Machine Learning for Beginners

Learn to Build Machine Learning Systems Using Python

HARSH BHASIN

bpb

Machine Learning for Beginners

Learn to Build Machine Learning Systems Using Python

Harsh Bhasin

www.bpbonline.com

FIRST EDITION 2020

ISBN: 978-93-89845-42-6

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

Distributors:

BPB PUBLICATIONS
20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990/23254991

DECCAN AGENCIES
4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967/24756400

MICRO MEDIA
Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

BPB BOOK CENTRE
376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

Published by Manish Jain for BPB Publications, 20 Ansari Road, Darya Ganj, New Delhi-110002 and Printed by him at Repro India Ltd, Mumbai

Dedicated to

My Mother

About the Author

Harsh Bhasin is an Applied Machine Learning researcher. Mr. Bhasin worked as Assistant Professor in Jamia Hamdard, New Delhi, and taught as a guest faculty in various institutes including Delhi Technological University. Before that, he worked in C# Client-Side Development and Algorithm Development.

Mr. Bhasin has authored a few papers published in renowned journals including Soft Computing, Springer, BMC Medical Informatics and Decision Making, AI and Society, etc. He is the reviewer of prominent journals and has been the editor of a few special issues. He has been a recipient of a distinguished fellowship.

Outside work, he is deeply interested in Hindi Poetry, progressive era; Hindustani Classical Music, percussion instruments.

His areas of interest include Data Structures, Algorithms Analysis and Design, Theory of Computation , Python, Machine Learning and Deep learning.

About the Reviewer

- "**Yogesh** is the Chief Technology Officer at Byprice, a price comparison platform powered by advanced machine learning and deep learning models. He has successfully deployed 4 business critical applications in the last 2 years by harnessing the power of machine learning.

  He has worked with recommendation systems, text similarity algorithms, deep learning models and image processing.

  He is a visionary who understands how to drive product market fit for highly scalable solutions . He has 8 years of experience and has successfully deployed more than a dozen large scale B2B and B2C applications. He has worked as a senior software developer in one of Latin America's largest e-commerce company, Linio, which serves 15 million users every month.

  His vast experience in different fields of Software Engineering, Data Science and Storage Engines helps him in creating simple solutions for complex problems.

  He graduated in Software Engineering from Delhi College of Engineering, INDIA.

  He loves music, gardening and answering technical questions on StackOverflow."

Acknowledgments

"YOU DON'T HAVE TO BE GREAT TO START,
BUT YOU HAVE TO START TO BE GREAT."

— ZIG ZIGLAR

I would like to thank a few people who helped me to start. Professor Moin Uddin, former Vice-Chancellor, Delhi Technological University has been a guiding light in my life. Late Professor A. K. Sharma had always encouraged me to do better and Professor Naresh Chauhan, YMCA Institute of Science and Technology, Faridabad has always been supportive.

I would also like to thank my students Aayush Arora, Arush Jasuja, and Deepanshu Goel for their help. I would also like to thank BPB Publications for giving all the support provided when needed. Also would like to thank Yogesh for his efforts, for the feedback given by him.

Lastly, I would like to thank my mother and sister, my friends, and my pets: Zoe and Xena for bearing me.

Preface

Data is being collected by websites, mobile applications, dispensations (on various pretexts), and even by devices. This data must be analyzed to become useful. The patterns extracted by this data can be used for targeted marketing, for national security, for propagating believes and myths, and for many other tasks. Machine Learning helps us in explaining the data by a simple model. It is currently being used in various disciplines ranging from Biology to Finance and hence has become one of the most important subjects.

There is an immediate need for a book that not only explains the basics but also includes implementations. The analysis of the models using various datasets needs to be explained, to find out which model can be used to explain a given data. Despite the presence of excellent books on the subject, none of the existing books covers all the above points.

This book covers major topics in Machine Learning. It begins with data cleansing and presents a brief overview of visualization. The first chapter of this book talks about introduction to Machine Learning, training and testing, cross-validation, and feature selection. The second chapter presents the algorithms and implementation of the most common feature selection techniques like Fisher Discriminant ratio and mutual information.

The third chapter introduces readers to Linear Regression and Gradient Descent. The later would be used by many algorithms that would be discussed later in the book. Some of the important classification techniques like K-nearest neighbors, logistic regression, Naïve Bayesian, and Linear Discriminant Analysis have been discussed and implemented in the next chapter. The next two chapters focus on Neural Networks and their implementation. The chapters systematically explain the biological background, the limitations of the perceptron, and the backpropagation model. The Support Vector Machines and Kernel methods have been discussed in the next chapter. This is followed by a brief overview and implementation of Decision Trees and Random Forests.

Various feature extraction techniques have been discussed in the book. These include Fourier Transform, STFT, and Local Binary patterns. The book also discusses Principle Component Analysis and its implementation.

The concept of Unsupervised Learning methods like K-means and Spectral clustering have been discussed and implemented in the last chapter.

The implementations have been given in Python, therefore cheat sheets of NumPy, Pandas, and Matplotlib have been included in the appendix.

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors if any, occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

BPB is searching for authors like you

If you're interested in becoming an author for BPB, please visit **www.bpbonline.com** and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Machine-Learning-for-Beginners**. In case there's an update to the code, it will be updated on the existing GitHub repository.

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

PIRACY

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

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**.

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com**.

Table of Contents

Chapter 1
An Introduction to Machine Learning

With the advancements in technology, data collection has become easy. When you turn on location in your mobile, upload your pictures on Facebook or Instagram, fill online forms, browse websites, or even order items from an e-commerce website, your data is collected. What do companies do with this huge data? They analyze it, find your preferences, and this helps them in marketing. The advertisements being shown to you, generally, depending on the above things. Marketing professionals must lure you into buying something that you need or are even remotely interested in. Your data helps them. Likewise, the dispensation may keep track of suspicious activities using this data, may tract the source of transactions, or gather other important information using this data. However, this is easier said than done. It is a huge data, and its analysis cannot be done using conventional methods.

Let us consider another example to understand this. Suppose Hari visits YouTube every day and watches videos related to Indian Classical Music, Hindi Poetry, and watch Lizzie McGuire. His friend Tarush goes to YouTube and watches Beer Biceps and other videos related to workouts. After some time, YouTube starts suggesting different relevant videos to both of them. While Hari is shown a video related to Lizzie McGuire's reboot or Dinkar, in the recommended videos' list, Tarush is not recommended any such video. On the other hand, Tarush is shown a recommendation for a workout video.

It may be stated that recommendation requires an in-depth analysis and cannot be done solely based on any conventional algorithms. Those using e-commerce websites or famous music streaming apps like YouTube must be knowing that the recommendations are mostly good, if not excellent. Here the task is prediction. Your browsing history helps in this task, and for sure, it cannot be accomplished by conventional algorithms. Moreover, the betterment in the output, with time, means there is a well-defined performance measure for the task.

Machine learning comes to the rescue of those wanting to analyze this huge data, predict trends, find patterns, and so on. This chapter introduces machine learning, discusses it's types, explains how the given data is divided, and discusses its pipeline. This chapter also presents an overview of the history of machine learning and its applications.

Structure

The main topics covered in this chapter are as follows:

- Conventional algorithm and machine learning
- Types of learning
- Working
- Applications of machine learning
- History of machine learning

Objective

After reading this chapter, the reader will be able to learn the following topics:

- Understand the definition and types of machine learning
- Understand the working of a machine learning algorithm
- Appreciate the applications of machine learning
- Learn about the history of machine learning

Conventional algorithm and machine learning

The algorithmic solution of a problem requires the input data and a program to produce an output. Here, a program is a set of instructions, and output is generated by applying those instructions to the input data. In a machine learning algorithm,

the system takes the Input Data along with the examples of Output (in the case of supervised learning). It creates a model, which establishes (or tries to establish) some relation between the input and the output. Learning, in general, is improving the outcome using experience (E). How do we know that we have improved? The performance measure tells the performance of our model. As per Tom Michel, machine learning can be defined as follows.

If the performance measure (P) improves with experience (E) on task (T), then the system is said to have learned.

Here, the Task (T) can be Classification, Regression, clustering, and so on. The data constitutes Experience (E). The Performance Measure (P) can be any accuracy, specificity, sensitivity, F measure, Sum of Squared errors, and so on. These terms will be defined as we proceed. To understand this, let us consider an example of disease classification using Magnetic Resonance Imagining. If the number of patients correctly classified (accuracy) as diseased is considered as a performance measure, then this problem can be defined as follows:

- **T**: Classify given patients as diseased or not-diseased
- **P**: Accuracy
- **E**: The MRI images of a patient

The task will be accomplished by pre-processing the given data, extracting relevant features from the pre-processed data, selecting the most important features, applying a classification algorithm followed by post-processing. In general, a machine learning pipeline constitutes the following steps (*Figure 1.1*):

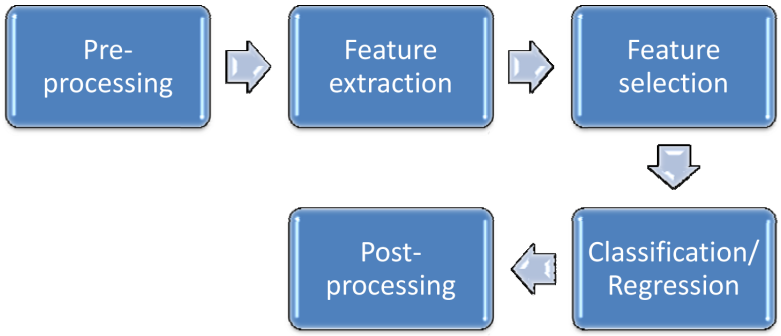


Figure 1.1: *Machine learning pipeline*

These terms will become clear in the following chapters. **Pre-processing** has been discussed in the second chapter. The chapter also introduces the idea of **Feature selection**. The next six chapters discuss supervised learning techniques, and the last chapter introduces **Feature extraction**. I decided to discuss Feature extraction at the

end because some of the techniques require the knowhow of concepts introduced in the previous seven chapters. Having seen the definition of machine learning, let us now have a look at its types.

Types of learning

Machine learning can be classified as supervised, unsupervised, or semi-supervised. This section gives a brief overview of the types.

Supervised machine learning

This type of learning uses the labels of the data in training set to predict the label of a sample in the test set. The training set acts as a teacher in this type of algorithm, which supervises the training process. The data in these algorithms contain samples and their correct labels. The training process tries to uncover the pattern hidden in the data. That is, the learning aims to relate the labels Y with the data X as $y = f(x)$, where x is a sample, and y is the label.

If this label is a discrete value, then the process is termed as classification. If y is a real value, then it is called regression. *Chapter 3* of this book introduces a regression, and *Chapter 4* to *Chapter 8* discusses classification algorithms.

Examples of classification are face detection, voice detection, object detection, and so on. Classification essentially means placing the given sample into one of the predefined categories. Examples of regression include predicting the price of a commodity, predicting temperature, housing price, and so on.

Unsupervised learning

This type of learning uses input *Data(X)* but no labels. The learning aims to learn about the data by grouping the like samples or by deducing the associations. Since there is no teacher involved in the algorithm, it is called unsupervised learning. Clustering and association come under unsupervised learning. Clustering uncovers the groupings in the data. Association, on the other hand, uncovers the rules which associate the events. *Chapter 9* of this book discusses clustering.

There is something in between supervised and unsupervised learning. It is called semi-supervised learning. In this type of learning, a part of the input data may be labeled. Many practical problems fall into this category.

Working

This section discusses the working of a machine learning algorithm. We begin with understanding the data. It is followed by the division of data into train and test sets. The learning algorithm is then applied to the training data, and the performance is then measured.

Data

In the discussion that follows, the data is represented by X, which is a matrix with n rows and m columns ($n \times m$ matrix). Here, n is the number of samples, and m is the number of features in each sample. The labels are represented by y, which is a ($n \times 1$ matrix). It may be noted that the i^{th} row of y contains the label corresponding to the i^{th} row of X.

For example, consider the Wine dataset available at the UCI Machine Learning Repository. The data considers attributes of wines from three different cultivars but from the same region in Italy. The dataset has 13 features, which are as follows (as per the official documentation at https://archive.ics.uci.edu/ml/datasets/Wine):

1. Alcohol
2. Malic acid
3. Ash
4. Alkalinity of ash
5. Magnesium
6. Total phenols
7. Flavanoids
8. Nonflavanoid phenols
9. Proanthocyanins
10. Color intensity
11. Hue
12. OD280/OD315 of diluted wines
13. Proline

The label is the class of the Wine (1, 2, or 3). The number of samples in the dataset is 178. That is, the values of the 13 features determine the class of Wine. The value of n is 178, and that of m is 13. The data, X, is 178×13 array, and the response variable, y, is a 178×1 array. It is followed by pre-processing, which involves many things, including removing null values. Some of these techniques have been discussed in the second chapter. Once you have got the data, create a train, and a test set out of the data.

Train test validation data

Suppose you are given the responsibility of teaching a topic to a group of students. How will you find whether your students have understood the topic? You will probably take a test, and based on the performance of the test; you will judge how well the topic has been understood. Wait! The performance is indicative of the learning only if the questions asked are not the same as those discussed while teaching (or during training). It is because giving the same questions in the test will judge how well the students can memorize, not their understanding of the topic.

The same is true for a machine learning model. The data used in the training phase should not be used for testing. So, to have confidence in the model, the given data is divided into two parts train and test *Figure 1.2*:

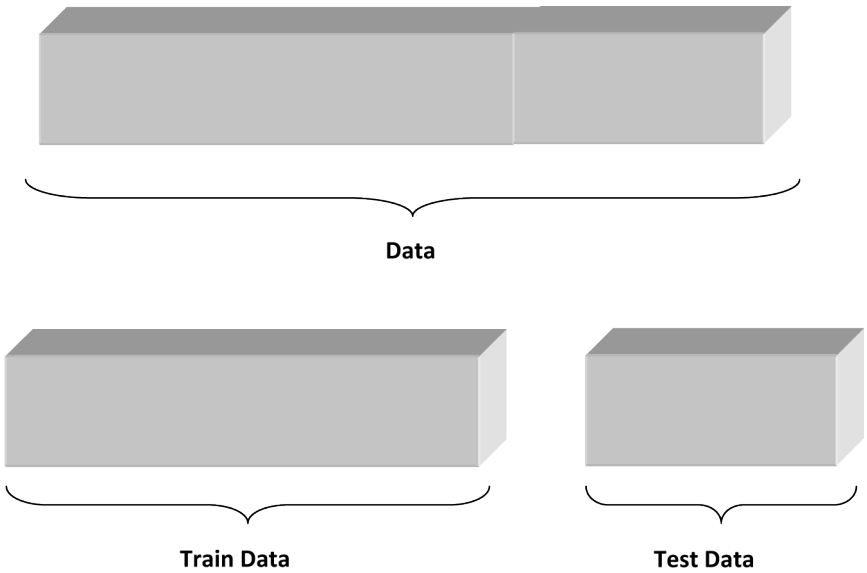


Figure 1.2: *Splitting the data into train set and test set*

Well! It may also mislead us in believing that the model so developed is good (or bad). So, we randomly split the data into train data (x %) and test data (100-x%) and find the accuracy. Repeat this process many times and take the average accuracy. It increases the confidence in the model so developed.

It may also be stated that while training, we may need to use some data for testing. We cannot use the test data because of the reasons stated above. So we divide the train data into train and validation. Train the model using the train data, and once the parameters are learned. Test the model using the test data:

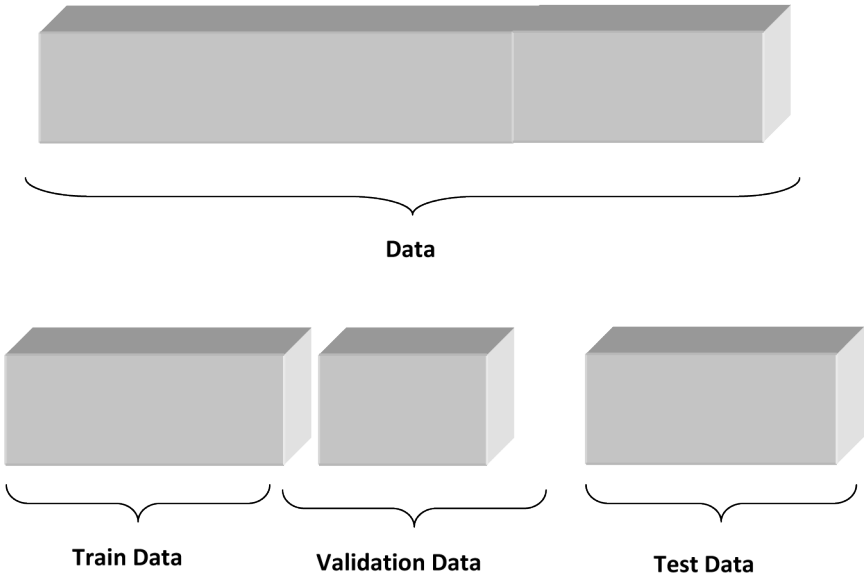


Figure 1.3: *Splitting the data into train set, test set, and validation set*

While learning, this validation data can be used to see the training performance. Once the model has learned, the test data can be used to test the model.

Another approach to validation is referred to as cross-validation. In this approach, the given data is divided into *k* parts. Out of these *k* parts (say part 1), one is used for testing and the rest for training. The process is repeated, taking part 2 as the test data and the rest as the train data. Likewise, *k* such models are created, and the average performance of these *k* models is reported. For example, in *Figure 1.4*, the value of *k* is 6. The data is split into six parts, and in each iteration, one of the parts is used for testing and the rest for training. The performance of the model is reported as the average of the six models:

To summarize, *K-Fold* is better than the train-test split as it gives more confidence in the results. However, the volume of the data must be considered before applying *K-Fold*. Also, in *K-Fold*, you take the average performance of *K* models and declare it as the output. Having seen the methods of division of data into train and test, let us move forward.

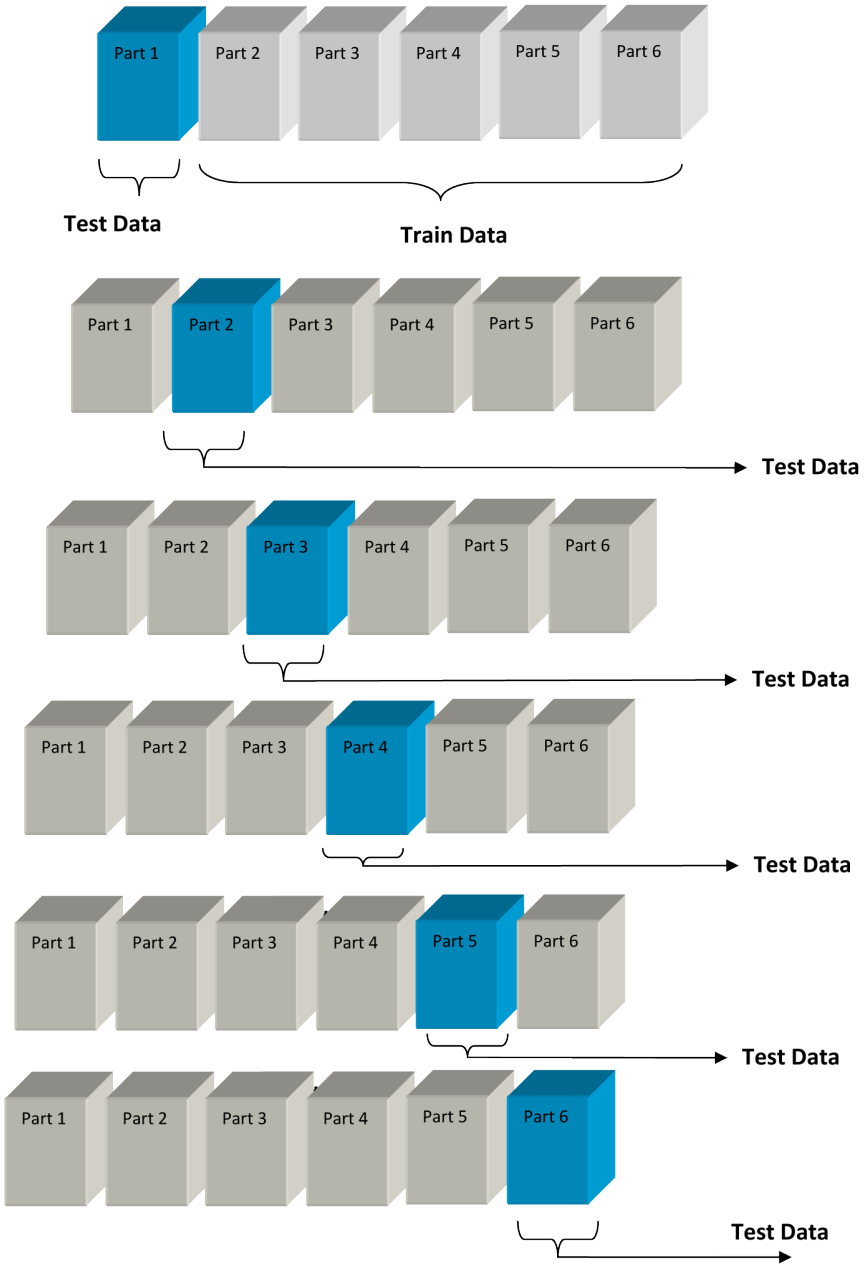


Figure 1.4: *K Fold Validation: K=6*

Rest of the steps

The division of data is followed by choosing the target function and its representation. The learning algorithms are then applied to the training set. The algorithm learns its

parameters using the training set and then applies them to the test set. We will learn about learning about our journey. The performance measures, which tell you how good your algorithm is, are discussed in the next chapter.

Having seen the outline of the machine learning pipeline, let us have a look at some of the exciting applications of machine learning.

Applications

Machine learning is an involved task, and along with other things, it also requires algorithms that learn from data. Machine learning has successfully been applied in many domains and disciplines. From Social Science to Drug Development, ML has proved it's mental everywhere. It is creating history, and we are a part of this history. Let us just not watch it, let us live this history and immerse ourselves in ML.

Natural Language Processing (NLP)

NLP aims to process and understand natural language. It involves linguistics, engineering, and artificial Intelligence. The advancements in ML have greatly helped the field. One of the fascinating examples of the advancement in this field is Alexa, voice assistance by Amazon.

Weather forecasting

This discipline aims at predicting the weather conditions at a particular location, using the past data available. It may be stated that even before the advent of machine learning, or even computers, the weather was being predicted. However, the latest technologies have helped improve this prediction.

Robot control

As per the literature review, the mechanical parts of the robot are generally controlled by software. This software may fail if it does not update itself with time, or learn. At this point, ML comes into play. ML helps a robot in making intelligent decisions using the training data.

Speech recognition

Speech recognition aims at the translation of spoken languages by computers. It requires computational linguistics and Computer Science. This field helps in understanding speech.

Business Intelligence

The data collected by a company must be converted into a form, which helps us in making decisions. This field helps in analyzing data to create the following:

- Reports
- Graphs
- Dashboards

The above helps in generating actionable insights into the data. Along with the above, ML has been successfully used in diagnosing diseases using medical imaging and many other things. So, machine learning is interesting. Let us have a brief look at how the story started.

History

One of the central characters in this play is the Neural Network. The story of this character started when Warren McCulloch and Walter Pitts published a paper explaining the working of Neurons in 1943. They created an electric circuit of this model. The model laid the foundation for machine learning. This model has been discussed in the fourth chapter of this book.

The Turing Test was conceived in the 1950s to find whether the user is a human or a computer. This test laid the foundation for accessing algorithms used in many ML applications, including chatbots. Many researchers contributed to incremental research on these topics.

One of the most exciting events in the history of ML is the creation of the first computer program to play checkers by Samuel. The McCulloch Pitts model was also being studied and analyzed. The perceptron model was created by Rosenblatt in 1958. Windrow and Hoff created two models ADELINE for binary classification and MADELINE, which could eliminate echo.

The researchers came up with novel ways to handle the inability of these models to detect non-linearly separable data. John Hopfield created a network that had bidirectional lines in 1982.

The development of Multi-Layer Perceptron helped in classifying the non-linearly separable data also. The discipline got a boost with the invention of Support Vector Machines, which did not need the whole data to create a separating hyperplane and were based on the concept of maximum margin classifier.

The second half of the ‘90s witnessed some amazing inventions. IBM developed Deep Blue in 1997, which was a chess-playing computer. It was followed by the

development of tools that could handle large data and hardware, which could process faster.

Currently, we are in the Deep Learning Age. GoogleBrain was developed in 2012. It was a deep neural network created by Jeff, which focused on pattern detection in images and videos.

AlexNet was also developed by 2012. It won the ImageNet competition by a large margin in 2012, which led to the use of GPUs and Convolutional Neural Networks in machine learning. DeepFace was developed in 2014. It is a Deep Neural Network created by Facebook, which they claimed can recognize people with the same accuracy as a human can. The research in the field of ML and DL continues and will hopefully amaze ourselves in the future also.

Conclusion

Machine learning is being used in diverse application domains. It is both useful and interesting. The creation of hand-crafted features, selection of features, application of models, and evaluation of performance is no alchemy. It involves a deep understanding of Computer Science, statistics, and many other disciplines. Moreover, it can be applied in many application domains and can be used for the good of the society. It is being used for predicting diseases, understanding the effects of medicines, and so on. This chapter introduced machine learning. The definition, types, and processes have been discussed in this chapter. The chapter also threw light on the applications and the history of machine learning.

This chapter helps you to get hold of the fundamentals. It also aims to help you motivate yourself towards learning machine learning. The reader must have also understood the pipeline and the division of data into train and test set. Also, this chapter forms the basis of the following chapters. The next chapter takes the discussion forward and introduces pre-processing and feature selection. Welcome to your journey towards becoming a machine learning professional! Before proceeding any further, let us spare a minute to see what we have learned.

Exercises

Multiple Choice Questions

1. Which of the following is not a type of machine learning?
   a. Classification
   b. Regression

c. Clustering
d. All of the above are the types of machine learning

2. Which of the following is a type of supervised learning?
a. Classification
b. Regression
c. Both
d. None of the above

3. Which of the following is a type of unsupervised learning?
a. Clustering
b. Finding association rules
c. Both
d. None of the above

4. In the case of supervised learning:
a. Both X and y are given
b. Only X is given
c. X and the labels of some of the samples are given
d. None of the above

5. In the case of unsupervised learning:
a. Both X and y are given
b. Only X is given
c. X and the labels of some of the samples are given
d. None of the above

6. In the case of semi-supervised learning,
a. Both X and y are given
b. Only X is given
c. X and the labels of some of the samples are given
d. None of the above

7. If the label y in the case of supervised learning is discrete, then it is referred to as
a. Classification
b. Regression
c. Both
d. None of the above

8. If the label y in the case of supervised learning is a real value, then it is referred to as
a. Classification
b. Regression
c. Both
d. None of the above

9. Can we use the whole input data for training?

a. Yes
b. No
c. Depends on the problem
d. None of the above

10. Which of the following may not produce better performance in training, but will lead to a robust model?

a. 70% train, 30% test
b. 30% train, 70% test
c. K-Fold
d. None of the above

11. The performance of K-Fold represents:

a. The performance of the final model
b. The average performance of all the models developed
c. Depends on the situation
d. None of the above

12. If the value of K in K-Fold is 1, then:

a. All the data would be used in training
b. All the data would be used in testing
c. Depends on the situation
d. None of the above

13. If the value of K in K-Fold is same as the number of input samples, then:

a. All the data would be used in training
b. All the data would be used in testing
c. Depends on the situation
d. None of the above

14. Which of the following is a part of an ML pipeline?

a. Feature extraction
b. Feature selection
c. Learning
d. All of the above

15. Which of the following must be performed, in case of data having very high dimensionality?

a. Feature selection
b. Pre-processing
c. Both
d. None of the above

Theory

1. Define machine learning. Explain the difference between a conventional algorithm and an ML algorithm.
2. What is supervised learning? Give examples to explain why it is called supervised learning.
3. What is unsupervised learning? Give examples to explain why it is called unsupervised learning.
4. Define classification.
5. Define regression. How is it different from classification?
6. Define clustering. Give examples.
7. Explain the importance of dividing the data into train and test set.
8. What is the validation data? How is it useful while training?
9. Explain K-Fold validation. How is the performance of a model measured using K-Fold validation?
10. State some of the applications of machine learning. Write a brief note on the history of machine learning.

Explore

As you proceed, you will need data for your experiments. Explore the following link to find out the various data available for:

- Classification
- Regression
- Clustering

https://archive.ics.uci.edu/ml/index.php

Perform the following tasks:

1. Download three datasets of each type and read the data using Python.
2. Find the number of features and the number of samples in each dataset.
3. Find out the data type of each feature.
4. Find the statistical description of each feature (mean, median, mode, standard deviation).

Chapter 2

The Beginning: Pre-Processing and Feature Selection

Introduction

Machine learning is an intricate task. Learning from the data involves cleaning of data, extracting features, selecting relevant features, and applying learning algorithms. The first and most important task is to clean the data. The data may contain missing values due to the reasons discussed in the chapter. The data may also contain "Not a Number" or "NaN"'s. Since missing data will hamper the learning process or, worse, will make the model learn incorrectly, dealing with such values is essential. This chapter gives an overview of how to deal with such values. Cleaning of data will enhance the performance of our Machine Learning model and make the results more meaningful. At times it is essential to convert the continuous data to categorical one. This chapter also introduces the discretization of data.

Once the data is clean, we extract the features and then move to the selection of relevant features. The feature extraction methods have been discussed in the following chapters. The second part of this chapter focuses on feature selection. Some of the statistical methods of feature selection have been discussed in this chapter. The chapter is important as it is the starting point of the road to machine learning. The methods discussed in the chapter will make your model efficient and effective.

Structure

The main topics covered in this chapter are as follows:

- Dealing with the missing values and 'NaN"s
- Feature selection
- Chi-Squared test
- Using variance to select features
- Pearson correlation

Objective

After reading this chapter, the reader will be able to:

- Understand the importance of dealing with the missing values
- Understand the importance of feature selection
- Understand the Chi-Squared test and Pearson's correlation

Dealing with missing values and 'NaN'

If your data contains missing values or 'Not-a-Number' type values, you might not be able to apply the standard procedures to carry out tasks like classification, regression, and so on. Therefore these missing values must be dealt-with before learning from the data. Knowing the source of the missing data can greatly help you in identifying the proper techniques to deal with them. These values may occur due to numerous reasons. Some of the most common ones are as follows:

- Incomplete filling of forms by the users
- If the database is migrated from some other, some data may have been lost
- Errors due to programs or due to other technical reasons

To deal with a missing value, one must:

- Find information about the feature which contains missing values. For example, if all the values in a feature are in a given range, the probability of the missing value lying in that range is high.
- Find the type of feature. For example, if a feature contains string type values, the missing value must also be a string.
- Find if the missing value can be replaced with obvious value. For example, if the column named country, in the regular employee table of a company located in Faridabad, has a missing value, its probability of being "India" is high.

There are many ways to deal with such values, but the application of any technique requires an in-depth analysis of the data at hand and due deliberations. It is because filling these placeholders with incorrect data may lead to the formation of a model that does not perform well.

Some of the most common ways to deal with the missing values are as follows:

- Ignoring the records having missing values: This solution is problematic if the number of samples is low. But in the case of data with many samples, this solution might work.
- Replacing the missing values with average/median: This may not be possible if the range of the values is too large. However, if this range is low, this solution might work.
- Replacing the missing values with those obtained from regression: This book contains a dedicated chapter on regression. Applying the regression techniques for finding out the missing values is one of the options, but this may not work in some of the cases.

According to many data scientists, replacing missing values with the mean is not a good idea. It is because replacing many values with mean will reduce the variance and would undermine its correlation with other features. To understand the procedure of dealing with the missing values, consider the following data.

This section uses a file called `Research_data.csv`. The file contains seven fields:

- `R_ID`: Research ID of the researcher
- `F_Name`: The first name of the researcher
- `L_Name`: The last name of the researcher
- `No_Books`: The number of books authored by him/her
- `No_Papers`: The number of papers authored by him/her
- `R_Score`: The Research Gate score of the researcher

The data types of the fields are as follows:

- `R_ID`: Integer
- `F_Name`: String
- `L_Name`: String
- `No_Books`: Integer
- `No_Papers`: Integer
- `R_Score`: Float

The contents of the file are as follows (*Table 2.1*):

	R_ID	F_Name	L_Name	No_Books	No_papers	R_Score
0	1001.0	Harsh	Bhasin	9	25	15.37
1	1002.0	Kumar	Gaurav	NaN	10	10.23
2	1003.0	Lovish	Kundu	NaN	8	8.20
3	NaN	Arush	Jasuja	2		NaN
4	1005.0	Kim	parsons	Nan	15	12.56
5	1006.0	Pulin	Verma	NaN	2	4.00
6	1007.0	ABC	XYZ	5	8	7.21
7	1008.0	LMN	QRS	1	13	9.76

Table 2.1: *The given table contains missing values and NaN's*

Note that the file contains NaN and missing values. The following steps take the reader on the journey of dealing with the missing values. The following code uses Pandas.

1. **Importing Numpy and Pandas:**

```
import pandas as pd
import numpy as np
```

2. **Observing a few rows of the given data:** Use `read_csv` method to read the file and the head method to see the first few rows of the file

```
df = pd.read_csv("property_data.csv")
# The first few rows
data=df.head()
print(data)
```

Output:

	R_ID	F_Name	L_Name	No_Books	No_papers	R_Score
0	1001.0	Harsh	Bhasin	9	25	15.37
1	1002.0	Kumar	Gaurav	NaN	10	10.23
2	1003.0	Lovish	Kundu	NaN	8	8.20
3		NaN	Arush	Jasuja	2	NaN
4	1005.0	Kim	parsons	Nan	15	12.56

Table 2.2: *Output of Step 2*

3. **Finding 'NAN' values and 'NAN' type-values:** The NAN values in the data can be found by the `isnull` method of a column of a dataframe:

```
# Finding Nan's in the No_Books column
print(df['No_Books'])
print(df['No_Books'].isnull())
```

Output:

```
0     9
1     NaN
2     NaN
3     2
4     Nan
5     NaN
6     5
7     1
Name: No_Books, dtype: object
0     False
1     True
2     True
3     False
4     False
5     True
6     False
7     False
Name: No_Books, dtype: bool
```

Note that the above method does not work if a field has 'Nan' or any other value in place of 'NaN.' Moreover, the method will not work if the value of a field having an integer data type is empty. For example, if the method is applied to the `No_Papers` field, the expected results are not obtained. Note that in the example that follows, `False` is displayed for row 3:

```
print(df['No_papers'])
print(df['No_papers'].isnull())
```

Output:

```
0     25
```

```
1     10
2     8
3
4     15
5     2
6     8
7     13
Name: No_papers, dtype: object
0     False
1     False
2     False
3     False
4     False
5     False
6     False
7     False
Name: No_papers, dtype: bool
```

The solution of the above problem is to find the unique values in the column (or manually observe the column) and replace each 'NaN' like values with 'NaN'. This can be done by assigning `na_values` to the list containing possible 'NaN' values in the column:

```
missing = ["n/a", "na", "Nan"," "]
df = pd.read_csv("Research_data.csv", na_values = missing)
```

The above statements will result in the replacement of all the values in the missing list with the standard 'NaN.' It can be observed by printing the values of the affected fields:

```
print(df['No_Papers'])
print(df['No_papers'].isnull())
```

Output:

```
0     3.0
1     3.0
2     NaN
3     1.0
```

```
4     3.0
5     NaN
6     2.0
7     1.0
8     NaN
Name: NUM_BEDROOMS, dtype: float64
0     False
1     False
2     True
3     False
4     False
5     True
6     False
7     False
8     True
Name: NUM_BEDROOMS, dtype: bool
```

4. **Dealing with the 'NAN' in an integer type column:** Note that the above method will not work if a column contains characters, and one of the values is an integer. To consider the integer value in a column having data type character, as 'NaN,' the following code can be used:

```
count=0
for item in df[]:
   try:
      int(item)
      df.loc[count, ]=np.nan
   except ValueError:
      pass count+=1
```

   Likewise, to convert any value having a particular data type, the value code can be used by replacing the type.

5. **Counting the total number of missing values in each column:** To count the total number of missing values in each column, the following statement can be used:

```
print(df.isnull().sum())
```

Output:

```
R_ID         1
F_Name       0
L_Name       0
No_Books     3
No_papers    0
R_Score      1
dtype: int64
```

6. **Finding the total number of missing values:** To find the total number of missing values in the data, the following statement can be used:

   ```
   print(df.isnull().sum().sum())
   ```

   Output:

   ```
   5
   ```

7. **Replacing the 'NaN' values with a particular value in a particular column:** To replace the 'NaN' values with a particular value in a particular column, the following statement can be used:

   ```
   df[<column name>].fillna(<value>, inplace=True)
   ```

8. **Replace 'NAN' with the mean:** To replace the 'NaN' with the mean of the column, the following statement can be used:

   ```
   m = df[]<column name>].mean()
   df[<column name>].fillna(m, inplace=True)
   ```

   As stated earlier, replacing the missing value with the `mean` is not recommended.

9. **To drop the records with 'NaN' values:** To drop the records with 'NaN' values, the following statement can be used:

   ```
   df.dropna()
   ```

   Output:

	R_ID	F_Name	L_Name	No_Books	No_papers	R_Score
0	1001.0	Harsh	Bhasin	9	25	15.37
4	1005.0	Kim	parsons	Nan	15	12.56
6	1007.0	ABC	XYZ	5	8	7.21
7	1008.0	LMN	QRS	1	13	9.76

Table 2.3: *The output of Step 9*

Having seen the ways to deal with the missing values, let us now move to the process of discretization. The next section gives a brief overview of the process.

Converting a continuous variable to categorical variable

A continuous variable can be converted into a categorical variable in many ways, some of which have been explored in this chapter.

To begin with, a continuous variable can be converted into binary by setting the value less than a threshold to 0 and those greater than the threshold as 1. This process is called Dichotomizing. To understand the need for this, consider a study about happiness with the present dispensation where your happiness question ranges from 1 to 10. You might be interested in categorizing the results as either great happiness or low happiness. To accomplish this task, you can also split a continuous variable into two parts. Likewise, to divide the values into *n* parts, a similar process can be applied.

The following code converts the Fisher-Iris data into categorical values. In the first step, the required modules have been imported. The second step extracts the data and asks for the value of the number of levels. The third step performs the required task:

1. Import the required modules:

```
import numpy as np
from sklearn import datasets
data= datasets.load_iris()
```

2. Extracting data:

```
x=data.data[:100,:]
x=np.array(x)
print(x.shape)
y=data.target[:100]
y=np.array(y)
print(y.shape)
n=int(input('Enter the value of n \t:'))
```

3. Carry out categorization:

```
for i in range (x.shape[1]):
```

```
        x1=x[:,i]
        max1=np.max(x1)
        min1=np.min(x1)
        step=(max1-min1)/n
        print(max1, ' ',min1, ' ',step)
        for k in range (n):
            a=min1+(step*k)
            b=min1+(step*(k+1))
        for j in range(x.shape[0]):
            if ((x[j,i]>=a) and (x[j,i]<=b)):
                x[j,i]=k
    print(x)
```

In case of data having a very large number of features, feature selection is used. The following sections give an in-depth overview of some of these methods.

Feature selection

The pre-processing of data follows feature extraction. These features will be used to create a feature set, which will help in learning. Feature extraction has been discussed in the last chapter of this book. At times, the features so obtained are huge in number. For example, if you have a 256×256 picture and wish to consider each pixel of the picture as a feature, you will have 2^{16} features. A picture with more pixels will have more features. Likewise, a video will have an even larger number of features. The extraction techniques, discussed in the following chapters, help to find better features. Some techniques even represent the given data with a lesser number of features. However, not all the features so obtained are equally important. Some of them are redundant, and some are noisy. The redundant features do not enhance the performance of a model, and the noisy features may degrade the performance of a model. Therefore, a smaller, more relevant subset of features needs to be selected to carry out the required learning task efficiently and effectively. So, having a larger number of features results in the following problems:

- Learning with a larger number of features is computationally expensive
- Some features are redundant
- Some features are noisy

Feature selection aims at better performance and reduced learning time. This selection can be classified as follows:

- Feature selection: This method aims at selecting the most relevant features from the given set of features.
- Feature elimination: This method aims at eliminating the irrelevant features from the given set of features.

Feature selection can again be classified as follows:

- Filter methods: The filter methods select the features considering their relation with the labels.
- *Wrapper methods*: The wrapper method uses the classifier/repressor to find the subset that gives the best performance.

Except for the above, some learning methods have inbuilt feature selection mechanisms. Feature selection methods generally arrange features in order of their importance. To do this, these methods may consider a single feature or a set of features. The former is referred to as Univariate methods, and the later are called Multivariate methods.

Univariate feature selection methods place features in order of their relation with the output variables. It can be done using statistical tests. Some of the prominent methods that can be used for this purpose are as follows:

- Chi-Squared
- Variance based
- Correlation-based
- ANOVA

The next three sections discuss Chi-Squared, Variance based, and Correlation-based methods for feature selection.

Chi-Squared test

Feature selection in case of data having categorical variables can be made using the χ^2 test. A categorical variable is one that can take values from a given set. The test is applicable for categorical variables, so continuous data is converted into categorical data by applying the techniques studied in the last-but-one section.

"The Pearson's Chi-Squared (χ^2) test supposes that the expected frequencies of a categorical variable match the observed frequencies for that variable [1]."

The observed frequencies of variables can be determined using the given data and their expected frequencies using the methods explained in the following discussion. If the value of the χ^2 is large, the expected and the observed frequencies are far apart. In case this value is low, the two are nearer.

The result obtained on applying the test needs to be compared with the critical value. This critical value is found using tables by calculating the degree of freedom. The degree of freedom is found using the following formula:

*degreesoffreedom: (numberofrows - 1) * (numberofcolumns - 1)*

Formally, the test can be stated as follows:

χ^2 Test:

- Reject Null hypothesis is χ^2 >= Critical Value
- Do not reject the Null hypothesis if χ^2 < Critical Value

The following steps explain the process of finding the χ^2. The code uses Pandas. The following examples use a file called `Dept_paper_Data.csv`. The file contains four fields:

- `R_ID`: Research ID of the researcher
- `Department`: The department of the researcher
- `No_Papers`: The number of papers authored by him/her
- `Patent`: If the researcher has a patent

The data types of the fields are as follows:

- `R_ID`: Integer
- `Department`: String
- `No_Papers`: Integer
- `Patent`: String

The contents of the file are as follows (*Table 2.4*):

	R_ID	Department	No_papers	Patent
0	H001	CS	3	Y
1	H002	CS	5	NA
2	H003	CS	2	NA
3	H004	CS	9	N
4	H005	CS	1	Y
5	H006	CS	6	NA

Contd…

6	H007	CS	1	N
7	H008	ECE	2	NA
8	H008	ECE	0	N
9	H009	ECE	4	N
10	H010	ECE	10	Y
11	H011	ECE	3	N
12	H012	ECE	1	Y

Table 2.4: *The research output table*

It is intended to find whether the value of the `Patent` field depends on the `Department` of the researcher. The following steps will help us to apply Chi-square to accomplish the given task.

In the above example, `Patent` can take values:

- 'Y'
- 'N'
- 'NA'

And the `Department` can take the following values:

- 'CS'
- 'ECE'

And are hence categorical variables. Therefore, the test can be used to accomplish the given task. Firstly, the summary of the values of categorical variables of the two features needs to be drawn. It can be done using a table called a **contingency table**. *Table 2.5* shows the format of the contingency table:

_	**'Y'**	**'NA'**	**'N'**
'CS'	2	3	2
'ECE'	2	1	3

Table 2.5: *Format of the contingency table*

The following steps will help us to apply The Pearson's Chi-Squared Chi-square to accomplish the given task:

1. Find unique values from the `Department` column and place them in a list called `departments`. Likewise, find unique values from the `Patent` column and place them in a list called `patent_values`:

```
departments=df['Department'].unique()
print(departments)
patent_values=df['Patent'].unique()
print(patent_values)
```

Output:

```
['CS' , 'ECE']
['Y' , 'NaN', 'N']
```

2. Find the records where `Patent` is 'Y' and `Department` is 'CS'. Likewise, find the number of records for all six combinations. The task can be accomplished by using the following code:

```
table=np.zeros((2,3))
i=0
j=0
for dept in departments:
    j=0
    for val in patent_values:
        a=(df[(df['Department']==dept) & (df['Patent']==val)].count())
        table[i,j]=a[0]
        j+=1
    i+=1
print(table)
```

Output:

```
array([[2., 3., 2.], [2., 1, 3.]])
```

Having obtained the contingency table, we can proceed further and find the expected frequencies, as shown in *Table 2.6*. Note that the last column contains the sum of values in that row, and the last row contains the sum of values of that column:

_	'Y'	'NA'	'N'	
'CS'	2	3	2	7
'ECE'	2	1	3	6
	4	4	5	

Table 2.6: *Calculating expected frequencies*

The expected frequency of the cell at the intersection of 'CS 'and 'Y' is:

$$\frac{7 \times 4}{13}$$

It is obtained by multiplying the value at the last cell of the corresponding column with the last cell of the corresponding row and dividing the result with the last cell of the table. Likewise, the expected frequency of each cell (*Ei*) can be calculated. The value of the chi-squared metric is calculated by using the following formula:

$$\chi^2 = \sum (O_i - E_i) / (E_i)$$

The table value corresponding to the degree of freedom, which in this case is two is then compared with the calculated value, and decision regarding dependence is made.

The implementation of this test using SKLearn is as follows: In the code that follows, the `chi2_contingency` and `scipy.stats` needs to be imported. The module has a method called `chi2_contingency`, which takes the contingency table as its argument and outputs the value of the statistics, the `p_value`, and the degree of freedom (`d_of_freedom`). The critical value can be obtained by using the `chi2.ppf` method, which takes probability and the degree of freedom as its arguments. The comparison between the two values is then made, and the result is printed:

```
from scipy.stats import
from chi2
value, p_value, d_of_freedom, expected = chi2_contingency (table)
print(expected)
probability = 0.95
critical = chi2.ppf(probability, d_of_freedom)
if abs(value) >= critical:
  print('The variables are Dependent')
else:
  print('The variables are not Dependent')
alpha = 1.0 - probability
print('significance=%.2f, p=%.2f' % (alpha, p_value))
if p_value<= alpha:
  print('Reject')
else:
  print('Do not reject')
```

Output:

```
[[2.15384615 2.15384615 2.69230769]
[1.84615385 1.84615385 2.30769231]]
The vairbles are not Dependent
significance=0.05, p=0.57
Do not reject
```

Having seen the implementation of Chi-squared, let us now move to another feature selection technique called the **Pearson Correlation**.

Pearson correlation

Pearson's coefficient of correlation between two variables X and y is given by:

$$corr(X,y)=\left(\sum_{i=1}^{n}\left(X_i-\overline{X}\right)\times\left(y_i-\overline{y}\right)\right)/\left(\sqrt{\sum_{i=1}^{n}\left(X_i-\overline{X}\right)^2}\times\sum_{i=1}^{n}\left(y_i-\overline{y}\right)^2\right)$$

Where X_i is the ith element of X and y_i is the ith element of y.

