UNIT I INTRODUCTION 9

Human Learning - Types – Machine Learning - Types - Problems not to be solved - Applications - Languages/Tools– Issues. Preparing to Model: Introduction - Machine Learning Activities - Types of data - Exploring structure of data - Data quality and remediation - Data Pre-processing

1. Human Learning – Types

LEARNING:

Learning is a change in behavior or in potential behavior that occurs as a result of experience. Learning occurs most rapidly on a schedule of continuous reinforcement. However it is fairly easy to extinguish… switching to variable reinforcement after the desired behavior has been reached prevents extinction.

BEHAVIORISM

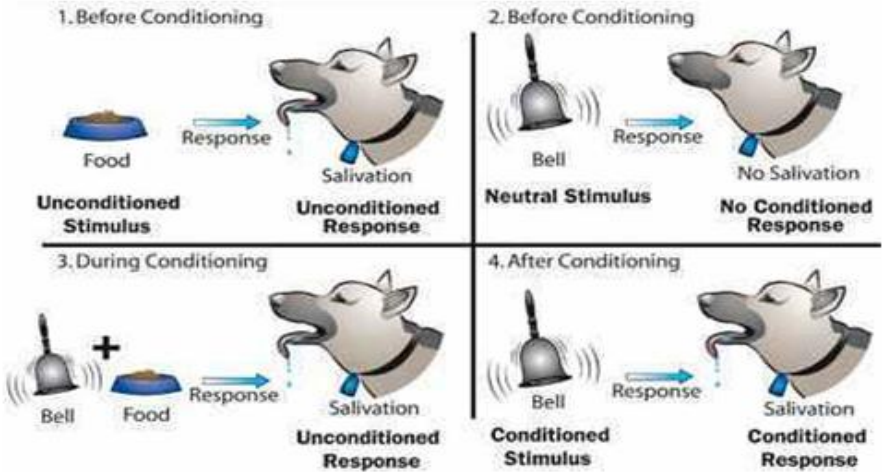
Is a theory of learning based upon the idea that all behaviors are acquired through conditioning. Conditioning occurs through interaction with the environment. - also known as Behavioral Psychology. Learning: acquiring new, or modifying and reinforcing, existing knowledge, behavior, skills, values or preferences and may involve synthesizing different types of information.

Learning is a process of progressive behaviour adaptation. –B.F Skinner

- Learning by association or Classical Conditioning
- Learning by consequences or Operant Conditioning.
- Learning through Observation or Modeling

Pavlov: Classical Conditioning

- Classical conditioning is a reflexive or automatic type of learning in which a stimulus acquires the capacity to evoke a response that was originally evoked by another stimulus.
- Classical conditioning is a form of learning whereby a conditioned stimulus becomes associated with an unrelated unconditioned stimulus, in order to produce a behavioral response known as a conditioned response.



Five key principles of classical conditioning:

1. Acquisition

Acquisition is the initial stage of learning when a response is first established and gradually strengthened. During the acquisition phase of classical conditioning, a neutral stimulus is repeatedly paired with an unconditioned stimulus. As you may recall, an unconditioned stimulus is something that naturally and automatically triggers a response without any learning. After an association is made, the subject will begin to emit a behavior in response to the previously neutral stimulus, which is now known as a conditioned stimulus. It is at this point that we can say that the response has been acquired.

2. Extinction

Extinction is when the occurrences of a conditioned response decrease or disappear. In classical conditioning, this happens when a conditioned stimulus is no longer paired with an unconditioned stimulus.

3. Spontaneous Recovery

Sometimes a learned response can suddenly reemerge even after a period of extinction. Spontaneous Recovery is the reappearance of the conditioned response after a rest period or period of lessened response. For example, imagine that after training a dog to salivate to the sound of a bell, you stop reinforcing the behavior and the response eventually becomes extinct. After a rest period during which the conditioned stimulus is not presented, you suddenly ring the bell and the animal spontaneously recovers the previously learned response.

4. Stimulus Generalization

Stimulus Generalization is the tendency for the conditioned stimulus to evoke similar responses after the response has been conditioned. For example, if a dog has been conditioned to salivate at the sound of a bell, the animal may also exhibit the same response to stimuli that are similar to the conditioned stimulus. In John B. Watson's famous Little Albert Experiment, for example, a small child was conditioned to fear a white rat. The child demonstrated stimulus generalization by also exhibiting fear in response to other fuzzy white objects including stuffed toys and Watson own hair.

5. Stimulus Discrimination

Discrimination is the ability to differentiate between a conditioned stimulus and other stimuli that have not been paired with an unconditioned stimulus. For example, if a bell tone were the conditioned stimulus, discrimination would involve being able to tell the difference between the bell tone and other similar sounds. Because the subject is able to distinguish between these stimuli, he or she will only respond when the conditioned stimulus is presented.

Operant Conditioning

Operant conditioning can be described as a process that attempts to modify behavior through the use of positive and negative reinforcement. Through operant conditioning, an individual makes an association between a particular behavior and a consequence.

- Example 1: Parents rewarding a child's excellent grades with candy or some other prize.
- Example 2: A schoolteacher awards points to those students who are the most calm and well-behaved. Students eventually realize that when they voluntarily become quieter and better behaved, that they earn more points.
- Example 3: A form of reinforcement (such as food) is given to an animal every time the animal (for example, a hungry lion) presses a lever.

The term "operant conditioning" originated by the behaviorist B. F. Skinner, who believed that one should focus on the external, observable causes of behavior (rather than try to unpack the internal thoughts and motivations)

Reinforcement comes in two forms: positive and negative.

Positive and negative reinforcers

- Positive reinforcers are favorable events or outcomes that are given to the individual after the desired behavior. This may come in the form of praise, rewards, etc.
- Negative reinforcers typically are characterized by the removal of an undesired or unpleasant outcome after the desired behavior. A response is strengthened as something considered negative is removed.

Four Important Principles in Operant Conditioning

- **Principle of Immediacy:** Verbal immediacy refers to calling on by the students or asks students how they feel about things. Non-verbal immediacy includes behaviors such as smiling, gesturing, moves around the class while teaching and having relaxed body language.
- **Principle of Deprivation/Satiation:**

  Deprivation: Not having access to something that is
  Deprivation: Not having access to something that is highly desirable.

  Satiation - is the opposite of deprivation -refers to having too much

- **Principle of Contingency:** a future event or circumstance that is possible but cannot be predicted with certainty.
- **Principle of Size:** The cost-benefit" determinant of whether a consequence will be effective. If the size, or amount, of the consequence is large enough to be worth the effort, the consequence will be more effective upon the behavior.

<u>Machine Learning</u>

Definition of Machine Learning: Arthur Samuel, an early American leader in the field of computer gaming and artificial intelligence, coined the term "Machine Learning " in 1959 while at IBM.

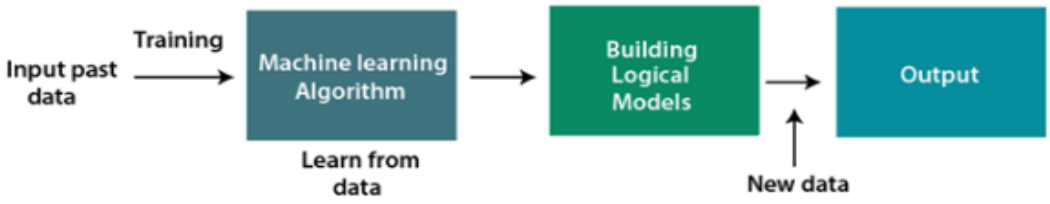
- Machine learning is programming computers to optimize a performance criterion using example data or past experience . We have a model defined up to some parameters, and learning is the execution of a computer program to optimize the parameters of the model using the training data or past experience. The model may be predictive to make predictions in the future, or descriptive to gain knowledge from data.

- The field of study known as machine learning is concerned with the question of how to construct computer programs that automatically improve with experience.

How does Machine Learning work

A Machine Learning system learns from historical data, builds the prediction models, and whenever it receives new data, predicts the output for it. The accuracy of predicted output depends upon the amount of data, as the huge amount of data helps to build a better model which predicts the output more accurately.

Suppose we have a complex problem, where we need to perform some predictions, so instead of writing a code for it, we just need to feed the data to generic algorithms, and with the help of these algorithms, machine builds the logic as per the data and predict the output. Machine learning has changed our way of thinking about the problem. The below block diagram explains the working of Machine Learning algorithm:



Features of Machine Learning:

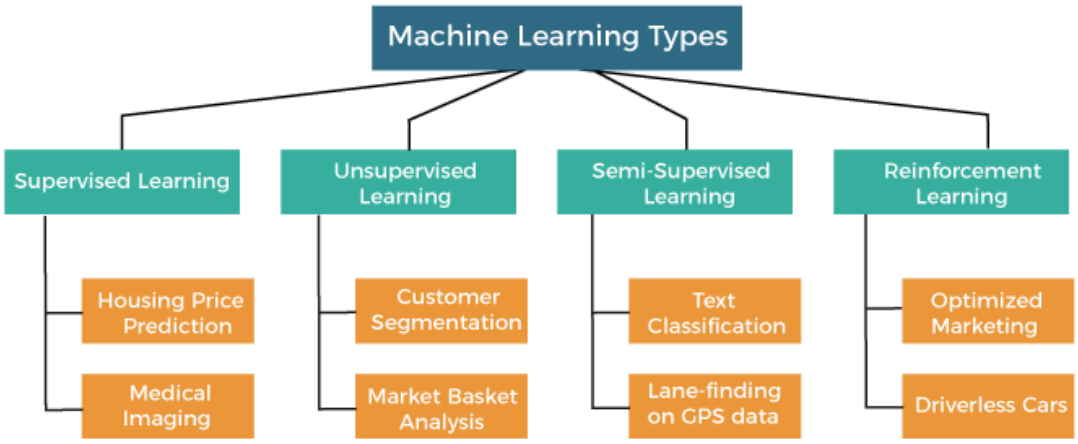
- Machine learning uses data to detect various patterns in a given dataset.
- It can learn from past data and improve automatically.
- It is a data-driven technology.
- Machine learning is much similar to data mining as it also deals with the huge amount of the data.

These ML algorithms help to solve different business problems like Regression, Classification, Forecasting, Clustering, and Associations, etc.

Based on the methods and way of learning, machine learning is divided into mainly four types, which are:

Types:

1. **Supervised Machine Learning**
2. **Unsupervised Machine Learning**
3. **Semi-Supervised Machine Learning**
4. **Reinforcement Learning**



1. Supervised Machine Learning

As its name suggests, Supervised machine learning is based on supervision. It means in the supervised learning technique, we train the machines using the "labelled" dataset, and based on the training, the machine predicts the output. Here, the labelled data specifies that some of the inputs are already mapped to the output. More preciously, we can say; first, we train the machine with the input and corresponding output, and then we ask the machine to predict the output using the test dataset.

Let's understand supervised learning with an example. Suppose we have an input dataset of cats and dog images. So, first, we will provide the training to the machine to understand the images, such as the shape & size of the tail of cat and dog, Shape of eyes, colour, height (dogs are taller, cats are smaller), etc. After completion of training, we input the picture of a cat and ask the machine to identify the object and predict the output. Now, the machine is well trained, so it will check all the features of the object, such as height, shape, colour, eyes, ears, tail, etc., and find that it's a cat. So, it will put it in the Cat category. This is the process of how the machine identifies the objects in Supervised Learning.

The main goal of the supervised learning technique is to map the input variable(x) with the output variable(y). Some real-world applications of supervised learning are Risk Assessment, Fraud Detection, Spam filtering, etc.

Categories of Supervised Machine Learning

Supervised machine learning can be classified into two types of problems, which are given below:

- **Classification**
- **Regression**

a) Classification

Classification algorithms are used to solve the classification problems in which the output variable is categorical, such as "Yes" or No, Male or Female, Red or Blue, etc. The classification algorithms predict the categories present in the dataset. Some real-world examples of classification algorithms are Spam Detection, Email filtering, etc.

Some popular classification algorithms are given below:

- Random Forest Algorithm
- Decision Tree Algorithm
- Logistic Regression Algorithm
- Support Vector Machine Algorithm

b) Regression

Regression algorithms are used to solve regression problems in which there is a linear relationship between input and output variables. These are used to predict continuous output variables, such as market trends, weather prediction, etc.
Some popular Regression algorithms are given below:

- Simple Linear Regression Algorithm
- Multivariate Regression Algorithm
- Decision Tree Algorithm
- Lasso Regression

Advantages and Disadvantages of Supervised Learning

Advantages:

- Since supervised learning work with the labelled dataset so we can have an exact idea about the classes of objects.
- These algorithms are helpful in predicting the output on the basis of prior experience.

Disadvantages:

- These algorithms are not able to solve complex tasks.
- It may predict the wrong output if the test data is different from the training data.
- It requires lots of computational time to train the algorithm.

2. Unsupervised Machine Learning

Unsupervised learning is different from the Supervised learning technique; as its name suggests, there is no need for supervision. It means, in unsupervised machine learning, the machine is trained using the unlabeled dataset, and the machine predicts the output without any supervision.
In unsupervised learning, the models are trained with the data that is neither classified nor labelled, and the model acts on that data without any supervision.
The main aim of the unsupervised learning algorithm is to group or categories the unsorted dataset according to the similarities, patterns, and differences. Machines are instructed to find the hidden patterns from the input dataset.
Let's take an example to understand it more preciously; suppose there is a basket of fruit images, and we input it into the machine learning model. The images are totally unknown to the model, and the task of the machine is to find the patterns and categories of the objects.

So, now the machine will discover its patterns and differences, such as colour difference, shape difference, and predict the output when it is tested with the test dataset.

Categories of Unsupervised Machine Learning

Unsupervised Learning can be further classified into two types, which are given below:

- **Clustering**
- **Association**

1) Clustering

The clustering technique is used when we want to find the inherent groups from the data. It is a way to group the objects into a cluster such that the objects with the most similarities remain in one group and have fewer or no similarities with the objects of other groups. An example of the clustering algorithm is grouping the customers by their purchasing behaviour.

Some of the popular clustering algorithms are given below:

- K-Means Clustering algorithm
- Mean-shift algorithm
- DBSCAN Algorithm
- Principal Component Analysis
- Independent Component Analysis

2) Association

Association rule learning is an unsupervised learning technique, which finds interesting relations among variables within a large dataset. The main aim of this learning algorithm is to find the dependency of one data item on another data item and map those variables accordingly so that it can generate maximum profit. This algorithm is mainly applied in Market Basket analysis, Web usage mining, continuous production, etc.

Some popular algorithms of Association rule learning are Apriori Algorithm, Eclat, FP-growth algorithm.

Advantages and Disadvantages of Unsupervised Learning Algorithm

Advantages:

- These algorithms can be used for complicated tasks compared to the supervised ones because these algorithms work on the unlabeled dataset.
- Unsupervised algorithms are preferable for various tasks as getting the unlabeled dataset is easier as compared to the labelled dataset.

Disadvantages:

- The output of an unsupervised algorithm can be less accurate as the dataset is not labelled, and algorithms are not trained with the exact output in prior.
- Working with Unsupervised learning is more difficult as it works with the unlabelled dataset that does not map with the output.

Applications of Unsupervised Learning

- Network Analysis: Unsupervised learning is used for identifying plagiarism and copyright in document network analysis of text data for scholarly articles.
- Recommendation Systems: Recommendation systems widely use unsupervised learning techniques for building recommendation applications for different web applications and e-commerce websites.
- Anomaly Detection: Anomaly detection is a popular application of unsupervised learning, which can identify unusual data points within the dataset. It is used to discover fraudulent transactions.
- Singular Value Decomposition: Singular Value Decomposition or SVD is used to extract particular information from the database. For example, extracting information of each user located at a particular location.

3. Semi-Supervised Learning

Semi-Supervised learning is a type of Machine Learning algorithm that lies between Supervised and Unsupervised machine learning. It represents the intermediate ground between Supervised (With Labelled training data) and Unsupervised learning (with no labelled training data) algorithms and uses the combination of labelled and unlabeled datasets during the training period.

Although Semi-supervised learning is the middle ground between supervised and unsupervised learning and operates on the data that consists of a few labels, it mostly consists of unlabeled data. As labels are costly, but for corporate purposes, they may have few labels. It is completely different from supervised and unsupervised learning as they are based on the presence & absence of labels.

To overcome the drawbacks of supervised learning and unsupervised learning algorithms, the concept of Semi-supervised learning is introduced. The main aim of semi-supervised learning is to effectively use all the available data, rather than only labelled data like in supervised learning. Initially, similar data is clustered along with an unsupervised learning algorithm, and further, it helps to label the unlabeled data into labelled data. It is because labelled data is a comparatively more expensive acquisition than unlabeled data.

We can imagine these algorithms with an example. Supervised learning is where a student is under the supervision of an instructor at home and college. Further, if that student is self-analysing the same concept without any help from the instructor, it comes under unsupervised learning. Under semi-supervised learning, the student has to revise himself after analyzing the same concept under the guidance of an instructor at college.

Advantages and disadvantages of Semi-supervised Learning

Advantages:

- It is simple and easy to understand the algorithm.
- It is highly efficient.
- It is used to solve drawbacks of Supervised and Unsupervised Learning algorithms.

Disadvantages:

- Iterations results may not be stable.

- We cannot apply these algorithms to network-level data.
- Accuracy is low.

4. Reinforcement Learning

Reinforcement learning works on a feedback-based process, in which an AI agent (A software component) automatically explore its surrounding by hitting & trail, taking action, learning from experiences, and improving its performance. Agent gets rewarded for each good action and get punished for each bad action; hence the goal of reinforcement learning agent is to maximize the rewards.

In reinforcement learning, there is no labelled data like supervised learning, and agents learn from their experiences only.

The reinforcement learning process is similar to a human being; for example, a child learns various things by experiences in his day-to-day life. An example of reinforcement learning is to play a game, where the Game is the environment, moves of an agent at each step define states, and the goal of the agent is to get a high score. Agent receives feedback in terms of punishment and rewards.

Due to its way of working, reinforcement learning is employed in different fields such as Game theory, Operation Research, Information theory, multi-agent systems.

A reinforcement learning problem can be formalized using Markov Decision Process(MDP). In MDP, the agent constantly interacts with the environment and performs actions; at each action, the environment responds and generates a new state.

Categories of Reinforcement Learning

Reinforcement learning is categorized mainly into two types of methods/algorithms:

- Positive Reinforcement Learning: Positive reinforcement learning specifies increasing the tendency that the required behaviour would occur again by adding something. It enhances the strength of the behaviour of the agent and positively impacts it.
- Negative Reinforcement Learning: Negative reinforcement learning works exactly opposite to the positive RL. It increases the tendency that the specific behaviour would occur again by avoiding the negative condition.

Real-world Use cases of Reinforcement Learning

- Video Games:
  RL algorithms are much popular in gaming applications. It is used to gain super-human performance. Some popular games that use RL algorithms are AlphaGO and AlphaGO Zero.
- Resource Management:
  The "Resource Management with Deep Reinforcement Learning" paper showed that how to use RL in computer to automatically learn and schedule resources to wait for different jobs in order to minimize average job slowdown.
- Robotics:
  RL is widely being used in Robotics applications. Robots are used in the industrial and manufacturing area, and these robots are made more powerful with reinforcement learning. There are different industries that have their vision of building intelligent robots using AI and Machine learning technology.

- Text Mining
  Text-mining, one of the great applications of NLP, is now being implemented with the help of Reinforcement Learning by Salesforce company.

Advantages and Disadvantages of Reinforcement Learning

Advantages

- It helps in solving complex real-world problems which are difficult to be solved by general techniques.
- The learning model of RL is similar to the learning of human beings; hence most accurate results can be found.
- Helps in achieving long term results.

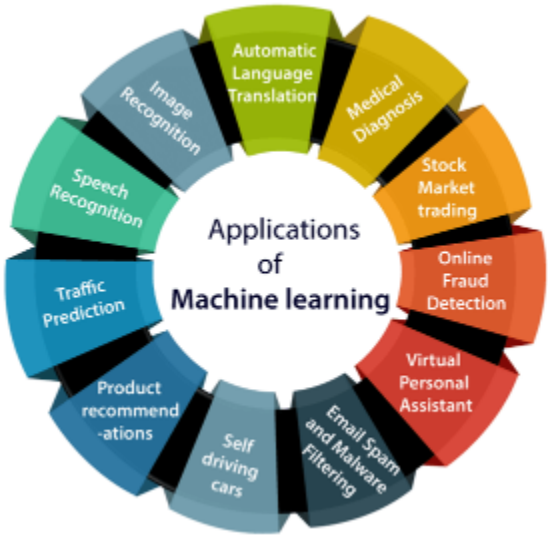
Disadvantage

- RL algorithms are not preferred for simple problems.
- RL algorithms require huge data and computations.
- Too much reinforcement learning can lead to an overload of states which can weaken the results.

The curse of dimensionality limits reinforcement learning for real physical systems.

Applications of Machine learning

Machine learning is a buzzword for today's technology, and it is growing very rapidly day by day. We are using machine learning in our daily life even without knowing it such as Google Maps, Google assistant, Alexa, etc. Below are some most trending real-world applications of Machine Learning:



1. Image Recognition:

Image recognition is one of the most common applications of machine learning. It is used to identify objects, persons, places, digital images, etc. The popular use case of image recognition and face detection is, Automatic friend tagging suggestion:

Facebook provides us a feature of auto friend tagging suggestion. Whenever we upload a photo with our Facebook friends, then we automatically get a tagging suggestion with name, and the technology behind this is machine learning's face detection and recognition algorithm.

It is based on the Facebook project named "Deep Face," which is responsible for face recognition and person identification in the picture.

2. Speech Recognition

While using Google, we get an option of "Search by voice," it comes under speech recognition, and it's a popular application of machine learning.

Speech recognition is a process of converting voice instructions into text, and it is also known as "Speech to text", or "Computer speech recognition." At present, machine learning algorithms are widely used by various applications of speech recognition. Google assistant, Siri, Cortana, and Alexa are using speech recognition technology to follow the voice instructions.

3. Traffic prediction:

If we want to visit a new place, we take help of Google Maps, which shows us the correct path with the shortest route and predicts the traffic conditions.

It predicts the traffic conditions such as whether traffic is cleared, slow-moving, or heavily congested with the help of two ways:

- Real Time location of the vehicle form Google Map app and sensors
- Average time has taken on past days at the same time.

Everyone who is using Google Map is helping this app to make it better. It takes information from the user and sends back to its database to improve the performance.

4. Product recommendations:

Machine learning is widely used by various e-commerce and entertainment companies such as Amazon, Netflix, etc., for product recommendation to the user. Whenever we search for some product on Amazon, then we started getting an advertisement for the same product while internet surfing on the same browser and this is because of machine learning.

Google understands the user interest using various machine learning algorithms and suggests the product as per customer interest.

As similar, when we use Netflix, we find some recommendations for entertainment series, movies, etc., and this is also done with the help of machine learning.

5. Self-driving cars:

One of the most exciting applications of machine learning is self-driving cars. Machine learning plays a significant role in self-driving cars. Tesla, the most popular car manufacturing company is working on self-driving car. It is using unsupervised learning method to train the car models to detect people and objects while driving.

6. Email Spam and Malware Filtering:

Whenever we receive a new email, it is filtered automatically as important, normal, and spam. We always receive an important mail in our inbox with the important symbol and spam emails in our spam box, and the technology behind this is Machine learning. Below are some spam filters used by Gmail:

- Content Filter
- Header filter
- General blacklists filter
- Rules-based filters
- Permission filters

Some machine learning algorithms such as Multi-Layer Perceptron, Decision tree, and Naïve Bayes classifier are used for email spam filtering and malware detection.

7. Virtual Personal Assistant:

We have various virtual personal assistants such as Google assistant, Alexa, Cortana, Siri. As the name suggests, they help us in finding the information using our voice instruction. These assistants can help us in various ways just by our voice instructions such as Play music, call someone, Open an email, Scheduling an appointment, etc.

These virtual assistants use machine learning algorithms as an important part.

These assistant record our voice instructions, send it over the server on a cloud, and decode it using ML algorithms and act accordingly.

8. Online Fraud Detection:

Machine learning is making our online transaction safe and secure by detecting fraud transaction. Whenever we perform some online transaction, there may be various ways that a fraudulent transaction can take place such as fake accounts, fake ids, and steal money in the middle of a transaction. So to detect this, Feed Forward Neural network helps us by checking whether it is a genuine transaction or a fraud transaction.

For each genuine transaction, the output is converted into some hash values, and these values become the input for the next round. For each genuine transaction, there is a specific pattern which gets change for the fraud transaction hence, it detects it and makes our online transactions more secure.

9. Stock Market trading:

Machine learning is widely used in stock market trading. In the stock market, there is always a risk of up and downs in shares, so for this machine learning's long short term memory neural network is used for the prediction of stock market trends.

10. Medical Diagnosis:

In medical science, machine learning is used for diseases diagnoses. With this, medical technology is growing very fast and able to build 3D models that can predict the exact position of lesions in the brain.

It helps in finding brain tumors and other brain-related diseases easily.

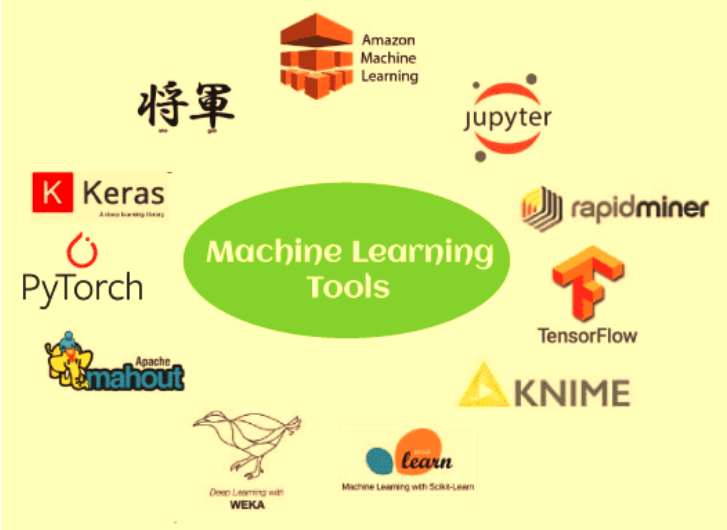
11. Automatic Language Translation:

Nowadays, if we visit a new place and we are not aware of the language then it is not a problem at all, as for this also machine learning helps us by converting the text into our known languages. Google's GNMT (Google Neural Machine Translation) provide this feature, which is a Neural Machine Learning that translates the text into our familiar language, and it called as automatic translation.

The technology behind the automatic translation is a sequence to sequence learning algorithm, which is used with image recognition and translates the text from one language to another language.

Machine Learning Tools

Machine learning is one of the most revolutionary technologies that is making lives simpler. It is a subfield of Artificial Intelligence, which analyses the data, build the model, and make predictions. Mastering machine learning tools will enable you to play with the data, train your models, discover new methods, and create algorithms.



1. TensorFlow

TensorFlow is one of the most popular open-source libraries used to train and build both machine learning and deep learning models. It provides a JS library and was developed by Google Brain Team. It is much popular among machine learning enthusiasts, and they use it for building different ML applications. It offers a powerful library, tools, and resources for numerical computation, specifically for large scale machine learning and deep learning projects. It enables data scientists/ML developers to build and deploy machine learning applications efficiently. For training and building the ML models, TensorFlow provides a high-level Keras API, which lets users easily start with TensorFlow and machine learning.

Features:

Below are some top features:

- TensorFlow enables us to build and train our ML models easily.
- It also enables you to run the existing models using the TensorFlow.js
- It provides multiple abstraction levels that allow the user to select the correct resource as per the requirement.
- It helps in building a neural network.
- Provides support of distributed computing.
- While building a model, for more need of flexibility, it provides eager execution that enables immediate iteration and intuitive debugging.
- This is open-source software and highly flexible.
- It also enables the developers to perform numerical computations using data flow graphs.
- Run-on GPUs and CPUs, and also on various mobile computing platforms.
- It provides a functionality of auto diff (Automatically computing gradients is called automatic differentiation or auto diff).
- It enables to easily deploy and training the model in the cloud.
- It can be used in two ways, i.e., by installing through NPM or by script tags.
- It is free to use.

2. PyTorch

PyTorch is an open-source machine learning framework, which is based on the Torch library. This framework is free and open-source and developed by FAIR(Facebook's AI Research lab). It is one of the popular ML frameworks, which can be used for various applications, including computer vision and natural language processing. PyTorch has Python and C++ interfaces; however, the Python interface is more interactive. Different deep learning software is made up on top of PyTorch, such as PyTorch Lightning, Hugging Face's Transformers, Tesla autopilot, etc.
It specifies a Tensor class containing an n-dimensional array that can perform tensor computations along with GPU support.

Features:

Below are some top features:

- It enables the developers to create neural networks using Autograde Module.
- It is more suitable for deep learning researches with good speed and flexibility.
- It can also be used on cloud platforms.
- It includes tutorial courses, various tools, and libraries.
- It also provides a dynamic computational graph that makes this library more popular.
- It allows changing the network behaviour randomly without any lag.
- It is easy to use due to its hybrid front-end.

- It is freely available.

3. Google Cloud ML Engine

While training a classifier with a huge amount of data, a computer system might not perform well. However, various machine learning or deep learning projects requires millions or billions of training datasets. Or the algorithm that is being used is taking a long time for execution. In such a case, one should go for the Google Cloud ML Engine. It is a hosted platform where ML developers and data scientists build and run optimum quality machine, learning models. It provides a managed service that allows developers to easily create ML models with any type of data and of any size.

Features:

Below are the top features:

- Provides machine learning model training, building, deep learning and predictive modelling.
- The two services, namely, prediction and training, can be used independently or combinedly.
- It can be used by enterprises, i.e., for identifying clouds in a satellite image, responding faster to emails of customers.
- It can be widely used to train a complex model.

4. Amazon Machine Learning (AML)

Amazon provides a great number of machine learning tools, and one of them is Amazon Machine Learning or AML. Amazon Machine Learning (AML) is a cloud-based and robust machine learning software application, which is widely used for building machine learning models and making predictions. Moreover, it integrates data from multiple sources, including Redshift, Amazon S3, or RDS.

Features

Below are some top features:

- AML offers visualization tools and wizards.
- Enables the users to identify the patterns, build mathematical models, and make predictions.
- It provides support for three types of models, which are multi-class classification, binary classification, and regression.
- It permits users to import the model into or export the model out from Amazon Machine Learning.
- It also provides core concepts of machine learning, including ML models, Data sources, Evaluations, Real-time predictions and Batch predictions.
- It enables the user to retrieve predictions with the help of batch APIs for bulk requests or real-time APIs for individual requests.

5. NET

Accord.Net is .Net based Machine Learning framework, which is used for scientific computing. It is combined with audio and image processing libraries that are written in C#. This framework provides different libraries for various applications in ML, such as Pattern Recognition, linear algebra, Statistical Data processing. One popular package of the Accord.Net framework is Accord. Statistics, Accord.Math, and Accord.MachineLearning.

Features

Below are some top features:

- It contains 38+ kernel Functions.
- Consists of more than 40 non-parametric and parametric estimation of statistical distributions.
- Used for creating production-grade computer audition, computer vision, signal processing, and statistics apps.
- Contains more than 35 hypothesis tests that include two-way and one way ANOVA tests, non-parametric tests such as the Kolmogorov-Smirnov test and many more.

6. Apache Mahout

Apache Mahout is an open-source project of Apache Software Foundation, which is used for developing machine learning applications mainly focused on Linear Algebra. It is a distributed linear algebra framework and mathematically expressive Scala DSL, which enable the developers to promptly implement their own algorithms. It also provides Java/Scala libraries to perform Mathematical operations mainly based on linear algebra and statistics.

Features:

Below are some top features:

- It enables developers to implement machine learning techniques, including recommendation, clustering, and classification.
- It is an efficient framework for implementing scalable algorithms.
- It consists of matrix and vector libraries.
- It provides support for multiple distributed backends(including Apache Spark)
- It runs on top of Apache Hadoop using the MapReduce paradigm.

7. Shogun

Shogun is a free and open-source machine learning software library, which was created by Gunnar Raetsch and Soeren Sonnenburg in the year 1999. This software library is written in C++ and supports interfaces for different languages such as Python, R, Scala, C#, Ruby, etc., using SWIG(Simplified Wrapper and Interface Generator). The main aim of Shogun is on different kernel-based algorithms such as Support Vector Machine (SVM), K-Means Clustering, etc., for regression and classification problems. It also provides the complete implementation of Hidden Markov Models.

Features:

Below are some top features:

- The main aim of Shogun is on different kernel-based algorithms such as Support Vector Machine (SVM), K-Means Clustering, etc., for regression and classification problems.
- It provides support for the use of pre-calculated kernels.
- It also offers to use a combined kernel using Multiple kernel Learning Functionality.
- This was initially designed for processing a huge dataset that consists of up to 10 million samples.
- It also enables users to work on interfaces on different programming languages such as Lua, Python, Java, C#, Octave, Ruby, MATLAB, and R.

8. Oryx2

It is a realization of the lambda architecture and built on Apache Kafka and Apache Spark. It is widely used for real-time large-scale machine learning projects. It is a framework for building apps, including end-to-end applications for filtering, packaged, regression, classification, and clustering. It is written in Java languages, including Apache Spark, Hadoop, Tomcat, Kafka, etc. The latest version of Oryx2 is Oryx 2.8.0.

Features:

Below are some top features:

- It has three tiers: specialization on top providing ML abstractions, generic lambda architecture tier, end-to-end implementation of the same standard ML algorithms.
- The original project of Oryx2 was Oryx1, and after some upgrades, Oryx2 was launched.
- It is well suited for large-scale real-time machine learning projects.
- It contains three layers which are arranged side-by-side, and these are named as Speed layer, batch layer, and serving layer.
- It also has a data transport layer that transfer data between different layers and receives input from external sources.

9. Apache Spark MLlib

Apache Spark MLlib is a scalable machine learning library that runs on Apache Mesos, Hadoop, Kubernetes, standalone, or in the cloud. Moreover, it can access data from different data sources. It is an open-source cluster-computing framework that offers an interface for complete clusters along with data parallelism and fault tolerance.

For optimized numerical processing of data, MLlib provides linear algebra packages such as Breeze and netlib-Java. It uses a query optimizer and physical execution engine for achieving high performance with both batch and streaming data.

Features

Below are some top features:

- MLlib contains various algorithms, including Classification, Regression, Clustering, recommendations, association rules, etc.
- It runs different platforms such as Hadoop, Apache Mesos, Kubernetes, standalone, or in the cloud against diverse data sources.
- It contains high-quality algorithms that provide great results and performance.
- It is easy to use as it provides interfaces In Java, Python, Scala, R, and SQL.

10. Google ML kit for Mobile

For Mobile app developers, Google brings ML Kit, which is packaged with the expertise of machine learning and technology to create more robust, optimized, and personalized apps. This

tools kit can be used for face detection, text recognition, landmark detection, image labelling, and barcode scanning applications. One can also use it for working offline.

Features:

Below are some top features:

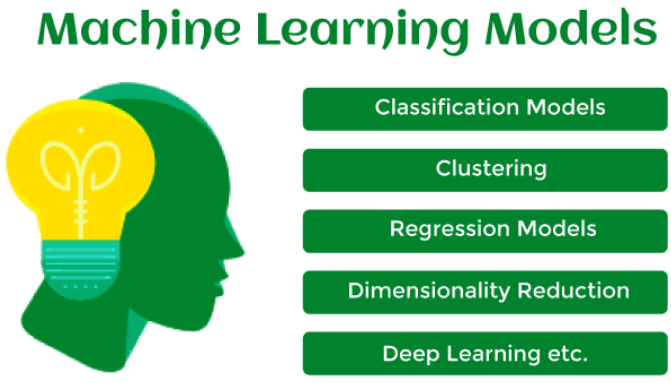
- The ML kit is optimized for mobile.
- It includes the advantages of different machine learning technologies.
- It provides easy-to-use APIs that enables powerful use cases in your mobile apps.
- It includes Vision API and Natural Language APIS to detect faces, text, and objects, and identify different languages & provide reply suggestions.

Issues in machine learning

- What algorithms can approximate functions well (and when)?
- How does number of training examples influence accuracy?
- How does complexity of hypothesis representation impact it?
- How does noisy data influence accuracy?
- What are the theoretical limits of learnability?
- How can prior knowledge of learner help?
- What clues can we get from biological learning systems?
- How can systems alter their own representations?

Machine Learning Models

A machine learning model is defined as a mathematical representation of the output of the training process. Machine learning is the study of different algorithms that can improve automatically through experience & old data and build the model. A machine learning model is similar to computer software designed to recognize patterns or behaviors based on previous experience or data. The learning algorithm discovers patterns within the training data, and it outputs an ML model which captures these patterns and makes predictions on new data.



Let's understand an example of the ML model where we are creating an app to recognize the user's emotions based on facial expressions. So, creating such an app is possible by Machine learning models where we will train a model by feeding images of faces with various emotions labeled on them. Whenever this app is used to determine the user's mood, it reads all fed data then determines any user's mood.
Hence, in simple words, we can say that a machine learning model is a simplified representation of something or a process. In this topic, we will discuss different machine learning models and their techniques and algorithms.

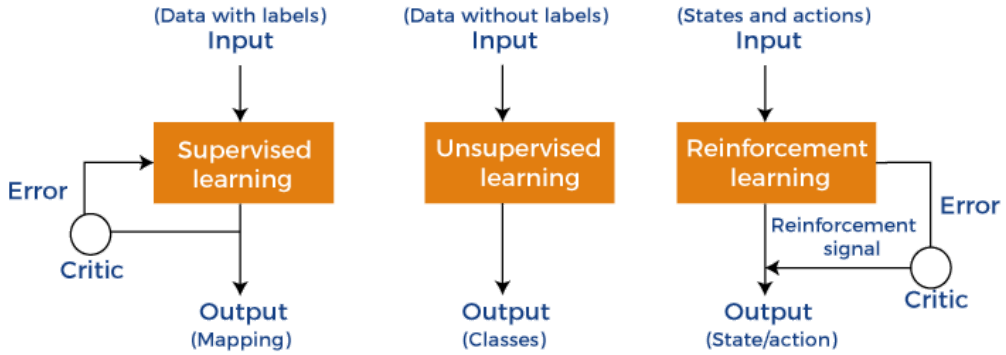
What is Machine Learning Model?

Machine Learning models can be understood as a program that has been trained to find patterns within new data and make predictions. These models are represented as a mathematical function that takes requests in the form of input data, makes predictions on input data, and then provides an output in response. First, these models are trained over a set of data, and then they are provided an algorithm to reason over data, extract the pattern from feed data and learn from those data. Once these models get trained, they can be used to predict the unseen dataset.
There are various types of machine learning models available based on different business goals and data sets.

Classification of Machine Learning Models:

Based on different business goals and data sets, there are three learning models for algorithms. Each machine learning algorithm settles into one of the three models:

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning



Supervised Learning is further divided into two categories:

- **Classification**
- **Regression**

Unsupervised Learning is also divided into below categories:

- **Clustering**
- **Association Rule**
- **Dimensionality Reduction**

1**. Supervised Machine Learning Models**

Supervised Learning is the simplest machine learning model to understand in which input data is called training data and has a known label or result as an output. So, it works on the principle of input-output pairs. It requires creating a function that can be trained using a training data set, and then it is applied to unknown data and makes some predictive performance. Supervised learning is task-based and tested on labeled data sets.

We can implement a supervised learning model on simple real-life problems. For example, we have a dataset consisting of age and height; then, we can build a supervised learning model to predict the person's height based on their age.

Supervised Learning models are further classified into two categories:

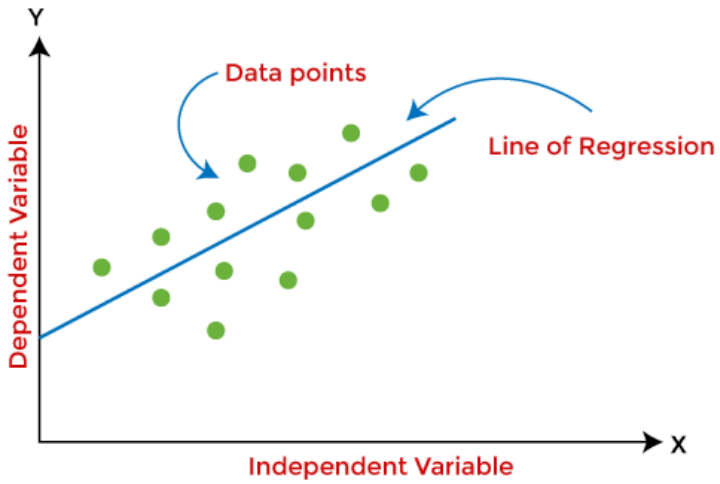
Regression

In regression problems, the output is a continuous variable. Some commonly used Regression models are as follows:

a) Linear Regression

Linear regression is the simplest machine learning model in which we try to predict one output variable using one or more input variables. The representation of linear regression is a linear equation, which combines a set of input values(x) and predicted output(y) for the set of those input values. It is represented in the form of a line:

Y = bx+ c.



The main aim of the linear regression model is to find the best fit line that best fits the data points.

Linear regression is extended to multiple linear regression (find a plane of best fit) and polynomial regression (find the best fit curve).

b) Decision Tree

Decision trees are the popular machine learning models that can be used for both regression and classification problems.

A decision tree uses a tree-like structure of decisions along with their possible consequences and outcomes. In this, each internal node is used to represent a test on an attribute; each branch is used to represent the outcome of the test. The more nodes a decision tree has, the more accurate the result will be.

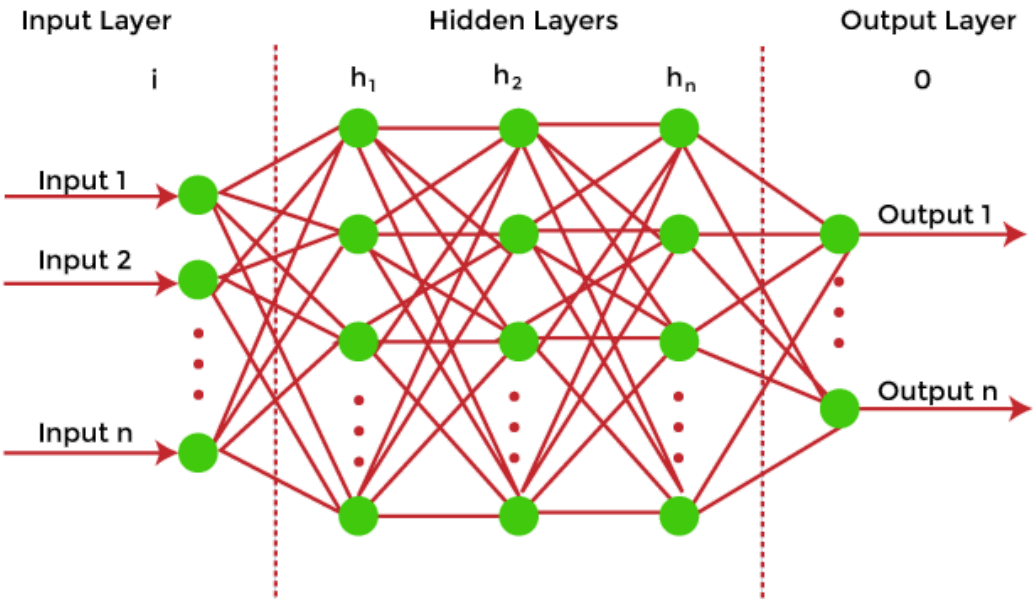
The advantage of decision trees is that they are intuitive and easy to implement, but they lack accuracy.
Decision trees are widely used in operations research, specifically in decision analysis, strategic planning, and mainly in machine learning.

c) Random Forest

Random Forest is the ensemble learning method, which consists of a large number of decision trees. Each decision tree in a random forest predicts an outcome, and the prediction with the majority of votes is considered as the outcome.
A random forest model can be used for both regression and classification problems.
For the classification task, the outcome of the random forest is taken from the majority of votes. Whereas in the regression task, the outcome is taken from the mean or average of the predictions generated by each tree.

d) Neural Networks

Neural networks are the subset of machine learning and are also known as artificial neural networks. Neural networks are made up of artificial neurons and designed in a way that resembles the human brain structure and working. Each artificial neuron connects with many other neurons in a neural network, and such millions of connected neurons create a sophisticated cognitive structure.



Neural networks consist of a multilayer structure, containing one input layer, one or more hidden layers, and one output layer. As each neuron is connected with another neuron, it transfers data from one layer to the other neuron of the next layers. Finally, data reaches the last layer or output layer of the neural network and generates output.
Neural networks depend on training data to learn and improve their accuracy. However, a perfectly trained & accurate neural network can cluster data quickly and become a powerful machine learning and AI tool. One of the best-known neural networks is Google's search algorithm.

Classification

Classification models are the second type of Supervised Learning techniques, which are used to generate conclusions from observed values in the categorical form. For example, the classification model can identify if the email is spam or not; a buyer will purchase the product or not, etc. Classification algorithms are used to predict two classes and categorize the output into different groups.

In classification, a classifier model is designed that classifies the dataset into different categories, and each category is assigned a label.

There are two types of classifications in machine learning:

- Binary classification: If the problem has only two possible classes, called a binary classifier. For example, cat or dog, Yes or No,
- Multi-class classification: If the problem has more than two possible classes, it is a multi-class classifier.

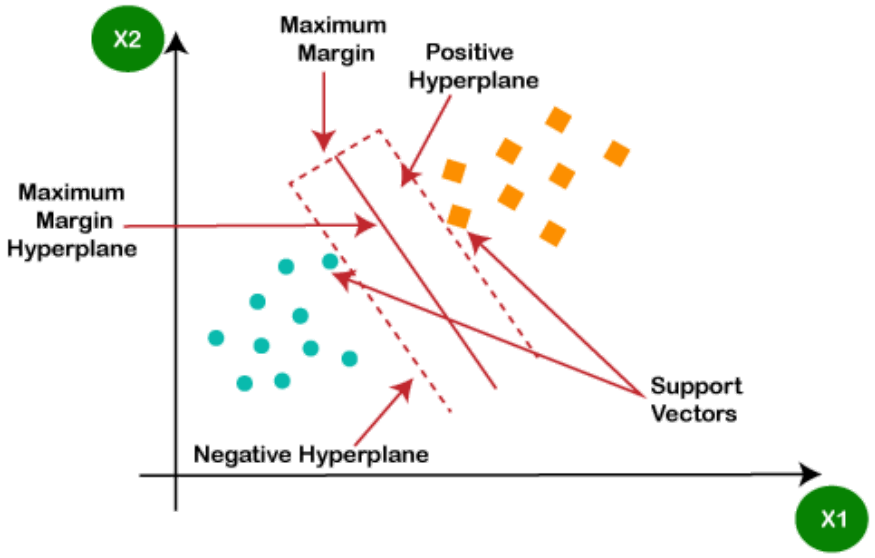
Some popular classification algorithms are as below:

a) Logistic Regression

Logistic Regression is used to solve the classification problems in machine learning. They are similar to linear regression but used to predict the categorical variables. It can predict the output in either Yes or No, 0 or 1, True or False, etc. However, rather than giving the exact values, it provides the probabilistic values between 0 & 1.

b) Support Vector Machine

Support vector machine or SVM is the popular machine learning algorithm, which is widely used for classification and regression tasks. However, specifically, it is used to solve classification problems. The main aim of SVM is to find the best decision boundaries in an N-dimensional space, which can segregate data points into classes, and the best decision boundary is known as Hyperplane. SVM selects the extreme vector to find the hyperplane, and these vectors are known as support vectors.



c) Naïve Bayes

Naïve Bayes is another popular classification algorithm used in machine learning. It is called so as it is based on Bayes theorem and follows the naïve(independent) assumption between the features which is given as:

$$P(y|X) = \frac{P(X|y) * P(y)}{P(X)}$$

Each naïve Bayes classifier assumes that the value of a specific variable is independent of any other variable/feature. For example, if a fruit needs to be classified based on color, shape, and taste. So yellow, oval, and sweet will be recognized as mango. Here each feature is independent of other features.

2. Unsupervised Machine learning models

Unsupervised Machine learning models implement the learning process opposite to supervised learning, which means it enables the model to learn from the unlabeled training dataset. Based on the unlabeled dataset, the model predicts the output. Using unsupervised learning, the model learns hidden patterns from the dataset by itself without any supervision.
Unsupervised learning models are mainly used to perform three tasks, which are as follows:

- **Clustering**
  Clustering is an unsupervised learning technique that involves clustering or groping the data points into different clusters based on similarities and differences. The objects with the most similarities remain in the same group, and they have no or very few similarities from other groups.
  Clustering algorithms can be widely used in different tasks such as Image segmentation, Statistical data analysis, Market segmentation, etc.
  Some commonly used Clustering algorithms are K-means Clustering, hierarchal Clustering, DBSCAN, etc.

- **Association** Rule Learning
  Association rule learning is an unsupervised learning technique, which finds interesting relations among variables within a large dataset. The main aim of this learning algorithm is to find the dependency of one data item on another data item and map those variables accordingly so that it can generate maximum profit. This algorithm is mainly applied in Market Basket analysis, Web usage mining, continuous production, etc.

Some popular algorithms of Association rule learning are Apriori Algorithm, Eclat, FP-growth algorithm.

- **Dimensionality** Reduction
  The number of features/variables present in a dataset is known as the dimensionality of the dataset, and the technique used to reduce the dimensionality is known as the dimensionality reduction technique.
  Although more data provides more accurate results, it can also affect the performance of the model/algorithm, such as overfitting issues. In such cases, dimensionality reduction techniques are used.
  "It is a process of converting the higher dimensions dataset into lesser dimensions dataset ensuring that it provides similar information."
  Different dimensionality reduction methods such as PCA(Principal Component Analysis), Singular Value Decomposition, etc.

Reinforcement Learning

In reinforcement learning, the algorithm learns actions for a given set of states that lead to a goal state. It is a feedback-based learning model that takes feedback signals after each state or action by interacting with the environment. This feedback works as a reward (positive for each good action and negative for each bad action), and the agent's goal is to maximize the positive rewards to improve their performance.
The behavior of the model in reinforcement learning is similar to human learning, as humans learn things by experiences as feedback and interact with the environment.
Below are some popular algorithms that come under reinforcement learning:

- **Q-learning**: Q-learning is one of the popular model-free algorithms of reinforcement learning, which is based on the Bellman equation.

It aims to learn the policy that can help the AI agent to take the best action for maximizing the reward under a specific circumstance. It incorporates Q values for each state-action pair that indicate the reward to following a given state path, and it tries to maximize the Q-value.

- **State-Action-Reward-State-Action (SARSA)**: SARSA is an On-policy algorithm based on the Markov decision process. It uses the action performed by the current policy to learn the Q-value. The SARSA algorithm stands for State Action Reward State Action, which symbolizes the tuple (s, a, r, s', a').
- **Deep Q Network**: DQN or Deep Q Neural network is Q-learning within the neural network. It is basically employed in a big state space environment where defining a Q-table would be a complex task. So, in such a case, rather than using Q-table, the neural network uses Q-values for each action based on the state.

Training Machine Learning Models

Once the Machine learning model is built, it is trained in order to get the appropriate results. To train a machine learning model, one needs a huge amount of pre-processed data. Here pre-processed data means data in structured form with reduced null values, etc. If we do not provide pre-processed data, then there are huge chances that our model may perform terribly.

How to choose the best model?

In the above section, we have discussed different machine learning models and algorithms. But one most confusing question that may arise to any beginner that "which model should I choose?". So, the answer is that it depends mainly on the business requirement or project requirement. Apart from this, it also depends on associated attributes, the volume of the available dataset, the number of features, complexity, etc. However, in practice, it is recommended that we always start with the simplest model that can be applied to the particular problem and then gradually enhance the complexity & test the accuracy with the help of parameter tuning and cross-validation.

Difference between Machine learning model and Algorithms

One of the most confusing questions among beginners is that are machine learning models, and algorithms are the same? Because in various cases in machine learning and data science, these two terms are used interchangeably.

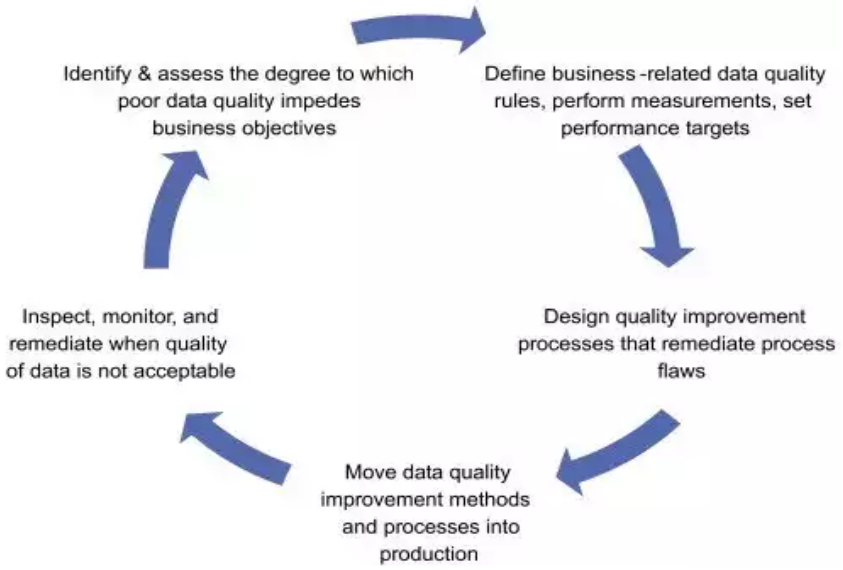
The answer to this question is No, and the machine learning model is not the same as an algorithm. In a simple way, an ML algorithm is like a procedure or method that runs on data to discover patterns from it and generate the model. At the same time, a machine learning model is like a computer program that generates output or makes predictions. More specifically, when we train an algorithm with data, it becomes a model.

Machine Learning ModelModel = Model Data + Prediction Algorithm

Data quality and remediation

Data quality management: how to implement and how it works

Data quality management (DQM) is a set of practices aimed at improving and maintaining the quality of data across a company's business units. Data management specialist David Loshin underlines the continuous nature of DQM. The expert notes that the process includes a "virtuous cycle" that's about ongoing observation, analysis, and improvement of information. The purpose of this cycle is to become proactive in controlling the health of data instead of fixing flaws once they are identified and dealing with the consequences of these flaws.



The virtuous cycle of data quality management.

1. Defining the impact of poor data on performance via data quality assessment

First of all, the data quality analyst reviews data to find potential issues that cause delays in specific operations and, consequently, decrease revenue and influence margins. The qualitative data review gives a basic understanding of what data flaws have the big impact on business processes. Then the specialist/s outlines data quality requirements and critical data quality dimensions that will be in a company.

Next, the team starts data quality assessment via top-down and bottom-up approaches. The top-down approach allows for learning how employees create and use data and what data-related problems they face along the way, and which of them are the most critical. Data assessment also helps defining operations that are the most affected by poor-quality data.

The data quality analyst may examine how data is organized in databases, interview users personally, or organize surveys in which users can document issues.

The bottom-up approach uses statistical and data analysis tools and techniques, for instance, data profiling. Data profiling employs various statistical and analytical algorithms and business rules to explore content of datasets and characteristics of their data elements.There are three types of data profiling:

- Structure discovery (structure analysis) is used to learn whether data consistent and formatted correctly. Pattern matching is one of the ways to explore data record structure. Analysts can also check statistics in data, such as the minimum and maximum values, medians, means, or standard deviations to learn about the validity of data.
- Content discovery entails examination of individual data records in a database to discover null or wrong values (incorrectly formatted).
- Relationship discovery is about understanding interconnections between datasets, data records, database fields, or cells. Relationship discovery starts from metadata review. This analysis allows for spotting and eliminating such issues as duplicates, which may occur in non-aligned datasets.

Analysts then may consult about found data issues with domain experts.

2. Defining data quality rules and metrics

First, data quality analysts compile data assessment results focusing on data elements that seem critical based on the specific user's needs. "The results of the empirical analysis will provide some types of measures that can be employed to assess the level of data quality within a particular business context," notes David Loshin in The Practitioner's Guide to Data Quality Improvement.
Then DQ analysts correlate business impacts to data flaws via defined business rules. That way, specialists define
metrics they will use to ensure data is accurate enough and can be used for operational or analytical needs. They consult with data users on acceptability thresholds for metric scores. Data with metric scores that are lower than acceptability levels, doesn't meet user expectations and must be improved to avoid negative impact on operations. Integrating acceptability thresholds with measurement methods allows for the framing of data quality metrics.

3. Defining data standards, metadata management standards, data validation rules

Once the impact of poor data is identified, data is examined, data quality rules and metrics are clear, the time comes to introduce techniques and activities on quality improvement. So, the goal of this stage is to document unified rules for data and metadata usage across the data lifecycle.
Data standards. Data quality standards are agreements on data entry, representation, formatting, and exchange used across the organization.
Metadata management standards. Policies and rules about metadata creation and maintenance are the baselines for successful data analytics initiatives and data governance. Metadata management standards can be grouped into three categories:

- Business – the use of business terms and definitions in different business contexts, the use of acronyms; data security levels and privacy level settings.
- Technical – structure, format, and rules for storing data (i.e., format and size for indexes, tables, and columns in databases, data models)
- Operational – rules for using metadata describing events and objects during the ETL process (i.e., ETL load date, update date, confidence level indicator)

Please note that some practitioners consider operational metadata as a type the technical one.
Data validity rules. Data validity rules are used to evaluate data against inconsistencies. Developers write data validity rules and integrate into applications so that tools can identify mistakes even during data entry, for instance. Data validity rules enable proactive data quality management.
It's also crucial to decide how to track data problems. Data quality issue tracking log provides information about flaws, their status, criticality, responsible employees, and includes the report notes. George Firican, director of data governance and BI at the University of British Columbia, has written an informative yet concise post, in which he advises on attributes to include in the log.
Another aspect to consider and approve is how to improve data. We'll talk about them in the following section.

4. Implementing data quality and data management standards

During this step, the data quality team implements data quality standards and processes it documented before to manage the solid quality of data across its lifecycle.
The team may organize meetings to explain to employees the new data management rules or/and introduce a business glossary – a document with common terminology approved by stakeholders and managers.
Also, the data quality team members can train employees on how to use a data quality tool to perform remediation, whether it's a custom or an-of-the-shelf solution.

