment140 dataset. Sentiment140 is a dataset containing tweets of brand, product, or topic on Twitter. The data format has 6 fields as follows:

- the polarity of the tweet (0 = negative, 4 = positive)
- the id of the tweet
- the date of the tweet
- the query (lyx). If there is no query, then this value is NO_QUERY.
- the user that tweeted
- the text of the tweet

To download the dataset, we use load method of `tfds`. The method has several parameters as follows:

- `with_info` is the parameter to get the information of the dataset. By default, the value is `False`.
- `as_supervised` is either `True` or `False`. If the value is `True`, the returned dataset will have a 2-tuple structure (input, label). However, for some dataset, the parameter is `None`. By default, the value is `False`.
- `split` is used to indicate which split of the data to load. By default, the dataset has been split into training and test sets and the method will load both sets. This parameter can be specified by passing either `'train'`, `'test'` or `['train','test']`. Furthermore, we can also specify the portion of each set to load such as `train[:25%]` which means, the first 25% of training data will be loaded. We can load the last 25% of training data by passing `train[25%:]`. We can also combine the samples of training and test sets by passing `'train+test'`. Note that, for some datasets, the values of split are `'train'` or `'validataion'`. For more information, please refer to the official documentation.

Now, let us load the dataset. We set both `with_info` and split to `['train[:25\%]', 'test[:10\%]']`. We load a quarter of the training set and 10% of the test set.

```
dataset, info = tfds.load(name='sentiment140', with_info=True,
                          split=['train[:50%]', 'test[:50%]'])
```

The function returns `dataset` as a list with two elements. The first element is the training set and the second element is the test set.

```
train_data, test_data = dataset[0], dataset[1]
```

We further split the training set into training and validation sets.

```
vald_size = int(len(train_data) * 0.2)
vald_data = train_data.skip(len(train_data) - vald_size)
train_data = train_data.take(len(train_data) - vald_size)
```

We create a function to get text and polarity from the dataset. The other columns (fields) will be ignored.

```
def get_text_polarity(data_dict):
    data_dict['polarity'] = tf.one_hot(data_dict['polarity'], depth=1)
    return data_dict['text'], data_dict['polarity']
```

Then, we use `map` method to apply the created function to the training, validation and test set.

```
train_data = train_data.map(get_text_polarity)
vald_data = vald_data.map(get_text_polarity)
test_data = test_data.map(get_text_polarity)
```

We set the training set to randomly shuffles the data. Then we set the batch size to 256.

```
buffer_size = 1000
batch_size = 256
train_data = train_data.shuffle(buffer_size).batch(batch_size)
vald_data = vald_data.batch(batch_size)
test_data = test_data.batch(batch_size)
```

We use `prefetch` method to allows the later samples to be prepared while the current samples are being processed. This will improves latency and throughput.

```
train_data = train_data.prefetch(buffer_size=buffer_size)
vald_data = vald_data.prefetch(buffer_size=buffer_size)
test_data = test_data.prefetch(buffer_size=buffer_size)
```

Let us take a batch from the training set and print out the first sample.

```
for b in train_data.take(1):
    data_batch, label_batch = b
print(data_batch[0].numpy())
```

We assume the size of the vocabulary is 1000 words. Vocabulary is the set of unique words used in the text corpus (dataset).

```
vocab_size = 1000
```

Now, we will be using `TextVectorization` preprocessing layer to turn the text data to integer representation. `TextVectorization` layer has a method called adapt to set the internal state of the layer. For example, `TextVectorization` layer should hold a mapping between string tokens (words) and integer indices. We pass the text data (without the labels) to adapt method to encode the text data.

```
text_vectorizer = tf.keras.layers.TextVectorization(max_tokens=vocab_size)
text_vectorizer.adapt(train_data.map(lambda text, label: text))
```

We can get the vocabulary from `TextVectorization` layer.

```
vocab = np.array(text_vectorizer.get_vocabulary())
```

14.7 Implementing Recurrent Neural Networks

Let us build the sequential model using RNN. A RNN is implemented using `SimpleRNN` module. The module has the following parameters:

- `units` is the dimensionality of the output space.
- `activation` is the activation function to use. The default value is '`tanh`'.
- `use_bias` is to indicate whether the layer uses a bias vector or not. By default, the value is `True`.
- `kernel_regularizer` is the regularizer function applied to the weights. This parameter is used to apply norm penalty regularization. By default, the value is `None`.
- `dropout` is the dropout rate for applying dropout regularization to the inputs. The default value is 0.0.
- `recurrent_dropout` is the dropout rate for applying dropout regularization to the recurrent state. The default value is 0.0.
- `return_sequences` is used to return the output of each time step or the output of the last time step only. By default, the value is `False`.
- `return_state` is used to return the last cell state in addition to the output. By default, the value is `False`.

Another type of layer that we need to use is `Embedding` layer. The purpose of this layer is to convert each word into a fixed length vector of defined size.

- `input_dim` is to define the size of the vocabulary. The value should be integer.
- `output_dim` is the output dimension of the layer i.e. the size of the output vector. The value should be integer.
- `mask_zero` is to pad the input with value of 0 or not. This is useful when using recurrent layers which takes different lengths of input. The value is either `True` or `False`. By default, the value is `False`.

First, we define the input layer using `Input`. Then, we pass the output of the input layer to `TextVectorization` layer to turn the text data to integer representation. The `Embedding` layer receives the integer representation to convert it to vector representation. The vector is the input to the RNN. We set the output units of the RNN to 64. The output of the RNN is passed to a fully connected layer with 32 units. Finally, the output of this layer is passed to another fully connected layer (output layer). The output layer has 1 unit representing 0 (negative) and 1 (positive).

```
from tensorflow.keras.layers import SimpleRNN, Dense, Input, Embedding
from tensorflow.keras.models import Model

inputs = Input(dtype=tf.string, shape=(1,))
x = text_vectorizer(inputs)
x = Embedding(input_dim=len(text_vectorizer.get_vocabulary()),
              output_dim=64, mask_zero=True)(x)
x = SimpleRNN(units=64)(x)
x = Dense(units=32, activation='relu')(x)
outputs = Dense(units=1)(x)
model = Model(inputs=inputs, outputs=outputs)
```

We can add another layer of RNN. Since the RNN receives a sequence of inputs, the first RNN needs to return the full sequence of outputs. To do this, we set `return_sequences` of the first RNN to `True`. Setting this parameter will make the RNN to return the output of each time step which will be the input to the second layer of the RNN.

```
inputs = Input(dtype=tf.string, shape=(1,))
x = text_vectorizer(inputs)
x = Embedding(input_dim=len(text_vectorizer.get_vocabulary()),
              output_dim=64, mask_zero=True)(x)
x = SimpleRNN(units=64, return_sequences=True)(x)
x = SimpleRNN(units=64)(x)
x = Dense(units=32, activation='relu')(x)
outputs = Dense(units=1, activation='sigmoid')(x)
model = Model(inputs=inputs, outputs=outputs)
```

Now, let us build the model using LSTM. The model consists of an LSTM, a hidden (fully connected) layer and an output layer. The module to implement LSTM is `LSTM`. `LSTM` has the following parameters.

- `units` is the dimensionality of the output space.
- `activation` is the activation function. By default, the value is '`tanh`'.
- `recurrent_activation` is the activation function to use for the recurrent step. By default, the value is '`sigmoid`'.
- `use_bias` is to indicate whether the layer uses a bias vector or not. By default, the value is `True`.
- `kernel_regularizer` is the regularizer function applied to the weights. This parameter is used to apply norm penalty regularization. By default, the value is `None`.
- `dropout` is the dropout rate for applying dropout regularization to the inputs. The default value is 0.0.
- `recurrent_dropout` is the dropout rate for applying dropout regularization to the recurrent state. The default value is 0.0.
- `return_sequences` is used to return the output of each time step or the output of the last time step only. By default, the value is `False`.
- `return_state` is used to return the last cell state in addition to the output. By default, the value is `False`.

Here, instead of `SimpleRNN`, we use `LSTM` as the RNN. The `LSTM` receives input vectors from the `Embedding` layer, and pass its output to a fully connected layer.

```
from tensorflow.keras.layers import Input, Embedding, LSTM, Dense
from tensorflow.keras.models import Model

inputs = Input(dtype=tf.string, shape=(1,))
x = text_vectorizer(inputs)
x = Embedding(input_dim=len(text_vectorizer.get_vocabulary()), output_dim=64)(x)
x = LSTM(units=64)(x)
x = Dense(units=64, activation='relu')(x)
outputs = Dense(units=1, activation='sigmoid')(x)
model = Model(inputs=inputs, outputs=outputs)
```

GRU can be implemented using `GRU` module. The module has similar parameters as LSTM.

- `units` is the dimensionality of the output space.
- `activation` is the activation function. By default, the value is `'tanh'`.
- `recurrent_activation` is the activation function to use for the recurrent step. By default, the value is `'sigmoid'`.
- `use_bias` is to indicate whether the layer uses a bias vector or not. By default, the value is `True`.
- `kernel_regularizer` is the regularizer function applied to the weights. This parameter is used to apply norm penalty regularization. By default, the value is `None`.
- `dropout` is the dropout rate for applying dropout regularization to the inputs. The default value is 0.0.
- `recurrent_dropout` is the dropout rate for applying dropout regularization to the recurrent state. The default value is 0.0.
- `return_sequences` is used to return the output of each time step or the output of the last time step only. By default, the value is `False`.
- `return_state` is used to return the last cell state in addition to the output. By default, the value is `False`.

Here, we build a model consists of a GRU, a hidden (fully connected) layer and an output layer.

```
from tensorflow.keras.layers import Input, Embedding, GRU, Dense
from tensorflow.keras.models import Model

inputs = Input(dtype=tf.string, shape=(1,))
x = text_vectorizer(inputs)
x = Embedding(input_dim=len(text_vectorizer.get_vocabulary()), output_dim=64)(x)
x = GRU(units=64)(x)
x = Dense(units=64, activation='relu')(x)
outputs = Dense(units=1, activation='sigmoid')(x)
gru_model = Model(inputs=inputs, outputs=outputs)
print(gru_model.summary())
```

Exercises

1. Explain the difference between RNN and fully-connected neural networks.
2. Give some examples where RNN can be used.
3. Explain if it is possible for a RNN to have more than one hidden layer. Justify your answer.