If the value of *corr(X,y)* is high, the samples are highly correlated, and if the value of this coefficient is low, they are less correlated. If the value of *corr(X,y)* is 1, it denotes a perfect positive correlation. In case this value is -1, it denotes a perfect negative correlation. The value 0 indicates that X and y are not related. To place the features of a given data in order of their importance using this test, we take one feature at a time and find its correlation with y. The coefficients so obtained are placed in a list. The values of this list indicate the importance of features. The following code implements the method. The implementation includes the following methods:

- `load_data`
- `pearson_cor`

The `load_data` function returns the data and the target of the Fisher Iris dataset:

```
Def load_data():
Data=load_iris()
data=Data.data
target=Data.target
return (data, target)
```

The Pearson's coefficient can then be calculated for each feature, using the formula stated above:

```
def pearson_cor(X,y):
    corr=[]
    for i in range(X.shape[1]):
        x=X[:,i]
        x_mean=np.mean(x)
        y_mean=np.mean(y)
        x1=x-x_mean
        y1=y-y_mean
        prod=x1*y1
        num=np.sum(prod)
        den=np.sqrt((np.sum(x1*x1))*(np.sum(y1*y1)))
        c=num/den
        corr.append(c)
    return corr
```

Finally, the above functions can be invoked to get the Person's coefficient of each feature and place them in order:

```
X,y=load_data()
corr=pearson_cor(X,y)
feat=np.argsort(np.abs(corr))
print(feat)
```

The above code results in a list consisting of indices placed in the order of importance of features. In the case of Fisher Iris data, the list is [2, 0, 1, 3], which means that the third feature is the most important, followed by the first and then the second. The fourth feature is the least important. If one aims to select the top two features, he can select the third and the first feature. In case the top three features are required, the second feature can also be selected.

The next section discusses feature selection using the variance threshold.

Variance threshold

If a feature has the same value in all the samples, it is not important for classification or regression. Therefore, such a feature can be excluded from the data. Extending this argument further, we can say that the feature which has low variance may not be very important. It may be stated that for Bernoulli random variables, the variance is given by:

$$Var = p \times (1 - p)$$

Where p is the probability. For the Boolean features, the above formula can be used as a threshold. For example, in a dataset with Boolean features, if it is intended to remove which are same for more 90% of the samples the threshold `.9 * (1 - .9)` can be used as an argument of the `VarianceThreshold` method and the final data, which does not contain the relevant features can be generated using the `fit_transform` method of SKLearn. The method can be implemented using the following code:

```
#Selecting features using variance threshold
import numpy as np
from sklearn.feature_selection import VarianceThreshold
X, y=load_data()
sel = VarianceThreshold(threshold=(.9* (1 - .9)))
sel.fit_transform(X)
print(sel)
```

Feature selection can also be accomplished using some of the methods which require the know-how of some of the learning algorithms discussed in the following chapters and hence have not been included here.

Conclusion

Like the Simpsons revolve around the Simpson family, but the absence of other characters will have a catastrophic effect on the series. You can understand Liza only if the rest of the characters show very little respect for nature. Homer is effective only because of Ned Flanders, and Marge shines as Homer does not. In machine learning, the learning algorithms are the protagonists. However, they will lose their cut if the procedures like data cleaning are absent.

This chapter is the first step towards machine learning and data science. The techniques discussed in this chapter not only make your model effective and efficient but will also help you in many other disciplines. The chapter begins with the methods of dealing with the missing values and 'NaN''s. It is followed by a brief discussion on converting continuous data into categorical ones.

The second part of this chapter introduces one of the most important topics of machine learning, which is feature selection. Some important techniques of feature selection, like the Chi-Squared test, variance-based method, and Pearson correlation, have been discussed in the chapter.

The methods introduced in this chapter would help you to clean the data and find missing values. It will not only improve the performance of the model but also make the results more meaningful. The reader is expected to analyze the methods and by varying the parameters.

The next chapter introduces the regression for finding the values of unknown samples. The reader should clean the data and deal with the missing values before applying the algorithms introduced in the next chapter.

Exercises

Multiple Choice Questions

1. Missing values may occur due to?
   a. Incomplete filling of forms by the users
   b. If the database is migrated from some other, some data may have been lost
   c. Errors due to programs or due to other technical reasons
   d. All of the above

2. To deal with a missing value, one must?
   a. Find information about the feature
   b. Find the type of feature
   c. Find if the missing value can be replaced with an obvious value
   d. All of the above

3. Some of the most common ways to deal with the missing values are as follows:
   a. Ignoring the records having missing values
   b. Replacing the missing values with average/median
   c. Replacing the missing values with those obtained from regression
   d. All of the above

4. Which of the following is true?
   a. Not all the features so obtained are equally important
   b. The redundant features do not enhance the performance of a model
   c. The noisy features may degrade the performance of a model
   d. All of the above

5. Having a larger number of features results in?
   a. Computationally inefficient learning
   b. Redundant features
   c. Noisy features
   d. All of the above
6. Feature selection aims at?
   a. Better performance
   b. Reduced learning time
   c. Both
   d. None of the above
7. Several features can be reduced by?
   a. Feature selection
   b. Feature elimination
   c. Both
   d. None of the above
8. Which of the following are the types of feature selection?
   a. Filter methods
   b. Wrapper methods
   c. Both
   d. None of the above
9. Which of the following are the types of feature selection?
   a. Univariate methods
   b. Multivariate methods
   c. Both
   d. None of the above
10. Some of the prominent methods that can be used for feature selection are?
    a. Chi-Squared
    b. Variance based
    c. Correlation-based
    d. All of the above

Programming/Numerical

Create a file called `Student.csv`, having the following data:

S_ID	F_name	L_name	No_Sub	Fees
H001	Pratham	Reehal	1	1500
H002	Tanishq	Chitkara		2750
H003	_	Arora	2	2750
H004	Krishna	Sharma	2	
H005	Tarush	_	1	3000
H006	Eliel	Joseph	1	3000
H007	Ram	Mohan	2	6000

The description of the fields is as follows:

- `S_ID`: Student ID
- `F_Name`: First name
- `L_Name`: Last name
- `No_Sub`: Number of subjects
- `Fees`: Total fees

The data type of each field is as follows:

- `S_ID`: String
- `F_Name`: String
- `L_Name`: String
- `No_Sub`: Integer
- `Fees`: Float

Now, perform the following tasks:

1. Read the CSV file using Pandas
2. Replace the 'NaN' type values with 'NaN'
3. Replace the unknown value in the `Fees` field with the mean
4. Replace the unknown value in the `No_Sub` field with the median

Another field called distinction is added to the table. The updated data is as follows:

S_ID	Distinction
H001	Y
H002	N
H003	Y
H004	NA
H005	N
H006	Y
H007	N

1. Find whether `No_Sub` and `Distinction` are correlated?
2. Apply the Chi-Squared test to find whether No_Sub and Distinction are correlated.
3. If Distinction is the label, find which feature is more important No_Sub or Fees. Accomplish this task using:
   a. Variance
   b. Correlation

4. Use the Breast Cancer dataset (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html) and place the features in terms of their importance using:
   a. Variance
   b. Correlation
5. Do you observe anything peculiar while applying the first method?
6. Perform the above task using the Fisher Iris dataset.

Theory

1. Explain the importance of feature selection.
2. What are the types of feature selection?
3. Differentiate between filter and wrapper methods.
4. What is Univariate feature selection? Give an example.
5. Explain the Chi-Squared test for feature selection.
6. How can variance be used for selecting features?
7. Explain the application of Pearson Correlation for feature selection.
8. How do you deal with missing values in the given data? Explain.
9. What are the problems in replacing the missing values with mean?
10. What are the problems in replacing the missing values with zeros?

Chapter 3
Regression

Introduction

At times we need to apply our skills to solve the problems of the people of the nation. The nation, which is different from what is being shown, where untimely rains cause farmer distress, where people work for their lifetime to buy a house, where weather determines whether a family will have the evening meals or not. Regression partially helps us to predict the above and accomplish the task of making people's lives better.

Regression is a supervised learning technique where the values of the dependent variable are real. This chapter introduces gradient descent, which will not only help in implementing regression but also in the classification algorithms discussed in the following chapters. The algorithm assumes that the dependent variables depend linearly on the independent variables, which may not always the case. The regression technique based on the values of the nearest neighbors will overcome this limitation. This chapter also presents the results of the application of the above algorithms on different datasets, hence uncovering the applicability of an algorithm on diverse datasets and hence its robustness.

Structure

The main topics covered in this chapter are as follows:

- Line of best fit
- Gradient descent method
- Implementation
- Linear regression using SKLearn
- Finding weights without iteration
- Regression using K-nearest neighbors

Objective

After reading this chapter, the reader will be able to:

- Define regression
- Find the line of best fit
- Understand gradient descent
- Implement Regression using SKLearn
- Use K-nearest neighbors for regression

The line of best fit

A survey was conducted in a start-up, in which the respondents were asked to state their income and the rent paid by them. The data collected has been shown in *Table 3.1*. Here, the income is the independent variable, and the rent is the dependent variable:

Income	Rent
23000	9500
14000	5000
24000	10000
52500	18000
43750	16000
18000	6000
15000	5000
16000	6000
41500	18000
45000	17500

Table 3.1: *Income and rent paid by ten employees*

Now, given the value of the independent variable, the value of the dependent variable is to be predicted, based on the given data. Let us start by plotting the data (Figure 3.1). Note that if the data is plotted, a positive correlation between the dependent and the independent variables can be observed:

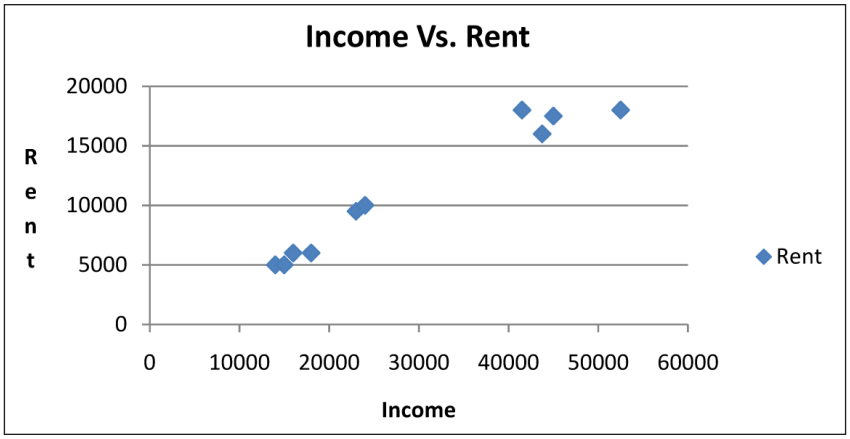


Figure 3.1: *Graph Income versus Data*

If the line of best fit is found, the value of the dependent variable can be predicted, given the value of the independent variable. If s_x is the standard deviation of X, s_y is the standard deviation of Y, r is the coefficient of correlation between X and Y, $\bar{x}$ is the mean of X and $\bar{y}$ is the mean of Y, the line of best fit for two data sets X and Y are given by the following equation:

$$y = mx + c$$

Where:

$$m = (r \times s_y)/s_x$$

And:

$$c = \bar{y} - m\bar{x}$$

Figure 3.2 shows the predicted values of rent and actual values. The slope of this line comes out to be *0.377552163*, and the y-intercept comes out to be *47.16042351*. The equation of the line, therefore, becomes:

$$y = 0.377552163\,x + 47.16042351$$

This line can be used to predict the values of rent, given the income:

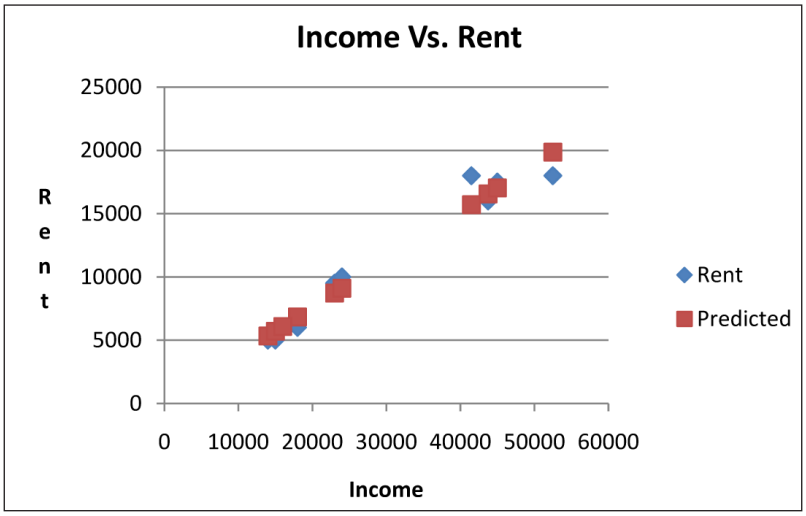


Figure 3.2: *The red squares depict the predicted values of y, and the blue rhombus depict the actual values of the rent*

So, if the income of a person is 15000, then the value of rent comes out to be 5710.442871.

In the above case, the number of independent variables was one. If there is more than one independent variable, then the regression is termed as multiple regression:

$$y = w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + \ldots + w_n x_n$$

Here:

x_1, x_2, x_3, …, x_n are the independent variables. y is the dependent variable, w_0, w_1, w_2, …, w_n are the weights, to be found using the given data, where weights correspond to the coefficient to of the features, in the line (or hyperplane) of best fit. To find the weights, the least square method can be applied. That is, from amongst many possible straight lines (or hyper-planes), we select the line having the least mean square distance from the given points.

"The goal of linear regression procedures is to fit a line or a hyper-plane through the points. Specifically, the program will compute a line so that the squared deviations of the observed points from that line are minimized.[1]"

Gradient descent method

Given X, a matrix having m columns and n rows, and y, a column matrix, having n rows. Each row in X, represent a sample, x_i. The values in x_i are the values of the m

features which determine the value of the dependent variable y_i. Here, we assume that the linear combination of x_i's determine the value y_i.

Let w_i be the weight associated with a feature x_i and b be the y-intercept. The expected value of the dependent variable would be $\hat{y}$, considering $\hat{y}$ to be the linear combination of the given features. That is:

$$\hat{y} = \sum_{i=1}^{m} x_i w_i + b$$

The above equation can be re-written by considering, $x_0 = 1$ and b = w_0 as:

$$\hat{y} = \sum_{i=0}^{m} x_i w_i$$

The difference between the predicted value $\hat{y}$ and the given value y should be minimum, and so should be its square and half of the resultant. The objective function therefore becomes:

$$f = \frac{1}{2} \times (\hat{y} - y)^2$$

To get the optimal weights, the gradient of f with respect to the weights must be found. The partial derivative of f with respect to the weight of the *ith* sample can be written as:

$$\frac{\delta f}{\delta w_i} = \frac{\delta f}{\delta \hat{y}} \times \frac{\delta \hat{y}}{\delta w_i}$$

It is because f depends on $\hat{y}$ and $\hat{y}$ depends on w_i:

$$\frac{\delta f}{\delta w_i} = (\hat{y} - y) \times \frac{\delta \sum_{i=0}^{m} x_i w_i}{\delta w_i}$$

The final value of the gradient becomes:

$$\frac{\delta f}{\delta w_i} = \left(\hat{y} - y\right) \times x_i$$

The weights must be updated in a direction opposite to the gradient as the gradient tells us the direction for the positive growth of the function. That is:

$$w_i = w_i - \frac{\delta f}{\delta w_i}$$

Or

$$w_i = w_i - \left(\hat{y} - y\right) \times x_i$$

Which is the equation used to update the weights of the *ith* sample?

The process of learning weights is as follows:

1. Initialize weights by small random numbers:
   w = List of random numbers conatining m values, where m is the number of features
2. Till convergence, change weights using the following equation:

$$w_i = w_i - \left(\hat{y} - y\right) \times x_i$$

3. Find the mean square error on the test set using the weights obtained in the above step.

Having seen the algorithm of regression, let us now implement the above algorithm.

Implementation

To understand the idea of linear regression, consider the following data (*Table 3.2*). The data has five features: **X1**, **X2**, **X3**, **X4**, and **X5**. The number of samples in the data is 21. Note that the last column depicts y, depends on the values in the first four columns. You can save the following data as a CSV file called `DataRegression.csv`:

X1	X2	X3	X4	X5	y
13	82	37	98	71	137.2167
87	89	87	68	98	138.9469
75	78	81	53	36	101.2451
87	74	26	51	47	100.9123
73	78	17	32	80	104.0818
43	97	87	63	33	118.059
99	59	24	71	71	109.5203
31	17	91	41	48	53.05253
33	37	18	84	61	94.39396
22	55	85	88	79	115.9572
36	35	28	91	93	107.8502
48	83	45	24	98	108.7555
92	51	75	25	35	66.94636
20	28	40	15	72	54.09701

Contd…

15	37	58	43	75	76.71684
10	43	31	91	61	101.6789
48	38	32	81	69	97.00749
24	12	69	61	73	68.1979
26	38	54	47	36	66.75108
96	91	56	16	48	95.35608
62	63	94	14	45	72.02378

Table 3.2: *Data for DataRegression.csv*

This section will discuss the procedure to develop a model that learns *w*, the weights and *w0*, the bias to fit the data such that:

$$y = w_0 + w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + w_5x_5$$

1. Read the CSV file and extract the data in *X* and *y*. This task can be accomplished using the CSV module and reading the file using the reader method. The extract `Data()` method fetches the data into the variables *X* and *y*. The code to read the file is as follows:

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import genfromtxt
import csv
with open('F:\Machine Learning\Regression\DataRegression.csv','r') as
f:
    data = csv.reader(f)
```

2. Initialize the weights by random numbers between 0 and 1. The `init_weights` function accomplishes this task:

```
def init_weights():
    w=np.random.random(X.shape[1])
    return w
```

3. Normalize the data by using the following formula:

$$x = \frac{x - m}{s}$$

Here, *s* is the standard deviation of the feature, and *m* is its mean. The normalize function takes `Data` as the input and returns the `Data`, the mean of each column,

and the standard deviation of each column. The last two can be used for converting the predicted value back to the same scale and to predict the error. The data can be normalized by subtracting mean from each value and dividing the difference by the standard deviation:

```
def normalize(Data):
    mean_arr=[]
    std_arr=[]
    for i in range(Data.shape[1]):
        col=Data[:,i]
        s=np.std(col)
        m=np.mean(col)
        mean_arr.append(m)
        std_arr.append(s)
        #print('col ',i,' mean ',m,' std ',s)
        for j in range(Data.shape[0]):
            Data[j,i]=(Data[j,i]-m)/s
            #print(Data)
    return Data, mean_arr, std_arr
```

4. The gradient descent method can be used to train the model and learn the weights. Note that there is no need to learn the bias separately as it is treated as one of the weights. The train function accomplishes the above task. It takes `X`, `y` and `w` as input arguments and returns the weight w. Note that the value of the learning rate in the following is set as 0.01:

```
def train(X, y, w):
    mse=0
    for i in range(X.shape[0]):
        x=X[i,:]
        sum1=np.matmul(np.transpose(w),x)
        diff=(sum1-y[i])
        mse=(diff**2)
        mse=np.sqrt(mse)
        if(mse>0.01):
```

```
            w=w-0.1*(diff)*x
            #print(w)
    return w
```

5. The mean squared error mse can be calculated using the following function, which takes `X`, `y` and `w` as the input arguments and returns the `mse`:

```
def calMSE(X, y, w):
mse=0
for i in range(len(y)):
    x=X[i,:]
    sum1=np.matmul(np.transpose(w),x)
    diff=(sum1-y[i])
    mse+=(diff**2)
    #print(mse)
    mse=np.sqrt(mse)
    return(mse)
```

6. The results obtained from the model, can be converted into the original scale, using the `verify` function, which takes the predicted value, `w`, the mean of features and their standard deviation as input arguments and returns the predicted values by using the formula *value=(value*s)+m*:

```
def verify(pred, mean_arr_y, std_arr_y):
    for j in range(pred.shape[0]):
        pred[j]=(pred[j]*std_arr[-1])+mean_arr[-1]
        #print(pred)
    return pred
```

7. The final program for regression is as follows:

```
Data = genfromtxt('F:\Machine Learning\Regression\DataRegression.
csv', delimiter=',')
Data,mean_arr, std_arr=normalize(Data)
X, y=extractData(Data)
w=init_weights()
w=train(X, y, w)
```

```
mse=calMSE(X, y, w)
pred=np.matmul(X,w)
predicted=verify(pred, w,mean_arr, std_arr)
```

The reader is expected to implement the above technique and observe and analyze the results. The reader can also use other datasets and compare the performance of the algorithm on various datasets.

Linear regression using SKLearn

The module `sklearn.linear_model.LinearRegression` implements linear regression. The following discussion uses the model. The important parameters of the constructor of the method are as follows (*Table 3.3*):

Parameter	Explanation
`fit_intercept`	This parameter is used to find the intercept. If the value of this parameter is set as `False`, the data is deemed to be centered. The default value of this parameter is `True`.
`normalize`	The data is normalized by subtracting mean and dividing the data by L2 norm. The default value of this parameter is `False`. Also, note that this parameter is not important of the `fit_intercept` parameter is set to `False`.

Table 3.3: *The parameters of LinearRegression*

Having seen the parameters of the function, let us now move to the attributes. The attributes of the `sklearn.linear_model.LinearRegression` are as follows (*Table 3.4*):

Attributes	Explanation
`coef_`	It returns the weights of the features. If there is a single target, a 1-D array is returned, whereas, in the case of multiple targets, the 2D array is returned. Here, the shape of the array would be (`n_targets`, `n_features`).
`intercept_`	The independent term or the bias in the linear model is returned by this.

Table 3.4: *The attributes of LinearRegression*

The module provides us with some functions. The fit and predict are the two most important functions. The fit function models the data X with y. The predict function predicts the output of the argument. Let us now use the above methods on some common datasets.

Experiments

This section presents two experiments in which linear regression has been applied to the Boston Housing dataset. In the first experiment, K-Fold validation has been used with K=10, and in the second experiment, the train test split has been used with test-size=0.33. The reader is expected to vary the value of K and `test_size` to see the effect of variation of these parameters on the mean squared errors. (make a plot of variation in test size and accuracy).

Experiment 1: Boston Housing Dataset, Linear Regression, 10-Fold Validation

1. Import requisite modules:

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
```

2. Load the data using the `load_boston()` function:

```
boston=load_boston()
X=boston.data
y=boston.target
```

3. Use the following code to split the data into train-test, using the K-Fold validation. Find the mean squared error of each test case and find the average of the mse's so obtained:

```
kf=KFold(n_splits=5)
kf.get_n_splits(X)
mse_arr=[]
for train_i,test_i in kf.split(X):
   X_train,X_test=X[train_i],X[test_i]
   y_train,y_test=y[train_i],y[test_i]
   modal= LinearRegression().fit(X_train, y_train)
   mse=0
```

```
    for l in range(len(y_test)):
        X_test_data=X_test[l,:]
        y1=modal.predict([X_test_data])
        d=y1-y_test[l]
        mse=mse+(d**2)
mse=mse//(y_test.shape[0])
mse=np.sqrt(mse)
mse_arr.append(mse)
print(np.mean(mse_arr))
```

Experiment 2: Boston Housing Dataset, Linear Regression, train-test split

1. Import requisite modules:

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
```

2. Load the data using the `load_boston()` function:

```
boston=load_boston()
X=boston.data
y=boston.target
```

3. Use the following code to split the data into the train-test, using the K-fold validation. Find the mean squared error of each test case and find the average of the mse's so obtained:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_
size=0.33, random_state=42)
modal= LinearRegression().fit(X_train, y_train)
mse=0
for l in range(len(y_test)):
    X_test_data=X_test[l,:]
    y1=modal.predict([X_test_data])
    d=y1-y_test[l]
    mse=mse+(d**2)
```

```
mse=mse//(y_test.shape[0])
mse=np.sqrt(mse)
print(np.mean(mse))
```

The above experiment can be repeated by not providing the random state, repeating the experiment 10 times. Moreover, the nested validation can also be done to make the final result more reliable.

Finding weights without iteration

Let X and Y are two matrices of order $n \times m$ and $n \times 1$, then the dimensions of $Y^T X$ is $m \times n$.

X be the matrix representing the Data, where each row of X represents a sample, and each column represents a feature. The vector y gives the value of the real label. That is:

$$X = \begin{bmatrix} x_{11} & \cdots & x_{1m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \cdots & x_{nm} \end{bmatrix} \text{ and } Y = \begin{matrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{matrix}$$

If W represents the weight matrix, of order $1 \times m$, W' represents the transpose of W. Then $(X \times W' - Y)$ should be minimized. It implies that $\frac{1}{2} \times (X \times W' - Y)^2$ is also minimized.

That is:

$$J(W) = \frac{1}{2} \times (X \times W' - Y)^2$$

Which is same as:

$$J(W) = \frac{1}{2} \times (X \times W' - Y)^T \times (X \times W' - Y)$$

To minimize the value of J, the derivative of J, with respect to W, should be equated to 0, to get the value of W, that is:

$$\frac{\delta J(W)}{\delta W} = 0$$

The derivative after simplification becomes:

$$X^T XW - X^T Y = 0$$

Which gives:

$$W = (X^T X)^{-1} X^T Y$$

The following code implements the above algorithm:

```
from sklearn.datasets import load_iris

import numpy as np

from numpy import linalg

from sklearn import model_selection

Data=load_iris()

X=Data.data[:100,:]

y=Data.target[:100]

X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
test_size=0.20, random_state=42)

W=np.matmul(np.matmul(linalg.inv(np.matmul(np.transpose(X_train),X_
train)),np.transpose(X_train)),y_train)
```

Let us now move to another technique of regression called **K-nearest neighbors**.

Regression using K-nearest neighbors

Suppose the salary of a person is to be guessed just by looking at the salaries of his five closest friends. What will you do? Probably take the average of the salaries of the five friends. It may appear very naïve, but it works most of the time.

In the above example, replace the friends with the train set, the salary with features, and the person whose salary is to be guessed with the test set. The salary, to be guessed, becomes the predicted value of the dependent variable. It is essential, K-nearest neighbors, for regression. The precise algorithm for this regression is as follows:

1. Divide the data into the train and the test set
2. For each test set:
    a. Find its distance from all the samples in the train set
    b. Arrange the distances in ascending order

c. Take the indices of the first K samples in 2. b
d. Find the average of the dependent variable of the test set for the above indices and declare the result

To ascertain the performance, you can find the root mean square error.

The implementation of the above algorithm is as follows:

```
Data=load_boston()

X=Data.data

y=Data.target

X_train,X_test,y_train,y_test=model_selection.train_test_split(X,y,test_
size=0.3)

y_pred=[]

for i in range(X_test.shape[0]):

    x=X_test[i]

    dist=[]

    for j in range(X_train.shape[0]):

        d=0

        for k in range(X_train.shape[1]):

            d+=((X_train[j,k]-X_test[i,k])**2)

        d=np.sqrt(d)

        dist.append(d)

    index=np.argsort(dist)

    mean=0

    for t in range(5):

        mean+=(y_train[index[t]])

    mean=mean/5

    y_pred.append(mean)
```

It may be stated that many classification algorithms described in the following chapters can be used for regression also.

Conclusion

This chapter described a supervised learning approach called regression. The statistical method of fitting a line has been discussed in the first section. The gradient descent method is generally used to learn the weights by minimizing the least squared error. The method has been discussed in detail, and regression using this method has been implemented both by using Scikit learn and without using it. The learning of weights without iteration has also been discussed in this chapter. Regression can also be done using other methods like K-nearest neighbors, SVM, and neural networks. Some of these methods would be discussed in the following chapters.

Regression finds its applications in diverse fields like weather forecasting, stock market prediction, and so on. The technique is exciting, and there is a scope of research in this field. The exercises given at the end of the chapter would prompt you to explore the algorithms studied on various datasets. You will continue learning regression both in ML and outside as and when you grow, professionally and personally.

The next chapter introduces classification. The chapter discusses some simple algorithms like K-nearest neighbors, logistic regression, and so on. The concept of Linear Discriminant Analysis has also been explained in the chapter.

Exercises

Multiple Choice Questions

1. Regression is?

a. Supervised learning
b. Unsupervised learning
c. Both
d. None of the above

2. Gradient descent changes weights?

a. In the direction of the gradient
b. In the opposite direction of the gradient
c. Both
d. None of the above

3. Which of the following can be used for regression?

a. Single-layer perceptron
b. Multi-layer perceptron
c. Decision trees
d. All of the above

4. Which of the following can be used for regression?
   a. KNN
   b. Support Vector Machines
   c. Decision trees
   d. All of the above
5. Which of the following can be used for regression?
   a. K means
   b. Support Vector Machines
   c. PCA
   d. All of the above
6. What is multiplied with the gradient in the gradient Descent to change the weights?
   a. Learning rate
   b. Accuracy
   c. Specificity
   d. None of the above
7. Which of the following can be used to ascertain the performance of a regression model?
   a. Mean Squared Error
   b. Accuracy
   c. Specificity
   d. None of the above
8. In which of the following regression can be used?
   a. Weather prediction
   b. Stock market prediction
   c. To predict the growth of a sector
   d. All of the above
9. Which of the following can be used to create a model of regression?
   a. Gradient descent
   b. Newton's method
   c. Both
   d. None of the above
10. If the mse of your model is 0, then it is the case?
    a. Underfitting
    b. Overfitting
    c. Insufficient information
    d. None of the above

Theory

1. Define regression. Explain any three applications of regression.
2. Differentiate between regression and classification.
3. Derive the formula for gradient descent.
4. Explain the statistical line fitting.
5. What is multiple regression?
6. How do you carry out regression using K-nearest neighbors?

7. Explain how you can accomplish the task of finding unknown values using the concept of Neural networks.
8. Is feature selection important for regression?

Experiments

1. Consider the 3D Road Network (North Jutland, Denmark) data set in the UCI data repository. The data is a text data, having 434874 instances and four attributes. Note that there are no missing values on the dataset. The data contains the following features:
   - OSM_ID: OpenStreetMap ID for each road segment or edge in the graph
   - Longitude
   - Latitude
   - Altitude

   As per the UCI machine learning repository.
   - Check if the altitude can be found by using the longitude and latitude.
   - Accomplish the above task using the gradient descent method and find the mean squared error.
   - Perform the above task using the KNN regression.
2. Give reasons to justify why regression should not be applied to the above problem?

   Repeat the above (Q1 and Q2) experiments on the Airfoil self-noise data set and compute the mean squared error.
3. The Alcohol QCM Sensor Dataset Data Set in the UCI repository has eight features. The data in the above question had six features and that in question 1 had four features. Apply the linear regression of SKLearn to all of them and see if having more number of features improves the results.
4. Now have a look at the Appliances energy prediction data set, having 29 features, and see if the feature selection has an impact on the results. You can select the relevant features by any of the feature selection methods explained in Chapter 2.
5. Based on the above experiments, write a note on whether having the optimal number of features enhances the performance of the regression models.

Chapter 4
Classification

Introduction

Hari decided not to see the face of a particular person. So, he sought to automate the process of distinguishing the pictures having the face of the person from those which do not contain the face. To accomplish this task, he decided to develop a pipeline. He used various classification algorithms and compared the performance of the algorithms using accuracy, specificity, and sensitivity. Finally, he selected a feature extraction method, a feature section algorithm, and a classification algorithm. He was able to accomplish the task. This chapter presents some of the most common classification algorithms and will help you if you are stuck in the same situation.

This chapter introduces classification. It assigns one of the designated labels to a test sample and comes under supervised learning. The techniques like K-Nearest Neighbor, Logistic Regression, and Naïve Bayes have been discussed and implemented in the chapter. The first is based on the determination of the majority behavior, in the vicinity, to find the behavior of the unknown sample. The second and third use the concepts of probability.

The chapter presents some basic experiments and expects the reader to understand the importance of empirical analysis in machine learning. This chapter will form the basis of complex ML-based projects like face recognition, and so on.

Structure

The main topics covered in this chapter are as follows:

- K- Nearest Neighbors
- Implementation of KNN
- Use of SKLearn to implement KNN
- Logistic Regression
- Implementation of Logistic Regression using SKLearn
- Naïve Bayes
- Implementation of Gaussian Naïve Bayes using SKLearn

Objective

After reading this chapter, the reader will be able to:

- Understand K-Nearest Neighbor algorithm
- Implement KNN
- Use SKLearn to implement KNN
- Understand the Logistic Regression algorithm
- Use SKLearn to implement Logistic Regression
- Understand the Naïve Bayes algorithm
- Use SKLearn to implement Gaussian Naïve Bayes

Basics

Classification is the process of assigning one of the designated classes to a given sample. It is preceded by feature extraction and feature selection. The feature extraction part of the system extracts the relevant features, which help us to distinguish a sample of a particular class from that of another class. This part is important and generally determines the performance of a system. In the case of an image or a video, since the number of features is colossal, the selection of relevant features using a good feature selection method is required. The feature extraction algorithms are discussed in the following chapters.

The performance of a classifier can be measured in terms of accuracy, specificity, and sensitivity. In a problem having just two classes, say **Class 0** and **Class 1**, one of the following cases may occur (*Table 4.1*):

The decision of the classifier	Actual class	Decision designated as
Class 0	Class 0	True Negative (TN)
Class 0	Class 1	False Negative (FN)
Class 1	Class 1	True Positive (TP)
Class 1	Class 0	False Positive (FP)

Table 4.1: *Decisions of a classifier, in case of a two-class problem*

The number of samples correctly classified by the classifier is termed as accuracy. The number of positive samples correctly classified is sensitivity, and the number of negative samples correctly classified is specificity. The formulae for accuracy, specificity, and sensitivity are as follows. The F-measure is the harmonic mean of sensitivity and specificity:

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

$$Specificity = \frac{TN}{TN + FP}$$

$$Senstivity = \frac{\left(TP\right)}{TP + FN}$$

$$F - measure = \frac{2 \times Specificity \times Senstivity}{Specificity + Senstivity}$$

It may be stated that there are many more ways of determining the performance of a model. However, this chapter uses the above performance measures.

Generally, the accuracy of a model is stated to show the performance of a model. However, the only accuracy does not always give the correct picture. For example, if you develop a system for segregating spam mails and your model gives 90% accuracy, it may not be a very good model. It is because more than 90% of the total emails are spam, and if your model classifies every sample as spam, even then, the accuracy can be 90%.

One may note that, most likely, there cannot be a model, which gives 100% performance. It is, therefore, desirable to find the probability of a sample belonging to a given class. This chapter introduces algorithms that help us to develop such models. Primarily, this chapter discusses K-nearest neighbors, Naïve Bayes, and logistic regression.

Classification using K-nearest neighbors

A man is known by the company he keeps. Let us use this proverb to find the class of an unknown sample by finding the classes of its neighbors. The majority class can be deemed as the class of the sample. For example, if a sample is to be marked as left or right, the leaning of its *k* neighbors can be determined. If the majority of its neighbors are left-leaning, the sample would be deemed as left-leaning, else it would be deemed as right-leaning.

The above algorithm is referred to as the K-nearest neighbors. It can be easily implemented. The distance of a given sample can be found from all the data in the train set. This distance is then arranged in increasing order. The majority-label of the first K elements of this ordered distance array is then returned as the output label. The formal algorithm for this method is as follows.

Algorithm

1. Divide the data into the train and the test set.
2. For each test set:
   a. Find its distance from all the samples in the train set.
   b. Arrange the distances in ascending order.
   c. Take the indices of the first K samples in 2.b
   d. Find the class of the neighbors identified in 2.c.
   e. The majority class in d. would be the class of the unknown sample.

To find the distances between the samples, any of the following formulae can be used. In the following formulae, the summation represents the addition of the values in the features. The three most common distances are as follows (*Table 4.2*):

Name of the distance	Formula
Euclidean Distance	$\sqrt{\sum_{i=1}^{m}(x_i^t - x_i^t)^2}$
Manhattan Distance	$\sqrt{\sum_{i=1}^{m}\left\|(x_i^t - x_i^t)\right\|}$
Minkowski Distance	$\sqrt[p]{\sum_{i=1}^{m}(\left\|x_i^t - x_i^t\right\|)^p}$

Table 4.2: *Distance used in KNN*

To ascertain the performance, you can find the accuracy, specificity, and sensitivity. The following section implements the above algorithm.

Implementation of K-nearest neighbors

The KNN algorithms explained in the above section can be easily implemented using NumPy. The implementation that follows constitutes four steps and uses the Euclidean Distance. The Fisher Iris dataset has been used in the following code:

1. To implement the algorithm, you need to import the following modules:

```
from sklearn.datasets import load_iris

import numpy as np

from sklearn import model_selection
```

2. This implementation divides the given data into train data and test data. This is done by creating the `load_data()` method:

```
def load_data():

    Data=load_iris()

    X=Data.data[:100,:]

    y=Data.target[:100]

    X_train, X_test, y_train, y_test = model_selection.train_test_
    split(X, y, test_size=0.30, random_state=42)

    return(X_train, X_test, y_train, y_test)
```

3. The performance measures can be calculated by finding the number of True Positive, True Negative, False Positive and False Negative. This is done by crafting the following function:

```
def cal_acc(y_pred, y_test):

    TP=0

    TN=0

    FP=0

    FN=0

    for i in range(len(y_test)):
```

```
        if(y_test[i]==y_pred[i]):
            if(y_pred[i]==1):
                TP+=1
            else:
                TN+=1
        else:
            if(y_pred[i]==1):
                FP+=1
            else:
                FN+=1
    return(TP, TN, FP, FN)
```

4. The classification is carried out using the following code. The following implementation uses Euclidean Distance:

```
X_train, X_test, y_train, y_test = load_data()
y_pred=[]
for i in range(X_test.shape[0]):
    dist=[]
    for j in range(X_train.shape[0]):
        d=0
        for k in range(X_train.shape[1]):
            d+=(X_test[i,k]-X_train[j,k])**2
        d=np.sqrt(d)
        dist.append(d)
    ord_dist=np.sort(dist)
index=np.argsort(dist)
s=0
```

```
for m in range(5):
    s+=y_train[index[m]]
    if(s>3):
        y_pred.append(1)
    else:
        y_pred.append(0)
TP, TN, FP, FN=cal_acc(y_pred, y_test)
accuracy=(TP+TN)/(TP+TN+FP+FN)
print(accuracy)
```

Let us now have a look at the implementation of KNN using SKLearn.

The KNeighborsClassifier in SKLearn

This classifier implements the K-nearest neighbor algorithm. The important parameters and methods of the classifier are given in *Table 4.3* and *Table 4.4*:

Parameter	Explanation	Details
`n_neighbors`	It depicts the number of neighbors, that is K.	It is an optional parameter. The default value of this parameter is 5.
`weights`	It depicts the functions of generating weights for prediction.	It is an optional parameter. The default value of this parameter is `uniform`. The value of weights can be `distance`, also, where the data points closer to the sample would have more weights.
`algorithm`	It depicts the algorithm used to carry out K-Means	The values of this parameter can be `auto`, `ball_tree`, `kd_tree`, or `brute`. It is an optional parameter
Power parameter	It is used in case of Minkowski metric	It is an optional parameter. Its default value is 2.

Table 4.3: *Parameters of KNeighborsClassifier*

The important methods of KNeighborsClassifier are given in *Table 4.4*. The reader may note that the fit and predict method of practically all the classifiers work in the same manner:

Name of the method	Purpose
`fit(self, X_train, y_train)`	This method takes two parameters: the train data and the train labels.
`kneighbors(self[, X, n_neighbors, …])`	This method finds the n-neighbors of a point.
`kneighbors_graph(self[, X, n_neighbors, mode])`	This method computes the graph of k-Neighbors.
`predict(self, X_test)`	This method predicts the class labels for `X_test`.
`predict_proba(self, X_test)`	This method finds the probability estimates for `X_test`.

Table 4.4: *Important methods of KNeighborsClassifier*

Having seen the important attributes, parameters, and methods of the classifier, let us now move to the experiment part.

Experiments – K-nearest neighbors

The following codes show how to use the above methods to carry out classification. *Code 1* uses the `train_test_split` to split the data into train and test data. *Code 2* uses the KFold to implement the KFold validation with k=10, and *Code 3* repeats the task with K=20. The functions used in the following codes (`load_data`, `cal_acc`) are the same, as shown in the earlier sections.

Code 1: Breast Cancer; Train Test Split

```
X_train, X_test, y_train, y_test=load_data()

Model_Knn = KNeighborsClassifier(n_neighbors=5)

Model_Knn.fit(X_train, y_train)

predicted=Model_Knn.predict(X_test)

TP, TN, FP, FN=cal_acc(predicted,y_test)

accuracy=(TP+TN)/(TP+TN+FP+FN)

print(accuracy)
```

The next experiment uses `train_test_split` to classify the Breast Cancer dataset.

Code 2: Breast Cancer; 10-Fold Cross Validation

```
acc_arr=[]
```

```
Data=datasets.load_breast_cancer()
X=Data.data
y=Data.target
kf = KFold(n_splits=5)
KFold(n_splits=10, random_state=None)
for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    Model_Knn = KNeighborsClassifier(n_neighbors=5)
    Model_Knn.fit(X_train, y_train)
    predicted=Model_Knn.predict(X_test)
    TP, TN, FP, FN=cal_acc(predicted,y_test)
    acc=(TP+TN)/(TP+TN+FP+FN)
    acc_arr.append(acc)
acc_av=np.mean(acc_arr)
print(acc_av)
```

The next experiment uses ten-fold cross-validation to classify the Breast Cancer dataset.

Code 3: Breast Cancer; 10 Fold Cross Validation

```
acc_arr=[]
Data=datasets.load_breast_cancer()
X=Data.data
y=Data.target
kf = KFold(n_splits=10, random_state=None)
Data=datasets.load_breast_cancer()
X=Data.data
y=Data.target
```

```
for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    Model_Knn = KNeighborsClassifier(n_neighbors=5)
    Model_Knn.fit(X_train, y_train)
    predicted=Model_Knn.predict(X_test)
    TP, TN, FP, FN=cal_acc(predicted,y_test)
    acc=(TP+TN)/(TP+TN+FP+FN)
    acc_arr.append(acc)
acc_av=np.mean(acc_arr)
print(acc_av)
```

Having seen the working of K-nearest neighbor, let us now shift our focus to the logistic regression, which will also help us to determine the probability of a sample belonging to a class.

Logistic regression

Logistic regression is a statistical model, which helps in modeling probability. This model can also be extended for classification. This section gives a brief overview of logistic regression. Let us first have a look at the mathematical foundations of this model. In the discussion that follows, X is a matrix of order $n \times m$ and W is a matrix of order $1 \times m$. The former depicts the data having n samples and m features and the later depict the weights assigned to each feature. y denotes the target. In classification problems, the values of y are discrete. Specifically, for a two-class problem, y will be either 1 or 0. In such cases, the value of y can be predicted by finding W^tX and then using a function which maps the obtained values to . The sigmoid function is one such function. The sigmoid function is:

$$f\left(z\right)=\frac{1}{1+e^{-Z}}$$

Note that the maximum value of the function is 1 when the value of Z is ∞, that is:

$$f(z)=\frac{1}{1+e^{-Z}}=1$$

The minimum value of the function is 0, when the value of Z is – ∞, that is:

$$f(z) = \frac{1}{1 + e^{-Z}} = 0$$

This function not only smoothly maps (– ∞, ∞) to [0, 1], but also can be differentiated with ease. The derivative of s sigmoid function can be expressed in terms of itself as follows:

$$f^{'}\left(Z\right) = f\left(Z\right) \times (1 - f\left(Z\right))$$

Here, *f(Z)* depicts the probability of the value of *y* being *1*, given the data. That is:

$$P\left(X; W\right) = f\left(x\right)$$

The probability of the value of *y* being *0*, given the data, is therefore 1 – *f*(*x*). That is:

$$P\left(X; W\right) = 1 - f\left(x\right)$$

The probability of y given *X* and *W* can, therefore, be written as follows:

$$P\left(X; W\right) = f(x)^{y}(1 - f(x))^{1-y}$$

Here we take the liberty of assuming that all the samples have independent features. The product of probabilities will give us the likelihood:

$$P\left(x_i; W\right) = f(x_i)^{y}(1 - f(x_i))^{1-y}$$

$$P\left(X; W\right) = \prod_{i=1}^{m} f(x_i)^{y}(1 - f(x_i))^{1-y}$$

Taking *log*, we get the log-likelihood:

$$\log\log\left(L\right) = \sum_{i=1}^{m} y \times \log\left(f(x)\right) + (1 - y) \times \log(1 - f(x))$$

The derivative of the log-likelihood can be written in terms of itself as follows:

$$\frac{\delta\left(\log\log\left(L\right)\right)}{\delta W_i} = (y - f\left(x\right))x_i$$

In each iteration of training, the change in weights can thus be found as follows:

$$w_i = w_i + \alpha \times (y - f\left(x\right))x_i$$

Having obtained the formula for updating the weights, let us now move to the formal algorithm.

Algorithm:

1. Split the given data into train data and test data.
2. Initialize the weights to random numbers.
3. Repeat the following steps for each training sample.
   a. Update the weights using the following formula:

$$w_i = w_i + \alpha \times (y_i - f(x_i))x_i$$

   Where, y_i is the value of the target variable for the *ith* sample, x_i is the data of the *ith* sample, and α is the learning rate.

4. For each test sample :
   a. Find $u = \sum x_j w_j$
   b. Find $f(u) = \frac{1}{1+e^{-u}}$
   c. The value obtained above gives the probability of a given sample belonging to a class
   d. Based on the value obtained above, decide whether the given sample belongs to the class or not

The reader is expected to implement the above using Numpy. The implementation of logistic regression using SKLearn has been explained in the next section.

Logistic regression using SKLearn

The LogisticRegression classifier implements the above algorithm is SKLearn. The parameters, attributes, and methods of LogisticRegression are as follows. *Table 4.5* shows the parameters of the classifier:

Parameter	Explanation	Details
`penalty`	The norm used in penalization is specified using this parameter. The values of this parameter can be `l1`, `l2`, `elasticnet`, or `none`.	It is an optional parameter. The default value of this parameter is `l2`.

Contd…

`tol`	This parameter specifies the tolerance for stopping criteria.	It is an optional parameter. The default value of this parameter is 1e-4.
`class_weight`	The weights associated with classes are specified using this parameter. The values of this parameter can be `dict` or `balanced`.	It is an optional parameter. The default value of this parameter is `none`.
`solver`	This parameter is used to specify the algorithm.	It is an optional parameter. The default value of this parameter is `liblinear`.
`max_iter`	This parameter represents the number of iterations.	It is an optional parameter. The default value of this parameter is 100.
`multi_class`	It is chosen if there are more than two labels. It can have the following values `ovr`, `auto`, or `multinomial`.	It is an optional parameter. The default value of this parameter is `ovr`.

Table 4.5: *Important parameters of logistic regression*

The attributes of the classifier have been shown in *Table 4.6*:

Attributes	Details
`classes_`	This attribute gives the class labels of each sample.
`coef_`	This attribute gives the coefficients of the decision function.
`intercept_`	This attribute gives the intercept of the decision function.

Table 4.6: *Important attributes of logistic regression*

The important methods of the LogisticRegression classifier are as follows (*Table 4.7*):

Name of the method	Purpose
`fit(self, X_train, y_train)`	This method takes two parameters: the train data and the train labels.
`predict(self, X_test)`	This method predicts the class labels for `X_test`.
`predict_proba(self, X_test)`	This method finds the probability estimates for `X_test`.

