Lecture Notes

Machine Learning

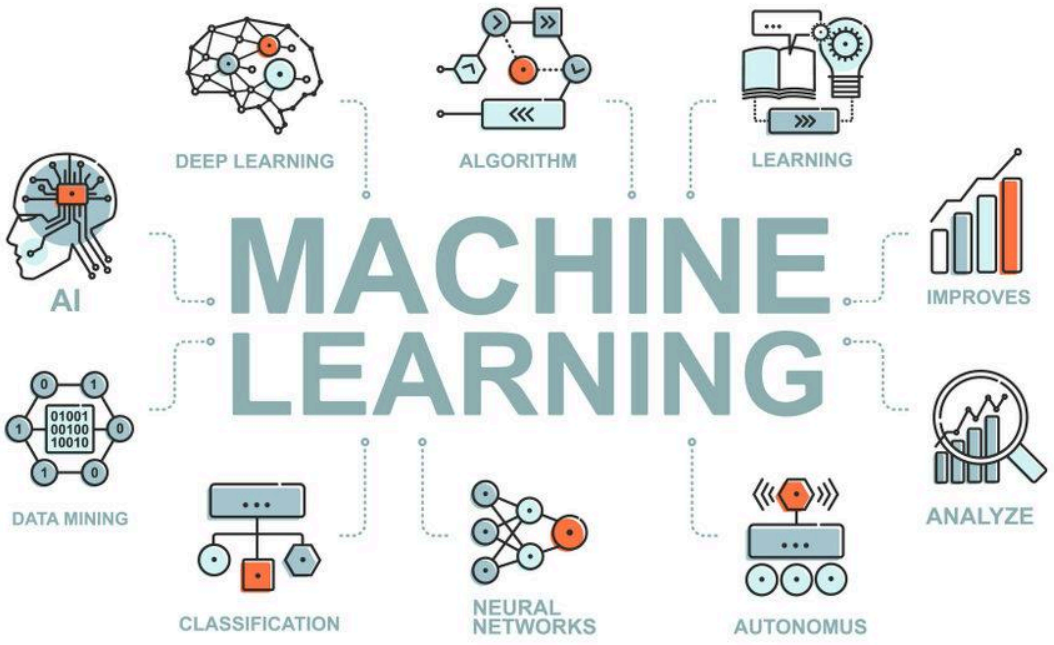


Table of Contents

Chapter-1

Introduction to machine learning

Instructor Name: B N V Narasimha Raju

1.1 Introduction

Machine learning is a subfield of artificial intelligence (AI). Machine learning is the systematic study of algorithms and systems that gain knowledge or performance with experience.

A computer program is said to be learning from experience *E* if its performance measure *P* on task *T* improves with experience *E*. A computer program that learns from experience is called a machine learning program, or simply a learning program. Machine learning powers everything from translation apps to self-driving cars. It offers a way to solve problems and answer complex questions. It is basically a process of training a piece of software called an algorithm or model to make useful predictions from data.

1.2 Importance of Machine Learning

Machine learning concepts are used almost everywhere, such as in healthcare, finance, infrastructure, marketing, self-driving cars, recommendation systems, chatbots, social sites, gaming, cyber security, and many more.

Currently, machine learning is in the development phase, and many new technologies are continuously being added to machine learning. It helps us in many ways, such as analyzing large chunks of data, data extractions, interpretations, etc. Hence, there are a number of uses for machine learning.

1.3 Types of Machine Learning

Machine learning implementations are classified into four major categories:

- Supervised learning
- Unsupervised learning
- Reinforcement learning
- Semi-supervised learning

1.3.1 Supervised learning

Supervised learning is the machine learning task of learning a function that maps an input to an output, for example, input-output pairs. The given data is labeled. Both classification and regression problems are supervised learning problems. The training process continues until the model achieves the desired level of accuracy on the training data.

A machine is said to be learning from past experiences with respect to some class of tasks if its performance in a given task improves with the experience. For example, assume that a machine has to predict whether a customer will buy a specific product, let's say "antivirus," this year or not. The machine will do it by looking at the previous knowledge or past experiences, i.e., the data of products that the customer has bought every year, and if he buys antivirus every year, then there is a high probability that the customer is going to buy antivirus this year as well. This is how machine learning works at the conceptual level.

Supervised learning is when the model is getting trained on a labeled dataset. A labeled dataset is one that has both input and output parameters. In this type of learning, both training and validation are done, and datasets are labeled as shown in Figure 1.1.

User ID	Gender	Age	Salary	Purchased
15624510	Male	19	19000	0
15810944	Male	35	20000	1
15668575	Female	26	43000	0
15603246	Female	27	57000	0
15804002	Male	19	76000	1
15728773	Male	27	58000	1
15598044	Female	27	84000	0
15694829	Female	32	150000	1
15600575	Male	25	33000	1
15727311	Female	35	65000	0
15570769	Female	26	80000	1
15606274	Female	26	52000	0
15746139	Male	20	86000	1
15704987	Male	32	18000	0
15628972	Male	18	82000	0
15697686	Male	29	80000	0
15733883	Male	47	25000	1

Figure A: CLASSIFICATION

Temperature	Pressure	Relative Humidity	Wind Direction	Wind Speed
10.69261758	986.882019	54.19337313	195.7150879	3.278597116
13.59184184	987.8729248	48.0648859	189.2951202	2.909167767
17.70494885	988.1119385	39.11965597	192.9273834	2.973036289
20.95430404	987.8500366	30.66273218	202.0752869	2.965289593
22.9278274	987.2833862	26.06723423	210.6589203	2.798230886
24.04233986	986.2907104	23.46918024	221.1188507	2.627005816
24.41475295	985.2338867	22.25082295	233.7911987	2.448749781
23.93361956	984.8914795	22.35178837	244.3504333	2.454271793
22.68800023	984.8461304	23.7538641	253.0864716	2.418341875
20.56425726	984.8380737	27.07867944	264.5071106	2.318677425
17.76400389	985.4262085	33.54900114	280.7827454	2.343950987
11.25680746	988.9386597	53.74139903	68.15406036	1.650191426
14.37810685	989.6819458	40.70884681	72.62069702	1.553469896
18.45114201	990.2960205	30.85038484	71.70604706	1.005017161
22.54895853	989.9562988	22.81738811	44.66042709	0.264133632
24.23155922	988.796875	19.74790765	318.3214111	0.329656571

Figure B: REGRESSION

Figure 1.1: Classification and Regression

While training the model, data is usually split in the ratio of 80:20, i.e., 80% as training data and the rest as testing data. In training data, we feed input as well as output for 80% of the data. The model learns from training data only. We use different machine learning algorithms to build our model. Learning means that the model will build some logic of its own.

Once the model is ready, it is good to be tested. At the time of testing, the input is fed from the remaining 20% of data that the model has never seen before. The model will predict some value, and we will compare it with the actual output and calculate the accuracy.

1.3.1.1 Types of Supervised Learning

Classification: It is a supervised learning task where the output has defined labels (discrete values). For example, in the above Figure A, Output – Purchased has defined labels, i.e., 0 or 1; 1 means the customer will purchase, and 0 means that the customer won't purchase. The goal here is to predict discrete values belonging to a particular class and evaluate them on the basis of accuracy.

It can be either a binary or multi-class classification. In binary classification, the model predicts either 0 or 1, yes or no, but in the case of multi-class classification, the model predicts more than one class. Example: Gmail classifies mail into more than one class, like social, promotions, updates, and forums.

Regression: It is a supervised learning task where the output has a continuous value. For example, in the above Figure B, Output – Wind Speed has no discrete value but is continuous in a particular range. The goal here is to predict a value as close to the actual output value as our model can, and then evaluation is done by calculating the error value. The smaller the error, the greater the accuracy of our regression model.

1.3.2 Unsupervised learning

Unsupervised learning is a type of machine learning algorithm used to draw inferences from datasets consisting of input data without labeled responses. In unsupervised learning algorithms, classification, or categorization, is not included in the observations.

No labels are given to the learning algorithm, leaving it on its own to find structure in its input. It is used for clustering populations into different groups. Unsupervised learning can be a goal in itself (discovering hidden patterns in data).

Unsupervised machine learning analyzes and clusters unlabeled datasets using machine learning algorithms. These algorithms find hidden patterns and data without any human intervention, i.e., we don't give output to our model. The training model has only input parameter values and discovers the groups or patterns on its own.

Data-set in figure 1.2 is mall data that contains information about its clients who subscribe to them. Once subscribed, they are provided a membership card, and the mall has complete information about the customer and his/her every purchase. Now, using this data and unsupervised learning techniques, the mall can easily group clients based on the parameters we are feeding in.

CustomerID	Genre	Age	Annual Income (k$)	Spending Score (1-100)
1	Male	19	15	39
2	Male	21	15	81
3	Female	20	16	6
4	Female	23	16	77
5	Female	31	17	40
6	Female	22	17	76
7	Female	35	18	6
8	Female	23	18	94
9	Male	64	19	3
10	Female	30	19	72
11	Male	67	19	14
12	Female	35	19	99
13	Female	58	20	15
14	Female	24	20	77
15	Male	37	20	13
16	Male	22	20	79
17	Female	35	21	35

Figure 1.2: Mall data

The input to the unsupervised learning models is as follows:

- **Unstructured data**: It may contain noisy data, missing values, or unknown data.
- **Unlabeled data**: data only contains a value for input parameters; there is no targeted value (output). It is easier to collect as compared to the labeled one in the supervised approach.

1.3.2.1 Types of Unsupervised Learning

Clustering: Broadly, this technique is applied to group data based on different patterns, such as similarities or differences, that our machine model finds. These algorithms are used to process raw, unclassified data objects into groups. For example, in the above figure, we have not given output parameter values, so this technique will be used to group clients based on the input parameters provided by our data.

Association: This technique is a rule-based ML technique that finds out some very useful relations between parameters of a large data set. This technique is basically used for market basket analysis, which helps to better understand the relationship between different products. For example, shopping stores use algorithms based on this technique to find out the relationship between the sale of one product and another's sales based on customer behavior. If a customer buys milk, he may also buy bread, eggs, or butter. Once trained well, such models can be used to increase sales by planning different offers.

1.3.3 Reinforcement learning

Reinforcement learning is the problem of getting an agent to act in the world so as to maximize its rewards. A learner is not told what actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them.

For example, consider teaching a dog a new trick: we cannot tell him what to do or what not to do, but we can reward/punish it if it does the right/wrong thing. A computer program interacts with a dynamic environment in which it must perform a certain goal (such as driving a vehicle or playing a game against an opponent). The program receives feedback in terms of rewards and punishments as it navigates its problem space.

1.3.4 Semi-supervised learning

Where an incomplete training signal is given: a training set with some (often many) of the target outputs missing. There is a special case of this principle known as transduction, where the entire set of problem instances is known at learning time, except that some of the targets are missing.

Semi-supervised learning is an approach to machine learning that combines small labeled data with a large amount of unlabeled data during training. Semi-supervised learning falls between unsupervised learning and supervised learning. For example, a photo archive where only some of the images are labeled (e.g., dog, cat, person) and the majority are unlabeled.

1.4 Designing a Learning System

Machine learning enables a machine to automatically learn from data, improve performance from an experience, and predict things without being explicitly programmed. When we feed the training data to machine learning algorithms, this algorithm will produce a mathematical model, and with the help of the mathematical model, the machine will make a prediction and take a decision without being explicitly programmed. Also, during training data, the more machines will work with it, the more experience it will gain, and the more efficient the result will be, as shown in Fig. 1.3.

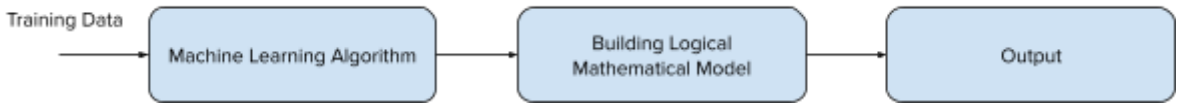


Figure 1.3: Learn from Data

For example, in driverless cars, the training data is fed to algorithms like how to drive a car on the highway or on busy and narrow streets with factors like speed limit, parking, stop at signal, etc. After that, a logical and mathematical model is created on the basis of that, and after that, the car will work according to the logical model. Also, the more data is fed, the more efficient output is produced.

A computer program is said to be learning from experience *E* if its performance measure *P* on task *T* improves with experience *E*. For example, in the Spam EMail Detection

- Task, *T*: To classify mails into spam or not spam.
- Performance measure, *P*: Total percent of emails being correctly classified as being "spam" or "not spam."
- Experience, *E*: Set of Mails with the Label "spam".

1.4.1 Steps for Designing Learning System

- Choosing the Training Experience
- Choosing target function
- Choosing Representation for Target function
- Choosing Function Approximation Algorithm
- Final Design

1.4.1.1 Choosing the Training Experience

The very important first task is to choose the training data or training experience that will be fed to the machine learning algorithm. It is important to note that the data or experience that we feed to the algorithm must have a significant impact on the success or failure of the model. So training data or experience should be chosen wisely. Below are the attributes which will impact the success and failure of data

- The training experience will be able to provide direct or indirect feedback regarding choices. For example: while playing chess, the training data will provide feedback to itself for choosing a move to increase the chance of success.
- The second important attribute is the degree to which the learner will control the sequences of training examples. For example: when training data is fed to the machine, at that time accuracy is very low, but when it gains experience while playing again and again, the machine algorithm will get feedback and control the chess game accordingly.
- The third important attribute is how it will represent the distribution of examples over which performance will be measured. For example, a machine learning algorithm will gain experience while going through a number of different cases and different examples. Thus, machine learning algorithms will gain more and more experience by passing through more and more examples, and hence, their performance will increase.

1.4.1.2 Choosing target function

The next important step is choosing the target function. It means that according to the knowledge fed to the algorithm, the machine learning will choose the NextMove function, which will describe what type of legal moves should be taken. For example : While playing chess with the opponent, when the opponent plays, the machine learning algorithm will decide the number of possible legal moves to take in order to get success.

1.4.1.3 Choosing Representation for Target function

When the machine algorithm knows all the possible legal moves, the next step is to choose the optimized move using any representation, i.e., linear equations, hierarchical graph representations, tabular forms, etc. The NextMove function will move the target move like this out of these moves, which will provide a higher success rate. For Example : while playing chess, machines have four possible moves, so the machine will choose the optimal move that will bring it success.

1.4.1.4 Choosing Function Approximation Algorithm

An optimized move cannot be chosen just from the training data. The training data had to go through a set of examples, and through these examples, the training data will approximate which steps are chosen, and after that, the machine will provide feedback on it. For Example: when training data for playing chess is fed to an algorithm, it is not a machine algorithm that will fail or get success, and again, from that failure or success, it will measure what step should be chosen and what the success rate is.

1.4.1.5 Final Design

The final design is created at last when the system goes through a number of examples, failures and success, correct and incorrect decisions, what will be the next step, etc. For example, DeepBlue is an intelligent computer that won an ML-based chess game against a chess expert, and it became the first computer to beat a human chess expert.

1.5 Issues in Machine Learning

In machine learning, there is a process of analyzing data to build or train models. It is used everywhere, and it holds great value throughout. There are a lot of challenges that machine learning professionals face to inculcate ML skills and create an application from scratch.

1.5.1 Poor Quality of Data

Data plays a significant role in the machine learning process. One of the significant issues that machine learning professionals face is the absence of good quality data. Unclean and noisy data can make the whole process extremely exhausting. We don't want our algorithm to make inaccurate or faulty predictions. Hence, the quality of the data is essential to enhancing the output. Therefore, we need to ensure that the process of data preprocessing, which includes removing outliers, filtering missing values, and removing unwanted features, is done with the utmost level of perfection.

1.5.2 Underfitting of Training Data

This process occurs when data is unable to establish an accurate relationship between input and output variables. It simply means trying to fit in undersized jeans. It signifies that the data is too simple to establish a precise relationship. To overcome this issue:

- Maximize the training time
- Enhance the complexity of the model
- Add more features to the data
- Reduce regular parameters

1.5.3 Overfitting of Training Data

Overfitting refers to a machine learning model trained with a massive amount of data that negatively affects its performance. It is like trying to fit into oversized jeans. Unfortunately, this is one of the significant issues faced by machine learning professionals. This means that the algorithm is trained with noisy and biased data, which will affect its overall performance. We can tackle this issue by:

- Analyzing the data with the utmost level of perfection
- Use data augmentation technique
- Remove outliers in the training set
- Select a model with fewer features

1.5.4 Machine Learning is a Complex Process

The machine learning industry is young and is continuously changing. Rapid hit and trial experiments are being carried out. The process is transforming, and hence there are high chances of error, which makes the learning complex. It includes analyzing the data, removing data bias, training the data, applying complex mathematical calculations, and a lot more. Hence, it is a really complicated process, which is another big challenge for machine learning professionals.

1.5.5 Lack of Training Data

The most important task you need to do in the machine learning process is to train the data to achieve an accurate output. Less training data will produce inaccurate or too biased predictions. But a machine-learning algorithm needs a lot of data to distinguish. For complex problems, it may even require millions of pieces of data to be trained. Therefore, we must ensure that machine learning algorithms are trained with sufficient amounts of data.

1.5.6 Slow Implementation

This is one of the common issues faced by machine learning professionals. The machine learning models are highly efficient in providing accurate results, but it takes a tremendous amount of time. Slow programs, data overload, and excessive requirements usually take a lot of time to provide accurate results. Further, it requires constant monitoring and maintenance to deliver the best output.

1.5.7 Imperfections in the Algorithm When Data Grows

So you have found quality data, trained it amazingly, and the predictions are really concise and accurate. The model may become useless in the future as data grows. The best model of the present may become inaccurate in the coming future and require further rearrangement. So you need regular monitoring and maintenance to keep the algorithm working. This is one of the most exhausting issues faced by machine learning professionals.

1.6 A Machine Learning Sampler

Most current email clients incorporate algorithms to identify and filter out spam email, also known as junk e-mail or unsolicited bulk e-mail. Earlier spam filters relied on hand-coded pattern matching techniques such as regular expressions and so-on. Machine learning techniques have achieved additional adaptivity and flexibility.

SpamAssassin is a widely used open-source spam filter. It calculates a score for an incoming e-mail, based on a number of built-in rules and adds a 'junk' flag and a summary report to the email headers if the score is 5 or more. Here is an example report for an email:

0.6	HTML_IMAGE_RATIO_02	BODY: HTML has a low ratio of text to image area
1.2	TVD_FW_GRAPHIC_NAME_MID	BODY: TVD_FW_GRAPHIC_NAME_MID
1.4	SARE_GIF_ATTACH	FULL: Email has an inline gif

From left to right, you see the score attached to a particular test, the test identifier, and a short description, including a reference to the relevant part of the e-mail. Scores for individual tests can be negative as well as positive. The overall score of 5.3 suggests the email might be spam. If the email was a notification from an intermediate server, which had a whopping score of 14.6, it was rejected as spam. This 'bounce' message included the original message and therefore inherited some of its characteristics, such as a low text-to-image ratio, which pushed the score over the threshold of 5.

SpamAssassin determines the scores or 'weights' for each of the dozens of tests by using machine learning. Suppose we have a large 'training set' of emails that have been hand-labeled spam or ham, and we know the results of all the tests for each of these emails.

The goal is now to come up with a weight for every test, such that all spam e-mails receive a score above 5, and all ham emails get less than 5. There are a number of machine learning techniques that solve exactly this problem. For the moment, a simple example will illustrate the main idea.

Linear classification: Suppose we have only two tests and four training emails, one of which is spam (see Table 1.1).

E-mail	x_1	x_2	Spam?	$4x_1 + 4x_2$
1	1	1	1	8
2	0	0	0	0
3	1	0	0	4
4	0	1	0	4

Table 1.1: A small training set for SpamAssassin.

The columns marked x_1 and x_2 indicate the results of two tests on four different emails. The fourth column indicates which of the emails are spam. The right-most column demonstrates that by thresholding the function $4x_1 + 4x_2$ at 5, we can separate spam from ham.

Both tests succeed for the spam email; for one ham e-mail neither test succeeds; for another, the first test succeeds and the second doesn't; and for the third ham email, the first test fails and the second succeeds. It is easy to see that assigning both tests a weight of 4 correctly 'classifies' these four emails into spam and ham. In mathematical notation, the classifier can be described as $4x_1 + 4x_2 > 5$.

There are a number of useful ways in which we can express the SpamAssassin classifier in mathematical notation. If we denote the result of the *i-th* test for a given email as x_i , where x_i = 1 if the test succeeds and 0 otherwise, and we denote the weight of the *i-th* test as w_i, then the total score of an email can be expressed as $\sum_{i=1}^{n} w_i x_i$, making use of the fact that w_i contributes to the sum only if x_i = 1, i.e., if the test succeeds for the email. Using t for the threshold above which an email is classified as spam (5 in our example), the 'decision rule' can be written as $\sum_{i=1}^{n} w_i x_i > t$.

The notation can be simplified by means of linear algebra, writing w for the vector of weights $(w_1, \ldots, w_n)$ and x for the vector of test results $(x_1, \ldots, x_n)$. The above inequality can then be written using a dot product: $w \cdot x > t$. Changing the inequality to an equality $w \cdot x = t$, we obtain the 'decision boundary', separating spam from ham. The decision boundary is a plane (a 'straight' surface) in the space spanned by the x_i variables because of the linearity of the left-hand side. The vector w is perpendicular to this plane and points in the direction of spam. Figure 1.4 visualizes this for two variables.

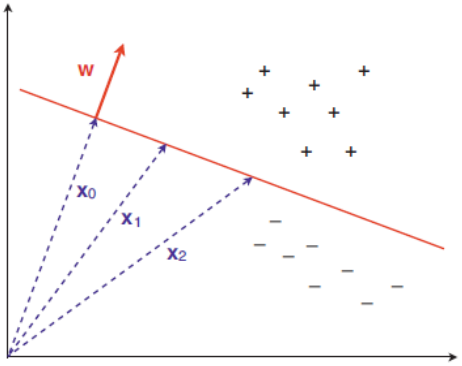


Figure 1.4 An example of linear classification in two dimensions.

The straight line separates the positives from the negatives. It is defined by $w \cdot x_i = t$. In fact, any weight between 2.5 and 5 will ensure that the threshold of 5 is only exceeded when both tests succeed. We could even consider assigning different weights to the tests.

SpamAssassin learns to recognise spam e-mail from examples and counter-examples. Moreover, the more training data is made available, the better SpamAssassin will become at this task. In the case of SpamAssassin, the 'experience' it learns from is some correctly labeled training data, and 'performance' refers to its ability to recognise spam email. A schematic view of how machine learning feeds into the spam e-mail classification task is given in Figure 1.5.

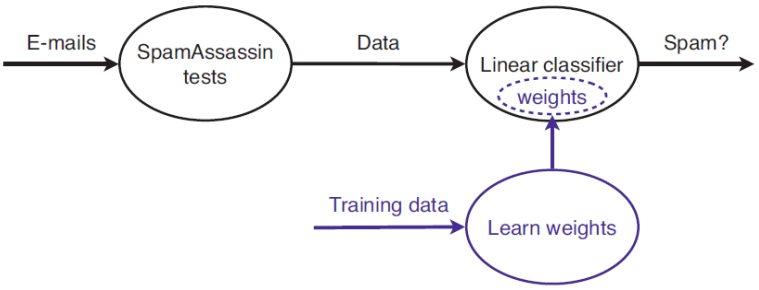


Figure 1.5: At the top, we see how SpamAssassin approaches the spam email classification task

The text of each email is converted into a data point by means of SpamAssassin's built-in tests, and a linear classifier is applied to obtain a 'spam or ham' decision. At the bottom (in blue), we see the bit that is done by machine learning.

We have already seen that a machine learning problem may have several solutions, even a simple one. Then how to choose among these solutions. One way is that we don't really care that much about performance on training data; we already know which of those emails are spam! What we care about is whether future emails are going to be classified correctly. While this appears to lead into a vicious circle—in order to know whether an email is classified correctly, I need to know its true class, but as soon as I know it's true class, I don't need the classifier anymore—it is important to keep in mind that good performance on training data is

only a means to an end, not a goal in itself. In fact, trying too hard to achieve good performance on the training data can easily lead to a fascinating but potentially damaging phenomenon called overfitting.

Overfitting: Imagine you are preparing for your machine learning exam. Previous exam papers and their answers are available online. You begin by trying to answer the questions from previous papers and comparing your answers with the model answers provided. Unfortunately, you spend all your time memorizing the model answers to all past questions. Now, if the upcoming exam completely consists of past questions, you are certain to do very well. But if the new exam asks different questions, you would be ill-prepared and get a much lower mark. In this case, one could say that you were overfitting the past exam papers and that the knowledge gained didn't generalize to future exam questions.

Generalization is probably the most fundamental concept in machine learning. If the knowledge that SpamAssassin has obtained from its training data carries over to your emails, you are happy; if not, you start looking for a better spam filter. However, overfitting is not the only possible reason for poor performance on new data. It may just be that the training data used by the SpamAssassin programmers to set its weights is not appropriate. The solution to the problem is to use different training data that exhibits the same characteristics.

Bayesian Classifier: Many spam email filters employ text classification techniques. Broadly speaking, such techniques maintain a vocabulary of words and phrases that are potential spam or ham indicators. For each of those words and phrases, statistics are collected from a training set. For instance, suppose that the word 'Free Ipod' occurred in four spam emails and in one ham email. If we then encounter a new email that contains the word 'Free Ipod', we might reason that the odds that this email is spam are 4:1, or the probability of it being spam is 0.80 and the probability of it being ham is 0.20.

The essence of rule-based classifiers is that they don't treat all emails in the same way but work on a case-by-case basis. In each case, they only invoke the most relevant features. Cases can be defined by several nested features

Does the email contain the word 'Free Ipod'?

- If so, does the email contain the word 'lottery'?
    - If so, estimate the odds of spam at 12:1.
    - If not, estimate the odds of spam at 4:1.
- If not, does the email contain the word 'lottery'?
    - If so, estimate the odds of spam at 3:1.
    - If not, estimate the odds of spam at 1:6.

We have seen practical examples of machine learning in spam email recognition. Machine learners call such a task binary classification, as it involves assigning objects (e-mails) to one of two classes: spam or ham. This task is achieved by describing each email in terms of a

number of variables or features. In the SpamAssassin example, these features were handcrafted by an expert in spam filtering, while in the Bayesian text classification example, we employed a large vocabulary of words. The question is then how to use the features to distinguish spam from ham. We have to somehow figure out a connection between the features and the class—machine learners call such a connection a model—by analyzing a training set of emails already labeled with the correct class.

Here we have, then, the main ingredients of machine learning: tasks, models, and features. Figure 1.6 shows how these ingredients relate. A task (red box) requires an appropriate mapping—a model—from data described by features to outputs. Obtaining such a mapping from training data is what constitutes a learning problem (blue box).

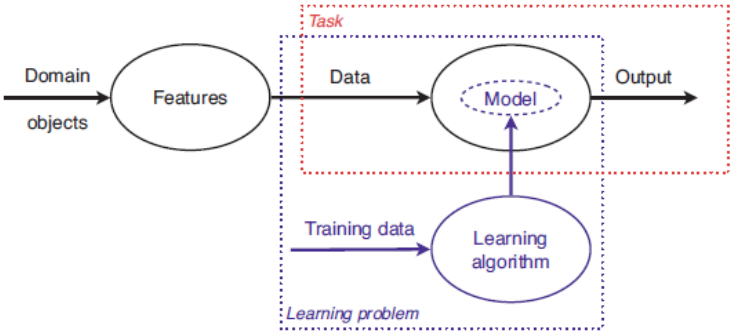


Figure 1.6. An overview of how machine learning is used to address a given task.

Chapter-2

The ingredients of machine learning

Instructor Name: B N V Narasimha Raju

Machine learning is all about using the right features to build the right models that achieve the right tasks. In essence, features define a 'language' in which we describe the relevant objects in our domain. A task is an abstract representation of a problem we want to solve regarding those domain objects. Many of these tasks can be represented as a mapping from data points to outputs. This mapping or model is itself produced as the output of a machine learning algorithm applied to training data.

2.1 Tasks: the problems that can be solved with machine learning

Learning from labeled data is called supervised learning. Both classification and regression problems are supervised learning problems. Spam email recognition constitutes a binary classification task that distinguishes between spam and ham emails. Consider classification problems with more than two classes. For instance, we may want to distinguish between different kinds of ham emails, e.g., work-related emails and private messages. It could be a combination of two binary classification tasks: the first task is to distinguish between spam and ham, and the second task is, among ham emails, to distinguish between work-related and private ones. Some spam emails will look like private messages, so some useful information will be lost. For this reason, utilizing multi-class classification as a machine learning task will be beneficial.

Sometimes it is useful to assess an incoming email's urgency on a sliding scale. This task is called regression and essentially involves learning a real-valued function from training examples labeled with true function values. For example, create a training set by randomly picking several emails from the inbox and labelling them with an urgency score ranging from 0 (ignore) to 10 (immediate action required). It works by choosing a class of functions and constructing a function that minimizes the difference between the predicted and true function values. It is different from SpamAssassin learning, where the training data is labeled with classes rather than 'true' spam scores.

Both classification and regression consist of a training set of examples labeled with true classes or function values. Providing the true labels for a data set is expensive. It is possible to distinguish spam from ham, or work emails from private messages, without a labeled training set. The task of grouping data without prior information on the groups is called clustering. Learning from unlabeled data is called unsupervised learning and is quite distinct from supervised learning. A clustering algorithm works by assessing the similarity between instances (e.g., emails) and putting similar instances in the same cluster and dissimilar instances in different clusters.

Measuring similarity: Email's similarity is determined by the words they share. For instance, take the number of common words in two emails and divide it by the number of words occurring in either email. This measure is called the Jaccard coefficient. Suppose that one email contains 42 (different) words and another contains 112 words, and the two emails have 23 words in common, then their similarity would be $\frac{23}{42+112-23} = \frac{23}{130} = 0.18$. We can then cluster our emails into groups, such that the average similarity of an email to the other emails in its group is much larger than the average similarity to emails from other groups.

There are many other patterns that can be learned from data in an unsupervised way. Association rules are a kind of pattern that are popular in marketing applications, and the result of such patterns can often be found on online shopping websites. Data mining algorithms find associations by identifying the items that appear frequently together. These algorithms work by only considering items that occur a minimum number of times. More interesting associations could be found by considering multiple items in your shopping basket. There are many other types of associations that can be learned and exploited.

2.1.1 Looking for structure

Like all other machine learning models, patterns are manifestations of the underlying structure of the data. Sometimes this structure takes the form of a single hidden or latent variable. Consider the following matrix:

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 2 \\ 1 & 0 & 1 & 1 \\ 0 & 2 & 2 & 3 \end{pmatrix}$$

Imagine these ratings by six different people (in rows), on a scale of 0 to 3, of four different films, say The Shawshank Redemption, The Usual Suspects, The Godfather, and the Big Lebowski (in columns, from left to right). The Godfather seems to be the most popular of the four, with an average rating of 1.5, and The Shawshank Redemption is the least appreciated, with an average rating of 0.5.

In this matrix, the columns or rows are combinations of other columns or rows. For instance, the third column turns out to be the sum of the first and second columns. Similarly, the fourth row is the sum of the first and second rows. What this means is that the fourth person combines the ratings of the first and second persons. Similarly, The Godfather's ratings are the sum of the ratings of the first two films. This is more clear by writing the matrix as the following product:

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 2 & 2 & 2 \\ 0 & 0 & 0 & 1 \\ 1 & 2 & 3 & 2 \\ 1 & 0 & 1 & 1 \\ 0 & 2 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

These matrices have a very natural interpretation in terms of film genres. The right-most matrix associates films (in columns) with genres (in rows): The Shawshank Redemption and The Usual Suspects belong to two different genres, say drama and crime; The Godfather belongs to both; and The Big Lebowski is a crime film that also introduces a new genre (say comedy). The tall, 6-by-3 matrix then expresses people's preferences in terms of genres: the first, fourth, and fifth person like drama, the second, fourth, and fifth person like crime films, and the third, fifth, and sixth person like comedies. Finally, the middle matrix states that the crime genre is twice as important as the other two genres in terms of determining people's preferences.

Methods for discovering hidden variables such as film genres really come into their own when the number of values of the hidden variable (here: the number of genres) is much smaller than the number of rows and columns of the original matrix. If the model output involves the target variable, we call it a predictive model, and a descriptive model if it does not. This leads to the four different machine learning settings summarized in Table 2.1.

Table 2.1: An overview of different machine learning settings.

	Predictive model	*Descriptive model*
Supervised learning	classification, regression	subgroup discovery
Unsupervised learning	predictive clustering	descriptive clustering, association rule discovery

The most common setting is supervised learning of predictive models. Typical tasks are classification and regression. It is also possible to use labeled training data to build a descriptive model that is not primarily intended to predict the target variable but instead identifies, say, subsets of the data that behave differently with respect to the target variable. This example of supervised learning using a descriptive model is called subgroup discovery.

Descriptive models can naturally be learned in an unsupervised setting; a few examples are clustering, association rule discovery, and matrix decomposition. A typical example of unsupervised learning of a predictive model occurs when we cluster data with the intention of using the clusters to assign class labels to new data.

2.1.2 Evaluating performance on a task

Assume that the perfect spam e-mail filter doesn't exist. In many cases, the data is 'noisy', examples may be mislabeled, or features may contain errors. If it tries hard to find a model that correctly classifies the training data, it will lead to overfitting. In some cases, the features of the data will give an indication of what their class might be, but they may not predict the class perfectly. For these and other reasons, machine learners take the performance evaluation of learning algorithms very seriously. We need to have some idea of how well an algorithm is expected to perform on new data, not in terms of runtime or memory usage but in terms of classification performance (if our task is a classification task).

Suppose we want to find out how well our newly trained spam filter does. One thing we can do is count the number of correctly classified emails, both spam and ham, and divide that by the total number of examples to get a proportion, which is called the accuracy of the classifier. However, this doesn't indicate whether overfitting is occurring. A better idea would be to use only 90% (say) of the data for training and the remaining 10% as a test set. If overfitting occurs, the test set performance will be considerably lower than the training set performance. In practice, this train-test split is repeated in a process called cross-validation. This works as follows: we randomly divide the data into ten parts of equal size and use nine parts for training and one part for testing. We do this ten times, using each part once for testing. At the end, we compute the average test set performance. Cross-validation can also be applied to other supervised learning problems, but unsupervised learning methods typically need to be evaluated differently.

2.2 Models: the output of machine learning

Models form the central concept in machine learning. Models are being learned from the data in order to solve a given task. There are three groups of models. They are

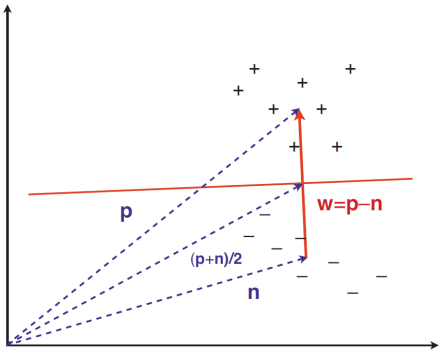
- Geometric models.
- Probabilistic models.
- Logical models.

2.2.1 Geometric models

The instance space is the set of all possible instances, whether they are present in our data set or not. A geometric model is constructed in instance space using geometric concepts

such as lines, planes, and distances. For example, the linear classifier is also a geometric classifier. One main advantage of geometric classifiers is that they are easy to visualize if they consist of two or three dimensions. The instance space has as many coordinates as features. Generally there will be more features,so the instance space will have more coordinates.

If there exists a linear decision boundary separating the two classes, we say that the data is linearly separable. As we have seen, a linear decision boundary is defined by the equation $w \cdot x = t$, where w is a vector perpendicular to the decision boundary, x is an arbitrary point on the decision boundary, and t is the decision threshold. The vector w is pointing from the 'center of mass' of the negative examples, n, to the center of mass of the positive examples, p. It is described by the equation $w \cdot x = t$, with $w = p - n$; the decision threshold can be found by noting that $(p+n) / 2$ is on the decision boundary, and hence $t = (p - n) \cdot (p + n)/2 = (\|p\|^2 - \|n\|^2) / 2$ ($\|x\|$ denotes the length of vector x). By setting the decision threshold appropriately, it can intersect the line from n to p half-way (Figure 2.1). This is called a basic linear classifier.



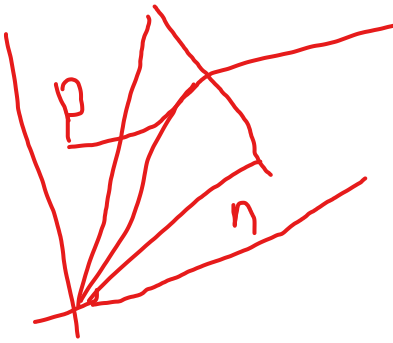


Figure 2.1: The basic linear classifier constructs a decision boundary by half-way intersecting the line between the positive and negative centers of mass.

However, if those assumptions do not hold, the basic linear classifier can perform poorly; for instance, it may not perfectly separate the positives from the negatives, even if the data is linearly separable. Because data is usually noisy. Consider a large vocabulary, say 10000 words. This means that the instance space has 10000 dimensions, but for any one document a small percentage of the features will be non-zero. As a result there is much 'empty space' between instances, which increases the possibility of linear separability. One option is to prefer large margin classifiers, where the margin is the distance between the decision boundary and the closest instance. Support vector machines (SVM) are a powerful kind of linear classifier that find a decision boundary whose margin is as large as possible.

A very useful geometric concept in machine learning is distance. If the distance between two instances is small, then the instances are similar, and so nearby instances would be expected to receive the same classification or belong to the same cluster. Distance can be measured by Euclidean distance, which is the square root of the sum of the squared distances along each coordinate $\sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$. A very simple distance based classifier works as follows: to

classify a new instance, retrieve the most similar training instance from memory with the smallest Euclidean distance from the instance to be classified, and simply assign that training instance's class. This classifier is known as the nearest-neighbor classifier. A variation of this classifier can retrieve the *k* most similar training instances and take a vote (*k*-nearest neighbor). This is simple yet powerful.

Suppose data is to be clustered into *K* clusters, and there is an initial guess of how the data should be clustered. Now calculate the means of each initial cluster and reassign each point to the nearest cluster mean. Unless our initial guess was wrong, this will have changed some of the clusters, so we repeat these two steps until no change occurs. This clustering algorithm, which is called *K*-means, is very widely used to solve a range of clustering tasks. The initial guess is usually done randomly: either by randomly partitioning the data set into *K* 'clusters' or by randomly guessing *K* 'cluster centers'.

2.2.2 Probabilistic models

Let *X* denote the variables (e.g., the instance's feature values) and let *Y* denote the target variables (e.g., the instance's class). The key issue in machine learning is to model the relationship between *X* and *Y*. The statistician's approach is a random process that generates the values for these variables according to an unknown probability distribution.

Since *X* is known for a particular instance, *Y* may not be. The conditional probabilities are *P*(*Y* | *X*). For instance, *Y* could indicate whether the email is spam, and *X* could indicate whether the email contains the words *free iPod* and *lottery*. The probability of interest is then *P*(*Y* | *free iPod*, *lottery*), where *free iPod* and *lottery* are two boolean variables of the feature vector *X*. For a particular email, the feature values are known, so the *P*(*Y* | *free ipod* = 1, *lottery* = 0) if the email contains the word *free iPod* but not the word *lottery*. This is called a posterior probability because it is used after the features *X* are observed.

Table 2.2 shows an example of a distributed posterior distribution. *Free iPod* and *lottery* are two Boolean features. *Y* is the class variable, with values of *spam* and *ham*. In each row, the most likely class is indicated in bold.

Free ipod	lottery	P(Y=spam/Free iPod, lottery)	P(Y=ham/Free iPod, lottery)
0	0	0.31	**0.69**
0	1	**0.65**	0.35
1	0	**0.80**	0.20
1	1	0.40	**0.60**

Table 2.2. An example of posterior distribution.

Assuming that *X* and *Y* are known variables, the posterior distribution *P*(*Y* | *X*) helps to answer many questions. For instance, to classify a new email, the corresponding probability *P*(*Y*=*spam* | *free iPod*, *lottery*) is used and predicts spam if this probability exceeds 0.5 and ham otherwise. Such a way to predict a value of *Y* on the basis of the values of *X* and the posterior distribution *P*(*Y* | *X*) is called a decision rule.

Bayes' rule states that

$$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}$$

Here, *P*(*Y*) is the prior probability, i.e., in the case of classification without observing the data *X*, it tells the objects belonging to which classes. *P*(*X*) is the probability of the data, which is independent of *Y*. *P*(*X* | *Y*) is a likelihood function. The first decision rule predicts the class with maximum posterior probability, which uses Bayes' rule can be written in terms of the likelihood function:

$$y_{\text{MAP}} = \arg\max_Y P(Y|X) = \arg\max_Y \frac{P(X|Y)P(Y)}{P(X)} = \arg\max_Y P(X|Y)P(Y)$$

This is usually called the maximum a posteriori (MAP) decision rule. Now, if we assume uniform prior distribution (i.e., *P(Y)* is the same for every value of *Y*), this reduces to the maximum likelihood (ML) decision rule

$$y_{\text{ML}} = \arg\max_Y P(X|Y)$$

A useful thumb rule is to use likelihoods if you want to ignore the prior distribution or assume it is uniform, and posterior probabilities otherwise.

If there are only two classes, it is convenient to work with ratios of posterior probabilities or likelihood ratios. To know how much the data favors one of two classes, calculate the posterior odds, for e.g.,

$$\frac{P(Y = \text{spam}|X)}{P(Y = \text{ham}|X)} = \frac{P(X|Y = \text{spam})}{P(X|Y = \text{ham})} \frac{P(Y = \text{spam})}{P(Y = \text{ham})}$$

The posterior odds are the product of the likelihood ratio and the prior odds. If the odds are greater than 1, then take the class as the class in the numerator; if it is smaller than 1, take the class in the denominator. In many cases, the prior odds are a simple constant factor that can be manually set, estimated from the data, or optimized to maximize performance on a test set.

If the likelihoods of individual words are independent within the same class, then it allows us to decompose the joint likelihood into a product of marginal likelihoods:

$$P(\textit{free iPod}, \textit{lottery} \mid Y) = P(\textit{free iPod} \mid Y)\, P(\textit{lottery} \mid Y)$$

The probabilities on the right are called marginal likelihoods. The independence assumption can also be called a product of marginal likelihoods 'naive'. The machine learners also refer to the simplified Bayesian classifier as naive Bayes.

2.2.3 Logical models

The third type of model is called ‘logical’ because models of this type can be easily translated into rules that are understandable by humans. Such rules are easily arranged in a tree structure, which is called a feature tree. The feature tree is constructed by iteratively selecting the features in the instance space. The leaves of the tree are represented in rectangular areas in the instance space, which we will call instance space segments. Depending on the task, the leaves can be labeled with a class, a probability, a real value, and so-on. Feature trees whose leaves are labeled with classes are called decision trees.

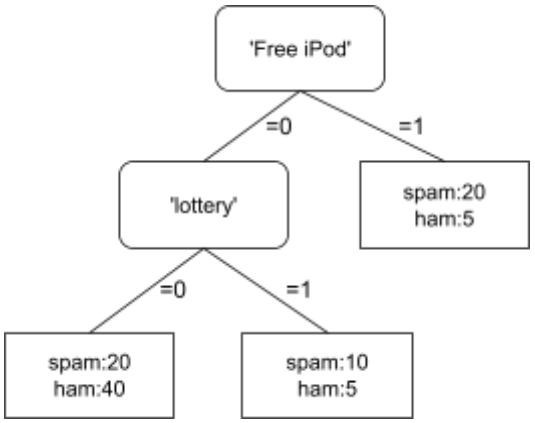


Figure 2.2:. A feature tree combining two Boolean features.