5. Data monitoring and **remediation**

Data cleaning (remediation, preparation) entails detecting erroneous or incomplete records in data, removing or modifying them. There are many ways of performing data preparation: manually, automatically with data quality tools, as batch processing through scripting, via data migration, or using some of these methods together.
Data remediation includes numerous activities, such as:

- Root cause analysis – identifying the source of erroneous data, reasons for the error to occur, isolating factors that contribute to the issue, and finding the solution.
- Parsing and standardization – reviewing records in database tables against defined patterns, grammar, and representations to identify erroneous data values or values in the wrong fields and formatting them. For example, a data quality analyst may standardize values from different metric systems (lbs and kg), geographic record abbreviations (CA and US-CA).
- Matching – identifying the same or similar entities in a dataset and merging them into one. Data matching is related to identity resolution and record linkage. The technique can be applied when joining datasets and when data from multiple sources were integrated into one destination (the ETL process). One uses identity resolution in datasets containing records about individuals to create a single view of the customer. Record linkage deals with records that may or may not refer to a common entity (i.e., database key, Social security number, URL) and which may be due to differences in record shape, storage location, or curator style or preference.
- Enhancement – adding extra data from internal and external sources.
- Monitoring – evaluating data in given intervals to ensure it can well serve its purposes.

Data Preprocessing in Machine learning

Data preprocessing is a process of preparing the raw data and making it suitable for a machine learning model. It is the first and crucial step while creating a machine learning model.
When creating a machine learning project, it is not always a case that we come across the clean and formatted data. And while doing any operation with data, it is mandatory to clean it and put in a formatted way. So for this, we use data preprocessing task.

Why do we need Data Preprocessing?

A real-world data generally contains noises, missing values, and maybe in an unusable format which cannot be directly used for machine learning models. Data preprocessing is required tasks for cleaning the data and making it suitable for a machine learning model which also increases the accuracy and efficiency of a machine learning model.
It involves below steps:

- Getting the dataset
- Importing libraries
- Importing datasets
- Finding Missing Data
- Encoding Categorical Data
- Splitting dataset into training and test set
- Feature scaling

---

1) Get the Dataset

To create a machine learning model, the first thing we required is a dataset as a machine learning model completely works on data. The collected data for a particular problem in a proper format is known as the dataset.
Dataset may be of different formats for different purposes, such as, if we want to create a machine learning model for business purpose, then dataset will be different with the dataset required for a liver patient. So each dataset is different from another dataset. To use the dataset in our code, we usually put it into a CSV file. However, sometimes, we may also need to use an HTML or xlsx file.

What is a CSV File?

CSV stands for "Comma-Separated Values" files; it is a file format which allows us to save the tabular data, such as spreadsheets. It is useful for huge datasets and can use these datasets in programs.
Here we will use a demo dataset for data preprocessing, and for practice, it can be downloaded from here, "https://www.superdatascience.com/pages/machine-learning. For real-world problems, we can download datasets online from various sources such as https://www.kaggle.com/uciml/datasets, https://archive.ics.uci.edu/ml/index.php etc.
We can also create our dataset by gathering data using various API with Python and put that data into a .csv file.

2) Importing Libraries

In order to perform data preprocessing using Python, we need to import some predefined Python libraries. These libraries are used to perform some specific jobs. There are three specific libraries that we will use for data preprocessing, which are:
Numpy: Numpy Python library is used for including any type of mathematical operation in the code. It is the fundamental package for scientific calculation in Python. It also supports to add large, multidimensional arrays and matrices. So, in Python, we can import it as:

```
1. import numpy as nm
```

Here we have used nm, which is a short name for Numpy, and it will be used in the whole program.

Matplotlib: The second library is matplotlib, which is a Python 2D plotting library, and with this library, we need to import a sub-library pyplot. This library is used to plot any type of charts in Python for the code. It will be imported as below:

1. import matplotlib.pyplot as mpt

Here we have used mpt as a short name for this library.

Pandas: The last library is the Pandas library, which is one of the most famous Python libraries and used for importing and managing the datasets. It is an open-source data manipulation and analysis library. It will be imported as below:

Here, we have used pd as a short name for this library. Consider the below image:

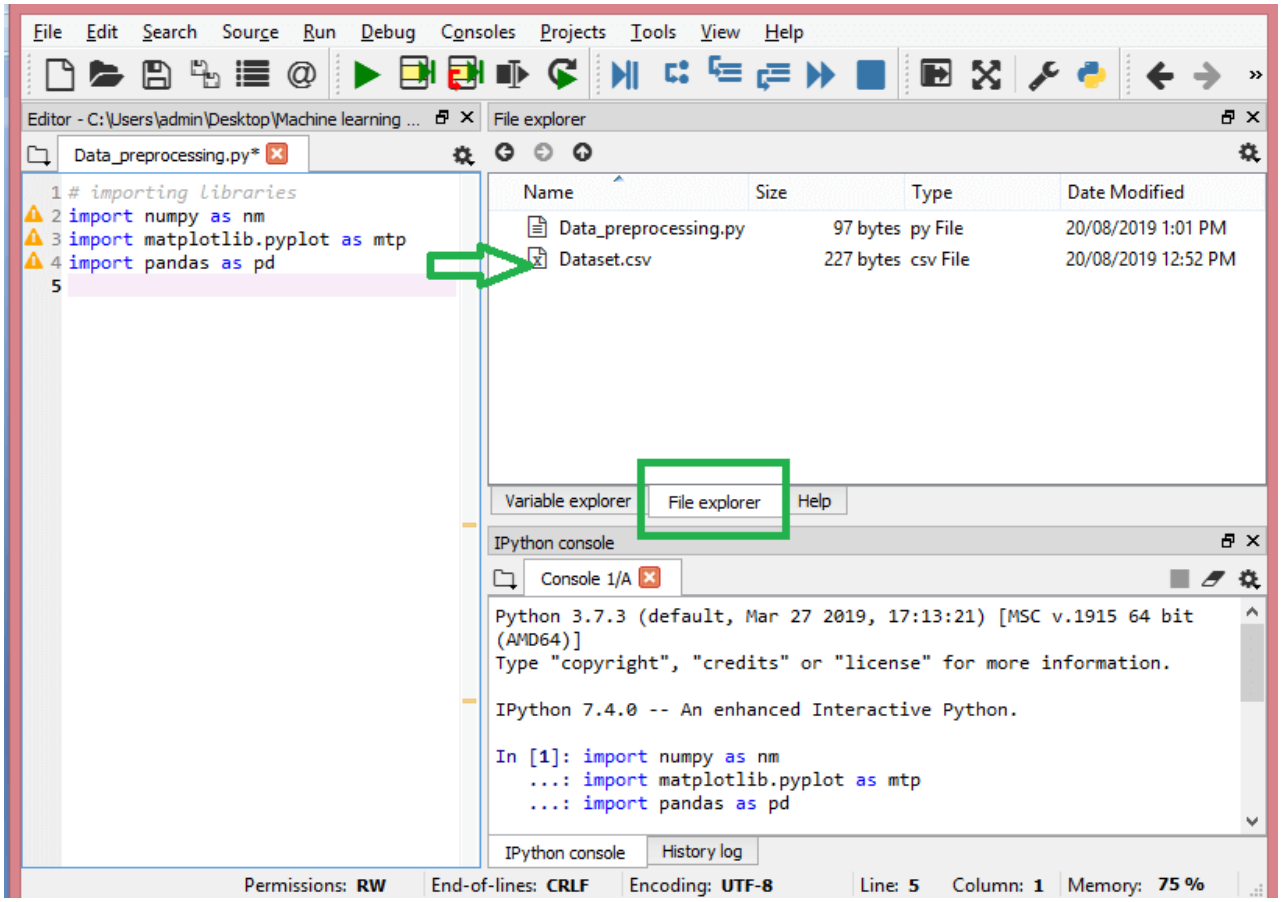
```
1 # importing libraries
2 import numpy as nm
3 import matplotlib.pyplot as mtp
4 import pandas as pd
5
```

3) Importing the Datasets

Now we need to import the datasets which we have collected for our machine learning project. But before importing a dataset, we need to set the current directory as a working directory. To set a working directory in Spyder IDE, we need to follow the below steps:

1. Save your Python file in the directory which contains dataset.
2. Go to File explorer option in Spyder IDE, and select the required directory.
3. Click on F5 button or run option to execute the file

   Here, in the below image, we can see the Python file along with required dataset. Now, the current folder is set as a working directory.



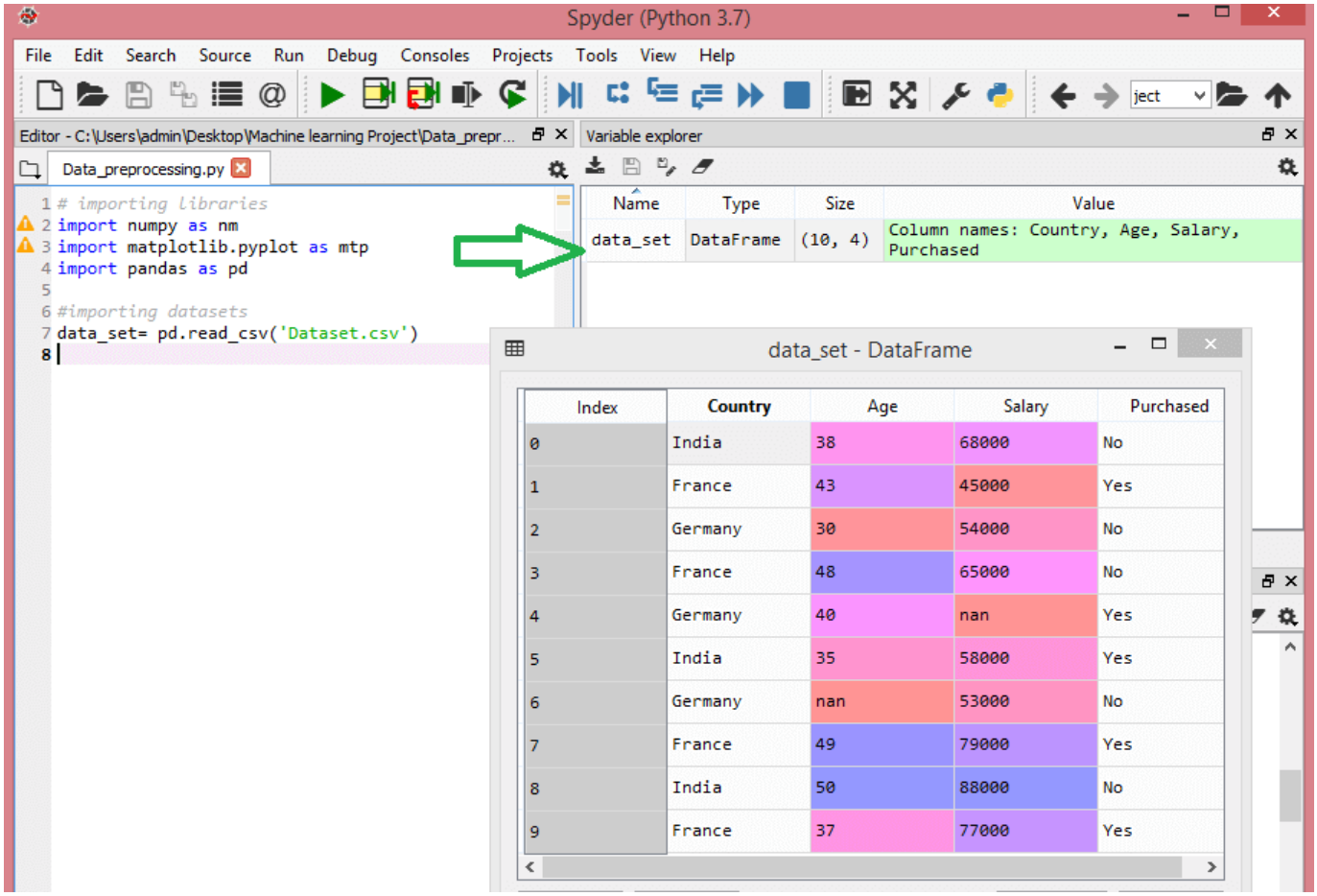
read_csv() function:

Now to import the dataset, we will use read_csv() function of pandas library, which is used to read a csv file and performs various operations on it. Using this function, we can read a csv file locally as well as through an URL.

We can use read_csv function as below:

1. data_set= pd.read_csv('Dataset.csv')

Here, data_set is a name of the variable to store our dataset, and inside the function, we have passed the name of our dataset. Once we execute the above line of code, it will successfully import the dataset in our code. We can also check the imported dataset by clicking on the section variable explorer, and then double click on data_set. Consider the below image:



As in the above image, indexing is started from 0, which is the default indexing in Python. We can also change the format of our dataset by clicking on the format option.

Extracting dependent and independent variables:

In machine learning, it is important to distinguish the matrix of features (independent variables) and dependent variables from dataset. In our dataset, there are three independent variables that are Country, Age, and Salary, and one is a dependent variable which is Purchased.
Extracting independent variable:
To extract an independent variable, we will use iloc[ ] method of Pandas library. It is used to extract the required rows and columns from the dataset.

1. x= data_set.iloc[:,:-1].values

In the above code, the first colon(:) is used to take all the rows, and the second colon(:) is for all the columns. Here we have used :-1, because we don't want to take the last column as it contains the dependent variable. So by doing this, we will get the matrix of features.
By executing the above code, we will get output as:

1. [['India' 38.0 68000.0]
2. ['France' 43.0 45000.0]
3. ['Germany' 30.0 54000.0]

```
['France' 48.0 65000.0]
['Germany' 40.0 nan]
['India' 35.0 58000.0]
['Germany' nan 53000.0]
['France' 49.0 79000.0]
['India' 50.0 88000.0]
['France' 37.0 77000.0]]
```

As we can see in the above output, there are only three variables.

Extracting dependent variable:

To extract dependent variables, again, we will use Pandas .iloc[] method.

```
y= data_set.iloc[:,3].values
```

Here we have taken all the rows with the last column only. It will give the array of dependent variables.

By executing the above code, we will get output as:

Output:

```
array(['No', 'Yes', 'No', 'No', 'Yes', 'Yes', 'No', 'Yes', 'No', 'Yes'],
      dtype=object)
```

4) Handling Missing data:

The next step of data preprocessing is to handle missing data in the datasets. If our dataset contains some missing data, then it may create a huge problem for our machine learning model. Hence it is necessary to handle missing values present in the dataset.

Ways to handle missing data:

There are mainly two ways to handle missing data, which are:

By deleting the particular row: The first way is used to commonly deal with null values. In this way, we just delete the specific row or column which consists of null values. But this way is not so efficient and removing data may lead to loss of information which will not give the accurate output.

By calculating the mean: In this way, we will calculate the mean of that column or row which contains any missing value and will put it on the place of missing value. This strategy is useful for the features which have numeric data such as age, salary, year, etc. Here, we will use this approach.

To handle missing values, we will use Scikit-learn library in our code, which contains various libraries for building machine learning models. Here we will use Imputer class of sklearn.preprocessing library. Below is the code for it:

```
#handling missing data (Replacing missing data with the mean value)
from sklearn.preprocessing import Imputer
imputer= Imputer(missing_values ='NaN', strategy='mean', axis = 0)
#Fitting imputer object to the independent variables x.
imputerimputer= imputer.fit(x[:, 1:3])
#Replacing missing data with the calculated mean value
x[:, 1:3]= imputer.transform(x[:, 1:3])
```

Output:

```
array([['India', 38.0, 68000.0],
       ['France', 43.0, 45000.0],
       ['Germany', 30.0, 54000.0],
       ['France', 48.0, 65000.0],
       ['Germany', 40.0, 65222.22222222222],
       ['India', 35.0, 58000.0],
       ['Germany', 41.111111111111114, 53000.0],
       ['France', 49.0, 79000.0],
       ['India', 50.0, 88000.0],
       ['France', 37.0, 77000.0]], dtype=object
```

As we can see in the above output, the missing values have been replaced with the means of rest column values.

5) Encoding Categorical data:

Categorical data is data which has some categories such as, in our dataset; there are two categorical variable, Country, and Purchased.
Since machine learning model completely works on mathematics and numbers, but if our dataset would have a categorical variable, then it may create trouble while building the model. So it is necessary to encode these categorical variables into numbers.

For Country variable:

Firstly, we will convert the country variables into categorical data. So to do this, we will use LabelEncoder() class from preprocessing library.

```
1. #Catgorical data
2. #for Country Variable
3. from sklearn.preprocessing import LabelEncoder
4. label_encoder_x= LabelEncoder()
5. x[:, 0]= label_encoder_x.fit_transform(x[:, 0])
```

Output:

```
Out[15]:
 array([[2, 38.0, 68000.0],
        [0, 43.0, 45000.0],
       [1, 30.0, 54000.0],
       [0, 48.0, 65000.0],
       [1, 40.0, 65222.22222222222],
       [2, 35.0, 58000.0],
       [1, 41.111111111111114, 53000.0],
       [0, 49.0, 79000.0],
       [2, 50.0, 88000.0],
      [0, 37.0, 77000.0]], dtype=object)
```

Explanation:

In above code, we have imported LabelEncoder class of sklearn library. This class has successfully encoded the variables into digits.

But in our case, there are three country variables, and as we can see in the above output, these variables are encoded into 0, 1, and 2. By these values, the machine learning model may assume that there is some correlation between these variables which will produce the wrong output. So to remove this issue, we will use dummy encoding.

Dummy Variables:

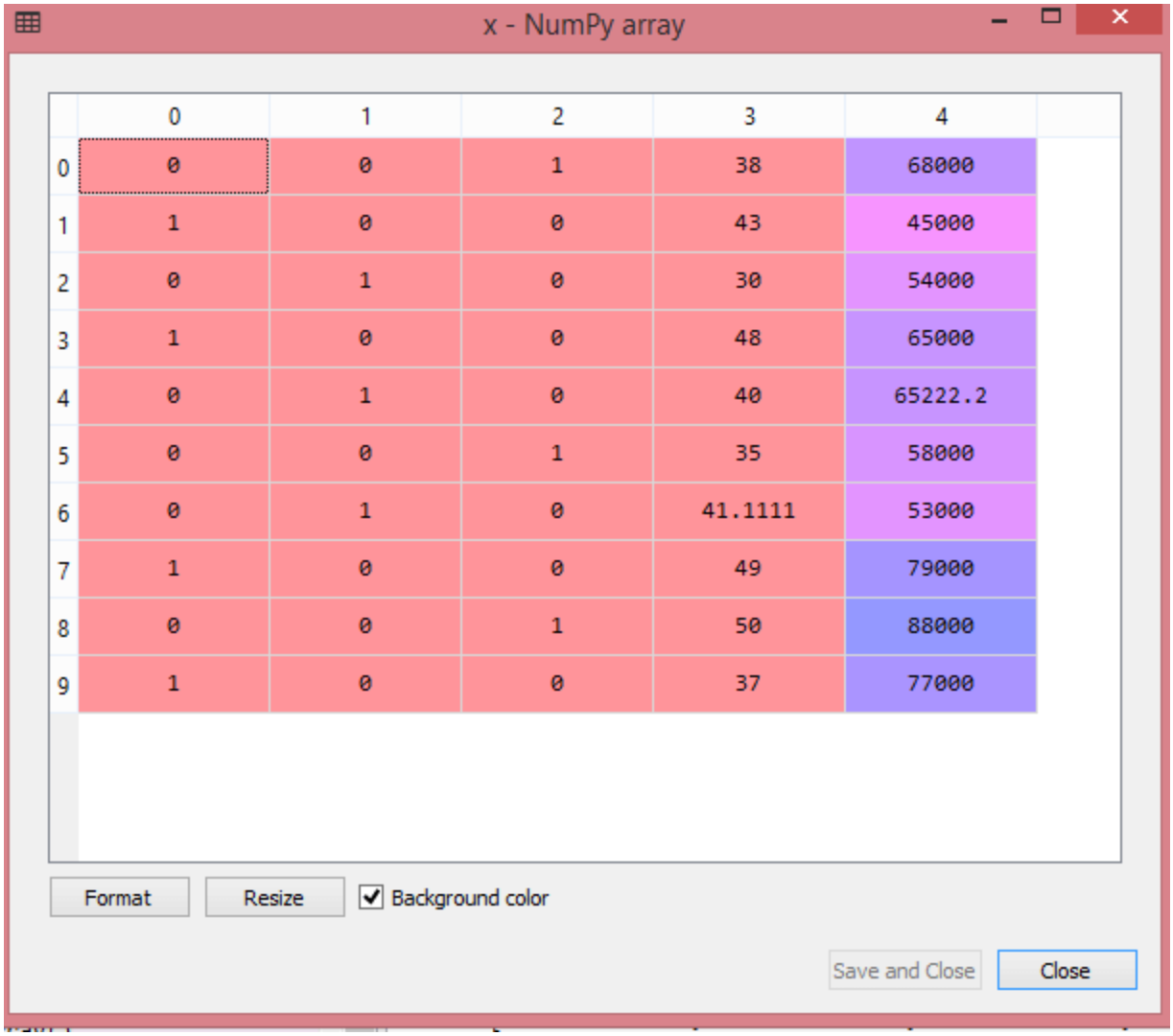
Dummy variables are those variables which have values 0 or 1. The 1 value gives the presence of that variable in a particular column, and rest variables become 0. With dummy encoding, we will have a number of columns equal to the number of categories.
In our dataset, we have 3 categories so it will produce three columns having 0 and 1 values. For Dummy Encoding, we will use OneHotEncoder class of preprocessing library.

```
#for Country Variable
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
label_encoder_x= LabelEncoder()
x[:, 0]= label_encoder_x.fit_transform(x[:, 0])
#Encoding for dummy variables
onehot_encoder= OneHotEncoder(categorical_features= [0])
x= onehot_encoder.fit_transform(x).toarray()
```

Output:

```
array([[0.00000000e+00, 0.00000000e+00, 1.00000000e+00, 3.80000000e+01,
        6.80000000e+04],
       [1.00000000e+00, 0.00000000e+00, 0.00000000e+00, 4.30000000e+01,
        4.50000000e+04],
       [0.00000000e+00, 1.00000000e+00, 0.00000000e+00, 3.00000000e+01,
        5.40000000e+04],
       [1.00000000e+00, 0.00000000e+00, 0.00000000e+00, 4.80000000e+01,
        6.50000000e+04],
       [0.00000000e+00, 1.00000000e+00, 0.00000000e+00, 4.00000000e+01,
        6.52222222e+04],
       [0.00000000e+00, 0.00000000e+00, 1.00000000e+00, 3.50000000e+01,
        5.80000000e+04],
       [0.00000000e+00, 1.00000000e+00, 0.00000000e+00, 4.11111111e+01,
        5.30000000e+04],
       [1.00000000e+00, 0.00000000e+00, 0.00000000e+00, 4.90000000e+01,
        7.90000000e+04],
       [0.00000000e+00, 0.00000000e+00, 1.00000000e+00, 5.00000000e+01,
        8.80000000e+04],
       [1.00000000e+00, 0.00000000e+00, 0.00000000e+00, 3.70000000e+01,
        7.70000000e+04]])
```

As we can see in the above output, all the variables are encoded into numbers 0 and 1 and divided into three columns.
It can be seen more clearly in the variables explorer section, by clicking on x option as:



	0	1	2	3	4
0	0	0	1	38	68000
1	1	0	0	43	45000
2	0	1	0	30	54000
3	1	0	0	48	65000
4	0	1	0	40	65222.2
5	0	0	1	35	58000
6	0	1	0	41.1111	53000
7	1	0	0	49	79000
8	0	0	1	50	88000
9	1	0	0	37	77000

For Purchased Variable:

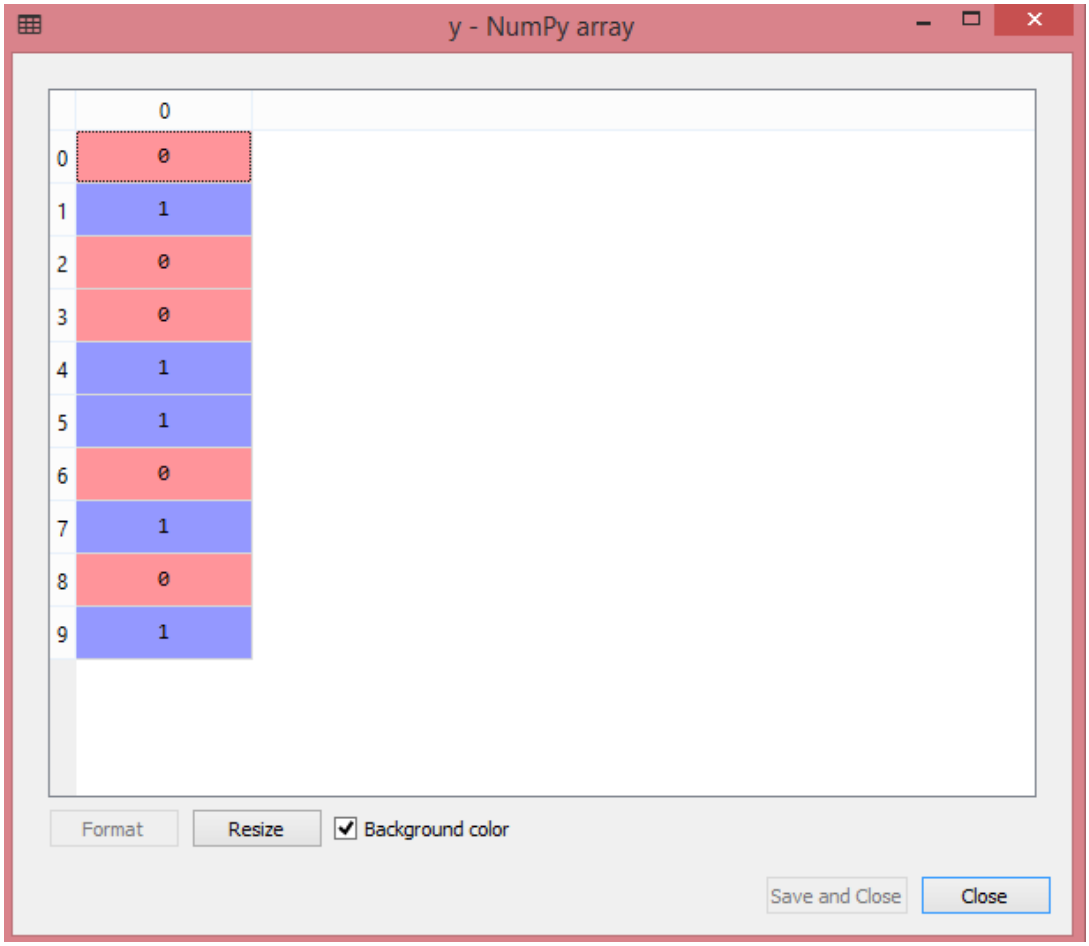
1. labelencoder_y= LabelEncoder()
2. y= labelencoder_y.fit_transform(y)

For the second categorical variable, we will only use labelencoder object of LableEncoder class. Here we are not using OneHotEncoder class because the purchased variable has only two categories yes or no, and which are automatically encoded into 0 and 1.

Output:

```
Out[17]: array([0, 1, 0, 0, 1, 1, 0, 1, 0, 1])
```

It can also be seen as:

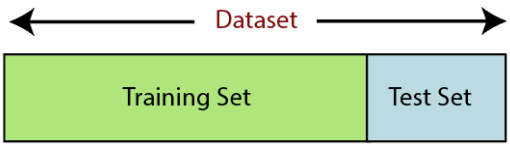


6) Splitting the Dataset into the Training set and Test set

In machine learning data preprocessing, we divide our dataset into a training set and test set. This is one of the crucial steps of data preprocessing as by doing this, we can enhance the performance of our machine learning model.

Suppose, if we have given training to our machine learning model by a dataset and we test it by a completely different dataset. Then, it will create difficulties for our model to understand the correlations between the models.

If we train our model very well and its training accuracy is also very high, but we provide a new dataset to it, then it will decrease the performance. So we always try to make a machine learning model which performs well with the training set and also with the test dataset. Here, we can define these datasets as:



Training Set: A subset of dataset to train the machine learning model, and we already know the output.

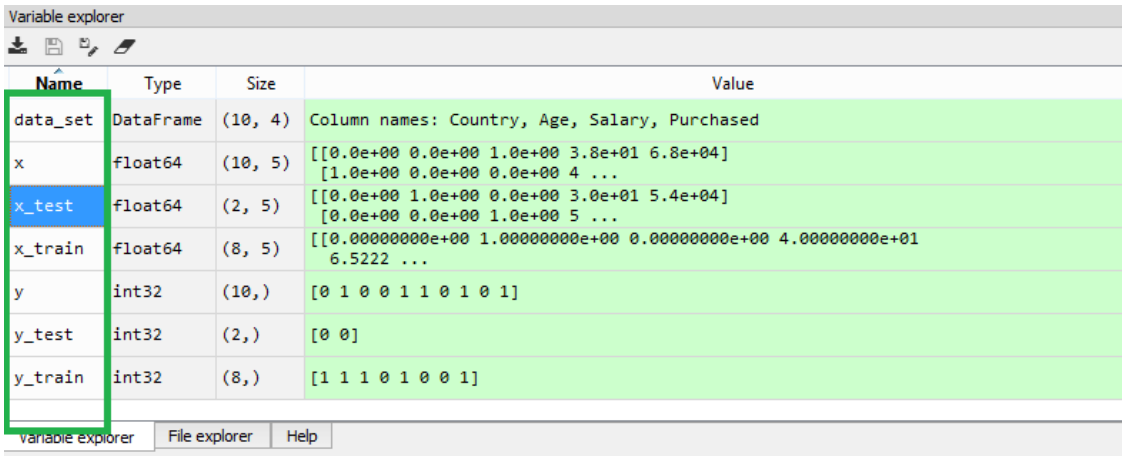
Test set: A subset of dataset to test the machine learning model, and by using the test set, model predicts the output.
For splitting the dataset, we will use the below lines of code:

```
1. from sklearn.model_selection import train_test_split
2. x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 0.2, random_state=0)
```

Explanation:

- In the above code, the first line is used for splitting arrays of the dataset into random train and test subsets.
- In the second line, we have used four variables for our output that are
  - x_train: features for the training data
  - x_test: features for testing data
  - y_train: Dependent variables for training data
  - y_test: Independent variable for testing data
- In train_test_split() function, we have passed four parameters in which first two are for arrays of data, and test_size is for specifying the size of the test set. The test_size maybe .5, .3, or .2, which tells the dividing ratio of training and testing sets.
- The last parameter random_state is used to set a seed for a random generator so that you always get the same result, and the most used value for this is 42.

Output:
By executing the above code, we will get 4 different variables, which can be seen under the variable explorer section.

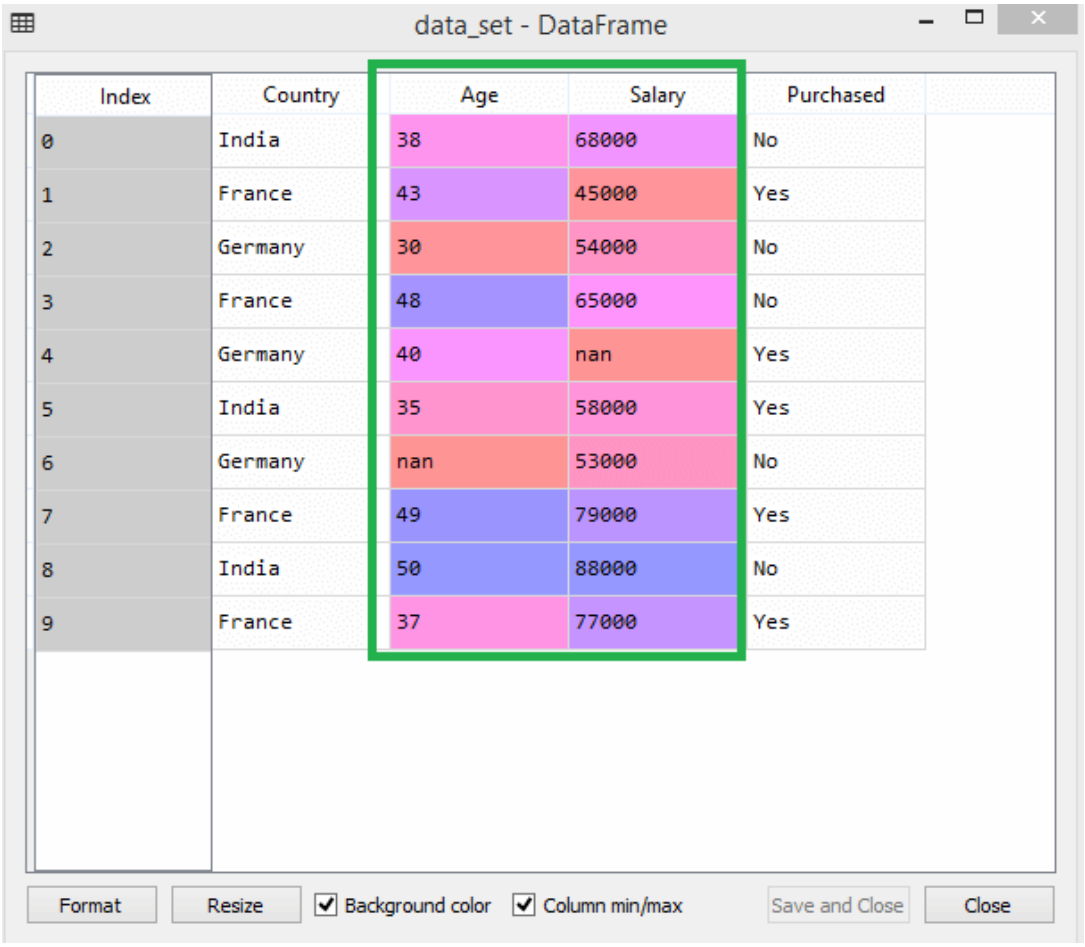


As we can see in the above image, the x and y variables are divided into 4 different variables with corresponding values.

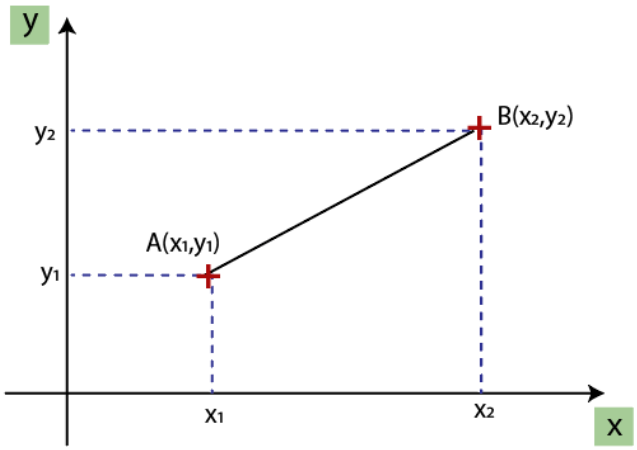
7) Feature Scaling

Feature scaling is the final step of data preprocessing in machine learning. It is a technique to standardize the independent variables of the dataset in a specific range. In feature scaling, we put our variables in the same range and in the same scale so that no any variable dominate the other variable.

Consider the below dataset:



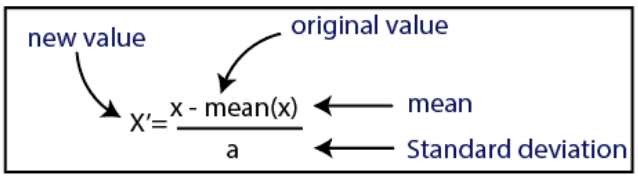
As we can see, the age and salary column values are not on the same scale. A machine learning model is based on Euclidean distance, and if we do not scale the variable, then it will cause some issue in our machine learning model.
Euclidean distance is given as:



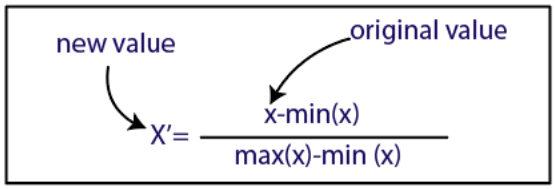
Euclidean Distance Between A and B=$\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$

If we compute any two values from age and salary, then salary values will dominate the age values, and it will produce an incorrect result. So to remove this issue, we need to perform feature scaling for machine learning.
There are two ways to perform feature scaling in machine learning:

Standardization



Normalization



Here, we will use the standardization method for our dataset.
For feature scaling, we will import StandardScaler class of sklearn.preprocessing library as:

1. from sklearn.preprocessing import StandardScaler
   Now, we will create the object of StandardScaler class for independent variables or features. And then we will fit and transform the training dataset.

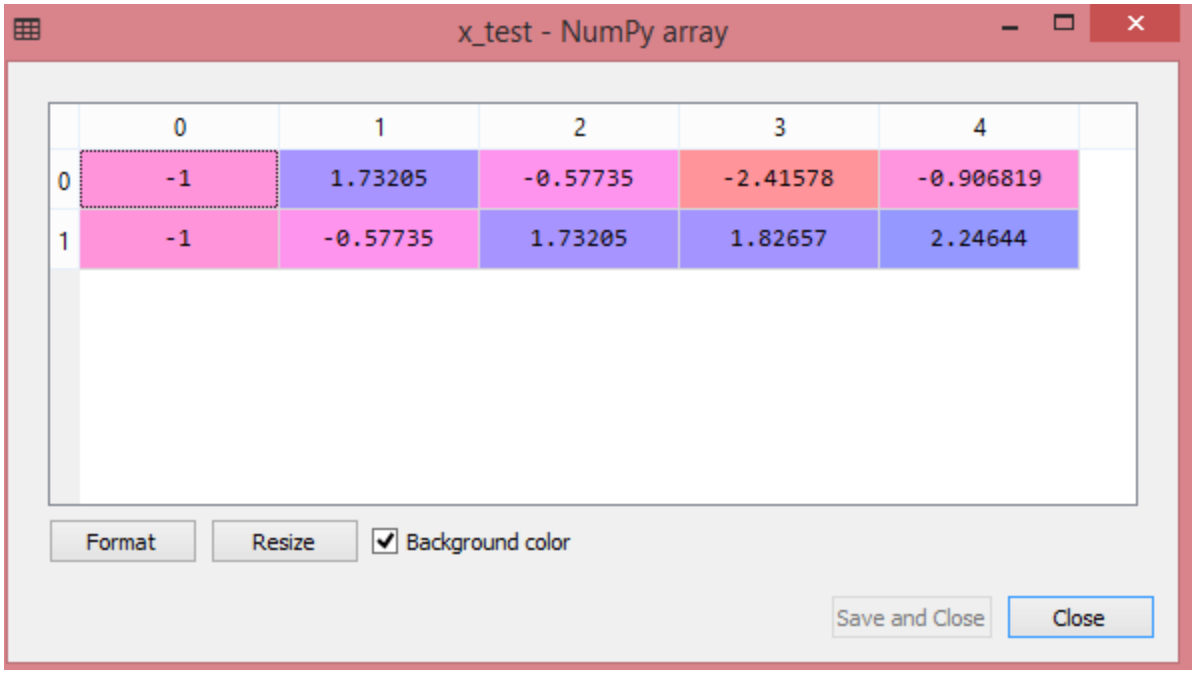
1. st_x= StandardScaler()
2. x_train= st_x.fit_transform(x_train)
   For test dataset, we will directly apply transform() function instead of fit_transform() because it is already done in training set.

1. x_test= st_x.transform(x_test)
   Output:
   By executing the above lines of code, we will get the scaled values for x_train and x_test as:

   x_train:

	0	1	2	3	4
0	-1	1.73205	-0.57735	-0.294607	0.133962
1	1	-0.57735	-0.57735	-0.930959	1.22627
2	1	-0.57735	-0.57735	0.341745	-1.7415
3	-1	1.73205	-0.57735	-0.0589215	-0.999562
4	1	-0.57735	-0.57735	1.61445	1.41175
5	1	-0.57735	-0.57735	1.40233	0.113352
6	-1	-0.57735	1.73205	-0.718842	0.391581
7	-1	-0.57735	1.73205	-1.35519	-0.535848

   x_test:



As we can see in the above output, all the variables are scaled between values -1 to 1.

Combining all the steps:

Now, in the end, we can combine all the steps together to make our complete code more understandable.

```
# importing libraries
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd

#importing datasets
data_set= pd.read_csv('Dataset.csv')

#Extracting Independent Variable
x= data_set.iloc[:, :-1].values

#Extracting Dependent variable
y= data_set.iloc[:, 3].values

#handling missing data(Replacing missing data with the mean value)
from sklearn.preprocessing import Imputer
imputer= Imputer(missing_values ='NaN', strategy='mean', axis = 0)

#Fitting imputer object to the independent varibles x.
imputerimputer= imputer.fit(x[:, 1:3])

```

```
#Replacing missing data with the calculated mean value
x[:, 1:3]= imputer.transform(x[:, 1:3])

#for Country Variable
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
label_encoder_x= LabelEncoder()
x[:, 0]= label_encoder_x.fit_transform(x[:, 0])

#Encoding for dummy variables
onehot_encoder= OneHotEncoder(categorical_features= [0])
x= onehot_encoder.fit_transform(x).toarray()

#encoding for purchased variable
labelencoder_y= LabelEncoder()
y= labelencoder_y.fit_transform(y)

# Splitting the dataset into training and test set.
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 0.2, random_state=0)

#Feature Scaling of datasets
from sklearn.preprocessing import StandardScaler
st_x= StandardScaler()
x_train= st_x.fit_transform(x_train)
x_test= st_x.transform(x_test)
```

In the above code, we have included all the data preprocessing steps together. But there are some steps or lines of code which are not necessary for all machine learning models. So we can exclude them from our code to make it reusable for all models.

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

Examples

- Handwriting recognition learning problem
    - Task T : Recognizing and classifying handwritten words within images
    - Performance P : Percent of words correctly classified
    - Training experience E : A dataset of handwritten words with given classifications
- A robot driving learning problem
    - Task T : Driving on highways using vision sensors
    - Performance P : Average distance traveled before an error
    - Training experience E : A sequence of images and steering commands recorded while observing a human driver

Definition: A computer program which learns from experience is called a machine learning program or simply a learning program .

Classification of Machine Learning

Machine learning implementations are classified into four major categories, depending on the nature of the learning "signal" or "response" available to a learning system which are as follows:

A. Supervised learning:

Supervised learning is the machine learning task of learning a function that maps an input to an output based on example input-output pairs. The given data is labeled . Both classification and regression problems are supervised learning problems .

- Example — Consider the following data regarding patients entering a clinic . The data consists of the gender and age of the patients and each patient is labeled as "healthy" or "sick".

gender	age	label
M	48	sick
M	67	sick
F	53	healthy
M	49	sick
F	32	healthy
M	34	healthy

M	21	healthy

B. 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. Example: Consider the following data regarding patients entering a clinic. The data consists of the gender and age of the patients.

gender	age
M	48
M	67
F	53
M	49
F	34
M	21

As a kind of learning, it resembles the methods humans use to figure out that certain objects or events are from the same class, such as by observing the degree of similarity between objects. Some recommendation systems that you find on the web in the form of marketing automation are based on this type of learning.

Types of machine learning problems

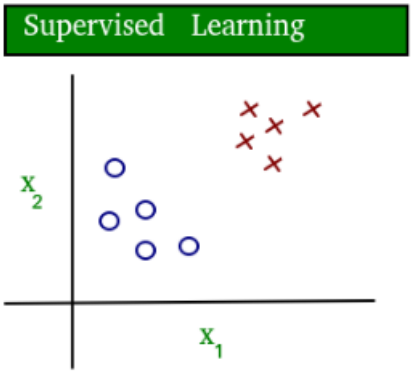
There are various ways to classify machine learning problems. Here, we discuss the most obvious ones.

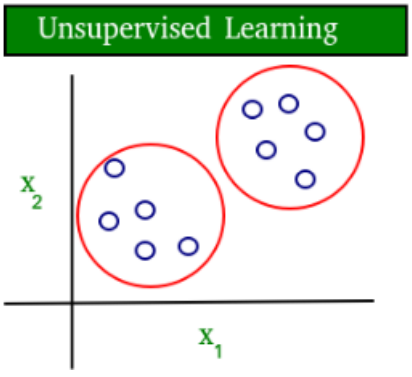
1. On basis of the nature of the learning "signal" or "feedback" available to a learning system

- Supervised learning: The model or algorithm is presented with example inputs and their desired outputs and then finding patterns and connections between the input and the output. The goal is to learn a general rule that maps inputs to outputs. The training process continues

until the model achieves the desired level of accuracy on the training data. Some real-life examples are:
    - Image Classification: You train with images/labels. Then in the future you give a new image expecting that the computer will recognize the new object.
    - Market Prediction/Regression: You train the computer with historical market data and ask the computer to predict the new price in the future.
- Unsupervised learning: No labels are given to the learning algorithm, leaving it on its own to find structure in its input. It is used for clustering population in different groups. Unsupervised learning can be a goal in itself (discovering hidden patterns in data).
    - Clustering: You ask the computer to separate similar data into clusters, this is essential in research and science.
    - High Dimension Visualization: Use the computer to help us visualize high dimension data.
    - Generative Models: After a model captures the probability distribution of your input data, it will be able to generate more data. This can be very useful to make your classifier more robust.

A simple diagram which clears the concept of supervised and unsupervised learning is shown below:

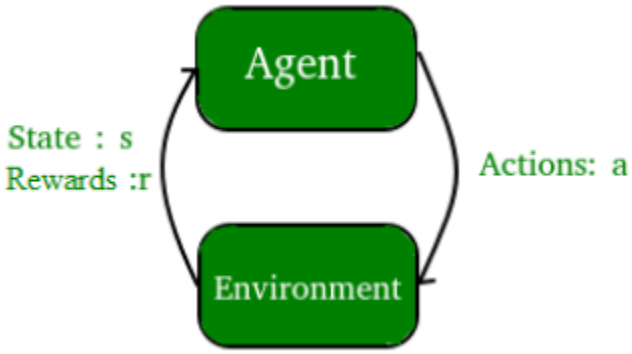




As you can see clearly, the data in supervised learning is labelled, where as data in unsupervised learning is unlabelled.

- Semi-supervised learning: Problems where you have a large amount of input data and only some of the data is labeled, are called semi-supervised learning problems. These problems sit in between both supervised and unsupervised learning. For example, a photo archive where only some of the images are labeled, (e.g. dog, cat, person) and the majority are unlabeled.

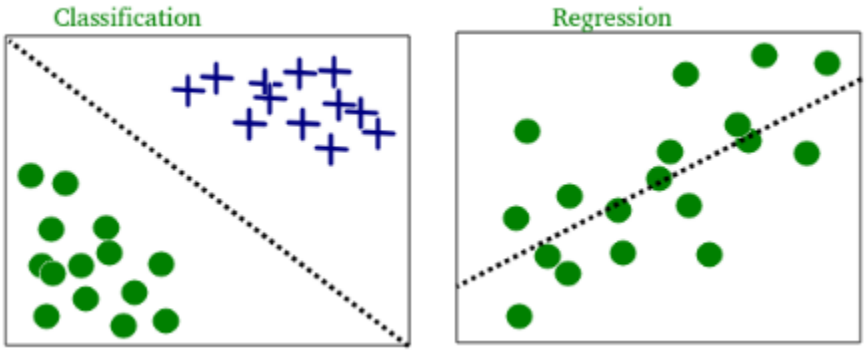
- Reinforcement learning: A computer program interacts with a dynamic environment in which it must perform a certain goal (such as driving a vehicle or playing a game against an opponent). The program is provided feedback in terms of rewards and punishments as it navigates its problem space.



2. Two most common use cases of Supervised learning are:

- Classification: Inputs are divided into two or more classes, and the learner must produce a model that assigns unseen inputs to one or more (multi-label classification) of these classes and predicting whether or not something belongs to a particular class. This is typically tackled in a supervised way. Classification models can be categorized in two groups: Binary classification and Multiclass Classification. Spam filtering is an example of binary classification, where the inputs are email (or other) messages and the classes are "spam" and "not spam".
- Regression: It is also a supervised learning problem, that predicts a numeric value and outputs are continuous rather than discrete. For example, predicting the stock prices using historical data.

An example of classification and regression on two different datasets is shown below:



3. Most common Unsupervised learning are:

- Clustering: Here, a set of inputs is to be divided into groups. Unlike in classification, the groups are not known beforehand, making this typically an unsupervised task. As you can see in the example below, the given dataset points have been divided into groups identifiable by the colors red, green and blue.
- Density estimation: The task is to find the distribution of inputs in some space.

- Dimensionality reduction: It simplifies inputs by mapping them into a lower-dimensional space. Topic modeling is a related problem, where a program is given a list of human language documents and is tasked to find out which documents cover similar topics.

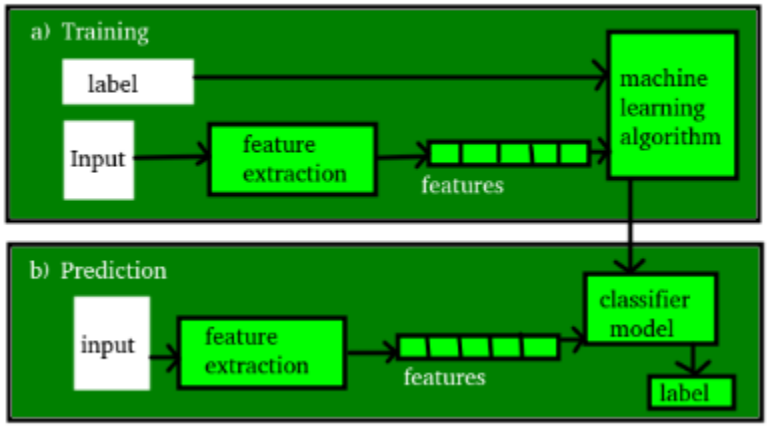
On the basis of these machine learning tasks/problems, we have a number of algorithms which are used to accomplish these tasks. Some commonly used machine learning algorithms are Linear Regression, Logistic Regression, Decision Tree, SVM(Support vector machines), Naive Bayes, KNN(K nearest neighbors), K-Means, Random Forest, etc. Note: All these algorithms will be covered in upcoming articles

Terminologies of Machine Learning

- Model A model is a specific representation learned from data by applying some machine learning algorithm. A model is also called hypothesis.
- Feature A feature is an individual measurable property of our data. A set of numeric features can be conveniently described by a feature vector. Feature vectors are fed as input to the model. For example, in order to predict a fruit, there may be features like color, smell, taste, etc. Note: Choosing informative, discriminating and independent features is a crucial step for effective algorithms. We generally employ a feature extractor to extract the relevant features from the raw data.
- Target (Label) A target variable or label is the value to be predicted by our model. For the fruit example discussed in the features section, the label with each set of input would be the name of the fruit like apple, orange, banana, etc.
- Training The idea is to give a set of inputs(features) and it's expected outputs(labels), so after training, we will have a model (hypothesis) that will then map new data to one of the categories trained on.
- Prediction Once our model is ready, it can be fed a set of inputs to which it will provide a predicted output(label). But make sure if the machine performs well on unseen data, then only we can say the machine performs well.

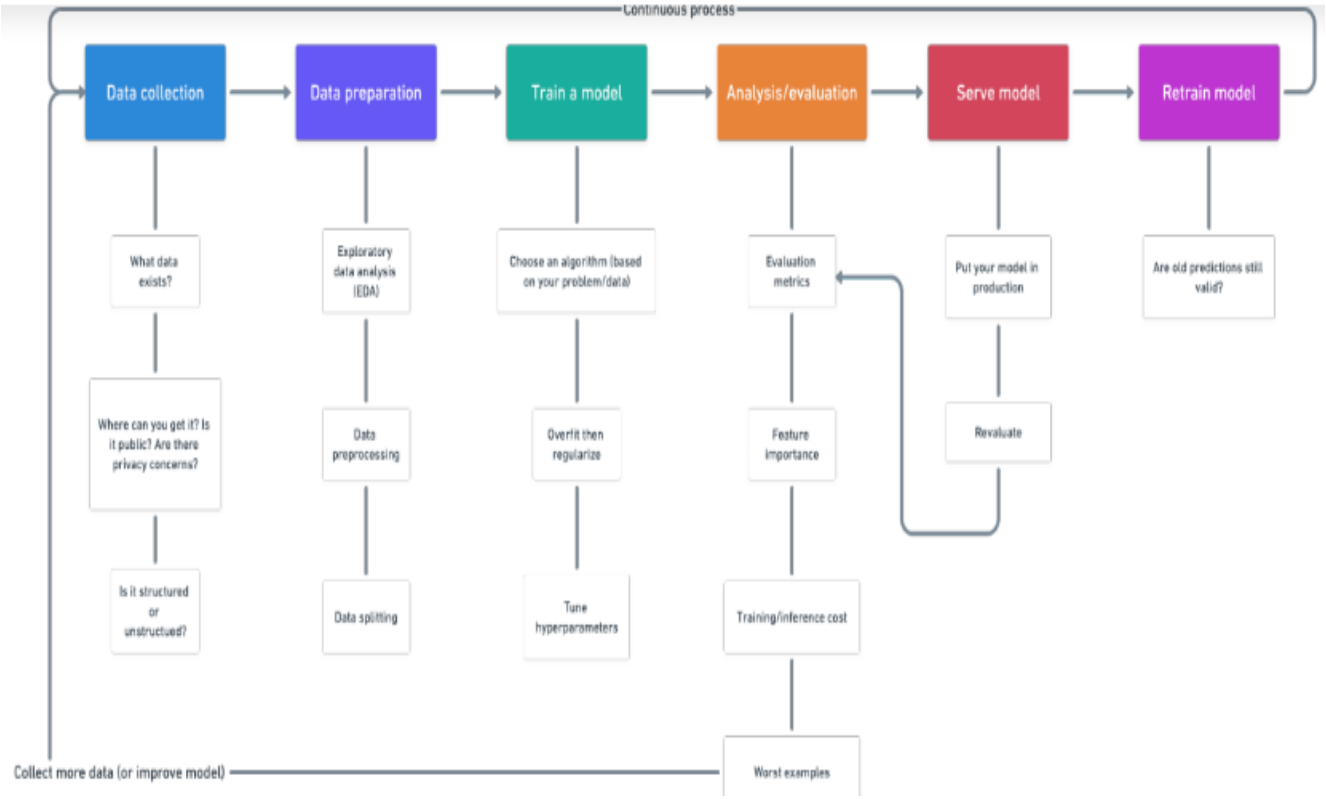
The figure shown below clears the above concepts:

The figure shown below clears the above concepts:



Preparing to Model:
(Steps to Complete a Machine Learning Project)

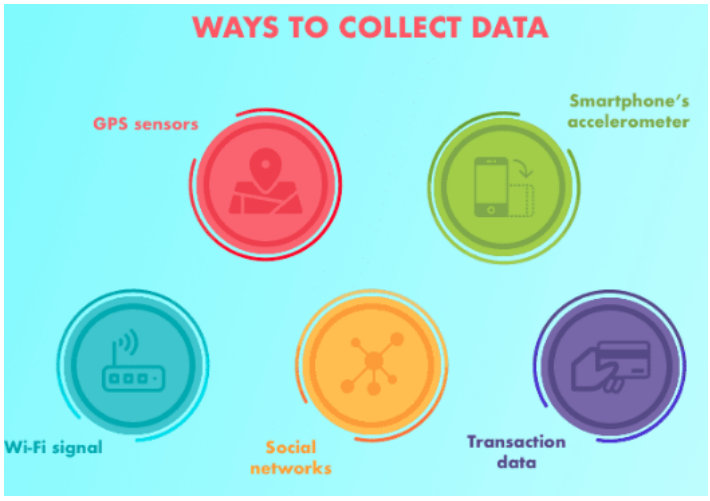
First, we have to collect the data according to our business needs. The next step is to clean the data like removing values, removing outliers, handling imbalanced datasets, changing categorical variables to numerical values, etc.
After that training of a model, use various machine learning and deep learning algorithms. Next, is model evaluation using different metrics like recall, f1 score, accuracy, etc. Finally, model deployment on the cloud and retrain a model.



1. Data Collection

- **Questions to ask?**

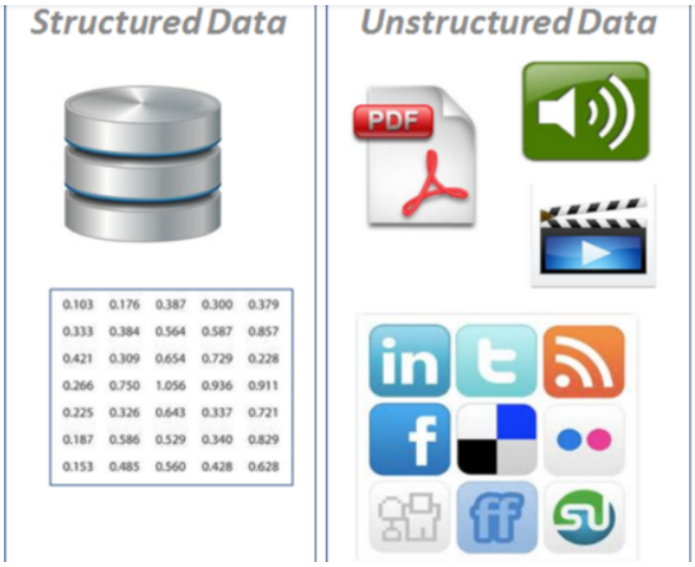
1. What kind of problem are we trying to solve?
2. What data sources already exist?
3. What privacy concerns are there?
4. Is the data public?
5. Where should we store the files?



- **Types of data**

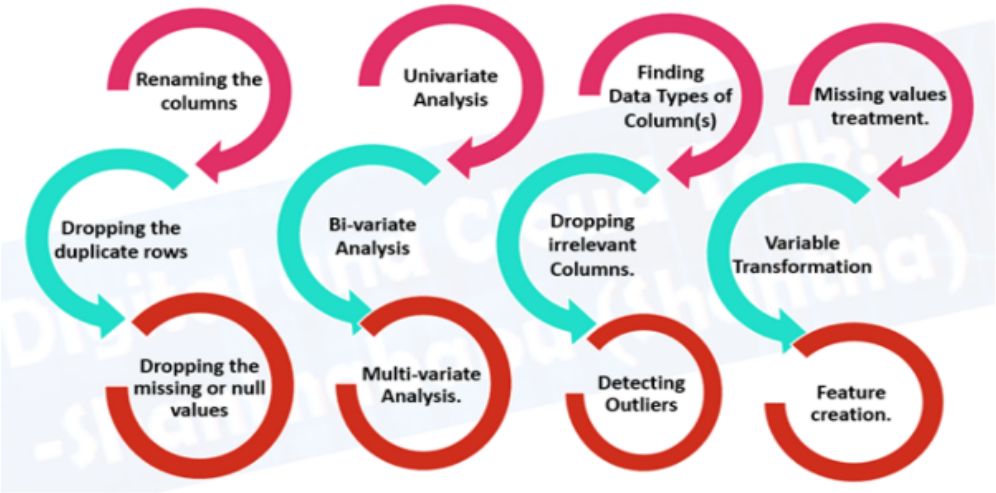
    1. **<u>Structured data</u>**: appears in tabulated format (rows and columns style, like what you'd find in an Excel spreadsheet). It contains different types of data, for example numerical, categorical, time series.