Table 4.7: *Important methods of Logistic Regression*

Having seen the important attributes, parameters, and methods of the classifier, let us now move to the experiment part.

Experiments – Logistic regression

The following codes show how to use the above methods to carry out classification. It may be noted that the loading of data, finding accuracy, and so on, can be done in the same manner as shown in the previous implementations. To use the classifier, one needs to import the following modules:

```
from sklearn.datasets import load_iris
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
import numpy as np
```

The rest of the code is the same as that explained earlier. However, the classifier can be constructed using the `LogisticRegression()` method, shown as follows:

```
clf=LogisticRegression()
clf.fit(X_train,y_train)
y_pred=clf.predict(X_test)
```

The reader is advised to use these functions on the Breast Cancer dataset and the Iris dataset and find the accuracy using:

- train_test_split
- KFold

And compare the results. Another example of a probability-based model has been explained in the following section.

Naïve Bayes classifier

The ethos of the data is generally not known, and therefore modeling is required, which is a random process. It may also be stated that we do not know all the variables affecting the outcome. Therefore, even if the process is deterministic, it will be difficult to accomplish the above task. However, the model can be crafted by using the probability of observable variable *P(x)*, which can be found easily. Using the above model, classification can be carried out. It can be done by finding the probability of a given sample belonging to a particular class. Bayes Theorem, which can be stated as follows, helps us to accomplish this task:

$$P(B_i / A) = (P(A / B_i) \times P(B_i)) / \sum_{i=1}^{C} (P(A / B_i) \times P(B_i))$$

Where, $P(B_i/A)$ is the conditional probability of B_i, provided A is given. Likewise, $P(A/B_i)$ is the conditional probability A provided B_i is given. Also, if $P(B_i/A)$ and $P(B_i)$ are known, the above theorem can be applied.

In the case of classification if there are two classes we can find out the probability of sample belonging to a particular class, say for that matter, C_1 and C_2.

Since there are only two classes:

$$P(C_1) + P\left(C_2\right) = 1$$

Where, $P(C_1)$ is the prior probability, and $P(x/C)$ is called likelihood, and $P(x)$ is the evidence, which can be found by using the law of total probability:

$$P\left(x\right) = P(x/C_1) \times P(C_1) + P(x/C_2) \times P(C_2)$$

The posterior probability is defined as the product of prior probability and the likelihood divided by the evidence. That is:

$$P(C_i/x) = (P(x/C_i) \times P(C_i)) / P(x)$$

The value of $P(x)$ can be found using the above equation.

The final decision can be made by choosing the class C_i for which $P(C_i/x)$ is maximum.

The decision can also be taken by minimizing the risk as there is always a loss incurred for taking the decision. We can define the risk for the action A_i as follows:

$$R(a_i/x) = \sum \in_{ik} P(C_k/x)$$

Where k varies from 1 to the number of classes, and the value of k is not the same as the correct class.

The discriminant function can, hence, be defined as the negative of the above risk. It is because the minimization of the risk would lead to the maximization of the discriminant function:

$$f\left(x\right) = -R(a_i/x)$$

The reader is expected to implement the above using Numpy. The in-built classifier has been explained in the next section.

The GaussianNB Classifier of SKLearn

SKLearn comes with various Naïve Bayes implementations. These include Gaussian Naïve Bayes, Bernoulli Naive Bayes, and Multinomial Naïve Bayes. The Gaussian

Naive Bayes is popular and simple. It assumes the Gaussian distribution of the data. The important parameters and methods of the classifier are given in *Table 4.8* and *Table 4.9*:

Parameter	Explanation	Details
`priors`	The prior probabilities of the classes can be found using this parameter.	It is an optional parameter.
`var_smoothing`	The portion of the largest variance of all features can be seen using this parameter.	It is an optional parameter. The default value of this parameter is 1e-9.

Table 4.8: *Parameters of Gaussian Naïve Bayes*

Attributes	Details
`class_prior_`	This attribute provides the prior probability of each class.
`theta_`	This attribute provides the mean of each feature per class.
`epsilon_`	This parameter provides the absolute additive value to variances.

Table 4.9: *Important attributes of GaussianNB*

Some of the most important methods of Gaussian Naïve Bayes are as follows (*Table 4.10*):

Name of the method	Purpose
`fit(self, X_train, y_train)`	This method takes two parameters: the train data and the train labels.
`predict(self, X_test)`	This method predicts the class labels for `X_test`.
`predict_proba(self, X_test)`	This method finds the probability estimates for `X_test`.

Table 4.10: *Important methods of GaussianNB*

Having seen the important parameters, attributes, and methods of Gaussian Naïve Bayes, let us now move to the experiment part.

Implementation of Gaussian Naïve Bayes

The Gaussian Naïve Bayes explained in the above section can be easily implemented using SciPy. The implementation that follows constitutes four steps and uses the GaussianNB, explained in the previous section. Breast Cancer dataset has been used in the following code:

1. To implement the algorithm, you need to import the following modules:

```
from sklearn.datasets import load_breast_cancer
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import KFold
import numpy as np
```

2. The performance measures can be calculated by finding the number of True Positive, True Negative, False Positive and False Negative. This is done by crafting the following function:

```
def cal_acc(y_pred, y_test):
    TP=0
    TN=0
    FP=0
    FN=0
for i in range(len(y_test)):
    if(y_test[i]==y_pred[i]):
        if(y_pred[i]==1):
            TP+=1
        else:
            TN+=1
    else:
        if(y_pred[i]==1):
            FP+=1
        else:
            FN+=1
return(TP, TN, FP, FN)
```

The classification is carried out using the following code:

```
acc=[]
Data=load_breast_cancer()
X=Data.data
y=Data.target
kf = KFold(n_splits=5)
kf.get_n_splits(X)
```

```
for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    clf = GaussianNB()
    clf.fit(X_train,y_train)
    y_pred=clf.predict(X_test)
    TP, TN, FP, FN=cal_acc(y_pred, y_test)
    accuracy=(TP+TN)/(TP+TN+FP+FN)
    acc.append(accuracy)
av_acc=np.mean(acc)
print(av_acc)
```

Having seen the use of the GaussianNB module, let us summarize the discussion.

Conclusion

This chapter discussed various classification algorithms, their implementation, and the use of pre-defined functions for the classification of standard datasets. The K-means algorithm finds the majority class of the neighbors of a given sample and declares the label. It is one of the simplest algorithms but performs exceptionally well not just in text data but also in images. The Naïve Bayes algorithm calculates the posterior probability of a sample belonging to a particular class and takes the decision accordingly. The algorithm makes some assumptions regarding the ethos of the sample and the dependence of features. Logistic regression also helps us to find the probability of a sample belonging to a given class.

The reader may note that it is important to choose the correct algorithm before finalizing the model for classification. Also, your model needs to be robust, so it is pointless to project the maximum accuracy in various experiments run. It is always good to run experiments many times and declare the average accuracy.

The next chapter introduces the reader to the fascinating world of neural networks. This world has not only empowered humans with the power of machine learning but also gave birth to a newer, better, and fascinating world of deep learning.

Exercises

Multiple Choice Questions

1. Which of the following finds the majority labels of the neighbors and declares the label of an unknown sample?

a. KNN
b. Naïve Bayes
c. Logistic Regression
d. None of the above

2. Which of the following distances are used in KNN?

a. Euclidean Distance
b. Manhattan Distance
c. Minkowski Distance
d. All of the above

3. Which of the following is the default value of the number of neighbors in the K-Neighbors Classifier?

a. 5
b. 3
c. 1
d. None of the above

4. Which function is generally used in the implementation of logistic regression?

a. Sigmoid
b. Ramp
c. Both
d. None of the above

5. Which of the following is not true with reference to Logistic Regression?

a. It uses a log of odds
b. It generates a probability of a sample belonging to a class
c. Both
d. None of the above

6. Logistic Regression is used for?

a. Classification
b. Regression
c. Both
d. None

7. K-nearest neighbors are used for?

a. Classification
b. Regression
c. Both
d. None

8. Naïve Bayes is used for?

a. Classification
b. Regression
c. Both
d. None

9. Which of the following should be true for applying Gaussian Naïve Bayes?
   a. The features should be independent
   b. The features should be dependent
   c. The model does not make any assumption regarding the dependence of features on each other.
   d. The coefficient of correlation between the features should be less than 0.5
10. Which of the following should be true for applying Gaussian Naïve Bayes?
   a. The data should follow Gaussian distribution
   b. The data should follow Binomial distribution
   c. None of the above
   d. The model does not make any assumption regarding the distribution of data

Theory

1. Write the algorithm for classification using K-Nearest Neighbors. Also, implement the algorithm using NumPy.
2. Write the algorithm for classification using Naïve Bayes. Also, implement the algorithm using NumPy.
3. Write the algorithm for classification using Logistic Regression. Also, implement the algorithm using NumPy.
4. What are the various distance measures used in KNN?
5. What are the assumptions regarding the data and features in Gaussian Naïve Bayes?
6. The algorithm for KNN, given in the text, uses Brute Force. Suggest another algorithm which is better in terms of efficiency but preserves the essence of the algorithm.
7. What is the difference between Single Layer Perceptron and Logistic Regression?
8. Can we claim that the accuracy of a model is 100%? If not, why?

Numerical/Programs

1. The coordinates of points and their respective classes have been shown in the following table. Use K-NN to find to which class the point (5,6) belongs?

Point	Class
(2, 2)	0
(2, 4)	0
(3, 2)	1
(3, 4)	1
(2, 5)	0
(6, 4)	1
(1, 2)	0
(1, 7)	0

2. In the above question, use LDA to find the class of the unknown sample.

3. Generate 50 random numbers (Normal Distribution: Mean=10, variance =5). Let the label of these samples be 0. Now, generate another set of 50 random numbers (Normal Distribution: Mean=15 and variance=3). Let the label of these samples be 1. Use K-Fold validation to carry out classification and state the accuracy of the model which uses:

a. KNN

b. Naïve Bayes

c. LDA

d. Logistic Regression

Chapter 5
Neural Network I – The Perceptron

Introduction

If you like music, your mind starts behaving like Alexa. When you see Phoebe Buffay in Friends, the song "Smelly Cat" comes to your mind; on seeing Sheldon Cooper, "Soft Kitty" starts playing, which immediately switches to "Jungle Jungle" on seeing Mowgli. Your mind drenches your thoughts with the melodious voices and takes you deep down, riding on the compositions by Faiz.

And there is a similarity between Alexa and your mind: both learn. While growing up, you listen to songs, memorize them, associate them with situations, characters, cartoons, and so on, and your mind becoming a Jukebox is the outcome of this process.

This chapter describes neural networks, which work similarly. They are inspired by the neurons in the brain, which is the most important and perhaps the most complex organ of the human body. It acts as the Central Processing System (CPU) of a computer system. Like a CPU, the brain receives information from the sense organs, integrates them, processes them, and takes decisions that are conveyed to the various parts of the body. It contains many nerves, connected by connections called neurons. The neural structure was proposed by Cajal in 1911. There are billions of

neurons and trillions of synapses in our bodies. This structure inspired computer scientists, and the Narnia of machine learning was discovered.

This chapter starts with a brief description of the brain and the structure of neurons. The models, learning algorithms, and limitations of neural networks have been divided into two chapters. This chapter deals with the single layer perceptron, and the next chapter discusses the multi-layer perceptron. This chapter also presents the Delta Learning Rule and discusses the applicability of Perceptron in the classification of two different datasets.

Structure

The main topics covered in this chapter are as follows:

- Introduction
- The brain
- The structure of a neuron
- The McCulloch Pitts model
- The Rosenblatt perceptron
- Activation functions
- Implementation of neural networks
- Learning rules
- Experiments with two datasets

Objective

After reading the chapter, the reader will be able to:

- Understand the basic structure of a neuron
- Understand the McCulloch Pitts Model
- Understand the principle of Rosenblatt perceptron
- Understand the idea behind learning
- Implement Rosenblatt perceptron
- Use perceptron to carry out classification

The brain

The brain is the most important and perhaps the most complex organ of the human body. It acts like a computer system. Like the **Central Processing System** or the

CPU of a computer system, the brain receives information from the sense organs, integrates them, processes them, and takes decisions that are conveyed to the various parts of the body. It contains many nerves connected by neurons. There are billions of neurons and trillions of synapses in the body, which forms the crux of the body's neural network.

Before initiating the discussion on the neural network, let us briefly discuss the parts of the brain. The cerebrum, cortex, brain stem, basal ganglia, and cerebellum are some of the important parts of the brain. The largest part, cerebrum, is divided into two hemispheres. The cerebral cortex is an outer layer of grey matter. It covers the core of the white matter. The spontaneous moments are controlled by this layer. The cerebrum is connected to the spinal cord via the brainstem. It contains:

- The midbrain
- The pons
- The medulla oblongata

Breathing is controlled by the brain stem. The coordination between the brain areas is controlled by the basal ganglia, which is the cluster of structures at the center. The coordination and balance are controlled by the cerebellum, which is at the base and the back of the brain. The cerebellum is connected to the brainstem by pairs of tracts. The layer surrounding the brain is meninges, and the skull protects the brain.

Each hemisphere is divided into four lobes. These include the frontal lobes, the parietal lobes, the temporal lobes, and the occipital lobes. The prime functions of the lobes are as follows.

The frontal lobes take care of:

- Problem solving and judgment
- Motor function

The parietal lobes are primarily concerned with:

- Sensation
- Handwriting
- Body position

The temporal lobes take care of:

- Memory and hearing

Finally, the occipital lobes primarily deal with the:

- Visual processing system

Having learned the basics of a human brain, let us now have a look at the structure of a neuron.

The neuron

The human brain and spinal cord constitute the **Central Nervous System** (**CNS**). The nerve impulses are responded by the release of neurotransmitters. The neurons connect to constitute the neural pathways, the neural circuits, and, ultimately, the network.

A neuron can be excited electrically and communicate with each other using synapse. We all have neurons, each one of us, except SpongeBob. It is because sponges do not have neurons neither do plants. Neurons can be classified as follows:

- **Sensory neurons**: Act in response to touch, sound, light
- **Motor neurons**: Get signals from the brain and spinal cord
- **Interneurons**: These neurons are known to connect neurons of the brain or spinal cord to other neurons within the same region

A neural circuit is a group of connected neurons. *Figure 5.1* shows the structure of a neuron. The components of a neuron are:

- Cell body (soma)
- Dendrites
- Axon

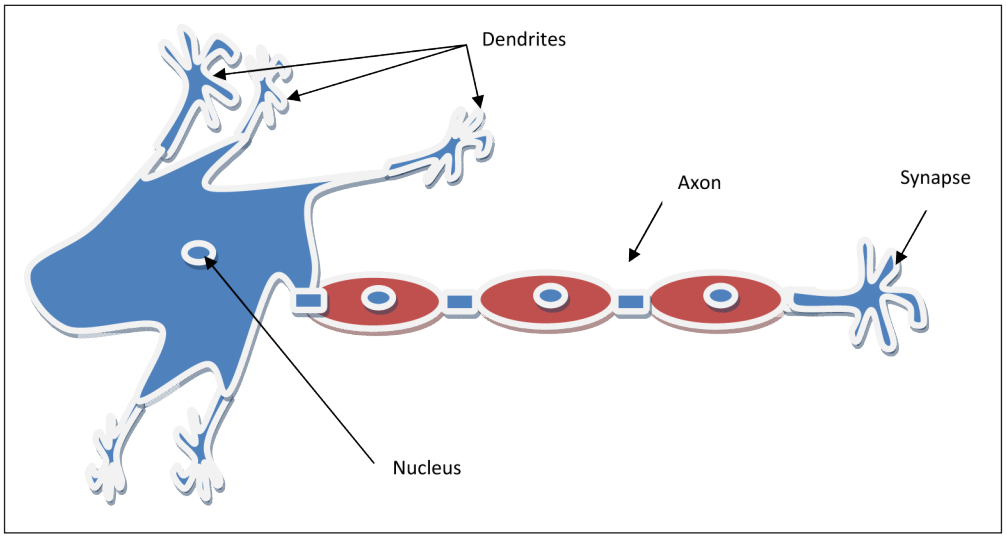


Figure 5.1: *A neuron*

Dendrites receive messages from other neurons. They have a large number of branches. The message is processed in the cell body. The axon takes the message to the other neuron. The process of sending a signal to another neuron is partially chemical and partially electrical.

Generally, neurons get input signals through the dendrites, process the signal, and send the output down the axon.

The structure of neuron inspired the first learning model, the McCulloch Pitts model.

The McCulloch Pitts model

An American Neuropsychologist Warren Sturgis McCulloch and Walter Harry Pitts, Jr., a logician, proposed the McCulloch Pitts model in 1943. The model mimicked the biological neuron and proposed an **LTU** or **Linear Threshold Unit**. The initial model had binary inputs and outputs and restrictions on weights. It is widely regarded by many as the first computational model based on neurons.

The Pitts model is perhaps one of the simplest models of learning. The model takes binary input x_i and summates them. The binary output x_i depends on whether the summation is greater than the threshold or not. So, the only thing that needs to be learned in the threshold. *Figure 5.2* depicts the McCulloch Pitts model:

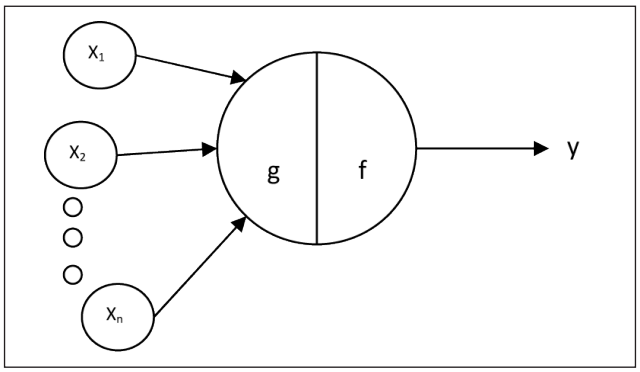


Figure 5.2: *The McCulloch Pitts model*

Here, x_1, x_2, …x_n are the binary inputs (can be 0 or 1), and y is the binary output. The function g is the summation of x_i's and f is the result of applying threshold (*Figure 5.3*). That is:

$$g = \sum_{i=1}^{n} x_i$$

$$f = threshold(g)$$

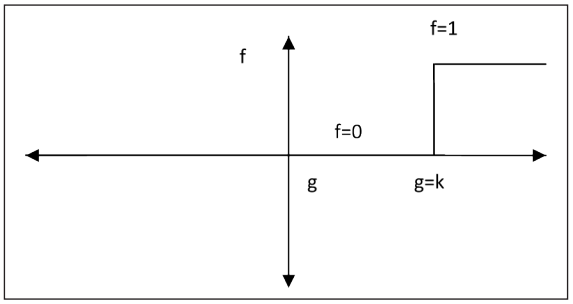


Figure 5.3: *Thresholding in McCulloch Pitts model*

Let us understand the model by keeping the weights of each synapse as 1 or -1. If $w_i = 1$, the input may be referred to as the excitatory input. If the value of $w_i = -1$, the input is referred to as the inhibitory input. The later can be used to model the not gate. Note that, in the discussion that follows if $w_i = 1 \; \forall i$ the weight has not been shown in the model.

The model can be used to create logic gates as well. For example, for creating an AND gate, the inputs are x_1 and x_2 and the output is . The function g is the summation of x_i's:

$$g = \sum_{i=1}^{2} x_i$$

If both the inputs are 1, the output should be high. In all other cases, the output should be low.

Therefore, the threshold of f should be 2.

$$f = threshold\left(g\right) = \begin{cases} 1, g \geq 2 \\ 0, g < 2 \end{cases}$$

The model can be interpreted geometrically in *Figure 5.4*:

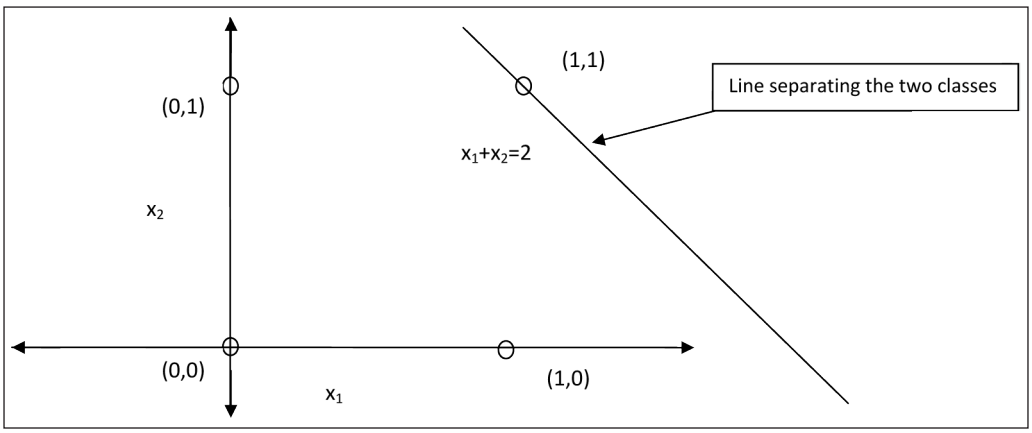


Figure 5.4: *The AND gate*

Likewise, for creating an OR gate, the inputs are x_1 and x_2 and the output is y. The function g is the summation of x_i's. If both the inputs are 0, the output should be low. In all other cases, the output should be high. Therefore, the value of f should be 1:

$$g = \sum_{i=1}^{2} x_i$$

$$f = threshold(g)$$

Figure 5.5 presents the geometrical interpretation of the OR gate created using perceptron:

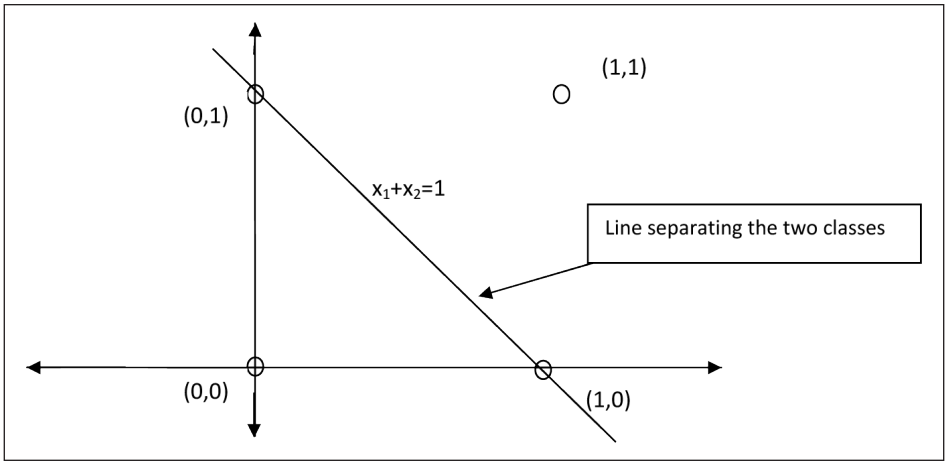


Figure 5.5: *The OR gate*

The above discussion can be extended to 3-dimensions also. That is, perceptron can be used to generate a three-input AND gate as well. Let the inputs to this gate be and the output be x_1, x_2 and x_3 is the output. The function y is the summation of x_i's. If all the inputs are 1, the output should be high. In all other cases, the output should be low. Therefore, the value of f should be 3:

$$g = \sum_{i=1}^{3} x_i$$

$$f = threshold\left(g\right) = \begin{cases} 1, g \geq 3 \\ 0, g < 3 \end{cases}$$

In this case, the separating hyperplane is $x_1 + x_2 + x_3 = 3$. Note that there can be many such hyperplanes which satisfy the condition that the output should be high when all the inputs are 1. *Figure 5.6* shows one such hyperplane:

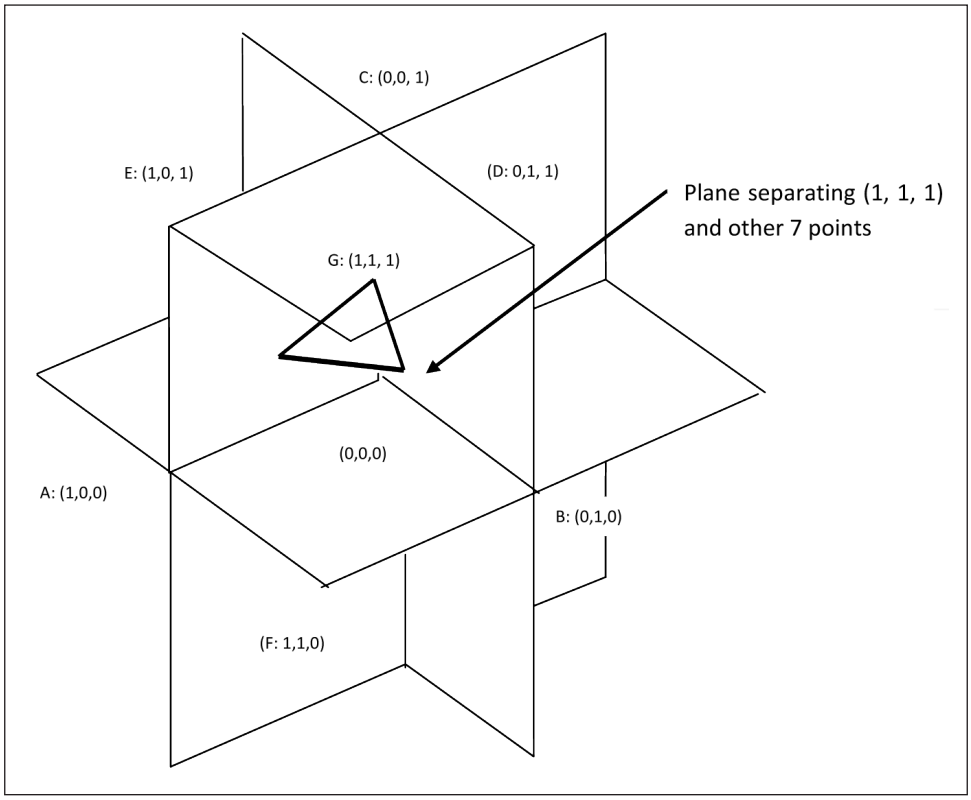


Figure 5.6: *The three input AND gate*

Likewise, for creating a three-input OR gate, x_1, x_2 and x_3 are the inputs, and y is the output. The function g is the summation of x_i's. If any of the inputs are 1, the output should be high. If all the inputs are low, the output should be low. Therefore, the value of f should be 1:

$$g = \sum_{i=1}^{3} x_i$$

$$f = threshold\left(g\right) = \begin{cases} 1, g \geq 1 \\ 0, g < 1 \end{cases}$$

In this case the separating hyperplane is $x_1 + x_2 + x_3 = 1$. Again, there can be many such hyperplanes which satisfy the condition that the output should be high when any of the inputs are 1. *Figure 5.7* shows one such hyperplane:

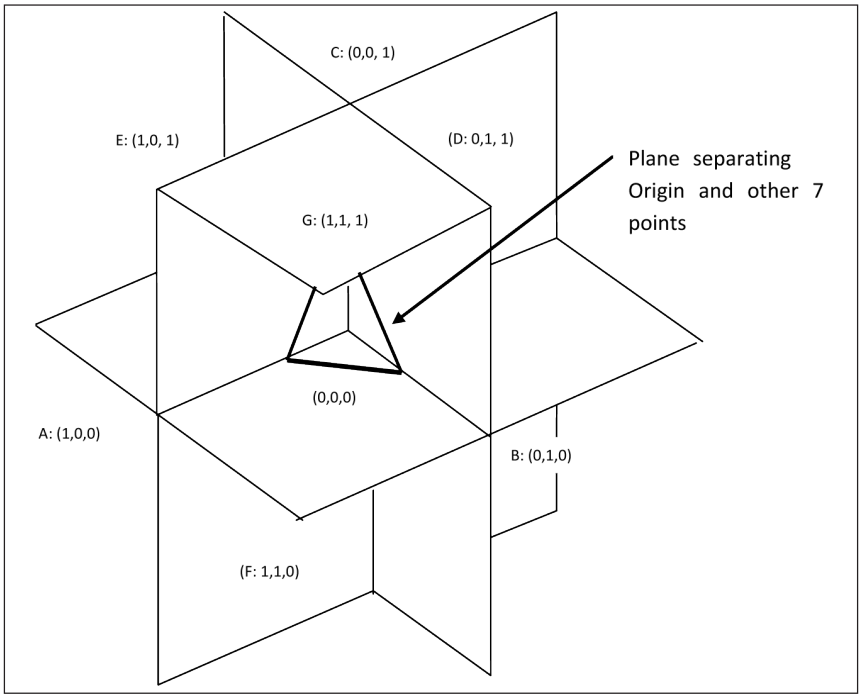


Figure 5.7: *The three input OR gate*

In a NAND gate, the output becomes low if all its outputs are high. In all other cases, it is high. In a NOR gate, on the other hand, the output becomes high, when all its inputs are low. In all other cases, it remains low. One can create a NOR gate and a NAND gate using the above idea. A XOR gate produces one if inputs are alternate. Otherwise, it produces 0. It may be stressed that it is not possible to craft a XOR gate using the above model. However, it can be created by a multi-layer McCulloch Pitts model.

Limitations of the McCulloch Pitts

Despite being extremely useful, the model had its limitations. The notable ones were as follows:

- The inputs to a model are not always binary. The next section discusses a model, which considers the real-valued inputs.
- In the McCulloch Pitts model, the Heaviside step function (the unit step function) is used for thresholding. The following sections explores other activation functions.
- The model cannot deal with inputs that are not linearly separable.

Some of these limitations are handled in the Rosenblatt Perceptron, described in the next section.

The Rosenblatt perceptron model

This model was proposed by Frank Rosenblatt, who was an American Psychologist. He created the first Perceptron device based on the principle discussed in the above section. The difference, however, was the ability of this model to deal with real inputs. Also, each input in this model can have different weights, which are initially random numbers and can change based on some learning rules. The model was simulated on IBM-704 in 1957.

It is a model, which takes input and develops a general linear model by learning the weights and using an activation function. The inputs are multiplied with the weights, and bias is added to the resultant, which acts as an input to the activation function. The weights and bias are initially random numbers and can be learned using the gradient descent method, explained in *Chapter 2* of this book. The original data is divided into two parts: the train data and the test data. The learning is done using the train data, and the testing is done using the test data. The model is shown in *Figure 5.8*.

In the model, the input is $[X_1, X_2, X_3, \ldots X_m]$, the weights are $[w_1, w_2, w_3, \ldots w_m]$ and the bias is b. The inputs are multiplied with weights, summated, and added with the bias to give u, as depicted in the following equation. It is then passed, as the argument to the activation function f, to generate v. For a classification problem, the threshold then determines the class. Note that the formula used to update the weights is the same as that derived in gradient descent:

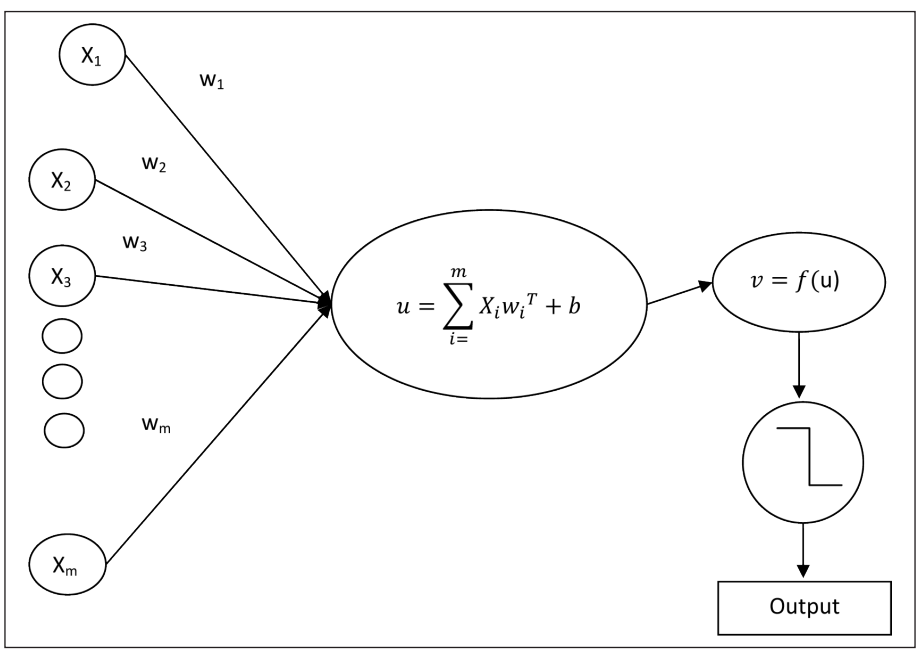


Figure 5.8: *Simple perceptron*

In the equations that follow, X is a matrix of order, $n \times m$ where n is the number of samples, and m is the number of features. The target y is a $n \times 1$ row matrix. The inputs are fed one row at a time to the above network. So, a $1 \times m$ vector is fed to the network. Since there are m features, the weight matrix of $1 \times m$ is required:

$$u = \sum_{i=}^{m} X_i w_i^T + b$$

$$v = f(u)$$

$$output = \begin{cases} 1, if v > \theta \\ 0, if v < \theta \end{cases}$$

If the output matches the expected output, weights would not be changed. Otherwise, they will be changed as per the following equation. The formula has been derived in the next section:

$$w(t+1) = w(t) + \propto \left(d[i] - y[i]\right) X[i]$$

The above discussion has been summarized in the algorithm that follows.

Algorithm

- **Creating train test data**: Divide the data into train and test sets.
- **Network creation**: If the data has n features, create a network with n input neurons and a bais. For the binary classification problem, there can be a single output neuron.
- **Initialization**: Initialize the weight (matrix of order $n \times 1$) randomly. Also, initialize the bais randomly.
- **Parameters**: Decide the learning rate.
- **Learning**: Apply the formula to change weights. The weights can be changes by taking one sample at a time or a set of inputs. Repeat till the changes in weights are small enough.

Figure 5.9 shows the steps of the learning algorithm. Note that, the bais can also be considered as an input with $xi = 1$. It would save the model from learning the bais separately:

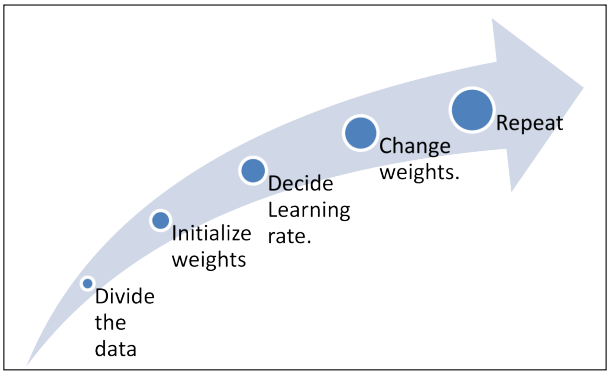


Figure 5.9: *Learning weights in perceptron*

Since activation functions play an important part in the learning, the next section presents a brief overview of the activation functions and their attributes.

Activation functions

The summation of products of weights and inputs, added to the bias, is given as input to the activation function. Some of the important activation functions for learning are as follows.

Unit step

The function can be mathematically represented as follows:

$$f(x) = \begin{cases} 1, if x > 0 \\ 0, if x < 0 \end{cases}$$

The graph of the function has been shown in *Figure 5.10*. The function simply allows the signal to pass through if it is positive. The derivative of the function is 0:

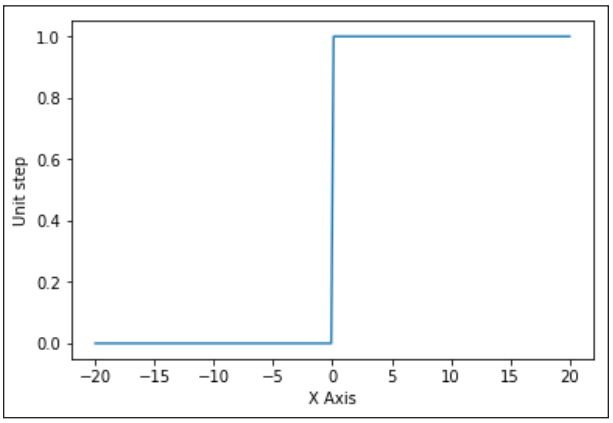


Figure 5.10: *The unit step function*

The function, being simple, can be used as the activation function in single layer perceptron, if the binary output is desired.

sgn

The function can be mathematically represented as follows:

$$f(x) = \begin{cases} 1, if x > 0 \\ -1, if x < 0 \end{cases}$$

The graph of the function has been shown in *Figure 5.11*. The function simply returns 1; if the function is positive, else it returns *a -1*. The derivative of the function is 0:

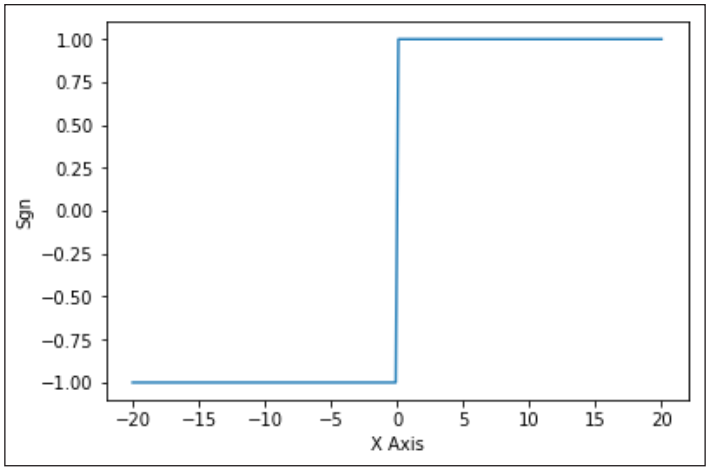


Figure 5.11: *The sgn function*

The function, like the unit step function, is simple and can be used as the activation function in single layer perceptron, if output [-1, 1] is desired.

Sigmoid

The `sigmoid` function is one of the most important activation functions. The function can be stated as follows:

$$f(x) = 1/(1+e^{-sx})$$

The function is shown in *Figure 5.12*. The maximum and the minimum value of the function can be calculated as shown. The following derivation proves that the derivative of this function can be expressed in terms of itself, which is an added advantage of using this function for learning:

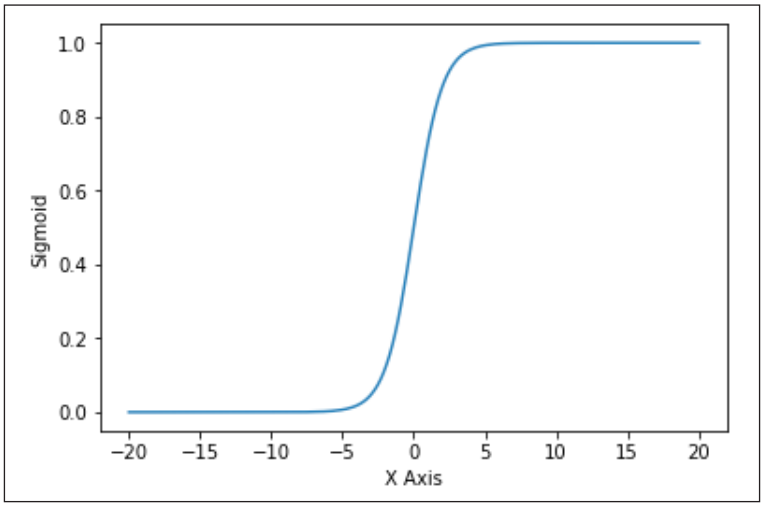


Figure 5.12: *The Sigmoid function*

Minimum value:

$$\text{When } x \to -\infty, \lim_{x\to-\infty}\lim_{x\to-\infty} f\left(x\right) = 1/(1+e^{-sx}) = 0$$

Maximum value:

$$\text{When } x \to \infty, \lim_{x\to\infty}\lim_{x\to\infty} f\left(x\right) = 1/(1+e^{-sx}) = 1$$

Since the value of $f(x)$ becomes 0 at $x = -\infty$ and 1 at $x = \infty$, the value of $f(x)$ lies between 0 and 1.

Derivative

Also, the derivative of $f(x)$ can be expressed in terms of itself:

$$f\left(x\right) = \frac{1}{1+e^{-sx}}$$

$$f'\left(x\right) = -\frac{s\times e^{-sx}}{\left(1+e^{-sx}\right)^2} = -s\times\left(\frac{1}{1+e^{-sx}} - \frac{1}{\left(1+e^{-sx}\right)^2}\right) = -s\times f(1-f)$$

Table 5.1 shows the effect of the variation of the parameter s:

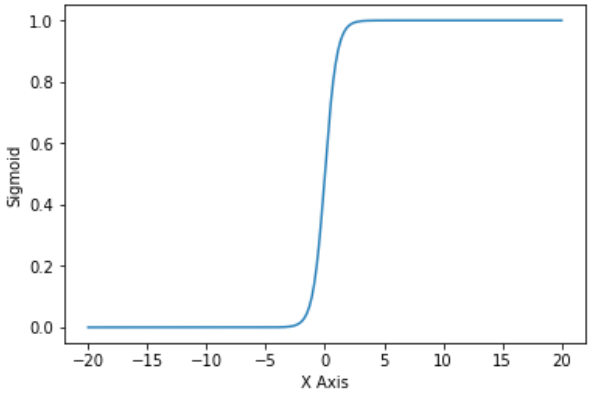
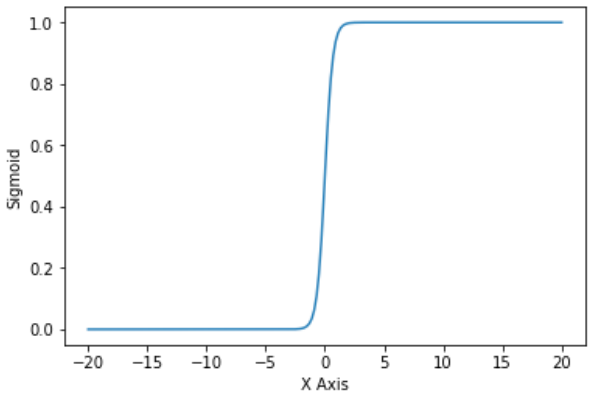
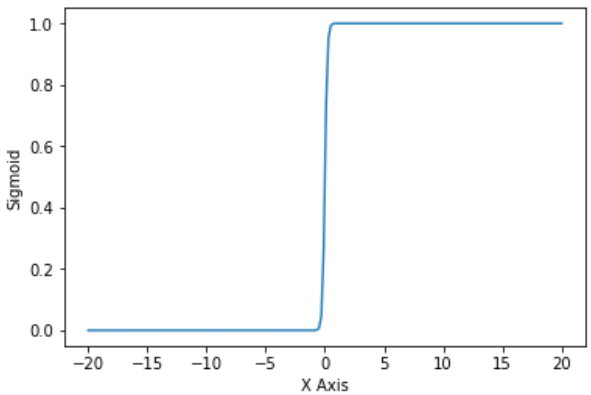
s	Sigmoid
2	
3	
10	

Table 5.1: *Effect of variation of s on Sigmoid*

It can be seen from the table that, for higher values of s, the function behaves like the unit step.

tan-hyperbolic

The tan-hyperbolic function is also one of the most important activation functions. The function can be stated as follows:

$$f(x) = (1 - e^{-sx}) / (1 + e^{-sx})$$

The function is shown in *Figure 5.13*. The maximum and the minimum value of the function can be calculated as follows. *Table 5.2* shows the effect of the variation of the parameter s:

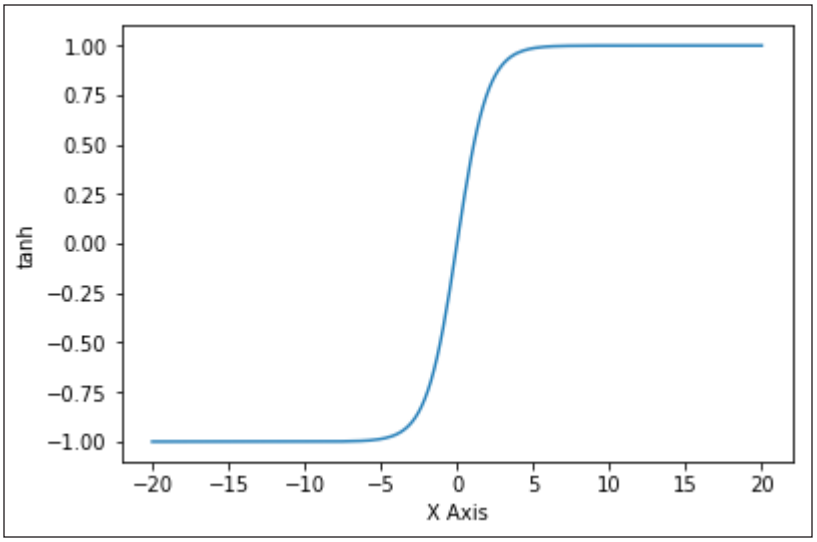


Figure 5.13: *The tanh function*

Minimum value:

$$\text{When } x \to -\infty, \lim_{x \to -\infty} \lim_{x \to -\infty} f(x) = (1 - e^{-sx}) / (1 + e^{-sx}) = -1$$

Maximum value:

$$\text{When } x \to \infty, \lim_{x \to \infty} \lim_{x \to \infty} f(x) = (1 - e^{-sx}) / (1 + e^{-sx}) = 1$$

Hence, the value of $f(x)$ lies between -1 and 1:

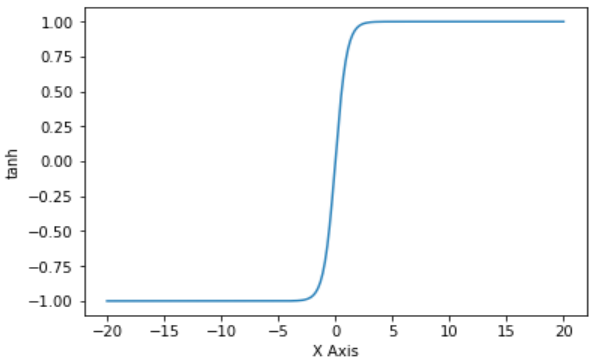
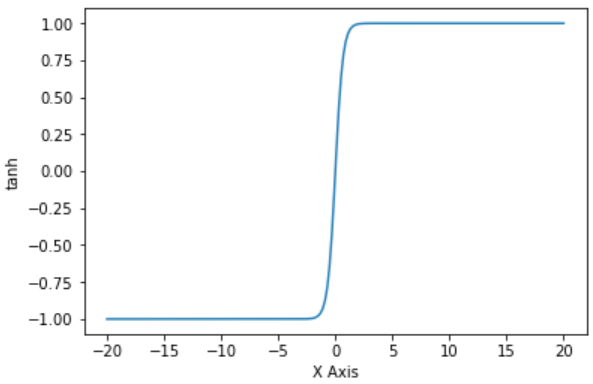
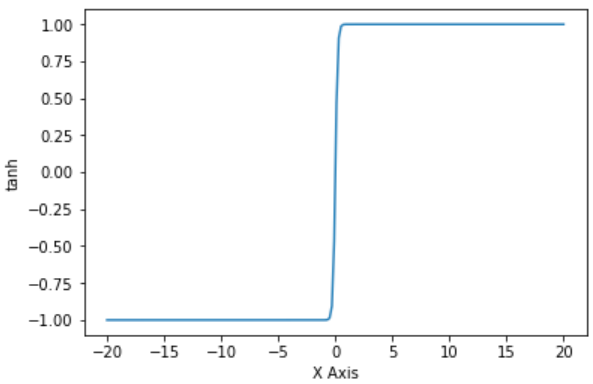
s	tanh(x)
2	
3	
10	

Table 5.2: *Effect of variation of s on tanh*

It can be observed from the table that, for higher values of s, the function behaves like the sgn function.