4. Explain the difference between LSTM and GRU.
5. Explain bidirectional RNN.
6. RNN takes the current input $\mathbf{x}^t$ and the previous state vector $\mathbf{h}^{(t-1)}$ and returns a new state $\mathbf{h}^t$ and output vector $\hat{y}^t$.

   $$\hat{y}^t = g(\mathbf{w}^\top \mathbf{h}^t + \mathbf{c})$$

   $$\mathbf{h}^t = f(\mathbf{u}^\top \mathbf{x}^t + \mathbf{b}_x + \mathbf{v}^\top \mathbf{h}^{t-1} + \mathbf{b}_h)$$

   Suppose that f is a step function and g is a linear function. Determine the weights $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$ and biases $\mathbf{b}_x$, $\mathbf{b}_h$ and $\mathbf{c}$ so that the RNN initially output 0, and as soon as it receives an input of 1, it switches to output 1 for all subsequent time steps. For example, an input sequence of 00011010 produces an output 00011111. Assuming the hidden unit has an initial value of 0.

Chapter 15

Attention Mechanism

15.1 Introduction

Attention mechanism has significantly improved the performance of various deep learning models in recent years. It is a powerful method that is meant to mimic cognitive attention which allows the model to focus on one or few inputs while ignoring others. Attention mechanism emerged as an improvement over the encoder-decoder model for neural machine translation. Since then, many advances have been made in natural language processing, including the bidirectional encoder representations from transformers (BERT) (Devlin et al. 2018) and generative pre-trained transformer 3 (GPT3) (Brown et al. 2020). In this chapter, we first introduce the encoder-decoder architecture. Then, we introduce the attention mechanism and the self-attention mechanism. We describe the attention methods in the context of machine translation.

15.2 Encoder-Decoder

The most common network architecture for many-to-many sequence learning is called encoder-decoder (Cho et al. 2014; Sutskever, Vinyals, and Le 2014). The architecture is the basis of state-of-the-art machine translation models. In general, the encoder accepts the input sequence and encodes it into a feature representation of the input sequence which is known as a context vector. Then, the decoder takes the context vector and decodes it to produce the output sequence word by word. Figure 15.1 illustrates the block diagram of the encoder-decoder model.

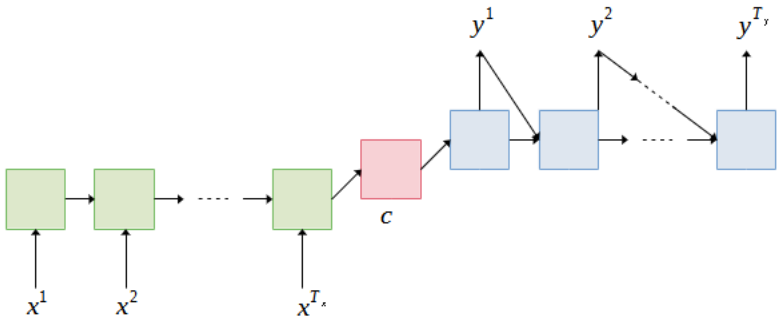


Figure 15.1 Architecture of encoder-decoder

Both encoder and decoder use RNN to model the sequence of the data. The RNN may refer to the

RNN or a variant of RNN such as LSTM and GRU. Let $\mathbf{x}$ of length T_x denotes the source sequence and $\mathbf{y}$ of length T_y denotes the target sequence. Note that the source and target variables are in bold to indicate they are vectors.

$$\mathbf{x} = \{\mathbf{x}^1, \mathbf{x}^2, ..., \mathbf{x}^{T_x}\} \tag{15.1}$$

$$\mathbf{y} = \{\mathbf{y}^1, \mathbf{y}^2, ..., \mathbf{y}^{T_y}\} \tag{15.2}$$

In the context of machine translation, the source is the set of sentences to be translated and the target is the set of sentences of the language that we want to translate to. The RNN of the encoder reads the input sequence whereby at each time step, the inputs are the current word $\mathbf{x}^t$ and the hidden state from the previous time step $\mathbf{h}^{t-1}$, and the output of the RNN is the new hidden state $\mathbf{h}^t$. The initial hidden state of the decoder $\mathbf{h}^0$ is typically zeros. The encoder $f_{\texttt{enc}}$ can be represented as a function of $\mathbf{x}^t$ and $\mathbf{h}^{t-1}$.

$$\mathbf{h}^t = f_{\texttt{enc}}(\mathbf{x}^t, \mathbf{h}^{t-1}) \tag{15.3}$$

Once the final word of the input sequence is passed to the RNN, the final hidden state is used as the context vector $\mathbf{c} = \mathbf{h}^{T_x}$. The context vector is passed to the decoder for decoding and generating the translation. The decoder is trained to generate the output or the target words $\mathbf{y}^t$ given hidden state $\mathbf{s}^t$ at each time step. Typically, fully connected layer(s) is used to predict the target words. The hidden state $\mathbf{s}^t$ is conditioned on the previous target word $\mathbf{y}^{t-1}$and the previous hidden state $\mathbf{s}^{t-1}$. The decoder $f_{\texttt{dec}}$ can be represented as follows:

$$\mathbf{y}^t = f_{\texttt{dec}}(\mathbf{s}^t) \tag{15.4}$$

where hidden state $\mathbf{s}^t$ is

$$\mathbf{s}^t = g(\mathbf{y}^{t-1}, \mathbf{s}^{t-1}) \tag{15.5}$$

$$\mathbf{s}^0 = \mathbf{c} \tag{15.6}$$

We use two tokens to indicate the start and end of the source and target sentences: `<start>` and `<end>`. The tokens are appended to the start and end of the sequence $\mathbf{x} = \{\mathbf{x}^1, \mathbf{x}^2..., \mathbf{x}^{T_x}\}$ where $\mathbf{x}^1 =< \texttt{start} >$ and $\mathbf{x}^{T_x} =< \texttt{end} >$. Thus, the first input to the encoder and decoder is `<start>`. The subsequent inputs are the source and target words, and the last input `<end>`. When training the encoder-decoder model, we always know the number of words in the sequences. During inference, the model will keep generating the words until `<end>` token is generated.

15.3 Attention Mechanism

In the previous section, we discussed encoder-decoder for sequence learning such as machine translation. In encoder-decoder model, the final hidden state of the encoder, also known as context vector is passed to the decoder for generating the target words. The context vector represents the compressed version of the input sequence. However, due to the fixed length of the context vector, the information needs to be compressed which may lead to information loss especially those early in the sequence. Bi-directional RNN may resolve the problem by processing the sequence in reverse order. However, the problem still persists when processing long input sequences. An attention mechanism was introduced to address the challenge of learning long input sequences (Bahdanau, Cho, and Bengio 2014). Attention allows the model to focus on different "parts" of

input sequence when generating the outputs at each time step. Unlike the standard encoder-decoder model, attention derives the context vector from the hidden states of both the encoder and decoder, and the alignment between the source and target.

The encoder is a bi-directional RNN that reads the sequence of inputs $\mathbf{x} = \{\mathbf{x}^1, \mathbf{x}^2, ..., \mathbf{x}^{T_x}\}$ and generates forward hidden states $\overrightarrow{\mathbf{h}}$ and backward hidden states $\overleftarrow{\mathbf{h}}$ at each time step. The concatenation of the hidden states represents the encoder state.

$$\mathbf{h} = [\overrightarrow{\mathbf{h}}; \overleftarrow{\mathbf{h}}] \tag{15.7}$$

where $\overrightarrow{\mathbf{h}} = \{\mathbf{h}^1, \mathbf{h}^2, ..., \mathbf{h}^{T_x}\}$ and $\overleftarrow{\mathbf{h}} = \{\mathbf{h}^{T_x}, \mathbf{h}^{T_{x-1}}, ..., \mathbf{h}^1\}$

At each time step, the decoder generates a hidden state as follows:

$$\mathbf{s}^t = f(\mathbf{y}^{t-1}, \mathbf{s}^{t-1}, \mathbf{c}^t) \tag{15.8}$$

where the initial hidden state $\mathbf{s}^0 = \overleftarrow{\mathbf{h}}$ and $\mathbf{c}^t$ denotes the context vector at time t. The context vector is computed as a weighted sum of the encoder hidden states, with weights given by the alignment scores $\alpha^{t,j}$:

$$\mathbf{c}^t = \sum_j^{T_x} \alpha^{t,j} \mathbf{h}^j \tag{15.9}$$

The alignment scores are the measures of how well the input at position j match the output at position t. The scores are calculated based on the decoder hidden state $\mathbf{s}^t - 1$ which is right before the output is generated and the encoder hidden state $\mathbf{h}^j$. The alignment scores $\alpha^{t,j}$ are computed by

$$\alpha^{t,j} = \frac{\texttt{exp}(\texttt{score}^{t,j})}{\sum_k^{T_x} \texttt{exp}(\texttt{score}^{t,k})} \tag{15.10}$$

where $\texttt{score}^{t,j} = a(\mathbf{s}^{t-1}, \mathbf{h}^j)$. Figure 15.2 illustrates how the alignment scores determine the contribution of each encoder hidden state to the context vector.

Bahdanau et al. parameterize the alignment score function $a(\mathbf{s}^{t-1}, \mathbf{h}^j)$ with fully connected layers that are jointly trained with the encoder and decoder (Bahdanau, Cho, and Bengio 2014). Specifically, the score function is defined as

$$a(\mathbf{s}^{t-1}, \mathbf{h}^j) = \mathbf{v}^\top \texttt{tanh}(\mathbf{w} \cdot \mathbf{s}^{t-1} + \mathbf{u} \cdot \mathbf{h}^j) \tag{15.11}$$

where $\mathbf{v}$, $\mathbf{w}$ and $\mathbf{u}$ are weight matrices to be learned by the model.