- ·**Nominal/categorical** – One thing or another (mutually exclusive). For example, for car scales, color is a category. A car may be blue but not white. An order does not matter.
- **Numerical**: Any continuous value where the difference between them matters. For example, when selling houses, $107,850 is more than $56,400.
- **Ordinal:** Data which has order but the distance between values is unknown. For example, a question such as, how would you rate your health from 1-5? 1 being poor, 5 being healthy. You can answer 1,2,3,4,5 but the distance between each value doesn't necessarily mean an answer of 5 is five times as good as an answer of 1.Time-series: Data across time. For example, the historical sale values of Bulldozers from 2012-2018.
- **Time-series**: Data across time. For example, the historical sale values of Bulldozers from 2012-2018.
    1. **<u>Unstructured data</u>***:* Data with no rigid structure(images, video, speech, natural language text)



2. Data preparation

- **<u>Exploratory data analysis(EDA), learning about the data you're working with</u>**
    1. What are the feature variables (input) and the target variable (output) For example, for predicting heart disease, the feature variables may be a person's age, weight, average heart rate, and level of physical activity. And the target variable will be whether or not they have a disease
    2. What kind of do you have? Structured, unstructured, numerical, time series. Are there missing values? Should you remove them or fill them feature imputation.
    3. Where are the outliers? How many of them are there? Why are they there? Are there any questions you could ask a domain expert about the data? For example, would a heart disease physician be able to shed some light on your heart disease dataset?

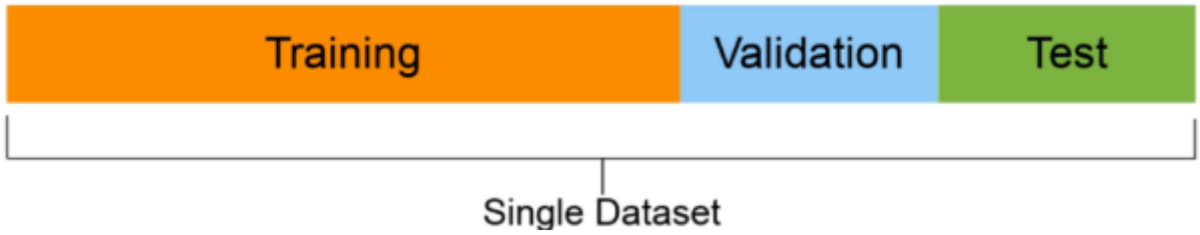


- **<u>Data preprocessing, preparing your data to be modelled.</u>**
- ***Feature imputation:*** filling missing values (a machine learning model can't learn on data that's isn't there)
    1. ***Single imputation:*** Fill with mean, a median of the column.

2. ***Multiple imputations:*** Model other missing values and with what your model finds.
3. ***KNN (k-nearest neighbors):*** Fill data with a value from another example that is similar.
4. Many more, such as random imputation, last observation carried forward (for time series), moving window, and most frequent.

- **Feature encoding** (turning values into numbers). A machine learning model requires all values to be numerical)
- ***One hot encoding:*** Turn all unique values into lists of 0's and 1's where the target value is 1 and the rest are 0's. For example, when a car colors green, red blue, a green, a car's color future would be represented as [1, 0, and 0] and a red one would be [0, 1, and 0].
- ***Label Encoder:*** Turn labels into distinct numerical values. For example, if your target variables are different animals, such as dog, cat, bird, these could become 0, 1, and 2, respectively.
- ***Embedding encoding:*** Learn a representation amongst all the different data points. For example, a language model is a representation of how different words relate to each other. Embedding is also becoming more widely available for structured (tabular) data.
- ***Feature normalization (scaling) or standardization:*** When you're numerical variables are on different scales (e.g. number_of_bathroom is between 1 and 5 and size_of_land between 500 and 20000 sq. feet), some machine learning algorithms don't perform very well. Scaling and standardization help to fix this.
- **Feature engineering:** transform data into (potentially) more meaningful representation by adding in domain knowledge
    1. Decompose
    2. Discretization: turning large groups into smaller groups
    3. Crossing and interaction features: combining two or more features
    4. The indicator features: using other parts of the data to indicate something potentially significant
- **Feature selection:** selecting the most valuable features of your dataset to model. Potentially reducing overfitting and training time(less overall data and less redundant data to train on) and improving accuracy.
    1. ***Dimensionality reduction:*** A common dimensionality reduction method, PCA or principal component analysis taken a large number of dimensions (features) and uses linear algebra to reduce them to fewer dimensions. For example, say you have 10 numerical features, you could run PCA to reduce it down to 3.
    2. ***Feature importance (post modelling):*** Fit a model to a set of data, then inspect which features were most important to the results, remove the least important ones.
    3. ***Wrapper methods*** such as genetic algorithms and recursive feature elimination involve creating large subsets of feature options and then removing the ones which don't matter.
- ***Dealing with imbalances:*** does your data have 10,000 examples of one class but only 100 examples of another?
    1. Collect more data (if you can)
    2. Use the scikit-learn-contrib imbalanced- learn package
    3. ***Use SMOTE:*** synthetic minority over-sampling technique. It creates synthetic samples of your minor class to try and level the playing field.
    4. A helpful paper to look at is "Learning from imbalanced Data".

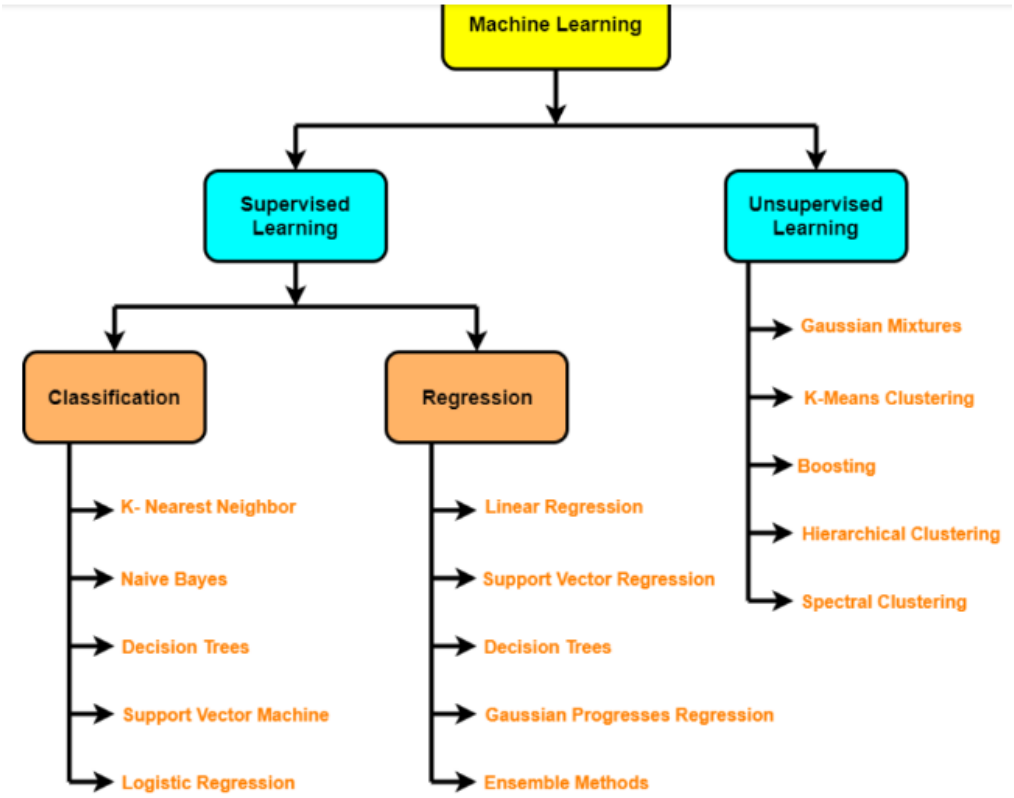
- **Data splitting**
    1. Training set (usually 70-80% of data): Model learns on this.
    2. Validation set (usually 10-15% of data): Model hyperparameters are tuned on this
    3. Test set (usually 10-15% of data): Models' final performance is evaluated on this. If you have done it right, hopefully, the results on the test set give a good indication of how the model should perform in the real world. Do not use this dataset to tune the model.



3. Train model on data(3 steps: Choose an algorithm, overfit the model, reduce overfitting with regularization)

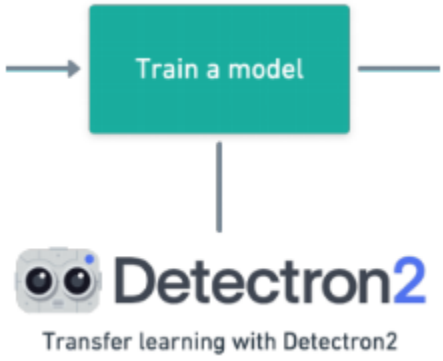
- **Choosing an algorithms**

    1. Supervised algorithms – Linear Regression, Logistic Regression, KNN, SVMs, Decision tree and Random forests, AdaBoost/Gradient Boosting Machine(boosting)
    2. Unsupervised algorithms- Clustering, dimensionality reduction(PCA, Autoencoders, t-SNE), An anomaly detection



- **Type of learning**

1. Batch learning
2. Online learning
3. Transfer learning
4. Active learning
5. Ensembling

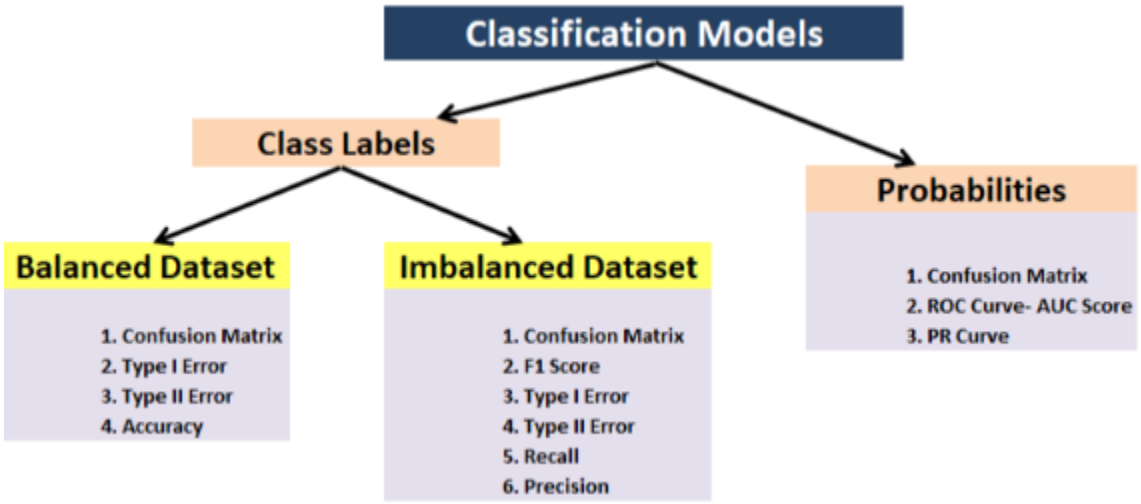


- **Underfitting** – happens when your model doesn't perform as well as you'd like on your data. Try training for a longer or more advanced model.
- **Overfitting**– happens when your validation loss starts to increase or when the model performs better on the training set than on the test set.
    1. **Regularization:** a collection of technologies to prevent/reduce overfitting (e.g. L1, L2, Dropout, Early stopping, Data augmentation, Batch normalization)

- **Hyperparameter Tuning** – run a bunch of experiments with different settings and see which works best

4. Analysis/Evaluation

- **Evaluation metrics**
    1. Classification- Accuracy, Precision, Recall, F1, Confusion matrix, Mean average precision (object detection)
    2. Regression – MSE,MAE,R^2
    3. Task-based metric – E.g. for the self-driving car, you might want to know the number of disengagements



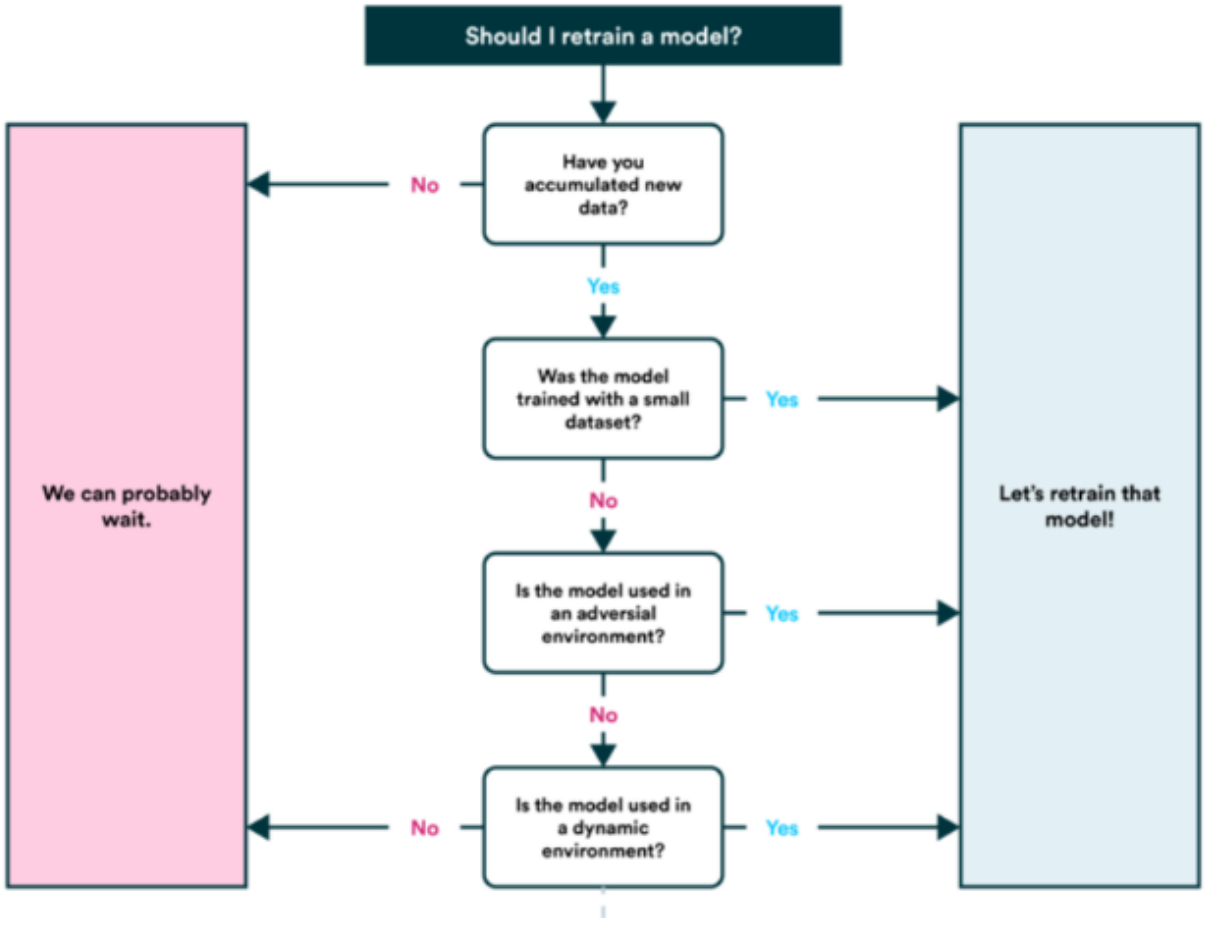
- **Feature importance**
- **Training/inference time/cost**
- **What if tool: how does my model compare to other models?**
- **Least confident examples: what does the model get wrong?**
- **Bias/variance trade-off**
-

5. Serve model (deploying a model)

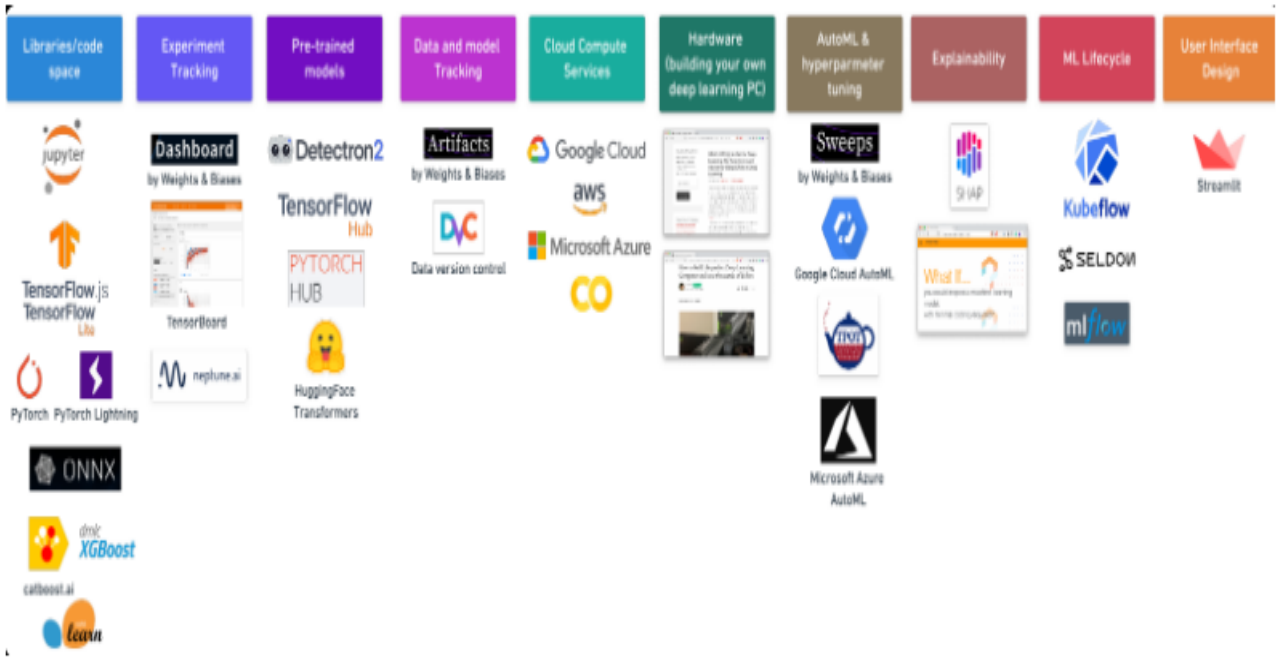
- Put the model into **production** and see how it goes.
- **Tools** you can use: TensorFlow Servinf, PyTorch Serving, Google AI Platform, Sagemaker
- **MLOps**: where software engineering meets machine learning, essentially all the technology required around a machine learning model to have it working in production

6. Retrain model

- See how the model performs after serving (or before serving) based on various evaluation metrics and revisit the above steps as required (remember, machine learning is very experimental, so this is where you'll want to track your data and experiments.
- You'll also find your model's predictions start to 'age' (usually not in a fine-wine style) or 'drift', as in when data sources change or upgrade(new hardware, etc.). This is when you'll want to retrain it.



7. Machine Learning Tools



UNIT II MODEL EVALUATION AND FEATURE ENGINEERING 9

Model Selection - Training Model - Model Representation and Interpretability - Evaluating Performance of a Model - Improving Performance of a Model - Feature Engineering: Feature Transformation - Feature Subset Selection

DataSet ML : https://www.ritchieng.com/machine-learning/datasets

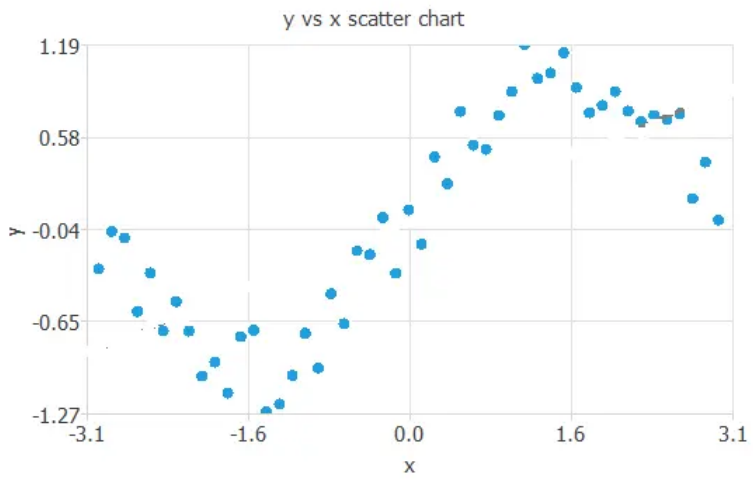
1. Model selection

The objective of model selection is to find the network architecture with the best generalization properties, that is, that which minimizes the error on the selected instances of the data set (the selection error).
There are two families of model selection algorithms:

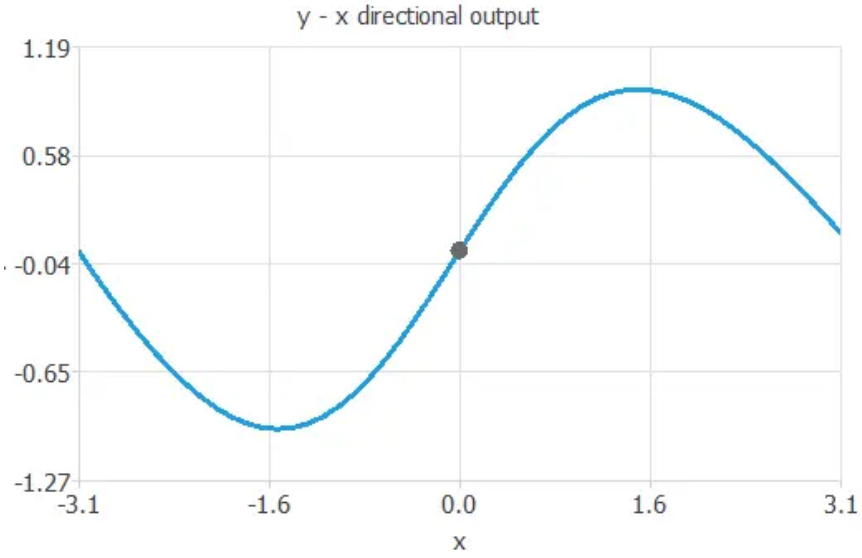
- **Neurons selection**
- **Inputs selection**

A) Neurons selection

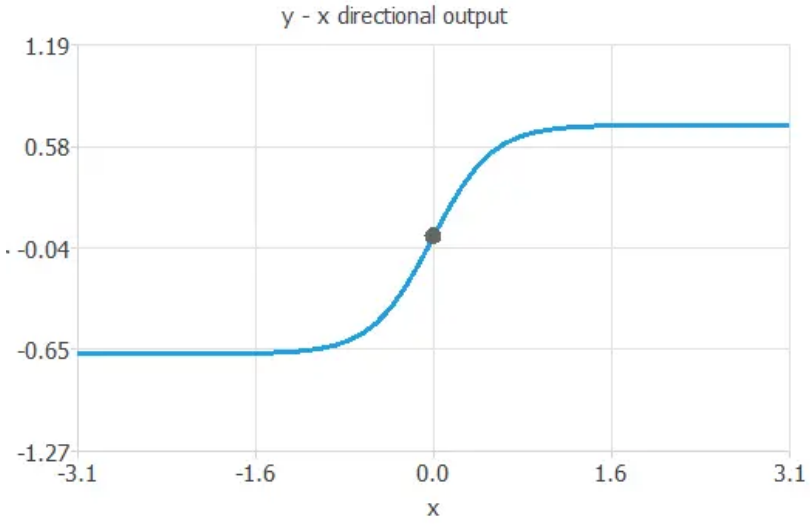
Two frequent problems in designing a neural network are called underfitting and overfitting. The best generalization is achieved by using a model with the most appropriate complexity to produce a good data fit.
To illustrate underfitting and overfitting, consider the following data set. It consists of data taken from a sine function to which white noise has been added.



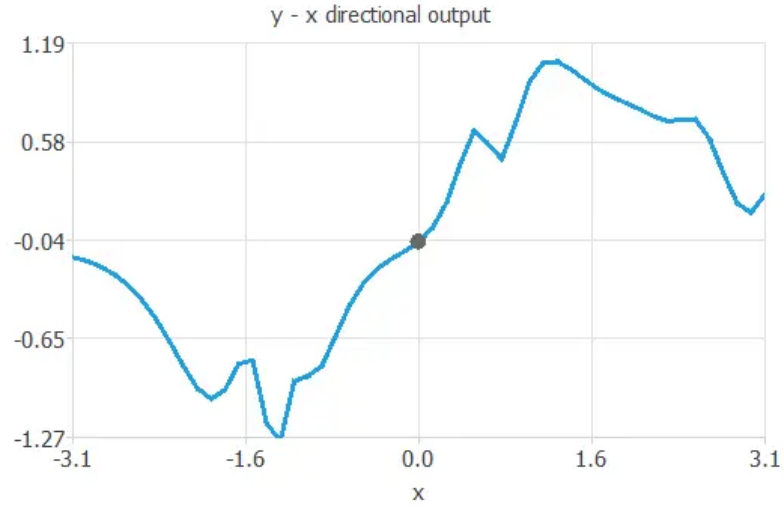
The best generalization is achieved by using a model whose complexity is the most appropriate to produce an adequate fit of the data. In this case, we use a neural network with one input (x), three hidden neurons, and one output (y).



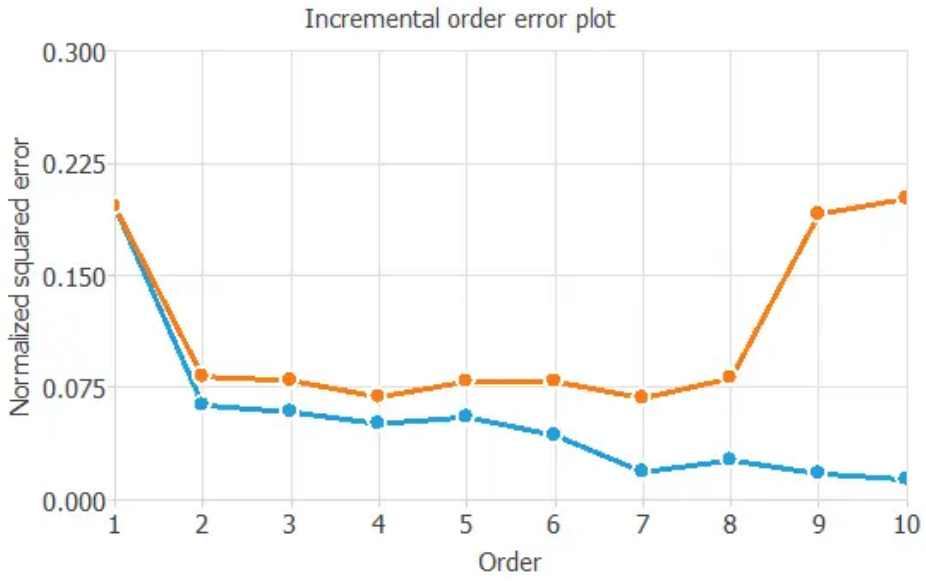
In this way, underfitting is defined as the effect of a selection error increasing due to a too-simple model. Here we have used one hidden neuron.



On the contrary, overfitting is defined as the effect of a selection error increasing due to a very complex model. In this case, we have used 10 hidden neurons.



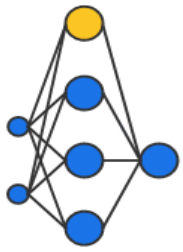
The error of a neural network on the training instances of the data set is called the training error. Similarly, the error on the selected instances is called the selection error.
The training error measures the ability of the neural network to fit the data that it sees. But the selection error measures the ability of the neural network to generalize to new data.
The following figure shows the training (blue) and selection (orange) errors as a function of the neural network complexity, represented by the number of hidden neurons.



As we can see, the more hidden neurons, the smaller the training error. However, for small and big complexities, the selection error is significant. In the first case, we have underfitting, and in the second one, overfitting. In our case, the neural network with the best generalization properties has four hidden neurons. Indeed, the selection error takes a minimum value at that point.
Neurons selection algorithms are used to find the neural network's complexity, yielding the best generalization properties. The most used algorithm is the growing neurons.

Growing neuons

Growing neurons is the most straightforward neurons selection algorithm. This method starts with small neurons and increases the complexity until any stopping criterion is met.



The algorithm returns the neural network with the optimal number of neurons obtained.

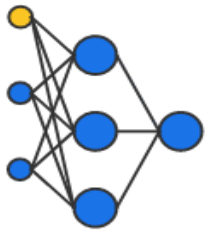
B) Inputs selection

Which features should you use to create a predictive model? This is a difficult question that may require in-depth knowledge of the problem domain.
Input selection algorithms automatically extract those features in the data set that provide the best generalization capabilities. They search for the subset of inputs that minimizes the selection error.
The most common input selection algorithms are:

- **Growing inputs.**
- **Pruning inputs.**
- **Genetic algorithm.**

Growing inputs

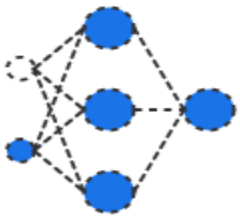
The growing inputs method calculates the correlation of every input with every output in the data set.
It starts with a neural network that only contains the most correlated input and calculates the selection error for that model.
It keeps adding the most correlated variables until the selection error increases.



The algorithm returns the neural network with the optimal subset of inputs found.

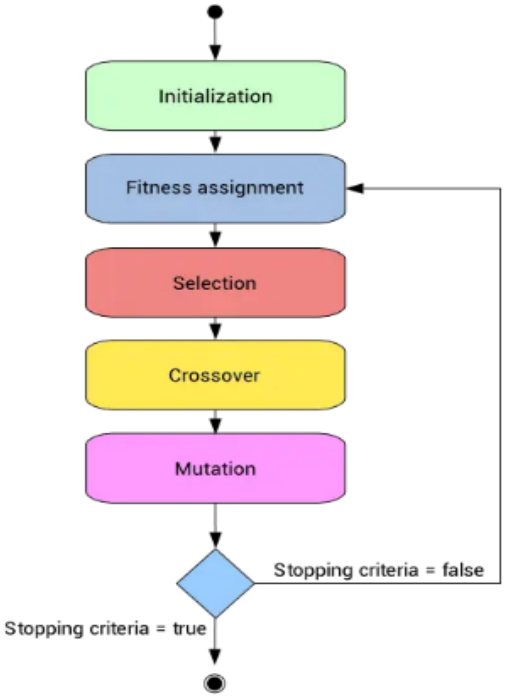
Pruning inputs

The pruning inputs method starts with all the inputs in the data set.
It keeps removing those inputs with the smallest correlation with the outputs until the selection error increases.



Genetic algorithm

A different class of inputs selection method is the genetic algorithm.
This is a stochastic method based on natural genetics and biological evolution mechanics.
Genetic algorithms usually include fitness assignment, selection, crossover, and mutation operators.



2. Evaluating Performance of a Model

Machine Learning Model Evaluation:

Machine learning continues to be an increasingly integral component of our lives, whether we're applying the techniques to research or business problems. Machine learning models ought to be able to give accurate predictions in order to create real value for a given organization.

While training a model is a key step, how the model generalizes on unseen data is an equally important aspect that should be considered in every machine learning pipeline. We need to know whether it actually works and, consequently, if we can trust its predictions. Could the model be merely memorizing the data it is fed with, and therefore unable to make good predictions on future samples, or samples that it hasn't seen before?

In this article, we explain the techniques used in evaluating how well a machine learning model generalizes to new, previously unseen data. We'll also illustrate how common model evaluation metrics are implemented for classification and regression problems using Python.

Model Evaluation Techniques:

The above issues can be handled by evaluating the performance of a machine learning model, which is an integral component of any data science project. Model evaluation aims to estimate the generalization accuracy of a model on future (unseen/out-of-sample) data.

Methods for evaluating a model's performance are divided into 2 categories:
namely, holdout and Cross-validation. Both methods use a test set (i.e data not seen by the

model) to evaluate model performance. It's not recommended to use the data we used to build the model to evaluate it. This is because our model will simply remember the whole training set, and will therefore always predict the correct label for any point in the training set. This is known as overfitting.

a) Holdout

The purpose of holdout evaluation is to test a model on different data than it was trained on. This provides an unbiased estimate of learning performance.
In this method, the dataset is randomly divided into three subsets:

1. Training set is a subset of the dataset used to build predictive models.
2. Validation set is a subset of the dataset used to assess the performance of the model built in the training phase. It provides a test platform for fine-tuning a model's parameters and selecting the best performing model. Not all modeling algorithms need a validation set.
3. Test set, or unseen data, is a subset of the dataset used to assess the likely future performance of a model. If a model fits to the training set much better than it fits the test set, overfitting is probably the cause.

The holdout approach is useful because of its speed, simplicity, and flexibility. However, this technique is often associated with high variability since differences in the training and test dataset can result in meaningful differences in the estimate of accuracy.

b) Cross-Validation

Cross-validation is a technique that involves partitioning the original observation dataset into a training set, used to train the model, and an independent set used to evaluate the analysis.
The most common cross-validation technique is k-fold cross-validation, where the original dataset is partitioned into k equal size subsamples, called folds. The k is a user-specified number, usually with 5 or 10 as its preferred value. This is repeated k times, such that each time, one of the k subsets is used as the test set/validation set and the other k-1 subsets are put together to form a training set. The error estimation is averaged over all k trials to get the total effectiveness of our model.
For instance, when performing five-fold cross-validation, the data is first partitioned into 5 parts of (approximately) equal size. A sequence of models is trained. The first model is trained using the first fold as the test set, and the remaining folds are used as the training set. This is repeated for each of these 5 splits of the data and the estimation of accuracy is averaged over all 5 trials to get the total effectiveness of our model.
As can be seen, every data point gets to be in a test set exactly once and gets to be in a training set k-1 times. This significantly reduces bias, as we're using most of the data for fitting, and it also significantly reduces variance, as most of the data is also being used in the test set. Interchanging the training and test sets also adds to the effectiveness of this method.
Model Evaluation Metrics:

Model evaluation metrics are required to quantify model performance. The choice of evaluation metrics depends on a given machine learning task (such as classification, regression, ranking, clustering, topic modeling, among others). Some metrics, such as precision-recall, are useful for multiple tasks. Supervised learning tasks such as classification and regression constitutes a majority of machine learning applications. In this article, we focus on metrics for these two supervised learning models.

Classification Metrics:
In this section we will review some of the metrics used in classification problems, namely:

- Classification Accuracy
- Confusion matrix
- Logarithmic Loss
- Area under curve (AUC)
- F-Measure

Classification Accuracy

Accuracy is a common evaluation metric for classification problems. It's the number of correct predictions made as a ratio of all predictions made. We use sklearn module to compute the accuracy of a classification task, as shown below:

```
 #import modules
import warnings
import pandas as pd
import numpy as np
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn import datasets
from sklearn.metrics import accuracy_score
#ignore warnings
warnings.filterwarnings('ignore')
# Load digits dataset
iris = datasets.load_iris()
# # Create feature matrix
X = iris.data
# Create target vector
y = iris.target
#test size
test_size = 0.33
#generate the same set of random numbers
seed = 7
#cross-validation settings
kfold = model_selection.KFold(n_splits=10, random_state=seed)
#Model instance
model = LogisticRegression()
```

```
#Evaluate model performance
scoring = 'accuracy'
results = model_selection.cross_val_score(model, X, y, cv=kfold,
scoring=scoring)
print('Accuracy -val set: %.2f%% (%.2f)' % (results.mean()*100,
results.std()))

#split data
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
test_size=test_size, random_state=seed)
#fit model
model.fit(X_train, y_train)
#accuracy on test set
result = model.score(X_test, y_test)
print("Accuracy - test set: %.2f%%" % (result*100.0))
```

The classification accuracy is 88% on the validation set.
By using cross-validation, we'd be "testing" our machine learning model in the "training" phase to check for overfitting and to get an idea about how our machine learning model will generalize to independent data (test data set).
Cross-validation techniques can also be used to compare the performance of different machine learning models on the same data set and can also be helpful in selecting the values for a model's parameters that maximize the accuracy of the model—also known as parameter tuning.

Confusion Matrix

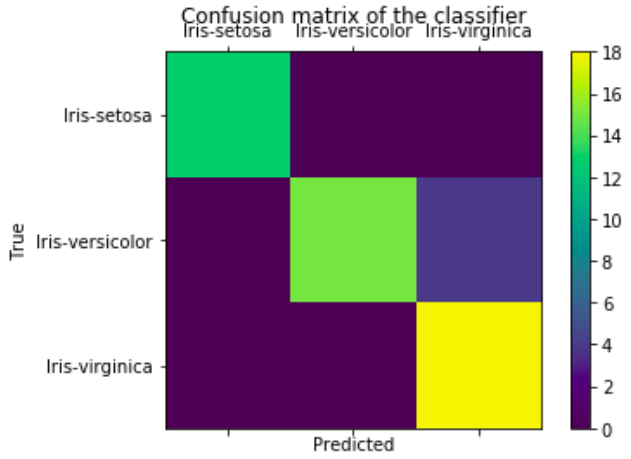
A confusion matrix provides a more detailed breakdown of correct and incorrect classifications for each class. We use the Iris dataset to classify and compute the confusion matrix for the predictions:

```
#import
module
s
        import warnings
        import pandas as pd
        from sklearn import model_selection
        from sklearn.linear_model import LogisticRegression
        from sklearn.metrics import confusion_matrix
        import matplotlib.pyplot as plt
        %matplotlib inline

        #ignore warnings
        warnings.filterwarnings('ignore')
        # Load digits dataset
        url = "http://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
        df = pd.read_csv(url)
```

```
# df = df.values
X = df.iloc[:,0:4]
y = df.iloc[:,4]
# print (y.unique())
#test size
test_size = 0.33
#generate the same set of random numbers
seed = 7
#Split data into train and test set.
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
test_size=test_size, random_state=seed)
#Train Model
model = LogisticRegression()
model.fit(X_train, y_train)
pred = model.predict(X_test)

#Construct the Confusion Matrix
labels = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
cm = confusion_matrix(y_test, pred, labels)
print(cm)
fig = plt.figure()
ax = fig.add_subplot(111)
cax = ax.matshow(cm)
plt.title('Confusion matrix of the classifier')
fig.colorbar(cax)
ax.set_xticklabels([''] + labels)
ax.set_yticklabels([''] + labels)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```

view rawmodel_evaluation_CM.py

```
[[13  0  0]
 [ 0 15  4]
 [ 0  0 18]]
```



The short explanation of how to interpret a confusion matrix is as follows: The diagonal elements represent the number of points for which the predicted label is equal to the true label, while anything off the diagonal was mislabeled by the classifier. Therefore, the higher the diagonal values of the confusion matrix the better, indicating many correct predictions.
In our case, the classifier predicted all the 13 setosa and 18 virginica plants in the test data perfectly. However, it incorrectly classified 4 of the versicolor plants as virginica.

Logarithmic Loss

Logarithmic loss (logloss) measures the performance of a classification model where the prediction input is a probability value between 0 and 1. Log loss increases as the predicted probability diverges from the actual label. The goal of machine learning models is to minimize this value. As such, smaller logloss is better, with a perfect model having a log loss of 0.

#Classification
LogLoss

```
import warnings
import pandas
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import log_loss

warnings.filterwarnings('ignore')
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabe
tes.data.csv"
dataframe = pandas.read_csv(url)
dat = dataframe.values
```

```
X = dat[:,:-1]
y = dat[:,-1]
seed = 7
#split data
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
test_size=test_size, random_state=seed)
model.fit(X_train, y_train)
#predict and compute logloss
pred = model.predict(X_test)
accuracy = log_loss(y_test, pred)
print("Logloss: %.2f" % (accuracy))
```

Area under Curve (AUC)

Area under ROC Curve is a performance metric for measuring the ability of a binary classifier to discriminate between positive and negative classes.

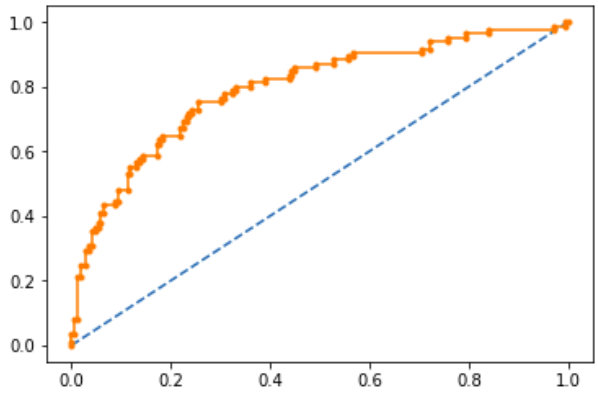
#Classification
Area under
curve

```
import warnings
import pandas
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score, roc_curve

warnings.filterwarnings('ignore')

url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabe
tes.data.csv"
dataframe = pandas.read_csv(url)
dat = dataframe.values
X = dat[:,:-1]
y = dat[:,-1]
seed = 7
#split data
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
test_size=test_size, random_state=seed)
model.fit(X_train, y_train)

# predict probabilities
probs = model.predict_proba(X_test)
# keep probabilities for the positive outcome only
probs = probs[:, 1]
```

```
auc = roc_auc_score(y_test, probs)
print('AUC - Test Set: %.2f%%' % (auc*100))

# calculate roc curve
fpr, tpr, thresholds = roc_curve(y_test, probs)
# plot no skill
plt.plot([0, 1], [0, 1], linestyle='--')
# plot the roc curve for the model
plt.plot(fpr, tpr, marker='.')
# show the plot
plt.show()
```

view rawmodel_evaluation_AUC.py

```
AUC - Test Set: 79.55%
```



In the example above, the AUC is relatively close to 1 and greater than 0.5. A perfect classifier will have the ROC curve go along the Y axis and then along the X axis.

F-Measure

F-measure (also F-score) is a measure of a test's accuracy that considers both the precision and the recall of the test to compute the score. Precision is the number of correct positive results divided by the total predicted positive observations. Recall, on the other hand, is the number of correct positive results divided by the number of all relevant samples (total actual positives).

```
import
warnings
        import pandas
        from sklearn import model_selection
        from sklearn.linear_model import LogisticRegression
        from sklearn.metrics import log_loss
        from sklearn.metrics import precision_recall_fscore_support as score, precision_score,
        recall_score, f1_score

        warnings.filterwarnings('ignore')
```

```
url =
"https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.da
ta.csv"
dataframe = pandas.read_csv(url)
dat = dataframe.values
X = dat[:,:-1]
y = dat[:,-1]
test_size = 0.33
seed = 7

model = LogisticRegression()
#split data
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
test_size=test_size, random_state=seed)
model.fit(X_train, y_train)
precision = precision_score(y_test, pred)
print('Precision: %f' % precision)
# recall: tp / (tp + fn)
recall = recall_score(y_test, pred)
print('Recall: %f' % recall)
# f1: tp / (tp + fp + fn)
f1 = f1_score(y_test, pred)
print('F1 score: %f' % f1)
```

view rawmodel_evaluation_f1.py

Regression Metrics

In this section we review 2 of the most common metrics for evaluating regression problems namely, Root Mean Squared Error and Mean Absolute Error.
The Mean Absolute Error (or MAE) is the sum of the absolute differences between predictions and actual values. On the other hand, Root Mean Squared Error (RMSE) measures the average magnitude of the error by taking the square root of the average of squared differences between prediction and actual observation.
The Python code snippet below shows how the two regression metrics can be implemented.

```
import
panda
s
    from sklearn import model_selection
    from sklearn.linear_model import LinearRegression
    from sklearn.metrics import mean_absolute_error, mean_squared_error
    from math import sqrt
    url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/housing.data"
    dataframe = pandas.read_csv(url, delim_whitespace=True)
    df = dataframe.values
    X = df[:,:-1]
```