Implementation

The following code presents the implementation of the Rosenblatt perceptron using Numpy. The reader may revisit *Chapter 2* for the details of the methods for loading the data and carrying out matrix operations.

Code:

```
from sklearn.datasets import load_iris
import numpy as np
from sklearn.utils import shuffle

#Loading the data
Data=load_iris()
X=Data.data
Y=Data.target

#Train test split
x=X[:100,:]
y=Y[:100]
x, y = shuffle(x, y, random_state=0)
x_train=x[:80,:]
y_train=y[:80]
x_test=x[80:,:]
y_test=y[80:]

#Setting parameters
alpha=0.01
s=X.shape

#Initial weights and bais
w=np.random.rand(1,s[1])
b=np.random.rand()
n=(x_train.shape)

#Learning: The Rosenblatt Perceptron
for i in range(n):
```

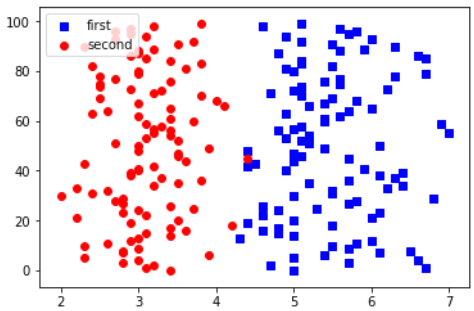
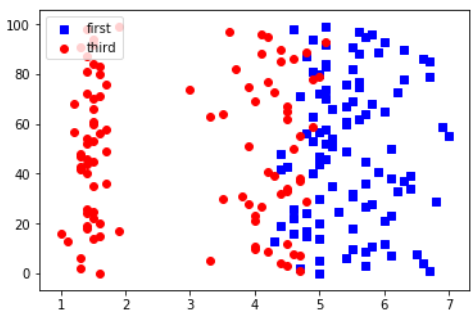
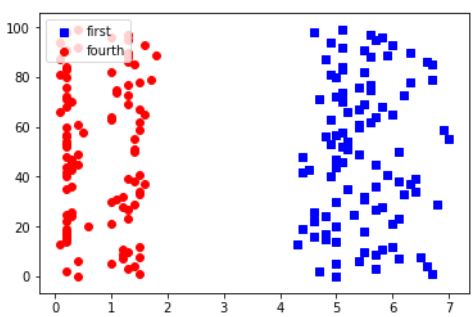
```
    x1=x_train[i,:]
    u=np.matmul(x1,np.transpose(w))
    v=1/(1+np.exp(-1*u))
    if (v>0.5):
        o1=1
    else:
        o1=0
    #print(v,' ',y_train[i])
    y1=y_train[i]
    w=w-alpha*(o1-y1)*x1
    b=b-alpha*(o1-y1)

#Finding accuracy, specificity and senstivity
tp=0
tn=0
fp=0
fn=0
    for i in range(20):
        x1=x_test[i,:]
        u=np.matmul(x1,np.transpose(w))+b
        v=1/(1+np.exp(-1*u))
        if (v>0.5):
            o1=1
        else:
            o1=0
        y1=y_test[i]
        if(o1==1 & y1==1):
            tp+=1
        elif(o1==0 & y1==0):
            tn+=1
        elif(o1==1 & y1==0):
            fp+=1
        else:
            fn+=1
```

```
acc=(tp+tn)/(tp+tn+fp+fn)*100
```

Output:

```
y_test: [0 0 1 0 0 1 1 0 1 1 1 0 0 1 0 1 1 1 0 0]
Accuracy: 100.0
```

Having seen the results, let us spend some time deliberating on why this model can classify the IRIS data with such good accuracy. The data has four features. The pairs of features have been shown in *Table 5.3*. One may note that the data seems linearly separable and hence can be classified using the perceptron:

Feature 1	Feature 2	Figure showing Feature 1 on the X-axis and Feature 2 on the Y-axis
1	2	
1	3	
1	4	

Contd…

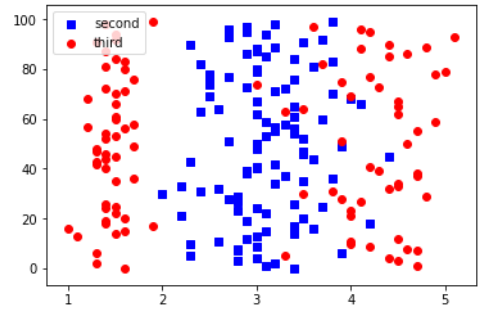
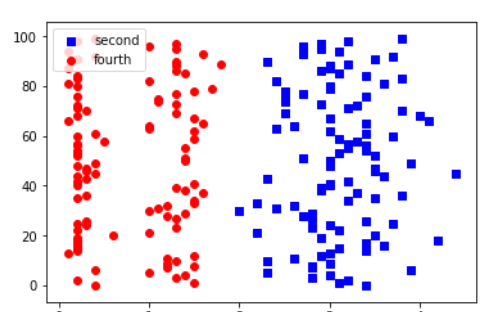
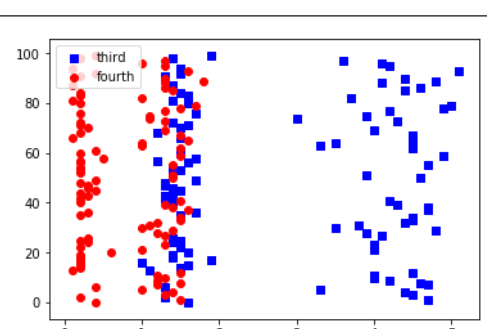
2	3	
2	4	
3	4	

Table 5.3: *Features of Fisher IRIS data*

If we only use two features for classification: 1 and 2, 1 and 4, or 2 and 4, for classification, a linear decision boundary can be formed. The Iris data is, as such, linearly separable and hence can be easily classified using the perceptron model.

Learning

Neural networks can perform many sophisticated tasks. These models find the relationship between the input and the output. The raw input is generally converted into feature space. And the model learns the weight of each feature, which is the same as learning the importance of a feature. For the time being, let us not go into

the conversion of raw input to feature space and assume that the model learns the weights of inputs. This learning can be understood as follows.

The sum of products of the inputs and the corresponding weights is fed into an activation function, which produces some output. The difference between what is being produced by the model and the desired output should be as low as possible. Therefore, the square of this difference should also be as low as possible.

That is, $E_j = (D_j - O_j)^2$ should be minimized, where E_j is the squared error, D_j is the desired output and O_j is the output produced by the model. Note that $O_j = f(u_j)$ and $u_j = x_j w_j$, for each input x_j having weight w_j, f being the activation function.

The gradient of E is given by:

$$\frac{\partial E}{\delta w_j} = \frac{\delta}{\delta w_j}(D_j - O_j)^2$$

$$= -2\left(D_j - O_j\right)\frac{\delta}{\delta w_j}(O_j)$$

$$= -2\left(D_j - O_j\right)\frac{\delta}{\delta w_j}(x_j w_j)$$

$$= -2\left(D_j - O_j\right)\frac{\delta}{\delta w_j}(x_j w_j)$$

$$= -2\left(D_j - O_j\right)x_j$$

The weights can be changed by adding the negative of this gradient to the previous weight, that is:

$$w_j\left(t\right) = w_j\left(t-1\right) - \frac{\partial E}{\delta w_j}$$

Or:

$$w_j\left(t\right) = w_j\left(t-1\right) + \eta\left(D_j - O_j\right)x_j$$

Here η is the learning rate. The value of η, should neither be too high or too low. If the value of η is large, we may overshoot the optimal solution. The small value of η poses the danger of learning too slow.

The above learning rule was proposed by Bernard Widrow and his doctoral student Ted Hoff in 1949 and is hence referred to as the Widrow-Hoff learning rule. This

rule was instrumental in the advances in neural networks. It is also called the Delta learning rule. Rosenblatt model uses the perceptron learning rule, which, though different in origin, is similar to the above. If a linearly separable pattern is presented to the perceptron, the perceptron is guaranteed to learn the weights, if they can be learned. As per Rosenblatt:

The perceptron learning rule is guaranteed to converge in a finite number of steps for all problems that can be solved by a perceptron. That is linearly separable classification problems.

It may be stated here that though we will use this rule in our models, this was certainly not the first learning rule. The first learning rule was proposed by Canadian Psychologist Donald Olding Hebb, who is considered as the father of neural networks. Hebb, in his book *The Organization of Behaviour*, stated that the synaptic efficiency of a cell increases by repeated and persistent stimulation of a cell. The rule can be loosely stated as follows:

"Neurons that fire together wire together."

It is not difficult to understand the rule. For example, team B is given a job, which was being done by another team, say A. The team members have no idea of what is to be done and hence fail in whatever they do in successive attempts. Since they are not very competent, owing up is simply out of the question. Few of them try to learn the job, and the rest start blaming the previous team for their failures. What happens after that is anyone's guess. In B, the learners bind together, and so does the other ones. The rule can be stated as follows:

$$w_j(t) = w_j(t-1) + \eta(O_j)\, x_j$$

Where the symbols have usual meanings. The interested readers can explore the references at the end of this chapter for a detailed discussion on Hebbian Learning.

Perceptron using sklearn

The module `sklearn.linear_model` implements a few general linear models, including the perceptron. The important parameters of the constructor of the perceptron class have been presented in *Table 5.4*:

Parameter	Type	Explanation
`fit_intercept`	Boolean	This parameter determines whether the bais term should be estimated or not. The default value of this parameter is `True`.
`max_iter`	Integer	This parameter determines the epochs. It is an optional parameter, and its default value is `1000`.

Contd…

`tol`	Float or None	This parameter determines the stopping criteria based on loss. It is an optional parameter, and its default value is 1e-3.
`shuffle`	Boolean	This parameter determines if the input should be shuffled after each iteration. It is an optional parameter, and its default value is `True`.
`random_state`	Integer	If you want the model to produce the same result every time it runs, this parameter can be used. It is an optional parameter, and its default value is `None`.
`n_iter_no_change`	Integer	This parameter sets the number of iterations after no change. The default value of this parameter is `5.`
`class_weight`	Dictionary or "balanced" or None	It is an optional parameter. The absence of this parameter assumes each class to be of weight 1. As per the official site: "The "balanced" mode uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as `n_samples / (n_classes * np.bincount(y))`.

Table 5.4: *Parameters of perceptron*

Table 5.5 presents the attributes of the model:

Attribute	Explanation
`coef_`	It gives the weights assigned to the features.
`intercept_`	It shows the constants in decision function.
`n_iter_`	It gives the number of iterations to reach the stopping criteria.

Table 5.4: *Attributes of the model*

The above functions have been exemplified in the following experiments.

Experiments

To understand the usage of the above methods, consider the following experiments:

- Experiment 1 uses the first 100 samples of the Fisher IRIS data, normalizes the data, and carries out classification using the SLP.
- Experiment 2 uses the first 100 samples of the IRIS data and uses the train-test-split to classify the data using the SLP.
- Experiments 3 use the Breast Cancer data, normalize it, and carry out classification using the SLP.

- Experiment 4 uses the Breast Cancer data and applies K-Fold validation (K=10) to classify the data using the SLP.

Note that K-Fold and train-test-split have been discussed in *Chapter 1*.

Experiment 1: Classification of Fisher Iris Data

```
from sklearn.datasets import load_digits
from sklearn.linear_model import Perceptron
from sklearn.datasets import load_iris
import numpy as np
import math

#Load Data
IRIS=load_iris()
X=IRIS.data
y=IRIS.target

#Normalize
max=[]
min=[]
S=X.shape
for i in range(S[1]):
   max.append(np.max(X[:,i]))
   min.append(np.min(X[:,i]))
for i in range(S[1]):
   for j in range(S[0]):
      X[j,i]=(X[j,i]-min[i])/(max[i]-min[i])

#Prepare test train data
arr=np.random.permutation(100)
X=IRIS.data[arr,:]
y=np.vstack((np.zeros((50,1)),np.ones((50,1))))
y=y[arr]
X_train=X[:40,:]
```

```
X_test=X[40:50,:]
y_train=y[:40]
y_test=y[40:50]
X_train1=X[50:90,:]
X_test1=X[90:100,:]
y_train1=y[50:90]
y_test1=y[90:100]
X_train=np.vstack((X_train,X_train1))
y_train=np.vstack((y_train,y_train1))
X_test=np.vstack((X_test,X_test1))
y_test=np.vstack((y_test,y_test1))

#Classify using SLP
clf=Perceptron(random_state=0)
clf.fit(X_train, y_train)
predicted=clf.predict(X_test)
TP=0
TN=0
FN=0
FP=0
for i in range(len(y_test)):
   if(y_test[i]==predicted[i]):
      if(y_test[i]==1):
         TP+=1
      else:
         TN+=1
   else:
      if(predicted[i]==1):
         FP+=1
      else:
         FN+=1
acc=(TP+TN)/(TP+TN+FP+FN)        #accuracy
```

```
sens=TP/(TP+FN)                    #sensitivity
spec=TN/(TN+FP)                    #specificity
```

Experiment 2: Classification of Fisher Iris Data, train-test split

```
from sklearn.linear_model import Perceptron
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import numpy as np
import math

#Load Data
IRIS=load_iris()
X=IRIS.data
y=IRIS.target

#Normalize
max=[]
min=[]
S=X.shape
for i in range(S[1]):
   max.append(np.max(X[:,i]))
   min.append(np.min(X[:,i]))
for i in range(S[1]):
   for j in range(S[0]):
      X[j,i]=(X[j,i]-min[i])/(max[i]-min[i])

#Train test split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.4,random_
state=4)

#Predict using SLP
clf=Perceptron(random_state=0)
```

```
clf.fit(X_train, y_train)
predicted=clf.predict(X_test)
TP=0
TN=0
FN=0
FP=0
for i in range(len(y_test)):
   if(y_test[i]==predicted[i]):
      if(y_test[i]==1):
         TP+=1
      else:
         TN+=1
   else:
      if(predicted[i]==1):
         FP+=1
      else:
         FN+=1
acc=(TP+TN)/(TP+TN+FP+FN)        #accuracy
sens=TP/(TP+FN)                  #sensitivity
spec=TN/(TN+FP)                  #specificity
```

Experiment 3: Classification of Breast Cancer Data

```
from sklearn.linear_model import Perceptron
from sklearn.datasets import load_breast_cancer
import numpy as np
import math

#Load Data
dataset=load_breast_cancer()       #constructor called
X=dataset.data
y=dataset.target
```

```
#Normalize
max=[]
min=[]
S=X.shape
for i in range(S[1]):
   max.append(np.max(X[:,i]))
   min.append(np.min(X[:,i]))
for i in range(S[1]):
   for j in range(S[0]):
      X[j,i]=(X[j,i]-min[i])/(max[i]-min[i])

#Train test data
X_train=X[:400,:]
X_test=X[400:,:]
y_train=y[:400]
y_test=y[400:]

#Classify using Perceptron
clf=Perceptron(random_state=0)
clf.fit(X_train, y_train)
predicted=clf.predict(X_test)
TP=0
TN=0
FN=0
FP=0
for i in range(len(y_test)):
   if(y_test[i]==predicted[i]):
      if(y_test[i]==1):
         TP+=1
      else:
         TN+=1
   else:
```

```
        if(predicted[i]==1):
            FP+=1
        else:
            FN+=1
acc=(TP+TN)/(TP+TN+FP+FN)        #accuracy
sens=TP/(TP+FN)                  #sensitivity
spec=TN/(TN+FP)                  #specificity
```

Experiment 4: Classification of Breast Cancer Data, 10 Fold Validation

```
from sklearn.linear_model import Perceptron
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import KFold
import numpy as np
import math

#Load dataset
dataset=load_breast_cancer()
X=dataset.data
y=dataset.target

#K Fold
kf=KFold(n_splits=10,random_state=None,shuffle=True)
kf.get_n_splits(X)

#Classification Using Perceptron
accur=[]
specificity=[]
senstivity=[]
for train_index, test_index in kf.split(X):
print("TRAIN:", train_index.shape, "TEST:", test_index.shape)
```

```
X_train, X_test = X[train_index], X[test_index]
y_train, y_test = y[train_index], y[test_index]
clf=Perceptron(random_state=0)
clf.fit(X_train, y_train)
predicted=clf.predict(X_test)
TP=0
TN=0
FN=0
FP=0
for i in range(len(y_test)):
   if(y_test[i]==predicted[i]):
      if(y_test[i]==1):
         TP+=1
      else:
         TN+=1
   else:
      if(predicted[i]==1):
         FP+=1
      else:
         FN+=1
acc=(TP+TN)/(TP+TN+FP+FN)
accur.append(acc)
sens=TP/(TP+FN)
senstivity.append(sens)
spec=TN/(TN+FP)
specificity.append(spec)

#Performance
print(np.mean(accur))
print(np.mean(senstivity))
print(np.mean(specificity))
```

The accuracy, specificity, and sensitivity obtained in the four experiments is shown in *Table 5.6*:

Experiment	Accuracy	Sensitivity	Specificity
1	0.95	1.0	0.93
2	0.76	1.0	0.67
3	0.69	0.6	1.0
4	0.96	0.98	0.94

Table 5.6: *Results of Experiments*

Conclusion

This chapter discussed the structure of a neuron and explained how the neural network of the body inspired the Artificial Neural Networks. The so-called first-generation neural networks have been introduced in this chapter. Right from the simplicity of the Mc-Culloch-Pitts model to the elegance of Rosenblatt perceptron has been talked about. The mechanism of learning has also been covered in this chapter.

The implementation of a single layer perceptron (SLP) and important functions in SciPy have also been discussed. One must note that experimentation and empirical analysis are involved tasks, the pre-requisites to which is a sound foundation of the related concepts. The reader is expected to analyze the outputs with various datasets and reason out the differences in the performance measures.

The discussion continues in the next chapter, where multi-layer perceptrons (MLP) and the backpropagation algorithm are discussed. The limitation of the models, given in this chapter, can be handled elegantly by the MLP's. MLP's can thus classify the inputs that are not linearly separable.

Exercises

Multiple Choice Questions

1. Which of the following is the derivative of Sigmoid function $(x) = \frac{1}{1+e^{-sx}}$?

a. $-s \times f \times (1-f)$
b. $-s \times (1-f)$
c. $f \times (1-f)$
d. None of the above

2. For larger values of s, the sigmoid function behaves like?

a. Unit Impulse b. Unit Step

c. Ramp d. None of the above

3. For larger values of s, the tanh function behaves like?

a. Unit Impulse b. Unit Step

c. sgn d. None of the above

4. Which of the following can be used if the probability of a test sample belonging to a particular class is to be determined?

a. Sigmoid b. Tanh

c. Unit step d. None of the above

5. Which of the following is used in the McCulloch Pitts model?

a. Unit step b. Sigmoid

c. tanh d. None of the above

6. The Rosenblatt Perceptron can learn the weights of?

a. Linear separable inputs

b. Any inputs

c. Input with a limited number of features

d. None of the above

7. In delta learning, which of the following can be avoided?

a. Learning of w's b. Learning of bais

c. None d. Both

8. Which of the following cannot be handled by the McCulloch Pitts model?

a. AND gate b. OR gate

c. XOR gate d. None of the above

9. Who is considered the father of neural networks?

a. Hebb b. McCulloch c. Rosenblatt d. Justin Trudeau

10. Which of the following learning rules can be used to train a Perceptron?

a. Delta b. Hebbian

c. Both d. None of the above

Theory

1. What is a neuron? Explain the structure of a neuron.
2. State the types of neurons.
3. Explain the McCulloch Pitts Model.
4. Implement the following using the McCulloch Pitts model.
   a. 2 Input AND gate
   b. 2 Input OR gate
   c. 3 Input AND gate
   d. 3 Input AND gate
   e. 2 Input NAND gate
   f. 2 Input NOR gate
   g. 3 Input NAND gate
   h. 3 Input NOR gate
   i. 4 Input AND gate
5. Implement the above using Rosenblatt perceptron, starting from random weights between 0 and 1.
6. Derive the formula for the change in weights using the Delta Learning Rule.
7. Explain the Hebbian Learning Rule.
8. Find the maximum and minimum value of the following. Also, analyze the effect of changing the value of parameter s.
   a. Sigmoid
   b. tanh
9. Prove that the derivative of sigmoid can be expressed in terms of itself.
10. Explain why the XOR gate cannot be handled by a single layer perceptron.

Programming/Experiments

1. Implement Rosenblatt Perceptron.
2. Analyze the effect of replacing random weights by zeros on the number of iterations, in which convergence is achieved.
3. Analyze the effect of replacing delta learning rule by Hebbian on the number of iterations, in which convergence is achieved. What do you observe about the weights if the number of iterations becomes large?
4. On the Breast Cancer Data set:
   a. Use train-test split (60% Train data and 40% test data)

b. Use train-test split (80% tarin data and 20% test data)
c. Use K-Fold (K=20)
d. Use K-Fold (K=10)
e. Use K-Fold(K=5)

5. Compare the specificity, selectivity, and sensitivity in each case.

6. Perform the above experiment on the Autism Screening Adult dataset (https://archive.ics.uci.edu/ml/datasets/Autism+Screening+Adult) and compare the accuracy in each case.

Chapter 6
Neural Network II – The Multi-Layer Perceptron

Introduction

You have been asked to develop a model capable of segregating the pictures of faces of Dr. Heinz Doofenshmirtz and Major Monogram of Phineas and Ferb fame. You will classify the pictures into the above classes by looking at the higher-level features like nose, eyes, hair, and so on. These features, in turn, can be constructed using various lines (horizontal, vertical, inclined) and curves. So, your model should probably:

- Take a given picture as input
- Find various lines and curves in the picture
- Construct higher-level features from the above features
- Classify the given picture based on the above features

The process can be perceived as a concatenation of layers, each performing some task. The first being the input layer, which takes the input and the last being the output layer, which declares whether the input image is of Dr. Doofenshmirtz or Major Monogram. Everything in between is not visible to the world and hence are hidden layers. These layers extract features. Just for the record, Dr. Doofenshmirtz is not a real doctor; he purchased his degree.

Now, consider each layer as being a single layer perceptron and the concatenation of these Perceptrons as being the multi-layer perceptron. This chapter briefly explores the fascinating world of multi-layer perceptrons and presents the feed-forward model and the back-propagation algorithm for learning. MLP's are capable of handling data that is not linearly separable. This chapter also presents an implementation of the multi-layer perceptrons and its applicability to some of the publically available datasets.

Structure

The main topics covered in this chapter are as follows:

- Introduction
- History of neural networks
- The architecture of the feed-forward neural network
- Back-propagation algorithm
- Feed-forward algorithm
- Implementation of MLP
- Experiments with two datasets

Objective

After reading the chapter, the reader will be able to:

- Appreciate the need of multi-layer perceptrons
- Understand the back-propagation algorithm
- Understand the feed-forward model
- Implement MLP
- Use MLP to carry out classification

History

Knowing the history helps us to deal with the present in a better way. We learn from the mistakes made in the past and ascertain the factors behind our successful endeavors. Also, creating history is fun but inconsequential. So, let us dwell on the history of neural networks, which is as exciting as each episode of Duck Tales. There was initial enthusiasm, followed by a period of dismay, which followed an influx

of funds bringing happiness along-with. *Table 6.1* shows the brief history of neural networks:

Year	Researcher/Group	What was proposed?
1943	Warren McCulloch and Walter Pitts	The proposed model, which takes inputs (excitatory or inhibitory), summates them and make decision-based on the summation
1949	Donald Hebb	He authored the organization of Behavior, which argued, "Neurons that fire together, wire together."
1950	Nathanial Rochester from the IBM research laboratories	It was the first Neural network to be simulated
1959	Bernard Widrow and Marcian Hoff	They developed Adaline and Madaline model. Adaline could predict a binary pattern. Madaline pioneered the application of Neural Network to a practical problem.
1962	Bernard Widrow and Marcian Hoff	They developed a learning paradigm, which adjusts weights after scrutinizing the input.
Problems	_	A paper suggested that the SLP could not be extended to a multilayer. The learning functions used at that time were problematic as they could not be differentiated at each point. The potential of Neural Networks was embroidered after initial success
1975	_	The first multilayer network was created.
1982	John Hopfield	He presented a work in the National Academy of Science, in which he used bidirectional lines in the networks.
1982	US-Japan Conference on Cooperative Neural Network.	Fear of being left behind increased funding.
1986	Three groups of researchers, including David Rumelhart.	Back-propagation networks introduced.

Table 6.1: *History of neural networks*

The table shows the important events in the history of Neural Networks. *Figure 6.1* summarizes the above discussion:

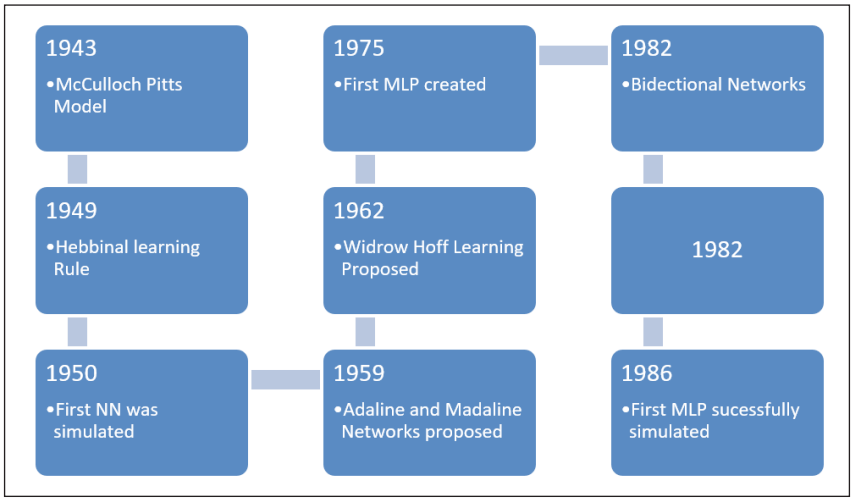


Figure 6.1: *The important events in the life of neural networks*

The discussion continues in the chapter on deep learning.

Introduction to multi-layer perceptrons

The **single layer perceptron** (**SLP**), discussed in the previous chapter, can handle patterns, which can be separated linearly but cannot handle the ones which behave otherwise and, therefore, cannot solve the XOR problem.

This chapter introduces the **multi-layer perceptron** (**MLP**), which does not suffer from the above limitations and hence can act as a universal approximator. In an MLP, each layer is a perceptron whose output acts as the input to the next layer. The sum of products of weights and inputs of a layer $\left(\Sigma_{i=1}^{n} w_{ij} x_i\right)$ added together with the bias (b_{i0}) is fed to the activation function and produce the value (v_i), which becomes the input to the next layer:

$$u_j = \sum_{i=1}^{n} w_{ij} x_i + b_{i0}$$

$$v_i = f(u_j)$$

Here, *wij* is the synaptic weight of the connection between the ith neuron in the first layer to the jth in the next. *Figure 6.2* depicts the above equations:

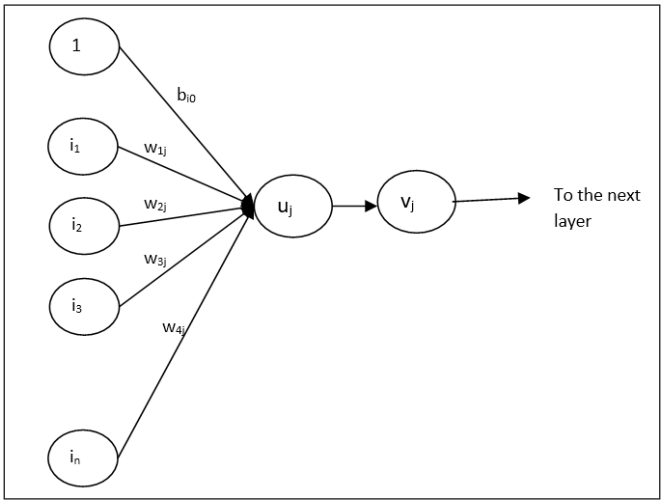


Figure 6.2: *The sum of weights and inputs, added with the bias is fed to the activation function to produce* v_j

So, an MLP contains input layers, hidden layers, and output layers. The network contains at least one hidden layer. It may be noted that theoretically there is no limit on the number of the hidden layers, though more hidden layers make the output unexplainable. Hence, it is better to have a small number of hidden layers.

Moreover, if the hidden layers outputs a linear function, the purpose of having many layers is defeated as the combination of linear combinations is linear. Hence the activation functions in the hidden layer are generally the sigmoid or tanh functions, owing to their non-linearity and also because any function can be expressed in terms of basis functions like an exponential function. So, these functions can generate any function. Though, recently the relu function has become popular.

Architecture

MLP has an input layer, an output layer, and at least one hidden layer. The number of neurons in the input layer can be:

- Same as the number of features of the given data, in which case bias is needed
- Number of features in the given data + 1, the last one for the bais, this extra input would always be one, and its weight would be equivalent to the bias in the above model

The number of neurons in the output layer is the same as the number of outputs. For example, if the given data has m features and n samples and a single response, then $X = \{x_1, x_2, x_3, \ldots, x_m\}$, where x_i is $n \times 1$ vector, and y is $1 \times n$ vector. In this case,

the number of neurons in the input layer would be $m + 1$ and that in the output layer would be 1. In the figure that follows, the neural network contains a single hidden layer which has p neurons (*Figure 6.3*):

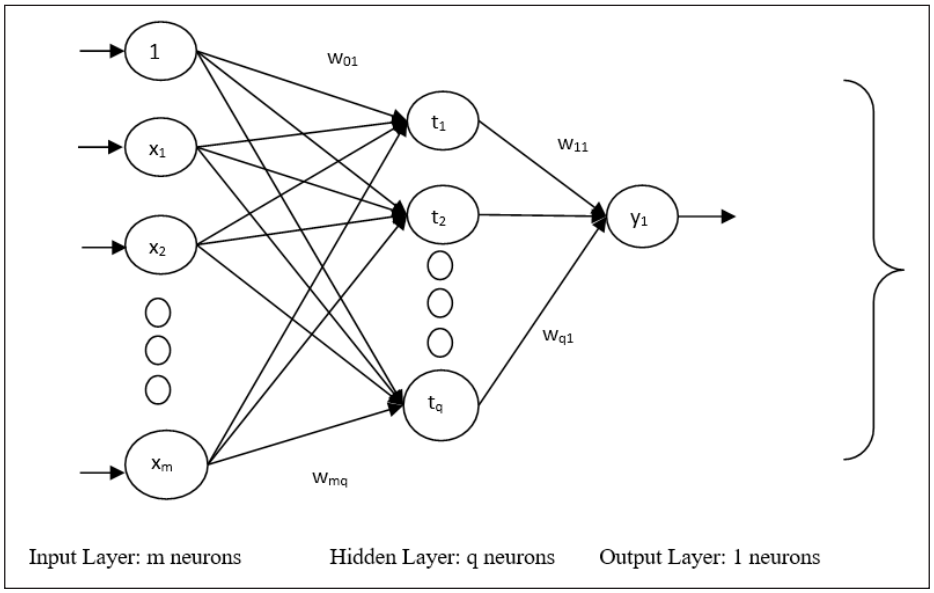


Figure 6.3: *Architecture of the feed-forward model*

The learning of weights, in the above network, is slightly tricky. It is because the input and the expected output is known to us. The network learns the weights between the hidden layer and output layer and uses these updated weights to update the weights between the input and the hidden layer. The algorithm for changing the weights has been discussed in the next section.

Backpropagation algorithm

Consider the weight of the synapse connecting the *jth* neuron in a layer to the *ith* in the previous layer. This weight determines the outcome y_j. The difference between the desired output and the output obtained, for the *nth* input and *jth* neuron is:

$$e(n) = d_i(n) - y_j(n)$$

The value of y_j is obtained by giving to the activation function :

$$y_j(n) = f(u_j)$$

And u_j is the summation of x_i and w_{ij}:

The square of this error multiplied ½, henceforth referred to as the loss function, needs to be minimized, to attain $d_j(x)$:

$$E(n) = \frac{1}{2}(d_i(n) - y_j(n))^2$$

To do so, the weights would have to be changed as per the delta rule. The weight of the synapse connecting the *ith* neuron of the previous layer to the *jth* neuron in the layer, *wij* would change as per the following equation:

$$Äw_{ji} = -\delta E(n) / \delta w_{ji}$$

Now, $E(n)$ depends on $y_j(n)$. This quantity is obtained by feeding $u_j(n)$ to the activation function *f*. Therefore:

$$\frac{\delta E(n)}{\delta w_{ji}} = \frac{\delta E(n)}{\delta y_j} \times \frac{\delta y_j}{\delta u_j} \times \frac{\delta u_j}{\delta w_{ij}}$$

The partial derivative of E with respect to y_j is found as follows:

$$\frac{\delta E(n)}{\delta y_j} = -1 \times \frac{1}{2} \times 2 \times \left(d_i(n) - y_j(n)\right) = -\left(d_i(n) - y_j(n)\right)$$

The partial derivative of y_j for u_j is found as follows:

$$\frac{\delta y_j}{\delta u_j} = f'(u_j)$$

In the case of the sigmoid function, this becomes $f \times (1 - f)$.

Finally, the partial derivative of u_j for w_{ij} is:

$$\frac{\delta u_j}{\delta w_{ij}} = x_i$$

Therefore, $\frac{\delta E(n)}{\delta w_{ji}}$ becomes:

$$\frac{\delta E(n)}{\delta w_{ji}} = -1 \times \left(d_i(n) - y_j(n)\right) \times f'(u_j) \times x_i$$

And finally, the change in the weights would be:

$$Äw_{ji} = -\frac{\delta E(n)}{\delta w_{ji}} = \left(d_i(n) - y_j(n)\right) \times f'(u_j) \times x_i$$

First of all, start by changing the weights of the outermost layer and then move inward, using the new weights obtained in the previous step.

Learning

The data in the neural network, being discussed, travels from the input layer to the output layer. It is the reason why it is referred to as the feed-forward model. The feed-forward backpropagation model works as follows:

- The neural network can have one more than the number of features neurons in the input layer (for the bias). Likewise, the number of neurons of the output layer generally is one less than the number of predictors.
- Initialize the weights by small random numbers.
- Feed input data to the input layer and calculate the output.
- Update the weights using the back-propagation algorithm. The change of weights starts from the output layer, and the weights of the inner layers are changed in the successive steps.
- Stop changing the weights when the change becomes less than the threshold or the maximum number of iterations is reached.

The process has been depicted in the figure below:

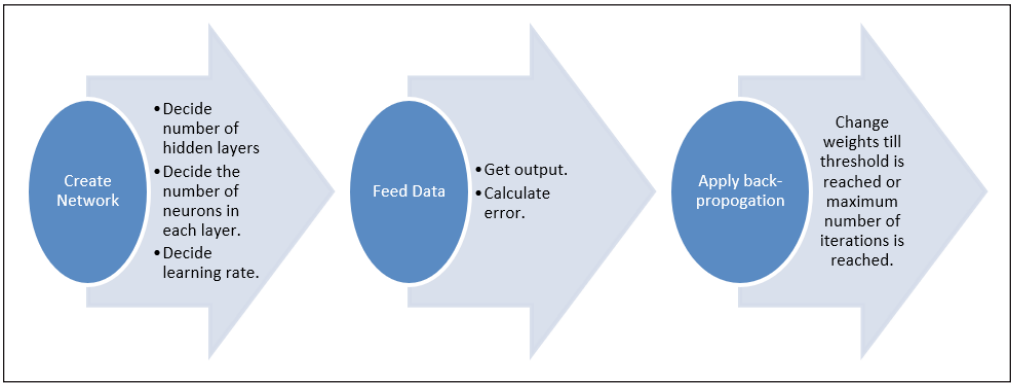


Figure 6.4: *Feed-forward, backpropagation model*

The next section discusses the implementation of MLP.

Implementation

The above algorithm has been implemented from scratch using NumPy. The implementation creates a network of a single hidden layer. The hidden layer contains two neurons. The value of the learning rate has been taken as 0.1 for the

bias. The code follows. The reader is expected to change the learning rate and the number of neurons in the hidden layer to analyze the effect of these parameters on the performance of the network. The MLP has been implemented in the following code:

```
from sklearn.datasets import load_iris
from matplotlib import pyplot as plt
import numpy as np
import math

#Loading Data
IRIS=load_iris()
X=IRIS.data
y=IRIS.target

#Normalization
max=[]
min=[]
S=X.shape
for i in range(S[1]):
   max.append(np.max(X[:,i]))
   min.append(np.min(X[:,i]))
for i in range(S[1]):
   for j in range(S[0]):
      X[j,i]=(X[j,i]-min[i])/(max[i]-min[i])
arr=np.random.permutation(100)
X=IRIS.data[arr,:]
y=np.vstack((np.zeros((50,1)),np.ones((50,1))))
y=y[arr]

#Train test split
X_train=X[:40,:]
X_test=X[40:50,:]
y_train=y[:40]
```

```
y_test=y[40:50]
X_train1=X[50:90,:]
X_test1=X[90:100,:]
y_train1=y[50:90]
y_test1=y[90:100]
X_train=np.vstack((X_train,X_train1))
y_train=np.vstack((y_train,y_train1))
X_test=np.vstack((X_test,X_test1))
y_test=np.vstack((y_test,y_test1))
print(X_train.shape)
print(X_test.shape)

#Initialize weights and bias
W1=np.random.random((4,2))
W2=np.random.random((2,1))
b1=np.random.random((1,2))
b2=np.random.random((1,1))

#Activation
def f(u):
    ans=1/(1+np.exp(-1*u))  #s=1
    return ans
#Learning
y_pred=np.zeros(y_train.shape[0])
for i in range(X_train.shape[0]):
    input_sample=X_train[i,:]
    u1=np.matmul(input_sample,W1)+b1      #1X2
    v1=f(u1)   #1X2
    u2=np.matmul(v1,W2)+b2  #1X1
    v2=f(u2)   #1X1
    if(v2>0.5):       #threshold=0.5
        y_pred[i]=1
    else:
```

```
      y_pred[i]=0
   W2=np.transpose(np.transpose(W2)+0.95*(y_train[i]-y_
pred[i])*(v2)*(1-v2)*v1)   #s=1,learning rate=0.475
   a=np.transpose(input_sample)  #4X1
   b=(y_train[i]-y_pred[i])*(v2)*(1-v2)  #1X1
   c=np.matmul(np.transpose(W2),np.matmul((np.transpose(v1)),(1-v1)))
#1X2
e=np.matmul(b,c)  #1X2
inp=np.zeros((4,1))
for j in range(a.shape[0]):
   inp[j]=a[j]
   delta=np.matmul(inp,e)
   W1=W1+0.95*delta
   b2=b2-0.1*y_pred[i]          #learning rate=0.1
   b1=b1-0.1*y_pred[i]          #learning rate=0.1
print(W2)  #2X1
print(W1)  #4X2

#Testing
corr=0
for i in range(X_test.shape[0]):
   input_sample=X_test[i,:]
   u1=np.matmul(input_sample,W1)+b1      #1X2
   v1=f(u1)   #1X2
   u2=np.matmul(v1,W2)+b2  #1X1
   v2=f(u2)   #1X1
   if(v2>0.5):       #threshold=0.5
      y_pred[i]=1
   else:
      y_pred[i]=0
   if(y_test[i]==y_pred[i]):
      corr+=1
   acc=corr/(y_test.shape[0])   #accuracy
```

```
    print(acc)

#Performance Measures
TP=0
TN=0
FN=0
FP=0
for i in range(len(y_test)):
    if(y_test[i]==y_pred[i]):
        if(y_test[i]==1):
            TP+=1
        else:
            TN+=1
    else:
        if(y_pred[i]==1):
            FP+=1
        else:
            FN+=1
acc=(TP+TN)/(TP+TN+FP+FN)        #accuracy
print(acc)
sens=TP/(TP+FN)                  #sensitivity
print(sens)
spec=TN/(TN+FP)                  #specificity
print(spec)
```

The next section discusses the in-built sklearn function for the implementation of MLP.

Multilayer perceptron using sklearn

The module sklearn.neural_network implements a few neural network models, including the forward backpropagation model. The following discussion uses the `sklearn.neural_network.MLPClassifier`. The important parameters of the constructor of the MLPClassifier have been presented in *Table 6.2*:

Parameter	Type	Explanation
`hidden_layer_sizes`	`tuple, length`	The tuple depicts the number of hidden layers, and the length depicts the number of neurons. The default value is `(100,)`.

`activation`	One of the values from `{'identity', 'logistic', 'tanh', 'relu'}`	It represents the activation function. The default value is `relu`.
		`'identity'` returns `f(x) = x`
		`'logistic'` returns `f(x) = 1 / (1 + exp(-x))`
		`'tanh'` returns `f(x) = tanh(x)`
		`'relu'` returns `f(x) = max(0, x)`

`solver`	One of the values from `{'lbfgs', 'sgd', 'adam'}`.	It represents the solver used in the model. The default value is `'adam'`. `'lbfgs'`: Optimizer in the family of quasi-Newton methods. `'sgd'`: Stochastic gradient descent. `'adam'`: Stochastic gradient-based optimizer proposed by Kingma, Diederik, and Jimmy Ba.
`alpha`	`float`	It represents the learning rate. This parameter is optional, and its default value is `0.0001`.
`batch_size`	`int`	It represents the size of the batches in stochastic optimizers. It is also an optional parameter, and its default value is `'auto'`.
`learning_rate`	One of the values from `{'constant', 'invscaling', 'adaptive'}`	The default value of this parameter is `'constant'`.
`max_iter`	`int`	It represents the maximum number of iterations. It is an optional parameter whose default value is `200`.
`random_state`	`int`	The `random_state` is the seed used by the random number generator.
`Tol`	`float`	It represents the tolerance. It is an optional parameter whose default value is `1e-4`.

Table 6.2: *Parameters of MLPClassifier*

Having seen the parameters of the function, let us now move to the attributes. *Table 6.3* presents the attributes of MLPClassifier:

Attribute	Type	Explanation
`classes_`	Array or List	It gives the class label for each attribute.
`loss_`	Float	It gives the current loss.
`intercepts_`	List	The elements represent the bias vectors.
`n_iter_`	int	It represents the number of iterations run.

Table 6.3: *The attributes of MLPClassifier*

The module provides us with some functions. The `fit`, `predict`, `predict_log_probas`, `predict_probas` are some of the most important such functions.

The `fit` function models the data *X* with *y*. The `predict` function predicts the output of the argument. The `predict_log_prob` returns the logarithms of the probability estimates. Likewise, the `predict_prob` provides us with the probabilities.

Experiments

To understand the usage of the above parameters, attributes, and methods, consider the following experiments. The first experiment uses the first 100 samples of the IRIS dataset. The data has been normalized, divided into the test and the train data, and classified using the MLPClassifier. The size of the network is (3,2), and the value of alpha is 10-3.

Experiment 1: IRIS DATA, Two classes, Normalization, MLP

```
from sklearn.datasets import load_iris
from matplotlib import pyplot as plt
from sklearn.neural_network import MLPClassifier
import numpy as np
import math

# Loading the data
IRIS=load_iris()
X=IRIS.data
y=IRIS.target

#Normalization
max=[]
min=[]
```

```
S=X.shape
for i in range(S[1]):
   max.append(np.max(X[:,i]))
   min.append(np.min(X[:,i]))
for i in range(S[1]):
   for j in range(S[0]):
      X[j,i]=(X[j,i]-min[i])/(max[i]-min[i])

# shuffling and creating test and train data
arr=np.random.permutation(100)
X=IRIS.data[arr,:]
y=np.vstack((np.zeros((50,1)),np.ones((50,1))))
y=y[arr]
X_train=X[:40,:]
X_test=X[40:50,:]
y_train=y[:40]
y_test=y[40:50]
X_train1=X[50:90,:]
X_test1=X[90:100,:]
y_train1=y[50:90]
y_test1=y[90:100]
X_train=np.vstack((X_train,X_train1))
y_train=np.vstack((y_train,y_train1))
X_test=np.vstack((X_test,X_test1))
y_test=np.vstack((y_test,y_test1))
#print(X_train.shape)

#Classification
clf=MLPClassifier(solver='lbfgs',alpha=1e-3,hidden_layer_sizes=(3,    2),
random_state=1)
clf.fit(X_train, y_train)
predicted=clf.predict(X_test)
```

```
#Performance evaluation
TP=0
TN=0
FN=0
FP=0

for i in range(len(y_test)):
   if(y_test[i]==predicted[i]):
      if(y_test[i]==1):
         TP+=1
      else:
         TN+=1
   else:
      if(predicted[i]==1):
         FP+=1
      else:
         FN+=1
acc=(TP+TN)/(TP+TN+FP+FN)        #accuracy
print(acc)
sens=TP/(TP+FN)                  #sensitivity
print(sens)
spec=TN/(TN+FP)                  #specificity
print(spec)
```

Output:

```
1.0
1.0
1.0
```

The second experiment uses the first 100 samples of the Fisher IRIS dataset. The data has been normalized to numbers between 0 and 1. It has been divided into the test and the train data using the train-test split, and the MLPClassifier has been used to classify the data. The size of hidden layers, in the network, is (3,2). The value of alpha is 10-3.

Experiment 2: IRIS DATA, two classes, Normalization, Train-test split, MLP

```
from sklearn.datasets import load_iris
from matplotlib import pyplot as plt
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
import numpy as np
import math

#Loading the data
IRIS=load_iris()
X=IRIS.data[:100,:]
y=IRIS.target[:100]

#Normalization
max=[]
min=[]
S=X.shape
for i in range(S[1]):
   max.append(np.max(X[:,i]))
   min.append(np.min(X[:,i]))
for i in range(S[1]):
   for j in range(S[0]):
      X[j,i]=(X[j,i]-min[i])/(max[i]-min[i])

#Train test split
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.4,random_
state=4)
#Classification
clf=MLPClassifier(solver='lbfgs',alpha=1e-3,hidden_layer_sizes=(3,
2),random_state=1)
clf.fit(X_train, y_train)
predicted=clf.predict(X_test)
```

```
#Performance measures
TP=0
TN=0
FN=0
FP=0
for i in range(len(y_test)):
   if(y_test[i]==predicted[i]):
      if(y_test[i]==1):
         TP+=1
      else:
         TN+=1
   else:
      if(predicted[i]==1):
         FP+=1
      else:
         FN+=1
acc=(TP+TN)/(TP+TN+FP+FN)        #accuracy
print(acc)
sens=TP/(TP+FN)                  #sensitivity
print(sens)
spec=TN/(TN+FP)                  #specificity
print(spec)
```

Output:

```
1.0
1.0
1.0
```

The third experiment deals with the Breast Cancer dataset. The given data has been normalized remove. The data has been divided into the test, and the train data and the MLPClassifier have been applied to classify the data. The size of hidden layers, in the network, is (10,2). The value of alpha is 10-3.