Given the significant improvement by integrating the attention mechanism, various forms of attention and alignment score functions have been introduced (Luong, Pham, and Manning 2015; Graves, Wayne, and Danihelka 2014). In (Luong, Pham, and Manning 2015), Luong et al. introduced several changes to generalize the attention mechanism which allows the alignment score function to be switched to different score functions. The key difference is the method of computing the context vector. Bahdanau attention first derives the context vector. Then, the decoder feeds the concatenation of the context vector with the previous target word $\mathbf{y}^{t-1}$ to the decoder's RNN to generate the current hidden state $\mathbf{s}^t$. Luong attention first generates $\mathbf{s}^t$ using the decoder's RNN and then uses it to derive the context vector. The decoder concatenates the context vector with $\mathbf{s}^t$

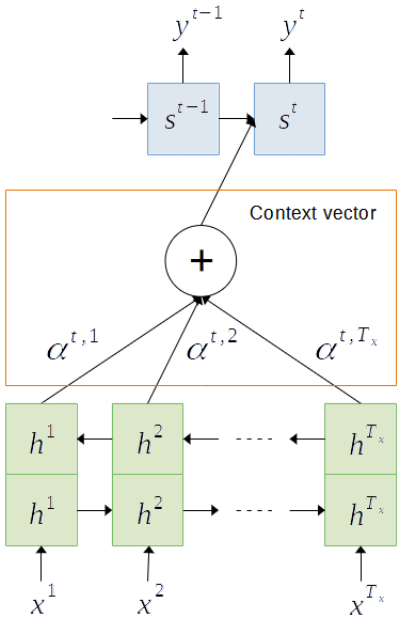


Figure 15.2 The context vector is derived by summing all the weighted hidden states of the encoder. Image is reproduced from (Bahdanau, Cho, and Bengio 2014).

and feeds it to the fully connected layer to generate the target word. Figure 15.3 illustrates the differences in the attention mechanisms, while Figure 15.4 presents an example of alignment scores computed by an encoder-decoder model with attention for English-French translation.

The attention that we discussed above attends to the entire hidden states of the encoder. This attention is called global attention model. This model has the drawback that it is computationally expensive for long sequences. Luong attention proposed the local attention model whereby the context vector is derived as a weighted average over a set of encoder hidden states $[p^t - D, p^t + D]$ where D is empirically selected. The context vector is therefore calculated as follows. Figure 15.5 illustrates the computation of global and local attention.

$$\mathbf{c}^t = \sum_{j=p^t-D}^{p^t+D} \alpha^{t,j} h^j \tag{15.12}$$

Another difference is that the encoder does not use a bi-directional RNN to simplify the computation. Thus, only forward hidden states are used when deriving the context vector. Finally, three alignment score functions are introduced: general, dot-product and location-based, as an alternative to the additive score function. Table 15.1 lists the popular attention mechanism and their alignment score functions.

15.4 Self-Attention

RNN is an essential component in attention model for learning sequential data. However, due to the sequential nature of recurrent neural network, it precludes parellization within the training data. For a sequence with L lengths, the model needs to perform $O(n)$ sequential operations to generate the output sequence. The problem is becoming more pronounced when processing long input sequences. The motivation for the development of self-attention is to reduce the computational complexity of

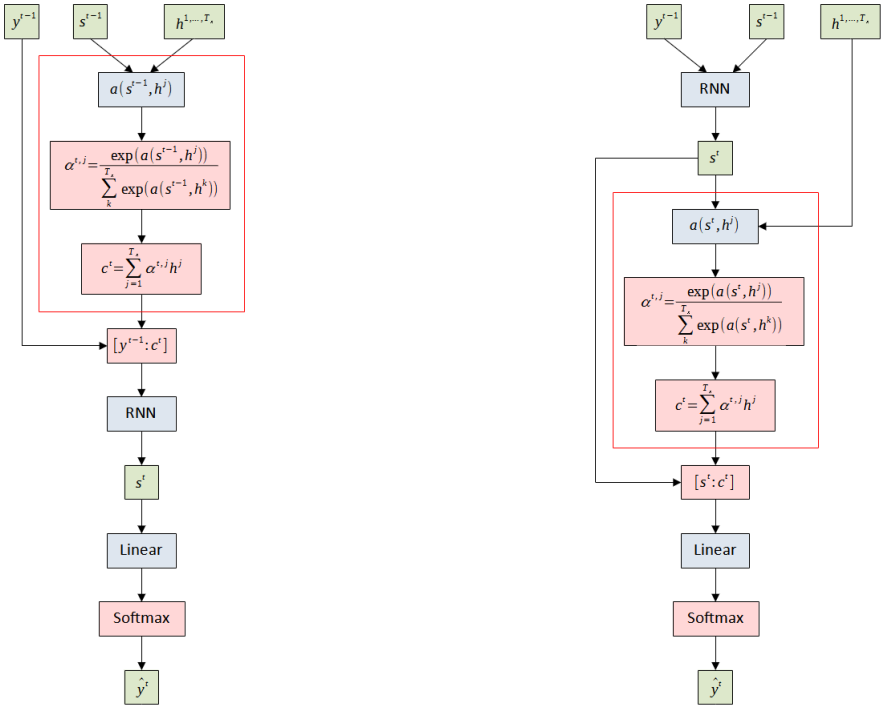


Figure 15.3 (a) Bahdanau (Bahdanau, Cho, and Bengio 2014) and (b) Luong (Luong, Pham, and Manning 2015) attention mechanisms. The red bounded box indicates the calculation of the context vector.

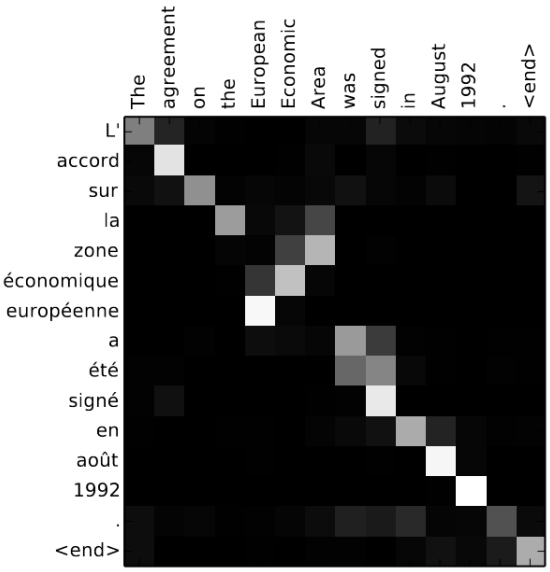


Figure 15.4 The alignment scores between the source words (English) and the generated translation (French). Image source from (Bahdanau, Cho, and Bengio 2014).

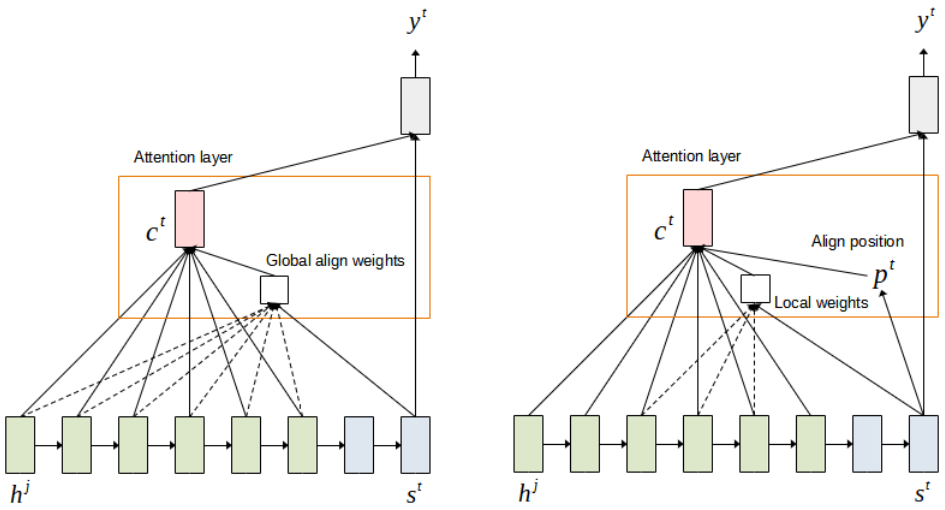


Figure 15.5 Global and local attention. Encoder in green and decoder in blue. Images are reproduced from (Luong, Pham, and Manning 2015).

Table 15.1 Popular alignment score functions

Alignment score function	Name
$\texttt{score}^{t,j} = \mathbf{v}^\top \texttt{tanh}(\mathbf{w} \cdot \mathbf{s}^{t-1} + \mathbf{u} \cdot \mathbf{h}^j)$	Additive or Concatenation
$\texttt{score}^{t,j} = \mathbf{h}^{jT}\mathbf{w}\mathbf{s}^t$ where $\mathbf{w}$ is the trainable weight matrix.	General
$\texttt{score}^{t,j} = \mathbf{s}^{tT}\mathbf{h}^j$	Dot-product
$\alpha^{t,j} = \texttt{softmax}(\mathbf{w}\mathbf{s}^t)$. This score function depends on the target position thus simplifying the softmax alignment.	Location-base
$\texttt{score}^{t,j} = \texttt{cosine}[\mathbf{s}^t, \mathbf{h}^j]$	Content-based attention

the attention model. Like attention which allows the model to focus attention on specific parts of the input sequence, self-attention enables interaction between words in the sequence to compute the context vector, resulting in better capturing of the syntactic and semantic structure of the sentences.

15.4.1 The Transformer

The Transformer architecture proposed in (Vaswani et al. 2017) is the first model that relied entirely on self-attention for the alignment between input and output sequence without the recurrent architecture. Although the model does not use recurrent and convolutional layers, it retains the popular encoder-decoder architecture as shown in Figure 15.6. The encoder (on the left side of the figure), consists of six identical layers whereby each has two sub-layers. The first sub-layer is multi-head attention which we will be explained later. The second sub-layer is position-wise feed forward network. The position-wise feed forward network is implemented using two fully connected layers with ReLU activation function in between.

$$\texttt{FFN}(\mathbf{x}) = \texttt{max}(0, \mathbf{x} \cdot \mathbf{W}_1 + b_1)\mathbf{W}_2 + b_2 \tag{15.13}$$

The sub-layers have a residual connection followed by layer normalization, that is the output of each sub-layer is

$$y = \texttt{LayerNorm(x + Sublayer(x))} \tag{15.14}$$

where `Sublayer(x)` is the output of the sub-layer itself. All sub-layers and the embedding layer produce outputs of dimension $d_m = 512$.

