Machine Learning Notes

CHAPTER 1 : INTRODUCTION

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

For example, a computer program that learns to play checkers might improve its performance as measured by its ability to win at the class of tasks involving playing checkers games, through experience obtained by playing games against itself. In general, to have a well-defined learning problem, we must identity these three features:

1. the class of tasks,
2. the measure of performance to be improved, and
3. the source of experience.

A checkers learning problem:

- **Task T:** playing checkers
- **Performance measure P:** percent of games won against opponents
- **Training experience E:** playing practice games against itself

We can specify many learning problems in this fashion, such as learning to recognize handwritten words, or learning to drive a robotic automobile autonomously.

A handwriting recognition learning problem:

Task T: recognizing and classifying handwritten words within images

Performance measure P: percent of words correctly classified

Training experience E: a database of handwritten words with given classifications

A robot driving learning problem:

Task T: driving on public four-lane highways using vision sensors

Performance measure P: average distance travelled before an error (as judged by human overseer)

Training experience E: a sequence of images and steering commands recorded while observing a human driver

CHAPTER 2: CONCEPT LEARNING AND THE GENERAL-TO-SPECIFIC 0RDERING

INTRODUCTION

Concept learning. Inferring a Boolean valued function from training examples of its input and output.

A CONCEPT LEARNING TASK

The attribute EnjoySport indicates whether or not Aldo enjoys his favourite water sport on this day. The task is to learn to predict the value of EnjoySport for an arbitrary day, based on the values of its other attributes.

Let us begin by considering a simple representation in which each hypothesis consists of a conjunction of constraints on the instance attributes. In particular, let each hypothesis be a vector of six constraints, specifying the values of the six attributes Sky, AirTemp, Humidity, Wind, Water, and Forecast. For each attribute, the hypothesis will either 0 indicate by a "?' that any value is acceptable for this attribute, 0 specify a single required value (e.g., Warm) for the attribute, or 0 indicate by a "0" that no value is acceptable.

- indicate by a "?' that any value is acceptable for this attribute,
- specify a single required value (e.g., Warm) for the attribute, or
- indicate by a "Φ" that no value is acceptable.

If some instance x satisfies all the constraints of hypothesis h, then h classifies x as a positive example (h(x) = 1). To illustrate, the hypothesis that Aldo enjoys his favourite sport only on cold days with high humidity (independent of the values of the other attributes) is represented by the expression

(?, Cold, High, ?, ?, ?)

Example Sky AirTemp Humidity Wind Water Forecast EnjoySport

1 Sunny Warm Normal Strong Warm Same Yes

2 Sunny Warm High Strong Warm Same Yes

3 Rainy Cold High Strong Warm Change No

4 Sunny Warm High Strong Cool Change Yes

The most general hypothesis-that every day is a positive example-is represented by

(?, ?, ?, ?, ?, ?)

and the most specific possible hypothesis-that no day is a positive example-is represented by

(Φ, Φ, Φ, Φ, Φ, Φ)

Notation

The set of items over which the concept is defined is called the set of instances, which we denote by X. In the current example, X is the set of all possible days, each represented by the attributes Sky, AirTemp, Humidity, Wind, Water, and Forecast. The concept or function to be learned is called the target concept, which we denote by c. In general, c can be any boolean-valued function defined over the instances X; that is, c : X -> {O, 1}.

In the current example, the target concept corresponds to the value of the attribute EnjoySport

(i.e., c(x) = 1 if EnjoySport = Yes, and c(x) = 0 if EnjoySport = No).

- Given: 0
    - Instances X: Possible days, each described by the attributes
        - Sky (with possible values Sunny, Cloudy, and Rainy),
        - AirTemp (with values Warm and Cold),
        - Humidity (with values Normal and High),
        - Wind (with values Strong and Weak),
        - Water (with values Warm and Cool), and
        - Forecast (with values Same and Change).

- Hypotheses H: Each hypothesis is described by a conjunction of constraints on the attributes Sky, AirTemp, Humidity, Wind, Water, and Forecast. The constraints may be "?" (any value is acceptable), " Φ"(no value is acceptable), or a specific value.
- Target concept c: EnjoySport : X -> {0,1}
- Training examples D: Positive and negative examples of the target function (see Table 2.1).
- Determine:
  - hypothesis h in H such that h(x) = c(x) for all x in X.

Instances for which c(x) = 1 are called positive examples, or members of the target concept. Instances for which c(x) = 0 are called negative examples, or nonmembers of the target concept.

In general, each hypothesis h in H represents a boolean-valued function defined over X; that is, h : X -> {O, 1}. The goal of the learner is to find a hypothesis h such that h(x) = c(x) for all x in X.

The Inductive Learning Hypothesis

The inductive learning hypothesis. Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.

General-to-Specific Ordering of Hypotheses(Not Necessary)

To illustrate the general-to-specific ordering, consider the two hypotheses

h1 = (Sunny, ?, ?, Strong, ?, ?)

h2 = (Sunny, ?, ?, ?, ?, ?)

Now consider the sets of instances that are classified positive by h1 and by h2. Because h2 imposes fewer constraints on the instance, it classifies more instances as positive. In fact, any instance classified positive by h1 will also be classified positive by h2. Therefore, we say that h2 is more general than h1.

This intuitive "more general than" relationship between hypotheses can be defined more precisely as follows. First, for any instance x in X and hypothesis h in H, we say that x satisfies h if and only if h(x) = 1. We now define the more_general_than_or_equal_to relation in terms of the sets of instances that satisfy the two hypotheses: Given hypotheses h_j and h_k, h_j is more_general_than_or_equal_to h_k if and only if any instance that satisfies h_k also satisfies h_i.

Definition: Let h_j and h_k be boolean-valued functions defined over X. Then h_j is more_general_than_or_equal_to h_k (written $h_j >=_g h_k$) if and only if

$$(\forall x \in X)[(h_k(x) = 1) \rightarrow (h_j(x) = 1)]$$

We will also find it useful to consider cases where one hypothesis is strictly more general than the other. Therefore, we will say that h_j is (strictly) more_general_than_or_equal_to h_k (written $h_j >_g h_k$) if and only if ($h_j >=_g h_k$) ($h_k >!=_g h_i$). Finally, we will sometimes find the inverse useful and will say that hj is more_specific_than h_k when h_k is more_general_than h_j.

The $>=_g$ relation is important because it provides a useful structure over the hypothesis space H for any concept learning problem. The following sections present concept learning algorithms that take advantage of this partial order to efficiently organize the search for hypotheses that fit the training data.

FIND-S: FINDING A MAXIMALLY SPECIFIC HYPOTHESIS

Find-S Algorithm is a basic concept learning algorithm which find the most specific hypothesis that fits all the positive example.

The **find-S Algorithm** also starts with most specific hypothesis & generalizes this hypothesis each time. It fails classify and observe positive training data.

Hence **find-S Algorithm** moves from the most specific hypothesis to most generalizes hypothesis.

CHAPTER 6: BAYESIAN LEARNING

INTRODUCTION

Bayesian learning methods are relevant to our study of machine learning for two different reasons. First, Bayesian learning algorithms that calculate explicit probabilities for hypotheses, such as the naive Bayes classifier, are among the most practical approaches to certain types of learning problems.

The second reason that Bayesian methods are important to our study of machine learning is that they provide a useful perspective for understanding many learning algorithms that do not explicitly manipulate probabilities.

Features of Bayesian learning methods include:

- 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

  (1) a prior probability for each candidate hypothesis, and

  (2) a probability distribution over observed data for each possible hypothesis.
- Bayesian methods can accommodate hypotheses that make probabilistic predictions (e.g., hypotheses such as "this pneumonia patient has a 93% chance of complete recovery").
- 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.

BAYES THEOREM

P(h) to denote the initial probability that hypothesis h holds, before we have observed the training data. P(h) is often called the prior probability of h.

P(D) to denote the prior probability that training data D will be observed (i.e., the probability of D given no knowledge about which hypothesis holds).

P(D|h) to denote the probability of observing data D given some world in which hypothesis h holds.

P(xly) to denote the probability of x given y.

P(h|D) that h holds given the observed training data D. P (h 1 D) is called the posterior probability of h.

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)).

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

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

(Formula used in example)

Example :- (10 marks confirm)

S. No.	Outlook	Temperature	Humidity	Windy	Play on
1	Rainy	Hot	High	FALSE	No
2	Rainy	Hot	High	TRUE	No
3	Overcast	Hot	High	FALSE	Yes
4	Sunny	Mild	High	FALSE	Yes
5	Sunny	Cool	Normal	FALSE	Yes
6	Sunny	Cool	Normal	TRUE	No
7	Overcast	Cool	Normal	TRUE	Yes
8	Rainy	Mild	High	FALSE	No
9	Rainy	Cool	Normal	FALSE	Yes
10	Sunny	Mild	Normal	FALSE	Yes
11	Rainy	Mild	Normal	TRUE	Yes
12	Overcast	Mild	High	TRUE	Yes
13	Overcast	Hot	Normal	FALSE	Yes
14	Sunny	Mild	High	TRUE	No

Outlook

	Yes	No	Probability of Yes	Probability of No
Rainy	2	3		
Sunny	3	2		
Overcast	4	0		
Total	9	5	100	100

Temperature

	Yes	No	Probability of Yes	Probability of No
Hot	2	2		
Mild	3	1		
Cool	4	2		
Total	9	5	100	100

Humidity

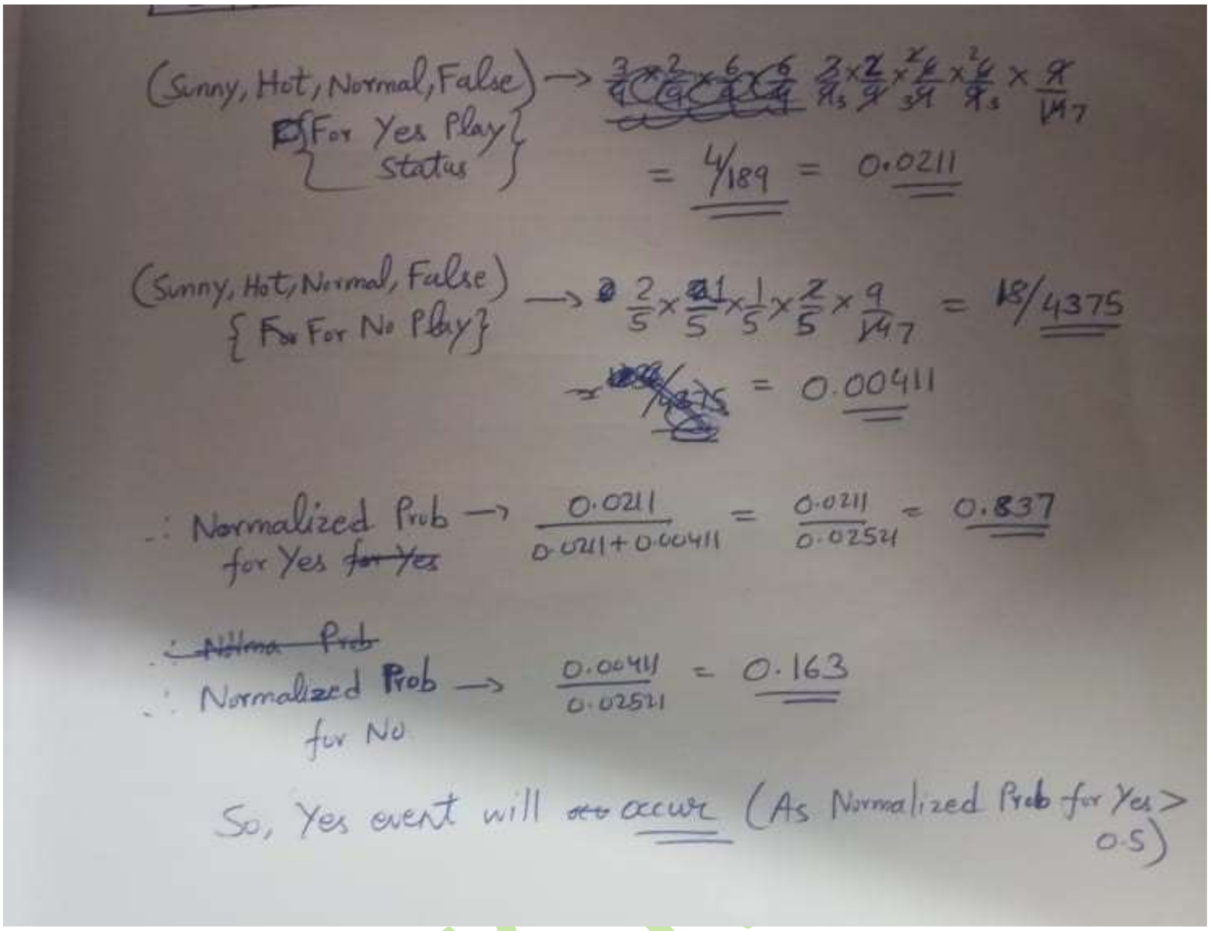
	Yes	No	Probability of Yes	Probability of No
High	3	4		
Normal	6	1		
Total	9	5	100	100

Windy

	Yes	No	Probability of Yes	Probability of No
TRUE	6	2		
FALSE	3	3		
Total	9	5	100	100

Play On

	Number	Probability
Yes	9	(9/14)
No	5	(5/14)
Total	14	



(Sunny, Hot, Normal, False) {For Yes Play Status} $\rightarrow \cancel{\frac{3}{9} \times \frac{2}{9} \times \frac{6}{9} \times \frac{6}{9}}$ $\frac{\cancel{3}}{\cancel{9}_3} \times \frac{2}{\cancel{9}} \times \frac{\cancel{6}^2}{\cancel{9}_3} \times \frac{\cancel{6}^2}{\cancel{9}_3} \times \frac{\cancel{9}}{\cancel{14}7}$

$= \frac{4}{189} = 0.0211$

(Sunny, Hot, Normal, False) {~~For~~ For No Play} $\rightarrow \frac{2}{5} \times \frac{\cancel{2}1}{5} \times \frac{1}{5} \times \frac{2}{5} \times \frac{9}{\cancel{14}7} = \frac{18}{4375}$

$= 0.00411$

∴ Normalized Prob for Yes ~~for Yes~~ $\rightarrow \frac{0.0211}{0.0211 + 0.00411} = \frac{0.0211}{0.02521} = 0.837$

~~∴ Nlma Prob~~

∴ Normalized Prob for No $\rightarrow \frac{0.00411}{0.02521} = 0.163$

So, Yes event will ~~oto~~ occur (As Normalized Prob for Yes > 0.5)

Pattern Recognition and Machine Learning

The Curse of Dimensionality

Curse of Dimensionality describes the explosive nature of increasing data dimensions and its resulting exponential increase in computational efforts required for its processing and/or analysis.

As the dimensionality increases, the number of data points required for good performance of any machine learning algorithm increases exponentially.

Scatter plot of the oil flow data for input variables x6 and x7, in which red denotes the 'homogenous' class, green denotes the 'annular' class, and blue denotes the 'laminar' class. Our goal is to classify the new test point denoted by '×'.

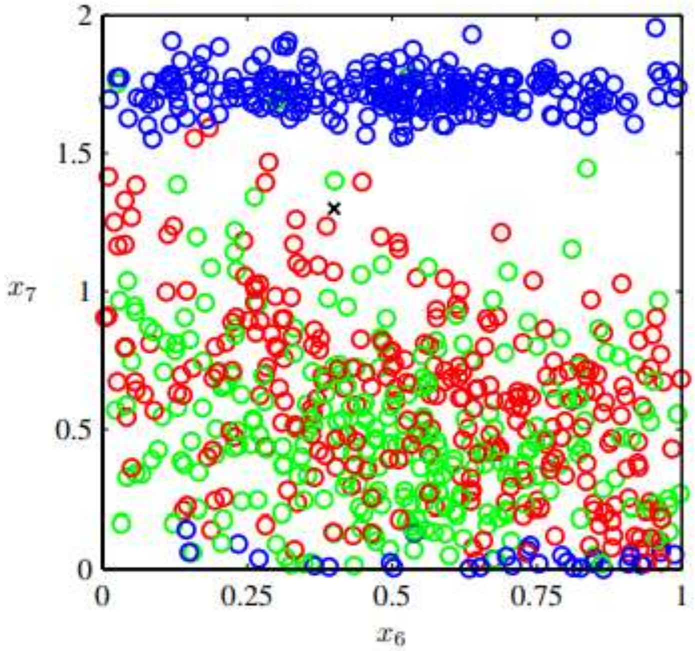
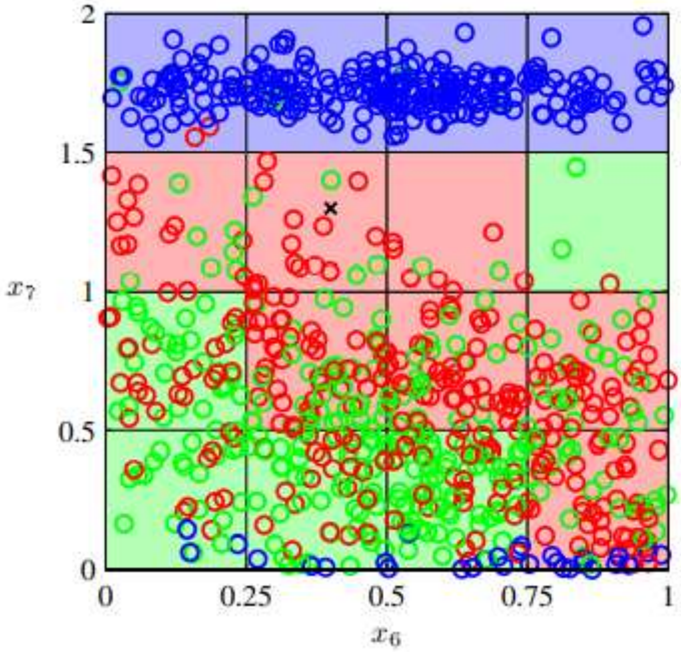


Illustration of a simple approach to the solution of a classification problem in which the input space is divided into cells and any new test point is assigned to the class that has a majority number of representatives in the same cell as the test point. As we shall see shortly, this simplistic approach has some severe shortcomings.



The problem with an exponentially large number of cells is that we would need an exponentially large quantity of training data in order to ensure that the cells are not empty. Clearly, we have no hope of applying such a technique in a space of more than a few variables, and so we need to find a more sophisticated approach

The Art and Science of Algorithms that Make Sense of Data

Kinds of feature

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

Calculations on features

Three main categories are **statistics of central tendency, statistics of dispersion** and **shape statistics.** Each of these can be interpreted either as a theoretical property of an unknown population or a concrete property of a given sample – here we will concentrate on sample statistics.

Starting with statistics of central tendency, the most important ones are:

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

Imagine a swimmer who swims the same distance d on two different days, taking a seconds one day and b seconds the next. On average, it took her therefore c = (a + b)/2 seconds, with an average speed of d/c = 2d/(a + b). Notice how this average speed is not calculated as the normal or **arithmetic mean** of the speeds, which would yield (d/a + d/b)/2: to calculate average speed over a fixed distance we use a different mean called the **harmonic mean**. Given two numbers x and y (in our swimming example these are the speeds on either day, d/a and d/b), the harmonic mean h is defined as

Since 1/h(x, y) = (1/x + 1/y)/2, we observe that calculating the harmonic mean on a scale with unit u corresponds to calculating the arithmetic mean on the **reciprocal scale** with unit 1/u.

In the example, speed with fixed distance is expressed on a scale reciprocal to the time scale, and since we use the arithmetic mean to average time, we use the harmonic mean to average speed. (If we average speed over a fixed time interval this is expressed on the same scale as distance and thus we would use the arithmetic mean.)

A good example of where the harmonic mean is used in machine learning arises when we average precision and recall of a classifier. Remember that precision is the proportion of positive predictions that is correct (prec = TP/(TP + FP)), and recall is the proportion of positives that is correctly predicted (rec = TP/(TP + FN)). Suppose we first calculate the number of mistakes averaged over the classes: this is the arithmetic mean Fm = (FP + FN)/2. We can then derive

We recognise the last term as the harmonic mean of precision and recall. Since the enumerator of both precision and recall is fixed, taking the arithmetic mean of the denominators corresponds to taking the harmonic mean of the ratios. In information retrieval this harmonic mean of precision and recall is very often used and called the **F-measure**.

Yet other means exist for other scales. In music, going from one note to a note one octave higher corresponds to doubling the frequency. So frequencies f and 4f are two octaves apart, and it makes sense to take the octave in between with frequency 2f as their mean. This is achieved by the **geometric mean**, which is defined as *

Since * it follows that the geometric mean corresponds to the arithmetic mean on a logarithmic scale. All these means have in common that the mean of two values is an interediate value, and that they can easily be extended to more than two values.

The second kind of calculation on features are statistics of **dispersion** or **'spread'**. Two well-known statistics of dispersion are the **variance** or **average** squared deviation from the (arithmetic) mean, and its square root, the **standard deviation**.

Variance and **standard deviation** essentially measure the same thing, but the latter has the advantage that it is expressed on the same scale as the feature itself. **For example,** the variance of the body weight in kilograms of a group of people is measured in kg 2 (kilograms-squared), whereas the standard deviation is measured in kilograms. The absolute difference between the mean and the median is never larger than the standard deviation – this is a consequence of **Chebyshev's inequality**, which states that at most 1/k 2 of the values are more than k standard deviations away from the mean.

A simpler dispersion statistic is the difference between maximum and minimum value, which is called the **range**. A natural statistic of central tendency to be used with the range is the **midrange point**, which is the mean of the two extreme values. These definitions assume a linear scale but can be adapted to other scales using suitable transformations. **For example**, for a feature expressed on a logarithmic scale, such as frequency, we would take the ratio of the highest and lowest frequency as the range, and the harmonic mean of these two extremes as the midrange point.

Other statistics of dispersion include **percentiles**. The p-th percentile is the value such that p per cent of the instances fall below it. If we have 100 instances, the 80th percentile is the value of the 81st instance in a list of increasing values. If p is a multiple of 25 the percentiles are also called **quartiles**, and if it is a multiple of 10 the percentiles are also called **deciles**. Note that the 50th percentile, the 5th decile and the second quartile are all the same as the median. **Percentiles**, **deciles** and **quartiles** are special cases of **quantiles**. Once we have quantiles we can measure dispersion as the distance between different quantiles. For instance, the **interquartile range** is the difference between the third and first quartile (i.e., the 75th and 25th percentile).

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

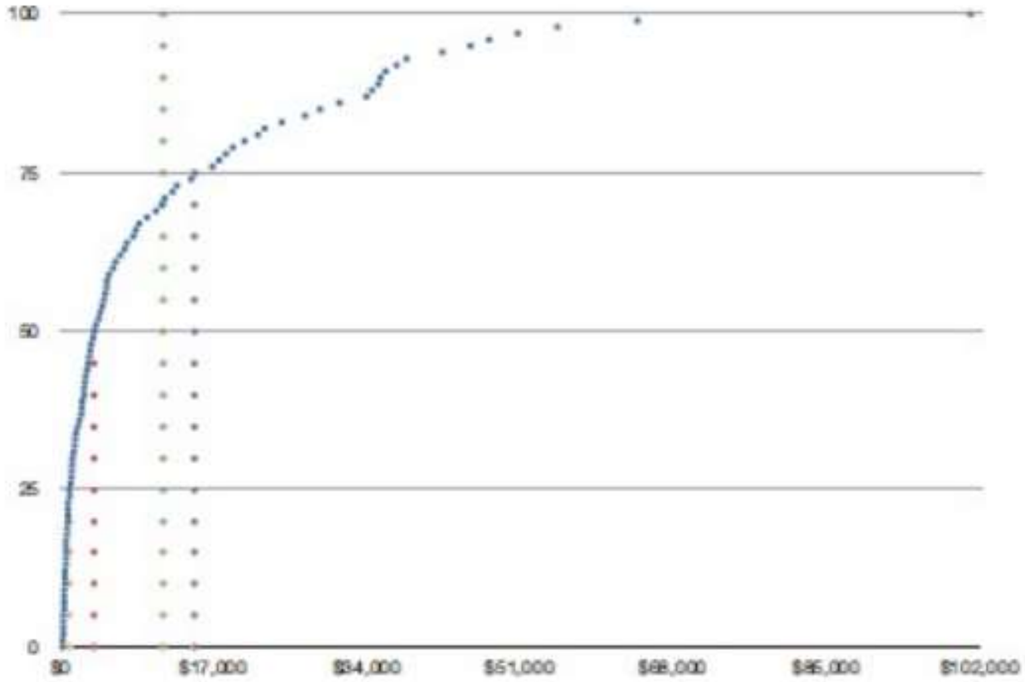


Figure 10.1. Percentile plot of GDP per capita for 231 countries (data obtained from www.wolframalpha.com by means of the query ‘GDP per capita’). The vertical dotted lines indicate, from left to right: the **first quartile** ($900); the **median** ($3600); the **mean** ($11284); and the **third quartile** ($14,750). The **interquartile range** is $13 850, while the **standard deviation** is $16,189.

Categorical, ordinal and quantitative features

Features with an ordering but without scale are called **ordinal features.**

Features with a meaningful numerical scale are called **quantitative**.