Experiment 3: Breast Cancer Dataset, Two classes, Normalization, MLP

```
from sklearn.datasets import load_breast_cancer
from matplotlib import pyplot as plt
from sklearn.neural_network import MLPClassifier
import numpy as np
import math

#Load Data
dataset=load_breast_cancer()
X=dataset.data
y=dataset.target

#Normalization
max=[]
min=[]
S=X.shape
for i in range(S[1]):
   max.append(np.max(X[:,i]))
   min.append(np.min(X[:,i]))
for i in range(S[1]):
   for j in range(S[0]):
      X[j,i]=(X[j,i]-min[i])/(max[i]-min[i])

# Train Test Split
X_train=X[:400,:]
X_test=X[400:,:]
y_train=y[:400]
y_test=y[400:]

#Classify
clf=MLPClassifier(solver='lbfgs',alpha=1e-5,hidden_layer_sizes=(5,
2),random_state=1)
clf.fit(X_train,y_train)
```

```
#Performance Measures
predicted=clf.predict(X_test)
TP=0
TN=0
FN=0
FP=0
for i in range(len(y_test)):
   if(y_test[i]==predicted[i]):
      if(y_test[i]==1):
         TP+=1
      else:
         TN+=1
   else:
      if(predicted[i]==1):
         FP+=1
      else:
         FN+=1
acc=(TP+TN)/(TP+TN+FP+FN)        #accuracy
print(acc)
sens=TP/(TP+FN)                  #sensitivity
print(sens)
spec=TN/(TN+FP)                  #specificity
print(spec)
```

Output:

```
0.93
0.96
0.89
```

The fourth experiment uses the Breast Cancer dataset. The data has been normalized, divided into the test, and the train data using the K fold validation and classified using the MLPClassifier. The size of hidden layers, in the network, is (3,2). The value of alpha is 10-3.

Experiment 4: Breast Cancer Dataset, Two classes, Normalization, K-Fold Split, MLP

```
from sklearn.datasets import load_breast_cancer
from matplotlib import pyplot as plt
from sklearn.neural_network import MLPClassifier
import numpy as np
fromsklearn.model_selection import KFold
import math

#Load Data
dataset=load_breast_cancer()
X=dataset.data
y=dataset.target

#K Fold Validation
kf=KFold(n_splits=10,random_state=None,shuffle=False)
kf.get_n_splits(X)
accur=[]
specificity=[]
senstivity=[]
for train_index, test_index in kf.split(X):
   #print("TRAIN:", train_index.shape, "TEST:", test_index.shape)
   X_train, X_test = X[train_index], X[test_index]
   y_train, y_test = y[train_index], y[test_index]
   clf=MLPClassifier(solver='lbfgs',alpha=1e-3,hidden_layer_sizes=(8,
   2))
   clf.fit(X_train,y_train)
   predicted=clf.predict(X_test)
   TP=0
   TN=0
   FN=0
   FP=0
   for i in range(len(y_test)):
      if(y_test[i]==predicted[i]):
```

```
            if(y_test[i]==1):
                TP+=1
            else:
                TN+=1
        else:
            if(predicted[i]==1):
                FP+=1
            else:
                FN+=1
    acc=(TP+TN)/(TP+TN+FP+FN)
    accur.append(acc)
    if((TP+FN)!=0):
        sens=TP/(TP+FN)
    else:
        sens=0
    senstivity.append(sens)
    if((TN+FP)!=0):
        spec=TN/(TN+FP)
    else:
        spec=0
    specificity.append(spec)
print(np.mean(accur))
print(np.mean(senstivity))
print(np.mean(specificity))
```

The results of the above experiments are shown in the following table (*Table 6.4*):

Experiment	Accuracy	Sensitivity	Specificity
1	1.0	1.0	1.0
2	1.0	1.0	1.0
3	0.93	0.96	0.89
4	0.71	0.89	0.18

Table 6.4: *Results of Experiments 1, 2, 3 and 4*

Conclusion

The single layer neural networks, discussed in the last chapter, were able to perform many tasks, but not all. Therefore, the interest in neural networks waned after an initial surge. One of the reasons for this was the inability of the single layer perceptrons to classify data, which was not linearly separable. This lead to the advent of the multi-layer neural networks, which could handle the non-linearly separable data and could learn any function.

This chapter presented a brief overview of multi-layer perceptrons. The history and architecture of the neural networks have also been discussed in this chapter. The last chapter discussed the activation functions, which can be discrete or continuous. The unit step function is an example of a discrete activation function. The examples of continuous activation functions are sigmoid and the hyperbolic tangent function. The later helps to develop a model capable of predicting a continuous function. This chapter introduced the feed-forward models, which used continuous activation functions.

It may also be noted that initially, the number of layers in the network could not be increased as there was no way of learning the weights since the output of the hidden layer was not known. The back-propagation model proposed in 1986 changed the discourse and set the bells ringing. The model helped to learn the weights of the output layer, after which the weights of the hidden layer can be learned. It is one of the most used algorithms for learning the weights in neural networks. It may be stated that this algorithm was certainly not the first to be proposed. Firstly, Rosenblatt proposed the random initialization of weights, followed by the learning of the weights of the outer layer. This chapter discussed the backpropagation algorithm and presented arguments in favor of multi-layer perceptrons as Universal Approximators.

The next chapter discusses the support vector machines, which is much better in classifying the patterns as it uses only some of the inputs (called support vectors) for classification. Also, the idea behind classification is markedly different from neural networks. The reader is expected to attempt the exercises to develop a better understanding of the chapter.

Exercises

Multiple Choice Questions

1. Generally, the initial weights in a Neural Network are small random numbers. Why?

a. So that the learning algorithm does not lead to large random numbers and hence saturate the network
b. So that computation time is saved
c. The given statement is not true
d. None of the above

2. Which of the two initializations of weights is generally better?
a. All the initial weights are equal
b. They are not equal
c. Both the above situations are equivalent
d. Cannot determine

3. Training of a network can be done using?
a. Single input at a time
b. Set of inputs
c. All the inputs together
d. All of the above

4. In backpropagation, the weights of which layer are modified first?
a. Last Layer
b. First Layer
c. Hidden Layer
d. All of the above

5. Which of the following activation functions can be used in MLP?
a. Relu
b. Sigmoid
c. tanh
d. All of the above

6. Which of the following activation functions cannot be used in MLP?
a. Relu
b. Sigmoid
c. Unit step
d. All of the above

7. Which of the following cannot handle the XOR problem?
a. MLP
b. SLP
c. Both
d. None of the above

8. What is the minimum number of hidden layers in MLP?
a. 0
b. 1
c. 2
d. None of the above

9. Which of the following can be classified using MLP?
a. Two class problem
b. Multi-class problem
c. Both
d. None of the above

10. A neural network can have?

a. Different activation functions at each neuron

b. Any number of hidden layers

c. Any number of neurons in a layer

d. All of the above

Theory

1. Explain the backpropagation algorithm and derive the formula for change in weights.

2. Examine the problems in SLP and argue in favor of MLP.

3. Prove that MLP can act as a universal approximator.

4. Write the algorithm for classification using a feed-forward back-propagation model.

5. Which activation functions can be used in MLP?

6. What is the importance of the learning rate?

7. Explore research papers, given in the reference at the end of this book, and write a note on deciding the number of hidden layers and neurons in each layer.

8. Write a short note on the history of neural networks.

Practical/Coding

1. Implement a neural network using numpy for classifying any classification dataset from the UCI machine learning Repository.

a. Analyze the effect of replacing random weights by zeros on the number of iterations. In which of the two convergence is achieved?

b. Analyze the effect of replacing delta learning rule by Hebbian on the number of iterations. In which convergence is achieved? What do you observe about the weights if the number of iterations becomes large?

2. On the Breast Cancer Data set:

a. Normalize the data

b. Use K-Fold (K=10)

Change the number of hidden layers and the number of neurons in each layer.

Report the average accuracy, selectivity, and sensitivity in each case.

3. Perform the above experiment on the Autism Screening Adult dataset (https://archive.ics.uci.edu/ml/datasets/Autism+Screening+Adult) and compare the accuracy in each case.

Chapter 7
Support Vector Machines

Introduction

The discussion so far leads to the conclusion that if a test sample is far away from the decision boundary, its probability of belonging to a particular class is more, as compared to a sample which is near to the decision boundary. It is because the decision boundary is crafted using the train data, and we intend to classify the test data. Therefore, the assignment of a label to a test sample, very near to the decision boundary, may not be correct. The above premise suggests that the samples further from the decision boundary are more likely to be correctly classified. The reader is requested to appreciate this idea before proceeding any further. This chapter introduces the reader to support vector machines. The classifier explained in this chapter is based on the idea of the Maximum Margin Classifier.

Support vector machines are perhaps one of the best machine learning algorithms. They are elegant, effective, and even work for data having very large dimensions. These machines, therefore, handle the curse of dimensionality gracefully. These machines do not use the whole data to craft the separating hyperplane, but only a small subset of the training data called the support vectors. It makes these machines' memory efficient. Though the algorithm is based on the creation of hyperplane for linearly separable data, the model can be extended to non-linearly separable data using the kernel trick. Also, the concept of cost has been explained in the chapter,

which allows the misclassification of the train data to achieve a better performance on the test data.

The following sections also explain the implementation of SVM using `sklearn.svm`. The reader will be able to appreciate the mathematical basis of SVM, use SVM for classifying the numeric data and the images using the experiments explained in this chapter. The discussion continues in the *Appendix* of this book, which introduces regression using the support vector machines.

Structure

The main topics covered in this chapter are as follows:

- Introduction
- The Maximum Margin Classifier
- Maximizing the margins
- The cost parameter
- The kernel trick
- Implementation of SVM
- Experiments with two datasets

Objective

After reading the chapter, the reader will be able to:

- Appreciate the importance of Maximum Margin Classifiers
- Understand the derivation of maximum margin
- Understand the cost parameter and kernel trick
- Implement SVM
- Use SVM to carry out classification

The Maximum Margin Classifier

The samples of the data, shown in *Figure 7.1*, belong to two classes, and it is desired to find a line that separates the space. The symbol **X** represents the samples of class I, and the symbol * represents the samples of class II. For the sake of mathematical convenience, we take the labels of the samples belonging to class I as -1 and of those belonging to class II as 1. Assume that the classifier used to accomplish the task comes up with the line shown in *Figure 7.1*, which is very near to class II. Another classifier comes up with that shown in *Figure 7.2*, which is very near to class I. Both

the lines can classify the samples into two classes but may not give good results for the test data. The above analysis takes into consideration only two features of the data. However, the concept developed in the following discussion can be extended to multiple features, in which case a hyper-plane instead of a line would be generated by the classifier:

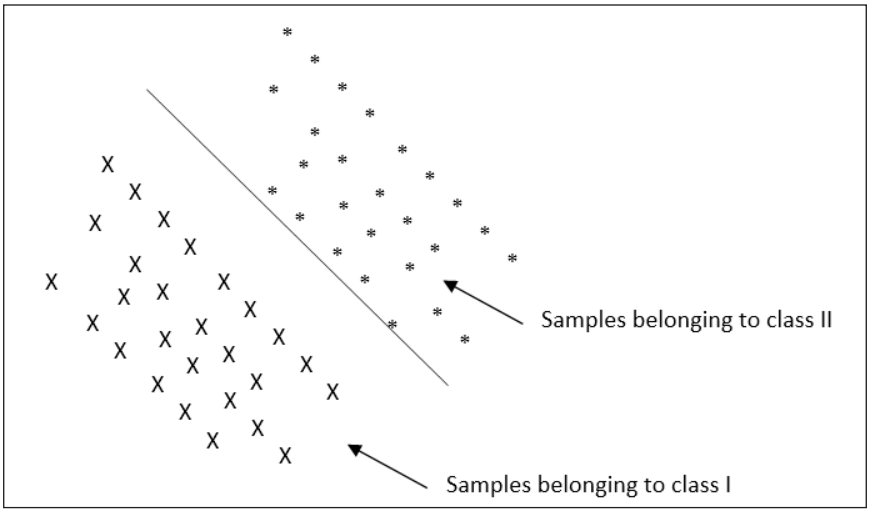


Figure 7.1: *The classifier gives a line very near to class II*

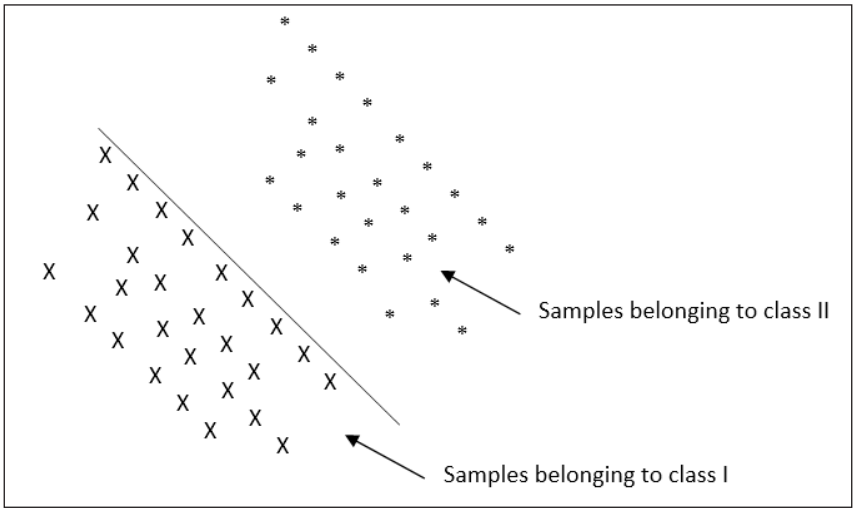


Figure 7.2: *The classifier gives a line very near to class I*

If we can maximize the width of the gutter shown in *Figure 7.3* and the classifier generates a line in the middle of the gutter, the chances of enhanced performance of the classifier with the test data would increase:

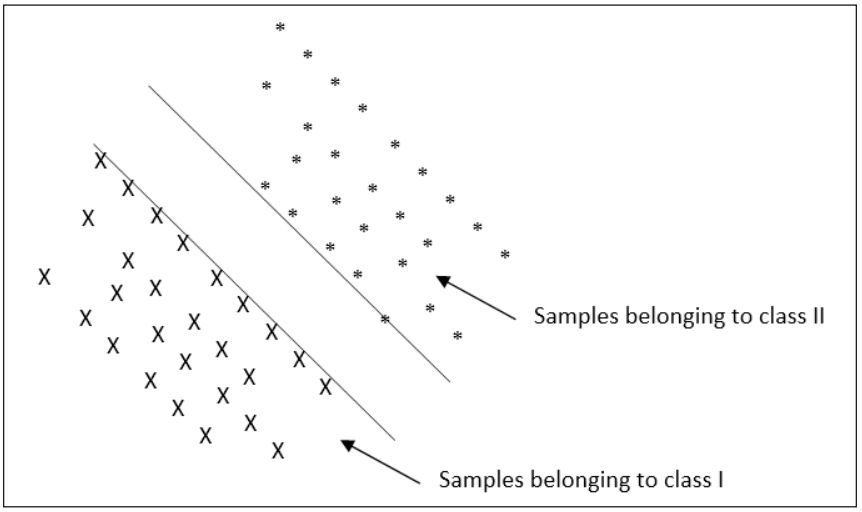


Figure 7.3: *It is desired to maximize the width of the gutter between the two classes*

To do so, we find a vector representing the width of this gutter. Let any point $\left(\overline{x}_1\right)$ belonging to class I satisfy:

$$\overline{x}_1.\overline{w}+b\le -1 \quad \ldots(1)$$

And any point () belonging to class II satisfy:

$$\overline{x}_2.\overline{w}+b\ge 1 \quad \ldots(2)$$

That is:

$$\overline{x}_1.\overline{w}+b\le -1,\ if\ y_i=-1 \quad \ldots(3)$$

And

$$\overline{x}_2.\overline{w}+b\ge 1,\ if\ y_i=1 \quad \ldots(4)$$

The above two equations can be written as:

$$y_i(\overline{x}_t.\overline{w}+b)\ge 1 \quad \ldots(5)$$

And x_i for, the samples in the gutter satisfies:

$$y_i(\overline{x}_t.\overline{w}+b)=1 \quad \ldots(6)$$

The hyperplane, in this case, would be represented by:

$$i(\overline{x}_t.\overline{w}+b)=0 \quad \ldots(7)$$

If the vectors $\overline{x}_1$ and $\overline{x}_2$ represent those to the two lines passing through the endpoints of the two classes (*Figure 7.4*), then $(\overline{x}_1 - \overline{x}_2)$ represents the width of the gutter (*Figure 7.5*):

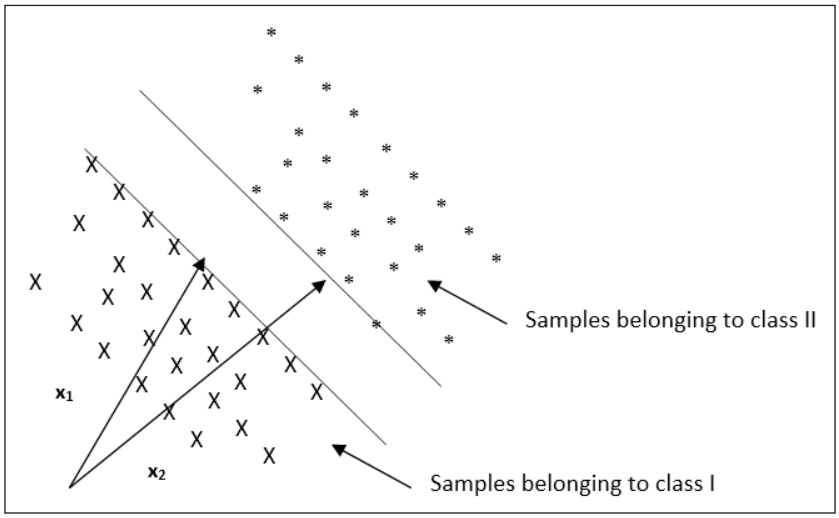


Figure 7.4: *The gutter between the two samples needs to be maximized*

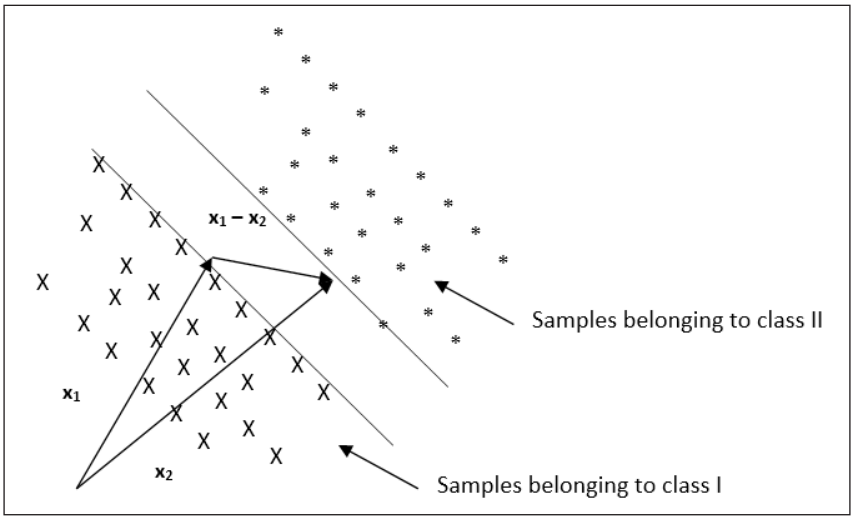


Figure 7.5: $(\overline{x}_1 - \overline{x}_2)$ *represents the width of the gutter*

The magnitude of this vector is $\left|(\overline{x}_1 - \overline{x}_2).\frac{w}{\|w\|}\right|$, which becomes $\frac{2}{\|w\|}$ by equation (6).

Having found the width of the gutter, let us move to maximize this width. The next section uses Lagrange's method to accomplish this task.

Maximizing the margins

In the following discussion, the labels $y \in \{-1, 1\}$ and the classifier is represented by $f = g(X \times W^T)$. The weights W are to be determined. For a given sample x_i, the weight and the value of g determine whether a sample belongs to a particular class. The discussion in the previous section suggests that in this case, a linear function represents the classifier.

The distance between two samples is proportional to:

$$d \propto 1 / \|w\| \quad \ldots(8)$$

To maximize this distance, we can minimize $\|w\|$ or for that matter $\frac{1}{2}\|w\|^2$ subject to constraint:

$$1 - y_i \left(x_i w^t + b \right) = 0 \quad \ldots(9)$$

Note that in the above equation both x_i and w are one-dimensional matrices. The problem can thus be solved using Lagrange's method. The Lagrange's would, therefore be:

$$L = \frac{1}{2}\|w\|^2 + \sum_{i=1}^{m} \alpha_i \left(1 - y_i \left(x_i w^t + b \right) \right) \quad \ldots(10)$$

To minimize L, we find its partial derivative for w and b:

$$\frac{\delta L}{\delta w} = \frac{\delta}{\delta w} \left(\frac{1}{2}\|w\|^2 + \sum_{i=1}^{m} \alpha_i \left(1 - y_i \left(x_i w^t + b \right) \right) \right) \quad \ldots(11)$$

$$\frac{\delta L}{\delta w} = w - \sum_{i=1}^{m} y_i \alpha_i x_i \quad \ldots(12)$$

Putting:

$$\frac{\delta L}{\delta w} = 0 \quad \ldots(13)$$

We get:

$$w = \sum_{i=1}^{m} y_i \alpha_i x_i \quad \ldots(14)$$

Similarly, differentiating L with respect to b, we get:

$$\frac{\delta L}{\delta b} = \frac{\delta}{\delta b} \left(\frac{1}{2}\|w\|^2 + \sum_{i=1}^{m} \alpha_i \left(1 - y_i \left(x_i w^t + b \right) \right) \right) \quad \ldots(15)$$

$$\frac{\delta L}{\delta b} = -\sum_{i=1}^{m} y_i \alpha_i \quad \ldots(16)$$

Putting:

$$\frac{\delta L}{\delta b} = 0 \quad \ldots(17)$$

We get:

$$\sum_{i=1}^{m} \alpha_i y_i = 0 \quad \ldots(18)$$

For all positive α_i's:

We can substitute $w = \sum_{i=1}^{m} y_i \alpha_i x_i$ in L, we get:

$$L = \frac{1}{2}\|w\|^2 + \sum_{i=1}^{m} \alpha_i \left(1 - y_i \left(x_i w^t + b\right)\right) \quad \ldots(19)$$

$$L = \frac{1}{2} w w^T + \sum_{i=1}^{m} \alpha_i \left(1 - y_i \left(x_i w^t + b\right)\right) \quad \ldots(20)$$

$$L = \sum_{i=1}^{m}\sum_{j=1}^{m} \frac{1}{2} y_i y_j \alpha_i \alpha_j x_i x_j^T + \sum_{i=1}^{m}\sum_{j=1}^{m} \alpha_i (1 - y_i \left(y_j \alpha_j x_j^T + b\right)) \quad \ldots(21)$$

$$L = -\sum_{i=1}^{m}\sum_{j=1}^{m} \frac{1}{2} y_i y_j \alpha_i \alpha_j x_i x_j^T + \sum_{j=1}^{m} \alpha_i \quad \ldots(22)$$

Subject to:

$$\sum_{i=1}^{m} \alpha_i y_i = 0 \text{ and } \alpha_i \geq 0 \quad \ldots(23)$$

Note that the values of y_i, y_j, x_i and x_i are known. Hence the above system of equations can be solved. However, some of the α_i's will be zeros. The non-zero α_i's will determine the decision boundary. The following function represents the decision:

$$\sum_{i=1}^{m} \alpha_i y_i \overline{x}_i x_i + b \geq 0, \textit{then } y_i = 1 \quad \ldots(24)$$

$$\textit{elsey}_i = -1 \quad \ldots(25)$$

Having discussed the mathematics, let us now move to the importance of the cost parameter and how to handle a non-linearly separable case using the cost parameter.

The non-separable patterns and the cost parameter

The above theory works if there is a wide margin between two linearly separable data. Now consider a situation where it is not the case. For example, in *Figure 7.6*, if the * and the X in the gutter can be ignored, the above theory can be applied:

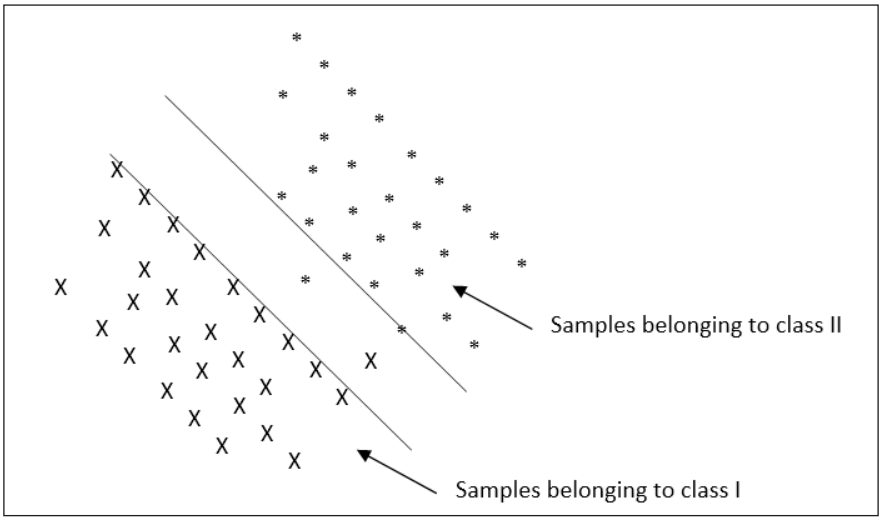


Figure 7.6: *The case of non-separable data and the importance of cost function*

Let us reframe the optimization problem so that a small amount of misclassification may be allowed if instead of that, the margins so formed are far apart. As per Haykin [2], the definition of hyperplane can be derived by introducing an error term in equation (5). That is:

$$y_i\left(\bar{x}_i.\bar{w}+b\right)\geq 1-\xi \quad \ldots(26)$$

If the value of the new variable introduced is between 0 and 1, the test sample will be in the gutter, on the correct side of the hyperplane. To find the hyperplane for which misclassification is minimized, we need to minimize:

$$f\left(\xi\right)=\sum_{i=1}^{m} g(\xi-1) \qquad \ldots(27)$$

where m is the number of training samples, and g is a function defined by

$$g\left(\xi\right)=\begin{cases}0, if \;\; \xi\leq 0\\ 1, if \;\; \xi>0\end{cases} \quad \ldots(28)$$

Interestingly, the above problem is NP-complete. So, to solve this, we relax the constraint in equation (28) to:

$$f\left(\xi\right)=\sum_{i=1}^{m} g(\xi) \qquad \ldots(29)$$

The problem then reduces to:

$$L=-\sum_{i=1}^{m}\sum_{j=1}^{m}\frac{1}{2}y_i y_j \alpha_i \alpha_j x_i x_j^{T}+\sum_{j=1}^{m}\alpha_i \quad \ldots(30)$$

Subject to:

$$\sum_{i=1}^{m} \alpha_i y_i = 0 \text{ and } \alpha_i \leq C \quad \ldots(31)$$

The large value of C allows the misclassification of training examples. One can vary the value of C and analyze the performance of the model so formed on the test data.

The kernel trick

Let $\phi(x)$ be a function that transforms the feature space from the input space. The decision surface, in terms of $\phi(x)$ can be defined as $\sum_{i=1}^{\infty} w_i \phi\left(x\right) = 0$. In this case, the weights can be found by modifying the equation (14) as:

$$w = \sum_{i=1}^{m} y_i \alpha_i \phi(x_i) \quad \ldots(32)$$

And the output decision function of the output space can be expressed as:

$$\sum_{i=1}^{m} \alpha_i y_i \phi^T(x_i)\phi(x_i) + b = 0 \quad \ldots(33)$$

Which contains , called the inner product:

$$K(x, x_i) = \phi^T(x)\phi(x_i)$$

This K represents a function that finds the inner product feature space under ϕ of the two data points in the space [2]. This function must be symmetric, that is K(x, x_i) = $K(x_i, x)$, and the total volume under the surface represented by K must be constant. The kernel trick allows us to transform the input data into space where non-linearly separable data points become linearly separable, find the hyperplane, and then transform the result back to the original space.

The kernel functions provided by `sklearn.svm` are as follows:

- **linear**: $<x, x^T>$
- **polynomial**: $(\gamma < x, x^T > + r)^d$, where d is the degree of the polynomial, r is specified by `coef0` and gamma is specified by the `gamma` parameter
- **rbf**: $e^{-\gamma\left\|x-x^T\right\|^2}$, Where the value of gamma must be greater than 0.
- **sigmoid**: *tanh* $(\gamma < x, x^T > + r)$, where r is specified by `coef0` and `gamma` is specified by the `gamma` parameter

Moreover, one can define his/her kernel by defining a method and setting the kernel parameter as the name of that method.

SKLEARN.SVM.SVC

The SVC class of the sklearn.svm provides a libsvm based implementation of Support Vector Machine. *Table 7.1* shows the parameters of the `sklearn.svm.svc` method:

Parameter name	Data Type	Optional/Default value	Description
`C`	float	It is an optional parameter. The default value of this parameter is 10	This is the Regularization parameter, which must be positive.
`kernel`	string	It is an optional parameter. The default value of this parameter is 'rbf'.	This parameter specifies the kernel. It can have the following values: 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable.
`degree`	int	It is an optional parameter. The default value of this parameter is 3.	This parameter depicts the degree of the polynomial kernel function.
`gamma`	{'scale', 'auto'} or float	It is an optional parameter. The default value of this parameter is 'scale'.	This parameter depicts the kernel coefficient for 'rbf', 'poly' and 'sigmoid'.
`coef0`	float	It is an optional parameter. The default value of this parameter is 0.0.	This parameter represents the independent term in kernel function.
`tol`	float	It is an optional parameter. The default value of this parameter is 1e-3.	This parameter represents the tolerance for stopping criterion.
`max_iter`	int	It is an optional parameter. The default value of this parameter is -1.	This parameter signifies the maximum number of iterations.
`random_state`	int	It is an optional parameter. The default value of this parameter is None.	This parameter states the random state used in the pseudo random number generator.

Table 7.1: *Parameters of sklearn.svm.svc*

The `fit` method crafts the SVM. The attributes of the method are presented in *Table 7.2*:

Attribute	Description
`support`	This attribute gives the indices of the support vectors.
`support_vectors`	This attribute gives the support vectors.
`n_support`	This attribute gives the number of support vectors for each class.

Contd…

`coef_array`	This attribute gives the weights assigned to the features.
`intercept`	This attribute gives the constants in decision function.
`classes`	This attribute gives the class labels.

Table 7.2: *Attributes of sklearn.svm.svc*

The methods of the SVC class are shown in *Table 7.3*. Note that, like other classifiers, the fit method is used to craft the model, and the predict method is used to predict the test data, and other methods help us to see the support vectors, labels, and so on:

Method	Description
`decision_function`	This method finds the decision function for the data passed as the argument.
`fit`	This method fits the SVM model as per the training data.
`predict`	This method is used for the classification of the test data.
`score`	This method returns the mean accuracy.

Table 7.3: *Methods of sklearn.svm.svc*

Having seen the parameters, attributes, and methods of `sklearn.svm`, let us now move to the next section, which presents some experiments using the above methods.

Experiments

The following experiments demonstrate the application of `sklearn.svm` for classification. The first experiment classifies the Breast Cancer dataset using the linear kernel of SVM, by dividing the dataset into train and test set using `train_test_split`. The second experiment uses K-Fold validation to accomplish the same task.

Experiment 1**: Classification of Breast Cancer Dataset using SVM, Linear Kernel**.

Specifications:

- Dataset: Breast Cancer
- Classifier: SVM
- Kernel: Linear
- Split: 70% train data, 30% test data

The following steps will take you through the process of classifying the Breast Cancer dataset using the SVC of `sklearn.svc`:

Step 1: Import modules

The following modules need to be imported to classify the Breast Cancer dataset using SVM:

```
import numpy as np
from sklearn.svm import SVC
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
```

Step 2: Load data

The `load_data` function returns the data and the labels:

```
def load_data():
   Data=load_breast_cancer()
   X=Data.data
   y=Data.target
   return (X, y)
```

Step 3: Evaluate performance

The `cal_acc` function calculates the accuracy by comparing the predicted values of the labels and the values of the labels of the test data:

```
defcal_acc(y_test, y_predict):
  tp=0
  tn=0
  fp=0
  fn=0
  s=np.shape(y_test)
  for i in range (s[0]):
    o1=y_predict[i]
    y1=y_test[i]
    if(o1==1 and y1==1):
      tp+=1
    elif(o1==0 and y1==0):
      tn+=1
    elif(o1==1 and y1==0):
      fp+=1
    else:
```

```
      fn+=1
  acc=(tp+tn)/(tp+tn+fp+fn)*100
  return(acc)
```

Step 4: The model

The following code makes use of the above functions to classify the data:

```
X, y=load_data()
X_train, X_test, y_train, y_test=train_test_split(X, y, test_size=0.3,
random_state=4)
clf=SVC(kernel='linear') #gamma='auto'
clf.fit(X_train, y_train)
y_predict=clf.predict(X_test)
accuracy=cal_acc(y_test, y_predict)
```

The accuracy, in this case, comes out to be 94.7368.

Experiment 2: Classification of Breast Cancer Dataset using SVM, Linear Kernel, K-Fold

Specifications:

- Dataset: Breast Cancer
- Classifier: SVM
- Kernel: Linear
- Split: K-Fold, K=5

The `load_data` and `cal_acc` functions of Experiment 1 are used in the following code. The reader is expected to write the functions again. The code that uses K-Fold split and calculates the average accuracy is as follows:

```
kf=KFold(n_splits=5)
kf
kf.get_n_splits(X)
acc=[]
for train_i,test_i in kf.split(X):
   X_train,X_test=X[train_i],X[test_i]
   y_train,y_test=y[train_i],y[test_i]
   clf=SVC(kernel='linear') #gamma='auto'
   clf.fit(X_train, y_train)
```

```
    y_predict=clf.predict(X_test)
    accuracy=cal_acc(y_test, y_predict)
    acc.append(accuracy)
print(np.mean(acc))
```

The average accuracy, in this case, comes out to be 95.25384257102935. Note that other conditions remaining the same, the accuracy in the case of K=10 is 95.25689223057643 and in the case of K=20 is 95.09852216748767. In this case, the variation of K does not have a great impact on the performance of the system. The accuracies in the 20 folds of 20-Fold cross-validation are as follows:

```
[96.55172413793103,        86.20689655172413,        93.10344827586206,
93.10344827586206,96.55172413793103,93.10344827586206,96.55172413793103,
96.55172413793103,93.10344827586206,96.42857142857143,92.85714285714286,
100.0, 100.0, 96.42857142857143, 92.85714285714286, 96.42857142857143,
89.28571428571429, 100.0, 96.42857142857143, 96.42857142857143]
```

The accuracies in the ten folds of 10-Fold cross-validation are as follows:

```
[91.22807017543859,        92.98245614035088,        94.73684210526315,
96.49122807017544,96.49122807017544,96.49122807017544,98.24561403508771,
94.73684210526315, 94.73684210526315, 96.42857142857143]
```

Experiment 3: The following code classifies two sets of images using SVM. The required modules can be imported using the following code:

```
from matplotlib import pyplot as plt
import matplotlib.image as mpimg
import numpy as np
from sklearn import svm
from sklearn.model_selection import train_test_split
```

The images can be converted to grayscale using the following function:

```
def rgb2gray(rgb):
return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])
```

The `perf_measure` function finds the performance of the model. It takes `test_y` and `y_predicted` as parameters:

```
def perf_measure(test_y,y_predicted):
tp=0
```

```
tn=0
fp=0
fn=0
for i in range(len(test_y)):
   predicted=y_predicted[i]
   actual=test_y[i]
   if(predicted==actual):
      if(predicted==1):
         tp+=1
      else:
         tn+=1
   else:
      if(predicted==1):
         fp+=1
      else:
         fn+=1
acc=(tp+tn)/  (tp+tn+fp+fn)
sens=(tp)/(tp+fn)
spec=(tn)/(tn+fp)
return (acc,sens,spec)
```

Suppose you have 20 images of class I and 20 images of class II, the following code would help you to classify the images, assuming that you have saved the image data in `final_data`. Note that the following code uses a linear kernel:

```
y1=np.zeros((20,1))
y2=np.ones((20,1))
y=np.vstack((y1,y2))
train_X,test_X,train_y,test_y=train_test_split(final_data,    y,    test_
size=0.3)
clf = svm.SVC(kernel=”linear”)
clf.fit(train_X,train_y)
y_predicted=clf.predict(test_X)
acc,sens,spec=perf_measure(test_y,y_predicted)
```

The exercises, given at the end of this chapter, take this experiment further. The reader is expected to carry out all the steps given in the Experiment section of the exercises to get a better hold of the working of SVM.

Conclusion

The Support Vector Machines, introduced in this chapter are like a thread for the tailor, rose for a floweriest and wheat for a cook. They are essential, very essential. They are good both in terms of computation time and memory. They do not use the whole training set for classification but only its subset for the crafting of separating hyperplane. The derivation of the separating hyperplane, presented in this chapter, is for linearly separable data. However, the kernel trick allows us to separate non-linearly separable data as well. This chapter explains the idea behind the Support Vector Machines, presents the derivation of the hyperplane, discusses the cost parameter, and finally discusses the importance of kernels.

The experiments presented in this chapter will help the reader in analyzing the performance of SVM with various datasets. The reader is also expected to take note of the variation of the performance of the classifier on changing the parameters.

It may be stated here that there are certain disadvantages to these machines. They include the problem in choosing the kernel and the regularization parameters. Also, the sklearn.svm uses 5-Fold CV for the estimation of probabilities, which is expensive.

Nevertheless, the reader will be able to carry out the classification of numeric, imaging data, optimize cost and choose kernel using the concepts introduced in this chapter. The next chapter introduces some of the most important feature extraction methods and presents its implementation. These include Fast Fourier Transform, STFT, patches, HOG, and transformation techniques like PCA. The knowledge of these feature extraction methods will help the reader to develop a robust and efficient decision model.

Exercises

Multiple Choice Questions

1. Which of the following is based on the principle of maximum margin?

a. Support Vector Machine
b. Single Layer Perceptron
c. Multi-Layer Perceptron
d. None of the above

2. Which of the following helps us to classify non-linearly separable data using SVM?

a. Kernel trick
b. Train test split
c. Both
d. None of the above

3. How is SVM better than MLP?

a. It uses lesser data points for the creation of hyperplane
b. It uses the idea of Maximum Margin Classifier
c. Both
d. None of the above

4. The data points used by SVM for the creation of separating hyperplane are?

a. Support vectors
b. All the data samples
c. Cannot say
d. Depends on the situation

5. The cost parameter?

a. Helps to improve testing performance
b. May allow misclassification
c. Both
d. None of the above

6. The Support Vector Machines are used for

a. Classification
b. Regression
c. Finding outliers
d. All of the above

7. Consider the derivation of the creation of hyperplane. The problem reduces to?

a. Quadratic Optimization Problem
b. Linear Optimization
c. None of the above
d. Both

8. The data points having non-zero are?

a. Support vectors
b. Non-support vectors
c. Uber
d. None of the above

9. Why are the labels in SVM taken as 1 and -1?

a. Mathematical convenience
b. It is necessary for Lagrange's method
c. Both
d. None of the above

10. While classifying data, which of the following should be the first preference?

a. Linear kernel
b. Polynomial kernel
c. rbf
d. None of the above

Theory

1. Explain the concept of Maximum Margin Classifier.
2. Derive the separating hyperplane in the case of Support Vector Machine.
3. Explain the cost parameter in SVM.
4. What is a kernel? Which functions can be used as kernels in SVM?
5. Explain the advantages and disadvantages of Support Vector Machines.
6. State various kernels provided by sklearn.
7. Write an algorithm to classify data using SVM.

Experiment

1. Take 20 images of the face of a person and 20 images of the face of another person. The images should have the same dimensions.

a. Convert the above to grayscale.
b. Divide the data into train and test samples.
c. Apply the following to find accuracy, specificity, and sensitivity:
   i. Linear
   ii. Polynomial, degree 3
   iii. rbf
   iv. Sigmoid
d. Vary the cost parameter in (4) to find the cost at which maximum accuracy is obtained.
e. Does the change in the value of , change results in 4(b).
f. Does the change in the value of , change results in 4(b).
g. Does the change in the value of , change results in 4(c).
h. Does the change in the value of , change results in 4(c).

2. Repeat the experiment by features extracted using PCA.

Chapter 8
Decision Trees

Introduction

So far, classification algorithms like K-nearest neighbors, neural networks, and support vector machines have been discussed. These algorithms perform well. However, a major problem with these algorithms is that the assignment of a label to a test sample cannot be explained in terms of decision rules.

Duda et al. points to the applicability of algorithms, that use some distance metric, in the problems related to nominal data [3]. The algorithms like K-nearest neighbors can be used if the "closeness" amongst the samples can be defined. However, this *closeness* does not make sense in many situations. For example, in the case of nominal data, the distance between two samples may not make sense, and hence such algorithms should not be applied. In such cases, decision trees come to our rescue.

Decision trees contain decision nodes and leaf nodes. The decision nodes lead us to one of the possible branches, depending upon the answer to the question asked at the node. The leaf nodes, on the other hand, represent labels. In deciding the class of a test sample, we start with the decision node and move towards the leaf, which declares the class of the sample. The first section introduces the reader with the basics of decision trees.

The formation of these trees requires a feature to be chosen at each level. This feature is chosen using information gain or Gini index. The algorithms for choosing a feature at each level are explained in the following sections. The number of branches originating out of each node depends on the number of discrete values in the feature represented by the node. If the data is continuous, algorithms for discretization can be used. This chapter also revisits discretization.

A node can be declared as a leaf if it is pure; that is, it contains only one type of label. If the decision tree becomes too large, procedures explained in this chapter can be used to curtail the depth of the tree. In such cases, the node, to be declared as a leaf, is assigned a label, which is the same as the majority of samples at that node.

Finally, this chapter presents the implementation of decision trees using SKLearn. The reader will be able to deal with the data containing nominal values after reading this chapter. This chapter will also help the reader to explain the answer obtained using a sum of product of the rules represented by each node.

Structure

The main topics covered in this chapter are as follows:

- Introduction
- Basics
- Discretization
- Information gain and Gini index
- Implementation

Objective

After reading the chapter, the reader will be able to:

- Appreciate the importance of decision trees
- Understand the concept of information gain and the formation of a tree using the concept of information gain
- Understand Gini index
- Implement decision trees using SKLearn
- Understand the procedures to stop splitting

Basics

In Computer Science, a tree is a non-linear data structure having nodes and branches. It does not have a cycle or isolated edges/branches. A rooted tree has a root, from which other nodes originate. Other nodes, in a rooted tree, may have children, and the leaf nodes do not have any children.

A **d**ecision **t**ree (**DT**) is a classifier in the form of a tree where each node, except for the leaves, is a decision node. The leaf nodes in these trees represent labels or probability of belonging to a label.

A DT is created from the train data. Once it is created, the label of the test data is found by starting from the root and traversing till a leaf node by evaluating conditions on each decision node, using the feature set of the test data.

To understand this concept, consider a decision tree to decide the **Category** of a composition, which has **7** or **14** beats. The train data consist of two features Beats and Sections and a label called Category. If the composition has **7** Beats, we check the Sections. If the composition has **223** as the value in the Sections, the Category is **A**; otherwise, it is **B**. Likewise, in the case of a composition having **14** Beats, the Category is **C** irrespective of the value in the field Sections (*Table 8.1*):

Beats	Sections	Category
7	223	A
7	322	B
14	5234	C
14	3434	C

Table 8.1: *Category of a composition having 7 or 14 beats*

The corresponding DT is shown in *Figure 8.1*. Note that the root node and the nodes at the next level are decision nodes and the leaf nodes are labels. So, for a test sample, `{Beats=7, Sections=322}`, the first node checks whether the number of Beats in the composition is 7 or 14. Since the test sample has 7 beats, the next decision node checks the value of Sections. Since the sample has **322** as the value of Sections, it belongs to the Category **B**:

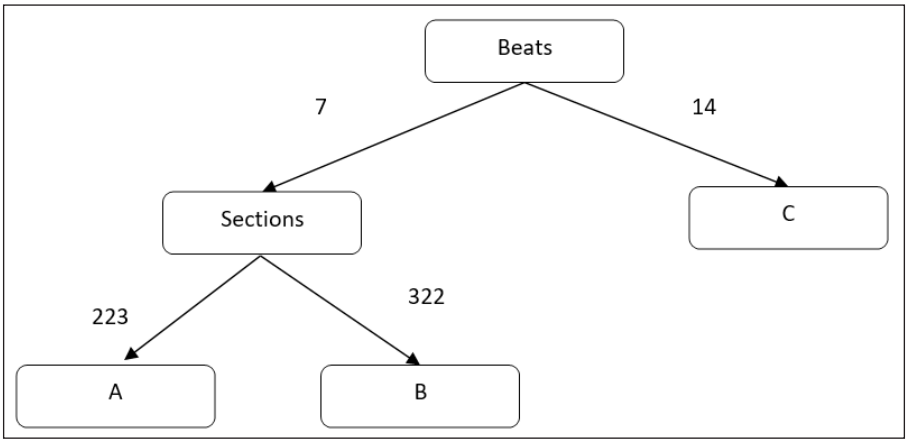


Figure 8.1: *Decision tree for Table 8.1*

The above tree has two decision nodes and three leaf nodes. The following rules can be inferred using this DT:

Rule 1: If((Beats == 7) and (Sections = 223)), thenCategory = A

Rule 2: If((Beats == 7) and (Sections = 322)), thenCategory = B

Rule 3: If((Beats == 14)), thenCategory = C

The above tree had discrete values in the columns **Beats** and **Sections**. The sections that follow explain the formation of a decision tree from data having discrete values. However, often the given data contain continuous values. The following section gives an idea of what to do in case of continuous values.

Discretization

Many algorithms require the values of data to be discrete. However, in most cases, the given data is continuous. The process of converting a continuous data into discrete values is called **discretization**. Generally, the given data is discretized into parts of equal length. Dichotomization is a type of discretization which involves splitting the measured variable at some fixed value to form two categories that can be described as *Low* and *High*. This splitting can be done along the sample median, called the median split, or along the midpoint of the measured range of the variable.

One of the algorithms used for converting the data into a discrete one is as follow:

Algorithm:

```
Discretization(Data, n)
   for each column
```

```
        min1 = minimum value in the column
        max1 = maximum value in the feature
        step = (max1 – min1)/n
        for k in range (n)
            a=min1+(step*k)
            b=min1+(step*(k+1))
            for each value in the column
                if ((value>=a) and (value<=b)):
                    value=k
                end
            end
        end
    end
```

The following code discretizes the Iris dataset having four features:

Code:

```
X=data.data[:100,:]
X=np.array(X)
y=data.target[:100]
y=np.array(y)
n=int(input('Enter the value of n \t:'))
for i in range (X.shape[1]):
    x1=X[:,i]
    max1=np.max(x1)
    min1=np.min(x1)
    step=(max1-min1)/n
    #print(max1, ' ',min1, ' ',step)
    for k in range (n):
        a=min1+(step*k)
        b=min1+(step*(k+1))
        for j in range(x.shape[0]):
        if ((X[j,i]>=a) and (X[j,i]<=b)):
            X[j,i]=k
```

```
X=pd.DataFrame(X)
print(X)
```

Output:

	0	1	2	3
0	1.0	3.0	0.0	0.0
1	1.0	2.0	0.0	0.0
2	0.0	2.0	0.0	0.0
3	0.0	2.0	0.0	0.0
4	1.0	3.0	0.0	0.0
95	2.0	2.0	3.0	3.0
96	2.0	1.0	3.0	3.0
97	3.0	1.0	4.0	3.0
98	1.0	1.0	2.0	2.0
99	2.0	1.0	3.0	3.0

```
100 rows × 4 columns
```

Having seen one of the ways to discretize data let us now come back to the creation of a decision tree.