```
y = df[:,-1]
seed = 7
model = LinearRegression()
#split data
X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,
test_size=test_size, random_state=seed)
model.fit(X_train, y_train)
#predict
pred = model.predict(X_test)
print("MAE test score:", mean_absolute_error(y_test, pred))
print("RMSE test score:", sqrt(mean_squared_error(y_test, pred)))
```

view rawmodel_evaluation_regression.py

3. Training Model

A machine learning training model is a process in which a machine learning (ML) algorithm is fed with sufficient training data to learn from.

ML models can be trained to benefit manufacturing processes in several ways. The ability of ML models to process large volumes of data can help manufacturers identify anomalies and test correlations while searching for patterns across the data feed. It can equip manufacturers with predictive maintenance capabilities and minimize planned and unplanned downtime.

What Is Model Training In Machine Learning?

A training model is a dataset that is used to train an ML algorithm. It consists of the sample output data and the corresponding sets of input data that have an influence on the output. The training model is used to run the input data through the algorithm to correlate the processed output against the sample output. The result from this correlation is used to modify the model. This iterative process is called “model fitting”. The accuracy of the training dataset or the validation dataset is critical for the precision of the model.

Model training in machine language is the process of feeding an ML algorithm with data to help identify and learn good values for all attributes involved. There are several types of machine learning models, of which the most common ones are supervised and unsupervised learning.

Supervised learning is possible when the training data contains both the input and output values. Each set of data that has the inputs and the expected output is called a supervisory signal. The training is done based on the deviation of the processed result from the documented result when the inputs are fed into the model.

Unsupervised learning involves determining patterns in the data. Additional data is then used to fit patterns or clusters. This is also an iterative process that improves the accuracy based on the correlation to the expected patterns or clusters. There is no reference output dataset in this method.

Creating A Model In Machine Learning

There are 7 primary steps involved in creating a machine learning model. Here is a brief summarized overview of each of these steps:

Defining The Problem

Defining the problem statement is the first step towards identifying what an ML model should achieve. This step also enables recognizing the appropriate inputs and their respective outputs;

Questions like “what is the main objective?”, “what is the input data?” and “what is the model trying to predict?” must be answered at this stage.

Data Collection

After defining the problem statement, it is necessary to investigate and gather data that can be used to feed the machine. This is an important stage in the process of creating an ML model because the quantity and quality of the data used will decide how effective the model is going to be. Data can be gathered from pre-existing databases or can be built from the scratch

Preparing The Data

The data preparation stage is when data is profiled, formatted and structured as needed to make it ready for training the model. This is the stage where the appropriate characteristics and attributes of data are selected. This stage is likely to have a direct impact on the execution time and results. This is also at the stage where data is categorized into two groups – one for training the ML model and the other for evaluating the model. Pre-processing of data by normalizing, eliminating duplicates and making error corrections is also carried out at this stage.

Assigning Appropriate Model / Protocols

Picking and assigning a model or protocol has to be done according to the objective that the ML model aims to achieve. There are several models to pick from, like linear regression, k-means and bayesian. The choice of models largely depends on the type of data that is being used. For instance, image processing convolutional neural networks would be the ideal pick and k-means would work best for segmentation.

Training The Machine Model Or “The Model Training”

This is the stage where the ML algorithm is trained by feeding datasets. This is the stage where the learning takes place. Consistent training can significantly improve the prediction rate of the ML model. The weights of the model must be initialized randomly. This way the algorithm will learn to adjust the weights accordingly.

Evaluating And Defining Measure Of Success

The machine model will have to be tested against the “validation dataset”. This helps assess the accuracy of the model. Identifying the measures of success based on what the model is intended to achieve is critical for justifying correlation.

Parameter Tuning

Selecting the correct parameter that will be modified to influence the ML model is key to attaining accurate correlation. The set of parameters that are selected based on their influence on the model architecture are called hyperparameters. The process of identifying the hyperparameters by tuning the model is called parameter tuning. The parameters for correlation should be clearly defined in a manner in which the point of diminishing returns for validation is as close to 100% accuracy as possible.

How Long Does It Take To Train A Machine Learning Model?

There is no definite period or a prefixed set of iterations for training an ML model. The factors influencing the duration of training could be the quality of training data, proper definition of the measures of success and complexity of model selection. Factors like method of training, allocation of weights and complexity of the model also play an important role. Other factors that are extraneous to the data or the models like computing power and skilled resources can also have an impact on the training duration. There is always a scope for optimizing training a model as the number of parameters influencing its duration are too many.

4. Model Representation:

Non-linear hypothesis, neurons and the brain, model representation, and multi-class classification

- Origins
  - Algorithms that try to mimic the brain
- Recent resurgence
  - State-of-the-art techniques for many applications
- The "one learning algorithm" hypothesis
  - Auditory cortex handles hearing
    - Re-wire to learn to see
  - Somatosensory cortex handles feeling
    - Re-wire to learn to see
  - Plug in data and the brain will learn accordingly

Neural Networks:

a. Model Representation I

- Neuron in the brain
  - Many neurons in our brain
  - Dendrite: receive input
  - Axon: produce output
    - When it sends a message through the Axon to another neuron
    - It sends to another neuron's Dendrite
- Neuron model: logistic unit
  - Yellow circle: body of neuron
  - Input wires: dendrites
  - Output wire: axon
- Neural Network
  - 3 Layers
    - 1 Layer: input layer
    - 2 Layer: hidden layer
      - Unable to observe values
      - Anything other than input or output layer
    - 3 Layer: output layer
    - We calculate each of the layer-2 activations based on the input values with the bias term (which is equal to 1)
      - i.e. x0 to x3
      - We then calculate the final hypothesis (i.e. the single node in layer 3) using exactly the same logic, except in input is not x values, but the activation values from the preceding layer
    - The activation value on each hidden unit (e.g. a12) is equal to the sigmoid function applied to the linear combination of inputs
      - Three input units
      - $\Theta(1)$ is the matrix of parameters governing the mapping of the input units to hidden units
        - $\Theta(1)$ here is a [3 x 4] dimensional matrix

        - Three hidden units
            - Then Θ(2) is the matrix of parameters governing the mapping of the hidden layer to the output layer
                - Θ(2) here is a [1 x 4] dimensional matrix (i.e. a row vector)
        - Every input/activation goes to every node in following layer
            - Which means each "layer transition" uses a matrix of parameters with the following significance
                - j (first of two subscript numbers)= ranges from 1 to the number of units in layer l+1
                - i (second of two subscript numbers) = ranges from 0 to the number of units in layer l
                - l is the layer you're moving FROM
- Notation

b. Model Representation II

- Here we'll look at how to carry out the computation efficiently through a vectorized implementation. We'll also consider why neural networks are good and how we can use them to learn complex non-linear things
- Forward propagation: vectorized implementation
    - g applies sigmoid-function element-wise to z
    - This process of calculating H(x) is called forward propagation
        - Worked out from the first layer
        - Starts off with activations of input unit
        - Propagate forward and calculate the activation of each layer sequentially
- Similar to logistic regression if you leave out the first layer
    - Only second and third layer
    - Third layer resembles a logistic regression node
    - The features in layer 2 are calculated/learned, not original features
    - Neural network, learns its own features
        - The features a's are learned from x's
        - It learns its own features to feed into logistic regression
        - Better hypothesis than if we were constrained with just x1, x2, x3
        - We can have whatever features we want to feed to the final logistic regression function
        - Implemention in Octave for a2
            - `a2 = sigmoid (Theta1 * x);`
- Other network architectures
    - Layer 2 and 3 are hidden layers

***Neural Network Application*:**

a. Examples and Intuitions I

- XOR/XNOR
    - XOR: or
    - XNOR: not or
- AND function
    - Outputs 1 only if x1 and x2 are 1

    - Draw a table to determine if OR or AND
- NAND function
    - NOT AND
- OR function

b. Examples and Intuitions II

- NOT function
    -
- XNOR function
    - NOT XOR
    - NOT an exclusive or
        - Hence we would want
            - AND
            - Neither

c. Multi-class Classification

- Example: identify 4 classes
    - You would want a 4 x 1 vector for h_theta(X)
    - 4 logistic regression classifiers in the output layer
    - There will be 4 output
    - y would be a 4 x 1 vector instead of an integer

UNIT III BAYESIAN LEARNING

Basic Probability Notation- Inference – Independence - Bayes' Rule.

Bayesian Learning: Maximum Likelihood and Least Squared error hypothesis- Maximum Likelihood hypotheses for predicting probabilities- Minimum description Length principle -Bayes optimal classifier – Naïve Bayes classifier - Bayesian Belief networks -EM algorithm.

Features of Bayesian learning methods:

• Each observed training example can incrementally **decrease or increase the estimated probability that a hypothesis is correct**. – This provides a more flexible approach to learning than algorithms that completely **eliminate a hypothesis if it is found to be inconsistent** with any single example.

• **Prior knowledge** can be combined with **observed data** to determine the **final probability of a hypothesis**. In **Bayesian learning**, prior knowledge is provided by asserting – a prior probability for each candidate hypothesis, and – a probability distribution over observed data for each possible hypothesis.

• Bayesian methods can accommodate hypotheses that make **probabilistic predictions**

• **New instances** can be classified by combining the **predictions of multiple hypotheses**, weighted by their probabilities.

• Even in cases where Bayesian methods prove computationally intractable, they can provide a standard of optimal decision making against which other practical methods can be measured.

Some notation:

D: the training data

H: the set of all hypotheses

h: a hypothesis $h \in H$

P(h): the prior probability of h: the initial probability that hypothesis h holds, before we have observed the training data

P(D): the prior probability that training data D will be observed

P(D | h): the probability of observing data D given some world in which h holds

P(x | y): the probability of x occurring given that y has been observed

P(h | D): the posterior probability of h given D: the probability that h holds given the observed training data D

Basic Probability Notation:

- **Prior Probability**
- **Conditional Probability**

Basic probability notation

- **Prior probability** :We will use the notation P(A) for the unconditional or prior probability that the proposition A is true.
- For example, if Cavity denotes the proposition that a particular patient has a cavity, P(Cavity) = 0.1 means that in the absence of any other information, the agent will assign a probability of 0.1(a 10% chance)
- It is important to remember that P(A) can only be used when there is no other information. As soon as some new information B is known, we have to reason with the **conditional probability** of A given B instead of P(A) to the event of the patient's having a cavity.
- Propositions can also include equalities involving so-called random variables.
- For example, if we are concerned about the random variable Weather,

we might have P(Weather = Sunny) = 0.7

P(Weather = Rain) = 0.2

P(Weather= Cloudy) = 0.08

P(Weather = Snow) = 0.02

Each random variable X has a domain of possible values (x1,...,xn) that it can take on.

- We can view proposition symbols as random variables as well, if we assume that they have a domain [true,false).

- Thus, the expression P(Cavity) can be viewed as shorthand for P(Cavity = true).
- Similarly, P(->Cavity) is shorthand for P(Cavity =false).
- Sometimes, we will want to talk about the probabilities of all the possible values of a random variable. In this case, we will use an expression such as P(Weather)
- for example, we would write P(Weather) = (0.7,0.2,0.08,0.02)

This statement defines a probability distribution

- We can also use logical connectives to make more complex sentences and assign probabilities to them.

For example, P(Cavity A ¬Insured)

Conditional probability:

- Once the agent has obtained some evidence concerning the previously unknown propositions making up the domain, prior probabilities are no longer applicable.

Instead, we use conditional or posterior probabilities, with the notation P(A|B)

- This is read as "the probability of A given that all we know is B."
- P(B|A) means "Event B given Event A"
- In other words, event A has already happened, now what is the chance of event B?
- P(B|A) is also called the "Conditional Probability" of B given A.

Ex:Drawing 2 Kings from a Deck

- **Event A is drawing a King first, and Event B is drawing a King second.**
- For the first card the chance of drawing a King is 4 out of 52 (there are 4 Kings in a deck of 52 cards):
- P(A) = 4/52
- But after removing a King from the deck the probability of the 2nd card drawn is less likely to be a King (only 3 of the 51 cards left are Kings):
- P(B|A) = 3/51

And so: P(A and B) = P(A) x P(B|A) = (4/52) x (3/51) = 12/2652 = 1/221

- So the chance of getting 2 Kings is 1 in 221, or about 0.5

Inference by enumeration

- A simple method for **probabilistic inference** uses observed evidence for computation of posterior probabilities.
- Start with the joint probability distribution:

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	.108	.012	.072	.008
¬ *cavity*	.016	.064	.144	.576

- For any proposition φ, sum the atomic events where it is true: $P(\varphi) = \Sigma_{\omega:\omega \models \varphi} P(\omega)$

- Start with the joint probability distribution:

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	.108	.012	.072	.008
¬ *cavity*	.016	.064	.144	.576

- For any proposition φ, sum the atomic events where it is true: $P(\varphi) = \Sigma_{\omega:\omega \models \varphi} P(\omega)$
- P(*toothache*) = 0.108 + 0.012 + 0.016 + 0.064 = 0.2

- Start with the joint probability distribution:

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	.108	.012	.072	.008
¬ *cavity*	.016	.064	.144	.576

- For any proposition φ, sum the atomic events where it is true: $P(\varphi) = \Sigma_{\omega:\omega \models \varphi} P(\omega)$
- P(*toothache* ∨ *cavity*) = 0.108 + 0.012 + 0.016 + 0.064 + 0.072 + 0.008 = 0.28

- Start with the joint probability distribution:

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	.108	.012	.072	.008
¬ *cavity*	.016	.064	.144	.576

- Conditional probabilities:

$$P(\neg cavity \mid toothache) = \frac{P(\neg cavity \wedge toothache)}{P(toothache)} = \frac{0.016+0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4$$

Marginalization

- One particularly common task is to extract the distribution over some subset of variables or a single variable.
- For example, adding the entries in the first row gives the **unconditional probability** of *cavity*:

$$P(cavity) = 0.108+0.012+0.072+0.008 = 0.2$$

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	.108	.012	.072	.008
¬ *cavity*	.016	.064	.144	.576

23

Marginalization

- This process is called **marginalization** or **summing out**, because the variables other than *Cavity* are summed out.
- General marginalization rule for any sets of variables **Y** and **Z**:

$$\mathbf{P}(\mathbf{Y}) = \Sigma_{\mathbf{z}} \mathbf{P}(\mathbf{Y}, \mathbf{z})$$

- A distribution over **Y** can be obtained by summing out all the other variables from any joint distribution containing **Y**.

Typically, we are interested in:
the posterior joint distribution of the query variables **X**
given specific values **e** for the evidence variables **E**.

Let the hidden variables be **Y**.

Then the required summation of joint entries is done by summing out the hidden variables:

$$P(X \mid E = e) = P(X,E = e) / P(e) = \Sigma_y P(X,E = e, Y = y) / P(e)$$

- **X**, **E** and **Y** together exhaust the set of random variables.

Normalization

- $P(cavity \mid toothache) = \frac{P(cavity \wedge toothache)}{P(toothache)} = \frac{0.108+0.012}{0.108 + 0.012 + 0.016 + 0.064}$

- $P(\neg cavity \mid toothache) = \frac{P(\neg cavity \wedge toothache)}{P(toothache)} = \frac{0.016+0.064}{0.108 + 0.012 + 0.016 + 0.064}$

- Notice that in these two calculations the term *1/P(toothache)* remains constant, no matter which value of *Cavity* we calculate.

26

- The denominator can be viewed as a normalization constant α for the distribution **P**(*Cavity* | *toothache*), ensuring it adds up to 1.
- With this notation and using marginalization, we can write the two preceding equations in one:

P(*Cavity* | *toothache*) = α **P**(*Cavity*,*toothache*)
= α [**P**(*Cavity*,*toothache*,*catch*) + **P**(*Cavity*,*toothache*,¬ *catch*)]
= α [<0.108,0.016> + <0.012,0.064>]
= α <0.12,0.08> = <0.6,0.4>

27

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	**.108**	**.012**	**.072**	**.008**
¬ *cavity*	**.016**	**.064**	**.144**	**.576**

P(*Cavity* | *toothache*) = α **P**(*Cavity*,*toothache*)
= α [**P**(*Cavity*,*toothache*,*catch*) + **P**(*Cavity*,*toothache*,¬ *catch*)]
= α [<0.108,0.016> + <0.012,0.064>]
= α <0.12,0.08> = <0.6,0.4>

General idea: compute distribution on query variable by fixing evidence variables and summing over hidden variables

Inference by enumeration

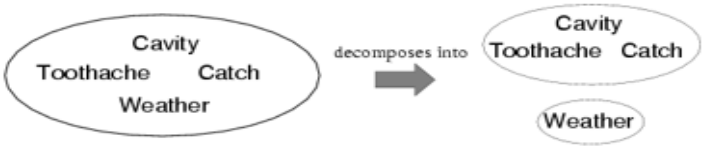
- Obvious problems:
  - Worst-case time complexity: $O(d^n)$ where d is the largest arity and n is the number of variables
  - Space complexity: $O(d^n)$ to store the joint distribution
  - How to define the probabilities for $O(d^n)$ entries, when variables can be hundreds or thousand?
- It quickly becomes completely impractical to define the vast number of probabilities required.

Independence

- Independence between propositions a and b can be written as:
  $P(a \mid b) = P(a)$ or $P(b \mid a) = P(b)$ or $P(a \wedge b) = P(a)\,P(b)$
- Independence assertions are usually based on knowledge of the domain.
- As we have seen, they can dramatically reduce the amount of information necessary to specify the full joint distribution.
- If the complete set of variables can be divided into independent subsets, then the full joint can be *factored* into separate joint distributions on those subsets.
- For example, the joint distribution on the outcome of n independent coin flips, $P(C_1, \ldots, C_n)$, can be represented as the product of n single-variable distributions $P(C_i)$.

Independence

- *A* and *B* are independent iff
  $\mathbf{P}(A|B) = \mathbf{P}(A)$ or $\mathbf{P}(B|A) = \mathbf{P}(B)$ or $\mathbf{P}(A, B) = \mathbf{P}(A)\,\mathbf{P}(B)$



$\mathbf{P}(Toothache, Catch, Cavity, Weather)$
$= \mathbf{P}(Toothache, Catch, Cavity)\,\mathbf{P}(Weather)$

- 32 entries reduced to 12
- For *n* independent biased coins, $O(2^n) \rightarrow O(n)$
- **Absolute independence** powerful but rare
- Dentistry is a large field with hundreds of variables, none of which are independent. What to do?

Conditional independence

- $\mathbf{P}(Toothache, Cavity, Catch)$ has $2^3 - 1$ (because the numbers must sum to 1) = 7 independent entries

- If I have a cavity, the probability that the probe catches in it doesn't depend on whether I have a toothache:
  $\mathbf{P}(catch \mid toothache, cavity) = \mathbf{P}(catch \mid cavity)$

- The same independence holds if I haven't got a cavity:
  $\mathbf{P}(catch \mid toothache, \neg cavity) = \mathbf{P}(catch \mid \neg cavity)$

- *Catch* is conditionally independent of *Toothache* given *Cavity*:
  $\mathbf{P}(Catch \mid Toothache, Cavity) = \mathbf{P}(Catch \mid Cavity)$

- Equivalent statements:
  $\mathbf{P}(Toothache \mid Catch, Cavity) = \mathbf{P}(Toothache \mid Cavity)$
  $\mathbf{P}(Toothache, Catch \mid Cavity) = \mathbf{P}(Toothache \mid Cavity)\,\mathbf{P}(Catch \mid Cavity)$

- Full joint distribution using product rule:
  P(*Toothache, Catch, Cavity*)
  = **P**(*Toothache* | *Catch, Cavity*) **P**(*Catch, Cavity*)
  = **P**(*Toothache* | *Catch, Cavity*) **P**(*Catch* | *Cavity*) **P**(*Cavity*)
  = **P**(*Toothache* | *Cavity*) **P**(*Catch* | *Cavity*) **P**(*Cavity*)

The resultant three smaller tables contain 5 independent entries ($2*(2^1-1)$ for each conditional probability distribution and 2^1-1 for the prior on *Cavity*)

- In most cases, the use of conditional independence reduces the size of the representation of the joint distribution from exponential in *n* to linear in *n*.
- Conditional independence is our most basic and robust form of knowledge about uncertain environments.

Bayes Rule

Bayes rule provides us with a way to update our **beliefs** based on the arrival of new, relevant pieces of **evidence**. For example, if we were trying to provide the probability that a given person has cancer, we would initially just say it is whatever percent of the population has cancer. However, given additional evidence such as the fact that the person is a smoker, we can update our probability, since the probability of having cancer is higher given that the person is a smoker. This allows us to utilize prior knowledge to improve our probability estimations.

The Rule

The below equation is Bayes rule:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

The rule has a very simple derivation that directly leads from the relationship between joint and conditional probabilities. First, note that P(A,B) = P(A|B)P(B) = P(B,A) = P(B|A)P(A). Next, we can set the two terms involving conditional probabilities equal to each other, so P(A|B)P(B) = P(B|A)P(A), and finally, divide both sides by P(B) to arrive at Bayes rule.

In this formula, **A** is the event we want the probability of, and **B** is the new evidence that is related to A in some way.

P(A|B) is called the **posterior**; this is what we are trying to estimate. In the above example, this would be the “probability of having cancer given that the person is a smoker”.

P(B|A) is called the **likelihood**; this is the probability of observing the new evidence, given our initial hypothesis. In the above example, this would be the “probability of being a smoker given that the person has cancer”.

P(A) is called the **prior**; this is the probability of our hypothesis without any additional prior information. In the above example, this would be the “probability of having cancer”.

P(B) is called the **marginal likelihood**; this is the total probability of observing

the evidence. In the above example, this would be the “probability of being a smoker”. In many applications of Bayes Rule, this is ignored, as it mainly serves as normalization.

Bayes' rule: example

C = breast cancer (having, not having)
M = mammographies (positive, negative)

P(C) = <0.01, 0.99>
P(m | c) = 0.8
P(m | ¬c) = 0.096

P(C | m) = **P**(m | C) **P**(C) / **P**(m) =
= α **P**(m | C) **P**(C) =
= α <**P**(m | c) **P**(c), **P**(m | ¬c) **P**(¬c)> =
= α <0.8 * 0.01, 0.096 * 0.99> =
= α <0.008, 0.095> = <0.078, 0.922>

P(c | m) = 7.8%

Bayesian learning:

Bayes theorem provides a way to calculate the probability of a hypothesis based on its prior probability, the probabilities of observing various data given the hypothesis, and the observed data itself.

Bayesian learning algorithms are among the most practical approaches to certain types of learning problems. 2 Bayesian methods aid in understanding other learning algorithms.

What is the relationship between Bayes theorem and the problem of concept learning?

Since Bayes theorem provides a principled way to calculate the posterior probability of each

hypothesis given the training data, and can use it as the basis for a straightforward learning algorithm that calculates the probability for each possible hypothesis, then outputs the most probable.

Notations

- P(h) *prior probability of* h, reflects any background knowledge about the chance that h is correct
- P(D) *prior probability of* D, probability that D will be observed
- P(D|h) probability of observing D given a world in which h holds
- P(h|D) *posterior probability of* h, reflects confidence that h holds after D has been observed

Bayes theorem is the cornerstone of Bayesian learning methods because it provides a way to calculate the posterior probability ***P(h|D),*** from the prior probability ***P(h),*** together with ***P(D)*** and ***P(D(h).***

6

Bayes Theorem:

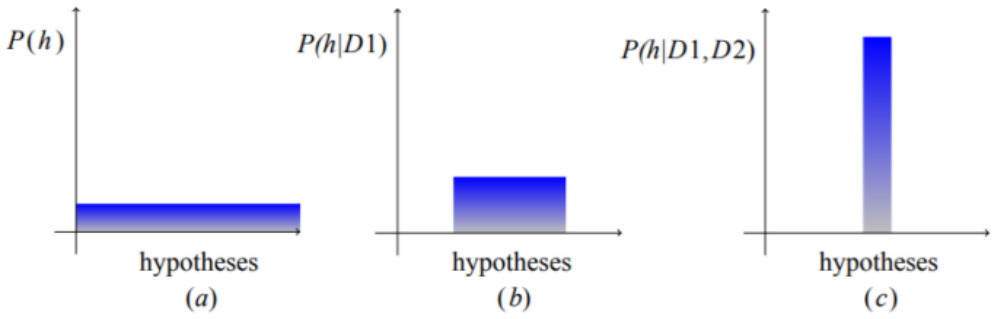
$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

P(h|D) increases with ***P(h)*** and with ***P(D|h)*** according to Bayes theorem.

P(h|D) decreases as ***P(D)*** increases, because the more probable it is that ***D*** will be observed independent of ***h,*** the less evidence ***D*** provides in support of ***h.***

The Evolution of Probabilities Associated with Hypotheses

- Figure (a) all hypotheses have the same probability.
- Figures (b) and (c), As training data accumulates, the posterior probability for inconsistent hypotheses becomes zero while the total probability summing to 1 is shared equally among the remaining consistent hypotheses.



MAP Hypotheses and Consistent Learners

A learning algorithm is a consistent learner if it outputs a hypothesis that commits zero errors over the training examples.

Every consistent learner outputs a MAP hypothesis, if we assume a uniform prior probability distribution over H ($P(h_i) = P(h_j)$ for all i, j), and deterministic, noise free training data (P(D|h) =1 if D and h are consistent, and 0 otherwise).

Example:

- FIND-S outputs a consistent hypothesis, it will output a MAP hypothesis under the probability distributions P(h) and P(D|h) defined above.
- Are there other probability distributions for P(h) and P(D|h) under which FIND-S outputs MAP hypotheses? Yes.
- Because FIND-S outputs a maximally specific hypothesis from the version space, its output hypothesis will be a MAP hypothesis relative to any prior probability distribution that favours more specific hypotheses.

- Bayesian framework is a way to characterize the behaviour of learning algorithms
- By identifying probability distributions P(h) and P(D|h) under which the output is a optimal hypothesis, implicit assumptions of the algorithm can be characterized (Inductive Bias)
- Inductive inference is modelled by an equivalent probabilistic reasoning system based on Bayes theorem

MAXIMUM LIKELIHOOD AND LEAST-SQUARED ERROR HYPOTHESES

Consider the problem of learning a continuous-valued target function such as neural network learning, linear regression, and polynomial curve fitting

A straightforward Bayesian analysis will show that under certain assumptions any learning algorithm that minimizes the squared error between the output hypothesis predictions and the training data will output a maximum likelihood (ML) hypothesis

Learning A Continuous-Valued Target Function

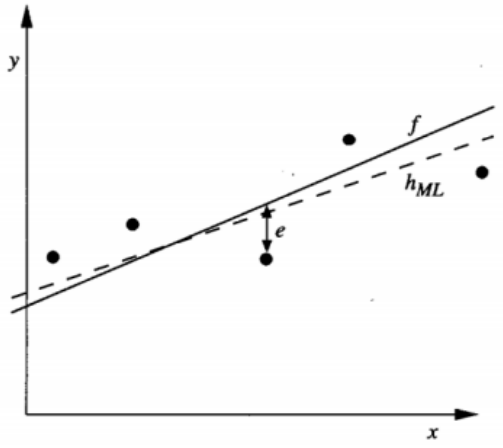
- Learner L considers an instance space X and a hypothesis space H consisting of some class of real-valued functions defined over X, i.e., ($\forall$ h $\in$ H)[h : X $\rightarrow$ R] and training examples of the form $<x_i,d_i>$
- The problem faced by L is to learn an unknown target function f : X $\rightarrow$ R
- A set of m training examples is provided, where the target value of each example is corrupted by random noise drawn according to a Normal probability distribution with zero mean ($d_i = f(x_i) + e_i$)
- Each training example is a pair of the form (x_i ,d_i) where di = f (x_i) + e_i .

  – Here f(x_i) is the noise-free value of the target function and e_i is a random variable representing the noise.

  – It is assumed that the values of the e_i are ***drawn independently*** and that they are distributed according to a ***Normal distribution*** with zero mean.
- The task of the learner is to output a ***maximum likelihood hypothesis***, or, equivalently, a MAP hypothesis assuming all hypotheses are equally probable a priori.

Learning A Linear Function

- The target function f corresponds to the solid line.
- The training examples (xi , di) are assumed to have Normally distributed noise ei with zero mean added to the true target value f (xi).
- The dashed line corresponds to the hypothesis h_{ML} with least-squared training error, hence the maximum likelihood hypothesis.
- Notice that the maximum likelihood hypothesis is not necessarily identical to the correct hypothesis, f, because it is inferred from only a limited sample of noisy training data



Before showing why a hypothesis that minimizes the sum of squared errors in this setting is also a maximum likelihood hypothesis, let us quickly review ***probability densities and Normal distributions***

Probability Density for continuous variables

$$p(x_0) \equiv \lim_{\epsilon \to 0} \frac{1}{\epsilon} P(x_0 \le x < x_0 + \epsilon)$$

e: a random noise variable generated by a Normal probability distribution

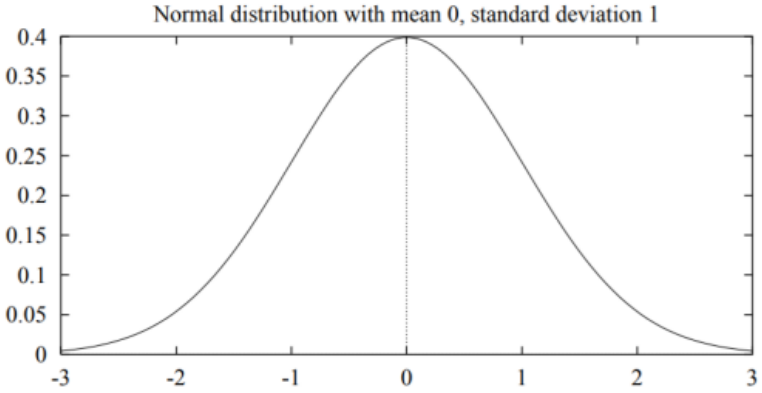
$<x_1 \ldots x_m>$: the sequence of instances (as before)

$<d_1 \ldots d_m>$: the sequence of target values with $d_i = f(x_i) + e_i$

Normal Probability Distribution (Gaussian Distribution)

A Normal distribution is a smooth, bell-shaped distribution that can be completely characterized by its mean μ and its standard deviation σ

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$



- A Normal distribution is fully determined by two parameters in the formula: μ and σ .
- If the random variable X follows a normal distribution:
  - The probability that X will fall into the interval (a, b) is $\int_a^b p(x)dx$
  - The expected, or ***mean value of X***, E[X] = μ
  - The ***variance of X***, Var(X) = σ^2
  - The ***standard deviation of X***, $\sigma_x = \sigma$
- The ***Central Limit Theorem*** states that the sum of a large number of independent, identically distributed random variables follows a distribution that is approximately ***Normal***.

Using the previous definition of h_{ML} we have

$$h_{ML} = \underset{h \in H}{argmax}\ p(D|h)$$

Assuming training examples are mutually independent given h, we can write P(D|h) as the product of the various (d_i|h)

$$h_{ML} = \underset{h \in H}{argmax} \prod_{i=1}^{m} p(d_i|h)$$

Given the noise e_i obeys a Normal distribution with zero mean and unknown variance σ^2 , each d_i must also obey a Normal distribution around the true targetvalue $f(x_i)$. Because we are writing the expression for P(D|h), we assume h is the correct description of f. Hence, $\mu = f(x_i) = h(x_i)$

$$h_{ML} = \underset{h \in H}{argmax} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(d_i - h(x_i))^2}$$

$$h_{ML} = \underset{h \in H}{\text{argmax}} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(d_i - \mu)^2}$$

$$= \underset{h \in H}{\text{argmax}} \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(d_i - h(x_i))^2}$$

It is common to maximize the less complicated logarithm, which is justified because of the monotonicity of function *p*.

$$= \underset{h \in H}{\text{argmax}} \sum_{i=1}^{m} ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2\sigma^2}(d_i - h(x_i))^2$$

The first term in this expression is a constant independent of ***h*** and can therefore be discarded

$$= \underset{h \in H}{\text{argmax}} \sum_{i=1}^{m} -\frac{1}{2\sigma^2}(d_i - h(x_i))^2$$

Maximizing this negative term is equivalent to minimizing the corresponding positive term.

$$= \underset{h \in H}{\text{argmin}} \sum_{i=1}^{m} \frac{1}{2\sigma^2}(d_i - h(x_i))^2$$

Finally Discard constants that are independent of ***h***

$$h_{ML} = \underset{h \in H}{argmin} \sum_{i=1}^{m} (d_i - h(x_i))^2$$

- the h_{ML} is one that minimizes the sum of the squared errors

Why is it reasonable to choose the Normal distribution to characterize noise?

- good approximation of many types of noise in physical systems
- Central Limit Theorem shows that the sum of a sufficiently large number of independent, identically distributed random variables itself obeys a Normal distribution

Only noise in the target value is considered, not in the attributes describing the instances themselves

---

MAXIMUM LIKELIHOOD HYPOTHESES FOR PREDICTING PROBABILITIES

Consider the setting in which we wish to learn a nondeterministic (probabilistic) function $f : X \rightarrow \{0, 1\}$, which has two discrete output values.

We want a function approximator whose output is the probability that f(x) = 1

In other words , learn the target function

$$f' : X \rightarrow [0, 1] \text{ such that } f'(x) = P(f(x) = 1)$$

How can we learn f' using a neural network?

Use of brute force way would be to first collect the observed frequencies of 1's and 0's for each possible value of x and to then train the neural network to output the target frequency for each x.

What criterion should we optimize in order to find a maximum likelihood hypothesis for f' in this setting?

- First obtain an expression for P(D|h)
- Assume the training data D is of the form $D = \{(x_1, d_1) \ldots (x_m, d_m)\}$, where d_i is the observed 0 or 1 value for $f(x_i)$.
- Both x_i and d_i as random variables, and assuming that each training example is drawn independently, we can write P(D|h) as

$$P(D \mid h) = \prod_{i=1}^{m} P(x_i, d_i \mid h)$$

Applying the product rule

$$P(D \mid h) = \prod_{i=1}^{m} P(d_i \mid h, x_i) P(x_i)$$

The probability $P(d_i|h, x_i)$

$$P(d_i|h, x_i) = \begin{cases} h(x_i) & \text{if } d_i = 1 \\ (1 - h(x_i)) & \text{if } d_i = 0 \end{cases} \qquad \text{equ (3)}$$

Re-express it in a more mathematically manipulable form, as

$$P(d_i|h, x_i) = h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} \qquad \text{equ (4)}$$

Equation (4) to substitute for $P(d_i|h, x_i)$ in Equation (5) to obtain

$$P(D|h) = \prod_{i=1}^{m} h(x_i)^{d_i}(1 - h(x_i))^{1-d_i} P(x_i) \qquad \text{equ (5)}$$

We write an expression for the maximum likelihood hypothesis

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} h(x_i)^{d_i} (1 - h(x_i))^{1-d_i} P(x_i)$$

The last term is a constant independent of h, so it can be dropped

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \prod_{i=1}^{m} h(x_i)^{d_i} (1 - h(x_i))^{1-d_i} \qquad \text{equ (6)}$$

It easier to work with the log of the likelihood, yielding

$$h_{ML} = \underset{h \in H}{\operatorname{argmax}} \sum_{i=1}^{m} d_i \ln h(x_i) + (1 - d_i) \ln(1 - h(x_i)) \qquad \text{equ (7)}$$

Equation (7) describes the quantity that must be maximized in order to obtain the maximum likelihood hypothesis in our current problem setting

---

Gradient Search to Maximize Likelihood in a Neural Net

Derive a weight-training rule for neural network learning that seeks to maximize G(h, D) using gradient ascent

- The gradient of G(h, D) is given by the vector of partial derivatives of G(h, D) with respect to the various network weights that define the hypothesis h represented by the learned network
- In this case, the partial derivative of G(h, D) with respect to weight w_{jk} from input k to unit j is

$$\begin{aligned} \frac{\partial G(h,D)}{\partial w_{jk}} &= \sum_{i=1}^{m} \frac{\partial G(h, D)}{\partial h(x_i)} \frac{\partial h(x_i)}{\partial w_{jk}} \\ &= \sum_{i=1}^{m} \frac{\partial (d_i \ln h(x_i) + (1 - d_i) \ln(1 - h(x_i)))}{\partial h(x_i)} \frac{\partial h(x_i)}{\partial w_{jk}} \\ &= \sum_{i=1}^{m} \frac{d_i - h(x_i)}{h(x_i)(1 - h(x_i))} \frac{\partial h(x_i)}{\partial w_{jk}} \end{aligned} \qquad \text{equ (1)}$$

Suppose our neural network is constructed from a single layer of sigmoid units. Then,

$$\frac{\partial h(x_i)}{\partial w_{jk}} = \sigma'(x_i)x_{ijk} = h(x_i)(1 - h(x_i))x_{ijk}$$

where x_{ijk} is the k^{th} input to unit j for the i^{th} training example, and d(x) is the derivative of the sigmoid squashing function.

Finally, substituting this expression into Equation (1), we obtain a simple expression for the derivatives that constitute the gradient

$$\frac{\partial G(h, D)}{\partial w_{jk}} = \sum_{i=1}^{m} (d_i - h(x_i))\ x_{ijk}$$

Because we seek to maximize rather than minimize P(D|h), we perform ***gradient ascent*** rather than ***gradient descent search***. On each iteration of the search the weight vector is adjusted in the direction of the gradient, using the weight update rule

$$w_{jk} \leftarrow w_{jk} + \Delta w_{jk}$$

Where,

$$\Delta w_{jk} = \eta \sum_{i=1}^{m} (d_i - h(x_i))\ x_{ijk} \qquad \text{equ (2)}$$

where η is a small positive constant that determines the step size of the i gradient ascent search

---

It is interesting to compare this weight-update rule to the weight-update rule used by the BACKPROPAGATION algorithm to minimize the sum of squared errors between predicted and observed network outputs.

The BACKPROPAGATION update rule for output unit weights, re-expressed using our current notation, is

$$w_{jk} \leftarrow w_{jk} + \Delta w_{jk}$$

Where,

$$\Delta w_{jk} = \eta \sum_{i=1}^{m} h(x_i)(1 - h(x_i))(d_i - h(x_i))\ x_{ijk}$$

MINIMUM DESCRIPTION LENGTH PRINCIPLE

- A Bayesian perspective on Occam's razor
- Motivated by interpreting the definition of h_{MAP} in the light of basic concepts from information theory.

$$h_{MAP} = \underset{h \in H}{argmax}\ P(D|h)P(h)$$

which can be equivalently expressed in terms of maximizing the $\log_2$

$$h_{MAP} = \underset{h \in H}{argmax}\ \log_2 P(D|h) + \log_2 P(h)$$

or alternatively, minimizing the negative of this quantity

$$h_{MAP} = \underset{h \in H}{argmin}\ -\log_2 P(D|h) - \log_2 P(h) \qquad \text{equ (1)}$$

- This equation can be interpreted as a statement that short hypotheses are preferred, assuming a particular representation scheme for encoding hypotheses and data

---

Introduction to a basic result of information theory

- Consider the problem of designing a code to transmit messages drawn at random
- i is the message
- The probability of encountering message i is p_i
- Interested in the most compact code; that is, interested in the code that minimizes the expected number of bits we must transmit in order to encode a message drawn at random
- To minimize the expected code length we should assign shorter codes to messages that are more probable
- Shannon and Weaver (1949) showed that the optimal code (i.e., the code that minimizes the expected message length) assigns - log, pi bitst to encode message i.
- The number of bits required to encode message i using code C as the ***description length of message i with respect to C***, which we denote by $L_c(i)$.

Interpreting the equation

$$h_{MAP} = \underset{h \in H}{argmin} \; -\log_2 P(D|h) - \log_2 P(h) \qquad \text{equ (1)}$$

- $-\log_2 P(h)$: the description length of h under the optimal encoding for the hypothesis space H $L_{CH}(h) = -\log_2 P(h)$, where C_H is the optimal code for hypothesis space H.
- $-\log_2 P(D \mid h)$: the description length of the training data D given hypothesis h, under the optimal encoding fro the hypothesis space H: $L_{CH}(D|h) = -\log_2 P(D| h)$, where $C_{D|h}$ is the optimal code for describing data D assuming that both the sender and receiver know the hypothesis h.

Rewrite Equation (1) to show that h_{MAP} is the hypothesis h that minimizes the sum given by the description length of the hypothesis plus the description length of the data given the hypothesis.

$$h_{MAP} = \underset{h \in H}{argmin} \; L_{C_H}(h) + L_{C_{D|h}}(D|h)$$

where C_H and $C_{D|h}$ are the optimal encodings for H and for D given h

---

The Minimum Description Length (MDL) principle recommends choosing the hypothesis that minimizes the sum of these two description lengths of equ.

$$h_{MAP} = \underset{h \in H}{argmin} \; L_{C_H}(h) + L_{C_{D|h}}(D|h)$$

Minimum Description Length principle:

$$h_{MDL} = \underset{h \in H}{\text{argmin}} \; L_{C_1}(h) + L_{C_1}(D \mid h)$$

Where, codes C_1 and C_2 to represent the hypothesis and the data given the hypothesis

The above analysis shows that if we choose C_1 to be the optimal encoding of hypotheses C_H, and if we choose C_2 to be the optimal encoding $C_{D|h}$, then $h_{MDL} = h_{MAP}$

Application to Decision Tree Learning

Apply the MDL principle to the problem of learning decision trees from some training data.

What should we choose for the representations C_1 and C_2 of hypotheses and data?

- **For C_1:** C_1 might be some obvious encoding, in which the description length grows with the number of nodes and with the number of edges
- **For C_2:** Suppose that the sequence of instances $(x_1 \ldots x_m)$ is already known to both the transmitter and receiver, so that we need only transmit the classifications $(f(x_1) \ldots f(x_m))$.

  Now if the training classifications $(f(x_1) \ldots f(x_m))$ are identical to the predictions of the hypothesis, then there is no need to transmit any information about these examples. The description length of the classifications given the hypothesis ZERO

  If examples are misclassified by h, then for each misclassification we need to transmit a message that identifies which example is misclassified as well as its correct classification

The hypothesis h_{MDL} under the encoding C_1 and C_2 is just the one that minimizes the sum of these description lengths.

- MDL principle provides a way for trading off hypothesis complexity for the number of errors committed by the hypothesis
- MDL provides a way to deal with the issue of overfitting the data.
- Short imperfect hypothesis may be selected over a long perfect hypothesis.

Bayes Optimal Classifier

- Normally we consider:
  - What is the most probable ***hypothesis*** given the training data?
- We can also consider:
  - what is the most probable ***classification*** of the new instance given the training data?

- Consider a hypothesis space containing three hypotheses, hl, h2, and h3.
  - Suppose that the posterior probabilities of these hypotheses given the training data are .4, .3, and .3 respectively.
  - Thus, hl is the MAP hypothesis.
  - Suppose a new instance x is encountered, which is classified positive by *hl,* but negative by *h2* and h3.
  - Taking all hypotheses into account, the probability that x is positive is .4 (the probability associated with *h1),* and the probability that it is negative is therefore .6.
  - The most probable classification (negative) in this case is different from the classification generated by the MAP hypothesis.

- The most probable classification of the new instance is obtained by combining the predictions of all hypotheses, weighted by their posterior probabilities.
- If the possible classification of the new example can take on any value v_j from some set V, then the probability $P(v_j \mid D)$ that the correct classification for the new instance is v_j :

$$P(v_j|D) = \sum_{h_i \in H} P(v_j|h_i)P(h_i|D)$$

- **Bayes optimal classification:**

$$\underset{v_j \in V}{\operatorname{argmax}} \sum_{h_i \in H} P(v_j|h_i)P(h_i|D)$$

Bayes Optimal Classifier - Ex

$$P(h_1|D) = .4,\ P(\ominus|h_1) = 0,\ P(\oplus|h_1) = 1$$
$$P(h_2|D) = .3,\ P(\ominus|h_2) = 1,\ P(\oplus|h_2) = 0$$
$$P(h_3|D) = .3,\ P(\ominus|h_3) = 1,\ P(\oplus|h_3) = 0$$

Probabilities:

$$\sum_{h_i \in H} P(\oplus|h_i)P(h_i|D) = .4$$

$$\sum_{h_i \in H} P(\ominus|h_i)P(h_i|D) = .6$$

Result:

$$\underset{v_j \in \{\oplus, \ominus\}}{\text{argmax}} \sum_{h_i \in H} P(v_j|h_i)P(h_i|D) = \ominus$$

- Although the Bayes optimal classifier obtains the best performance that can be achieved from the given training data, it can be quite costly to apply.
  - The expense is due to the fact that it computes the posterior probability for every hypothesis in H and then combines the predictions of each hypothesis to classify each new instance.

- An alternative, less optimal method is the Gibbs algorithm:

1. Choose a hypothesis ***h*** from H at random, according to the posterior probability distribution over H.
2. Use ***h*** to predict the classification of the next instance ***x.***

Naive Bayes Classifier

- One highly practical Bayesian learning method is Naive Bayes Learner (***Naive Bayes Classifier).***
- The naive Bayes classifier applies to learning tasks where each instance ***x*** is described by a conjunction of attribute values and where the target function f (x) can take on any value from some finite set V.
- A set of training examples is provided, and a new instance is presented, described by the tuple of attribute values ***(a_1, a_2 ...a_n).***
- The learner is asked to predict the target value (classification), for this new instance.

- The Bayesian approach to classifying the new instance is to assign the most probable target value v_{MAP}, given the attribute values (a_1, a_2 ... a_n) that describe the instance.

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j | a_1, a_2 \ldots a_n)$$

- By Bayes theorem:

$$v_{MAP} = \underset{v_j \in V}{\operatorname{argmax}} \frac{P(a_1, a_2 \ldots a_n | v_j) P(v_j)}{P(a_1, a_2 \ldots a_n)}$$

$$= \underset{v_j \in V}{\operatorname{argmax}} P(a_1, a_2 \ldots a_n | v_j) P(v_j)$$

- It is easy to estimate each of the $\mathbf{P(v_j)}$ simply by counting the frequency with which each target value $\mathbf{v_j}$ occurs in the training data.
- However, estimating the different $\mathbf{P(a_1,a_2 \ldots a_n \mid v_j)}$ terms is not feasible unless we have a very, very large set of training data.
  - The problem is that the number of these terms is equal to the number of possible instances times the number of possible target values.
  - Therefore, we need to see every instance in the instance space many times in order to obtain reliable estimates.
- The naive Bayes classifier is based on the simplifying assumption that the attribute values are conditionally independent given the target value.
- For a given the target value of the instance, the probability of observing conjunction $a_1,a_2 \ldots a_n$, is just the product of the probabilities for the individual attributes:

  $$P(a_1, a_2 \ldots a_n | v_j) = \prod_i P(a_i | v_j)$$

- **Naive Bayes classifier:** $$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_i P(a_i | v_j)$$

Naive Bayes Classifier - Ex

Day	Outlook	Temp.	Humidity	Wind	Play Tennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Weak	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cold	Normal	Weak	Yes
D10	Rain	Mild	Normal	Strong	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

- New instance to classify:

  (Outlook=sunny, Temperature=cool, Humidity=high, Wind=strong)
- Our task is to predict the target value (yes or no) of the target concept *PlayTennis* for this new instance.

$$v_{NB} = \underset{v_j \in \{yes,no\}}{\text{argmax}} \; P(v_j) \prod_i P(a_i|v_j)$$

$$= \underset{v_j \in \{yes,no\}}{\text{argmax}} \; P(v_j) \; \mathbf{P(Outlook=sunny|v_j)\; P(Temperature=cool|v_j)\; P(Humidity=high|v_j)\; P(Wind=strong|v_j)}$$

- P(PlayTennis = yes) = 9/14 = .64
- P(PlayTennis = no) = 5/14 = .36

$$P(yes)\; P(sunny|yes)\; P(cool|yes)\; P(high|yes)\; P(strong|yes) = .0053$$

$$P(no)\; P(sunny|no)\; P(cool|no)\; P(high|no)\; P(strong|no) = .0206$$

➔ Thus, the naive Bayes classifier assigns the target value ***PlayTennis*** = ***no*** to this new instance, based on the probability estimates learned from the training data.

- Furthermore, by normalizing the above quantities to sum to one we can calculate the conditional probability that the target value is ***no,*** given the observed attribute values.

$$.0206 / (.0206 + .0053) = .795$$

Estimating Probabilities

- **P(Wind=strong | PlayTennis=no)** by the fraction $\mathbf{n_c/n}$ where $n = 5$ is the total number of training examples for which **PlayTennis=no,** and $n_c = 3$ is the number of these for which **Wind=strong.**
- When n_c is zero
  - $\mathbf{n_c/n}$ will be zero too
  - this probability term will dominate
- To avoid this difficulty we can adopt a Bayesian approach to estimating the probability, using the m-estimate defined as follows.

  m-estimate of probability: $(n_c + m*p) / (n + m)$
- if an attribute has k possible values we set $p = 1/k$.
  - p=0.5 because Wind has two possible values.
- m is called the equivalent sample size
  - augmenting the n actual observations by an additional m virtual samples distributed according to p.

Learning To Classify Text

LEARN_NAIVE_BAYES_TEXT(Examples,V)

- Examples is a set of text documents along with their target values. V is the set of all possible target values.
- This function learns the probability terms $P(w_k|v_j)$, describing the probability that a randomly drawn word from a document in class v_j will be the English word w_k.
- It also learns the class prior probabilities $P(v_j)$.

1. *collect all words, punctuation, and other tokens that occur in* ***Examples***
   - **Vocabulary** ← the set of all distinct words and other tokens occurring in any text document from **Examples**

LEARN_NAIVE_BAYES_TEXT(Examples,V)

2. *calculate the required* $\mathbf{P(v_j)}$ *and* $\mathbf{P(w_k|v_j)}$ *probability terms*

   For each target value $\boldsymbol{v_j}$ in **V** do
   - $\mathbf{docs_j}$ ← the subset of documents from **Examples** for which the target value is $\mathbf{v_j}$
   - $\mathbf{P(v_j)}$ ← $\mathbf{|docs_j| / |Examples|}$
   - $\mathbf{Text_j}$ ← a single document created by concatenating all members of $\mathbf{docs_j}$
   - **n** ← total number of distinct word positions in **Examples**
   - for each word $\mathbf{w_k}$ in **Vocabulary**
     - $\mathbf{n_k}$ ← number of times word $\mathbf{w_k}$ occurs in $\mathbf{Text_j}$
     - $\mathbf{P(w_k|v_j)}$ ← $\mathbf{(n_k + 1) / (n + |Vocabulary|)}$

CLASSIFY_NAIVE_BAYES_TEXT(Doc)

- *Return the estimated target value for the document* **Doc**.
- $\mathbf{a_i}$ *denotes the word found in the* i^{th} *position within* **Doc**.
  - **positions** ← all word positions in **Doc** that contain tokens found in **Vocabulary**
  - Return $\mathbf{V_{NB}}$, where

$$v_{NB} = \underset{v_j \in V}{\operatorname{argmax}} P(v_j) \prod_{i \in positions} P(a_i | v_j)$$

- $P(h \mid D) = \frac{P(D|h)P(h)}{P(D)}$

This means: the posterior probability of D given h equals the probability of observing data D given some world in which h holds times the prior probability of h all over the prior probability of D.

maximum a posteriori (MAP) hypothesis: the most probable hypothesis $h \in H$ given the observed data D

$$\begin{aligned} h_{MAP} &\equiv \underset{h \in H}{\operatorname{argmax}} P(h \mid D) \\ &= \underset{h \in H}{\operatorname{argmax}} \frac{P(D \mid h)P(h)}{P(D)} \\ &= \underset{h \in H}{\operatorname{argmax}} P(D \mid h)P(h) \end{aligned}$$

Assuming equal probabilities for all $h \in H$ (*maximum likelihood hypothesis*):

$$h_{ML} \equiv \underset{h \in H}{\operatorname{argmax}} P(D \mid h)$$

Any set H of mutually exclusive propositions whose probabilities sum to one.

An example:

A grim scenario at the doctor's office:

- a patient has cancer
- the patient does not have cancer

$$\begin{aligned} P(cancer) &= .008 & P(\neg cancer) &= .992 \\ P(\oplus \mid cancer) &= .98 & P(\ominus \mid cancer) &= .02 \\ P(\oplus \mid \neg cancer) &= .98 & P(\ominus \mid \neg cancer) &= .02 \end{aligned}$$

$$\begin{aligned} P(\oplus \mid cancer)P(cancer) &= (.98).008 = .0078 \\ P(\oplus \mid \neg cancer)P(\neg cancer) &= (.03).992 = .0298 \\ \therefore h_{MAP} &= \neg cancer \end{aligned}$$

Basic Probability Formulas

- Product Rule: $P(A \wedge B) = P(A \mid B)P(B) = P(B \mid A)P(A)$
- Sum Rule: $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$
- Bayes Theorem: $P(h \mid D) = \frac{P(D|h)P(h)}{P(D)}$
- Theorem of Total Probability (if events $A_1, \ldots, A_n$ are mutually exclusive with $\sum_{i=1}^{n} P(A_i) = 1$): $P(B) = \sum_{i=1}^{n} P(B \mid A_i)P(A_i)$

"Since Bayes theorem provides a principled way to calculate the posterior probability of each hypothesis given the training data, we can use it as the basis for a straightforward learning algorithm that calculates the probability for each possible hypothesis, then outputs the most

probable."(ML pg 158)

More Terminology

- Bayesian Belief Networks

. Naive Bayes assumption of conditional independence is too restrictive.
. But it is intractable without some such assumptions
. Bayesian belief networks describe conditional independence among subsets of variables.
-> allow combining prior knowledge about (in)dependencies among variables with observed training data
. Bayes belief networks are also called Bayes nets.

- Conditional Independence

. definition: X is conditionally independent of Y given Z if the probability distribution governing X is independent of the value of Y given the value of Z, that is, if

$$(\forall x_i, y_j, z_k) P(X = x_i | Y = y_j, Z = z_k) = P(X = x_i | Z = z_k)$$

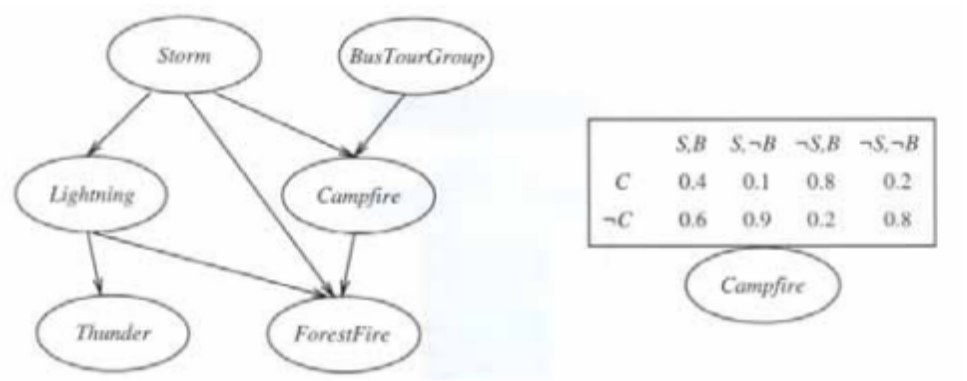
more compactly, we write

$$P(X | Y, Z) = P(X | Z)$$

. Example: Thunder is conditionally independent of Rain, given Lightning

$$P(Thunder | Rain, Lightning) = P(Thunder | Lightning)$$

- Bayesian Belief Network



. Each node represents a probability table of an attribute given its parents.

eg. $P(Campfire = T | Storm = T, Bus\,Tour\,Group = T) = 0.4$

. Each branch in the graph represents the conditional dependence.

. Bayesian belief network represents the joint probability distribution over all variables.

eg. $P(Storm, Bus\,Tour\,Group, \cdots, ForestFire)$

. In general,

$$P(y_1, \cdots, y_n) = \prod_{i=1}^{n} P(y_i | Parents(Y_i))$$

where $Parents(Y_i)$ denotes immediate predecessors of Y_i in graph.

. So, the joint distribution is fully defined by graph plus the conditional probability $P(y_i | Parents(Y_i))$.

- Inference in Bayesian Networks

. How can one infer the (probabilities of) values of one or more network variables, given observed values of others?
  > Bayes net contains all information needed for this inference.
  > If only one variable with unknown value, easy to infer it.
  > In general case, problem is NP hard.
. In practice, can succeed in many cases
  > Exact inference methods work well for some network structures.
  > Monte Carlo methods simulate the network randomly to calculate approximate solutions.

Learning of Bayesian Networks

Several variants of this learning task

- Network structure might be *known* or *unknown*
- Training examples might provide values of *all* network variables, or just *some*

If structure known and observe all variables

- Then it is easy as training a Naïve Bayes classifier

Suppose structure known, variables partially observable

e.g., observe *ForestFire, Storm, BusTourGroup, Thunder,* but not *Lightning, Campfire, …*

- Similar to training neural network with hidden units
- In fact, can learn network conditional probability tables using gradient ascent!
- Converge to network h that (locally) maximizes $P(D|h)$

EM algorithm

The Expectation-Maximization (EM) algorithm is defined as the combination of various unsupervised machine learning algorithms, which is used to determine the **local maximum likelihood estimates (MLE)** or **maximum a posteriori estimates (MAP)** for unobservable variables in statistical models. Further, it is a technique to find maximum likelihood estimation when the latent variables are present. It is also referred to as the **latent variable model.**

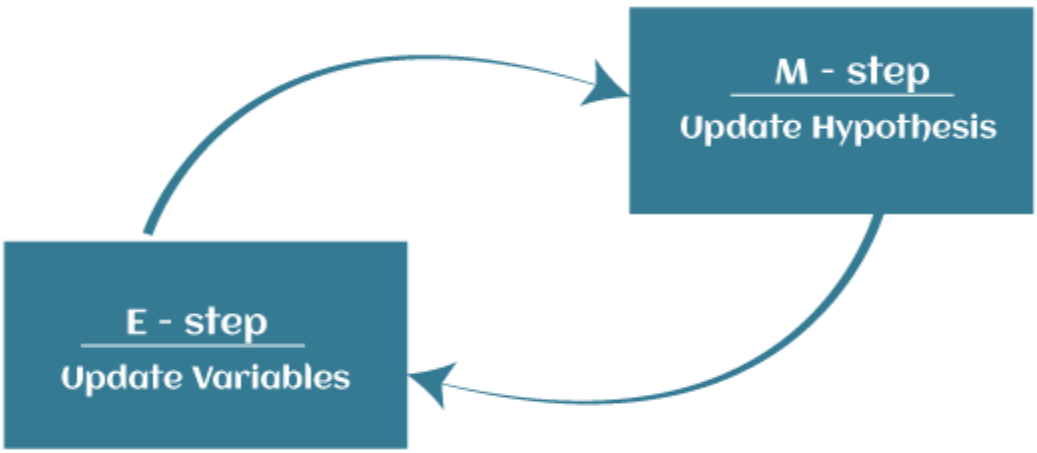
A latent variable model consists of both observable and unobservable variables where observable can be predicted while unobserved are inferred from the observed variable. These unobservable variables are known as latent variables.

Key Points:

- It is known as the latent variable model to determine MLE and MAP parameters for latent variables.
- It is used to predict values of parameters in instances where data is missing or unobservable for learning, and this is done until convergence of the values occurs.

EM Algorithm

The EM algorithm is the combination of various unsupervised ML algorithms, such as the **k-means clustering algorithm**. Being an iterative approach, it consists of two modes. In the first mode, we estimate the missing or latent variables. Hence it is referred to as the **Expectation/estimation step (E-step**). Further, the other mode is used to optimize the parameters of the models so that it can explain the data more clearly. The second mode is known as the **maximization-step or M-step.**



- **Expectation step (E - step):** It involves the estimation (guess) of all missing values in the dataset so that after completing this step, there should not be any missing value.

- **Maximization step (M - step):** This step involves the use of estimated data in the E-step and updating the parameters.
- **Repeat E-**step and M-step until the convergence of the values occurs.

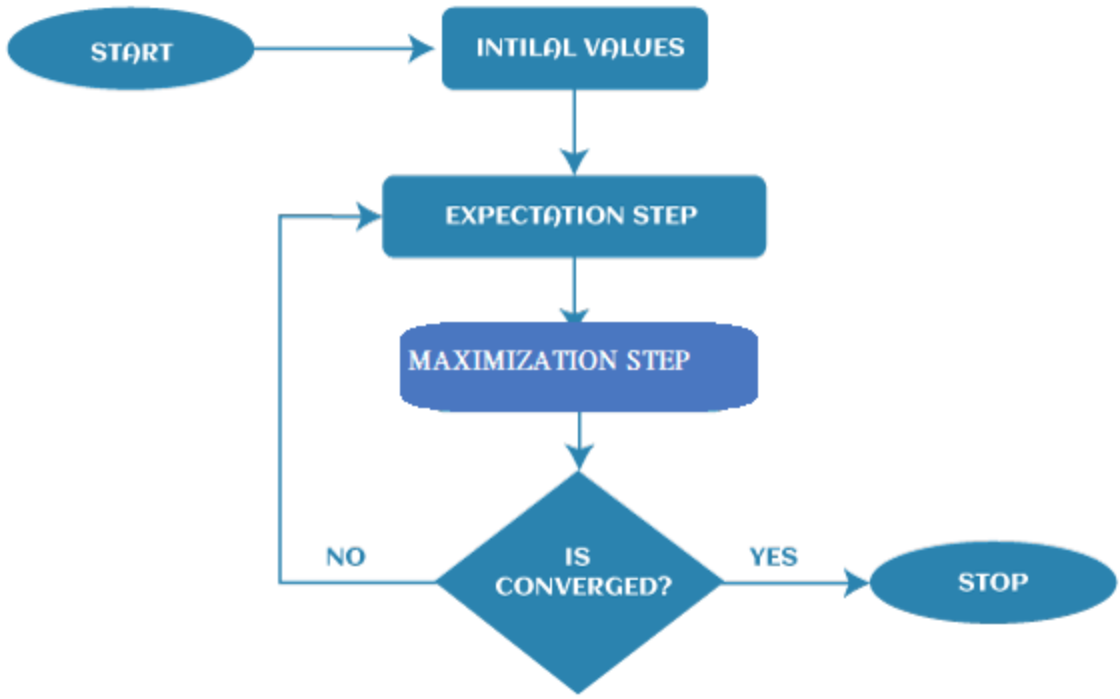
The primary goal of the EM algorithm is to use the available observed data of the dataset to estimate the missing data of the latent variables and then use that data to update the values of the parameters in the M-step.

What is Convergence in the EM algorithm?

Convergence is defined as the specific situation in probability based on intuition, e.g., if there are two random variables that have very less difference in their probability, then they are known as converged. In other words, whenever the values of given variables are matched with each other, it is called convergence.

Steps in EM Algorithm

The EM algorithm is completed mainly in 4 steps, which include I***nitialization Step, Expectation Step, Maximization Step, and convergence Step***. These steps are explained as follows:



- **1st Step:** The very first step is to initialize the parameter values. Further, the system is provided with incomplete observed data with the assumption that data is obtained from a specific model.

- **2nd Step:** This step is known as Expectation or E-Step, which is used to estimate or guess the values of the missing or incomplete data using the observed data. Further, E-step primarily

updates the variables.

- **3rd Step:** This step is known as Maximization or M-step, where we use complete data obtained from the 2nd step to update the parameter values. Further, M-step primarily updates the hypothesis.
- **4th step:** The last step is to check if the values of latent variables are converging or not. If it gets "yes", then stop the process; else, repeat the process from step 2 until the convergence occurs.

Gaussian Mixture Model (GMM)

The Gaussian Mixture Model or GMM is defined as a mixture model that has a combination of the unspecified probability distribution function. Further, GMM also requires estimated statistics values such as mean and standard deviation or parameters. It is used to estimate the parameters of the probability distributions to best fit the density of a given training dataset. Although there are plenty of techniques available to estimate the parameter of the Gaussian Mixture Model (GMM), the **Maximum Likelihood Estimation** is one of the most popular techniques among them.

Let's understand a case where we have a dataset with multiple data points generated by two different processes. However, both processes contain a similar Gaussian probability distribution and combined data. Hence it is very difficult to discriminate which distribution a given point may belong to.

The processes used to generate the data point represent a latent variable or unobservable data. In such cases, the Estimation-Maximization algorithm is one of the best techniques which helps us to estimate the parameters of the gaussian distributions. In the EM algorithm, E-step estimates the expected value for each latent variable, whereas M-step helps in optimizing them significantly using the Maximum Likelihood Estimation (MLE). Further, this process is repeated until a good set of latent values, and a maximum likelihood is achieved that fits the data.

Applications of EM algorithm

The primary aim of the EM algorithm is to estimate the missing data in the latent variables through observed data in datasets. The EM algorithm or latent variable model has a broad range of real-life applications in machine learning. These are as follows:

- The EM algorithm is applicable in data clustering in machine learning.
- It is often used in computer vision and NLP (Natural language processing).
- It is used to estimate the value of the parameter in mixed models such as the **Gaussian Mixture Model**and quantitative genetics.
- It is also used in psychometrics for estimating item parameters and latent abilities of item response theory models.
- It is also applicable in the medical and healthcare industry, such as in image reconstruction and

structural engineering.

- It is used to determine the Gaussian density of a function.

Advantages of EM algorithm

- It is very easy to implement the first two basic steps of the EM algorithm in various machine learning problems, which are E-step and M- step.
- It is mostly guaranteed that likelihood will enhance after each iteration.
- It often generates a solution for the M-step in the closed form.

Disadvantages of EM algorithm

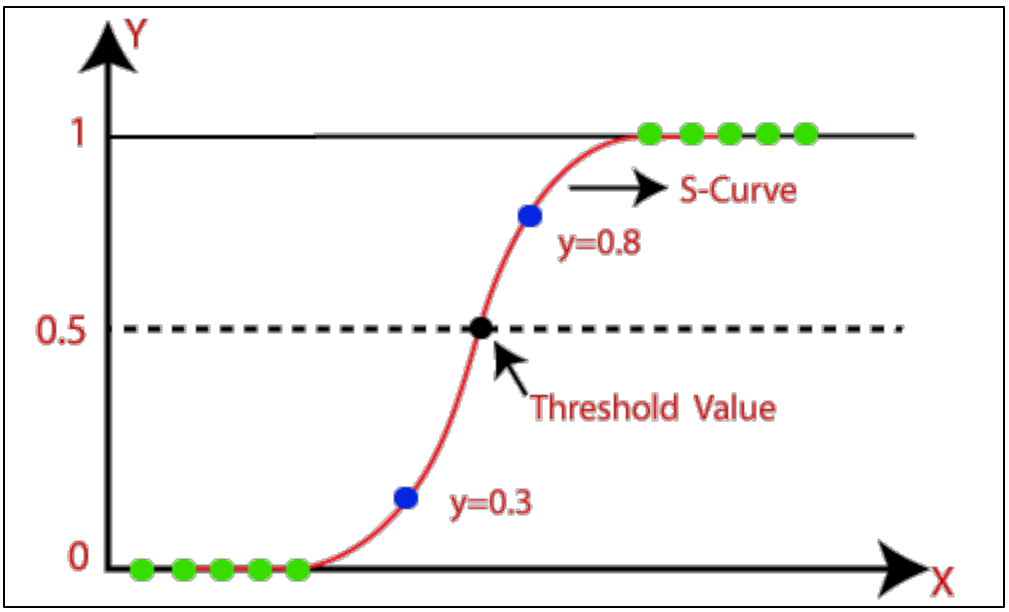
- The convergence of the EM algorithm is very slow.
- It can make convergence for the local optima only.
- It takes both forward and backward probability into consideration. It is opposite to that of numerical optimization, which takes only forward probabilities.

UNIT VI PARAMETRIC MACHINE LEARNING

Logistic Regression: Classification and representation – Cost function – Gradient descent – Advanced optimization – Regularization - Solving the problems on overfitting. Perceptron – Neural Networks – Multi – class Classification - Backpropagation – Non-linearity with activation functions (Tanh, Sigmoid, Relu, PRelu) - Dropout as regularization

Logistic Regression: Classification and representation :

- Logistic regression is one of the most popular Machine Learning algorithms, which comes under the Supervised Learning technique. It is used for predicting the categorical dependent variable using a given set of independent variables.
- 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**.
- Logistic Regression is much similar to the Linear Regression except that how they are used. Linear Regression is used for solving Regression problems, whereas **Logistic regression is used for solving the classification problems**.
- In Logistic regression, instead of fitting a regression line, we fit an "S" shaped logistic function, which predicts two maximum values (0 or 1).
- The curve from the logistic function indicates the likelihood of something such as whether the cells are cancerous or not, a mouse is obese or not based on its weight, etc.
- Logistic Regression is a significant machine learning algorithm because it has the ability to provide probabilities and classify new data using continuous and discrete datasets.
- Logistic Regression can be used to classify the observations using different types of data and can easily determine the most effective variables used for the classification. The below image is showing the logistic function:



Note: ***Logistic regression uses the concept of predictive modeling as regression; therefore, it is called logistic regression, but is used to classify samples; Therefore, it falls under the classification algorithm.***

Logistic Function (Sigmoid Function):

- The sigmoid function is a mathematical function used to map the predicted values to probabilities.
- It maps any real value into another value within a range of 0 and 1.
- The value of the logistic regression must be between 0 and 1, which cannot go beyond this limit, so it forms a curve like the "S" form. The S-form curve is called the Sigmoid function or the logistic function.
- In logistic regression, we use the concept of the threshold value, which defines the probability of either 0 or 1. Such as values above the threshold value tends to 1, and a value below the threshold values tends to 0.

Assumptions for Logistic Regression:

- The dependent variable must be categorical in nature.
- The independent variable should not have multi-collinearity.

Logistic Regression Equation:

The Logistic regression equation can be obtained from the Linear Regression equation. The mathematical steps to get Logistic Regression equations are given below:

- We know the equation of the straight line can be written as:

$$y= b_0 + b_1x_1 + b_2x_2 + b_3x_3 + \cdots + b_nx_n$$

- In Logistic Regression y can be between 0 and 1 only, so for this let's divide the above equation by (1-y):

$\frac{y}{1-y}$; 0 for y= 0, and infinity for y=1

- But we need range between -[infinity] to +[infinity], then take logarithm of the equation it will become:

$$log\left[\frac{y}{1-y}\right] = b_0 + b_1x_1 + b_2x_2 + b_3x_3 + \cdots + b_nx_n$$

The above equation is the final equation for Logistic Regression.

Type of Logistic Regression:

On the basis of the categories, Logistic Regression can be classified into three types:

- **Binomial:** In binomial Logistic regression, there can be only two possible types of the dependent variables, such as 0 or 1, Pass or Fail, etc.
- **Multinomial:** In multinomial Logistic regression, there can be 3 or more possible unordered types of the dependent variable, such as "cat", "dogs", or "sheep"
- **Ordinal:** In ordinal Logistic regression, there can be 3 or more possible ordered types of dependent variables, such as "low", "Medium", or "High".

Python Implementation of Logistic Regression (Binomial)

To understand the implementation of Logistic Regression in Python, we will use the below example:

Example: There is a dataset given which contains the information of various users obtained from the social networking sites. There is a car making company that has recently launched a new SUV car. So the company wanted to check how many users from the dataset, wants to purchase the car.

For this problem, we will build a Machine Learning model using the Logistic regression algorithm. The dataset is shown in the below image. In this problem, we will predict the **purchased variable (Dependent Variable)** by using **age and salary (Independent variables)**.

User ID	Gender	Age	EstimatedSalary	Purchased
15624510	Male	19	19000	0
15810944	Male	35	20000	0
15668575	Female	26	43000	0
15603246	Female	27	57000	0
15804002	Male	19	76000	0
15728773	Male	27	58000	0
15598044	Female	27	84000	0
15694829	Female	32	150000	1
15600575	Male	25	33000	0
15727311	Female	35	65000	0
15570769	Female	26	80000	0
15606274	Female	26	52000	0
15746139	Male	20	86000	0
15704987	Male	32	18000	0
15628972	Male	18	82000	0
15697686	Male	29	80000	0
15733883	Male	47	25000	1
15617482	Male	45	26000	1
15704583	Male	46	28000	1
15621083	Female	48	29000	1
15649487	Male	45	22000	1
15736760	Female	47	49000	1

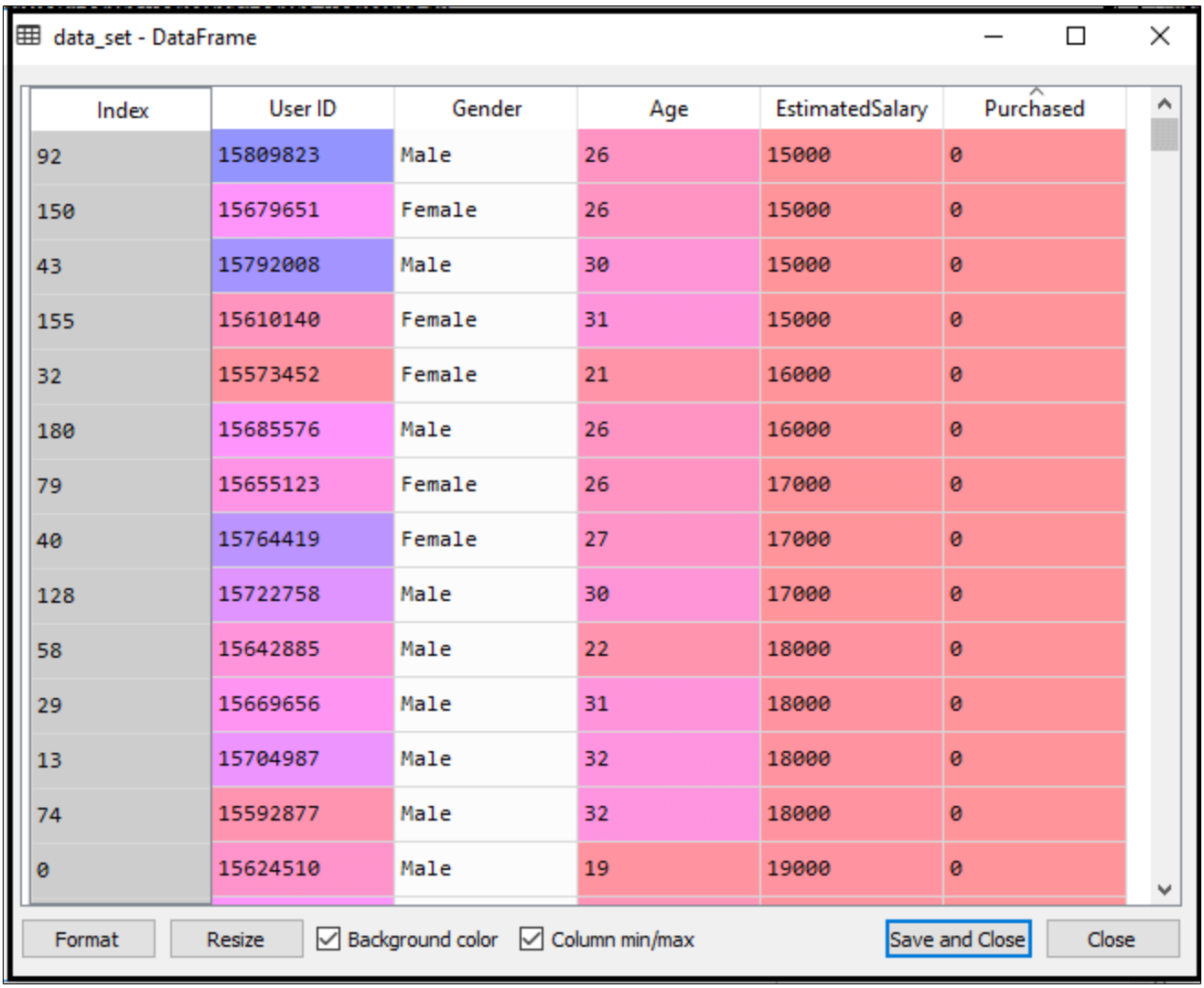
Steps in Logistic Regression: To implement the Logistic Regression using Python, we will use the same steps as we have done in previous topics of Regression. Below are the steps:

- Data Pre-processing step
- Fitting Logistic Regression to the Training set
- Predicting the test result
- Test accuracy of the result(Creation of Confusion matrix)
- Visualizing the test set result.

1. Data Pre-processing step: In this step, we will pre-process/prepare the data so that we can use it in our code efficiently. It will be the same as we have done in Data pre-processing topic. The code for this is given below:

```
#Data Pre-procesing Step
# importing libraries
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd

#importing datasets
data_set= pd.read_csv('user_data.csv')
```

By executing the above lines of code, we will get the dataset as the output. Consider the given image:

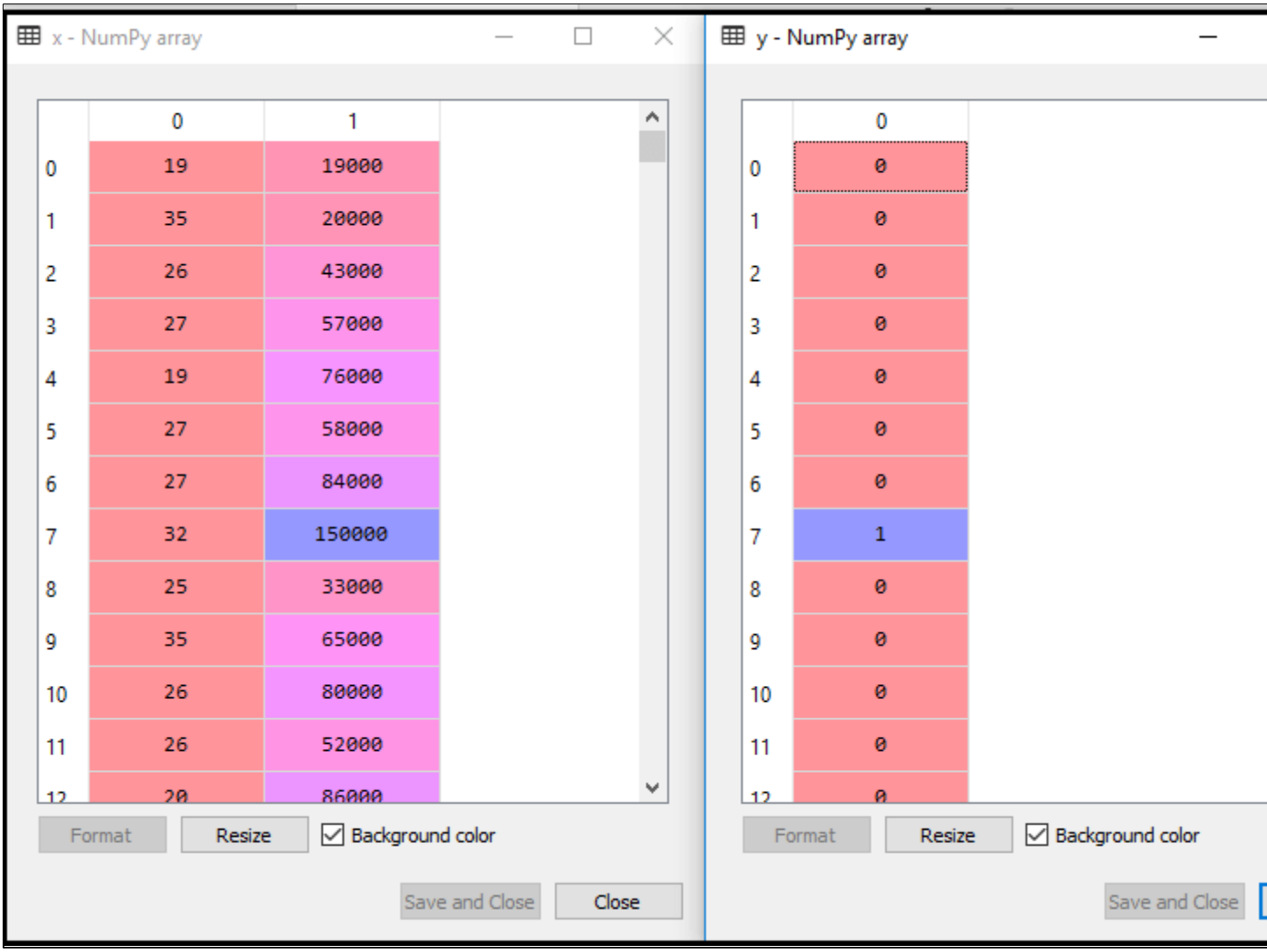


Index	User ID	Gender	Age	EstimatedSalary	Purchased
92	15809823	Male	26	15000	0
150	15679651	Female	26	15000	0
43	15792008	Male	30	15000	0
155	15610140	Female	31	15000	0
32	15573452	Female	21	16000	0
180	15685576	Male	26	16000	0
79	15655123	Female	26	17000	0
40	15764419	Female	27	17000	0
128	15722758	Male	30	17000	0
58	15642885	Male	22	18000	0
29	15669656	Male	31	18000	0
13	15704987	Male	32	18000	0
74	15592877	Male	32	18000	0
0	15624510	Male	19	19000	0

Now, we will extract the dependent and independent variables from the given dataset. Below is the code for it:

1. #Extracting Independent and dependent Variable
2. x= data_set.iloc[:, [2,3]].values
3. y= data_set.iloc[:, 4].values

   In the above code, we have taken [2, 3] for x because our independent variables are age and salary, which are at index 2, 3. And we have taken 4 for y variable because our dependent variable is at index 4. The output will be:

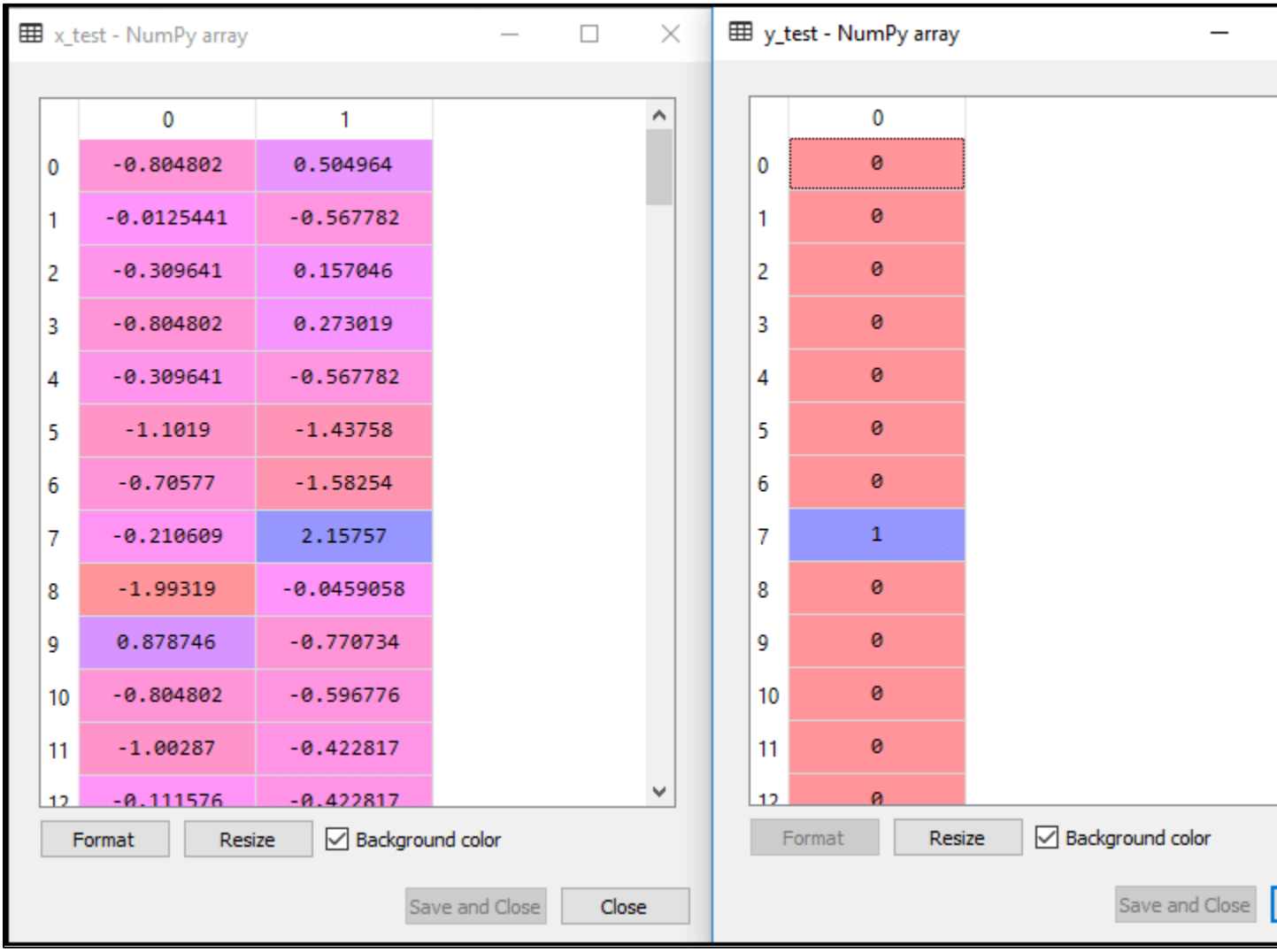


Now we will split the dataset into a training set and test set. Below is the code for it:

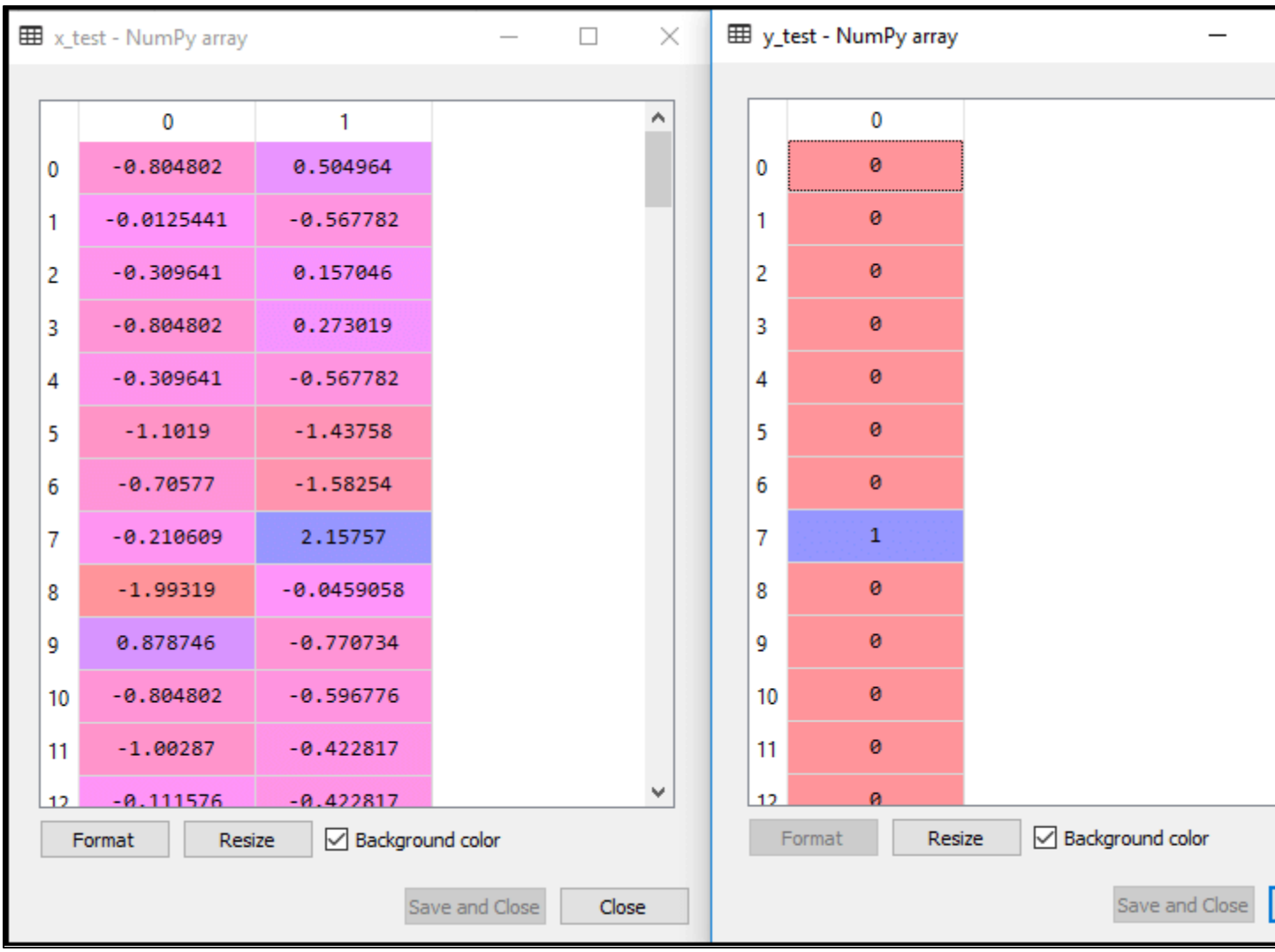
```
# Splitting the dataset into training and test set.
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 0.25, random_state=0)
```

The output for this is given below:

For test set:



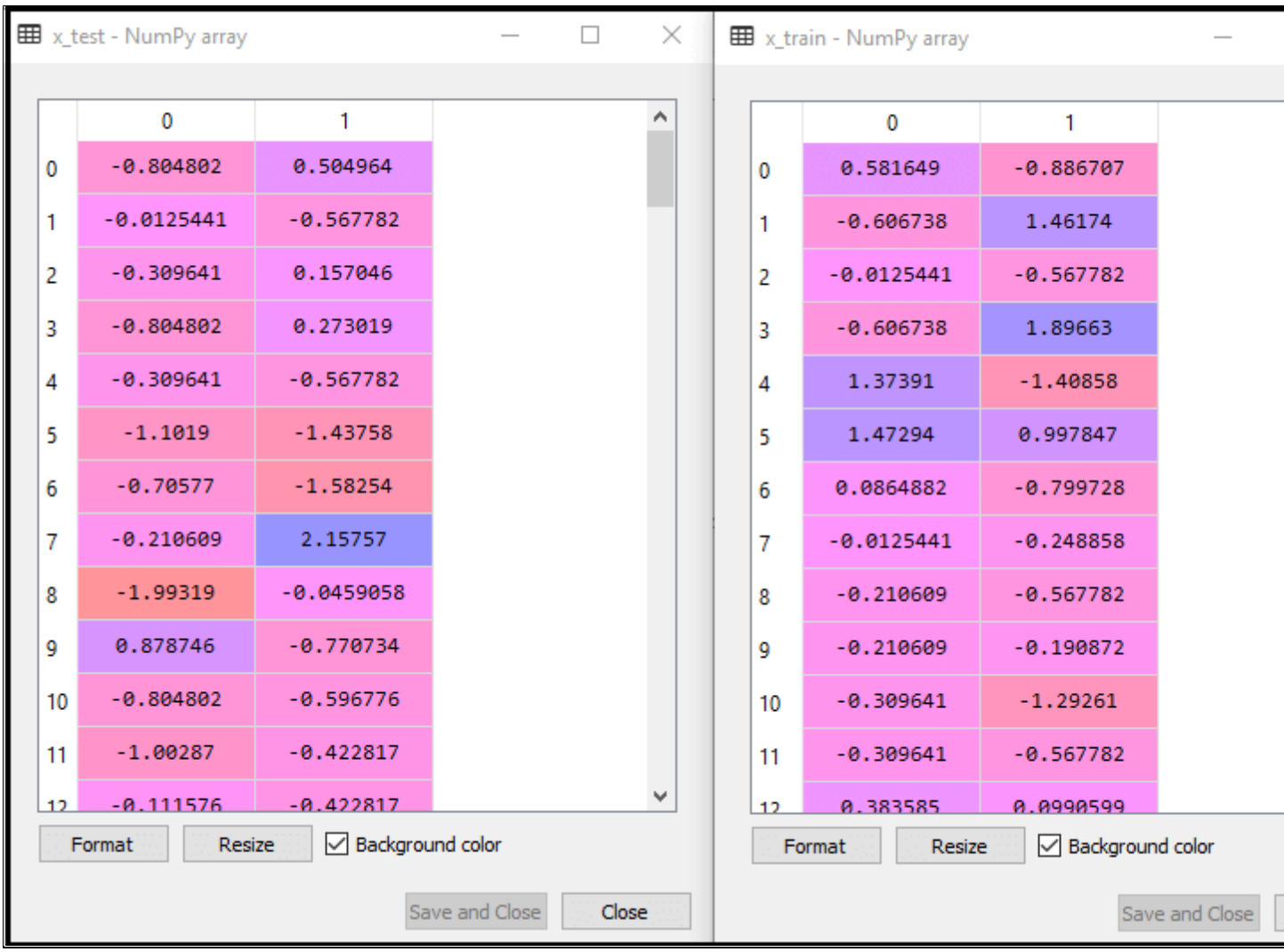
For training set:



In logistic regression, we will do feature scaling because we want accurate result of predictions. Here we will only scale the independent variable because dependent variable have only 0 and 1 values. Below is the code for it:

```
1. #feature Scaling
2. from sklearn.preprocessing import StandardScaler
3. st_x= StandardScaler()
4. x_train= st_x.fit_transform(x_train)
5. x_test= st_x.transform(x_test)
```

The scaled output is given below:



2. Fitting Logistic Regression to the Training set:

We have well prepared our dataset, and now we will train the dataset using the training set. For providing training or fitting the model to the training set, we will import the **LogisticRegression** class of the **sklearn** library.

After importing the class, we will create a classifier object and use it to fit the model to the logistic regression. Below is the code for it:

```
1. #Fitting Logistic Regression to the training set
2. from sklearn.linear_model import LogisticRegression
3. classifier= LogisticRegression(random_state=0)
4. classifier.fit(x_train, y_train)
```

Output: By executing the above code, we will get the below output:

Out[5]:

```
1. LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
2.                    intercept_scaling=1, l1_ratio=None, max_iter=100,
3.                    multi_class='warn', n_jobs=None, penalty='l2',
4.                    random_state=0, solver='warn', tol=0.0001, verbose=0,
5.                    warm_start=False)
```

Hence our model is well fitted to the training set.

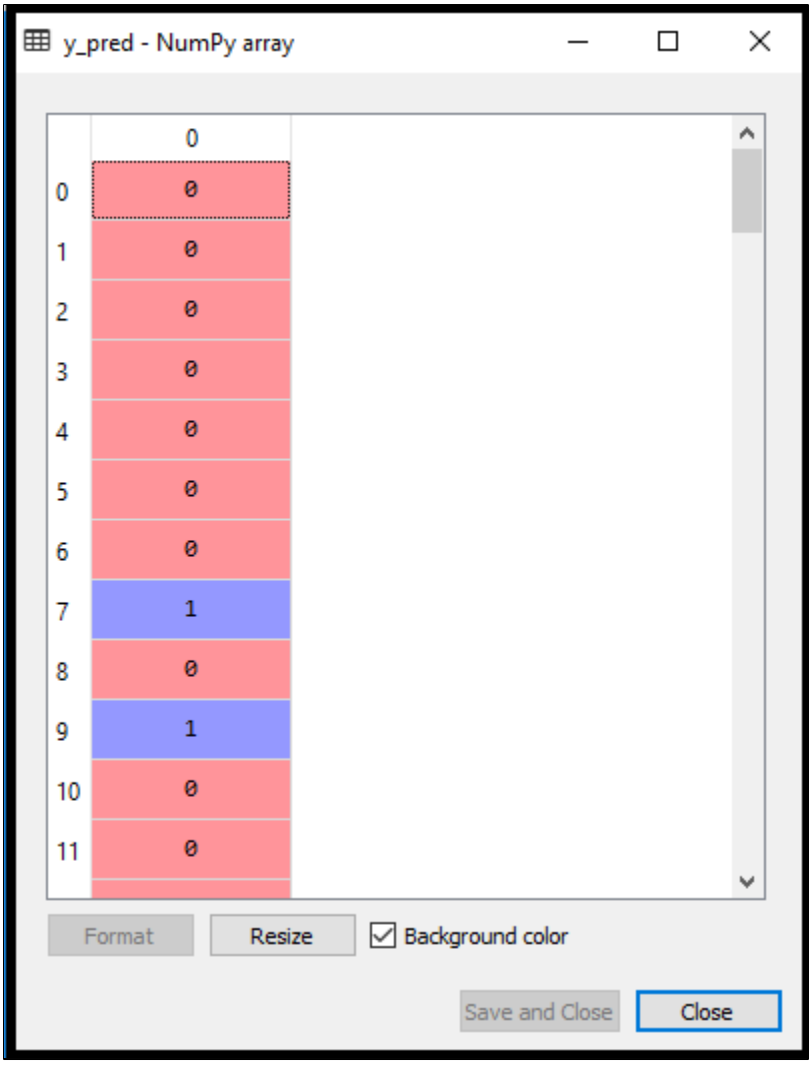
3. Predicting the Test Result

Our model is well trained on the training set, so we will now predict the result by using test set data. Below is the code for it:

```
1. #Predicting the test set result
2. y_pred= classifier.predict(x_test)
```

In the above code, we have created a y_pred vector to predict the test set result.

Output: By executing the above code, a new vector (y_pred) will be created under the variable explorer option. It can be seen as:



The above output image shows the corresponding predicted users who want to purchase or not purchase the car.

4. Test Accuracy of the result

Now we will create the confusion matrix here to check the accuracy of the classification. To create it, we need to import the **confusion_matrix** function of the sklearn library. After importing the function, we will call it using a new variable **cm**. The function takes two parameters, mainly **y_true**(the actual values) and **y_pred** (the targeted value return by the classifier). Below is the code for it:

```
1. #Creating the Confusion matrix
2. from sklearn.metrics import confusion_matrix
3. cm= confusion_matrix()
```

Output:

By executing the above code, a new confusion matrix will be created. Consider the below image:

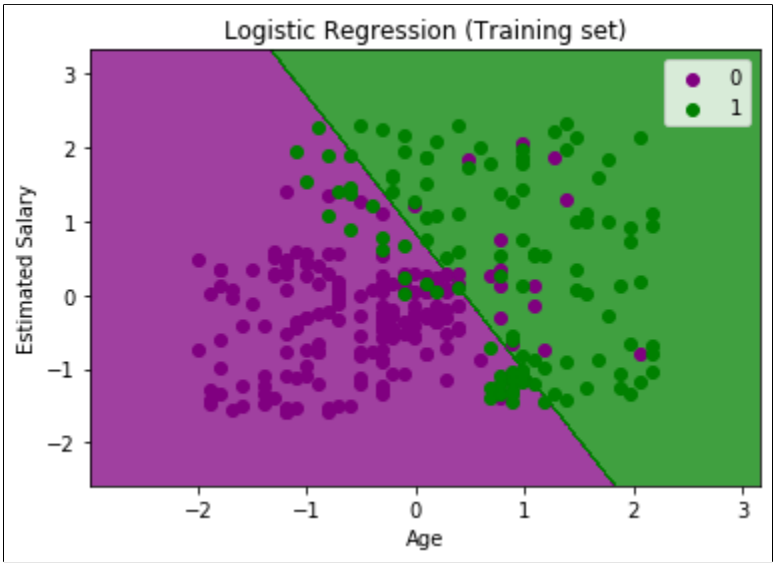
We can find the accuracy of the predicted result by interpreting the confusion matrix. By above output, we can interpret that 65+24= 89 (Correct Output) and 8+3= 11(Incorrect Output).

5. Visualizing the training set result

Finally, we will visualize the training set result. To visualize the result, we will use **ListedColormap** class of matplotlib library. Below is the code for it:

```
1. #Visualizing the training set result
2. from matplotlib.colors import ListedColormap
3. x_set, y_set = x_train, y_train
4. x1, x2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() -
    1, stop = x_set[:, 0].max() + 1, step  =0.01),
5. nm.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))
6. mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x
   1.shape),
7. alpha = 0.75, cmap = ListedColormap(('purple','green' )))
8. mtp.xlim(x1.min(), x1.max())
9. mtp.ylim(x2.min(), x2.max())
10.for i, j in enumerate(nm.unique(y_set)):
11.    mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
12.        c = ListedColormap(('purple', 'green'))(i), label = j)
13.mtp.title('Logistic Regression (Training set)')
14.mtp.xlabel('Age')
15.mtp.ylabel('Estimated Salary')
16.mtp.legend()
17.mtp.show()
```

In the above code, we have imported the **ListedColormap** class of Matplotlib library to create the colormap for visualizing the result. We have created two new variables **x_set** and **y_set** to replace **x_train** and **y_train**. After that, we have used the **nm.meshgrid** command to create a rectangular grid, which has a range of -1(minimum) to 1 (maximum). The pixel points we have taken are of 0.01 resolution. To create a filled contour, we have used **mtp.contourf** command, it will create regions of provided colors (purple and green). In this function, we have passed the **classifier.predict** to show the predicted data points predicted by the classifier.
Output: By executing the above code, we will get the below output:



The graph can be explained in the below points:

- In the above graph, we can see that there are some **Green points** within the green region and **Purple points** within the purple region.
- All these data points are the observation points from the training set, which shows the result for purchased variables.
- This graph is made by using two independent variables i.e., **Age on the x-axis** and **Estimated salary on the y-axis**.
- The **purple point observations** are for which purchased (dependent variable) is probably 0, i.e., users who did not purchase the SUV car.
- The **green point observations** are for which purchased (dependent variable) is probably 1 means user who purchased the SUV car.
- We can also estimate from the graph that the users who are younger with low salary, did not purchase the car, whereas older users with high estimated salary purchased the car.
- But there are some purple points in the green region (Buying the car) and some green points in the purple region(Not buying the car). So we can say that younger users with a high estimated salary purchased the car, whereas an older user with a low estimated salary did not purchase the car.

The goal of the classifier:
We have successfully visualized the training set result for the logistic regression, and our goal for this classification is to divide the users who purchased the SUV car

and who did not purchase the car. So from the output graph, we can clearly see the two regions (Purple and Green) with the observation points. The Purple region is for those users who didn't buy the car, and Green Region is for those users who purchased the car.

Linear Classifier:

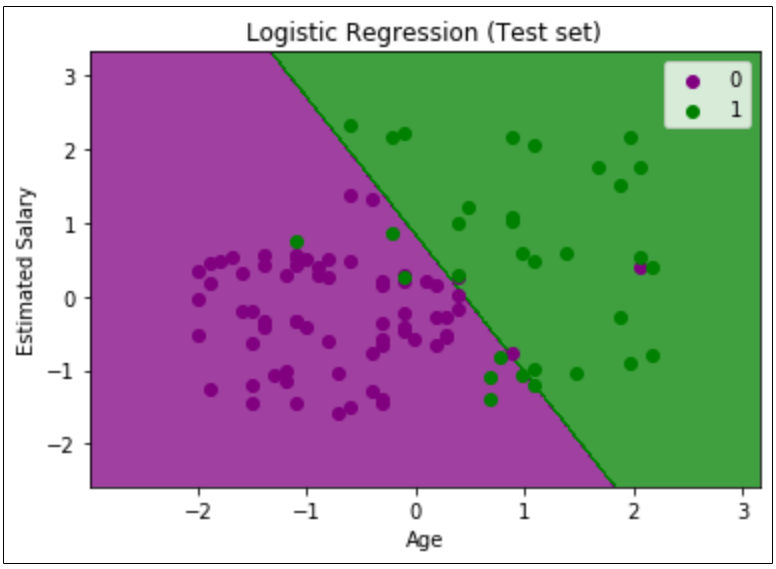
As we can see from the graph, the classifier is a Straight line or linear in nature as we have used the Linear model for Logistic Regression. In further topics, we will learn for non-linear Classifiers.

Visualizing the test set result:

Our model is well trained using the training dataset. Now, we will visualize the result for new observations (Test set). The code for the test set will remain same as above except that here we will use **x_test and y_test** instead of **x_train and y_train**. Below is the code for it:

```
#Visulaizing the test set result
from matplotlib.colors import ListedColormap
x_set, y_set = x_test, y_test
x1, x2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() -
 1, stop = x_set[:, 0].max() + 1, step  =0.01),
nm.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))
mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x
1.shape),
alpha = 0.75, cmap = ListedColormap(('purple','green' )))
mtp.xlim(x1.min(), x1.max())
mtp.ylim(x2.min(), x2.max())
for i, j in enumerate(nm.unique(y_set)):
    mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
        c = ListedColormap(('purple', 'green'))(i), label = j)
mtp.title('Logistic Regression (Test set)')
mtp.xlabel('Age')
mtp.ylabel('Estimated Salary')
mtp.legend()
mtp.show()
```

Output:



The above graph shows the test set result. As we can see, the graph is divided into two regions (Purple and Green). And Green observations are in the green region, and Purple observations are in the purple region. So we can say it is a good prediction and model. Some of the green and purple data points are in different regions, which can be ignored as we have already calculated this error using the confusion matrix (11 Incorrect output).
Hence our model is pretty good and ready to make new predictions for this classification problem.

Gradient Descent:

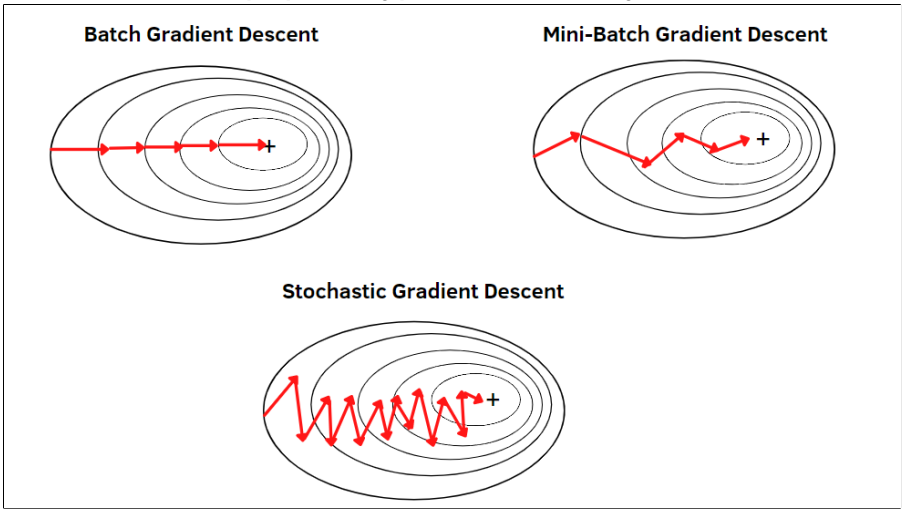
Gradient descent adjusts parameters to minimize particular functions to local minima. In linear regression, it finds weight and biases, The algorithm objective is to identify model parameters like weight and bias that reduce model error on training data.
Dy/dx
dy = change in y
dx = change in x

1. A gradient measures how much the output of a function changes if you change the inputs a little bit.
2. In machine learning, a gradient is a derivative of a function that has more than one input variable. Known as the slope of a function in mathematical terms, the gradient simply measures the change in all weights about the change in error.

Learning Rate:
The algorithm designer can set the learning rate. If we use a learning rate that is too small, it will cause us to update very slowly, requiring more iterations to get a better solution.

Types of Gradient Descent:

There are three popular types that mainly differ in the amount of data they use:



1. BATCH GRADIENT DESCENT:

Batch gradient descent, also known as vanilla gradient descent, calculates the error for each example within the training dataset. Still, the model is not changed until every training sample has been assessed. The entire procedure is referred to as a cycle and a training epoch.

Some benefits of batch are its computational efficiency, which produces a stable error gradient and a stable convergence. Some drawbacks are that the stable error gradient can sometimes result in a state of convergence that isn't the best the model can achieve. It also requires the entire training dataset to be in memory and available to the algorithm.

```
class GDRegressor:

    def __init__(self,learning_rate=0.01,epochs=100):

        self.coef_ = None
        self.intercept_ = None
        self.lr = learning_rate
        self.epochs = epochs

    def fit(self,X_train,y_train):
        # init your coefs
        self.intercept_ = 0
        self.coef_ = np.ones(X_train.shape[1])

        for i in range(self.epochs):
            # update all the coef and the intercept
            y_hat = np.dot(X_train,self.coef_) + self.intercept_
            #print("Shape of y_hat",y_hat.shape)
            intercept_der = -2 * np.mean(y_train - y_hat)
            self.intercept_ = self.intercept_ - (self.lr * intercept_der)

            coef_der = -2 * np.dot((y_train - y_hat),X_train)/X_train.shape[0]
            self.coef_ = self.coef_ - (self.lr * coef_der)
```

```
        print(self.intercept_,self.coef_)

    def predict(self,X_test):
        return np.dot(X_test,self.coef_) + self.intercept_
```

Advantages

1. Fewer model updates mean that this variant of the steepest descent method is more computationally efficient than the stochastic gradient descent method.
2. Reducing the update frequency provides a more stable error gradient and a more stable convergence for some problems.
3. Separating forecast error calculations and model updates provides a parallel processing-based algorithm implementation.

Disadvantages

1. A more stable error gradient can cause the model to prematurely converge to a suboptimal set of parameters.
2. End-of-training epoch updates require the additional complexity of accumulating prediction errors across all training examples.
3. The batch gradient descent method typically requires the entire training dataset in memory and is implemented for use in the algorithm.
4. Large datasets can result in very slow model updates or training speeds.
5. Slow and require more computational power.

2. STOCHASTIC GRADIENT DESCENT:

By contrast, stochastic gradient descent (SGD) changes the parameters for each training sample one at a time for each training example in the dataset. Depending on the issue, this can make SGD faster than batch gradient descent. One benefit is that the regular updates give us a fairly accurate idea of the rate of improvement.
However, the batch approach is less computationally expensive than the frequent updates. The frequency of such updates can also produce noisy gradients, which could cause the error rate to fluctuate rather than gradually go down.

Advantages

1. You can instantly see your model's performance and improvement rates with frequent updates.
2. This variant of the steepest descent method is probably the easiest to understand and implement, especially for beginners.
3. Increasing the frequency of model updates will allow you to learn more about some issues faster.
4. The noisy update process allows the model to avoid local minima (e.g., premature convergence).
5. Faster and require less computational power.
6. Suitable for the larger dataset.

Disadvantages

1. Frequent model updates are more computationally intensive than other steepest descent configurations, and it takes considerable time to train the model with large datasets.
2. Frequent updates can result in noisy gradient signals. This can result in

model parameters and cause errors to fly around (more variance across the training epoch).
3. A noisy learning process along the error gradient can also make it difficult for the algorithm to commit to the model's minimum error.

Implementation of sgd classifier in sklearn:

```
from sklearn.linear_model import SGDClassifier
X = [[0., 0.], [1., 1.]]
y = [0, 1]
clf = SGDClassifier(loss="hinge", penalty="l2", max_iter=5)
clf.fit(X, y)
SGDClassifier(max_iter=5)
```

3. MINI-BATCH GRADIENT DESCENT:

Since mini-batch gradient descent combines the ideas of batch gradient descent with SGD, it is the preferred technique. It divides the training dataset into manageable groups and updates each separately. This strikes a balance between batch gradient descent's effectiveness and stochastic gradient descent's durability. Mini-batch sizes typically range from 50 to 256, although, like with other machine learning techniques, there is no set standard because it depends on the application. The most popular kind in deep learning, this method is used when training a neural network.

```
class MBGDRegressor:

    def __init__(self,batch_size,learning_rate=0.01,epochs=100):

        self.coef_ = None
        self.intercept_ = None
        self.lr = learning_rate
        self.epochs = epochs
        self.batch_size = batch_size

    def fit(self,X_train,y_train):
        # init your coefs
        self.intercept_ = 0
        self.coef_ = np.ones(X_train.shape[1])

        for i in range(self.epochs):

            for j in range(int(X_train.shape[0]/self.batch_size)):

                idx = random.sample(range(X_train.shape[0]),self.batch_size)

                y_hat = np.dot(X_train[idx],self.coef_) + self.intercept_
                #print("Shape of y_hat",y_hat.shape)
                intercept_der = -2 * np.mean(y_train[idx] - y_hat)
                self.intercept_ = self.intercept_ - (self.lr * intercept_der)
```

```
            coef_der = -2 * np.dot((y_train[idx] - y_hat),X_train[idx])
            self.coef_ = self.coef_ - (self.lr * coef_der)

        print(self.intercept_,self.coef_)

    def predict(self,X_test):
        return np.dot(X_test,self.coef_) + self.intercept_
```

Advantages

1. The model is updated more frequently than the stack gradient descent method, allowing for more robust convergence and avoiding local minima.
2. Batch updates provide a more computationally efficient process than stochastic gradient descent.
3. Batch processing allows for both the efficiency of not having all the training data in memory and implementing the algorithm.

Disadvantages

1. Mini-batch requires additional hyperparameters “mini-batch size” to be set for the learning algorithm.
2. Error information should be accumulated over a mini-batch of training samples, such as batch gradient descent.
3. it will generate complex functions.

Configure Mini-Batch Gradient Descent:

The mini-batch steepest descent method is a variant of the steepest descent method recommended for most applications, intense learning.

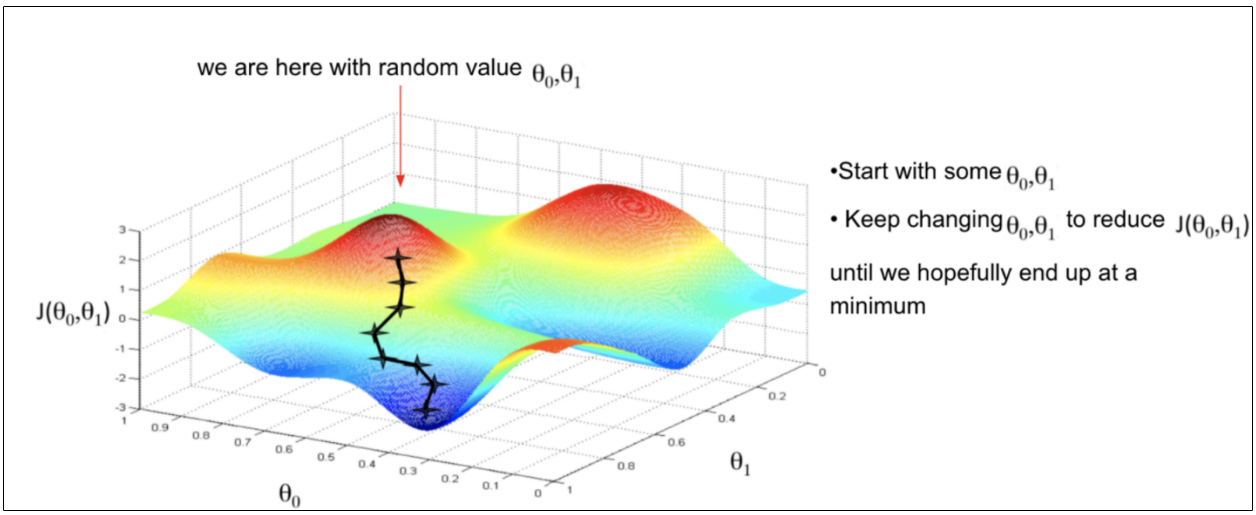
Mini-batch sizes, commonly called “batch sizes” for brevity, are often tailored to some aspect of the computing architecture in which the implementation is running. For example, a power of 2 that matches the memory requirements of the GPU or CPU hardware, such as 32, 64, 128, and 256.

The stack size is a slider for the learning process.

Smaller values allow the learning process to converge quickly at the expense of noise in the training process. Larger values result in a learning process that slowly converges to an accurate estimate of the error gradient.

Advanced Optimization Algorithms:

Gradient Descent is one of the most popular and widely used optimization algorithms. Most of you must have implemented it, for finding the values of parameters that will minimize the cost function. In this article, I’ll tell you about some advanced optimization algorithms, through which you can run logistic regression (or even linear regression) much more quickly than gradient descent. Also, this will let the algorithms scale much better, to very large machine learning problems i.e. where we have a large number of features.



If we have a cost function, say J, and we want to minimize it. We write a code that takes input parameters, say Θ (theta), and computes J(Θ) and its partial derivatives. So, given the code that does these two things, gradient descent will repeatedly perform an update, to minimize the function for us.

Similarly, there are some advanced algorithms, if we provide a way to compute these two things, they can minimize the cost function with their sophisticated strategies.

Types of Advanced Algorithms

We'll discuss the three main types of such algorithms which are very useful where a large number of features are involved.

1. Conjugate Gradient:

It is an iterative algorithm, for solving large sparse systems of linear equations. Mainly, it's used for optimization, neural net training, and image restoration. Theoretically, it is defined as a method that produces an exact solution after a finite number of iterations. However, practically we can't get the exact solution as it is unstable w.r.t small perturbations. This is a good option for high-dimensional models.

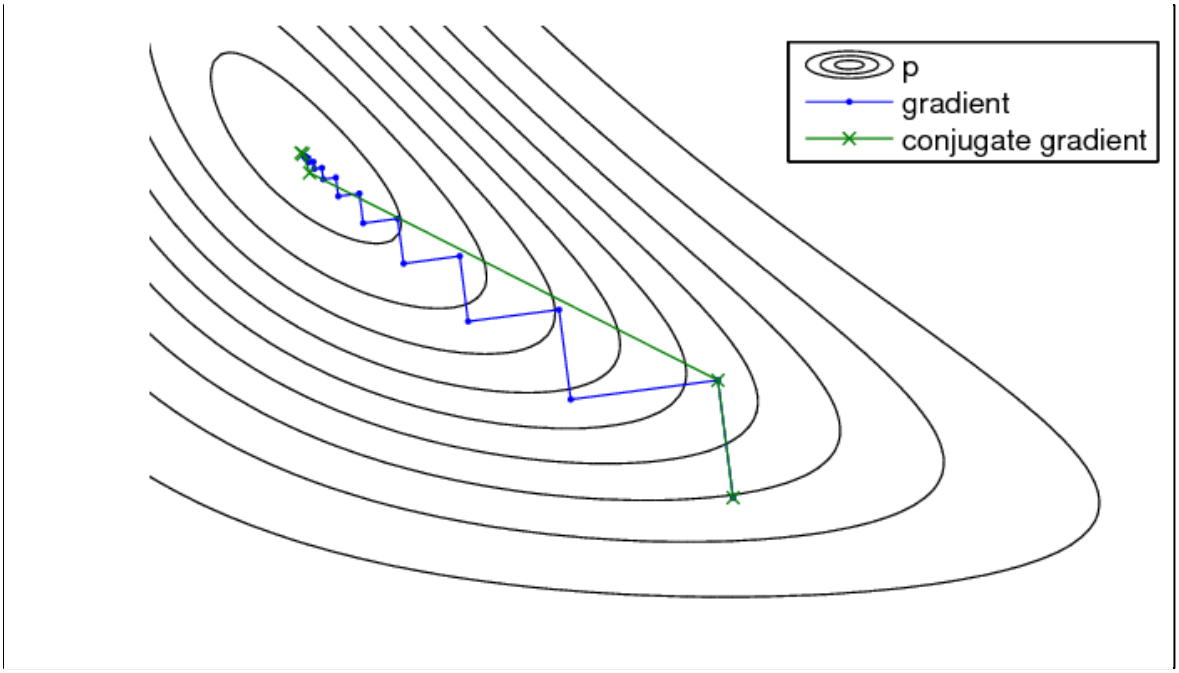


Image source: researchgate.net

2. BFGS:

It stands for Broyden Fletcher Goldfarb Shanno. It is also an iterative algorithm that is used for solving unconstrained non-linear optimization problems. It basically determines the descent direction, by preconditioning the gradient with curvature information. It is done gradually by improving approximation to the Hessian matrix, of the loss function.

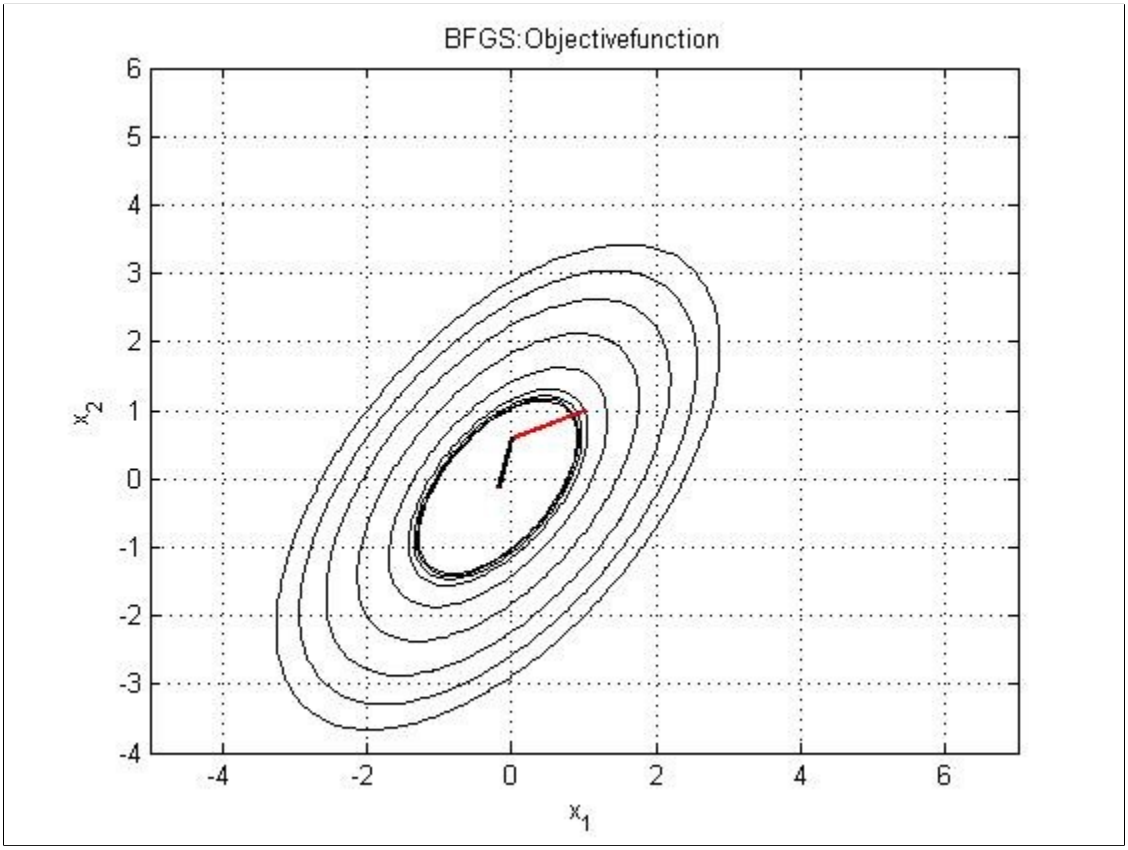


Image source:mathworks.com

3. L-BFGS:

This is basically a limited memory version of BFGS, mostly suited to problems with many variables (more than 1000). We can get a better solution with a smaller number of iterations. The L-BFGS line search method uses log-linear convergence rates, which reduces the number of line search iterations. This is a good option for low-dimensional models.

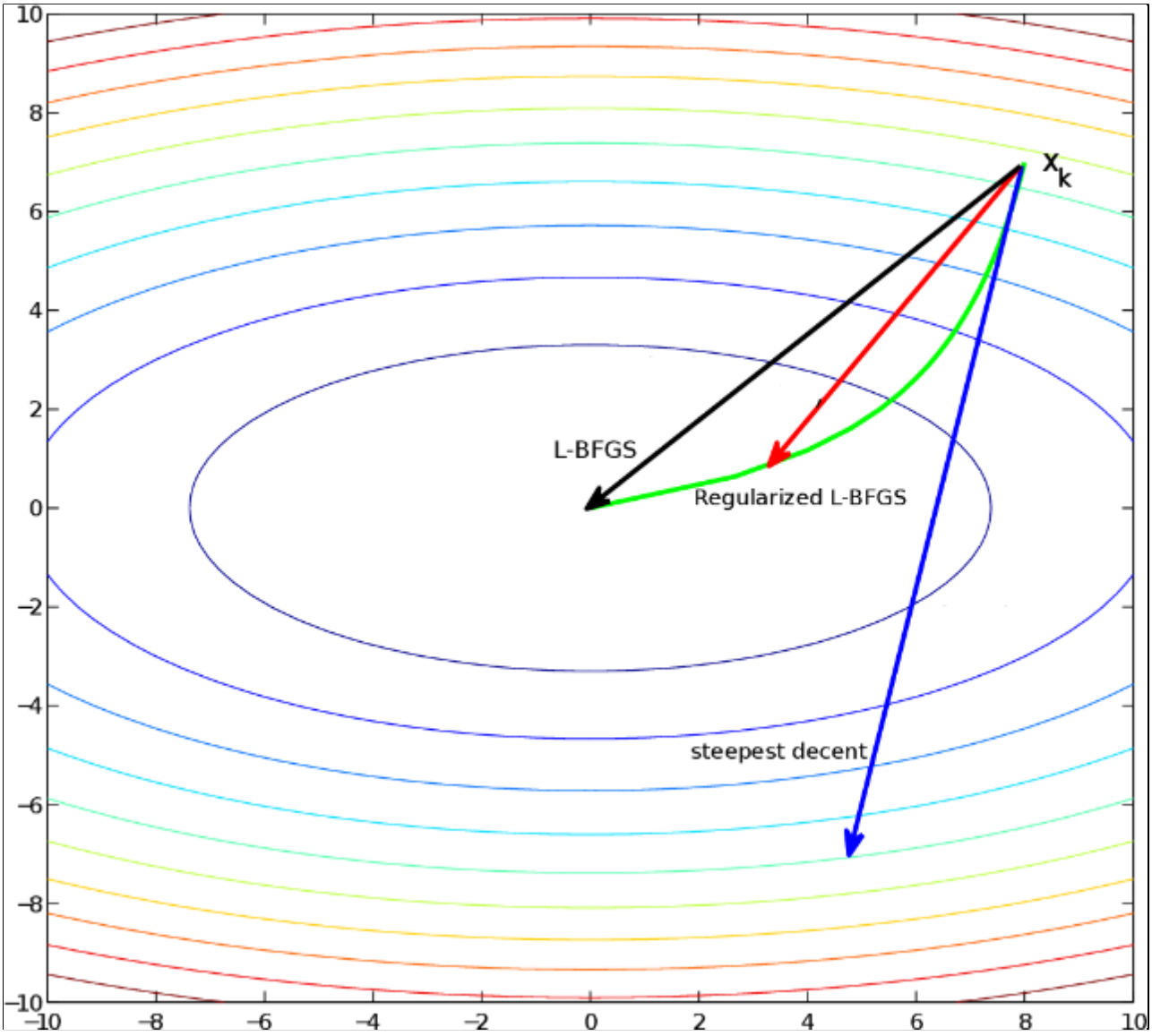


Image source: semanticscholar.org

Advantages and disadvantages over Gradient Descent:

These algorithms have a number of advantages –
1. You don't have to choose the learning rate manually. They have a clever inter loop called line search algorithm that automatically chooses a good learning rate and even a different learning rate for every iteration.

2. They end up converging much faster than gradient descent.

Example and its Implementation

I recommend you to use Octave or MATLAB, as it has a very good library that can implement these algorithms. So, just by using this library, we can get pretty cool results.

Now, I'll explain how to use these algorithms with an example. Suppose you have been given a problem with two parameters and a cost function as shown below.

- **Given problem**

$$\Theta = \begin{bmatrix} \theta 1 \\ \theta 2 \end{bmatrix}$$

$$J(\Theta) = (\Theta_1 - 12)^2 + (\Theta_2 - 12)^2$$

$$\partial/\partial\Theta_1 \, J(\Theta) = 2(\Theta_1 - 12)$$

$$\partial/\partial\Theta_2 \, J(\Theta) = 2(\Theta_2 - 12)$$

An example showing cost function and its derivatives

In the above figure, we are given Θ (theta) which is a 2×1 vector, a cost function J(Θ) and the last two equations are the partial derivates of the cost function. So, we clearly come to know that the values, $\Theta_1 = 12, \Theta_2 = 12$ will minimize the cost function. Now, we'll apply one of the advanced optimization algorithms to minimize this cost function and verify the result. I'll show you an octave function, which can perform this.

- **Implementing cost function**

```
function [jVal, gradient] = costFunction(theta)
  jVal = (theta(1)-12)^2 +(theta(2)-12)^2  ;
  gradient = zeros(2,1);
  gradient(1) = 2*(theta(1)-12);
  gradient(2) = 2*(theta(1)-12);
```

This function basically returns 2 arguments –

1) *J-Val*: this computes the cost function

2) *Gradient*: It's a 2×1 vector. The two elements here correspond to the partial derivative terms (shown above)

Save this code in a file costFunction.m

- **Using *fminunc()***

After implementing the cost function, we can now call the advanced optimization function, ***fminunc*** (function minimization unconstrained) in Octave. It is called as shown below –

```
options = optimset('GradObj','on','MaxIter',100);
initialTheta = zeros(2,1)
[optTheta, functionVal, exitFlag] = fminunc(@costFunction, initialTheta, options)
```

First, you set options, it is basically like a data structure that stores the options you

want. The 'GradObj' 'on' sets the gradient objective parameter to ON, which means that you will be providing a gradient. I've set the maximum iterations to 100. Then, we'll provide an initial guess for theta, which is a 2×1 vector. The command below it, calls the *fminunc* function. The '@' symbol there, represents a pointer to the cost function which we defined above. Now, this function, once called, will use one of the advanced optimization algorithms to minimize our cost function.

- **Code and output on Octave window**
  I've implemented the above example in Octave and below you can see the command and output.