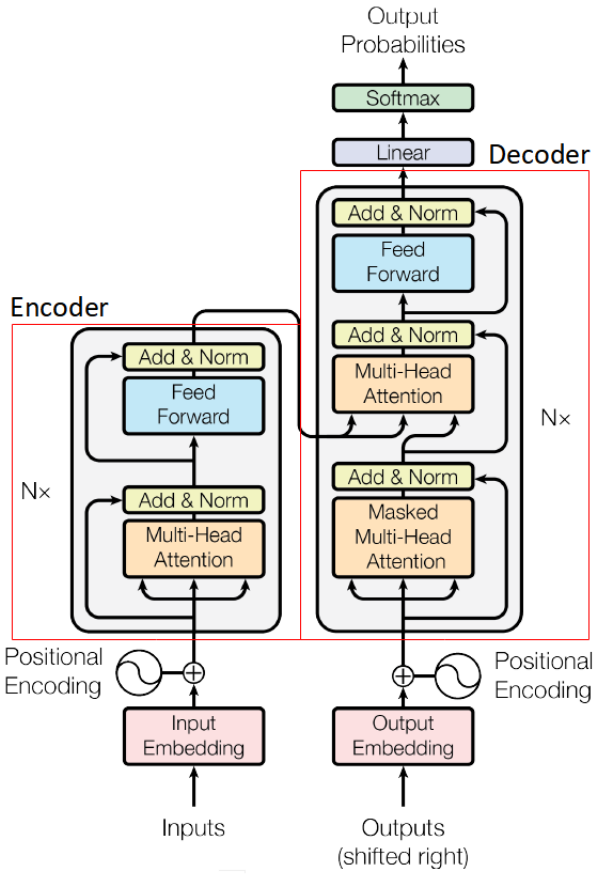


Figure 15.6 The Transformer architecture consists of encoder (left) and decoder (right). Image source from (Vaswani et al. 2017).

The decoder as shown in Figure 15.6 is also consists of six identical layers. Each layer has two sub-layers as in the encoder and a third sub-layer, which also performs multi-head attention. This sub-layer receives one input from the first multi-head attention and two inputs from the encoder stack. Similar to the sub-layers of the encoder, a residual connection followed by layer normalization. The output of the decoder is passed to a fully connected layer with softmax function for prediction.

Since the Transformer does not use any recurrent layer, it needs a method to "know" the order of the words in the sequence. This is achieved by adding the positional encoding to create positional vectors which to be added to the input vectors. The positional encoding uses the sine and cosine

functions of different frequencies to create the positional vectors.

$$\texttt{PE}(p, 2i) = \texttt{sin}(\frac{p}{1000^{\frac{2i}{d_m}}}) \tag{15.15}$$

$$\texttt{PE}(p, 2i+1) = \texttt{cos}(\frac{p}{1000^{\frac{2i}{d_m}}}) \tag{15.16}$$

where p is the position and i is the dimension.

15.4.2 Multi-head Attention

The multi-head attention consists of several components, of which the main component is the scaled dot-product attention as shown in Figure 15.7. The multi-head attention accepts three inputs denoted by $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$, representing query, key and value vectors. The query and key have the same dimension d_k and value has d_v dimension. Note that the vectors may have the same dimension $d_k = d_v$. Each vector is passed to a fully connected layer, for linear projection.

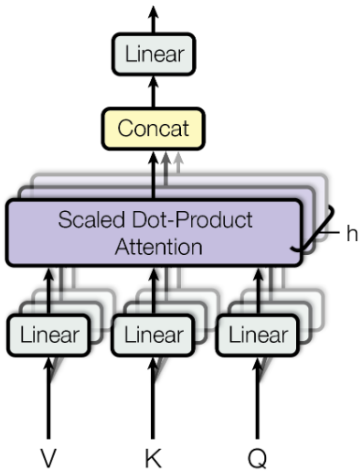


Figure 15.7 Multi-head attention. Image source from (Vaswani et al. 2017)

$$\mathbf{Q} = \mathbf{w}_q^T \mathbf{x} \tag{15.17}$$

$$\mathbf{K} = \mathbf{w}_k^T \mathbf{x} \tag{15.18}$$

$$\mathbf{V} = \mathbf{w}_v^T \mathbf{x} \tag{15.19}$$

where $\mathbf{w}_q$, $\mathbf{w}_k$ and $\mathbf{w}_v$ are weight matrices to be learned. $\mathbf{Q} = \mathbb{R}^{m \times d_k}$, $\mathbf{K} = \mathbb{R}^{m \times d_k}$ and $\mathbf{V} = \mathbb{R}^{m \times d_v}$ where m is the length of sequence.

The scaled dot-product attention computes the attention scores of the words. The sequence alignment is treated as a mapping between a query and a set of key-value pairs, and an output. The concept of query, key and value in self-attention is similar to the regular python dictionary. The following code shows a python dictionary `Courses` with three keys (`CPT111`, `CPT112` and `CPT113`) and three values (`Programming 1`, `Mathematic` and `Programming 2`). A query `Courses['CPT112']` is passed to the dictionary to fetch a piece of information which is stored in `result`. The query is a request

for information, key is what kind of information the dictionary has and value is the information the dictionary has. When the query is passed to the dictionary, the search engine will map the query against a set of keys and return the information that the matching key is associated with. In the context of machine translation, the query is the target sequence, value is the source sequence and key is some information associated with the source sequence.

```
Courses = {'CPT111': 'Programming 1' , 'CPT112': 'Mathematic', 'CPT113': 'Programming 2'}
result = Courses['CPT112']
```

The query, key and value vectors are derived from the same item in the input sequence. Using these vectors, the output is calculated by adding the weighted values, where the weights are determined by how well the query matches the associated keys. The attention weights (or scores) are calculated as follows. We compute the dot products between $\mathbf{Q}$ with $\mathbf{K}$ and perform scaling by dividing the resultant matrix with $\sqrt{d_k}$, followed by the softmax function to obtain the weights. Then, the resulting weights are applied to the value.

$$\texttt{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \texttt{softmax}(\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{d_k}})\mathbf{V} \tag{15.20}$$

The attention function allows the model to learn dependencies between different parts of the sequence. In the context of machine translation, understand the syntactic and semantics between words in the sentence. Figure 15.8 illustrates the computation of the scaled dot-product attention. As can be seen in Figure 15.7, the multi-head attention has several scaled dot-product attention runs in parallel. Each attention produces d_v dimensional output, and the outputs are concatenated and fed to a linear fully connected layer.

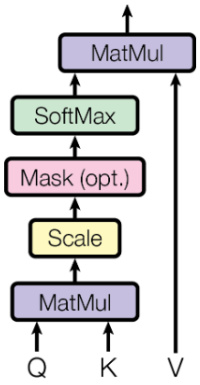


Figure 15.8 The scaled dot-product attention. Image source from (Vaswani et al. 2017)

15.5 Implementing Encoder-Decoder

In this section, we will be implementing encoder-decoder model for machine translation. We will be using the TensorFlow Text, a library that provides a collection of text related classes and functions ready to use with TensorFlow. The library can perform the preprocessing required by text-based models, and other features for sequence modeling. Use pip to install TensorFlow Text. Specify the library version that is similar to TensorFlow library.

```
pip install -U tensorflow-text==<version>
```

We import the libraries including TensorFlow Text.

```
import tensorflow as tf
import tensorflow_datasets as tfds
import tensorflow_text as tf_text
import numpy as np
```

We will be using a dataset from Opus. Opus is a collection of translated texts from the web. The dataset contains over 1.9 million sentences in Malay and English. First, download the dataset from Kaggle using the following link: `https://www.kaggle.com/datasets/mohalim/english-malay`

Once the dataset has been downloaded, unzip the zip file and move "`ms_en.txt`" to your working folder. Then, we define the path to the dataset file and create a function to read the file. Replace "`path_to_file`" with the path to the file.

```
import pathlib
dataset_path = 'path_to_file/ms_en.txt'
dataset_path = pathlib.Path(dataset_path)
```

The function reads the file, splits into lines and splits each line by '`\t`' (tab). Finally, we create the arrays of the Malay and English sentences.

```
def read_file(path):
    data = path.read_text(encoding='utf-8')
    lines = data.splitlines()
    pairs = [line.split('\t') for line in lines]
    source = np.array([english for malay, english in pairs])
    target = np.array([malay for malay, english in pairs])
    return source, target
```

Use the function by passing `dataset_path`. The function returns two arrays: `source` and `target`. `source` contains English sentences and `target` contains Malay sentences.

```
source, target = read_file(dataset_path)
```

From the arrays of strings, we create a `tf.data.Dataset`. We will be using only 20% of the data.

```
total = len(source)
subset_size = int(0.2 * total)
dataset = tf.data.Dataset.from_tensor_slices((source[:subset_size],
                                              target[:subset_size]))
```

Now we split the dataset into training and validation sets. We put aside 20% of the data for validation set.

```
vald_size = int(len(dataset) * 0.2)
vald_data = dataset.skip(len(dataset) - vald_size)
train_data = dataset.take(len(dataset) - vald_size)
```

Set the buffer size and batch size of the dataset.

```
buffer_size = 1000
batch_size = 256
train_data = train_data.shuffle(buffer_size).batch(batch_size)
vald_data = vald_data.batch(batch_size)
```