Coming back

In creating a DT node at each level is to be selected, and each child of this node will deal with a smaller subset of the data. At each node, the process of selecting a feature at this level and dividing the data is repeated. The testing of a smaller DT will take less time as compared to a deep one. However, from all trees possible, selecting the one with minimum depth is a computationally hard problem. So, we may use the Greedy approach to find the optimal DT. This chapter discusses one such approach.

Step 1: The first step of this algorithm requires us to calculate the entropy of the target (`Entropy_orig`). This entropy is defined as follows:

$$\text{Entropy\_orig} = -\sum_{i=1}^{\text{Nmberofclasses}} p_i \times \log_2 \log_2 p_i$$

So, if the target contains two classes, then *p1* is the probability of a sample belonging to the first class, and *p2* is the probability of a sample belonging to the second class.

Step 2: This is followed by the division of the dataset on the different attributes. It can be done by calculating the entropy for each branch and adding them proportionally, to get the total entropy for the split.

Step 3: The above entropy is then subtracted from the entropy before the split (`Entropy_orig`). It results in the information gain. The attribute with the maximum information gain is then selected.

Step 4: Repeat the above steps for each of the so formed branches.

To understand the above algorithm, consider the following dataset having four features F1, F2, F3, and F4. The dataset has ten samples, and the Labels belong to **{Y, N}**:

F1	F2	F3	F4	Label
0	1	3	1	Y
1	2	2	2	N
1	2	0	1	Y
0	1	1	2	N
0	0	3	2	Y
0	0	0	2	Y
1	1	2	2	Y
1	2	2	1	N
1	1	0	1	N
0	1	1	1	Y

The number of *Y* in the Label is 6, and that of N is 4. The probabilities of *Y* and *N* are, therefore, as follows:

$$P(Y) = \frac{6}{10} = \frac{3}{5}$$

$$P(N) = \frac{4}{10} = \frac{2}{5}$$

Step 1: Find *Entropy_orig* by using the formula:

$$Entropy\_orig = -\sum_{i=1}^{Nmberofvalue} p_i \times \log_2 \log_2 p_i$$

In this case, it comes out to be:

$$Entropy\_orig = -1 \times (P(Y) \times \log\log(P(Y)) + P(N) \times \log\log(P(N))) = 0.9709$$

(Note that the base of the logarithm in the above calculation is 2).

To calculate the Information Gain by splitting the data, take one feature at a time as the root and perform the following steps.

Step 2 a: Find the probability of each discrete value in the column.

The first column consists of two discrete values: 0 and 1 (say). 5 rows of this column have 0's and 5 rows having 1's. So, the probabilities are:

$$p_1 = \frac{5}{10} = \frac{1}{2}$$

$$p_2 = \frac{5}{10} = \frac{1}{2}$$

Step 2 b: Now, consider the values of labels for each of the discrete values in the column.

For 0's in the first field, there are four '*Y*'s and one '*N*'s in Label. The corresponding entropy is:

$$E1 = -(\frac{4}{5} \times \log_2 \log_2 \frac{4}{5} + \frac{1}{5} \times \log_2 \log_2 \frac{1}{5})$$

F1	Label1
0	Y
0	N
0	Y
0	Y
0	Y

For 1's in the first field, there are two 'Y's and three 'N's in the label. The corresponding entropy is:

$$E1 = -(\frac{2}{5} \times \log_2 \log_2 \frac{2}{5} + \frac{3}{5} \times \log_2 \log_2 \frac{3}{5})$$

F1	Label1
1	N
1	Y
1	Y
1	N
1	N

Step 2 c: For each field find $\sum_{i=1}^{\text{number of values}} p_i \times E_i$.

$$= -\frac{1}{2} \times (\frac{4}{5} \times \log_2 \log_2 \frac{4}{5} + \frac{1}{5} \times \log_2 \log_2 \frac{1}{5}) - \frac{1}{2} \times (\frac{2}{5} \times \log_2 \log_2 \frac{2}{5} + \frac{3}{5} \times \log_2 \log_2 \frac{3}{5}) = 0.8464$$

Step 2 d: Find the information gain by subtracting the value obtained in 2 c) from *Entropy_orig*:

Information Gain = 0.9709 – 0.8464 = 0.1244

Likewise, the calculation of information gain for the second field will be as follows:

F2	Label1
1	Y
1	N
1	Y
1	N
1	Y

F2	Label1
2	N
2	Y
2	N

F2	Label1
0	Y
0	Y

The probabilities of individual discrete values are $p_{1=\frac{5}{10}=\frac{1}{2}} p_{2=\frac{3}{10}} p_{3=\frac{2}{10}}$

The individual *Ei's* are $E_1 == -(\frac{2}{5} \times \log_2 \log_2 \frac{2}{5} + \frac{3}{5} \times \log_2 \log_2 \frac{3}{5})$

$$E_2 == -(\frac{1}{3} \times \log_2 \log_2 \frac{1}{3} + \frac{2}{3} \times \log_2 \log_2 \frac{2}{3})$$

$$E_3 == -(\frac{2}{2} \times \log_2 \log_2 \frac{2}{2} + 0 \times \log_2 \log_2 ()) = 0$$

The value of weighted entropy is:

$$\sum_{i=1}^{number\ of\ values} p_i \times E_i = 0.8360$$

And the corresponding information gain is 0.1349. For the third feature, the information gain is 0.2168, and for the fourth feature, it is 0. The maximum information gain is for the third feature. This feature would, therefore, be the root node of the decision tree. *Figure 8.2* shows the root and row number of the data to be used by each of the branches, in the next step:

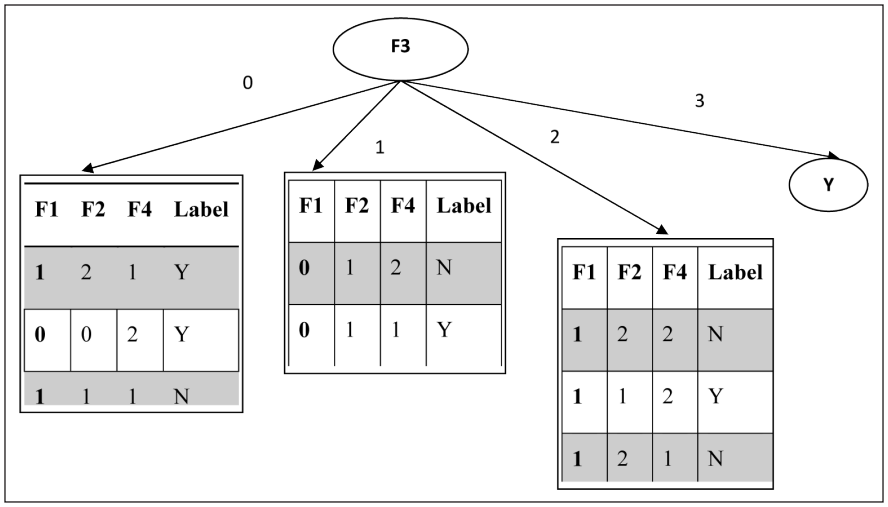


Figure 8.2: *Selection of Root of a Decision Tree in the given dataset*

Now, repeat the steps for the first, second, and third branches. While doing this, do not consider F3. Also note that for the fourth branch. The labels are Y, so there is no need to proceed in this branch.

The reader is expected to repeat the steps in the next level and verify the tree obtained is the same as that shown in *Figure 8.3:*

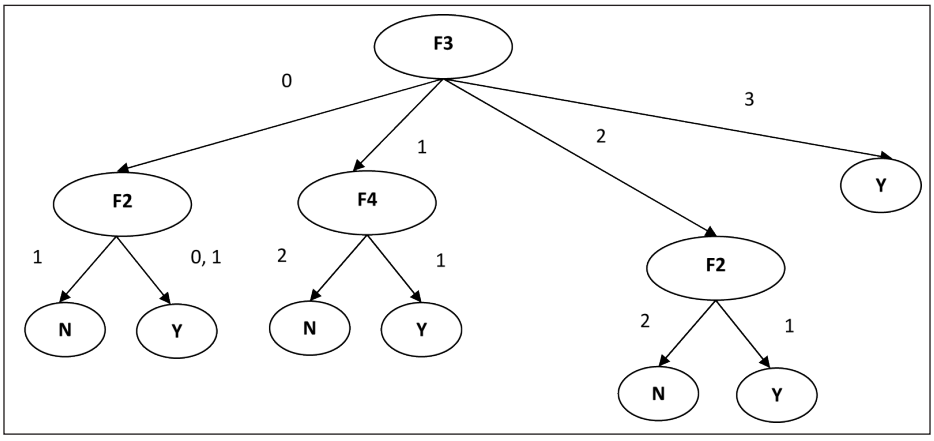


Figure 8.3: *Decision tree creation using information gain*

It may be noted that the Gini index can also be used to create a decision tree.

Containing the depth of a tree

Ideally, we continue spitting the tree until the nodes are pure. That is, all the samples at that node belong to the same class. However, in many cases, this may lead to a situation wherein the leaf corresponds to a single sample. It also leads to overfitting. On the other hand, if the number of levels in the tree is deliberately kept very low, the performance of the tree is affected. We may apply one of the following approaches to stop splitting:

- **Using the validation set**: In this approach, we continue spitting until the error in the validation set is minimized [3]. While creating a model, the data is divided into train and test set. The train data is further divided into training and validation sets. While developing the model, the validation error is considered. Once we have reached the point of minimum error in the validation set, the tree can be used for testing.
- **Thresholding**: In this technique, splitting is stopped when the reduction in impurity is less than the pre-decided threshold. In this technique, the tree is created using the whole data and not just the training data. Also, here the leaves can be at different levels. In the case of decision trees, this is considered good. The major problem with this technique is the decision to find the threshold. As per the literature, this can be done when the leaves have fewer than a certain percentage of training samples.
- **Objective function minimization**: Another method is to create an objective function, consisting of size and the sum of impurities of the leaves. As per Duda et al. [3], the following objective function can be used to accomplish this task:

  $$f = \alpha \times size + \sum_{leafnodes} i(N)$$

  We can stop spitting when the global minimum is reached.
- **Using statistical tests**: as per the literature, statistical tests like chi-square can be used to find the stopping criteria.
- **Pruning**: Another way to contain the depth of a tree is to craft a complete tree and then start from leaves. The sibling leaves, in this approach, maybe merged, if their merging creates only a very marginal increase in the impurity.

Implementation of a decision tree using sklearn

The decision tree in sklearn can be implemented using the class `sklearn.tree.DecisionTree`.

To instantiate this class, the constructor `DecisionTreeClassifier` is used. This method takes the following parameters (*Table 8.2*):

Parameter	Explanation
Criterion	This parameter specifies the function with the help of which splitting is done. This parameter can take the following values i) `Gini` and ii) `entropy`. The default value of this parameter is `Gini`.
`max_depth`	This parameter specifies the depth. Its default value is `None`, which means that the nodes are split till the leaves are pure.
`min_samples_split`	This parameter specifies the minimum number of samples required to split. The default value of this parameter is `2`.
`min_samples_leaf`	This parameter specifies the minimum number of samples at the leaf. The default value of this parameter is `1`.
`max_features`	This parameter represents the number of features to consider when looking for the best split. Its default value is `None`. The possible values of this parameter are `auto`, `sqrt` and `log2`.
`random_state`	The algorithm uses the `random_state` as the seed used by the random number generator. So giving a particular number of results is getting the same results. The default value of this parameter is `None`.

Table 8.2: *Parameters in decision tree classifier*

The attributes of the decision tree classifier have been presented in *Table 8.3*:

Attributes	Explanation
`n_classes`	This parameter represents the number of classes in cases of single output problems. In the case of multiple output problem, this parameter represents a list containing the number of classes for each output.
`n_features`	This parameter represents the number of features used for constructing the tree when the fit is performed.
`tree_`	This parameter denotes the underlying Tree object.

Table 8.3: *Attributes of a decision tree classifier*

The next section uses the above methods to implement decision trees.

Experiments

This section presents two experimenters on two different datasets. The first experiment uses the Iris dataset, and the second uses the Breast Cancer dataset. The models have been developed, and accuracies have been reported.

Experiment 1 – Iris Dataset, three classes

Step 1: The Iris dataset is loaded, and the data and labels are saved in `df` and `target`, respectively. It is followed by the creation of the train and the test data using the `train_test_split` module. The classifier is trained using the `X_train` and `y_train`.

Code:

```
data = datasets.load_iris()
df = pd.DataFrame(data.data, columns = data.feature_names)
target = data.target
X_train, X_test, y_train, y_test =train_test_split(df, target, test_
size=0.33, random_state=42)
clf = DecisionTreeClassifier(max_depth=3) #max_depthis maximum number of
levels in the tree
clf.fit(X_train, y_train)
```

The decision tree so formed is shown in *Figure 8.4*. The tree is created using GraphViz():

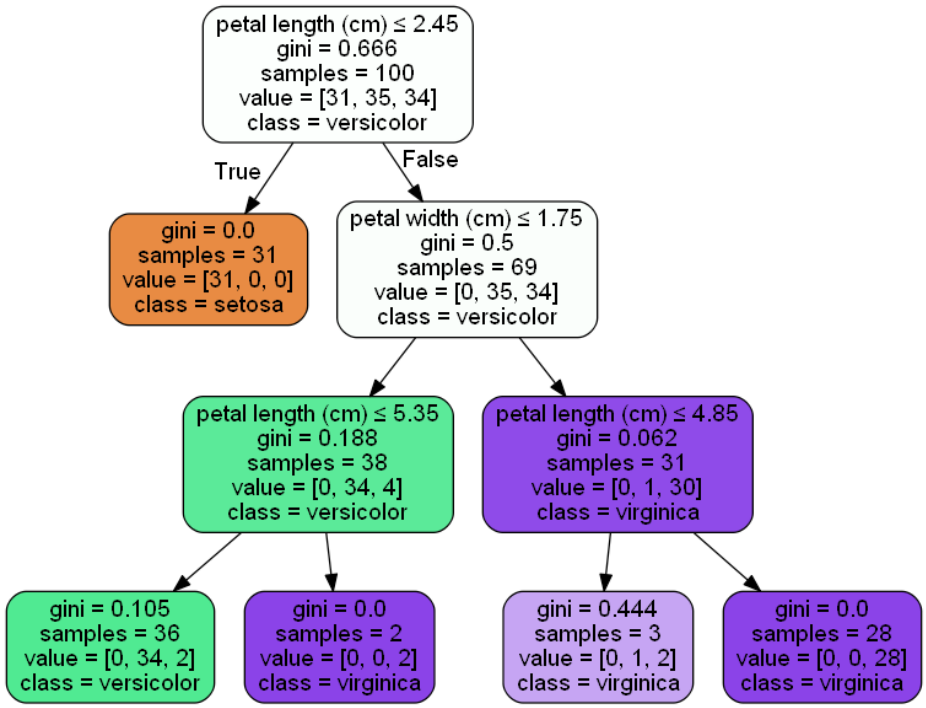


Figure 8.4: *Decision tree for the Iris dataset, using 66 percent of the data with random_state=42*

Step 2: The following code finds the accuracy of the test data using the above model. Note that a maximum accuracy of 98.00 is achieved using this classifier.

Code:

```
y_pred=clf.predict(X_test)
TP=0
TN=0
FP=0
FN=0
for i in range(X_test.shape[0]):
   if(y_test[i]==y_pred[i]):
      if(y_test[i]==1):
         TP+=1
      else:
         TN+=1
   else:
      if(y_pred[i]==1):
         FP+=1
      else:
         FN+=1
acc=(TP+TN)/(TP+TN+FP+FN)
print(acc)
```

Output:

```
0.98
```

It may be noted that the tree shown in *Figure 8.4* is not created using the entire data, but only the train data. The tree crafted using the entire data is shown in *Figure 8.5*. The tree is created using GraphViz:

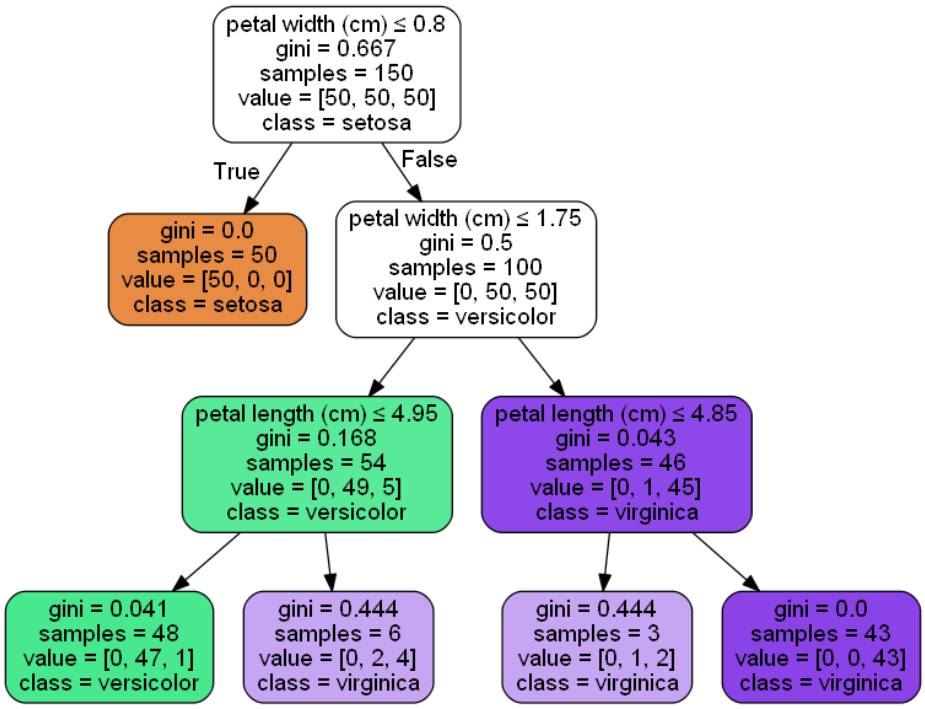


Figure 8.5: *Decision tree for the Iris dataset, using the entire data*

The reader may note the difference in trees in *Figure 8.4* and *Figure 8.5*. The idea here is to create the decision tree using only the train data and not the entire data. The second tree has been shown just to reinforce the fact that the tree formed by the entire data may be different from that formed by using the train data.

Experiment 2 – Breast Cancer dataset, two classes

Step 1: The breast cancer dataset is loaded, and the data and labels are saved in `df` and `target`, respectively. It is followed by the creation of the train and the test data using the `train_test_split` module. The classifier is trained using the `X_train` and `y_train`.

Code:

```
breast_cancer = datasets.load_breast_cancer()
df = pd.DataFrame(breast_cancer.data, columns = breast_cancer.feature_
names)
target = breast_cancer.target
```

```
X_train, X_test, y_train, y_test =train_test_split(df, target, test_
size=0.33, random_state=42)
clf = DecisionTreeClassifier(max_depth=3) #max_depth is maximum number of
levels in the tree
clf.fit(X_train, y_train)
```

The decision tree so formed is shown in *Figure 8.6*. The tree is created using GraphViz:

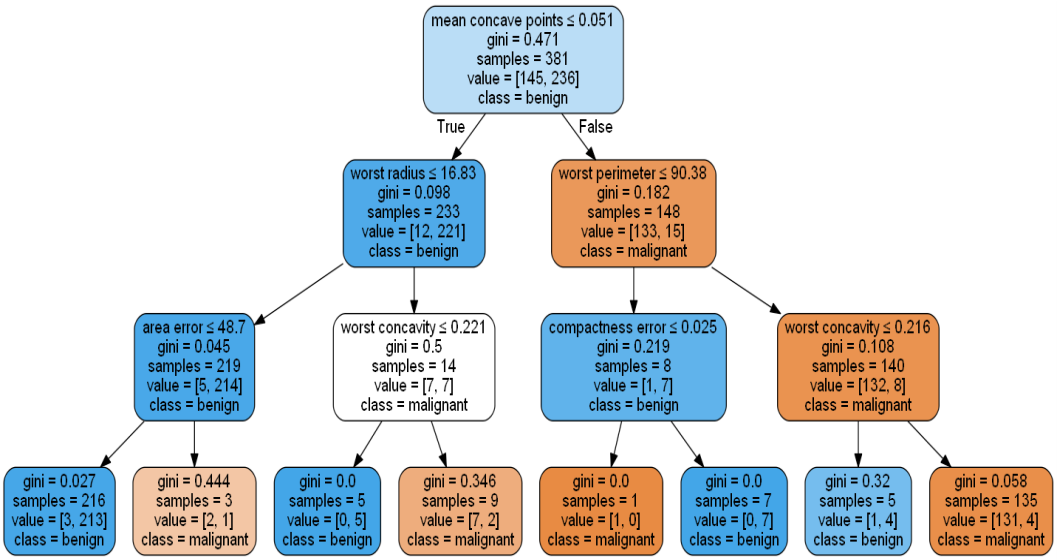


Figure 8.6: *Decision Tree for the Breast Cancer dataset, using 66 percent of the data with random_state=42*

Step 2: The following code finds the accuracy of the test data using the above model. Note that a maximum accuracy of 92.02 is achieved using this classifier.

Code:

```
y_pred=clf.predict(X_test)
TP=0
TN=0
FP=0
FN=0
for i in range(X_test.shape[0]):
   if(y_test[i]==y_pred[i]):
      if(y_test[i]==1):
```

```
            TP+=1
        else:
            TN+=1
    else:
        if(y_pred[i]==1):
            FP+=1
        else:
            FN+=1
acc=(TP+TN)/(TP+TN+FP+FN)
print(acc)
```

Output:

```
0.9202127659574468
```

It may be noted that the tree shown in *Figure 8.6* is not created using the entire data, but only the train data. The tree crafted using the entire data is shown in *Figure 8.7*. The tree is created using GraphViz ().

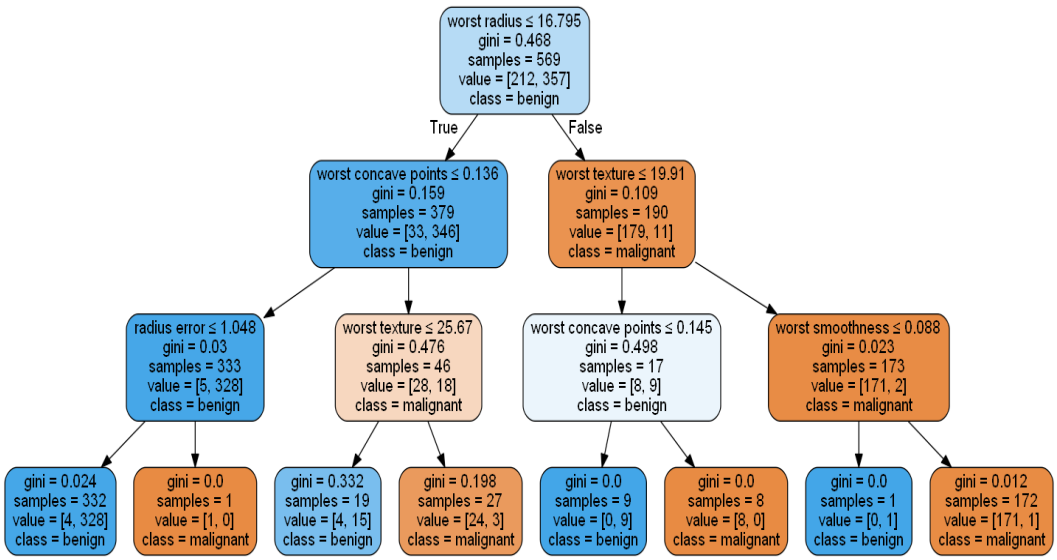


Figure 8.7: *Decision tree for the complete Breast Cancer dataset*

Again, the reader should consider the difference between the two trees (*Figure 8.6* and *Figure 8.7*) and keep in mind that the tree is formed using the train data and not the entire data.

Conclusion

This chapter introduced the decision tree classifier. The creation of a tree using the concept of information gain has been discussed in detail. The chapter also discusses the ways to prepare data for applying the decision tree algorithms. A brief discussion on the splitting has also been included in the chapter.

The reader will be able to implement the decision tree using sklearn and set parameters as per the requirement. Also, these classifiers will help the reader to handle multiple class problems as well.

The decision trees are good, can classify multi-class problems, and generally perform well. However, there are some unresolved issues. The problems are gracefully handled by the Random Forests, which makes use of many decision trees and comes under the category of ensemble methods. DT can also be used to perform regression. The next chapter introduces clustering, which is an unsupervised learning technique. Let us now hit the exercises to get hold of the concepts studied.

Exercises

Multiple Choice Questions

1. Which of the following can be used to create a decision tree?

a. Information gain
b. Gini index
c. Both
d. None of the above

2. Which of the following is an algorithm for creating a decision tree?

a. C 4.5
b. CART
c. ID3
d. All of the above

3. Which of the following can be used for nominal data?

a. K-nearest neighbors
b. Decision trees
c. Both
d. None of the above

4. Which of the following can be used to split a tree?

a. Statistical measures
b. Least reduction in purity
c. Both
d. None of the above

5. Which of the following can be done using decision trees?

a. Classification of 2-class problems
b. Classification of multi-class problems
c. Finding outliers
d. All of the above

Theory

1. What is a decision tree? How are the different types of nodes in a decision tree?
2. Explain the idea of information gain. How does it help in choosing a feature for the root node?
3. Explain the Gini index. How does it help in choosing a feature for the root node?
4. Explain the various procedures to split a tree.
5. Explain the advantages of using the decision tree as a classifier.
6. Explain the disadvantages of using the decision tree as a classifier.
7. Explain how both information gain and Gini index techniques come under the preview of Greedy algorithms?

Numerical/Programming

1. Explain the procedure to select the root node in the following dataset:

Feat1	Feat2	Feat3	Feat4
2	1	1	3
1	2	1	1
2	1	2	3
1	1	2	2
2	2	1	1
1	2	1	1
3	2	2	2
3	1	2	3
1	1	1	3
2	2	1	1

2. Perform the task using information gain.
3. Perform the above task using the Gini index. Repeat the process for the selection of nodes at the succeeding levels and complete the tree.
4. How will you curtail the depth of the tree to three?
5. Discretize the Breast Cancer dataset using any of the procedures explained in the chapter. Divide the data into the train (70% data) and test set (30% data). Create the decision tree using the train data and evaluate the performance using the test data.
6. Lung Cancer Dataset.

   This dataset is available at https://archive.ics.uci.edu/ml/datasets/Lung+Cancer. It is a multivariate dataset, having 32 instances and 56 attributes.

The data contains missing values. Hence the samples that contain missing values need to be dealt with. The reader may replace the "?" with np.nan and then remove the rows with NaN by issuing the following commands. Note that each feature contains discrete values, which is good as this saves the extra effort of converting the data into a discrete one.

A portion of the modified data is shown as follows:

1	2	3	4	5	6	7	8	9	10		47	48	49	50	51	52	53	54	55	56
0	3	3	1	0	3	1	3	1	1	...	2	2	2	2	2	2	2	1	2	2
0	3	3	2	0	3	3	3	1	1	...	2	2	2	2	2	2	2	2	1	2
0	2	3	2	1	3	3	3	1	2	...	2	2	2	2	2	2	2	2	2	2
0	3	2	1	1	3	3	3	2	2	...	2	2	2	2	2	2	2	1	2	2
0	3	3	2	0	3	3	3	1	2	...	2	2	2	2	2	2	2	2	1	2
0	3	2	1	0	3	3	3	1	2	...	2	2	2	2	1	2	2	2	1	2
0	2	2	1	0	3	1	3	3	3	...	2	2	1	2	2	2	2	1	2	2
0	3	1	1	0	3	1	3	1	1	...	2	2	2	2	2	2	2	1	2	2
0	2	3	2	0	2	2	2	1	2	...	2	2	2	1	3	2	1	1	2	2
0	2	2	0	0	3	2	3	1	1	...	2	2	2	2	2	2	2	2	2	2
0	2	3	2	0	1	2	1	1	2	...	2	2	2	2	2	1	1	2	2	1
0	2	1	1	0	1	2	2	1	2	...	2	2	2	2	2	2	2	1	2	2
0	2	2	1	1	2	3	3	1	1	...	2	2	2	2	2	1	1	1	2	2
1	3	0	NaN	1	1	2	2	1	1	...	2	2	2	2	2	2	2	1	2	1
0	3	2	2	1	2	2	2	1	1	...	2	2	2	2	2	2	2	2	2	2
0	3	2	2	0	1	1	3	1	1	...	2	2	2	2	2	2	2	1	2	2
0	2	1	1	0	2	1	3	1	1	...	2	2	2	2	2	1	1	1	2	2
0	2	0	NaN	0	2	3	3	3	2	...	2	2	2	2	2	2	2	2	1	2
0	1	2	1	0	3	3	3	1	2	...	2	2	2	2	2	1	1	2	2	1
0	2	0	NaN	1	3	3	3	1	2	...	2	2	2	2	1	2	2	1	2	2
0	3	3	2	0	2	1	3	1	1	...	2	2	1	2	2	2	2	2	1	2
0	2	3	1	1	2	2	1	1	1	...	3	3	3	3	1	3	3	2	2	1
0	2	3	1	1	1	2	1	1	1	...	2	2	2	2	2	2	2	2	2	1
0	3	3	1	0	3	3	1	1	1	...	2	2	2	2	3	2	2	2	2	1
0	2	3	2	0	1	2	2	1	2	...	2	2	2	1	3	1	2	2	1	2
0	2	2	2	0	2	1	2	1	1	...	2	2	2	2	2	2	2	1	2	1
0	2	2	1	0	2	2	2	1	1	...	3	3	2	2	3	2	2	2	2	1
0	3	2	2	0	2	2	2	1	1	...	2	2	2	3	1	2	2	2	2	2
0	2	1	1	0	2	2	1	1	1	...	2	2	3	2	2	2	2	2	2	1
0	2	3	2	1	2	2	3	1	1	...	2	2	2	2	2	2	2	1	2	2
0	2	3	1	0	2	3	3	1	1	...	2	2	2	2	2	2	2	2	2	2

The label of the dataset contains two values 0 and 1. Write the steps in creating a decision tree from the above data.

Chapter 9
Clustering

Introduction

The previous chapters discussed various supervised learning algorithms. In these algorithms, the data (X) and corresponding labels (Y) are given. The objective is to find a function that maps X to Y. The mapping is found using the training dataset. This mapping is then used to find the value of y_i for an unknown x_i. In unsupervised learning, X is given, and the aim is to extract the hidden patterns in the data. This chapter introduces clustering, which comes under unsupervised learning. To understand the meaning of clustering, consider the following examples.

Let us consider the task of segregating two types of flowers. This task is relatively easy. Even if you do not know about the two types of flowers, you can segregate them considering features like sepal length, sepal width, and so on. Note that labels are not provided to us, and we need to accomplish the task of dividing the flowers into two groups. This task, therefore, comes under the preview of unsupervised learning.

The second example is even more interesting. In the Kerala assembly elections in 2011, the two fronts polled 45.83% and 44.94% percent of the total votes polled. The difference in the percent was very less, and the results could have been different had the undecided voters had voted for the other front. It would have been possible

if there was a way out to detect the undecided voters and present your roadmap of improving economy, education, and healthcare facilities, assuming elections are fought on these issues.

If you are burdened with the responsibility of identifying undecided voters using machine learning methods, what will you do? You are only provided with the relevant data and cannot identify them by their appearance, or other features for that matter. In such cases, the unsupervised learning methods like clustering may be used to segregate the group into various clusters. Unsupervised learning does not use labels.

The creation of groups from unorganized data is referred to as clustering. Ideally, the items in a cluster should be as similar to each other as possible and should be distinct from items of other groups. This similarity can be found by any standard similarity measure like Euclidian distance, Manhattan distance, and so on. To carry out clustering, one needs to decide the measure of similarity, figure out the way of evaluating a cluster, and an algorithm for clustering. The evaluation of a cluster requires finding inter-cluster separation and intra-cluster cohesion. This chapter discusses the above issues. This chapter also addresses the question of finding the number of clusters.

One of the most important applications of clustering is segmentation. It is exciting and has been widely used in diverse applications. The technique has been used for detecting objects, identifying the regions of the brain affected by the tumor, and so on.

Structure

The main topics covered in this chapter are as follows:

- K-means
- Spectral clustering
- Agglomerative clustering
- Experiments with datasets

Objective

After reading the chapter, the reader will be able to:

- Appreciate the importance of clustering
- Understand the working of K-means
- Understand the limitations of K-means

- Understand spectral clustering
- Understand agglomerative clustering

K-means

Supervised learning techniques have been discussed in detail in the previous chapters. In supervised learning, we are provided with the training data, which contains feature vectors and the corresponding labels. The task is to predict the labels of the test data, which has only a feature vector. It is accomplished by constructing a function which takes a feature vector as its input and generates the label. The goodness of this function can be determined by the methods discussed in the second chapter of this book. Tasks like classification and regression come under the ambit of supervised learning.

Unsupervised learning, on the other hand, only deals with the feature vectors. Tasks like grouping data and some techniques of reducing dimensionality come under unsupervised learning. To understand the concept, consider the points shown in *Figure 9.1*:

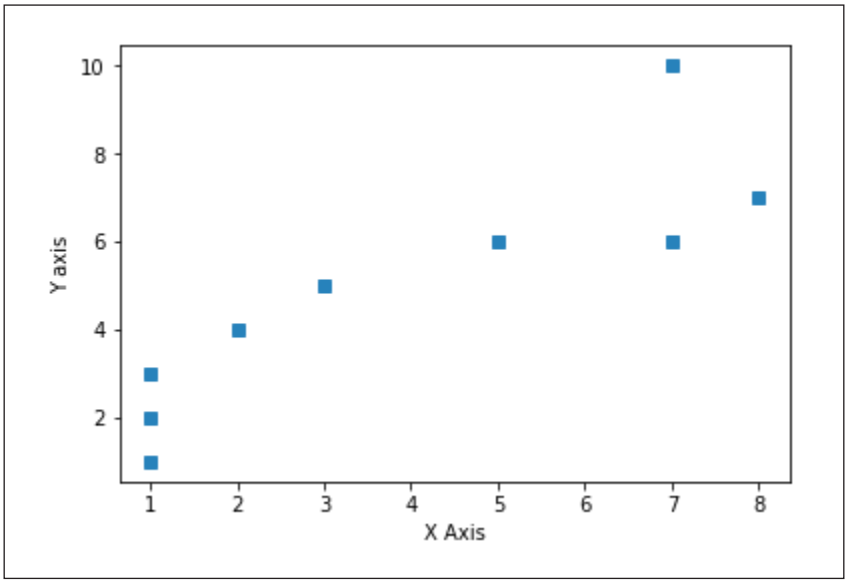


Figure 9.1: *The points shown in the figure needs to be grouped into two clusters*

The most intuitive way to create two groups out of the above points would be, as shown in *Figure 9.2*:

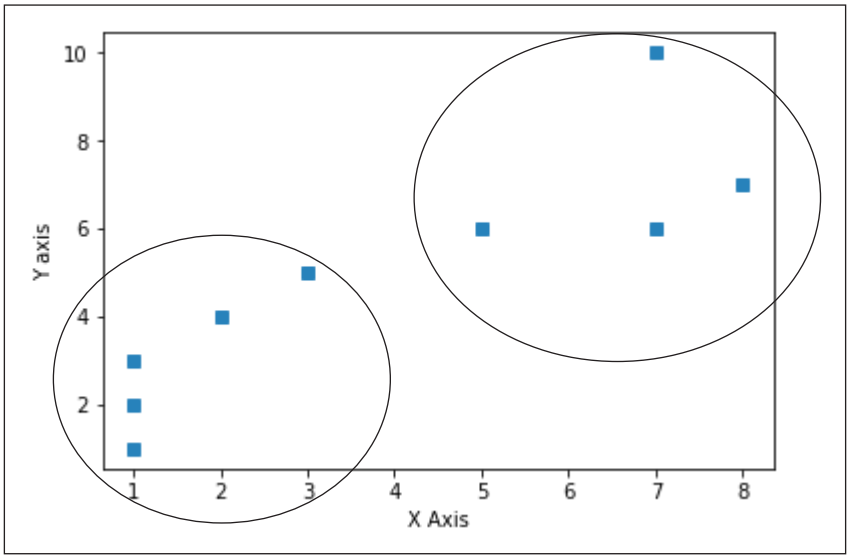


Figure 9.2: *The clusters are formed by taking the points near to each other in one group*

This section presents one of the simplest algorithms to accomplish this task. This algorithm is called K-means clustering. This algorithm requires the number of clusters as it's input. Say this number is K. The first step is to take K random samples and consider them as the centers of the K clusters. It is followed by finding the distances of each sample from each of these K centroids. The sample is put in the group from whose centroid it is nearest to. The new groups are formed, and their mean is found. These means would act as the new centroids. This process is repeated until there is no change as far as the creation of new centroids is concerned. The algorithm is as follows.

Algorithm: K Means

Input: The number of clusters, K:

1. Randomly select K random data points as the centroids.
2. Repeat the following steps until there is no change to the centroids:
    - Find the distance between each data point and all centroids
    - Allocate each data point to the closest cluster
    - Find the average of all data points that belong to each cluster and take this average as the new centroids for the clusters

The sklearn implementation of K-means has been discussed in the following sections. This algorithm works well in many cases. However, if the data is not

linearly separable, the algorithm does not work. *Figure 9.3* shows K-means applied to a non-linearly placed points:

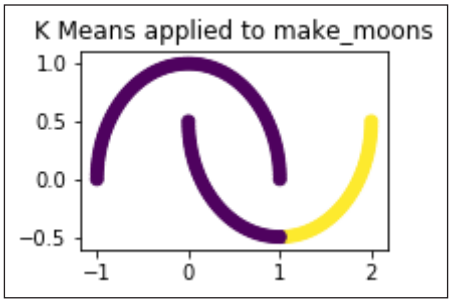


Figure 9.3: *The K-Means algorithm does not work well on such data*

In such cases, spectral clustering comes to our rescue. The next section discusses the algorithm and implementation of spectral clustering.

Spectral clustering

Clustering aims to assign data points to some groups with the intent of assigning the same group to similar data points and different groups to the different ones. The measure of similarity and the choice of algorithm to segregate the data into groups, therefore, becomes important. The algorithm that follows finds the distance between the data points, creates a graph, and finds a Laplacian matrix. It is followed by finding the Eigenvalues and Eigenvectors of this matrix to determine the groups.

This method aims to create a similarity graph, wherein each vertex represents a data point, and each edge represents the distance between them. The technique proves better than the existing techniques. The first step in the algorithm is the construction of an adjacency matrix. The adjacency matrix of a graph can be found by finding all possible distances and placing the distance between x_i and x_j at A_{ij} of the corresponding matrix. It is followed by the creation of the degree matrix, which can be found as follows:

$$D_{ii} = \sum_{j=1}^{n} A_{ij}$$

The difference between the adjacency matrix and the degree matrix is called the normal Laplacian. That is:

$$L = A - D$$

The eigenvalues and eigenvectors of the Laplacian matrix is significant. The second eigen vector of the matrix gives us the graph cut needed to separate the graph into

two components. The vector corresponding to the value here gives the direction along which the graph can be divided to split it into two components. The algorithm for spectral clustering is as follows.

Algorithm – Spectral clustering

1. Create a similarity graph
2. Find the first *k* eigenvectors of the Laplacian matrix
3. The above matrix is subjected to k-means to create *k* classes

This algorithm is computationally expensive and hence cannot be easily applied to datasets having a very large number of features. Let us now move to another type of clustering called hierarchical clustering.

Hierarchical clustering

The methods like K-means, despite being simple, are limited in the sense that they need the number of clusters as the input to the procedure. It may work if you know the data. However, in most cases, you will not know the number of groups and hence will not be able to use these algorithms. The algorithm discussed in this section does not need this information.

Hierarchical clustering can be classified as top-down or bottom-up. Let us start with the bottom-up approach. In this approach, we start with each sample as a separate cluster and use some similarity measure to find the two nearest samples. The group formed would now act as one of the samples, and the above process is repeated until a single group is created. To accomplish this task if the distance between the two samples *a* and *b* is *d*, then the distance between *(a,b)* and a new sample c can be calculated using either of the following methods:

- ***SingleLink****distance((a,b),c) = minimum(distance(a,c), distance(b,c))*
- ***ComplexLink****distance((a,b),c) = maximum(distance(a,c), distance(b,c))*
- ***AverageLink****distance((a,b),c) = average(distance(a,c), distance(b,c))*

To understand the above, consider the following example. The number of features in the Iris dataset is four. The first five samples of the data are given by the matrix *A*:

$$A = \begin{matrix} 5.1 & 3.5 & 1.4 & 0.2 \\ 4.9 & 3 & 1.4 & 0.2 \\ 4.7 & 3.2 & 1.3 & 0.2 \\ 4.6 & 3.1 & 1.5 & 0.2 \\ 5 & 3.6 & 1.4 & 0.2 \end{matrix}$$

The distance amongst samples x_i and x_j can be found by using the following formula (assuming that the number of features is k):

$$d_{i,j} = \sqrt{\sum_{k=1}^{k}(x_{i,k} - x_{j,k})^2}$$

The following matrix shows the distance between the two points. Note that W_{ij} represent the distance between the i^{th} and the j^{th} the sample:

$$W =$$

_	1	2	3	4	5
1	0.0	0.29	0.26	0.42	0.02
2	0.29	0.0	0.09	0.11	0.37
3	0.26	0.09	0.	0.06	0.26
4	0.42	0.11	0.06	0.	0.42
5	0.02	0.37	0.26	0.42	0.

Note that the sec0ond minimum distance amongst the points is 0.02 (the minimum distance is 0). It is the distance between the first and the fifth sample. Let us club together these two samples into one group and compute the distance of (1, 5) from all other points using the complex link:

- Distance of (1, 5) from Sample 2: 0.37
- Distance of (1, 5) from Sample 3: 0.26
- Distance of (1, 5) from Sample 4: 0.42

The matrix now becomes:

_	(1,5)	2	3	4
(1,5)	0.0	0.29	0.26	0.42
2	0.37	0.0	0.09	0.11
3	0.26	0.09	0.	0.06
4	0.42	0.11	0.06	0.

Repeat the above procedure with the new matrix. We can see that the second minimum distance is that between (3, 4):

	(1, 5)	2	(3, 4)
(1, 5)	0.0	0.29	0.42
2	0.37	0.0	0.11
(3, 4)	0.42	0.011	0.0

From the above matrix, it is evident that (3, 4) will now be clubbed with 2, thereby making ((3, 4), 2). In the last step, the item (1, 5) is clubbed with ((3, 4), 2), therefore creating ((1, 5), ((3, 4), 2)). The corresponding dendrogram is shown in *Figure 9.4*:

Result: ((1, 5), ((3, 4), 2))

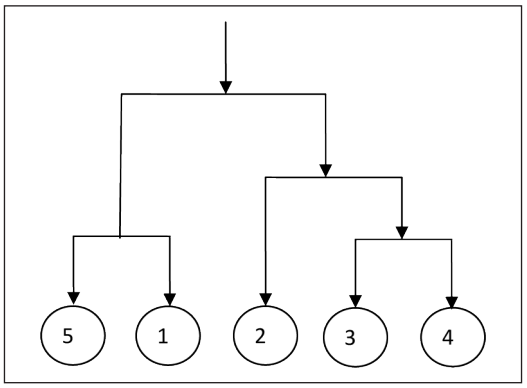


Figure 9.4: *Dendrogram – furthest distance clustering algorithm*

The above process is also referred to as the furthest distance clustering algorithm. Instead of creating the complete dendrogram, we can also stop when this distance exceeds a threshold. In this case, the algorithm becomes the complete linkage algorithm.

If the clubbing of samples is done by taking the minimum of the two distances, that is:

- *distance((a,b),c) = minimum(distance(a,c), distance(b,c))*

The algorithm is referred to as the minimum distance clustering algorithm. The steps involved in the application of this algorithm are shown as follows.

In the following matrix *W*, *Wij* represents the distance between the i^{th} and the j^{th} the sample.

_	1	2	3	4	5
1	0.0	0.29	0.26	0.42	0.02
2	0.29	0.0	0.09	0.11	0.37
3	0.26	0.09	0.	0.06	0.26
4	0.42	0.11	0.06	0.0	0.42
5	0.02	0.37	0.26	0.42	0.0

Note that the second minimum distance amongst the points is 0.02 (the minimum distance is 0). It is the distance between the first and the fifth sample. Let us club

together these two samples into one group and compute the distance of (1,5) from all other points:

- Distance of (1,5) from Sample 2: 0.29
- Distance of (1,5) from Sample 3: 0.26
- Distance of (1,5) from Sample 4: 0.42

The matrix now becomes:

_	(1,5)	2	3	4
(1,5)	0.0	0.29	0.26	0.42
2	0.29	0.0	0.09	0.11
3	0.26	0.09	0.0	0.06
4	0.42	0.11	0.06	0.0

Repeat the above procedure with the new matrix. We can see that the second minimum distance is that between (3, 4):

	(1, 5)	2	(3, 4)
(1, 5)	0.0	0.29	0.26
2	0.37	0.0	0.09
(3, 4)	0.26	0.09	0.0

From the above matrix, it is evident that (3, 4) will now be clubbed with 2, thereby making ((3, 4), 2). In the last step, the item (1, 5) is clubbed with ((3, 4), 2), therefore creating ((1, 5), ((3, 4), 2)). The corresponding dendrogram is shown in *Figure 9.5*.

Result: ((1, 5), ((3, 4), 2))

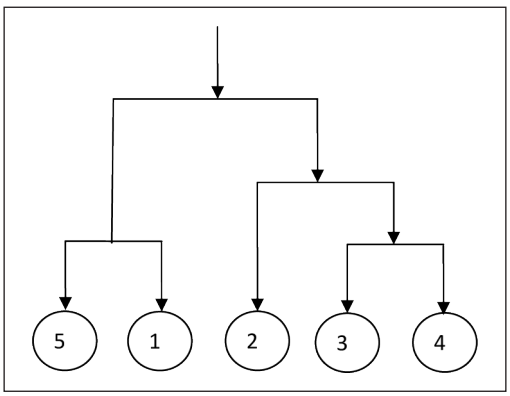


Figure 9.5: *Dendrogram – A minimum distance clustering algorithm*

The grouping of samples can also be done by taking the average of the two distances, that is:

- *distance((a,b),c) = average(distance(a,c),distance(b,c))*

The algorithm is referred to as the average distance clustering algorithm. The reader is expected to create a dendrogram using the above formula for the matrix *A*.

Implementation

This section consists of a few experiments. For each of the four methods:

- K-means
- Spectral clustering
- Agglomerative clustering
- DBSCAN

Three experiments have been designed. These experiments have been designed to ascertain the goodness of a method on data having three clusters with the same variances, different variances, and different numbers of samples. The first nine experiments have been presented, and the last three experiments are left for the reader.