```
>> options = optimset('GradObj','on','MaxIter',100);
>> initialTheta = zeros(2,1)
initialTheta =

   0
   0

>> [optTheta, functionVal, exitFlag] = fminunc(@costFunction, initialThe
optTheta =

   12.000
   12.000

functionVal = 3.1554e-30
exitFlag = 1
>>
```

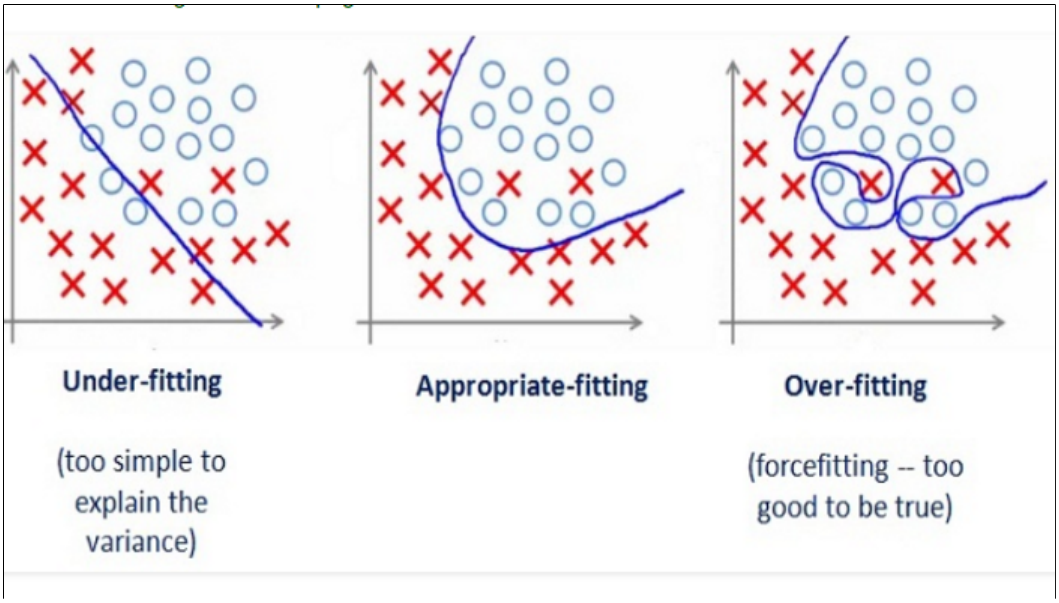
Octave window

As you can see, we got our **optimal theta ($\Theta_1 = 12, \Theta_2 = 12$) as we expected**. The exit flag basically shows the convergence status, that our algorithm has converged. So, this is the way you can implement these advanced algorithms. Also, I'd like to mention that, **this *fminunc* function will work only if the parameter theta, is a vector and not a real number.**

Regularization - Solving the problems on overfitting :

Regularization ,When training a machine learning model, the model can be easily overfitted or under fitted. To avoid this, we use regularization in machine learning to properly fit the model to our test set. Regularization techniques help reduce the possibility of overfitting and help us obtain an optimal model.

In machine learning, training and testing are based on the data; information from which a solution or set of solutions can be generated using data models. Regularization techniques address the prevention of ill-posed problems; problems where "the solution is highly sensitive to changes in the final data" (Wikipedia). Errors or problems with the data or method of inputting the data can lead to larger errors in the solutions. Making modifications to the algorithm to compensate for error and limitations which may occur creates more accurate and useful models.



Overfitting (see above), for example, is when the model learns too well how to fit the data based on the training model. This leads the model to be less accurate and "negatively impacts the performance of the model on new data" (Brownlee). If the model is only looking for and responding to a particular set of data based on the training data it will overlook changes and trends in new data. Regularization can omit and include segments of data in such a manner from training set to training set. The same overall data is used but different segments of data, fluctuating the data prevents overfitting.

Regularization Techniques

Regularization techniques modify how the data is input to compensate for any errors that may occur with the data; making the solutions more useful and accurate.

L1 & L2 Regularization

The activation function is a mathematical equation that acts on the input thus producing an output based on the needs of the learning model. Weights are multiplied by inputs and determine how fast the activation function will act upon the input. Both L1 Regularization and L2 Regularization impact the weights used in a model. L1 and L2 Regularization are based on the Cost Function; a function that measures the performance of models based on data (towardsdatascience.com). To the Cost function is added the regularization term and the sum of the weight, with variations between the two techniques. Weight values decrease as a result and create simpler models. Simpler models, for example, help to reduce overfitting.

L1 Regularization

Weight value can be reduced to zero. This impacts the output by increasing the speed at which the activation function will act upon the data. L1 regularization is a useful technique when compressing a model.

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum ||w||$$

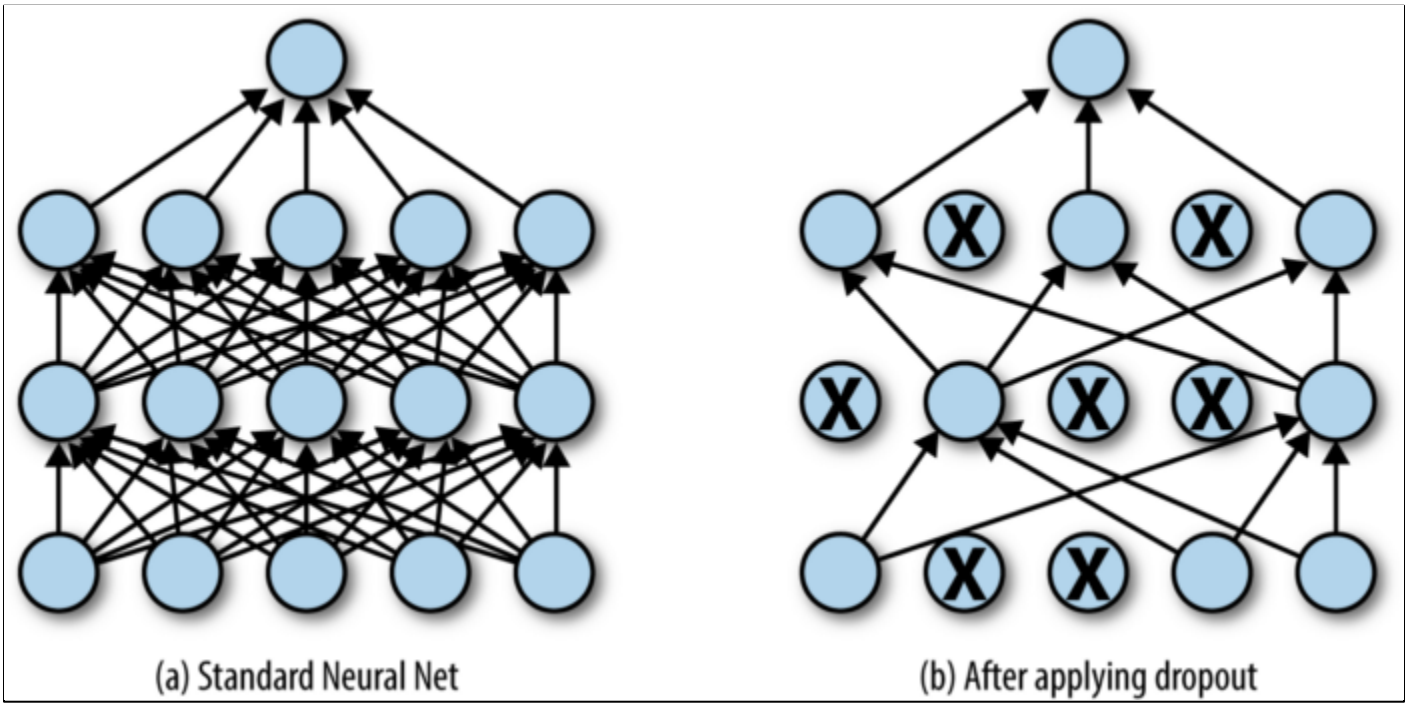
The sum of the absolute value of the weights is always going to be positive and can be zero; useful when compressing a model. Lambda represents the regularization parameter which is determined based on which number presents the best results needed. L1 regularization produces a sparse model. The norm is not differentiable and for gradient-based learning model changes in the learning, an algorithm may be needed.

L2 Regularization

L2 Regularization is referred to as “weight decay”. The purpose of this technique is to reduce overfitting by reducing the magnitude or size of the weights. This concept is based on the idea that the larger the weight value the larger to errors that can occur. The smaller the weight value is meant to reduce the number of errors that can occur.

$$Cost\,function = Loss + \frac{\lambda}{2m} * \sum ||w||^2$$

Unlike with L1 regularization, the value of the weights cannot be zero. Above a function for L2 regularization using the Cost function. Weights (w) is squared and multiplied by the regularization parameter (lambda) expression. The larger the lambda value the smaller the magnitude of weights will become. When choosing the best value for lambda difficulties may arise; cross-validation methods to verify the validity of the results by accurately figuring out what the response would be to the unseen data.



Dropout Regularization:

Dropout regularization randomly ignores some nodes from the neural network as well as the input and output connections. Each node contains weighted input, transfer functions, input and output connections. Each node impacts the output from a Neural Network. There can be many nodes in each network During dropout a node is completely removed from the network. Ignored nodes changes with each iteration impacting the results. Dropout is commonly used because of its consistency and good results. It is good for "training a large number of neural networks with different architectures in parallel" (Brownlee 2019). Challenges presented by Dropout include making the training process noisy. Because Dropout simulates sparse activation a network will have to learn sparse representation (Brownlee 2019). Layer outputs are randomly subsampled which, during training, reduces the capacity of the network.

Perceptron:
Building Block of Artificial Neural Network

Biological inspiration of Neural Networks

A neuron (nerve cell) is the basic building block of the nervous system. A human brain consists of billions of neurons that are interconnected to each other. They are responsible for receiving and sending signals from the brain. As seen in the below diagram, a typical neuron consists of the three main parts – dendrites, an axon, and cell body or soma. Dendrites are tree-like branches originating from the cell body. They receive information from the other neurons. Soma is the core of a neuron. It is responsible for processing the information received from dendrites. Axon is like a cable through which the neurons send the information. Towards its end, the axon splits up into many branches that make connections with the other neurons through their dendrites. The connection between the axon and other neuron dendrites is called synapses.

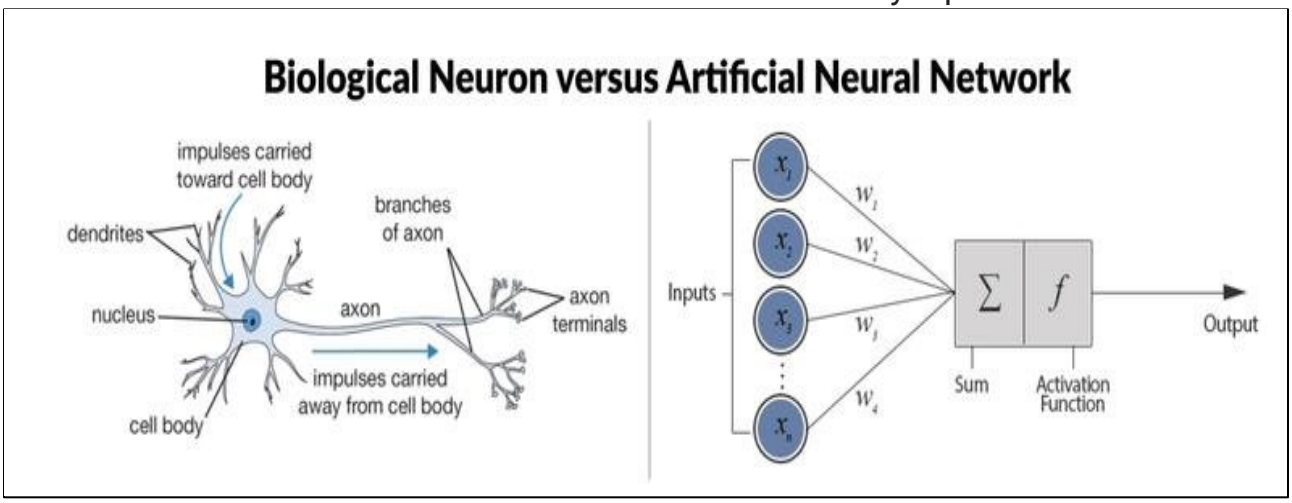


Image Source: Willems, K. (2017, May 2). Keras Tutorial: Deep Learning in Python.

As ANN is inspired by the functioning of the brain, let us see how the brain works. The brain consists of a network of billions of neurons. They communicate by means of electrical and chemical signals through a synapse, in which, the information from one neuron is transmitted to other neurons. The transmission process involves an electrical impulse called 'action potential'. For the information to be transmitted, the input signals (impulse) should be strong enough to cross a certain threshold barrier, then only a neuron activates and transmits the signal further (output).

Inspired by the biological functioning of a neuron, an American scientist Franck Rosenblatt came up with the concept of perceptron at Cornell Aeronautical Laboratory in 1957.

- A neuron receives information from other neurons in form of electrical impulses of varying strength.
- Neuron integrates all the impulses it receives from the other neurons.
- If the resulting summation is larger than a certain threshold value, the neuron 'fires', triggering an action potential that is transmitted to the other connected neurons.

Main Components of Perceptron

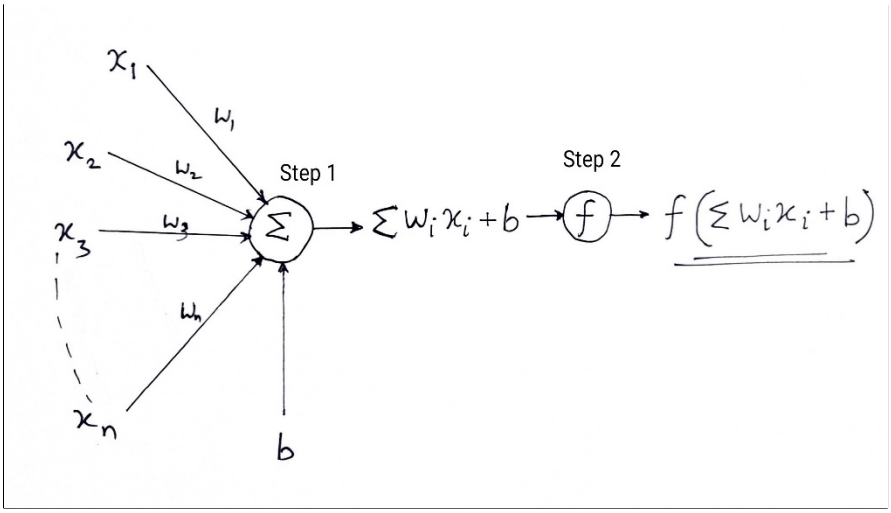
Rosenblatt's perceptron is basically a binary classifier. The perceptron consists of 3 main parts:

- Input nodes or input layer: The input layer takes the initial data into the system for further processing. Each input node is associated with a numerical value. It can take any real value.
- Weights and bias: Weight parameters represent the strength of the connection between units. Higher is the weight, stronger is the influence of the associated input neuron to decide the output. Bias plays the same as the intercept in a linear equation.
- Activation function: The activation function determines whether the neuron will fire or not. At its simplest, the activation function is a step function, but based on the scenario, different activation functions can be used.
  We shall see more about these in the subsequent section.

Working of a Perceptron

In the first step, all the input values are multiplied with their respective weights and added together. The result obtained is called weighted sum $\sum wi*xi$, or stated differently, $x1*w1 + x2*w2 + \ldots wn*xn$. This sum gives an appropriate representation of the inputs based on their importance. Additionally, a bias term *b* is added to this sum $\sum wi*xi + b$. Bias serves as another model parameter (in addition to weights) that can be tuned to improve the model's performance.

In the second step, an activation function *f* is applied over the above sum $\sum wi*xi + b$ to obtain output $Y = f(\sum wi*xi + b)$. Depending upon the scenario and the activation function used, the Output is either binary {1, 0} or a continuous value.



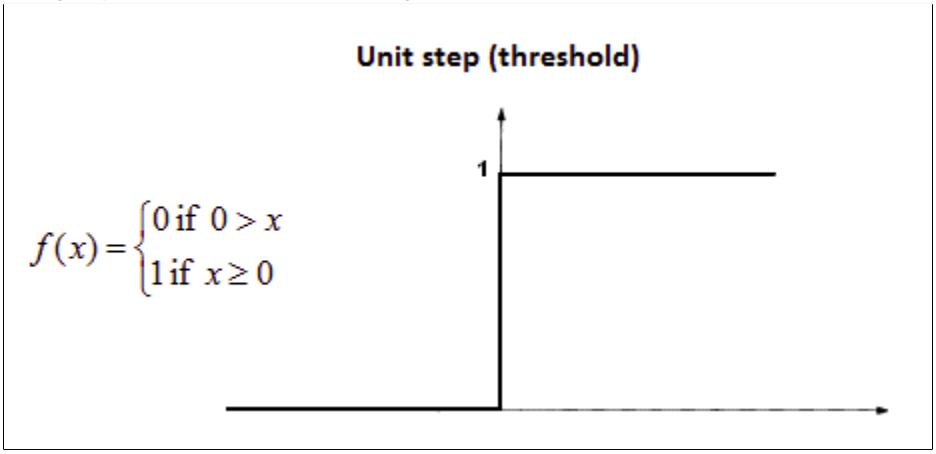
(Often both these steps are represented as a single step in multi-layer perceptrons, here I have shown them as two different steps for better understanding)

Activation Functions

A biological neuron only fires when a certain threshold is exceeded. Similarly, the artificial neuron will also only fire when the sum of the inputs (weighted sum) exceeds a certain threshold value, let's say 0. Intuitively, we can think of a rule-based approach like this –

```
If ∑wi*xi + b > 0:
   output = 1
else:
   output = 0
```

Its graph will be something like this:



This is in fact the Unit Step (Threshold) activation function which was originally used by Rosenblatt. But as you can see, this function is discontinuous at 0, so it causes problems in mathematical computations. A smoother version of the above function is the sigmoid function. It outputs between 0 and 1. Another one is the Hyperbolic tangent(tan*h*) function, which produces the output between -1 and 1. Both sigmoid and tanh functions suffer from vanishing gradients problems. Nowadays, ReLU and Leaky ReLU are the most popularly used activation functions. They are comparatively stable over deep networks.

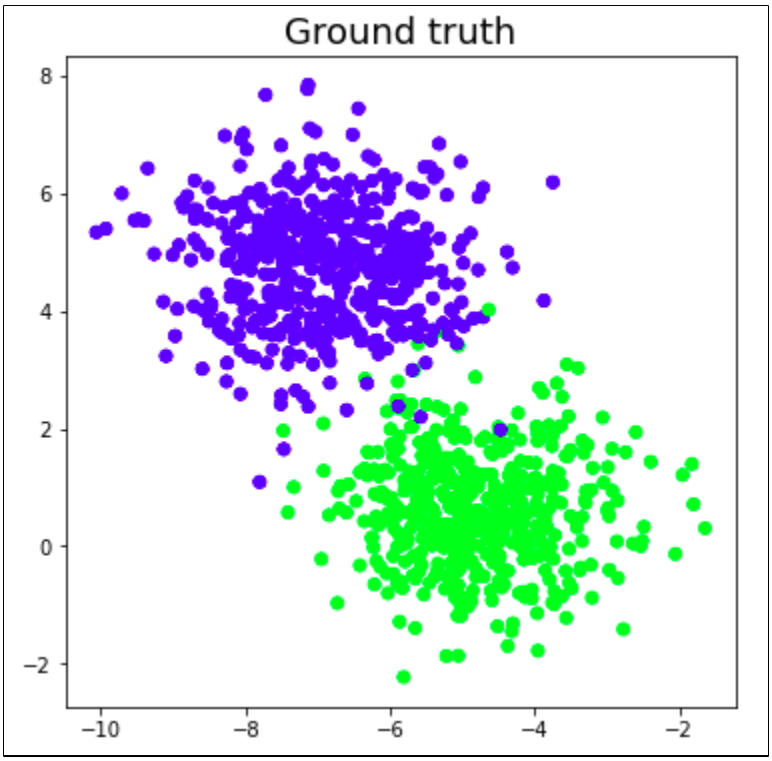
Perceptron as a Binary Classifier

So far, we have seen the biological inspiration and the mathematics of the perceptron. In this section, we shall see how a perceptron solves a linear classification problem.
Importing some libraries –

```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

Generating a dummy dataset using *make_blobs* functionality provided by scikit learn –

```
# Generate dataset
X, Y = make_blobs(n_features = 2, centers = 2, n_samples = 1000, random_state = 12)
# Visualize dataset
plt.figure(figsize = (6, 6))
plt.scatter(X[:, 0], X[:, 1], c = Y)
plt.title('Ground truth', fontsize = 18)
plt.show()
```



Let's say the blue dots are 1s and the green dots are 0s. Using perceptron logic, we can create a decision boundary(hyperplane) for classification which separates different data points on the graph.
Before we proceed further, let's add a bias term (ones) to the input vector –

```
# Add a bias to the input vector
```

```
X_bias = np.ones([X.shape[0], 3])
X_bias[:, 1:3] = X
```

The dataset will look something like this –

```
array([[ 1.        , -4.82009419,  0.31265986],
       [ 1.        , -4.85798902, -0.75532612],
       [ 1.        , -5.87148063,  0.18708564],
       ...,
       [ 1.        , -5.41918388,  1.18580916],
       [ 1.        , -8.33802005,  3.63740559],
       [ 1.        , -3.54957757,  1.52533045]])
```

Here each row of the above dataset represents the input vector (a datapoint). In order to create a decision boundary, we need to find out the appropriate weights. The weights are 'learned' from the training using the below rule –

w = w + (expected — predicted) * x

$$w \equiv w + (Y_i - y)X_i$$

It means that subtracting the estimated outcome from the ground truth and then multiplying this by the current input vector and adding old weights to it in order to obtain the new value of the weights. If our output is the same as the actual class, then the weights do not change. But if our estimation is different from the ground truth, then the weights increase or decrease accordingly. This is how the weights are progressively adjusted in each iteration.
We start by assigning arbitrary values to the weight vector, then we progressively adjust them in each iteration using the error and data at hand –

```
# initialize weights with random values
w = np.random.rand(3, 1)
print(w)
```

Output:

```
[[0.37547448]
 [0.00239401]
 [0.18640939]]
```

Define the activation function of perceptron –

```
def activation_func(z):
   if z >= 1:
      return 1
   else:
      return 0
```

Next, we apply the perceptron learning rule –

```
for _ in range(100):
```

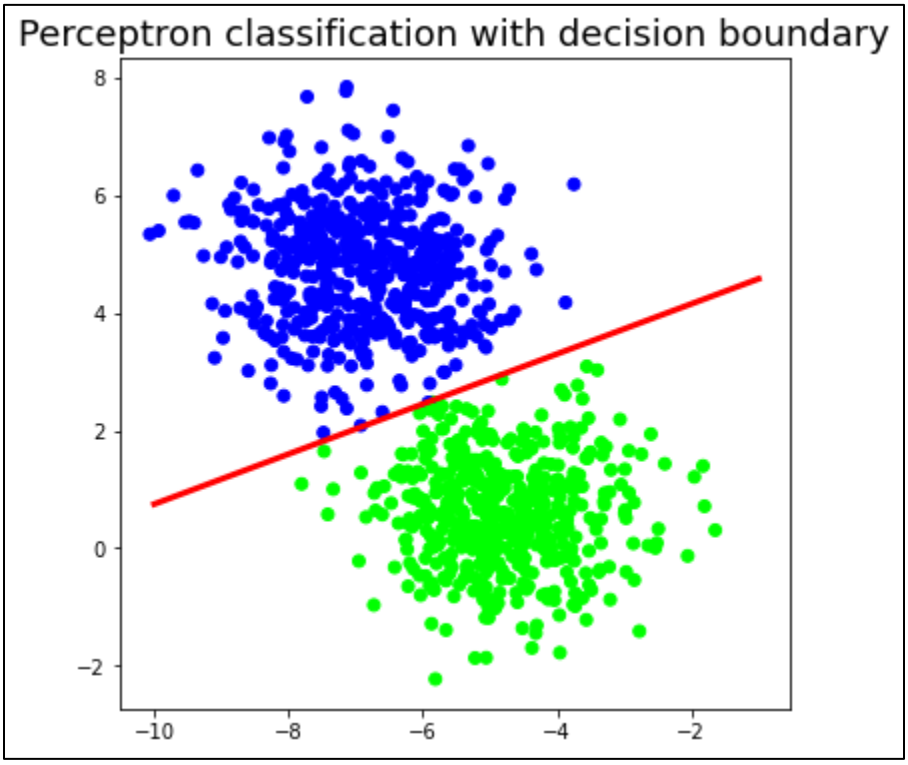
```
    for i in range(X_bias.shape[0]):
        y = activation_func(w.transpose().dot(X_bias[i, :]))
        # Update weights
        w = w + ((Y[i] - y) * X_bias[i, :]).reshape(w.shape[0], 1)
```

It is not guaranteed that the weights will converge in one pass, so we feed all the training data into the perceptron algorithm 100 times while constantly applying the learning rule so that eventually we manage to obtain the optimal weights.
Now, that we have obtained the optimal weights, we predict the class for each datapoint using Y = *f*(∑wi*xi + b) or Y = wT.x in vector form.

```
# predicting the class of the datapoints
result_class = [activation_func(w.transpose().dot(x)) for x in X_bias]
```

Visualize the decision boundary and the predicted class labels –

```
# convert to unit vector
w = w/np.sqrt(w.transpose().dot(w))
# Visualize results
plt.figure(figsize = (6, 6))
plt.scatter(X[:, 0], X[:, 1], c = result_class)
plt.plot([-10, -1], hyperplane([-10, -1], w), lw = 3, c = 'red')
plt.title('Perceptron classification with decision boundary')
plt.show()
```



You can compare the ground truth image with the predicted outcome image and see some points that are misclassified. If we calculate the accuracy, it comes to about 98% (I leave this as an exercise to the readers).
If you see, here our original data was fairly separated, so we are able to get such good accuracy. But this is not the case with real-world data. Using a single

perceptron, we can only construct a linear decision boundary, so if the data is intermixed the perceptron algorithm will perform poorly. This is one of the limitations of the single perceptron model.

Multi class Classification

A multi-class classification is a classification technique that allows us to categorize data with more than two class labels. Trained multi-class classifiers are able to predict labels for test data based on those that are present in training data. One-Vs-All Classification is a method of multi-class classification. It can be broken down by splitting up the multi-class classification problem into multiple binary classifier models. For k class labels present in the dataset, k binary classifiers are needed in One-vs-All multi-class classification.

Since binary classification is the foundation of One-vs-All classification, here is a quick review of binary classification before we explore One-vs-All classification further.

1.1 Review of Binary Classification Model

In binary classification, the given data $D = \{x_i, y_i\}_{i=1}^{n}$ is classified into two discrete classes:

$$y_i = \begin{cases} 0 & \text{class 1} \\ 1 & \text{class 2} \end{cases}$$

Binary classification problems requires only one classifier and its effectiveness is easily visualized and understood using a confusion matrix.[1].

		Predicted Class: Normal	Predicted Class: Attack
Actual Class: Normal		True Negative (TN)	False Positive (FP)
Actual Class: Attack		False Negative (FN)	True Positive (TP)

If you can solve a binary classification problem, you can solve a multi-class classification problem.

1.2 One-vs-All Classification

Again, one-vs-all classification breaks down k classes present in our dataset D into k binary classifier models that aims to classify a data point as either part of the current class k_i or as part of all other classes. Each model can discriminate the i^{th} class with everything else.

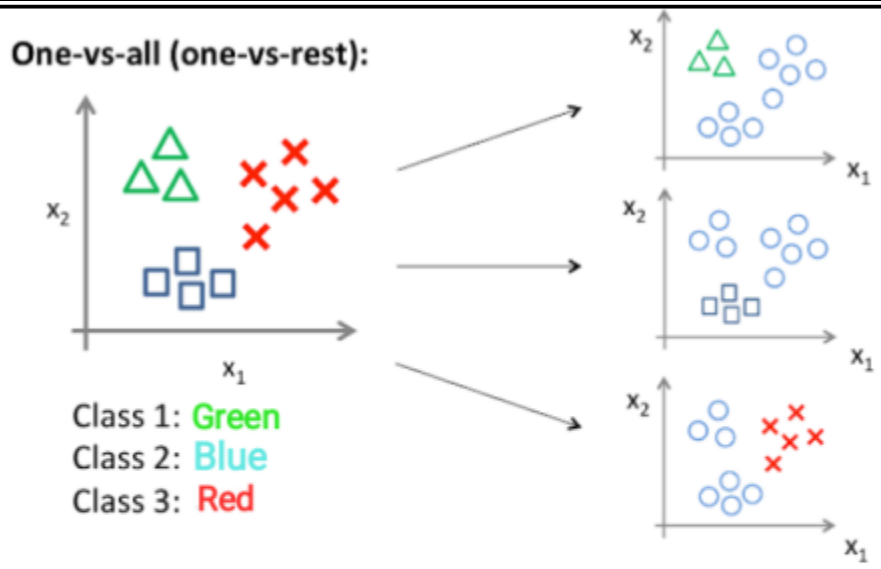
Example: Suppose you have classes A, B, and C. We will build one model for each class:

Model 1: A or BC
Model 2: B or AC
Model 3: C or AB

Another way to think about the models is each class vs everything else (hence the name):

Model 1: A or not A
Model 2: B or not B
Model 3: C or not C

A visual representation of One-vs-All classification can be seen below[2].



1.2.1 Confidence (or score) about a Prediction

The Problem: Suppose your confidence that the chances a data point belongs to Class 1 is very low. Logically, we know this should increase our confidence that this particular data point belongs to all the other classes. However, in One-vs-All classification, each binary classifier (Perceptron) is completely independent from all other $k-1$ classifiers that have been built to model the dataset at hand, meaning the probabilities output from each binary classifier need not sum to 1.

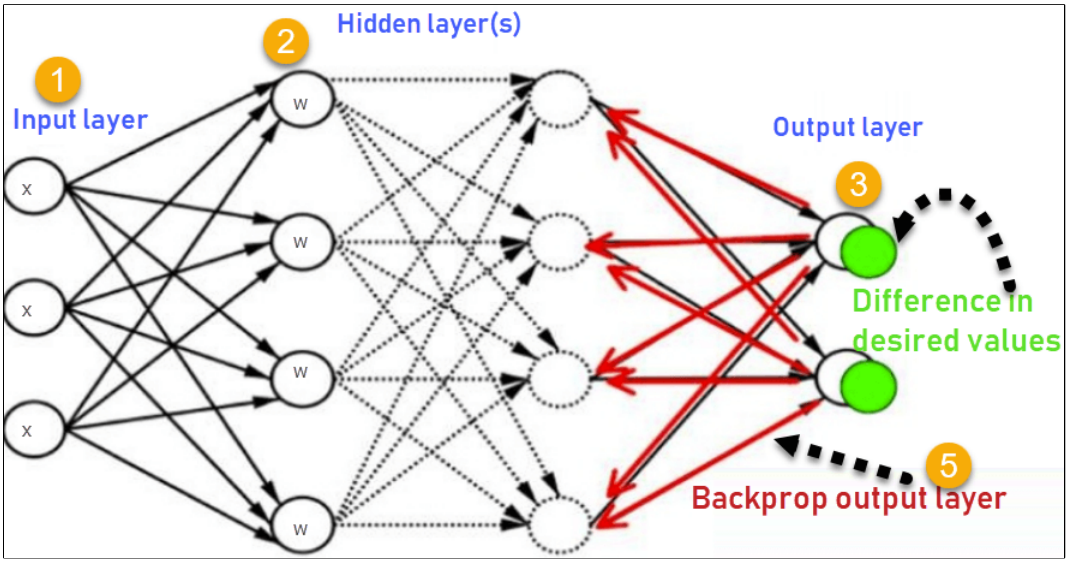
Back Propagation in Neural Network:

Backpropagation is the essence of neural network training. It is the method of fine-tuning the weights of a neural network based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights allows you to reduce error rates and make the model reliable by increasing its generalization.

Backpropagation in neural network is a short form for “backward propagation of errors.” It is a standard method of training artificial neural networks. This method helps calculate the gradient of a loss function with respect to all the weights in the network.

How Backpropagation Algorithm Works

The Back propagation algorithm in neural network computes the gradient of the loss function for a single weight by the chain rule. It efficiently computes one layer at a time, unlike a native direct computation. It computes the gradient, but it does not define how the gradient is used. It generalizes the computation in the delta rule. Consider the following Back propagation neural network example diagram to understand:



How Backpropagation Algorithm Works

1. Inputs X, arrive through the preconnected path
2. Input is modeled using real weights W. The weights are usually randomly selected.
3. Calculate the output for every neuron from the input layer, to the hidden layers, to the output layer.
4. Calculate the error in the outputs

$Error_B$= Actual Output – Desired Output

5. Travel back from the output layer to the hidden layer to adjust the weights such that the error is decreased.

Keep repeating the process until the desired output is achieved

Why We Need Backpropagation?

Most prominent advantages of Backpropagation are:

- Backpropagation is fast, simple and easy to program
- It has no parameters to tune apart from the numbers of input
- It is a flexible method as it does not require prior knowledge about the

network
- It is a standard method that generally works well
- It does not need any special mention of the features of the function to be learned.

What is a Feed Forward Network?

A feedforward neural network is an artificial neural network where the nodes never form a cycle. This kind of neural network has an input layer, hidden layers, and an output layer. It is the first and simplest type of artificial neural network.

Types of Backpropagation Networks

Two Types of Backpropagation Networks are:

- Static Back-propagation
- Recurrent Backpropagation

Static back-propagation:

It is one kind of backpropagation network which produces a mapping of a static input for static output. It is useful to solve static classification issues like optical character recognition.

Recurrent Backpropagation:

Recurrent Back propagation in data mining is fed forward until a fixed value is achieved. After that, the error is computed and propagated backward.
The main difference between both of these methods is: that the mapping is rapid in static back-propagation while it is nonstatic in recurrent backpropagation.

History of Backpropagation

- In 1961, the basics concept of continuous backpropagation were derived in the context of control theory by J. Kelly, Henry Arthur, and E. Bryson.
- In 1969, Bryson and Ho gave a multi-stage dynamic system optimization method.
- In 1974, Werbos stated the possibility of applying this principle in an artificial neural network.
- In 1982, Hopfield brought his idea of a neural network.
- In 1986, by the effort of David E. Rumelhart, Geoffrey E. Hinton, Ronald J. Williams, backpropagation gained recognition.
- In 1993, Wan was the first person to win an international pattern recognition contest with the help of the backpropagation method.

Backpropagation Key Points

- Simplifies the network structure by elements weighted links that have the least effect on the trained network
- You need to study a group of input and activation values to develop the relationship between the input and hidden unit layers.

- It helps to assess the impact that a given input variable has on a network output. The knowledge gained from this analysis should be represented in rules.
- Backpropagation is especially useful for deep neural networks working on error-prone projects, such as image or speech recognition.
- Backpropagation takes advantage of the chain and power rules allows backpropagation to function with any number of outputs.

Best practice Backpropagation

Backpropagation in neural network can be explained with the help of “Shoe Lace” analogy

Too little tension =

- Not enough constraining and very loose

Too much tension =

- Too much constraint (overtraining)
- Taking too much time (relatively slow process)
- Higher likelihood of breaking

Pulling one lace more than other =

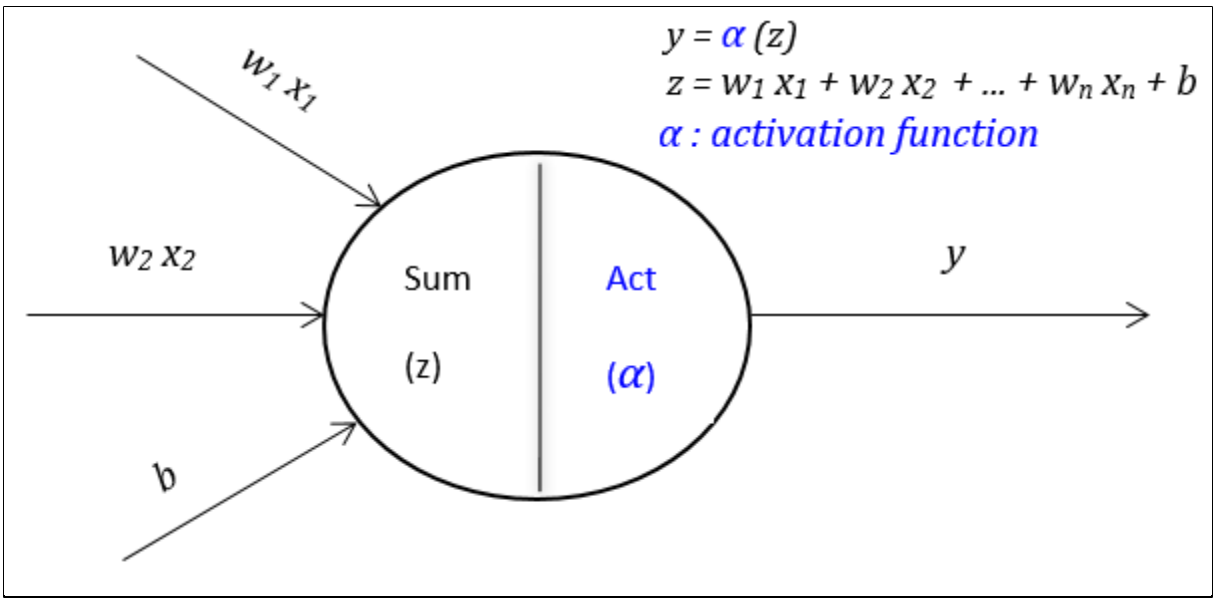
- Discomfort (bias)

Disadvantages of using Backpropagation

- The actual performance of backpropagation on a specific problem is dependent on the input data.
- Back propagation algorithm in data mining can be quite sensitive to noisy data
- You need to use the matrix-based approach for backpropagation instead of mini-batch.

Non-linearity with activation functions (Tanh, Sigmoid, Relu, PRelu)

Activation functions are functions used in a neural network to compute the weighted sum of inputs and biases, which is in turn used to decide whether a neuron can be activated or not. It manipulates the presented data and produces an output for the neural network that contains the parameters in the data. The activation functions are also referred to as *transfer functions* in some literature. These can either be linear or nonlinear
depending on the function it represents and is used to control the output of neural networks across different domains.
Why Activation Functions?



The need for these activation functions includes converting the linear input signals and models into non-linear output signals, which aids the learning of high order polynomials for deeper networks.

How to use it?

In a neural network every neuron will do two computations:

- *Linear summation of inputs:* In the above diagram, it has two inputs x_1, x_2 with weights w_1, w_2, and bias b. And the linear sum $z = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$
- *Activation computation:* This computation decides, whether a neuron should be activated or not, by calculating the weighted sum and further adding bias with it. The purpose of the activation function is to introduce non-linearity into the output of a neuron.

Most neural networks begin by computing the weighted sum of the inputs. Each node in the layer can have its own unique weighting. However, the activation function is the same across all nodes in the layer. They are typical of a fixed form whereas the weights are considered to be the learning parameters.

What is a Good Activation Function?

A proper choice has to be made in choosing the activation function to improve the results in neural network computing. All activation functions must be monotonic, differentiable, and quickly converging with respect to the weights for optimization purposes.

Types of Activation Functions

The different kinds of activation functions include:

1) Linear Activation Functions

A linear function is also known as a straight-line function where the activation is proportional to the input i.e. the weighted sum from neurons. It has a simple function with the equation:

$f(x) = ax + c$

The problem with this activation is that it cannot be defined in a specific range. Applying this function in all the nodes makes the activation function work like linear regression. The final layer of the Neural Network will be working as a linear function of the first layer. Another issue is the gradient descent when differentiation is done, it has a constant output which is not good because during backpropagation the rate of change of error is constant that can ruin the output and the logic of backpropagation.

2) Non-Linear Activation Functions

The non-linear functions are known to be the most used activation functions. It makes it easy for a neural network model to adapt with a variety of data and to differentiate between the outcomes.

These functions are mainly divided basis on their range or curves:

a) Sigmoid Activation Functions

Sigmoid takes a real value as the input and outputs another value between 0 and 1. The sigmoid activation function translates the input ranged in $(-\infty,\infty)$ to the range in (0,1)

b) Tanh Activation Functions

The tanh function is just another possible function that can be used as a non-linear activation function between layers of a neural network. It shares a few things in common with the sigmoid activation function. Unlike a sigmoid function that will map input values between 0 and 1, the Tanh will map values between -1 and 1. Similar to the sigmoid function, one of the interesting properties of the tanh function is that the derivative of tanh can be expressed in terms of the function itself.

c) ReLU Activation Functions

The formula is deceptively simple: $max(0,z)$. Despite its name, Rectified Linear Units, it's not linear and provides the same benefits as Sigmoid but with better performance.

(i) Leaky Relu

Leaky Relu is a variant of ReLU. Instead of being 0 when $z<0$, a leaky ReLU allows a small, non-zero, constant gradient α (normally, $\alpha=0.01$). However, the consistency of the benefit across tasks is presently unclear. Leaky ReLUs attempt to fix the "dying ReLU" problem.

(ii) Parametric Relu

PReLU gives the neurons the ability to choose what slope is best in the negative region. They can become ReLU or leaky ReLU with certain values of α.

d) Maxout:

The Maxout activation is a generalization of the ReLU and the leaky ReLU functions. It is a piecewise linear function that returns the maximum of inputs, designed to be used in conjunction with the dropout regularization technique. Both ReLU and leaky ReLU are special cases of Maxout. The Maxout neuron, therefore, enjoys all the benefits of a ReLU unit and does not have any drawbacks like dying ReLU. However, it doubles the total number of parameters for each neuron, and hence, a higher total number of parameters need to be trained.

e) ELU

The Exponential Linear Unit or ELU is a function that tends to converge faster and produce more accurate results. Unlike other activation functions, ELU has an extra alpha constant which should be a positive number. ELU is very similar to ReLU except for negative inputs. They are both in the identity function form for non-negative inputs. On the other hand, ELU becomes smooth slowly until its output equal to -α whereas ReLU sharply smoothes.

f) Softmax Activation Functions

Softmax function calculates the probabilities distribution of the event over ‘n’ different events. In a general way, this function will calculate the probabilities of each target class over all possible target classes. Later the calculated probabilities will help determine the target class for the given inputs.

When to use which Activation Function in a Neural Network?

Specifically, it depends on the problem type and the value range of the expected output. For example, to predict values that are larger than 1, tanh or sigmoid are not suitable to be used in the output layer, instead, ReLU can be used. On the other hand, if the output values have to be in the range (0,1) or (-1, 1) then ReLU is not a good choice, and sigmoid or tanh can be used here. While performing a classification task and using the neural network to predict a probability distribution over the mutually exclusive class labels, the softmax activation function should be used in the last layer. However, regarding the hidden layers, as a rule of thumb, use ReLU as an activation for these layers.

In the case of a binary classifier, the Sigmoid activation function should be used. The sigmoid activation function and the tanh activation function work terribly for the hidden layer. For hidden layers, ReLU or its better version leaky ReLU should be used. For a multiclass classifier, Softmax is the best-used activation function. Though there are more activation functions known, these are known to be the most used activation functions.

Activation Functions and their Derivatives

Function Type	Equation	Derivative
Linear	$f(x) = ax + c$	$f'(x) = a$
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)\,(1 - f(x))$
TanH	$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ReLU	$f(x) = \begin{cases} 0 \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$
Parametric ReLU	$f(x) = \begin{cases} \alpha x \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$
ELU	$f(x) = \begin{cases} \alpha(e^x - 1) \text{ for } x<0 \\ x \text{ for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha \text{ for } x<0 \\ 1 \text{ for } x \geq 0 \end{cases}$

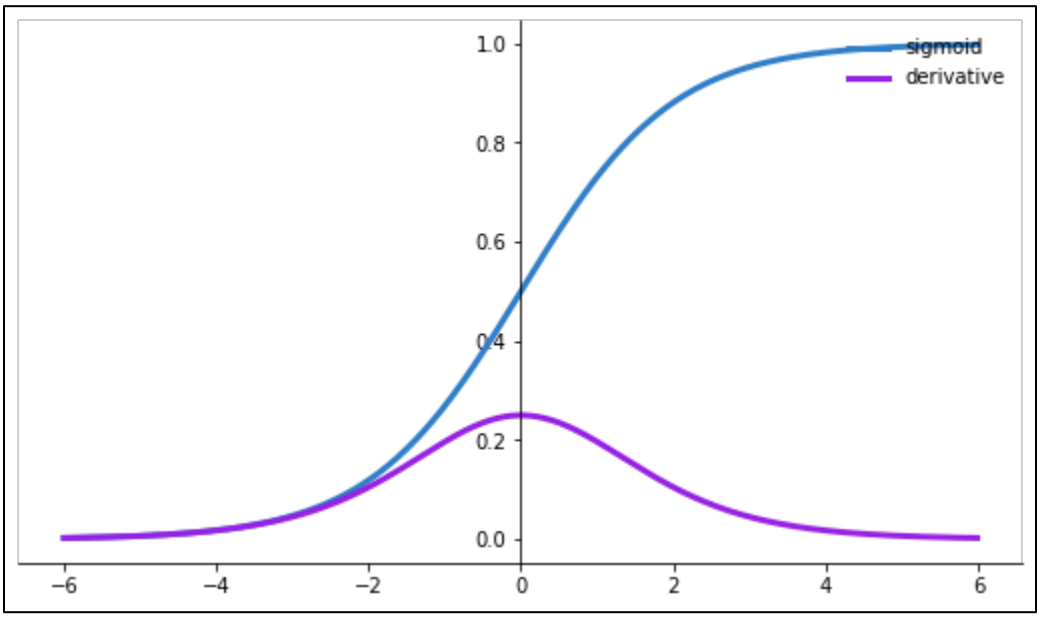
Implementation using Python

Having learned the types and significance of each activation function, it is also essential to implement some basic (non-linear) activation
functions using python code and observe the output for more clear understanding of the concepts:

Sigmoid Activation Function

```
import matplotlib.pyplot as plt
import numpy as np
def sigmoid(x):
   s=1/(1+np.exp(-x))
   ds=s*(1-s)
   return s,ds
x=np.arange(-6,6,0.01)
sigmoid(x)
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
```

```
ax.xaxis.set_ticks_position('bottom')
ax.yaxis.set_ticks_position('left')
ax.plot(x,sigmoid(x)[0], color="#307EC7", linewidth=3, label="sigmoid")
ax.plot(x,sigmoid(x)[1], color="#9621E2", linewidth=3, label="derivative")
ax.legend(loc="upper right", frameon=False)
fig.show()
```



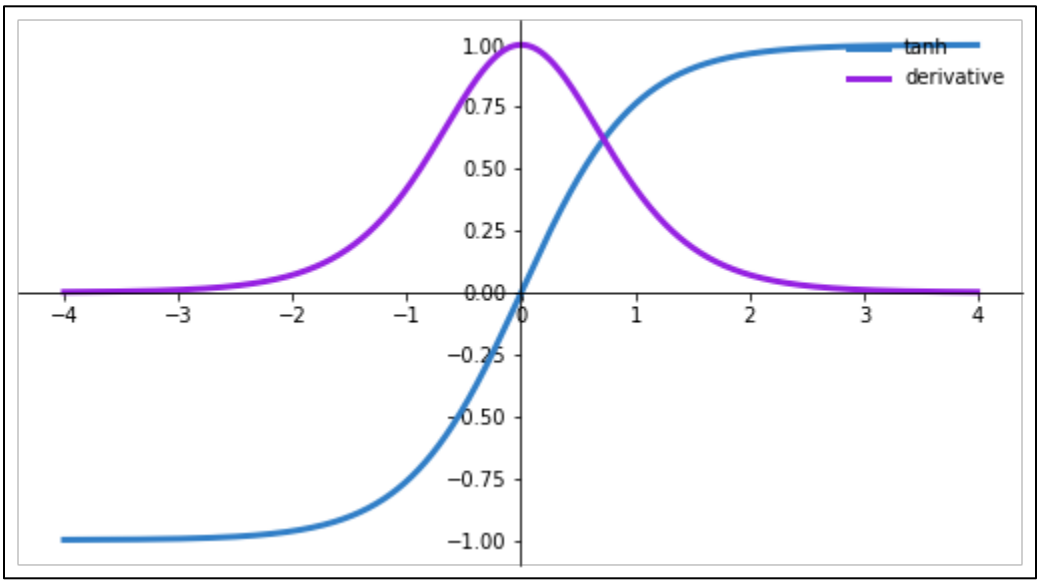
Observations:

- The sigmoid function has values between 0 to 1.
- The output is not zero-centered.
- Sigmoids saturate and kill gradients.
- At the top and bottom level of sigmoid functions, the curve changes slowly, the derivative curve above shows that the slope or gradient it is zero.

Tanh Activation Function

```
import matplotlib.pyplot as plt
import numpy as np
def tanh(x):
    t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
    dt=1-t**2
    return t,dt
z=np.arange(-4,4,0.01)
tanh(z)[0].size,tanh(z)[1].size
fig, ax = plt.subplots(figsize=(9, 5))
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('center')
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')
ax.xaxis.set_ticks_position('bottom')
```

```
ax.yaxis.set_ticks_position('left')
ax.plot(z,tanh(z)[0], color="#307EC7", linewidth=3, label="tanh")
ax.plot(z,tanh(z)[1], color="#9621E2", linewidth=3, label="derivative")
ax.legend(loc="upper right", frameon=False)
fig.show()
```



Observations:

- Its output is zero-centered because its range is between -1 to 1. i.e. -1 < output < 1.
- Optimization is easier in this method hence in practice it is always preferred over the Sigmoid function.

Pros and Cons of Activation Functions

Type of Function	*Pros*	*Cons*
Linear	It gives a range of activations, so it is not binary activation. It can definitely connect a few neurons together and if more than 1 fire, take the max and decide based on that.	It is a constant gradient and the descent is going to be on a constant gradient. If there is an error in prediction, the changes made by backpropagation are constant and not depending on the change in input.
Sigmoid	It is nonlinear in nature. Combinations of this function are also nonlinear. It will give an analog activation, unlike the step function.	Sigmoids saturate and kill gradients. It gives rise to a problem of “vanishing gradients” The network refuses to learn further or is

		drastically slow.
Tanh	The gradient is stronger for tanh than sigmoid i.e. derivatives are steeper.	Tanh also has a vanishing gradient problem.
ReLU	It avoids and rectifies the vanishing gradient problem. ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations.	It should only be used within hidden layers of a Neural Network Model. Some gradients can be fragile during training and can die. It can cause a weight update which will make it never activate on any data point again. Thus, ReLu could even result in Dead Neurons.
Leaky ReLU	Leaky ReLUs is one attempt to fix the "dying ReLU" problem by having a small negative slope	As it possesses linearity, it can't be used for complex Classification. It lags behind the Sigmoid and Tanh for some of the use cases.
ELU	Unlike ReLU, ELU can produce negative outputs.	For x>0, it can blow up the activation with the output range of [0, ∞].

When all of this is said and done, the actual purpose of an activation function is to feature some reasonably non-linear property to the function, which could be a neural network. A neural network, without the activation functions, might perform solely linear mappings from the inputs to the outputs, and also the mathematical operation throughout the forward propagation would be the dot-products between an input vector and a weight matrix.

Since one dot product could be a linear operation, sequent dot products would be nothing more than multiple linear operations repeated one after another. And sequent linear operations may be thought of as mutually single learn operations. To be able to work out extremely attention-grabbing stuff, the neural networks should be able to approximate the nonlinear relations from input features to the output labels.

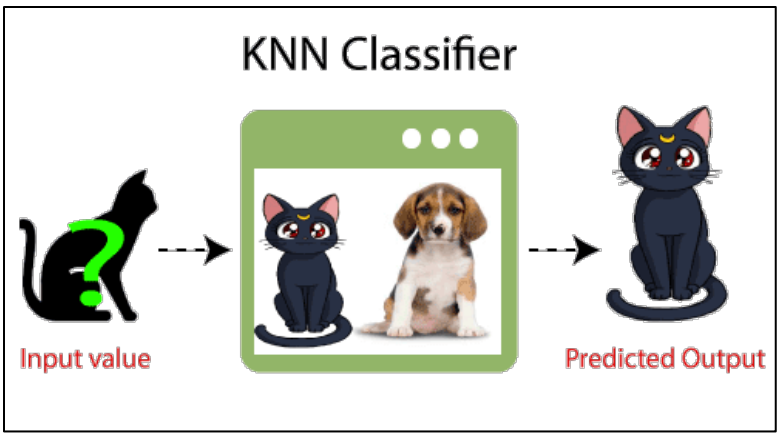
The more complicated the information, the more non-linear the mapping of features to the bottom truth label will usually be. If there is no activation function in a neural network, the network would in turn not be able to understand such complicated mappings mathematically and wouldn't be able to solve tasks that the network is really meant to resolve.

UNIT V NON PARAMETRIC 9
MACHINE LEARNING

k- Nearest Neighbors- Decision Trees – Branching – Greedy Algorithm - Multiple Branches – Continuous attributes – Pruning. Random Forests: ensemble learning. Boosting – Adaboost algorithm. Support Vector Machines – Large Margin Intuition – Loss Function - Hinge Loss – SVM Kernels

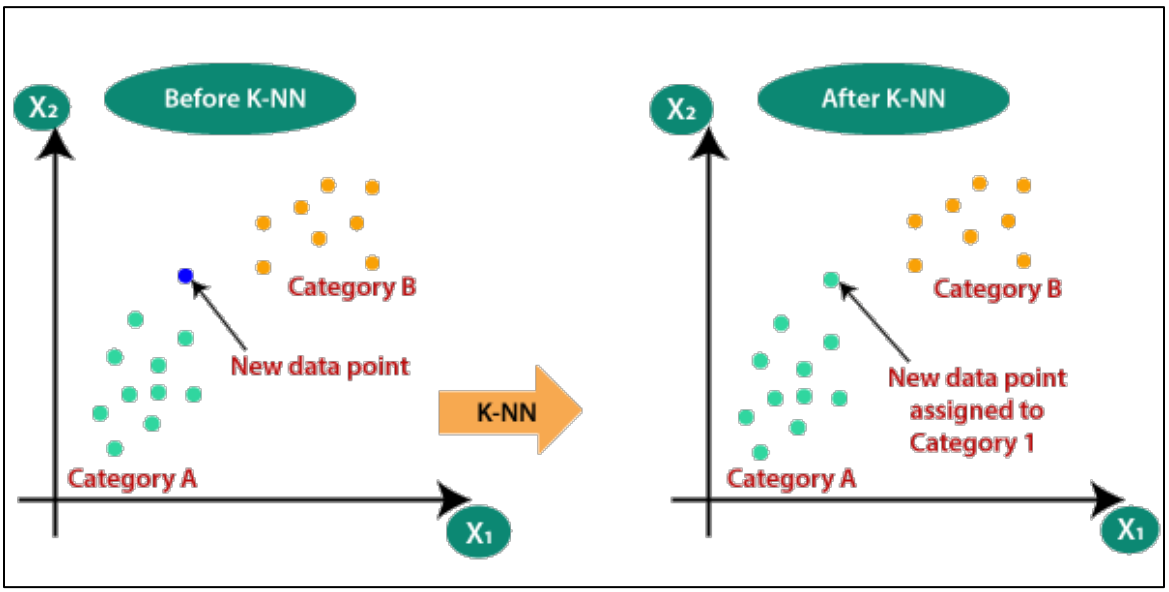
K-Nearest Neighbor (KNN)

- K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on Supervised Learning technique.
- K-NN algorithm assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories.
- K-NN algorithm stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using K- NN algorithm.
- K-NN algorithm can be used for Regression as well as for Classification but mostly it is used for the Classification problems.
- K-NN is a **non-parametric algorithm**, which means it does not make any assumption on underlying data.
- It is also called a **lazy learner algorithm** because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.
- KNN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similar to the new data.
- **Example:** Suppose, we have an image of a creature that looks similar to cat and dog, but we want to know either it is a cat or dog. So for this identification, we can use the KNN algorithm, as it works on a similarity measure. Our KNN model will find the similar features of the new data set to the cats and dogs images and based on the most similar features it will put it in either cat or dog category.



Why do we need a K-NN Algorithm?

Suppose there are two categories, i.e., Category A and Category B, and we have a new data point x1, so this data point will lie in which of these categories. To solve this type of problem, we need a K-NN algorithm. With the help of K-NN, we can easily identify the category or class of a particular dataset. Consider the below diagram:

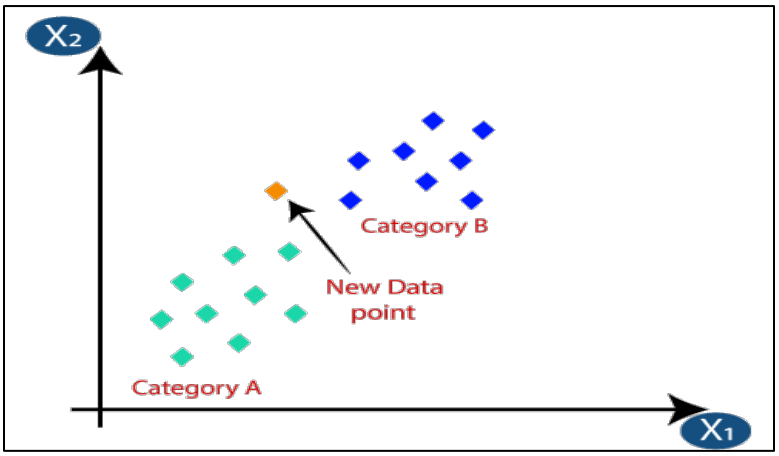


How does K-NN work?

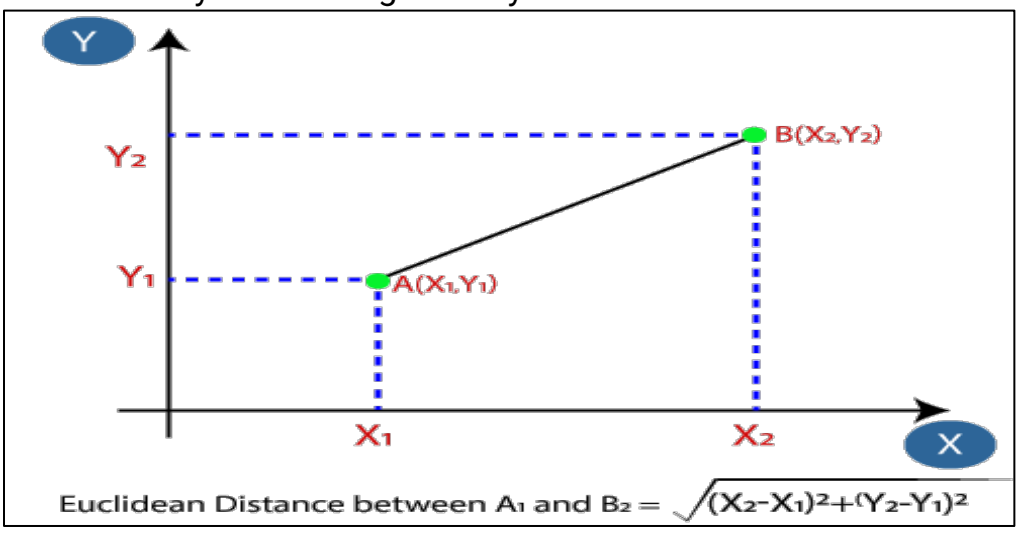
The K-NN working can be explained on the basis of the below algorithm:

- **Step-1:** Select the number K of the neighbors
- **Step-2:** Calculate the Euclidean distance of **K number of neighbors**
- **Step-3:** Take the K nearest neighbors as per the calculated Euclidean 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.
- **Step-6:** Our model is ready.