The feature tree is as shown in Figure 2.2. Each internal node or split is labeled with a feature, and each edge is labeled with a feature value. Each leaf, therefore, corresponds to a unique combination of feature values. The feature tree in Figure 2.2 is also called a decision tree because leaves are labeled with classes (e.g., spam and ham).

The feature tree in Figure 2.3 is a complete feature tree, which contains all features, one at each level of the tree (Figure 2.3). Naive Bayes made a wrong prediction in the rightmost leaf. Since the leaf covers only a single example, there is a danger of overfitting the data. So the previous tree is a better model. Decision tree learners will use pruning techniques to delete this type of split.

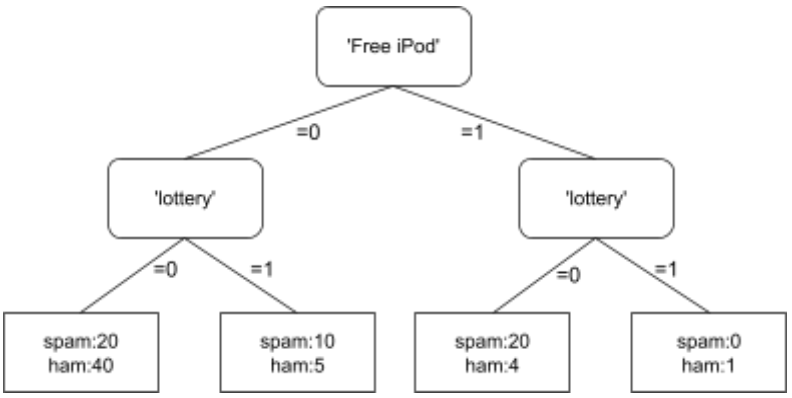


Figure 2.3: A complete feature tree built from two Boolean features

A feature list is a binary feature tree that always branches in the same direction, either left or right. The tree in figure 2.2 is a left-branching feature list. Such feature lists can be written as nested if–then–else statements. For instance, to label the leaves in Figure 2.2 by majority class, the following is the decision list:

if free iPod= 1 then Class = Y = spam.
else if lottery = 1 then Class = Y = spam·
else Class = Y = ham·

Logical models often have different, equivalent formulations. For instance, two alternative formulations for this model are

if Free iPod= 1 $\vee$ lottery = 1 then Class = Y = spam·
else Class = Y = ham·

if Free iPod= 0 $\wedge$ lottery = 0 then Class = Y = ham·
else Class = Y = spam·

The first formulation combines the two rules by means of disjunction (‘or’), denoted by $\vee$. This selects a single non rectangular area in the instance space. The second formulation is a conjunctive condition (‘and’, denoted by $\wedge$) for the opposite class (ham) and declares everything else as spam.

Tree-learning algorithms typically work in a top–down fashion. The first task is to find a good feature to split on at the top of the tree. Once the algorithm has found such a feature, the training set is partitioned into subsets, one for each node. For each of these subsets, again find a good feature to split on, and so-on. An algorithm that works by repeatedly splitting a problem into smaller subproblems is called a divide-and-conquer algorithm. Stop splitting a node when all training examples belonging to that node are of the same class.

Rule learning algorithms also work in a top–down fashion. A single rule is learned by repeatedly adding conditions to the rule until the rule only covers examples of a single class. Then remove the covered examples from that class and repeat the process. This is sometimes called a separate-and-conquer approach.

An interesting aspect of logical models is that they can provide explanations for their predictions to some extent. For example, a prediction assigned by a decision tree could be explained by reading off the conditions that led to the prediction from root to leaf. The model itself can also be easily inspected by humans, which is why it is sometimes called declarative.

2.2.4 Grouping and grading

There are three general types of models, and there are some underlying principles pertaining to each of these groups of models. The key difference between grouping models and grading models is the way they handle the instance space.

Grouping models do this by breaking up the instance space into groups or segments, the number of which is determined at training time. One could say that grouping models have a fixed and finite 'resolution' and cannot distinguish between individual instances beyond this resolution. The finest resolution is very simple, such as assigning the majority class to all instances that fall into the segment. The main emphasis of training a grouping model is then on determining the right segments.

Grading models, on the other hand, do not employ such a notion of segment. Rather than applying very simple local models, they form one global model over the instance space. Consequently, grading models are (usually) able to distinguish between arbitrary instances, no matter how similar they are. Their resolution is infinite, particularly when working in a cartesian instance space.

The distinction between grouping and grading models is relative rather than absolute, and some models combine both features. On the other end of the spectrum, regression trees combine grouping and grading features.

2.3 Features: the workhorses of machine learning

Features determine much of the success of a machine learning application, because a model is only as good as its features. A feature can be thought of as a kind of measurement that can be easily performed in any instance. Mathematically, they are functions that map from the instance space to some set of feature values called the domain of the feature. The feature domain contains real numbers, integers, booleans etc.

2.3.1 Two uses of features

The two uses of features are

- features as splits
- features as predictors

In features as splits, it is worth noting that features and models are intimately connected; models defined in terms of a single feature can be called a univariate model. We can therefore distinguish two uses of features that echo the distinction between grouping and grading models. A very common use of features, particularly in logical models, is to zoom in on a particular area of the instance space. Binary splits divide the instance space into two groups. One group satisfies the condition, and the other group doesn't satisfy the condition. Non-binary splits are also possible.

The use of features as predictors is a second use of features, particularly in supervised learning. A linear classifier employs a decision rule of the form $\sum_{i=1}^{n} w_i x_i > t$. where x_i is a numerical feature. The linearity of this decision rule means that each feature makes an independent contribution to the score of an instance. This contribution depends on the weight w_i. If this is large and positive, a positive x_i increases the score; if $w_i \ll 0$, a positive x_i decreases the score; if $w_i \approx 0$, x_i influence is negligible. Thus, the feature makes a precise and measurable contribution to the final prediction. These two uses of features—features as splits and features as predictors—are sometimes combined in a single model.

2.3.2 Feature construction and transformation

There is a lot of scope in machine learning for playing around with features. In the spam filter example and text classification more generally, the messages or documents don't come with built-in features; rather, they need to be constructed by the developer of the machine learning application. This feature construction process is absolutely crucial for the success of a machine learning application. Indexing an e-mail by the words that occur in it is a carefully engineered representation that manages to amplify the signal and remove the noise in spam e-mail filtering and related classification tasks.

However, it is easy to create problems, but this is the wrong thing to do: For instance, if we aim to train a classifier to distinguish between grammatical and ungrammatical sentences, word order is clearly signal rather than noise, and a different representation is called for.

It is often natural to build a model in terms of the given features. However, we are free to change the features or even introduce new features. For instance, real-valued features often contain unnecessary detail that can be removed by discretization. Imagine you want to analyze the body weight of a relatively small group of, say, 100 people by drawing a

histogram. If you measure everybody's weight in kilograms with one position after the decimal point (i.e., your precision is 100 grams), then your histogram will be sparse and spiky. It is hard to draw any general conclusions from such a histogram. It would be much more useful to discretize the body weight measurements in intervals of 10 kilograms.

In more extreme cases of feature construction, we transform the entire instance space. Consider Figure 2.4: the data on the left is clearly not linearly separable, but by mapping the instance space into a new 'feature space' consisting of the squares of the original features, we see that the data becomes almost linearly separable.

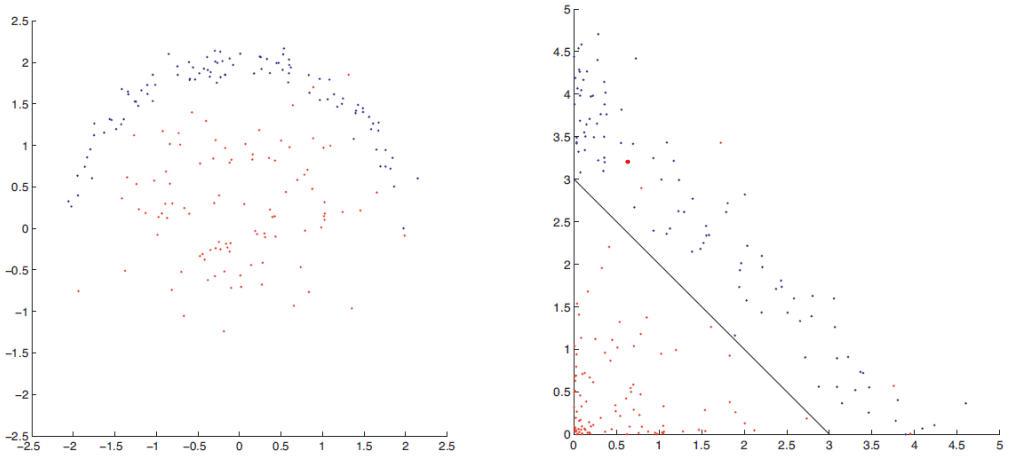


Figure 2.4: (left) A linear classifier would perform poorly on this data. (right) By transforming the original (x, y) data into (x, y) = (x^2, y^2)

Chapter-3

Preliminaries

Instructor Name: B N V Narasimha Raju

3.1 The curse of dimensionality

The curse of dimensionality basically refers to the difficulties a machine learning algorithm faces when working with data in the higher dimensions that did not exist in the lower dimensions. This happens because when you add dimensions (features), the minimum data requirements also increase rapidly. This means that as the number of features (columns) increases, you need an exponentially growing number of samples (rows) to have all combinations of feature values well-represented in our sample.

The essence of the curse is the realization that as the number of dimensions increases, the volume of the unit hypersphere does not increase with it. The unit hypersphere is the region we get if we start at the origin (the center of our coordinate system) and draw all the points that are distance 1 away from the origin. In 2 dimensions we get a circle of radius 1 around (0, 0) (drawn in Figure 3.1), and in 3D we get a sphere around (0, 0, 0) (Figure 3.2). The sphere does not reach as far into the corners as the circle does, and this gets more noticeable as the number of dimensions increases. In higher dimensions, the sphere becomes a hypersphere.

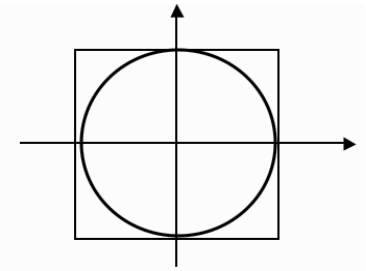


FIGURE 3.1 The unit circle in 2D with its bounding box.

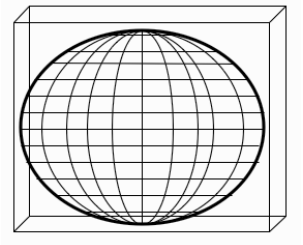


FIGURE 3.2 The unit sphere in 3D with its bounding cube.

The Figure 3.3 graph shows the size of the unit hypersphere for the first few dimensions, but also shows clearly that as the number of dimensions tends to infinity, so the volume of the hypersphere tends to zero.

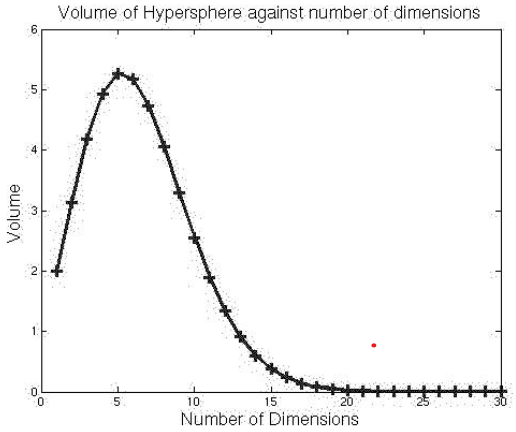


FIGURE 3.3 The volume of the unit hypersphere for different numbers of dimensions.

At first this seems completely the reverse for consideration. However, think about enclosing the hypersphere in a box of width 2 (between -1 and 1 along each axis), so that the box just touches the sides of the hypersphere. For the circle, almost all of the area inside the box is included in the circle, except for a little bit at each corner (see Figure 3.1) The same is true in 3D (Figure 3.2), but for the 100-dimensional hypersphere the volume of the hypersphere is obviously shrinking as the number of dimensions grows. The graph in Figure 3.3 shows that when the number of dimensions is above about 20, the volume is effectively zero. It was computed using the formula for the volume of the hypersphere of dimension n as $v_n = (2\Pi/n)v_{n-2}$. So as soon as $n > 2\Pi$, the volume starts to shrink.

The curse of dimensionality will apply to our machine learning algorithms because as the number of input dimensions gets larger, we will need more data to enable the algorithm to generalize sufficiently well. Our algorithms try to separate data into classes based on the features. So be careful about the information given to the algorithm. Regardless of how many input dimensions there are, the point of machine learning is to make predictions on data inputs.

3.2 Overfitting

Unfortunately, to know how well the algorithm is generalizing as it learns, make sure that enough training is given for the algorithm. In fact, there is at least as much danger in over-training as there is in under-training. The number of degrees of variability in most machine learning algorithms is huge - for a neural network there are lots of weights, and each of them can vary. This is undoubtedly more variation, so be careful while training for too long because it will overfit the data. It will learn about the noise and inaccuracies in the data as well

as the actual function. Therefore, the model will be too complicated, and won't be able to generalize.

Figure 3.4 shows this by plotting the predictions of some algorithm (as the curve) at two different points in the learning process. On the left of the figure 3.4 the curve fits the overall trend of the data well. The effect of overfitting is that rather than finding the generating function (as shown on the left in figure 3.4), the neural network matches the inputs perfectly, including the noise in them (on the right in figure 3.4). This reduces the generalization capabilities of the network. So that it has overfitted the training data.

Stop the learning process before the algorithm overfits, which means there is a need to know how well it is generalizing at each timestep. The training data is not useful for this because it wouldn't detect overfitting. The testing data is used for the final tests. So there is a need for a third set of data to use for this purpose, which is called the validation set because it is useful to validate the learning. This is known as cross-validation in statistics. It is part of model selection: choosing the right parameters for the model so that it generalizes as well as possible.

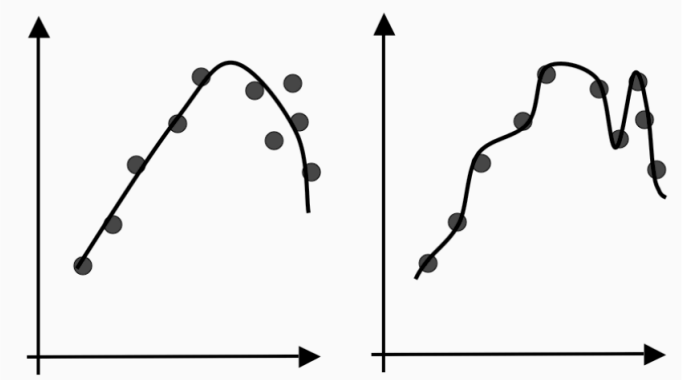


Figure 3.4 The effect of overfitting

3.3 Training, Testing, and Validation Sets

Three sets of data are needed. The training set to actually train the algorithm, the validation set to keep track of how well it is doing as it learns, and the test set to produce the final results. This is becoming expensive in data because for supervised learning all the data has the target values attached. Even for unsupervised learning, the validation and test sets need targets. It is not always easy to get accurate labels. The area of semi-supervised learning attempts to deal with this need for large amounts of labeled data.

Clearly, each algorithm needs some reasonable amount of data to learn from. However, the validation and test sets should also be reasonably large. Generally, the exact proportion of training to testing to validation data is up to you, but it is typical to do something like 50:25:25 if you have plenty of data, and 60:20:20 if you don't. Many datasets are presented with the first set of data points being in class 1, the next in class 2, and so on. If you pick the first few

points to be the training set, the next the test set, etc., then the results are going to be pretty bad, since the training did not see all the classes. This can be dealt with by randomly reordering the data first, or by assigning each datapoint randomly to one of the sets, as is shown in Figure 3.5.

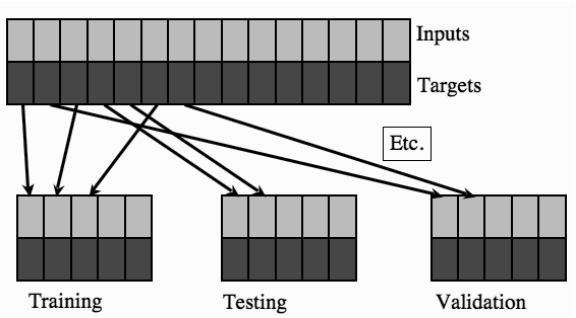


FIGURE 3.5 The dataset is split into different sets

If you are really short of training data, so that if you have a separate validation set there is a worry that the algorithm won't be sufficiently trained; then it is possible to perform leave-some-out, multi-fold cross-validation. The idea is shown in Figure 3.6. The dataset is randomly partitioned into K subsets, and one subset is used as a validation set and another for testing, while the algorithm is trained on all of the others. A different subset is then left out and a new model is trained on that subset, repeating the same process for all of the different subsets. Finally, the model that produced the lowest validation error is tested and used.

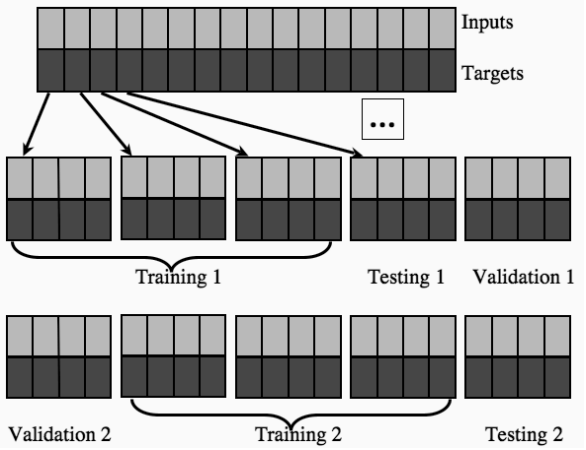


FIGURE 3.6 Leave-some-out, multi-fold cross-validation

3.4 The Confusion Matrix

A much better way to evaluate the performance of a classifier is to look at the confusion matrix. Regardless of how much data is used for testing still there is need to know whether the result is good or not. A method that is suitable for classification problems is the confusion matrix. For regression problems it is more complicated because the results are continuous, so the commonly used method is the sum-of-squares error.

In a confusion matrix, make a square matrix that contains all the possible classes in both the horizontal and vertical directions. Each row in a confusion matrix represents an actual class, while each column represents a predicted class. The general idea is to count the number of times instances of class C_1 are classified as class C_2 and so-on. Anything on the leading diagonal (the diagonal that starts at the top left of the matrix and runs down to the bottom right) is a correct answer. Suppose that we have three classes: C_1, C_2, and C_3. Now we count the number of times that the output was class C_1 when the target was C_1, then when the target was C_2, and so on until we've filled in the table:

	Outputs		
	C_1	C_2	C_3
C_1	5	1	0
C_2	1	4	1
C_3	2	0	4

This table tells us that, for the three classes, most examples were classified correctly, but two examples of class C_3 were misclassified as C_1, and so on. For a small number of classes this is a nice way to look at the outputs.

3.5 Accuracy Metrics

Results can be analyzed rather than just measuring the accuracy. If you consider the possible outputs of the classes, then they can be arranged in a simple chart where a true positive is an observation correctly put into class 1, while a false positive is an observation incorrectly put into class 1, while negative examples both true and false are those put into class 2. The entries on the leading diagonal of this chart are correct.

True Positives	False Positives
False Negatives	True Negatives

Accuracy is then defined as the sum of the number of true positives and true negatives divided by the total number of examples (where # means 'number of', and *TP* stands for True Positive, etc.).

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

The problem with accuracy is that it doesn't tell us everything about the results, since it turns four numbers into just one. There are two pairs of measurements that can help us to interpret the performance of a classifier, namely sensitivity and specificity, and precision and recall. Their definitions are shown next, followed by some explanation.

$$Sensitivity = \frac{\#TP}{\#TP+\#FN}$$

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

$$Precision = \frac{\#TP}{\#TP+\#FP}$$

$$Recall = \frac{\#TP}{\#TP+\#FN}$$

Sensitivity (also known as the true positive rate) is the ratio of the number of correct positive examples to the number classified as positive, while specificity is the same ratio for negative examples. Precision is the ratio of correct positive examples to the number of actual positive examples, while recall is the ratio of the number of correct positive examples out of those that were classified as positive, which is the same as sensitivity.

Either of these pairs of measures gives more information than the accuracy. If you consider precision and recall they are inversely related, in that if the number of false positives increases then the number of false negatives often decreases, and vice versa. They can be combined to give a single measure called F_1, which can be written in terms of precision and recall as

$$F_1 = 2\frac{precision \times recall}{precision + recall}$$

and in terms of the numbers of false positives, etc. as

$$F_1 = \frac{\#TP}{\#TP+(\#FN+\#FP)/2}$$

3.6 The Receiver Operator Characteristic (ROC) Curve

An ROC curve is a graph showing the performance of a classification model at all classification thresholds. The measures are useful to evaluate a particular classifier and can also compare the same classifier with different learning parameters, or completely different classifiers. This is a plot of the percentage of true positives on the y axis against false positives on the x axis; an example is shown in Figure 3.7. A single run of a classifier produces a single point on the ROC plot, and a perfect classifier would be a point at (0, 1) (100% true positives, 0% false positives), while the anti-classifier that got everything wrong would be at (1,0). The result of a classifier is closer to the top-left-hand corner. Any classifier that sits on the diagonal line from (0,0) to (1,1) behaves exactly at the chance level (assuming that the positive and negative classes are equally common) and so a lot of learning effort is wasted.

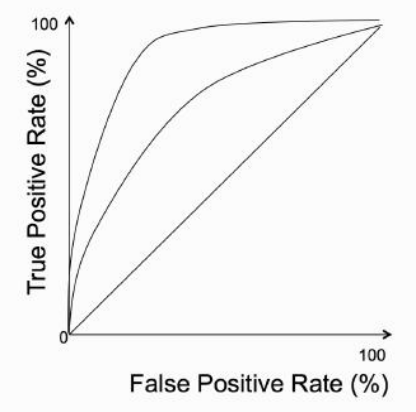


FIGURE 3.7 An example of an ROC curve

In order to compare classifiers, you could just compute the point that is far from the 'chance' line along the diagonal. However, it is normal to compute the area under the curve (AUC) instead. If you only have one point for each classifier, the curve is the trapezoid that runs from point (0,0) to (1,1). If there are more points, then they are just included along the diagonal line.

The key to getting a curve rather than a point on the ROC curve is to use cross validation. If you use 10-fold cross-validation, then you have 10 classifiers, with 10 different test sets, and you also have the 'ground truth' labels. By producing an ROC curve for each classifier it is possible to compare their results.

3.7 Unbalanced Datasets

For accuracy, assume that there are the same number of positive and negative examples in the dataset which is known as a balanced dataset. However, this is often not true. In the case where it is not, we can compute the balanced accuracy as the sum of sensitivity and specificity divided by 2. However, a more correct measure is Matthew's Correlation Coefficient, which is computed as:

$$MCC = \frac{\#TP\ X\ \#TN - \#FP\ X\ \#FN}{\sqrt{(\#TP + \#FP)(\#TP + \#FN)(\#TN + \#FP)(\#TN + \#FN)}}$$

If any of the brackets in the denominator are 0, then the whole of the denominator is set to 1. This provides a balanced accuracy computation. In methods of evaluation, if there are more than two classes and it is useful to distinguish the different types of error, then the calculations are a little complicated. Instead of one set of false positives and one set of false negatives, you have some for each class. In this case, specificity and recall are not the same. However, it is possible to create a set of results, where you use one class as the positives and everything else as the negatives, and repeat this for each of the different classes.

3.8 The Naïve Bayes Classifier

The naive Bayesian classifier, or simple Bayesian classifier, works as follows, Let D be a training set of tuples and their associated class labels. As usual, each tuple is represented by an n-dimensional attribute vector, $X = (x_1, x_2, ..., x_n)$, depicting n measurements made on the tuple from n attributes, respectively, $A_1, A_2, ..., A_n$.

Suppose that there are m classes, $C_1, C_2, ..., C_m$. Given a tuple, X, the classifier will predict that X belongs to the class having the highest posterior probability, conditioned on X. That is, the naive Bayesian classifier predicts that tuple X belongs to the class C_i if and only if

$$P(C_i \mid X) > P(C_j \mid X) \; for \; 1 \leq j \leq m, j \neq i$$

Thus, we maximize $P(C_i \mid X)$. The class C_i for which $P(C_i \mid X)$ is maximized is called the maximum posteriori hypothesis. By Bayes theorem,

$$P(C_i \mid X) = \frac{P(X \mid C_i)P(C_i)}{P(X)}$$

As $P(X)$ is constant for all classes, only $P(X \mid C_i)P(C_i)$ needs to be maximized. If the class prior probabilities are not known, then it is commonly assumed that the classes are equally likely, that is, $P(C_1) = P(C_2) = ... = P(C_m)$, and we would therefore maximize $P(X \mid C_i)$. Otherwise, we maximize $P(X \mid C_i)P(C_i)$. Note that the class prior probabilities may be estimated by $P(C_i) = |C_{i,D}| / |D|$ where $|C_{i,D}|$ is the number of training tuples of class C_i in D.

Given data sets with many attributes, it would be extremely computationally expensive to compute $P(X \mid C_i)$. To reduce computation in evaluating $P(X \mid C_i)$, the naive assumption of class-conditional independence is made. This presumes that the attribute values are conditionally independent of one another, given the class label of the tuple (i.e., that there are no dependence relationships among the attributes). Thus,

$$P(X \mid C_i) = \prod_{k=1}^{n} P(x_k \mid C_i) = P(x_1 \mid C_i) \times P(x_2 \mid C_i) \times ... \times P(x_n \mid C_i).$$

We can easily estimate the probabilities $P(x_1 \mid C_i)$, $P(x_2 \mid C_i)$, ..., $P(x_n \mid C_i)$ from the training tuples. To predict the class label of X, $P(X \mid C_i)P(C_i)$ is evaluated for each class C_i. The classifier predicts that the class label of tuple X is the class C_i if and only if

$$P(X \mid C_i) P(C_i) > P(X \mid C_j) P(C_j) \; for \; 1 \leq j \leq m, j \neq i$$

In other words, the predicted class label is the class C_i for which $P(X \mid C_i)\, P(C_i)$ is the maximum.

To predict the class label of a tuple using naive Bayesian classification, the training data is shown in figure 3.8. The data tuples are described by the attributes *age*, *income*, *student*, and *credit_rating*. The class label attribute, buys computer, has two distinct values {yes, no}. Let C_1 correspond to the class *buys_computer* = *yes* and C_2 correspond to *buys_computer* = *no*. The tuple we wish to classify is

X = (*age* = *youth*, *income* = *medium*, *student* = *yes*, *credit_rating* = *fair*)

RID	age	income	student	credit_rating	Class: buys_computer
1	youth	high	no	fair	no
2	youth	high	no	excellent	no
3	middle_aged	high	no	fair	yes
4	senior	medium	no	fair	yes
5	senior	low	yes	fair	yes
6	senior	low	yes	excellent	no
7	middle_aged	low	yes	excellent	yes
8	youth	medium	no	fair	no
9	youth	low	yes	fair	yes
10	senior	medium	yes	fair	yes
11	youth	medium	yes	excellent	yes
12	middle_aged	medium	no	excellent	yes
13	middle_aged	high	yes	fair	yes
14	senior	medium	no	excellent	no

Figure 3.8: Class-Labeled Training Tuples from *AllElectronics* Customer Database

To maximize $P(X \mid C_i)\, P(C_i)$, for i = 1, 2. $P(C_i)$, the prior probability of each class, can be computed based on the training tuples

P(*buys_computer* = *yes*) = 9/14 = 0.643

P(*buys_computer* = *no*) = 5/14 = 0.357

To compute$P(X \mid C_i)$, for i = 1, 2, we compute the following conditional probabilities:

P(*age* = *youth* | *buys_computer* = *yes*) = 2/9 = 0.222

P(*age* = *youth* | *buys_computer* = *no*) = 3/5 = 0.600

P(*income* = *medium* | *buys_computer* = *yes*) = 4/9 = 0.444

P(*income* = *medium* | *buys_computer* = *no*) = 2/5 = 0.400

P(*student* = *yes* | *buys_computer* = *yes*) = 6/9 = 0.667

P(*student* = *yes* | *buys_computer* = *no*) = 1/5 = 0.200

P(credit_rating = fair | *buys_computer* = *yes*) = 6/9 = 0.667

P(*credit_rating* = *fair* | *buys_computer* = *no*) = 2/5 = 0.400

Using these probabilities, we obtain

P(*X* | *buys_computer* = *yes*) = *P*(*age* = *youth* / *buys_computer* = *yes*) x *P*(*income* = *medium* | *buys_computer* = *yes*) x *P*(*student* = *yes* | *buys_computer* = *yes*) x *P*(*credit_rating* = *fair* | *buys_computer* = *yes*)

= 0.222 x 0.444 x 0.667 x 0.667 = 0.044.

Similarly,

P(X | buys_computer = no) = 0.600 x 0.400 x 0.200 x 0.400 = 0.019.

To find the class, C_i , that maximizes $P(X \mid C_i)\, P(C_i)$, we compute

P(*X* | *buys_computer* = *yes*) *P*(*buys_computer* = *yes*) = 0.044 x 0.643 = 0.028

P(*X* | *buys_computer* = *no*) *P*(*buys_computer* = *no*) = 0.019 x 0.357 = 0.007

Therefore, the naive Bayesian classifier predicts *buys_computer* = *yes* for tuple *X*.

3.9 Some Basic Statistics

3.9.1 Mean

Mean is the average of a given set of data. The formula for mean is $\mu = \frac{\sum_{i=1}^{n} x_i}{N}$, Where μ is mean and x_1, x_2, x_3,...., x_i are elements. Also note that mean is sometimes denoted by $\bar{x}$. Let us consider the example 2, 4, 4, 4, 5, 5, 7, 9. These eight data points have the mean (average) of 5.

3.9.2 Variance

Variance is the sum of squares of differences between all numbers and means. The formula for variance is $\sigma^2 = \frac{\sum_{i=1}^{n} (x_i - \mu)^2}{N}$, Where μ is Mean, *N* is the total number of elements or frequency of distribution. Let us consider the example 2, 4, 4, 4, 5, 5, 7, 9. First, calculate the deviations of each data point from the mean, and square the result of each

$(2 - 5)^2 = (-3)^2 = 9$

$(4 - 5)^2 = (-1)^2 = 1$

$(4 - 5)^2 = (-1)^2 = 1$

$(4 - 5)^2 = (-1)^2 = 1$

$(5 - 5)^2 = (0)^2 = 0$

$(5 - 5)^2 = (0)^2 = 0$

$(7 - 5)^2 = (2)^2 = 4$

$(9 - 5)^2 = (4)^2 = 16$

Variance = $\frac{9+1+1+1+0+0+4+16}{8}$ = 4

3.9.3 Standard Deviation

Standard Deviation is the square root of variance. It is a measure of the extent to which data varies from the mean. Let us consider the example 2, 4, 4, 4, 5, 5, 7, 9. The standard deviation is $\sqrt{4} = 2$

3.9.4 Covariance

Covariance is the relationship between a pair of random variables where change in one variable causes change in another variable. It can take any value between -infinity to +infinity, where the negative value represents the negative relationship whereas a positive value represents the positive relationship. It is used for the linear relationship between variables and gives the direction of the relationship between variables.

$$cov(X, Y) = \frac{1}{n}\sum_{i=1}^{n}(x_i - x')(y_i - y')$$

Where *x'* and *y'* are the means of two given sets.

3.9.5 Bias-Variance Tradeoff

It is important to understand prediction errors (bias and variance) when it comes to accuracy in any machine learning algorithm. There is a tradeoff between a model's ability to minimize bias and variance which is referred to as the best solution for selecting a value of Regularization constant. Proper understanding of these errors would help to avoid the overfitting and underfitting of a data set while training the algorithm.

3.9.5.1 Bias

The bias is known as the difference between the prediction of the values by the ML model and the correct value. Being high in biasing gives a large error in training as well as testing data. It is recommended that an algorithm should always be low biased to avoid the problem of underfitting.

By high bias, the data predicted is in a straight line format, thus not fitting accurately in the data in the data set. Such fitting is known as Underfitting of Data. This happens when the hypothesis is too simple or linear in nature. Refer to the graph in figure 3.9 for an example of such a situation.

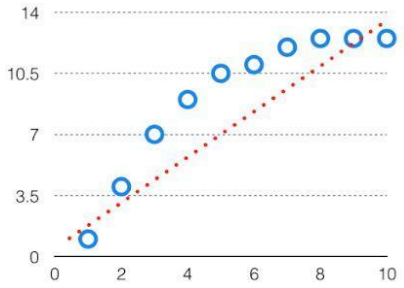


Figure 3.9: High Bias

In such a problem, a hypothesis looks like follows

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

3.9.5.2 Variance

The variability of model prediction for a given data point which tells us spread of our data is called the variance of the model. The model with high variance has a very complex fit to the training data and thus is not able to fit accurately on the data which it hasn't seen before. As a result, such models perform very well on training data but have high error rates on test data.

When a model is high on variance, it is then said to be Overfitting of Data. Overfitting is fitting the training set accurately via complex curve and high order hypothesis but is not the solution as the error with unseen data is high. While training a data model variance should be kept low. The high variance data looks as shown in figure 3.10.

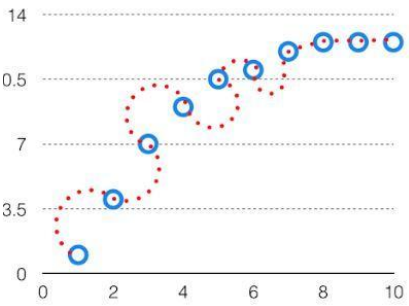


Figure 3.10: High Variance

In such a problem, a hypothesis looks like follows

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4)$$

3.9.5.3 Bias Variance Tradeoff

If the algorithm is too simple (hypothesis with linear eq.) then it may be on high bias and low variance condition and thus is error-prone. If algorithms fit too complex (hypothesis with high degree eq.) then it may be high variance and low bias. In the latter condition, the new entries will not perform well. Well, there is something between both of these conditions, known as Trade-off or Bias Variance Trade-off.

This tradeoff in complexity is why there is a tradeoff between bias and variance. An algorithm can't be more complex and less complex at the same time. For the graph, the perfect tradeoff is shown in figure 3.11.

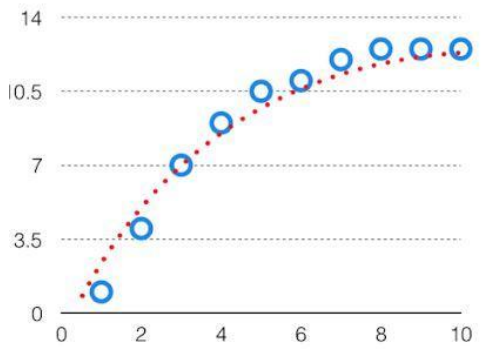


Figure 3.11: Tradeoff

The best fit will be given by hypothesis on the tradeoff point. The error to complexity graph to show trade-off is shown in figure 3.12.

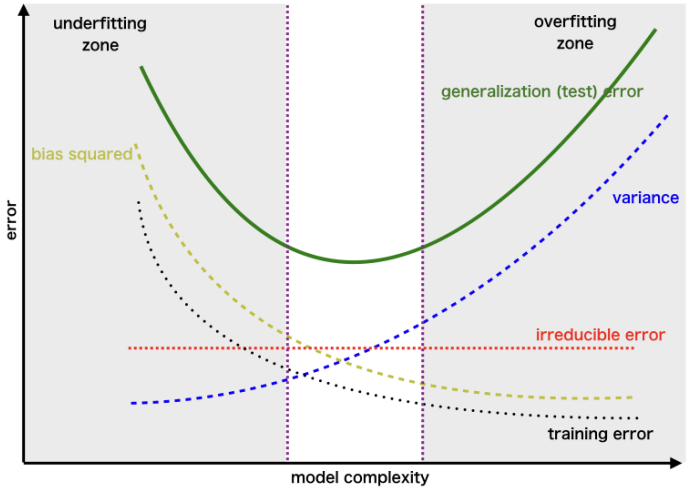


Figure 3.12: Error to Complexity Graph to show Trade-off

This is referred to as the best point chosen for the training of the algorithm which gives low error in training as well as testing data.

Chapter-4

Tree Models

Instructor Name: B N V Narasimha Raju

Tree models are among the most popular models in machine learning. Trees are expressive and easy to understand, and computer scientists are interested in this due to their recursive 'divide-and-conquer' nature. In fact, the paths through the logical hypothesis space constitute a very simple kind of tree. For instance, consider the feature tree. A feature tree is a compact way of representing a number of conjunctive concepts in the hypothesis space.

A feature tree is a tree such that each internal node (the nodes that are not leaves) is labeled with a feature, and each edge coming from an internal node is labeled with a literal. The set of literals at a node is called a split. Each leaf of the tree represents a logical expression, which is the conjunction of literals on the path from the root of the tree to the leaf.

4.1 Decision tree

Decision tree induction is the learning of decision trees from class-labeled training tuples. A decision tree is a flowchart-like tree structure, where each internal node denotes a test on an attribute, each branch represents an outcome of the test, and each leaf node holds a class label. The topmost node in a tree is the root node. A typical decision tree is shown in Figure 4.1. It represents the concept *buys_computer*, that is, it predicts whether a customer at *AllElectronics* is likely to purchase a computer.

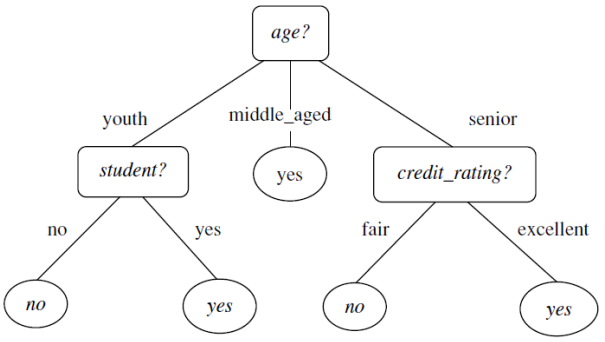


Figure 4.1 A decision tree for the concept *buys_computer*

Decision trees are used for classification when a tuple, *X*, for which the associated class label is unknown, the attribute values of the tuple are tested against the decision tree. A path is traced from the root to a leaf node, which holds the class prediction for that tuple. Decision trees can easily be converted to classification rules.

During tree construction, attribute selection measures are used to select the attribute that best partitions the tuples into distinct classes. When decision trees are built, many of the branches may reflect noise or outliers in the training data. Tree pruning attempts to identify and remove such branches, with the goal of improving classification accuracy on unseen data.

ID3 (Iterative Dichotomiser), C4.5, CART (Classification and Regression Trees) are approaches for learning decision trees from training tuples. They adopt a greedy (i.e., non backtracking) approach in which decision trees are constructed in a top-down recursive divide-and-conquer manner. Most algorithms for decision tree induction also follow a top-down approach, which starts with a training set of tuples and their associated class labels. The training set is recursively partitioned into smaller subsets as the tree is being built. A basic decision tree algorithm is summarized in figure 4.2.

Algorithm: Generate_decision_tree. Generate a decision tree from the training tuples of data partition, *D*.

Input:

- Data partition, *D*, which is a set of training tuples and their associated class labels;
- *attribute_list*, the set of candidate attributes;
- *Attribute_selection_method*, a procedure to determine the splitting criterion that "best" partitions the data tuples into individual classes. This criterion consists of a *splitting_attribute* and, possibly, either a *split-point* or *splitting subset*.

Output: A decision tree.

Method:

```
(1)  create a node N;
(2)  if tuples in D are all of the same class, C, then
(3)      return N as a leaf node labeled with the class C;
(4)  if attribute_list is empty then
(5)      return N as a leaf node labeled with the majority class in D; // majority voting
(6)  apply Attribute_selection_method(D, attribute_list) to find the "best" splitting_criterion;
(7)  label node N with splitting_criterion;
(8)  if splitting_attribute is discrete-valued and
         multiway splits allowed then // not restricted to binary trees
(9)      attribute_list ← attribute_list − splitting_attribute; // remove splitting_attribute
(10) for each outcome j of splitting_criterion
     // partition the tuples and grow subtrees for each partition
(11)     let Dj be the set of data tuples in D satisfying outcome j; // a partition
(12)     if Dj is empty then
(13)         attach a leaf labeled with the majority class in D to node N;
(14)     else attach the node returned by Generate_decision_tree(Dj, attribute_list) to node N;
     endfor
(15) return N;
```

Figure 4.2 Basic algorithm for inducing a decision tree from training tuples

The algorithm has three parameters: *D*, *attribute_list*, and *Attribute_selection_method*. *D* is a data partition. Initially, it is the complete set of training tuples and their associated class labels. The parameter *attribute_list* is a list of attributes describing the tuples. *Attribute_selection_method* specifies a procedure for selecting the best attribute that differentiates the given tuples according to class. This procedure requires an attribute selection measure such as information gain or the Gini index.

The tree starts as a single node, *N*, representing the training tuples in *D*. If the tuples in *D* are all of the same class, then node *N* becomes a leaf and is labeled with that class. Otherwise, the algorithm calls Attribute_selection_method to determine the splitting criterion. The splitting criterion tells us which attribute to test at node *N* and the branches to grow from node *N*. The splitting criterion is determined as that, the resulting partitions at each branch are as "pure" as possible. A partition is pure if all the tuples in it belong to the same class.

The node *N* is labeled with the splitting criterion. A branch is grown from node *N* for each of the outcomes of the splitting criterion. The tuples in *D* are partitioned accordingly. There are three possible scenarios, as illustrated in Figure 4.3. Let *A* be the splitting attribute. *A* has *v* distinct values, $\{a_1, a_2, ..., a_v\}$, based on the training data.

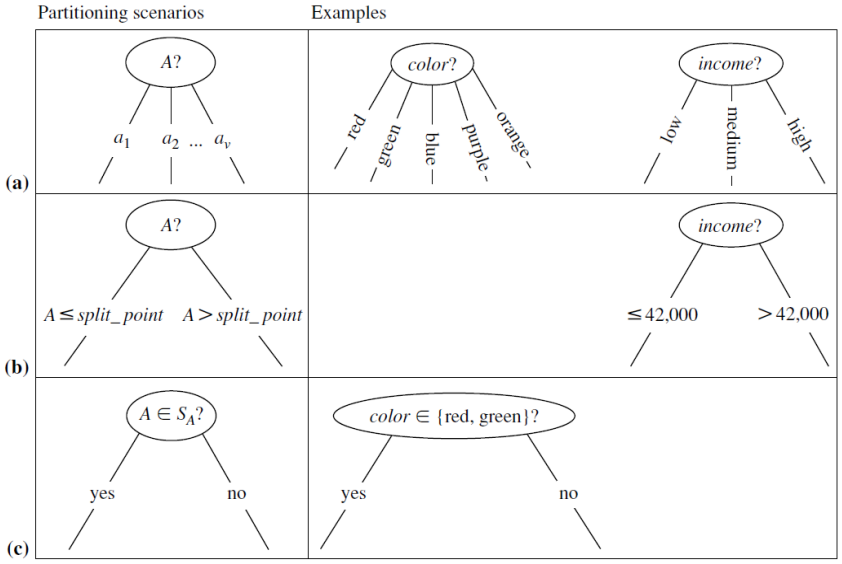


Figure 4.3 This figure shows three possibilities for partitioning tuples based on the splitting criterion

- *A* is discrete-valued: In this case, the outcomes of the test at node *N* correspond directly to the known values of *A*. A branch is created for each known value, a_j, of *A* and labeled with that value (Figure 4.3a) and it is removed from *attribute_list*.
- *A* is continuous-valued: In this case, the test at node *N* has two possible outcomes, corresponding to the conditions $A \leq split\_point$ and $A \geq split\_point$, respectively, where *split_point* is returned by *Attribute_selection_method* as part of the splitting criterion. Two branches are grown from *N* and labeled according to the previous outcomes (Figure 4.3b).
- *A* is discrete-valued and a binary tree must be produced: The test at node *N* is of the form "$A \in S_A$?," where S_A is the splitting subset for *A*, returned by *Attribute_selection_method* as part of the splitting criterion. It is a subset of the known values of *A*. If a given tuple has value a_j of *A* and if $a_j \in S_A$, then the test at node *N* is satisfied. Two branches are grown from *N* (Figure 4.3c). By convention, the left branch out of *N* is labeled *yes* that satisfy the test. The right branch out of *N* is labeled *no* so that do not satisfy the test.