Features without ordering or scale are called **categorical features** (or sometimes ‘nominal’ features). They do not allow any statistical summary except the mode. One subspecies of the categorical features is the **Boolean feature**, which maps into the truth values **true** and **false**.

Feature transformations

Feature transformations aim at improving the utility of a feature by removing, changing or adding information. We could order feature types by the amount of detail they convey: quantitative features are more detailed than ordinal ones, followed by categorical features, and finally Boolean features.

Binarisation transforms a categorical feature into a set of Boolean features, one for each value of the categorical feature. This loses information since the values of a single categorical feature are mutually exclusive, but is sometimes needed if a model cannot handle more than two feature values.

Unordering trivially turns an ordinal feature into a categorical one by discarding the ordering of the feature values. This is often required since most learning models cannot handle ordinal features directly.

Thresholding transforms a quantitative or an ordinal feature into a Boolean feature by finding a feature value to split on.

Without knowing how this feature is to be used in a model, the most sensible thresholds are the statistics of central tendency such as the mean and the median. This is referred to as **unsupervised thresholding**.

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

Unsupervised discretisation methods typically require one to decide the number of bins beforehand. A simple method that often works reasonably well is to choose the bins so that each bin has approximately the same number of instances: this is referred to as **equal-frequency discretisation**.

Another unsupervised discretisation method is **equal-width discretisation**, which chooses the bin boundaries so that each interval has the same width.

Switching now to **supervised discretisation** methods, we can distinguish between **top–down** or **divisive** discretisation methods on the one hand, and **bottom–up** or **agglomerative** discretisation methods on the other. Divisive methods work by progressively splitting bins, whereas agglomerative methods proceed by initially assigning each instance to its own bin and successively merging bins. In either case an important role is played by the **stopping criterion**, which decides whether a further split or merge is worthwhile.

Feature normalisation is often required to neutralise the effect of different quantitative features being measured on different scales. If the features are approximately normally distributed, we can convert them into **z-scores** by centring on the mean and dividing by the standard deviation.

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

Feature construction and selection

Forward selection methods start with an empty set of features and add features to the set one at a time, as long as they improve the performance of the model. **Backward elimination** starts with the full set of features and aims at improving performance by removing features one at a time.

One of the best-known algebraic feature construction methods is **principal component analysis (PCA)**. Principal components are new features constructed as linear combinations of the given features. The first principal component is given by the direction of maximum variance in the data; the second principal component is the direction of maximum variance orthogonal to the first component, and so on.

PCA can be explained in a number of different ways: here, we will derive it by means of the **singular value decomposition (SVD)**. Any n-by-d matrix can be uniquely written as a product of three matrices with special properties:

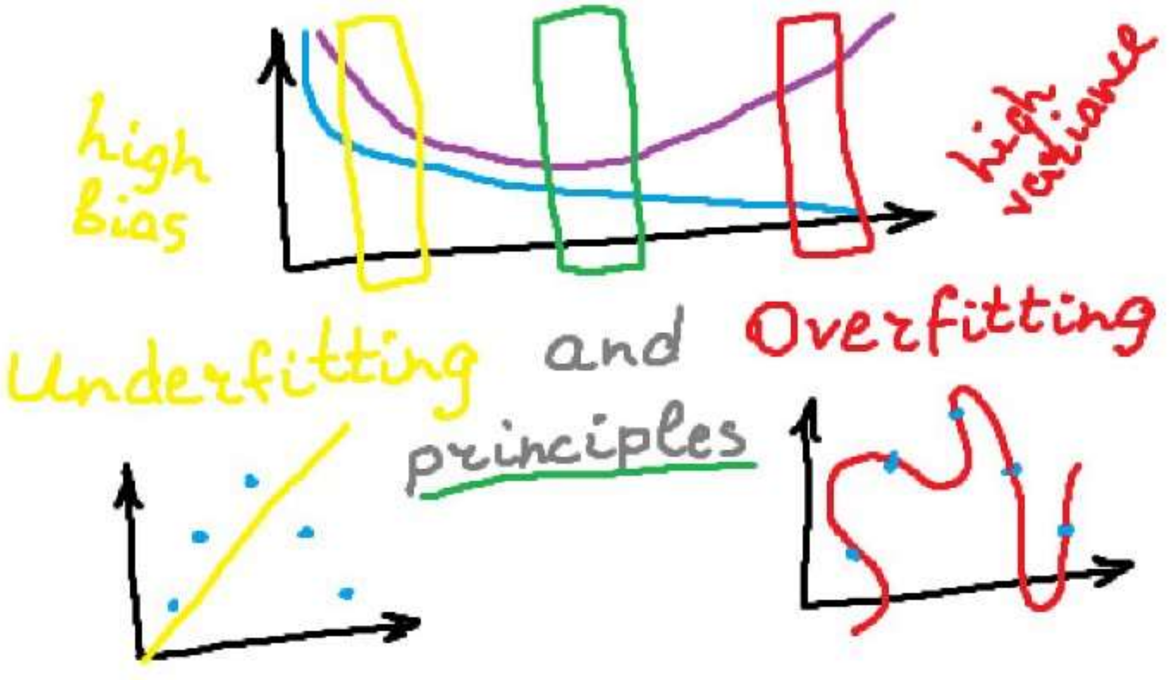
$$X = U\Sigma V^{T}$$

Here, U is an n-by-r matrix, Σ is an r-by-r matrix and V is an d-by-r matrix (for the moment we will assume r = d < n). Furthermore, U and V are orthogonal (hence rotations) and Σ is diagonal (hence a scaling). The columns of U and V are known as the left and right singular vectors, respectively; and the values on the diagonal of Σ are the corresponding singular values. It is customary to order the columns of V and U so that the singular values are decreasing from top-left to bottom-right.

https://towardsdatascience.com/overfitting-and-underfitting-principles-ea8964d9c45c

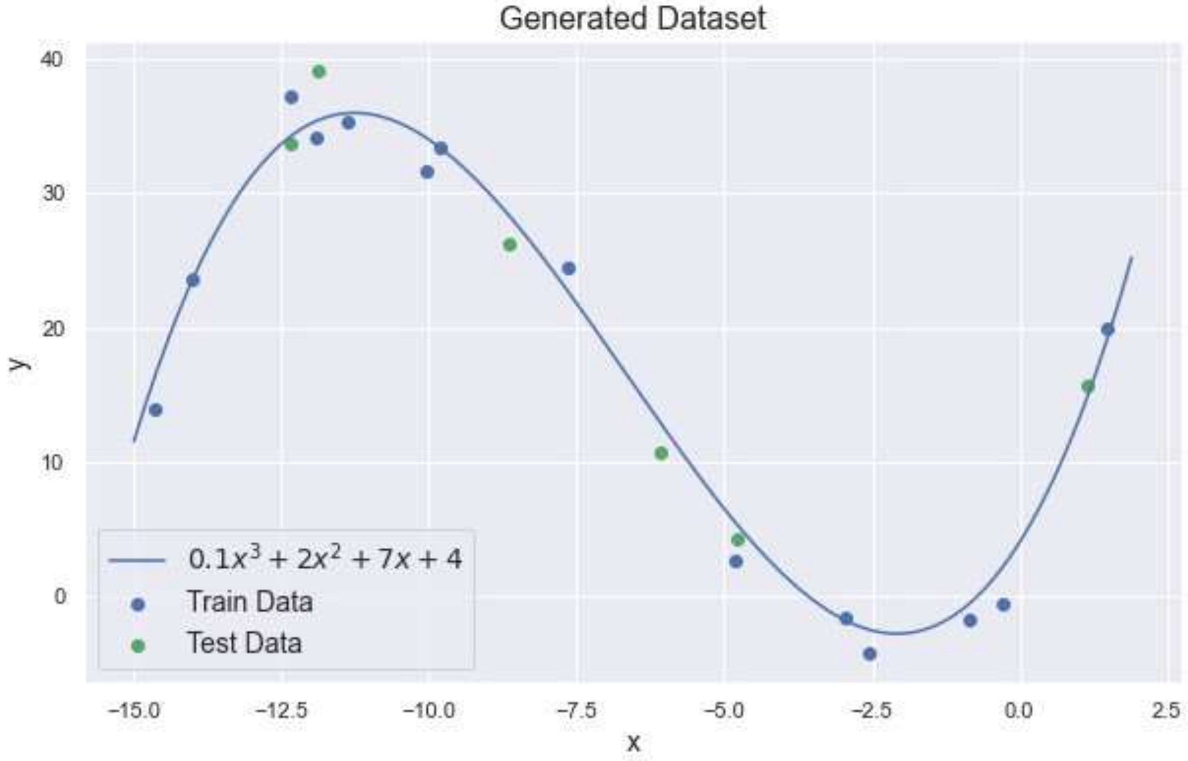
Overfitting and Underfitting Principles

Understand basic principles of underfitting and overfitting and why you should use particular techniques to **deal with them**



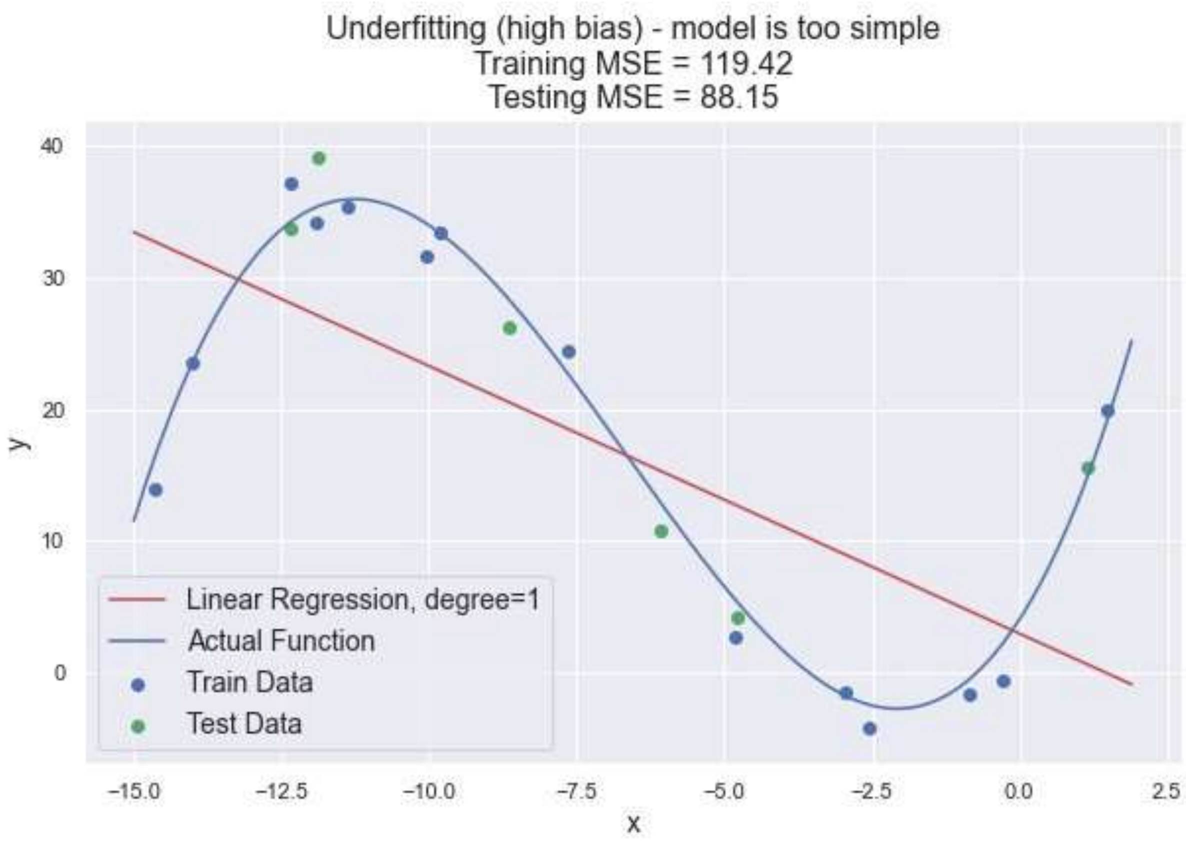
Underfitting and overfitting principles. Image by Author

Underfitting and Overfitting and Bias/Variance Trade-off



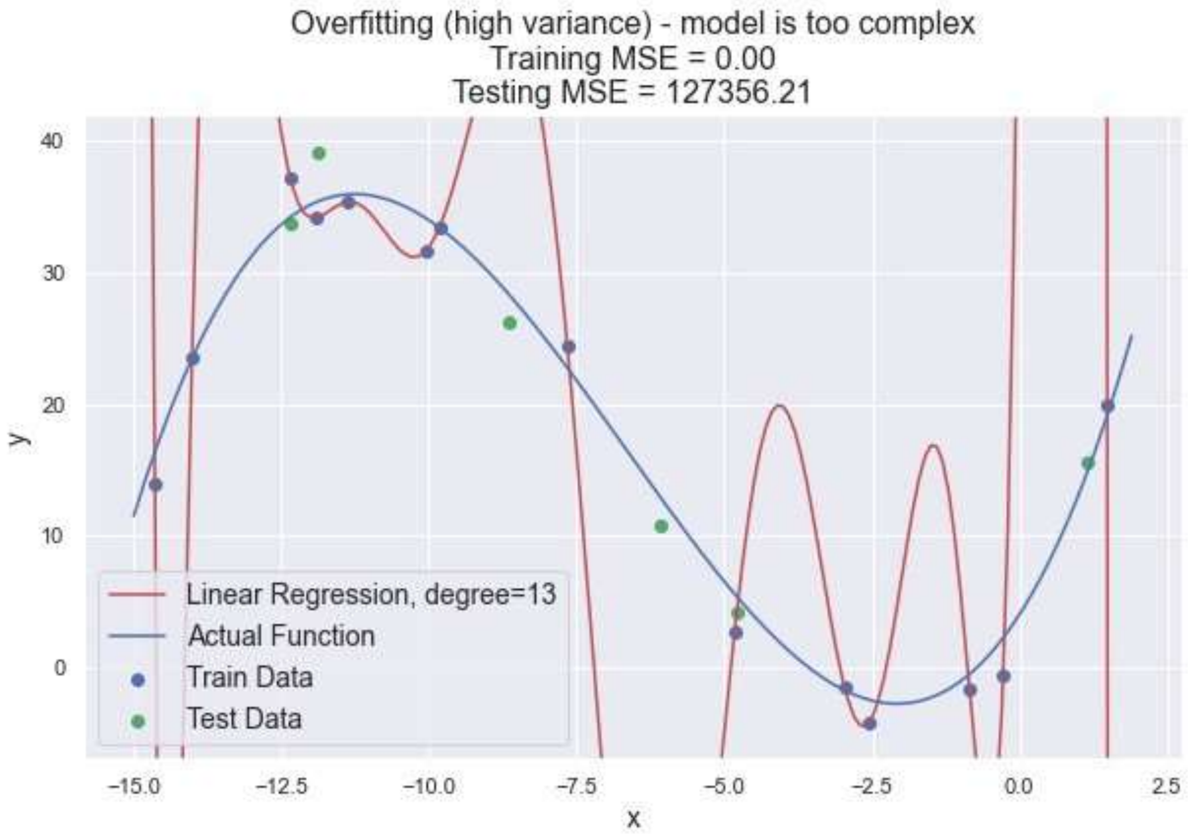
Generated dataset. Image by Author

Underfitting is a situation when your model is **too simple** for your data. More formally, your hypothesis about data distribution is wrong and too simple — for example, your data is quadratic and your model is linear. This situation is also called **high bias**. This means that your algorithm can do accurate predictions, but the initial assumption about the data is incorrect.



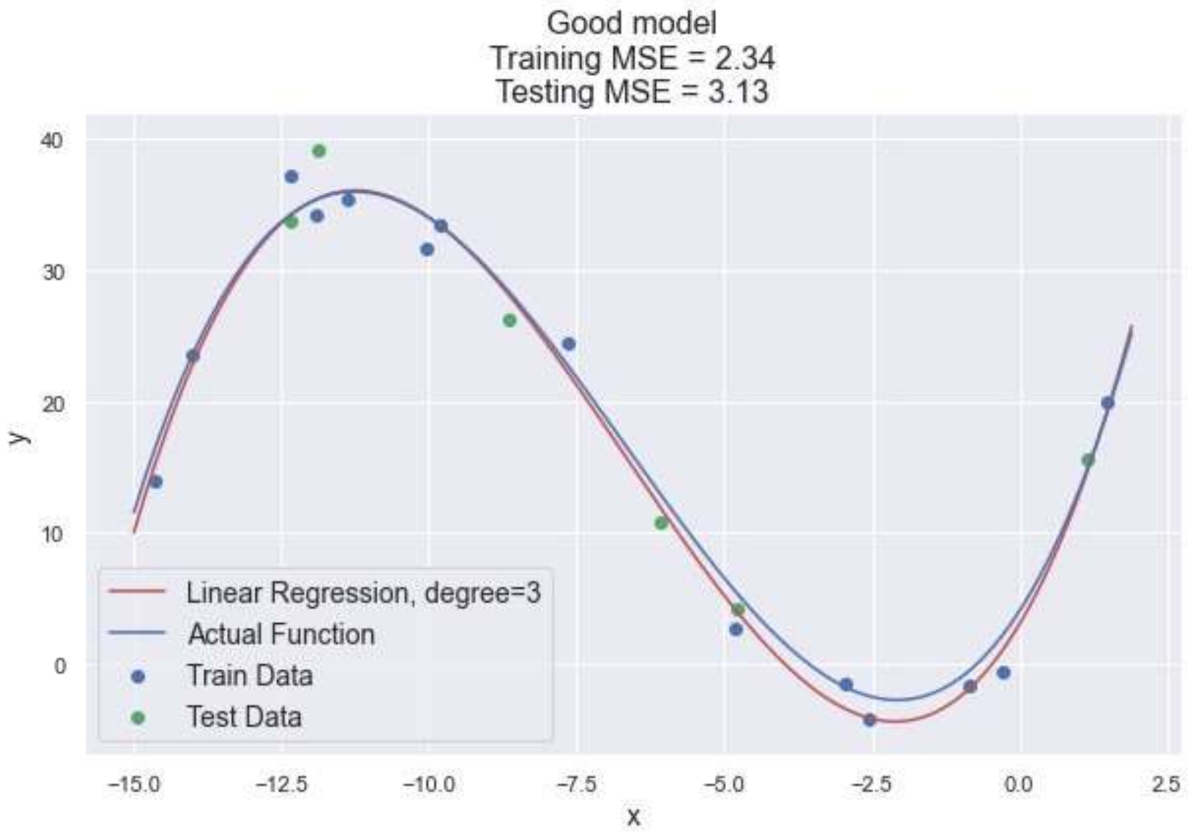
Underfitting. The linear model trained on cubic data. Image by Author

Opposite, **overfitting** is a situation when your model is **too complex** for your data. More formally, your hypothesis about data distribution is wrong and too complex — for example, your data is linear and your model is high-degree polynomial. This situation is also called **high variance**. This means that your algorithm can’t do accurate predictions — changing the input data only a little, the model output changes very much.



Overfitting. The 13-degree polynomial model trained on cubic data. Image by Author

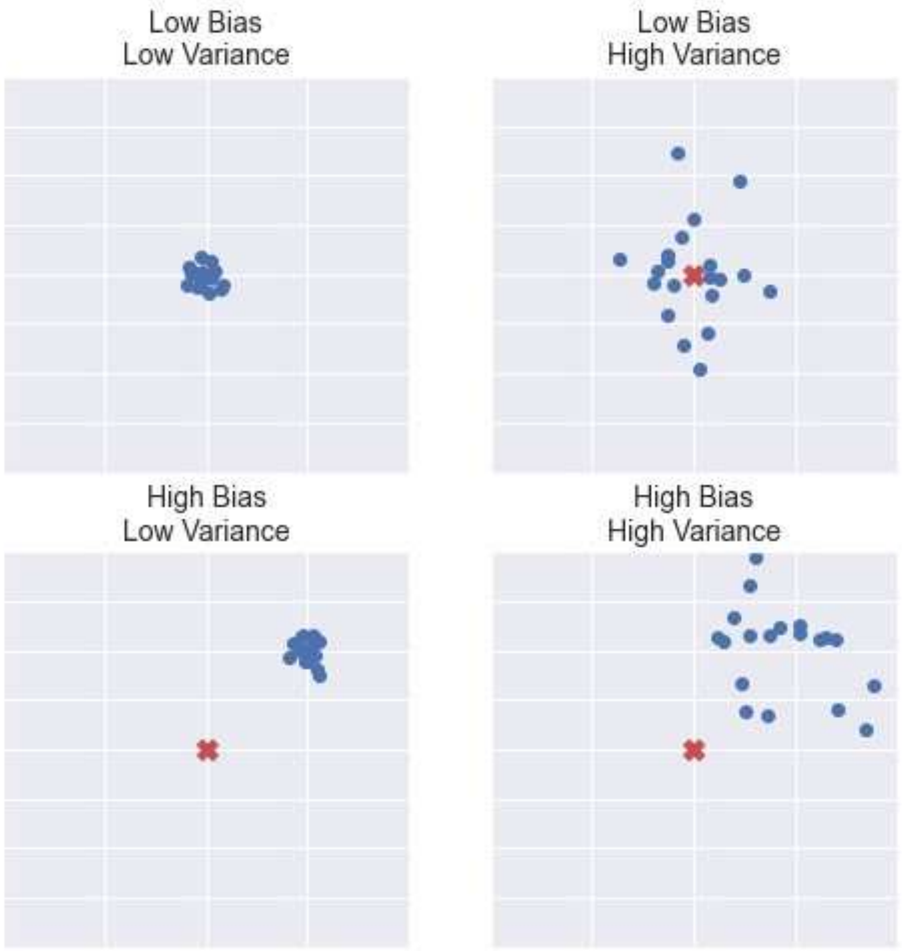
These are two extremes of the same problem and the optimal solution always lies somewhere in the middle.



Good model. The cubic model trained on cubic data. Image by Author

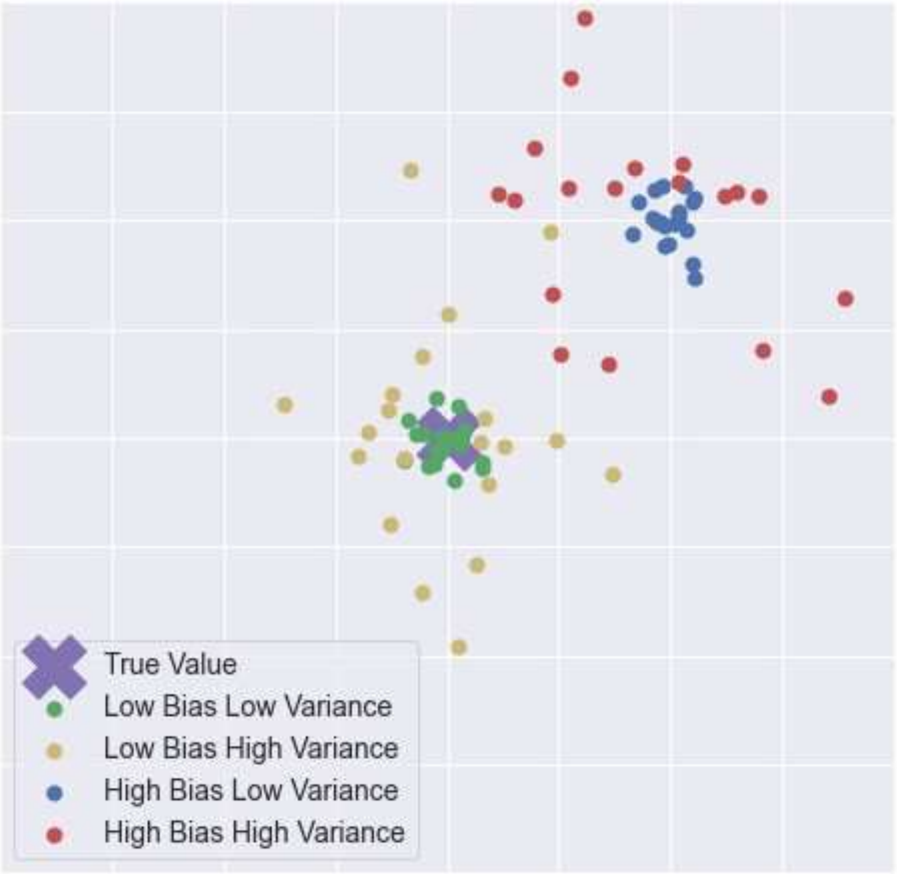
I will not talk much about bias/variance trade-off (you can read the basics in this article), but let me briefly mention possible options:

- low bias, low variance — is a good result, just right.
- low bias, **high variance** — **overfitting** — the algorithm outputs very different predictions for similar data.
- **high bias**, low variance — **underfitting** — the algorithm outputs similar predictions for similar data, but predictions are wrong (algorithm "miss").
- high bias, high variance — very bad algorithm. You will most likely never see this.



Bias and Variance options on four plots. Image by Author

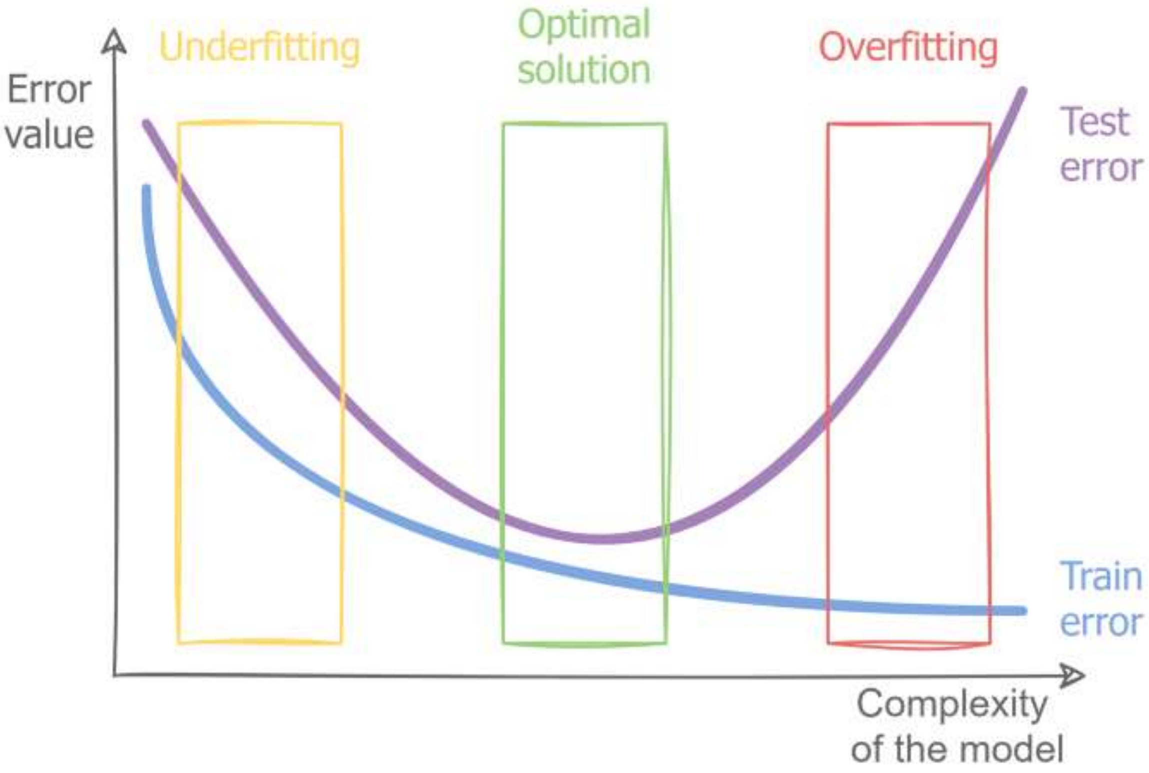
All these cases can be placed on the same plot. It is a bit less clear than the previous one but more compact.



Bias and Variance options on one plot. Image by Author

How to Detect Underfitting and Overfitting

Before we move on to the tools, let's understand how to "diagnose" underfitting and overfitting.



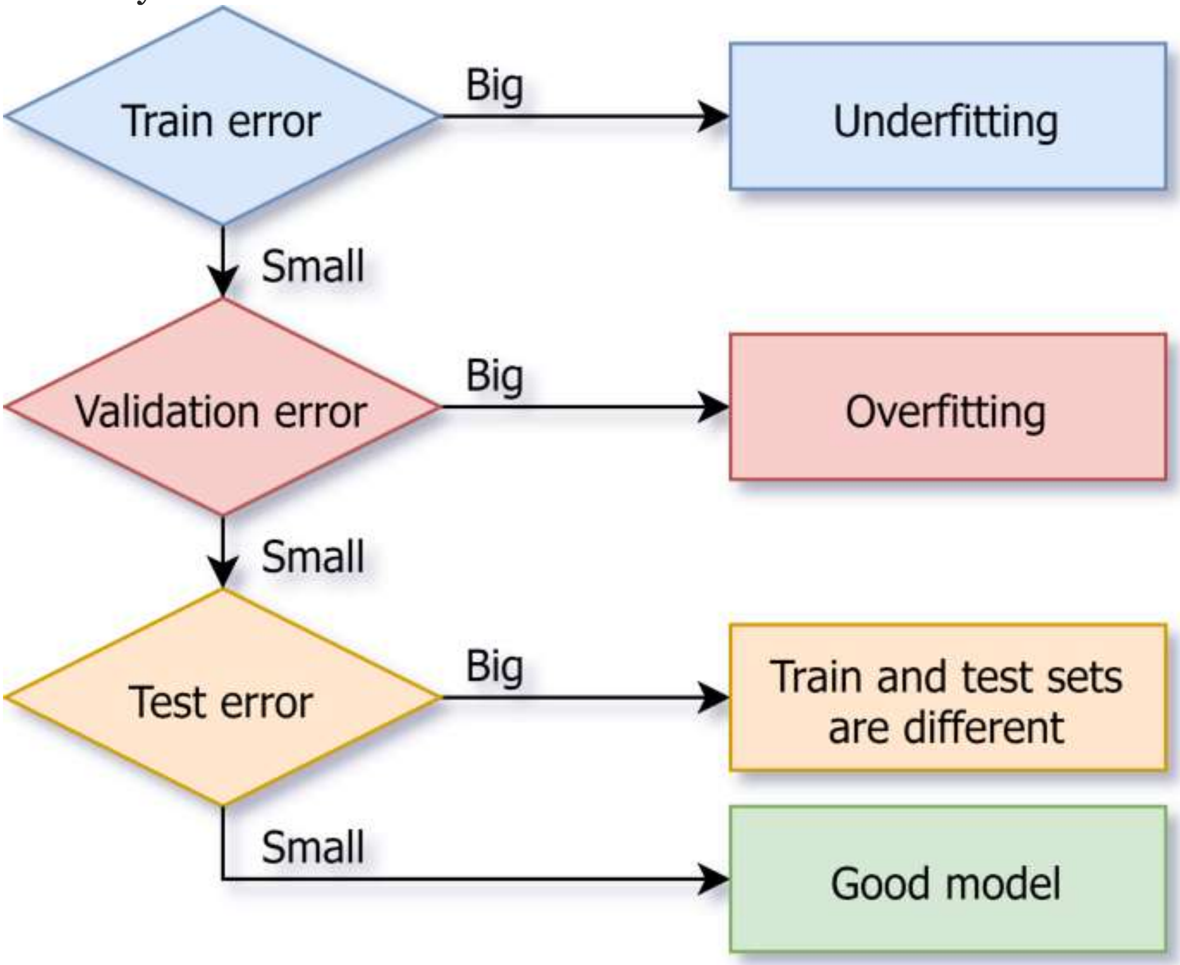
Train/test error and underfitting/overfitting. Image by Author

Underfitting means that your model makes accurate, but initially incorrect predictions. In this case, **train error is large** and **val/test error is large** too.

Overfitting means that your model makes not accurate predictions. In this case, **train error is very small** and **val/test error is large**.

When you find a **good model**, **train error is small** (but larger than in the case of overfitting), and **val/test error is small** too.

In the case above, the test error and validation error are approximately the same. This happens when everything is fine, and your *train, validation, and test data have the same distributions*. If validation and test error are very different, then you need to get more data similar to test data and make sure that you split the data correctly.



How to detect underfitting and overfitting. Image by Author

General Intuition You Should Remember

As we remember:

- **underfitting** occurs when your model is **too simple** for your data.
- **overfitting** occurs when your model is **too complex** for your data.

Based on this, simple intuition you should keep in mind is:

- to fix **underfitting**, you should **complicate** the model.
- to fix **overfitting**, you should **simplify** the model.

More Simple / Complex Model

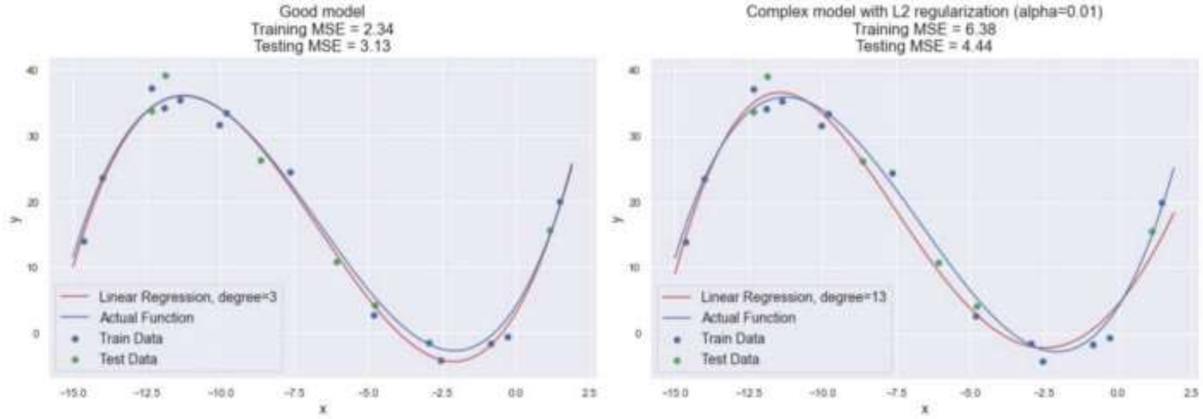
The easiest way that comes to mind based on the intuition above is to try a more simple or more complex algorithm (model).

To complicate the model, you need to add more parameters (*degrees of freedom*). Sometimes this means to directly try a more powerful model — one that is a priori capable to restore more complex dependencies (*SVM with different kernels instead of logistic regression*). **If the algorithm is already quite complex** (neural network or some ensemble model), **you need to add more parameters** to it, for example, increase the number of models in boosting. In the context of neural networks, this means adding more layers / more neurons in each layer / more connections between layers / more filters for CNN, and so on.

To simplify the model, you need contrariwise to reduce the number of parameters. Either completely change the algorithm (*try random forest instead of deep neural network*), or reduce the number of degrees of freedom. Fewer layers, fewer neurons, and so on.

More Regularization / Less Regularization

This point is very closely related to the previous one. In fact, **regularization is an indirect and forced simplification of the model**. The regularization term requires the model to keep parameters values as small as possible, so requires the model to be as simple as possible. Complex models with strong regularization often perform better than initially simple models, so this is a very powerful tool.



Good model and complex model with regularization. Image by Author

More regularization (simplifying the model) means increasing the impact of the *regularization term*. This process is strictly individual — depending on the algorithm, the regularization parameters are different (for example, **to reduce the regularization, the alpha for Ridge regression should be decreased, and C for SVM — increased**). So you should study the parameters of the algorithm and pay attention to whether they should be increased or decreased in a particular situation. There are a lot of such parameters — L1/L2 coefficients for linear regression, C and gamma for SVM, maximum tree depth for decision trees, and so on. In the context of neural networks, the main regularization methods are:

- Early stopping,
- Dropout,
- L1 and L2 Regularization.

You can read about them in this article.

Opposite, in the case when the model needs to be complicated, you should reduce the influence of regularization terms or abandon the regularization at all and see what happens.

More Features / Fewer Features

This may not be so obvious, but adding new features also complicates the model. Think about it in the context of a *polynomial regression* — adding quadratic features to a dataset allows a linear model to recover quadratic data.

Adding new “natural” features (if you can call it that) — obtaining new features for existing data is used infrequently, mainly due to the fact that it is very expensive and long. But you can keep in mind that sometimes this can help.

But artificial obtaining of additional features from existing ones (the so-called *feature engineering*) is used quite often for classical machine learning models. There are as many examples of such transformations as you can imagine, but here are the main ones:

- polynomial features — from x_1, x_2 to $x_1, x_2, x_1x_2, x_1^2, x_2^2$, ... (`sklearn.preprocessing.PolynomialFeatures` class)

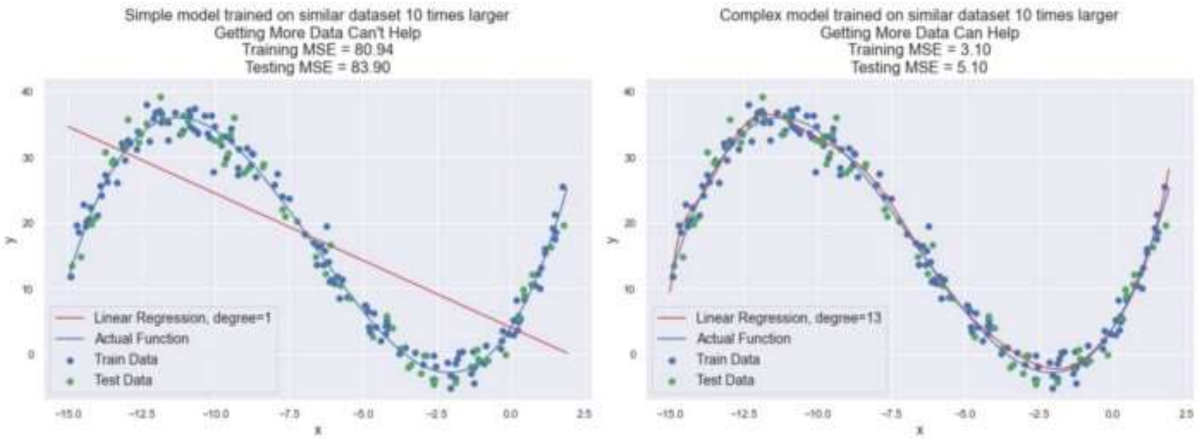
- log(x), for data with not-normal distribution
- ln(|x| + 1) for data with heavy right tail
- transformation of categorical features
- other non-linear data transformation (from length and width to area (length*width)) and so on.

If you need to simplify the model, then you should use a smaller quantity of features. First of all, remove all the additional features that you added earlier if you did so. But it may turn out that in the original dataset there are features that do not carry useful information, and sometimes cause problems. Linear models often work worse if some features are dependent — *highly correlated*. In this case, you need to use *feature selection* approaches to select only those features that carry the maximum amount of useful information.

It is worthwhile to say that in the context of neural networks, *feature engineering* and *feature selection* make almost no sense because **the network finds dependencies in the data itself**. This is actually why deep neural networks can restore such complex dependencies.

Why Getting More Data Sometimes Can't Help

One of the techniques to combat overfitting is to get more data. However, surprisingly, this may not always help. Let's generate a similar dataset 10 times larger and train the same models on it.

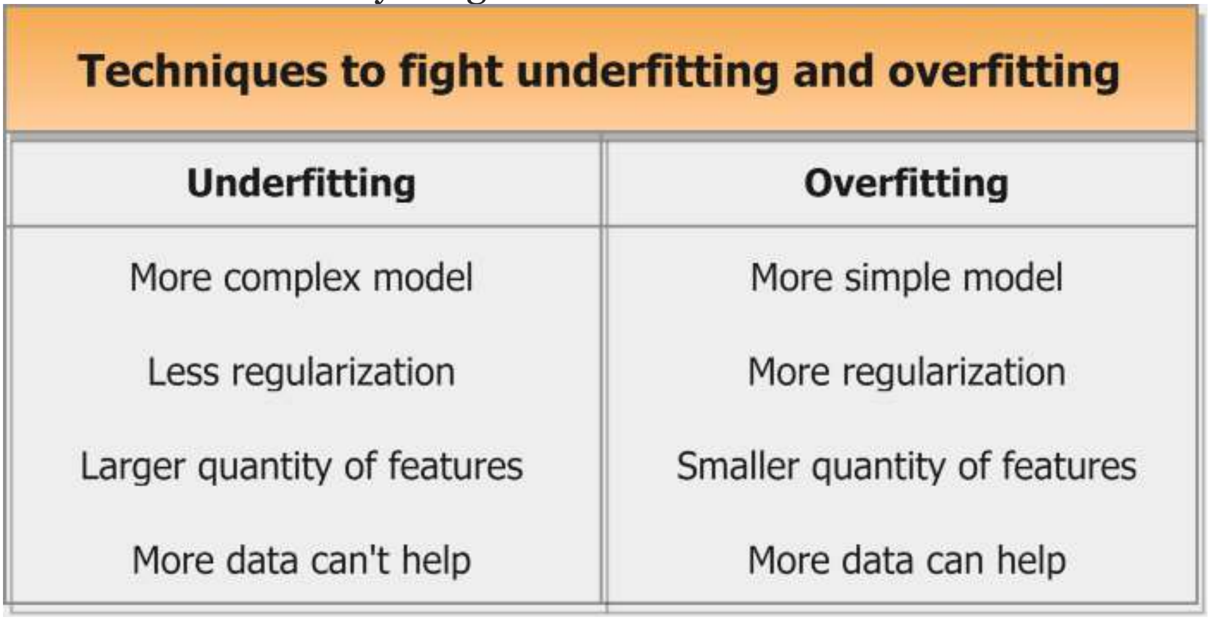


Why getting more data sometimes can’t help. Image by Author

A very simple model (degree 1) has remained simple, almost nothing has changed. So **getting more data will not help in case of underfitting**.

Summary

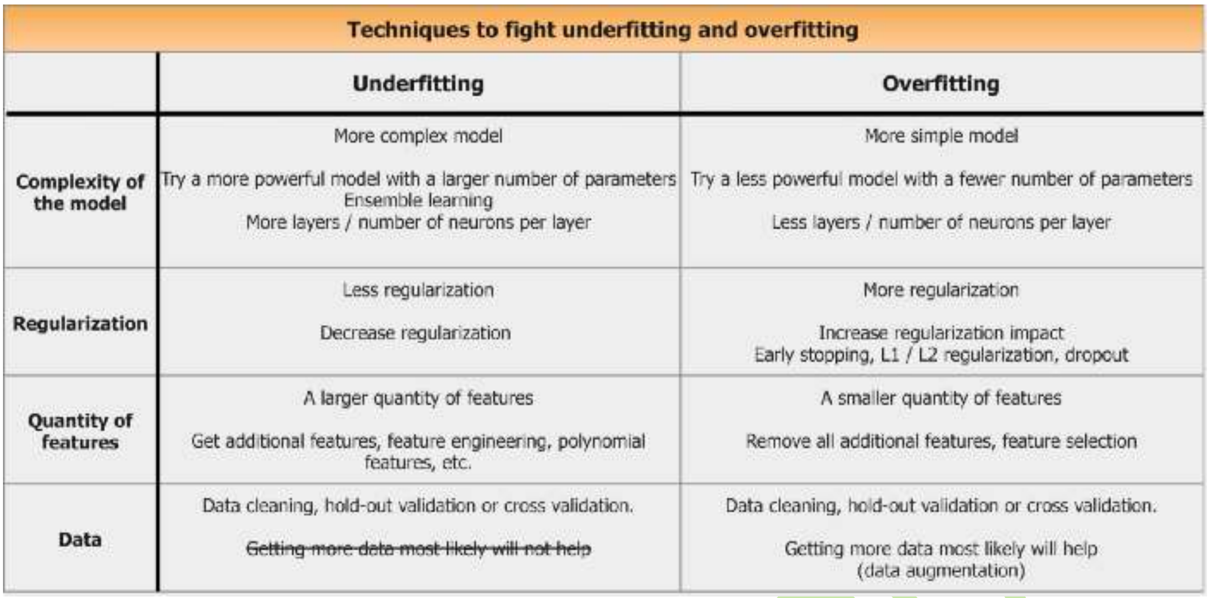
Let’s summarize everything in one table.



Techniques to fight underfitting and overfitting	
Underfitting	**Overfitting**
More complex model	More simple model
Less regularization	More regularization
Larger quantity of features	Smaller quantity of features
More data can't help	More data can help

Techniques to fight underfitting and overfitting. Image by Author

Well, better in two.



Techniques to fight underfitting and overfitting

	Underfitting	Overfitting
Complexity of the model	More complex model Try a more powerful model with a larger number of parameters Ensemble learning More layers / number of neurons per layer	More simple model Try a less powerful model with a fewer number of parameters Less layers / number of neurons per layer
Regularization	Less regularization Decrease regularization	More regularization Increase regularization impact Early stopping, L1 / L2 regularization, dropout
Quantity of features	A larger quantity of features Get additional features, feature engineering, polynomial features, etc.	A smaller quantity of features Remove all additional features, feature selection
Data	Data cleaning, hold-out validation or cross validation. ~~Getting more data most likely will not help~~	Data cleaning, hold-out validation or cross validation. Getting more data most likely will help (data augmentation)

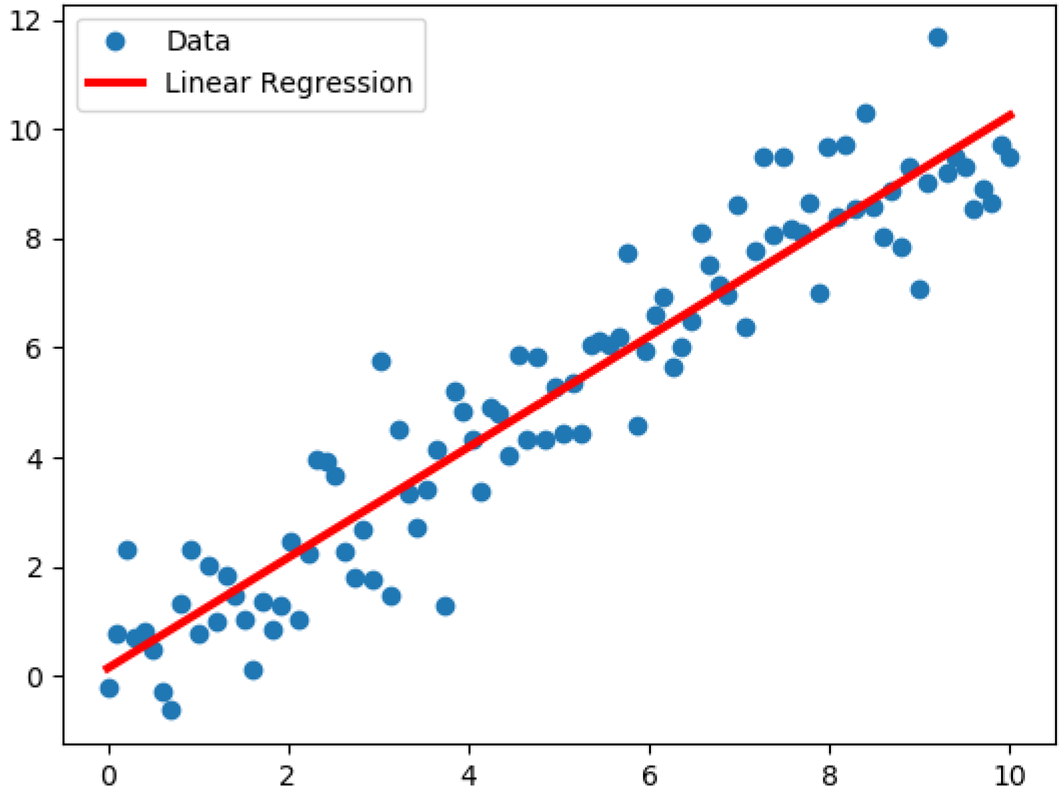
Techniques to fight underfitting and overfitting (extended). Image by Author

Some tools and techniques have not been covered in this article. For example, I consider **data cleaning** and **cross-validation** or **hold-out validation** to be common practices in any machine learning project, but they can also be considered as tools to combat overfitting.

You may notice that to eliminate underfitting or overfitting, you need to apply **diametrically opposite actions**. So if you initially "misdiagnosed" your model, you can spend a lot of time and money on empty work (for example, getting new data when in fact you need to complicate the model). That's why it is so important — hours of analysis can save your days and weeks of work.

https://medium.com/analytics-vidhya/linear-regression-with-gradient-descent-derivation-c10685ddf0f4

What is Linear Regression ?



Simple Linear Regression is basically a modelling of linear relationship between linearly dependent variables that can be later used to predict dependent variable values for new independent variables

For this, we use the equation of the line : `y = m * x + c.`

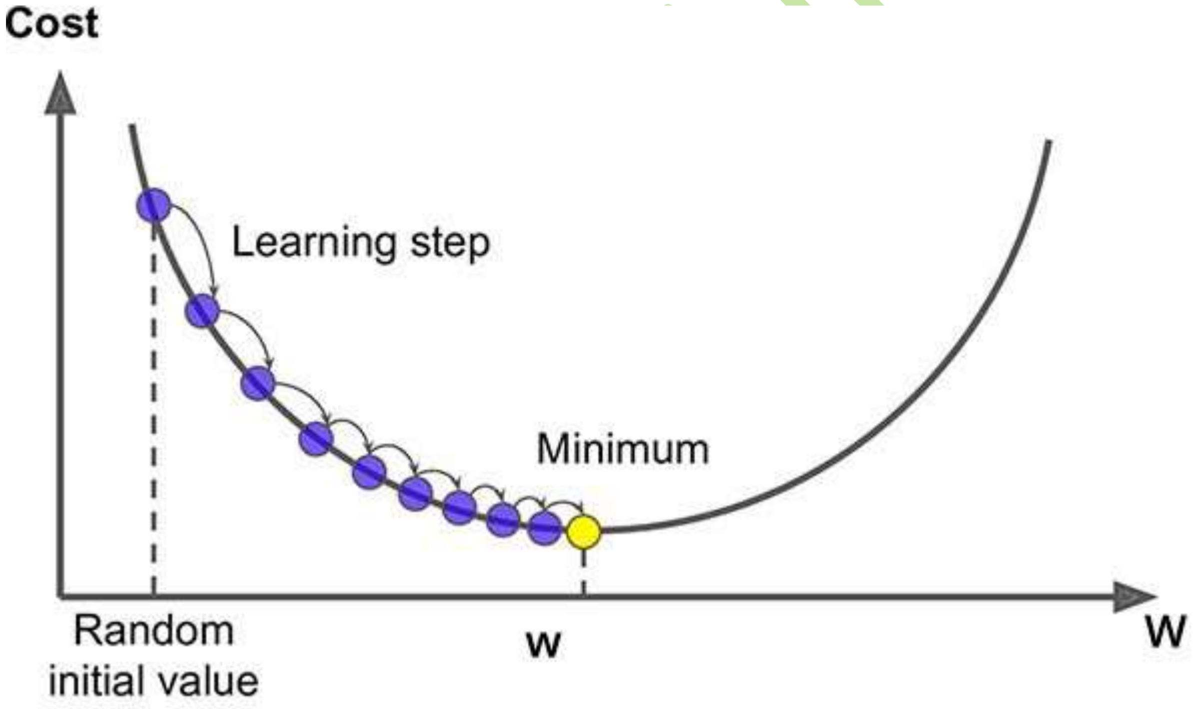
where y is the dependent variable and x is the independent variable

For example: We could predict the salary of a person with the years of experience of the person.

Here, Salary is the dependent variable and experience is the independent variable since, we are predicting salary with the help of experience.

When to use Linear Regression: Linear regression can be performed on data where there is a good linear relationship between dependent and independent variables.
The degree of linear relationship can be found with help of correlation.

Gradient Descent



Gradient Descent is a an optimization algorithm that can be used to find the global or local minima of a differentiable function.

Various Assumptions and definitions before we begin.

Let’s say we have an independent variable x and a dependent variable y.
in order to form the relationship between these 2 variables, we have the equation:

```
y = x * w + b
```

where **w** is weight (or slope) ,
b is the bias (or intercept),
x is the independent variable column vector(examples),
y is the dependent variable column vector(examples)
Our main goal is to find the **w** and **b** that defines the relationship between variable **x** and **y** correctly. We accomplish this with the help of something called as the **Loss Function.**

Loss Function: A loss function is a function that signifies how much our predicted values is deviated from the actual values of the dependent variable.

Important Note: *we are trying to find the values for **w** and **b** such that it minimizes our loss function.*

Steps Involved in Linear Regression with Gradient Descent Implementation

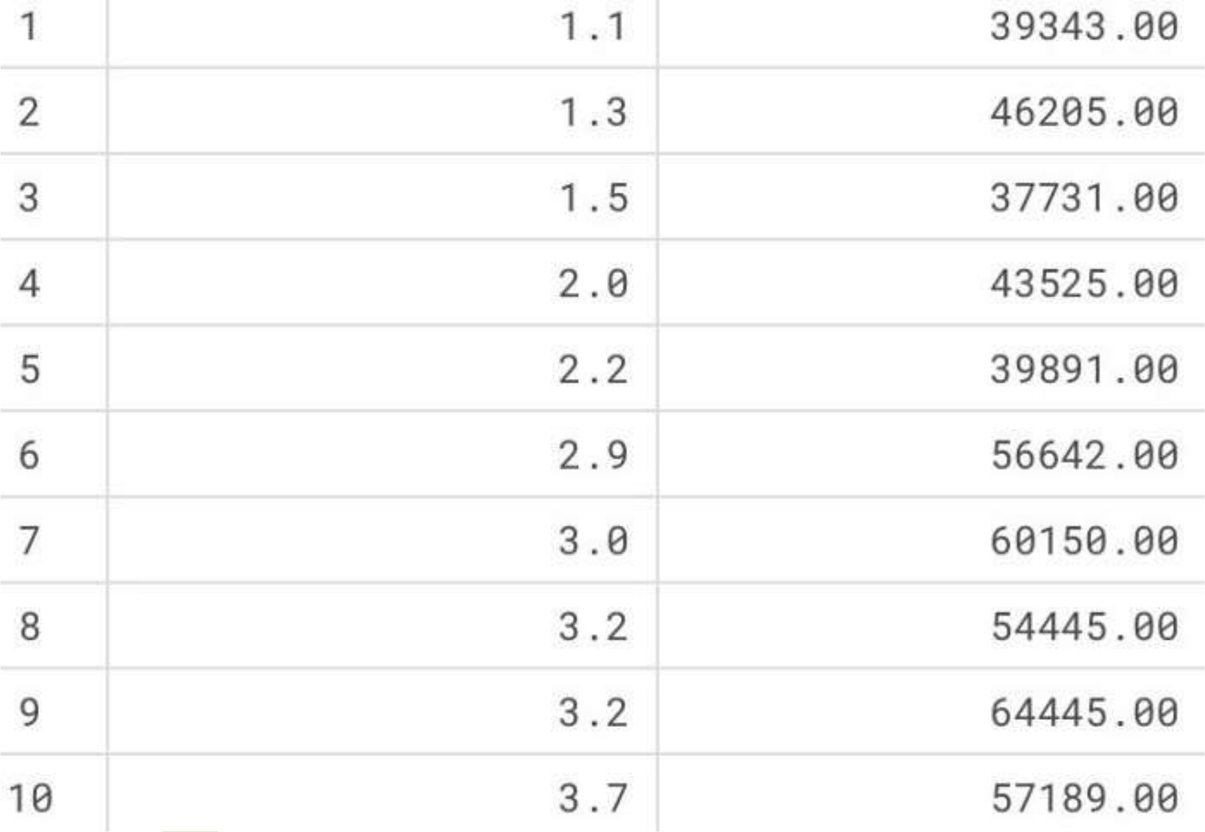
1. Initialize the weight and bias randomly or with 0(both will work).
2. Make predictions with this initial weight and bias.

3. Compare these predicted values with the actual values and define the loss function using both these predicted and actual values.

4. With the help of differentiation, calculate how loss function changes with respect to weight and bias term.

5. Update the weight and bias term so as to minimize the loss function.

Implementation with Math



1	1.1	39343.00
2	1.3	46205.00
3	1.5	37731.00
4	2.0	43525.00
5	2.2	39891.00
6	2.9	56642.00
7	3.0	60150.00
8	3.2	54445.00
9	3.2	64445.00
10	3.7	57189.00

Example of Independent and dependent variables respectively

1 . Assumption

Let us say we have an **x** and **y** vectors like shown in the above pic (*The above one is only for an example*).

2. Initialize w and b to 0

w = 0, **b** = 0

3. Make some predictions with the current w and b. Of course, it’s going to be wrong.

y_pred = x*w + b, where y_pred stands for predicted y values.
This y_pred will also be a vector like y.

4 . Define a loss function

loss = (y_pred — y)^{2}/n

where n is the number of examples in the dataset.
It is obvious that this loss function represents the deviation of the predicted values from the actual.
This loss function will also be a vector. But, we will be summing all the elements in that vector to convert it into scalar.

5. Calculate (∂(loss)/ ∂w)

The derivative of a function of a real variable measures the sensitivity to change of the function value with respect to a change in its argument.

We can use calculus to find how loss changes with respect to ***w***.

```
loss = (y_pred — y)²/n
loss = (y_pred² + y² — 2y*y_pred)/n (expanding the whole square)
       =>( (x*w+b)² + y² — 2y*(x*w+b))/n (substitute y_pred)
       => ((x*w+b)²/n ) + (y²/n) + ((-2y(x*w+b))/n) (splitting
the terms)
Let A = ((x*w+b)²/n )
Let B = (y²/n),
Let C = ((-2y(x*w+b))/n)A = ( x²w² + b² + 2xwb )/n (expanding)
∂A/∂w = ( 2x²w + 2xb )/n (differentiating)
```

```
∂B/∂w = 0 (differentiating)
C = (-2yxw — 2yb)/n
∂C/∂w = (-2yx)/n (differentiating)
```

So, ∂loss/∂w will be the addition of all these terms:

```
∂loss/∂w = (2x²w + wxb — 2yx)/n
=> (2x(x*w + b — y))/n
```

So, The derivative of loss with respect to ***w*** was found to be:

`(2/n)*(y_pred — y)*x`. Let us call this dw.

If we perform the same differentiation for loss with respect to ***b***, we'll get:

`(2/n)*(y_pred — y)`. Let us call this db.

This dw and db are what we call *"gradients"*

6. Update w and b

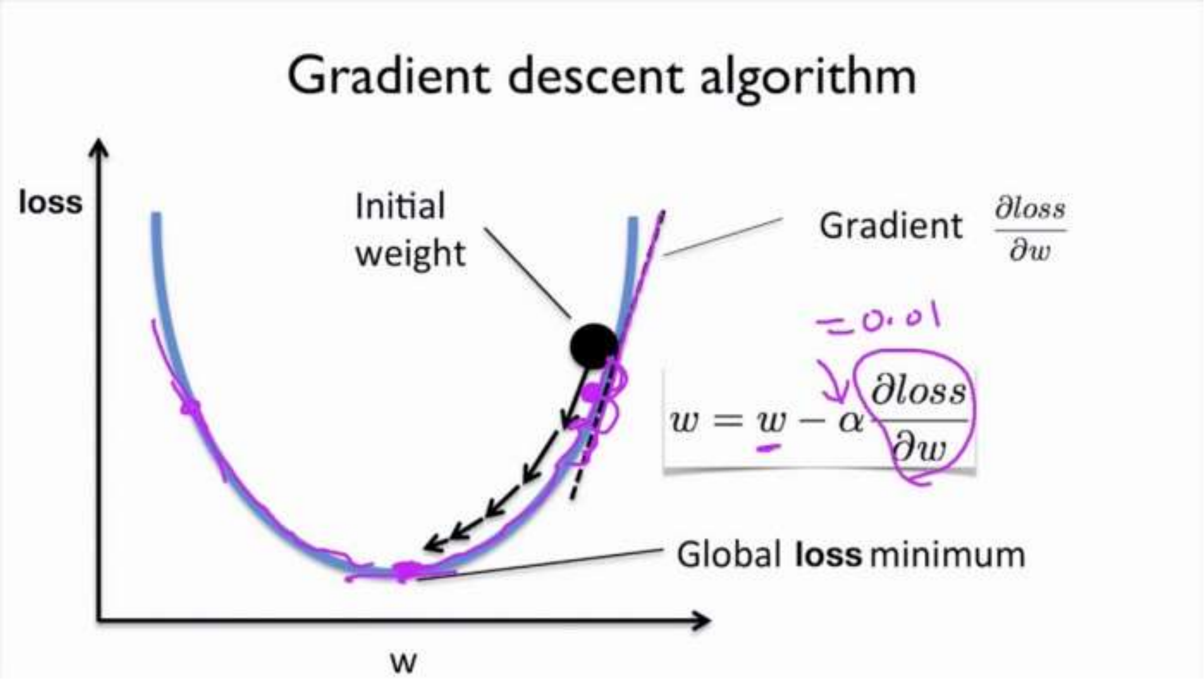

Gradient descent algorithm
loss
Initial weight
Gradient $\frac{\partial loss}{\partial w}$
=0.01
$w = w - \alpha \frac{\partial loss}{\partial w}$
Global loss minimum
w

Figure 6.1

As we can see in the figure 6.1, if we initialize the weight randomly, it might not result in a global minimum of the loss function.
It is our duty to update the weights to the point where the loss is minimum. We have calculated **dw** above.

dw is nothing but the slope of the tangent of the loss function at point w.
Considering the initial position of **w**.

Important point to understand :

In the above diagram, The slope of the tangent of the loss will be positive as initial value of w is greater and it needs to be reduced so as to attain global minimum.
If the value of w is low and we want to increase it to attain global minimum, the slope of the tangent of loss at point w will be negative

We want the value of w to be a little lower so as to attain global minima of the loss function as shown in the figure 6.1.

We know that dw is positive in the above graph and we need to reduce w.
This can be done by:

```
w = w — alpha*dw
b = b — alpha*bw
```

where alpha is a small number ranging between 0.1 to 0.0000001 (approx) and this alpha is knows as the learning rate.

*Doing this, we can reduce **w** if slope of tangent of loss at **w** is positive and increase **w** if slope is negative*

7. Learning Rate

Learning rate alpha is something that we have to manually choose and it is something which we don’t know beforehand. Choosing it is a matter of trial and error.
The reason we do not directly subtract **dw** from **w** is because, it might result in too much change in the value of w and might not end up in global minimum but, even further away from it.

8. Training Loop

The process of calculating the gradients and updating the weight and bias is repeated for several times, which results in the optimized value for weight and bias.

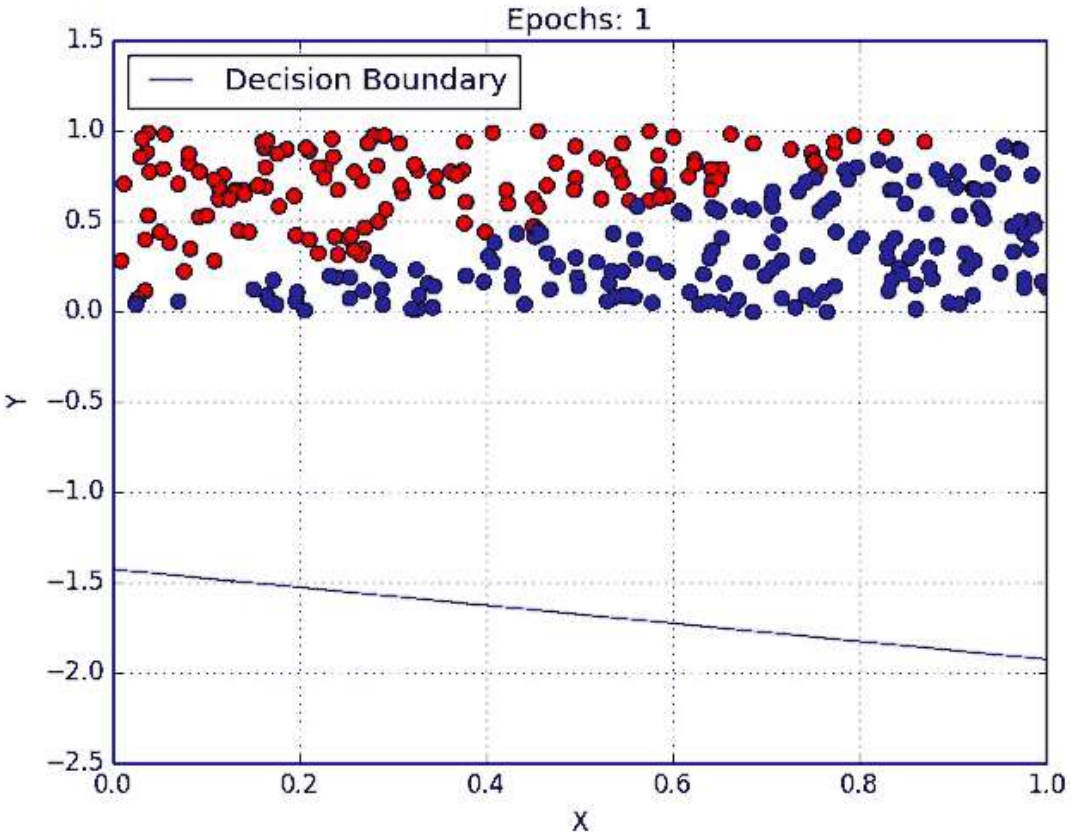
9. Prediction

After the training loop, the value of weight and bias is now optimized, which can be used to predict new values given new x values.

y = x*w + b

https://towardsdatascience.com/introduction-to-logistic-regression-66248243c148

Introduction to Logistic Regression



Logistic regression is a classification algorithm used to assign observations to a discrete set of classes. Some of the examples of classification problems are Email spam or not spam, Online transactions Fraud or not Fraud, Tumor Malignant or Benign.

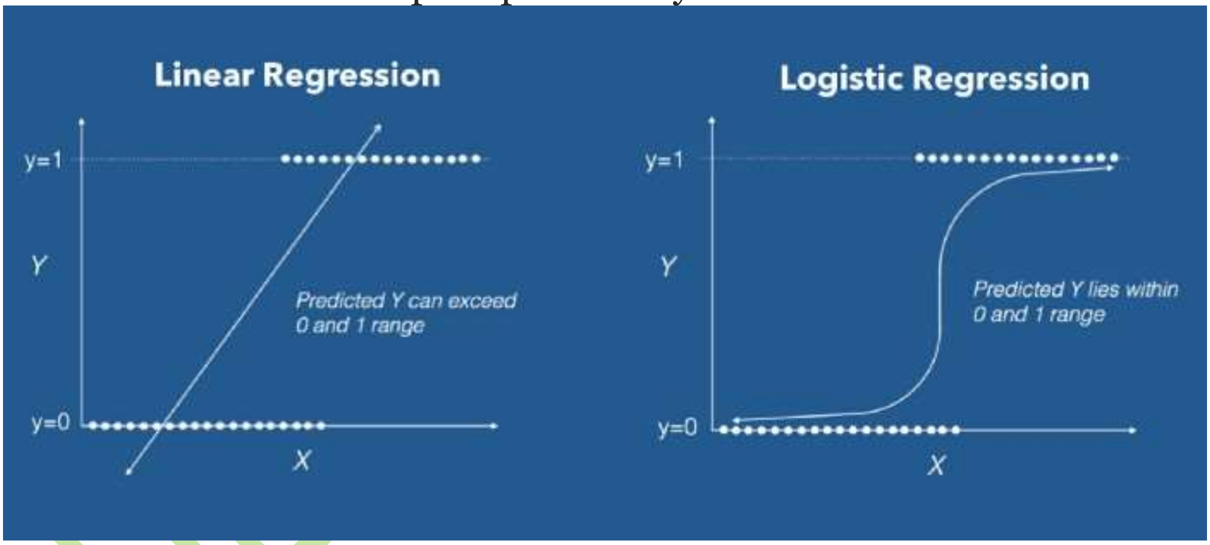
Logistic regression transforms its output using the logistic sigmoid function to return a probability value.

What are the types of logistic regression

1. Binary (eg. Tumor Malignant or Benign)
2. Multi-linear functions failsClass (eg. Cats, dogs or Sheep's)

Logistic Regression

Logistic Regression is a Machine Learning algorithm which is used for the classification problems, it is a predictive analysis algorithm and based on the concept of probability.



Linear Regression VS Logistic Regression Graph| Image: Data Camp

We can call a Logistic Regression a Linear Regression model but the Logistic Regression uses a more complex cost function, this cost function can be defined as the '**Sigmoid function**' or also known as the 'logistic function' instead of a linear function.

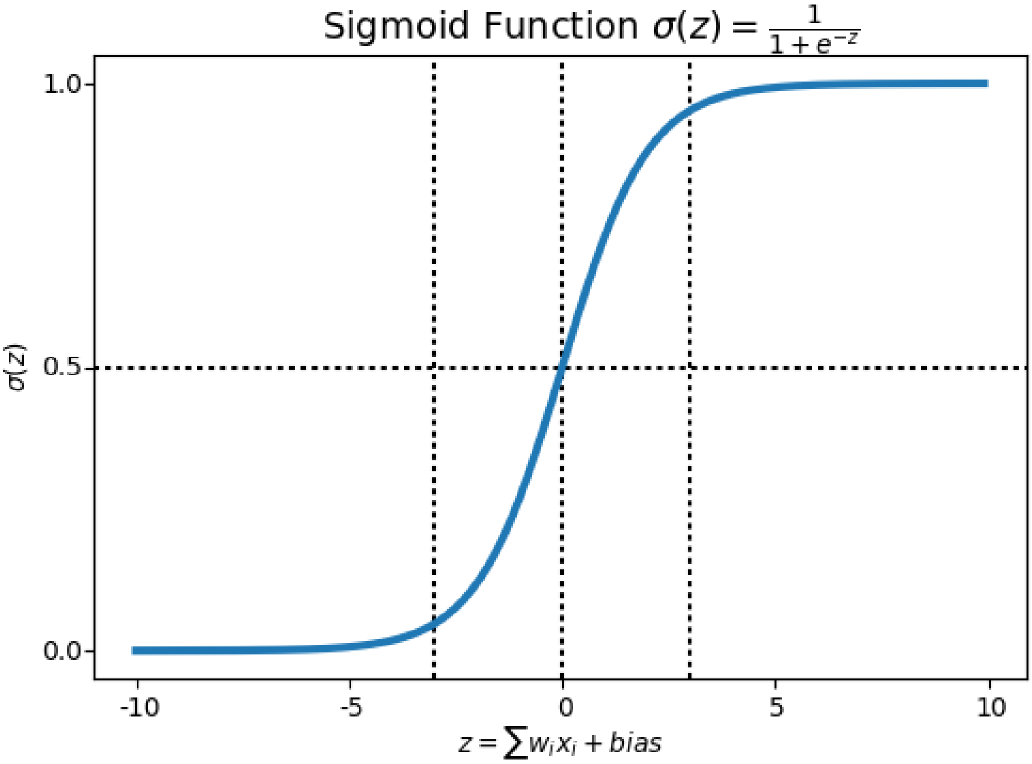
The hypothesis of logistic regression tends it to limit the cost function between 0 and 1. Therefore linear functions fail to represent it as it can have a value greater than 1 or less than 0 which is not possible as per the hypothesis of logistic regression.

$$0 \leq h_\theta(x) \leq 1$$

Logistic regression hypothesis expectation

What is the Sigmoid Function?

In order to map predicted values to probabilities, we use the Sigmoid function. The function maps any real value into another value between 0 and 1. In machine learning, we use sigmoid to map predictions to probabilities.



Sigmoid Function Graph

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

Formula of a sigmoid function | Image: Analytics India Magazine

Hypothesis Representation

When using *linear regression* we used a formula of the hypothesis i.e.

$h\Theta(x) = \beta 0 + \beta 1X$

For logistic regression we are going to modify it a little bit i.e.

$\sigma(Z) = \sigma(\beta 0 + \beta 1X)$

We have expected that our hypothesis will give values between 0 and 1.

$Z = \beta_0 + \beta_1 X$

$h\Theta(x) = \text{sigmoid}(Z)$

i.e. $h\Theta(x) = 1/(1 + e\text{^}-(\beta_0 + \beta_1 X)$

$$h\theta(X) = \frac{1}{1+e^{-(\beta_0+\beta_1 X)}}$$

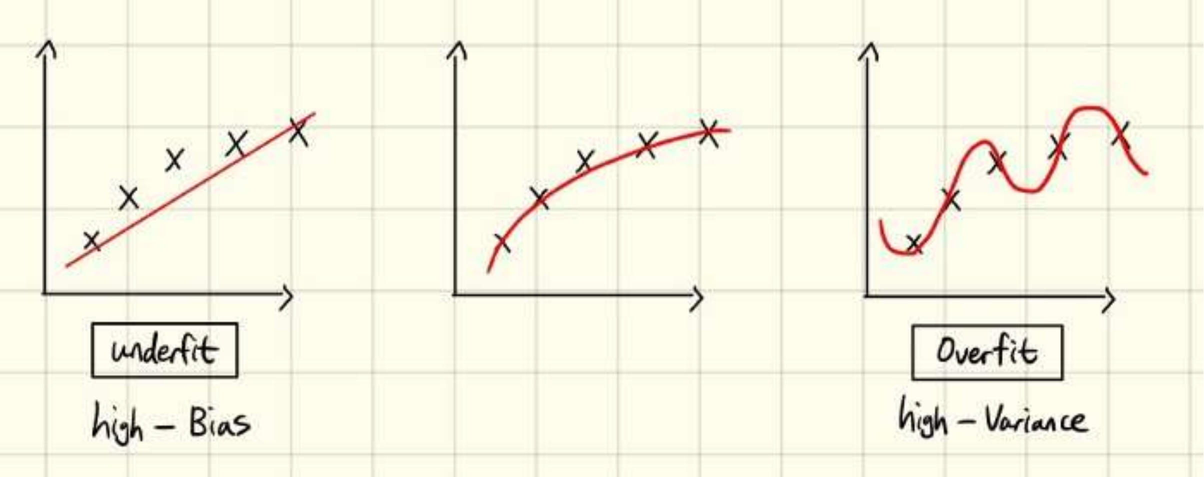
The Hypothesis of logistic regression

ML-Regularization

https://medium.com/@shiny_jay/ml-regularization-79a081666fbc

The Problem of Overfitting

Having too many features can cause a problem. The hypothesis function can become too complicated. It would predict the training example very well, but our purpose is to perform prediction on a new data, a data not included in the training set. Hypothesis that is too fit to the training data can be very poor when predicting a new data. When a hypothesis is too fit to the training data, we address this problem as “**Overfitting**”.



- Bias: The accuracy of our predictions
- Variance: The difference between many model's predictions

If we have high bias, we cannot predict the data well so we have an "underfit" situation. In contrast, if we have high variance, it can be told that the model is not generalized well and would perform poorly on a new data. We have an "overfit" situation.

Options to deal with overfitting

There are some options to deal with overfitting.

- Option 1: Reduce feature number

Manually select features you want to use/Model selection algorithm

However, you might not want to throw away the features. Maybe all of the features might be a help to our prediction. In this case, we retain all of the features and try to avoid overfitting. This method is called "Regularization".

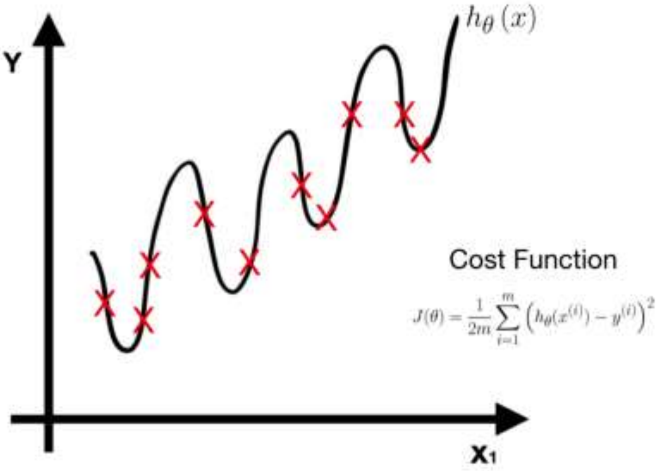
- Option 2: Regularization

Retain all of the features/reduce the magnitude(value) of parameter theta.

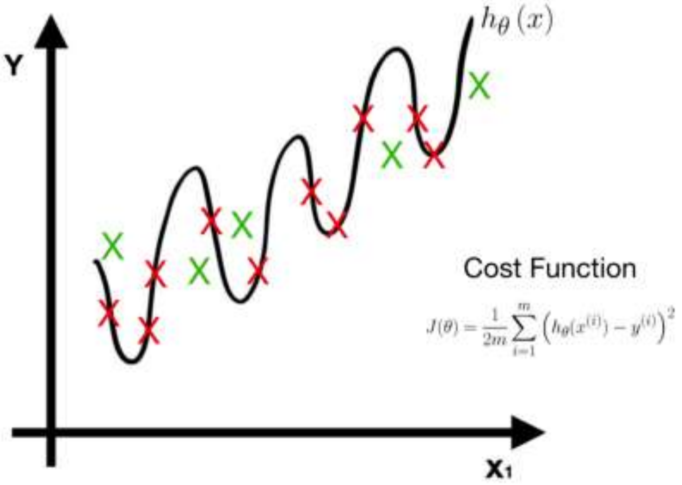
What is a good model?

The definition of a 'good' model is the one with least training error. First, define a hypothesis function ***H*** (i.e. model). Second, define a Cost Function ***J(θ)*** for error measuring. Third, learn parameters of ***H*** by minimizing ***J(θ)***. In the end, we can get a learned model with least training error. In the extreme case, this 'good' model has zero training error and fits training data perfectly, that is, the predict label of all training data is equal to their corresponding truth label.

Taking linear regression as example, the model fits all training data (red points) perfectly.


$h_\theta(x)$
Y
Cost Function
$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$
X1

However, when new data (green point) comes in, this model has terrible performance with huge prediction error.



The problem here is called Overfitting. A truly good model must have both little training error and little prediction error.

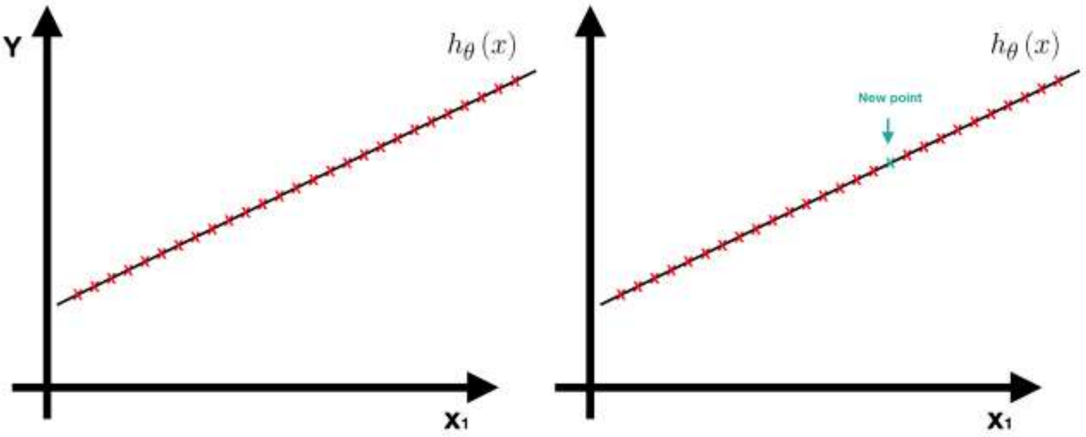
https://medium.com/@qempsil0914/courseras-machine-learning-notes-week3-overfitting-and-regularization-partii-3e3f3f36a287

Overfitting

The learned model works well for training data but terrible for testing data (unknown data). In other words, the model has little training error but has huge perdition error.

If we have a whole dataset that captured all possibilities of the problem, we don't need to worry about overfitting. When new data comes in, it shall fall into one possibility, hence, the model can predict it perfectly.

For example, suppose we have a training dataset that captures all points on the line and also get a perfect model h(x) with zero training error. Whenever new data comes in, it shall be one of the points, hence, the prediction error must be zero. In this special case, overfitting is not a problem.



Unfortunately, the training data we get in reality is normally a small part of the whole dataset. Therefore, even if the model fits these training data perfectly, this is still not a good model and overfitting must occur.

When overfitting occurs, we get an over complex model with too many features. One way to avoid it is to apply Regularization and then we can get a better model with proper features.

Regularization

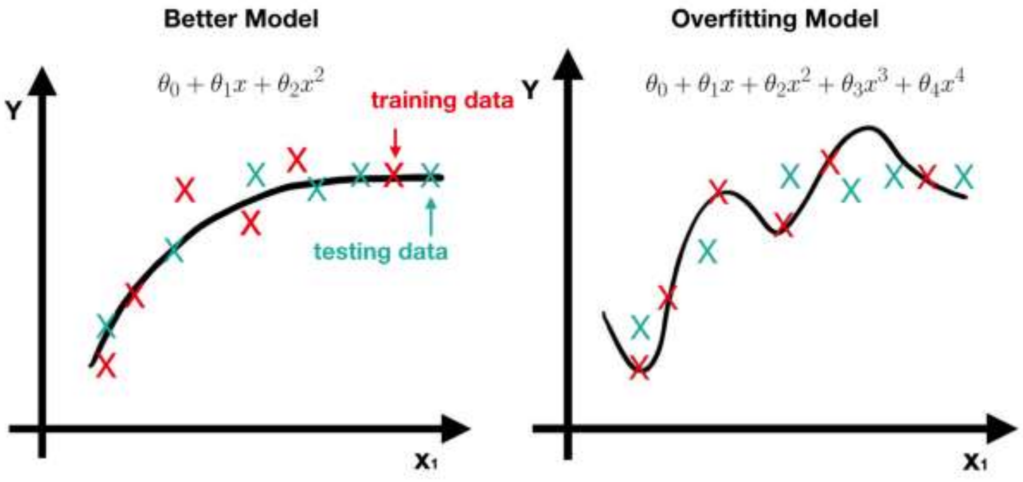
*It's a technique applied to Cost Function **J(θ)** in order to avoid Overfitting.*

The core idea in Regularization is to keep more important features and ignore unimportant ones. The importance of feature is measured by the value of its parameter ***θj*.**

In linear regression, we modify its cost function by adding regularization term. The value of ***θj*** is controlled by regularization parameter **λ**. Note that ***m*** is the number of data and ***n*** is the number of features(parameters.

$$J(\theta)=\frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)})-y^{(i)}\right)^2+\overbrace{\boxed{\underset{\uparrow\ \text{Regularization Parameter}}{\boxed{\lambda}}\sum_{j=1\ \leftarrow\ \text{start at }\theta_1}^{n}\theta_j{}^2}}^{\text{Regularization Term}}$$

For instance, if we want to get a better model instead of the overfitting one. Obviously, we don't need features X³ and X⁴ since they are unimportant. The procedure describes below.



First, we modify the Cost Function ***J(θ)*** by adding regularization. Second, apply gradient descent in order to minimize ***J(θ)*** and get the values of ***θ3*** and ***θ4.*** After the minimize procedure, the values of ***θ3*** and ***θ4*** must be near to zero if **λ=1000.**

Regularization Term

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 + \boxed{1000{\theta_3}^2 + 1000{\theta_4}^2}$$

$$\min_{\theta} J(\theta), \text{ getting } \theta_3 \approx 0,\ \theta_4 \approx 0$$

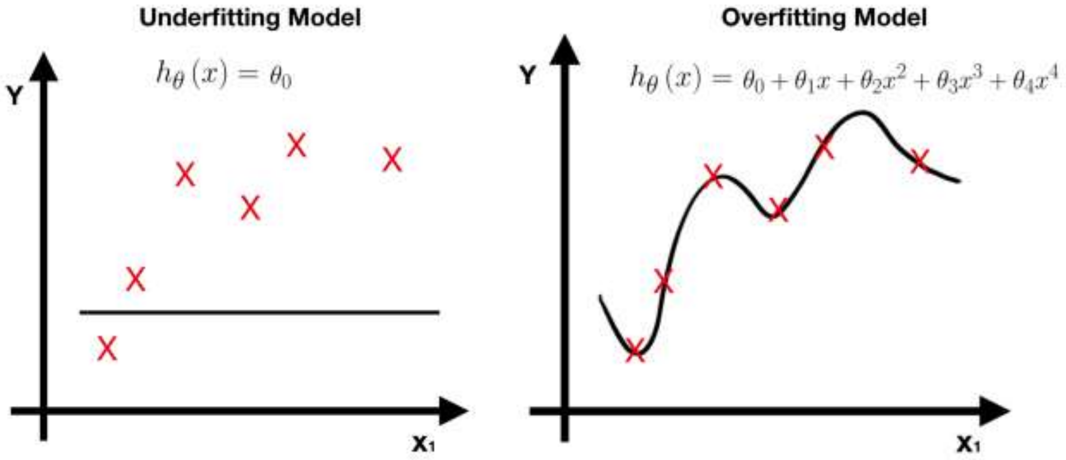
Remember, the value of ***J(θ)*** represents training error and this value must be positive (≥0). The parameter **λ=1000** has significant effect on ***J(θ),*** therefore, ***θ3*** and ***θ4*** must be near to zero (e.g 0.000001) so as to eliminate error value.

Positive

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\boxed{\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2} + 1000\boxed{{\theta_3}^2} + 1000\boxed{{\theta_4}^2}$$

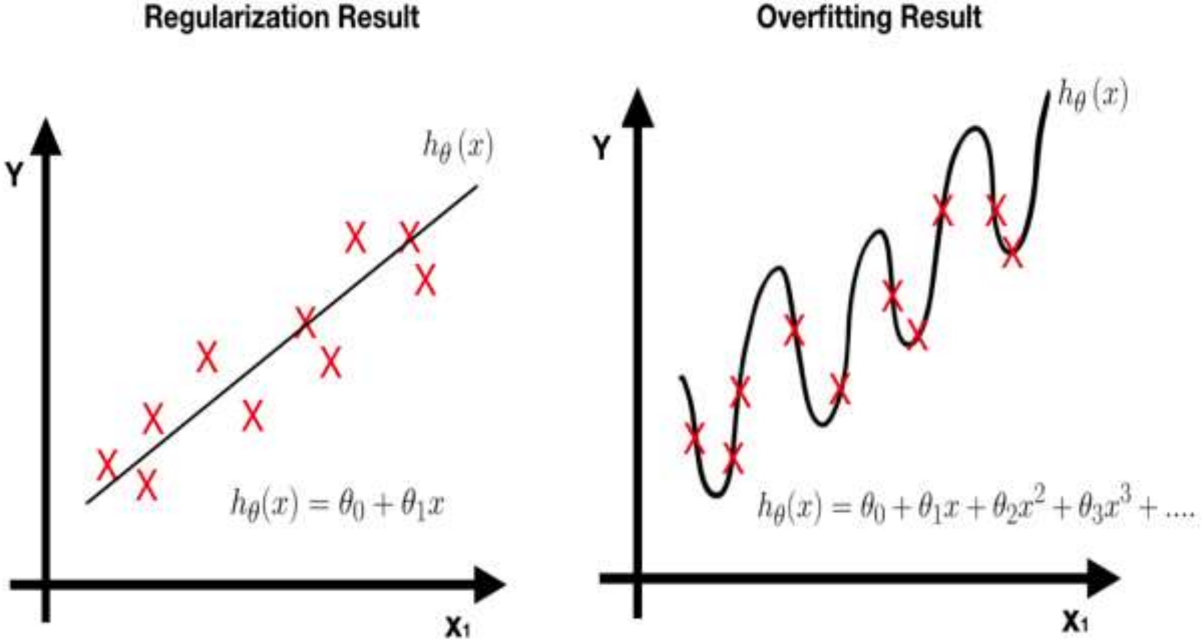
Regularization Parameter λ

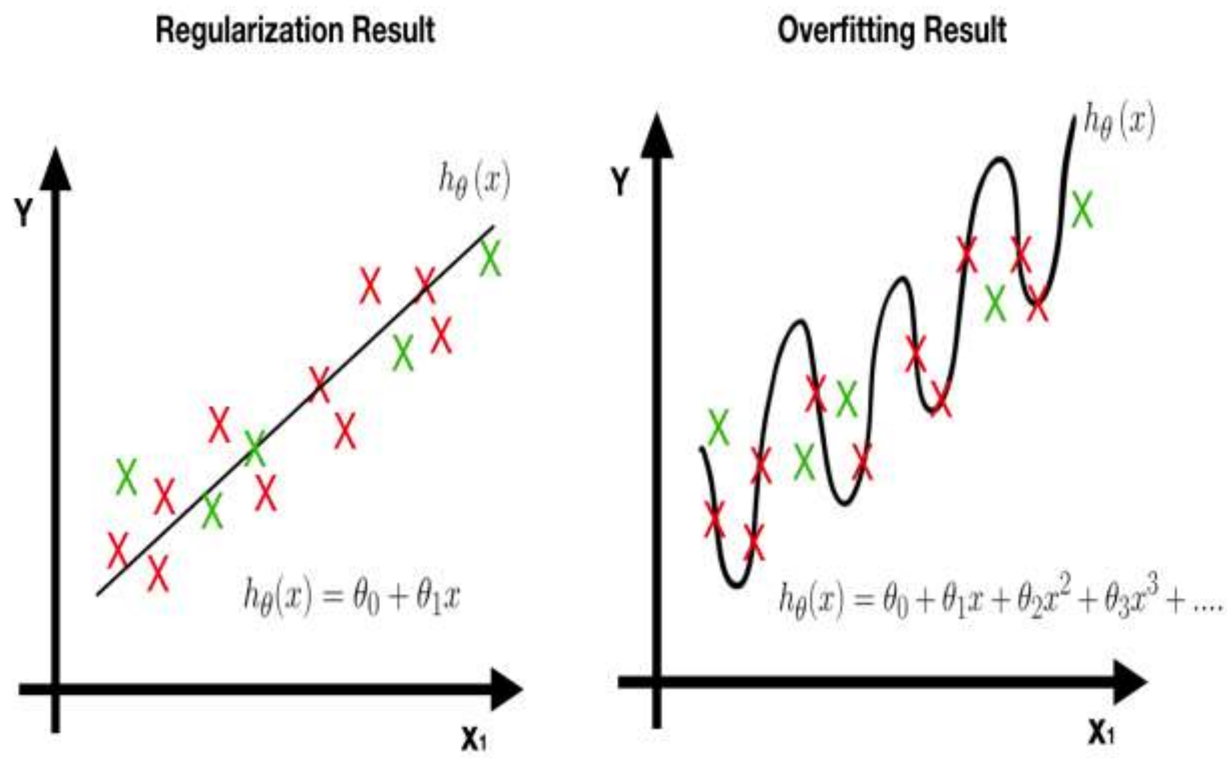
- If **λ** is too large, then all the values of ***θ*** may be near to zero and this may cause Underfitting. In other words, this model has both large training error and large prediction error. (Note that the regularization term starts from ***θ1***)
- If **λ** is zero or too small, its effect on parameters ***θ*** is little. This may cause Overfitting.



To sum up, there are two advantages of using regularization.

- The prediction error of the regularized model is lesser, that is, it works well in testing data (green points).
- The regularization model is simpler since it has less features (parameters).





So far, we have discussed the concept of regularization.

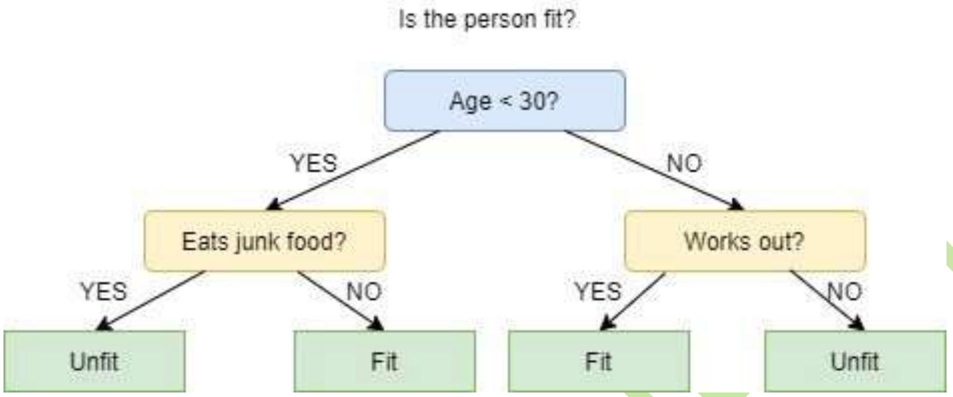
https://towardsdatascience.com/decision-trees-for-classification-id3-algorithm-explained-89df76e72df1

What are Decision Trees?

In simple words, a decision tree is a structure that contains nodes (rectangular boxes) and edges(arrows) and is built from a dataset (table of columns representing features/attributes and rows corresponds to records). Each node is either used to ***make a***

decision (known as decision node) or ***represent an outcome*** (known as leaf node).

Decision tree Example



The picture above depicts a decision tree that is used to classify whether a person is **Fit** or **Unfit.** The decision nodes here are questions like ‘’*‘Is the person less than 30 years of age?’*, *‘Does the person eat junk?’*, etc. and the leaves are one of the two possible outcomes viz. **Fit** and **Unfit**. Looking at the Decision Tree we can say make the following decisions: if a person is less than 30 years of age and doesn’t eat junk food then he is Fit, if a person is less than 30 years of age and eats junk food then he is Unfit and so on.

The initial node is called the **root node** *(colored in blue),* the final nodes are called the **leaf nodes** *(colored in green)* and the rest of the nodes are called **intermediate** or **internal** nodes. The root and intermediate nodes represent the decisions while the leaf nodes represent the outcomes.

ID3 in brief

ID3 stands for Iterative Dichotomiser 3 and is named such because the algorithm iteratively (repeatedly) dichotomizes(divides) features into two or more groups at each step.

Invented by Ross Quinlan, ID3 uses a **top-down greedy** approach to build a decision tree. In simple words, the **top-down** approach means that we start building the tree from the top and the **greedy** approach means that at each iteration we select the best feature at the present moment to create a node.

Most generally ID3 is only used for classification problems with nominal features only.

Dataset description

In this article, we'll be using a sample dataset of COVID-19 infection. A preview of the entire dataset is shown below.

```
+----+-------+-------+------------------+----------+
| ID | Fever | Cough | Breathing issues | Infected |
+----+-------+-------+------------------+----------+
| 1  | NO    | NO    | NO               | NO       |
+----+-------+-------+------------------+----------+
| 2  | YES   | YES   | YES              | YES      |
+----+-------+-------+------------------+----------+
| 3  | YES   | YES   | NO               | NO       |
+----+-------+-------+------------------+----------+
| 4  | YES   | NO    | YES              | YES      |
+----+-------+-------+------------------+----------+
| 5  | YES   | YES   | YES              | YES      |
+----+-------+-------+------------------+----------+
| 6  | NO    | YES   | NO               | NO       |
+----+-------+-------+------------------+----------+
| 7  | YES   | NO    | YES              | YES      |
+----+-------+-------+------------------+----------+
| 8  | YES   | NO    | YES              | YES      |
+----+-------+-------+------------------+----------+
| 9  | NO     | YES    | YES                | YES       |
+----+-------+-------+------------------+----------+
| 10 | YES    | YES    | NO                 | YES       |
+----+-------+-------+------------------+----------+
| 11 | NO     | YES    | NO                 | NO        |
+----+-------+-------+------------------+----------+
| 12 | NO     | YES    | YES                | YES       |
+----+-------+-------+------------------+----------+
| 13 | NO     | YES    | YES                | NO        |
+----+-------+-------+------------------+----------+
| 14 | YES    | YES    | NO                 | NO        |
+----+-------+-------+------------------+----------+
```

The columns are self-explanatory. Y and N stand for Yes and No respectively. The values or **classes** in Infected column Y and N represent Infected and Not Infected respectively.

The columns used to make decision nodes viz. ‘Breathing Issues’, ‘Cough’ and ‘Fever’ are called feature columns or just features and the column used for leaf nodes i.e. ‘Infected’ is called the target column.

Metrics in ID3

As mentioned previously, the ID3 algorithm selects the best feature at each step while building a Decision tree. Before you ask, the answer to the question: ‘How does ID3 select the best feature?’ is that ID3 uses **Information Gain** or just **Gain** to find the best feature.

Information Gain calculates the reduction in the entropy and measures how well a given feature separates or classifies the target classes. The feature with the **highest Information Gain** is selected as the **best** one.

In simple words, **Entropy** is the measure of disorder and the Entropy of a dataset is the measure of disorder in the target feature of the dataset.
In the case of binary classification (where the target column has only two types of classes) entropy is **0** if all values in the target column are homogenous(similar) and will be **1** if the target column has equal number values for both the classes.

We denote our dataset as **S,** entropy is calculated as:

$$\text{Entropy}(S) = - \sum p_i * \log_2(p_i) \;\; ; \; i = 1 \text{ to } n$$

where,
n is the total number of classes in the target column (in our case n = 2 i.e YES and NO). $\mathbf{p_i}$ is the **probability of class 'i'** or the ratio of "*number of rows with class i in the target column*" to the "*total number of rows*" in the dataset.

Information Gain for a feature column **A** is calculated as:

$$\text{IG}(S,\ A) = \text{Entropy}(S) - \sum((|S_v| \ / \ |S|) * \text{Entropy}(S_v))$$

where $\mathbf{S_v}$ is the set of rows in **S** for which the feature column **A** has value **v**, |$\mathbf{S_v}$| is the number of rows in $\mathbf{S_v}$ and likewise **|S|** is the number of rows in **S.**

ID3 Steps

1. Calculate the Information Gain of each feature.

2. Considering that all rows don't belong to the same class, split the dataset **S** into subsets using the feature for which the Information Gain is maximum.

3. Make a decision tree node using the feature with the maximum Information gain.

4. If all rows belong to the same class, make the current node as a leaf node with the class as its label.

5. Repeat for the remaining features until we run out of all features, or the decision tree has all leaf nodes.

Implementation on our Dataset

As stated in the previous section the first step is to find the best feature i.e. the one that has the maximum Information Gain**(IG)**. We'll calculate the IG for each of the features now, but for that, we first need to calculate the entropy of **S**

From the total of 14 rows in our dataset **S**, there are **8** rows with the target value **YES** and **6** rows with the target value **NO**. The entropy of **S** is calculated as:

$$\texttt{Entropy(S)} = -\ (8/14)\ *\ \log_2(8/14)\ -\ (6/14)\ *\ \log_2(6/14)\ =\ 0.99$$

Note: If all the values in our target column are same the entropy will be zero (meaning that it has no or zero randomness).

We now calculate the Information Gain for each feature:

IG calculation for Fever:
In this(Fever) feature there are **8** rows having value **YES** and **6** rows having value **NO.**
As shown below, in the **8** rows with **YES for** Fever, there are **6** rows having target value **YES** and **2** rows having target value **NO.**

```
+-------+-------+------------------+----------+
| Fever | Cough | Breathing issues | Infected |
+-------+-------+------------------+----------+
| YES   | YES   | YES              | YES      |
+-------+-------+------------------+----------+
| YES   | YES   | NO               | NO       |
+-------+-------+------------------+----------+
| YES   | NO    | YES              | YES      |
+-------+-------+------------------+----------+
| YES   | YES   | YES              | YES      |
+-------+-------+------------------+----------+
| YES   | NO    | YES              | YES      |
+-------+-------+------------------+----------+
| YES   | NO    | YES              | YES      |
+-------+-------+------------------+----------+
| YES   | YES   | NO               | YES      |
+-------+-------+------------------+----------+
| YES   | YES   | NO               | NO       |
+-------+-------+------------------+----------+
```

As shown below, in the **6** rows with **NO**, there are **2** rows having target value **YES** and **4** rows having target value **NO.**

```
+-------+-------+------------------+----------+
| Fever | Cough | Breathing issues | Infected |
+-------+-------+------------------+----------+
| NO    | NO    | NO               | NO       |
+-------+-------+------------------+----------+
| NO    | YES   | NO               | NO       |
+-------+-------+------------------+----------+
| NO    | YES   | YES              | YES      |
+-------+-------+------------------+----------+
| NO    | YES   | NO               | NO       |
+-------+-------+------------------+----------+
| NO    | YES   | YES              | YES      |
+-------+-------+------------------+----------+
| NO    | YES   | YES              | NO       |
+-------+-------+------------------+----------+
```

The block, below, demonstrates the calculation of Information Gain for **Fever.**

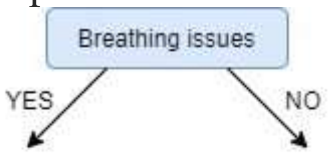
```
# total rows
|S| = 14For v = YES, |Sᵥ| = 8
Entropy(Sᵥ) = - (6/8) * log₂(6/8) - (2/8) * log₂(2/8) = 0.81For v
= NO, |Sᵥ| = 6
Entropy(Sᵥ) = - (2/6) * log₂(2/6) - (4/6) * log₂(4/6) = 0.91#
Expanding the summation in the IG formula:
IG(S, Fever) = Entropy(S) - (|Sʏᴇs| / |S|) * Entropy(Sʏᴇs) -
(|Sɴᴏ| / |S|) * Entropy(Sɴᴏ)∴ IG(S, Fever) = 0.99 - (8/14) * 0.81
- (6/14) * 0.91 = 0.13
```

Next, we calculate the **IG** for the features **“Cough”** and **“Breathing issues”.**

You can use this free online tool to calculate the Information Gain.

```
IG(S, Cough) = 0.04
IG(S, BreathingIssues) = 0.40
```

Since the feature **Breathing issues** have the highest Information Gain it is used to create the root node. Hence, after this initial step our tree looks like this:



Next, from the remaining two unused features, namely, **Fever** and **Cough**, we decide which one is the best for the left branch of **Breathing Issues**. Since the left branch of **Breathing Issues** denotes **YES,** we will work with the subset of the original data i.e the set of rows having **YES** as the value in the Breathing Issues column. These **8 rows** are shown below:

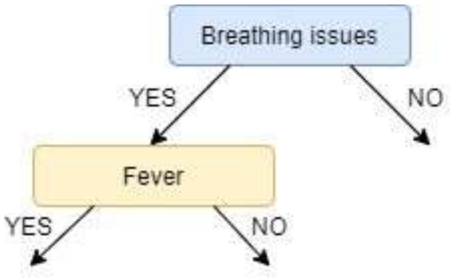
```
+-------+-------+------------------+----------+
| Fever | Cough | Breathing issues | Infected |
+-------+-------+------------------+----------+
| YES   | YES   | YES              | YES      |
+-------+-------+------------------+----------+
| YES   | NO    | YES              | YES      |
+-------+-------+------------------+----------+
| YES   | YES   | YES              | YES      |
+-------+-------+------------------+----------+
| YES   | NO    | YES              | YES      |
+-------+-------+------------------+----------+
| YES   | NO    | YES              | YES      |
+-------+-------+------------------+----------+
| NO    | YES   | YES              | YES      |
+-------+-------+------------------+----------+
| NO    | YES   | YES              | YES      |
+-------+-------+------------------+----------+
| NO    | YES   | YES              | NO       |
+-------+-------+------------------+----------+
```

Next, we calculate the IG for the features Fever and Cough using the subset **SBY** (**S**et **Breathing** Issues **Y**es) which is shown above :

*Note: For **IG** calculation the Entropy will be calculated from the subset **SBY** and not the original dataset **S.***

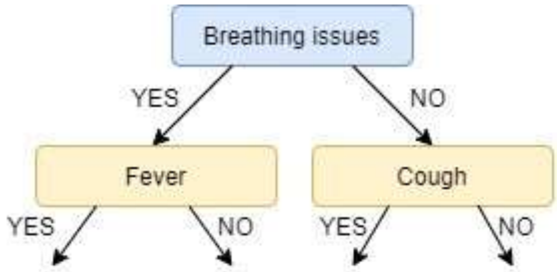
```
IG(SBY, Fever) = 0.20
IG(SBY, Cough) = 0.09
```

IG of Fever is greater than that of Cough, so we select **Fever** as the left branch of Breathing Issues: Our tree now looks like this:



Next, we find the feature with the maximum IG for the right branch of **Breathing Issues**. But, since there is only one unused feature left we have no other choice but to make it the right branch of the root node.

So our tree now looks like this:



There are no more unused features, so we stop here and jump to the final step of creating the leaf nodes.

For the left leaf node of Fever, we see the subset of rows from the original data set that has **Breathing Issues** and **Fever** both values as **YES**.

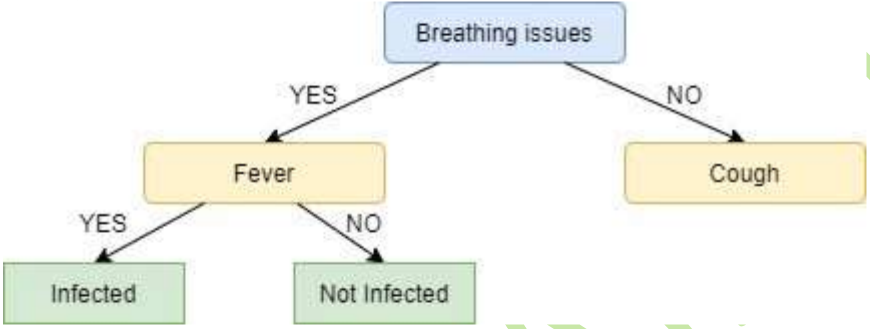
```
+-------+-------+------------------+----------+
| Fever | Cough | Breathing issues | Infected |
+-------+-------+------------------+----------+
| YES   | YES   | YES              | YES      |
+-------+-------+------------------+----------+
| YES   | NO    | YES              | YES      |
+-------+-------+------------------+----------+
| YES   | YES   | YES              | YES      |
+-------+-------+------------------+----------+
| YES   | NO    | YES              | YES      |
+-------+-------+------------------+----------+
| YES   | NO    | YES              | YES      |
+-------+-------+------------------+----------+
```

Since all the values in the target column are **YES,** we label the left leaf node as **YES**, but to make it more logical we label it **Infected.**

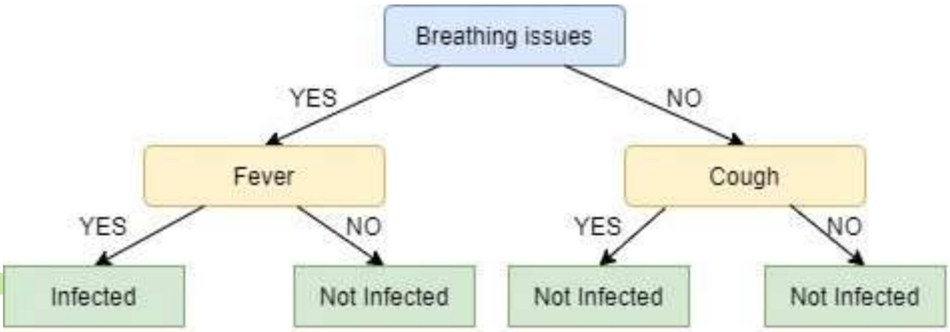
Similarly, for the right node of Fever we see the subset of rows from the original data set that have **Breathing Issues** value as **YES** and **Fever** as **NO**.

```
+-------+-------+------------------+----------+
| Fever | Cough | Breathing issues | Infected |
+-------+-------+------------------+----------+
| NO    | YES   | YES              | YES      |
+-------+-------+-----------------+----------+
| NO    | YES   | YES             | NO       |
+-------+-------+-----------------+----------+
| NO    | YES   | YES             | NO       |
+-------+-------+-----------------+----------+
```

Here not all but **most** of the **values** are **NO,** hence **NO** or **Not Infected** becomes our **right leaf node.** Our tree, now, looks like this:



We repeat the same process for the node **Cough**, however here both left and right leaves turn out to be the same i.e. **NO** or **Not Infected** as shown below:



https://www.javatpoint.com/k-nearest-neighbor-algorithm-for-machine-learning

K-Nearest Neighbor(KNN) Algorithm for Machine Learning

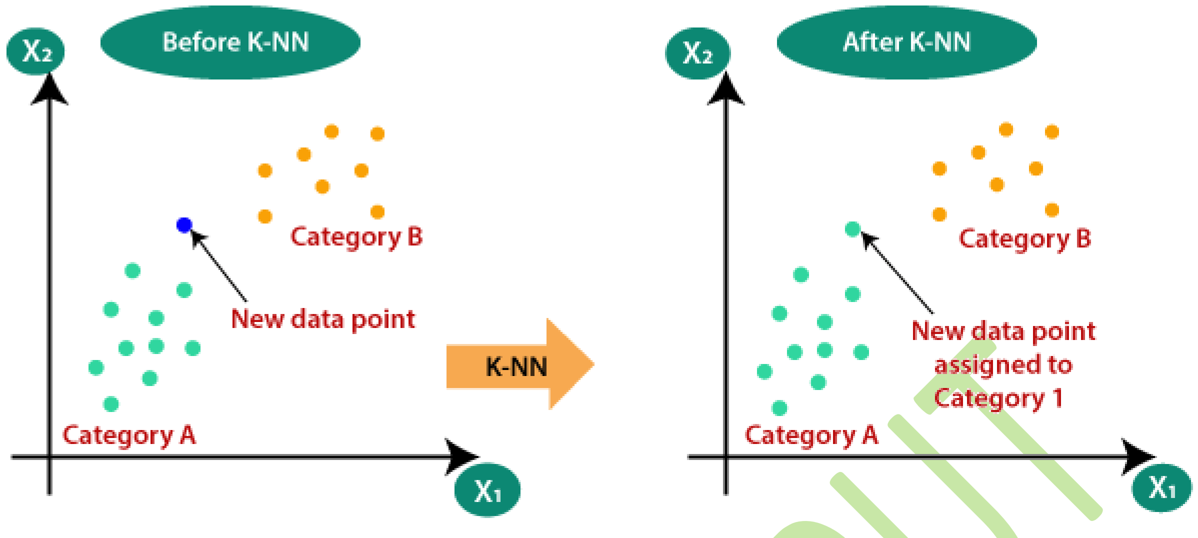
- K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on Supervised Learning technique.

- 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.

KNN Classifier
Input value
Predicted Output

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:

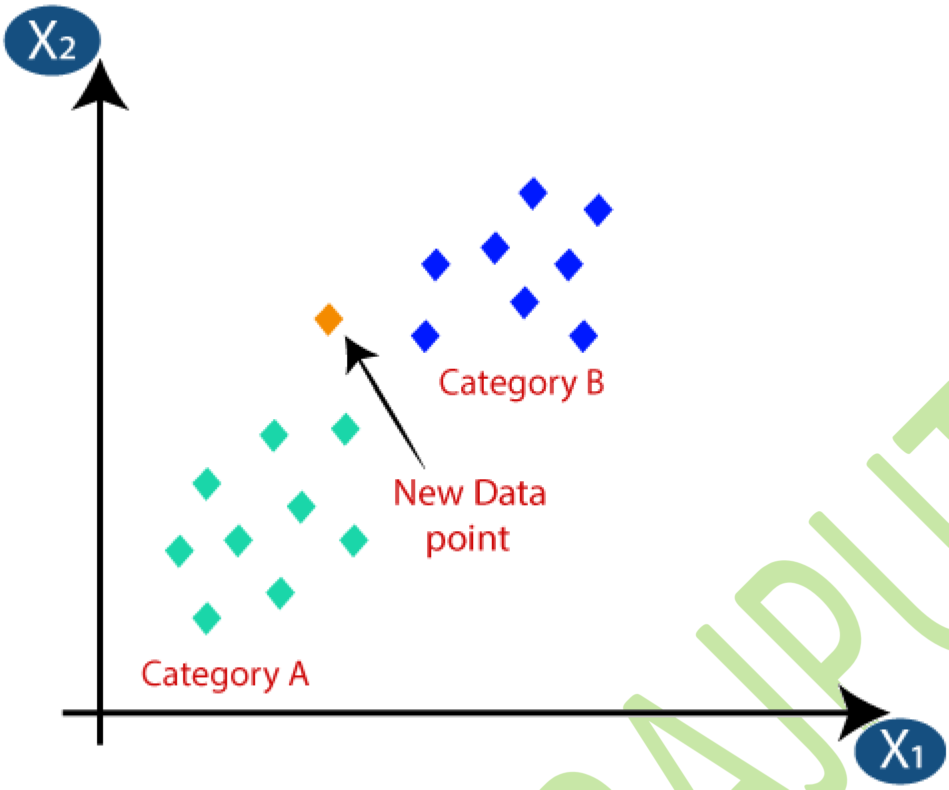


How does K-NN work?

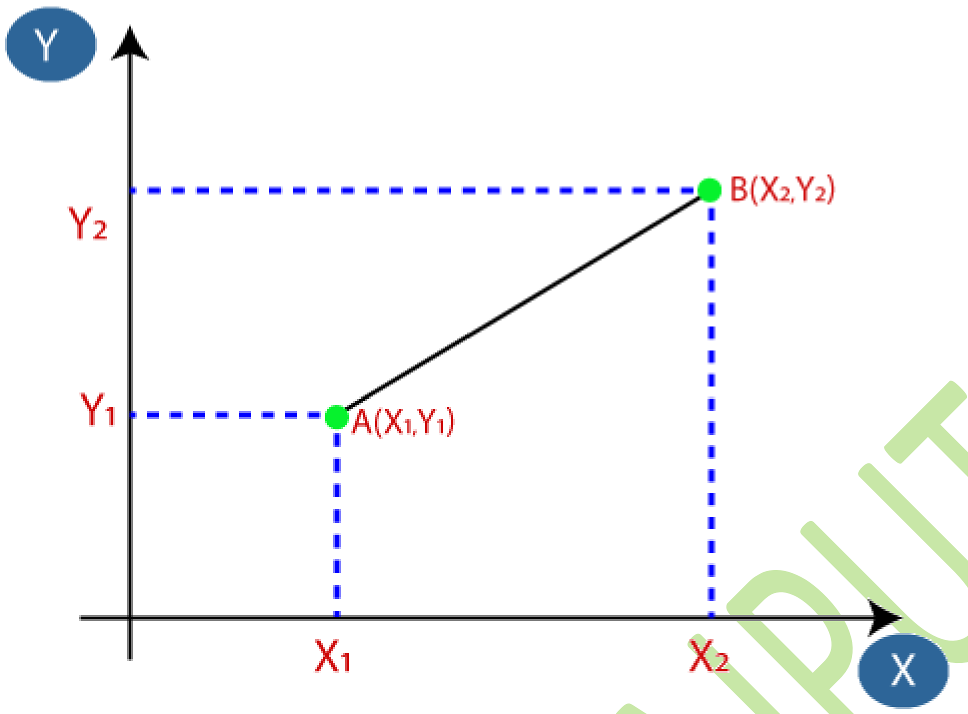
The K-NN working can be explained on the basis of the below algorithm:

- **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:

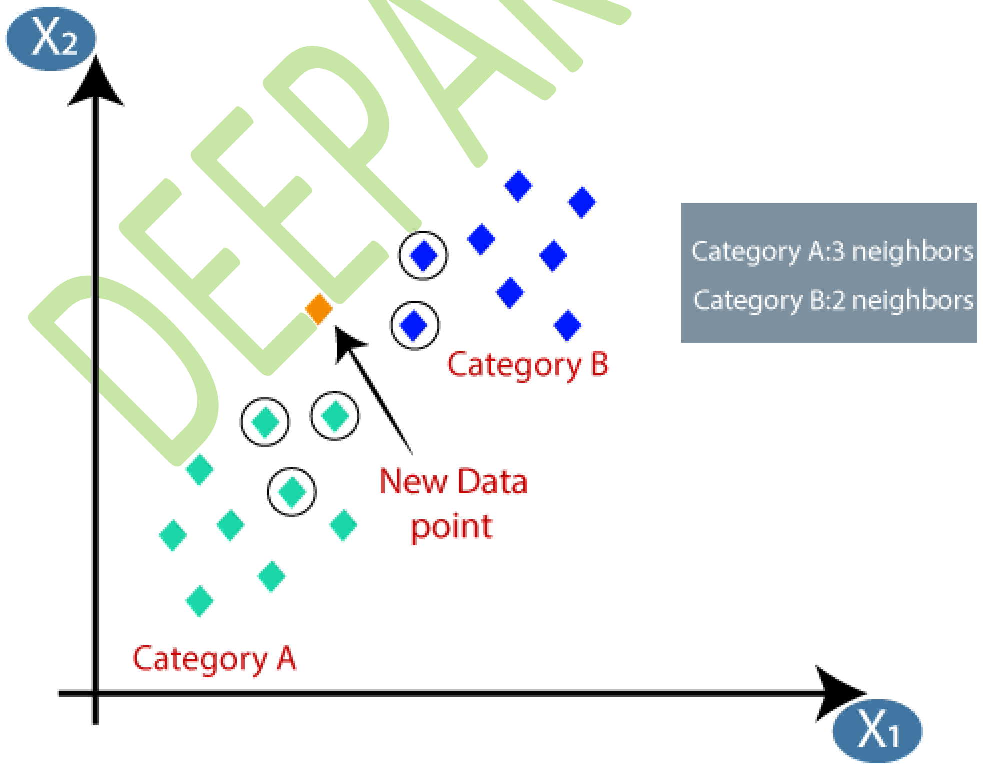


- Firstly, we will choose the number of neighbors, so we will choose the k=5.
- Next, we will calculate the **Euclidean distance** between the data points. The Euclidean distance is the distance between two points, which we have already studied in geometry. It can be calculated as:



Euclidean Distance between A_1 and $B_2 = \sqrt{(X_2-X_1)^2+(Y_2-Y_1)^2}$

- 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.

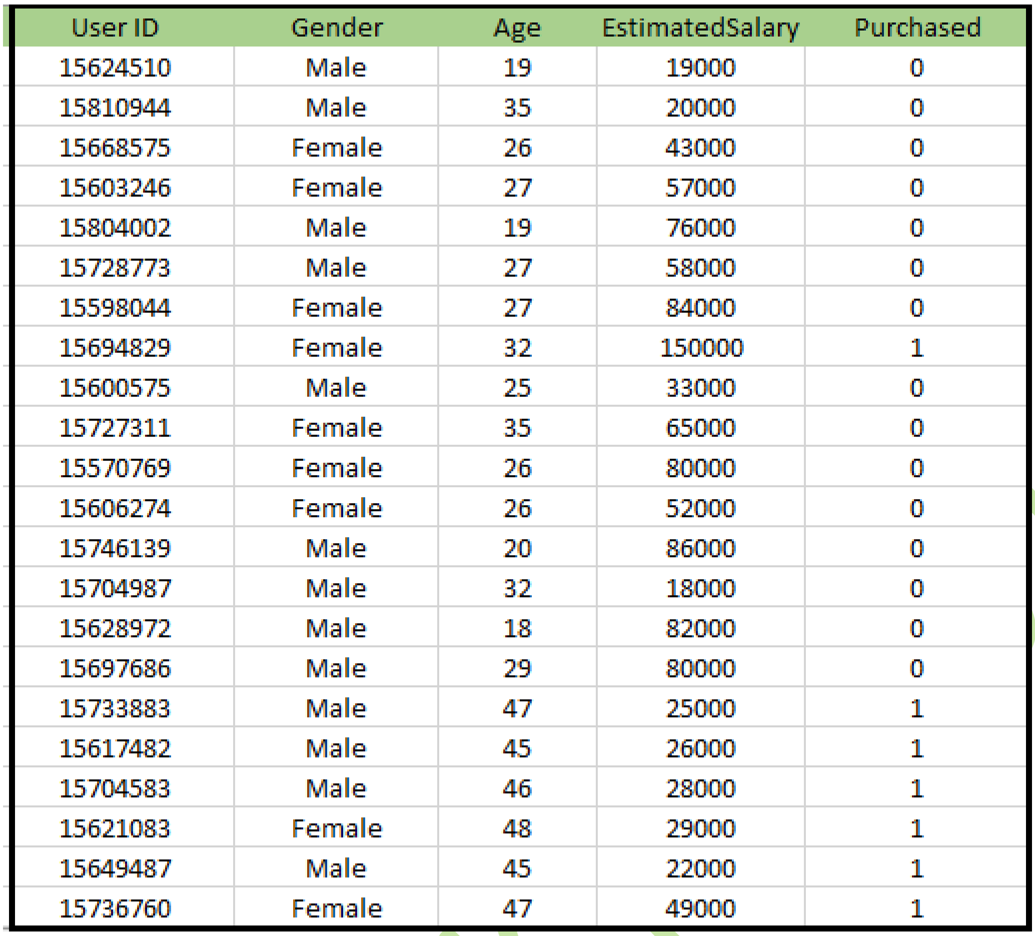
Disadvantages of KNN Algorithm:

- 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

```

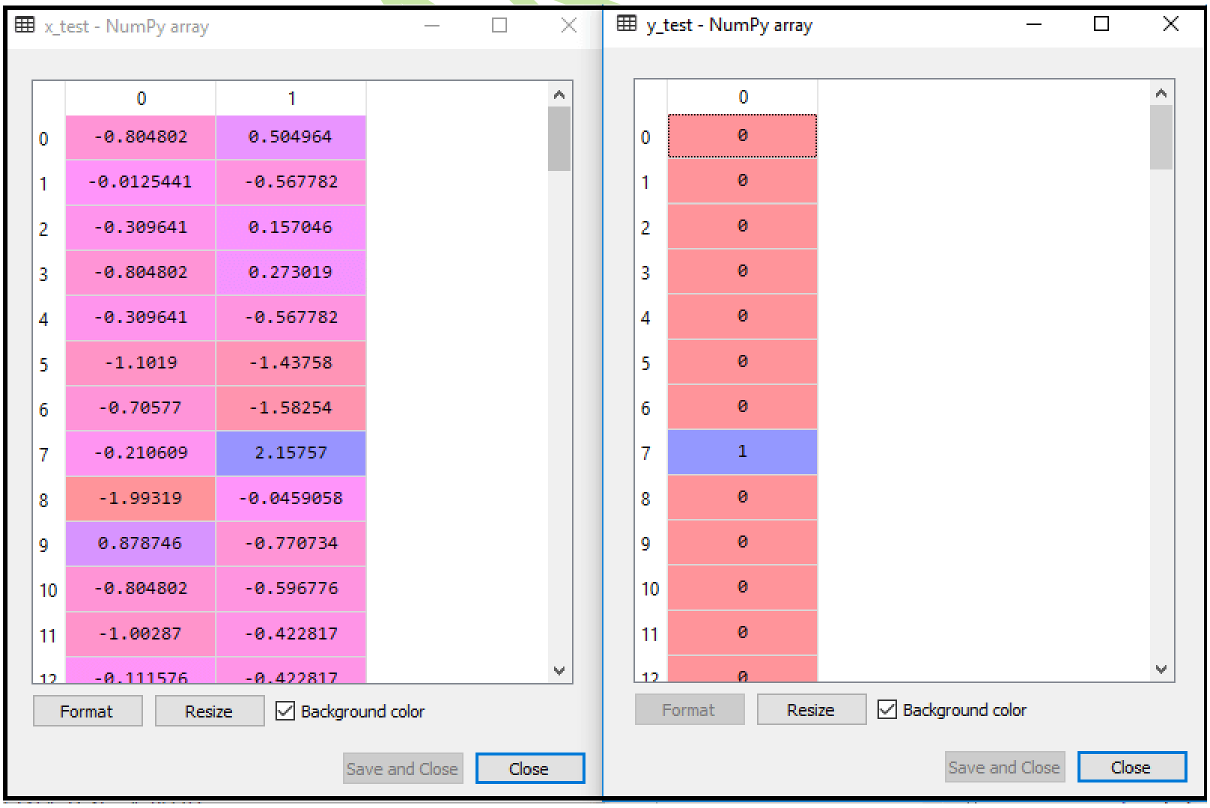
```
#importing datasets
data_set= pd.read_csv('user_data.csv')

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

# 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)

#feature Scaling
from sklearn.preprocessing import StandardScaler
st_x= StandardScaler()
x_train= st_x.fit_transform(x_train)
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:



	0	1
0	-0.804802	0.504964
1	-0.0125441	-0.567782
2	-0.309641	0.157046
3	-0.804802	0.273019
4	-0.309641	-0.567782
5	-1.1019	-1.43758
6	-0.70577	-1.58254
7	-0.210609	2.15757
8	-1.99319	-0.0459058
9	0.878746	-0.770734
10	-0.804802	-0.596776
11	-1.00287	-0.422817
12	-0.111576	-0.422817

	0
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	1
8	0
9	0
10	0
11	0
12	0

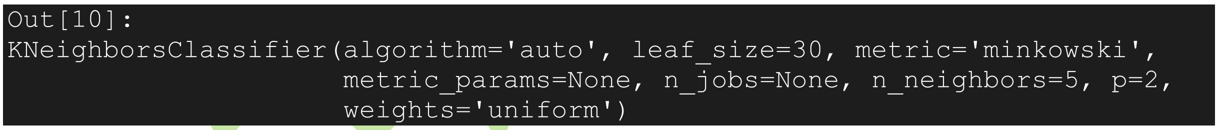
From the above output image, we can see that our data is successfully scaled.

- **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:



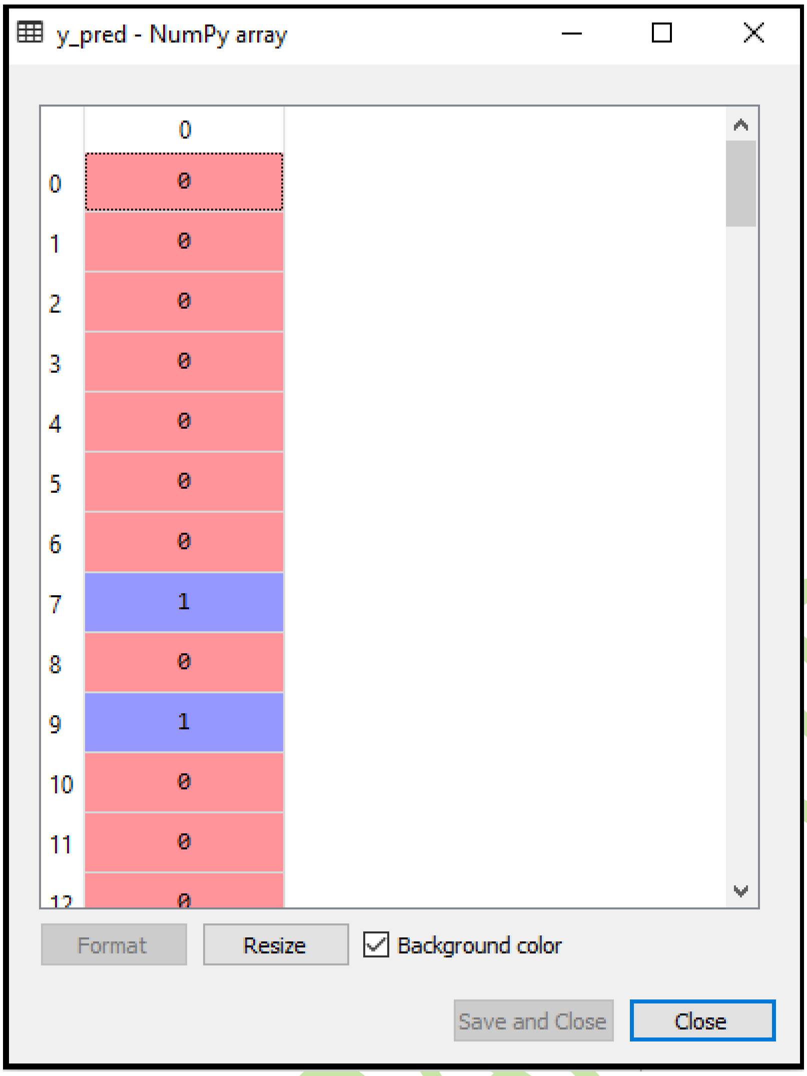
```
Out[10]:
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

- **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:



	0
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	1
8	0
9	1
10	0
11	0
12	0

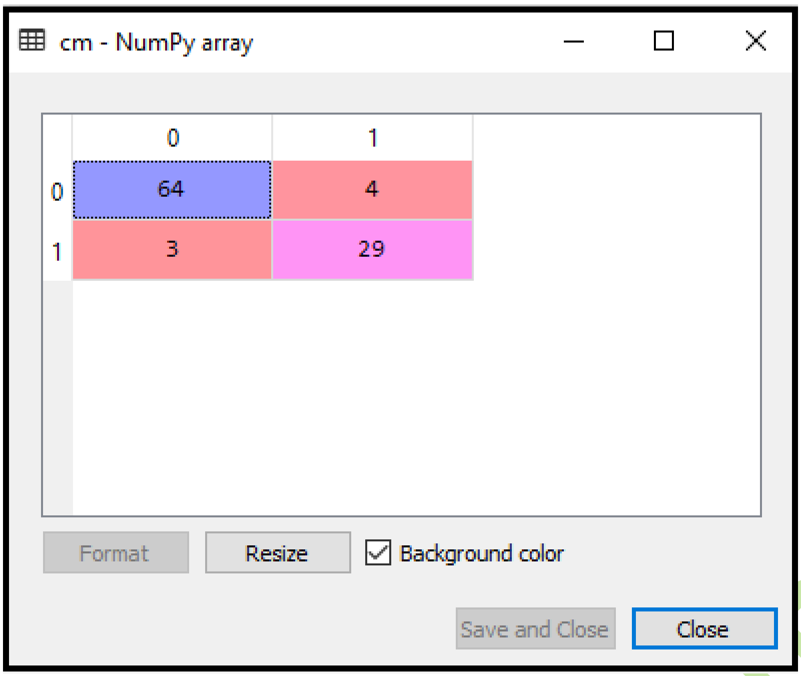
- **Creating the Confusion Matrix:**

  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
  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:


cm - NumPy array

	0	1
0	64	4
1	3	29

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:**

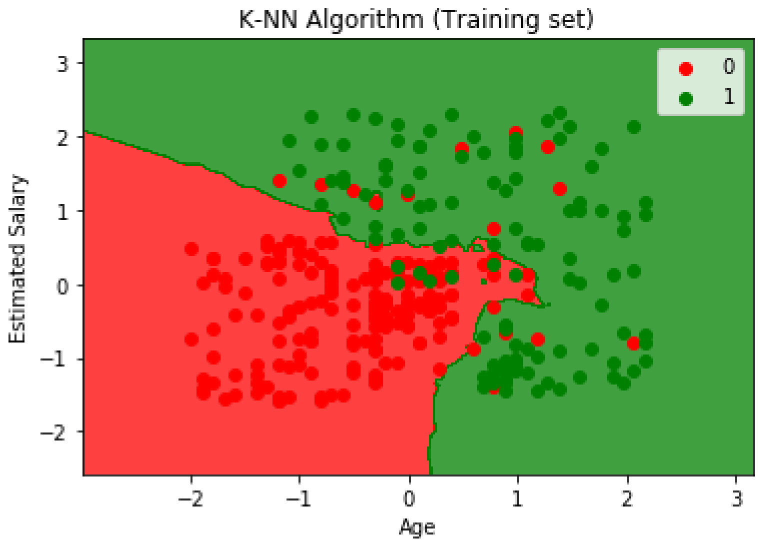
  Now, we will visualize the training set result for K-NN model. The code will remain same as we did in Logistic Regression, except the name of the graph. Below is the code for it:

```
#Visulaizing the trianing set result
from matplotlib.colors import ListedColormap
x_set, y_set = x_train, y_train
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(x1.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 (Training set)')
mtp.xlabel('Age')
mtp.ylabel('Estimated Salary')
mtp.legend()
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.

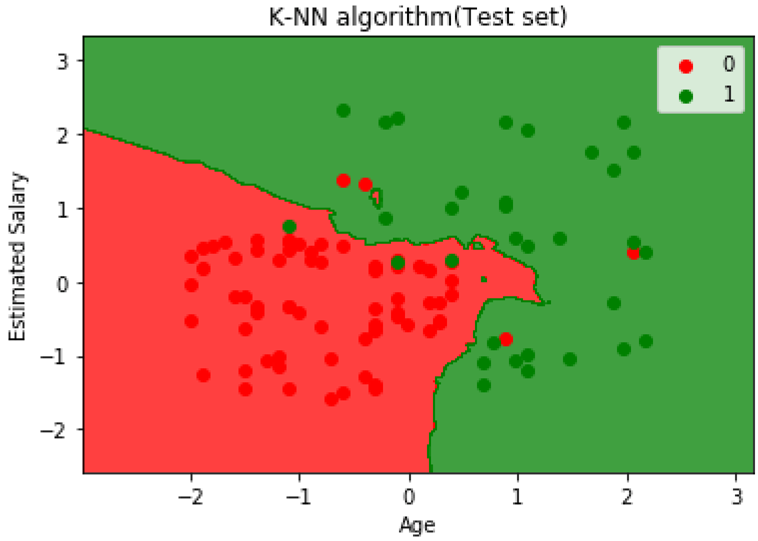
- Hence our model is well trained.

- **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:

```
#Visualizing 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(x1.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.

Artificial Neural Networks

Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples. Algorithms such as BACKPROPAGATION use gradient descent to tune network parameters to best fit a training set of input-output pairs.

ANN learning is robust to errors in the training data and has been successfully applied to problems such as interpreting visual scenes, speech recognition, and learning robot control strategies.

To develop a feel for this analogy, let us consider a few facts from neurobiology. The human brain, for example, is estimated to contain a densely interconnected network of approximately 1011 neurons, each connected, on average, to lo4 others. Neuron activity is typically excited or inhibited through connections to other neurons. The fastest neuron switching times are known to be on the order of loe3 seconds--quite slow compared to computer switching speeds of 10-lo seconds. Yet humans are able to make surprisingly complex decisions, surprisingly quickly. For example, it requires approximately lo-' seconds to visually recognize your mother. Notice the sequence of neuron firings that can take place during this 10-'-second interval cannot possibly be longer than a few hundred steps, given the switching speed of single neurons. This observation has led many to speculate that the information-processing abilities of biological neural systems must follow from highly parallel processes operating on representations that are distributed over many neurons.

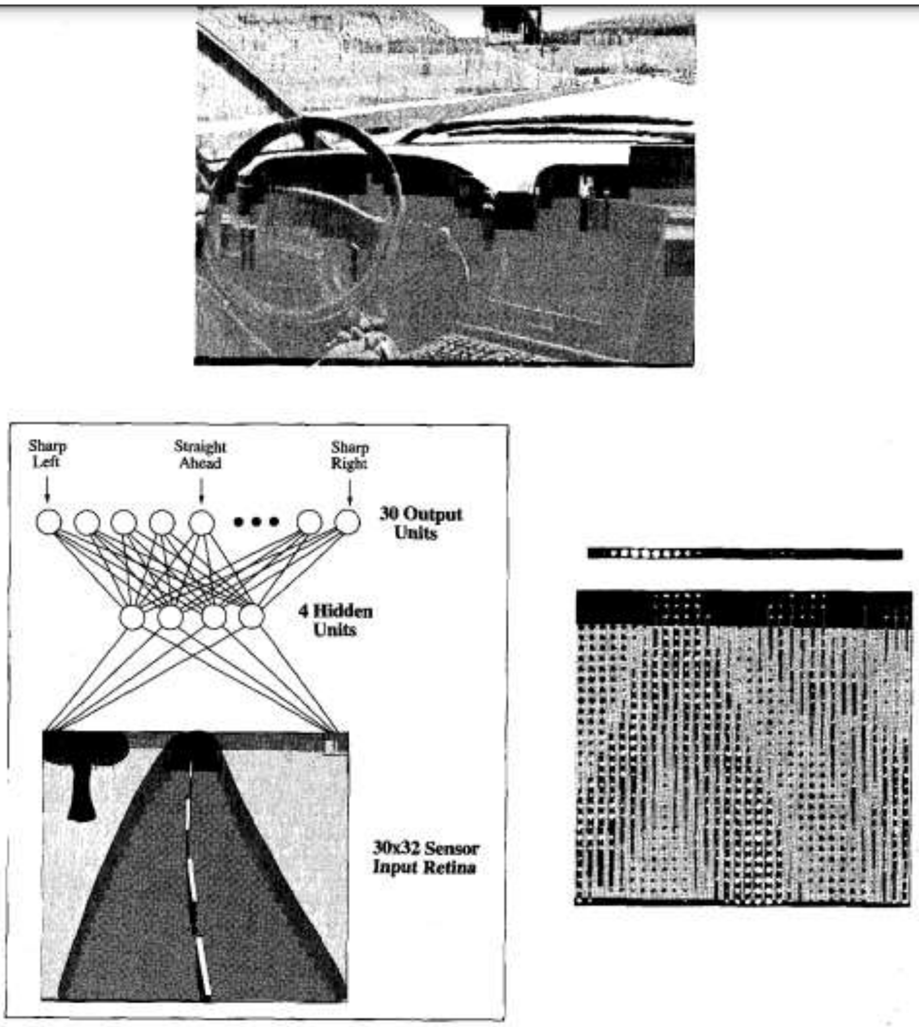


FIGURE 4.1
Neural network learning to steer an autonomous vehicle. The ALVINN system uses BACKPROPAGATION to learn to steer an autonomous vehicle (photo at top) driving at speeds up to 70 miles per hour. The diagram on the left shows how the image of a forward-mounted camera is mapped to 960 neural network inputs, which are fed forward to 4 hidden units, connected to 30 output units. Network outputs encode the commanded steering direction. The figure on the right shows weight values for one of the hidden units in this network. The 30 × 32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights. The weights from this hidden unit to the 30 output units are depicted by the smaller rectangular block directly above the large block. As can be seen from these output weights, activation of this particular hidden unit encourages a turn toward the left.

Figure 4.1 illustrates the neural network representation used in one version of the ALVINN system, and illustrates the kind of representation typical of many ANN systems. The network is shown on the left side of the figure, with the input camera image depicted below it. Each node (i.e., circle) in the network diagram corresponds to the output of a single network unit, and the lines entering the node from below are its inputs. As can be seen, there are four units that receive inputs directly from all of the 30 x 32 pixels in the image. These are called "**hidden**" units because their output is available only within the network and is not available as part of the global network output. Each of these four hidden units computes a single real-valued output based on a weighted combination of its 960 inputs. These hidden unit outputs are then used as inputs to a second layer of 30 "output" units. Each output unit corresponds to a particular steering direction, and the output values of these units determine which steering direction is recommended most strongly.

The **BACKPROPAGATION** algorithm is the most commonly used ANN learning technique. It is appropriate for problems with the following characteristics:

- Instances are represented by many attribute-value pairs. The target function to be learned is defined over instances that can be described by a vector of predefined features, such as the pixel values in the ALVINN example. These input attributes may be highly correlated or independent of one another. Input values can be any real values.
- The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes. For example, in the ALVINN system the output is a vector of 30 attributes, each corresponding to a recommendation regarding the steering direction. The value of each output is some real number between 0 and 1, which in this case corresponds to the confidence in predicting the corresponding steering direction. We can also train a single network to output both the steering command and suggested acceleration, simply by concatenating the vectors that encode these two output predictions.
- The training examples may contain errors. ANN learning methods are quite robust to noise in the training data.
- Long training times are acceptable. Network training algorithms typically require longer training times than, say, decision tree learning algorithms. Training times can range from a few seconds to many hours, depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.
- Fast evaluation of the learned target function may be required. Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast. For

example, ALVINN applies its neural network several times per second to continually update its steering command as the vehicle drives forward.

- The ability of humans to understand the learned target function is not important. The weights learned by neural networks are often difficult for humans to interpret. Learned neural networks are less easily communicated to humans than learned rules.

https://www.javatpoint.com/perceptron-in-machine-learning

What is the Perceptron model in Machine Learning?

Perceptron is Machine Learning algorithm for supervised learning of various binary classification tasks. Further, ***Perceptron is also understood as an Artificial Neuron or neural network unit that helps to detect certain input data computations in business intelligence***.

Perceptron model is also treated as one of the best and simplest types of Artificial Neural networks. However, it is a supervised learning algorithm of binary classifiers. Hence, we can consider it as a single-layer neural network with four main parameters, i.e., **input values, weights and Bias, net sum, and an activation function.**

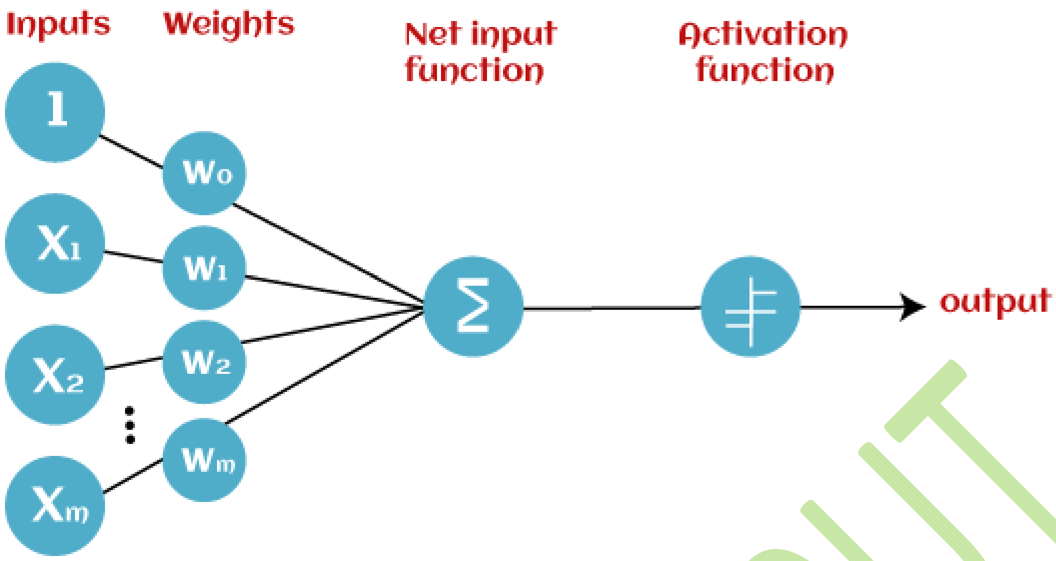
What is Binary classifier in Machine Learning?

In Machine Learning, binary classifiers are defined as the function that helps in deciding whether input data can be represented as vectors of numbers and belongs to some specific class.

Binary classifiers can be considered as linear classifiers. In simple words, we can understand it as a ***classification algorithm that can predict linear predictor function in terms of weight and feature vectors.***

Basic Components of Perceptron

Mr. Frank Rosenblatt invented the perceptron model as a binary classifier which contains three main components. These are as follows:



- **Input Nodes or Input Layer:**

This is the primary component of Perceptron which accepts the initial data into the system for further processing. Each input node contains a real numerical value.

- **Wight and Bias:**

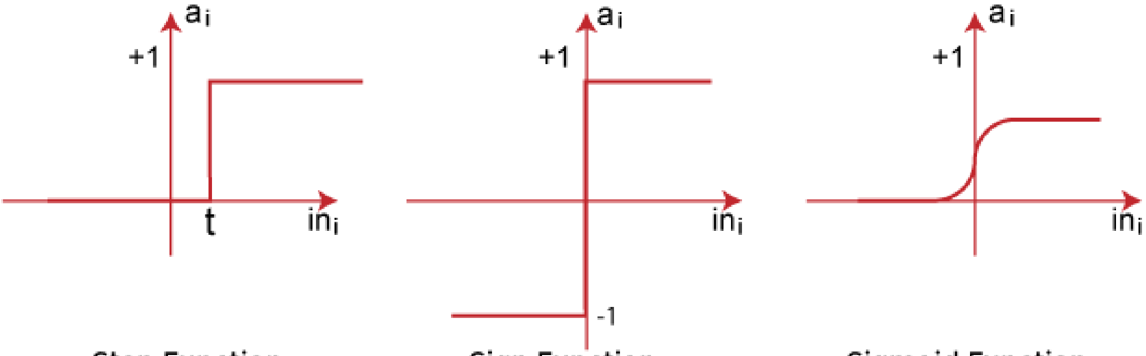
Weight parameter represents the strength of the connection between units. This is another most important parameter of Perceptron components. Weight is directly proportional to the strength of the associated input neuron in deciding the output. Further, Bias can be considered as the line of intercept in a linear equation.

- **Activation Function:**

These are the final and important components that help to determine whether the neuron will fire or not. Activation Function can be considered primarily as a step function.

Types of Activation functions:

- Sign function
- Step function, and
- Sigmoid function

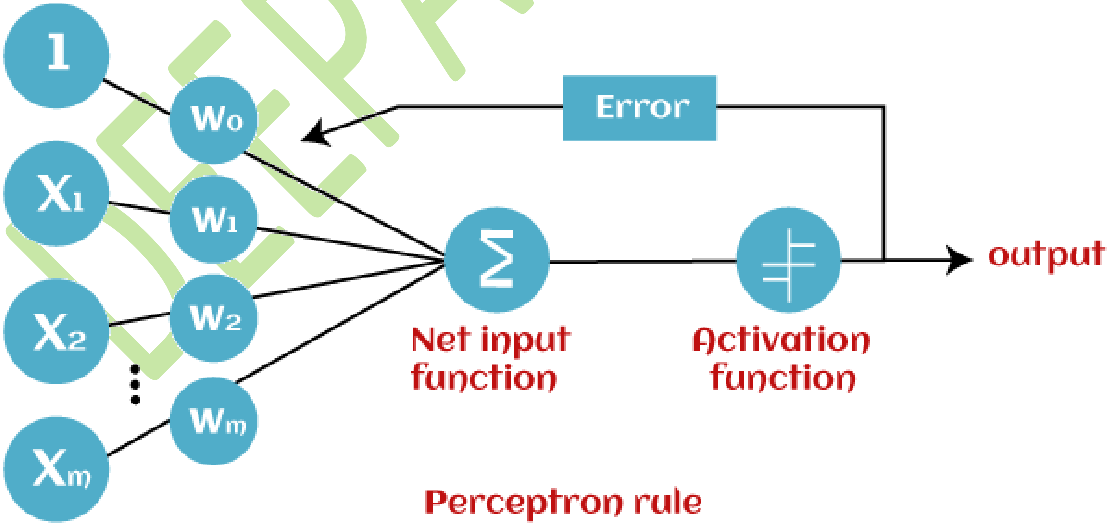


Step Function Sign Function Sigmoid Function

The data scientist uses the activation function to take a subjective decision based on various problem statements and forms the desired outputs. Activation function may differ (e.g., Sign, Step, and Sigmoid) in perceptron models by checking whether the learning process is slow or has vanishing or exploding gradients.

How does Perceptron work?

In Machine Learning, Perceptron is considered as a single-layer neural network that consists of four main parameters named input values (Input nodes), weights and Bias, net sum, and an activation function. The perceptron model begins with the multiplication of all input values and their weights, then adds these values together to create the weighted sum. Then this weighted sum is applied to the activation function 'f' to obtain the desired output. This activation function is also known as the **step function** and is represented by **'f'**.



Perceptron rule

This step function or Activation function plays a vital role in ensuring that output is mapped between required values (0,1) or (-1,1). It is important to note that the weight of input is indicative of the strength of a node. Similarly, an input's bias value gives the ability to shift the activation function curve up or down.

Perceptron model works in two important steps as follows:

Step-1

In the first step first, multiply all input values with corresponding weight values and then add them to determine the weighted sum. Mathematically, we can calculate the weighted sum as follows:

$\sum wi*xi = x1*w1 + x2*w2 +\dots wn*xn$

Add a special term called **bias 'b'** to this weighted sum to improve the model's performance.

$\sum wi*xi + b$

Step-2

In the second step, an activation function is applied with the above-mentioned weighted sum, which gives us output either in binary form or a continuous value as follows:

$Y = f(\sum wi*xi + b)$

Types of Perceptron Models

Based on the layers, Perceptron models are divided into two types. These are as follows:

1. Single-layer Perceptron Model
2. Multi-layer Perceptron model

Single Layer Perceptron Model:

This is one of the easiest Artificial neural networks (ANN) types. A single-layered perceptron model consists feed-forward network and also includes a threshold transfer function inside the model. The main objective of the single-layer perceptron model is to analyze the linearly separable objects with binary outcomes.

In a single layer perceptron model, its algorithms do not contain recorded data, so it begins with inconstantly allocated input for weight parameters. Further, it sums up all inputs (weight). After adding all inputs, if the total sum of all inputs is more than a pre-determined value, the model gets activated and shows the output value as +1.

If the outcome is same as pre-determined or threshold value, then the performance of this model is stated as satisfied, and weight demand does not change. However, this model consists of a few discrepancies triggered when multiple weight inputs values are fed into the model. Hence, to find desired output and minimize errors, some changes should be necessary for the weights input.

"Single-layer perceptron can learn only linearly separable patterns."

Multi-Layered Perceptron Model:

Like a single-layer perceptron model, a multi-layer perceptron model also has the same model structure but has a greater number of hidden layers.

The multi-layer perceptron model is also known as the Backpropagation algorithm, which executes in two stages as follows:

- **Forward Stage:** Activation functions start from the input layer in the forward stage and terminate on the output layer.
- **Backward Stage:** In the backward stage, weight and bias values are modified as per the model's requirement. In this stage, the error between actual output and demanded originated backward on the output layer and ended on the input layer.

Hence, a multi-layered perceptron model has considered as multiple artificial neural networks having various layers in which activation function does not remain linear, similar to a single layer perceptron model. Instead of linear, activation function can be executed as sigmoid, TanH, ReLU, etc., for deployment.

A multi-layer perceptron model has greater processing power and can process linear and non-linear patterns. Further, it can also implement logic gates such as AND, OR, XOR, NAND, NOT, XNOR, NOR.

Advantages of Multi-Layer Perceptron:

- A multi-layered perceptron model can be used to solve complex non-linear problems.
- It works well with both small and large input data.
- It helps us to obtain quick predictions after the training.
- It helps to obtain the same accuracy ratio with large as well as small data.

Disadvantages of Multi-Layer Perceptron:

- In Multi-layer perceptron, computations are difficult and time-consuming.

- In multi-layer Perceptron, it is difficult to predict how much the dependent variable affects each independent variable.
- The model functioning depends on the quality of the training.

Perceptron Function

Perceptron function "f(x)" can be achieved as output by multiplying the input 'x' with the learned weight coefficient 'w'.

Mathematically, we can express it as follows:

$f(x)=1$; if $w.x+b>0$

otherwise, $f(x)=0$

- 'w' represents real-valued weights vector
- 'b' represents the bias
- 'x' represents a vector of input x values.

Characteristics of Perceptron

The perceptron model has the following characteristics.

1. Perceptron is a machine learning algorithm for supervised learning of binary classifiers.
2. In Perceptron, the weight coefficient is automatically learned.
3. Initially, weights are multiplied with input features, and the decision is made whether the neuron is fired or not.
4. The activation function applies a step rule to check whether the weight function is greater than zero.
5. The linear decision boundary is drawn, enabling the distinction between the two linearly separable classes +1 and -1.
6. If the added sum of all input values is more than the threshold value, it must have an output signal; otherwise, no output will be shown.

Limitations of Perceptron Model

A perceptron model has limitations as follows:

- The output of a perceptron can only be a binary number (0 or 1) due to the hard limit transfer function.
- Perceptron can only be used to classify the linearly separable sets of input vectors. If input vectors are non-linear, it is not easy to classify them properly.

Future of Perceptron

The future of the Perceptron model is much bright and significant as it helps to interpret data by building intuitive patterns and applying them in the future. Machine learning is a rapidly growing technology of Artificial Intelligence that is continuously evolving and in the developing phase; hence the future of perceptron technology will continue to support and facilitate analytical behavior in machines that will, in turn, add to the efficiency of computers.

The perceptron model is continuously becoming more advanced and working efficiently on complex problems with the help of artificial neurons.

https://medium.com/edureka/backpropagation-bd2cf8fdde81

What Is Backpropagation?

Backpropagation is a supervised learning algorithm, for training Multi-layer Perceptrons (Artificial Neural Networks).

Why We Need Backpropagation?

While designing a Neural Network, in the beginning, we initialize weights with some random values or any variable for that fact.

Now obviously, we are not *superhuman*. So, it's not necessary that whatever weight values we have selected will be correct, or it fits our model the best.

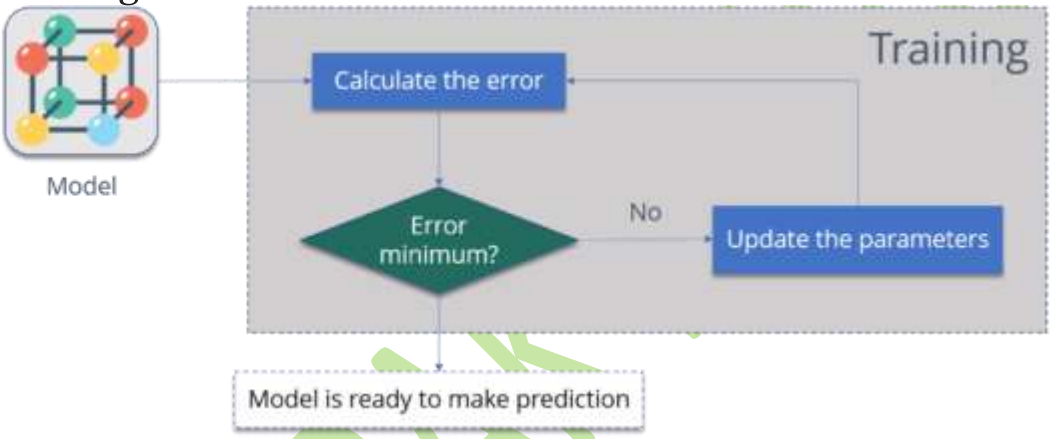
Okay, fine, we have selected some weight values in the beginning, but our model output is way different than our actual output i.e. the error value is huge.

Now, how will you reduce the error?

Basically, what we need to do, we need to somehow explain the model to change the parameters (weights), such that error becomes minimum.

Let's put it in another way, we need to train our model.

One way to train our model is called as Backpropagation. Consider the diagram below:



Let me summarize the steps for you:

- **Calculate the error** — How far is your model output from the actual output.
- **Error minimum** — Check whether the error is minimized or not.
- **Update the parameters** — If the error is huge then, update the parameters (weights and biases). After that again check the error. Repeat the process until the error becomes minimum.

- **Model is ready to make a prediction** — Once the error becomes minimum, you can feed some inputs to your model and it will produce the output.

I am pretty sure, now you know, why we need Backpropagation or why and what is the meaning of training a model.

Now is the correct time to understand what is Backpropagation.

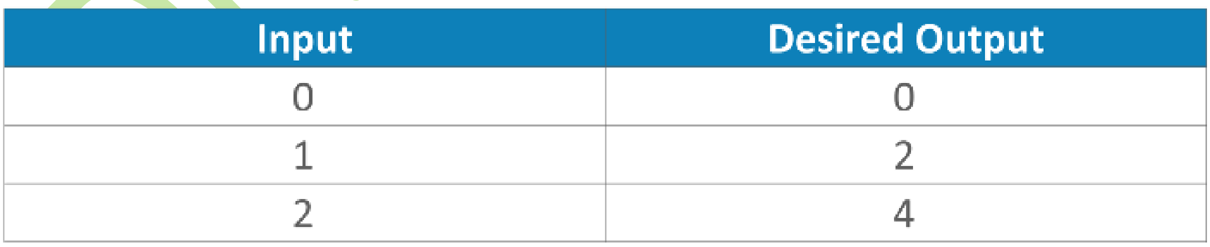
What is Backpropagation?

The Backpropagation algorithm looks for the minimum value of the error function in weight space using a technique called the delta rule or gradient descent. The weights that minimize the error function is then considered to be a solution to the learning problem.

Let’s understand how it works with an example:

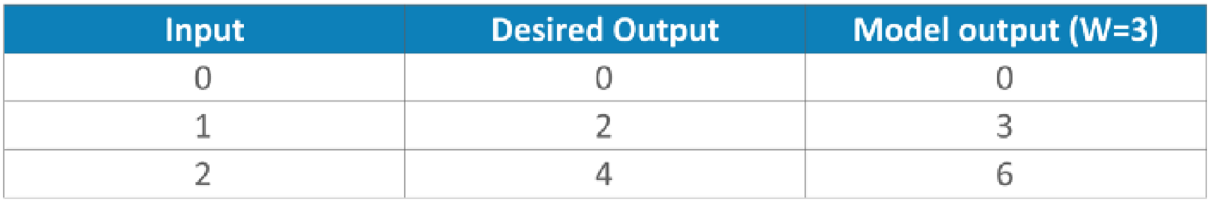
You have a dataset, which has labels.

Consider the below table:



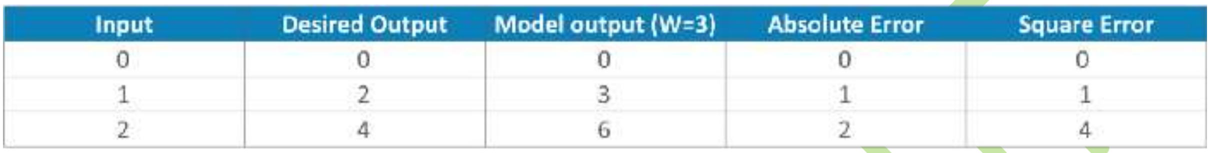
Input	Desired Output
0	0
1	2
2	4

Now the output of your model when ‘W” value is 3:



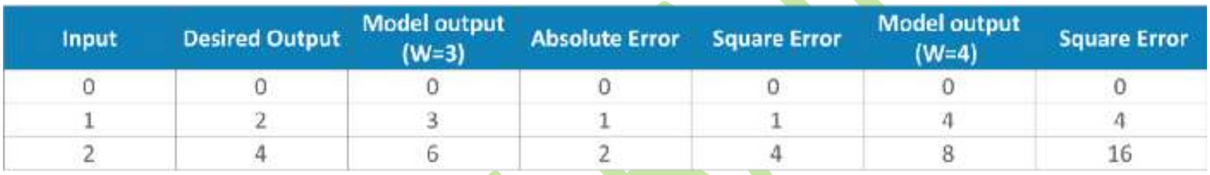
Input	Desired Output	Model output (W=3)
0	0	0
1	2	3
2	4	6

Notice the difference between the actual output and the desired output:



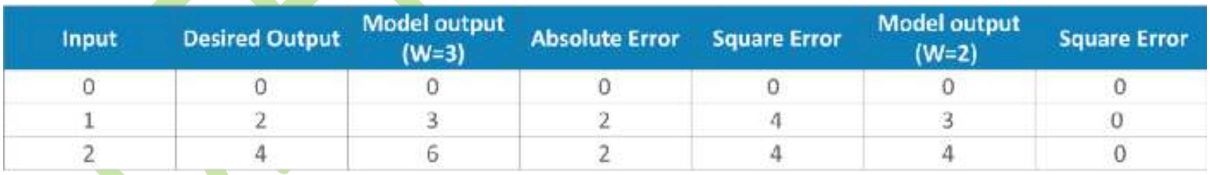
Input	Desired Output	Model output (W=3)	Absolute Error	Square Error
0	0	0	0	0
1	2	3	1	1
2	4	6	2	4

Let’s change the value of ‘W’. Notice the error when ‘W’ = ‘4’



Input	Desired Output	Model output (W=3)	Absolute Error	Square Error	Model output (W=4)	Square Error
0	0	0	0	0	0	0
1	2	3	1	1	4	4
2	4	6	2	4	8	16

Now if you notice, when we increase the value of ‘W’ the error has increased. So, obviously there is no point in increasing the value of ‘W’ further. But, what happens if I decrease the value of ‘W’? Consider the table below:



Input	Desired Output	Model output (W=3)	Absolute Error	Square Error	Model output (W=2)	Square Error
0	0	0	0	0	0	0
1	2	3	2	4	3	0
2	4	6	2	4	4	0

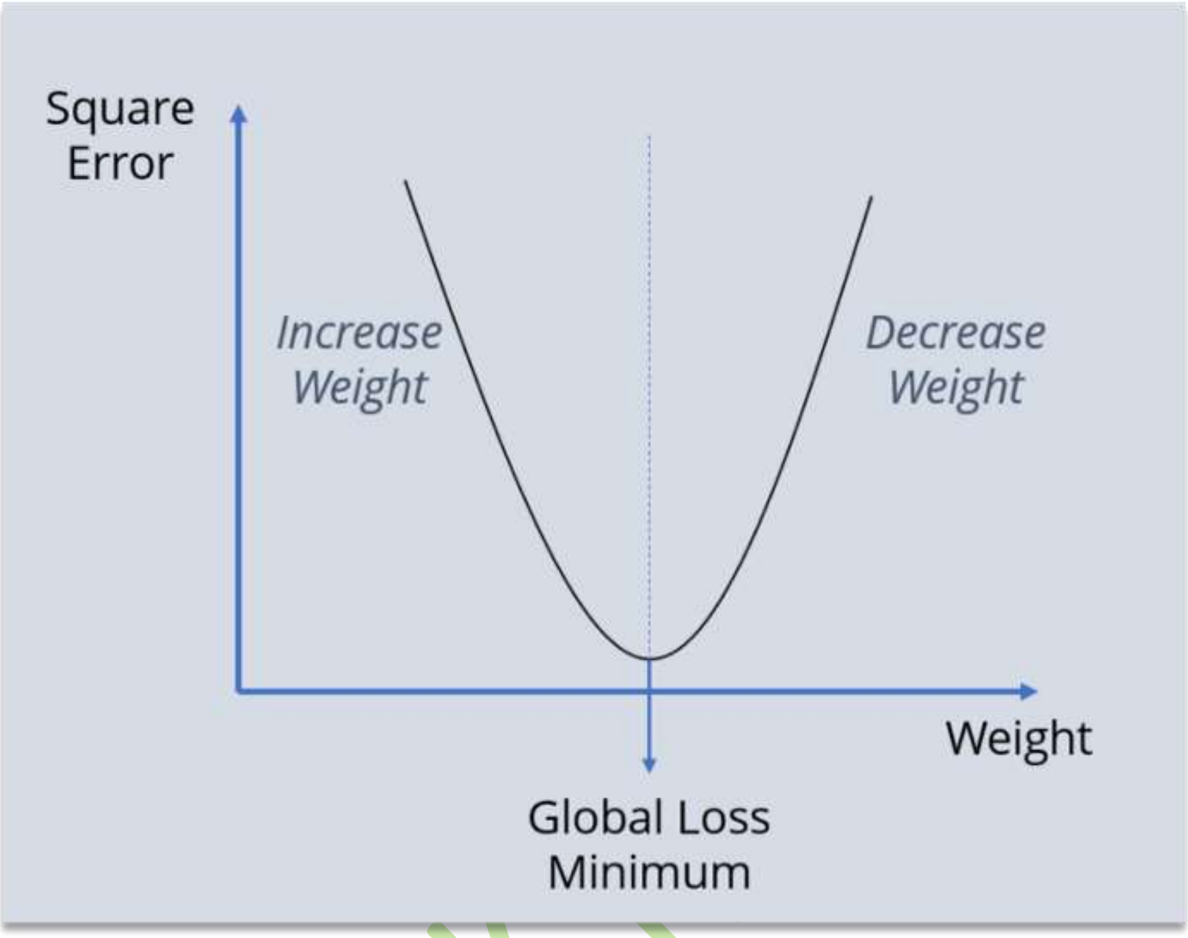
Now, what we did here:

- We first initialized some random value to ‘W’ and propagated forward.

- Then, we noticed that there is some error. To reduce that error, we propagated backward and increased the value of 'W'.
- After that, also we noticed that the error has increased. We came to know that, we can't increase the 'W' value.
- So, we again propagated backward and we decreased 'W' value.
- Now, we noticed that the error has reduced.

So, we are trying to get the value of weight such that the error becomes minimum. Basically, we need to figure out whether we need to increase or decrease the weight value. Once we know that, we keep on updating the weight value in that direction until error becomes minimum. You might reach a point, where if you further update the weight, the error will increase. At that time you need to stop, and that is your final weight value.

Consider the graph below:



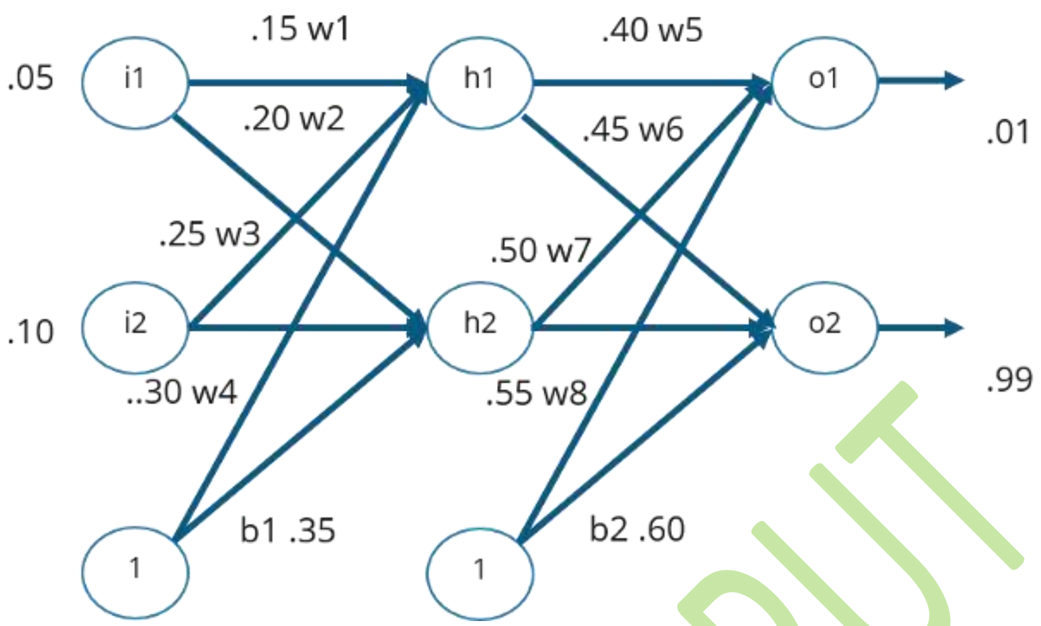
We need to reach the ‘Global Loss Minimum’.

This is nothing but Backpropagation.

Let’s now understand the math behind Backpropagation.

How Backpropagation Works?

Consider the below Neural Network:



The above network contains the following:

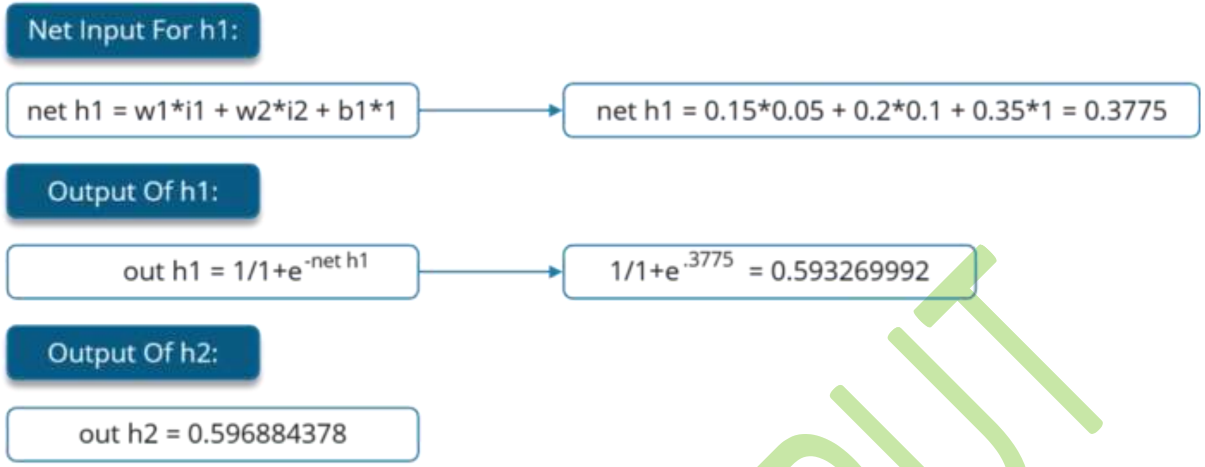
- two inputs
- two hidden neurons
- two output neurons
- two biases

Below are the steps involved in Backpropagation:

- Step — 1: Forward Propagation
- Step — 2: Backward Propagation
- Step — 3: Putting all the values together and calculating the updated weight value

Step — 1: Forward Propagation

We will start by propagating forward.



Net Input For h1:

net h1 = w1*i1 + w2*i2 + b1*1 → net h1 = 0.15*0.05 + 0.2*0.1 + 0.35*1 = 0.3775

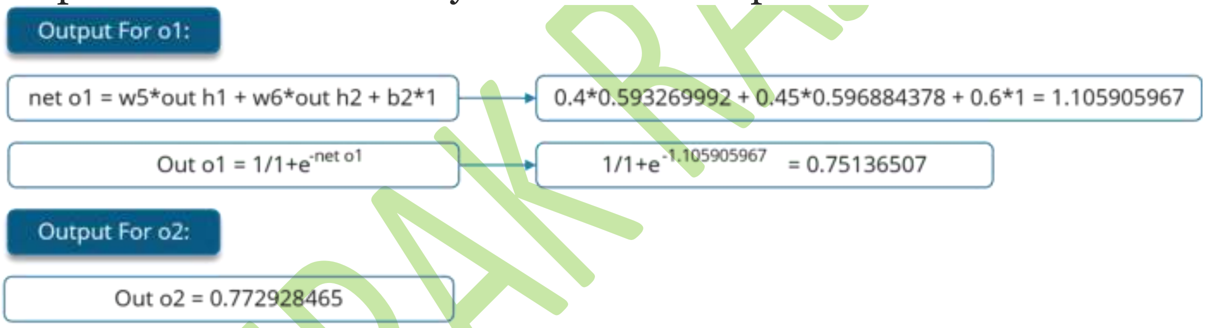
Output Of h1:

out h1 = $1/1+e^{-net\ h1}$ → $1/1+e^{.3775}$ = 0.593269992

Output Of h2:

out h2 = 0.596884378

We will repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.



Output For o1:

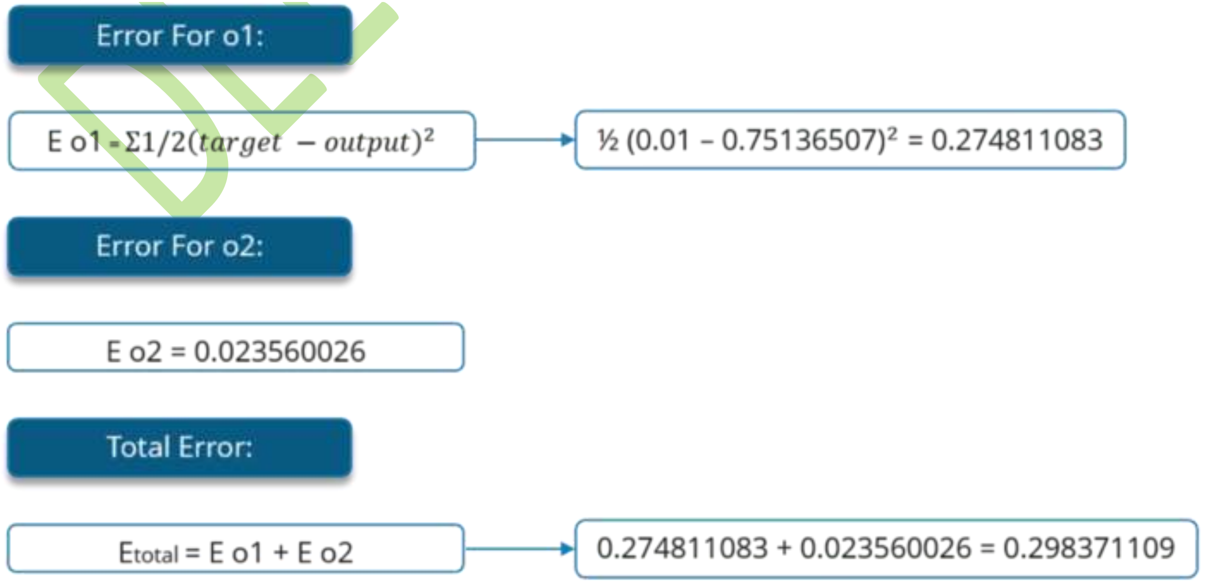
net o1 = w5*out h1 + w6*out h2 + b2*1 → 0.4*0.593269992 + 0.45*0.596884378 + 0.6*1 = 1.105905967

Out o1 = $1/1+e^{-net\ o1}$ → $1/1+e^{-1.105905967}$ = 0.75136507

Output For o2:

Out o2 = 0.772928465

Now, let's see what is the value of the error:



Error For o1:

E o1 = $\Sigma 1/2(target\ - output)^2$ → ½ (0.01 – 0.75136507)² = 0.274811083

Error For o2:

E o2 = 0.023560026

Total Error:

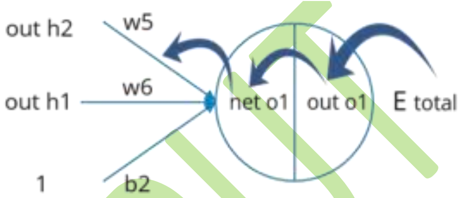
E_{total} = E o1 + E o2 → 0.274811083 + 0.023560026 = 0.298371109

Step — 2: Backward Propagation

Now, we will propagate backwards. This way we will try to reduce the error by changing the values of weights and biases.

Consider W5, we will calculate the rate of change of error w.r.t change in weight W5.

$$\frac{\delta Etotal}{\delta w5} = \frac{\delta Etotal}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w5}$$



Since we are propagating backwards, first thing we need to do is, calculate the change in total errors w.r.t the output O1 and O2.

$$E_{total} = 1/2(\text{target o1} - \text{out o1})^2 + 1/2(\text{target o2} - \text{out o2})^2$$

$$\frac{\delta Etotal}{\delta out\ o1} = -(\text{target o1} - \text{out o1}) = -(0.01 - 0.75136507) = 0.74136507$$

Now, we will propagate further backwards and calculate the change in output O1 w.r.t to its total net input.

$$\text{out o1} = 1/1{+}e^{-neto1}$$

$$\frac{\delta out\ o1}{\delta net\ o1} = \text{out o1 (1 - out o1)} = 0.75136507\ (1 - 0.75136507) = 0.186815602$$

Let's see now how much does the total net input of O1 changes w.r.t W5?

$$\text{net o1} = \text{w5} * \text{out h1} + \text{w6} * \text{out h2} + \text{b2} * 1$$

$$\frac{\delta net\ o1}{\delta w5} = 1 * \text{out h1}\ w5^{(1-1)} + 0 + 0 = 0.593269992$$

Step — 3: Putting all the values together and calculating the updated weight value

Now, let's put all the values together:

$$\boxed{\frac{\delta Etotal}{\delta w5} = \frac{\delta Etotal}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w5}} \rightarrow 0.082167041$$

Let's calculate the updated value of W5:

$$\boxed{w5^{+} = w5 - n\frac{\delta Etotal}{\delta w5}} \rightarrow \boxed{w5^{+} = 0.4 - 0.5 * 0.082167041}$$

$$\text{Updated w5} \rightarrow 0.35891648$$

- Similarly, we can calculate the other weight values as well.
- After that we will again propagate forward and calculate the output. Again, we will calculate the error.
- If the error is minimum we will stop right there, else we will again propagate backwards and update the weight values.
- This process will keep on repeating until error becomes minimum.

Support Vector Machine

https://youtu.be/VJ7WF_Dr3Os

Support Vector Machine Example

https://youtu.be/ivPoCcYfFAw

K-means Clustering Algorithm

https://youtu.be/J19eBNfC4dM

K-means Clustering Algorithm Example

https://youtu.be/FllcPjvztTI