15.5.1 Text Preprocessing

Now, we need to create a function to preprocess the text data. The first is to perform Unicode normalization to split accented characters and replace compatibility characters with their ASCII equivalents. The tensorflow_text library has a function `normalize_utf8` which normalizes each UTF-8 string in the input tensor using the specified rule. The four Unicode normalization forms: `NFD`, `NFC`, `NFKD` and `NFKC`. By default, the normalization form is `NFKD`. Then, the convert all uppercase characters into their respective lowercase replacements followed by removing all characters except characters (a-z) and selected punctuations. We also add spaces around punctuations and removing whitespaces. Finally, "start of sequence" and "end of sequence" tokens are added to the input sequences.

```
def preprocess_text(text):
    text = tf_text.normalize_utf8(text, 'NFKD')
    text = tf.strings.lower(text)
    text = tf.strings.regex_replace(text, '[^ a-z.?!,]', '')
    text = tf.strings.regex_replace(text, '[.?!,]', r' \0 ')
    text = tf.strings.strip(text)
    text = tf.strings.join(['[START]', text, '[END]'], separator=' ')
    return text
```

This `preprocess_text` function is used by the `TextVectorization` layer which will handle the vocabulary extraction and preprocessing of input text to sequences of tokens. We create two `TextVectorization` for source and target sentences. We assume the vocabulary size is 1000. Note that, the vocabulary size defines the number of units of the output layer.

```
from tensorflow.keras.layers import TextVectorization
vocab_size = 1000
source_text_vectorizer = TextVectorization(standardize=preprocess_text,
                                           max_tokens=vocab_size)
source_text_vectorizer.adapt(train_data.map(lambda source, target : source))
target_text_vectorizer = TextVectorization(standardize=preprocess_text,
                                           max_tokens=vocab_size)
target_text_vectorizer.adapt(train_data.map(lambda source, target : target))
```

We need to create a function for the training and validation sets. This `prepare_text` function converts the text data into a sequence of tokens using the `TextVectorization` layers. It also converts from a `(source, target)` pair to an `((source, target_in), target_out)` pair for training. The `target_in` is used as input to the decoder while `target_out` is used as label. The difference between `target_in` and `target_out` is that `target_out` is shifted by one step relative to each other, so that the label is the next token at each position.

```
def prepare_text(source, target):
    source = source_text_vectorizer(source)
    target = target_text_vectorizer(target)
    target_in = target[:,:-1]
    target_out = target[:,1:]
    return (source, target_in), target_out
```

Then, we use `map` method to apply the created function to the training and validation sets.

```
train_data = train_data.map(prepare_text, tf.data.AUTOTUNE)
vald_data = vald_data.map(prepare_text, tf.data.AUTOTUNE)
```

Let us display the first sequence of the training set.

```
for (src, tgt_in), tgt_out in train_data.take(1):
    print(src[0,:10].numpy())
    print()
    print(tgt_in[0,:10].numpy())
    print(tgt_out[0,:10].numpy())
```

15.5.2 The Encoder

Now, let us create the encoder-decoder model. Figure 15.9 shows the overview of the encoder-decoder model. The encoder is on the left and the decoder is on the right. The final hidden state of the encoder is the context vector is passed to the decoder to predict the target words. The encoder consists of an embedding layer and an LSTM. The embedding layer takes the input sequence and passes the output to the LSTM. Both `return_sequences` and `return_state` of the LSTM are set to `True`. The `call` function returns the final hidden state and the carry state of the LSTM.

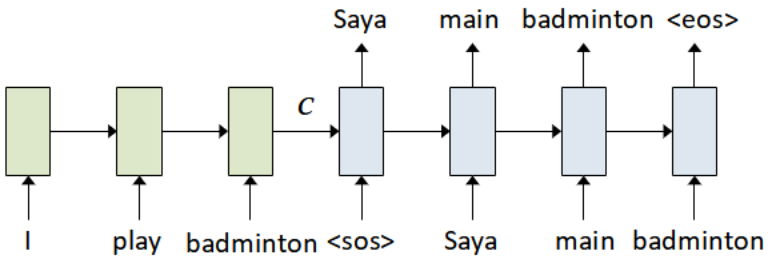


Figure 15.9 An overview of the encoder-decoder model

A function is needed for inference (predicting new text data) whereby the function accepts text as input, and passes it to the `TextVectorization` layer to convert the text to integer representation. Then, the integer representation is passed to the embedding layer and the LSTM.

```
from tensorflow.keras.layers import Layer, LSTM, Embedding, Bidirectional

class Encoder(Layer):
    def __init__(self, text_vectorizer, units=64):
        super(Encoder, self).__init__()
        self.text_vectorizer = text_vectorizer
        self.embedding = Embedding(input_dim=text_vectorizer.vocabulary_size(),
                                   output_dim=units, mask_zero=True)
        rnn = LSTM(units=units, return_sequences=True,
                   return_state=True, dropout=0.3)
        self.birnn = Bidirectional(rnn)

    def call(self, x):
        x = self.embedding(x)
        seq_out, fw_hidden, fw_carry, bw_hidden, bw_carry = self.birnn(x)
        return [bw_hidden, bw_carry]

    def process_input_text(self, texts):
        texts = tf.convert_to_tensor(texts)
        if len(texts.shape) == 0:
            texts = tf.convert_to_tensor(texts)[tf.newaxis]
        source = self.text_vectorizer(texts)
        state = self(source)
        return state
```

15.5.3 The Decoder

In addition to the embedding and LSTM, the decoder includes a fully connected layer and string lookup tables for prediction. Two string lookup tables are defined using `StringLookup` module. The first lookup table (`word_to_id`) translates a string to its corresponding integer representation (index) while the second lookup table (`id_to_word`) translates an index to its corresponding string representation. The parameter `invert` is set to `True` when creating `id_to_word` to reverse the mapping. The vocabulary `parameter` of the lookup tables is set using the `text_vectorizer` to ensure a consistent mapping of words based on the training data. At each time step, the LSTM takes the input embeddings and produces both output predictions and the hidden states. The output predictions, processed through the fully connected layer, represent the likelihood of each word in the vocabulary, allowing for word generation. The decoder also includes special tokens generated by `word_to_id`, such as the start and end tokens, which help delineate the boundaries of the generated sequences.

Three additional functions are needed for inference. The first function is called `get_start_token`, creates the start token based on the size of the input. The second function is called `get_next_token`, is used to predict the next token (word). The previously predicted token and the previous hidden state is passed to the LSTM to predict the next token. Also the function compared the generated word with end token. If the comparison is true, set `done` flag to `True`. The last function is called `tokens_to_text` converts the predicted tokens to words. This function uses the second string lookup table (`id_to_word`) to translate an index to its corresponding string representation.

```
from tensorflow.keras.layers import Layer, LSTM, Dense, Embedding, StringLookup

class Decoder(Layer):
    def __init__(self, text_vectorizer, units=64):
        super(Decoder, self).__init__()
        self.vocab_size = text_vectorizer.vocabulary_size()
        self.embedding = Embedding(input_dim=self.vocab_size,
                                   output_dim=units, mask_zero=True)
        self.rnn = LSTM(units=units, return_sequences=True,
                        return_state=True, dropout=0.3)
        self.output_layer = Dense(units=self.vocab_size)

        self.word_to_id = StringLookup(
            vocabulary=text_vectorizer.get_vocabulary(),
            mask_token='', oov_token='[UNK]')
        self.id_to_word = StringLookup(
            vocabulary=text_vectorizer.get_vocabulary(),
            mask_token='', oov_token='[UNK]', invert=True)

        self.start_token = self.word_to_id(['[START]'])
        self.end_token = self.word_to_id(['[END]'])

    def call(self, x, state=None, return_state=False):
        x = self.embedding(x)
        x, hidden_state, carry_state = self.rnn(x)
        x = self.output_layer(x)
        if return_state:
            return x, [hidden_state, carry_state]
        else:
            return x

    def get_start_token(self, source):
        batch_size = tf.shape(source)[0]
        start_tokens = tf.fill([batch_size, 1], self.start_token)
        done = tf.zeros([batch_size, 1], dtype=tf.bool)
        return start_tokens, done

    def get_next_token(self, next_token, done, state):
        logits, state = self(next_token, state=state, return_state=True)
        next_token = tf.argmax(logits, axis=-1)
        done = done | (next_token == self.end_token)
        next_token = tf.where(done, tf.constant(0, dtype=tf.int64), next_token)
        return next_token, done, state

    def tokens_to_text(self, tokens):
        words = self.id_to_word(tokens)
        result = tf.strings.reduce_join(words, axis=-1, separator=' ')
        result = tf.strings.regex_replace(result, '^ *\[START\] *', '')
        result = tf.strings.regex_replace(result, ' *\[END\] *$', '')
        return result
```

15.5.4 The Model

Now that we have defined the encoder and the decoder, we combine them to build the model for training. The `call` function receives the `source` and `target_in` as inputs. The source sequence is passed to the encoder. Then, the hidden state, the carry state and `target_in` are passed to the decoder for predicting the target sequence. We define a function for inference called translate. The function accepts a list of strings (text). The list of strings is passed to the encoder's `process_input_text` function and the encoder produces the hidden state and carry state. Then, the decoder's `get_start_token` function is used to create the start token. Using for loop, the decoder's `get_next_token` is used to predict the next token. At each iteration, `done` flag is checked to stop the prediction. Finally, the decoder's `token_to_text` function is used to convert the predicted tokens to words.

```
from tensorflow.keras.models import Model

class Machine_Translation(Model):
    def __init__(self, units, source_text_vectorizer, target_text_vectorizer):
        super(Machine_Translation, self).__init__()
        self.encoder = Encoder(source_text_vectorizer, units=units)
        self.decoder = Decoder(target_text_vectorizer, units=units)

    def call(self, x):
        source, target = x
        state = self.encoder(source)
        logits = self.decoder(target, state=state)
        return logits

    def translate(self, texts, *, max_length=50, temperature=0.0):
        state = self.encoder.process_input_text(texts)

        tokens = []
        next_token, done = self.decoder.get_start_token(state[0])

        for _ in range(max_length):
            next_token, done, state = self.decoder.get_next_token(next_token,
                                                                  done, state,
                                                                  temperature)
            tokens.append(next_token)
            if tf.executing_eagerly() and tf.reduce_all(done):
                break

        tokens = tf.concat(tokens, axis=-1)
        result = self.decoder.tokens_to_text(tokens)
        return result
```

15.5.5 Training the Model

We need to implement functions to calculate the loss and accuracy. The loss function utilizes the `SparseCategoricalCrossentropy`. A mask is created based on the ground truth and the mask is applied to the loss value to consider only the positions with actual tokens. Similarly, the accuracy is implemented with masking to consider the positions with actual tokens only.

```
from tensorflow.keras.losses import SparseCategoricalCrossentropy

loss_fn = SparseCategoricalCrossentropy(from_logits=True, reduction='none')

def masked_loss(y_true, y_pred):
    loss = loss_fn(y_true, y_pred)
    mask = tf.cast(y_true != 0, loss.dtype)
    loss *= mask
    return tf.reduce_sum(loss)/tf.reduce_sum(mask)

def masked_acc(y_true, y_pred):
    y_pred = tf.argmax(y_pred, axis=-1)
    y_pred = tf.cast(y_pred, y_true.dtype)
    match = tf.cast(y_true == y_pred, tf.float32)
    mask = tf.cast(y_true != 0, tf.float32)
    return tf.reduce_sum(match)/tf.reduce_sum(mask)
```

We create the object of the model.

```
model = Machine_Translation(units=128,
                            source_text_vectorizer=source_text_vectorizer,
                            target_text_vectorizer=target_text_vectorizer)
```

Then, we compile the model using the created masked loss and accuracy functions.

```
model.compile(optimizer=optimizer, loss=masked_loss,
              metrics=[masked_acc, masked_loss])
```

Define the callbacks and train the model by calling `fit`.

```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
checkpoints_dir = 'checkpoints_ms_en/'
callbacks = [EarlyStopping(monitor='val_masked_acc', patience=10),
             ModelCheckpoint(filepath=checkpoints_dir,
                             monitor='val_masked_acc',
                             save_weights_only=True)]
history = model.fit(train_data, epochs=100,
                    validation_data=vald_data,
                    callbacks=callbacks)
```

To predict (translate) new text data, pass the string(s) as a list.

```
new_data = ['And thank you for your help, my friend.',
            'It is special.',
            'Let me spell it out for you.',
            'Just go with it.',
            'Just let me know if you need anything.',
            'The President is waiting for you.']
model.translate(new_data)
```

15.6 Implementing Attention

In this section, we will be implementing the encoder-decoder model with attention. First, we will create the attention layer. Then we will create the encoder and decoder, and use the attention layer in the decoder. The attention layer can be created using `MultiHeadAttention` module. The module has the following parameter:

- `num_heads` is the number of attention heads.
- `key_dim` is the size of each attention head for query and key. .
- `value_dim` is the size of each attention head for value.
- `dropout` is the dropout probability of the attention output. The default value is 0.0.

This module is an implementation of multi-head attention as described in (Vaswani et al. 2017). Thus, if query, key and value are the same, then this is self-attention. However, this module can also be used to implement attention mechanism by specifying different inputs for query and value. In this example, we create an attention layer with number of heads equal 1. We also implement the residual connection with layer normalization. The `call` function has two parameters `x` and `enc_hidden` where `x` is the output the decoder's RNN. The attention layer accepts `x` (query) and `enc_hidden` (value). The attention is set to return the attention scores. The residual connection is implemented by adding the attention's output with `x` and the resultant is fed to the layer normalization.

```
from tensorflow.keras.layers import MultiHeadAttention, LayerNormalization, Add

class Attention(Layer):
    def __init__(self, units=64):
        super(Attention, self).__init__()
        self.mha = MultiHeadAttention(num_heads=1, key_dim=units)
        self.add = Add()
        self.layernorm = LayerNormalization()
        self.attn_scores = 0

    def call(self, x, enc_hidden):
        attn_output, attn_scores = self.mha(query=x, value=enc_hidden,
                                            return_attention_scores=True)
        self.attn_scores = attn_scores
        x = self.add([attn_output, x])
        x = self.layernorm(x)
        return x
```

We need to make a few changes to the Encoder class and Decoder class. For the Encoder class, we remove `return_state=True` because the LSTM does not need to return the last state. Thus, in `call()`, we change the line for RNN to `hidden_states = self.rnn(x)` and the function to return `hidden_states` only. Similar changes is done to `process_input_text` function. The affected lines are in italic.

```
from tensorflow.keras.layers import Layer, LSTM, Embedding, Bidirectional

class Encoder(Layer):
    def __init__(self, text_vectorizer, units=64):
        super(Encoder, self).__init__()
        self.text_vectorizer = text_vectorizer
        self.embedding = Embedding(input_dim=text_vectorizer.vocabulary_size(),
                                   output_dim=units, mask_zero=True)
        self.rnn = LSTM(units=units, return_sequences=True, dropout=0.3)
        self.birnn = Bidirectional(rnn)

    def call(self, x):
        x = self.embedding(x)
        hidden_states = self.birnn(x)
        return hidden_states

    def process_input_text(self, texts):
        texts = tf.convert_to_tensor(texts)
        if len(texts.shape) == 0:
            texts = tf.convert_to_tensor(texts)[tf.newaxis]
        source = self.text_vectorizer(texts)
        hidden_states = self(source)
        return hidden_states
```

For the Decoder class, we create the Attention layer in `__init__()` and a variable to store the attention scores. The `call()` function accepts an additional parameter which is the encoder's hidden states (`enc_hidden`). In `call()`, we add the attention layer after the RNN and the attention layer accepts `x` (the output of decoder's RNN) and `enc_hidden` as inputs. We store the attention score so that we can retrieve it later. The `get_next_token` function needs a few modification. It needs to accept the encoder's output (`enc_hidden`) which will be used in `call()` function. The affected lines are in italic.

```
from tensorflow.keras.layers import Layer, LSTM, Dense, Embedding, StringLookup

class Decoder(Layer):
    def __init__(self, text_vectorizer, units=64):
        super(Decoder, self).__init__()
        self.vocab_size = text_vectorizer.vocabulary_size()
        self.embedding = Embedding(input_dim=self.vocab_size,
                                   output_dim=units, mask_zero=True)
        self.rnn = LSTM(units=units, return_sequences=True,
                        return_state=True, dropout=0.3)
        self.output_layer = Dense(units=self.vocab_size)

        self.word_to_id = StringLookup(
            vocabulary=text_vectorizer.get_vocabulary(),
            mask_token='', oov_token='[UNK]')

        self.id_to_word = StringLookup(
            vocabulary=text_vectorizer.get_vocabulary(),
            mask_token='', oov_token='[UNK]', invert=True)

        self.start_token = self.word_to_id(['[START]'])
```

```
        self.end_token = self.word_to_id(['[END]'])

        self.attention = Attention(units=units)

    def call(self, x, enc_hidden, state=None, return_state=False):
        x = self.embedding(x)
        x, dec_hidden_state, dec_carry_state = self.rnn(x, initial_state=state)
        x = self.attention(x, enc_hidden)
        self.attn_scores = self.attention.attn_scores
        x = self.output_layer(x)
        if return_state:
            return x, [dec_hidden_state, dec_carry_state]
        else:
            return x

    def get_start_token(self, source):
        batch_size = tf.shape(source)[0]
        start_tokens = tf.fill([batch_size, 1], self.start_token)
        done = tf.zeros([batch_size, 1], dtype=tf.bool)
        embedded = self.embedding(start_tokens)
        return start_tokens, done, self.rnn.get_initial_state(embedded)

    def get_next_token(self, next_token, enc_hidden, done, state):
        logits, state = self(next_token, enc_hidden,state=state, return_state=True)
        next_token = tf.argmax(logits, axis=-1)
        done = done | (next_token == self.end_token)
        next_token = tf.where(done, tf.constant(0, dtype=tf.int64), next_token)
        return next_token, done, state

    def tokens_to_text(self, tokens):
        words = self.id_to_word(tokens)
        result = tf.strings.reduce_join(words, axis=-1, separator=' ')
        result = tf.strings.regex_replace(result, '^ *\[START\] *', '')
        result = tf.strings.regex_replace(result, ' *\[END\] *$', '')
        return result
```

Finally, we make a few modifications to the `translate` function in `Machine_Translation` class. The line where we call decoder's `get_start_token()` should return three values. In the for loop, we get the attention scores from the decoder and append it to `attention_scores` list. We create a new function `plot_attention` to display the attention matrix which shows the correlation between the source sequence and the target sequence. The function accepts a string (text), produces the translation, gets the attention scores, splits the sentences into tokens and displays the attention matrix. The affected lines are in italic.

```
from tensorflow.keras.models import Model
from matplotlib import pyplot as plt
from matplotlib import ticker

class Machine_Translation(Model):
    def __init__(self, units, source_text_vectorizer, target_text_vectorizer):
        super(Machine_Translation, self).__init__()
        self.encoder = Encoder(source_text_vectorizer, units=units)
        self.decoder = Decoder(target_text_vectorizer, units=units)
```

```
def call(self, x):
    source, target = x
    enc_hidden_states = self.encoder(source)
    logits = self.decoder(target, enc_hidden_states)
    return logits

def translate(self, texts, max_length=50, temperature=0.0):
    enc_hidden = self.encoder.process_input_text(texts)

    tokens = []
    attention_scores = []
    next_token, done, state = self.decoder.get_start_token(enc_hidden)

    for _ in range(max_length):
        next_token, done, state =
self.decoder.get_next_token(next_token,enc_hidden,done, state,temperature)
        tokens.append(next_token)
        attention_scores.append(self.decoder.attn_scores)
        if tf.executing_eagerly() and tf.reduce_all(done):
            break

    tokens = tf.concat(tokens, axis=-1)
    self.last_attention_scores = tf.concat(attention_scores, axis=1)
    result = self.decoder.tokens_to_text(tokens)
    return result

def plot_attention(self, text):
    output = self.translate([text])[0]

    attention_scores = self.last_attention_scores[0]
    attention_scores = tf.squeeze(attention_scores, axis=1)

    source = preprocess_text(text)
    source = source.numpy().decode().split()

    output = preprocess_text(output)
    output = output.numpy().decode().split()[1:]

    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(1, 1, 1)
    mt = ax.matshow(attention_scores, cmap='viridis', vmin=0.0)
    fig.colorbar(mt)
    fontdict = {'fontsize':  14}

    ax.set_xticklabels([''] + source, fontdict=fontdict, rotation=90)
    ax.set_yticklabels([''] + output, fontdict=fontdict)

    ax.xaxis.set_major_locator(ticker.MultipleLocator(1))
    ax.yaxis.set_major_locator(ticker.MultipleLocator(1))

    ax.set_xlabel('Input text')
    ax.set_ylabel('Output text')
```

We train the model and test the model with new text data. In this example, we pass a string to `plot_attention` function. Figure 15.10 shows the attention matrix between the input sequence and the output sequence generated by the model.

```
model.plot_attention('Just let me know if you need anything.')
```

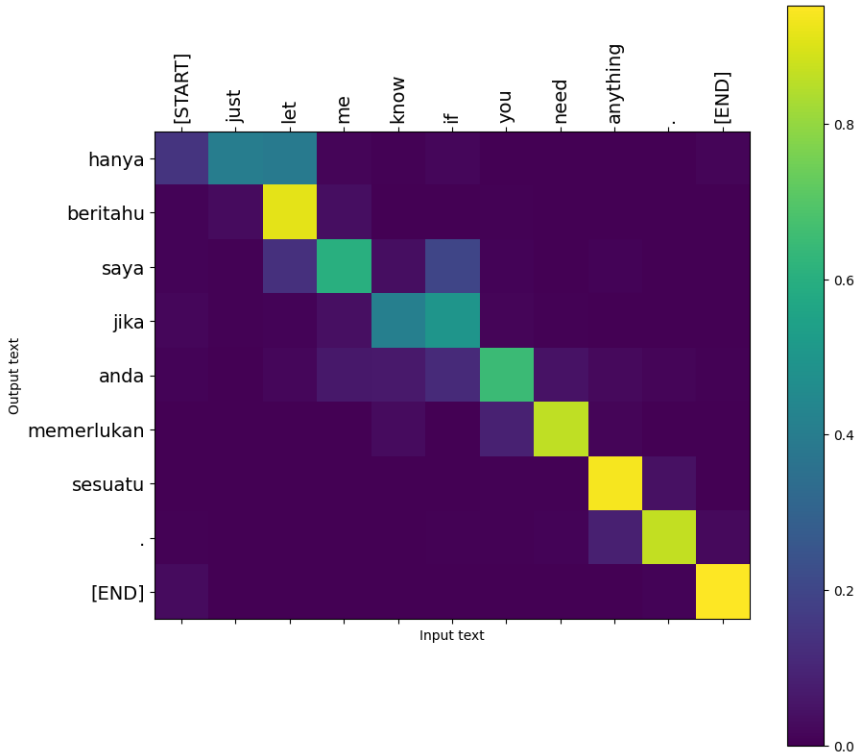


Figure 15.10 The matrix of the attention scores

Exercises

1. Explain how does the attention mechanism works in a neural network.
2. Explain the problem that attention mechanism addresses in neural networks.
3. Explain the difference between global attention and local attention mechanisms.
4. Explain if attention mechanisms can be applied to non-sequence data.
5. Explain self-attention in neural networks.

References

Arthur, David and Sergei Vassilvitskii (Jan. 2007). "k-means++: the advantages of careful seeding". In: *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. SODA '07. Society for Industrial and Applied Mathematics, USA, pp. 1027–1035. ISBN: 978-0-89871-624-5. (Visited on 07/31/2023).

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2014). "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473*.

Bayes, Thomas (Dec. 1763). "LII. An essay towards solving a problem in the doctrine of chances. By the late Rev. Mr. Bayes, F. R. S. communicated by Mr. Price, in a letter to John Canton, A. M. F. R. S". In: *Philosophical Transactions* 53, pp. 370–418. ISSN: 0260-7085. DOI: 10.1098/rstl.1763.0053. URL: https://doi.org/10.1098/rstl.1763.0053 (visited on 12/04/2025).

Bickel, Peter J. and Kjell A. Doksum (Apr. 2015). *Mathematical Statistics: Basic Ideas and Selected Topics, Volume I, Second Edition*. English. 2nd edition. Chapman and Hall/CRC, Boca Raton. ISBN: 978-1-4987-2380-0.

Bishop, Christopher M. (Aug. 2006). *Pattern Recognition and Machine Learning*. English. Springer, New York. ISBN: 978-0-387-31073-2.

Blömer, Johannes et al. (2016). "Theoretical analysis of the k-means algorithm–a survey". In: *Algorithm Engineering: Selected Results and Surveys*. Lecture Notes in Computer Science. Springer, pp. 81–116.

Boser, Bernhard E., Isabelle M. Guyon, and Vladimir N. Vapnik (July 1992). "A training algorithm for optimal margin classifiers". In: *Proceedings of the fifth annual workshop on Computational learning theory*. COLT '92. Association for Computing Machinery, New York, NY, USA, pp. 144–152. ISBN: 978-0-89791-497-0. DOI: 10.1145/130385.130401. URL: https://doi.org/10.1145/130385.130401 (visited on 04/03/2022).

Breiman, Leo (Aug. 1996). "Bagging predictors". In: *Machine Learning* 24.2, pp. 123–140. ISSN: 1573-0565. DOI: 10.1007/BF00058655. URL: https://doi.org/10.1007/BF00058655.

– (Oct. 2001). "Random Forests". In: *Machine Learning* 45.1, pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324.

Breiman, Leo et al. (Oct. 2017). *Classification And Regression Trees*. Routledge, New York. ISBN: 978-1-315-13947-0. DOI: 10.1201/9781315139470.

Breslow, Leonard A. and David W. Aha (Jan. 1997). "Simplifying decision trees: A survey". en. In: *The Knowledge Engineering Review* 12.01. Publisher: Cambridge University Press, pp. 1–40. ISSN: 1469-8005, 0269-8889. DOI: 10.1017/S0269888997000015. URL: https://www.cambridge.org/core/journals/knowledge-engineering-review/article/abs/simplifying-decision-trees-a-survey/CEE7A6994E66E821DB4A12DD83DC3810 (visited on 08/07/2023).

Brown, Tom et al. (2020). "Language models are few-shot learners". In: *Advances in neural information processing systems* 33, pp. 1877–1901.

Buitinck, Lars et al. (Sept. 2013). "API design for machine learning software: experiences from the scikit-learn project". en. In: URL: https://hal.inria.fr/hal-00856511 (visited on 11/01/2022).

Cho, Kyunghyun et al. (Oct. 2014). "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, pp. 1724–1734. DOI: 10.3115/v1/D14-1179. URL: https://aclanthology.org/D14-1179 (visited on 10/20/2022).

Chung, Junyoung et al. (Dec. 2014). *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. arXiv:1412.3555 [cs]. DOI: 10.48550/arXiv.1412.3555. URL: http://arxiv.org/abs/1412.3555 (visited on 10/20/2022).

Cohen, David, Theodore B Lee, and David Sklar (2016). *Precalculus*. Cengage Learning.

Cortes, Corinna and Vladimir Vapnik (Sept. 1995). "Support-vector networks". en. In: *Machine Learning* 20.3, pp. 273–297. ISSN: 1573-0565. DOI: 10.1007/BF00994018. URL: https://doi.org/10.1007/BF00994018 (visited on 04/03/2022).

Cover, T. and P. Hart (Jan. 1967). "Nearest neighbor pattern classification". In: *IEEE Transactions on Information Theory* 13.1. Conference Name: IEEE Transactions on Information Theory, pp. 21–27. ISSN: 1557-9654. DOI: 10.1109/TIT.1967.1053964.

Davies, P (1988). *Kendall's Advanced Theory of Statistics. Volume 1. Distribution Theory*.

Devlin, Jacob et al. (2018). "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805*.

Drucker, Harris et al. (1996). "Support vector regression machines". In: *Advances in neural information processing systems* 9.

Efron, Bradley and Robert J Tibshirani (1994). *An introduction to the bootstrap*. CRC press.

Fisher, R. A. (1936). "The Use of Multiple Measurements in Taxonomic Problems". en. In: *Annals of Eugenics* 7.2, pp. 179–188. ISSN: 2050-1439. DOI: 10.1111/j.1469-1809.1936.tb02137.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-1809.1936.tb02137.x (visited on 05/15/2022).

Forgy, Edward W (1965). "Cluster analysis of multivariate data: efficiency versus interpretability of classifications". In: *Biometrics* 21, pp. 768–769.

Freund, Yoav and Robert E Schapire (Aug. 1997). "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting". en. In: *Journal of Computer and System Sciences* 55.1, pp. 119–139. ISSN: 0022-0000. DOI: 10.1006/jcss.1997.1504. URL: https://www.sciencedirect.com/science/article/pii/S002200009791504X (visited on 06/12/2022).

Friedman, Jerome H. (2001). "Greedy Function Approximation: A Gradient Boosting Machine". In: *The Annals of Statistics* 29.5. Publisher: Institute of Mathematical Statistics, pp. 1189–1232. ISSN: 0090-5364. URL: https://www.jstor.org/stable/2699986 (visited on 08/18/2022).

– (Feb. 2002). "Stochastic gradient boosting". en. In: *Computational Statistics & Data Analysis*. Nonlinear Methods and Data Mining 38.4, pp. 367–378. ISSN: 0167-9473. DOI: 10.1016/S0167-9473(01)00065-2. URL: https://www.sciencedirect.com/science/article/pii/S0167947301000652 (visited on 08/18/2022).

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2017). "Deep learning (adaptive computation and machine learning series)". In: *Cambridge Massachusetts*, pp. 321–359.

Graves, Alex, Greg Wayne, and Ivo Danihelka (2014). "Neural turing machines". In: *arXiv preprint arXiv:1410.5401*.

Guyon, Isabelle and André Elisseeff (Mar. 2003). "An introduction to variable and feature selection". In: *The Journal of Machine Learning Research* 3.null, pp. 1157–1182. ISSN: 1532-4435.

Hand, David J and Keming Yu (2001). "Idiot's Bayes—not so stupid after all?" In: *International statistical review* 69.3. Publisher: Wiley Online Library, pp. 385–398.

Hastie, Trevor et al. (2009). *The elements of statistical learning: data mining, inference, and prediction*. **2**. Springer.

Hinton, Geoffrey E et al. (2012). "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580*.

Ho, Tin Kam (Aug. 1995). "Random decision forests". In: *Proceedings of 3rd International Conference on Document Analysis and Recognition*. **1**, 278–282 vol.1. DOI: 10.1109/ICDAR.1995.598994.

Hochreiter, Sepp, Yoshua Bengio, et al. (2001). *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*.

Hochreiter, Sepp and Jürgen Schmidhuber (Nov. 1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735 (visited on 10/17/2022).

Hosmer Jr, David W, Stanley Lemeshow, and Rodney X Sturdivant (2013). *Applied logistic regression*. **398**. John Wiley & Sons.

Ioffe, Sergey and Christian Szegedy (July 2015). "Batch normalization: accelerating deep network training by reducing internal covariate shift". In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. JMLR.org, Lille, France, pp. 448–456. (Visited on 04/02/2022).

John, George H and Pat Langley (2013). "Estimating continuous distributions in Bayesian classifiers". In: *arXiv preprint arXiv:1302.4964*.

Jordan, M. I. and T. M. Mitchell (July 2015). "Machine learning: Trends, perspectives, and prospects". In: *Science* 349.6245. Publisher: American Association for the Advancement of Science, pp. 255–260. DOI: 10.1126/science.aaa8415. URL: https://www.science.org/doi/10.1126/science.aaa8415 (visited on 07/31/2023).

Kohavi, Ron and George H. John (Dec. 1997). "Wrappers for feature subset selection". en. In: *Artificial Intelligence*. Relevance 97.1, pp. 273–324. ISSN: 0004-3702. DOI: 10.1016/S0004-3702(97)00043-X. URL: https://www.sciencedirect.com/science/article/pii/S000437029700043X (visited on 05/09/2022).

Kohavi, Ron and David Wolpert (July 1996). "Bias plus variance decomposition for zero-one loss functions". In: *Proceedings of the Thirteenth International Conference on International Conference on Machine Learning*. ICML'96. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 275–283. ISBN: 978-1-55860-419-3. (Visited on 04/04/2023).

Kolen, John F. and Jordan B. Pollack (Oct. 1990). "Back propagation is sensitive to initial conditions". In: *Proceedings of the 1990 conference on Advances in neural information processing systems 3*. NIPS-3. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 860–867. ISBN: 978-1-55860-184-0. (Visited on 06/14/2022).

Kononenko, Igor (1990). *Comparison of inductive and naive Bayesian learning approaches to automatic knowledge acquisition*. **331**. Current trends in knowledge acquisition. IOS Press.

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (May 2015). "Deep learning". en. In: *Nature* 521.7553. Number: 7553 Publisher: Nature Publishing Group, pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539. URL: https://www.nature.com/articles/nature14539 (visited on 06/05/2022).

LeCun, Yann A. et al. (2012). "Efficient BackProp". en. In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 9–48. ISBN: 978-3-642-35289-8. DOI:

10.1007/978-3-642-35289-8_3. URL: https://doi.org/10.1007/978-3-642-35289-8_3 (visited on 03/26/2022).

Liu, Fei Tony, Kai Ming Ting, and Wei Fan (2005). "Maximizing Tree Diversity by Building Complete-Random Decision Trees". en. In: *Advances in Knowledge Discovery and Data Mining*. Ed. by Tu Bao Ho, David Cheung, and Huan Liu. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, pp. 605–610. ISBN: 978-3-540-31935-1. DOI: 10.1007/11430919_70.

Lloyd, S. (Mar. 1982). "Least squares quantization in PCM". In: *IEEE Transactions on Information Theory* 28.2. Conference Name: IEEE Transactions on Information Theory, pp. 129–137. ISSN: 1557-9654. DOI: 10.1109/TIT.1982.1056489.

Lu, Lu et al. (June 2020). "Dying ReLU and Initialization: Theory and Numerical Examples". In: *Communications in Computational Physics* 28.5. arXiv: 1903.06733, pp. 1671–1706. ISSN: 1815-2406, 1991-7120. DOI: 10.4208/cicp.OA-2020-0165. URL: http://arxiv.org/abs/1903.06733 (visited on 03/26/2022).

Luong, Minh-Thang, Hieu Pham, and Christopher D Manning (2015). "Effective approaches to attention-based neural machine translation". In: *arXiv preprint arXiv:1508.04025*.

Martinez, A.M. and A.C. Kak (Feb. 2001). "PCA versus LDA". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23.2. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 228–233. ISSN: 1939-3539. DOI: 10.1109/34.908974.

Mitchell, Tom M. (Sept. 1997). *Machine Learning*. English. 1st edition. McGraw-Hill Education, New York. ISBN: 978-0-07-115467-3.

Murtagh, Fionn and Pedro Contreras (2012). "Algorithms for hierarchical clustering: an overview". en. In: *WIREs Data Mining and Knowledge Discovery* 2.1. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/widm.53, pp. 86–97. ISSN: 1942-4795. DOI: 10.1002/widm.53. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/widm.53 (visited on 07/31/2023).

Nielsen, Frank (2016). *Introduction to HPC with MPI for Data Science*. Springer.

Nocedal, Jorge and Stephen J Wright (1999). *Numerical optimization*. Springer.

Opitz, D. and R. Maclin (Aug. 1999). "Popular Ensemble Methods: An Empirical Study". en. In: *Journal of Artificial Intelligence Research* 11, pp. 169–198. ISSN: 1076-9757. DOI: 10.1613/jair.614. URL: https://jair.org/index.php/jair/article/view/10239 (visited on 06/12/2022).

Pearson, Karl (Nov. 1901). "LIII. On lines and planes of closest fit to systems of points in space". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/14786440109462720, pp. 559–572. ISSN: 1941-5982. DOI: 10.1080/14786440109462720. URL: https://doi.org/10.1080/14786440109462720 (visited on 06/04/2022).

Quinlan, J. R. (Mar. 1986). "Induction of decision trees". en. In: *Machine Learning* 1.1, pp. 81–106. ISSN: 1573-0565. DOI: 10.1007/BF00116251. URL: https://doi.org/10.1007/BF00116251 (visited on 08/07/2023).

Rennie, Jason D. M. et al. (Aug. 2003). "Tackling the poor assumptions of naive bayes text classifiers". In: *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*. ICML'03. AAAI Press, Washington, DC, USA, pp. 616–623. ISBN: 978-1-57735-189-4. (Visited on 07/31/2023).

Rosenblatt, Frank (1957). *The Perceptron—a perceiving and recognizing automaton*. Tech. rep. Cornell Aeronautical Laboratory.

Rossi, Richard J (2018). *Mathematical statistics: an introduction to likelihood based inference*. John Wiley & Sons.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (Oct. 1986). "Learning representations by back-propagating errors". en. In: *Nature* 323.6088. Number: 6088 Publisher:

Nature Publishing Group, pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://www.nature.com/articles/323533a0 (visited on 04/03/2022).

Rumelhart, David E. and James L. McClelland (1987). "Learning Internal Representations by Error Propagation". In: *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*. Conference Name: Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations. MIT Press, pp. 318–362. ISBN: 978-0-262-29140-8. URL: https://ieeexplore.ieee.org/document/6302929 (visited on 10/17/2022).

Seber, George AF (2009). *Multivariate observations*. John Wiley & Sons.

Skansi, Sandro (2018). *Introduction to Deep Learning: from logical calculus to artificial intelligence*. Springer.

Sollich, Peter and Anders Krogh (1995). "Learning with Ensembles: How over-Fitting Can Be Useful". In: *Proceedings of the 8th International Conference on Neural Information Processing Systems*. NIPS'95. event-place: Denver, Colorado. MIT Press, Cambridge, MA, USA, pp. 190–196.

Srivastava, Nitish et al. (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56, pp. 1929–1958. ISSN: 1533-7928. URL: http://jmlr.org/papers/v15/srivastava14a.html (visited on 04/02/2022).

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le (2014). "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems* 27.

Utgoff, Paul E. (Nov. 1989). "Incremental Induction of Decision Trees". en. In: *Machine Learning* 4.2, pp. 161–186. ISSN: 1573-0565. DOI: 10.1023/A:1022699900025. URL: https://doi.org/10.1023/A:1022699900025 (visited on 08/07/2023).

Van Houdt, Greg, Carlos Mosquera, and Gonzalo Nápoles (Dec. 2020). "A review on the long short-term memory model". en. In: *Artificial Intelligence Review* 53.8, pp. 5929–5955. ISSN: 1573-7462. DOI: 10.1007/s10462-020-09838-1. URL: https://doi.org/10.1007/s10462-020-09838-1 (visited on 10/17/2022).

Vaswani, Ashish et al. (2017). "Attention is all you need". In: *Advances in neural information processing systems* 30.

Waggener, Bill and William N Waggener (1995). *Pulse code modulation techniques*. Springer Science & Business Media.

Werbos, Paul J. (Jan. 1988). "Generalization of backpropagation with application to a recurrent gas market model". en. In: *Neural Networks* 1.4, pp. 339–356. ISSN: 0893-6080. DOI: 10.1016/0893-6080(88)90007-X. URL: https://www.sciencedirect.com/science/article/pii/089360808890007X (visited on 10/17/2022).

Wolpert, David H. (Jan. 1992). "Stacked generalization". en. In: *Neural Networks* 5.2, pp. 241–259. ISSN: 0893-6080. DOI: 10.1016/S0893-6080(05)80023-1. URL: https://www.sciencedirect.com/science/article/pii/S0893608005800231 (visited on 06/12/2022).

Zhou, Zhi-Hua (2012). *Ensemble Methods: Foundations and Algorithms*. 1st. Chapman & Hall/CRC. ISBN: 1-4398-3003-7.