The algorithm uses the same process recursively to form a decision tree for the tuples at each resulting partition, D_j , of D. The recursive partitioning stops only when any one of the following terminating conditions is true:

- All the tuples in partition D belong to the same class.
- There are no remaining attributes on which the tuples may be further partitioned. In this case, majority voting is employed. This involves converting node N into a leaf and labeling it with the most common class in D.
- There are no tuples for a given branch, that is, a partition D_j is empty. In this case, a leaf is created with the majority class in D.

The resulting decision tree is returned. An attribute selection measure is a heuristic for selecting the splitting criterion. Attribute selection measures are also known as splitting rules because they determine how the tuples at a given node are to be split. The attribute selection measure provides a ranking for each attribute describing the given training tuples. The attribute having the best score for the measure is chosen as the splitting attribute for the given tuples. The tree node created for partition D is labeled with the splitting criterion, branches are grown for each outcome of the criterion, and the tuples are partitioned accordingly. There are three popular attribute selection measures—information gain, gain ratio, and Gini index.

4.1.1 Information Gain

ID3 uses information gain as its attribute selection measure. Let node N has the tuples of partition D. The attribute with the highest information gain is chosen as the splitting attribute for node N. The expected information needed to classify a tuple in D is given by

$$Info(D) = -\sum_{i=1}^{m} P_i \log_2 (P_i)$$

where P_i is the nonzero probability that an arbitrary tuple in D belongs to class C_i and is estimated by $|C_{i,D}| / |D|$ (i.e. $|C_{i,D}|$ and $|D|$ denotes the number of tuples in $C_{i,D}$ and D). A log function to the base 2 is used, because the information is encoded in bits. *Info*(D) is just the average amount of information needed to identify the class label of a tuple in D. *Info*(D) is also known as the entropy of D.

Now, partition the tuples in D on some attribute A having v distinct values. These partitions would correspond to the branches grown from node N. These partitioning need to produce an exact classification of the tuples. The amount of information need for exact classification is measured by

$$Info_A(D) = \sum_{j=1}^{v} \frac{|D_j|}{|D|} \times Info(D_j)$$

The term $\frac{|D_j|}{|D|}$ acts as the weight of the j^{th} partition. $Info_A(D)$ is the expected information required to classify a tuple from D based on the partitioning by A. The smaller the expected information (still) required, the greater the purity of the partitions.

Information gain is defined as the difference between the original information requirement (i.e., based on just the proportion of classes) and the new requirement (i.e., obtained after partitioning on A). That is,

$$Gain(A) = Info(D) - Info_A(D)$$

The attribute A with the highest information gain, $Gain(A)$, is chosen as the splitting attribute at node N.

4.1.1.1 Induction of a decision tree using information gain

Table 4.1 presents a training set, D, of class-labeled tuples randomly selected from the *AllElectronics* customer database. The class label attribute, *buys_computer*, has two distinct values, therefore there are two distinct classes. Let class C_1 correspond to *yes* and class C_2 correspond to *no*. There are nine tuples of class *yes* and five tuples of class *no*. A (root) node N is created for the tuples in D. To find the splitting criterion for these tuples, we must compute the information gain of each attribute.

RID	age	income	student	credit_rating	Class: buys_computer
1	youth	high	no	fair	no
2	youth	high	no	excellent	no
3	middle_aged	high	no	fair	yes
4	senior	medium	no	fair	yes
5	senior	low	yes	fair	yes
6	senior	low	yes	excellent	no
7	middle_aged	low	yes	excellent	yes
8	youth	medium	no	fair	no
9	youth	low	yes	fair	yes
10	senior	medium	yes	fair	yes
11	youth	medium	yes	excellent	yes
12	middle_aged	medium	no	excellent	yes
13	middle_aged	high	yes	fair	yes
14	senior	medium	no	excellent	no

Table 4.1 Class-Labeled Training Tuples from the AllElectronics Customer Database

First compute the expected information needed to classify a tuple in D:

$$Info(D) = -\frac{9}{14}\log_2\left(\frac{9}{14}\right) - \frac{5}{14}\log_2\left(\frac{5}{14}\right) = 0.940 \text{ bits.}$$

Next, we need to compute the expected information requirement for each attribute. Let's start with the attribute *age*. We need to look at the distribution of *yes* and *no* tuples for each category of *age*. For the *age* there are three category they are "youth", "middle aged" and

"senior". The expected information needed to classify a tuple in *D* if the tuples are partitioned according to *age* is

$$Info_{age}(D) = \frac{5}{14} \times \left(-\frac{2}{5}\log_2\frac{2}{5} - \frac{3}{5}\log_2\frac{3}{5}\right) + \frac{4}{14} \times \left(-\frac{4}{4}\log_2\frac{4}{4}\right) + \frac{5}{14} \times \left(-\frac{3}{5}\log_2\frac{3}{5} - \frac{2}{5}\log_2\frac{2}{5}\right)$$

$$= 0.694 \text{ bits}.$$

Hence, the gain in information from such a partitioning would be

$$Gain(age) = Info(D) - Info_{age}(D) = 0.940 - 0.694 = 0.246 \text{ bits}.$$

Similarly, we can compute *Gain*(*income*) = 0.029 bits, *Gain*(*student*) = 0.151 bits, and *Gain*(*credit_rating*) = 0.048 bits. Because *age* has the highest information gain among the attributes, it is selected as the splitting attribute. Node *N* is labeled with age, and branches are grown for each of the attribute's values. The tuples are then partitioned accordingly, as shown in figure 4.4.

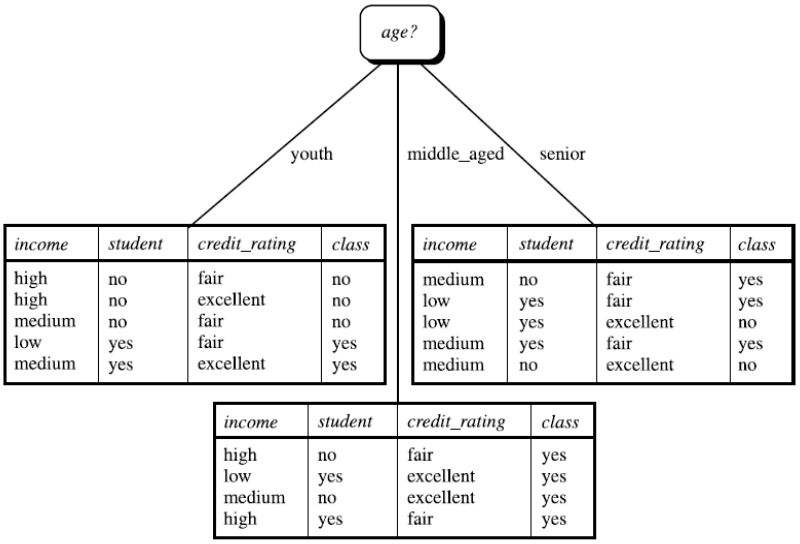


Figure 4.4 The attribute age has the highest information gain

4.1.2 Gain Ratio

The information gain measure is biased toward tests with many outcomes. C4.5 uses an extension to information gain known as gain ratio, which attempts to overcome this bias. It applies a kind of normalization to information gain using a "split information" value defined analogously with Info(D) as

$$SplitInfo_A(D) = -\sum_{j=1}^{v} \frac{|D_j|}{|D|} \times \log_2\left(\frac{|D_j|}{|D|}\right).$$

This value represents the potential information generated by splitting the training data set, *D*, into *v* partitions, corresponding to the v outcomes of a test on attribute *A*. The gain ratio is defined as

$$GainRatio(A) = \frac{Gain(A)}{SplitInfo_A(D)}.$$

The attribute with the maximum gain ratio is selected as the splitting attribute

4.1.2.1 Computation of gain ratio for the attribute income.

A test on income splits the data into three partitions, namely low, medium, and high, containing four, six, and four tuples, respectively. To compute the gain ratio of income, first obtain

$$SplitInfo_{income}(D) = -\frac{4}{14} \times \log_2\left(\frac{4}{14}\right) - \frac{6}{14} \times \log_2\left(\frac{6}{14}\right) - \frac{4}{14} \times \log_2\left(\frac{4}{14}\right)$$

$$= 1.557.$$

The *Gain*(*income*) = 0.029. Therefore, *GainRatio*(*income*) = 0.029 / 1.557 = 0.019.

4.1.3 Gini Index

The Gini index is used in CART. Using the notation previously described, the Gini index measures the impurity of D, a data partition or set of training tuples, as

$$Gini(D) = 1 - \sum_{i=1}^{m} p_i^2$$

where P_i is the probability that a tuple in *D* belongs to class C_i and is estimated by $|C_{i,D}| / |D|$. The sum is computed over *m* classes. The Gini index considers a binary split for each attribute. Let's first consider the case where *A* is a discrete-valued attribute having *v* distinct values, occurring in *D*. To determine the best binary split on *A*, we examine all the possible subsets that can be formed using known values of *A*.

If *A* has *v* possible values, then there are 2^v possible subsets. For example, if income has three possible values, namely {*low*, *medium*, *high*}, then the possible subsets are {*low*, *medium*, *high*}, {*low*, *medium*}, {*low*, *high*}, {*medium*, *high*}, {*low*}, {*medium*}, {*high*} and { }. Exclude the power set, {*low*, *medium*, *high*}, and the empty set from consideration since they do not represent a split. Therefore, there are 2^v - 2 possible ways to form two partitions of the data, *D*, based on a binary split on *A*.

When considering a binary split, we compute a weighted sum of the impurity of each resulting partition. For example, if a binary split on *A* partitions *D* into D_1 and D_2, the Gini index of *D* given that partitioning is

$$Gini_A(D) = \frac{|D_1|}{|D|} Gini(D_1) + \frac{|D_2|}{|D|} Gini(D_2).$$

For each attribute, each of the possible binary splits is considered. For a discrete-valued attribute, the subset that gives the minimum Gini index for that attribute is selected as its splitting subset.

For continuous-valued attributes, each possible split-point must be considered. The reduction in impurity that would be incurred by a binary split on a discrete- or continuous-valued attribute *A* is

$$\Delta Gini(A) = Gini(D) - Gini_A(D).$$

The attribute that maximizes the reduction in impurity (or, equivalently, has the minimum Gini index) is selected as the splitting attribute. This attribute and either its splitting subset (for a discrete-valued splitting attribute) or split-point (for a continuous-valued splitting attribute) together form the splitting criterion.

4.1.3.1 Induction of a decision tree using the Gini index

Let *D* be the training data, where there are nine tuples belonging to the class *buys_computer* = *yes* and the remaining five tuples belong to the class *buys_computer* = *no*. A (root) node *N* is created for the tuples in *D*. First use the Gini index to compute the impurity of *D*:

$$Gini(D) = 1 - \left(\frac{9}{14}\right)^2 - \left(\frac{5}{14}\right)^2 = 0.459.$$

To find the splitting criterion for the tuples in D, we need to compute the Gini index for each attribute. Let's start with the attribute income and consider each of the possible splitting subsets. Consider the subset {low, medium}. This would result in 10 tuples in partition D_1 satisfying the condition "income $\in$ {low, medium}." The remaining four tuples of D would be assigned to partition D_2. The Gini index value computed based on

$$\begin{aligned} &Gini_{income \in \{low,medium\}}(D) \\ &\quad = \frac{10}{14} Gini(D_1) + \frac{4}{14} Gini(D_2) \\ &\quad = \frac{10}{14}\left(1 - \left(\frac{7}{10}\right)^2 - \left(\frac{3}{10}\right)^2\right) + \frac{4}{14}\left(1 - \left(\frac{2}{4}\right)^2 - \left(\frac{2}{4}\right)^2\right) \\ &\quad = 0.443 \\ &\quad = Gini_{income \in \{high\}}(D). \end{aligned}$$

Similarly, the Gini index values for splits on the remaining subsets are 0.458 (for the subsets {*low*, *high*} and {*medium*}) and 0.450 (for the subsets {*medium*, *high*} and {*low*}). Therefore, the best binary split for attribute income is on {*low*, *medium*} (or {*high*}) because it minimizes the Gini index. Evaluating age, we obtain {*youth*, *senior*} (or {*middle_aged*}) as the best split for age with a Gini index of 0.375; the attributes *student* and *credit_rating* are both binary, with Gini index values of 0.367 and 0.429, respectively.

4.1.4 Tree Pruning

When a decision tree is built, many of the branches will reflect anomalies in the training data due to noise or outliers. Tree pruning methods address this problem of overfitting the data. Such methods typically use statistical measures to remove the least-reliable branches. An unpruned tree and a pruned version of it are shown in Figure 4.5. Pruned trees tend to be smaller and less complex and, thus, easier to comprehend. They are usually faster and better at correctly classifying independent test data than unpruned trees.

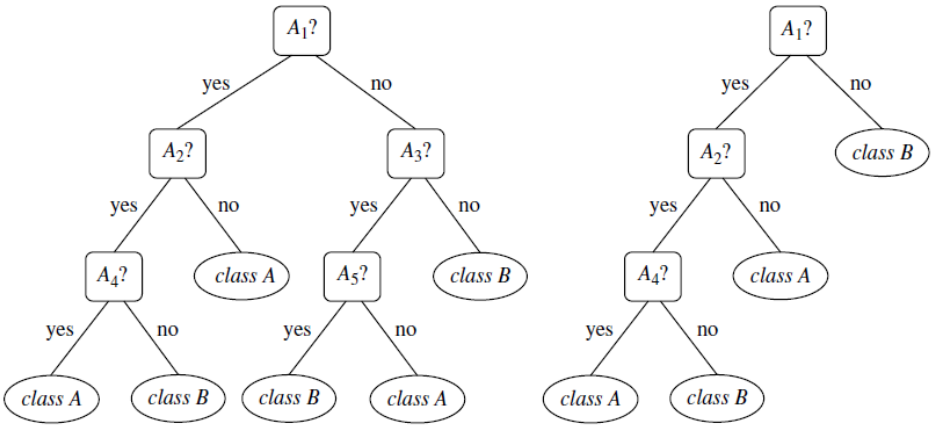


Figure 4.5 An unpruned decision tree and a pruned version of it.

There are two common approaches to tree pruning: prepruning and postpruning. In the prepruning approach, a tree is "pruned" by halting its construction early. Upon halting, the node becomes a leaf. The leaf may hold the most frequent class among the subset tuples or the probability distribution of those tuples.

When constructing a tree, partitioning the tuples at a node would result in a split that falls below a prespecified threshold, then further partitioning of the given subset is halted. There are difficulties, however, in choosing an appropriate threshold. High thresholds could result in oversimplified trees, whereas low thresholds could result in very little simplification.

The second and more common approach is postpruning, which removes subtrees from a "fully grown" tree. A subtree at a given node is pruned by removing its branches and replacing it with a leaf. The leaf is labeled with the most frequent class among the subtree being replaced. For example, notice the subtree at node "A3?" in the unpruned tree of Figure 4.5. Suppose that the most common class within this subtree is "class B." In the pruned version of the tree, the subtree in question is pruned by replacing it with the leaf "class B."

Postpruning requires more computation than prepruning, yet generally leads to a more reliable tree. Although pruned trees tend to be more compact than their unpruned counterparts, they may still be rather large and complex.

Decision trees can suffer from repetition and replication (Figure 4.6), making them overwhelming to interpret. Repetition occurs when an attribute is repeatedly tested along a given branch of the tree (e.g., “age < 60?,” followed by “age < 45?,” and so on). In replication, duplicate subtrees exist within the tree. These situations can impede the accuracy and comprehensibility of a decision tree. The use of multivariate splits (splits based on a combination of attributes) can prevent these problems. Another approach is to use a different form of knowledge representation, such as rules, instead of decision trees.

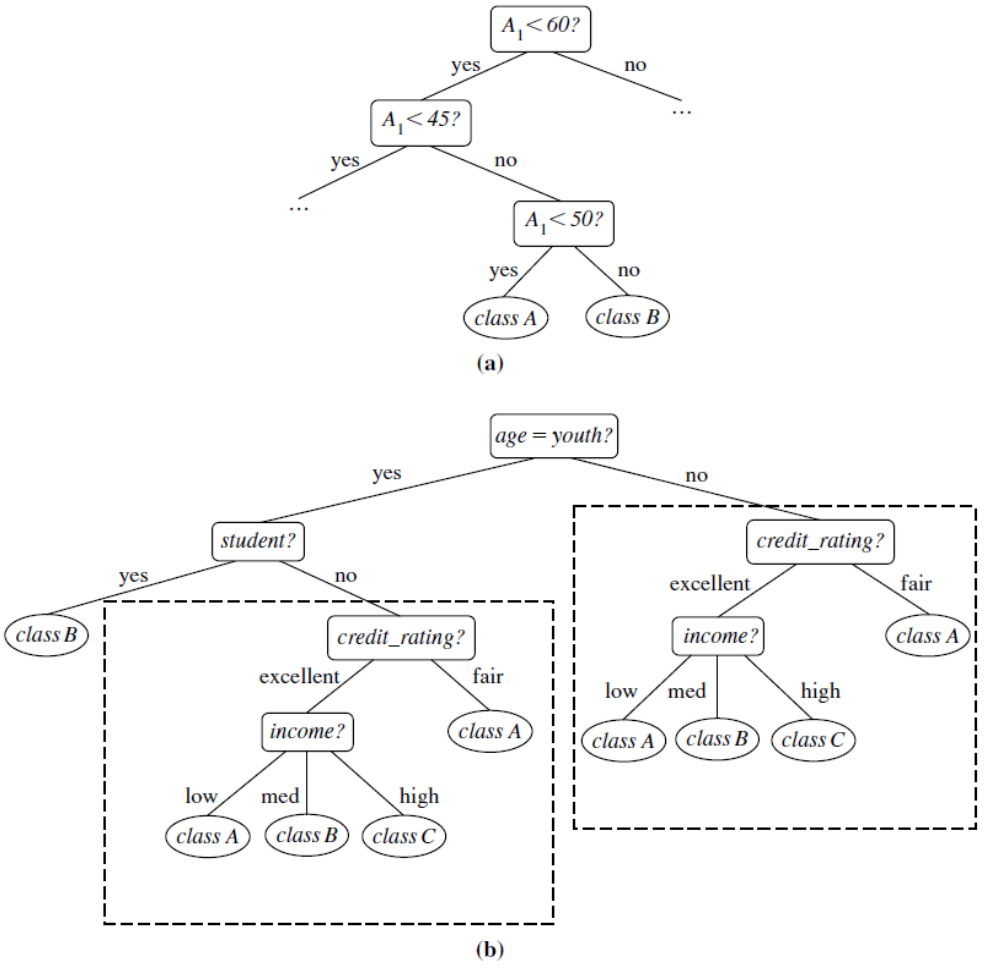


Figure 4.6 An example of: (a) subtree repetition, and (b) subtree replication.

Chapter-5

Linear Models

Instructor Name: B N V Narasimha Raju

In machine learning, linear models are of particular interest because of their simplicity. The following are the simplicity of linear models.

- Linear models are parametric, meaning that they have a fixed form with a small number of numeric parameters that need to be learned from data.
- Linear models are stable, which is to say that small variations in the training data have only a limited impact on the learned model.
- Linear models are less likely to overfit the training data than some other models, largely because they have relatively few parameters. The flipside of this is that they sometimes lead to underfitting.

Linear models have low variance but high bias. Such models are often preferable when you have limited data and want to avoid overfitting. Linear models exist for all predictive tasks, including classification, probability estimation and regression.

5.1 The least-squares method - Univariate linear regression

This is about the basics of linear regression and its implementation in the Python programming language. Linear regression is a statistical method for modeling relationships between a dependent variable with a given set of independent variables.

Note: In this, we refer to dependent variables as responses and independent variables as features for simplicity. In order to provide a basic understanding of linear regression, we start with the most basic version of linear regression, i.e. Simple linear regression.

5.2 Simple Linear Regression

Simple linear regression is an approach for predicting a response using a single feature. It is assumed that the two variables are linearly related. Hence, we try to find a linear function that predicts the response value(y) as accurately as possible as a function of the feature or

independent variable(x). Let us consider a dataset where we have a value of response y for every feature x:

x	0	1	2	3	4	5	6	7	8	9
y	1	3	2	5	7	8	8	9	10	12

For generality, we define x as feature vector, i.e x = [x_1, x_2, ..., x_n], y as response vector, i.e y = [y_1, y_2, ..., y_n] for n observations (in above example, n=10). A scatter plot of the above dataset is shown in figure 5.1

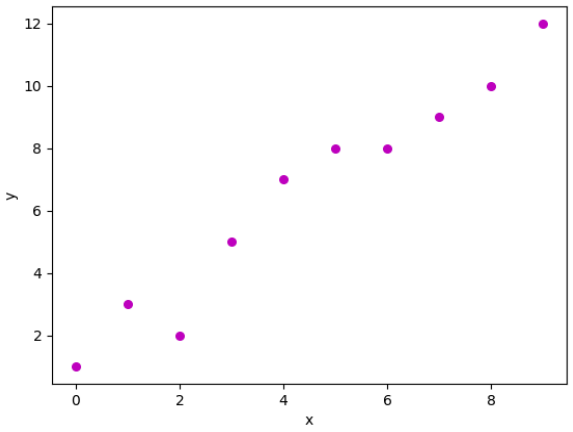


Figure 5.1 Scatter Plot

Now, the task is to find a line that fits best in the above scatter plot so that we can predict the response for any new feature values. (i.e a value of x not present in a dataset). This line is called a regression line. The equation of regression line is represented as:

$$h(x_i) = \beta_0 + \beta_1 x_i$$

Here, $h(x_i)$ represents the predicted response value for ith observation. β_0 and β_1 are regression coefficients and represent y-intercept and slope of regression line respectively. To create our model, we must learn or estimate the values of regression coefficients β_0 and β_1 and once we've estimated these coefficients, we can use the model to predict responses.

In this, we are going to use the principle of Least Squares. Now consider:

$$h(x_i) = \beta_0 + \beta_1 x_i + \varepsilon_i = h(x_i) + \varepsilon_i = \varepsilon_i = y_i - h(x_i)$$

Here, ε_i is a residual error in the i^{th} observation. So, our aim is to minimize the total residual error. We define the squared error or cost function, J as:

$$J(\beta_0, \beta_1) = \frac{1}{2n}\sum_{i=1}^{n} \varepsilon_i^2$$

Our task is to find the value of β_0 and β_1 for which $J(\beta_0, \beta_1)$ is minimum. Without going into the mathematical details, we present the result here:

$$\beta_1 = \frac{SS_{xy}}{SS_{xx}}$$

$$\beta_0 = \bar{y} - \beta_1 \bar{x}$$

where SS_{xy} is the sum of cross-deviations of *y* and *x*

$$SS_{xy} = \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y}) = \sum_{i=1}^{n} y_i x_i - n\bar{x}\,\bar{y})$$

and SS_{xx} is the sum of squared deviations of *x*

$$SS_{xx} = \sum_{i=1}^{n} (x_i - \bar{x})^2 = \sum_{i=1}^{n} x_i^2 - n(\bar{x})^2$$

5.3 Logistic Regression

Logistic regression is basically a supervised classification algorithm. It is a statistical method that is used for building machine learning models where the dependent variable is dichotomous: i.e. binary. Logistic regression is used to describe data and the relationship between one dependent variable and one or more independent variables. The independent variables can be nominal, ordinal, or of interval type.

Logistic regression predicts the output of a categorical dependent variable. Therefore the outcome must be a categorical or discrete value. It can be either Yes or No, 0 or 1, True or False, etc. but instead of giving the exact value as 0 and 1, it gives the probabilistic values which lie between 0 and 1.

The name “logistic regression” is derived from the concept of the logistic function that it uses. The logistic function, also known as the sigmoid function, was developed by statisticians to describe properties of population growth in ecology, rising quickly and maxing out at the carrying capacity of the environment. It’s an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1, but never exactly at those limits.

$$\frac{1}{1+e^{-value}}$$

Where *e* is the base of the natural logarithms (Euler’s number or the EXP() function in your spreadsheet) and value is the actual numerical value that you want to transform. Below is a

plot of the numbers between -5 and 5 transformed into the range 0 and 1 using the logistic function as shown in figure 5.2.

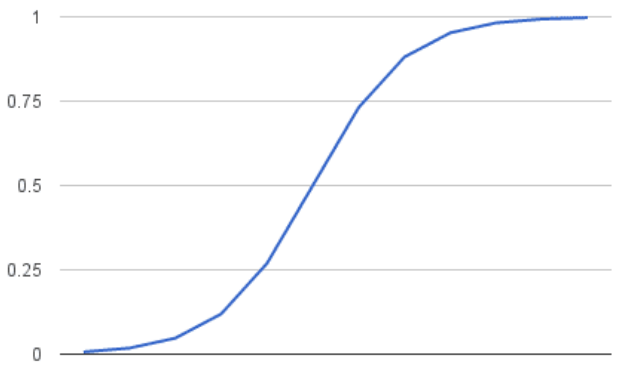


Figure 5.2 Logistic function

In order to map this to a discrete class, we select a threshold value or tipping point above which we will classify values into class 0 and below which we classify values into class 1.

$$p >= 0.5, \text{class} = 0$$

$$p < 0.5, \text{class} = 1$$

If our threshold was .5 and our prediction function returned .7, we would classify this observation as belonging to class 0. If our prediction was .2 we would classify the observation as belonging to class 1. So, the line with 0.5 is called the decision boundary. In order to map predicted values to probabilities, we use the Sigmoid function as shown in figure 5.3.

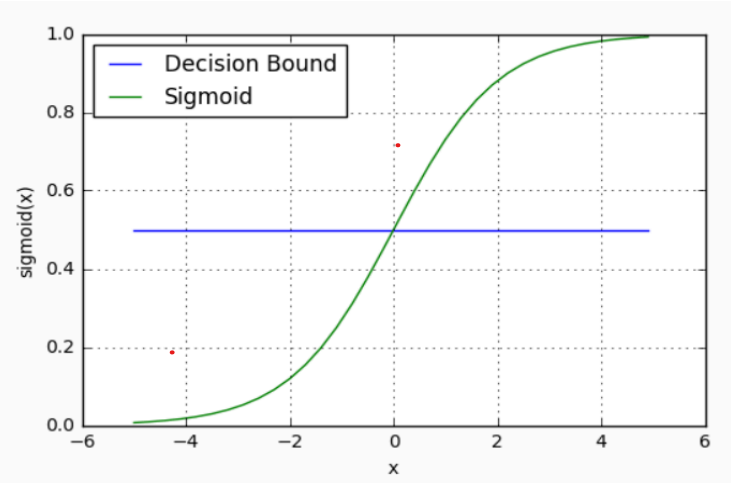


Figure 5.3 Sigmoid function

5.3.1 Representation Used for Logistic Regression

Logistic regression uses an equation as the representation, very much like linear regression. Input values (x) are combined linearly using weights or coefficient values (referred to as the

Greek capital letter Beta) to predict an output value (y). A key difference from linear regression is that the output value being modeled is a binary value (0 or 1) rather than a numeric value.

Below is an example logistic regression equation:

$$y = \frac{e^{(b_0 + b_1 x)}}{1 + e^{(b_0 + b_1 x)}}$$

Where *y* is the predicted output, b_0 is the bias or intercept term and b_1 is the coefficient for the single input value (x). Each column in your input data has an associated *b* coefficient (a constant real value) that must be learned from your training data.

The actual representation of the model that you would store in memory or in a file are the coefficients in the equation (the beta value or b's).

5.3.2 Logistic Regression Predicted Probabilities

Logistic regression models the probability of the default class (e.g. the first class). For example, if we are modeling people's sex as male or female from their height, then the first class could be male and the logistic regression model could be written as the probability of male given a person's height, or more formally:

$$P(sex = male \mid height)$$

In another way, modeling the probability that an input (*X*) belongs to the default class (*Y*=1), can be written as:

$$P(X) = P(Y = 1 \mid X)$$

Note that the probability prediction must be transformed into a binary value (0 or 1) in order to actually make a probability prediction.

Logistic regression is a linear method, but the predictions are transformed using the logistic function. The impact of this is that we can no longer understand the predictions as a linear combination of the inputs as we can with linear regression, for example, continuing on from above, the model can be stated as:

$$p(X) = \frac{e^{(b_0 + b_1 x)}}{1 + e^{(b_0 + b_1 x)}}$$

The above equation can be written as follows (remember we can remove the *e* from one side by adding a natural logarithm (ln) to the other):

$$ln\left(\frac{P(X)}{1-P(X)}\right) = b_0 + b_1 x$$

This is useful because we can see that the calculation of the output on the right is linear again (just like linear regression), and the input on the left is a log of the probability of the default class.

This ratio on the left is called the odds of the default class. Odds are calculated as a ratio of the probability of the event divided by the probability of not the event, e.g. 0.8/(1-0.8) which has the odds of 4. So we could instead write:

$$ln(odds) = b_0 + b_1 x$$

Because the odds are log transformed, we call this left hand side the log-odds or the probit. It is possible to use other types of functions for the transform (which is out of scope), but as such it is common to refer to the transform that relates the linear regression equation to the probabilities as the link function, e.g. the probit link function.

We can move the exponent back to the right and write it as:

$$odds = e^{(b_0 + b_1 x)}$$

All of this helps us understand that indeed the model is still a linear combination of the inputs, but that this linear combination relates to the log-odds of the default class.

5.3.3 Learning the Logistic Regression Model

The coefficients (Beta values b) of the logistic regression algorithm must be estimated from your training data. This is done using maximum-likelihood estimation. Maximum-likelihood estimation is a common learning algorithm used by a variety of machine learning algorithms, although it does make assumptions about the distribution of your data (more on this when we talk about preparing your data).

The best coefficients would result in a model that would predict a value very close to 1 (e.g. male) for the default class and a value very close to 0 (e.g. female) for the other class. The intuition for maximum-likelihood for logistic regression is that a search procedure seeks values for the coefficients (Beta values) that minimize the error in the probabilities predicted by the model to those in the data (e.g. probability of 1 if the data is the primary class).

It is enough to say that a minimization algorithm is used to optimize the best values for the coefficients for your training data. This is often implemented in practice using efficient numerical optimization algorithms (like the Quasi-newton method).

While learning logistic, it can be implemented from scratch using the much simpler gradient descent algorithm.

5.3.4 Making Predictions with Logistic Regression

Making predictions with a logistic regression model is as simple as plugging in numbers into the logistic regression equation and calculating a result.

Let's make this concrete with a specific example. Let's say we have a model that can predict whether a person is male or female based on their height (completely fictitious). Given a height of 150cm is the person male or female.

We have learned the coefficients of b_0 = -100 and b_1 = 0.6. Using the equation above we can calculate the probability of male given a height of 150cm or more formally *P*(*male* | *height* = 150). We will use EXP() for e, because that is what you can use if you type this example into your spreadsheet:

$$y = \frac{e^{(b_0 + b_1 x)}}{1 + e^{(b_0 + b_1 x)}}$$

$$y = \frac{EXP^{(-100 + 0.6*150)}}{1 + EXP^{(-100 + 0.6*150)}}$$

y = 0.0000453978687

Or a probability of near zero that the person is a male. In practice we can use the probabilities directly. Because this is classification and we want a crisp answer, we can snap the probabilities to a binary class value, for example:

0 if p(male) < 0.5

1 if p(male) >= 0.5

Now that we know how to make predictions using logistic regression, let's look at how we can prepare our data to get the most from the technique.

Sample calculations

Suppose a cancer study yields:

log odds = –2.6837 + 0.0812 SurvRate

Consider a patient with SurvRate = 40

log odds = –2.6837 + 0.0812(40) = 0.5643

odds = $e^{0.5643}$ = 1.758

patient is 1.758 times more likely to be improved than not

Consider another patient with SurvRate = 41

$$\text{log odds} = -2.6837 + 0.0812(41) = 0.6455$$

$$\text{odds} = e^{0.6455} = 1.907$$

Patient's odds are 1.907/1.758 = 1.0846 times (or 8.5%) better than those of the previous patient

5.4 Support Vector Machines

Support Vector Machine(SVM) is a supervised machine learning algorithm used for both classification and regression. It can be used for the classification of both linear and nonlinear data. Data from two classes can always be separated by a hyperplane. The SVM finds this hyperplane using support vectors ("essential" training tuples) and margins (defined by the support vectors).

5.4.1 The Case When the Data Are Linearly Separable

To explain the mystery of SVMs, let's first look at the simplest case—a two-class problem where the classes are linearly separable. Let the data set D be given as (X_1, y_1), (X_2, y_2), ..., $(X_{|D|}, y_{|D|})$, where X_i is the set of training tuples with associated class labels, y_i. Each y_i can take one of two values, either +1 or -1 corresponding to the classes buys_computer = *yes* and buys_computer = *no*, respectively. To aid in visualization, let's consider an example based on two input attributes, A_1 and A_2, as shown in figure 5.4. From the graph, we see that the 2-D data are linearly separable (or "linear," for short), because a straight line can be drawn to separate all the tuples of class +1 from all the tuples of class -1.

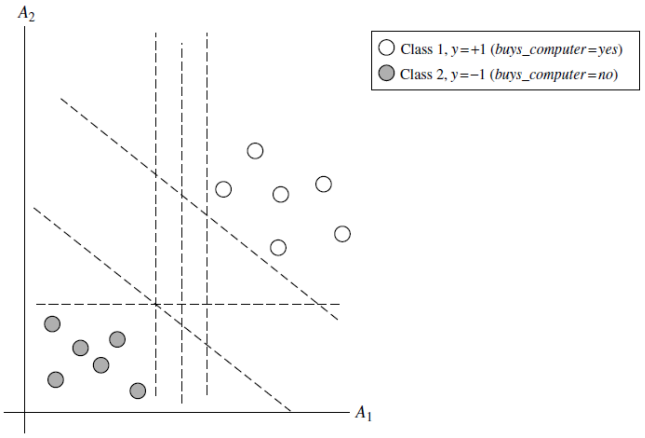


Figure 5.4 The 2-D training data are linearly separable

There are an infinite number of separating lines that could be drawn. Find the "best" one, that is, one that (we hope) will have the minimum classification error on previously unseen tuples.

An SVM approaches this problem by searching for the maximum marginal hyperplane. Consider figure 5.5, which shows two possible separating hyperplanes and their associated margins. Both hyperplanes can correctly classify all the given data tuples. Intuitively, however, we expect the hyperplane with the larger margin to be more accurate at classifying future data tuples than the hyperplane with the smaller margin. This is why (during the learning or training phase) the SVM searches for the hyperplane with the largest margin, that is, the maximum marginal hyperplane (MMH). The associated margin gives the largest separation between classes.

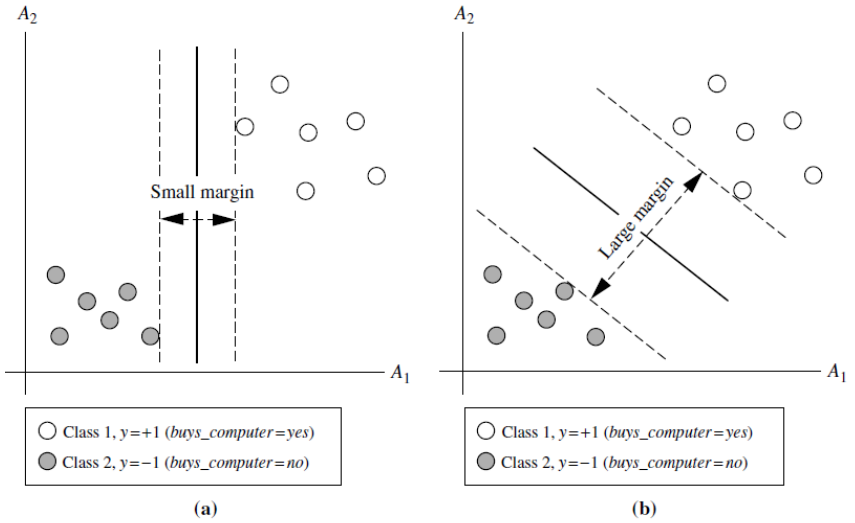


Figure 5.5 Two possible separating hyperplanes and their associated margins

Getting to an informal definition of margin, we can say that the shortest distance from a hyperplane to one side of its margin is equal to the shortest distance from the hyperplane to the other side of its margin, where the "sides" of the margin are parallel to the hyperplane. When dealing with the MMH, this distance is, in fact, the shortest distance from the MMH to the closest training tuple of either class. A separating hyperplane can be written as

$$\boldsymbol{W} \cdot \boldsymbol{X} + b = 0,$$

where W is a weight vector, namely, $W = \{w_1, w_2, ..., w_n\}$; n is the number of attributes and b is a scalar, often referred to as a bias. To aid in visualization, let's consider two input attributes, A_1 and A_2, as in figure 5.5(b). Training tuples are 2-D (e.g., $X = (x_1, x_2)$), where x_1 and x_2 are the values of attributes A_1 and A_2, respectively, for X. If we think of b as an additional weight, w_0, we can rewrite as

$$w_0 + w_1x_1 + w_2x_2 = 0.$$

Thus, any point that lies above the separating hyperplane satisfies

$$w_0 + w_1x_1 + w_2x_2 > 0$$

Similarly, any point that lies below the separating hyperplane satisfies

$$w_0 + w_1x_1 + w_2x_2 < 0.$$

The weights can be adjusted so that the hyperplanes defining the "sides" of the margin can be written as

$$H_1 : w_0 + w_1x_1 + w_2x_2 \geq 1 \text{ for } y_i = +1,$$

$$H_2 : w_0 + w_1x_1 + w_2x_2 \leq -1 \text{ for } y_i = -1.$$

That is, any tuple that falls on or above H1 belongs to class +1, and any tuple that falls on or below H2 belongs to class -1. Combining the two inequalities of Eqs. we get

$$y_i(w_0 + w_1x_1 + w_2x_2) \geq 1, \ \forall i$$

Any training tuples that fall on hyperplanes H_1 or H_2 (i.e., the "sides" defining the margin) are called support vectors. That is, they are equally close to the (separating) MMH. In figure 5.6, the support vectors are shown encircled with a thicker border. Essentially, the support vectors are the most difficult tuples to classify and give the most information regarding classification.

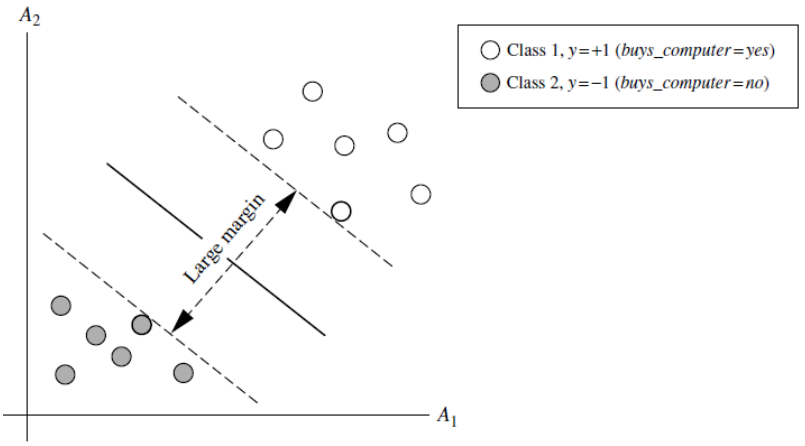


Figure 5.6 Support vectors.

From this, we can obtain a formula for the size of the maximal margin. The distance from the separating hyperplane to any point on H_1 is $\frac{1}{||W||}$, where $||W||$ is the Euclidean norm of W, that is, $\sqrt{W.W^2}$. By definition, this is equal to the distance from any point on H_2 to the separating hyperplane. Therefore, the maximal margin is $\frac{2}{||W||}$.

SVM finds the MMH and the support vectors by using some "fancy math tricks," known as a constrained (convex) quadratic optimization problem. If the data are small (say, less than 2000 training tuples), any optimization software package for solving constrained convex quadratic problems can then be used to find the support vectors and MMH. For larger data, special and more efficient algorithms for training SVMs can be used instead. Once we've found the

support vectors and MMH (note that the support vectors define the MMH!), we have a trained support vector machine. The MMH is a linear class boundary, and so the corresponding SVM can be used to classify linearly separable data. We refer to such a trained SVM as a linear SVM.

The trained support vector machine, can be used to classify test (i.e., new) tuples basing on the Lagrangian formulation, the MMH can be rewritten as the decision boundary

$$d(\boldsymbol{X}^T) = \sum_{i=1}^{l} y_i \alpha_i \boldsymbol{X}_i \boldsymbol{X}^T + b_0.$$

where y_i is the class label of support vector X_i; X^T is a test tuple; α_i and b_0 are numeric parameters that were determined automatically by the optimization or SVM algorithm noted before; and l is the number of support vectors. For linearly separable data, the support vectors are a subset of the actual training tuples.

Given a test tuple, X^T , and then check to see the sign of the result. This tells us on which side of the hyperplane the test tuple falls. If the sign is positive, then X^T falls on or above the MMH, and so the SVM predicts that X^T belongs to class +1 (representing buys_computer = yes, in our case). If the sign is negative, then X^T falls on or below the MMH and the class prediction is -1 (representing buys_computer = no).

5.4.2 The Case When the Data Are Linearly Inseparable

If the data is not linearly separable, as in figure 5.7 then there is no straight line that would separate the classes. The linear SVMs we studied would not be able to find a feasible solution here.

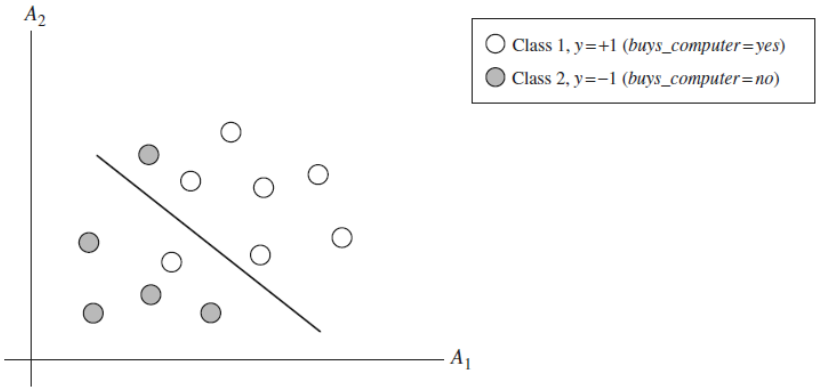


Figure 5.7 A simple 2-D case showing linearly inseparable data

Extend the linear SVMs approach to obtain a nonlinear SVM as follows. There are two main steps. In the first step, we transform the original input data into a higher dimensional space using a nonlinear mapping. Once the data have been transformed into the new higher space, the second step searches for a linear separating hyperplane in the new space. We again end

up with a quadratic optimization problem that can be solved using the linear SVM formulation. The maximal marginal hyperplane found in the new space corresponds to a nonlinear separating hypersurface in the original space.