K-means

The theory of K-means has already been discussed in the second section. This section presents the implementation of the algorithm using the methods provided by the `sklearn`. Note that the data is generated using `make_blobs`, which takes the number of samples and the random state as the input. The clustering is carried out using the `fit_predict` method of `KMeans`. The predicted values are stored in `y_predicted`.

In the first experiment, 200 samples are generated using `make_blobs`. The data contains three clusters having an almost equal number of samples. Note that since no input is provided for setting the variance of the data of the three clusters, all of them will have the same variance.

Experiment 1

1. Importing the modules:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
```

2. Generating data for clustering:

```
n_samples = 200
random_state = 10
X, y = make_blobs(n_samples=n_samples, random_state=random_state)
```

3. Applying `sklearn.KMeans` to predict the clusters:

```
y_predicted = KMeans(n_clusters=3, random_state=random_state).fit_
predict(X)
plt.scatter(X[:, 0], X[:, 1], c=y_predicted)
plt.title("K Means Clustering I")
plt.show()
```

The output is shown in *Figure 9.6*:

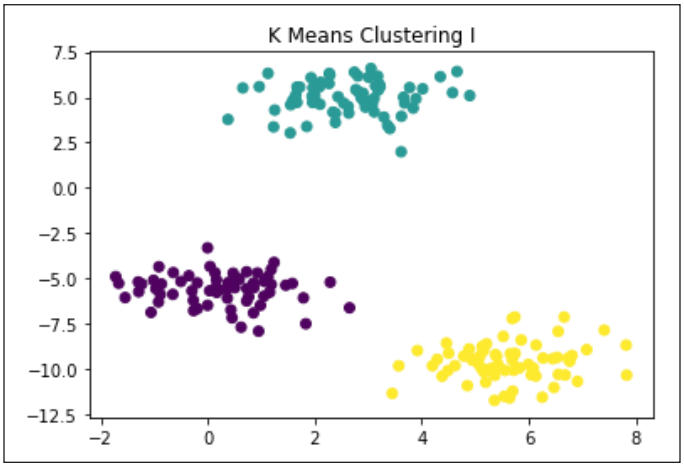


Figure 9.6: *Applying sklearn.KMeans for clustering on the data generated by make_blobs*

In the second experiment, 200 samples are generated using `make_blobs`. The data contains three clusters having an almost equal number of samples. Note that input is provided for setting the variance of the data of the three clusters as 1, 0.5, and 3.0.

Experiment 2

Step 1 and *Step 2* of *Experiment 1* to be used on as-it-is basis:

```
X_1, y_1 = make_blobs(n_samples=n_samples, cluster_std=[1, 0.5, 3.0],
random_state=random_state)
y_predicted = KMeans(n_clusters=3, random_state=random_state).fit_
predict(X_1)
```

```
plt.scatter(X_1[:, 0], X_1[:, 1], c=y_predicted)
plt.title("K Means II")
plt.show()
```

The output is shown in *Figure 9.7*:

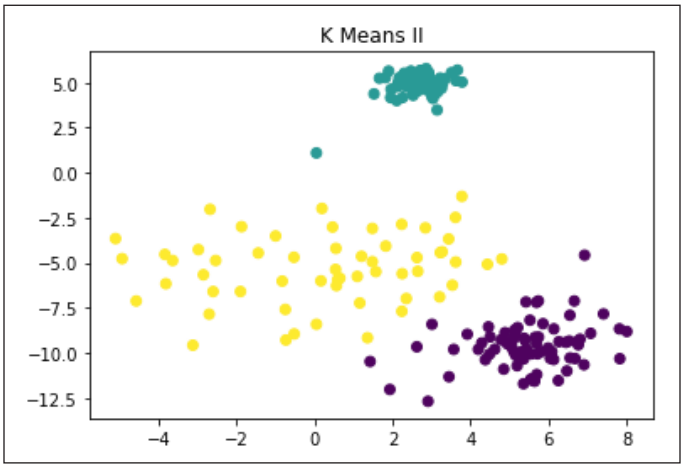


Figure 9.7: *Applying sklearn for clustering on the data generated by make_blobs. The three clusters have different values of variances*

In the third experiment, 200 samples are generated using `make_blobs`. The data contains three clusters having an odd number of samples. Note that since no input is provided for setting the variance of the data, all the three groups will have the same variance.

Experiment 3

Step 1 and *Step 2* of *Experiment 1* to be used on as-it-is basis:

```
X_not_balanced = np.vstack((X[y == 0][:500], X[y == 1][:200], X[y == 2]
[:10]))
y_predicted = KMeans(n_clusters=3,random_state=random_state).fit_
predict(X_not_balanced)
plt.scatter(X_not_balanced[:, 0], X_not_balanced[:, 1], c=y_pred)
plt.title("Blobs having differnt number of elements")
plt.show()
```

The output is shown in *Figure 9.8*:

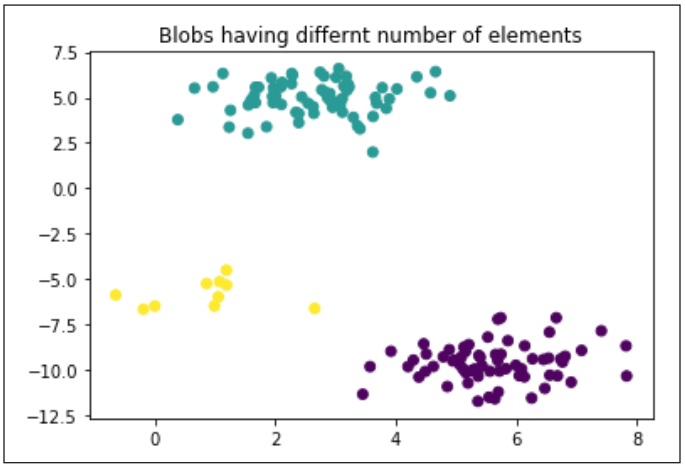


Figure 9.8: *Applying sklearn.KMeans for clustering on the data generated by make_blobs. The three clusters have a different number of samples*

The above experiments bring forth the following points:

- `KMeans` works well for balanced data wherein each block has the same variance
- The performance of `KMeans` is fairly good even in balanced data wherein each block has a different variance, provided that this value is not too large
- The algorithm works well even for data in which different clusters have a varying number of samples

However, the algorithm does not produce good results for clusters having non-linear shapes. The reader is expected to refer to the exercises given at the end of the chapter to appreciate this point.

Spectral clustering

In `sklearn`, spectral clustering finds the affinity matrix, which represents the similarity amongst the samples. The Laplacian of this matrix is quite informative, and the second eigenvector of this Laplacian is used to find the cut which divides the graph corresponding to the affinity matrix. It is followed by the application of K-means to the groups so formed. It may be stated that the implementation provided by `sklearn` **"`amgsolver`"** is used for solving the eigenvalue problem, which makes it rather efficient.

The spectral clustering finds the normalized cut in the similarity graph, which works appealingly well in the case of images wherein weights of the edges are computed as the gradient of an image. As per the official documentation in `sklearn` the similarity is given by the following formula:

*similarity = np.exp(-beta * distance / distance.std())*

[https://scikit-learn.org/stable/modules/clustering.html]. The following code uses the `sklearn` implementation of the algorithm.

In the fourth experiment, 200 samples are generated using `make_blobs`. The data contains three clusters having an almost equal number of samples. Note that since no input is provided for setting the variance of the data of the three clusters, all of them will have the same variance.

Experiment 4

1. Importing the modules:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import SpectralClustering
from sklearn.datasets import make_blobs
```

2. Generating data for clustering

```
n_samples = 200
random_state = 10
X, y = make_blobs(n_samples=n_samples, random_state=random_state)
```

3. Applying `sklearn.SpectralClustering` to predict the clusters:

```
y_predicted = SpectralClustering(n_clusters=3, random_state=random_
state).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=y_predicted)
plt.title("Spectral Clustering")
plt.show()
```

The output is shown in *Figure 9.9*:

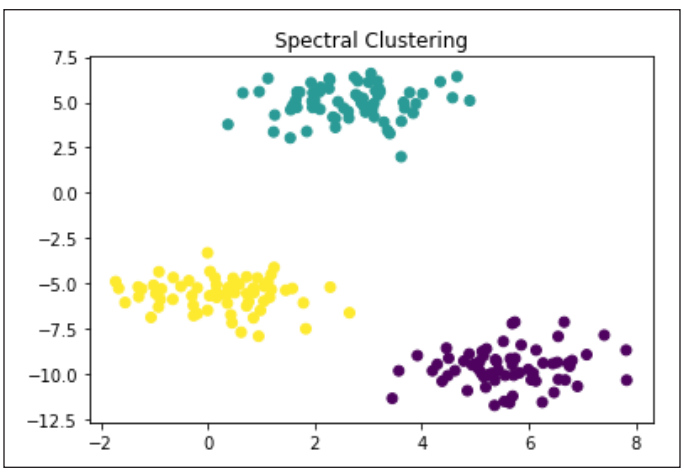


Figure 9.9: *Applying sklearn.SpectralClustering for clustering data generated by make_blobs*

In the fifth experiment, 200 samples are generated using `make_blobs`. The data contains three clusters having an almost equal number of samples. Note that input is provided for setting the variance of the data as of the three clusters as 1, 0.5, and 3.0.

Experiment 5

Step 1 and *Step 2* of *Experiment 4* to be used on as-it-is basis:

```
X_1, y_1 = make_blobs(n_samples=n_samples, cluster_std=[1, 0.5, 3.0], random_state=random_state)
y_predicted = SpectralClustering(n_clusters=3, random_state=random_state).fit_predict(X_1)
plt.scatter(X_1[:, 0], X_1[:, 1], c=y_predicted)
plt.title("Spectral Clustering II")
plt.show()
```

The output is shown in *Figure 9.10*:

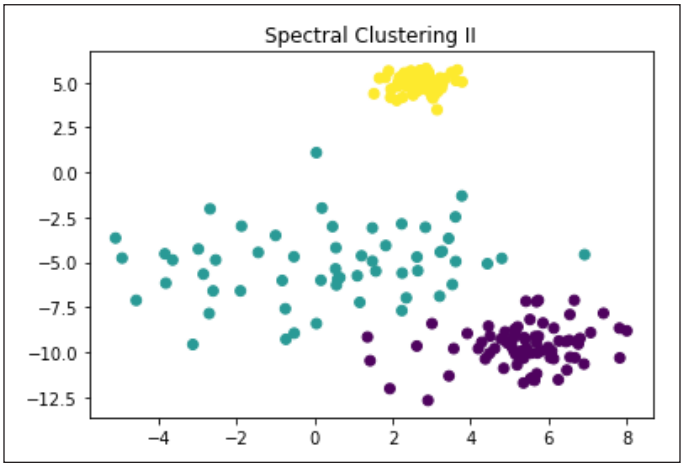


Figure 9.10: *Applying sklearn.SpectralClustering for clustering data generated by make_blobs. The three clusters have different values of variance*

In the sixth experiment, 200 samples are generated using `make_blobs`. The data contains three clusters having an odd number of samples. Note that since no input is provided for setting the variance of the data, all the three groups will have the same variance.

Experiment 6

Step 1 and *Step 2* of *Experiment 4* to be used on as-it-is basis:

```
X_not_balanced = np.vstack((X[y == 0][:500], X[y == 1][:200], X[y == 2]
[:10]))
y_predicted= SpectralClustering(n_clusters=3,random_state=random_state).
fit_predict(X_not_balanced)
plt.scatter(X_not_balanced[:, 0], X_not_balanced[:, 1], c=y_predicted)
plt.title("Blobs having unequal number of elements in each cluster")
plt.show()
```

The output is shown in *Figure 9.11*:

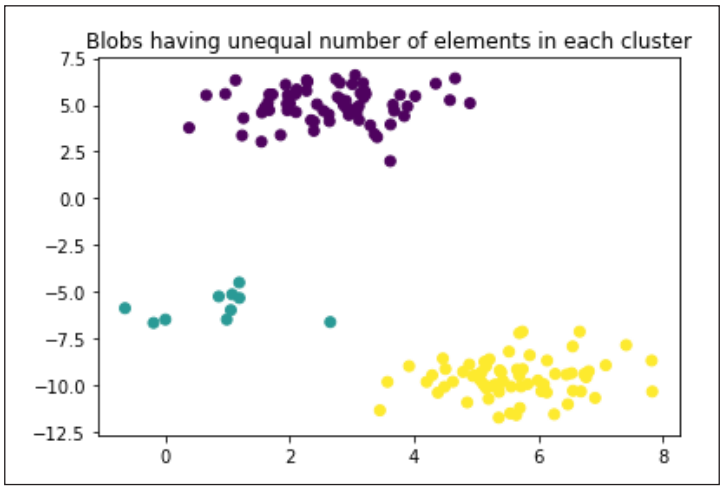


Figure 9.11: *Applying sklearn.SpectralClustering for clustering on the data generated by make_blobs. The three clusters have a different number of samples*

Experiments 4, 5, and 6 bring forth the following points:

- Spectral Clusterin works well for balanced data wherein each block has the same variance.
- The performance of Spectral Clusterin is fairly good even in balanced data wherein each block has a different variance, provided this value is not very large.
- The algorithm works well even for data in which different clusters have a varying number of samples.
- The algorithm also produces good results for clusters having non-linear shapes. The reader is expected to refer to the exercises given at the end of the chapter to appreciate this point.

Agglomerative clustering

As explained in the previous section, the agglomerative clustering is a type of clustering which crafts nested clusters by repeatedly combining or splitting the samples. The agglomerative clustering of `sklearn` uses the bottom-up approach in which the samples are merged. This merging uses any one of the following criteria:

- WARD
- Maximum
- Average
- Single

In the WARD criterion, the sum of squares within all clusters is minimized. The rest of the algorithms have been explained in the previous section. The following code uses the sklearn implementation of the algorithm.

In the seventh experiment, 200 samples are generated using `make_blobs`. The data contains three clusters having an almost equal number of samples. Note that since no input is provided for setting the variance of the data of the three clusters, all of them will have the same variance.

Experiment 7

1. Importing the modules:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets import make_blobs
```

2. Generating data for clustering:

```
n_samples = 200
random_state = 10
X, y = make_blobs(n_samples=n_samples, random_state=random_state)
```

3. Applying `sklearn.AgglomerativeClustering` to predict the clusters

```
y_predicted = AgglomerativeClustering(n_clusters=3).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=y_predicted)
plt.title("Agglomerative Clustering")
plt.show()
```

The output is shown in *Figure 9.12*:

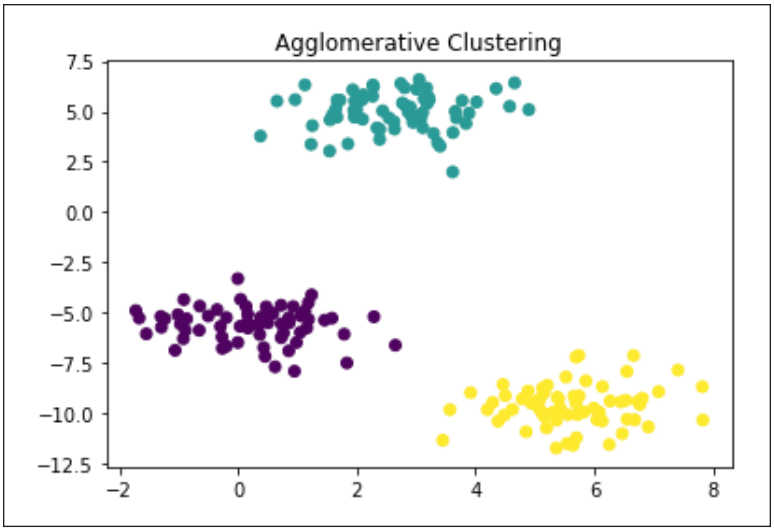


Figure 9.12: *Applying sklearn.AgglomerativeClustering for clustering on the data generated by make_blobs*

In the eighth experiment, 200 samples are generated using `make_blobs`. The data contains three clusters having an almost equal number of samples. Note that the input is provided for setting the variance of the data as of the three clusters as 1, 0.5, and 3.0.

Experiment 8

Step 1 and *Step 2* of *Experiment 7* to be used on as-it-is basis

```
X_1, y_1 = make_blobs(n_samples=n_samples, cluster_std=[1,0.5,3.0], random_state=random_state)
y_predicted = AgglomerativeClustering(n_clusters=3).fit_predict(X_1)
plt.scatter(X_1[:, 0], X_1[:, 1], c=y_predicted)
plt.title("Agglomerative Clustering II")
plt.show()
```

The output is shown in *Figure 9.13*:

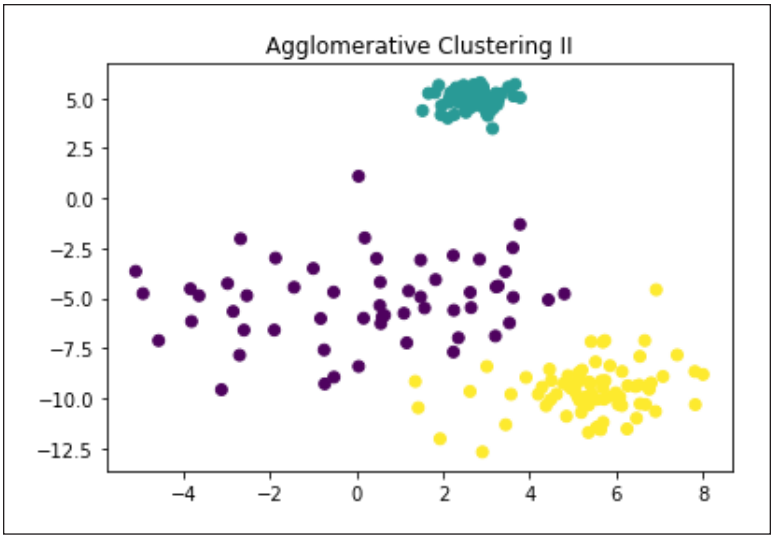


Figure 9.13: *Applying sklearn.AgglomerativeClustering for clustering on the data generated by make_blobs. The three clusters have different values of variance.*

In the ninth experiment, 200 samples are generated using `make_blobs`. The data contains three clusters having an odd number of samples. Note that since no input is provided for setting the variance of the data, all the three groups will have the same variance.

Experiment 9

Step 1 and *Step 2* of *Experiment 1* to be used on as-it-is basis:

```
X_not_balanced = np.vstack((X[y == 0][:500], X[y == 1][:200], X[y == 2]
[:10]))
y_predicted = AgglomerativeClustering(n_clusters=3).fit_predict(X_not_
balanced)
plt.scatter(X_not_balanced[:, 0], X_not_balanced[:, 1], c=y_predicted)
plt.title("Blobs having differnt number of elements")
plt.show()
```

The output is shown in *Figure 9.14*:

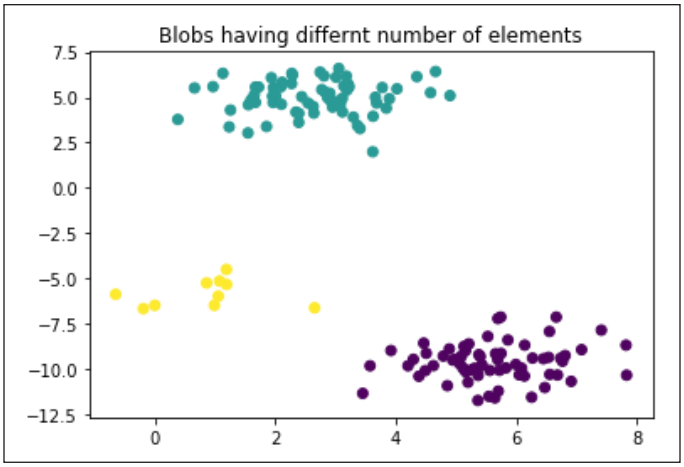


Figure 9.14: *Applying sklearn.AgglomerativeClustering for clustering on the data generated by make_blobs. The three clusters have a different number of samples.*

The above discussion brings forth the following points:

- `AgglomerativeClustering` works well for balanced data wherein each block has the same variance.
- The performance of `AgglomerativeClustering` is fairly good even in balanced data wherein each block has a different variance, provided this value is not very large.
- The algorithm works well even for data in which different clusters have a varying number of samples.

DBSCAN

In this algorithm, the clusters are formed by identifying the areas with low density. This concept makes it computationally expensive, but it can find clusters with any shape. This algorithm takes two parameters:

- minimum samples
- eps

As per the official documentation:

"We define a core sample as being a sample in the dataset such that there exist min_samples and other samples within a distance of eps, which are defined as neighbors of the core sample. This tells us that the core sample is in a dense area of the vector space.

A cluster is a set of core samples that can be built by recursively taking a core sample, finding all of its neighbors that are core samples, finding all of their neighbors that are core samples, and so on. A cluster also has a set of non-core samples, which are samples that are neighbors of a core sample in the cluster but are not themselves core samples. Intuitively, these samples are on the fringes of a cluster."

[2.3. Clustering — scikit-learn 0.22.1 documentation].

The `DBSCAN(eps=<value>, min_samples=<value>)` method is used for accomplishing the task using this algorithm. The reader is expected to carry out experiments similar to experiments 1, 2, and 3 using the DBSCAN function of `sklearn`.

Conclusion

As per Shimon Ullman et al., "Clustering is the organization of unlabeled data into similarity groups called clusters.[4]" A clustering algorithm aims to come up with clusters that are similar to each other and dissimilar to elements in the other clusters. The technique finds its applications in diverse fields. It was first used to analyze the location of cholera deaths on a map. It not only helped to find that these locations had dirty wells but also helped the doctors to control the spread of disease. You can use this technique to suggest the causes of various problems by identifying the locations and analyzing the data about these locations. This technique is useful for the segmentation of images, finding patterns, finding target groups for analysis, and so on.

To form clusters, we need to decide the similarity measure, the way to evaluate clusters, and the algorithm for clustering. Euclidean and Manhattan distances, which special cases of Minslowski distance, have already been discussed in the book. To evaluate clusters, the sum of squared errors can be used. This chapter discusses two types of clustering:

- Hierarchical
- Partitional

The hierarchical clustering can be segregated as divisive and agglomerative. The partitional clustering can be segregated as centroid based and model-based. This chapter discusses the above algorithms and compares them.

The next chapter discusses various feature extraction techniques. These would help the reader not only to improve the performance but also to analyze the results. Let us now hit the exercises.

Exercises

Multiple Choice Questions

1. Clustering comes under the ambit of?

a. Supervised Learning
b. Unsupervised Learning
c. Semi-supervised Learning
d. None of the above

2. Which of the following initially identifies K centroids and then allocates samples to each cluster based on its similarity from it?

a. K-means
b. Spectral clustering
c. Both
d. None of the above

3. Which of the following uses Laplacian in finding clusters?

a. K-means
b. Spectral clustering
c. Both
d. None of the above

4. In the above question, what is done after finding the Laplacian?

a. Finding eigenvalues Aand vectors
b. Finding the shortest distance path in the graph
c. Both
d. None of the above

5. Which of the following is not a type of hierarchical clustering?

a. Top-down
b. Bottom-up
c. Depth First Search
d. None of the above

6. Which of the following term is generally not associated with hierarchical clustering?

a. Single Link
b. Complex Link
c. Average Link
d. All the terms are associated with hierarchical clustering

7. Which of the following can be used if the shapes of the two clusters are non-linearly embedded?

a. K-means
b. Spectral clustering
c. Both
d. None of the above

8. Which of the following can be used to find the number of clusters?
   a. K_means b. W c. Both d. None of the above
9. Which of the following is sensitive to the number of clusters?
   a. K-means b. Spectral clustering
   c. Both d. None of the above
10. Which of the following can be used for segmentation?
   a. Clustering b. Classification
   c. Both d. None of the above

Theory

1. What is clustering? What are the types of clustering?
2. Explain the K-means algorithm. Write the pseudo-code for the same. What are the limitations of this algorithm?
3. Explain the spectral clustering algorithm. Write the pseudo-code for the same. What are the limitations of this algorithm?
4. Explain the hierarchical clustering algorithm. Write the pseudo-code for the same. What are the limitations of this algorithm?
5. Explain the process of finding the number of clusters for clustering.
6. Write some applications of clustering.

Numerical

1. The following data contains six samples and four features. Write the steps of finding clusters using hierarchical clustering for this data:

x1	12	4	3	2
x2	2	23	2	1
x3	1	12	7	3
x4	3	9	4	3
x5	1	6	7	2
x6	8	9	4	5

2. In question number 1, draw dendrogram for each of the following technique:
   a. Single link b. Average link c. Complex link
3. For the above question, apply K-means for K=2.
4. For the above question, apply K-means for K=3.
5. Create a Laplacian matrix for the matrix provided in Question 1.

Programming

Consider the following code:

Step 1: Include the modules:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import AgglomerativeClustering
from sklearn.datasets import make_moons
```

Step 2: Apply `AgglomerativeClustering`:

```
n= 200
random_state = 10
X, y = make_moons(n_samples=n, random_state=random_state)
y_pred = AgglomerativeClustering(n_clusters=2).fit_predict(X)
plt.scatter(X[:, 0], X[:, 1], c=y_pred)
plt.title("Agglomerative Clustering applied to make_moons")
plt.show()
```

The output is shown in *Figure 9.15*:

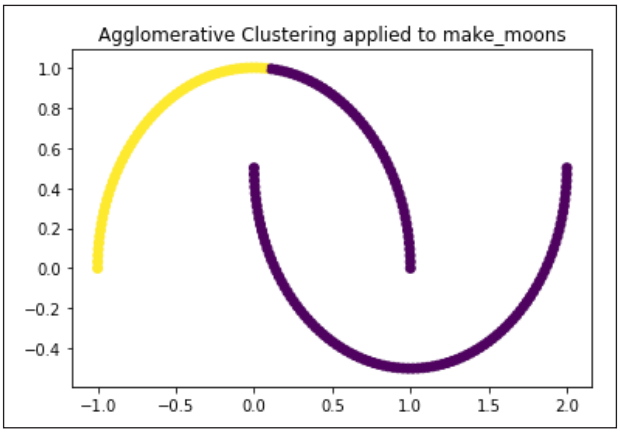


Figure 9.15: *Applying Agglomerative Clustering to make_moons*

Based on the above, perform the following task:

a. Apply K-means to the data
b. Apply spectral clustering to the data
c. Analyze the above results and state why spectral clustering works wonders here
d. Apply various methods in agglomerative clustering (like a ward, single link, complex link, and average link) to the above data and analyze the output

Chapter 10
Feature Extraction

Introduction

So far, we have learned the preprocessing of data, selection of features, and basic machine learning tasks like classification and regression. Let us shift our focus to improving the performance of the model. It can be done by a) varying the parameters of the classifier, b) selecting the relevant features, or c) taking the data into an altogether different dimension(s), in which data becomes better for the required tasks. To understand this, consider a boy called Hari, who is a good poet but writes technical books as a profession. It is because the latter pays more in this world as compared to the former. If he is transferred to another dimension in which poetry pays more than academic writing or teaching, he may peruse poetry as his career and academic writing as his hobby as he is good in poetry. The absence of any pressure may improve his academic writing skills. So Hari, in the transformed dimension, is a more useful person for the society. Alas, we cannot shift him to new dimensions, but at least we can shift the data in some other dimensions to make it more useful.

If the number of features in the given dataset is large, then the machine learning model may suffer from overfitting. To handle this problem, feature selection and feature extraction can be used. Feature selection places the features in order of their relevance to the labels. The second chapter of this book introduced some of the most important feature selection methods. This problem can also be handled using

feature extraction, which extracts features from the given data and hence takes the given data to a new set of dimensions where the data becomes more suitable for a given machine learning task. It may also result in improved performance with a reduced number of features. The reduction in the number of features will lead to faster training and may also result in improved visualization.

Feature extraction aims to create new features from the existing ones. This chapter introduces the reader to some of the most important feature extraction methods. These include:

- The frequency-based methods
- Finding patches in an image
- Histogram of orientated gradients
- Principal component analysis

These feature extraction methods have been explained and implemented in the sections that follow. These methods can be used for text data, images, and even sound. The reader is expected to use these methods and apply classification methods learned in the previous chapters to analyze the effect of these methods.

Structure

The main topics covered in this chapter are as follows:

- Introduction
- Fourier Transform and Short Term Fourier Transform
- Patches
- Histogram of orientations
- Principal Component Analysis
- Experiments with datasets

Objective

After reading the chapter, the reader will be able to:

- Appreciate the importance of feature extraction
- Understand the use of Fourier Transform
- Understand the shortcomings of Fourier Transform
- Understand feature extraction using patches

- Understand the implementation of the histogram of oriented gradients
- Use principal component analysis

Fourier Transform

Fourier analysis helps us to express a given function as a sum of periodic signals. The Fourier Transform of a signal takes it from the time domain to the frequency domain. Moreover, the inverse Fourier Transform helps us to recover the original signal from the components obtained by the Fourier Transform of a signal. The corresponding discrete counterpart is referred to as the **Discrete Fourier Transform** (**DFT**). However, the complexity of the most common method of finding the transform is $O(n^2)$. The **Fast Fourier Transform** (**FFT**) is an efficient implementation of DFT. The input to FFT is in the time domain, and the output is in the frequency domain. The FFT finds its applications in numerous domains.

The `np.fft` uses the following implementation of FFT:

$$A_K = \sum_{n=0}^{n-1} a_m \exp\left(-2\pi i\left(\frac{mk}{n}\right)\right)$$

Input is $\{a_0, a_1, \ldots, a_m\}$ and k varies from 0 to $(n-1)$.

The output of `np.fft` follows a standard order. The first half of the output contains positive frequencies, and the second half from $(n+1)/2$ to the last term contains negative frequencies. You can find the modulus of the output using `np.mod` and the phase spectrum by `np.angle`. The inverse DFT is given by the expression:

$$a_m = \frac{1}{n}\sum_{k=0}^{n-1} A_k \exp\left(2\pi i\left(\frac{mk}{n}\right)\right)$$

Where m varies from 0 to $(n-1)$. The parameters of the `np.fft` function are as follows (*Table 10.1*):

Parameter	Type	Explanation
`a`	`array_like`	This parameter represents the input array.
`n`	`int`	It is an optional parameter. It represents the length of the transformed axis of the output. The input is truncated or padded with zeros if the value of n is not the same as the input.
`axis`	`int`	It is an optional parameter. It represents the axis over which to compute the FFT

Table 10.1: *The parameters of np.fft*

The attributes of the function are as follows (*Table 10.2*):

Attribute	Type	Explanation
`out`	`complex`	It is the truncated or zero-padded input.

Table 10.2: *The attributes of np.fft*

This function raises the `IndexError` exception. One can find out the most prominent frequency (or frequencies) by using the `fft`. The following steps take the reader through the implementation of `fft` using `np.fft`:

Step 1: Import the following modules:

```
import matplotlib.pyplot as plt
import numpy as np
from scipy import signal
```

Step 2: Ask the user to enter the amplitude and frequency of and plot the signal:

```
A=float(input('Enter amplitude\t:'))
f=int(input('Enter frequency\t:'))
t=np.linspace(-np.pi, np.pi,256)
y=A*np.sin(2*np.pi*f*t)
plt.plot(t,y)
plt.show()
```

Output: The output of the above code is as follows (*Figure 10.1*):

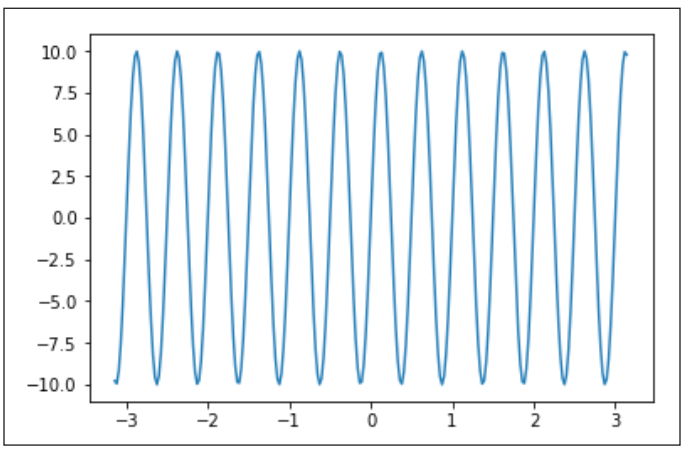


Figure 10.1: *The sin signal*

Step 3: Find the Fast Fourier Transform of the signal using `np.fft`:

```
sp = np.fft.fft(y)
freq = np.fft.fftfreq(y.shape[-1])
plt.plot(freq, sp.real, freq, sp.imag)
plt.show()
```

Output: The output of the above code is as follows (*Figure 10.2*):

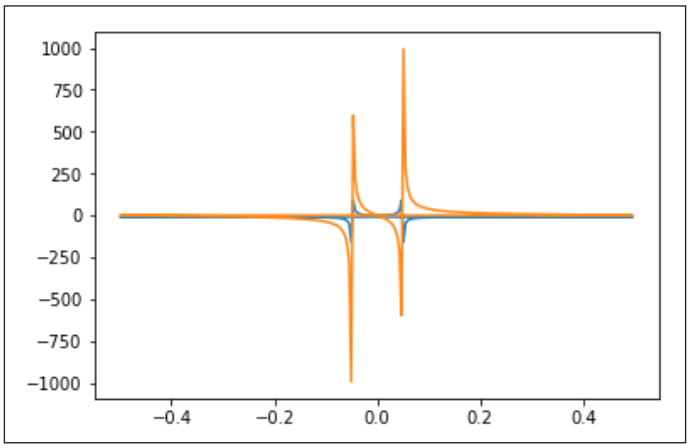


Figure 10.2: *Fourier Transform of the sin signal*

We can also find the magnitude of this complex Fourier Transform thus generated. The plot of the magnitude of FFT of the sin signal is shown in *Figure 10.3*:

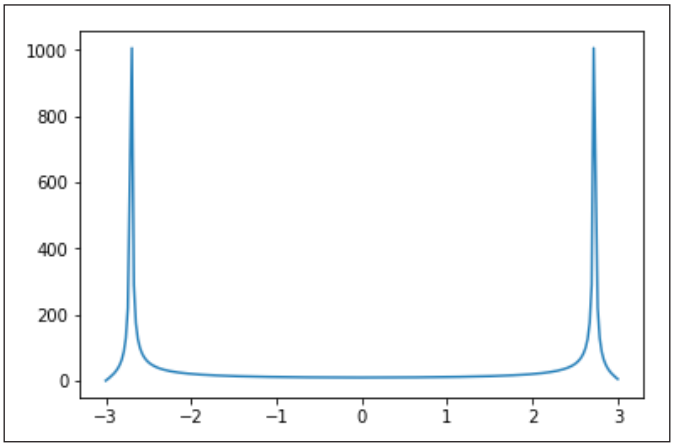


Figure 10.3: *Magnitude of the complex Fourier Transform of sin signal*

Note that the graph is symmetric about the *y*-axis. In the positive direction, there is a single frequency. Now consider the following signal (*Figure 10.4*):

$$y = A_1 \sin\sin\left(2\pi f_1 t\right) + A_2 \sin\sin\left(2\pi f_2 t\right)$$

The FFT of this signal would generate two frequencies and hence would help us to see what frequencies (and how many) constitute the signal. The following code asks the user to enter two frequencies and generates the sin signal. The FFT of this signal is then calculated. The FFT and corresponding magnitude are shown in figures that follow.

Code:

```
A1=float(input('Enter amplitude\t:'))
f1=int(input('Enter frequency\t:'))
t=np.linspace(-np.pi,np.pi,256)
y1=A1*np.sin(2*np.pi*f1*t)
A2=float(input('Enter amplitude\t:'))
f2=int(input('Enter frequency\t:'))
y2=A2*np.sin(2*np.pi*f2*t)
y=y1+y2
plt.plot(t,y)
plt.show()
```

Output: The output of the above code is as follows:

```
Enter amplitude :10
Enter frequency :2
Enter amplitude :10
Enter frequency :4
```

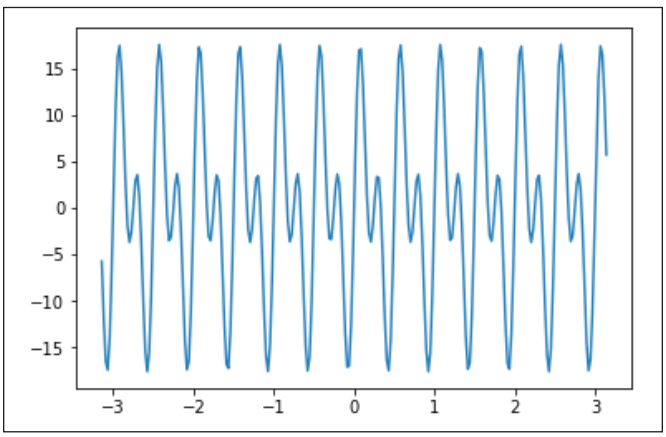


Figure 10.4: $y = A_1 \sin(2\pi f_1 t) + A_2 \sin(2\pi f_2 t)$

The FFT of the above signal can be found as follows. The output is shown in *Figure 10.5*:

Code:

```
sp=np.fft.fft(y)
freq=np.fft.fftfreq(y.shape[-1])
plt.plot(freq, sp.real, freq, sp.imag)
plt.show()
f=np.linspace(-3,3,256)
mod1=[np.sqrt((i.real**2+i.imag**2)) for i in sp]
mod2=np.abs(sp)
plt.plot(f,mod2)
```

Output: The output of the above code is as follows (*Figure 10.5*):

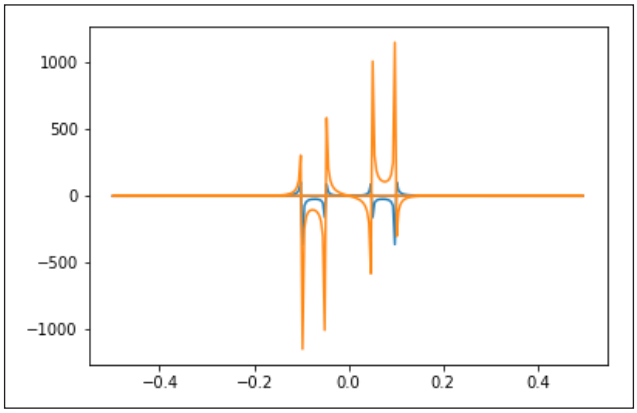


Figure 10.5: *The FFT of*

The magnitude of the complex Fourier Transform of is shown in *Figure 10.6*:

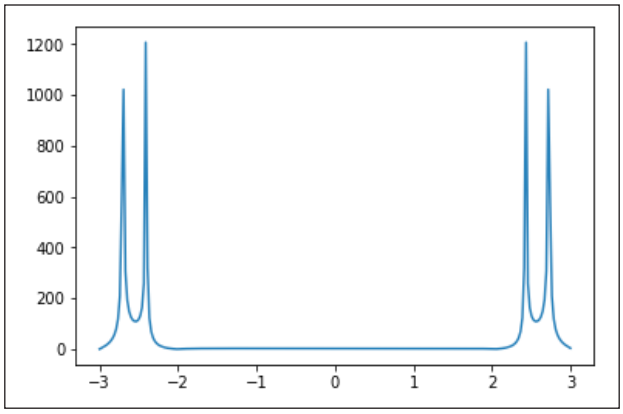


Figure 10.6: *Magnitude of the complex Fourier Transform of*

Likewise, the FFT of a signal containing three frequencies would generate three frequencies. The following code asks the user to enter three frequencies and generates the sin signal (*Figure 10.7*). The FFT of this signal is then calculated. The FFT and corresponding magnitude are shown in *Figure 10.8* and *Figure 10.9*.

Code:

```
A1=float(input('Enter amplitude\t:'))
f1=int(input('Enter frequency\t:'))
t=np.linspace(-np.pi,np.pi,256)
y1=A1*np.sin(2*np.pi*f1*t)
A2=float(input('Enter amplitude\t:'))
f2=int(input('Enter frequency\t:'))
y2=A2*np.sin(2*np.pi*f2*t)
A3=float(input('Enter amplitude\t:'))
f3=int(input('Enter frequency\t:'))
y3=A3*np.sin(2*np.pi*f3*t)
y=y1+y2+y3
plt.plot(t,y)
plt.show()
```

Output: The output of the above code is as follows:

```
Enter amplitude  :10
Enter frequency  :2
Enter amplitude  :10
Enter frequency  :4
Enter amplitude  :10
Enter frequency  :8
```

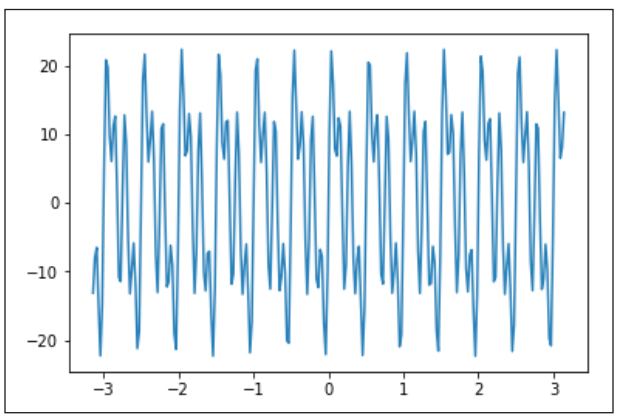


Figure 10.7: $y = A_1 \sin(2\pi f_1 t) + A_2 \sin(2\pi f_2 t) + A_3 \sin(2\pi f_3 t)$

The following code finds the Fourier Transform of this signal.

Code:

```
sp = np.fft.fft(y)
freq = np.fft.fftfreq(y.shape[-1])
plt.plot(freq, sp.real, freq, sp.imag)
plt.show()
f=np.linspace(-3,3,256)
mod1=[np.sqrt((i.real**2+i.imag**2)) for i in sp]
mod2=np.abs(sp)
plt.plot(f,mod2)
```

The output of the above code is shown in *Figure 10.8*:

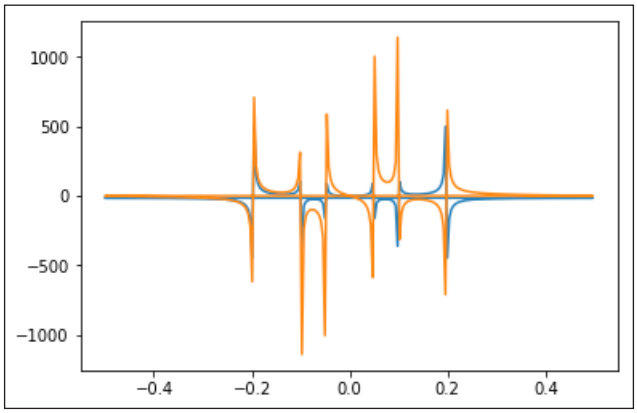


Figure 10.8: *The FFT of* $y = A_1 \sin(2\pi f_1 t) + A_2 \sin(2\pi f_2 t) + A_3 \sin(2\pi f_3 t)$

The magnitude of the signal is as follows (*Figure 10.9*):

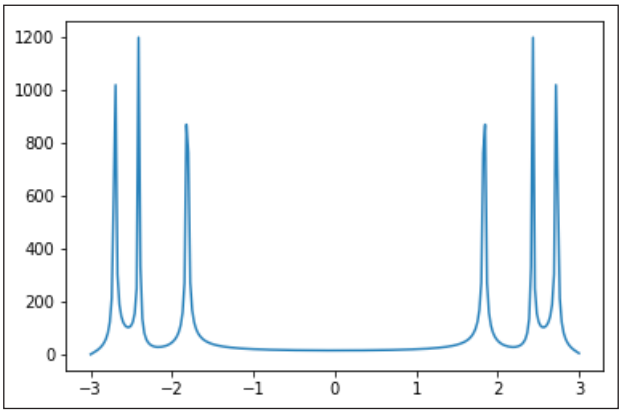


Figure 10.9: *The magnitude of FFT of* $y = A_1 \sin(2\pi f_1 t) + A_2 \sin(2\pi f_2 t) + A_3 \sin(2\pi f_3 t)$

So far, we have seen that the Fourier Transform can extract frequencies of simple signals. These frequencies may help us to distinguish between two signals. However, this method does not give desirable results with non-stationary signals. Although the FFT of a signal composed of two parts having different frequencies can also extract the various frequencies, the Fourier Transform of signals shown in *Figure 10.10* and *Figure 10.13* generate the same Fourier Transforms.

Code:

```
A=float(input('Enter amplitude\t:'))
f1=int(input('Enter frequency 1\t:'))
f2=int(input('Enter frequency 2\t:'))
t=np.linspace(-np.pi, np.pi,256)
y=np.zeros(256)
for i in range(128):
y[i]=A*np.sin(2*np.pi*f1*t[i])
for i in range(128,256):
y[i]=A*np.sin(2*np.pi*f2*t[i])
plt.plot(t,y)
plt.show()
#print(t)
```

Output: The output of the above code is as follows:

```
Enter amplitude  :10
Enter frequency 1   :2
Enter frequency 2   :4
```

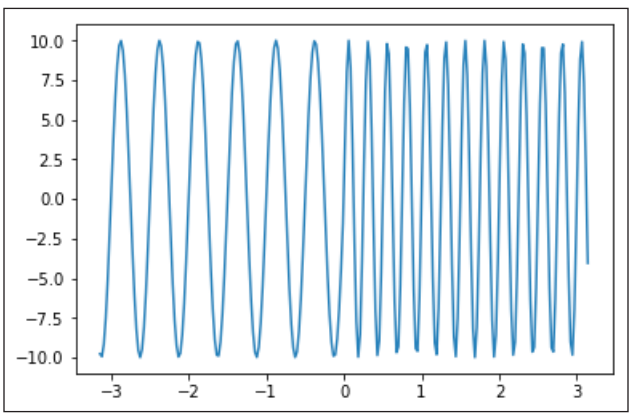


Figure 10.10: *A non-stationary signal*

The following code finds the FFT of this signal.

Code:

```
sp = np.fft.fft(y)
freq = np.fft.fftfreq(y.shape[-1])
plt.plot(freq, sp.real, freq, sp.imag)
plt.show()
f=np.linspace(-3,3,256)
mod1=[np.sqrt((i.real**2+i.imag**2)) for i in sp]
mod2=np.abs(sp)
plt.plot(f,mod2)
```

Output: The output of the above code is shown in *Figure 10.11*:

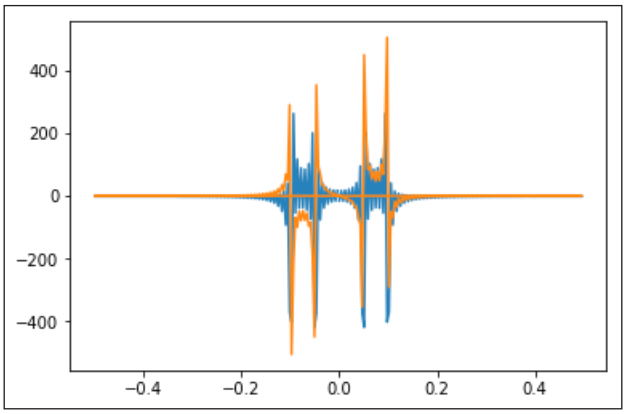


Figure 10.11: *The FFT of the signal shown in Figure 10.10*

The magnitude of the output is shown in *Figure 10.12*:

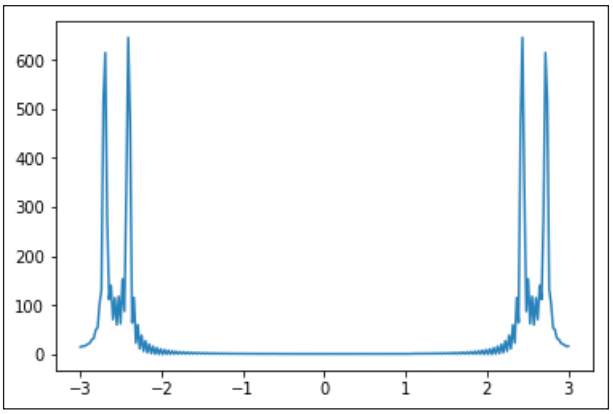


Figure 10.12: *The magnitude of the FFT of the signal shown in Figure 10.10*

Now, if the signal is changed so that the first part contains the higher frequency and the second part contains the lower one (*Figure 10.13*), the FFT of the signal remains the same (*Figure 10.14* and *Figure 10.15*).