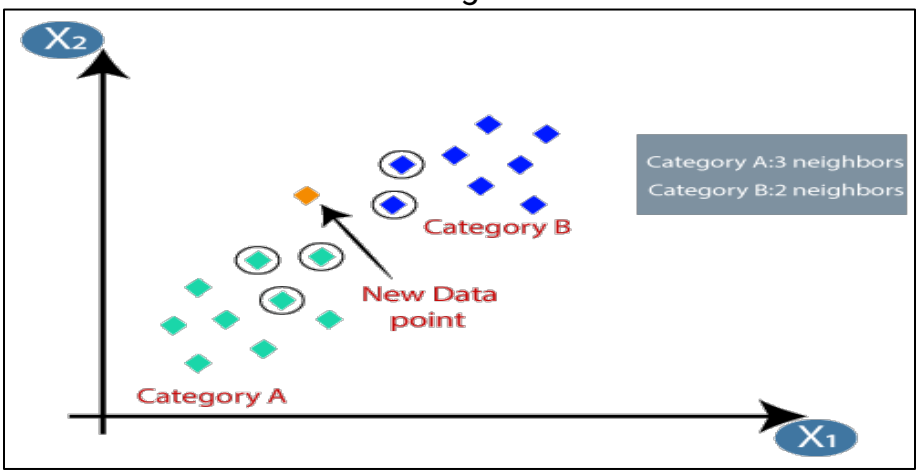
Suppose we have a new data point and we need to put it in the required category. Consider the below image:



- Firstly, we will choose the number of neighbors, so we will choose the k=5.
- Next, we will calculate the **Euclidean distance** between the data points. The Euclidean distance is the distance between two points, which we have already studied in geometry. It can be calculated as:



- By calculating the Euclidean distance we got the nearest neighbors, as three nearest neighbors in category A and two nearest neighbors in category B. Consider the below image:



- As we can see the 3 nearest neighbors are from category A, hence this new data point must belong to category A.

How to select the value of K in the K-NN Algorithm?

Below are some points to remember while selecting the value of K in the K-NN algorithm:

- There is no particular way to determine the best value for "K", so we need to try some values to find the best out of them. The most preferred value for K is 5.
- A very low value for K such as K=1 or K=2, can be noisy and lead to the effects of outliers in the model.
- Large values for K are good, but it may find some difficulties.

Advantages of KNN Algorithm:

- It is simple to implement.
- It is robust to the noisy training data
- It can be more effective if the training data is large.

Disadvantages of KNN Algorithm:

- Always needs to determine the value of K which may be complex some time.
- The computation cost is high because of calculating the distance between the data points for all the training samples.

Python implementation of the KNN algorithm

To do the Python implementation of the K-NN algorithm, we will use the same problem and dataset which we have used in Logistic Regression. But here we will improve the performance of the model. Below is the problem description:

Problem for K-NN Algorithm: There is a Car manufacturer company that has manufactured a new SUV car. The company wants to give the ads to the users who are interested in buying that SUV. So for this problem, we have a dataset that contains multiple user's information through the social network. The dataset contains lots of information but the **Estimated Salary** and **Age** we will consider for the independent variable and the **Purchased variable** is for the dependent variable. Below is the dataset:

User ID	Gender	Age	EstimatedSalary	Purchased
15624510	Male	19	19000	0
15810944	Male	35	20000	0
15668575	Female	26	43000	0
15603246	Female	27	57000	0
15804002	Male	19	76000	0
15728773	Male	27	58000	0
15598044	Female	27	84000	0
15694829	Female	32	150000	1
15600575	Male	25	33000	0
15727311	Female	35	65000	0
15570769	Female	26	80000	0
15606274	Female	26	52000	0
15746139	Male	20	86000	0
15704987	Male	32	18000	0
15628972	Male	18	82000	0
15697686	Male	29	80000	0
15733883	Male	47	25000	1
15617482	Male	45	26000	1
15704583	Male	46	28000	1
15621083	Female	48	29000	1
15649487	Male	45	22000	1
15736760	Female	47	49000	1

Steps to implement the K-NN algorithm:

- Data Pre-processing step
- Fitting the K-NN algorithm to the Training set
- Predicting the test result
- Test accuracy of the result(Creation of Confusion matrix)
- Visualizing the test set result.

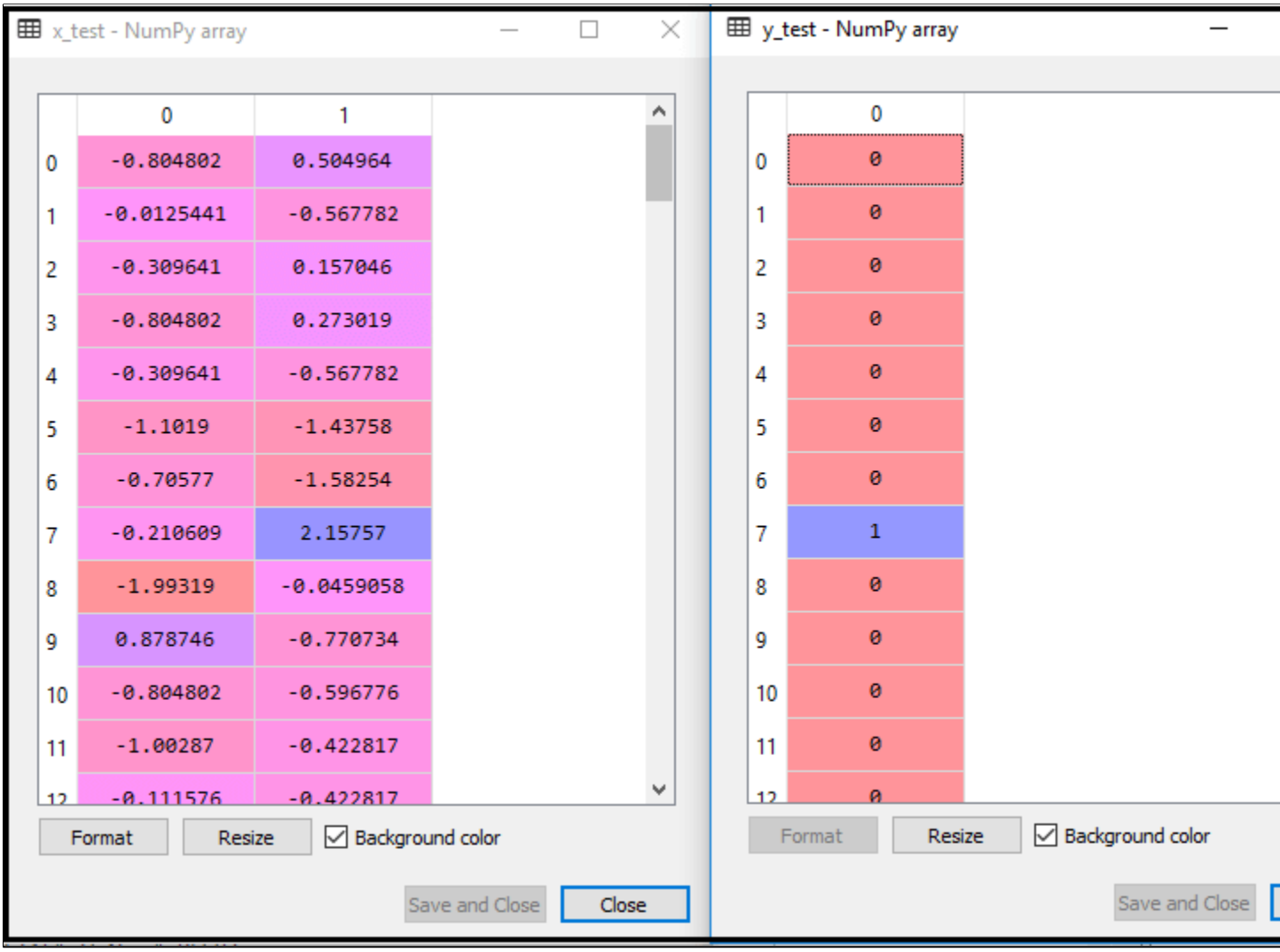
Data Pre-Processing Step:

The Data Pre-processing step will remain exactly the same as Logistic Regression. Below is the code for it:

```
# importing libraries
import numpy as nm
import matplotlib.pyplot as mtp
import pandas as pd

#importing datasets
data_set= pd.read_csv('user_data.csv')

#Extracting Independent and dependent Variable
x= data_set.iloc[:, [2,3]].values
y= data_set.iloc[:, 4].values
```

```
13.# Splitting the dataset into training and test set.
14.from sklearn.model_selection import train_test_split
15.x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= 0.25, random_state=0)

16.
17.#feature Scaling
18.from sklearn.preprocessing import StandardScaler
19.st_x= StandardScaler()
20.x_train= st_x.fit_transform(x_train)
21.x_test= st_x.transform(x_test)
```

By executing the above code, our dataset is imported to our program and well pre-processed. After feature scaling our test dataset will look like:



From the above output image, we can see that our data is successfully scaled.

- **Fitting K-NN classifier to the Training data:**
  Now we will fit the K-NN classifier to the training data. To do this we will import the **KNeighborsClassifier** class of **Sklearn Neighbors** library. After importing the class, we will create the **Classifier** object of the class. The Parameter of this class will be
  - **n_neighbors:** To define the required neighbors of the algorithm. Usually,

it takes 5.

- **metric='minkowski':** This is the default parameter and it decides the distance between the points.
- **p=2:** It is equivalent to the standard Euclidean metric.

And then we will fit the classifier to the training data. Below is the code for it:

```
#Fitting K-NN classifier to the training set
from sklearn.neighbors import KNeighborsClassifier
classifier= KNeighborsClassifier(n_neighbors=5, metric='minkowski', p=2 )
classifier.fit(x_train, y_train)
```

Output: By executing the above code, we will get the output as:

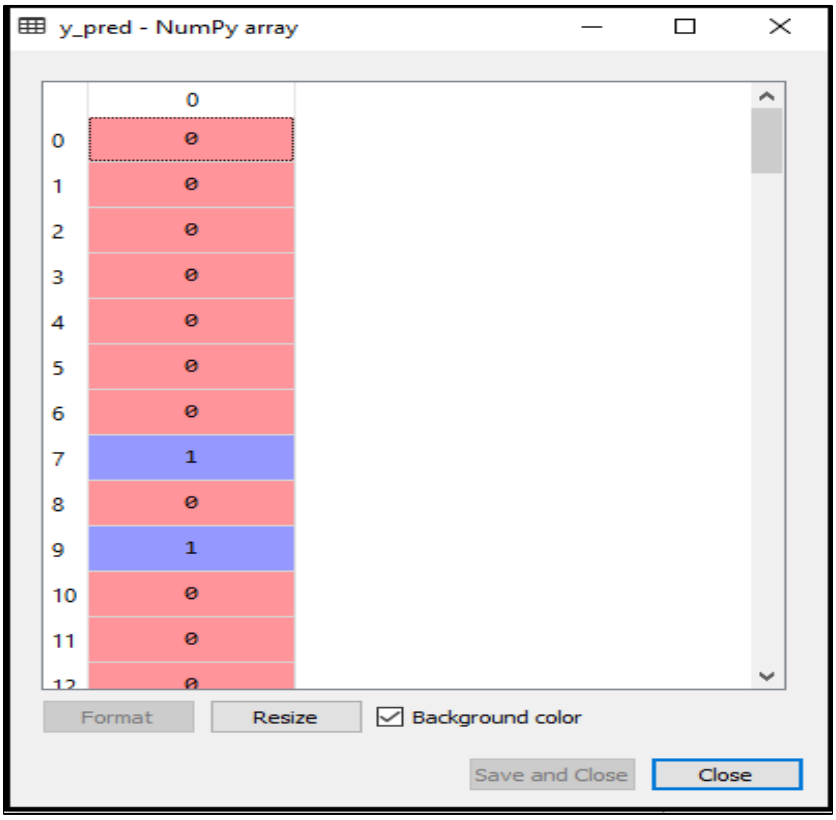
```
Out[10]:
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

- **Predicting the Test Result:** To predict the test set result, we will create a **y_pred** vector as we did in Logistic Regression. Below is the code for it:

```
#Predicting the test set result
y_pred= classifier.predict(x_test)
```

Output:

The output for the above code will be:



- **Creating the Confusion Matrix:**

  Now we will create the Confusion Matrix for our K-NN model to see the accuracy of the classifier. Below is the code for it:

```
#Creating the Confusion matrix
from sklearn.metrics import confusion_matrix
```

3. cm= confusion_matrix(y_test, y_pred)

In above code, we have imported the confusion_matrix function and called it using the variable cm.

Output: By executing the above code, we will get the matrix as below:

In the above image, we can see there are 64+29= 93 correct predictions and 3+4= 7 incorrect predictions, whereas, in Logistic Regression, there were 11 incorrect predictions. So we can say that the performance of the model is improved by using the K-NN algorithm.

- **Visualizing the Training set result:**

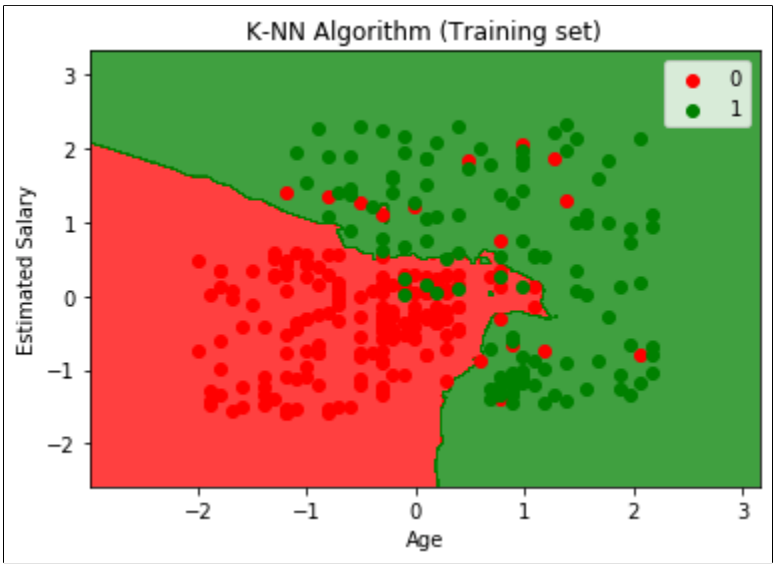
  Now, we will visualize the training set result for K-NN model. The code will remain same as we did in Logistic Regression, except the name of the graph. Below is the code for it:

```
1. #Visulaizing the trianing set result
2. from matplotlib.colors import ListedColormap
3. x_set, y_set = x_train, y_train
4. x1, x2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() -
    1, stop = x_set[:, 0].max() + 1, step  =0.01),
5. nm.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))
6. mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x
   1.shape),
7. alpha = 0.75, cmap = ListedColormap(('red','green' )))
8. mtp.xlim(x1.min(), x1.max())
9. mtp.ylim(x2.min(), x2.max())
10.for i, j in enumerate(nm.unique(y_set)):
11.    mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
12.        c = ListedColormap(('red', 'green'))(i), label = j)
13.mtp.title('K-NN Algorithm (Training set)')
14.mtp.xlabel('Age')
```

```
15.mtp.ylabel('Estimated Salary')
16.mtp.legend()
17.mtp.show()
```

Output:

By executing the above code, we will get the below graph:



The output graph is different from the graph which we have occurred in Logistic Regression. It can be understood in the below points:

- As we can see the graph is showing the red point and green points. The green points are for Purchased(1) and Red Points for not Purchased(0) variable.
- The graph is showing an irregular boundary instead of showing any straight line or any curve because it is a K-NN algorithm, i.e., finding the nearest neighbor.
- The graph has classified users in the correct categories as most of the users who didn't buy the SUV are in the red region and users who bought the SUV are in the green region.
- The graph is showing good result but still, there are some green points in the red region and red points in the green region. But this is no big issue as by doing this model is prevented from overfitting issues.
- Hence our model is well trained.

- **Visualizing the Test set result:**

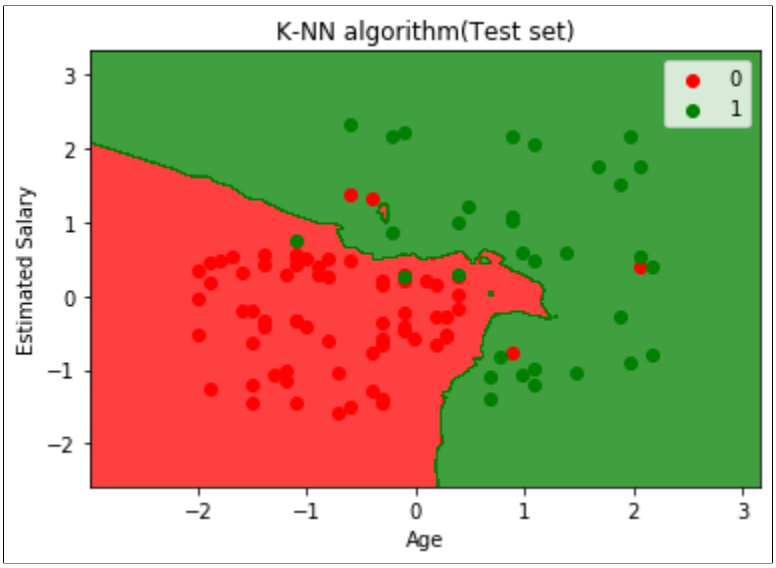
  After the training of the model, we will now test the result by putting a new dataset, i.e., Test dataset. Code remains the same except some minor changes: such as **x_train and y_train** will be replaced by **x_test and y_test**.

  Below is the code for it:

```
1. #Visualizing the test set result
2. from matplotlib.colors import ListedColormap
3. x_set, y_set = x_test, y_test
4. x1, x2 = nm.meshgrid(nm.arange(start = x_set[:, 0].min() -
   1, stop = x_set[:, 0].max() + 1, step  =0.01),
5. nm.arange(start = x_set[:, 1].min() - 1, stop = x_set[:, 1].max() + 1, step = 0.01))
```

```
mtp.contourf(x1, x2, classifier.predict(nm.array([x1.ravel(), x2.ravel()]).T).reshape(x
1.shape),
alpha = 0.75, cmap = ListedColormap(('red','green' )))
mtp.xlim(x1.min(), x1.max())
mtp.ylim(x2.min(), x2.max())
for i, j in enumerate(nm.unique(y_set)):
   mtp.scatter(x_set[y_set == j, 0], x_set[y_set == j, 1],
       c = ListedColormap(('red', 'green'))(i), label = j)
mtp.title('K-NN algorithm(Test set)')
mtp.xlabel('Age')
mtp.ylabel('Estimated Salary')
mtp.legend()
mtp.show()
```

Output:



The above graph is showing the output for the test data set. As we can see in the graph, the predicted output is well good as most of the red points are in the red region and most of the green points are in the green region.
However, there are few green points in the red region and a few red points in the green region. So these are the incorrect observations that we have observed in the confusion matrix(7 Incorrect output).

Decision Tree?

It is a tool that has applications spanning several different areas. Decision trees can be used for classification as well as regression problems. The name itself suggests that it uses a flowchart like a tree structure to show the predictions that result from a series of feature-based splits. It starts with a root node and ends with a decision made by leaves.

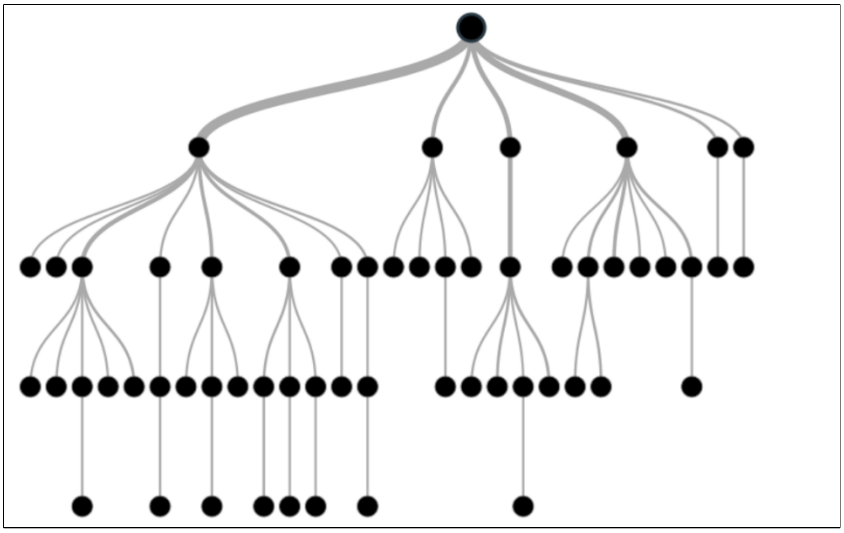


Image 1

Before learning more about decision trees let's get familiar with some of the terminologies.

Root Nodes – It is the node present at the beginning of a decision tree from this node the population starts dividing according to various features.
Decision Nodes – the nodes we get after splitting the root nodes are called Decision Node
Leaf Nodes – the nodes where further splitting is not possible are called leaf nodes or terminal nodes
Sub-tree – just like a small portion of a graph is called sub-graph similarly a sub-section of this decision tree is called sub-tree.
Pruning – is nothing but cutting down some nodes to stop overfitting.

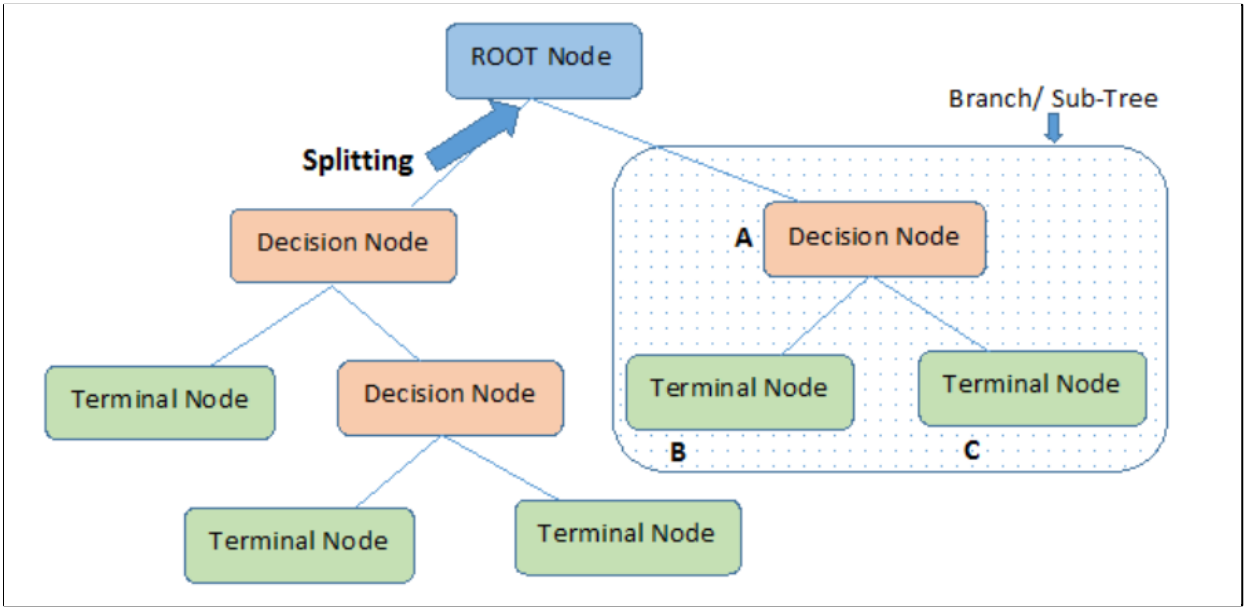


Image 2

Example of a decision tree

Let’s understand decision trees with the help of an example.

Day	Weather	Temperature	Humidity	Wind	Play?
1	Sunny	Hot	High	Weak	No
2	Cloudy	Hot	High	Weak	Yes
3	Sunny	Mild	Normal	Strong	Yes
4	Cloudy	Mild	High	Strong	Yes
5	Rainy	Mild	High	Strong	No
6	Rainy	Cool	Normal	Strong	No
7	Rainy	Mild	High	Weak	Yes
8	Sunny	Hot	High	Strong	No
9	Cloudy	Hot	Normal	Weak	Yes
10	Rainy	Mild	High	Strong	No

Image 3
Decision trees are upside down which means the root is at the top and then this root is split into various several nodes. Decision trees are nothing but a bunch of if-else statements in layman terms. It checks if the condition is true and if it is then it goes to the next node attached to that decision.

In the below diagram the tree will first ask what is the weather? Is it sunny, cloudy, or rainy? If yes then it will go to the next feature which is humidity and wind. It will again check if there is a strong wind or weak, if it’s a weak wind and it’s rainy then the person may go and play.

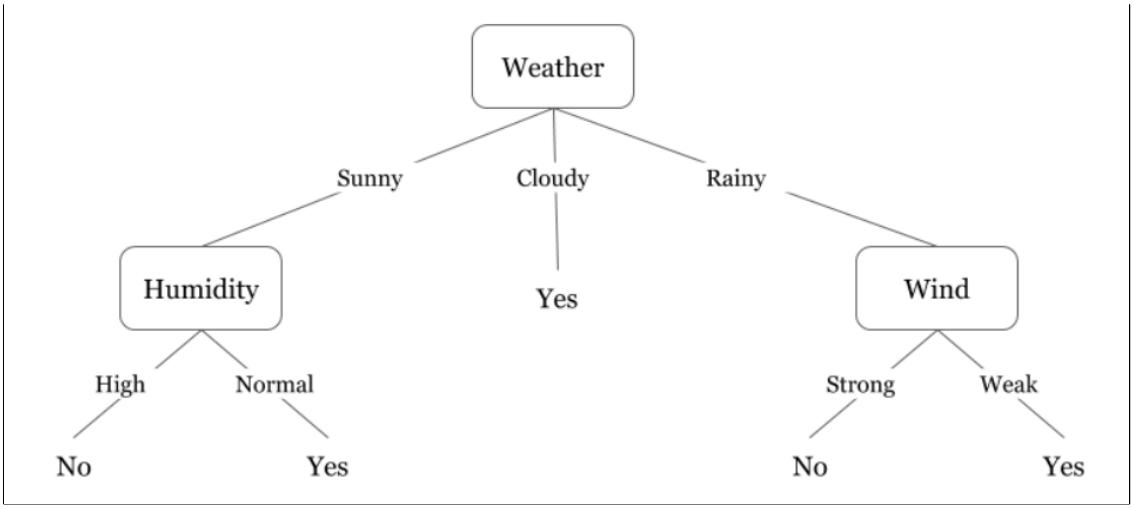


Image 4

Did you notice anything in the above flowchart? We see that if the *weather is cloudy* then we must go to play. Why didn't it split more? Why did it stop there?

To answer this question, we need to know about few more concepts like entropy, information gain, and Gini index. But in simple terms, I can say here that the output for the training dataset is always yes for cloudy weather, since there is no disorderliness here we don't need to split the node further.

The goal of machine learning is to decrease uncertainty or disorders from the dataset and for this, we use decision trees.

Now you must be thinking how do I know what should be the root node? what should be the decision node? when should I stop splitting? To decide this, there is a metric called "Entropy" which is the amount of uncertainty in the dataset.

Entropy

Entropy is nothing but the uncertainty in our dataset or measure of disorder. Let me try to explain this with the help of an example.

Suppose you have a group of friends who decides which movie they can watch together on Sunday. There are 2 choices for movies, one is ***"Lucy"*** and the second is ***"Titanic"*** and now everyone has to tell their choice. After everyone gives their answer we see that *"Lucy" gets 4 votes* and *"Titanic" gets 5 votes*. Which movie do we watch now? Isn't it hard to choose 1 movie now because the votes for both the movies are somewhat equal.

This is exactly what we call disorderness, there is an equal number of votes for both the movies, and we can't really decide which movie we should watch. It would have been much easier if the votes for "Lucy" were 8 and for "Titanic" it was 2. Here we could easily say that the majority of votes are for "Lucy" hence everyone will be watching this movie.

In a decision tree, the output is mostly "yes" or "no"

The formula for Entropy is shown below:

$$E(S) = -p_{(+)}\log p_{(+)} - p_{(-)}\log p_{(-)}$$

Here p_+ is the probability of positive class
p_- is the probability of negative class
S is the subset of the training example

How do Decision Trees use Entropy?

Now we know what entropy is and what is its formula, Next, we need to know that how exactly does it work in this algorithm.
Entropy basically measures the impurity of a node. Impurity is the degree of randomness; it tells how random our data is. A **pure sub-split** means that either you should be getting "yes", or you should be getting "no".
Suppose a *feature* has 8 "yes" and 4 "no" initially, after the first split the left node *gets 5 'yes' and 2 'no'* whereas right node *gets 3 'yes' and 2 'no'.*
We see here the split is not pure, why? Because we can still see some negative classes in both the nodes. In order to make a decision tree, we need to calculate the impurity of each split, and when the purity is 100%, we make it as a leaf node.
To check the impurity of feature 2 and feature 3 we will take the help for Entropy formula.

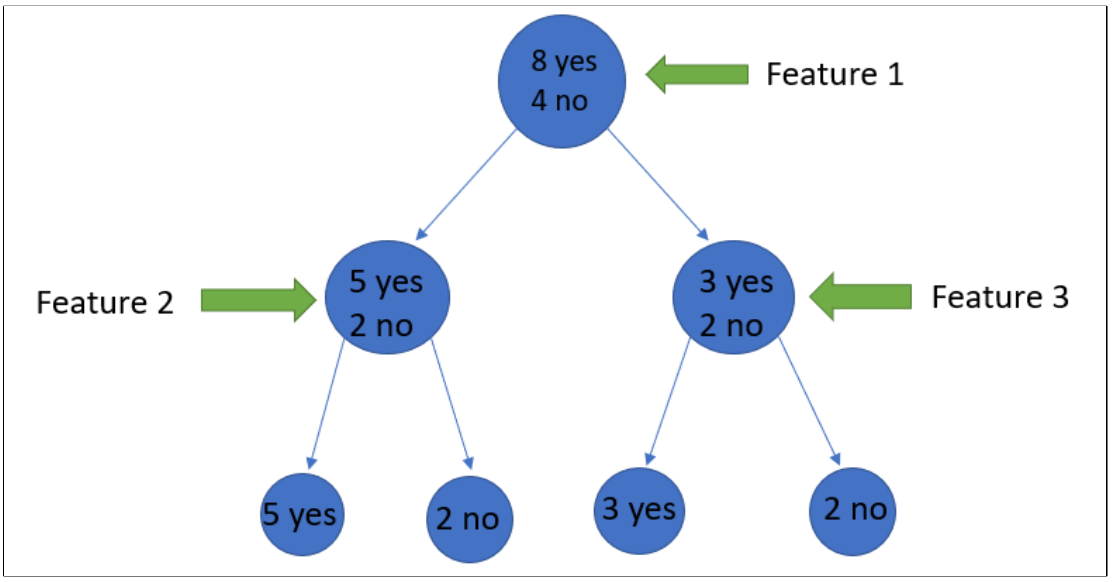


Image Source: Author

$$\Rightarrow -\left(\frac{5}{7}\right)log_2\left(\frac{5}{7}\right) - \left(\frac{2}{7}\right)log_2\left(\frac{2}{7}\right)$$

$$\Rightarrow -(0.71 * -0.49) - (0.28 * -1.83)$$

$$\Rightarrow -(-0.34) - (-0.51)$$

$$\Rightarrow 0.34 + 0.51$$

$$\Rightarrow 0.85$$

For feature 3,

$$\Rightarrow -\left(\frac{3}{5}\right)log_2\left(\frac{3}{5}\right) - \left(\frac{2}{5}\right)log_2\left(\frac{2}{5}\right)$$
$$\Rightarrow -(0.6 * -0.73) - (0.4 * -1.32)$$
$$\Rightarrow -(-0.438) - (-0.528)$$
$$\Rightarrow 0.438 + 0.528$$
$$\Rightarrow 0.966$$

We can clearly see from the tree itself that left node has low entropy or more purity than right node since left node has a greater number of “yes” and it is easy to decide here.
Always remember that the higher the Entropy, the lower will be the purity and the higher will be the impurity.
As mentioned earlier the goal of machine learning is to decrease the uncertainty or impurity in the dataset, here by using the entropy we are getting the impurity of a particular node, we don’t know if the parent entropy or the entropy of a particular node has decreased or not.
For this, we bring a new metric called “Information gain” which tells us how much the parent entropy has decreased after splitting it with some feature.

Information Gain

Information gain measures the reduction of uncertainty given some feature and it is also a deciding factor for which attribute should be selected as a decision node or root node.

$$Information\ Gain\ =\ E(Y)\ -\ E(Y|X)$$

It is just entropy of the full dataset – entropy of the dataset given some feature.
To understand this better let’s consider an example:
Suppose our entire population has a total of 30 instances. The dataset is to predict whether the person will go to the gym or not. Let’s say 16 people go to the gym and 14 people don’t
Now we have two features to predict whether he/she will go to the gym or not.
Feature 1 is **“Energy”** which takes two values *“high” and “low”*
Feature 2 is **“Motivation”** which takes 3 values *“No motivation”, “Neutral” and “Highly motivated”.*
Let’s see how our decision tree will be made using these 2 features. We’ll use information gain to decide which feature should be the root node and which feature should be placed after the split.

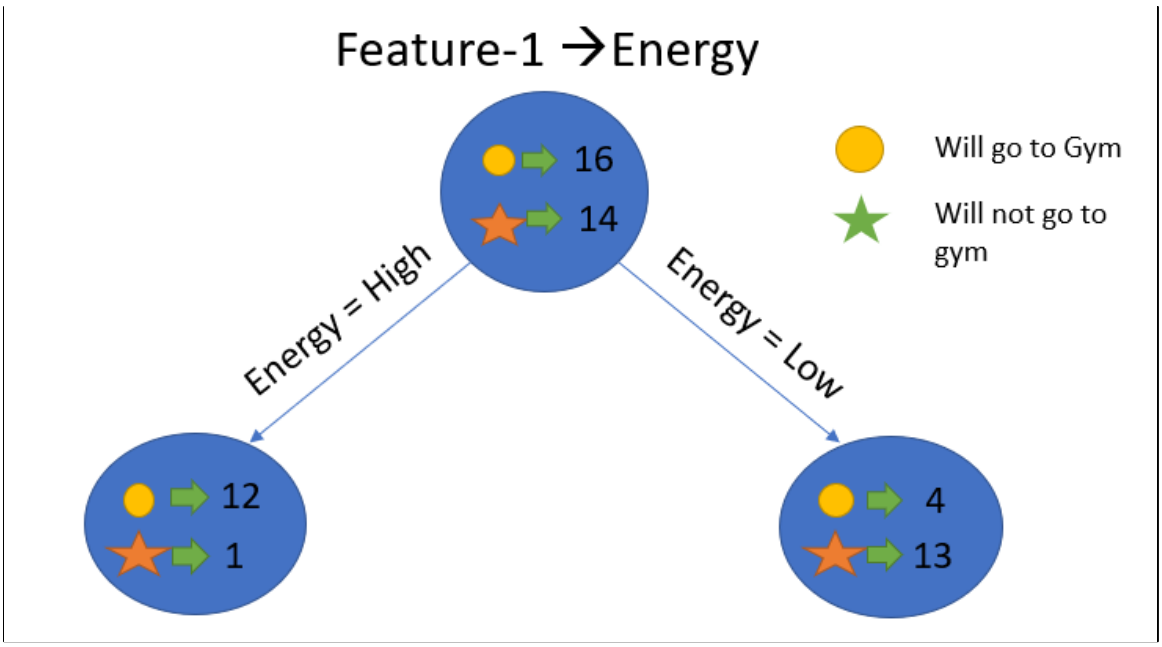


Image Source: Author

Let's calculate the entropy:

$$E(Parent) = -\left(\frac{16}{30}\right)\log_2\left(\frac{16}{30}\right) - \left(\frac{14}{30}\right)\log_2\left(\frac{14}{30}\right) \approx 0.99$$

$$E(Parent|Energy = "high") = -\left(\frac{12}{13}\right)\log_2\left(\frac{12}{13}\right) - \left(\frac{1}{13}\right)\log_2\left(\frac{1}{13}\right) \approx 0.39$$

$$E(Parent|Energy = "low") = -\left(\frac{4}{17}\right)\log_2\left(\frac{4}{17}\right) - \left(\frac{13}{17}\right)\log_2\left(\frac{13}{17}\right) \approx 0.79$$

To see the weighted average of entropy of each node we will do as follows:

$$E(Parent|Energy) = \frac{13}{30} * 0.39 + \frac{17}{30} * 0.79 = 0.62$$

Now we have the value of E(Parent) and E(Parent|Energy), information gain will be:

$$\begin{aligned} Information\ Gain &= E(parent) - E(parent|energy \\ &= 0.99 - 0.62 \\ &= 0.37 \end{aligned}$$

Our parent entropy was near 0.99 and after looking at this value of information gain, we can say that the entropy of the dataset will decrease by 0.37 if we make "Energy" as our root node.

Similarly, we will do this with the other feature “Motivation” and calculate its information gain.

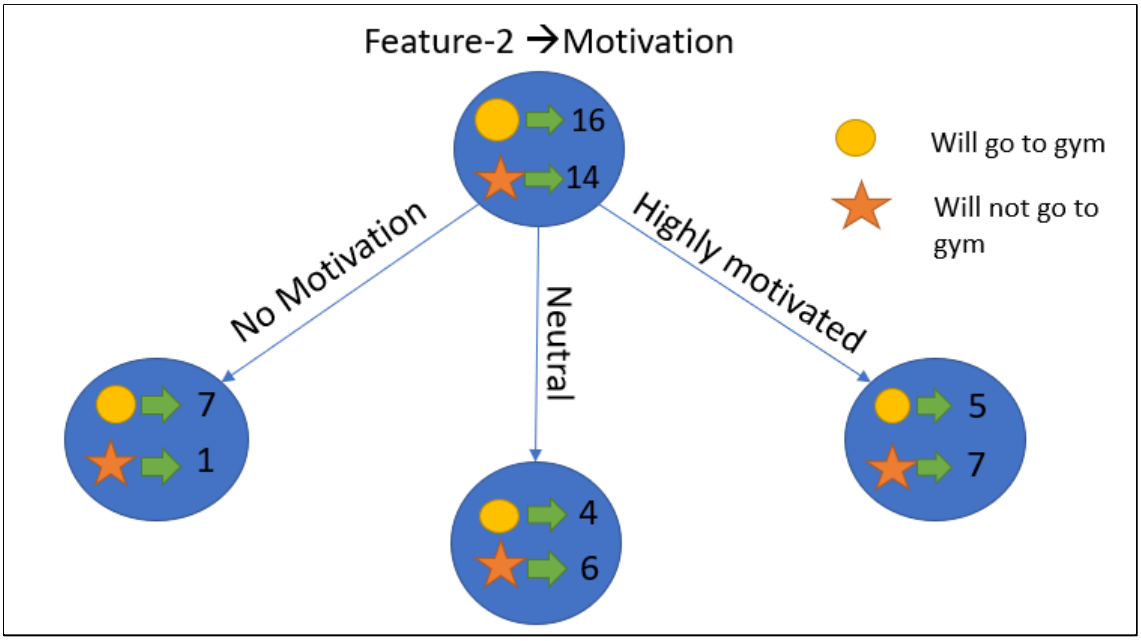


Image Source: Author
Let’s calculate the entropy here:

$$E(Parent) = 0.99$$

$$E\Big(Parent|Motivation = "No\ motivation"\Big) = -\left(\frac{7}{8}\right)log_2\left(\frac{7}{8}\right) - \frac{1}{8}\log_2\left(\frac{1}{8}\right) = 0.54$$

$$E\big(Parent|Motivation = "Neutral"\big) = -\left(\frac{4}{10}\right)log_2\left(\frac{4}{10}\right) - (\frac{6}{10})\log_2\left(\frac{6}{10}\right) = 0.97$$

$$E\big(Parent|Motivation = "Highly\ motivated"\big) = -\left(\frac{5}{12}\right)log_2\left(\frac{5}{12}\right) - (\frac{7}{12})\log_2\left(\frac{7}{12}\right) = 0.98$$

To see the weighted average of entropy of each node we will do as follows:

$$E(Parent|Motivation) = \frac{8}{30} * 0.54 + \frac{10}{30} * 0.97 + \frac{12}{30} * 0.98 = 0.86$$

Now we have the value of E(Parent) and E(Parent|Motivation), information gain will be:

$$\begin{aligned} Information\ Gain &= E(Parent) - E(Parent|Motivation) \\ &= 0.99 - 0.86 \\ &= 0.13 \end{aligned}$$

We now see that the “Energy” feature gives more reduction which is 0.37 than the “Motivation” feature. Hence we will select the feature which has the highest information gain and then split the node based on that feature.

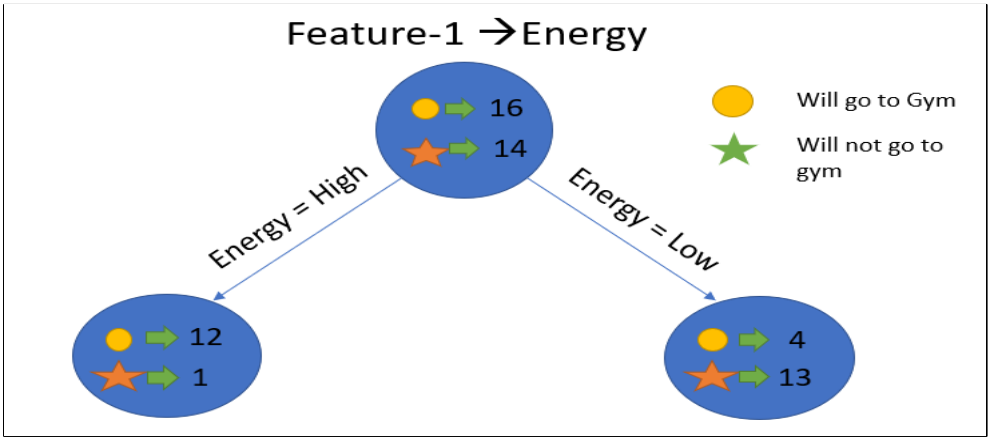


Image Source: Author

In this example “Energy” will be our root node and we’ll do the same for sub-nodes. Here we can see that when the energy is “high” the entropy is low and hence we can say a person will definitely go to the gym if he has high energy, but what if the energy is low? We will again split the node based on the new feature which is “Motivation”.

When to stop splitting?

You must be asking this question to yourself that when do we stop growing our tree? Usually, real-world datasets have a large number of features, which will result in a large number of splits, which in turn gives a huge tree. Such trees take time to build and can lead to overfitting. That means the tree will give very good accuracy on the training dataset but will give bad accuracy in test data.
There are many ways to tackle this problem through hyperparameter tuning. We can set the maximum depth of our decision tree using the ***max_depth*** parameter. The more the value of ***max_depth***, the more complex your tree will be. The training error will off-course decrease if we increase the ***max_depth*** value but when our test data comes into the picture, we will get a very bad accuracy. Hence you need a value that will not overfit as well as underfit our data and for this, you can use GridSearchCV.
Another way is to set the minimum number of samples for each spilt. It is denoted by ***min_samples_split***. Here we specify the minimum number of samples required to do a spilt. For example, we can use a minimum of 10 samples to reach a decision. That means if a node has less than 10 samples then using this parameter, we can stop the further splitting of this node and make it a leaf node.
There are more hyperparameters such as :
min_samples_leaf – represents the minimum number of samples required to be in the leaf node. The more you increase the number, the more is the possibility of overfitting.
max_features – it helps us decide what number of features to consider when looking for the best split.
To read more about these hyperparameters you can read it here.

Pruning

It is another method that can help us avoid overfitting. It helps in improving the performance of the tree by cutting the nodes or sub-nodes which are not significant. It removes the branches which have very low importance.
There are mainly 2 ways for pruning:
(i) **Pre-pruning** – we can stop growing the tree earlier, which means we can prune/remove/cut a node if it has low importance **while growing** the tree.

(ii) **Post-pruning** – once our **tree is built to its depth**, we can start pruning the nodes based on their significance.

Greedy Algorithm

Greedy algorithm or search is an efficient tool that is usually applied in optimization problems. The main steps of all greedy algorithms are as follows:
1) Choice of a candidate set. The problem is divided into a finite set of sub-problems. At initial step a candidate set is arbitrarily chosen as a solution to the first sub-problem, so that the proceeding sub-problems are solved on the basis of this candidate set.
2) Choice of a selection function. A selection function is chosen to test what is the best candidate to be added to the solution.
3) Choice of a feasibility function. The chosen selection function involves a feasibility function, which first determines the set of candidates that can possibly contribute to the solution.
4) Choice of an objective function. This function allows to make choice of a candidate at each step in some sense optimal.
5) Choice of a solution function. Finally, a solution function is chosen appropriately to establish when a desired precision in solution approximation is achieved.
Greedy algorithms can be really efficient when dealing with large sets of data, in the sense that if the globally optimal solution of the problem consists of optimal solutions of locally optimal sub-problems, then greedy searches will find the global solution in reasonable time. Nevertheless, in some specific problems, such as the famous traveling salesman problem, greedy algorithm may result in unique worst possible solution [37] . This failure comes from the fact that the greedy algorithm does not use all the data of the problem (recall step 1).
Note that the successful application of the algorithm, first of all, depends on success at every partial step, since any globally optimal solution consists of optimal sub-solutions. Moreover, since the selection function is different from identity, then each candidate can be involved in only one step of the algorithm, i.e. neither of previous steps (except directly previous one) plays a role in proceeding computations/searches. This is called greedy choice property and helps to save quite much computational cost.

Integration of Greedy Algorithm into Neural Model of User Behavior Prediction

Problem solving using neural computations in principle is a systematic search through given data in order to reach required solution. Therefore, from the description of the greedy algorithm above it follows that it can be efficiently

involved in neural computing procedure to save computational time, especially when the analyzed data sets are quite large. However, using of greedy algorithm may not be a guarantee for optimality of found solution.

Let us demonstrate how a simple greedy algorithm can be involved in prediction of user behavior based on his/her clicks, search history, time spent on viewing certain items etc., within a specific online shop. Analysis of user behavior and prediction of his/her intentions are crucial for online sellers in order to e.g. make advertisement more targeted, recommend items that will be more likely bought, etc. Several approaches exist to involve artificial intelligence in real time prediction of user intentions including recent studies [38] [39] [40] and some references therein.

We are mainly interested in interactions between guaranteed purchase and

1) view of a product page,
2) view of the basket.

The set of training data (DD) is composed of

1) all sessions (SS) of all users,
2) all items (IsIs) that were displayed within session s∈Ss∈S ,
3) all purchases (PsPs) corresponding to session s∈Ss∈S .

Thus, D=S∪Is∪PsD=S∪Is∪Ps . The set of sessions is split into two subsets: SpSp containing sessions with purchase and SnpSnp containing those without purchase (see Figure 2). Eventually, D=Sp∪Snp∪Is∪Ps.D=Sp∪Snp∪Is∪Ps.

The problem is to predict the set PsPs .

In this specific example, there exist 2.5·1042.5·104 products with full description. Evidently, the data are very sparse and high dimensional (depend on many parameters), therefore their analysis is too costly. To explore any pattern between

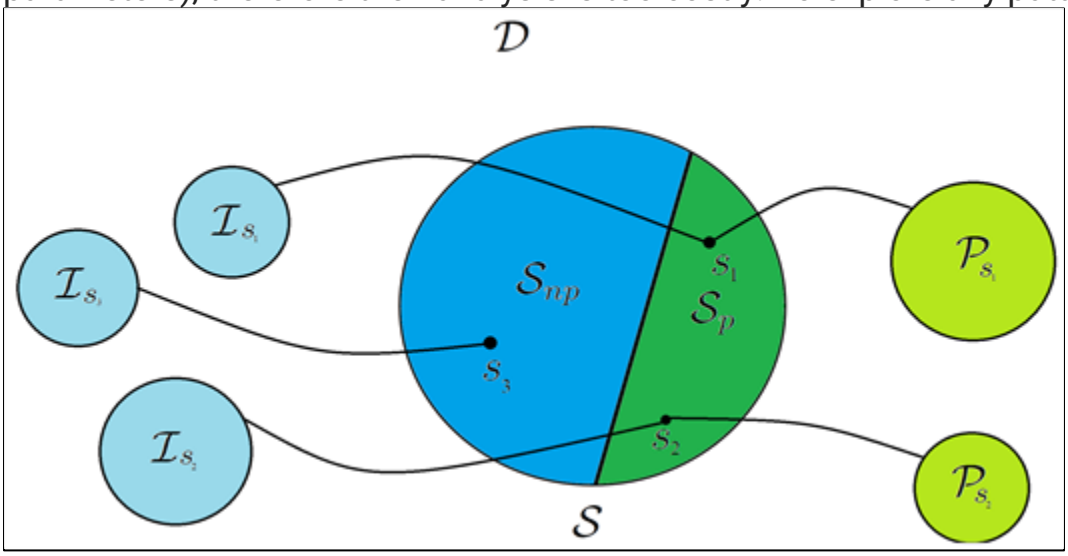


Figure 2. Schematic representation of the data set DD .

these data, we involve the neural network borrowed from [40] with greedy search integrated into the data analysis step in order to reduce the dimensionality. More specifically, at the initial step a candidate set is chosen heuristically. Then, a selection function

σ(s)=χD\(Sp∪Snp∪Is)(s),σ(s)=χD\(Sp∪Snp∪Is)(s),

where χDχD is the characteristic function of DD defined as follows:

χD(s)=(1;s∈D,0;else.χD(s)=(1;s∈D,0;else.

Thus, the selection function checks whether a chosen element ss belongs to D\(Sp∪Snp∪Is)D\(Sp∪Snp∪Is) or not. If it does, then it is out of consideration, otherwise it is a potential candidate for PsPs to be complemented by. Then, the objective function, chosen as the following convex functional (quadratic error function):

$$\kappa[s]=\frac{1}{n}\sum_{i=1}^{n}(s-s_i)^2,\kappa[s]=\frac{1}{n}\sum_{i=1}^{n}(s-s_i)^2,$$

where ii is the iteration index, nn is the amount of data points, sisi is the currently chosen element; allows to choose only the data which are (in the sense of quadratic error) close to the desired set.

The last step of the greedy algorithm defines the solution function as follows:

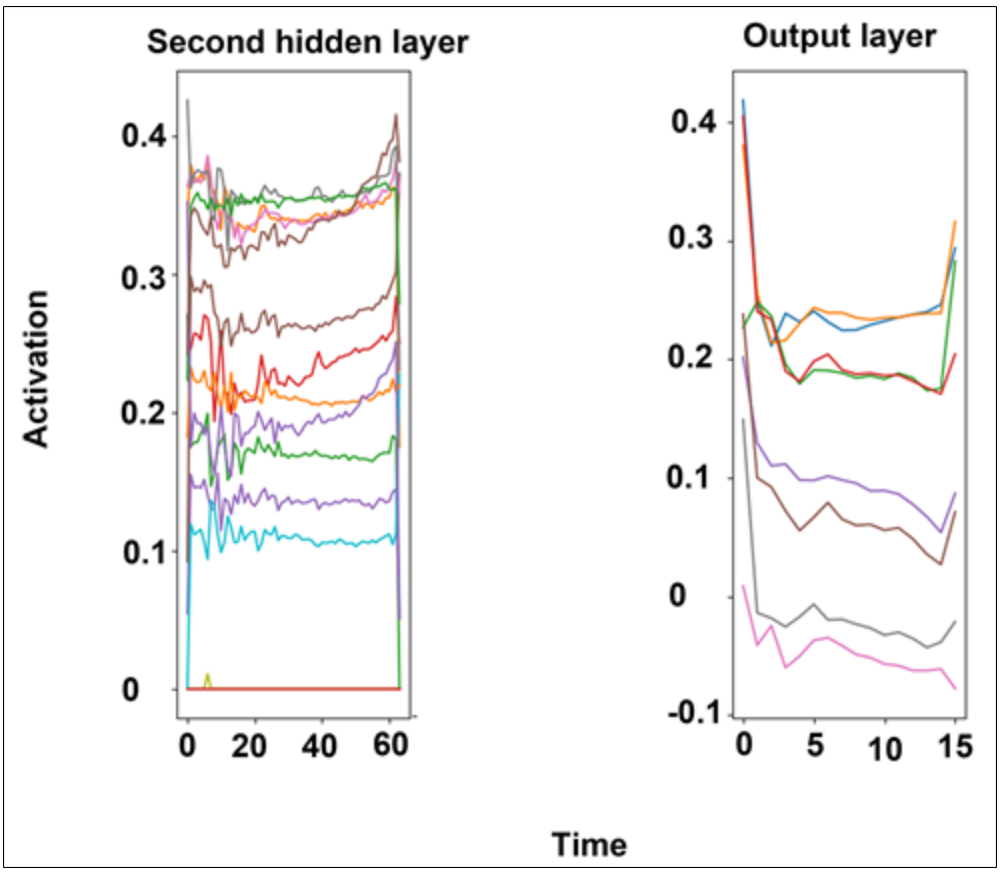
$$S(s)=\operatorname{argmin}\kappa[s]=\operatorname{argmin}\left[\frac{1}{n}\sum_{i=1}^{n}(s-s_i)^2\right].S(s)=\operatorname{argmin}\kappa[s]=\operatorname{argmin}\left[\frac{1}{n}\sum_{i=1}^{n}(s-s_i)^2\right].$$

Apparently, for the above chosen simple objective function, the solution function, SS , is easy to compute. However, more complicated forms of κκ can be considered.

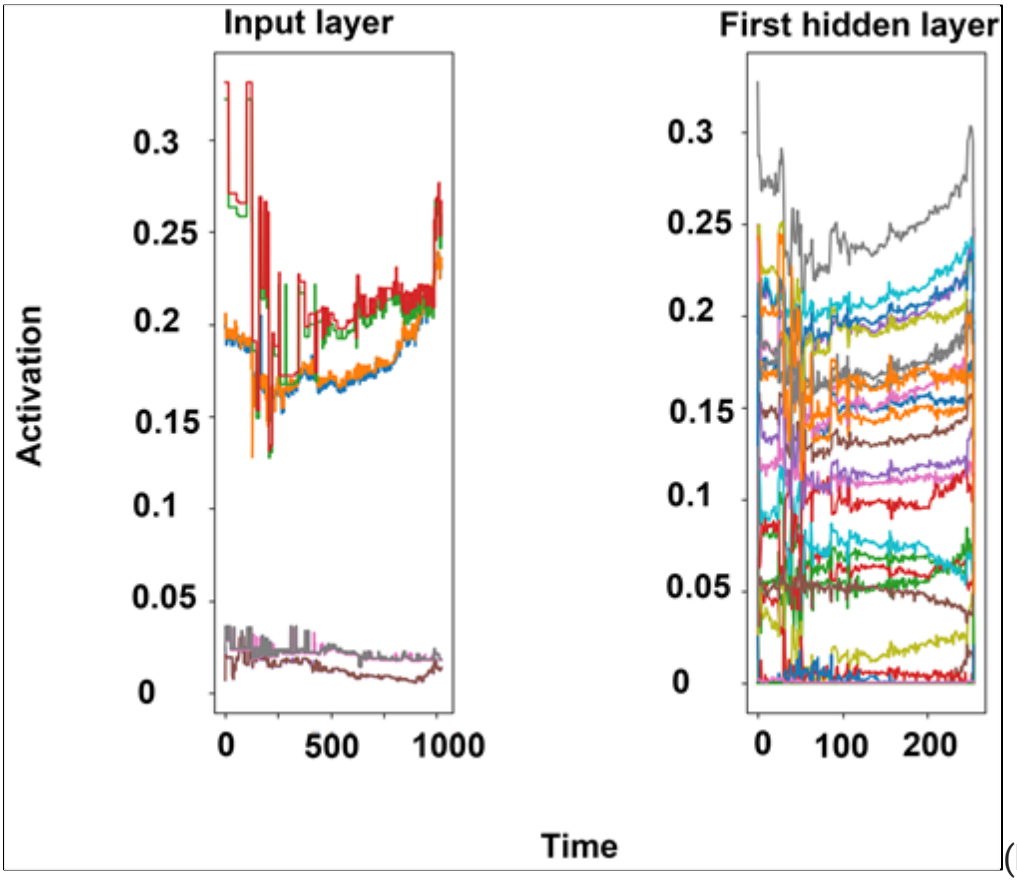
The mathematical background of the minimization problem is the same as in [40] , therefore we will not bring it here to be concise. Refer to [40] for further details.

3. Numerics and Discussions

Numerical experiment shows that in the chosen particular case involvement of greedy algorithm allows to reduce the computation time significantly, compared



(a)



(b)

Figure 3. Activation vs Time in neural network layers.

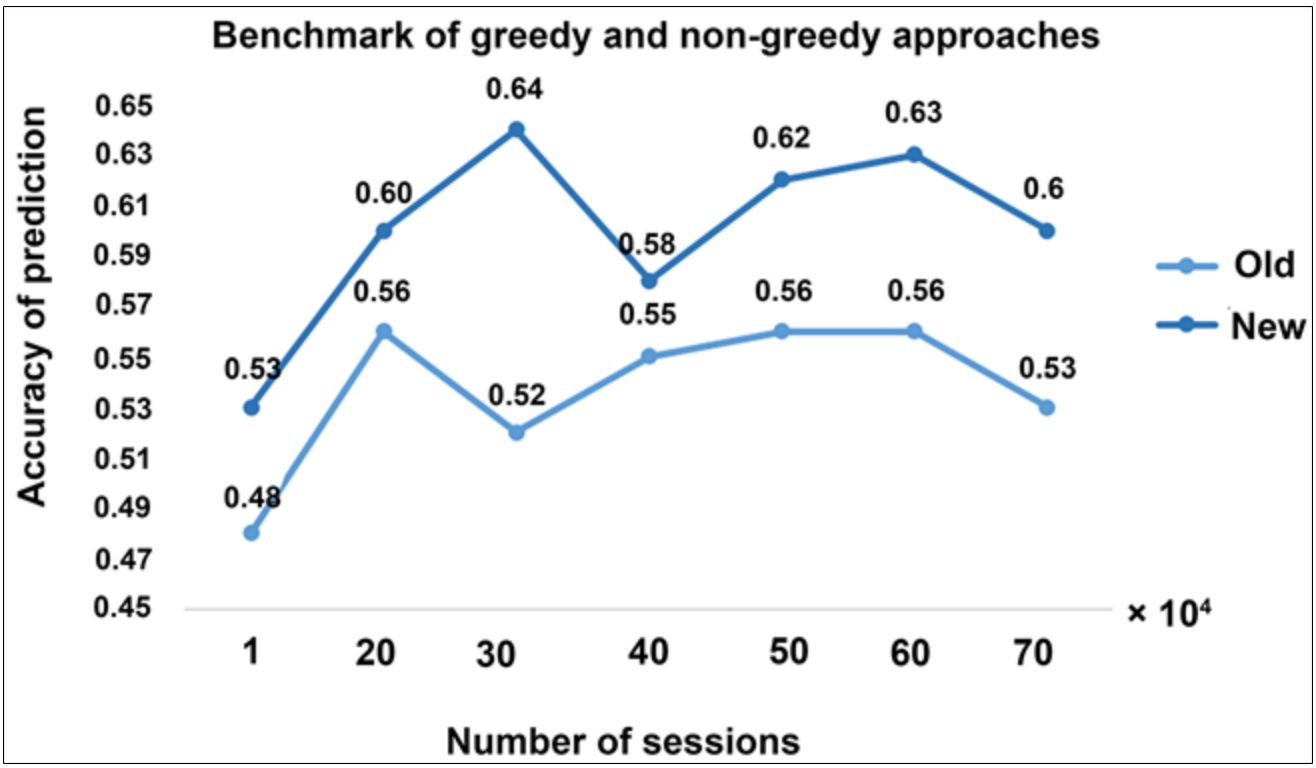


Figure 4. Benchmark of greedy and non-greedy approaches.

with [40] . Though, the relative error, i.e. the prediction error of the set PsPs of purchases within session s∈Sps∈Sp , is ~2% - 4%, which is pretty much acceptable. On Figure 3 the time spent on computations within input, first hidden layer, second hidden layer, and output. It is seen that for basic prediction of PsPs , only t≈1400t≈1400 s, which is a good improvement as well.

Figure 4 shows the accuracy of prediction implemented using the proposed approach against number of sessions. It shows a good tendency to be applicable in time consuming predictions.

Conclusion

Aiming to reduce the complexity of computations (machinery time) in the problem of user behavior analysis, prediction and decision support based on his/her online behavior, in this paper, we suggest to involve the well-known greedy algorithm. For heuristically chosen candidate set, we choose appropriate selection function, classifying the data at each iteration. Then, we choose a quadratic error function as objective function for classification. The output is used to construct the solution function. Numerical analysis of given data shows a significant decrease in computation time compared with the method without a greedy algorithm.

Random Forest:

Random Forest is one of the most popular bagging methods. What's bagging you ask? It's an abbreviation for **bootstrapping + aggregating**. The goal of bagging is to reduce the variance of a single estimator, i.e. the variance of a single Decision Tree in the case of Random Forest.

To be concrete, let's use a dummy dataset throughout this story. Suppose you have the following tax evasion dataset. Your task is to predict whether a person will

comply to pay taxes (the Evade column) based on features like Taxable Income (in thousand dollars), Marital Status, and whether a Refund is implemented or not.

Train data

Id	Refund	Marital Status	Taxable Income	Evade
1	Yes	Single	125	No
2	No	Married	100	No
3	No	Single	70	No
4	Yes	Married	120	No
5	No	Divorced	95	Yes
6	No	Married	60	No
7	Yes	Divorced	220	No
8	No	Single	85	Yes
9	No	Married	75	No
10	No	Single	90	Yes

Test data

Id	Refund	Marital Status	Taxable Income	Evade
11	Yes	Single	80	?

Dataset used in this story | Image by author

Random Forest consists of these three steps:

Step 1. Create a bootstrapped data

Given an original train data with *m* observations and *n* features, sample *m* observations randomly **with repetition**. The keyword here is "with repetition". This means it's most likely that some observations are picked more than once.
Below is an example of bootstrapped data from the original train data above. You might have different bootstrapped data due to randomness.

Bootstrapped data

Id	Refund	Marital Status	Taxable Income	Evade
6	No	Married	60	No
1	Yes	Single	125	No
4	Yes	Married	120	No
4	Yes	Married	120	No
8	No	Single	85	Yes
10	No	Single	90	Yes
4	Yes	Married	120	No
6	No	Married	60	No
3	No	Single	70	No
5	No	Divorced	95	Yes

A bootstrapped data | Image by author

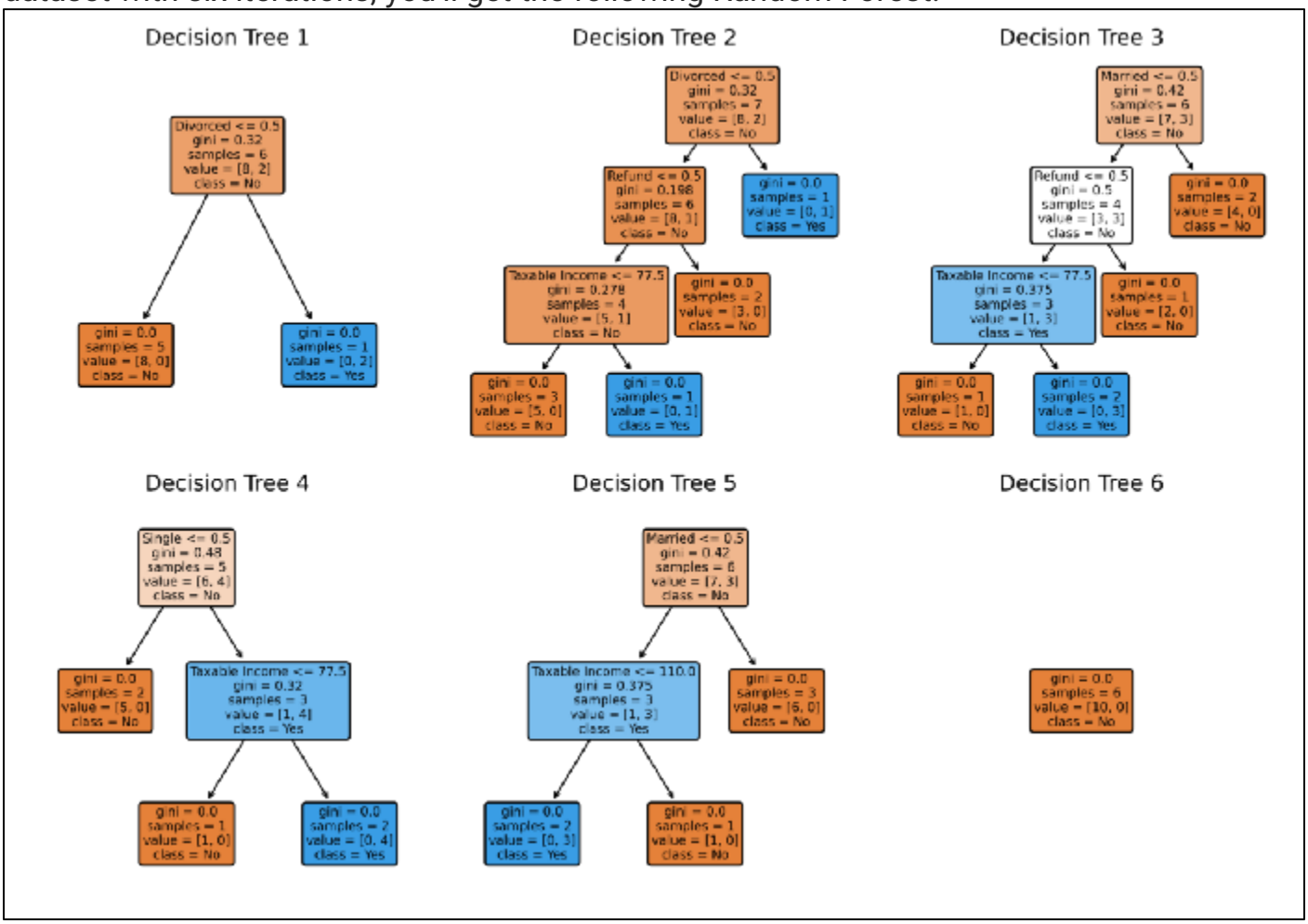
Step 2. Build a Decision Tree

The decision tree is built:

1. using the bootstrapped data, and
2. considering a random subset of features at each node (the number of features considered is usually the square root of *n*).

Step 3. Repeat

Step 1 and Step 2 are iterated many times. For example, using the tax evasion dataset with six iterations, you'll get the following Random Forest.



A Random Forest with six Decision Trees using the tax evasion dataset. Note that Decision Tree 6 consists of only one node since the bootstrapped data has Evade = No target for all ten observations | Image by author

Note that Random Forest is not deterministic since there's a random selection of observations and features at play. These random selections are why Random Forest reduces the variance of a Decision Tree.

Predict, aggregate, and evaluate

To make predictions using Random Forest, traverse each Decision Tree using the test data. For our example, there's only one test observation on which six Decision Trees respectively give predictions **No**, **No**, **No**, **Yes**, **Yes**, and **No** for the "Evade"

target. Do majority voting from these six predictions to make a single Random Forest prediction: **No**.
Combining predictions of Decision Trees into a single prediction of Random Forest is called **aggregating**. While majority voting is used for classification problems, the aggregating method for regression problems uses the mean or median of all Decision Tree predictions as a single Random Forest prediction.
The creation of bootstrapped data allows some train observations to be unseen by a subset of Decision Trees. These observations are called **out-of-bag samples** and are useful for evaluating the performance of Random Forest. To obtain validation accuracy (or any metric of your choice) of Random Forest:

1. run out-of-bag samples through all Decision Trees that were built *without* them,
2. aggregate the predictions, and
3. compare the aggregated predictions with the true labels.

AdaBoost:

Unlike Random Forest, AdaBoost (Adaptive Boosting) is a boosting ensemble method where *simple* Decision Trees are built sequentially. How simple? The trees consist of only a root and two leaves, so simple that they have their own name: **Decision Stumps**. Here are two main differences between Random Forest and AdaBoost:

1. Decision Trees in Random Forest have equal contributions to the final prediction, whereas Decision Stumps in AdaBoost have different contributions, i.e. some stumps have more say than others.
2. Decision Trees in Random Forest are built independently, whereas each Decision Stump in AdaBoost is made by taking the previous stump's mistake into account.

Decision Stumps are too simple; they are slightly better than random guessing and have a high bias. The goal of boosting is to reduce the bias of a single estimator, i.e. the bias of a single Decision Stump in the case of AdaBoost.
Recall the tax evasion dataset. You will use AdaBoost to predict whether a person will comply to pay taxes based on three features: Taxable Income, Marital Status, and Refund.

Train data

Id	Refund	Marital Status	Taxable Income	Evade
1	Yes	Single	125	No
2	No	Married	100	No
3	No	Single	70	No
4	Yes	Married	120	No
5	No	Divorced	95	Yes
6	No	Married	60	No
7	Yes	Divorced	220	No
8	No	Single	85	Yes
9	No	Married	75	No
10	No	Single	90	Yes

Test data

Id	Refund	Marital Status	Taxable Income	Evade
11	Yes	Single	80	?

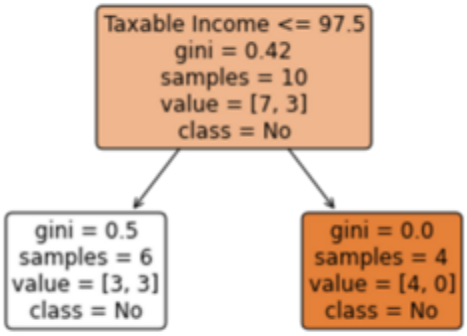
Tax evasion dataset | Image by author

Step 1. Initialize sample weight and build a Decision Stump

Remember that a Decision Stump in AdaBoost is made by taking the previous stump's mistake into account. To do that, AdaBoost assigns a sample weight to each train observation. Bigger weights are assigned to misclassified observations by the current stump, informing the next stump to pay more attention to those misclassified observations.
Initially, since there are no predictions made yet, all observations are given the same weight of $1/m$. Then, a Decision Stump is built just like creating a Decision Tree with depth 1.

Train data

Id	Refund	Marital Status	Taxable Income	Evade	Sample Weight
1	Yes	Single	125	No	1/10
2	No	Married	100	No	1/10
3	No	Single	70	No	1/10
4	Yes	Married	120	No	1/10
5	No	Divorced	95	Yes	1/10
6	No	Married	60	No	1/10
7	Yes	Divorced	220	No	1/10
8	No	Single	85	Yes	1/10
9	No	Married	75	No	1/10
10	No	Single	90	Yes	1/10



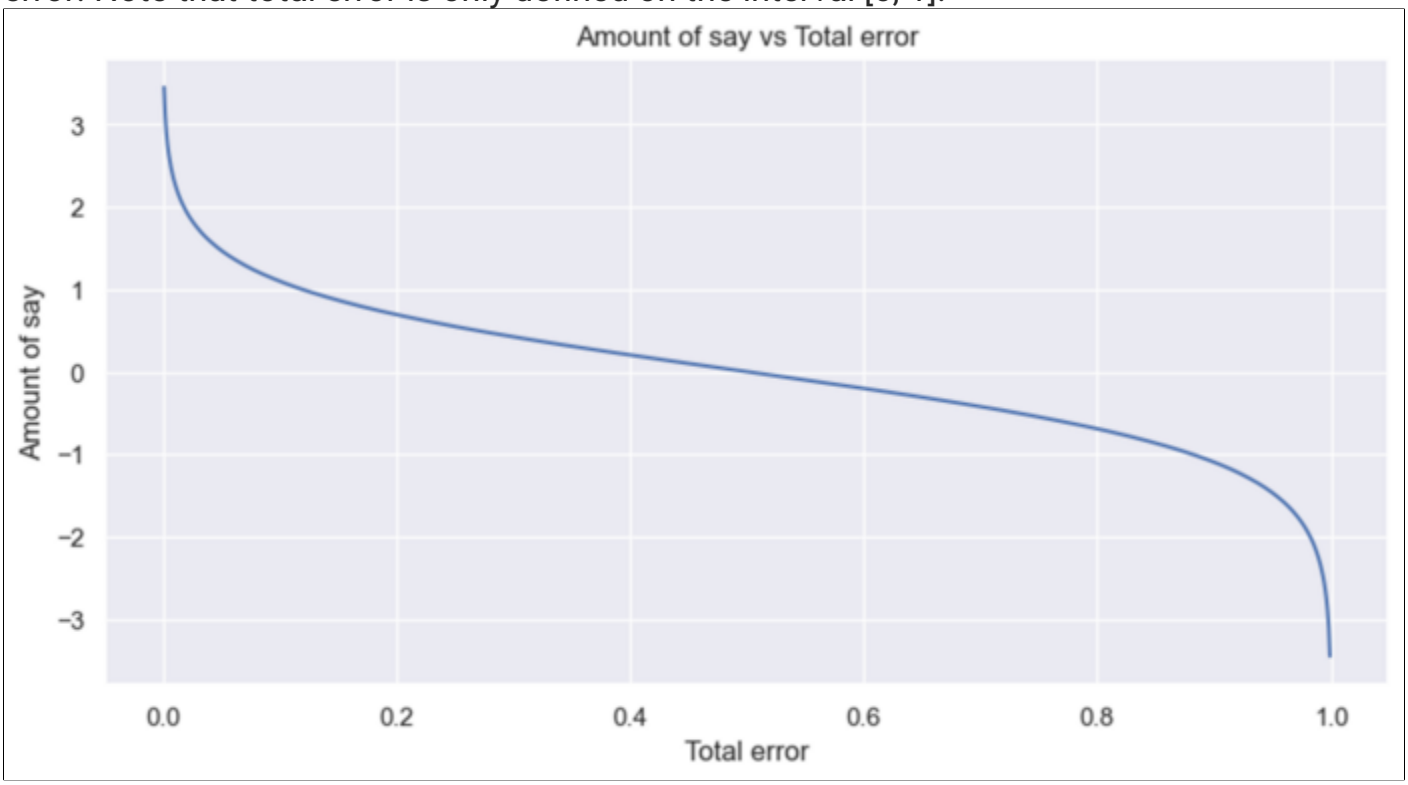
Initializing sample weight and building the first Decision Stump | Image by author

Step 2. Calculate the amount of say for the Decision Stump

This Decision Stump yields 3 misclassified observations. Since all observations have the same sample weight, the **total error** in terms of sample weight is 1/10 + 1/10 + 1/10 = 0.3.
The **amount of say** can then be calculated using the formula

$$\begin{aligned}\text{amount of say} &= \frac{1}{2}\ln\left(\frac{1-\text{total error}}{\text{total error}}\right)\\ &= \frac{1}{2}\ln\left(\frac{1-0.3}{0.3}\right)\\ &= 0.424\end{aligned}$$

To understand why the formula makes sense, let's plot the amount of say vs total error. Note that total error is only defined on the interval [0, 1].



Amount of say vs Total error | Image by author

If the total error is close to 0, the amount of say is a big positive number, indicating the stump gives many contributions to the final prediction. On the other hand, if the total error is close to 1, the amount of say is a big negative number, giving much penalty for the misclassification the stump produces. If the total error is 0.5, the stump doesn't do anything for the final prediction.

Step 3. Update sample weight

AdaBoost will increase the weights of incorrectly labeled observations and decrease the weights of correctly labeled observations. The new sample weight influences the next Decision Stump: observations with larger weights are paid more attention to.
To update the sample weight, use the formula

- For incorrectly labeled observations:

$$\begin{aligned}\text{new sample weight} &= \text{sample weight} \times e^{\text{amount of say}} \\ &= 0.1 \times e^{0.424} \\ &= 0.153\end{aligned}$$

- For correctly labeled observations:

$$\begin{aligned}\text{new sample weight} &= \text{sample weight} \times e^{-\text{amount of say}} \\ &= 0.1 \times e^{-0.424} \\ &= 0.065\end{aligned}$$

You need to normalize the new sample weights such that their sum equals 1 before continuing to the next step, using the formula

$$\text{norm. sample weight} = \frac{\text{new sample weight}}{\sum \text{new sample weight}}$$

These calculations are summarized in the table below. The green observations are correctly labeled and the red ones are incorrectly labeled by the current stump.

Train data

Id	Refund	Marital Status	Taxable Income	Evade	Sample Weight	Evade Pred.	New Sample Weight	Norm. Sample Weight
1	Yes	Single	125	No	0.100	No	0.065	0.071
2	No	Married	100	No	0.100	No	0.065	0.071
3	No	Single	70	No	0.100	No	0.065	0.071
4	Yes	Married	120	No	0.100	No	0.065	0.071
5	No	Divorced	95	Yes	0.100	No	0.153	0.167
6	No	Married	60	No	0.100	No	0.065	0.071
7	Yes	Divorced	220	No	0.100	No	0.065	0.071
8	No	Single	85	Yes	0.100	No	0.153	0.167
9	No	Married	75	No	0.100	No	0.065	0.071
10	No	Single	90	Yes	0.100	No	0.153	0.167

Updating sample weight for the first Decision Stump | Image by author

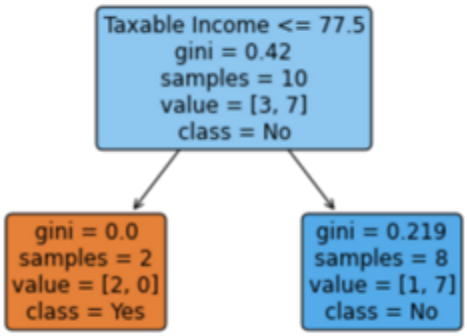
Step 4. Repeat

A new Decision Stump is created using the **weighted** impurity function, i.e. weighted Gini Index, where normalized sample weight is considered in determining the proportion of observations per class. Alternatively, another way to create a Decision Stump is to use bootstrapped data based on normalized sample weight.
Since normalized sample weight has a sum of 1, it can be considered as a probability that a particular observation is picked when bootstrapping. So, observations with Ids 5, 8, and 10 are more probable to be picked and hence the Decision Stump is more focused on them.

The bootstrapped data is shown below. Note that you may get different bootstrapped data due to randomness. Build a Decision Stump as before.

Train data

Id	Refund	Marital Status	Taxable Income	Evade	Sample Weight
5	No	Divorced	95	Yes	0.100
10	No	Single	90	Yes	0.100
8	No	Single	85	Yes	0.100
8	No	Single	85	Yes	0.100
3	No	Single	70	No	0.100
3	No	Single	70	No	0.100
1	Yes	Single	125	No	0.100
10	No	Single	90	Yes	0.100
8	No	Single	85	Yes	0.100
8	No	Single	85	Yes	0.100



Initializing sample weight and building the second Decision Stump | Image by author

You see that there's one misclassified observation. So, the amount of say is 1.099. Update the sample weight and normalize with respect to the sum of weights of only bootstrapped observations. These calculations are summarized in the table below.

Train data

Id	Refund	Marital Status	Taxable Income	Evade	Sample Weight	Evade Pred.	New Sample Weight	Norm. Sample Weight
5	No	Divorced	95	Yes	0.100	Yes	0.033	0.036
10	No	Single	90	Yes	0.100	Yes	0.033	0.036
8	No	Single	85	Yes	0.100	Yes	0.033	0.036
8	No	Single	85	Yes	0.100	Yes	0.033	0.036
3	No	Single	70	No	0.100	No	0.033	0.036
3	No	Single	70	No	0.100	No	0.033	0.036
1	Yes	Single	125	No	0.100	Yes	0.300	0.321
10	No	Single	90	Yes	0.100	Yes	0.033	0.036
8	No	Single	85	Yes	0.100	Yes	0.033	0.036
8	No	Single	85	Yes	0.100	Yes	0.033	0.036

Updating sample weight for the second Decision Stump | Image by author

The bootstrapped observations go back to the pool of train data and the new sample weight becomes as shown below. You can then bootstrap the train data again using the new sample weight for the third Decision Stump and the process goes on.

Train data

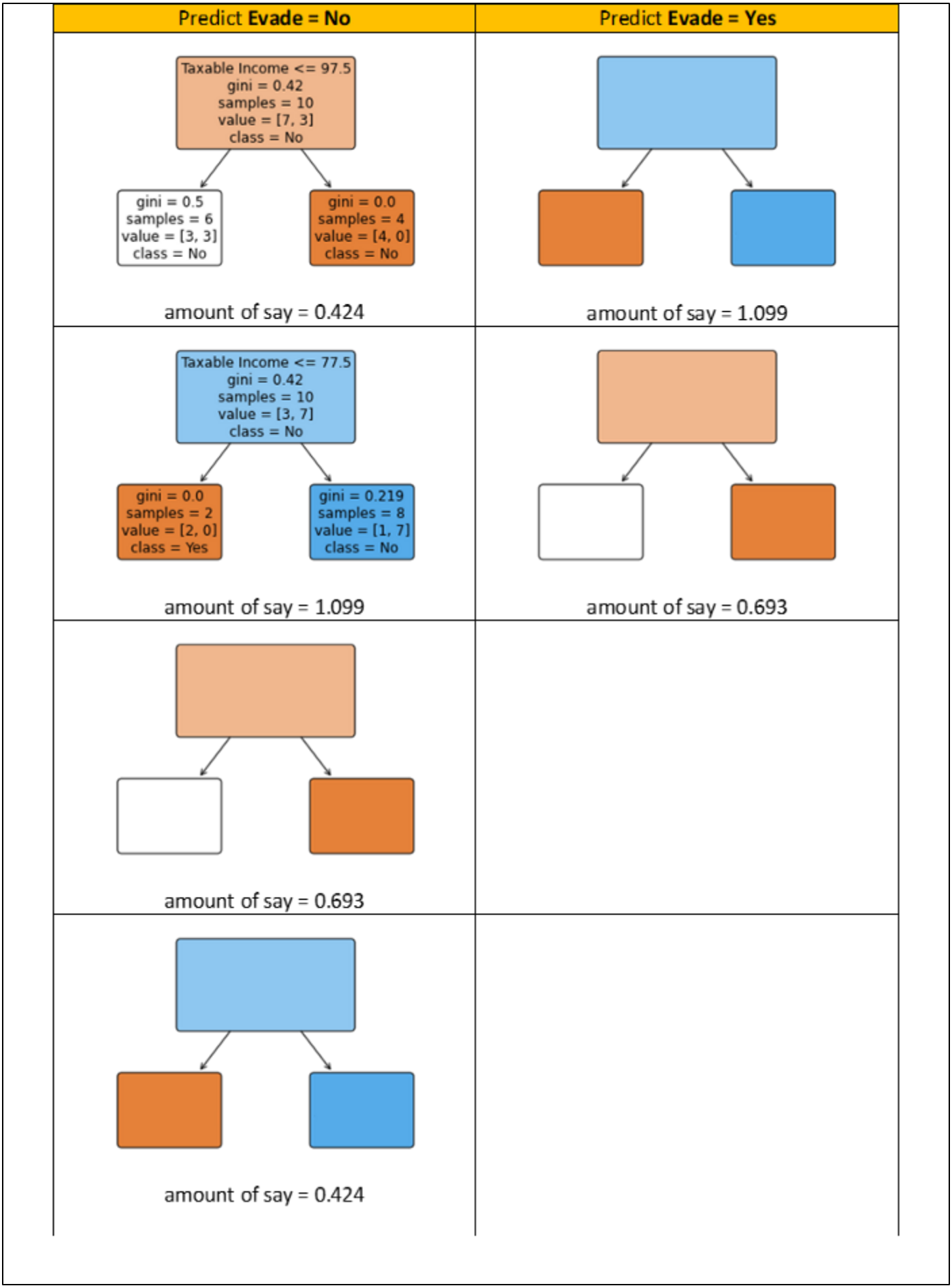
Id	Refund	Marital Status	Taxable Income	Evade	Sample Weight
1	Yes	Single	125	No	0.321
2	No	Married	100	No	0.071
3	No	Single	70	No	0.071
4	Yes	Married	120	No	0.071
5	No	Divorced	95	Yes	0.036
6	No	Married	60	No	0.071
7	Yes	Divorced	220	No	0.071
8	No	Single	85	Yes	0.143
9	No	Married	75	No	0.071
10	No	Single	90	Yes	0.071

Train data with its sample weight | Image by author

Prediction

Let's say you iterate the steps six times. Thus, you have six Decision Stumps. To make predictions, traverse each stump using the test data and group the predicted classes. The group with the largest amount of say represents the final prediction of AdaBoost.

To illustrate, the first two Decision Stumps you built have the amount of say 0.424 and 1.099, respectively. Both stumps predict **Evade = No** since for the taxable income, $77.5 < 80 \leq 97.5$ (80 is the value of Taxable Income for the test observation). Let's say the other four Decision Stumps predict and have the following amount of say.



Decision Stumps' predictions and their amount of say | Image by [author]

Since the sum of the amount of say of all stumps that predict **Evade = No** is 2.639, bigger than the sum of the amount of say of all stumps that predict **Evade = Yes** (which is 1.792), then the final AdaBoost prediction is **Evade = No**.

Gradient Boosting:

Since both are boosting methods, AdaBoost and Gradient Boosting have a similar workflow. There are two main differences though:

1. Gradient Boosting uses trees larger than a Decision Stump. In this story, we limit the trees to have a maximum of 3 leaf nodes, which is a hyperparameter that can be changed at will.
2. Instead of using sample weight to guide the next Decision Stump, Gradient Boosting uses the residual made by the Decision Tree to guide the next tree.

Recall the tax evasion dataset. You will use Gradient Boosting to predict whether a person will comply to pay taxes based on three features: Taxable Income, Marital Status, and Refund.

Train data

Id	Refund	Marital Status	Taxable Income	Evade
1	Yes	Single	125	No
2	No	Married	100	No
3	No	Single	70	No
4	Yes	Married	120	No
5	No	Divorced	95	Yes
6	No	Married	60	No
7	Yes	Divorced	220	No
8	No	Single	85	Yes
9	No	Married	75	No
10	No	Single	90	Yes

Test data

Id	Refund	Marital Status	Taxable Income	Evade
11	Yes	Single	80	?

Tax evasion dataset | Image by author

Step 1. Initialize a root

A root is a Decision Tree with zero depth. It doesn't consider any feature to do prediction: it uses Evade itself to predict on Evade. How does it do this? Encode Evade using the mapping **No → 0** and **Yes → 1**. Then, take the average, which is 0.3. This is called the **prediction probability** (the value is always between 0 and 1) and is denoted by *p*.
Since 0.3 is less than the threshold of 0.5, the root will predict **No** for every train observation. In other words, the prediction is always the *mode* of Evade. Of course, this is a very bad initial prediction. To improve the performance, you can calculate the **log(odds)** to be used in the next step using the formula

$$\begin{aligned}\log(\text{odds}) &= \ln\left(\frac{p}{1-p}\right)\\ &= \ln\left(\frac{0.3}{1-0.3}\right)\\ &= -0.847\end{aligned}$$

Step 2. Calculate the residual to fit a Decision Tree into

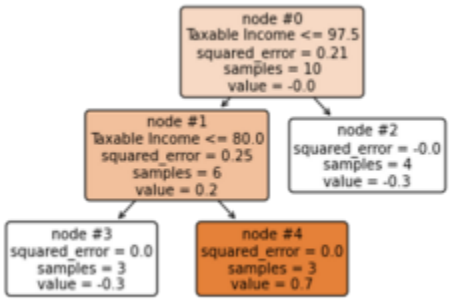
To simplify notations, let the encoded column of Evade be denoted by *y*. Residual is defined simply by $r = y - p$, where p is the prediction probability calculated from the latest log(odds) using its inverse function

$$\begin{aligned}p &= \frac{e^{\log(\text{odds})}}{1+e^{\log(\text{odds})}}\\ &= \frac{e^{-0.847}}{1+e^{-0.847}}\\ &= 0.3\end{aligned}$$

Build a Decision Tree to predict the residual using all train observations. Note that this is a regression task.

Train data

Id	Refund	Marital Status	Taxable Income	y	log(odds)	p	r	Node Id
1	Yes	Single	125	0	-0.847	0.300	-0.300	#2
2	No	Married	100	0	-0.847	0.300	-0.300	#2
3	No	Single	70	0	-0.847	0.300	-0.300	#3
4	Yes	Married	120	0	-0.847	0.300	-0.300	#2
5	No	Divorced	95	1	-0.847	0.300	0.700	#4
6	No	Married	60	0	-0.847	0.300	-0.300	#3
7	Yes	Divorced	220	0	-0.847	0.300	-0.300	#2
8	No	Single	85	1	-0.847	0.300	0.700	#4
9	No	Married	75	0	-0.847	0.300	-0.300	#3
10	No	Single	90	1	-0.847	0.300	0.700	#4



Calculating the residual and fitting a Decision Tree to predict the residual | Image by author

Step 3. Update log(odds)

Note that I also added a Node Id column that explains the index of the node in the Decision Tree where each observation is predicted. In the table above, each observation is color-coded to easily see which observation goes to which node.

To combine the predictions of the initial root and the Decision Tree you just built, you need to transform the predicted values in the Decision Tree such that they match the log(odds) in the initial root.

For example, let d_i denote the set of observation Ids in node #*i*, i.e. $d_2 = \{1, 2, 4, 7\}$. Then the transformation for node #2 is

$$\gamma_{d_2} = \frac{\sum_{i \in d_2} r_i}{\sum_{i \in d_2} p_i(1-p_i)}$$
$$= \frac{-0.3 - 0.3 - 0.3 - 0.3}{0.3 \times (1-0.3) + 0.3 \times (1-0.3) + 0.3 \times (1-0.3) + 0.3 \times (1-0.3)}$$
$$= -1.429$$

Do the same calculation to node #3 and node #4, you get γ for all training observations as the following table. Now you can update the log(odds) using the formula

$$\log(\text{odds}) \leftarrow \log(\text{odds}) + \alpha\gamma,$$

where α is the learning rate of Gradient Boosting and is a hyperparameter. Let's set $\alpha = 0.4$, then the complete calculation of updating log(odds) can be seen below.

Train data

Id	Refund	Marital Status	Taxable Income	y	log(odds)	p	r	Node Id	gamma	New log(odds)
1	Yes	Single	125	0	-0.847	0.300	-0.300	#2	-1.429	-1.419
2	No	Married	100	0	-0.847	0.300	-0.300	#2	-1.429	-1.419
3	No	Single	70	0	-0.847	0.300	-0.300	#3	-1.429	-1.419
4	Yes	Married	120	0	-0.847	0.300	-0.300	#2	-1.429	-1.419
5	No	Divorced	95	1	-0.847	0.300	0.700	#4	3.333	0.486
6	No	Married	60	0	-0.847	0.300	-0.300	#3	-1.429	-1.419
7	Yes	Divorced	220	0	-0.847	0.300	-0.300	#2	-1.429	-1.419
8	No	Single	85	1	-0.847	0.300	0.700	#4	3.333	0.486
9	No	Married	75	0	-0.847	0.300	-0.300	#3	-1.429	-1.419
10	No	Single	90	1	-0.847	0.300	0.700	#4	3.333	0.486

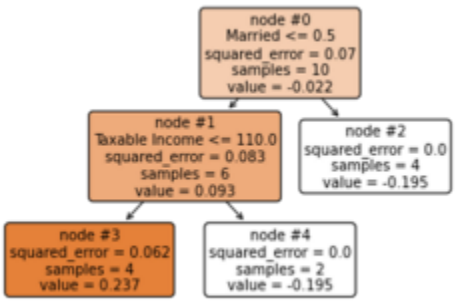
Updating log(odds) | Image by author

Step 4. Repeat

Using the new log(odds), repeat Step 2 and Step 3. Here's the new residual and the Decision Tree built to predict it.

Train data

Id	Refund	Marital Status	Taxable Income	y	log(odds)	p	r	Node Id
1	Yes	Single	125	0	-1.419	0.195	-0.195	#4
2	No	Married	100	0	-1.419	0.195	-0.195	#2
3	No	Single	70	0	-1.419	0.195	-0.195	#3
4	Yes	Married	120	0	-1.419	0.195	-0.195	#2
5	No	Divorced	95	1	0.486	0.619	0.381	#3
6	No	Married	60	0	-1.419	0.195	-0.195	#2
7	Yes	Divorced	220	0	-1.419	0.195	-0.195	#4
8	No	Single	85	1	0.486	0.619	0.381	#3
9	No	Married	75	0	-1.419	0.195	-0.195	#2
10	No	Single	90	1	0.486	0.619	0.381	#3



Calculating the residual and fitting a Decision Tree to predict the residual | Image by author

Calculate γ as before. For example, consider node #3, then

$$\gamma_{d_3} = \frac{\sum_{i \in d_3} r_i}{\sum_{i \in d_3} p_i(1-p_i)}$$
$$= \frac{-0.195 + 0.381 + 0.381 + 0.381}{0.195(1-0.195) + 0.619(1-0.619) + 0.619(1-0.619) + 0.619(1-0.619)}$$
$$= 1.096$$

Using γ, calculate the new log(odds) as before. Here's the complete calculation

Train data

Id	Refund	Marital Status	Taxable Income	y	log(odds)	p	r	Node Id	gamma	New log(odds)
1	Yes	Single	125	0	-1.419	0.195	-0.195	#4	-1.242	-1.916
2	No	Married	100	0	-1.419	0.195	-0.195	#2	-1.242	-1.916
3	No	Single	70	0	-1.419	0.195	-0.195	#3	1.096	-0.980
4	Yes	Married	120	0	-1.419	0.195	-0.195	#2	-1.242	-1.916
5	No	Divorced	95	1	0.486	0.619	0.381	#3	1.096	0.925
6	No	Married	60	0	-1.419	0.195	-0.195	#2	-1.242	-1.916
7	Yes	Divorced	220	0	-1.419	0.195	-0.195	#4	-1.242	-1.916
8	No	Single	85	1	0.486	0.619	0.381	#3	1.096	0.925
9	No	Married	75	0	-1.419	0.195	-0.195	#2	-1.242	-1.916
10	No	Single	90	1	0.486	0.619	0.381	#3	1.096	0.925

Updating log(odds) | Image by author

You can repeat the process as many times as you like. For this story, however, to avoid things getting out of hand, we will stop the iteration now.

Prediction

To do prediction on test data, first, traverse each Decision Tree until reaching a leaf node. Then, add the log(odds) of the initial root with all the γ of the corresponding leaf nodes scaled by the learning rate.
In our example, the test observation lands on node #3 of both Decision Trees. Hence, the summation becomes -0.847 + 0.4 × (-1.429 + 1.096) = -0.980. Convert to prediction probability

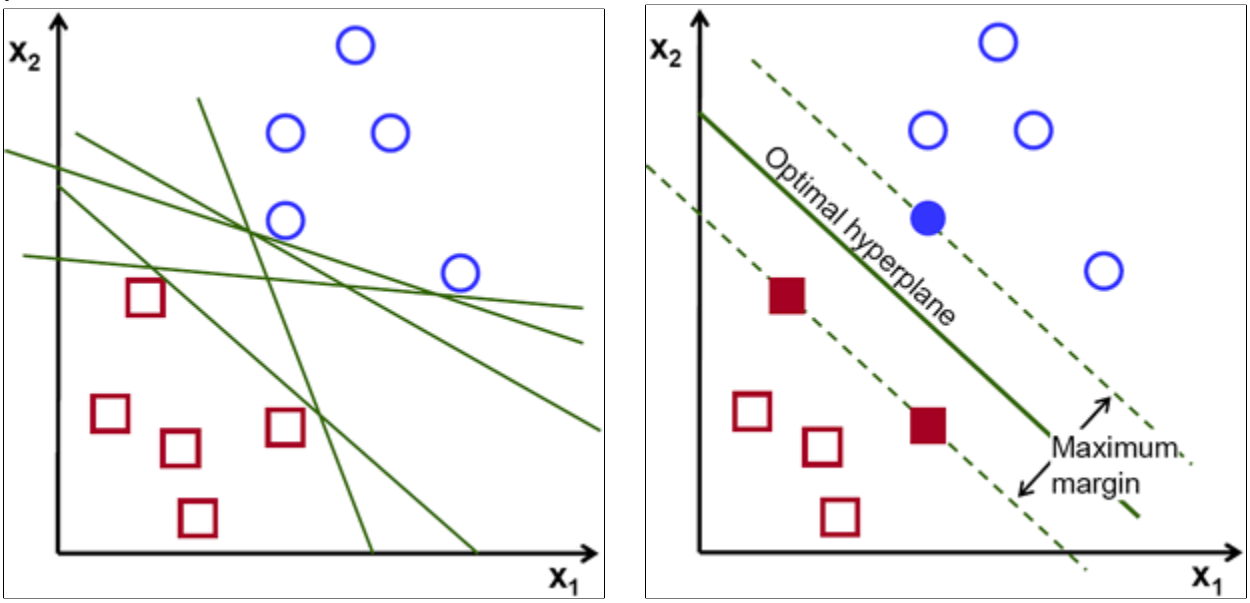
$$p = \frac{e^{\log(\text{odds})}}{1 + e^{\log(\text{odds})}}$$
$$= \frac{e^{-0.980}}{1 + e^{-0.980}}$$
$$= 0.273$$

Since 0.273 < 0.5, predict **Evade = No**.

Support Vector Machines – Large Margin Intuition – Loss Function - Hinge Loss – SVM Kernels

Support Vector Machine?

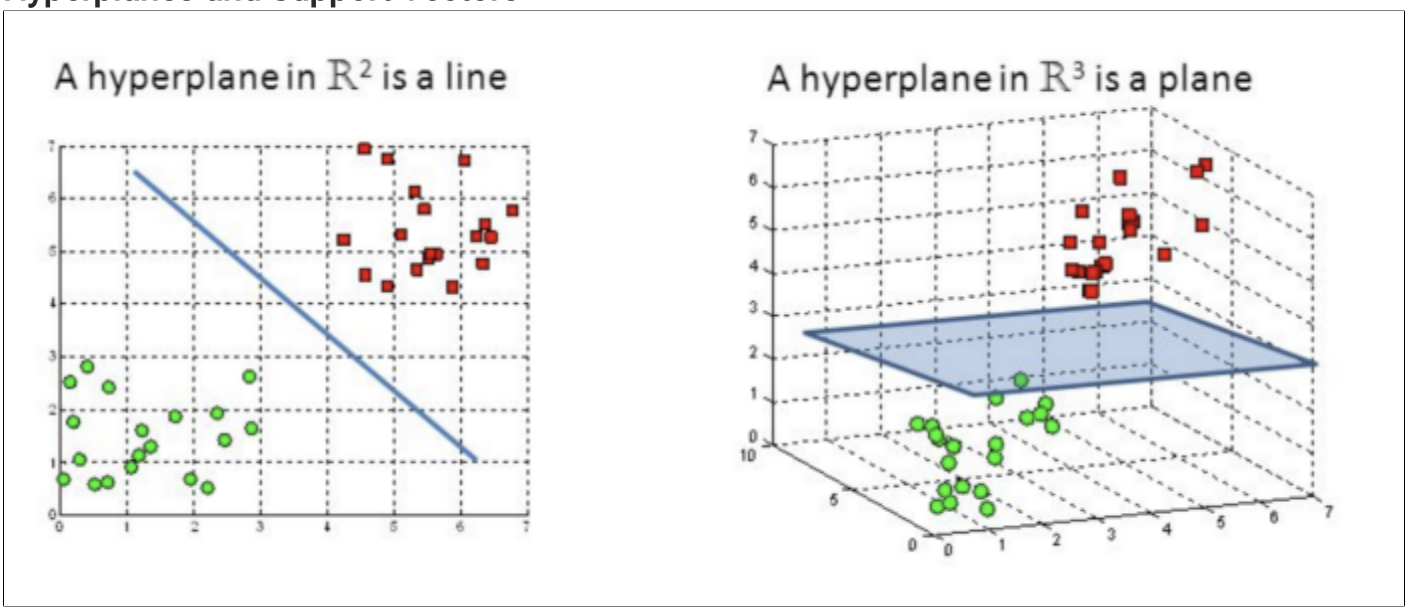
The objective of the support vector machine algorithm is to find a hyperplane in an N-dimensional space(N — the number of features) that distinctly classifies the data points.



Possible hyperplanes

To separate the two classes of data points, there are many possible hyperplanes that could be chosen. Our objective is to find a plane that has the maximum margin, i.e the maximum distance between data points of both classes. Maximizing the margin distance provides some reinforcement so that future data points can be classified with more confidence.

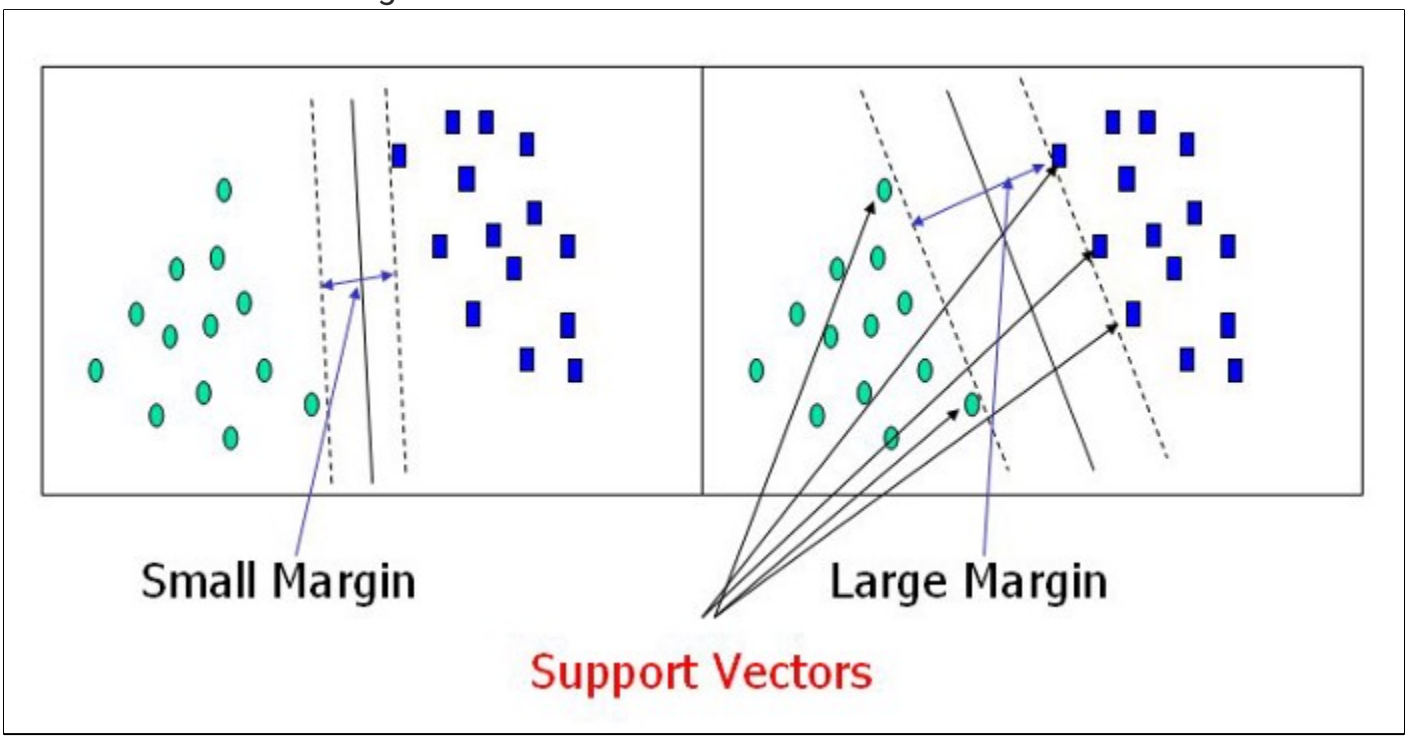
Hyperplanes and Support Vectors



Hyperplanes in 2D and 3D feature space

Hyperplanes are decision boundaries that help classify the data points. Data points

falling on either side of the hyperplane can be attributed to different classes. Also, the dimension of the hyperplane depends upon the number of features. If the number of input features is 2, then the hyperplane is just a line. If the number of input features is 3, then the hyperplane becomes a two-dimensional plane. It becomes difficult to imagine when the number of features exceeds 3.



Support Vectors

Support vectors are data points that are closer to the hyperplane and influence the position and orientation of the hyperplane. Using these support vectors, we maximize the margin of the classifier. Deleting the support vectors will change the position of the hyperplane. These are the points that help us build our SVM.

Large Margin Intuition

In logistic regression, we take the output of the linear function and squash the value within the range of [0,1] using the sigmoid function. If the squashed value is greater than a threshold value(0.5) we assign it a label 1, else we assign it a label 0. In SVM, we take the output of the linear function and if that output is greater than 1, we identify it with one class and if the output is -1, we identify is with another class. Since the threshold values are changed to 1 and -1 in SVM, we obtain this reinforcement range of values([-1,1]) which acts as margin.

Loss Functions

In order to better understand the behavior of SVMs, and how they compare to other methods, we will analyze them in terms of their **loss functions.**[1] In some cases, this loss function might come from the problem being solved: for example, we might pay a certain dollar amount if we incorrectly classify a vector, and the penalty for a false positive might be very different for the penalty for a false negative. The rewards and losses due to correct and incorrect classification depend on the particular problem being optimized. Here, we will simply attempt to minimize the total number of classification errors, using a penalty is called the **0-1 Loss:**

$$L_{0-1}(\mathbf{x}, y) = \begin{cases} 1 & yf(\mathbf{x}) < 0 \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

(Note that $yf(\mathbf{x}) > 0$ is the same as requiring that y and $f(\mathbf{x})$ have the same sign.) This loss function says that we pay a penalty of 1 when we misclassify a new input, and a penalty of zero if we classify it correctly.

Ideally, we would choose the classifier to minimize the loss over the new test data that we are given; of course, we don't know the true labels, and instead we optimize the following surrogate objective function over the training data:

$$E(\mathbf{w}) = \sum_i L(\mathbf{x}_i, y_i) + \lambda R(\mathbf{w}) \tag{13}$$

[1] A loss function specifies a measure of the quality of a solution to an optimization problem. It is the penalty function that tell us how badly we want to penalize errors in a models ability to fit the data. In probabilistic methods it is typically the negative log likelihood or the negative log posterior.

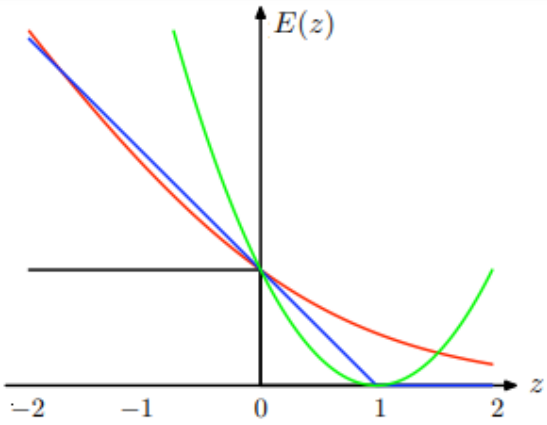


Figure 3: Loss functions, $E(z)$, for learning, for $z = y\,f(x)$. Black: 0-1 loss. Red: LR loss. Green: Quadratic loss ($(z-1)^2$). Blue: Hinge loss. (Figure from *Pattern Recognition and Machine Learning* by Chris Bishop.)

where $R(\mathbf{w})$ is a regularizer meant to prevent overfitting (and thus improve performance on future test data). The basic assumption is that loss on the training set should correspond to loss on the test set. If we can get the classifier to have small loss on the training data, while also being smooth, then the loss we pay on new data ought to not be too big either. This optimization framework is equivalent to MAP estimation as discussed previously[2]; however, here we are not at all concerned with probabilities. We only care about whether the classifier gets the right answers or not.

Unfortunately, optimizing a classifier for the 0-1 loss is very difficult: it is not differentiable everywhere, and, where it is differentiable, the gradient is zero everywhere. There are a set of algorithms called Perceptron Learning which attempt to do this; of these, the Voted Perceptron algorithm is considered one of the best. However, these methods are somewhat complex to analyze and we will not discuss them further. Instead, we will use other loss functions that approximate 0-1 loss.

We can see that maximum likelihood logistic regression is equivalent to optimization with the following loss function:

$$L_{\text{LR}} = \ln\left(1 + e^{-yf(\mathbf{x})}\right) \tag{14}$$

which is the negative log-likelihood of a single data vector. This function is a poor approximation to the 0-1 loss, and, if all we care about is getting the labels right (and not the class probabilities), then we ought to search for a better approximation.

SVMs minimize the slack variables, which, from the constraints, can be seen to give the **hinge loss**:

$$L_{hinge} = \begin{cases} 1 - yf(\mathbf{x}) & 1 - yf(\mathbf{x}) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

[2]However, not all loss functions can be viewed as the negative log of a valid likelihood function, although all negative-log likelihoods can be viewed as loss functions for learning.

That is, when a data point is correctly classified and further from the decision boundary than the margin, then the loss is zero. In this way it is insensitive to correctly-classified points far from the boundary. But when the point is within the margin or incorrectly classified, then the loss is simply the magnitude of the slack variable, i.e. $\xi = 1 - yf(\mathbf{x})$, where $f(\mathbf{x}) = \mathbf{w}^T\phi(\mathbf{x}) + b$. The hinge loss therefore increases linearly for misclassified points, which is not nearly as quickly as the LR loss.

Large Margin SVMs as regularized hinge loss

Recall, binary linear classifiers aim to differentiate observations into 2 categories. The classification is accomplished by way of a decision boundary dividing the data space into 2 regions, where $f(w) < 0$ and $f(w) > 0$. The classification boundary, also known as the discriminate, corresponds to $f(w) = 0$. In 2D, the boundary is a line whereas it is a plane or hyper-plane in higher dimensions.

Linear classifiers suffer from what is called the "Identifiability Problem" whereby multiple models that lead to 0 training errors exist. Furthermore, the identification of the best model among the 0 training error options is not possible.

Support Vector Machines

Support Vector Machines (SVMs) are supervised learning models used for binary classification problems. They have been successful in classifying observations as either positive or negative instances by exploiting the concept of regularized hinge loss. Hinge loss is a cost function that employs a margin from the classification boundary as a measure of confidence. Observations residing farther from the classification boundary are considered higher confidence classifications.

The equation for an SVM model is

$$f_w(w) = w^T x + w_0 \tag{6}$$

The outputs are

$$\hat{y} = sign(f_w(w)) \in \{-1, +1\} \tag{7}$$

Accordingly, the classification boundary whose margin, defined as the perpendicular distance in both the positive and negative directions from the classification boundary prior to encountering a point in the training data set, is maximal is selected. The margin is equidistant from $f(w) = 0$ in the positive and negative direction and therefore, parallel to $f(w) = 0$. This method enables the selection of an optimal model by seeking the boundary with the largest margin. The margin distance can be defined by constant k as follows

$$w^T x + w_0 \geq k \quad \text{when} \quad y_i = +1 \tag{8}$$

$$w^T x + w_0 \leq -k \quad \text{when} \quad y_i = -1 \tag{9}$$

The concept in 2D is depicted in Figure 1.

The optimization of an SVM is given by

$$min \quad C \sum_{i=1} max(0, 1 - y_i[w^T x_i + w_0]) + \lambda \parallel w \parallel_2^2 \tag{10}$$

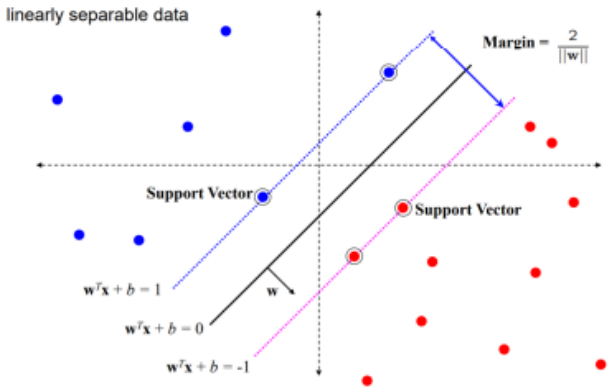


Figure 1: SVM Margin

The first term is the hinge loss and represents the classification errors. The second term is the regularization term. The hinge loss is zero if the data point is classified correctly and increases when data points reside close to the classification boundary inside the margin. A data point residing on the margin will have a hinge loss of 0 (Figure 2).

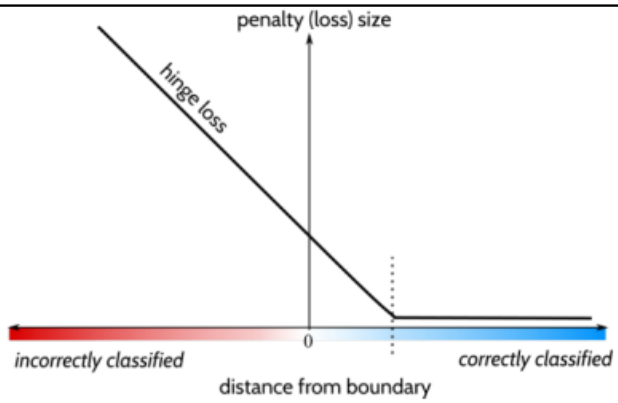


Figure 2: Hinge Loss

The parameter C is the regularization parameter. The regularization parameter becomes relevant when data sets are overlapping, i.e. not definitively separable by a classification boundary. This is known as a soft margin SVM. Large regularization parameter values correspond to smaller margins leading to an increased number of correct classifications. Conversely, large margins are associated with smaller parameter values and more instances of misclassification.

The distance, $d_w(x_i)$, of a data point x_i from the line $f(w) = w^T x_i + w_0$ is

$$d_w(x_i) = \frac{w^T x_i + w_0}{\| w \|_2} \tag{11}$$

Zero loss occurs beyond the margin, specifically when,

$$max(0, 1 - y_i[w^T x_i + w_0]) = 0 \tag{12}$$

which then implies

$$y_i[w^T x_i + w_0] = 1 \tag{13}$$

when a data point is on the margin.

Then, by equation 11,

$$d_w(x_i) = \frac{1}{y_i \parallel w \parallel_2} = \frac{1}{\parallel w \parallel_2} \tag{14}$$

As a result, minimizing the loss function is equivalent to maximizing the margin because the training errors are fewest when the margin distance is maximal. Mathematically, the loss function is minimized when the hinge loss term in equation 5 is 0 and the L2 Norm is minimized

$$min \parallel w \parallel_2^2 \tag{16}$$

Minimizing the L2 Norm (equation 11) is equivalent to maximizing the margin (equation 10).

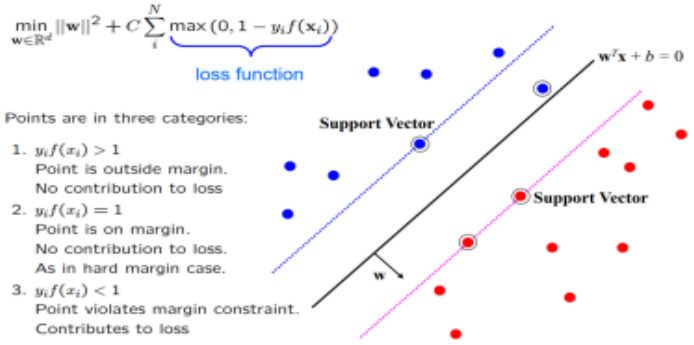


Figure 3: Hinge Loss

Support Vector Machines (Kernels)

The SVM algorithm is implemented in practice using a kernel.
The learning of the hyperplane in linear SVM is done by transforming the problem using some linear algebra, which is out of the scope of this introduction to SVM.
A powerful insight is that the linear SVM can be rephrased using the inner product of any two given observations, rather than the observations themselves. The inner product between two vectors is the sum of the multiplication of each pair of input values.
For example, the inner product of the vectors [2, 3] and [5, 6] is 2*5 + 3*6 or 28.
The equation for making a prediction for a new input using the dot product between the input (x) and each support vector (xi) is calculated as follows:
f(x) = B0 + sum(ai * (x,xi))
This is an equation that involves calculating the inner products of a new input vector (x) with all support vectors in training data. The coefficients B0 and ai (for each input) must be estimated from the training data by the learning algorithm.

Linear Kernel SVM

The dot-product is called the kernel and can be re-written as:
K(x, xi) = sum(x * xi)
The kernel defines the similarity or a distance measure between new data and the support vectors. The dot product is the similarity measure used for linear SVM or a linear kernel because the distance is a linear combination of the inputs.
Other kernels can be used that transform the input space into higher dimensions such as a Polynomial Kernel and a Radial Kernel. This is called the Kernel Trick.
It is desirable to use more complex kernels as it allows lines to separate the classes that are curved or even more complex. This in turn can lead to more accurate classifiers.

Polynomial Kernel SVM

Instead of the dot-product, we can use a polynomial kernel, for example:
K(x,xi) = 1 + sum(x * xi)^d
Where the degree of the polynomial must be specified by hand to the learning algorithm. When d=1 this is the same as the linear kernel. The polynomial kernel allows for curved lines in the input space.

Radial Kernel SVM

Finally, we can also have a more complex radial kernel. For example:
K(x,xi) = exp(-gamma * sum((x – xi^2))
Where gamma is a parameter that must be specified to the learning algorithm. A good default value for gamma is 0.1, where gamma is often 0 < gamma < 1. The radial kernel is very local and can create complex regions within the feature space, like closed polygons in two-dimensional space.