For nonlinear transformation of original input data into a higher dimensional space, consider the following example. A 3-D input vector X = (x_1, x_2, x_3) is mapped into a 6-D space, *Z*, using the mappings $\Phi_1(X) = x_1$, $\Phi_2(X) = x_2$, $\Phi_3(X) = x_3$, $\Phi_4(X) = (x_1)^2$, $\Phi_5(X) = x_1x_2$, and $\Phi_6(X) = x_1x_3$. A decision hyperplane in the new space is $d(Z) = WZ + b$, where *W* and *Z* are vectors. This is linear. We solve for *W* and *b* and then substitute back so that the linear decision hyperplane in the new (*Z*) space corresponds to a nonlinear second-order polynomial in the original 3-D input space:

$$\begin{aligned} d(\boldsymbol{Z}) &= w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 (x_1)^2 + w_5 x_1 x_2 + w_6 x_1 x_3 + b \\ &= w_1 z_1 + w_2 z_2 + w_3 z_3 + w_4 z_4 + w_5 z_5 + w_6 z_6 + b. \end{aligned}$$

A nonlinear function that creates a new dimension is referred to as a kernel. The SVM kernel is also a function that takes low-dimensional input space and transforms it into higher-dimensional space, i.e. it converts non separable problem to separable problem. It is mostly useful in non-linear separation problems. Simply put the kernel, does some extremely complex data transformations then finds out the process to separate the data based on the labels or outputs defined.

Chapter-6

Distance Based Models

Instructor Name: B N V Narasimha Raju

6.1 Nearest Neighbours classification

K Nearest Neighbor (KNN) is very simple, easy to understand, versatile and one of the topmost machine learning algorithms. It is used in a variety of applications such as finance, healthcare, political science, handwriting detection, image recognition and video recognition. For example:

- In Credit ratings, financial institutes use KNN to predict the credit rating of customers
- In loan disbursement, banking institutes use KNN to predict whether a loan is safe or risky
- In political science, KNN is used to classify potential voters in two classes (those who will vote or won't vote)

In this you will learn all about the K-Nearest Neighbor (KNN) algorithm and build a KNN classifier using the Python scikit-learn package.

6.1.1 K-Nearest Neighbors

KNN is a non-parametric and lazy learning algorithm. Non-parametric means there is no assumption for underlying data distribution. In other words, the model structure is determined from the dataset. This will be very helpful in practice where most of the real world datasets do not follow mathematical theoretical assumptions. Lazy algorithm means it does not need any training data points for model generation. All training data used in the testing phase. This makes training faster and the testing phase slower and costlier. Costly testing phase means time and memory. In the worst case, KNN needs more time to scan all data points and scanning all data points will require more memory for storing training data.

6.1.2 KNN algorithm

The K-NN working can be explained as follows

- Step-1: Select the number *K* of the neighbors
- Step-2: Calculate the distance of *K* number of neighbors
- Step-3: Take the *K* nearest neighbors as per the calculated distance.

- Step-4: Among these *K* neighbors, count the number of the data points in each category.
- Step-5: Assign the new data points to that category for which the number of the neighbor is maximum.

In KNN, *K* is the number of nearest neighbors. The number of neighbors is the core deciding factor. *K* is generally an odd number if the number of classes is 2. When *K*=1, then the algorithm is known as the nearest neighbor algorithm as shown in figure 6.1. This is the simplest case. Suppose *P1* is the point, for which the label needs to predict. First, you find the one closest point to *P1* and then the label of the nearest point assigned to *P1*.

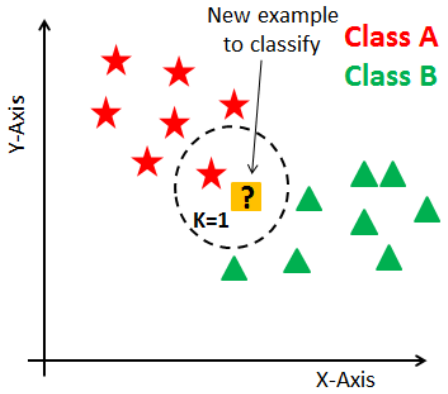


Figure 6.1. Nearest Neighbor Classification when K=1

Suppose *P1* is the point, for which the label needs to predict. First, you find the k closest point to *P1* and then classify points by majority vote of its *K* neighbors. Each object votes for their class and the class with the most votes is taken as the prediction. For finding closest similar points, you find the distance between points using distance measures such as Euclidean distance, Hamming distance, Manhattan distance and Minkowski distance. KNN has the following basic steps as shown in Figure 6.2

- Calculate distance
- Find closest neighbors
- Vote for labels

6.1.3 Eager Vs. Lazy Learners

Eager learners mean when given training points will construct a generalized model before performing prediction on given new points to classify. You can think of such learners as being ready, active and eager to classify unobserved data points.

Lazy Learning means there is no need for learning or training of the model and all of the data points used at the time of prediction. Lazy learners wait until the last minute before classifying any data point. Lazy learner stores merely the training dataset and waits until classification needs to perform. Only when it sees the test tuple does it perform generalization to classify the tuple based on its similarity to the stored training tuples. Unlike eager learning methods,

lazy learners do less work in the training phase and more work in the testing phase to make a classification. Lazy learners are also known as instance-based learners because lazy learners store the training points or instances, and all learning is based on instances.

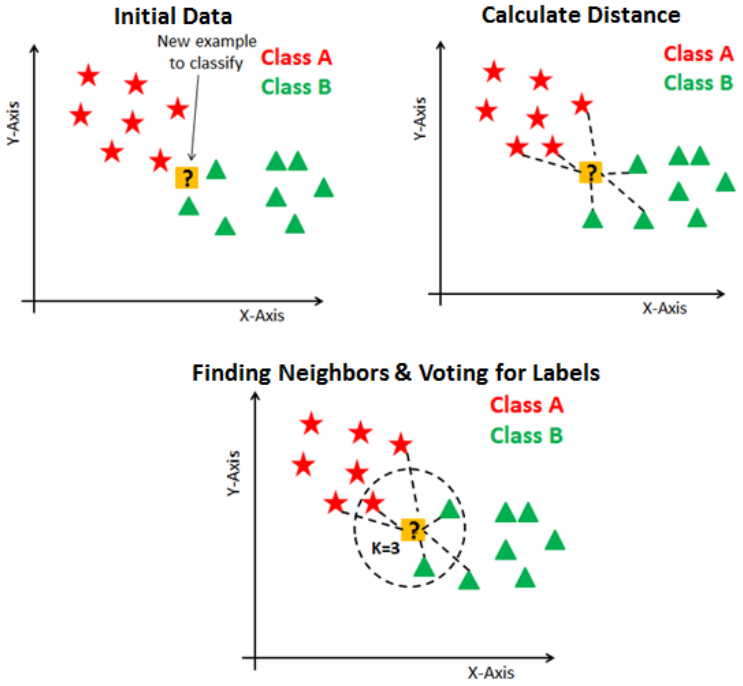


Fig 6.2 KNN Classification when K=3

6.1.4 Curse of Dimensionality

KNN performs better with a lower number of features than a large number of features. You can say that when the number of features increases then it requires more data. Increase in dimension also leads to the problem of overfitting. To avoid overfitting, the needed data will need to grow exponentially as you increase the number of dimensions. This problem of higher dimension is known as the Curse of Dimensionality.

To deal with the problem of the curse of dimensionality, you need to perform principal component analysis before applying any machine learning algorithm, or you can also use feature selection approach. Research has shown that in large dimensions Euclidean distance is not useful anymore. Therefore, you can prefer other measures such as cosine similarity, which get decidedly less affected by high dimensions.

6.1.5 Number of Neighbors in KNN

Now, you understand the KNN algorithm working mechanism. At this point, the question arises: How to choose the optimal number of neighbors? And what are its effects on the classifier? The number of neighbors(*K*) in KNN is a hyperparameter that you need to choose at the time of model building. You can think of K as a controlling variable for the prediction model.

Research has shown that no optimal number of neighbors suits all kinds of data sets. Each dataset has its own requirements. In the case of a small number of neighbors, the noise will have a higher influence on the result, and a large number of neighbors make it computationally expensive. Research has also shown that a small number of neighbors are the most flexible fit which will have low bias but high variance and a large number of neighbors will have a smoother decision boundary which means lower variance but higher bias.

Generally, Data scientists choose an odd number if the number of classes is even. You can also check by generating the model on different values of *K* and check their performance. You can also try the Elbow method as shown in figure 6.3.

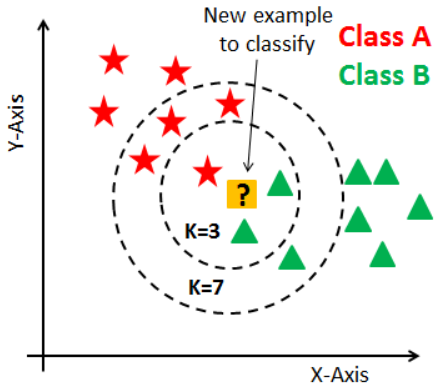


Figure 6.3 Elbow Method

The Elbow method is one of the most popular methods to determine this optimal value of *K*. The user must specify in advance what *K* to choose. It assigns all members to K clusters even if that is not the right *K* for the dataset. Lets create a for loop that trains various KNN models with different *K* values, then keep track of the error rate for each of these models. The *K* value having the least error rate will be the optimal value. Retrain with new *K* Value.

For example We have data from questionnaires, surveys (to ask people's opinion) and objective testing with two attributes (acid durability and strength) to classify whether a special paper tissue is good or not. Here is four training samples

X1 = Acid Durability (seconds)	**X2 = Strength (kg/square meter)**	**Y = Classification**
7	7	Bad
7	4	Bad
3	4	Good
1	4	Good

Now the factory produces a new paper tissue that passes the laboratory test with X1 = 3 and X2 = 7. Without another expensive survey, can we guess what the classification of this new tissue is?

Determine parameter K = number of nearest neighbors. Suppose use K = 3, calculate the distance between the query-instance and all the training samples

Coordinate of query instance is (3, 7), instead of calculating the distance we compute square distance which is faster to calculate (without square root)

X1 = Acid Durability (seconds)	**X2 = Strength (kg/square meter)**	**Square Distance to query instance (3, 7)**
7	7	$(7-3)^2 + (7-7)^2 = 16$
7	4	$(7-3)^2 + (4-7)^2 = 25$
3	4	$(3-3)^2 + (4-7)^2 = 9$
1	4	$(1-3)^2 + (4-7)^2 = 13$

Sort the distance and determine nearest neighbors based on the K-th minimum distance

X1 = Acid Durability (seconds)	**X2 = Strength (kg/square meter)**	**Square Distance to query instance (3, 7)**	**Rank minimum distance**	**Is it included in 3-Nearestneighbors?**
7	7	$(7-3)^2 + (7-7)^2 = 16$	3	Yes
7	4	$(7-3)^2 + (4-7)^2 = 25$	4	No
3	4	$(3-3)^2 + (4-7)^2 = 9$	1	Yes
1	4	$(1-3)^2 + (4-7)^2 = 13$	2	Yes

Gather the category *Y* of the nearest neighbors. Notice in the second row last column that the category of nearest neighbor (Y) is not included because the rank of this data is more than 3 (=K).

X1 = Acid Durability (seconds)	**X2 = Strength (kg/square meter)**	**Square Distance to query instance (3, 7)**	**Rank minimum distance**	**Is it included in 3-Nearestneighbors?**	**Y = Category of nearest Neighbor**
7	7	$(7-3)^2 + (7-7)^2 = 16$	3	Yes	Bad

7	4	$(7-3)^2 + (4-7)^2 = 25$	4	No	-
3	4	$(3-3)^2 + (4-7)^2 = 9$	1	Yes	Good
1	4	$(1-3)^2 + (4-7)^2 = 13$	2	Yes	Good

Use simple majority of the category of nearest neighbors as the prediction value of the query instance. We have 2 good and 1 bad, since 2 > 1 then we conclude that a new paper tissue that passes the laboratory test with X1 = 3 and X2 = 7 is included in the Good category.

6.2 Distance Based Clustering

Clustering is the process of grouping a set of data objects into multiple groups or clusters so that objects within a cluster have high similarity, but are very dissimilar to objects in other clusters. Dissimilarities and similarities are assessed based on the attribute values describing the objects and often involve distance measures. Distance-based cluster analysis is based on *k*-means, *k*-medoids, and on several other methods. Clustering is known as unsupervised learning because the class label information is not present. For this reason, clustering is a form of learning by observation, rather than learning by examples.

The simplest and most fundamental version of cluster analysis is partitioning. Most partitioning methods are distance-based. Clustering is also called data segmentation in some applications because clustering partitions large data sets into groups according to their similarity.

Assume that the number of clusters is given as background knowledge. This parameter is the starting point for partitioning methods. Given a data set, *D*, of *n* objects, and *k*, the number of partitions to construct where each partition represents a cluster and $k \leq n$. That is, it divides the data into *k* groups such that each group must contain at least one object. A partitioning method creates an initial partitioning. It then uses an iterative relocation technique that attempts to improve the partitioning by moving objects from one group to another. The general criterion of a good partitioning is that objects in the same cluster are “similar” to one another and “dissimilar” to objects in other clusters.

6.2.1 k-Means: A Centroid-Based Technique

Suppose a data set, *D*, contains *n* objects in Euclidean space. Partitioning methods distribute the objects in *D* into *k* clusters, C_1, C_2, ...,C_k. An objective function is used to assess the partitioning quality. This is, the objective function aims for high intra cluster similarity and low inter cluster similarity.

A centroid-based partitioning technique uses the centroid of a cluster, C_i, to represent that cluster. Conceptually, the centroid of a cluster is its center point. The centroid can be defined in various ways such as by the mean or medoid of the objects (or points) assigned to the cluster. The difference between an object $p \in C_i$ and c_i, is measured by dist(p, c_i), where dist(x, y) is the Euclidean distance between two points x and y. The quality of cluster C_i can be measured by the within cluster variation, which is the sum of squared error between all objects in C_i and the centroid c_i, defined as

$$E = \sum_{i=1}^{k} \sum_{p \in C_i} dist(\boldsymbol{p}, \boldsymbol{c_i})^2$$

where E is the sum of the squared error for all objects in the data set; p is the point in space representing a given object; and c_i is the centroid of cluster C_i.

The k-means algorithm defines the centroid of a cluster as the mean value of the points within the cluster. It proceeds as follows. First, it randomly selects k of the objects in D, each of which initially represents a cluster mean or center. For each of the remaining objects, an object is assigned to the cluster to which it is the most similar, based on the Euclidean distance between the object and the cluster mean. The k-means algorithm then iteratively improves the within-cluster variation. For each cluster, it computes the new mean using the objects assigned to the cluster in the previous iteration. All the objects are then reassigned using the updated means as the new cluster centers. The iterations continue until the assignment is stable, that is, the clusters formed in the current round are the same as those formed in the previous round. The k-means procedure is summarized in Figure 6.4.

Algorithm: *k*-means. The *k*-means algorithm for partitioning, where each cluster's center is represented by the mean value of the objects in the cluster.

Input:

- k: the number of clusters,
- D: a data set containing n objects.

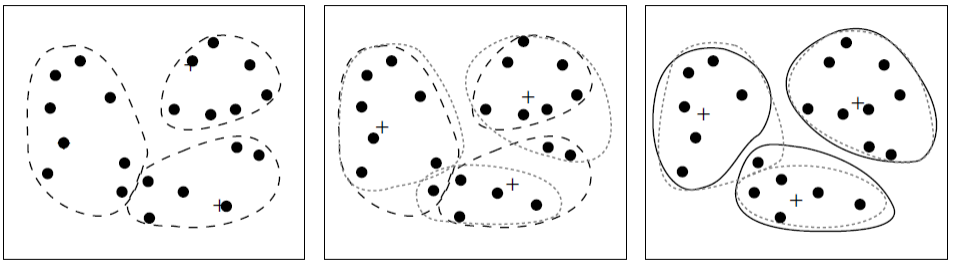
Output: A set of k clusters.

Method:

(1) arbitrarily choose k objects from D as the initial cluster centers;
(2) **repeat**
(3) (re)assign each object to the cluster to which the object is the most similar, based on the mean value of the objects in the cluster;
(4) update the cluster means, that is, calculate the mean value of the objects for each cluster;
(5) **until** no change;

Figure 6.4 The k-means partitioning algorithm.

Clustering by *k*-means partitioning. Consider a set of objects located in 2-D space, as depicted in Figure 6.5(a). Let $k = 3$, that is, the user would like the objects to be partitioned into three clusters.



(a) Initial clustering (b) Iterate (c) Final clustering

Figure 6.5 Clustering of a set of objects using the k-means method

According to the algorithm, we arbitrarily choose three objects as the three initial cluster centers, where cluster centers are marked by a +. Each object is assigned to a cluster based on the cluster center to which it is the nearest. Such a distribution forms silhouettes encircled by dotted curves, as shown in Figure 6.5(a).

Next, the cluster centers are updated. That is, the mean value of each cluster is recalculated based on the current objects in the cluster. Using the new cluster centers, the objects are redistributed to the clusters based on which cluster center is the nearest. Such a redistribution forms new silhouettes encircled by dashed curves, as shown in Figure 6.5(b).

This process iterates, leading to Figure 6.5(c). The process of iteratively reassigning objects to clusters to improve the partitioning is referred to as iterative relocation. Eventually, no reassignment of the objects in any cluster occurs and so the process terminates. The resulting clusters are returned by the clustering process.

There are several variants of the *k*-means method. These can differ in the selection of the initial *k*-means, the calculation of dissimilarity, and the strategies for calculating cluster means. The k-means method can be applied only when the mean of a set of objects is defined. The *k*-modes method is a variant of *k*-means, which extends the *k*-means to cluster nominal data by replacing the means of clusters with modes. It uses new dissimilarity measures to deal with nominal objects and a frequency-based method to update modes of clusters.

The necessity for users to specify *k*, the number of clusters, in advance can be seen as a disadvantage. To overcome this difficulty, provide an approximate range of *k* values, and then use an analytical technique to determine the best *k* by comparing the clustering results obtained for the different *k* values. The *k*-means method is not suitable for discovering clusters with nonconvex shapes or clusters of very different sizes. It is also not suitable when the data has noise and outliers because it can influence the mean value.

Example: Use K-Means Algorithm to create two clusters for the dataset in figure 6.6

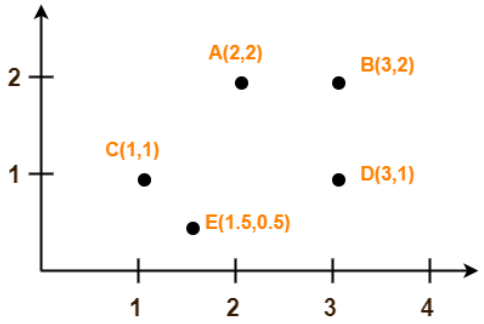


Figure 6.6 Dataset for k-means clustering

We follow the above discussed *K*-Means Clustering Algorithm. Assume A(2, 2) and C(1, 1) are centers of the two clusters.

Step-1:

We calculate the distance of each point from each of the centers of the two clusters. The distance is calculated by using the euclidean distance formula.

Calculate the distance between A(2, 2) and C1(2, 2)-

$P(A, C1) = \text{sqrt}\ [\ (x2 - x1)^2 + (y2 - y1)^2\ ] = \text{sqrt}\ [\ (2 - 2)^2 + (2 - 2)^2\ ] = \text{sqrt}\ [\ 0 + 0\ ] = 0$

Calculate distance between A(2, 2) and C2(1, 1)

$P(A, C2) = \text{sqrt}\ [\ (x2 - x1)^2 + (y2 - y1)^2\ ] = \text{sqrt}\ [\ (1 - 2)^2 + (1 - 2)^2\ ] = \text{sqrt}\ [\ 1 + 1\ ] = \text{sqrt}\ [\ 2\ ] = 1.41$

In the similar manner, we calculate the distance of other points from each of the centers of the two clusters.

Given Points	Distance from center (2, 2) of Cluster-01	Distance from center (1, 1) of Cluster-02	Point belongs to Cluster
A(2, 2)	0	1.41	C1
B(3, 2)	1	2.24	C1
C(1, 1)	1.41	0	C2
D(3, 1)	1.41	2	C1
E(1.5, 0.5)	1.58	0.71	C2

New clusters are

Cluster-01: First cluster contains points-

A(2, 2)

B(3, 2)

D(3, 1)

Cluster-02: Second cluster contains points-

C(1, 1)

E(1.5, 0.5)

Now, We re-compute the new cluster centers. The new cluster center is computed by taking the mean of all the points contained in that cluster.

For Cluster-01: Center of Cluster-01

= ((2 + 3 + 3)/3, (2 + 2 + 1)/3)

= (2.67, 1.67)

For Cluster-02: Center of Cluster-02

= ((1 + 1.5)/2, (1 + 0.5)/2)

= (1.25, 0.75)

This is the completion of Iteration-01. Next, we go to iteration-02, iteration-03 and so on until the centers do not change anymore.

6.2.2 *k*-Medoids: A Representative Object-Based Technique

A drawback of *k*-means. Consider six points in 1-D space having the values 1, 2, 3, 8, 9, 10, and 25, respectively. Intuitively, by visual inspection we may imagine the points partitioned into the clusters {1, 2, 3} and {8, 9, 10}, where point 25 is excluded because it appears to be an outlier. If we apply *k*-means using $k = 2$, the partitioning {{1, 2, 3}, {8, 9, 10, 25}} has the within-cluster variation

$$(1-2)^2+(2-2)^2+(3-2)^2+(8-13)^2+(9-13)^2+(10-13)^2+(25-13)^2=196,$$

given that the mean of cluster {1, 2, 3} is 2 and the mean of {8, 9, 10, 25} is 13. Compare this to the partitioning {{1, 2, 3, 8}, {9, 10, 25}}, for which *k*-means computes the within cluster variation as

$$\begin{aligned}(1-3.5)^2+(2-3.5)^2+(3-3.5)^2+(8-3.5)^2+(9-14.67)^2\\+(10-14.67)^2+(25-14.67)^2=189.67,\end{aligned}$$

given that 3.5 is the mean of cluster {1, 2, 3,8} and 14.67 is the mean of cluster {9, 10, 25}. The latter partitioning has the lowest within-cluster variation; therefore, the *k*-means method assigns the value 8 to a cluster different from that containing 9 and 10 due to the outlier point

25.Moreover, the center of the second cluster, 14.67, is substantially far from all the members in the cluster.

Instead of taking the mean value of the objects in a cluster as a reference point, we can pick actual objects to represent the clusters, using one representative object per cluster. Each remaining object is assigned to the cluster of which the representative object is the most similar. The partitioning method is then performed based on the principle of minimizing the sum of the dissimilarities between each object p and its corresponding representative object. That is, an absolute-error criterion is used, defined as

$$E = \sum_{i=1}^{k} \sum_{\boldsymbol{p} \in C_i} dist(\boldsymbol{p}, \boldsymbol{o}_i)$$

where E is the sum of the absolute error for all objects p in the data set, and o_i is the representative object of C_i. This is the basis for the k-medoids method, which groups n objects into k clusters by minimizing the absolute error.

The Partitioning Around Medoids (PAM) algorithm is a popular realization of k-medoids clustering as shown in figure 6.7. It tackles the problem in an iterative, greedy way. Like the k-means algorithm, the initial representative objects (called seeds) are chosen arbitrarily. We consider whether replacing a representative object by a nonrepresentative object would improve the clustering quality. All the possible replacements are tried out. The iterative process of replacing representative objects by other objects continues until the quality of the resulting clustering cannot be improved by any replacement. This quality is measured by a cost function of the average dissimilarity between an object and the representative object of its cluster.

Algorithm: k-medoids. PAM, a k-medoids algorithm for partitioning based on medoid or central objects.

Input:

- k: the number of clusters,
- D: a data set containing n objects.

Output: A set of k clusters.

Method:

(1) arbitrarily choose k objects in D as the initial representative objects or seeds;
(2) **repeat**
(3) assign each remaining object to the cluster with the nearest representative object;
(4) randomly select a nonrepresentative object, $\boldsymbol{o}_{random}$;
(5) compute the total cost, S, of swapping representative object, $\boldsymbol{o}_j$, with $\boldsymbol{o}_{random}$;
(6) **if** $S < 0$ **then** swap $\boldsymbol{o}_j$ with $\boldsymbol{o}_{random}$ to form the new set of k representative objects;
(7) **until** no change;

Figure 6.7 PAM, a k-medoids partitioning algorithm.

Specifically, let o_1, ..., o_k be the current set of representative objects (i.e., medoids). To determine whether a nonrepresentative object, denoted by o_{random}, is a good replacement for

a current medoid o_j ($1 \leq j \leq k$), we calculate the distance from every object p to the closest object in the set $\{o_1, \ldots, o_{j-1}, o_{random}, o_{j+1}, \ldots, o_k\}$, and use the distance to update the cost function. The reassignments of objects to $\{o_1, \ldots, o_{j-1}, o_{random}, o_{j+1}, \ldots, o_k\}$ are simple. Suppose object p is currently assigned to a cluster represented by medoid o_j (Figure 6.8 a or b). If o_j is being replaced by o_{random} then object p needs to be reassigned to either o_{random} or some other cluster represented by o_i ($i \neq j$)whichever is the closest.

For example, in Figure 6.8 (a), p is closest to o_i and therefore is reassigned to o_i. In Figure 6.8(b), however, p is closest to o_{random} and so is reassigned to o_{random}. If p is assigned to some other object, object o remains assigned to the cluster represented by o_i as long as o is still closer to o_i than to o_{random} (Figure 6.8c). Otherwise, o is reassigned to o_{random} (Figure 6.8d).

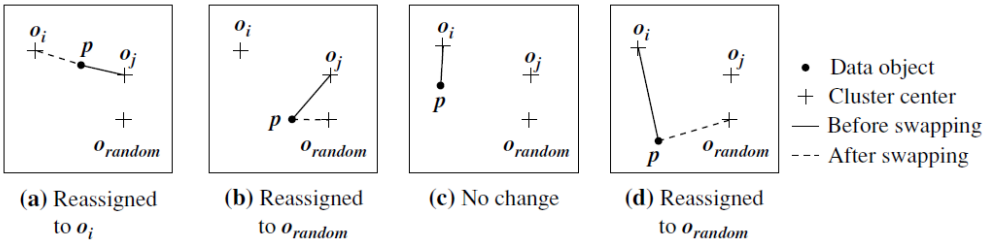


Figure 6.8 Four cases of the cost function for k-medoids clustering

Each time a reassignment occurs, a difference in absolute error, E, is contributed to the cost function. If the total cost is negative, then o_j is replaced or swapped with o_{random} because the actual absolute-error E is reduced. If the total cost is positive, the current representative object, o_j, is considered acceptable, and nothing is changed in the iteration.

6.3 Hierarchical Methods

A hierarchical clustering method works by grouping data objects into a hierarchy or "tree" of clusters. Hierarchical clustering methods can encounter difficulties regarding the selection of merge or split points. Such a decision is critical, because once a group of objects is merged or split, the process at the next step will operate on the newly generated clusters. It will neither undo what was done previously, nor perform object swapping between clusters. Thus, merge or split decisions, if not well chosen, may lead to low-quality clusters. Moreover, the methods do not scale well because each decision of merge or split needs to examine and evaluate many objects or clusters.

6.3.1 Agglomerative versus Divisive Hierarchical Clustering

A hierarchical clustering method can be either agglomerative or divisive, depending on whether the hierarchical decomposition is formed in a bottom-up (merging) or top-down (splitting) fashion.

An agglomerative hierarchical clustering method uses a bottom-up strategy. It typically starts by letting each object form its own cluster and iteratively merges clusters into larger and larger clusters, until all the objects are in a single cluster or certain termination conditions are satisfied. The single cluster becomes the hierarchy's root. For the merging step, it finds the two clusters that are closest to each other (according to some similarity measure), and combines the two to form one cluster.

A divisive hierarchical clustering method employs a top-down strategy. It starts by placing all objects in one cluster, which is the hierarchy's root. It then divides the root cluster into several smaller subclusters, and recursively partitions those clusters into smaller ones. The partitioning process continues until each cluster at the lowest level is coherent enough—either containing only one object, or the objects within a cluster are sufficiently similar to each other.

In either agglomerative or divisive hierarchical clustering, a user can specify the desired number of clusters as a termination condition.

In Figure 6.9 shows the application of AGNES (AGglomerative NESting), an agglomerative hierarchical clustering method, and DIANA (DIvisive ANAlysis), a divisive hierarchical clustering method, on a data set of five objects, {a, b, c, d, e}. Initially, AGNES, the agglomerative method, places each object into a cluster of its own. The clusters are then merged step-by-step according to some criterion.

For example, clusters C_1 and C_2 may be merged if an object in C_1 and an object in C_2 form the minimum Euclidean distance between any two objects from different clusters. This is a single-linkage approach in that each cluster is represented by all the objects in the cluster, and the similarity between two clusters is measured by the similarity of the closest pair of data points belonging to different clusters. The cluster-merging process repeats until all the objects are eventually merged to form one cluster

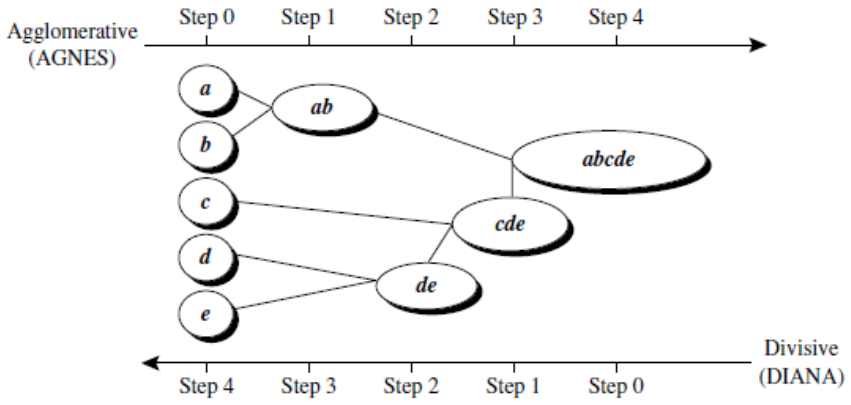


Figure 6.9 Agglomerative and divisive hierarchical clustering on data objects {a, b, c, d, e}.

DIANA, the divisive method, proceeds in the contrasting way. All the objects are used to form one initial cluster. The cluster is split according to some principle such as the maximum

Euclidean distance between the closest neighboring objects in the cluster. The cluster-splitting process repeats until, eventually, each new cluster contains only a single object.

A tree structure called a dendrogram is commonly used to represent the process of hierarchical clustering. It shows how objects are grouped together (in an agglomerative method) or partitioned (in a divisive method) step-by-step.

Figure 6.10 shows a dendrogram for the five objects presented in Figure 6.9, where $l = 0$ shows the five objects as singleton clusters at level 0. At $l = 1$, objects a and b are grouped together to form the first cluster, and they stay together at all subsequent levels.We can also use a vertical axis to show the similarity scale between clusters. For example, when the similarity of two groups of objects, {a, b} and {c, d, e}, is roughly 0.16, they are merged together to form a single cluster.

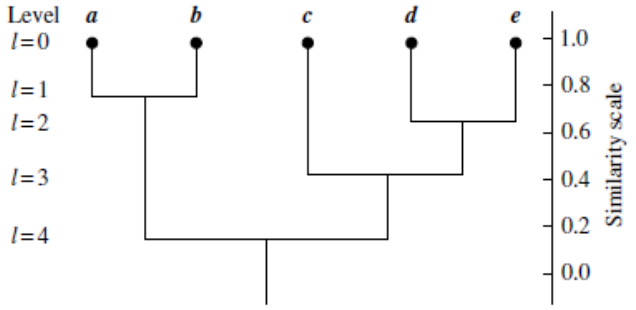


Figure 6.10 Dendrogram representation for hierarchical clustering of data objects {a, b, c, d, e}.

A challenge with divisive methods is how to partition a large cluster into several smaller ones. For example, there are $2^{n-1} - 1$ possible ways to partition a set of n objects into two exclusive subsets, where n is the number of objects. When n is large, it is computationally prohibitive to examine all possibilities.

Consequently, a divisive method typically uses heuristics in partitioning, which can lead to inaccurate results. For the sake of efficiency, divisive methods typically do not backtrack on partitioning decisions that have been made. Once a cluster is partitioned, any alternative partitioning of this cluster will not be considered again. Due to the challenges in divisive methods, there are many more agglomerative methods than divisive methods.

6.3.2 Distance Measures in Algorithmic Methods

Whether using an agglomerative method or a divisive method, a core need is to measure the distance between two clusters, where each cluster is generally a set of objects. Four widely used measures for distance between clusters are as follows, where $|p - p'|$ is the distance between two objects or points, p and p'; m_i is the mean for cluster, C_i; and n_i is the number of objects in C_i. They are also known as linkage measures.

$$\textbf{Minimum distance:} \quad dist_{min}(C_i, C_j) = \min_{\boldsymbol{p} \in C_i, \boldsymbol{p}' \in C_j} \{|\boldsymbol{p} - \boldsymbol{p}'|\}$$

$$\textbf{Maximum distance:} \quad dist_{max}(C_i, C_j) = \max_{\boldsymbol{p} \in C_i, \boldsymbol{p}' \in C_j} \{|\boldsymbol{p} - \boldsymbol{p}'|\}$$

$$\textbf{Mean distance:} \quad dist_{mean}(C_i, C_j) = |\boldsymbol{m_i} - \boldsymbol{m_j}|$$

$$\textbf{Average distance:} \quad dist_{avg}(C_i, C_j) = \frac{1}{n_i n_j} \sum_{\boldsymbol{p} \in C_i, \boldsymbol{p}' \in C_j} |\boldsymbol{p} - \boldsymbol{p}'|$$

When an algorithm uses the minimum distance, $d_{min}(C_i, C_j)$, to measure the distance between clusters, it is sometimes called a nearest-neighbor clustering algorithm. Moreover, if the clustering process is terminated when the distance between nearest clusters exceeds a user-defined threshold, it is called a single-linkage algorithm.

Viewing the data points as nodes of a graph, with edges forming a path between the nodes in a cluster, then the merging of two clusters, C_i and C_j, corresponds to adding an edge between the nearest pair of nodes in C_i and C_j. Because edges linking clusters always go between distinct clusters, the resulting graph will generate a tree.

Agglomerative hierarchical clustering algorithm that uses the minimum distance measure is also called a minimal spanning tree algorithm, where a spanning tree of a graph is a tree that connects all vertices, and a minimal spanning tree is the one with the least sum of edge weights.

When an algorithm uses the maximum distance, $d_{max}(C_i, C_j)$, to measure the distance between clusters, it is sometimes called a farthest-neighbor clustering algorithm. If the clustering process is terminated when the maximum distance between nearest clusters exceeds a user-defined threshold, it is called a complete-linkage algorithm.

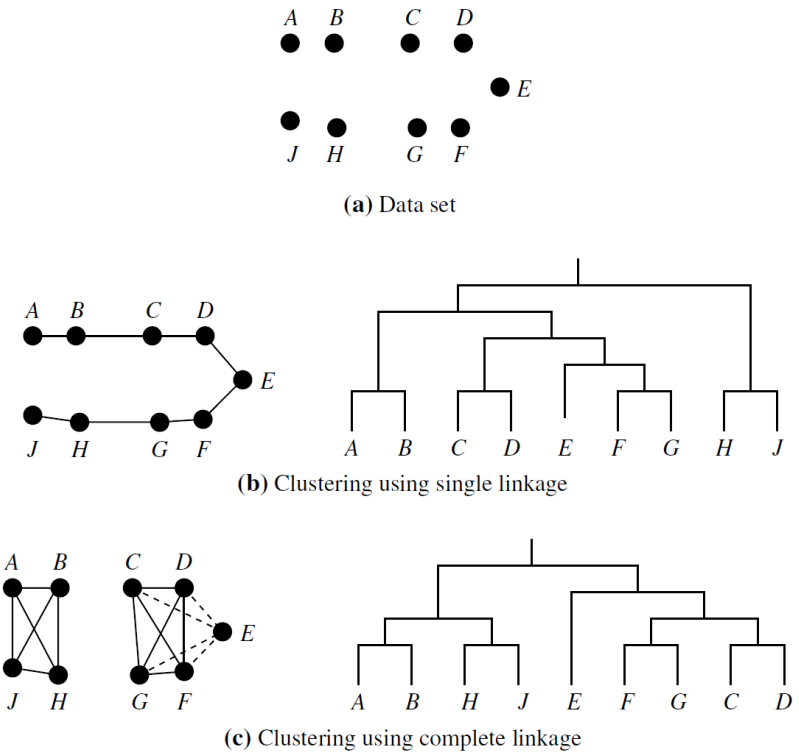
Viewing data points as nodes of a graph, with edges linking nodes, consider each cluster as a complete subgraph, that is, with edges connecting all the nodes in the clusters. The distance between two clusters is determined by the most distant nodes in the two clusters.

Farthest-neighbor algorithms tend to minimize the increase in diameter of the clusters at each iteration. If the true clusters are rather compact and approximately equal size, the method will produce high-quality clusters. Otherwise, the clusters produced can be meaningless.

6.3.3 Single versus complete linkages

Let us apply hierarchical clustering to the data set of figure 6.11(a). Figure 6.11(b) shows the dendrogram using single linkage. Figure 6.11(c) shows the case using complete linkage, where the edges between clusters {A, B, J, H} and {C, D, G, F, E} are omitted for ease of presentation.

This example shows that by using single linkages we can find hierarchical clusters defined by local proximity, whereas complete linkage tends to find clusters opting for global closeness.



(a) Data set

(b) Clustering using single linkage

(c) Clustering using complete linkage

Figure 6.11 Hierarchical clustering using single and complete linkages.

Chapter-7

Features

Instructor Name: B N V Narasimha Raju

Features, also called attributes, are defined as mappings f_i: $X \rightarrow F_i$ from the instance space X to the feature domain F_i. We can distinguish features by their domain: common feature domains include real and integer numbers, but also discrete sets such as colors, the Booleans, and so on. We can also distinguish features by the range of permissible operations. For example, we can calculate a group of people's average age but not their average blood type, so taking the average value is an operation that is permissible on some features but not on others.

Although many data sets come with pre-defined features, they can be manipulated in many ways. For example, we can change the domain of a feature by rescaling or discretization; we can select the best features from a larger set and only work with the selected ones; or we can combine two or more features into a new feature.

7.1 Kinds of feature

Consider two features, one describing a person's age and the other their house number. Both features map into the integers, but the way we use those features can be quite different. Calculating the average age of a group of people is meaningful, but an average house number is probably not very useful! In other words, what matters is not just the domain of a feature, but also the range of permissible operations. These, in turn, depend on whether the feature values are expressed on a meaningful scale. Despite appearances, house numbers are not really integers but ordinals: we can use them to determine that number 10's neighbors are number 8 and number 12, but we cannot assume that the distance between 8 and 10 is the same as the distance between 10 and 12. Because of the absence of a linear scale it is not meaningful to add or subtract house numbers, which precludes operations such as averaging.

7.1.1 Calculations on features

The range of possible calculations on features, often referred to as aggregates or statistics. Three main categories are statistics of central tendency, statistics of dispersion and shape statistics. Each of these can be interpreted either as a theoretical property of an unknown population or a concrete property of a given sample – here we will concentrate on sample statistics. Starting with statistics of central tendency, the most important ones are

- The mean or average value;

- The median, which is the middle value if we order the instances from lowest to highest feature value; and
- The mode, which is the majority value or values.

Of these statistics, the mode is the one we can calculate whatever the domain of the feature: so, for example, we can say that the most frequent blood type in a group of people is O+. In order to calculate the median, we need to have an ordering on the feature values: so we can calculate both the mode and the median house number in a set of addresses. In order to calculate the mean, we need a feature expressed on some scale: most often this will be a linear scale for which we calculate the familiar arithmetic mean. It is often suggested that the median tends to lie between the mode and the mean, but there are plenty of exceptions to this 'rule'. The famous statistician Karl Pearson suggested a more specific rule of thumb (with therefore even more exceptions): the median tends to fall one-third of the way from mean to mode.

The second kind of calculation on features are statistics of dispersion or 'spread'. Two well-known statistics of dispersion are the variance or average squared deviation from the (arithmetic) mean, and its square root, the standard deviation. Variance and standard deviation essentially measure the same thing, but the latter has the advantage that it is expressed on the same scale as the feature itself.

A simpler dispersion statistic is the difference between maximum and minimum value, which is called the range. A natural statistic of central tendency to be used with the range is the midrange point, which is the mean of the two extreme values.

Other statistics of dispersion include percentiles. The p-th percentile is the value such that p percent of the instances fall below it. If we have 100 instances, the 80th percentile is the value of the 81st instance in a list of increasing values. If p is a multiple of 25 the percentiles are also called quartiles, and if it is a multiple of 10 the percentiles are also called deciles. Percentiles, deciles and quartiles are special cases of quantiles. Once we have quantiles we can measure dispersion as the distance between different quantiles. For instance, the interquartile range is the difference between the third and first quartile (i.e., the 75th and 25th percentile).

7.1.1.1 Percentile plot

Suppose you are learning a model over an instance space of countries, and one of the features you are considering is the gross domestic product (GDP) per capita. Figure 10.1 shows a so-called percentile plot of this feature. In order to obtain the p-th percentile, you intersect the line $y = p$ with the dotted curve and read off the corresponding percentile on the x-axis. Indicated in the figure are the 25th, 50th and 75th percentile. Also indicated is the mean (which has to be calculated from the raw data). As you can see, the mean is considerably higher than the median; this is mainly because of a few countries with very high GDP per capita. In other words, the mean is more sensitive to outliers than the median, which is why themedian is often preferred to the mean for skewed distributions like this one.

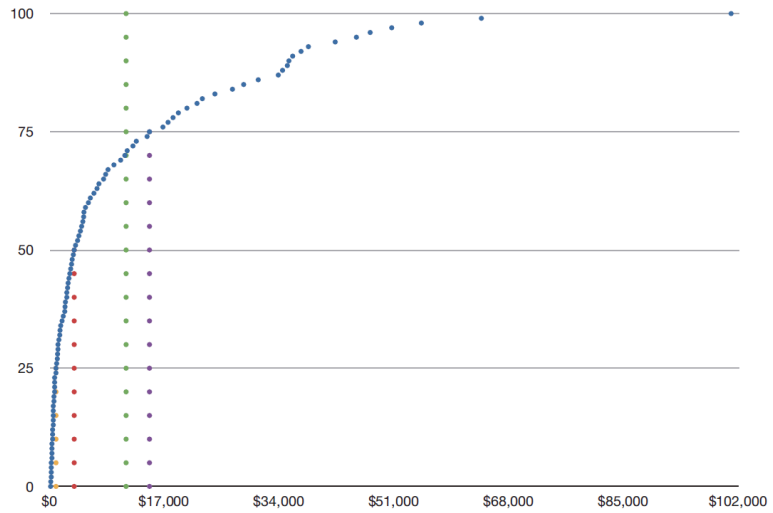


Figure 10.1. Percentile plot of GDP per capita for 231 countries.

In figure 10.1 the vertical dotted lines indicate, from left to right: the first quartile ($900); the median ($3600); the mean ($11 284); and the third quartile ($14 750). The interquartile range is $13 850, while the standard deviation is $16 189.

7.1.1.2 Symmetric data distribution

In a unimodal frequency curve with perfect symmetric data distribution, the mean, median, and mode are all at the same center value, as shown in Figure 2.1(a). Data in most real applications are not symmetric. They may instead be either positively skewed, where the mode occurs at a value that is smaller than the median (Figure 2.1b), or negatively skewed, where the mode occurs at a value greater than the median (Figure 2.1c).

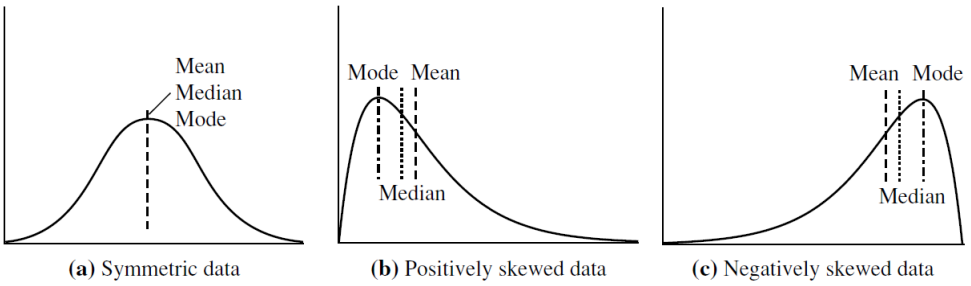


Figure 2.1 Mean, median, and mode of symmetric versus positively and negatively skewed data.

7.1.1.3 Cumulative probability distribution

By interpreting the y-axis as probabilities, the plot can be read as a cumulative probability distribution: a plot of $P(X \leq x)$ against x for a random variable X. For example, the plot shows that $P(X \leq \mu)$ is approximately 0.70, where μ = $11284 is the mean GDP per capita. In other words, if you choose a random country the probability that its GDP per capita is less than the average is about 0.70.

Since GDP per capita is a real-valued feature, it doesn't necessarily make sense to talk about its mode, since if you measure the feature precisely enough every country will have a

different value. We can get around this by means of a histogram, which counts the number of feature values in a particular interval or bin.

7.1.1.4 Histogram

A histogram of the data is shown in Figure 10.2. The left-most bin is the mode, with well over a third of the countries having a GDP per capita of not more than $2000. This demonstrates that the distribution is extremely right-skewed (i.e., has a long right tail), resulting in a mean that is considerably higher than the median.

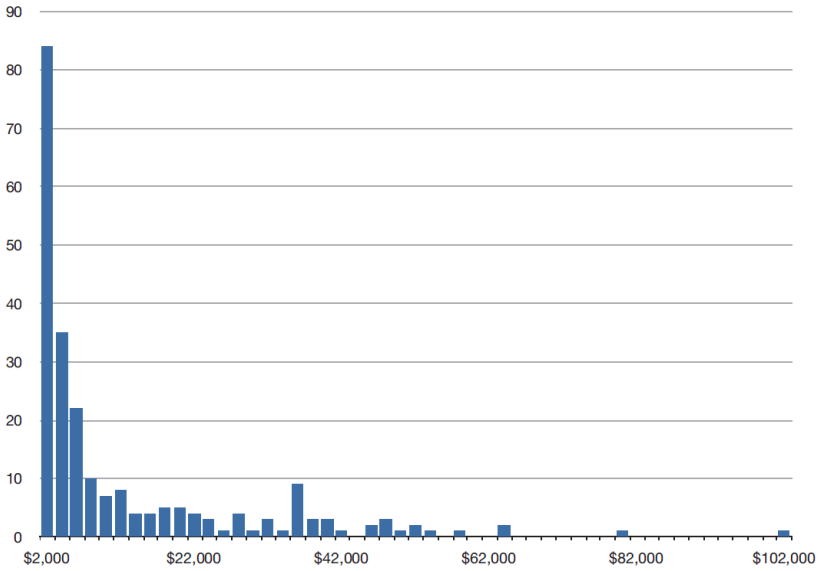


Figure 10.2. Histogram of the data from Figure 10.1, with bins of $2000 wide.

7.1.1.5 Skewness and Kurtosis

The skew and 'peakedness' of a distribution can be measured by shape statistics such as skewness and kurtosis. The main idea is to calculate the third and fourth central moment of the sample. In general, the k-th central moment of a sample $\{x_i, ..., x_n\}$ is defined as

$$m_k = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^k$$

where μ is the sample mean. Clearly, the first central moment is the average deviation from the mean – this is always zero, as the positive and negative deviations cancel each other out – and the second central moment is the average squared deviation from the mean, otherwise known as the variance. The third central moment m_3 can again be positive or negative.

Skewness is then defined as m_3 / σ^3, where σ is the sample's standard deviation. A positive value of skewness means that the distribution is right-skewed, which means that the right tail is longer than the left tail. Negative skewness indicates the opposite, left-skewed case.

Kurtosis is defined as m_4 / σ^4. As it can be shown that a normal distribution has kurtosis 3, people often use excess kurtosis $m_4 / \sigma^4 - 3$ as the statistic of interest. Briefly, positive excess kurtosis means that the distribution is more sharply peaked than the normal distribution.

In the GDP per capita example we can calculate skewness as 2.12 and excess kurtosis as 2.53. This confirms that the distribution is heavily right-skewed, and also more sharply peaked than the normal distribution.

7.1.2 Categorical, ordinal and quantitative features

Given these various statistics we can distinguish three main kinds of features: those with a meaningful numerical scale, those without a scale but with an ordering, and those without either. We will call features of the first type quantitative; they most often involve a mapping into the reals (another term in common use is 'continuous').

Features with an ordering but without scale are called ordinal features. The domain of an ordinal feature is some totally ordered set, such as the set of characters or strings. Another common example are features that express a rank order: first, second, third, and so on. Ordinal features allow the mode and median as central tendency statistics, and quantiles as dispersion statistics.

Features without ordering or scale are called categorical features (or sometimes 'nominal' features). They do not allow any statistical summary except the mode. One subspecies of the categorical features is the Boolean feature, which maps into the truth values true and false. The situation is summarized in Table 10.1.

Kind	*Order*	*Scale*	*Tendency*	*Dispersion*	*Shape*
Categorical	×	×	mode	n/a	n/a
Ordinal	√	×	median	quantiles	n/a
Quantitative	√	√	mean	range, interquartile range, variance, standard deviation	skewness, kurtosis

Table 10.1. Kinds of features, their properties and allowable statistics.

Models treat these different kinds of features in distinct ways. First, consider tree models such as decision trees. A split on a categorical feature will have as many children as there are feature values. Ordinal and quantitative features, on the other hand, give rise to a binary split. It follows that tree models are insensitive to the scale of quantitative features. For example, whether a temperature feature is measured on the Celsius scale or on the Fahrenheit scale will not affect the learned tree. Neither will switching from a linear scale to a logarithmic scale have any effect. In general, tree models are insensitive to monotonic transformations on the scale of a feature, which are those transformations that do not affect the relative order of the feature values. In effect, tree models ignore the scale of quantitative features, treating them as ordinal. The same holds for rule models.

While naive Bayes only really handles categorical features,many geometric models go in the other direction: they can only handle quantitative features. Linear models are a case in point: the very notion of linearity assumes a Euclidean instance space in which features act as Cartesian coordinates, and thus need to be quantitative.

7.1.3 Structured features

It is usually tacitly assumed that an instance is a vector of feature values. In other words, the instance space is a Cartesian product of d feature domains: $X = F_1 \times \ldots \times F_d$. This means that there is no other information available about an instance apart from the information conveyed by its feature values. Identifying an instance with its vector of feature values is what computer scientists call an abstraction, which is the result of filtering out unnecessary information. Representing an email as a vector of word frequencies is an example of an abstraction.

However, sometimes it is necessary to avoid such abstractions, and to keep more information about an instance than can be captured by a finite vector of feature values. For example, we could represent an email as a long string; or as a sequence of words and punctuation marks; or as a tree that captures the HTML mark-up; and so on. Features that operate on such structured instance spaces are called structured features.

Suppose an e-mail is represented as a sequence of words. This allows us to define, apart from the usual word frequency features, a host of other features, including:

- whether the phrase ‘machine learning’ – or any other set of consecutive words – occurs in the email.
- whether the email contains at least eight consecutive words in a language other than English.
- whether the email is palindromic.

Furthermore, we could go beyond properties of single emails and express relations such as whether one email is quoted in another email, or whether two emails have one or more passages in common.

7.2 Feature transformations

Feature transformations aim at improving the utility of a feature by removing, changing or adding information. The best-known feature transformations are those that turn a feature of one type into another of the next type down this list. But there are also transformations that change the scale of quantitative features, or add a scale (or order) to ordinal, categorical and Boolean features. Table 10.2 introduces the terminology we will be using.

↓ to, from →	*Quantitative*	*Ordinal*	*Categorical*	*Boolean*
Quantitative	**normalisation**	**calibration**	**calibration**	**calibration**
Ordinal	**discretisation**	**ordering**	**ordering**	**ordering**
Categorical	**discretisation**	**unordering**	**grouping**	
Boolean	**thresholding**	**thresholding**	**binarisation**	

Table 10.2. An overview of possible feature transformations

Normalization and calibration adapt the scale of quantitative features, or add a scale to features that don't have one. Ordering adds or adapts the order of feature values without reference to a scale. The other operations abstract away from unnecessary detail, either in a deductive way (unordering, binarisation) or by introducing new information (thresholding, discretisation).

Binarisation transforms a categorical feature into a set of Boolean features, one for each value of the categorical feature. Unordering trivially turns an ordinal feature into a categorical one by discarding the ordering of the feature values. An interesting alternative that we will explore below to add a scale to the feature by means of calibration.

7.2.1 Thresholding and discretisation

Thresholding transforms a quantitative or an ordinal feature into a Boolean feature by finding a feature value to split on. Unsupervised thresholding typically involves calculating some statistics over the data, whereas supervised thresholding requires sorting the data on the feature value and traversing down this ordering to optimize a particular objective function such as information gain.

Discretisation transforms a quantitative feature into an ordinal feature. Each ordinal value is referred to as a bin and corresponds to an interval of the original quantitative feature. There are supervised and unsupervised approaches.

Unsupervised discretisation methods typically require one to decide the number of bins beforehand. A simple method that often works reasonably well is to choose the bins so that each bin has approximately the same number of instances: this is referred to as equal-frequency discretisation. Another unsupervised discretisation method is equal-width discretisation, which chooses the bin boundaries so that each interval has the same width. The interval width can be established by dividing the feature range by the number of bins if the feature has upper and lower limits. An interesting alternative is to treat feature discretisation as a univariate clustering problem. For example, in order to generate K bins we can uniformly sample K initial bin centers and run K-means until convergence.

In supervised discretisation methods, there are top–down or divisive discretisation methods and bottom–up or agglomerative discretisation methods. Divisive methods work by progressively splitting bins, whereas agglomerative methods proceed by initially assigning

each instance to its own bin and successively merging bins. In either case an important role is played by the stopping criterion, which decides whether a further split or merge is worthwhile. A natural generalization of thresholding leads to a top–down recursive partitioning algorithm (Algorithm 10.1). This discretisation algorithm finds the best threshold according to some scoring function Q, and proceeds to recursively split the left and right bins. One scoring function that is often used is information gain.

Algorithm 10.1: RecPart(S, f, Q) – supervised discretisation by means of recursive partitioning.

Input : set of labelled instances S ranked on feature values $f(x)$; scoring function Q.

Output : sequence of thresholds $t_1, \ldots, t_{k-1}$.

1 **if** stopping criterion applies **then return** $\varnothing$;
2 Split S into S_l and S_r using threshold t that optimises Q;
3 T_l = RecPart(S_l, f, Q);
4 T_r = RecPart(S_r, f, Q);
5 **return** $T_l \cup \{t\} \cup T_r$;

Consider the following feature values, which are ordered on increasing value for convenience.

Instance	*Value*	*Class*
e_1	−5.0	⊖
e_2	−3.1	⊕
e_3	−2.7	⊖
e_4	0.0	⊖
e_5	7.0	⊖
e_6	7.1	⊕
e_7	8.5	⊕
e_8	9.0	⊖
e_9	9.0	⊕
e_{10}	13.7	⊖
e_{11}	15.1	⊖
e_{12}	20.1	⊖

This feature gives rise to the following ranking: ⊖⊕⊖⊖⊖⊕⊕[⊖⊕]⊖⊖⊖, where the square brackets indicate a tie between instances e_8 and e_9. Tracing information gain isometrics through each possible split, we see that the best split is ⊖⊕⊖⊖⊖⊕⊕[⊖⊕] | ⊖⊖⊖. Repeating the process once more gives the discretisation ⊖⊕⊖⊖⊖ | ⊕⊕[⊖⊕] | ⊖⊖⊖.

Algorithm 10.2: AggloMerge(S, f, Q) – supervised discretisation by means of agglomerative merging.

Input : set of labelled instances S ranked on feature values $f(x)$; scoring function Q.
Output : sequence of thresholds.

```
initialise bins to data points with the same scores;
merge consecutive pure bins ;                    // optional optimisation
repeat
    evaluate Q on consecutive bin pairs;
    merge the pairs with best Q (unless they invoke the stopping criterion);
until no further merges are possible;
return thresholds between bins;
```

In Agglomerative merging using χ^2, Algorithm 10.2 initializes the bins to ⊖ | ⊕ | ⊖⊖⊖ | ⊕⊕ | [⊖⊕] | ⊖⊖⊖. We illustrate the calculation of the χ^2 statistic for the last two bins. We construct the following contingency table:

	Left bin	*Right bin*	
⊕	**1**	**0**	1
⊖	**1**	**3**	4
	2	3	5

At the basis of the χ^2 statistic lies a comparison of these observed frequencies with expected frequencies obtained from the row and column marginals. For example, the marginals say that the top row contains 20% of the total mass and the left column 40%; so if rows and columns were statistically independent we would expect 8% of the mass – or 0.4 of the five instances – in the top-left cell.

Following a clockwise direction, the expected frequencies for the other cells are 0.6, 2.4 and 1.6. If the observed frequencies are close to the expected ones, this suggests that these two bins are candidates for merging since the split appears to have no bearing on the class distribution. The χ^2 statistic sums the squared differences between the observed and expected frequencies, each term normalized by the expected frequency:

$$\chi^2 = \frac{(1-0.4)^2}{0.4} + \frac{(0-0.6)^2}{0.6} + \frac{(3-2.4)^2}{2.4} + \frac{(1-1.6)^2}{1.6} = 1.88$$

Going left-to-right through the other pairs of consecutive bins, the χ^2 values are 2, 4, 5 and 1.33. This tells us that the fourth and fifth bin are first to be merged, leading to ⊖ | ⊕ | ⊖⊖⊖ | ⊕⊕[⊖⊕] | ⊖⊖⊖. We then recompute the χ^2 values (in fact, only those involving the newly merged bin need to be re-computed), yielding 2, 4, 3.94 and 3.94. We now merge the first two bins, giving the partition ⊖⊕ | ⊖⊖⊖ | ⊕⊕[⊖⊕] | ⊖⊖⊖. This changes the first χ^2 value to 1.88, so we again merge the first two bins, arriving at ⊖⊕⊖⊖⊖ | ⊕⊕[⊖⊕] | ⊖⊖⊖.

7.2.2 Normalization and calibration

Thresholding and discretisation are feature transformations that remove the scale of a quantitative feature. We now turn our attention to adapting the scale of a quantitative feature, or adding a scale to an ordinal or categorical feature. If this is done in an unsupervised fashion it is usually called normalization, whereas calibration refers to supervised approaches taking in the (usually binary) class labels.

Feature normalization is often required to neutralize the effect of different quantitative features being measured on different scales. If the features are approximately normally distributed, we can convert them into z-scores by centering on the mean and dividing by the standard deviation. If we don't want to assume normality we can center on the median and divide by the interquartile range.

Sometimes feature normalization is understood in the stricter sense of expressing the feature on a [0, 1] scale. This can be achieved in various ways. If we know the feature's highest and lowest values h and l, then we can simply apply the linear scaling $f \mapsto (f - l) / (h - l)$. We sometimes have to guess the value of h or l, and truncate any value outside $[l, h]$.

For example, if the feature measures age in years, we may take $l = 0$ and $h = 100$, and truncate any $f > h$ to 1. If we can assume a particular distribution for the feature, then we can work out a transformation such that almost all feature values fall in a certain range. For instance, we know that more than 99% of the probability mass of a normal distribution falls within $\pm 3\sigma$ of the mean, where σ is the standard deviation, so the linear scaling $f \mapsto (f - \mu) / 6\sigma + 1/2$ virtually removes the need for truncation.

Feature calibration is understood as a supervised feature transformation adding a meaningful scale carrying class information to arbitrary features. This has a number of important advantages. For instance, it allows models that require scale, such as linear classifiers, to handle categorical and ordinal features. It also allows the learning algorithm to choose whether to treat a feature as categorical, ordinal or quantitative.

We will assume a binary classification context, and so a natural choice for the calibrated feature's scale is the posterior probability of the positive class, conditioned on the feature's value. This has the additional advantage that models that are based on such probabilities, such as naive Bayes, do not require any additional training once the features are calibrated, as we shall see. The problem of feature calibration can thus be stated as follows: given a feature F: $X \mapsto F$, construct a calibrated feature F^c: $X \mapsto [0, 1]$ such that $F^c(x)$ estimates the probability $F^c(x) = P(\oplus \mid v)$, where $v = F(x)$ is the value of the original feature for x.

For categorical features this is as straightforward as collecting relative frequencies from a training set. Suppose we want to predict whether or not someone has diabetes from categorical features including whether the person is obese or not, whether he or she smokes,

and so on. We collect some statistics which tell us that 1 in every 18 obese persons has diabetes while among non-obese people this is 1 in 55. If $F(x) = 1$ for person x who is obese and $F(y) = 0$ for person y who isn't, then the calibrated feature values are $F^c(x) = 1 / 18 = 0.055$ and $F^c(y) = 1 / 55 = 0.018$.

7.2.3 Incomplete features

Missing feature values at training time are trickier to handle. First of all, the very fact that a feature value is missing may be correlated with the target variable. For example, the range of medical tests carried out on a patient is likely to depend on their medical history. For such features it may be best to have a designated 'missing' value so that, for instance, a tree model can split on it. However, this would not work for, say, a linear model. In such cases we can complete the feature by 'filling in' the missing values, a process known as imputation.

For instance, in a classification problem we can calculate the per-class means, medians or modes over the observed values of the feature and use this to impute the missing values. A somewhat more sophisticated method takes feature correlation into account by building a predictive model for each incomplete feature and uses that model to 'predict' the missing value. It is also possible to invoke the Expectation-Maximisation algorithm, which goes roughly as follows: assuming a multivariate model over all features, use the observed values for maximum-likelihood estimation of the model parameters, then derive expectations for the unobserved feature values and iterate.

7.3 Feature construction and selection

New features are constructed from several original features. Feature extraction techniques are to analyze the similarities between pieces of text. In text classification applications there is a feature for every word in the vocabulary. This means that sentences such as 'they write about machine learning' and 'they are learning to write about a machine' will be virtually indistinguishable, even though the former is about machine learning and the latter is not. It may therefore sometimes be necessary to include phrases consisting of multiple words in the dictionary and treat them as single features. In the information retrieval literature, a multi-word phrase is referred to as an n-gram (unigram, bigram, trigram and so on).

A new feature can be constructed from two Boolean or categorical features by forming their Cartesian product. For example, if we have one feature Shape with values Circle, Triangle and Square, and another feature Color with values Red, Green and Blue, then their Cartesian product would be the feature (Shape, Colour) with values (Circle, Red), (Circle, Green), (Circle, Blue), (Triangle, Red), and so on.

There are many other ways of combining features. For instance, we can take arithmetic or polynomial combinations of quantitative features. One attractive possibility is to first apply

concept learning or subgroup discovery, and then use these concepts or subgroups as new Boolean features. Once we have constructed new features it is often a good idea to select a suitable subset of them prior to learning. Not only will this speed up learning as fewer candidate features need to be considered, it also helps to guard against overfitting.

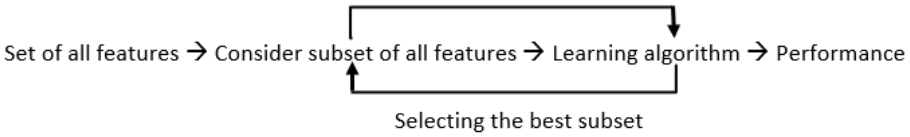
Feature selection is a process that chooses a subset of features from the original features so that the feature space is optimally reduced according to a certain criterion. Feature selection is a critical step in the feature construction process. There are two main approaches to feature selection. They are Filter methods and wrapper methods.

Filter methods are generally used while doing the pre-processing step. These methods select features from the dataset irrespective of the use of any machine learning algorithm. In terms of computation, they are very fast and inexpensive and are very good for removing duplicated, correlated, redundant features but these methods do not remove multicollinearity. Selection of features is evaluated individually which can sometimes help when features are in isolation (don't have a dependency on other features) but will lag when a combination of features can lead to increase in the overall performance of the model.

Set of all features → Selecting the best subset → Learning algorithm → Performance

An interesting variation is the Relief method, measures the quality of attributes by randomly sampling an instance from the dataset and updating each feature and distinguishing between instances that are near to each other based on the difference between the selected instance and two nearest instances of same and opposite classes.

Wrapper methods, also referred to as greedy algorithms, train the algorithm by using a subset of features in an iterative manner. Based on the conclusions made from training prior to the model, addition and removal of features takes place. Stopping criteria for selecting the best subset are usually pre-defined by the person training the model such as when the performance of the model decreases or a specific number of features has been achieved. The main advantage of wrapper methods over the filter methods is that they provide an optimal set of features for training the model, thus resulting in better accuracy than the filter methods but are computationally more expensive.



Forward selection method is an iterative approach where we initially start with an empty set of features and keep adding a feature which best improves our model after each iteration. The stopping criterion is till the addition of a new variable does not improve the performance of the model.

Backward elimination method is also an iterative approach where we initially start with all features and after each iteration, we remove the least significant feature. The stopping criterion is till no improvement in the performance of the model is observed after the feature is removed.

Apart from the methods discussed above, there are many other methods of feature selection. Using hybrid methods for feature selection can offer a selection of best advantages from other methods, leading to reduction in the disadvantages of the algorithms. These models can provide greater accuracy and performance when compared to other methods. Dimensionality reduction techniques such as Principal Component Analysis (PCA), Heuristic Search Algorithms, etc. don't work in the way of feature selection techniques but can help us to reduce the number of features.

Feature selection is a wide, complicated field and a lot of studies have already been made to figure out the best methods. It depends on the machine learning engineer to combine and innovative the approaches, test them and then see what works best for the given problem.

Chapter-8

Model Ensembles

Instructor Name: B N V Narasimha Raju

8.1 Ensemble learning

Ensemble learning helps improve machine learning results by combining several models. This approach allows the production of better predictive performance compared to a single model. Basic idea is to learn a set of classifiers (experts) and to allow them to vote. The advantage of ensemble learning is improving predictive accuracy and the disadvantage is that it is difficult to understand an ensemble of classifiers.

In learning models, noise, variance, and bias are the major sources of error. The ensemble methods in machine learning help minimize these error-causing factors, thereby ensuring the accuracy and stability of machine learning (ML) algorithms.

For example, assume that you are developing an app for the travel industry. It is obvious that before making the app public, you will want to get crucial feedback on bugs and potential loopholes that are affecting the user experience. What are your available options for obtaining critical feedback? 1) Soliciting opinions from your parents, spouse, or close friends. 2) Asking your co-workers who travel regularly and then evaluating their response. 3) Rolling out your travel and tourism app in beta to gather feedback from non-biased audiences and the travel community.

Think for a moment about what you are doing. You are taking into account different views and ideas from a wide range of people to fix issues that are limiting the user experience. The ensemble neural network and ensemble algorithm do precisely the same thing.

8.1.1 Simple Ensemble Methods

In statistical terminology, "mode" is the number or value that most often appears in a dataset of numbers or values. In this ensemble technique, machine learning professionals use a number of models for making predictions about each data point. The predictions made by different models are taken as separate votes. Subsequently, the prediction made by most models is treated as the ultimate prediction.

In the mean/average ensemble technique, data analysts take the average predictions made by all models into account when making the ultimate prediction.

In the weighted average ensemble method, data scientists assign different weights to all the models in order to make a prediction, where the assigned weight defines the relevance of each model.

8.1.2 Advanced Ensemble Methods

In Bagging (Bootstrap Aggregating), the primary goal is to minimize variance errors in decision trees. The objective here is to randomly create samples of training datasets with replacement (subsets of the training data). The subsets are then used for training decision trees or models. Consequently, there is a combination of multiple models, which reduces variance, as the average prediction generated from different models is much more reliable and robust than a single model or a decision tree.

Boosting is an iterative ensemble technique, "boosting," adjusts an observation's weight based on its last classification. In case observation is incorrectly classified, "boosting" increases the observation's weight, and vice versa. Boosting algorithms reduce bias errors and produce superior predictive models.

8.2 Bagging

Bootstrap Aggregating, also known as bagging, is a machine learning ensemble meta-algorithm designed to improve the stability and accuracy of machine learning algorithms used in statistical classification and regression. It decreases the variance and helps to avoid overfitting. It is usually applied to decision tree methods. Bagging is a special case of the model averaging approach.

Suppose a set D of d tuples, at each iteration i, a training set D_i of d tuples is selected via row sampling with a replacement method (i.e., there can be repetitive elements from different d tuples) from D (i.e., bootstrap). Then a classifier model M_i is learned for each training set D < i. Each classifier M_i returns its class prediction. The bagged classifier M* counts the votes and assigns the class with the most votes to X (unknown sample). Bagging is shown in figure 8.1.

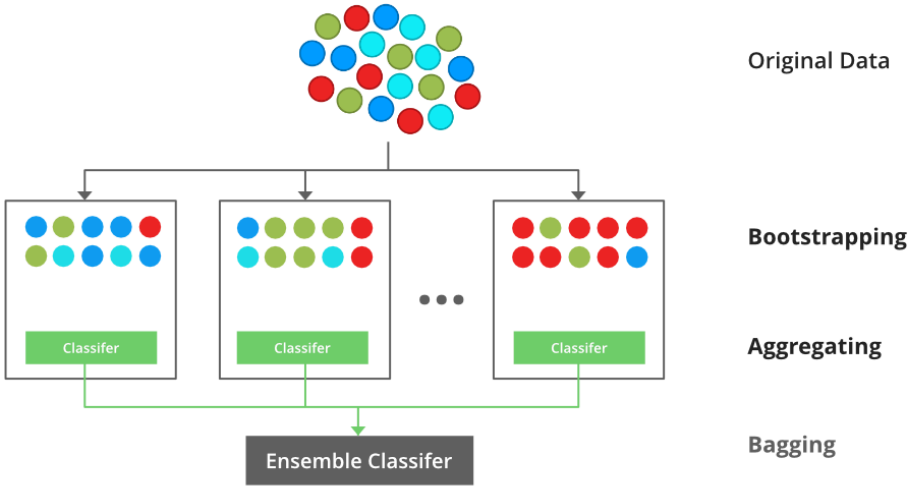


Figure 8.1: Bagging

The steps for Bagging are

- Multiple subsets are created from the original data set with equal tuples, selecting observations with replacement.
- A base model is created on each of these subsets.
- Each model is learned in parallel with each training set and independent of each other.
- The final predictions are determined by combining the predictions from all the models.

8.3 Random Forest

The Random Forest model uses Bagging, where decision tree models with higher variance are present. It makes random feature selection to grow trees. Several random trees make a Random Forest.

A Random Forest Algorithm is a supervised machine learning algorithm that is extremely popular and is used for Classification and Regression problems in Machine Learning. We know that a forest comprises numerous trees, and the more trees there are, the more it will be robust. Similarly, the greater the number of trees in a Random Forest Algorithm, the higher its accuracy and problem-solving ability.

8.3.1 Classification in random forests

Random Forest is a classifier that contains several decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset. It is based on the concept of ensemble learning which is a process of combining multiple classifiers to solve a complex problem and improve the performance of the model.

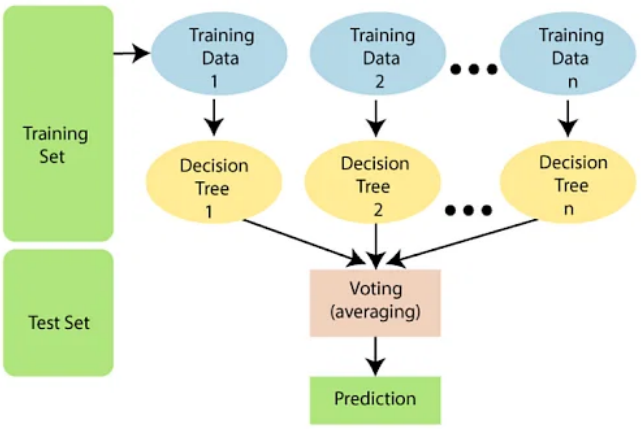


Figure 8.2: Random Forest

The following steps explain the working Random Forest Algorithm:

- Select random samples from a given data or training set.
- This algorithm will construct a decision tree for every training data.
- Voting will take place by averaging the decision tree.
- Finally, select the most voted prediction result as the final prediction result.

Let's take an example of a training dataset consisting of various fruits such as bananas, apples, pineapples, and mangoes as shown in figure 8.3. The random forest classifier divides this dataset into subsets. These subsets are given to every decision tree in the random forest system. Each decision tree produces its specific output. For example, the prediction for trees 1 and 2 is apples.

Another decision tree (n) has predicted banana as the outcome. The random forest classifier collects the majority voting to provide the final prediction. The majority of the decision trees have chosen apples as their prediction. This makes the classifier choose apples as the final prediction.

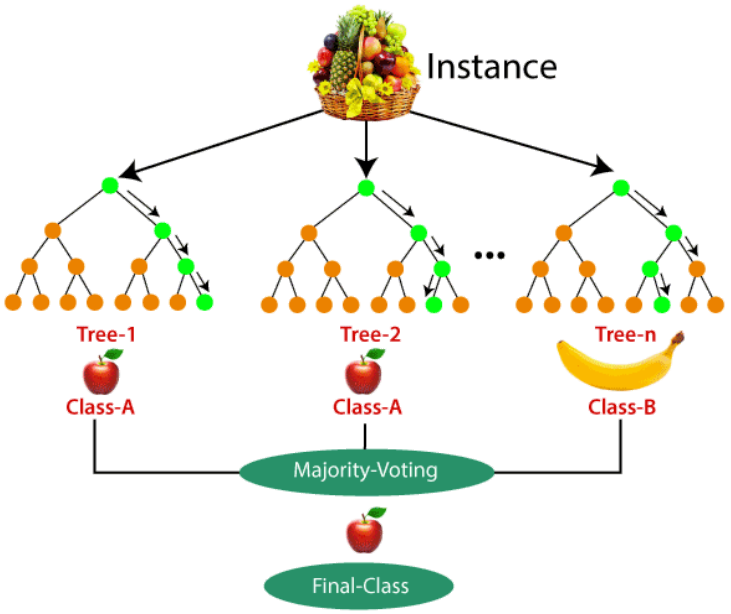


Figure 8.3: Example of random forest training dataset consisting of various fruits

8.3.2 Regression in random forests

Regression is the other task performed by a random forest algorithm. A random forest regression follows the concept of simple regression. Values of dependent (features) and independent variables are passed in the random forest model.

In a random forest regression, each tree produces a specific prediction. The mean prediction of the individual trees is the output of the regression. This is contrary to random forest classification, whose output is determined by the mode of the decision trees class.

Although random forest regression and linear regression follow the same concept, they differ in terms of functions. The function of linear regression is $y = bx + c$, where y is the dependent variable, x is the independent variable, b is the estimation parameter, and c is a constant. The function of a complex random forest regression is like a black box.

8.3.3 Essential Features of Random Forest

- Miscellany: Each tree has a unique attribute, variety and features concerning other trees. Not all trees are the same.
- Immune to the curse of dimensionality: Since a tree is a conceptual idea, it requires no features to be considered. Hence, the feature space is reduced.
- Parallelization: We can fully use the CPU to build random forests since each tree is created autonomously from different data and features.
- Train-Test split: In a Random Forest, we don't have to differentiate the data for train and test because the decision tree never sees 30% of the data.
- Stability: The final result is based on Bagging, meaning the result is based on majority voting or average.

8.3.4 Difference between Decision Tree and Random Forest

Decision Trees	Random Forest
They usually suffer from the problem of overfitting if it's allowed to grow without any control.	Since they are created from subsets of data and the final output is based on average or majority ranking, the problem of overfitting doesn't happen here.
A single decision tree is comparatively faster in computation.	It is slower.
They use a particular set of rules when a data set with features is taken as input.	Random Forest randomly selects observations, builds a decision tree and then the result is obtained based on majority voting. No formulas are required here.

There are a lot of benefits to using Random Forest Algorithm, but one of the main advantages is that it reduces the risk of overfitting and the required training time. Additionally, it offers a high level of accuracy. Random Forest algorithm runs efficiently in large databases and produces highly accurate predictions by estimating missing data.

8.3.5 Applications of random forest

Some of the applications of the random forest may include

- Banking: Random forest is used in banking to predict the creditworthiness of a loan applicant. This helps the lending institution make a good decision on whether to give the customer the loan or not. Banks also use the random forest algorithm to detect fraudsters.
- Health care: Health professionals use random forest systems to diagnose patients. Patients are diagnosed by assessing their previous medical history. Past medical records are reviewed to establish the right dosage for the patients.
- Stock market: Financial analysts use it to identify potential markets for stocks. It also enables them to identify the behavior of stocks.
- E-commerce: Through rain forest algorithms, e-commerce vendors can predict the preference of customers based on past consumption behavior.

8.4 Boosting

Boosting is an ensemble modeling technique that attempts to build a strong classifier from the number of weak classifiers. It is done by building a model by using weak models in series. Firstly, a model is built from the training data. Then the second model is built which tries to correct the errors present in the first model. This procedure is continued and models are

added until either the complete training data set is predicted correctly or the maximum number of models are added.

Boosting improves machine models predictive accuracy and performance by converting multiple weak learners into a single strong learning model. Machine learning models can be weak learners or strong learners.

Weak learners have low prediction accuracy, similar to random guessing. They are prone to overfitting—that is, they can't classify data that varies too much from their original dataset. For example, if you train the model to identify cats as animals with pointed ears, it might fail to recognize a cat whose ears are curled.

Strong learners have higher prediction accuracy. Boosting converts a system of weak learners into a single strong learning system. For example, to identify the cat image, it combines a weak learner that guesses for pointy ears and another learner that guesses for cat-shaped eyes. After analyzing the animal image for pointy ears, the system analyzes it once again for cat-shaped eyes. This improves the system's overall accuracy.

Boosting creates an ensemble model by combining several weak decision trees sequentially. It assigns weights to the output of individual trees. Then it gives incorrect classifications from the first decision tree a higher weight and input to the next tree. After numerous cycles, the boosting method combines these weak rules into a single powerful prediction rule.

Boosting and bagging are the two common ensemble methods that improve prediction accuracy. The main difference between these learning methods is the method of training. In bagging, data scientists improve the accuracy of weak learners by training several of them at once on multiple datasets. In contrast, boosting trains weak learners one after another.

8.4.1 Training in boosting

In boosting, the training method varies depending on the type of boosting process called the boosting algorithm. However, an algorithm takes the following general steps to train the boosting model:

- The boosting algorithm assigns equal weight to each data sample. It feeds the data to the first machine model, called the base algorithm. The base algorithm makes predictions for each data sample.
- The boosting algorithm assesses model predictions and increases the weight of samples with a more significant error. It also assigns a weight based on model performance. A model that outputs excellent predictions will have a high amount of influence over the final decision.
- The algorithm passes the weighted data to the next decision tree.
- The algorithm repeats steps 2 and 3 until instances of training errors are below a certain threshold.

8.4.2 Types of boosting

Boosting algorithms can differ in how they create and aggregate weak learners during the sequential process. Three popular types of boosting methods include

- Adaptive boosting: Adaptive Boosting (AdaBoost) was one of the earliest boosting models developed. It adapts and tries to self-correct in every iteration of the boosting process. AdaBoost initially gives the same weight to each dataset. Then, it automatically adjusts the weights of the data points after every decision tree. It gives more weight to incorrectly classified items to correct them for the next round. It repeats the process until the residual error, or the difference between actual and predicted values, falls below an acceptable threshold. AdaBoost is a suitable type of boosting for classification problems.
- Gradient boosting: Gradient Boosting (GB) is similar to AdaBoost in that it, too, is a sequential training technique. The difference between AdaBoost and GB is that GB does not give incorrectly classified items more weight. Instead, GB software optimizes the loss function by generating base learners sequentially so that the present base learner is always more effective than the previous one. This method attempts to generate accurate results initially instead of correcting errors throughout the process, like AdaBoost. For this reason, GB software can lead to more accurate results. Gradient Boosting can help with both classification and regression-based problems.
- Extreme gradient boosting: Extreme Gradient Boosting (XGBoost) improves gradient boosting for computational speed and scale in several ways. XGBoost uses multiple cores on the CPU so that learning can occur in parallel during training. It is a boosting algorithm that can handle extensive datasets, making it attractive for big data applications. The key features of XGBoost are parallelization, distributed computing, cache optimization, and out-of-core processing.

8.4.3 Benefits of boosting

- Ease of implementation: Boosting has easy-to-understand and easy-to-interpret algorithms that learn from their mistakes. These algorithms don't require any data preprocessing, and they have built-in routines to handle missing data. In addition, most languages have built-in libraries to implement boosting algorithms with many parameters that can fine-tune performance.
- Reduction of bias: Bias is the presence of uncertainty or inaccuracy in machine learning results. Boosting algorithms combine multiple weak learners in a sequential method, which iteratively improves observations. This approach helps to reduce high bias that is common in machine learning models.
- Computational efficiency: Boosting algorithms prioritize features that increase predictive accuracy during training. They can help to reduce data attributes and handle large datasets efficiently.

8.4.4 Challenges of boosting

- Vulnerability to outlier data: Boosting models are vulnerable to outliers or data values that are different from the rest of the dataset. Because each model attempts to correct the faults of its predecessor, outliers can skew results significantly.
- Real-time implementation: You might also find it challenging to use boosting for real-time implementation because the algorithm is more complex than other processes. Boosting methods have high adaptability, so you can use a wide variety of model parameters that immediately affect the model's performance.

8.5 AdaBoost

AdaBoost, also called Adaptive Boosting, is a technique in Machine Learning used as an Ensemble Method. AdaBoost was the first really successful boosting algorithm developed for the purpose of binary classification. AdaBoost is a very popular boosting technique that combines multiple "weak classifiers" into a single "strong classifier". The most common estimator used with AdaBoost is decision trees with one level which means Decision trees with only 1 split. These trees are also called Decision Stumps as shown in figure 8.4.

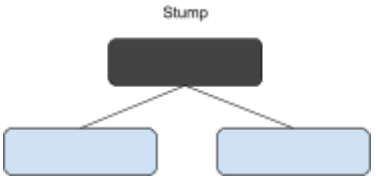


Figure 8.4: Decision Stumps

This algorithm builds a model and gives equal weights to all the data points. It then assigns higher weights to points that are wrongly classified. Now all the points with higher weights are given more importance in the next model. It will keep training models until and unless a lower error is received as shown in figure 8.5.

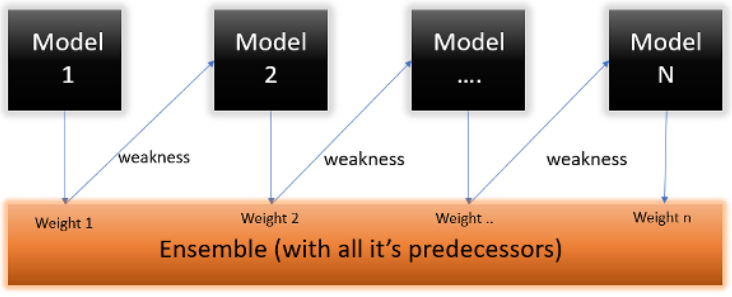


Figure 8.5: Working of AdaBoost algorithm

Let's take an example to understand this, suppose you built a decision tree algorithm on the Titanic dataset, and from there, you get an accuracy of 80%. After this, you apply a different

algorithm and check the accuracy, and it comes out to be 75% for KNN and 70% for Linear Regression.

Accuracy will differ when we build a different model on the same dataset. To make the final prediction the combination of all these algorithms is used to get more accurate results by taking the average of the results from these models. The prediction power can be increased in this way.

8.5.1 Working of the AdaBoost Algorithm

Step 1 – The dataset is shown in Table 8.1. Since the target column is binary, it is a classification problem. First of all, these data points will be assigned some weights. Initially, all the weights will be equal

Row No.	Gender	Age	Income	Illness	Sample Weights
1	Male	41	40000	Yes	1/5
2	Male	54	30000	No	1/5
3	Female	42	25000	No	1/5
4	Female	40	60000	Yes	1/5
5	Male	46	50000	Yes	1/5

Table 8.1: Dataset

The formula to calculate the sample weights is $w(x_i, y_i) = \frac{1}{N} i = 1, 2,.., n$. Where *N* is the total number of data points. Here since we have 5 data points, the sample weights assigned will be 1/5.

Step 2 – See how well "Gender" classifies the samples and see how the variables (Age, Income) classify the samples. Create a decision stump for each of the features and then calculate the Gini Index of each tree. The tree with the lowest Gini Index will be our first stump. Here in our dataset, let's say Gender has the lowest gini index, so it will be our first stump.

Step 3 – Calculate the "Amount of Say" or "Importance" or "Influence" for this classifier in classifying the data points. The total error is nothing but the summation of all the sample weights of misclassified data points. Here in our dataset, let's assume there is 1 wrong output, so our total error will be 1/5, and the alpha (performance of the stump) will be:

α (Performance of the stump) = $\frac{1}{2} log_e(\frac{1-Total\ Error}{Total\ Error})$

$\alpha = \frac{1}{2} log_e(\frac{1-1/5}{1/5})$

$$\alpha = \frac{1}{2} log_e(\frac{0.8}{0.2})$$

$$\alpha = \frac{1}{2} log_e(4) = \frac{1}{2} * (1.38)$$

$$\alpha = 0.69$$

Total error will always be between 0 and 1. 0 Indicates perfect stump, and 1 indicates horrible stump.

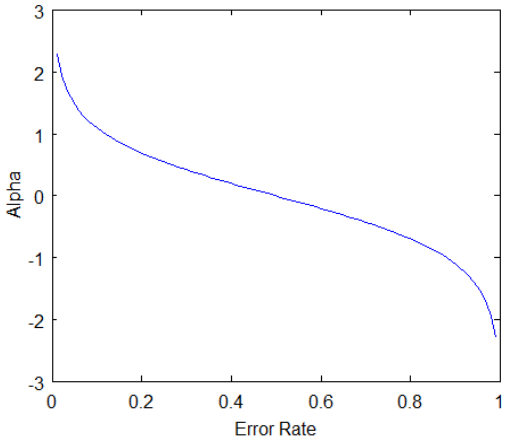


Figure 8.6: Total Error Rate

From the figure 8.6, see that when there is no misclassification, then no error (Total Error = 0), so the "amount of say (alpha)" will be a large number. When the classifier predicts half right and half wrong, then the Total Error = 0.5, and the importance (amount of say) of the classifier will be 0. If all the samples have been incorrectly classified, then the error will be very high (approx. to 1), and hence our alpha value will be a negative integer.

Step 4 – You must be wondering why it is necessary to calculate the TE and performance of a stump. Well, the answer is very simple, we need to update the weights because if the same weights are applied to the next model, then the output received will be the same as what was received in the first model.

The wrong predictions will be given more weight, whereas the correct predictions weights will be decreased. Now when we build our next model after updating the weights, more preference will be given to the points with higher weights.

After finding the importance of the classifier and total error, we need to finally update the weights, and for this, we use the following formula:

$$New\ sample\ weight\ =\ old\ weight\ *\ e^{\pm Amount\ of\ say(\alpha)}$$

The amount of, say (alpha) will be negative when the sample is correctly classified. The amount of, say (alpha) will be positive when the sample is miss-classified. There are four correctly classified samples and 1 wrong. Here, the sample weight of that datapoint is 1/5, and

the amount of say/performance of the stump of Gender is 0.69. New weights for correctly classified samples are:

$$New\ sample\ weight\ =\ 1/5\ *\ exp(-\ 0.69)$$

$$New\ sample\ weight\ =\ 0.2\ *\ 0.502\ =\ 0.1004$$

For wrongly classified samples, the updated weights will be

$$New\ sample\ weight\ =\ 1/5\ *\ \ exp(0.69)$$

$$New\ sample\ weight\ =\ 0.2\ *\ 1.994\ =\ 0.3988$$

See the sign of alpha when I am putting the values, the alpha is negative when the data point is correctly classified, and this decreases the sample weight from 0.2 to 0.1004. It is positive when there is misclassification, and this will increase the sample weight from 0.2 to 0.3988

Row No.	Gender	Age	Income	Illness	Sample Weights	New Sample Weights
1	Male	41	40000	Yes	1/5	0.1004
2	Male	54	30000	No	1/5	0.1004
3	Female	42	25000	No	1/5	0.1004
4	Female	40	60000	Yes	1/5	0.3988
5	Male	46	50000	Yes	1/5	0.1004

The total sum of the sample weights must be equal to 1, but here the sum of all the new sample weights is 0.8004. To bring this sum equal to 1, normalize these weights by dividing all the weights by the total sum of updated weights, which is 0.8004. So, after normalizing the sample weights, now the sum is equal to 1.

Row No.	Gender	Age	Income	Illness	Sample Weights	New Sample Weights
1	Male	41	40000	Yes	1/5	0.1004/0.8004 = 0.1254
2	Male	54	30000	No	1/5	0.1004/0.8004 = 0.1254
3	Female	42	25000	No	1/5	0.1004/0.8004 = 0.1254
4	Female	40	60000	Yes	1/5	0.3988/0.8004 = 0.4982
5	Male	46	50000	Yes	1/5	0.1004/0.8004 = 0.1254

Step 5 – Now, make a new dataset to see if the errors decreased or not. For this, remove the "sample weights" and "new sample weights" columns and then, based on the "new sample weights," divide our data points into buckets.

Row No.	Gender	Age	Income	Illness	New Sample Weights	Buckets

1	Male	41	40000	Yes	0.1004/0.8004 = 0.1254	0 to 0.1254
2	Male	54	30000	No	0.1004/0.8004 = 0.1254	0.1254 to 0.2508
3	Female	42	25000	No	0.1004/0.8004 = 0.1254	0.2508 to 0.3762
4	Female	40	60000	Yes	0.3988/0.8004 = 0.4982	0.3762 to 0.8744
5	Male	46	50000	Yes	0.1004/0.8004 = 0.1254	0.8744 to 0.9998

Step 6 – Now, what the algorithm does is selects random numbers from 0-1. Since incorrectly classified records have higher sample weights, the probability of selecting those records is very high. Suppose the 5 random numbers our algorithm takes are 0.38, 0.19, 0.26, 0.40, 0.55. Now see where these random numbers fall in the bucket, and according to it, make our new dataset shown below.

Row No.	Gender	Age	Income	Illness
1	Female	40	60000	Yes
2	Male	54	30000	No
3	Female	42	25000	No
4	Female	40	60000	Yes
5	Female	40	60000	Yes

This is the new dataset, and see the data point, which was wrongly classified, has been selected 3 times because it has a higher weight.

Step 7 – Now this is the new dataset, and repeat all the above steps.

Iterate through these steps until and unless a low training error is achieved. Suppose, with respect to our dataset, we have constructed 3 decision trees (DT1, DT2, DT3) in a sequential manner. If we send our test data now, it will pass through all the decision trees, and finally, we will see which class has the majority, and based on that, we will do predictions for our test dataset.

8.6 Gradient Boosting

The main idea behind this algorithm is to build models sequentially and these subsequent models try to reduce the errors of the previous model. Error is reduced by building a new model on the errors or residuals of the previous model.

When the target column is continuous, Gradient Boosting Regressor is used whereas when it is a classification problem, Gradient Boosting Classifier is used. The only difference between

the two is the “Loss function”. The objective here is to minimize this loss function by adding weak learners using gradient descent. Since it is based on loss function hence for regression problems there are different loss functions like Mean squared error (MSE) and for classification there are different loss functions like log-likelihood.

8.6.1 Gradient Boosting Algorithm

Let’s understand the intuition behind Gradient boosting with the help of an example. Here our target column is continuous hence we will use Gradient Boosting Regressor.

Following is a sample from a random dataset where we have to predict the car price based on various features. The target column is price and other features are independent features.

Row No.	Cylinder Number	Car Height	Engine Location	Price
1	Four	48.4	Front	12000
2	Six	48.4	Back	16500
3	Five	52.4	Back	15500
4	Four	54.3	Front	14000

Step 1 - The first step in gradient boosting is to build a base model to predict the observations in the training dataset. For simplicity, take the average of the target column and assume that to be the predicted value.

Row No.	Cylinder Number	Car Height	Engine Location	Price	Prediction 1
1	Four	48.4	Front	12000	14500
2	Six	48.4	Back	16500	14500
3	Five	52.4	Back	15500	14500
4	Four	54.3	Front	14000	14500

Taking the average of the target column has math involved behind this. Mathematically the first step can be written as:

$$F_0(x) = arg_\gamma min \sum_{i=1}^{n} L(y_i, \gamma)$$

Here *L* is our loss function, *Gamma* is our predicted value and *argmin* means we have to find a predicted value/gamma for which the loss function is minimum. Since the target column is continuous our loss function will be:

$$L = \frac{1}{n} \sum_{i=0}^{n} (y_i - \gamma_i)^2$$

Here y_i is the observed value and γ_i is the predicted value. Now we need to find a minimum value of gamma such that this loss function is minimum. To find minima and maxima simply differentiate this loss function and then put it equal to 0.

$$\frac{dL}{d\gamma} = \frac{2}{2}\left(\sum_{i=0}^{n}(y_i - \gamma_i)\right) = -\sum_{i=0}^{n}(y_i - \gamma_i)$$

Remember that y_i is our observed value and γ_i is our predicted value, by plugging the values in the above formula we get:

Now $\frac{dL}{d\gamma}$ = 0 and taking (-) common

⇒ -[12000 - γ + 16500 - γ + 15500 - γ + 14400 - γ] = 0

⇒ [58000 - 4γ] = 0

⇒ 58000 = 4γ

⇒ γ = 58000/4 =14500

It is an average of the observed car price and it is assumed to be your first prediction. Hence for γ = 14500, the loss function will be minimum so this value will become our prediction for the base model.

Step 2 - The next step is to calculate the pseudo residuals which are (observed value – predicted value)

Row No.	Cylinder Number	Car Height	Engine Location	Price	Prediction 1	Residual 1
1	Four	48.4	Front	12000	14500	-2500
2	Six	48.4	Back	16500	14500	2000
3	Five	52.4	Back	15500	14500	1000
4	Four	54.3	Front	14000	14500	-500

We are just taking the derivative of loss function w.r.t the predicted value

$$\frac{dL}{d\gamma} = -(y_i - \gamma_i) = -(Observed - Predicted)$$

The derivative of the loss function is multiplied by a negative sign, so now we get $(observed - Predicted)$. The predicted value here is the prediction made by the previous

model. In our example the prediction made by the previous model (initial base model prediction) is 14500, to calculate the residuals our formula becomes $(observed - 14500)$

In the next step, build a model on these pseudo residuals and make predictions. This is to minimize the residuals and minimizing the residuals will eventually improve our model accuracy and prediction power. So, using the Residual as target and the original feature Cylinder number, Cylinder height, and Engine location can generate new predictions.

Note that the predictions, in this case, will be the error values, not the predicted car price values since our target column is an error now. Let's say $h_m(x)$ is our decision tree made on these residuals.

Step 3 - In this step find the output values for each leaf of our decision tree. That means there might be a case where 1 leaf gets more than 1 residual, hence we need to find the final output of all the leaves. To find the output simply take the average of all the numbers in a leaf, it doesn't matter if there is only 1 number or more than 1. Mathematically this step can be represented as:

$$\gamma_m = arg_\gamma min \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i))$$

Here $h_m(x_i)$ is the decision tree made on residuals and m is the number of decision trees. When m = 1 means talking about the 1st decision tree and when it is "M" we are talking about the last decision tree.

The output value for the leaf is the value of γ that minimizes the Loss function. The left-hand side "γ" is the output value of a particular leaf. On the right-hand side is similar to step 1 but here the difference is that we are taking previous predictions whereas earlier there was no previous prediction. Let's understand this even better with the help of an example. Suppose our regression tree is shown in figure 8.7

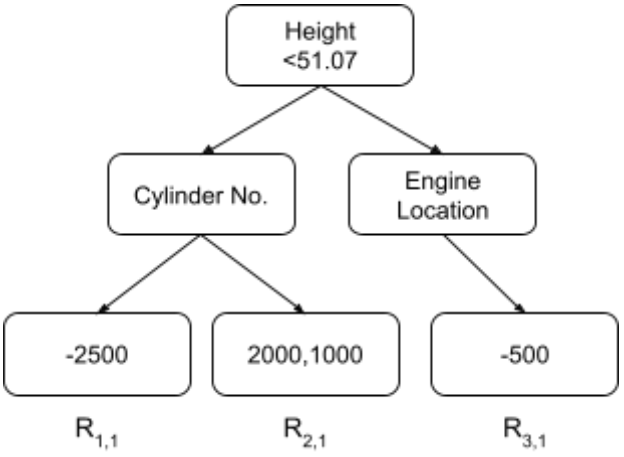


Figure 8.7: Regression Tree

We see 1st residual goes in $R_{1,1}$,2nd and 3rd residuals go in $R_{2,1}$ and 4th residual goes in $R_{3,1}$. Let's calculate the output for the first leaf that is $R_{1,1}$.

$$\gamma_{1,1} = argmin\frac{1}{2}\Big(12000-\big(14500+\gamma\big)\Big)^2$$

$$\gamma_{1,1} = argmin\frac{1}{2}\Big(-2500-\gamma\Big)^2$$

Now find the value for γ for which this function is minimum. So find the derivative of this equation w.r.t γand put it equal to 0.

$$\frac{d}{d\gamma}\frac{1}{2}\Big(-2500-\gamma\Big)^2 = 0$$

$$-2500 -\gamma = 0$$

$$\gamma = -2500$$

Hence the leaf $R_{1,1}$ has an output value of -2500. Now let's solve for the $R_{2,1}$

$$\gamma_{2,1} = \text{argmin}\left[\frac{1}{2}\Big(16500-\big(14500+\gamma\big)\Big)^2+\frac{1}{2}\Big(15500-\big(14500+\gamma\big)\Big)^2\right]$$

$$\gamma_{2,1} = \text{argmin}[\frac{1}{2}\big(2000-\gamma\big)^2+\frac{1}{2}(1000-\gamma)^2]$$

Let's take the derivative to get the minimum value of gamma for which this function is minimum:

$$\frac{d}{d\gamma}\left[\frac{1}{2}\big(2000-\gamma\big)^2+\frac{1}{2}\Big(1000-\gamma\big)^2\right] = 0$$

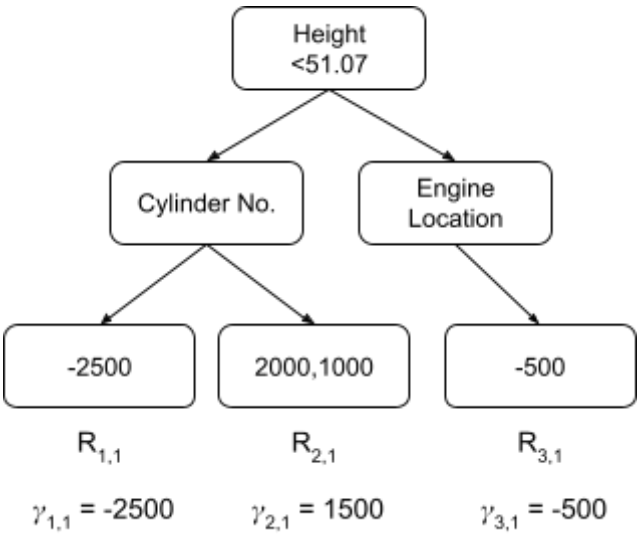
$$2000-\gamma+1000-\gamma=0$$

$$3000-2\gamma=0$$

$$\frac{3000}{2}=\gamma$$

$$\gamma=1500$$

End up with the average of the residuals in the leaf $R_{2,1}$. Hence if any leaf with more than 1 residual, simply find the average of that leaf and that will be our final output. Now after calculating the output of all the leaves



Step 4 - This is finally the last step is to update the predictions of the previous model. It can be updated as

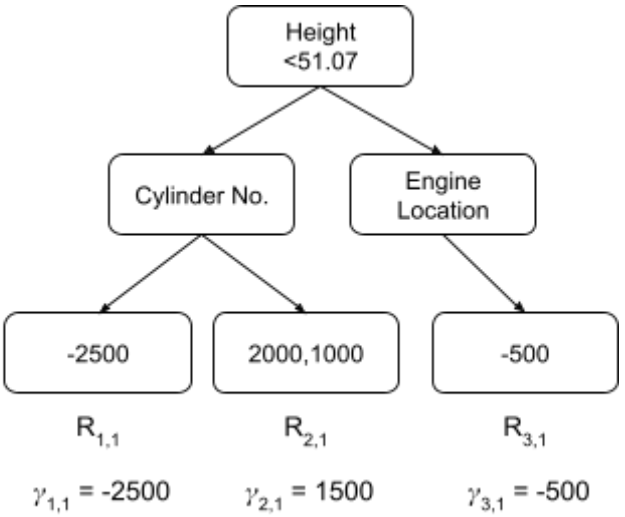
$$F_m(x) = F_{m-1}(x) + v_m h_m(x)$$

where *m* is the number of decision trees made. Since we have just started building our model so our *m* = 1. Now to make a new decision tree our new predictions will be:

New Predicition	=	Previous Predicition	+	Learning Rate	*	The Tree made on Residuals

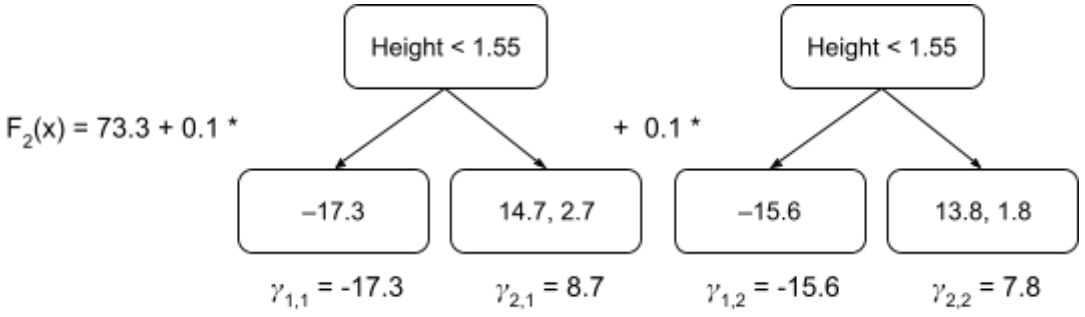
Here $F_{m-1}(x)$ is the prediction of the base model (previous prediction) since $F_{1\text{-}1=0}$, F_0 is our base model hence the previous prediction is 14500. *nu* is the learning rate that is usually selected between 0 - 1. It reduces the effect each tree has on the final prediction, and this improves accuracy in the long run. Let's take *nu* = 0.1 in this example. $H_m(x)$ is the recent decision tree made on the residuals. Let's calculate the new prediction now:

New Prediction $F_1(x)$ = 14500 + 0.1 *



Suppose we want to find a prediction of our first data point which has a car height of 48.8. This data point will go through this decision tree and the output it gets will be multiplied with the learning rate and then added to the previous prediction.

Now let's say m = 2 means 2 decision trees are built and now to have new predictions. This time add the previous prediction that is $F_1(x)$ to the new decision tree made on residuals. Iterate through these steps again and again till the loss is negligible. Considering the hypothetical example here just to make you understand how this predicts for a new dataset:



If a new data point says height = 1.40 comes, it'll go through all the trees and then will give the prediction. Here there are only 2 trees hence the datapoint will go through these 2 trees and the final output will be $F_2(x)$.

Gradient Boosting Classifier

A gradient boosting classifier is used when the target column is binary. All the steps explained in the Gradient boosting regressor are used here, the only difference is to change the loss function. Earlier Mean squared error is used when the target column was continuous but this time, use log-likelihood as our loss function.

Let's see how this loss function works. The loss function for the classification problem is given below:

$$L = -\sum_{i=1}^{n} y_i \log(p) + (1-p)\log(1-p)$$

Our first step in the gradient boosting algorithm was to initialize the model with some constant value and use the average of the target column but here use log(odds) to get that constant value. When the loss function is differentiated, it will get a function of log(odds) and then find a value of log(odds) for which the loss function is minimum. Let's first transform this loss function so that it is a function of log(odds)

$$L = -\left[\sum_{i=1}^{n} y_i \log(p) + \left(1-y_i\right) log(1-p)\right]$$

$$L = -y * log(odds) + log\left(1 + e^{log(odds)}\right)$$

Now this is our loss function and minimize it, for this take the derivative of this w.r.t to log(odds) and then put it equal to 0.

$$\frac{dL}{d[log(odds)]} = -y + \frac{e^{log(odds)}}{1 + e^{log(odds)}}$$

$$\text{We know } \frac{e^{log(odds)}}{1 + e^{log(odds)}} = p, \textit{ hence we can substitue } p$$

$$\frac{dL}{d[log(odds)]} = -y + p$$

Here *y* are the observed values. The transformation of the loss function into the function of log(odds) is due to the fact that sometimes it is easy to use the function of log(odds), and sometimes it's easy to use the function of predicted probability "*p*".

Hence the minimum value of this loss function will be our first prediction (base model prediction). Now in the Gradient boosting regressor our next step was to calculate the pseudo residuals and multiplied the derivative of the loss function with -1. Do the same but now the loss function is different, and dealing with the probability of an outcome now.

$$\frac{dL}{d[log(odds)]} = -y + p$$

$$\frac{dL}{d[log(odds)]} = -(-y + p) = (y - p) = (\text{observed} - \text{predicted})$$

After finding the residuals, build a decision tree with all independent variables and target variables as "Residuals".

Now for the first decision tree, find the final output of the leaves because there might be a case where a leaf gets more than 1 residual, so calculate the final output value. To calculate the output of a leaf:

$$\gamma = \frac{\sum_{i=1}^{n} \text{Residual}_i}{\sum_{i=1}^{n} [\text{Previous probability}_i \times (1 - \text{Previous probability}_i)]}$$

Finally, get new predictions by adding our base model with the new tree we made on residuals.

XGBoost

XGBoost is an optimized distributed gradient boosting library designed for efficient and scalable training of machine learning models. It is an ensemble learning method that combines the predictions of multiple weak models to produce a stronger prediction. XGBoost stands for “Extreme Gradient Boosting” and it has become one of the most popular and widely used machine learning algorithms due to its ability to handle large datasets and its ability to achieve state-of-the-art performance in many machine learning tasks such as classification and regression.

One of the key features of XGBoost is its efficient handling of missing values, which allows it to handle real-world data with missing values without requiring significant pre-processing. Additionally, XGBoost has built-in support for parallel processing, making it possible to train models on large datasets in a reasonable amount of time.

XGBoost is one of the well-known gradient boosting techniques(ensemble) having enhanced performance and speed in tree-based (sequential decision trees) machine learning algorithms. It is the most common algorithm used for applied machine learning in competitions and has gained popularity through winning solutions in structured and tabular data. It is open-source software. To understand XGboost first, a clear understanding of decision trees and ensemble learning algorithms is needed.

XGBoost falls under the category of Boosting techniques in Ensemble Learning. Ensemble learning consists of a collection of predictors which are multiple models to provide better prediction accuracy. It is also highly customizable and allows for fine-tuning of various model parameters to optimize performance. In this algorithm, decision trees are created in sequential form. Weights play an important role in XGBoost. Weights are assigned to all the independent variables which are then fed into the decision tree which predicts results. The weight of variables predicted wrong by the tree is increased and these variables are then fed to the second decision tree. These individual classifiers/predictors then ensemble to give a strong and more precise model. It can work on regression, classification, ranking, and user-defined prediction problems.

Unlike other boosting algorithms where weights of misclassified branches are increased, in Gradient Boosted algorithms the loss function is optimized. XGBoost is an advanced implementation of gradient boosting along with some regularization factors.

Features of XGBoost

- Can be run on both single and distributed systems (Hadoop, Spark).
- XGBoost is used in supervised learning (regression and classification problems).
- Supports parallel processing.

- Cache optimization.
- Efficient memory management for large datasets exceeding RAM.
- Has a variety of regularizations which helps in reducing overfitting.
- Auto tree pruning – Decision trees will not grow further after certain limits internally.
- Can handle missing values.
- Has inbuilt Cross-Validation.
- Take care of outliers to some extent.

XGBoost Algorithm

Let's look at how XGboost works with an example. Here I'll try to predict a child's IQ based on age. For any basic assumption in such statistical data, we can take the average IQ and find how much variance(loss) is present. Residuals are the losses incurred will be calculated after each model predicts.

CHILD's AGE	CHILD's IQ	RESIDUALS
10	20	-10
15	34	4
16	38	8

So the average of 20, 34, and 38 is 30.67, for simplicity let's take it as 30. If we plot a graph keeping y-axis as IQ and x-axis as Age and then we can see the variance in points from the average mark.

At first, our base model (M0) will give a prediction of 30. As from the graph, we know this model suffers a loss which will have some optimisation in the next model (M1). Model M1 will have input as age (independent features) and target as the loss suffered (variances) in M0. Until now it is the same as the gradient boosting technique.

For XGboost some new terms are introduced, ƛ -> regularization parameter, ɣ -> for auto tree pruning, *eta* -> how much model will converge.

Now calculate the *similarity score* (*S.S*) = $(S.R^2) / (N + ƛ)$. Here, *S.R* is the sum of residuals, *N* is Number of Residuals.

At first let's put ƛ =0, then *Similarity Score* = $(-10+4+8)^2 / 3+0 = 4/3 = 1.33$

Let's make the decision tree using these residuals and similarity scores. I've set the tree splitting criteria as Age >10. Again for these two leaves, we calculate the similarity scores which are 100 and 72. After this, we calculate the gain

Gain = S.S of the branch before split - S.S of the branch after the split.

Gain = (100 + 72) – 1.3

Now we set our ɣ, which is a value provided to the model at starting and its used during splitting. If *Gain* > ɣ then split will happen otherwise not. Let's assume that ɣ for this problem is 130 then since the gain is greater than ɣ, further split will occur. By this method, auto tree pruning will be achieved. The greater the ɣ value more pruning will be done.

For regularization and preventing overfitting, we must increase the ƛ which was initially set to 0. But this should be done carefully as greater the ƛ value lesser the Similarity score, lesser the gain and more the pruning.

New prediction = *Previous Prediction* + *Learning rate* * *Output*

XGboost calls the learning rate as *eta* and its value is set to 0.3. For the 2nd reading (Age=15).

New prediction = 30 + (0.3 * 6) = 31.8.

The outcome is 6 is calculated from the average residuals 4 and 8.

New Residual = 34 – 31.8 = 2.2

Age	**IQ**	**Residual**
10	20	-7
15	34	2.2
16	38	6.2

This way model M1 will be trained and residuals will keep on decreasing, which means the loss will be optimized in further models. XGboost has proven to be the most efficient Scalable Tree Boosting Method. The system runs way faster on a single machine than any other machine learning technique with efficient data and memory handling. The algorithm's optimization techniques improve performance and thereby provide speed using the least amount of resources.

Chapter-9

Dimensionality Reduction

Instructor Name: B N V Narasimha Raju

Dimensionality reduction is the process of reducing the number of features (or dimensions) in a dataset while retaining as much information as possible. This can be used for a variety of reasons, such as to reduce the complexity of a model, to improve the performance of a learning algorithm, or to make it easier to visualize the data. There are several techniques for dimensionality reduction, including principal component analysis (PCA) and linear discriminant analysis (LDA). Each technique uses a different method to project the data onto a lower-dimensional space while preserving important information.

In machine learning classification problems, there are often too many factors on the basis of which the final classification is done. These factors are basically variables called features. The higher the number of features, the harder it gets to visualize the training set and then work on it. Sometimes, most of these features are correlated, and hence redundant. This is where dimensionality reduction algorithms are useful. Dimensionality reduction is the process of reducing the number of random variables under consideration, by obtaining a set of principal variables. There are two components for dimensionality reduction. They are

- Feature selection: In this, try to find a subset of the original set of variables, or features, to get a smaller subset which can be used to model the problem.
- Feature extraction: This reduces the data in a high dimensional space to a lower dimension space, i.e. a space with lesser no. of dimensions.

An intuitive example of dimensionality reduction can be discussed through a simple email classification problem, where we need to classify whether the email is spam or not. This can involve a large number of features, such as whether or not the e-mail has a generic title, the content of the e-mail, whether the email uses a template, etc. However, some of these features may overlap. Hence, we can reduce the number of features in such problems. A 3-D classification problem can be hard to visualize, whereas a 2-D one can be mapped to a simple 2 dimensional space, and a 1-D problem to a simple line.

9.1 Principal Component Analysis

Principal Component Analysis (PCA) is one of the popular and unsupervised algorithms that has been used across several applications like data analysis, data compression, de-noising, reducing the dimension of data and a lot more. PCA analysis helps you reduce or eliminate

similar data in the line of comparison that does not even contribute a bit to decision making. PCA analysis reduces dimensionality without any data loss.

PCA helps you find out the most common dimensions of your project and makes result analysis easier. Consider a scenario where you deal with a project with significant variables and dimensions. Not all these variables will be critical. Some may be the primary key variables, whereas others are not. So, the Principal Component Method of factor analysis gives you a calculative way of eliminating a few extra less important variables, thereby maintaining the transparency of all information. PCA is thus called a dimensionality-reduction method. With reduced data and dimensions, you can easily explore and visualize the algorithms without wasting the time. Therefore, PCA statistics is the science of analyzing all the dimensions and reducing them as much as possible while preserving the exact information.

The Principal Component Method of Factor Analysis is useful when you are clueless about when to employ the techniques of PCA analysis. If this is the case, the following guidelines will help us.

- To reduce the number of dimensions in factor analysis but can't decide upon the variable. The principal component method of factor analysis will be useful.
- To categorize the dependent and independent variables in data, this algorithm will be useful.
- Also, to eliminate the noise components in your dimension analysis, PCA is the best computation method.

9.1.1 PCA Example

Let's take a situation where you have to recognize a few patterns of good quality apples in the food processing industry. The factory will contain thousands of quantities. When you have to detect and recognize thousands of samples, you would require an algorithm to sort this out. PCA in machine learning helps you fix this problem.

In the first step, all possible features are categorized as vector components and all the samples are passed out through an algorithm (simply like a sensor that scans the samples) for analysis. After analyzing the bulk reports of the algorithm, you may categorize the apple samples that are having greater variances like (very small/ very large in size, rotten samples, damaged samples, etc.) and at the same time, you may categorize other apple samples that are having smaller variances like (samples with leaves or branches, samples that are not under vector component values, etc).

The samples that are having greater variances will act as FIRST PRINCIPAL COMPONENT and the samples that are having smaller variances will act as SECOND PRINCIPAL COMPONENT. When you represent these two principal components over a pictorial representation in separate dimensions on the correct scale, you will get a clear view of the report. Also, the

components that are out of the border can be considered additional (noise) components and can be ignored if needed. This is just an example intended to give you a clear view of the process. Apart from this, there are a few other calculative steps to give you more preservative information without losing any data.

9.1.2 Explanation of PCA

In this you will learn about the steps involved in the Principal Component Analysis technique.

9.1.2.1 STANDARDIZATION

The range of variables is calculated and standardized in this process to analyze the contribution of each variable equally. Calculating the initial variables will help you categorize the variables that are dominating the other variables of small ranges which will lead to biased results. So, transforming the data to comparable scales can prevent this problem. To transform the variables of the same standard, you can follow the following formula.

$$z = \frac{Value - Mean}{Standard\ deviation}$$

$$mean = \frac{\sum_{i=1}^{n} x_i}{N}$$

$$standard\ Deviation = \sqrt{\frac{\sum_{i=1}^{n} (x_i - mean)^2}{N-1}}$$

where x_1, x_2, x_3,...., x_i are values in a data set and N is the number of values in the data set.

Let us consider the same scenario that we have taken as an example previously. Let us assume the following features of dimensions as F1, F2, F3, and F4.

LARGE SIZE APPLES	ROTTEN APPLES	DAMAGED APPLES	SMALL APPLES
F1	F2	F3	F4
1	5	3	1
4	2	6	3
1	4	3	2
4	4	1	1
5	5	2	3

Calculate the Mean and Standard Deviation for each feature and then, tabulate the same as follows.

	F1	F2	F3	F4
MEAN	3	4	3	2
STANDARD DEVIATION	1.87	1.223	1.87	1

Then, after the Standardization of each variable, the results are tabulated below. This is the Standardized data set.

F1	F2	F3	F4
-1.0695	0.8196	0	-1
0.5347	-1.6393	1.6042	1
-1.0695	0	0	0
0.5347	0	-1.0695	-1
1.0695	0.8196	-0.5347	1

9.1.2.2 COVARIANCE MATRIX COMPUTATION

In this step, you will get to know how the variables of the given data are varying with the mean value calculated. Any interrelated variables can also be sorted out at the end of this step. Because sometimes, variables are highly correlated in such a way that they contain redundant information. So, in order to identify these correlations, we compute the covariance matrix.

A covariance matrix is a *N* x *N* symmetrical matrix that contains the covariances of all possible data sets. The covariance matrix of two-dimensional data is, given as follows:

$$Covariance\ matrix\ = \begin{bmatrix} COV\ (X,X) & COV\ (X,Y) \\ COV\ (Y,X) & COV\ (Y,Y) \end{bmatrix}$$

Where,

$$Covariance = \frac{Sum((X - Mean\ of\ X)(Y - Mean\ of\ Y))}{Number\ of\ data\ points}$$

Make a note that, the covariance of a number with itself is its variance ($COV(X, X) = Var(X)$), the values at the top left and bottom right will have the variances of the same initial number. Likewise, the entries of the Covariance Matrix at the main diagonal will be symmetric concerning the fact that covariance is commutative ($COV(X, Y) = COV(Y, X)$).

If the value of the Covariance Matrix is positive, then it indicates that the variables are correlated. (If *X* increases, *Y* also increases and vice versa). If the value of the Covariance Matrix is negative, then it indicates that the variables are inversely correlated. (If *X* increases, *Y* also decreases and vice versa). As a result, at the end of this step, you will come to know which pair of variables are correlated with each other, so that you might categorize them much easier. The formula to calculate the covariance matrix of the given example will be:

	F1	F2	F3	F4
F1	VAR(F1)	COV(F1,F2)	COV(F1,F3)	COV(F1,F4)
F2	COV(F2,F1)	VAR(F2)	COV(F2, F3)	COV(F2, F4)
F3	COV(F3,F1)	COV(F3,F2)	VAR(F3)	COV(F3,F4)
F4	COV(F4,F1)	COV(F4,F2)	COV(F4,F3)	VAR(F4)

Since you have already standardized the features, you can consider Mean as 0 and Standard Deviation as 1 for each feature.

$VAR(F1) = ((-1.0695-0)^2 + (0.5347-0)^2 + (-1.0695-0)^2 + (0.5347-0)^2 + (1.069-0)^2)/5$

On solving the equation, you get, $VAR(F1) = 0.78$

$COV(F1, F2) = ((-1.0695-0)(0.8196-0) + (0.5347-0)(-1.6393-0) + (-1.0695-0)(0.0000-0) + (0.5347-0)(0.0000-0) + (1.0695-0)(0.8196-0))/5$

On solving the equation, you get, $COV(F1, F2) = -0.8586$. Similarly solving all the features, the covariance matrix will be,

	F1	F2	F3	F4
F1	0.78	-0.8586	-0.055	0.424
F2	-0.8586	0.78	-0.607	-0.326
F3	-0.055	-0.607	0.78	0.426
F4	0.424	-0.326	0.426	0.78

9.1.2.3 FEATURE VECTOR

To determine the principal components of variables, you have to define eigenvalue and eigenvectors for the same. Let A be any square matrix. A nonzero vector v is an eigenvector of A if $Av = \lambda v$ for some number λ, called the corresponding eigenvalue.

Once you have computed the eigenvector components, define eigenvalues in descending order (for all variables) and now you will get a list of principal components. So, the eigenvalues represent the principal components and these components represent the direction of data. This indicates that if the line contains large variables of large variances, then there are many data points on the line. Thus, there is more information on the line too.

Finally, these principal components form a line of new axes for easier evaluation of data and also the differences between the observations can also be easily monitored. Let v be a non-zero vector and λ a scalar. As per the rule, $Av = \lambda v$, then λ is called the eigenvalue associated with the eigenvector v of A *and* I is the Identity Matrix. Upon substituting the values in $det(A- \lambda I) = 0$, you will get the following matrix.

	F1	F2	F3	F4
F1	0.78 - λ	-0.8586	-0.055	0.424
F2	-0.8586	0.78 - λ	-0.607	-0.326
F3	-0.055	-0.607	0.78 - λ	0.426
F4	0.424	-0.326	0.426	0.78 - λ

When you solve the following matrix by considering 0 on the right-hand side, you can define eigenvalues as λ = 2.11691 , 0.855413 , 0.481689 , 0.334007. Then, substitute each eigenvalue in $(A-\lambda I)v = 0$ equation and solve the same for different eigenvectors *v1*, *v2*, *v3* and *v4*. For instance, For $\lambda = 2.11691$, solving the above equation using Cramer's rule, the values for the v vector are *v1* = 0.515514, *v2* = -0.616625, *v3* = 0.399314, *v4* = 0.441098

Follow the same process and you will form the following matrix by using the eigenvectors calculated as instructed.

E1	E2	E3	E4
0.515514	–0.623012	0.0349815	–0.587262
-0.616625	0.113105	0.452326	–0.634336
0.399314	0.744256	–0.280906	–0.455767
0.441098	0.212477	0.845736	0.212173

Now, calculate the sum of each Eigen column, arrange them in descending order and pick up the topmost Eigenvalues. These are your Principal components.

E1	E2
0.515514	–0.623012
-0.616625	0.113105
0.399314	0.744256
0.441098	0.212477

9.1.2.4 RECAST THE DATA ALONG THE PRINCIPAL COMPONENTS AXES

Still now, apart from standardization, you haven't made any changes to the original data. You have just selected the Principal components and formed a feature vector. Yet, the initial data remains the same on their original axes.

This step aims at the reorientation of data from their original axes to the ones you have calculated from the Principal components. This can be done by the following formula.

*Final Data Set= Standardized Original Data Set * Feature Vector*

So, the final data set becomes as follows:

Standardized Original Data Set is

F1	F2	F3	F4
-1.0695	0.8196	0	-1
0.5347	-1.6393	1.6042	1
-1.0695	0	0	0
0.5347	0	-1.0695	-1
1.0695	0.8196	-0.5347	1

The Feature Vector is

E1	E2
0.515514	0.744256
0.441098	0.212477
0.399314	0.113105
-0.616625	–0.623012

By solving the above equations, you will get the transformed data as follows.

LARGE SIZE APPLES	ROTTEN APPLES
0.4268066978	0.00116114000000012
-0.4234920968	-0.39182856
-0.551342223	-0.7959219
0.4652040128	0.89996329
0.0827279480000002	0.28653037

The large dataset is now compressed into a small dataset without any loss of data. This is the significance of Principal Component Analysis.

9.1.3 Advantages of Principal Component Analysis

- Easy to calculate and compute.
- Speeds up machine learning computing processes and algorithms.
- Prevents predictive algorithms from data overfitting issues.
- Increases performance of ML algorithms by eliminating unnecessary correlated variables.
- Principal Component Analysis results in high variance and increases visualization.
- Helps reduce noise that cannot be ignored automatically.

9.1.4 Disadvantages of Principal Component Analysis

- Sometimes, PCA is difficult to interpret. In rare cases, you may find it difficult to identify the most important features even after computing the principal components.
- You may face some difficulties in calculating the covariances and covariance matrices.
- Sometimes, the computed principal components can be more difficult to read rather than the original set of components.

9.2 Linear Discriminant Analysis

Linear Discriminant Analysis (LDA) or Normal Discriminant Analysis or Discriminant Function Analysis is a supervised learning algorithm used for classification tasks in machine learning. It is a technique used to find a linear combination of features that best separates the classes in a dataset.

LDA works by projecting the data onto a lower-dimensional space that maximizes the separation between the classes. It does this by finding a set of linear discriminants that maximize the ratio of between-class variance to within-class variance. In other words, it finds the directions in the feature space that best separate the different classes of data.

LDA assumes that the data has a Gaussian distribution and that the covariance matrices of the different classes are equal. It also assumes that the data is linearly separable, meaning that a linear decision boundary can accurately classify the different classes.

9.2.1 Explanation of LDA

LDA is used for modeling differences in groups i.e. separating two or more classes. It is used to project the features in higher dimension space into a lower dimension space. For example, we have two classes and we need to separate them efficiently. Classes can have multiple features. Using only a single feature to classify them may result in some overlapping as shown in the figure 9.1. So, we will keep on increasing the number of features for proper classification.

Figure 9.1: Overlapping

Suppose we have two sets of data points belonging to two different classes that we want to classify. As shown in the 2D graph in figure 9.2, when the data points are plotted on the 2D plane, there's no straight line that can separate the two classes of the data points completely. Hence, in this case, LDA (Linear Discriminant Analysis) is used which reduces the 2D graph into a 1D graph in order to maximize the separability between the two classes.

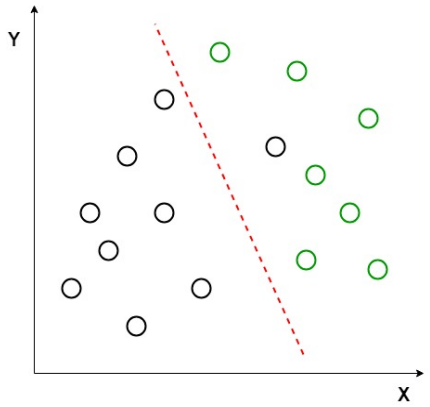


Figure 9.2: 2D-Graph

Here, Linear Discriminant Analysis uses both the axes (X and Y) to create a new axis and projects data onto a new axis in a way to maximize the separation of the two categories and hence, reducing the 2D graph into a 1D graph.

Two criteria are used by LDA to create a new axis:

- Maximize the distance between means of the two classes.
- Minimize the variation within each class.

In the figure 9.3, it can be seen that a new axis (in red) is generated and plotted in the 2D graph such that it maximizes the distance between the means of the two classes and minimizes the variation within each class. In simple terms, this newly generated axis increases the separation between the data points of the two classes. After generating this new axis using the above-mentioned criteria, all the data points of the classes are plotted on this new axis and are shown in the figure 9.4.

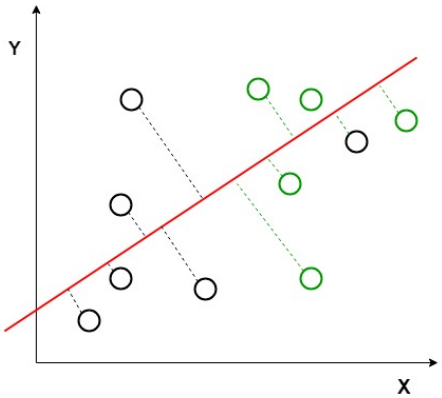


Figure 9.3: 2D-Graph

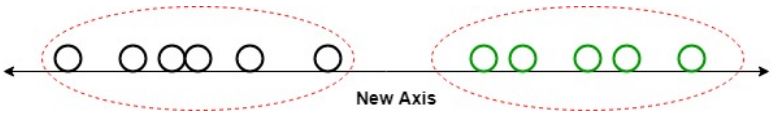


Figure 9.4: New Axis

But Linear Discriminant Analysis fails when the mean of the distributions are shared, as it becomes impossible for LDA to find a new axis that makes both the classes linearly separable. In such cases, we use non-linear discriminant analysis.

Let's suppose we have two classes and m-dimensional samples such as $x_1, x_2,..., x_n$, where N_1 samples come from the class (w_1) and N_2 coming from the class (w_2). We seek to obtain a scalar y by projecting the samples x onto a line

$$y = w^T x \quad where \quad x = \begin{bmatrix} x_1 \\ . \\ . \\ x_m \end{bmatrix} \quad and \quad w = \begin{bmatrix} w_1 \\ . \\ . \\ w_m \end{bmatrix}$$

where w is the projection vectors used to project x to y. Of all the possible lines we would like to select the one that maximizes the separability of the scalars. In order to find a good projection vector, we need to define a measure of separation between the projections.

Let's consider μ_1 and μ_2 be the means of samples class w_1 and w_2 respectively before projection and $\tilde{\mu}_i$ denotes the mean of the samples of class after projection and it can be calculated by:

$$\tilde{\mu}_i = \frac{1}{N_i} \sum_{y \in \omega_i} y = \frac{1}{N_i} \sum_{x \in \omega_i} w^T x$$

$$= w^T \frac{1}{N_i} \sum_{x \in \omega_i} x = w^T \mu_i$$

For each class we define the scatter. Now, In LDA we need to normalize $\tilde{\mu}1$ and $\tilde{\mu}2$. Let $y_i = w^T x_i$ be the projected samples, then scatter for the samples of w_i is

$$\tilde{s}_i^2 = \sum_{y \in \omega_i} (y - \tilde{\mu}_i)^2$$

$\tilde{s}_i^2$ measures the variability within class w_i after projecting it on the y-space. Thus $\tilde{s}_1^2 + \tilde{s}_2^2$ measures the variability within the two classes at hand after projection, hence it is called within-class scatter of the projected samples. Now, we need to project our data on the line having direction w which maximizes

$$J(w) = \frac{\left| \tilde{\mu}_1 - \tilde{\mu}_2 \right|^2}{\tilde{s}_1^2 + \tilde{s}_2^2}$$

For maximizing the above equation we need to find a projection vector that maximizes the difference of means of reducing the scatters of both classes. Now, scatter matrix of s_i of class w_i are:

$$S_i = \sum_{x \in \omega_i} (x - \mu_i)(x - \mu_i)^T$$

Now, we define, scatter within the classes(s_w) and scatter b/w the classes (s_B):

$$S_w = S_1 + S_2$$

$$S_B = (\mu_1 - \mu_2)(\mu_1 - \mu_2)^T$$

Now, we try to simplify the numerator part of *J*(*w*)

$$J(w) = \frac{\left|\tilde{\mu}_1 - \tilde{\mu}_2\right|^2}{\tilde{s}_1^2 + \tilde{s}_2^2} = \frac{w^T S_B w}{w^T S_W w}$$

Now, To maximize the above equation we need to calculate differentiation with respect to *w*

$$\frac{d}{dw} J(w) = \frac{d}{dw}\left(\frac{w^T S_B w}{w^T S_W w}\right) = 0$$

$$\Rightarrow S_B w - \left(\frac{w^T S_B w}{w^T S_W w}\right) S_W w = 0$$

$$\Rightarrow S_B w - J(w) S_W w = 0$$

$$\Rightarrow S_W^{-1} S_B w - J(w) w = 0$$

Here, for the maximum value of J(w) we will use the value corresponding to the highest eigenvalue. This will provide us with the best solution for LDA. Solving the generalized eigenvalue problem

$$S_W^{-1} S_B w = \lambda w \qquad where \quad \lambda = J(w) = scalar$$

Yields

$$w^* = S_W^{-1}(\mu_1 - \mu_2)$$

This is known as Fisher's Linear Discriminant, although it is not a discriminant but rather a specific choice of direction for the projection of the data down to one dimension.

9.2.2 LDA Example

Compute the Linear Discriminant projection for the following two dimensional dataset.

Samples for class ω_1 : $\mathbf{X}_1=(x_1,x_2)=\{(4,2),(2,4),(2,3),(3,6),(4,4)\}$

Sample for class ω_2 : $\mathbf{X}_2=(x_1,x_2)=\{(9,10),(6,8),(9,5),(8,7),(10,8)\}$

The classes mean are

$$\mu_1 = \frac{1}{N_1}\sum_{x\in\omega_1} x = \frac{1}{5}\left[\begin{pmatrix}4\\2\end{pmatrix}+\begin{pmatrix}2\\4\end{pmatrix}+\begin{pmatrix}2\\3\end{pmatrix}+\begin{pmatrix}3\\6\end{pmatrix}+\begin{pmatrix}4\\4\end{pmatrix}\right] = \begin{pmatrix}3\\3.8\end{pmatrix}$$

$$\mu_2 = \frac{1}{N_2}\sum_{x\in\omega_2} x = \frac{1}{5}\left[\begin{pmatrix}9\\10\end{pmatrix}+\begin{pmatrix}6\\8\end{pmatrix}+\begin{pmatrix}9\\5\end{pmatrix}+\begin{pmatrix}8\\7\end{pmatrix}+\begin{pmatrix}10\\8\end{pmatrix}\right] = \begin{pmatrix}8.4\\7.6\end{pmatrix}$$

Scatter matrix of the first class

$$\begin{aligned} S_1 = \sum_{x\in\omega_1}(x-\mu_1)(x-\mu_1)^T &= \left[\begin{pmatrix}4\\2\end{pmatrix}-\begin{pmatrix}3\\3.8\end{pmatrix}\right]^2+\left[\begin{pmatrix}2\\4\end{pmatrix}-\begin{pmatrix}3\\3.8\end{pmatrix}\right]^2 \\ &+\left[\begin{pmatrix}2\\3\end{pmatrix}-\begin{pmatrix}3\\3.8\end{pmatrix}\right]^2+\left[\begin{pmatrix}3\\6\end{pmatrix}-\begin{pmatrix}3\\3.8\end{pmatrix}\right]^2+\left[\begin{pmatrix}4\\4\end{pmatrix}-\begin{pmatrix}3\\3.8\end{pmatrix}\right]^2 \\ &= \begin{pmatrix}1 & -0.25\\ -0.25 & 2.2\end{pmatrix} \end{aligned}$$

Scatter matrix of the second class

$$\begin{aligned} S_2 = \sum_{x\in\omega_2}(x-\mu_2)(x-\mu_2)^T &= \left[\begin{pmatrix}9\\10\end{pmatrix}-\begin{pmatrix}8.4\\7.6\end{pmatrix}\right]^2+\left[\begin{pmatrix}6\\8\end{pmatrix}-\begin{pmatrix}8.4\\7.6\end{pmatrix}\right]^2 \\ &+\left[\begin{pmatrix}9\\5\end{pmatrix}-\begin{pmatrix}8.4\\7.6\end{pmatrix}\right]^2+\left[\begin{pmatrix}8\\7\end{pmatrix}-\begin{pmatrix}8.4\\7.6\end{pmatrix}\right]^2+\left[\begin{pmatrix}10\\8\end{pmatrix}-\begin{pmatrix}8.4\\7.6\end{pmatrix}\right]^2 \\ &= \begin{pmatrix}2.3 & -0.05\\ -0.05 & 3.3\end{pmatrix} \end{aligned}$$

Within-class scatter matrix

$$\begin{aligned} S_w = S_1 + S_2 &= \begin{pmatrix}1 & -0.25\\ -0.25 & 2.2\end{pmatrix}+\begin{pmatrix}2.3 & -0.05\\ -0.05 & 3.3\end{pmatrix} \\ &= \begin{pmatrix}3.3 & -0.3\\ -0.3 & 5.5\end{pmatrix} \end{aligned}$$

Between-class scatter matrix

$$\begin{aligned} S_B &= (\mu_1-\mu_2)(\mu_1-\mu_2)^T \\ &= \left[\begin{pmatrix}3\\3.8\end{pmatrix}-\begin{pmatrix}8.4\\7.6\end{pmatrix}\right]\left[\begin{pmatrix}3\\3.8\end{pmatrix}-\begin{pmatrix}8.4\\7.6\end{pmatrix}\right]^T \\ &= \begin{pmatrix}-5.4\\-3.8\end{pmatrix}\begin{pmatrix}-5.4 & -3.8\end{pmatrix} \\ &= \begin{pmatrix}29.16 & 20.52\\ 20.52 & 14.44\end{pmatrix} \end{aligned}$$

The LDA projection is then obtained as the solution of the generalized eigenvalue problem. The optimal projection is the one that is given maximum $\lambda = J(w)$. Directly

$$w^* = S_W^{-1}(\mu_1 - \mu_2) = \begin{pmatrix} 3.3 & -0.3 \\ -0.3 & 5.5 \end{pmatrix}^{-1} \left[\begin{pmatrix} 3 \\ 3.8 \end{pmatrix} - \begin{pmatrix} 8.4 \\ 7.6 \end{pmatrix} \right]$$

$$= \begin{pmatrix} 0.3045 & 0.0166 \\ 0.0166 & 0.1827 \end{pmatrix} \begin{pmatrix} -5.4 \\ -3.8 \end{pmatrix}$$

$$= \begin{pmatrix} 0.9088 \\ 0.4173 \end{pmatrix}$$

Transforming the samples onto the new subspace i.e. $y = x\ w$ (where x is a $n \times d$-dimensional matrix representing the n samples, and y are the transformed $n \times k$-dimensional samples in the new subspace).

9.2.3 Advantages of LDA

- It is a simple and computationally efficient algorithm.
- It can work well even when the number of features is much larger than the number of training samples.
- It can handle multicollinearity (correlation between features) in the data.

9.2.4 Limitations of LDA

- It assumes that the data has a Gaussian distribution, which may not always be the case.
- It assumes that the covariance matrices of the different classes are equal, which may not be true in some datasets.
- It assumes that the data is linearly separable, which may not be the case for some datasets.
- It may not perform well in high-dimensional feature spaces.

Chapter-10

Model Evaluation and Optimization

Instructor Name: B N V Narasimha Raju

10.1 Cross Validation

In machine learning, we couldn't fit the model on the training data and can't say that the model will work accurately for the real data. For this, we must assure that our model got the correct patterns from the data, and it is not getting too much noise. For this purpose, we use the cross-validation technique.

Cross validation is a technique used in machine learning to evaluate the performance of a model on unseen data. It involves dividing the available data into multiple folds or subsets, using one of these folds as a validation set, and training the model on the remaining folds. This process is repeated multiple times, each time using a different fold as the validation set. Finally, the results from each validation step are averaged to produce a more robust estimate of the model's performance.

The main purpose of cross validation is to prevent overfitting, which occurs when a model is trained too well on the training data and performs poorly on new, unseen data. By evaluating the model on multiple validation sets, cross validation provides a more realistic estimate of the model's generalization performance, i.e., its ability to perform well on new, unseen data.

Cross-validation is a technique in which we train our model using the subset of the data-set and then evaluate using the complementary subset of the data-set. The three steps involved in cross-validation are as follows :

- Reserve some portion of sample data-set.
- Using the rest data-set, train the model.
- Test the model using the reserve portion of the data-set.

There are several types of cross validation techniques, including Hold Out method, k-fold cross validation, leave-one-out cross validation, and stratified cross validation. The choice of technique depends on the size and nature of the data, as well as the specific requirements of the modeling problem.

10.1.1 Hold Out method

This is the simplest evaluation method and is widely used in Machine Learning projects. Here the entire dataset is divided into 2 sets – train set and test set. The data can be divided into 70-30 or 60-40, 75-25 or 80-20, or even 50-50 depending on the use case. As a rule, the proportion of training data has to be larger than the test data as shown in figure 10.1.

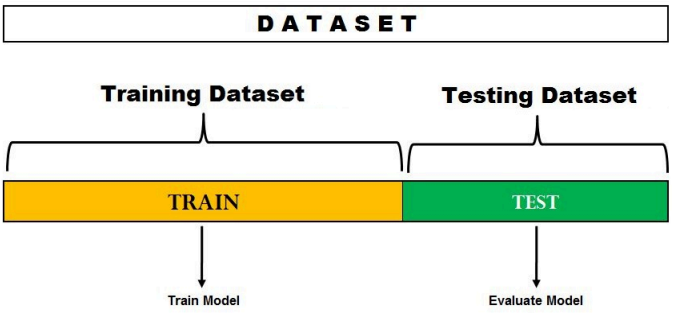


Figure 10.1: Split of Train and Test Datasets

The data split happens randomly, and we can't be sure which data ends up in the train and test bucket during the split unless we specify random_state. This can lead to extremely high variance and every time the split changes, the accuracy will also change.

There are some drawbacks to this method

- In the Hold out method, the test error rates are highly variable (high variance) and it totally depends on which observations end up in the training set and test set
- Only a part of the data is used to train the model (high bias) which is not a very good idea when data is not huge and this will lead to overestimation of test error.

One of the major advantages of this method is that it is computationally inexpensive compared to other cross-validation techniques.

10.1.2 Validation Set Approach

We divide our input dataset into a training set and test or validation set in the validation set approach. Both the subsets are given 50% of the dataset. But one of the big disadvantages is that we are just using a 50% dataset to train our model, so the model may miss out on capturing important information of the dataset. It also tends to give the under fitted model.

10.1.3 Leave One Out Cross-Validation

In this method, we divide the data into train and test sets – but with a twist. Instead of dividing the data into 2 subsets, we select a single observation as test data, and everything else is labeled as training data and the model is trained. Now the 2nd observation is selected as test data and the model is trained on the remaining data.

This process continues ‘n’ times and the average of all these iterations is calculated and estimated as the test set error as shown in figure 10.2.

$$CV_{(n)} = \frac{1}{n}\sum_{i=1}^{n} MSE_i$$

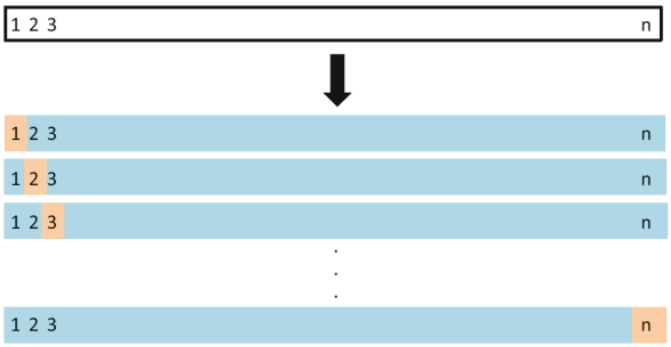


Figure 10.2: Leave One Out Cross-Validation

When it comes to test-error estimates, LOOCV gives unbiased estimates (low bias). But bias is not the only matter of concern in estimation problems. We should also consider variance.

LOOCV has an extremely high variance because we are averaging the output of n-models which are fitted on an almost identical set of observations, and their outputs are highly positively correlated with each other.

It is computationally expensive as the model is run ‘n’ times to test every observation in the data. Our next method will tackle this problem and give us a good balance between bias and variance.

10.1.4 Leave-P-out cross-validation

In this approach, the p datasets are left out of the training data. It means, if there are total n data points in the original input dataset, then n-p data points will be used as the training dataset and the p data points as the validation set. This complete process is repeated for all the samples, and the average error is calculated to know the effectiveness of the model. There is a disadvantage of this technique is, it can be computationally difficult for the large p.

10.1.5 K-Fold Cross-Validation

In this resampling technique, the whole data is divided into k sets of almost equal sizes. The first set is selected as the test set and the model is trained on the remaining k-1 sets. The test error rate is then calculated after fitting the model to the test data.

In the second iteration, the 2nd set is selected as a test set and the remaining k-1 sets are used to train the data and the error is calculated. This process continues for all the k sets as shown in figure 10.3.

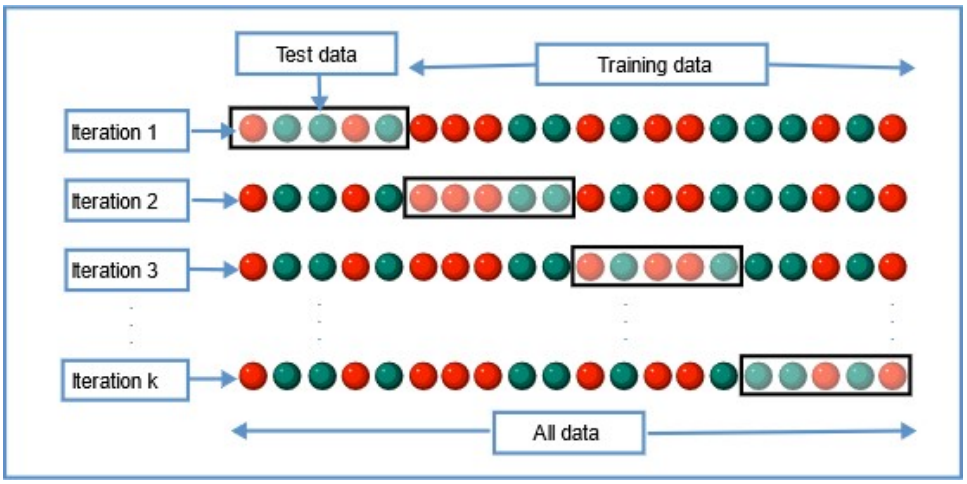


Figure 10.3: K-Fold Cross-Validation

The mean of errors from all the iterations is calculated as the CV test error estimate.

$$CV_{(k)} = \frac{1}{k}\sum_{i=1}^{k} MSE_i$$

In K-Fold CV, the number of folds k is less than the number of observations in the data (k<n) and we are averaging the outputs of k fitted models that are somewhat less correlated with each other since the overlap between the training sets in each model is smaller. This leads to low variance then LOOCV.

The best part about this method is each data point gets to be in the test set exactly once and gets to be part of the training set k-1 times. As the number of folds k increases, the variance also decreases (low variance). This method leads to intermediate bias because each training set contains fewer observations (k-1)n/k than the Leave One Out method but more than the Hold Out method.

Typically, K-fold Cross Validation is performed using k=5 or k=10 as these values have been empirically shown to yield test error estimates that neither have high bias nor high variance.

The major disadvantage of this method is that the model has to be run from scratch k-times and is computationally expensive than the Hold Out method but better than the Leave One Out method.

10.1.6 Stratified K-Fold Cross-Validation

This is a slight variation from K-Fold Cross Validation, which uses 'stratified sampling' instead of 'random sampling.' Let's quickly understand what stratified sampling is and how it is different from random sampling.

Suppose your data contains reviews for a cosmetic product used by both the male and female population. When we perform random sampling to split the data into train and test sets, there is a possibility that most of the data representing males is not represented in training data but might end up in test data. When we train the model on sample training data that is not a correct representation of the actual population, the model will not predict the test data with good accuracy.

This is where Stratified Sampling comes to the rescue. Here the data is split in such a way that it represents all the classes from the population.

Let's consider the above example which has a cosmetic product review of 1000 customers out of which 60% is female and 40% is male. I want to split the data into train and test data in proportion (80:20). 80% of 1000 customers will be 800 which will be chosen in such a way that there are 480 reviews associated with the female population and 320 representing the male population. In a similar fashion, 20% of 1000 customers will be chosen for the test data (with the same female and male representation).

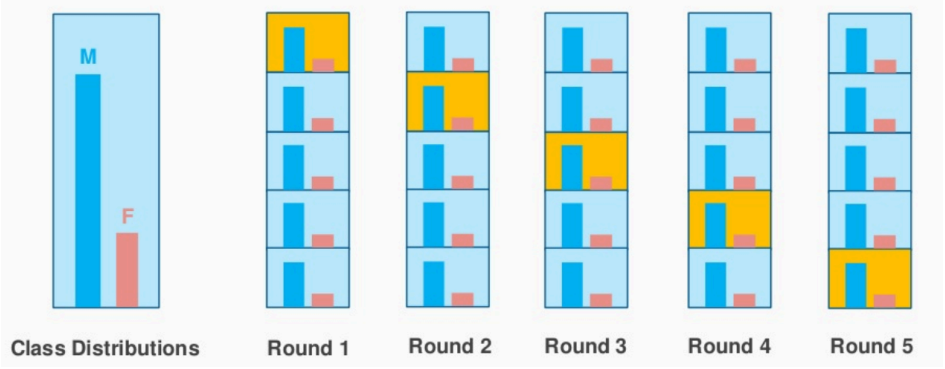


Figure 10.4: Stratified K-Fold Cross-Validation

This is exactly what stratified K-Fold CV does and it will create K-Folds by preserving the percentage of sample for each class as shown in figure 10.4. This solves the problem of random sampling associated with Hold out and K-Fold methods.

10.2 Grid Search

Grid Search is a tool that is used for hyperparameter tuning. Machine Learning in practice comes down to comparing different models to each other and trying to find the best working model.

Apart from selecting the right data set, there are generally two aspects of optimizing a predictive model

- Optimize the choice of the best model
- Optimize a model's fit using hyperparameters tuning

10.2.1 Optimize the choice of the best model

In some datasets, there may exist a simple linear relationship that can predict a target variable from the explanatory variables. In other datasets, these relationships may be more complex or highly nonlinear.

At the same time, many models exist. This ranges from simple models like the Linear Regression, up to very complex models like Deep Neural Networks. It is key to use a model that is appropriate for our data.

For example, if we use a Linear Regression on a very complex task, the model will not be performant. But if we use a Deep Neural Network on a very simple task, this will also not be performant!

To find a well-fitting Machine Learning model, the solution is to split data into train and test data, then fit many models on the training data and test each of them on the test data. The model that has the smallest error on the test data will be kept.

10.2.2 Optimize a model's fit using hyperparameters tuning

After choosing one well-performing model (or a few), the second thing to optimize is the hyperparameters of a model. Hyperparameters are like a configuration of the training phase of the model. They influence what a model can or cannot learn.

Tuning hyperparameters can, therefore, lower the error on the test data set even more. The way of estimating is different for each model, and thus each model has its own hyperparameters to optimize. One way to do a thorough search for the best hyperparameters is to use a tool called GridSearch.

10.2.3 Explanation of Grid Search

GridSearch is an optimization tool that we use when tuning hyperparameters. We define the grid of parameters that we want to search through, and we select the best combination of parameters for our data.

The hypothesis is that there is a specific combination of values of the different hyperparameters that will minimize the error of our predictive model. Our goal using Grid Search is to find this specific combination of parameters.

Grid Search's idea for finding this best parameter combination is simple: just test each parameter combination possible and select the best one. Not really each combination possible though, since for a continuous scale there would be infinitely many combinations to test. The solution for this is to define a Grid. This Grid defines for each hyperparameter, which values should be tested.

In an example case where two hyperparameters — Alpha and Beta— are tuned: we could give both of them the values [0.1, 0.01, 0.001, 0.0001] resulting in the following "Grid" of values as shown in figure. At each crossing point, our GridSearch will fit the model to see what the error at this point is.

And after checking all the grid points, we know which parameter combination is best for our prediction as shown in figure 10.5.

	Alpha 0.1	**0.01**	**0.001**	**0.0001**
Beta 0.1	Found Accuracy: 0.716	Found Accuracy: 0.76	Found Accuracy: 0.81	Found Accuracy: 0.78
0.01	Found Accuracy: 0.721	Found Accuracy: 0.81	Found Accuracy: 0.94	Found Accuracy: 0.86
0.001	Found Accuracy: 0.709	Found Accuracy: 0.88	Found Accuracy: 0.80	Found Accuracy: 0.73
0.0001	Found Accuracy: 0.72	Found Accuracy: 0.69	Found Accuracy: 0.71	Found Accuracy: 0.70

Figure 10.5: A schematic overview of GridSearch on two hyperparameters Alpha and Beta

10.2.4 The Cross-Validation in GridSearch

At this point, only one thing remains to be added: the Cross-Validation Error. When testing the performance of a model with each combination of hyperparameters, there could be a risk of overfitting. This means that just by pure chance, only the training data set corresponded well to this particular hyperparameter combination. The performance on new, real-life data, could be much worse. To get a more reliable estimate of the performances of a hyperparameter combination, we take the Cross Validation Error.

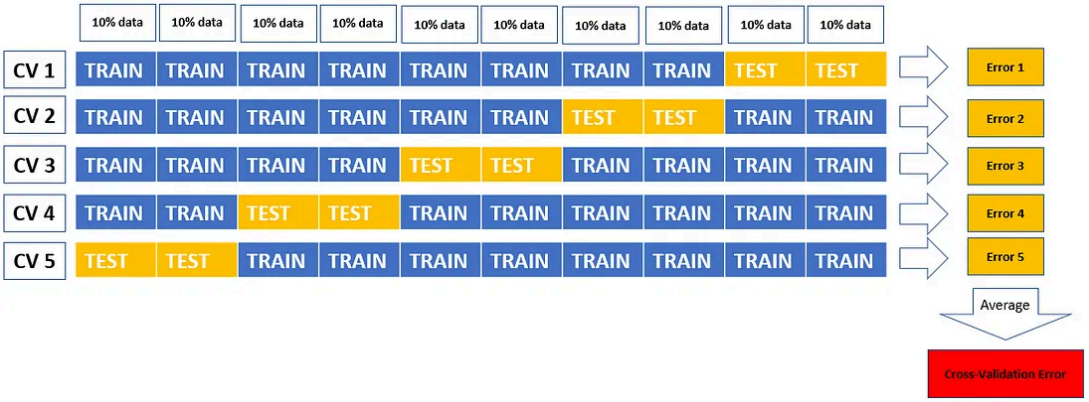


Figure 10.6: A schematic overview of Cross-Validation

In Cross-Validation, the data is split in multiple parts. For example 5 parts. Then the model is fit 5 times while leaving out one-fifth of the data. This one-fifth left-out data is used to measure the performances.

For one combination of hyperparameter values, the average of the 5 errors constitutes the cross-validation error. This makes the selection of the final combination more reliable as shown in figure 10.6.

GridSearch allows us to find the best model given a data set very easily. It actually makes the Machine Learning part of the Data Scientists role much easier by automating the search.

On the Machine Learning side, some things that still remain to be done is deciding on the right way to measure error, deciding on which models to try out and which hyperparameters to test for. And the most important part, the work on data preparation, is also left for the data scientist.

10.3 Regularization

Regularization refers to techniques that are used to calibrate machine learning models in order to minimize the adjusted loss function and prevent overfitting or underfitting.

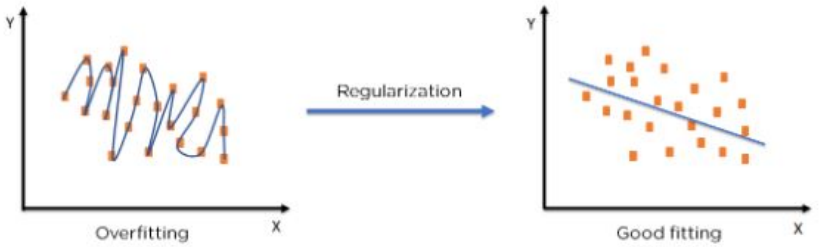


Figure 10.7: Regularization on an over-fitted model

Using Regularization, we can fit our machine learning model appropriately on a given test set and hence reduce the errors in it. There are two main types of regularization techniques: Ridge Regularization and Lasso Regularization.

10.3.1 Ridge Regularization

Also known as Ridge Regression, it modifies the over-fitted or under-fitted models by adding the penalty equivalent to the sum of the squares of the magnitude of coefficients.

This means that the mathematical function representing our machine learning model is minimized and coefficients are calculated. The magnitude of coefficients is squared and added. Ridge Regression performs regularization by shrinking the coefficients present. The function depicted below shows the cost function of ridge regression

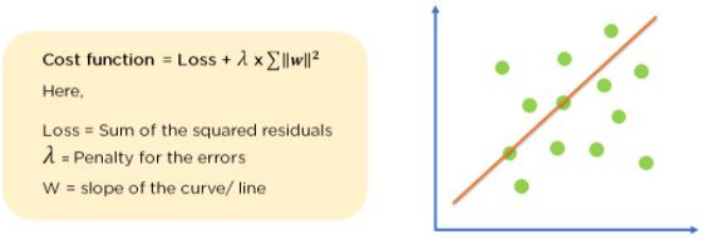


Figure 10.7: Cost Function of Ridge Regression

In the cost function, the penalty term is represented by Lambda λ. By changing the values of the penalty function, we are controlling the penalty term. The higher the penalty, it reduces the magnitude of coefficients. It shrinks the parameters. Therefore, it is used to prevent multicollinearity, and it reduces the model complexity by coefficient shrinkage. Consider the graph illustrated below which represents Linear regression :

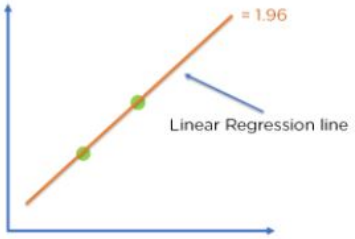


Figure 10.8: Linear regression model

Cost function = Loss + λ x ∑||w||^2

For Linear Regression line, let's consider two points that are on the line,

Loss = 0 (considering the two points on the line), λ= 1, w = 1.4

Then, Cost function = 0 + 1 x 1.42 = 1.96

For Ridge Regression, let's assume,

Loss = 0.32 + 0.22 = 0.13, λ = 1, w = 0.7

Then, Cost function = 0.13 + 1 x 0.72 = 0.62.

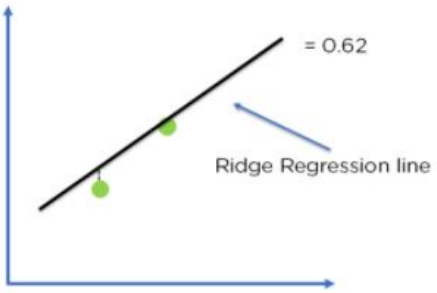


Figure 10.9: Ridge regression model

Comparing the two models, with all data points, we can see that the Ridge regression line fits the model more accurately than the linear regression line.

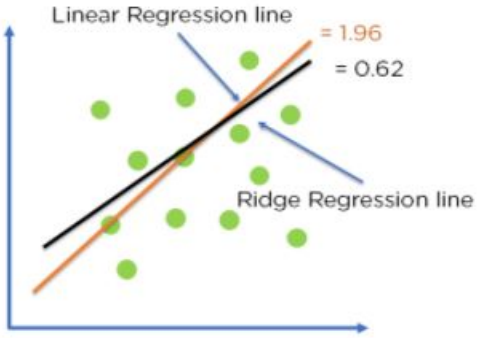


Figure 10.10: Optimization of model fit using Ridge Regression

10.3.2 Lasso Regression

It modifies the over-fitted or under-fitted models by adding the penalty equivalent to the sum of the absolute values of coefficients.

Lasso regression also performs coefficient minimization, but instead of squaring the magnitudes of the coefficients, it takes the true values of coefficients. This means that the coefficient sum can also be 0, because of the presence of negative coefficients. Consider the cost function for Lasso regression.

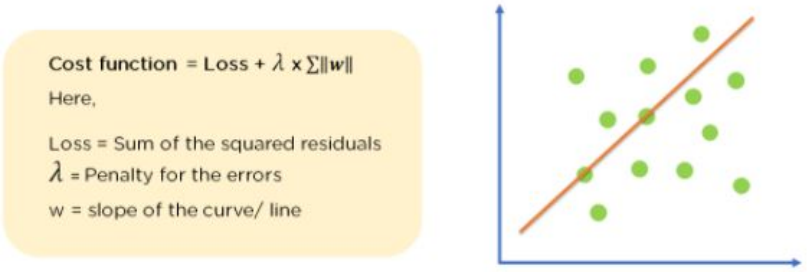


Figure 10.10: Cost function for Lasso Regression

We can control the coefficient values by controlling the penalty terms, just like we did in Ridge Regression. Again consider a Linear Regression model.

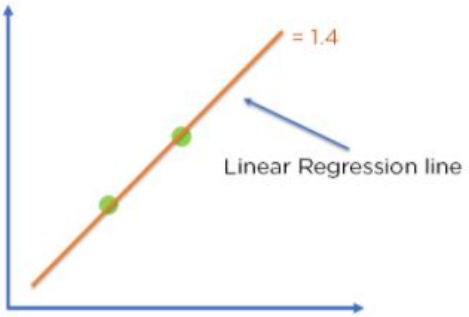


Figure 10.12: Linear Regression Model

Cost function = Loss + λ x ∑||w||

For Linear Regression line, let's assume,

Loss = 0 (considering the two points on the line), λ = 1, w = 1.4

Then, Cost function = 0 + 1 x 1.4 = 1.4

For Ridge Regression, let's assume,

Loss = 0.32 + 0.12 = 0.1, λ = 1, w = 0.7

Then, Cost function = 0.1 + 1 x 0.7 = 0.8

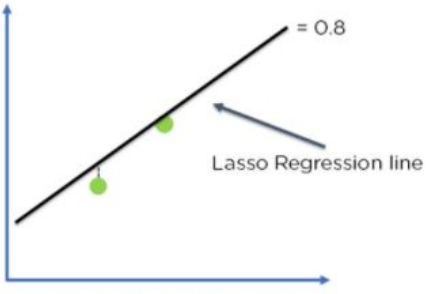


Figure 10.12: Lasso Regression Model

Comparing the two models, with all data points, we can see that the Lasso regression line fits the model more accurately than the linear regression line.

Chapter-11

Neurons, NNs, Linear Discriminants

Instructor Name: B N V Narasimha Raju

11.1 Neural Networks

Neural networks are used to mimic the basic functioning of the human brain and are inspired by how the human brain interprets information. It is used to solve various real-time tasks because of its ability to perform computations quickly and its fast responses.

An Artificial Neural Network model contains various components that are inspired by the biological nervous system. Artificial Neural Network has a huge number of interconnected processing elements, also known as Nodes. Each node is designed to behave similarly to a neuron in the brain. These nodes are connected with other nodes using a connection link. The connection link contains weights, these weights contain the information about the input signal. Each iteration and input in turn leads to updation of these weights.

After inputting all the data instances from the training data set, the final weights of the Neural Network along with its architecture is known as the Trained Neural Network. This process is called Training of Neural Networks. This trained neural network is used to solve specific problems as defined in the problem statement. Types of tasks that can be solved using an artificial neural network include Classification problems, Pattern Matching, Data Clustering, etc.

We use artificial neural networks because they learn very efficiently and adaptively. They have the capability to learn “how” to solve a specific problem from the training data it receives. After learning, it can be used to solve that specific problem very quickly and efficiently with high accuracy.

An artificial neuron can be thought of as a simple or multiple linear regression model with an activation function at the end. A neuron from layer i will take the output of all the neurons from the later i-1 as inputs calculate the weighted sum and add bias to it. After this is sent to an activation function as shown in figure 11.1. The activation function calculates the output value for the neuron.

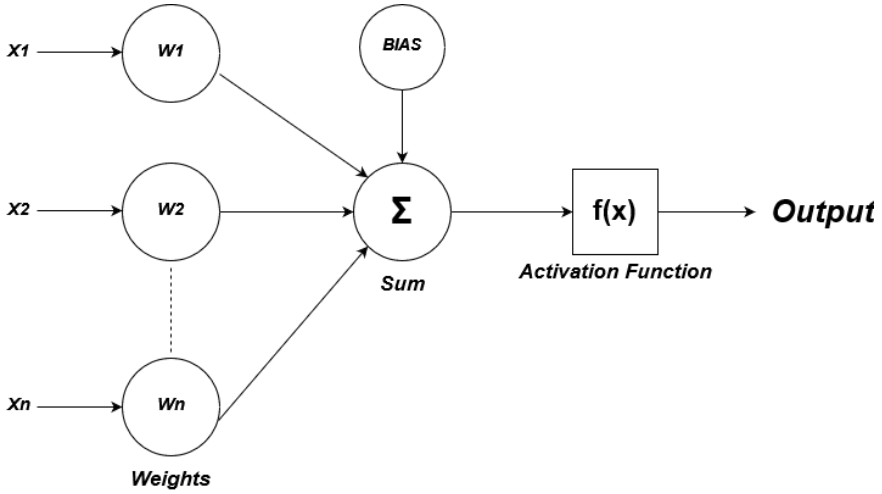


Figure 11.1: Working of Neural Network

The first neuron from the first layer is connected to all the inputs from the previous layer, Similarly, the second neuron from the first hidden layer will also be connected to all the inputs from the previous layer and so on for all the neurons in the first hidden layer as shown in figure 11.2.

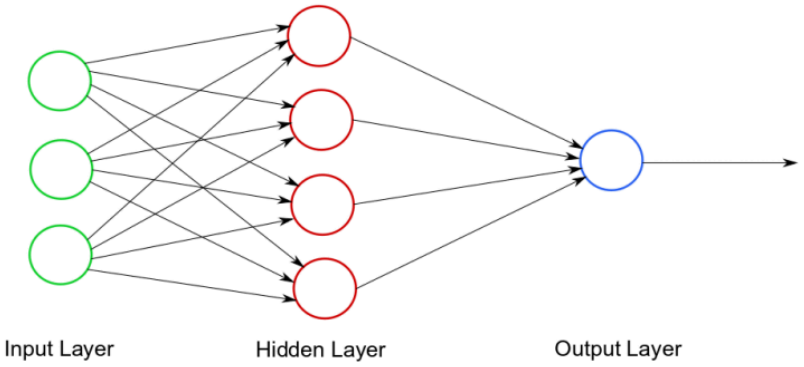


Figure 11.2: Layers of Neural Network

For neurons in the second hidden layer, the outputs of the previously hidden layer are considered as inputs and each of these neurons are connected to previous neurons, likewise. This whole process is called Forward propagation.

Once we have predicted the output it is then compared to the actual output. We then calculate the loss and try to minimize it. To minimize the loss use Back Propagation. In Back propagation, first the loss is calculated then weights and biases are adjusted in such a way that they try to minimize the loss. Weights and biases are updated with the help of an algorithm called gradient descent.

11.2 The perceptron

A perceptron is the smallest element of a neural network. Perceptron is a single-layer neural network or a Machine Learning algorithm used for supervised learning of various binary classifiers. It works as an artificial neuron to perform computations by learning elements and processing them for detecting the business intelligence and capabilities of the input data. A perceptron network is a group of simple logical statements that come together to create an array of complex logical statements, known as the neural network.

The human brain is a complex network of billions of interconnected cells known as Neurons. These cells process and transmit signals. Biological neurons respond to both chemical and electrical signals to create the Biological Neural Network (BNN). The input and output signals can either increase or decrease the potential of the neuron to fire.

11.2.1 Artificial Neuron

An artificial neuron is based on a model of biological neurons but it is a mathematical function. The neuron takes inputs in the form of binary values i.e. 1 or 0, meaning that they can either be ON or OFF. The output of an artificial neuron is usually calculated by applying a threshold function to the sum of its input values.

The threshold function can be either linear or nonlinear. A linear threshold function produces an output of 1 if the sum of the input values is greater than or equal to a certain threshold, and an output of 0 if the sum of the input values is less than that threshold. A nonlinear threshold function, on the other hand, can produce any output value between 0 and 1, depending on the inputs.

An Artificial Neural Network (ANN) is built on artificial neurons and based on a Feed-Forward strategy. It is known as the simplest type of neural network as it continues learning irrespective of the data being linear or nonlinear. The information flow through the nodes is continuous and stops only after reaching the output node.

11.2.2 Biological Neural Network Vs Artificial Neural Network

The structure of artificial neurons is derived from biological neurons and the network is also formed on a similar principle but there are some differences between a biological neural network and an artificial neural network.

Biological Neural Network	Artificial Neural Network
The speed of processing information is slow	Faster compared to BNN
Storage allocation to new process can be	Storage allocation to new processes

done easily by adjusting the interconnection strengths	is not possible as each location is dedicated to a specified process
Massive parallel applications can be conducted simultaneously	Sequential operations only
Information can be retrieved from the sub-nodes even if it gets corrupted	Once corrupted the information cannot be retrieved
The information being transmitted and processed into the network is not monitored	A control unit monitors all information and activities on the network

11.2.3 Perceptron Vs Neuron

The perceptron is a mathematical model of the biological neuron. It produces binary outputs from input values while taking into consideration weights and threshold values. Though created to imitate the working of biological neurons, the perceptron model has since been replaced by more advanced models like backpropagation networks for training artificial neural networks. Perceptrons use a brittle activation function to give a positive or negative output based on a specific value.

A neuron, also known as a node in a backpropagation artificial neural network produces graded values between 0 and 1. It is a generalization of the idea of the perceptron as the neuron also adds weighted inputs. However, it does not produce a binary output but a graded value based on the proximity of the input to the desired value of 1. The results are biased towards the extreme values of 0 or 1 as the node uses a sigmoidal output function.

11.2.4 Components of a Perceptron

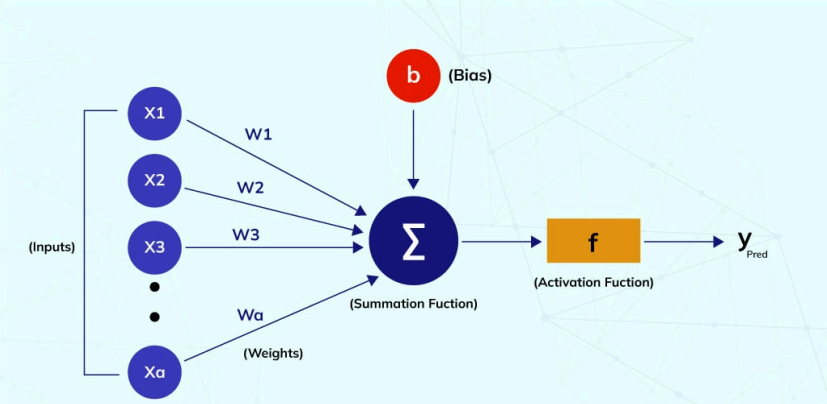


Figure 11.3: Perceptron

As shown in figure 11.3 each perceptron comprises different parts

Input Values: A set of values or a dataset for predicting the output value. They are also described as a dataset's features and dataset.

Weights: The real value of each feature is known as weight. It tells the importance of that feature in predicting the final value.

Bias: The activation function is shifted towards the left or right using bias.

Summation Function: The summation function binds the weights and inputs together. It is a function to find their sum.

Activation Function: It introduces non-linearity in the perceptron model.

11.2.5 Need of Weight and Bias

Weight and bias are two important aspects of the perceptron model. These are learnable parameters and as the network gets trained it adjusts both parameters to achieve the desired values and the correct output.

Weights are used to measure the importance of each feature in predicting output value. Features with values close to zero are said to have lesser weight or significance. These have less importance in the prediction process compared to the features with values further from zero known as weights with a larger value. Besides high-weighted features having greater predictive power than low-weighting ones, the weight can also be positive or negative. If the weight of a feature is positive then it has a direct relation with the target value, and if it is negative then it has an inverse relationship with the target value.

In contrast to weight in a neural network that increases the speed of triggering an activation function, bias delays the trigger of the activation function as shown in figure 11.4. Bias is a constant used to adjust the output and help the model to provide the best fit output for the given data.

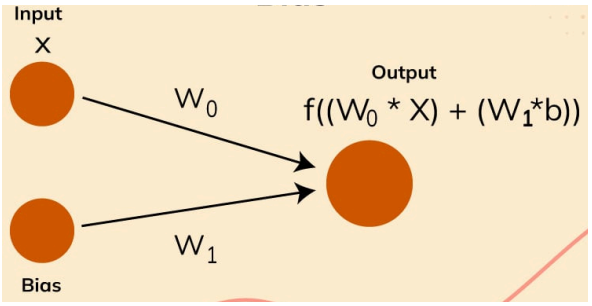


Figure 11.4: Bias

11.2.6 Perceptron Learning Rule

A new type of neural network called perceptrons, which were similar to the neurons. Train these networks with perceptron learning rules. According to the rule, perceptrons can learn automatically to generate the desired results through optimal weight coefficients as shown in figure 11.5.

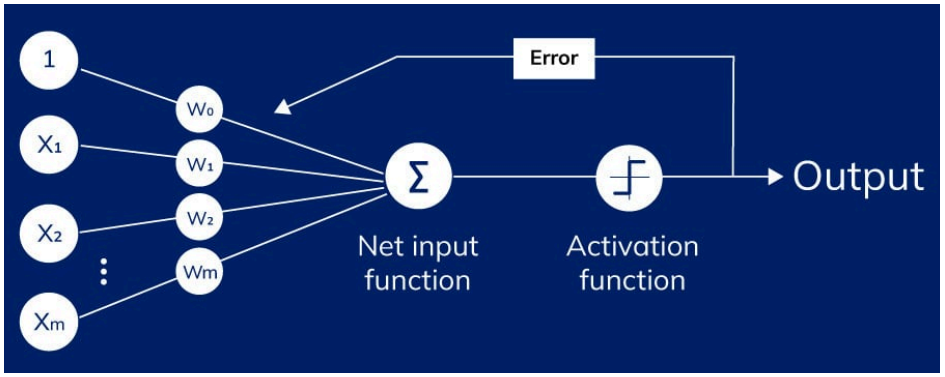


Figure 11.5: Perceptron Rule

11.2.7 Perceptron in Machine Learning

Perceptron in machine learning is used for the supervised learning of the algorithm through various binary classification tasks. The perceptron model in neural networks is one of the simplest artificial neural networks. However, the perceptron learning algorithm is a type of supervised machine-learning system that uses binary classifiers for decision-making.

In Machine Learning, a binary classifier is used to decide whether input data can be represented as vectors of numbers and belongs to some specific category. Binary classifiers are linear because they take into account weight values along with features. It helps the algorithm determine the classification value or probability distribution around the prediction point.

11.2.8 The Perceptron in Neural Network

Neural networks are computational algorithms or models that understand the data and process information. As these artificial neural networks are designed as per the structure of the human brain, the role of neurons in the brain is played by the perceptron in a neural network.

The perceptron model in a neural network is a convenient model of supervised machine learning. Being the early algorithm of binary classifiers it incorporates visual inputs and organizes captions into one of two classes. Machine learning algorithms exploit the crucial element of classification to process, identify and analyze patterns. Perceptron algorithms help in the linear separation of classes and patterns based on the numerical or visual input data.

The perceptron model was used for machine-driven image recognition. Being the first-ever artificial neural network it was claimed to be the most notable AI-based innovation. The perceptron algorithm however had some technical constraints. Being single-layered the perceptron model was only applicable for linearly separable classes. The issue was later resolved by the discovery of multi-layered perceptron algorithms.

11.2.9 Single Layer Perceptron Model

A single-layer perceptron model is the simplest type of artificial neural network. It includes a feed-forward network that can analyze only linearly separable objects while being dependent on a threshold transfer function. The model returns only binary outcomes(target) i.e. 1, and 0.

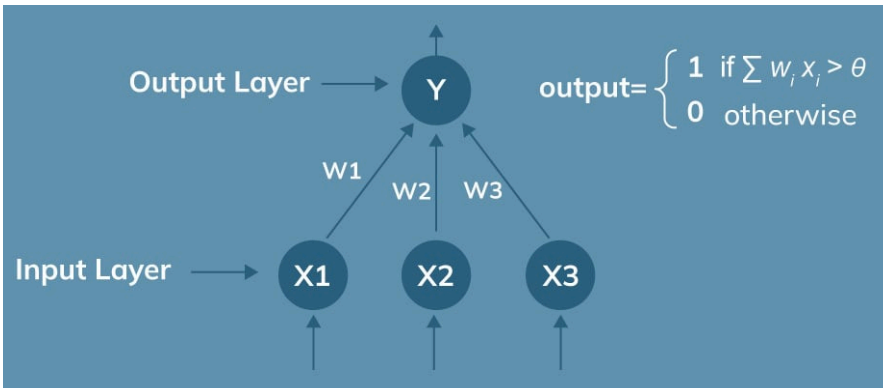


Figure 11.5: Single Layer Perceptron

The algorithm in a single-layered perceptron model does not have any previous information initially. The weights are allocated inconsistently, so the algorithm simply adds up all the weighted inputs. If the value of the sum is more than the threshold or a predetermined value then the output is delivered as 1 and the single-layer perceptron is considered to be activated.

$$w_1x_1 + w_2x_2 + ... + w_nx_n > \theta \rightarrow 1 \text{ (output)}$$

$$w_1x_1 + w_2x_2 + ... + w_nx_n \leq \theta \rightarrow 0 \text{ (output)}$$

When the values of input are similar to those desired for its predicted output, then we can say that the perceptron has performed satisfactorily. If there is any difference between what was expected and obtained, then the weights will need adjusting to limit how much these errors affect future predictions based on unchanged parameters.

$$\Delta w = \eta \times d \times X$$

Where

- d is predicted output - desired output
- η is learning rate, usually less than 1
- X is the input data

However, since the single-layer perceptron is a linear classifier and it does not classify cases if they are not linearly separable. So, due to the inability of the perceptron to solve problems with linearly non-separable cases, the learning process will never reach the point with all cases properly classified.

11.2.10 Multilayer Perceptron Model

A multi-layer perceptron model uses the backpropagation algorithm. Though it has the same structure as that of a single-layer perceptron, it has one or more hidden layers.

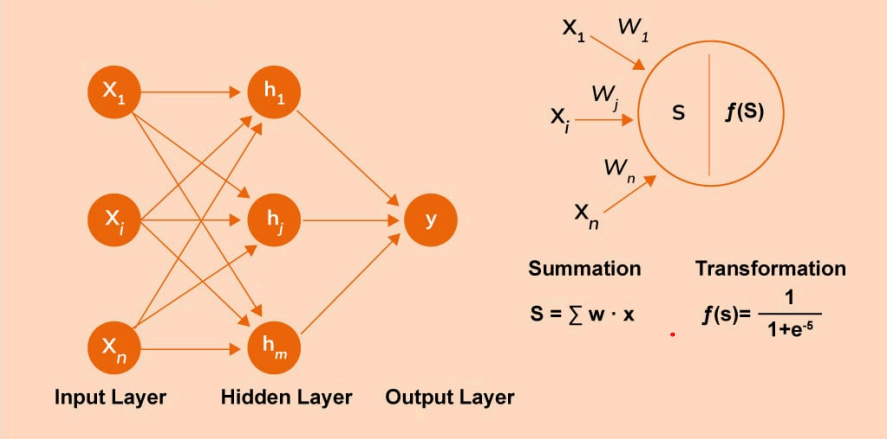


Figure 11.6: Multi Layer Perceptron

The backpropagation algorithm is executed in two phases:

Forward phase- Activation functions propagate from the input layer to the output layer. All weighted inputs are added to compute outputs using the sigmoid threshold.

Backward phase- The errors between the observed actual value and the demanded nominal value in the output layer are propagated backward. The weights and bias values are modified to achieve the requested value. The modification is done by apportioning the weights and bias to each unit according to its impact on the error.

- Error in any output neuron is $d_0 = y \times (1 - y) \times (t - y)$
- Error in any hidden neuron is $d_i = y_i \times (1 - y_i) \times (w_i - d_0)$
- Change the weights $\Delta w = \eta \times d \times X$

11.2.11 Perceptron Function

Perceptron function *f*(*x*) is generated by multiplying the input '*x*' with the learned weight coefficient '*w*'. The same can be expressed through the following mathematical equation:

$$f(x) = 1; \text{ if } w.x + b > 0$$

Otherwise, $f(x) = 0$

where *w* is the real-valued weight, *b* is the bias and *x* is a vector of input *x* values

11.2.12 Limitations of the Perceptron Model

A perceptron model has the following limitations

- The input vectors must be presented to the network one at a time or in batches so that the corrections can be made to the network based on the results of each presentation.
- The perceptron generates only a binary number (0 or 1) as an output due to the hard limit transfer function.
- It can classify linearly separable sets of inputs easily whereas non-linear input vectors cannot be classified properly.

11.3 Backpropagation

Backpropagation is a neural network learning algorithm. Roughly speaking, a neural network is a set of connected input/output units in which each connection has a weight associated with it. During the learning phase, the network learns by adjusting the weights so as to be able to predict the correct class label of the input tuples.

11.3.1 A Multilayer Feed-Forward Neural Network

The backpropagation algorithm performs learning on a multilayer feed-forward neural network. It iteratively learns a set of weights for prediction of the class label of tuples. A multilayer feed-forward neural network consists of an input layer, one or more hidden layers, and an output layer. An example of a multilayer feed-forward network is shown in Figure 11.7.

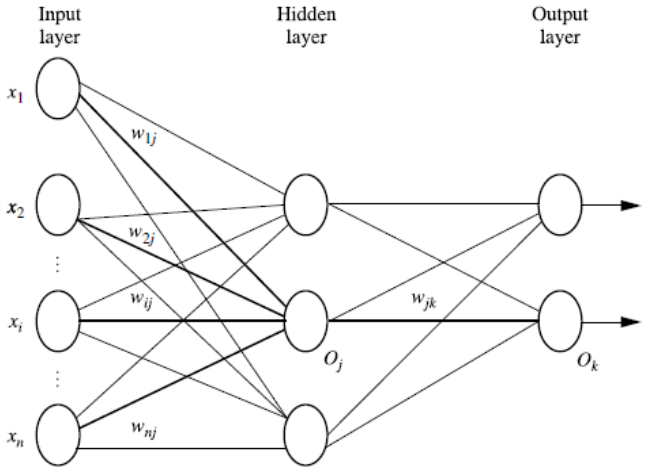


Figure 11.7: Multilayer feed-forward neural network

Each layer is made up of units. The inputs to the network correspond to the attributes measured for each training tuple. The inputs are fed simultaneously into the units making up the input layer. These inputs pass through the input layer and are then weighted and fed

simultaneously to a second layer of "neuronlike" units, known as a hidden layer. The outputs of the hidden layer units can be input to another hidden layer, and so on. The number of hidden layers is arbitrary, although in practice, usually only one is used. The weighted outputs of the last hidden layer are input to units making up the output layer, which emits the network's prediction for given tuples.

The units in the input layer are called input units. The units in the hidden layers and output layer are sometimes referred to as neurodes, due to their symbolic biological basis, or as output units. The multilayer neural network shown in Figure 11.7 has two layers of output units. Therefore, we say that it is a two-layer neural network. (The input layer is not counted because it serves only to pass the input values to the next layer.) Similarly, a network containing two hidden layers is called a three-layer neural network, and so on. It is a feed-forward network since none of the weights cycle back to an input unit or to a previous layer's output unit. It is fully connected in that each unit provides input to each unit in the next forward layer.

Each output unit takes, as input, a weighted sum of the outputs from units in the previous layer. It applies a nonlinear (activation) function to the weighted input. Multilayer feed-forward neural networks are able to model the class prediction as a nonlinear combination of the inputs. From a statistical point of view, they perform nonlinear regression. Multilayer feed-forward networks, given enough hidden units and enough training samples, can closely approximate any function.

Backpropagation is learned by iteratively processing a data set of training tuples, comparing the network's prediction for each tuple with the actual known target value. The target value may be the known class label of the training tuple (for classification problems) or a continuous value (for numeric prediction). For each training tuple, the weights are modified so as to minimize the mean-squared error between the network's prediction and the actual target value. These modifications are made in the "backwards" direction (i.e., from the output layer) through each hidden layer down to the first hidden layer (hence the name backpropagation). Although it is not guaranteed, in general the weights will eventually converge, and the learning process stops.

Initialize the weights: The weights in the network are initialized to small random numbers (e.g., ranging from -1.0 to 1.0, or -0.5 to 0.5). Each unit has a bias associated with it, as explained later. The biases are similarly initialized to small random numbers.

Each training tuple, X, is processed by the following steps.

Propagate the inputs forward: First, the training tuple is fed to the network's input layer. The inputs pass through the input units, unchanged. That is, for an input unit, j, its output, O_j , is equal to its input value, I_j . Next, the net input and output of each unit in the hidden and output layers are computed. The net input to a unit in the hidden or output layers is computed as a linear combination of its inputs. To help illustrate this point, a hidden layer or output layer unit

is shown in Figure 9.4. Each such unit has a number of inputs to it that are, in fact, the outputs of the units connected to it in the previous layer. Each connection has a weight. To compute the net input to the unit, each input connected to the unit is multiplied by its corresponding weight, and this is summed. Given a unit, j in a hidden or output layer, the net input, I_j , to unit *j* is

$$I_j = \sum_i w_{ij} O_i + \theta_j,$$

where w_{ij} is the weight of the connection from unit *i* in the previous layer to unit *j*; O_i is the output of unit *i* from the previous layer; and θ_j is the bias of the unit. The bias acts as a threshold in that it serves to vary the activity of the unit.

Each unit in the hidden and output layers takes its net input and then applies an activation function to it, as illustrated in Figure 9.4. The function symbolizes the activation of the neuron represented by the unit. The logistic, or sigmoid, function is used. Given the net input I_j to unit *j*, then O_j, the output of unit *j*, is computed as

$$O_j = \frac{1}{1 + e^{-I_j}}.$$

This function is also referred to as a squashing function, because it maps a large input domain onto the smaller range of 0 to 1. The logistic function is nonlinear and differentiable, allowing the backpropagation algorithm to model classification problems that are linearly inseparable.

We compute the output values, O_j, for each hidden layer, up to and including the output layer, which gives the network's prediction. In practice, it is a good idea to cache (i.e., save) the intermediate output values at each unit as they are required again later when backpropagating the error. This trick can substantially reduce the amount of computation required.

Backpropagate the error: The error is propagated backward by updating the weights and biases to reflect the error of the network's prediction. For a unit *j* in the output layer, the error Err_j is computed by

$$Err_j = O_j(1 - O_j)(T_j - O_j)$$

where O_j is the actual output of unit *j*, and T_j is the known target value of the given training tuple. Note that $O_j(1 - O_j)$ is the derivative of the logistic function. To compute the error of a hidden layer unit *j*, the weighted sum of the errors of the units connected to unit *j* in the next layer are considered. The error of a hidden layer unit *j* is

$$Err_j = O_j(1 - O_j) \sum_k Err_k w_{jk}$$

where w_{jk} is the weight of the connection from unit j to a unit k in the next higher layer, and Err_k is the error of unit k. The weights and biases are updated to reflect the propagated errors. Weights are updated by the following equations, where Δw_{ij} is the change in weight w_{ij}:

$$\Delta w_{ij} = (l) Err_j O_i.$$
$$w_{ij} = w_{ij} + \Delta w_{ij}.$$

The variable l is the learning rate, a constant typically having a value between 0.0 and 1.0. Backpropagation learns using a gradient descent method to search for a set of weights that fits the training data so as to minimize the mean-squared distance between the network's class prediction and the known target value of the tuples. The learning rate helps avoid getting stuck at a local minimum in decision space (i.e., where the weights appear to converge, but are not the optimum solution) and encourages finding the global minimum. If the learning rate is too small, then learning will occur at a very slow pace. If the learning rate is too large, then oscillation between inadequate solutions may occur. A rule of thumb is to set the learning rate to $1/t$, where t is the number of iterations through the training set so far.

Biases are updated by the following equations, where $\Delta\theta_j$ is the change in bias θ_j:

$$\Delta\theta_j = (l) Err_j.$$
$$\theta_j = \theta_j + \Delta\theta_j.$$

Note that here we are updating the weights and biases after the presentation of each tuple. This is referred to as case updating. Alternatively, the weight and bias increments could be accumulated in variables, so that the weights and biases are updated after all the tuples in the training set have been presented. This latter strategy is called epoch updating, where one iteration through the training set is an epoch. In theory, the mathematical derivation of backpropagation employs epoch updating, yet in practice, case updating is more common because it tends to yield more accurate results.

Terminating condition: Training stops when

- All Δw_{ij} in the previous epoch are so small as to be below some specified threshold, or
- The percentage of tuples misclassified in the previous epoch is below some threshold, or
- A prespecified number of epochs has expired.

Sample calculations for learning by the backpropagation algorithm. Figure 11.8 shows a multilayer feed-forward neural network. Let the learning rate be 0.9. The initial weight and bias values of the network are given in Table 11.1, along with the first training tuple, X = (1, 0, 1), with a class label of 1.

This example shows the calculations for backpropagation, given the first training tuple, X. The tuple is fed into the network, and the net input and output of each unit are computed. These

values are shown in Table 11.2. The error of each unit is computed and propagated backward. The error values are shown in Table 11.3. The weight and bias updates are shown in Table 11.4.

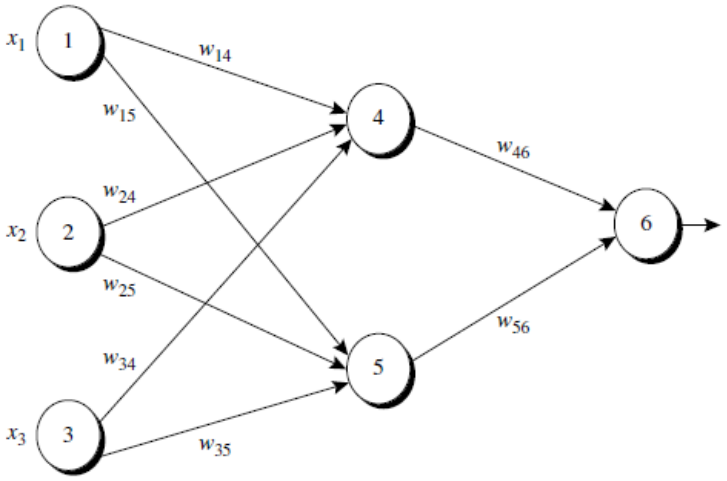


Figure 11.8: Example of a multilayer feed-forward neural network

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5	θ_6
1	0	1	0.2	−0.3	0.4	0.1	−0.5	0.2	−0.3	−0.2	−0.4	0.2	0.1

Table 11.1: Initial Input,Weight, and Bias Values

Unit, j	*Net Input, I_j*	*Output, O_j*
4	$0.2+0-0.5-0.4=-0.7$	$1/(1+e^{0.7})=0.332$
5	$-0.3+0+0.2+0.2=0.1$	$1/(1+e^{-0.1})=0.525$
6	$(-0.3)(0.332)-(0.2)(0.525)+0.1=-0.105$	$1/(1+e^{0.105})=0.474$

Table 11.2: Net Input and Output Calculations

Unit, j	*Err$_j$*
6	$(0.474)(1-0.474)(1-0.474)=0.1311$
5	$(0.525)(1-0.525)(0.1311)(-0.2)=-0.0065$
4	$(0.332)(1-0.332)(0.1311)(-0.3)=-0.0087$

Table 11.3: Calculation of the Error at Each Node

Weight or Bias	New Value
w_{46}	$-0.3 + (0.9)(0.1311)(0.332) = -0.261$
w_{56}	$-0.2 + (0.9)(0.1311)(0.525) = -0.138$
w_{14}	$0.2 + (0.9)(-0.0087)(1) = 0.192$
w_{15}	$-0.3 + (0.9)(-0.0065)(1) = -0.306$
w_{24}	$0.4 + (0.9)(-0.0087)(0) = 0.4$
w_{25}	$0.1 + (0.9)(-0.0065)(0) = 0.1$
w_{34}	$-0.5 + (0.9)(-0.0087)(1) = -0.508$
w_{35}	$0.2 + (0.9)(-0.0065)(1) = 0.194$
θ_6	$0.1 + (0.9)(0.1311) = 0.218$
θ_5	$0.2 + (0.9)(-0.0065) = 0.194$
θ_4	$-0.4 + (0.9)(-0.0087) = -0.408$

Table 11.4: Calculations forWeight and Bias Updating

To classify an unknown tuple, *X*, the tuple is input to the trained network, and the net input and output of each unit are computed. (There is no need for computation and/or backpropagation of the error.) If there is one output node per class, then the output node with the highest value determines the predicted class label for X. If there is only one output node, then output values greater than or equal to 0.5 may be considered as belonging to the positive class, while values less than 0.5 may be considered negative.

Chapter-12

Reinforcement Learning

Instructor Name: B N V Narasimha Raju

The best way to train your dog is by using a reward system. You give the dog a treat when it behaves well, and you punish it when it does something wrong. This same policy can be applied to machine learning models also. This type of machine learning method, where we use a reward system to train our model, is called Reinforcement Learning.

12.1 Need for Reinforcement Learning

A major drawback of machine learning is that a tremendous amount of data is needed to train models. The more complex a model, the more data it may require. But this data may not be available to us. It may not exist or we simply may not have access to it. Further, the data collected might not be reliable. It may have false or missing values or it might be outdated.

Also, learning from a small subset of actions will not help expand the vast realm of solutions that may work for a particular problem. This is going to slow the growth that technology is capable of. Machines need to learn to perform actions by themselves and not just learn from humans.

All of these problems are overcome by reinforcement learning. In reinforcement learning, we introduce our model to a controlled environment which is modeled after the problem statement to be solved instead of using actual data to solve it.

12.2 Reinforcement Learning

Reinforcement learning is a sub-branch of Machine Learning that trains a model to return an optimum solution for a problem by taking a sequence of decisions by itself.

Reinforcement learning is about taking suitable action to maximize reward in a particular situation. It is employed by various software and machines to find the best possible behavior or path it should take in a specific situation. Reinforcement learning differs from supervised learning in a way that in supervised learning the training data has the answer key with it so the model is trained with the correct answer itself whereas in reinforcement learning, there is no answer but the reinforcement agent decides what to do to perform the given task. In the absence of a training dataset, it is bound to learn from its experience.

Reinforcement learning uses algorithms that learn from outcomes and decide which action to take next. After each action, the algorithm receives feedback that helps it determine whether the choice it made was correct, neutral or incorrect. It is a good technique to use for automated systems that have to make a lot of small decisions without human guidance.

Reinforcement learning is an autonomous, self-teaching system that essentially learns by trial and error. It performs actions with the aim of maximizing rewards, or in other words, it is learning by doing in order to achieve the best outcomes.

We model an environment after the problem statement. The model interacts with this environment and comes up with solutions all on its own, without human interference. To push it in the right direction, we simply give it a positive reward if it performs an action that brings it closer to its goal or a negative reward if it goes away from its goal.

To understand reinforcement learning better, consider a dog that we have to house train. Here, the dog is the agent and the house, the environment as shown in figure 12.1

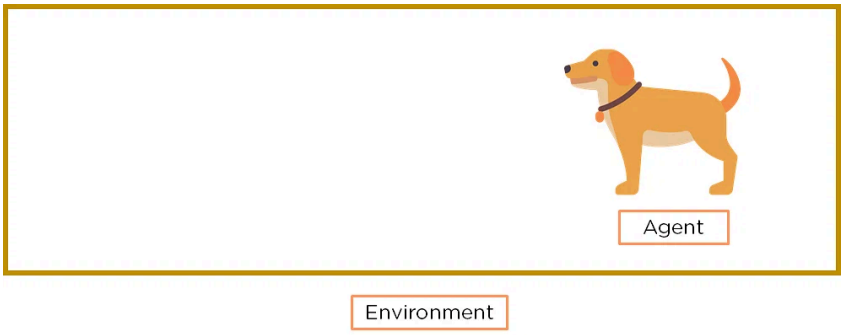


Figure 12.1: Agent and Environment

We can get the dog to perform various actions by offering incentives such as dog biscuits as a reward as shown in figure 12.2

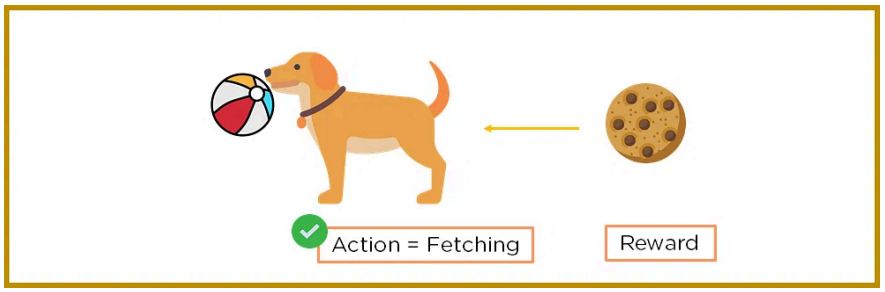


Figure 12.2: Performing an Action and getting Reward

The dog will follow a policy to maximize its reward and hence will follow every command and might even learn a new action, like begging, all by itself as shown in figure 12.3.

Figure 12.3: Learning new actions

The dog will also want to run around and play and explore its environment. This quality of a model is called Exploration. The tendency of the dog to maximize rewards is called Exploitation. There is always a tradeoff between exploration and exploitation, as exploration actions may lead to lesser rewards as shown in figure 12.4.

Figure 12.4: Exploration vs Exploitation

12.2.1 Supervised vs Unsupervised vs Reinforcement Learning

Supervised Learning	Unsupervised Learning	Reinforcement Learning
Data provided is labeled data, with output values specified	Data provided is unlabeled data, the outputs are not specified, machine makes its own prediction	The machine learns from its environment using rewards and errors
Used to solve regression and classification problems	Used to solve Association and clustering problems	Used to solve reward based problems
Labeled data is used	Unlabeled data is used	No predefined data is used
External Supervision	No supervision	No supervision
Solves problems by mapping labeled input to known output	Solves problem by understanding patterns and discovering output	Follows Trail and Error problem solving approach

12.2.2 Important Terms in Reinforcement Learning

- Agent: Agent is the model that is being trained via reinforcement learning

- Environment: The training situation that the model must optimize to is called its environment
- Action: All possible steps that can be taken by the model
- State: The current position/ condition returned by the model
- Reward: To help the model move in the right direction, it is rewarded/points are given to it to appraise some action
- Policy: Policy determines how an agent will behave at any time. It acts as a mapping between Action and present State.

The important terms in Reinforcement Learning is shown in figure 12.5

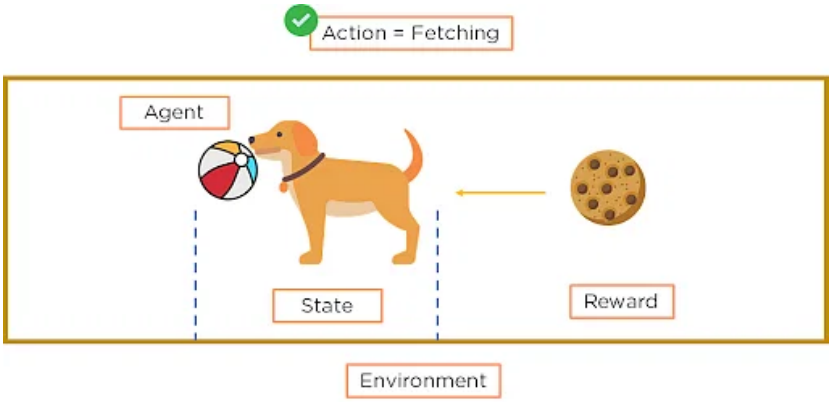


Figure 12.5: Important terms in Reinforcement Learning

There are two types of Reinforcement:

Positive: Positive Reinforcement is defined as when an event, occurs due to a particular behavior, increases the strength and the frequency of the behavior. In other words, it has a positive effect on behavior.

Negative: Negative Reinforcement is defined as strengthening of behavior because a negative condition is stopped or avoided.

12.3 Markov's Decision Process

Markov's Decision Process is a Reinforcement Learning policy used to map a current state to an action where the agent continuously interacts with the environment to produce new solutions and receive rewards as shown in figure 12.6

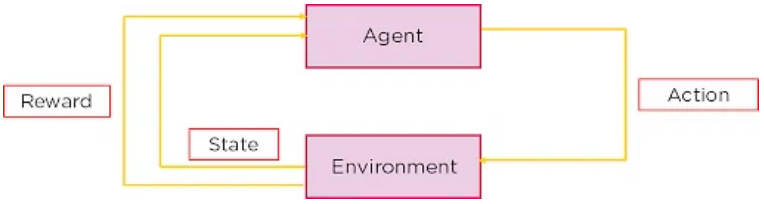


Figure 12.6: Markov's Decision Process

First, let's understand Markov's Process. Markov's Process states that the future is independent of the past, given the present. This means that, given the present state, the next state can be predicted easily, without the need for the previous state.

This theory is used by Markov's Decision Process to get the next action in our machine learning model. Markov's Decision Process (MDP) uses:

- A set of States (S)
- A set of Models
- A set of all possible actions (A)
- A reward function that depends on the state and action R(S, A)
- A policy which is the solution of MDP

State: A State is a set of tokens that represent every state that the agent can be in.

Model: A Model (sometimes called Transition Model) gives an action's effect in a state. In particular, T(S, a, S') defines a transition T where being in state S and taking an action 'a' takes us to state S' (S and S' may be the same). For stochastic actions (noisy, non-deterministic) we also define a probability P(S'|S, a) which represents the probability of reaching a state S' if action 'a' is taken in state S. Markov property states that the effects of an action taken in a state depend only on that state and not on the prior history.

Action: An Action A is a set of all possible actions. A(S) defines the set of actions that can be taken being in state S.

Reward: A Reward is a real-valued reward function. R(S) indicates the reward for simply being in the state S. R(S,a) indicates the reward for being in a state S and taking an action 'a'. R(S,a,S') indicates the reward for being in a state S, taking an action 'a' and ending up in a state S'.

Policy: A Policy is a solution to the Markov Decision Process. A policy is a mapping from S to a. It indicates the action 'a' to be taken while in state S.

Let us take the example of a grid world:

The policy of Markov's Decision Process aims to maximize the reward at each state. The Agent interacts with the Environment and takes Action while it is at one State to reach the next future State. We base the action on the maximum Reward returned.

In the figure 12.7 shown, we need to find the shortest path between node A and D. Each path has a reward associated with it, and the path with maximum reward is what we want to choose. The nodes; A, B, C, D; denote the nodes. To travel from node to node (A to B) is an action. The reward is the cost at each path, and policy is each path taken.

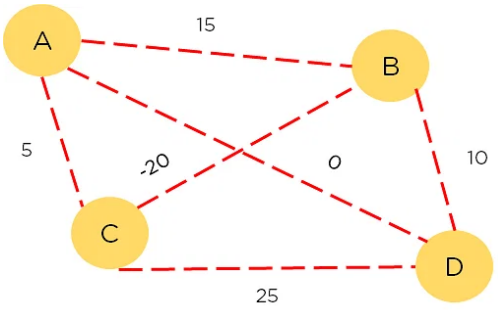


Figure 12.7: Nodes to traverse

The process will maximize the output based on the reward at each step and will traverse the path with the highest reward. This process does not explore but maximizes reward as shown in figure 12.8.

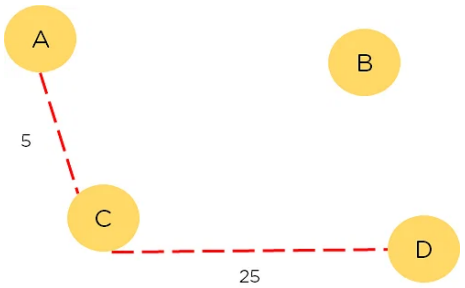


Figure 12.8: Path taken by MDP

12.4 Q-Learning

Q-Learning is a Reinforcement learning policy that will find the next best action, given a current state. It chooses this action at random and aims to maximize the reward as shown in figure 12.9.

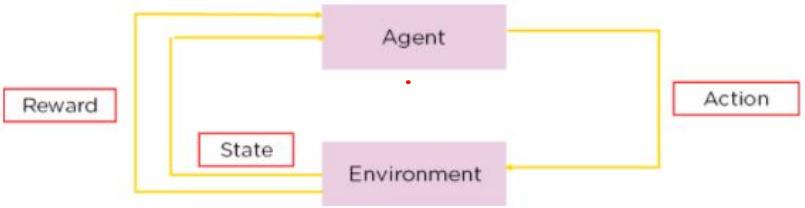


Figure 12.9: Components of Q-Learning

Q-learning is a model-free, off-policy reinforcement learning that will find the best course of action, given the current state of the agent. Depending on where the agent is in the environment, it will decide the next action to be taken.

The objective of the model is to find the best course of action given its current state. To do this, it may come up with rules of its own or it may operate outside the policy given to it to follow. This means that there is no actual need for a policy, hence we call it off-policy.

Model-free means that the agent uses predictions of the environment's expected response to move forward. It does not use the reward system to learn, but rather, trial and error.

An example of Q-learning is an Advertisement recommendation system. In a normal ad recommendation system, the ads you get are based on your previous purchases or websites you may have visited. If you've bought a TV, you will get recommended TVs of different brands as shown in figure 12.10.

Figure 12.10: Ad Recommendation System

Using Q-learning, we can optimize the ad recommendation system to recommend products that are frequently bought together. The reward will be if the user clicks on the suggested product as shown in figure 12.11.

Figure 12.11: Ad Recommendation System with Q-Learning

12.4.1 Important Terms in Q-Learning

- States: The State, S, represents the current position of an agent in an environment.
- Action: The Action, A, is the step taken by the agent when it is in a particular state.
- Rewards: For every action, the agent will get a positive or negative reward.
- Episodes: When an agent ends up in a terminating state and can't take a new action.
- Q-Values: Used to determine how good an Action, A, taken at a particular state, S, is. Q (A, S).
- Temporal Difference: A formula used to find the Q-Value by using the value of current state and action and previous state and action.

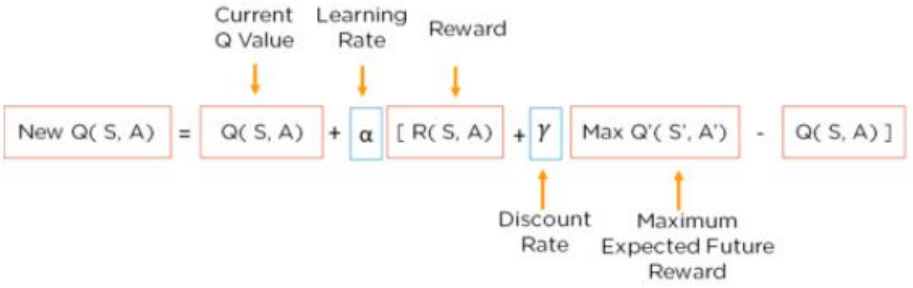
Q-Values or Action-Values: Q-values are defined for states and actions. Q(S, A) is an estimation of how good it is to take the action A at the state S . This estimation of Q(S, A) will be iteratively computed using the TD- Update rule

Rewards and Episodes: An agent over the course of its lifetime starts from a start state, makes a number of transitions from its current state to a next state based on its choice of action and also the environment the agent is interacting in. At every step of transition, the agent from a state takes an action, observes a reward from the environment, and then transits to another state. If at any point of time the agent ends up in one of the terminating states that means there are no further transition possible. This is said to be the completion of an episode.

Temporal Difference or TD-Update: The Temporal Difference or TD-Update rule can be represented by the Bellman equation.

The Bellman Equation is used to determine the value of a particular state and deduce how good it is to be in/take that state. The optimal state will give us the highest optimal value.

The equation is given below. It uses the current state, and the reward associated with that state, along with the maximum expected reward and a discount rate, which determines its importance to the current state, to find the next state of our agent. The learning rate determines how fast or slow the model will be learning.



This update rule to estimate the value of Q is applied at every time step of the agent's interaction with the environment. The terms used are explained below. :

- S : Current State of the agent.
- A : Current Action Picked according to some policy.
- S' : Next State where the agent ends up.
- A' : Next best action to be picked using current Q-value estimation, i.e. pick the action with the maximum Q-value in the next state.
- R : Current Reward observed from the environment in Response of current action.
- γ (>0 and <=1) : Discounting Factor for Future Rewards. Future rewards are less valuable than current rewards so they must be discounted. Since Q-value is an estimation of expected rewards from a state, the discounting rule applies here as well.
- α : It is a learning rate, step length taken to update the estimation of Q(S, A).

12.4.2 Q-Table

While running our algorithm, we will come across various solutions and the agent will take multiple paths. How do we find out the best among them? This is done by tabulating our findings in a table called a Q-Table.

A Q-Table helps us to find the best action for each state in the environment. We use the Bellman Equation at each state to get the expected future state and reward and save it in a table to compare with other states.

Let us create a q-table for an agent that has to learn to run, fetch and sit on command. The steps taken to construct a q-table are :

Step 1: Create an initial Q-Table with all values initialized to 0

When we initially start, the values of all states and rewards will be 0. Consider the Q-Table shown below which shows a dog simulator learning to perform actions as shown in figure 12.12.

Action	Fetching	Sitting	Running
Start	0	0	0
Idle	0	0	0
Wrong Action	0	0	0
Correct Action	0	0	0
End	0	0	0

Figure 12.12: Initial Q-Table

Step 2: Choose an action and perform it. Update values in the table

This is the starting point. We have performed no other action as of yet. Let us say that we want the agent to sit initially, which it does. The table will change to as shown in figure 12.13.

Action	Fetching	Sitting	Running
Start	0	1	0
Idle	0	0	0
Wrong Action	0	0	0
Correct Action	0	0	0
End	0	0	0

Figure 12.13: Q-Table after performing an action

Step 3: Get the value of the reward and calculate the value Q-Value using Bellman Equation

For the action performed, we need to calculate the value of the actual reward and the Q(S, A) value as shown in figure 12.14

Action	Fetching	Sitting	Running
Start	0	1	0
Idle	0	0	0
Wrong Action	0	0	0
Correct Action	0	34	0
End	0	0	0

Figure 12.14: Updating Q-Table with Bellman Equation

Step 4: Continue the same until the table is filled or an episode ends

The agent continues taking actions and for each action, the reward and Q-value are calculated and it updates the table as shown in figure 12.15.

Action	Fetching	Sitting	Running
Start	5	7	10
Idle	2	5	3
Wrong Action	2	6	1
Correct Action	54	34	17
End	3	1	4

Figure 12.15: Final Q-Table at end of an episode

12.5 Gradient descent reduces the cost function

After training your model, you need to see how well your model is performing. While accuracy functions tell you how well the model is performing, they do not provide you with an insight on how to better improve them. Hence, you need a correctional function that can help you compute when the model is the most accurate, as you need to hit that small spot between an undertrained model and an overtrained model.

A Cost Function is used to measure just how wrong the model is in finding a relation between the input and output. It tells you how badly your model is behaving/predicting

Consider a robot trained to stack boxes in a factory. The robot might have to consider certain changeable parameters, called Variables, which influence how it performs. Let's say the robot comes across an obstacle, like a rock. The robot might bump into the rock and realize that it is not the correct action.

It will learn from this, and next time it will learn to avoid rocks. Hence, your machine uses variables to better fit the data. The outcome of all these obstacles will further optimize the robot and help it perform better. It will generalize and learn to avoid obstacles in general, say like a fire that might have broken out. The outcome acts as a cost function, which helps you optimize the variable, to get the best variables and fit for the model.

12.5.1 Gradient Descent

Gradient Descent is an algorithm that is used to optimize the cost function or the error of the model. It is used to find the minimum value of error possible in your model. Gradient Descent can be thought of as the direction you have to take to reach the least possible error. The error in your model can be different at different points, and you have to find the quickest way to minimize it, to prevent resource wastage.

Gradient Descent can be visualized as a ball rolling down a hill. Here, the ball will roll to the lowest point on the hill. It can take this point as the point where the error is least as for any model, the error will be minimum at one point and will increase again after that.

In gradient descent, you find the error in your model for different values of input variables. This is repeated, and soon you see that the error values keep getting smaller and smaller. Soon you'll arrive at the values for variables when the error is the least, and the cost function is optimized as shown in figure 12.16.

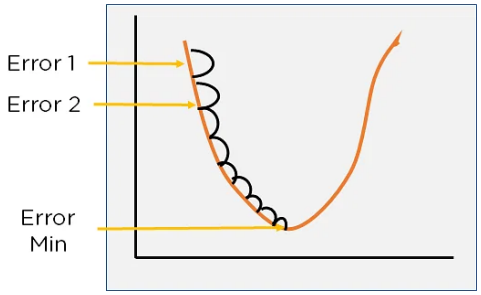


Figure 12.16: Gradient Descent

12.5.2 Cost Function For Linear Regression

A Linear Regression model uses a straight line to fit the model. This is done using the equation for a straight line as shown :

$$Output = a * Input + b$$

In the equation, you can see that two entities can have changeable values (variable) a, which is the point at which the line intercepts the x-axis, and b, which is how steep the line will be, or slope.

At first, if the variables are not properly optimized, you get a line that might not properly fit the model. As you optimize the values of the model, for some variables, you will get the perfect fit.

The perfect fit will be a straight line running through most of the data points while ignoring the noise and outliers. A properly fit Linear Regression model looks as shown in figure 12.17.

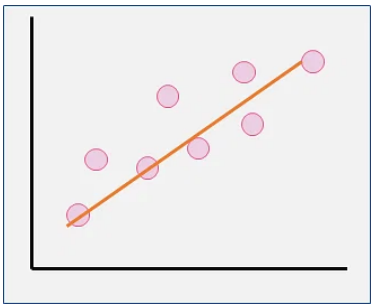


Figure 12.17: Linear regression graph

For the Linear regression model, the cost function will be the minimum of the Root Mean Squared Error of the model, obtained by subtracting the predicted values from actual values. The cost function will be the minimum of these error values.

$$\text{Cost Function (J)} = \frac{1}{n}\sum_{i=0}^{n}(h_{\theta}(x^i) - y^i)^2$$

By the definition of gradient descent, you have to find the direction in which the error decreases constantly. This can be done by finding the difference between errors. The small difference between errors can be obtained by differentiating the cost function and subtracting it from the previous gradient descent to move down the slope.

$$\text{Gradient Descent } \theta_j = \theta_j - \alpha \frac{\partial J}{\partial \theta}$$

After substituting the value of the cost function (J) in the above equation, a simplified Linear regression gradient descent function is obtained.

$$\text{Gradient Descent} = \theta_j - \frac{\alpha}{n}\sum_{i=0}^{n}((h_{\theta}(x^i) - y^i)x^i)^2$$

In the above equations, α is known as the learning rate. It decides how fast you move down the slope. If alpha is large, you take big steps, and if it is small; you take small steps. If alpha is too large, you can entirely miss the least error point and our results will not be accurate. If it is too small it will take too long to optimize the model and you will also waste computational power. Hence you need to choose an optimal value of alpha as shown in figure 12.18.

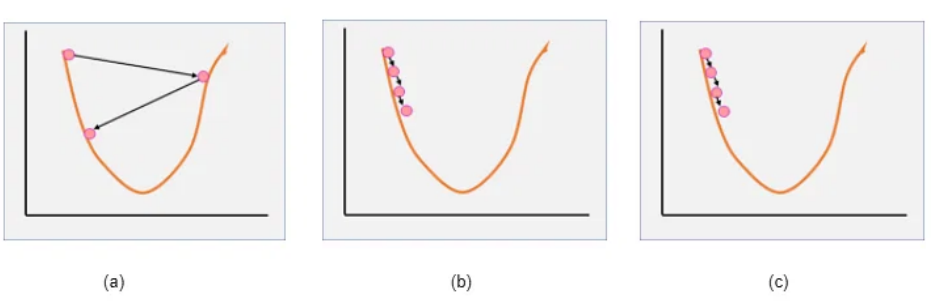


Figure 12.18: (a) Large learning rate, (b) Small learning rate, (c) Optimum learning rate

12.5.3 Cost Function for Neural Networks

A neural network is a machine learning algorithm that takes in multiple inputs, runs them through an algorithm, and essentially sums the output of the different algorithms to get the final output. The cost function of a neural network will be the sum of errors in each layer. This is done by finding the error at each layer first and then summing the individual error to get the total error. In the end, it can represent a neural network with cost function optimization as shown in figure 12.19.

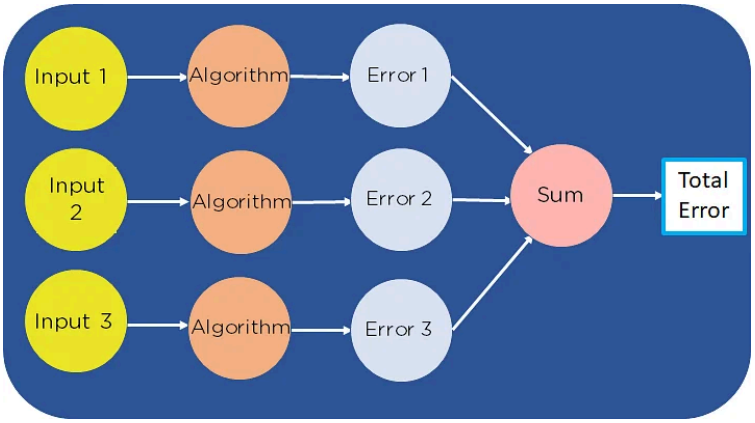


Figure 12.19: Neural network with the error function

For neural networks, each layer will have a cost function, and each cost function will have its own least minimum error value. Depending on where you start, you can arrive at a unique value for the minimum error. You need to find the minimum value out of all local minima. This value is called the global minima as shown in figure 12.20.

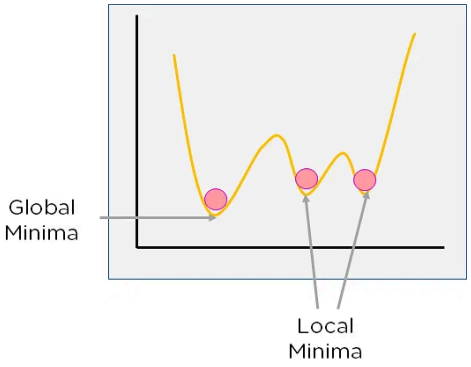


Figure 12.20: Cost function graph for Neural Networks

The cost function for neural networks is given as

$$\text{Cost Function (J)} = \frac{1}{n}\sum_{i=0}^{n}(y^i - (mx^i + b))^2$$

Gradient descent is just the differentiation of the cost function. It is given as

$$\text{Gradient Descent } (\frac{\partial J}{\partial \theta}) = \begin{bmatrix} \frac{1}{N}\sum_{i=0}^{n}(-2\, x_i(y_i - (mx_i + b)) \\ \frac{1}{N}\sum_{i=0}^{n}(-2\,(y_i - (mx_i + b)) \end{bmatrix}$$