The output of the re-run:

```
Enter amplitude :10
Enter frequency 1 :2
Enter frequency 2 :4
```

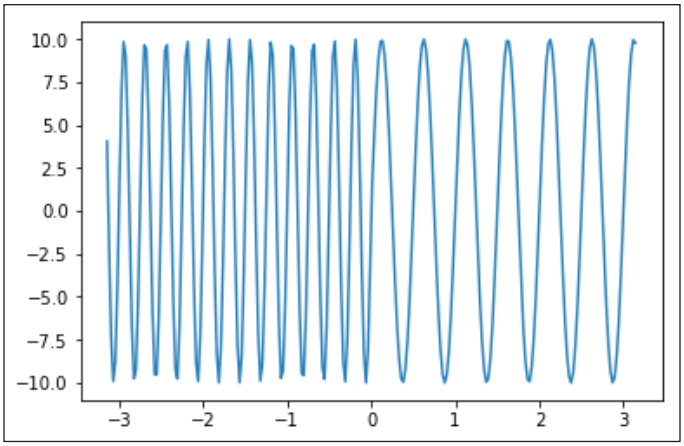


Figure 10.13: *Another example of a non-stationary signal*

The Fourier Transform of *Figure 10.13* is shown in *Figure 10.14*:

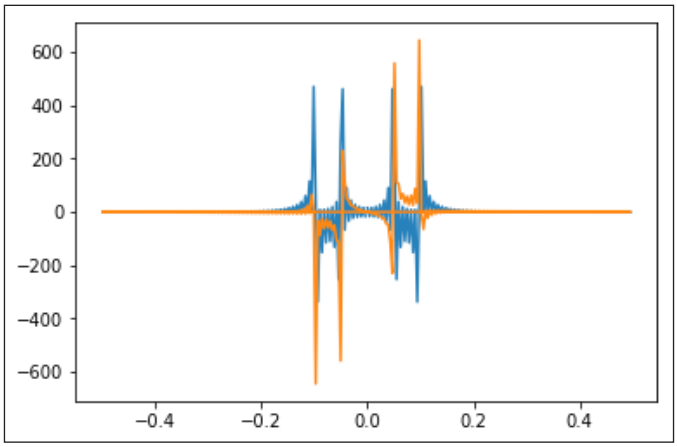


Figure 10.14: *The FFT of the signal shown in Figure 10.13*

The magnitude of the Fourier Transform of the *Figure 10.13* is shown in *Figure 10.15*:

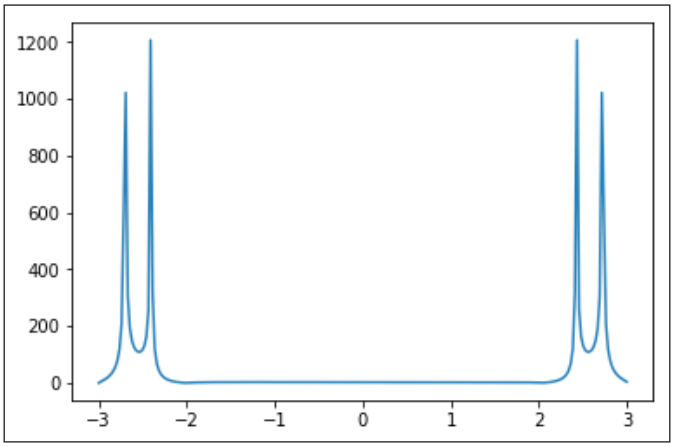


Figure 10.15: *The magnitude of the FFT of the signal shown in Figure 10.13*

The implementation and use of FFT have been elaborately explained. However, it has its shortcomings. The FFT cannot find the order of frequencies or the positions at which a particular frequency occurred. To find this, we use **Short Term Fourier Transform** (**STFT**). The STFT can be used for determining changes in the frequency and phases of non-stationary signals. The `scipy.signal` module can be used to find the `stft` of a given signal. *Table 10.3* presents the parameters of the `stft` function of the signal module of scipy:

Parameter	Type	Explanation
`x`	`array_like`	This parameter denotes the time series of measurement values
`fs`	`float`	It is an optional parameter. This parameter represents the sampling frequency of the x time series. The default value of this parameter is 1.0.
`window`	`str` or `tuple` or `array_like`	It is an optional parameter. The default value of this parameter is the Hann window. The `get_window` can be seen for a list of windows.
`nperseg`	`int`	It is an optional parameter. This parameter represents the length of each segment. The default value of this parameter is 256.
`boundary`	`str` or `None`	It is an optional parameter. This parameter specifies whether the input signal is extended at both ends.
`axis`	`int`	It is an optional parameter. It represents the axis along which the STFT is computed. The default is axis=-1.

Table 10.3: *The parameters of scipy.signal.stft*

The following table (*Table 10.4*) shows the attributes of the `stft` function:

Attribute	Type	Explanation
`f`	`ndarray`	This attribute represents the array of sample frequencies.
`t`	`ndarray`	This attribute represents the array of segment times.
`Zxx`	`ndarray`	This attribute represents the STFT of x.

Table 10.4: *The attributes of scipy.signal.stft*

The following code finds the `stft` of the given signal.

Code:

```
f, t,sp = signal.stft(y, 1000, nperseg=1000)
#plt.plot(freq, sp.real, freq, sp.imag)
plt.show()
f=np.linspace(-3,3,256)
mod1=[np.sqrt((i.real**2+i.imag**2)) for i in sp]
mod2=np.abs(sp)
plt.plot(f[127:],mod2[:,0])
plt.show()
plt.plot(f[127:],mod2[:,1])
plt.show()
plt.plot(f[127:],mod2[:,2])
plt.show()
```

Output: The output of the code is shown in figures (*Figure 10.16(a), Figure 10.16(b)* and *Figure 10.16(c)*:

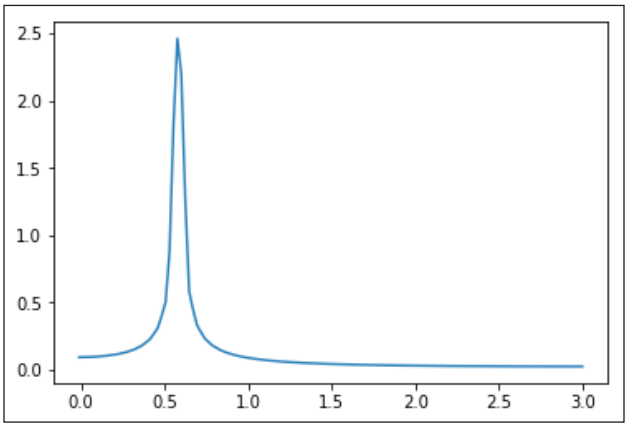


Figure 10.16 *(a): The STFT of the signal shown in Figure 10.13*

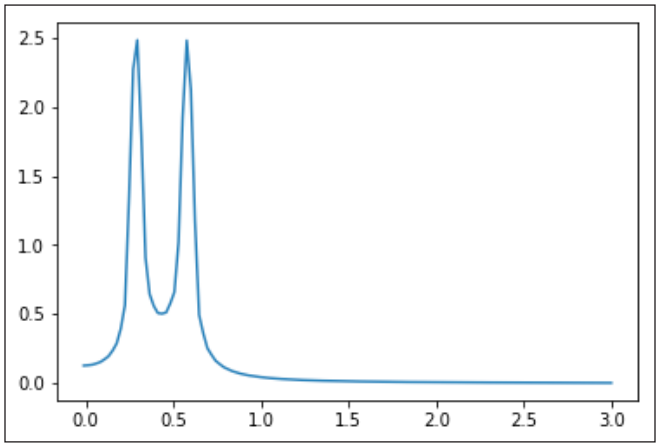


Figure 10.16 *(b): The STFT of the signal shown in Figure 10.13*

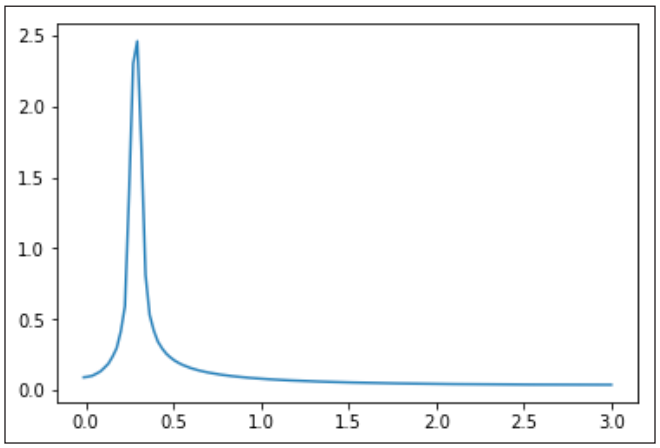


Figure 10.16 *(c): The STFT of the signal shown in Figure 10.13*

The frequency-based feature extraction is useful in the case of audio signals. The next section introduces patches, which can be used in the case of images.

Patches

Consider an image of size . A naïve way of extracting information from this image is to consider all the pixels as features, thus making an array containing 25 elements. This array will act as a feature vector.

One can also extract local information from the given image using patches. If we consider a patch, which moves 1 pixel at a time, as shown in *Figure 10.17*, 16 sub-arrays are generated for a image. Assume that we find the mean and the standard deviation of each of these 16 patches and consider the 32 values so obtained as the

features of the given image. Although the number of features will increase, but the features vector will become more informative.

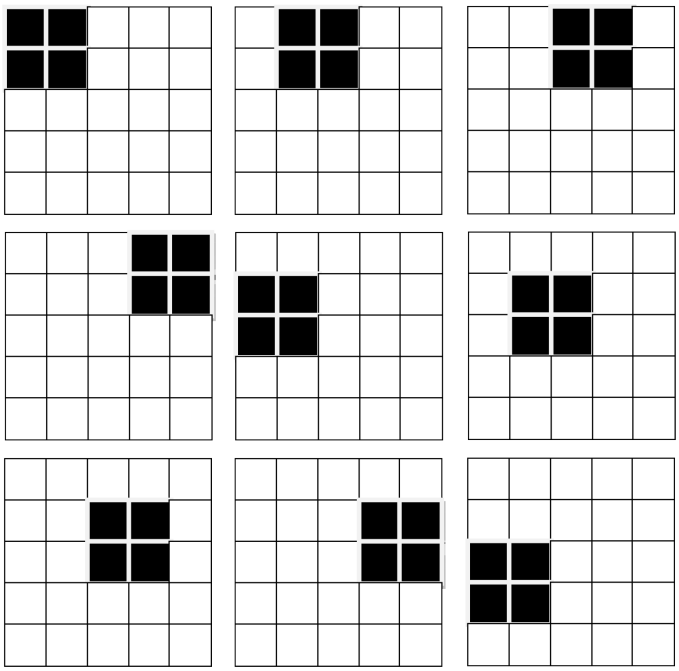


Figure 10.17: *A patch moves on a image*

Scipy provides an in-build function for extracting patches in `sklearn`. The details of the function are presented in the next section.

sklearn.feature_extraction.image.extract_patches_2d

This function reshapes a 2D image into a collection of patches. The resulting patches are allocated in a dedicated array. The parameters of the function are shown in *Table 10.5*:

Parameter	Type	Explanation
image	**array**	This parameter represents the original image data.
patch_size	**tuple of integers**	This parameter represents the dimension of each patch.
max_patches	**integer**	It is an optional parameter. The default value of the parameter is none. it denotes the maximum number of patches to extract.

Contd…

random_state	**int**	This parameter denotes the `RandomState`. It is an optional parameter, and its default value is `None`.

Table 10.5: *Parameters of extract_patches_2d*

The parameters and attributes of the function are as follows (*Table 10.6*):

Attribute	Type	Explanation
`patches`	`array`	It is the collection of patches extracted from the image.

Table 10.6: *Attributes of extract_patches_2d*

The following code extracts patches from the given image using the above function.

Code:

```
from sklearn.datasets import load_sample_image
from sklearn.feature_extraction import image
from matplotlib import pyplot as plt
import numpy as np
from skimage.color import rgb2gray

img1=load_sample_image('flower.jpg')
img1=rgb2gray(img1)
plt.imshow(img1)
plt.show()
patches=image.extract_patches_2d(img1,(2,2))
plt.imshow(patches[10000,:,:])
plt.show()
```

The above code results in a set of patches, from which relevant features can be extracted and used as features in classification or regression. The next section introduces a *Histogram of oriented gradients*, which can be used to represent an image compactly.

Histogram of oriented gradients

The **Histograms of Oriented Gradients** (**HOG**) is a popular feature descriptor. This technique finds the frequency of orientations in a localized portion of a given image. To calculate features using HOG, sliding window traverses over the whole image and gradients from a block are calculated from the change in intensities of a pixel within a block.

First of all, we convert a given image into a grayscale. Take a block, say of that matter, of . This block will have 25 pixels, and for any pixel at the horizontal and vertical gradient is calculated as follows:

$$H = I(i, j+1) - I(i, j-1)$$

$$V = I(i+1, j) - I(i-1, j)$$

$$Magnitude = \sqrt{(H^2 + V^2)}$$

$$Theta = \tan^{-1}\tan^{-1}\left(\frac{V}{H}\right)$$

Now, a histogram of theta is as formed, which acts as the feature set. In `sklearn`, the HOG has been implemented in the `skimage.feature.hog`. The following code demonstrates the implementation of HOG using `sklearn`. Note that the HOG image corresponding to *Figure 10.18* is shown in *Figure 10.19*:

Code:

```
def rgb2gray(rgb):
    r,g,b=rgb[:,:,0],rgb[:,:,1],rgb[:,:,2]
    gray=0.2989*r+0.5870*g+0.1140*b
    return gray_img
#Original image
data = load_sample_images()
len(data.images)
img1 = data.images[0]
img1.shape
plt.imshow(img1)
```

Output:

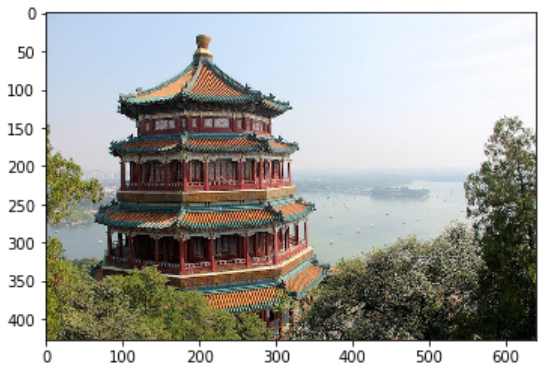


Figure 10.18: *Original Image*

```
#Finding HOG image
img2=rgb2gray(img1)
plt.imshow(img2)
fd, hog_image = hog(img1, orientations=8, pixels_per_cell=(16, 16),cells_
per_block=(1, 1), visualize=True, multichannel=True)
plt.imshow(hog_image)
```

Output:

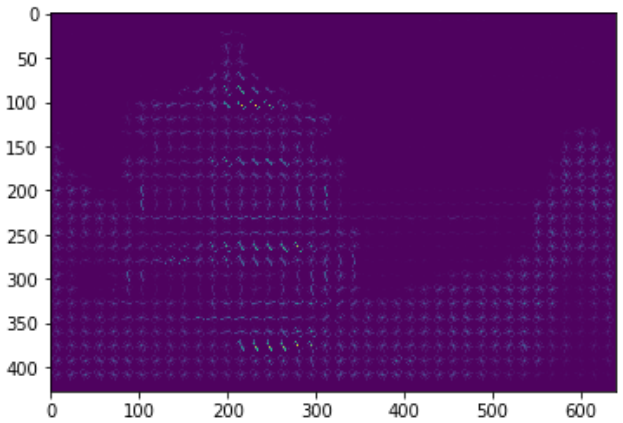


Figure 10.19: *The HOG image*

Likewise, for data.images[1] (*Figure 10.20 (a)*), the HOG is shown in *Figure 10.20 (b)*:

Figure 10.20 *(a): Original Image of flower*

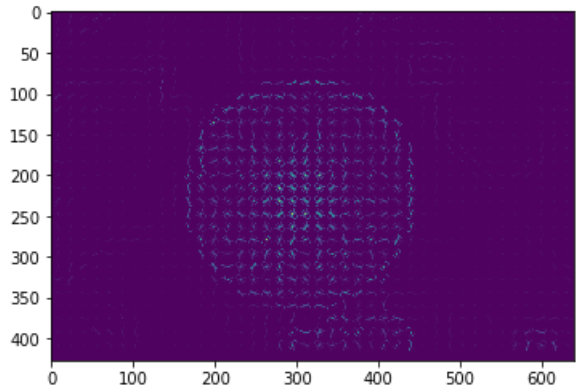


Figure 10.20 *(b): HOG of the image shown in Figure 10.20 (a)*

The user can use the `fd`, obtained in the above code, as the feature of the given image. This feature vector can be used to construct the features of the two given classes in a classification problem, and the data so obtained can be used for classification. The next section introduces the principal component analysis. The method transforms the data in the new dimensions considering the variance.

Principal component analysis

The principal component analysis transforms the given data into another set of dimensions and finds the direction cut of maximum scatter. It is accomplished by using the eigenvectors. Take, for example, the data of students of a particular school. The data contains m features, including the age of a student and his date of birth. Since both of them are dependent and assuming the rest of the attributes are not dependent, the relevant information is contained in the rest of $(m - 1)$ features. Now imagine if we do not know about the dependency of features and still want to remove the redundant features. The method described in this section helps us to achieve this task.

The given data X is a $n \times m$ matrix, having n samples and m features.

Find mean $\overline{X} = \sum_{i=1}^{n} X$, which becomes a $1 \times m$ matrix:

1. Subtract $\overline{X}$ from X by broadcasting $\overline{X}$
2. Find $S = (X - \overline{X})(X - \overline{X})^T$
3. Find the eigenvalues and eigenvectors of S
4. Place the vectors in the increasing order of their eigenvalues

To get hold of the method, consider *Figure 10.21*. The first figure shows the original data, which requires two dimensions to separate the data. The second figure shows

the new dimension, found by applying principal component analysis. Note that the transformation takes the data into new dimensions. Here, just one axis is sufficient to classify the data:

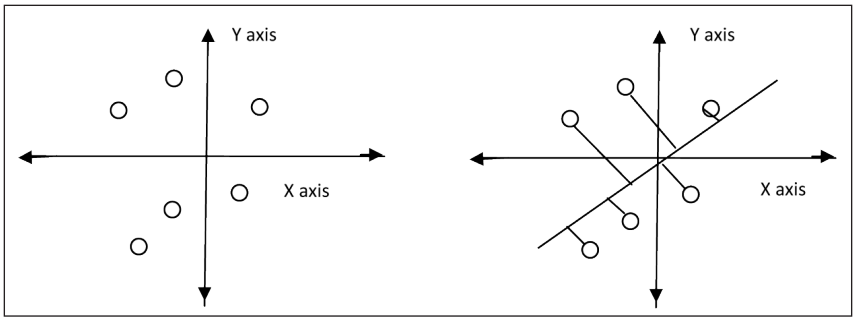


Figure 10.21: *Data before and after applying PCA*

The following steps will take you through the process of transforming the data using the principal component analysis:

Step 1: Import the following modules:

```
from sklearn.datasets import load_iris
import numpy as np
from matplotlib import pyplot as plt
from numpy import linalg
```

Figure 10.22 shows the first feature. The red and the blue color denote the samples of the two classes. Note that the given data cannot be classified easily using only one feature:

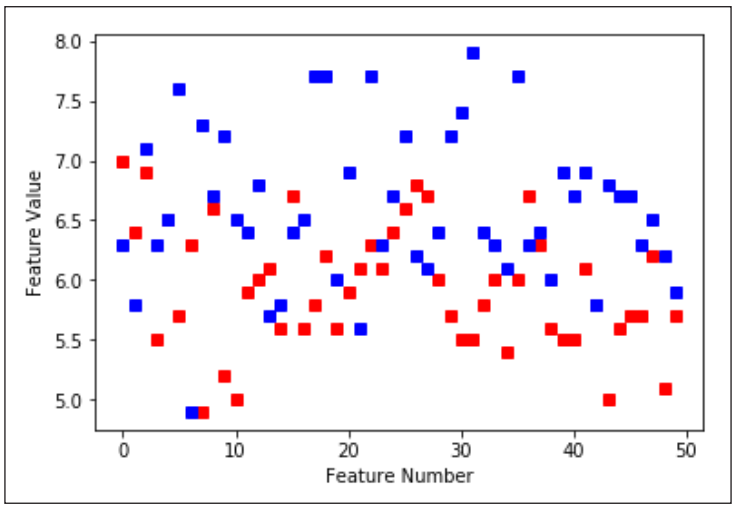


Figure 10.22: *The first feature of the data. The red and blue dots represent the two classes.*

Next, we find the PCA of the given data.

Step 2: Find PCA:

```
mean=np.mean(Data, axis=0)
s=np.matmul(np.transpose(Data-mean),(Data-mean))
val, vec=linalg.eig(s)
Data_transformed=np.matmul(Data,vec)
print(Data_transformed.shape)
Data1=Data_transformed[50:100,:]
Data2=Data_transformed[100:150,:]
plt.plot(index,Data1[:,0],'rs')
plt.plot(index,Data2[:,0],'bs',)
plt.xlabel('Feature Number')
plt.ylabel('Feature Value')
plt.show()
plt.plot(index,Data1[:,1],'rs')
plt.plot(index,Data2[:,1],'bs',)
plt.xlabel('Feature Number')
plt.ylabel('Feature Value')
plt.show()
```

Figure 10.23 shows the first feature of the transformed data. The red and the blue color denote the samples of the two classes. Note that the classification of the given data has become easy using only this feature:

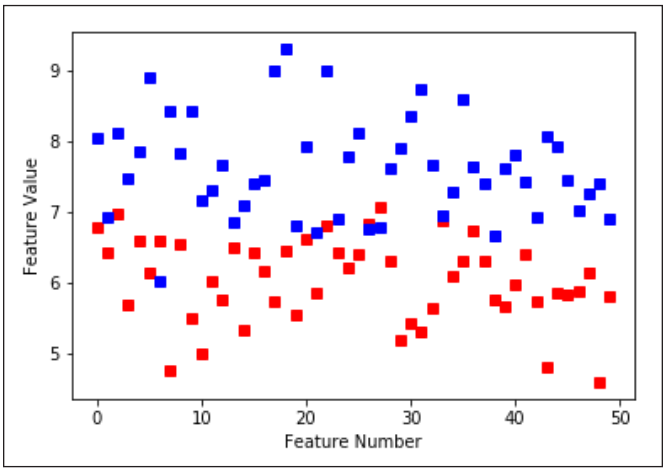


Figure 10.23: *The First feature of the transformed data. The red and blue dots represent the two classes.*

Note that the classification becomes easy after the transformation. In general, you can use only a few features of the transformed data to accomplish your task after applying PCA.

Conclusion

Feature extraction is an important constituent of any machine learning model. It helps us to extract relevant features from the given data. It may also help us to reduce dimensionality, hence making the model efficient and effective.

This chapter introduces four important feature extraction methods, namely, Fourier Transform, patches, HOG, and principal component analysis. The basics, theory, and implementation of these topics have been explained in the chapter.

The reader is expected to implement the above techniques and use the classifiers studied the previous chapters to see the effect of these techniques on the performance of the model. Moreover, there are some other feature extraction techniques like ICA and LDA. LDA has already been discussed in the book. The reader is requested to explore the Bibliography at the end of this book for more feature extraction methods like Local Binary Patterns and Wavelet Transform.

Exercises

Multiple Choice Questions

1. Which technique helps us to transform the data into a frequency domain?

a. Fourier Transform
b. Local binary pattern
c. Principal component analysis
d. All of the above

2. Which technique helps us to transform the data into new dimensions based on variance?

a. Fourier Transform
b. Local binary pattern
c. Principal component analysis
d. All of the above

3. Which technique helps us to find the edges of a picture?

a. Fourier Transform
b. Local binary pattern
c. Principal component analysis
d. All of the above

4. Which technique is more efficient Discrete Fourier Transform or Fast Fourier Transform?
   a. Discrete Fourier Transform
   b. Fast Fourier Transform
   c. Both are equally efficient
   d. Cannot say
5. Which of the following helps you to analyze non-stationary signals?
   a. Discrete Fourier Transform
   b. Fast Fourier Transform
   c. Short Term Fourier Transform
   d. None of the above
6. Which of the following signal cannot be analyzed using FFT?
   a. Sin signal consisting of a single frequency
   b. Sin signal consisting of two frequencies
   c. Non-stationary signal
   d. None of the above
7. Patches are generally used for?
   a. Audio data
   b. Video data
   c. Imaging data
   d. None of the above
8. Fourier is used for?
   a. Audio data
   b. Video data
   c. Imaging data
   d. None of the above
9. PCA is generally used for?
   a. Audio data
   b. Video data
   c. Imaging data
   d. All of the above
10. Which of the following is not a feature extraction method?
    a. PCA
    b. LDA
    c. LBP
    d. FDR

Theory

1. Explain the concept of principal component analysis and write the algorithm to find the PCA of a given data.
2. Explain the Fourier Transform. Write an algorithm to find the Fourier Transform of a given signal.
3. What changes will you make in the above algorithm to improve its complexity?
4. Which type of signals cannot be analyzed using the Fourier Transform? Explain Short Term Fourier Transform.
5. Explain the importance of patches.

Programming

1. Find the first-order statistical features of an image.
   a. Can you use the feature obtained so obtained for classification? Take any imaging dataset with two classes and verify.
   b. Explain the results so obtained.
2. Take an image and convert it into grayscale.
   a. Check if the intensity of each pixel is between 0 and255.
   b. Now, for each pixel, find whether its intensity is greater than each of its eight neighbors or not.
   c. Replace the neighbor's value by 1, if the neighbor's value is greater than the intensity of the central pixel; else replace the neighbor's value by 0.
   d. Find the decimal equivalent of the eight-bit binary number obtained by the neighbor's binary value obtained in c).
   e. Observe the image, what difference do you see vis-à-vis the original one.
3. Now repeat the above steps for two sets of images and use SVM to classify the dataset. Report the accuracy, specificity, and sensitivity.
4. Repeat the above experiment by replacing the transformed image with the histogram of intensity values.
5. Can you reduce the number of bins in the above histogram?
6. Why do you think the above system can classify the images? The reader may refer to http://www.scholarpedia.org/article/Local_Binary_Patterns.
7. The Histogram of Gradients (HoG) is a technique that counts occurrences of gradient orientation in localized portions of an image. The reader may refer to https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients for a detailed explanation of the method. Implement the method and classify any imagining dataset using this feature extraction technique.
8. Apply PCA to an imaging dataset (2-class problem) and classify the dataset. Report accuracy, specificity, and accuracy. Can you explain the results?
9. Refer to the bibliography at the end of this book and implement Wavelet Transforms. Can this feature extraction technique for classification.
10. Can you suggest a feature extraction method based on the combination of Q2 and Q1? Verify your claim by taking an imaging dataset and performing classification.

Appendix 1
Cheat Sheet – Pandas

Pandas: Software Library, mainly used for Data manipulation and Analysis.

Developed By: Wes McKinney.

Released On: 11th January 2008.

Free: It is released under the three-clause BSD license.

Important Data Structures: (a) Series and (b) Data Frame.

Series: A Pandas Series represents a one-dimensional array of indexed data [5].

Data Frame: A Data Frame is a two-dimensional labeled array that stores ordered collection of columns [5].

Creating a Pandas series

The Series function helps us to create Series data type. It can be done using:

- List
- An Array
- A dictionary.

Using a List

Syntax:

```
Pandas.Series(L)
```

Where `L` is a list.

Example:

```
L= [10, 20, 30, 40, 50]
Series(L)
L=[10, 20, 30, 40, 50]
S1= pd.Series(L)
S1
```

Output:

```
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

Using NumPy Array

Syntax:

```
Pandas.Series(Arr)
```

Where `Arr` is an NumPy array.

Example:

```
Arr1=np.random.randint(3, 89, 10)
S2=pd.Series(Arr1)
S2
```

Output:

```
0    61
1     9
2    82
```

```
3    63
4    11
5    41
6    37
7    68
8    10
9    48
dtype: int32
```

Using Dictionary

Syntax:

`Pandas.Series(D)`

Where `D` is a dictionary

Example:

```
D1={'Harsh':100, 'Aayush':22, 'Arush':22}
S3=pd.Series(D1)
S3
```

Output:

```
Harsh     100
Aayush     22
Arush      22
dtype: int64
```

Indexing

Procedure	Example	Code
Using Keys	Access the value corresponding to the key `'Arush'` in Series `S1`	`S1['Arush']`
Using index	Access the value at index `0`	`S1[0]`
_	Access value at the last position	`S3[-1]`
Accessing values using loc	Access value corresponding to the key `'Harsh'` in Series `S1`	`S1.loc['Harsh']`
Accessing values using `iloc`	Access value at index =1 in the Series `S1`	`S1.iloc[1]`

Slicing

Slicing produces a Sub-Series from a given Series.

Example:

Access element from index 3 up to 5 (5 not included) from Series S1.

```
S1[3:5]
```

Common methods

The common methods of the Series data type are presented in *Table 2*:

Function	Purpose	Example
`head()`	This method displays the top 5 values of the Series.	`S1.head()`
`tail()`	This method displays the last five values of the Series.	`S1.tail()`
`index`	It displays the index(s) of the given Series.	`S1.index()`
`describe()`	This method shows the count, mean, min, max, 25%, 50%, 75% and std (standard Deviation)	`S1.describe()`
`sort_values()`	This method sorts the items of the given Series and displays the indexes of the sorted arrays.	`S1.sort_values()`
`max`	It finds the maximum value from a given series.	`S1.max()`
`min`	It finds the minimum value from a given series.	`S1.min()`
`sum`	It finds the sum of values from a given series.	`S1.sum()`
`median`	It finds the median of the value from a given series.	`S1.median()`
`value_counts`	It counts the frequencies of values in a given Series.	`S1.value_counts()`

Table 1: *Common methods of the Series Data Type*

Boolean index

In a Series, the necessary condition can be specified inside the square brackets to get the elements that satisfy the given condition. For example, the following statement selects the elements of `S1`, which are greater than 20 and put them in `S2`:

```
S2=S1[S1>20]
```

DataFrame

Creation

a) By passing a dictionary in which each index is associated with a list of values in the `DataFrame` method of Pandas.

Example:

```
D={'Harsh':[1, 2, 3, 4], 'Arsh':[4, 5, 6, 7], 'Sparsh': [7, 9, 8, 10]}
DF_=pd.DataFrame(D)
```

Output:

	Harsh	Arsh	Sparsh
0	1	4	7
1	2	5	8
2	3	6	9
3	4	7	10

b) By passing a two-dimensional NumPy array in the `DataFrame` method of Pandas:

Example:

```
arr2=np.random.randint(2, 89, (3,3))
DF_2=pd.DataFrame(arr2)
```

Output:

	0	1	2
0	64	85	66
1	5	80	84
2	23	60	38

c) By passing some Series in the `DataFrame` method of Pandas:

Example:

```
S1=pd.Series(np.random.randint(2,89,10))
S2=pd.Series(np.random.randint(0,10,10))
S3=pd.Series(np.random.randint(20,70,10))
DF_3=pd.DataFrame({'Score':S1, 'Papers':S2, 'Age':S3})
DF_3
```

Output:

	Score	Papers	Age
0	25	4	46
1	77	0	33
2	16	9	67
3	43	1	61
4	48	8	33
5	72	5	22
6	79	3	49
7	80	6	23
8	20	6	20
9	53	0	27

Adding a Column in a Data Frame

Syntax:

```
<name of the Data Frame>['<Column Name'>] = L
```

Where `L` is a List.

Example:

```
Sal=np.random.randint(10000,100000,10)
DF_3['Sal']=Sal
DF_3
```

Output:

	Score	Papers	Age	Sal
0	25	4	46	19850
1	77	0	33	17093
2	16	9	67	90831
3	43	1	61	77055
4	48	8	33	58274
5	72	5	22	95604
6	79	3	49	29780
7	80	6	23	88793
8	20	6	20	29931
9	53	0	27	22357

Deleting column

The `drop` function helps us to delete a column from a DataFrame.

Example:

```
DF_3.drop('Sal', axis=1)

DF_3
```

Output:

	Score	Papers	Age
0	25	4	46
1	77	0	33
2	16	9	67
3	43	1	61
4	48	8	33
5	72	5	22
6	79	3	49
7	80	6	23
8	20	6	20
9	53	0	27

Addition of Rows

The `concat` function helps to concatenate a `DataFrame` with another Data Frame.

Example:

```
DF_4=pd.DataFrame(np.random.randint(10,1000,(5,3)))

pd.concat([DF_3, DF_4])
```

Output:

0	1	2	Age	Papers	Sal	Score	
0	NaN	NaN	NaN	46.0	4.0	19850.0	25.0
1	NaN	NaN	NaN	33.0	0.0	17093.0	77.0
2	NaN	NaN	NaN	67.0	9.0	90831.0	16.0
3	NaN	NaN	NaN	61.0	1.0	77055.0	43.0
4	NaN	NaN	NaN	33.0	8.0	58274.0	48.0

Contd…

5	NaN	NaN	NaN	22.0	5.0	95604.0	72.0
6	NaN	NaN	NaN	49.0	3.0	29780.0	79.0
7	NaN	NaN	NaN	23.0	6.0	88793.0	80.0
8	NaN	NaN	NaN	20.0	6.0	29931.0	20.0
9	NaN	NaN	NaN	27.0	0.0	22357.0	53.0
0	566.0	436.0	746.0	NaN	NaN	NaN	NaN
1	597.0	331.0	666.0	NaN	NaN	NaN	NaN
2	997.0	407.0	702.0	NaN	NaN	NaN	NaN
3	958.0	447.0	741.0	NaN	NaN	NaN	NaN
4	906.0	226.0	688.0	NaN	NaN	NaN	NaN

Deletion of Rows

The `drop` function is used to drop rows/columns from a Data Frame.

Example:

```
DF_4.drop([4], axis=0, inplace=True)
```

Output:

0	1	2	
0	566	436	746
1	597	331	666
2	997	407	702
3	958	447	741

unique

The `unique` function finds unique values in a column of a Data Frame.

Example: To see the unique values of the age column of the `Students_df` Data Frame issue the following command:

```
Students_df.Age.unique()
```

nunique

The `nunique` function finds the number of unique values in a Data Frame column.

Example:

```
Students_df.Age.unique()
```

Iterating a Pandas Data Frame

a) `iterrows()`:

The `pandas.DataFrame.iterrows` method helps us to iterate through the rows of a Data Frame.

Example:

```
for index, row in DF_4.iterrows():
   print(index,' : ', row)
```

b) `index`:

The `pandas.DataFrame.index` attribute may also be used to iterate over a given Data Frame rows.

Example:

```
for ind in DF_3.index:
   print(DF_3['Age'][ind],' year old, Papers= ', DF_3['Papers'][ind])
```

Output:

```
46 year old, Papers= 4 33 year old, Papers= 0 67 year old, Papers= 9
61 year old, Papers= 1 33 year old, Papers= 8 22 year old, Papers= 5
49 year old, Papers= 3 23 year old, Papers= 6 20 year old, Papers= 6
27 year old, Papers= 0
```

c) `iteritems()`:

The `DataFrame.iteritems()` method can also be used to iterate through a Pandas Data Frame.

Example:

```
for label, col in DF_4.iteritems():
   print(label,' : ', col)
```

Output:

```
0 : 0 566 1 597 2 997 3 958 Name: 0, dtype: int32 1 : 0 436 1 331 2
407 3 447 Name: 1, dtype: int32 2 : 0 746 1 666 2 702 3 741 Name: 2,
dtype: int32
```

Some of the important methods and procedures to deal with the Pandas data types have been presented in this Appendix. The reader should visit https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html for a detailed discussion on DataFrames and https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.html for a detailed discussion on Series.

Appendix 2
Face Classification

Introduction

Face classification is the process of identifying the face of a person from an image. It is an important problem but not an easy one. The conventional algorithms fail at detecting or classifying faces. We are given two sets of images: first containing the face of a person and the second containing the face of another person. The task is to identify the first person from an image that is not provided in the training set. In this process, the training set contains images of a face, and the corresponding label and the label of the test set needs to be predicted.

Here, machine learning comes to our rescue and helps us tackle this problem. The Pipeline used in this problem requires the pre-processing of the data, followed by feature extraction, feature selection and finally applying the classification algorithm.

In this appendix the Local Binary Pattern (LBP) is used for feature extraction, Fischer Discriminant Ratio (FDR) is used for feature section and the classification is carried out using the Support Vector Machine. The results of the application of the pipeline on the given dataset are encouraging.

Data

Source: The project uses two sets of images having 28 images each belonging to two classes which are henceforth called Actor1 and Actor2. The images are of the shape 60 × 60 × 3. The data set was divided into test and train by taking 70% of the images for training and 30% for testing.

Conversion to grayscale:

Before applying feature extraction, the images are converted into the grayscale. This is because LBP has been used for feature extraction. One of the aims of this extraction method is to detect edges, which can be detected using the grayscale images also. The shape of each image after this conversion becomes 60 × 60.

Methods

Feature extraction

LBP aims at finding the relevant features and at the same time reducing the number of features. Local Binary Pattern (LBP) is a simple yet very efficient texture operator which labels the pixels of an image by thresholding the neighborhood of each pixel and considers the result as a binary number [6]. The original LBP operator [6] forms labels for the image pixels by thresholding the 3 x 3 neighborhood of each pixel with the center value and considering the result as a binary number. The histogram of these $2^8 = 256$ different labels can then be used as a texture descriptor [6]. The application of feature extraction reduces the dataset to a () matrix.

Splitting of data

Sklearn module train_test_split is used to spit the data in training data and test data with testing data of size 30%.

Feature Selection

Fisher Discriminant Ratio (FDR) aims at assigning value to each feature based on its mean and the variance, in accordance with its relevance to the label. The features are then arranged in the decreasing order of their FDR values.

Forward Feature Selection

Forward feature selection is an iterative method in which we start with zero feature in the model. We continue adding a feature in each iteration which improves the performance of the model until addition of a new variable starts to degrade the performance of the model.

Classifier

The Support Vector Machine is a popular classification algorithm. SVM classifier is used as it can perform maximum separation hyperplane. SVM classifier with the linear kernel is used as a classifier and is trained using training data and then used to predict labels of testing data.

Observation and Conclusion

The variation of accuracy with the number of features is shown in Figure 1. It can be observed that the highest accuracy achieved is 91% when only 4 features are taken for model training.

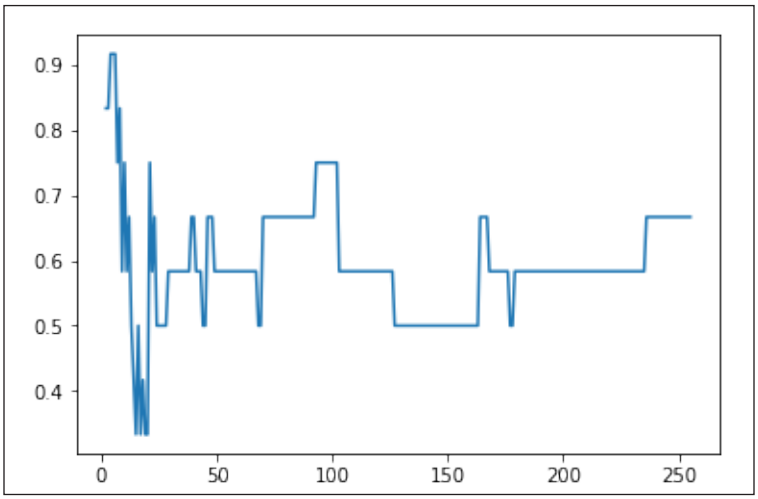


Figure 1: *Variation of accuracy with the number of features.*

The reader is encouraged to try out the combinations of different feature extraction methods, classifiers, and feature selection methods and compare the results.

Bibliography

General

Richard O. Duda, Peter E. Hart, and David G. Stork. (2000). Pattern Classification (2nd Edition). Wiley-Interscience, USA.

Mitchell Tom M., (1986). Machine Learning, Mg. Graw Hill.

Simon Haykin. (1998), Neural Networks: A Comprehensive Foundation (2nd. ed.). Prentice Hall PTR, USA.

Russell, S. & Norvig, P. (2003). *Artificial intelligence, a modern approach* (2nd ed.). Englewood Cliffs: Prentice Hall.

T. Ojala, M. Pietikäinen, and D. Harwood (1994), "Performance evaluation of texture measures with classification based on Kullback discrimination of distributions", Proceedings of the 12th IAPR International Conference on Pattern Recognition (ICPR 1994), vol. 1, pp. 582 - 585.

T. Ojala, M. Pietikäinen, and D. Harwood (1996), "A Comparative Study of Texture Measures with Classification Based on Feature Distributions", Pattern Recognition, vol. 29, pp. 51-59.

Nearest Neighbors

[1] Cover, T. M. (1968). Estimation by the nearest neighbor rule. *IEEE Transactions on Information Theory, IT-14*, 50–55.a

[2] Cover, T. M. & Hart, P. E. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory, IT-13*, 21–27. Dasarathy, B. V. (1991). *Nearest-neighbor classification techniques*. Los Alomitos: IEEE Computer Society Press.

[3] Dudani, S. A. (1975). The distance-weighted *k*-nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics, SMC-6*, 325–327.

Neural Networks

[1] Minsky, M. & Papert, S. (1969). *Perceptrons*. Cambridge, MA: MIT.

[2] Hopfield, J. J. (1982). "Neural networks and physical systems with emergent collective computational abilities". Proc. Natl. Acad. Sci. U.S.A. 79 (8): 2554–2558.

[3] *McCulloch, Warren; Walter Pitts (1943). «A Logical Calculus of Ideas Immanent in Nervous Activity». Bulletin of Mathematical Biophysics. 5 (4): 115–133.*

[4] Rosenblatt, F. (1958). "The Perceptron: A Probalistic Model For Information Storage And Organization In The Brain". Psychological Review. 65 (6): 386–408.

[5] Minsky, M.; S. Papert (1969). An Introduction to Computational Geometry. MIT Press.

[6] McCulloch, Warren; Pitts, Walter (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics. 5 (4): 115–133.

[7] Hebb, Donald (1949). The Organization of Behavior. New York: Wiley.

Support Vector Machines

[1] Cortes, Corinna*; Vapnik, Vladimir N. (1995).* "Support-vector networks" (PDF). Machine Learning. *20 (3): 273–297.*

Decision Trees

[1] Breiman, L., Friedman, J., Olshen, R., & Stone, C. J. (1984). *Classification and regression trees*. Belmont: Wadsworth International Group.

[2] Dietterich, T. (1998). Approximate statistical tests for comparing supervised classification learning algorithms. *Neural Computation, 10*, 1895–1923.

[3] Murty, M. N. & Krishna, G. (1980). A computationally efficient technique for data clustering. *Pattern Recognition, 12*, 153–158.

[4] Quinlan, J. R. (1987). "Simplifying decision trees". International Journal of Man-Machine Studies. 27 *(3): 221–234.*

[5] K. Karimi and H.J. Hamilton (2011), "Generation and Interpretation of Temporal Decision Rules", International Journal of Computer Information Systems and Industrial Management Applications, Volume 3

[6] Wagner, Harvey M. (1 September 1975). Principles of Operations Research: With Applications to Managerial Decisions (2nd ed.). Englewood Cliffs, NJ: Prentice Hall.

[7] R. Quinlan, "Learning efficient classification procedures", Machine Learning: an artificial intelligence approach, Michalski, Carbonell & Mitchell (eds.), Morgan Kaufmann, 1983, p. 463–482.

[8] Utgoff, P. E. (1989). Incremental induction of decision trees. Machine learning, 4(2), 161–186.

Clustering

[1] Cheeger, Jeff (1969). "A lower bound for the smallest eigenvalue of the Laplacian". Proceedings of the Princeton Conference in Honor of Professor S. Bochner.

[2] William Donath and Alan Hoffman (1972). "Algorithms for partitioning of graphs and computer logic based on eigenvectors of connections matrices". IBM Technical Disclosure Bulletin.

[3] Fiedler, Miroslav (1973). "Algebraic connectivity of graphs". Czechoslovak Mathematical Journal.

[4] Stephen Guattery and Gary L. Miller (1995). "On the performance of spectral graph partitioning methods". Annual ACM-SIAM Symposium on Discrete Algorithms.

[5] Daniel A. Spielman and Shang-Hua Teng (1996). "Spectral Partitioning Works: Planar graphs and finite element meshes". Annual IEEE Symposium on Foundations of Computer Science.

[6] Ng, Andrew Y and Jordan, Michael I and Weiss, Yair (2002). "On spectral clustering: analysis and an algorithm" (PDF). Advances in Neural Information Processing Systems.

[7] Rokach, Lior, and Oded Maimon. "Clustering methods." Data mining and knowledge discovery handbook. Springer US, 2005. 321-352.

[8] Frank Nielsen (2016). "Chapter 8: Hierarchical Clustering". Introduction to HPC with MPI for Data Science. Springer.

[9] R. Sibson (1973). "SLINK: an optimally efficient algorithm for the single-link cluster method" (PDF). The Computer Journal. British Computer Society. 16 *(1): 30–34.*

[10] D. Defays (1977). "An efficient algorithm for a complete-link method". The Computer Journal. British Computer Society. 20 *(4): 364–366.*

Fourier Transform

[1] *Taneja, H.C. (2008),* "Chapter 18: Fourier integrals and Fourier transforms", *Advanced Engineering Mathematics, Vol. 2, New Delhi, India: I. K. International Pvt Ltd,* ISBN 978-8189866563.

[2] *Jont B. Allen (June 1977). «Short Time Spectral Analysis, Synthesis, and Modification by Discrete Fourier Transform». IEEE Transactions on Acoustics, Speech, and Signal Processing. ASSP-25 (3): 235–238.*

Principal Component Analysis

[1] Jackson, J.E. (1991). A User's Guide to Principal Components (Wiley).

[2] *Jolliffe, I. T. (1986).* Principal Component Analysis. *Springer Series in Statistics. Springer-Verlag. pp. 487.*

[3] Jolliffe, I.T. (2002). Principal Component Analysis, second edition (Springer).

Histogram of Oriented Gradients

[1] Navneet Dalal and Bill Triggs, Histograms of Oriented Gradients for Human Detection, lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf