St Joseph's University, Bangalore

By

R Nithish

Implementing Custom Machine Learning Algorithms for Enhanced Performance and Understanding

[1]**Nithish R**
Research Scholar
St. Joseph's University
Bangalore - 27, Karnataka, India
rnithish248@gmail.com
+91 9663614603

[2]**Prem Sagar**
Assistant Professor
Department of Computer Science
St. Joseph's University
Bangalore - 27, Karnataka, India
Postal Address:
#3/2, 6th cross, Seppings Road, Shivaji Nagar
Bangalore - 01, Karnataka, India
premg563@gmail.com
prem.sagar_@sju.edu.in
+91 7795799358

Abstract:

In the contemporary landscape of artificial intelligence (AI), there's a notable emphasis on Generative AI, a subset of Deep Learning. However, delving into the fundamental mathematics underlying Machine Learning (ML) and Deep Learning (DL) is crucial. This understanding lays the groundwork for the introduction of new architectures like the Transformer. Many large language models (LLMs), including ChatGPT, heavily rely on the Transformer architecture.

The Transformer was introduced in the research paper titled **Attention Is All You Need** by Google researchers. Their seminal work highlights the importance of understanding that the context of a word depends not only on the words before it but also on those after it.

This research paper aims to explore various ML and DL algorithms by implementing them from scratch, without relying on third-party libraries like sklearn. By doing so, we aim to comprehend these architectures deeply and independently. Through this process, we engage in the mathematical intricacies, enabling us to grasp and innovate upon revolutionary architectures like the Transformer.

Expanding upon these foundational concepts and implementations is instrumental in our quest to advance AI research and development, facilitating the refinement of existing architectures and the creation of novel solutions.

Furthermore, a deeper understanding of these mathematical principles could lead to more efficient and effective AI systems across various domains, such as natural language processing, computer vision, and reinforcement learning. This knowledge can have a transformative impact on real-world problems, including healthcare, finance, and climate modeling, where advanced AI architectures could revolutionize decision-making processes.

Encouraging open-source contributions and collaborative research efforts is vital for advancing the field of AI. By fostering knowledge sharing and interdisciplinary collaboration, researchers can collectively address some of the most pressing challenges and unlock new opportunities in artificial intelligence.

1) Linear regression

Linear Regression is the most basic algorithm in Machine Learning. It is a regression algorithm, which means that it is useful when we are required to predict continuous values, that is, the output variable ‘y’ is continuous in nature.

A few examples of the regression problem can be the following

1. “What is the market value of the house?”

2. “Stock price prediction”

3. “Sales of a shop”

4. “Predicting height of a person”

Terms to be used here:

1. Features - These are the independent variables in any dataset represented by **X1, X2, X3, X4**,... In for 'n' features.

2. Target / Output Variable - This is the dependent variable whose value depends on the independent variable by a relation (given below) and is represented by 'y'.

3. Function or Hypothesis of Linear Regression is represented by - $\mathbf{y = M_1.X_1 + M_2.X_2 + M_3.X_3 + ... + Mn.xn + b}$

4. Intercept - Here b is the intercept of the line. We usually include this 'b' in the equation of 'm' and take 'x' values for that 'm' to be 1. So the modified form of the above equation is as follows: y = mx Where $\mathbf{mx = M_1.X_1 + M_2.X_2 + M_3.X_3 + ... + M_n. X_n + M_n+1-X_n+1}$. Here mn+1 is b and Xn+1 = 1

5. Training Data - This data contains a set of dependent variables that is 'x' and a set of output variables, 'y'. This data is given to the machine for it to learn or get trained on some function (here the function is the equation given above) such that in future on giving some new values of 'x', our machine is able to predict values of 'y' based on that function.

6. Testing Data - Once our model is ready, we need to get an idea of how well it performs. For this we use testing data. The machine is given the value of 'x', on which it predicts the value of 'y'. We then compare the predicted 'y' with the testing 'y' to get an idea of the error and accuracy of our machine.

Linear regression assumes linear relation between x and y. The hypothesis function for linear regression is:

$\mathbf{y = M_1.X_1 + M_2.X_2 + M_3.X_3 + ... + Mn.xn + b}$

where **m1, m2, m3** are called the parameters and b is the intercept of the line. This equation shows that the output variable y is linearly dependent on the features **x1, x2, x3**. The more you are dependent on a particular feature, more will be the value of corresponding m for that feature. We can find out which feature is more important or which feature is more affecting the result by varying the values of m one at a time and see if it is affecting the result, that is, the value of y. So, here in order to predict the values of y for given features values (x values) we use this equation.

But what we are missing here is the values of parameters (m1, m2, m3,... and b). So, we will be using our training data (where the values of x and y are already given) to find out values of parameters and later on predict the value of y for a set of new values of x.

The core idea is to obtain a line that best fits the data. The best fit line is considered to be the line for which the error between the predicted values and the observed values is minimum. It is also called the regression line and the errors are also known as residuals.

Types of Linear Regression

Linear regression can be further divided into two types of the algorithm:

1. Simple Linear Regression: If a single independent variable is used to predict the value of a numerical dependent variable, then such a Linear Regression algorithm is called Simple Linear Regression.

2. Multiple Linear regression: If more than one independent variable is used to predict the value of a numerical dependent variable, then such a Linear Regression algorithm is called Multiple Linear Regression.

Cost Function

1. Sum of residuals ∑(Yactual - Ypredict) - This is usually not used since it might result in cancelling out the positive and negative errors.

2. Sum of the absolute value of residuals ∑ | Yactual - Ypredict | - Taking absolute value would prevent cancellation of positive and negative errors

3. Sum of square of residuals ∑ (Yactual - Ypredict)^2 - This is the method mostly used in practice since here we penalize higher error values much more as compared to smaller ones, so that there is a significant difference between making big errors and small errors, which makes it easy to differentiate and select the best fit line.

Linear Regression Using Sklearn

```
import numpy as np
data = np.loadtxt("data.csv", delimiter = ",")
X = data[:, 0].reshape(-1,1)
Y = data[:, 1]
X.shape
```

This outputs (100,1) which says its 2d array with 100 rows and 1 column

```
from sklearn import model_selection
X_train,  X_test,  Y_train,  Y_test  =  model_selection.train_test_split(X,  Y,
test_size = 0.3)
X_train.shape
```

this outputs (70,1)

Let's Generate the model now

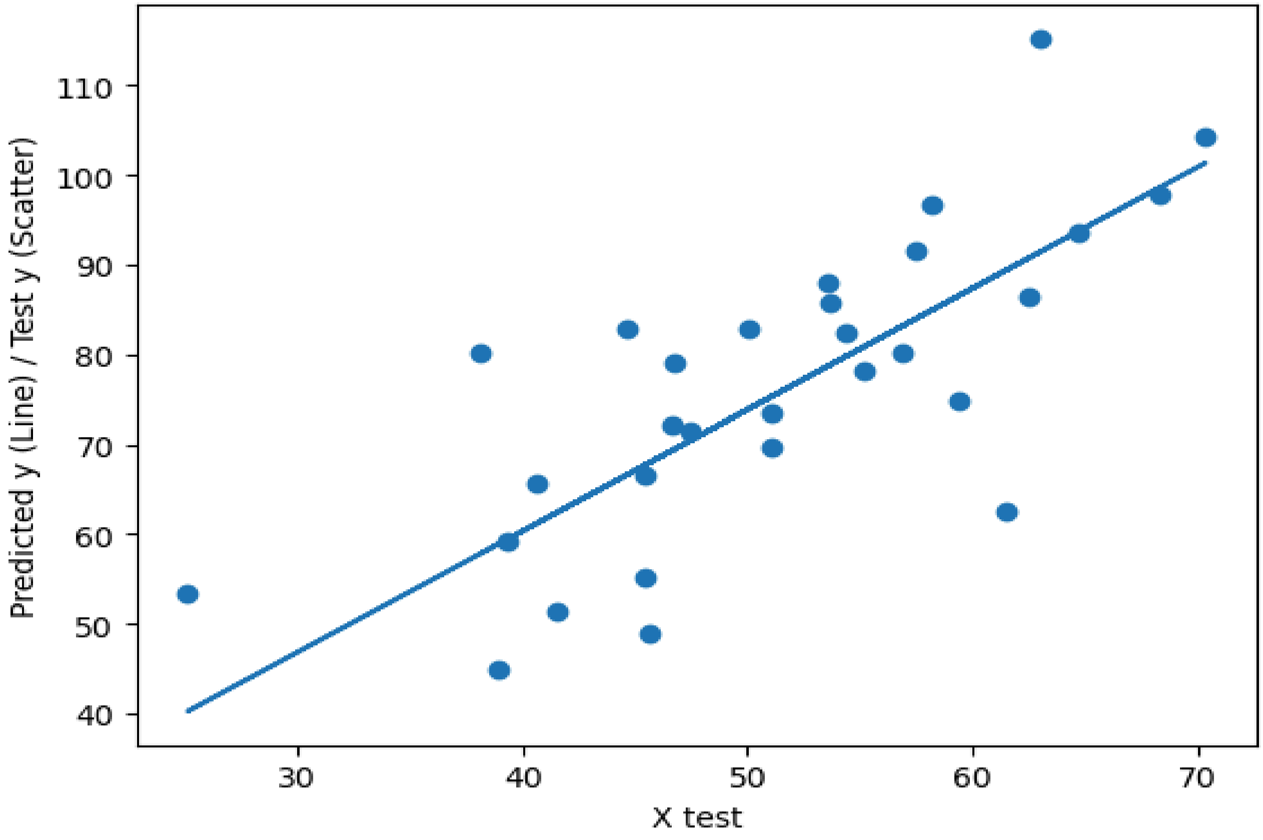
```
from sklearn.linear_model import LinearRegression
reg = LinearRegression()
reg.fit(X_train, Y_train)
y_predict = reg.predict(X_test)
```

Now The Model Is Trained

```
from sklearn.metrics import mean_squared_error

mse_self_implement = mean_squared_error(Y_test, y_predict)

print("MSE", mse_self_implement)

weights = reg.coef_

intercept = reg.intercept_

test_score=reg.score(X_test,Y_test)

train_score=reg.score(X_train, Y_train)

print("train score:", train_score)

print("test score:", test_score)

print("Coeffecient for x is:", weights[0])

print("Intercept value is:", intercept)
```

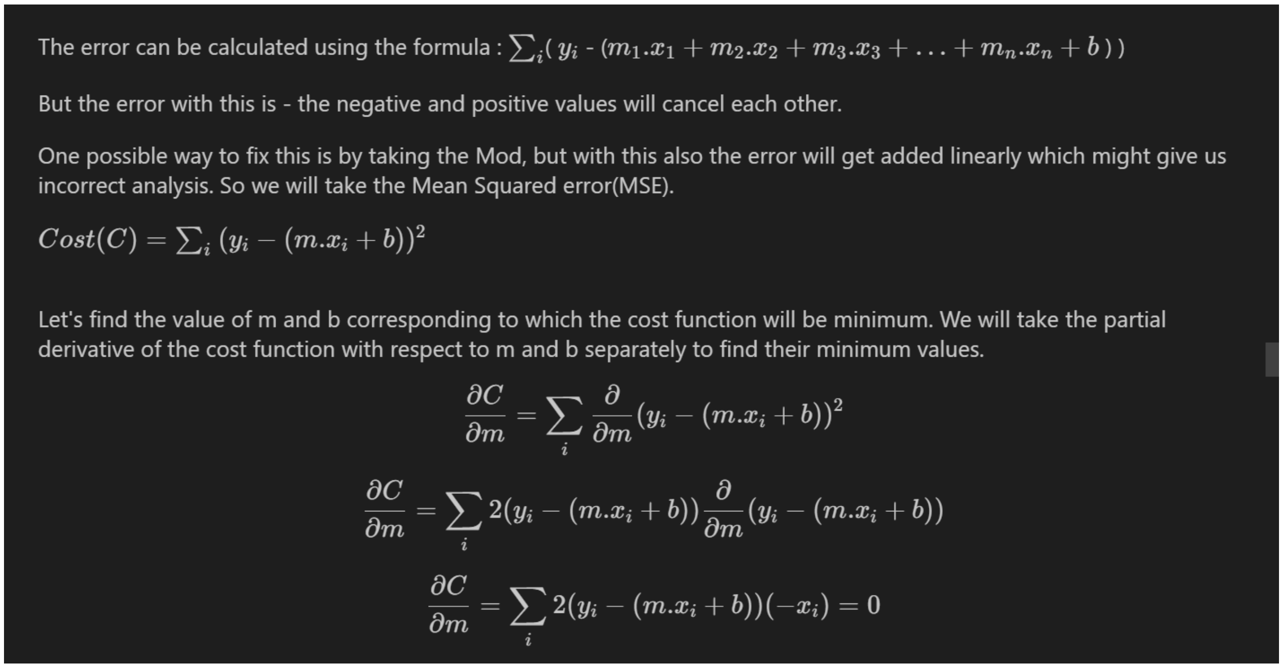
```
MSE 127.56610888204038

train score: 0.6108944484430867

test score: 0.5403990799824994

Coeffecient for x is: 1.3518549830478617

Intercept value is: 6.26343380269239
```

```python
import matplotlib.pyplot as plt

plt.scatter(X_test, Y_test)

plt.plot(X_test, y_predict)

plt.xlabel("X test")

plt.ylabel("Predicted y (Line) / Test y (Scatter)")

plt.show()
```



Writing Own Linear Regression

The different values for coefficient of lines gives different lines of regression, and the cost function is used to find the values of the coefficient for the best fit line. Cost function optimizes the regression coefficients. It measures how a linear regression model is performing.



The error can be calculated using the formula : $\sum_i(y_i$ - $(m_1.x_1 + m_2.x_2 + m_3.x_3 + \ldots + m_n.x_n + b\,)\,)$

But the error with this is - the negative and positive values will cancel each other.

One possible way to fix this is by taking the Mod, but with this also the error will get added linearly which might give us incorrect analysis. So we will take the Mean Squared error(MSE).

$Cost(C) = \sum_i\,(y_i - (m.x_i + b))^2$

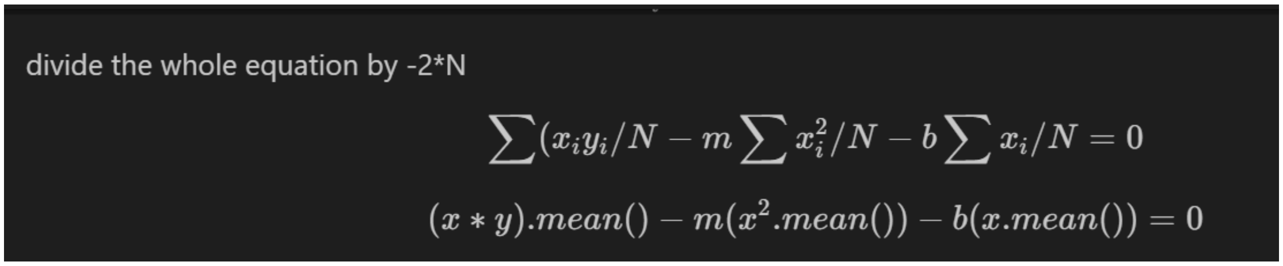
Let's find the value of m and b corresponding to which the cost function will be minimum. We will take the partial derivative of the cost function with respect to m and b separately to find their minimum values.

$$\frac{\partial C}{\partial m} = \sum_i \frac{\partial}{\partial m}(y_i - (m.x_i + b))^2$$

$$\frac{\partial C}{\partial m} = \sum_i 2(y_i - (m.x_i + b))\frac{\partial}{\partial m}(y_i - (m.x_i + b))$$

$$\frac{\partial C}{\partial m} = \sum_i 2(y_i - (m.x_i + b))(-x_i) = 0$$

we now have derivative of cost function with respect to m

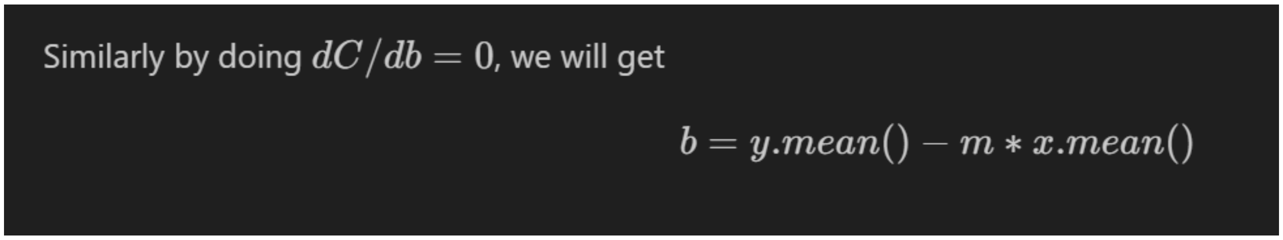


divide the whole equation by -2*N

$$\sum(x_i y_i/N - m\sum x_i^2/N - b\sum x_i/N = 0$$

$$(x * y).mean() - m(x^2.mean()) - b(x.mean()) = 0$$

we divide by n to reduce large values and take average values let this be eq1.

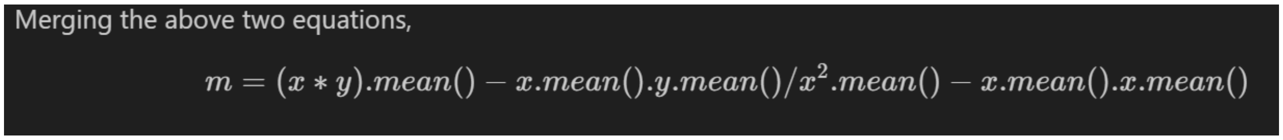


Similarly by doing $dC/db = 0$, we will get

$$b = y.mean() - m * x.mean()$$

We now have a derivative of cost function with respect to b let this be eq2.

let us substitute eq2 in eq1



Merging the above two equations,

$$m = (x * y).mean() - x.mean().y.mean()/x^2.mean() - x.mean().x.mean()$$

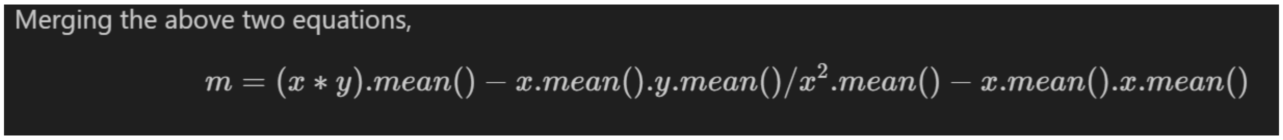
Now We are ready to implement Linear Regression from scratch.

```
import numpy as np
from sklearn import model_selection
import pandas as pd
from sklearn import model_selection
data = np.loadtxt("data.csv", delimiter = ",")
X = data[:, 0]
Y = data[:, 1]
X_train,  X_test,  Y_train,  Y_test  =  model_selection.train_test_split(X,  Y,
test_size = 0.3)
```

we loaded the data and split it into train and test.

```
# This function is used to find the best fit line using the training data

def fit(x_train, y_train):
    num = (x_train * y_train).mean() - x_train.mean() * y_train.mean()
    den = (x_train ** 2).mean() - x_train.mean() ** 2
    m = num / den
    c = y_train.mean() - m * x_train.mean()
    return m, c
```

We used the below formula to arrive at the above code.



Merging the above two equations,

$$m = (x * y).mean() - x.mean().y.mean()/x^2.mean() - x.mean().x.mean()$$

```
def predict(x, m, c):
    return m * x + c
def cost (x, y, m , c):
    return ((y - m * x - c)**2).mean()

def score(y_truth, y_pred):
    u = ((y_truth-y_pred)**2).sum()
    v = ((y_truth-y_truth.mean())**2).sum()
    return 1-(u/v)
```

The m and c calculated by fit() function will be required by predict() function.cost and score function is also calculated programmatically.

```
m, c = fit(X_train, Y_train)
# Test data
y_test_pred = predict(X_test, m, c)
print("Test Score: ",score(Y_test, y_test_pred))

# Train data
y_train_pred = predict(X_train, m, c)
print("Train Score:", score(Y_train, y_train_pred))
print("M:", m)
print("C:", c)
print("Cost on training data:", cost(X_train,Y_train, m, c ))
```

```
Test Score:  0.6735871842919778

Train Score: 0.5663635523498989

M: 1.3507498764664958

C: 6.025311629600068

Cost on training data: 121.36177147943415
```

we can observe these values are very similar to that of sklearn produced output.

Multiple Linear Regression

We implemented simple linear regression which cannot handle multiple features or multiple dependent variables.

Gradient Descent

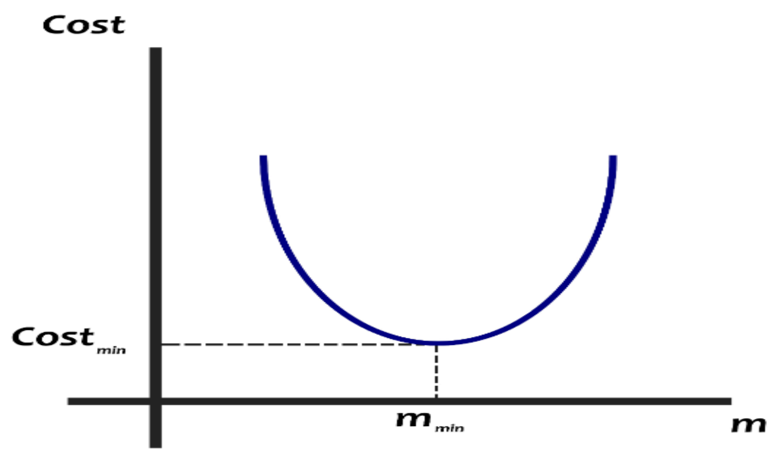
Gradient descent is an optimization algorithm used to find the values of parameters (coefficients) of a function (f) that minimizes a cost function (cost).

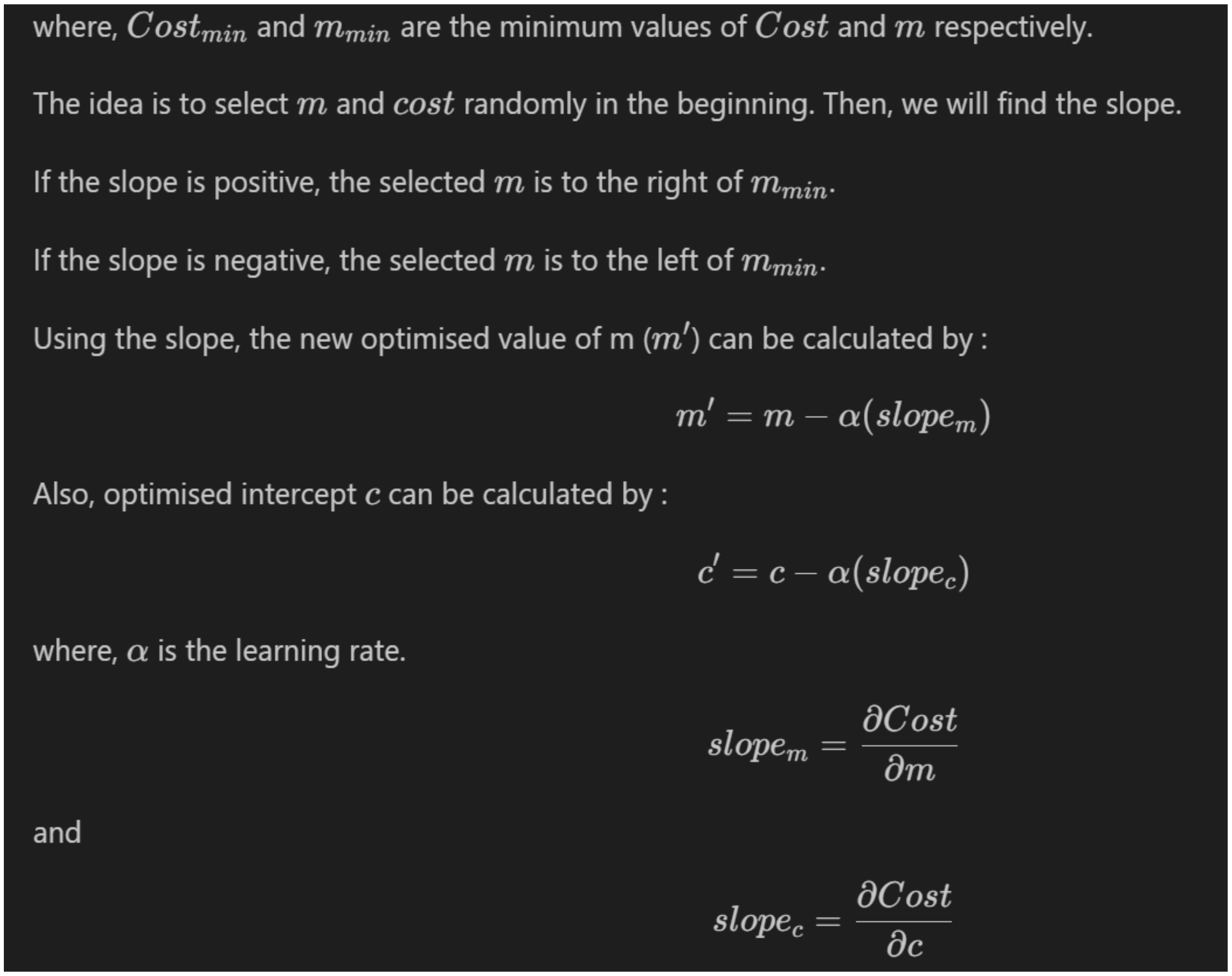
Gradient descent is best used when the parameters cannot be calculated analytically (e.g. using linear algebra) and must be searched for by an optimization algorithm.

Intuition of Gradient Descent

Think of a large bowl like what you would eat cereal out of or store fruit in. This bowl is a plot of the cost function (f) in 3D space.A random position on the surface of the bowl is the cost of the current values of the coefficients (cost). The bottom of the bowl is the cost of the best set of coefficients, the minimum of the function. The goal is to continue to try different values for the coefficients, evaluate their cost and select new coefficients that have a slightly better (lower) cost. Repeating this process enough times will lead to the bottom of the bowl and you will know the values of the coefficients that result in the minimum cost.

The graph of our cost function will look like this :




where, $Cost_{min}$ and m_{min} are the minimum values of $Cost$ and m respectively.

The idea is to select m and $cost$ randomly in the beginning. Then, we will find the slope.

If the slope is positive, the selected m is to the right of m_{min}.

If the slope is negative, the selected m is to the left of m_{min}.

Using the slope, the new optimised value of m (m') can be calculated by :

$$m' = m - \alpha(slope_m)$$

Also, optimised intercept c can be calculated by :

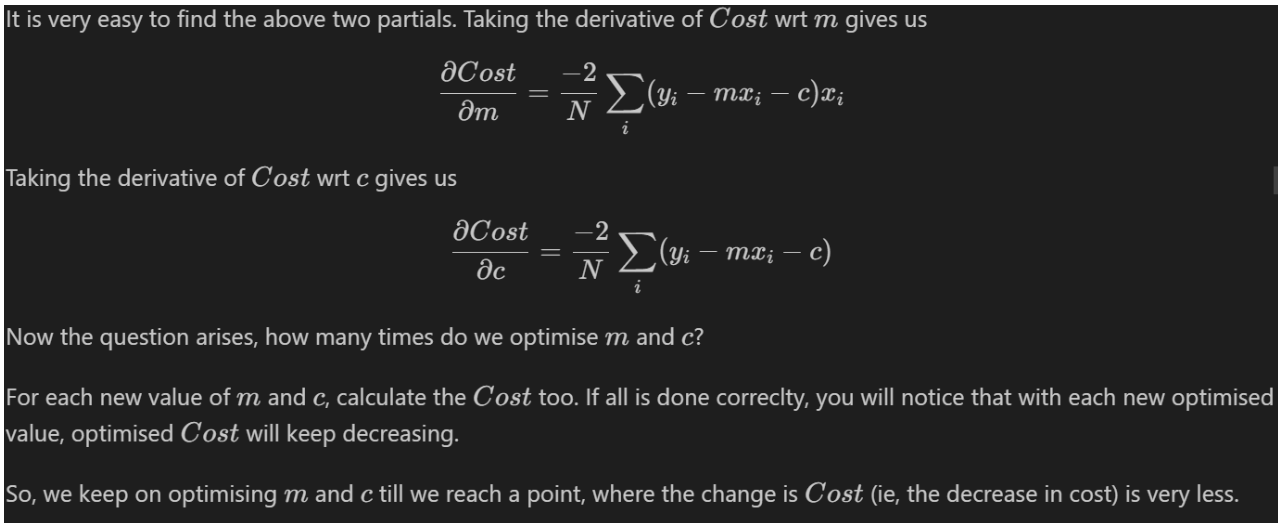
$$c' = c - \alpha(slope_c)$$

where, α is the learning rate.

$$slope_m = \frac{\partial Cost}{\partial m}$$

and

$$slope_c = \frac{\partial Cost}{\partial c}$$


It is very easy to find the above two partials. Taking the derivative of $Cost$ wrt m gives us

$$\frac{\partial Cost}{\partial m} = \frac{-2}{N}\sum_i (y_i - mx_i - c)x_i$$

Taking the derivative of $Cost$ wrt c gives us

$$\frac{\partial Cost}{\partial c} = \frac{-2}{N}\sum_i (y_i - mx_i - c)$$

Now the question arises, how many times do we optimise m and c?

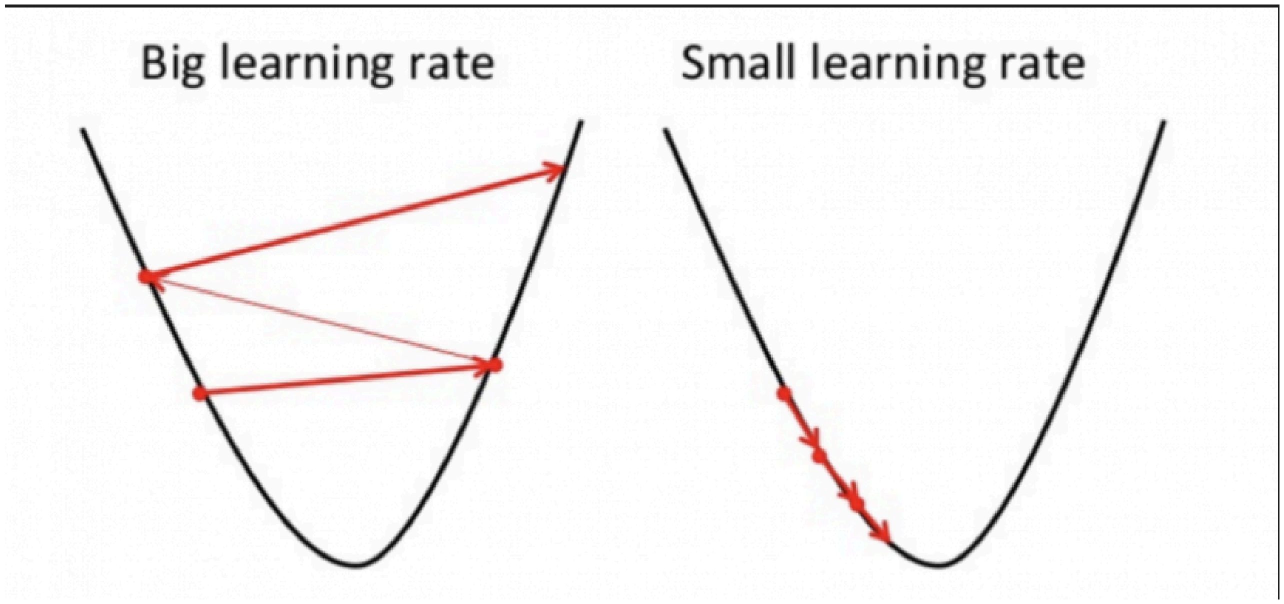
For each new value of m and c, calculate the $Cost$ too. If all is done correclty, you will notice that with each new optimised value, optimised $Cost$ will keep decreasing.

So, we keep on optimising m and c till we reach a point, where the change is $Cost$ (ie, the decrease in cost) is very less.

Learning Rate (α) and its Importance

How big the steps gradient descent takes into the direction of the local minimum are determined by the learning rate, which figures out how fast or slow we will move towards the optimal weights.

For gradient descent to reach the local minimum we must set the learning rate to an appropriate value, which is neither too low nor too high. This is important because if the steps it takes are too big, it may not reach the local minimum because it bounces back and forth between the convex function of gradient descent (see left image below). If we set the learning rate to a very small value, gradient descent will eventually reach the local minimum but that may take a while (see the right image).



So, the learning rate should never be too high or too low for this reason. You can check if your learning rate is doing well by plotting it on a graph.

Adaptive Learning Rate

The performance of the model on the training dataset can be monitored by the learning algorithm and the learning rate can be adjusted in response. This is called an adaptive learning rate. Perhaps the simplest implementation is to make the learning rate smaller once the performance of the model plateaus, such as by decreasing the learning rate by a factor of two.

An adaptive learning rate method will generally outperform a model with a badly configured learning rate.

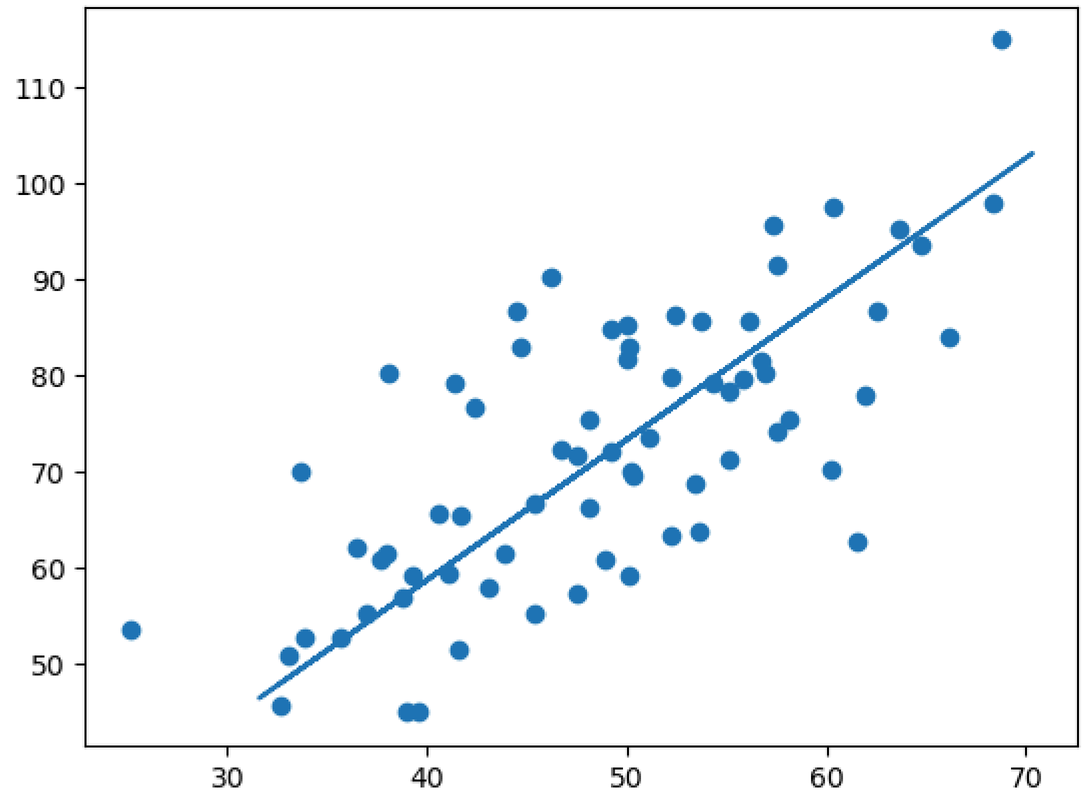
Let's code Multiple Linear Regression Using Gradient Descent.

```python
class LinearRegression:
    def __init__(self,lr=0.0001,n=1000):
        self.lr = lr
        self.iters = n
        self.weights = None
        self.bias = None

    def fit(self,x_train,y_train):
        n_points,n_features=x_train.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for i in range(self.iters):
            y_pred = (np.dot(x_train,self.weights) + self.bias)

            m_slope = (1/n_points) * np.dot((y_pred - y_train) , x_train)
            b_slope = (1/n_points) * sum(y_pred - y_train)

            self.weights -= self.lr * m_slope
            self.bias -= self.lr * b_slope

    def predict(self,x_test):
            y_pred  =  [(np.dot(point.T,self.weights)[0]  +  self.bias)  for  point  in
x_test]
        return y_pred
```

$$m' = m - \alpha(slope_m)$$ $$c' = c - \alpha(slope_c)$$

we put these 2 formulas to play in above code.

```
lr = LinearRegression()
lr.fit(X_train.reshape(-1,1),Y_train)
y_pred = lr.predict(X_test)
import matplotlib.pyplot as plt
plt.scatter(X_train,Y_train)
plt.plot(X_test,y_pred)
plt.show()
```



Now we have Linear Regression working for multiple features as well.

Types of Gradient Descent

1) Batch Gradient Descent

Batch gradient descent, also called vanilla gradient descent, calculates the error for each example within the training dataset, but only after all training examples have been evaluated does the model get updated. This whole process is like a cycle and it's called a training epoch.

Some advantages of batch gradient descent are that it's computationally efficient, it produces a stable error gradient and a stable convergence. Some disadvantages are the stable error gradient can sometimes result in a state of convergence that isn't the best the model can achieve. It also requires the entire training dataset be in memory and available to the algorithm.

2) Stochastic Gradient Descent

By contrast, stochastic gradient descent (SGD) does this for each training example within the dataset, meaning it updates the parameters for each training example one by one. Depending on the problem, this can make SGD faster than batch gradient descent. One advantage is the frequent updates allow us to have a pretty detailed rate of improvement.

The frequent updates, however, are more computationally expensive than the batch gradient descent approach. Additionally, the frequency of those updates can result in noisy gradients, which may cause the error rate to jump around instead of slowly decreasing.

3) Mini-Batch Gradient Descent

Mini-batch gradient descent is the go-to method since it's a combination of the concepts of SGD and batch gradient descent. It simply splits the training dataset into small batches and performs an update for each of those batches. This creates a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent.

Common mini-batch sizes range between 50 and 256, but like any other machine learning technique, there is no clear rule because it varies for different applications. This is the go-to algorithm when training a neural network and it is the most common type of gradient descent within deep learning.

Using Linear Regression for Classification

Linear Regression is a regression algorithm but can be used in classification problems as well.

Let's consider an example where the results of linear regression are between 0 and 5 (real numbers like 3.567) and we need to classify the data into 6 categories i.e. 0, 1, 2, 3, 4, 5. A simple way of using this for classification is to just round off the result to the nearest integer between 0 and 5. So, if the result from linear regression is [1.23, 0.43, 4.32, 3.49] we get the results of our classification as [1, 0, 4, 3] .

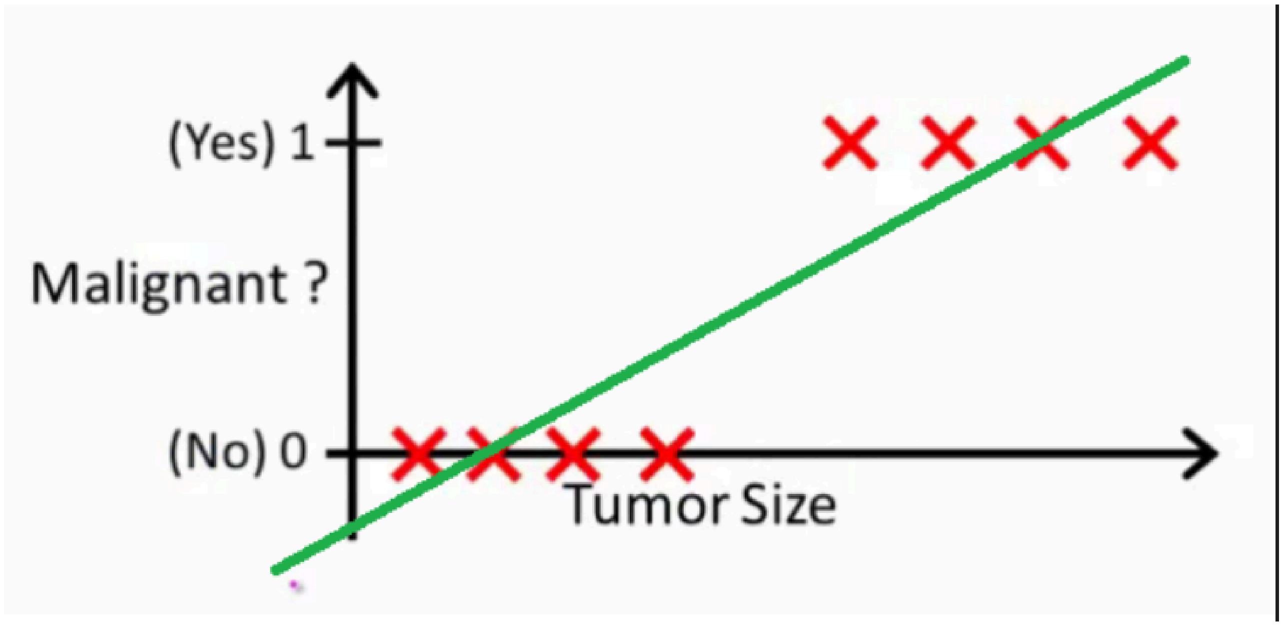
We need to set the thresholds for classification. In the above example thresholds were: [0-0.5) for 0, [0.5,1.5) for 1, [1.5,2.5) for 2 and so on. You can decide your own threshold values depending on the data you have

Why is linear regression bad for classification?

Linear Regression can be used for classification by defining appropriate threshold values, but it is not the right algorithm for classification problems because:

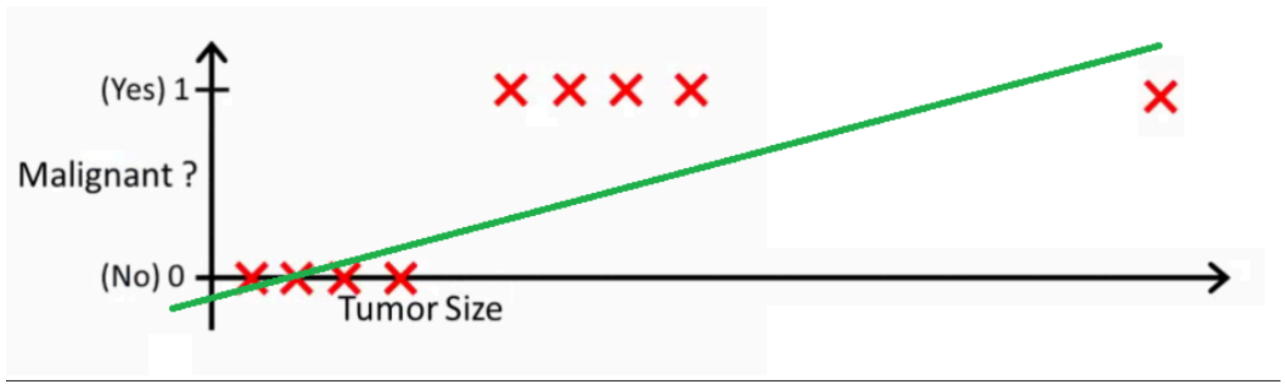
1. Outliers can affect the best fit line and thus the decision boundary. Values predicted by Linear Regression will be continuous, whereas expected results will be discrete.

2. With linear regression you fit a polynomial through the data - say, like in the example below we're fitting a straight line through {tumor size, tumor type} sample set:



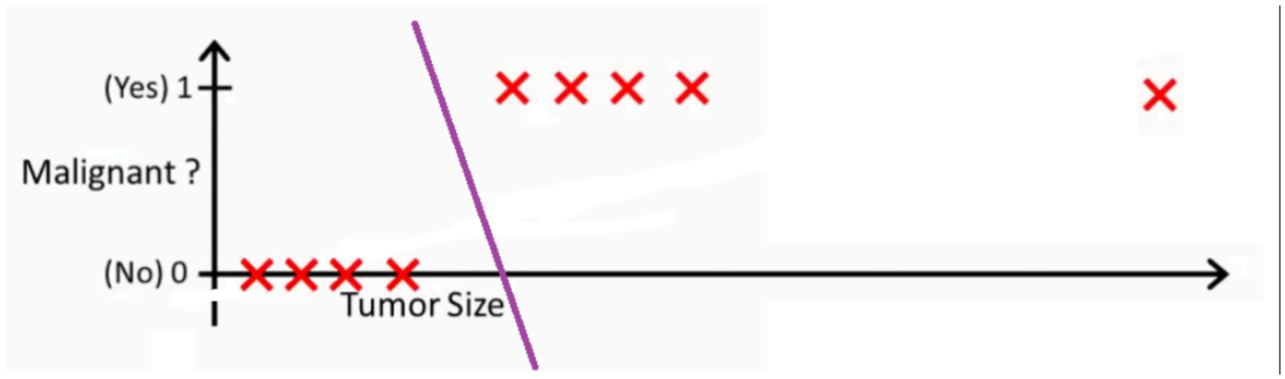
In the above case, **malignant tumors** get 1 and **non-malignant (benign)** tumors get 0, and the green line is our hypothesis h(x). To make predictions we may say that for any given tumor size x, if h(x) is greater than 0.5 we predict malignant tumor, otherwise we predict benign.

Lets change the data a little bit and for a quite large value of tumor size lets add a Malignant cancer data point. Now our line h(x) begins to look somewhat like this:



We can clearly see that now the predictions are not correct. Because we are trying to fit a line through the data we are getting, the line will be dependant on the quality and type of data we get. We cannot change the hypothesis each time a new sample arrives. Instead, we should learn it off the training set data, and then (using the hypothesis we've learned) make correct predictions for the data we haven't seen before. This can be done by creating a **decision boundary**.

Decision Boundary is the boundary which separates the two regions in classification. If we have a binary classification with values 0 and 1 then one side of this boundary will be 0 and the other will be 1. Take a look at the image below.



Here, the purple colour line is the decision boundary. All points on the left side of this line correspond to 0 and points on the right side correspond to 1. In the above problem (and also many other classification problems) we place greater importance on the fact that the points are placed on the correct side of the decision boundary and not so much on their distance from it. What this means is that it's alright if the points are quite close to the decision boundary, as long as they are on the right side of it.

The above decision boundary (purple line) is similar to the one which would have been generated if we would have used Logistic Regression in the above problem. Both linear regression and logistic regression give us a straight line (or a higher order polynomial) but those lines have different meaning:

1. The line for Linear Regression interpolates, or extrapolates, the output and predicts the value for x we haven't seen.

2. h(x) for Logistic Regression tells you the measure (e.g. probability) that x belongs to the "positive" class. You can see the line formed as the decision boundary.

Logistic Regression

Logistic Regression is a statistical method for analyzing a dataset in which there are one or more independent variables that determine an outcome. Logistic Regression is actually a classification algorithm. Logistic regression is used to describe data and to explain the relationship between one dependent binary variable and one or more nominal, ordinal, interval or ratio-level independent variables.

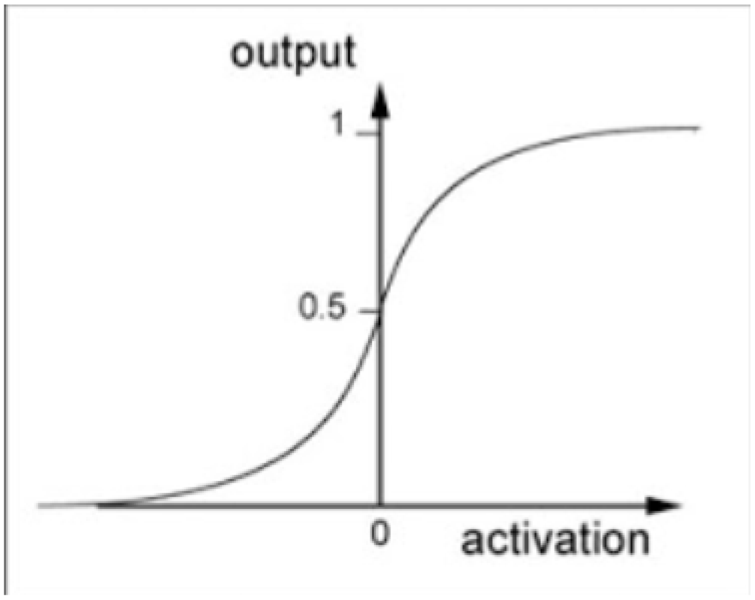
Before starting with Logistic regression we need to know about a function called **Sigmoid Function** and its properties.

Sigmoid Function

A sigmoid function is a mathematical function having an "S" shaped curve (sigmoid curve). Mathematically , the function is :

$$S(t) = \frac{1}{1 + e^{-t}}$$

Its curve looks like :



With its output ranging between 0 and 1. As we can clearly see that the curve quickly goes toward 1 when t > 0 and toward 0 when t < 0. At t = 0 it is equal to 0.5.

Value of the above function for t = 2 is 0.88 and for t = -2 is 0.119 , which shows how sharply it goes towards 0 and 1.

Because of the sigmoid function's property to give output between 0 and 1, we can use its output like a probability, but not exactly as probability. For example, the property of probability that P(true) + P(false) = 1 may not be true is case of sigmoid function i.e. S(true) + S(false) **may not be equal to 1**.

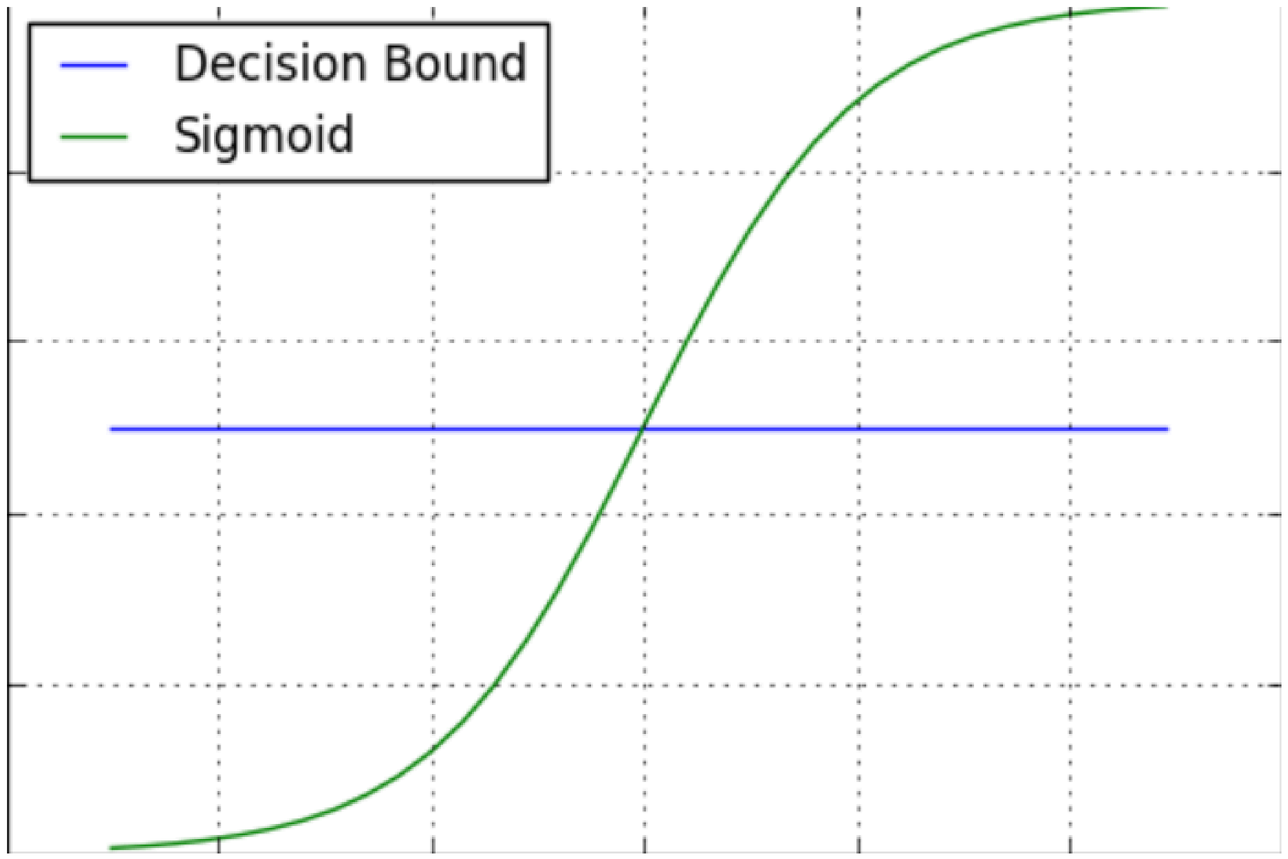
The logistic function has this further, **important property**, that its derivative can be expressed by the function itself,

As at **t = 0** we have **S(t) = 0.5** , and for **t > 0** we have **S(t) > 0.5** (sharply rising to 1 so we consider it 1) and for **t < 0** we have **S(t) < 0.5** (sharply falling to 0 so we consider it 0) , we have our decision boundary as 0.5.

For example, if our threshold is 0.5 and our prediction function returned 0.7, we would classify this observation as 'positive' (1). If our prediction was 0.2 we would classify the observation as 'negative' (0).

For logistic regression with multiple classes we could select the class with the highest predicted probability.

Observe the image below for better understanding :



Cost Function of Logistic Regression

Earlier in linear regression we have used the following cost function:

$$\sum_{k=1}^{n} (y_t - y_p)^2$$

where in Y_p was replaced by its value mx.

$$\sum_{k=1}^{n}(y-mx)^2$$

But in the case of Logistic Regression, if we use the same cost function then the cost function will end up having many minimas because of the hypothesis function of Logistic Regression. Therefore while estimating a minimum value we may end up at some **local minimum rather than a global minimum**.

$$\sum_{k=1}^{n}(y-\frac{1}{1+e^{-mx}})^2$$

So here we use **cross entropy** to measure the cost of our model.

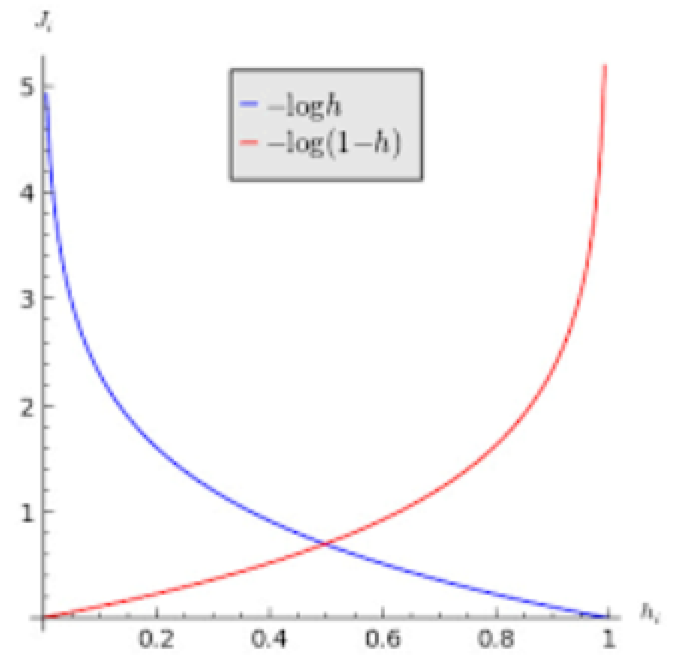
$$\begin{aligned} cost &= -log(h(x)) \;\; if \;\; y=1 \\ &\;\;\;\; -log(1-h(x)) \;\; if \;\; y=0 \end{aligned}$$

here **h(x)** is the hypothesis function for Logistic Regression (i.e sigmoid function) and **y** is the actual true output, i.e. the actual label, for the current considered values of features.

This error function penalizes for the wrong predictions we make.

For example, consider that y (the actual label) is 1 and we classify it as e (e is tending to zero), then our cost will be **-log(e)** i.e a very very high positive value, and the cost reduces as we classify it near to 1 . The cost becomes zero when we classify it correctly as 1 . Also consider that y (the actual label) is 0 and we classify it as t (nearly equal to 1), then the cost for that will be **-log(1-t) = -log(e)**, where e is tending to zero, and again the cost has a very very high positive value. And the cost tends to zero when we classify it correctly as 0.

The plot for -log(h) and -log(1-h)



We can combine the two cases of our cost function as:

$$cost = [-y * log(h(x))] - [(1-y) * log(1-h(x))]$$

Cost function can also be labelled as the error function for our model.
For **ith** point

$$E(h(x^i), y^i) = [-y^i * log(h(x^i))] - [(1-y^i) * log(1-h(x^i))]$$

So, for the entire dataset containing **x** and **y** will be

$$E(x,y) = \frac{1}{M}\sum_{i=0}^{M}(-y^i * log(h(x^i))] - [(1-y^i) * log(1-h(x^i)))$$

where M is the number of training points.

The only parameter which we can vary in our cost function is h(x) and therein the variable is m, shown below :

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

Hence our aim is to improve the accuracy of our model and reduce the cost by selecting a proper value(s) for m , and we do so by **gradient descent** as it was done in linear regression.

Our cost function is a convex function, i.e it has only one local minimum therefore we can use **Gradient Descent** approach to find the apt value of m.

Finding Optimal values of 'm'

From gradient descent, we know that

$$m_j = m_j - \alpha \frac{\partial E}{\partial m_j}$$

Solving this derivative gives us the value as

$$\frac{\partial E}{\partial m_j} = \frac{1}{m} \sum_i (y^i - h(x^i))x_j^i$$

Multi-Class Classification

Binary classification tasks are those tasks where examples are assigned exactly one of two classes. Multi-class classification tasks are those tasks where examples are assigned exactly one of more than two classes.

Binary Classification: Classification tasks with two classes.

Multi-class Classification: Classification tasks with more than two classes.

As such, logistic regression cannot be used for multi-class classification tasks, at least not directly.

Instead, heuristic methods can be used to split a multi-class classification problem into multiple binary classification datasets and train a binary classification model each.

Two examples of these heuristic methods include:

1. One-vs-Rest (OvR)
2. One-vs-One (OvO)

1) One-Vs-Rest for Multi-Class Classification

One-vs-rest (OvR for short, also referred to as One-vs-All or OvA) is a heuristic method for using binary classification algorithms for multi-class classification.

It involves splitting the multi-class dataset into multiple binary classification problems. A binary classifier is then trained on each binary classification problem and predictions are made using the model that is the most confident.

For example, given a multi-class classification problem with examples for each class **'red'**, **'blue'** and **'green'**. This could be divided into three binary classification datasets as follows:

Binary Classification Problem 1: red vs [blue, green]

Binary Classification Problem 2: blue vs [red, green]

Binary Classification Problem 3: green vs [red, blue]

A possible downside of this approach is that it requires one model to be created for each class. For example, three classes requires three models. This could be an issue for large datasets (e.g. millions of rows), slow models (e.g. neural networks), or very large numbers of classes (e.g. hundreds of classes).

This approach requires that each model predicts a class membership probability or a probability-like score. The argmax of these scores (class index with the largest score) is then used to predict a class.

2) One-vs-One

One-vs-One (OvO for short) is another heuristic method for using binary classification algorithms for multi-class classification.

Like one-vs-rest, one-vs-one splits a multi-class classification dataset into binary classification problems. Unlike one-vs-rest that splits it into one binary dataset for each class, the one-vs-one approach splits the dataset into one dataset for each class versus every other class.

For example, consider a multi-class classification problem with four classes: ‘red,’ ‘blue,’ and ‘green,’ ‘yellow.’ This could be divided into six binary classification datasets as follows:

Binary Classification Problem 1: red vs. blue

Binary Classification Problem 2: red vs. green

Binary Classification Problem 3: red vs. yellow

Binary Classification Problem 4: blue vs. green

Binary Classification Problem 5: blue vs. yellow

Binary Classification Problem 6: green vs. yellow

This is significantly more datasets, and in turn, models than the one-vs-rest strategy described in the previous section.

The formula for calculating the number of models, is as follows:

$$\frac{NumClasses * (NumClasses - 1)}{2}$$

We can see that for four classes, this gives us the value of six binary classification problems:

$$\frac{4 * (4 - 1)}{2}$$

$$\frac{4 * 3}{2}$$

$$6$$

Each binary classification model may predict one class label and the model with the most predictions or votes is predicted by the one-vs-one strategy.

Similarly, if the binary classification models predict a numerical class membership, such as a probability, then the argmax of the sum of the scores (class with the largest sum score) is predicted as the class label.

Creating useful features from given features and regularisation

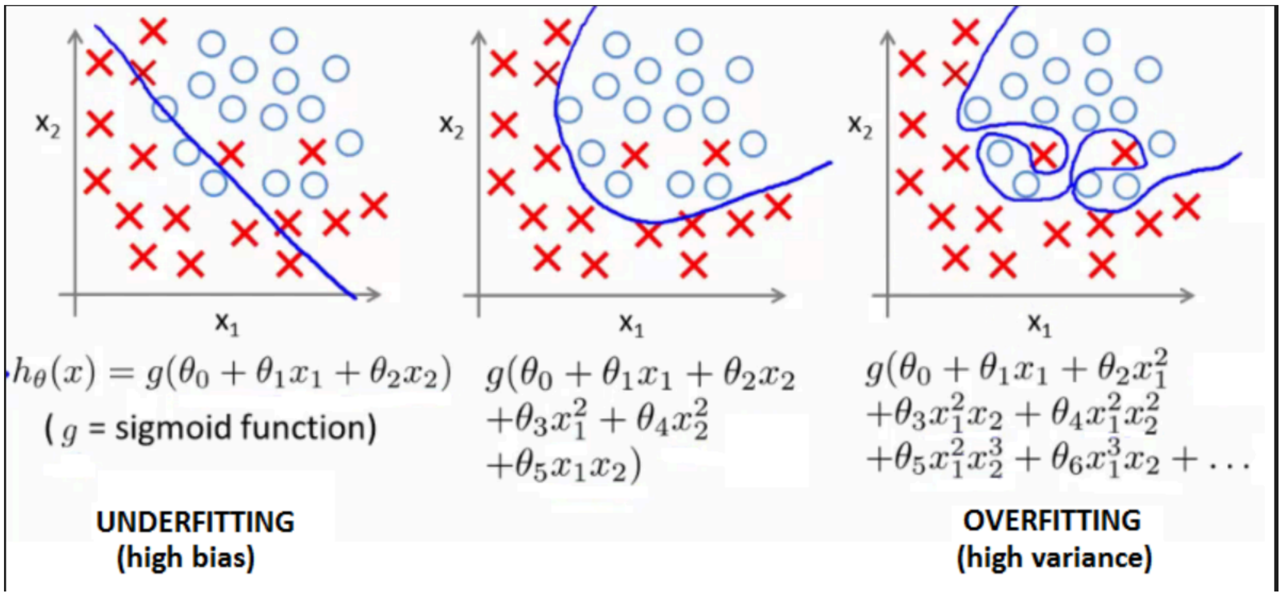
A common practice is to use the extended form by creating extra features from the given set of features. One way of doing this is shown here. This is the most general way and is used widely . Let's take an example in which we have 3 features **f1, f2, and f3**. For these features we will have an equation for input to sigmoid with at most one degree in **f1, f2, and f3**. By degree 1 we mean equations of the form **a(f1)+b(f2)+c(f3)+d**, where a, b, c, and d are some coefficients.

What we plan to do is add more features such as **f1*f1 , f2*f2 , f3*f3 , f1*f3, f1*f2 and f2*f3** so that our equation can be of 2nd degree. The main point to note here is that we have increased our number of features from just 3 to 9, but the newly added 6 features are derived from the already existing 3 features.

You can create and add features of any degree you feel like. **f1*f2*f3** will make an equation of 3rd degree. The more features you add the better your decision boundary tries to fit in the training data. Generally we get better results with higher degree terms in the equation.

We must acknowledge the fact that if the dependence of the output is very low on the a certain factor say **f1*f2** then the model will assign a value to its coefficient which ensures that it has less effect in the output.
With higher degree features being added to the dataset we are now able to achieve boundaries of many shapes such as parabolic and even some complex shapes.

This addition of features comes with a cost. If we keep on adding more and more features of higher degree, our model to try to fit itself more to the training data and may cause the problem of **overfitting**.
Consider the example below for more clarity:


x₂
x₁
$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$
(g = sigmoid function)
$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2)$
$g(\theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^2 x_2 + \theta_4 x_1^2 x_2^2 + \theta_5 x_1^2 x_2^3 + \theta_6 x_1^3 x_2 + \dots$
UNDERFITTING
(high bias)
OVERFITTING
(high variance)

In the above example we see that in the first figure when we use equation of degree one we just get a line which does not depict the data nicely.
For the second figure we use equation of degree 2 as we have terms like x1*x2. The curve we obtain here is depicting the data quite well. This should be the optimal solution even though it has a couple of wrong classifications. In the third example we use an equation with degree 5. We can clearly see that the decision boundary is trying to fit itself to the data. This is the case of over-fitting.

One way of reducing overfitting is by **regularisation**. A very simple explanation of regularisation can be that it discourages complex features in the dataset. Let's try to understand regularisation with an example.

Assume that we have:

$$(m_1 * x_1) + (m_2 * x_2) + (m_3 * x_1 * x_1) + (m_4 * x_2 * x_2) + (m_5 * x_1 * x_2)$$

as our input to the sigmoid function in case of Logistic regression.
We add this term to the cost function:

$$\beta(\sum_{k=1}^{n}(m_k)^2)$$

where **β** is the regularisation parameter and m are the coefficients given to n features. We are actually telling our model that the more importance you give to a particular feature the more it will add to the cost function. So this automatically acts as a deciding factor for the model. A particular feature will be given a higher valued coefficient only if it is significantly important in deciding the outcome. This will naturally reduce overfitting.

This type of regularisation in which we add square of the coefficient's value multiplied by **β** to the cost function is called L2 Regularisation and if we just added the coefficients value multiplied by **β** to the cost function, it is called L1 Regularisation.

Using Sklearn for Logistic Regression

```
from sklearn import datasets
from sklearn import model_selection
from sklearn.linear_model import LogisticRegression
cancer_ds = datasets.load_breast_cancer()
X = cancer_ds.data
Y = cancer_ds.target
X_train, X_test, Y_train, Y_test =
model_selection.train_test_split(X,Y,test_size=0.3)
```

now we are done with data pre processing using breast cancer dataset.

```
clf = LogisticRegression()
clf.fit(X_train, Y_train)
print("Training Score", clf.score(X_train, Y_train))
print("Testing Score", clf.score(X_test, Y_test))
```

the output will be

Training Score 0.9698492462311558

Testing Score 0.935672514619883

```
class Logistic_Regression:
    def __init__(self,lr=0.0001,n=1000):
        self.lr = lr
        self.iters = n
        self.weights = None
        self.bias = None

    def sigmoid(self,x):
        return 1/(1+np.exp(-x))

    def fit(self,x_train,y_train):
        n_points,n_features=x_train.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for i in range(self.iters):
            y_pred_linear = (np.dot(x_train,self.weights) + self.bias)
            y_pred = [self.sigmoid(point)  for point in y_pred_linear]
            m_slope = (1/n_points) * np.dot((y_pred - y_train) , x_train)
            b_slope = (1/n_points) * sum(y_pred - y_train)

            self.weights -= self.lr * m_slope
            self.bias -= self.lr * b_slope

    def predict(self,x_test):
        y_pred_linear = [(np.dot(point.T,self.weights) + self.bias) for point in x_test]
        y_pred = [0 if self.sigmoid(predicted) < 0.5 else 1  for predicted in y_pred_linear]
        return y_pred
```

in fit() function we used

$$\frac{\partial E}{\partial m_j} = \frac{1}{m}\sum_i (y^i - h(x^i))x_j^i$$

to optimise using gradient descent used

$$E(x,y) = \frac{1}{M}\sum_{i=0}^{M}(-y^i * log(h(x^i))] - [(1-y^i) * log(1-h(x^i)))$$

this cost function to get to that derivative.

```
log_reg =  Logistic_Regression()
log_reg.fit(x_train,y_train)
ypred =  log_reg.predict(x_test)
from sklearn.metrics import accuracy_score,confusion_matrix
accuracy_score(y_test,ypred)
```

0.9300699300699301
we get the above thing as output.

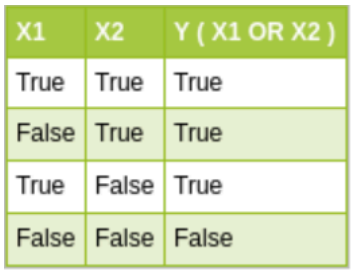
Decision Tree

A tree has many analogies in real life, and turns out that it has influenced a wide area of machine learning, covering both classification and regression. A decision tree is a tree-like graph which uses a branching method to illustrate every possible outcome of a decision.

Its structure is such that the nodes represent the place where we pick an attribute and ask a question. The edges represent the answers to the question and the leaves represent the actual output or class label. They are used in non-linear decision making with simple linear decision surface.

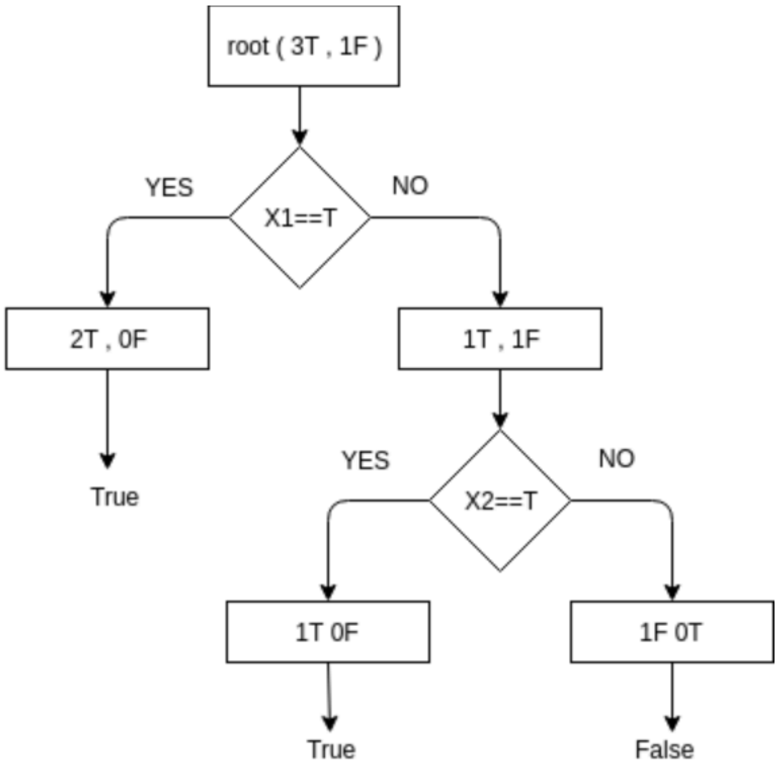
Decision trees classify the examples by sorting them down the tree from the root to some leaf node, with the leaf node providing the classification to the example. Each node in the tree acts as a test case for some attribute, and each edge descending from that node corresponds to one of the possible answers to the test case. This process is recursive in nature and is repeated for every subtree rooted at the new nodes.

Lets start by taking a example of OR of two variables X1 and X2. The decision tree for the same is shown below :



X1	X2	Y (X1 OR X2)
True	True	True
False	True	True
True	False	True
False	False	False

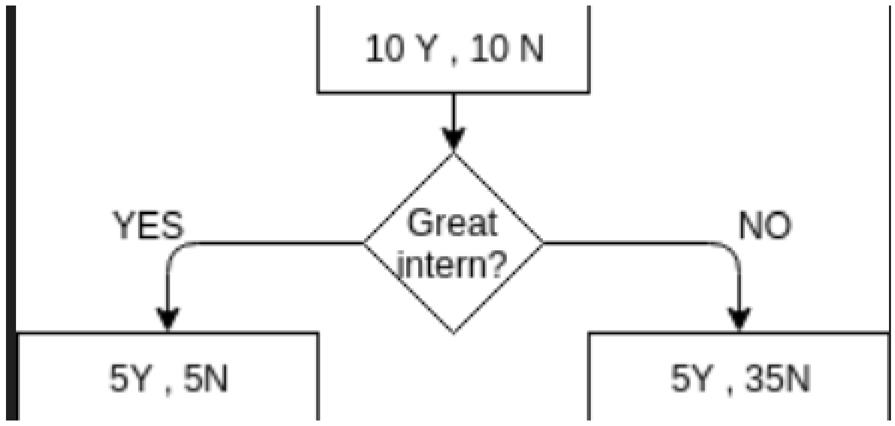
Truth Table for OR



We have a total of four possible combinations as shown in the truth table. So at the beginning we have 2 True and 2 False outcomes to start with. The first condition we check for is the value of X1. If X1 is true then we are sure that the result will be true and hence we arrive at 2 true and no false. Note that ,we only consider those rows of the table which are having X1 as true i.e. the first and the third row in the current example .That is why we have 2 total outcomes out of which 2 are true and none is false. This node is having a definite answer.If we arrive at this node there is no confusion as to which what our tree should return, therefore these are called **pure nodes.**
If X1 is false then the answer is dependent on the value of X2. If X2 is false the answer is false and if the answer is true the final result is true. We have three pure nodes here as shown in the pictorial representation of the decision tree. Now if we are given any new data for prediction we just need to run it through our tree. For example if we get A and B as two values for testing. We check first if A is true. If A is true we have true as our answer else we need to check for the value of B. If B is true then our answer agin is true else we have false as the final outcome.

Another Example

Let's consider another example which is not as straightforward as the previous one. Suppose we need to predict if a person will get an interview call or not based on some factors. There can be many factors but for simplicity let's consider that we focus on the level of projects, good intern and whether the person is from top 50 colleges or not.
Unlike the previous example in which we just picked X1 for the first decision, it is quite arguable which factor we should pick first here. Let's assume that we start by picking whether the person has done a good internship or not(which is in the form of true or false).



Similarly we continue to break down the nodes further on the basis of the features which are still left(type of college and projects in this case).
Consider a case where we have used up all the features and still we do not arrive at a pure node. Cases like this can surely be true. For eg: we may have three students lacking in all the three fields but still one gets the interview call and the other two do not. In such a case, our node which corresponds to "no" at each of the above three decisions cannot give a pure no as answer i.e. it is not a pure node. What should we do then? As we do not have anymore features left to judge on, we can simply favour the majority class of the node.
Therefore, there are two cases when we need to stop with the breaking of nodes into subparts, those are :

1. When we have a pure node, there is no need to proceed further.

2. When all the features are used and we don't have any other features present.

Depending upon which factor we pick to split on we can have different Decision trees and their accuracy will also vary. We will discuss how to pick a decent tree in the next section. What we should note is that till now we have only considered the examples where we have binary(true or false) as outcomes. If a particular feature has n different values, we may have to break a node into n subnodes. If a feature has all unique entries then we reach the pure nodes in one step but for many cases that is useless. For example, dividing a class of students on the basis of their roll number into subnodes may result in all pure nodes but it is of no use.

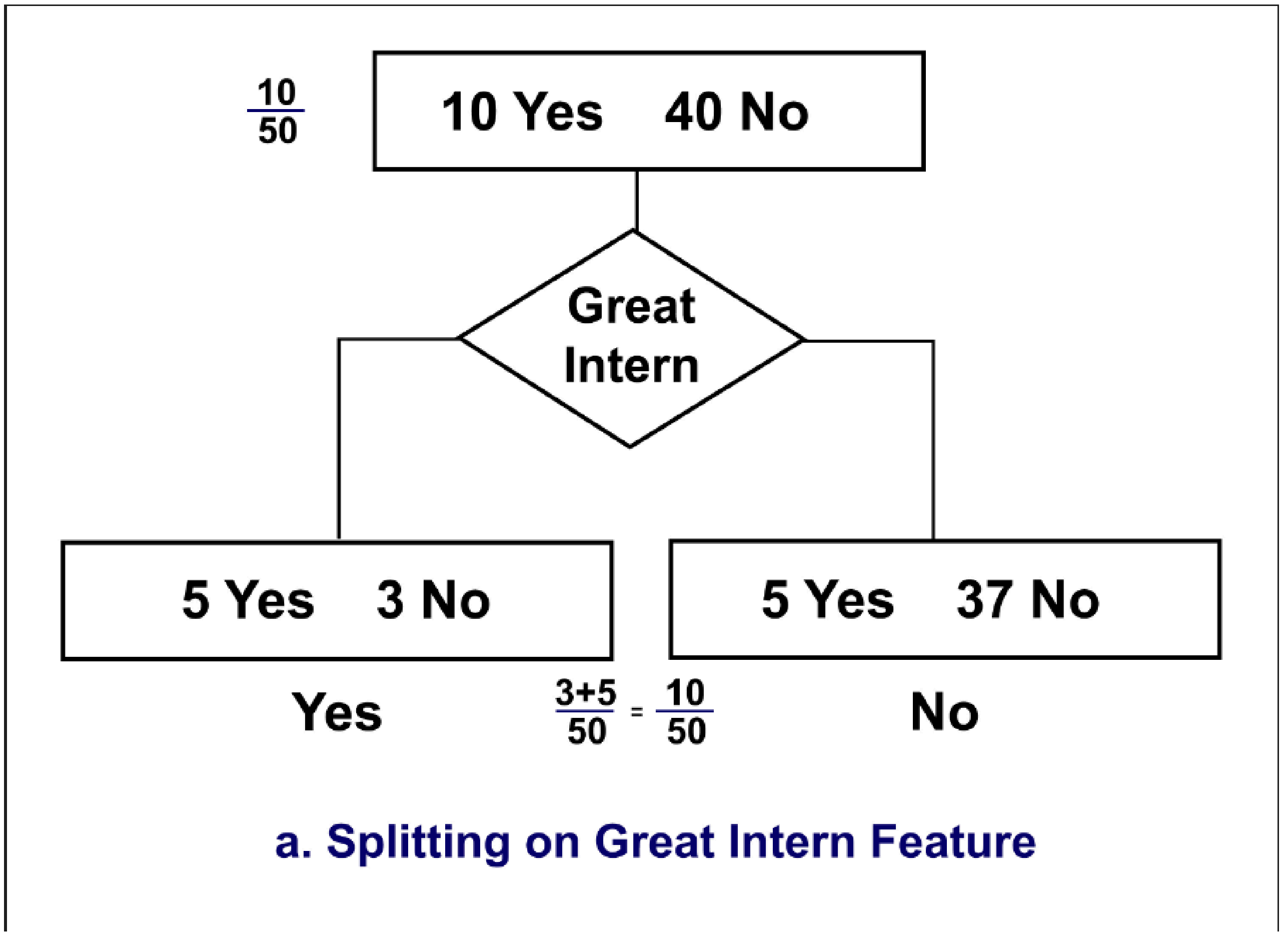
Getting to Best Decision Tree

Building a decision tree involves deciding on which feature to choose and what condition to use on splitting, alongwith knowing when to stop. In the first split on the root, all features are considered and the data points are divided into groups based on this split. Let's suppose, we have n features. Then we will be having n candidate splits at the first level. Now, we will calculate how much accuracy each split will cost us, using a function. The split(feature) which results in maximum accuracy is chosen at this level and data points are divided into child nodes according to that feature only. The child nodes formed are recursively divided into deeper levels, resulting in formation of the entire tree.

In the case we have n features, then we can possibly make an exponential number of decision trees. It is categorised into NP-HARD Problem. For finding out the best tree all possible combinations of tree possible should be taken care off. So, we are interested here to find out the good tree and not the best one. Using the **GREEDY approach**, we will try to lower the cost (and also maximize the accuracy) and according to this, build a good decision tree.

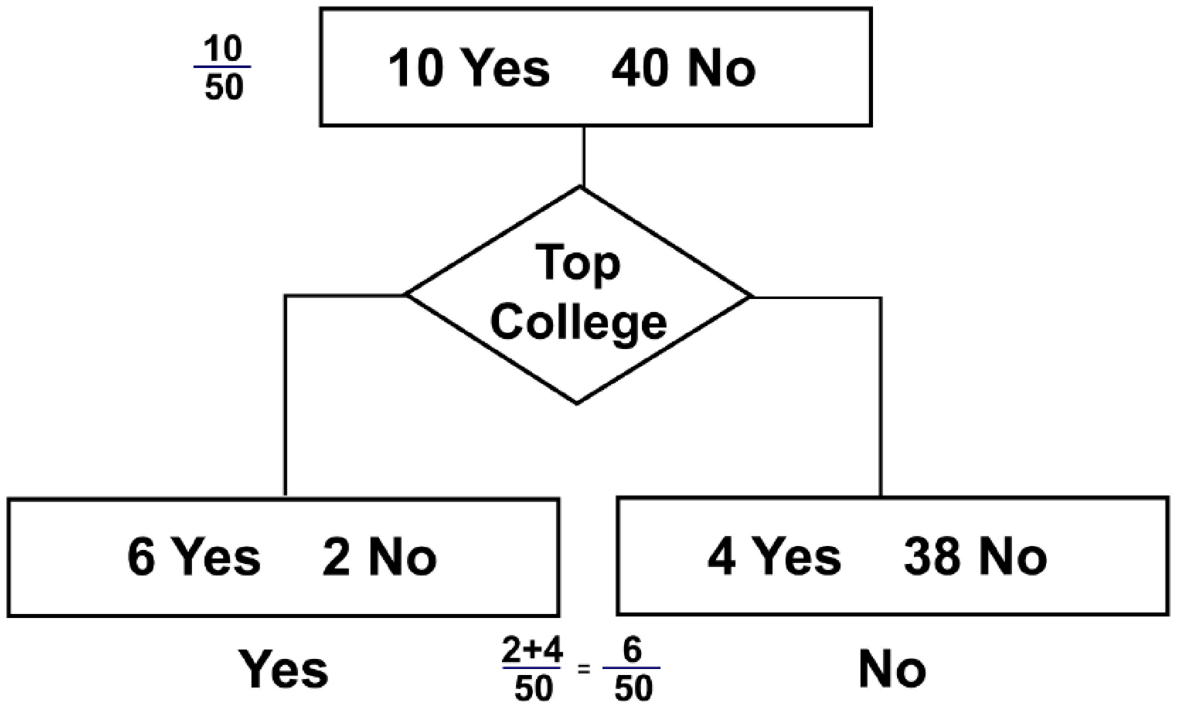
Deciding Features to Split On (On the Basis of Accuracy)

Taking the same example of the student applying for an internship in college will get a call for the interview or not. Considering total 50 students, we will pick up all the features one by one and see how many mistakes we make at each level, taking the decision on the basis of majority. Majority decision means that if we have 10 YES and 40 NO in particular node, then we will take our decision as NO for that node.



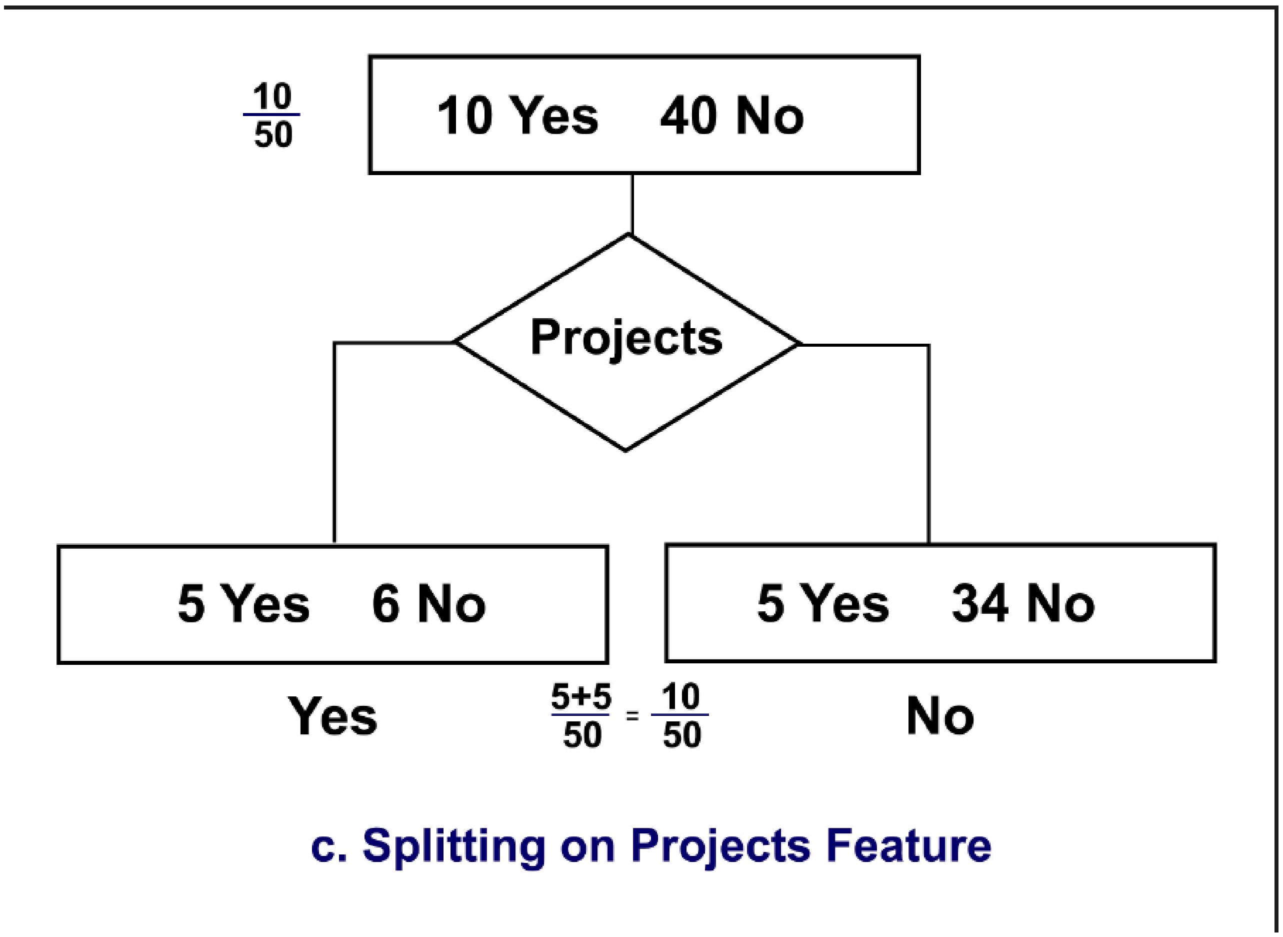
At the root node, we will take our decision as NO according to majority, then we make a total of 10 mistakes. So, it is represented as 10/50(10 out of 50) are wrong decisions.
Further, if we split the data points on the feature "Great Intern", then left node represents students who are great interns, total of 8 students, out of which 5 got a call for interview and 3 didn't. Right node similarly represents students who are not great interns, total of 42 students, and again out of

which 5 got a call for interview and 37 didn't. So, we take our decision as YES for left side child node and NO for side side child node (according to majority only). And we make a total of 8(3+5) mistakes at this level, which actually means that before using this features, if we took decision at root node only we make 10 mistakes in our decision, while after splitting on this feature we make a total of 8 mistakes at this level. Since, we are making less number of mistakes in our decision at lower level, so it will be beneficial and serve as an advantage in decision making if we make a split at this level.



b. Splitting on Top College Feature

Similarly, here we are making 10 mistakes at the root level, but after splitting we are making a total of 6(2+4) mistakes in decision at the lower level, and hence this split is also favoured at this step of splitting.



c. Splitting on Projects Feature

Here, we are making total of 10 mistakes at root level, but after splitting also we are making 10(5+5) mistakes, so we might possibly avoid split using this feature.

We will make a split using that feature by which number of mistakes made at the lower level of the tree gets reduced after a split is performed using that feature. Therefore, here we will make a split using second feature i.e. Top College

Handle Discrete and Continuous Value Features

If we have discrete value feature, say labelled data for example gender of a person we have Male and Females. Now, making a split on gender of person results in 2 child nodes, one for males and other for females.
Consider continuous valued features, say salary. Every person has different salary and values are spread over a wide range. If we have to make a split on Salary, then an option is to make different child node for every different value of salary we obtain. But unfortunately, it will result in large overfitting of data.
So, to avoid this difficulty for continuous value features. We follow the procedure mentioned to achieve the better split using this feature.

1. Spread all the salaries(values for feature chosen) on the straight line from lowest to highest order.

2. Split the data according to mid point values, taking all the pairwise adjacent points.

3. Take the salary value for that particular split that results in maximum accuracy or minimum mistakes made while making the decision.
Example, we have 4 values of salaries, 5000, 10000, 20000 and 50000. We take the mid point of all these which comes out to be 7500, 15000 and 35000. We will make a split on all these salary values one by one, first on 7500 salary, which means people with salary less than 7500 come on left side and all others on the right side. Similarly, doing for salary values 15000 and 35000. We choose that particular salary value to make a split at this level which results in maximum accuracy.

This process is followed for making a binary split for the continuous value features. We can choose salary feature again at the deeper level and make a split again using this feature below, with decreased range.

Advantages of Decision Trees

1. Decision trees are simple to understand, interpret, visualize.

2. Decision trees can handle both numerical and categorical data. They can also handle multi-output problems.

3. Decision trees require relatively little effort from users for data preparation.

4. Nonlinear relationships between parameters do not affect tree performance.

5. Decision trees provide a clear indication of which fields are most important for prediction or classification.

Disadvantages of Decision Trees

1. Decision tree learners can create over-complex trees that do not generalize the data well. This is called overfitting.

2. Decision trees are less appropriate for estimation tasks where the goal is to predict the value of a continuous attribute.

3. Decision trees are prone to errors in classification problems with many classes and relatively small number of training examples.

4. Decision trees can be computationally expensive to train. The process of growing a decision tree is computationally expensive. At each node, each candidate splitting field must be sorted before its best split can be found. In some algorithms, combinations of fields are used and a search must be made for optimal combining weights. Pruning algorithms can also be expensive since many candidate sub-trees must be formed and compared.

5. Greedy algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees, where the features and samples are randomly sampled with replacement.

6. Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the data set prior to fitting with the decision tree.

Implementation Using Sklearn

We will use the iris dataset for this.

```
from sklearn import datasets
from sklearn.metrics import confusion_matrix
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_graphviz
from sklearn.model_selection import train_test_split
import pydotplus

iris =datasets.load_iris()
model = DecisionTreeClassifier()

x_train,x_test,y_train,y_test=train_test_split(iris.data,iris.target,random_state=
1)

model.fit(x_train,y_train)

y_train_predict=model.predict(x_train)
y_predict=model.predict(x_test)

confusion_matrix(y_train,y_train_predict)
confusion_matrix(y_test,y_predict)
```

```
array([[13,  0,  0],
       [ 0, 15,  1],
       [ 0,  0,  9]], dtype=int64)
```

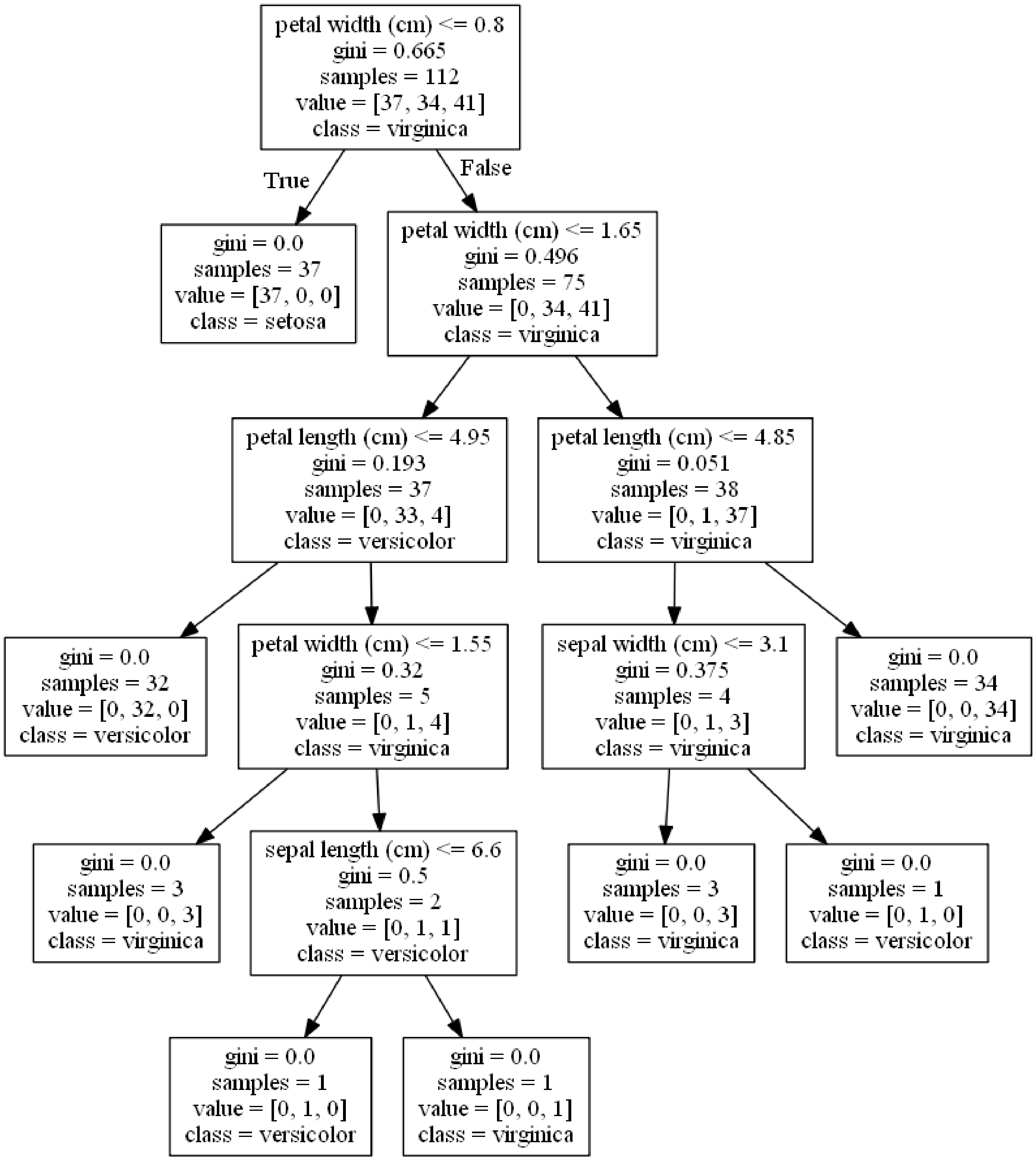
we can see only 1 error or wrong prediction in the testing data.

```

dot_data=export_graphviz(model,out_file=None,
                feature_names=iris.feature_names,
                class_names=iris.target_names)

graph = pydotplus.graph_from_dot_data(dot_data)
```

```
# graph.write_pdf("iris.pdf")
Image(graph.create_png())
```

after running this code

sklearn gives us with nice visualisation of this decision tree.



Entropy

Entropy controls how a Decision Tree decides to split the data. It actually affects how a Decision Tree draws its boundaries.

A decision tree is built top-down from a root node and involves partitioning the data into subsets that contain instances with similar values (homogenous). We use entropy to calculate the homogeneity of a sample. If the sample is completely homogeneous the entropy is zero and if the sample is equally divided, it has an entropy of one.

$$Entropy(randomness) = -\sum_i (p_i * log(p_i))$$

Here, p_i = probability of i^{th} class = $\frac{no\ of\ elements\ of\ i^{th} class}{total\ no\ of\ elements}$

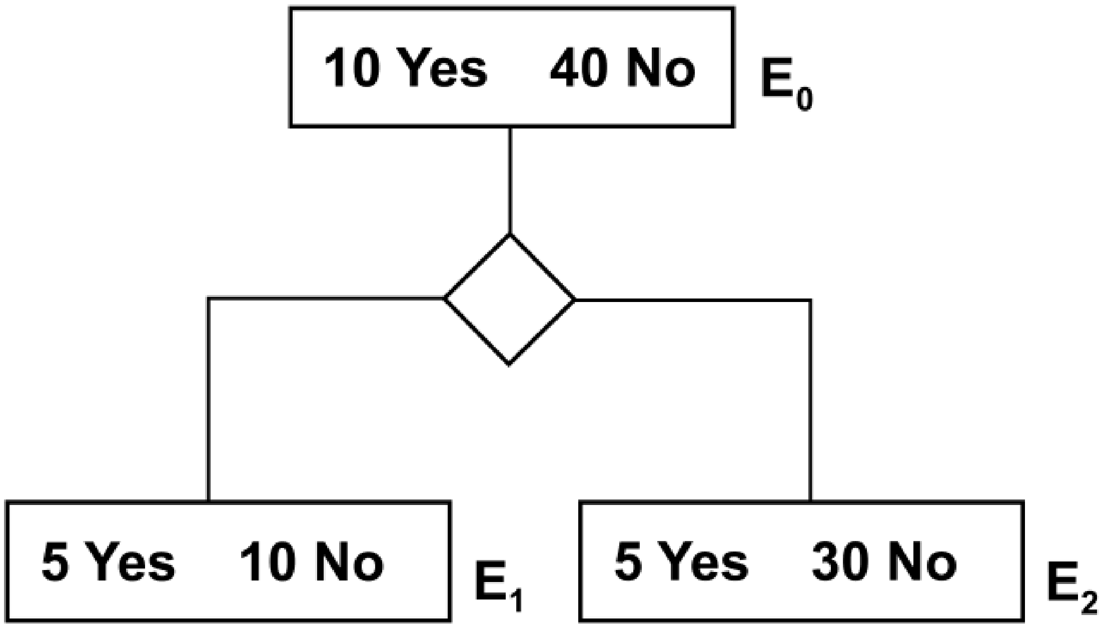
for example - entropy for 10Y, 40N

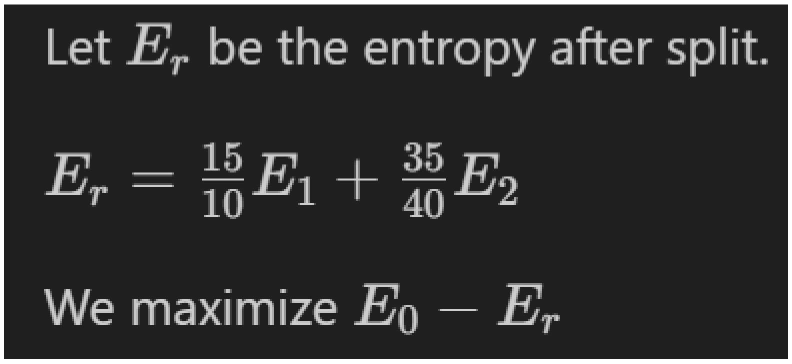
$= -\frac{10}{50}log(\frac{10}{50}) - \frac{40}{50}log(\frac{40}{50})$

Information Gain

The information gain is based on the decrease in entropy after a dataset is split on an attribute. Constructing a decision tree is all about finding attribute that returns the highest information gain (i.e., the most homogeneous branches).

Maximizing **[Entropy$_{(old)}$ - Entropy$_{(new)}$]** is information gain as we are going towards more pure or homogeneous nodes. We take weighted average of the entropies of two nodes to get the resultant entropy at the level




Let E_r be the entropy after split.

$$E_r = \frac{15}{10}E_1 + \frac{35}{40}E_2$$

We maximize $E_0 - E_r$

Now consider a situation where we predict whether a person will get a loan. If we split our data on the basis of name, then we will be getting lowest entropy and pure nodes, but that would majorly result in one node having one data point and overfitting will take place. Therefore Information Gain cannot be the only criteria on the basis of which we make our decision to split, rather we should also take in consideration the number of resulting nodes.

Split Number

Split number determines the degree of split i.e more the number of split more will be the split number.

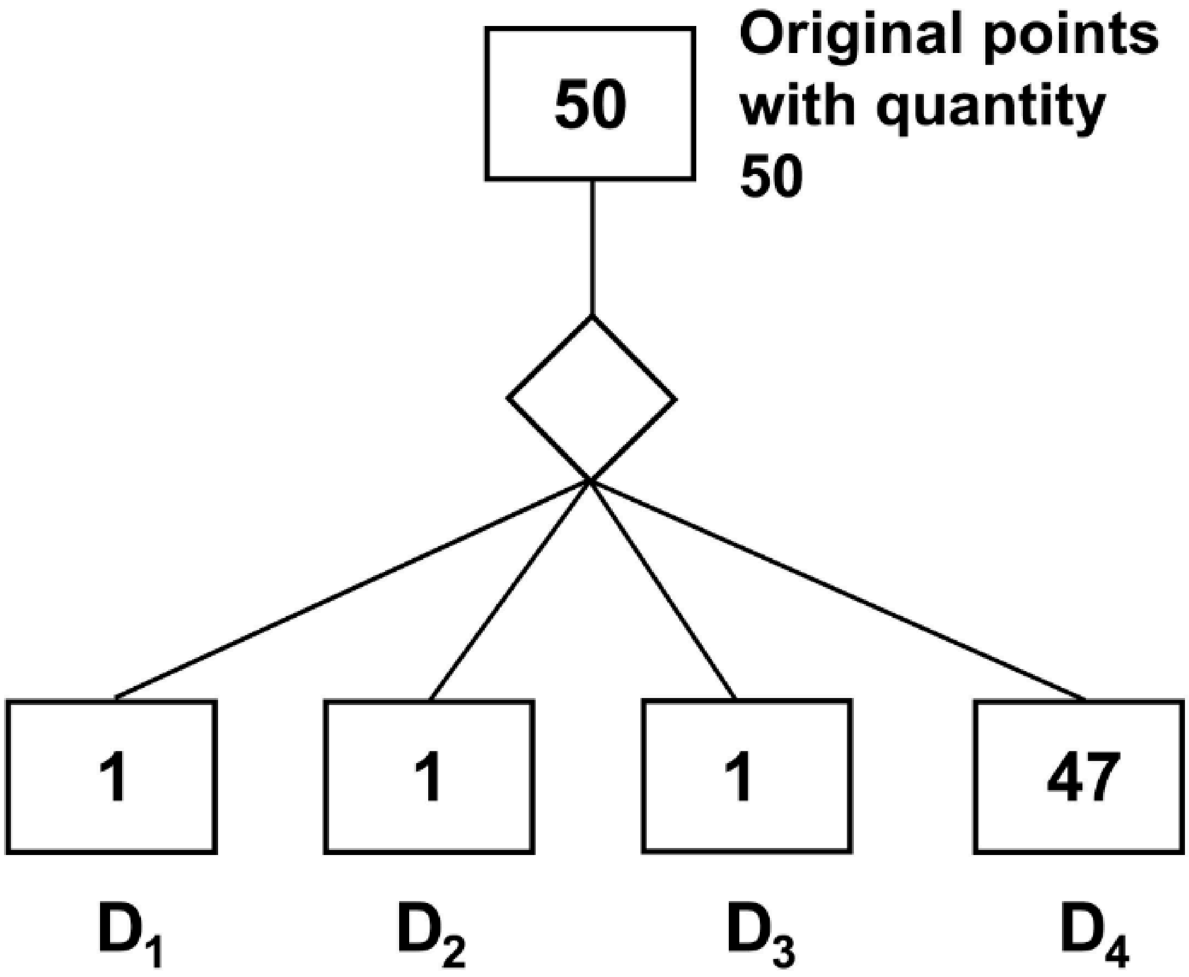
$$Split\ Number = -\sum_{j=1}^{n}(\frac{|D_j|}{|D|} * log(\frac{|D_j|}{|D|}))$$

Here, n = no of nodes into which we split

D_j = no of elements in original node on which split happened

D = no of elements in jth node

Consider an example:



D_1, D_2, D_3, D_4 = no of data points in each node after split

$$Split\ Number = -[\frac{1}{50}log(\frac{1}{50}) + \frac{1}{50}log(\frac{1}{50}) + \frac{1}{50}log(\frac{1}{50}) + \frac{47}{50}log(\frac{47}{50})]$$

Gain Ratio

We define our measure on the basis of which we will split on an attribute(or feature) considering both Information gain and Split number which is called the **gain ratio.**

$$Gain\ Ratio = \frac{Information\ Gain}{Split\ Number}$$

Gini Index

Gini Index calculates the amount of probability of a specific feature that is classified incorrectly when selected randomly. If all the elements are linked with a single class then it can be called pure.

It is calculated by subtracting the sum of squared probabilities of each class from one.

The Gini index varies between values 0 and 1, where 0 expresses the purity of classification, i.e. all the elements belong to a specified class or only one class exists there. And 1 indicates the random distribution of elements across various classes. The value of 0.5 of the Gini Index shows an equal distribution of elements over some classes.

$$Gini\ Index = 1 - \sum_{i=1}^{c}(p_i)^2$$

Here, c = number of classes

for Example:-

gn, $\boxed{50, 50, 50}$

P_i (probability of each class) $= 1/3$

$P_i^2 = 1/9$

$\therefore \sum P_i^2 = 3/9 = 1/3$

Gini Index $= 1 - 1/3 = 2/3 = 0.666$

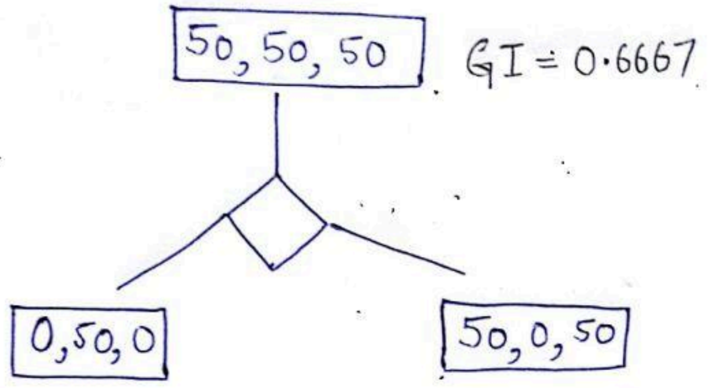
$\boxed{0, 1, 0} \rightarrow$ pure node.

$P_i = 0, 1, 0$

$P_i^2 = 0, 1, 0.$

$GI = 1 - (0+1+0) = \underline{\underline{0}}$

Gini Index of a pure node is zero



$GI \doteq 0.6667$

$GI = 0$
(pure class)

$GI = 0.5$

$p_i = 1/2, 0, 1/2$

$p_i^2 = 1/4, 0, 1/4$

$\sum p_i^2 = 1/2$

$GI = 1 - \sum p_i^2 = 1 - 1/2 = 0.5$

$$\text{Resulting GI} = \underbrace{\frac{50}{150}(0) + \frac{100}{150}(0.5)}_{\text{weighted average of both nodes}}$$

$$\text{Resultant Gini Index} = \sum \frac{D_i}{D}\left(\text{Gini Index}_i\ (GI_i)\right)$$

D_i = no. of data points in node i

D = Total no. of data points

In the above shown split we have moved from GI 0.6666 to GI of 0.3333 . A low value of Gini Index is desirable.

Inbuilt decision trees also use Gini Index.

Gini Index vs Information Gain

1. The Gini Index facilitates bigger distributions easy to implement whereas the Information Gain favors lesser distributions having small count with multiple specific values.

2. The method of the Gini Index is used by CART algorithms, in contrast to it, Information Gain is used in ID3, C4.5 algorithms.

3. Gini index operates on the categorical target variables in terms of “success” or “failure” and performs only binary split, in opposite to that Information Gain computes the difference between entropy before and after the split and indicates the impurity in classes of elements.

Overfitting in Decision Trees

In decision trees, overfitting occurs when the tree is designed so as to perfectly fit all samples in the training data set. Thus it ends up with branches with strict rules of sparse data. Thus, this affects the accuracy when predicting samples that are not part of the training set.

In simpler words, overfitting happens when our algorithm continues to develop hypotheses that reduce training set error at the cost of an
increased test set error.

We may end up overfitting the tree so much that it leads to 0 training error. For a simple dataset like Iris Dataset, such a tree is simple. But as the data sets become more detailed, the complexity of such a build increases too much.

There are a couple of ways to stop this from happening.

Stop Early (Pre-Pruning)

This method dictates that we add more stopping criteria. Right now, the only two stopping criteria we have is when we reach a pure node, or when the number of features exceeds a certain limit. By doing this, we stop early and avoid building of the entire tree before it perfectly classifies the training set.

This can be done in two simple ways:

Define a max depth (k)

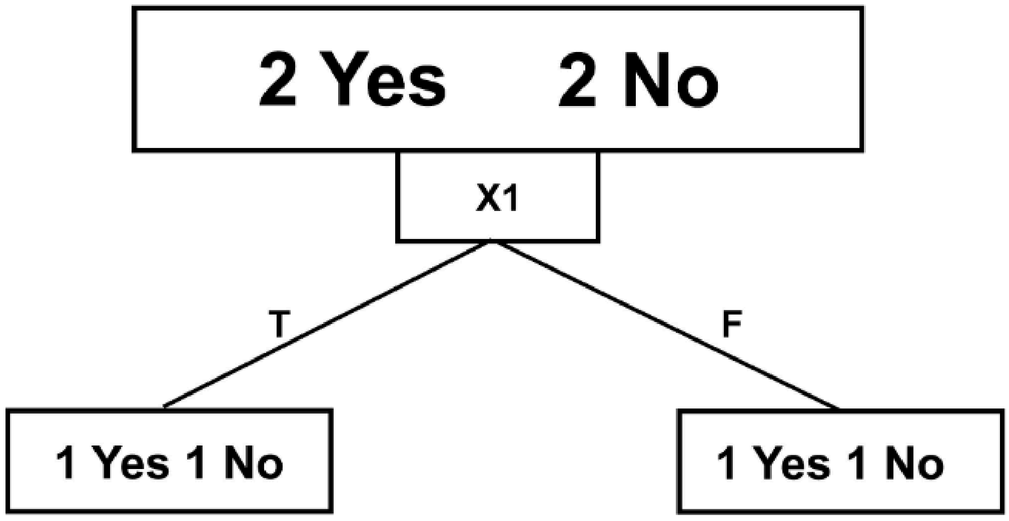
We shall decide a depth k, till which our tree will build. Beyond that, we will stop the training.

The problem with this method is that, we may not always choose the optimal k value.

Another problem with this method is that, in some cases, we require unbalanced trees for our classification. But selecting k value would restrict us towards building only balanced decision trees, thus wasting computation time, as well as leading to unwanted errors in classification.

Stopping when there is no significant gain in Score

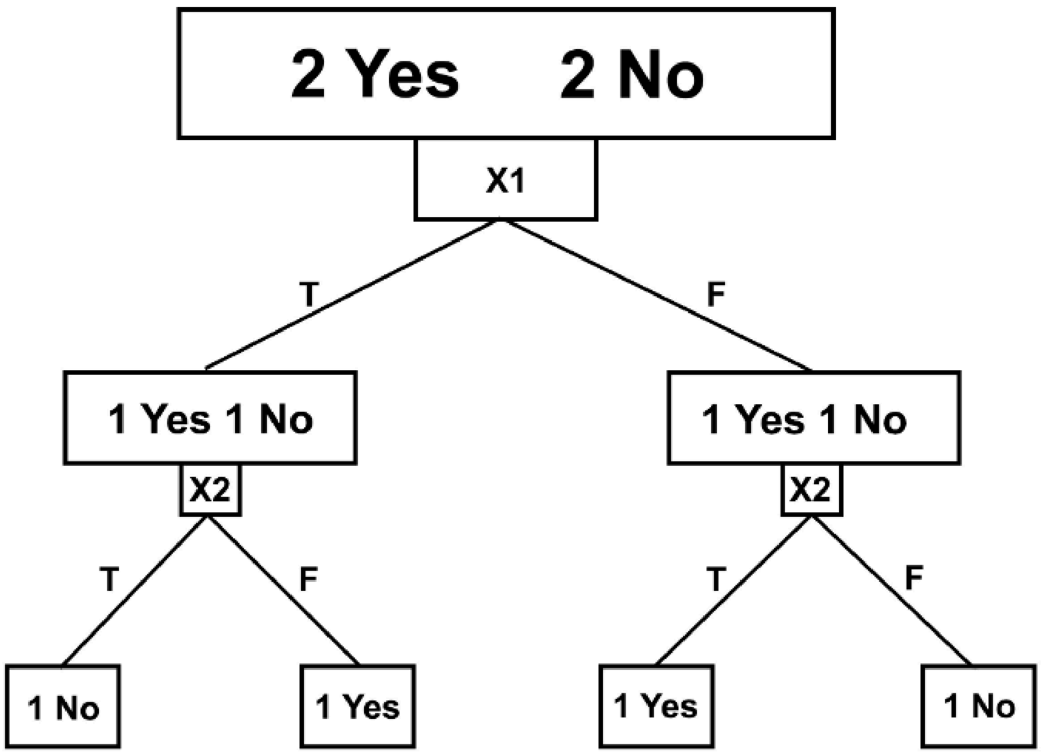
Though this method sounds good theoretically, it has some practical limitations. Let us look at them through the example of XOR.



At the initial level, we have **one node** with two YES and two NO. This means the accuracy is of 50%.

At the second level, we have **two nodes**, each with one YES and one NO. Again, the accuracy is of 50%.

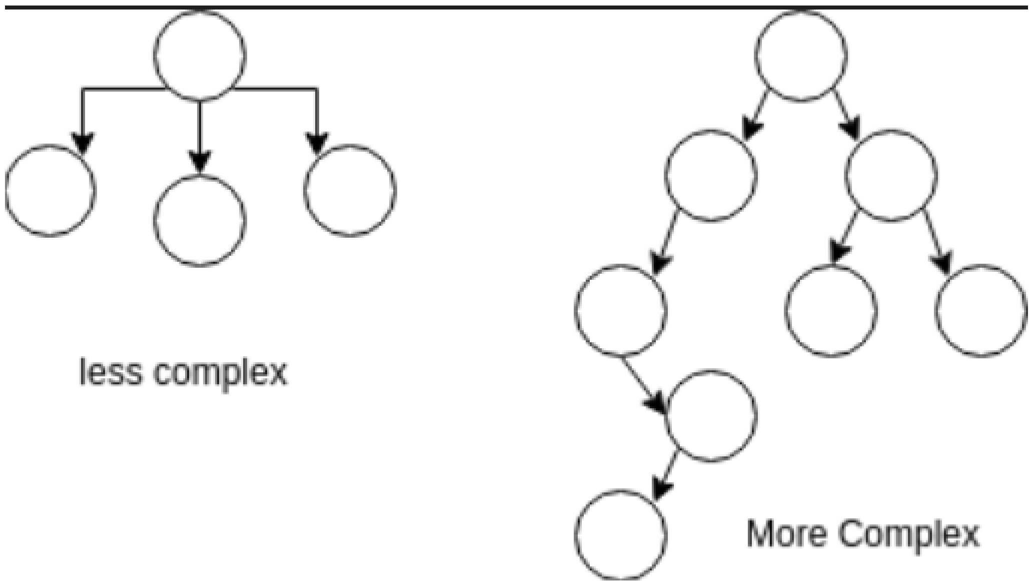
Since there is no significant change in the accuracy, our decision tree will stop itself from further training. But this is a huge loss for us, as, if another split is performed on second level, we shall achieve 100% accuracy, as it builds a perfect decision tree for XOR.



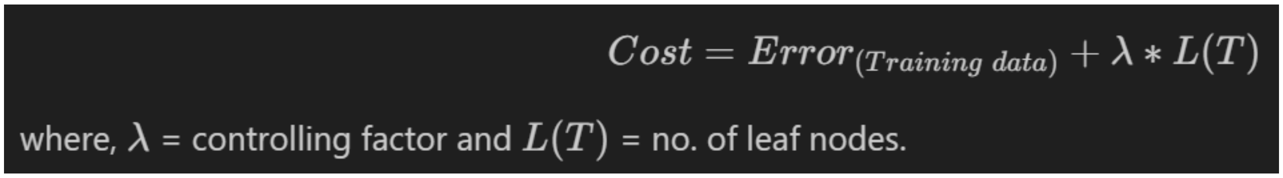
Due to these issues, early stopping using these methods is generally avoided. To avoid overfitting, the most significant method is **Pruning**.

Pruning

Pruning is a technique that allows the tree to perfectly classify the training set, and then trim the tree, such that it reduces the size of decision trees by removing sections of the tree that provide little power to classify instances. Pruning reduces the complexity of the final classifier, and hence improves predictive accuracy by the reduction of overfitting.
Consider the two Decision Tree shown below:



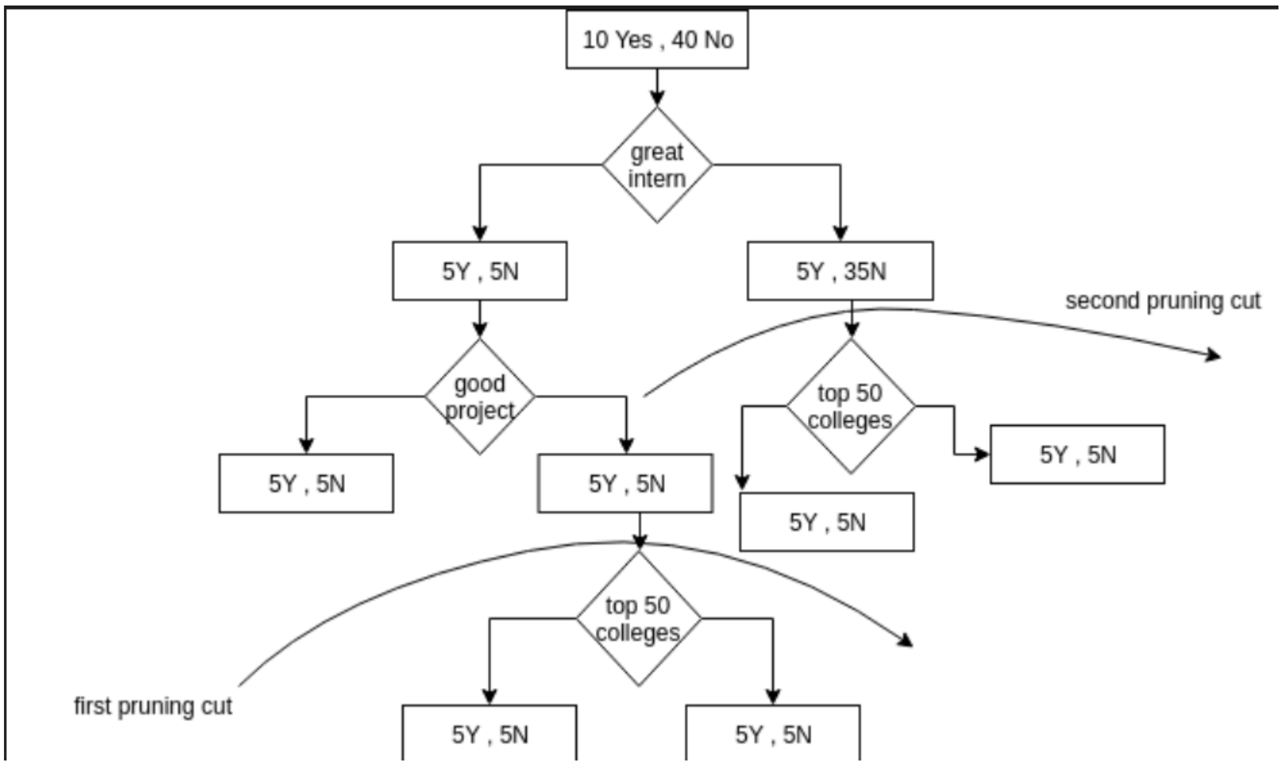
As we can see , the first tree is less complex than the second tree. Complexity can increase due to Depth as well as the nodes at a level. We cannot decide just on the basis of Depth or just on the basis of number of nodes at a level.
We add to our original cost a parameter lambda(λ) so that



$$Cost = Error_{(Training\ data)} + \lambda * L(T)$$

where, λ = controlling factor and $L(T)$ = no. of leaf nodes.

We start from the bottom of the tree and check for each split. As we encounter a split we decide if we want to keep that split or if we should keep the parent node of that split instead based upon the entropy and cost function of both the situations. If we decide to keep the split, we also have to keep all the splits above that particular split.
Consider the example shown below of a Decision Tree of getting an interview call based upon a good project, great intern and the type of college.



Own decision Tree Implementation.

```
import  numpy as np
from collections import Counter
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

brest_cancer = load_breast_cancer()
x_train,x_test,y_train,y_test =
train_test_split(brest_cancer.data,brest_cancer.target)
```

Load the dataset

```
class Node:
    def
__init__(self,feature=None,split_point_of_feature=None,left=None,right=None,value=
None):
        self.feature=feature
        self.threshold=split_point_of_feature
        self.left=left
        self.right=right
        self.value=value
    def is_leaf_node(self):
        return self.value is not None
```

we define Node class

Let us define and understand few methods

```
    def most_common_class(self,y):
        return Counter(y).most_common(1)[0][0]
```

given Y this will return the most common Y_i.

```
    def calculate_entropy(self,Y):

        probabality_of_each_class_in_y = np.bincount(Y) / len(Y)

       return -np.sum([p*np.log(p) for p in probabality_of_each_class_in_y if p>0])
```

this function calculates the entropy if array y is given

```
    def split(self,feature_to_split,split_feature_point):

        left_idx = np.argwhere(feature_to_split <= split_feature_point).flatten()

        right_idx = np.argwhere(feature_to_split > split_feature_point).flatten()

        return left_idx,right_idx
```

separates data points into right or left depending on feature and threshold.

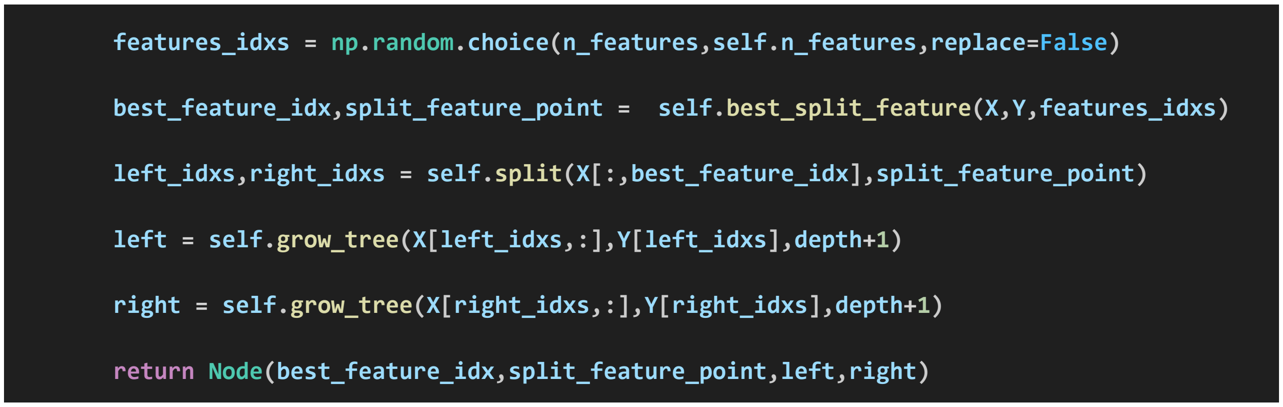
```
    def calculate_info_gain(self,Y,feature_to_cal,split_feature_point):

        parent_entropy = self.calculate_entropy(Y)

        left_children_idxs,right_children_idxs =

self.split(feature_to_cal,split_feature_point)

        if len(left_children_idxs) == 0 or len(right_children_idxs) == 0:

            return 0

        n = len(Y)

        n_l,n_r=len(left_children_idxs),len(right_children_idxs)

        entropy_left, entropy_right =

self.calculate_entropy(Y[left_children_idxs]),self.calculate_entropy(Y[right_children_idxs])

        child_entropy = (n_l/n)*entropy_left + (n_r/n)*entropy_right

        return parent_entropy - child_entropy
```

we use calculate_entropy() and split() defined before and calculate the information gain for a particular tree for both childrens and parents.

```
def best_split_feature(self,X,Y,features_idxs):
    best_gain = -10000
    split_feature_index, split_feature_point = None,None
    for feature in features_idxs:
        X_column = X[:,feature]
        unique_points = np.unique(X_column)
        for unique_point in unique_points:
            info_gain = self.calculate_info_gain(Y,X_column,unique_point)
            if info_gain > best_gain:
                best_gain=info_gain
                split_feature_index = feature
                split_feature_point = unique_point
    return split_feature_index, split_feature_point
```

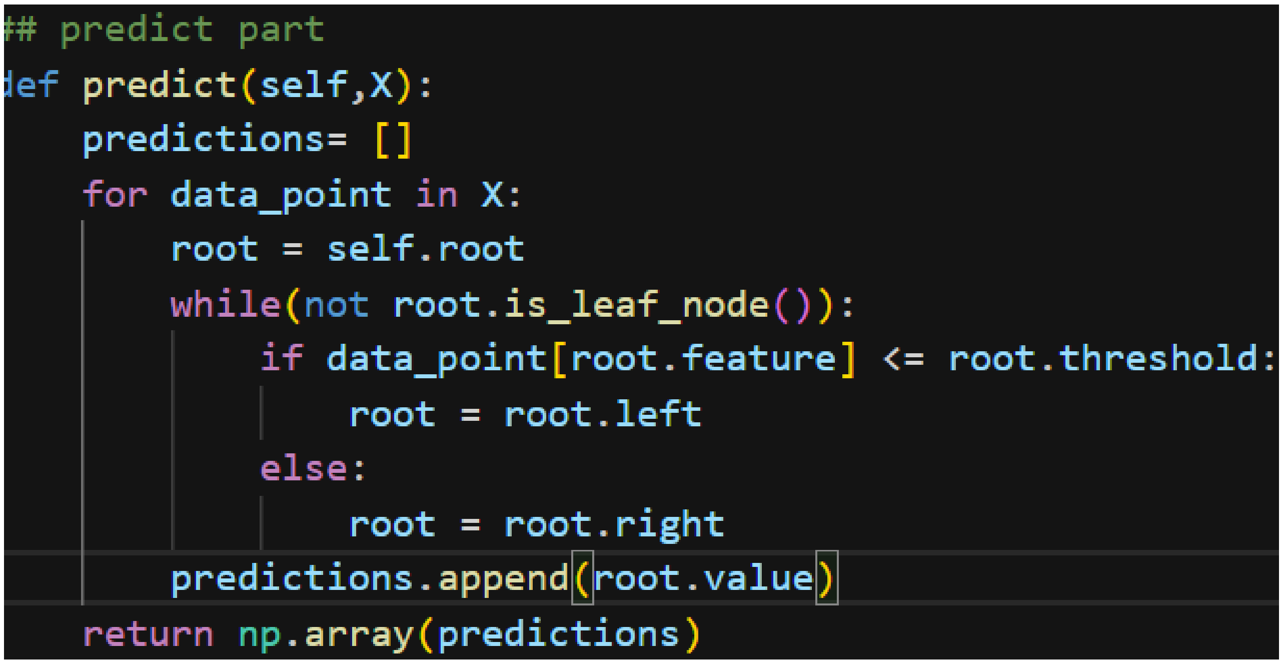
considers all the features and all the unique thresholds to split upon and gets best info gain.

```
def grow_tree(self,X,Y,depth=0):
    n_samples,n_features = X.shape
    n_classes = len(np.unique(Y))

    if(depth >= self.max_depth or n_classes == 1 or n_samples < self.min_samples_split):
        leaf_value = self.most_common_class(Y)
        return Node(value = leaf_value)
```



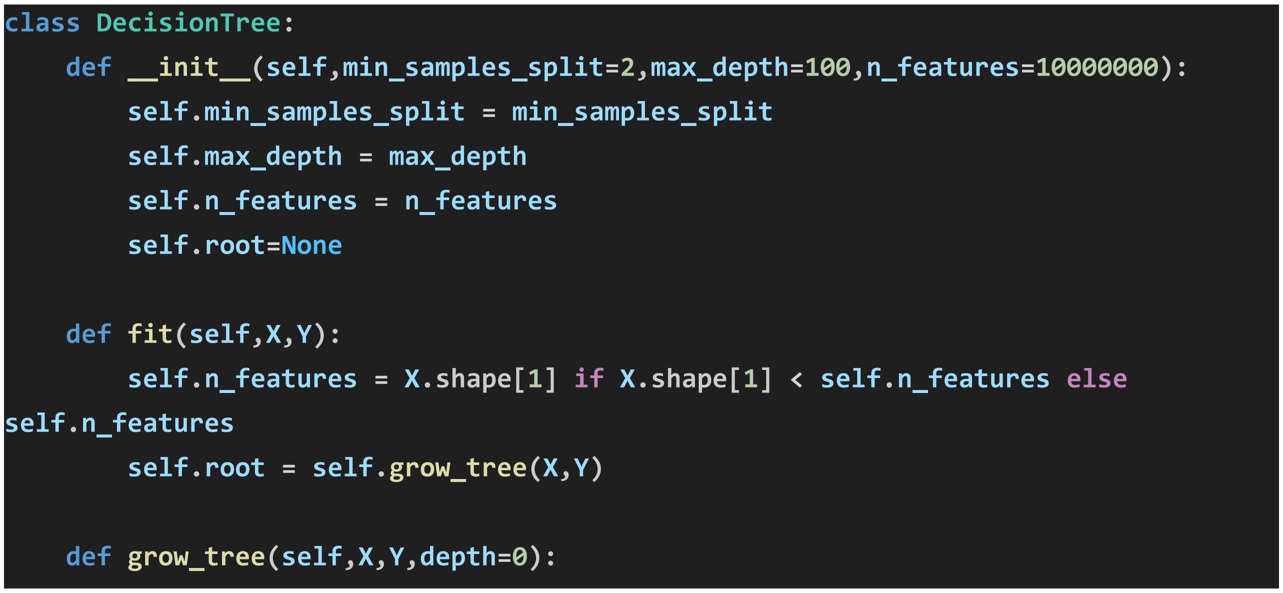
```
        features_idxs = np.random.choice(n_features,self.n_features,replace=False)

        best_feature_idx,split_feature_point =  self.best_split_feature(X,Y,features_idxs)

        left_idxs,right_idxs = self.split(X[:,best_feature_idx],split_feature_point)

        left = self.grow_tree(X[left_idxs,:],Y[left_idxs],depth+1)

        right = self.grow_tree(X[right_idxs,:],Y[right_idxs],depth+1)

        return Node(best_feature_idx,split_feature_point,left,right)
```

splits upon a feature using all the functions defined above recursively.



```
## predict part
def predict(self,X):
    predictions= []
    for data_point in X:
        root = self.root
        while(not root.is_leaf_node()):
            if data_point[root.feature] <= root.threshold:
                root = root.left
            else:
                root = root.right
        predictions.append(root.value)
    return np.array(predictions)
```

Takes X and predicts Y by traversing the tree formed.



```
class DecisionTree:
    def __init__(self,min_samples_split=2,max_depth=100,n_features=10000000):
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth
        self.n_features = n_features
        self.root=None

    def fit(self,X,Y):
        self.n_features = X.shape[1] if X.shape[1] < self.n_features else
self.n_features
        self.root = self.grow_tree(X,Y)

    def grow_tree(self,X,Y,depth=0):
        n_samples,n_features = X.shape
        n_classes = len(np.unique(Y))

        if(depth >= self.max_depth or n_classes == 1 or n_samples <
self.min_samples_split):
            leaf_value = self.most_common_class(Y)
            return Node(value = leaf_value)

        features_idxs = np.random.choice(n_features,self.n_features,replace=False)
        best_feature_idx,split_feature_point =
self.best_split_feature(X,Y,features_idxs)
        left_idxs,right_idxs =
self.split(X[:,best_feature_idx],split_feature_point)
        left = self.grow_tree(X[left_idxs,:],Y[left_idxs],depth+1)
        right = self.grow_tree(X[right_idxs,:],Y[right_idxs],depth+1)
        return Node(best_feature_idx,split_feature_point,left,right)

    def most_common_class(self,y):
        return Counter(y).most_common(1)[0][0]

    def best_split_feature(self,X,Y,features_idxs):
        best_gain = -10000
        split_feature_index, split_feature_point = None,None
        for feature in features_idxs:
            X_column = X[:,feature]
            unique_points = np.unique(X_column)
            for unique_point in unique_points:
                info_gain = self.calculate_info_gain(Y,X_column,unique_point)
                if info_gain > best_gain:
                    best_gain=info_gain
                    split_feature_index = feature
                    split_feature_point = unique_point
        return split_feature_index, split_feature_point

    def calculate_info_gain(self,Y,feature_to_cal,split_feature_point):
        parent_entropy = self.calculate_entropy(Y)
        left_children_idxs,right_children_idxs =
self.split(feature_to_cal,split_feature_point)

        if len(left_children_idxs) == 0 or len(right_children_idxs) == 0:
            return 0
        n = len(Y)
        n_l,n_r=len(left_children_idxs),len(right_children_idxs)
        entropy_left, entropy_right = 
self.calculate_entropy(Y[left_children_idxs]),self.calculate_entropy(Y[right_child
ren_idxs])
        child_entropy = (n_l/n)*entropy_left + (n_r/n)*entropy_right
        return parent_entropy - child_entropy

    def split(self,feature_to_split,split_feature_point):
        left_idx = np.argwhere(feature_to_split <= split_feature_point).flatten()
        right_idx = np.argwhere(feature_to_split > split_feature_point).flatten()
        return left_idx,right_idx

    def calculate_entropy(self,Y):
        probabality_of_each_class_in_y = np.bincount(Y) / len(Y)
        return -np.sum([p*np.log(p) for p in probabality_of_each_class_in_y if
p>0])

    ## predict part
    def predict(self,X):
        predictions= []
        for data_point in X:
            root = self.root
            while(not root.is_leaf_node()):
                if data_point[root.feature] <= root.threshold:
                    root = root.left
                else:
                    root = root.right
            predictions.append(root.value)
        return np.array(predictions)
```

Putting Everything Together.

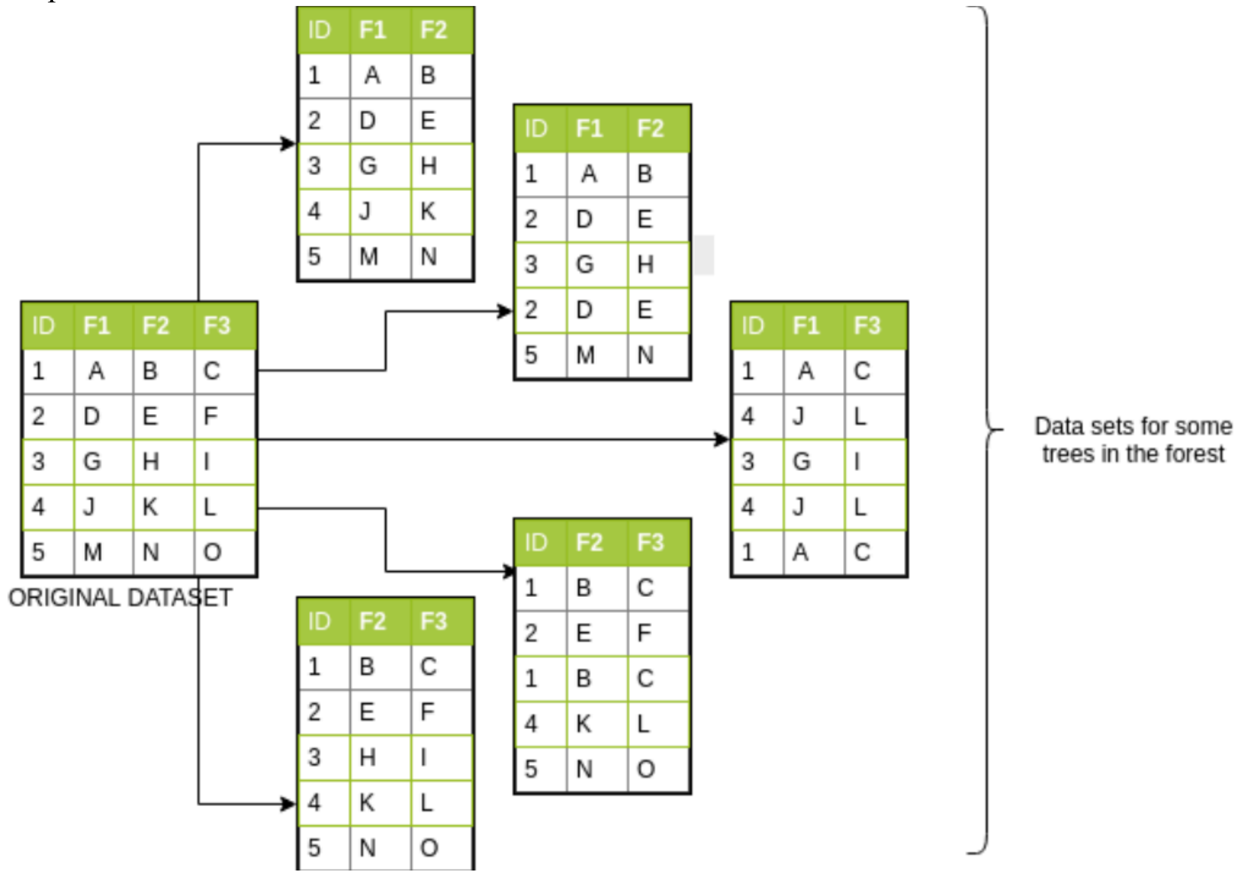
```
dcc = DecisionTree()
dcc.fit(x_train,y_train)
ypred = dcc.predict(x_test)
from sklearn.metrics import accuracy_score,confusion_matrix
accuracy_score(y_test,ypred),confusion_matrix(y_test,ypred)
```

we now defined our own decision tree class

Random Forest

Random forest builds multiple decision trees and merges them together to get a more accurate and stable prediction.

Random forest is a way to reduce overfitting in decision trees and it can also be used to find importance of features we are using.
Each decision tree built will have a randomly selected set of features and randomly selected set of data points. Number of features in each decision tree in the forest will be less than the total number of features we have in our dataset. So if we have a feature 'A', it may appear in some of the decision trees of the forest and not in others. Duplication is generally allowed in selecting the data-points from our data-set for the decision trees in the forest.



ID	F1	F2	F3
1	A	B	C
2	D	E	F
3	G	H	I
4	J	K	L
5	M	N	O

ID	F1	F2
1	A	B
2	D	E
3	G	H
4	J	K
5	M	N

ID	F1	F2
1	A	B
2	D	E
3	G	H
2	D	E
5	M	N

ID	F1	F3
1	A	C
4	J	L
3	G	I
4	J	L
1	A	C

ID	F2	F3
1	B	C
2	E	F
1	B	C
4	K	L
5	N	O

ID	F2	F3
1	B	C
2	E	F
3	H	I
4	K	L
5	N	O

As shown above, some of the trees have features F1 and F2 but not F3. There is no point of repeating a feature in a decision tree of the forest but as shown data point may be duplicate. This randomness in selecting the features and data-points helps in reducing overfitting. Note that we will make multiple decision trees so that there is very less chance of a feature/data-point getting missed out.

Random forest consists of many decision tress. Final answer of prediction is the majority of the answers from decision trees of the forest.

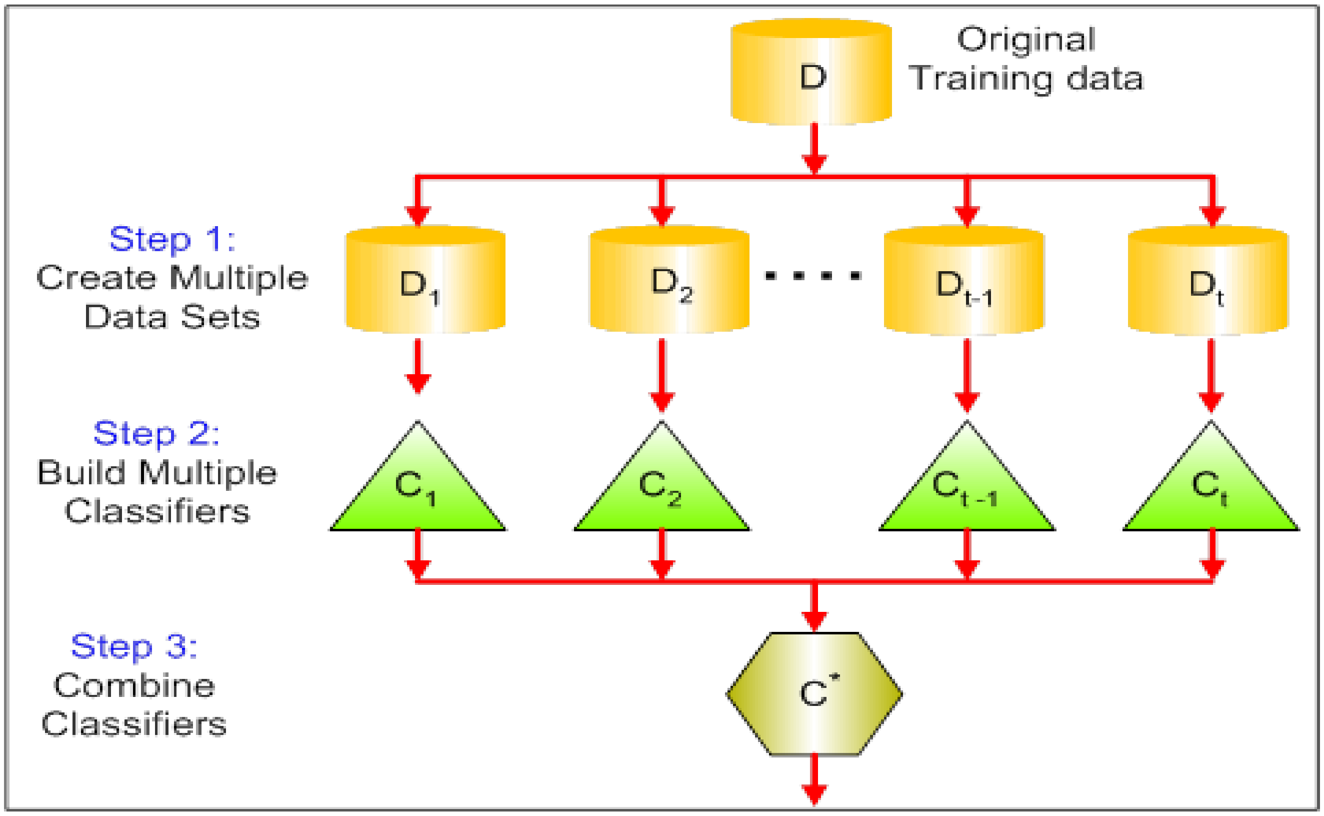
There are a few important ways to do so, let's have a look.

Data Bagging and Feature Selection

Bootstrap Aggregation or Bagging is a simple but very powerful ensemble method.

An ensemble method is a technique that combines the predictions from multiple machine learning algorithms together to make more accurate predictions than any individual model .

Bagging is used when our goal is to reduce the variance of a decision tree. It is a general purpose procedure for reducing the variance of a predictive model. When applied to trees the basic idea is to grow multiple trees which are then combined to give a single prediction. Combining multiple trees helps in improving precision and accuracy at the expense of interpretation.
In bagging, we take multiple smaller data-sets in which we also allow repetition of data points and randomly select some features. Bagging is generally done in reference of data-points.


D
Original Training data
Step 1: Create Multiple Data Sets
D_1
D_2
D_{t-1}
D_t
Step 2: Build Multiple Classifiers
C_1
C_2
C_{t-1}
C_t
Step 3: Combine Classifiers
C^*

As shown in the diagram above we have created multiple data-sets from the original dataset shown as D1, D2 etc. Classifiers C1, C2 etc are actually individual trees in our forest. To find the final answer we just take majority of the answers given by these trees. These smaller data-sets are obtained by choosing the data-points and the features in the following manner:

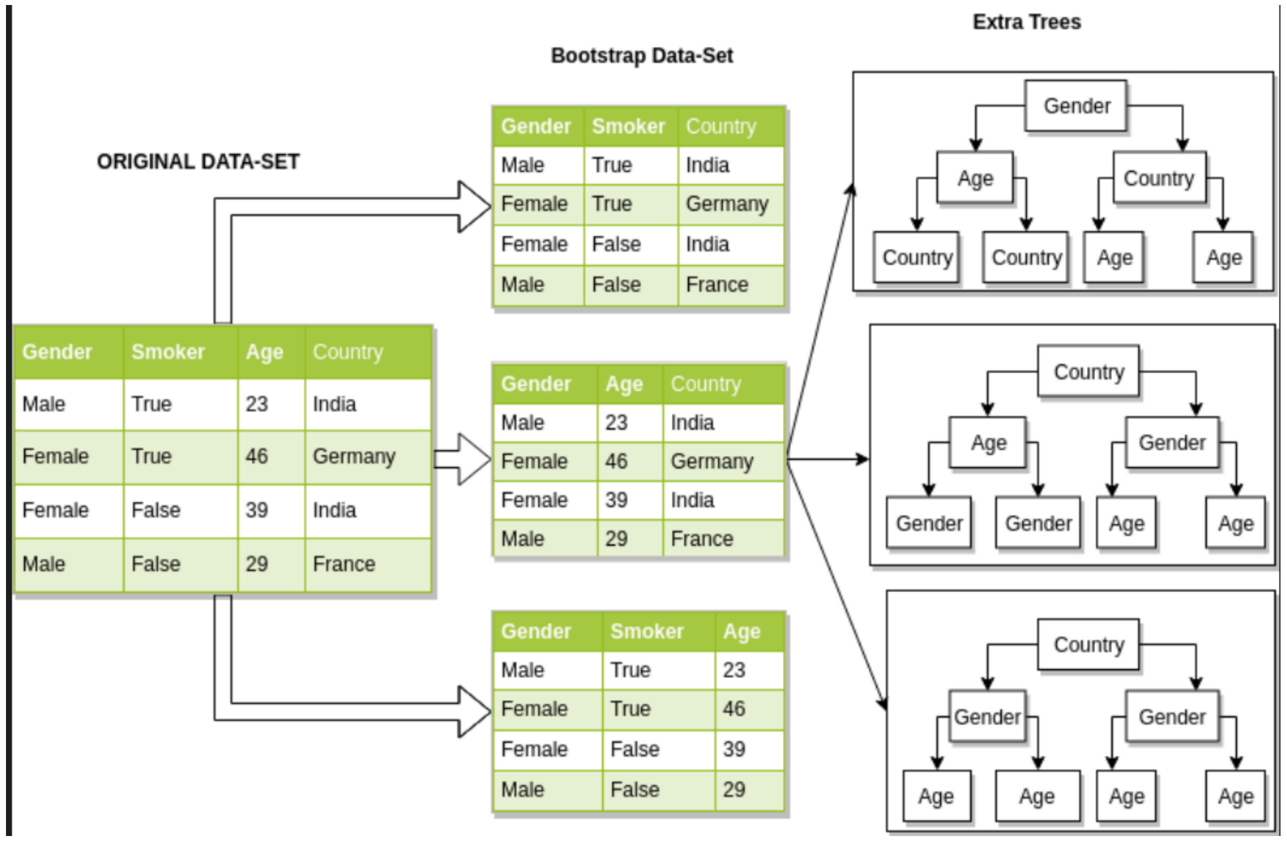
1. Features are selected at random without repetition
2. Data-points are selected at random with repetition (which is actually bagging)

While doing bagging, we must be sure that no data point is left out. Increasing the number of trees in the random forest significantly reduces the chances of missing out on any data points. Same goes for the features. As already discussed, selecting features in random forest helps us in knowing the relative importance of each feature.

Extra Trees

The Extra-Tree method stands for extremely randomized trees. With respect to random forests, the method drops the idea of using bootstrap copies of the learning sample, and instead of trying to find an optimal cut-point for each one of the K randomly chosen features at each node, it selects a cut-point at random.
In the implementation of Random Forest, we randomly select some features from the main data-set. Then for these randomly selected features we make a decision tree. The decision tree is made by calculating which feature should be selected to split at a particular node. The cost of choosing each feature was calculated and the feature which gave the least cost was selected. Multiple trees were made using the same approach to form a forest.
In the Extra Trees approach, we do not choose the features randomly to form a tree. We take all the features to form a tree. This is the first difference. The second one is in selecting the feature to split the data points at each node. Rather than considering the cost due to taking a certain feature to split Extra Trees just pick a feature at random. So in this case, any two trees will be different in terms of the feature selected to partition the data-points at each node. It is quite possible in both the approaches discussed that there exists a pair of trees that are exactly same.

We can combine Extra Trees approach with Random forests as well. This can be done by selecting randomly some features as Random forest does and then applying Extra Trees to this feature subset and make multiple Trees out of the bootstrapped data-set formed.



ORIGINAL DATA-SET

Gender	Smoker	Age	Country
Male	True	23	India
Female	True	46	Germany
Female	False	39	India
Male	False	29	France

Bootstrap Data-Set

Gender	Smoker	Country
Male	True	India
Female	True	Germany
Female	False	India
Male	False	France

Gender	Age	Country
Male	23	India
Female	46	Germany
Female	39	India
Male	29	France

Gender	Smoker	Age
Male	True	23
Female	True	46
Female	False	39
Male	False	29

As shown in the image above, the data-set is first converted into several bootstrap data-sets and then Extra Trees are made for each of these bootstrap data-sets.

Random Forest Using Sklearn

```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn import datasets,tree
from sklearn.model_selection import train_test_split
from sklearn.feature_selection import SelectFromModel
from sklearn.metrics import accuracy_score
import pandas as pd
from IPython.display import Image
import pydotplus
import graphviz
```

we start by importing all the packages.

```
iris = datasets.load_iris()
features = iris.feature_names
X = iris.data
Y = iris.target
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.4,
random_state = 14)
```

we load iris dataset and split inti training and testing.

```
clf = RandomForestClassifier(n_estimators=10000, n_jobs=-1, random_state = 14)
clf.fit(X_train, Y_train)
clf.score(X_test, Y_test)
```

the score that we get is : 0.95 which is better than decision trees that too on testing data.

```
sfm = SelectFromModel(clf, threshold = 0.15)
sfm.fit(X_train, Y_train)
X_important_train = sfm.transform(X_train)
X_important_test = sfm.transform(X_test)
clf_important = RandomForestClassifier(n_estimators=10000, n_jobs=-1, random_state
= 14)
clf_important.fit(X_important_train, Y_train)
clf_important.score(X_important_test, Y_test)
```

after applying feature scaling our score increases from 0.95 to 0.96

writing own random forest

```
class RandomForests:
    def
__init__(self,num_trees=10,min_sample_split=2,max_depth=10,n_features=1000000):
        self.n_trees=num_trees
        self.min_sample_split=min_sample_split
        self.max_depth=max_depth
        self.n_features=n_features
        self.trees=[]

    def fit(self,X,Y):
        n_data_points,n_features = X.shape
        self.n_features = n_features if self.n_features > n_features else
self.n_features
        self.trees=[]
        for i in range(self.n_trees):
            tree =
DecisionTree(max_depth=self.max_depth,min_samples_split=self.min_sample_split,n_fe
atures=self.n_features)
            X_sample,Y_sample = self.bootstrap_samples(X,Y)
            tree.fit(X_sample,Y_sample)
            self.trees.append(tree)

    def bootstrap_samples(self,X,Y):
        n_sample = X.shape[0]
        idxs = np.random.choice(n_sample,n_sample,replace=True)
        return X[idxs],Y[idxs]

    def most_common_class(self,y):
        return Counter(y).most_common(1)[0][0]

    def predict(self,X):
        predictions = np.array([tree.predict(X) for tree in self.trees])
        predictions = np.array([self.most_common_class(arr) for arr in
predictions.T])
        return predictions
```

we use previous code of decision tress and perform all the operations.

```
rf = RandomForests()
rf.fit(x_train,y_train)
ypred = rf.predict(x_test)
from sklearn.metrics import accuracy_score,confusion_matrix
accuracy_score(y_test,ypred),confusion_matrix(y_test,ypred)
```

the accuracy score will be 0.97 here we can see we get better score than that of sklearn the confusion matrix are as follows.

```
[55,  3],
[ 1, 84]
```

confusion matrix for our classifier.

Naive Bayes

Bayes Theorem defines probability of an event based on the prior knowledge of factors that might be related to an event.

$$P(A \mid B) = \frac{P(B \mid A) * P(A)}{P(B)}$$

where **A** and **B** are events and **P(B) != 0**

Here, a few points to note are :

P(A|B) is a conditional probability: the likelihood of event **A** occurring given that event **B** has occurred.

P(B|A) is also a conditional probability: the likelihood of event **B** occurring given that event **A** has occurred.

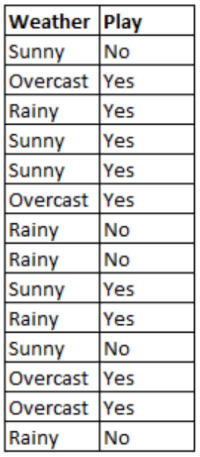
P(A) and **P(B)** are the probabilities of observing A and B independently of each other.

How Naive Bayes Works?

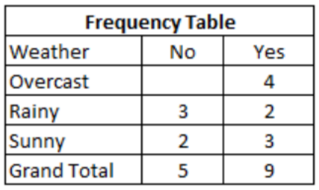
Let's understand it by using an example. Below we have a training data set of weather and corresponding target variable 'Play' (suggesting possibilities of playing). Now, we need to classify whether players will play or not based on weather. Let's follow the following steps to perform it.

Step 1: Convert the data set into a frequency table

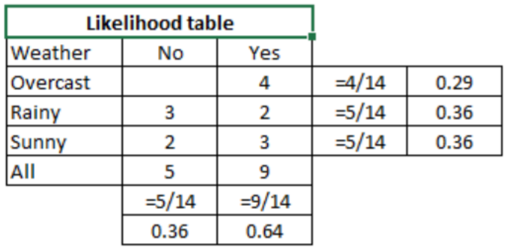
Step 2: Create a Likelihood table by finding the probabilities like Overcast probability = 0.29 and probability of playing = 0.64.



Weather	Play
Sunny	No
Overcast	Yes
Rainy	Yes
Sunny	Yes
Sunny	Yes
Overcast	Yes
Rainy	No
Rainy	No
Sunny	Yes
Rainy	Yes
Sunny	No
Overcast	Yes
Overcast	Yes
Rainy	No



Frequency Table		
Weather	No	Yes
Overcast		4
Rainy	3	2
Sunny	2	3
Grand Total	5	9



Likelihood table				
Weather	No	Yes		
Overcast		4	=4/14	0.29
Rainy	3	2	=5/14	0.36
Sunny	2	3	=5/14	0.36
All	5	9		
	=5/14	=9/14		
	0.36	0.64		

(In this image, a blank cell represents '0')

Step 3: Now, use Naive Bayesian equation to calculate the posterior probability for each class. The class with the highest posterior probability is the outcome of prediction.

Problem: Players will play if weather is sunny. Is this statement is correct?

We can solve it using above discussed method of posterior probability.

$$P(Yes \mid Sunny) = \frac{P(Sunny \mid Yes) * P(Yes)}{P(Sunny)}$$

Here we have $P(Sunny|Yes) = 3/9 = 0.33$, $P(Sunny) = 5/14 = 0.36$, $P(Yes) = 9/14 = 0.64$

Now,

$$P(Yes \mid Sunny) = \frac{0.33 * 0.64}{0.36} = 0.59$$

which is a high probability.

Naive Bayes uses a similar method to predict the probability of different classes based on various attributes. This algorithm is mostly used in text classification and with problems having multiple classes.

Now, basically for a data point $\mathbf{x_i}$, we have to predict the class that the current output **y** belongs to. Assume, there are total **j** number of classes for output.
Then,

$P(y=c_1 \mid x=x_i)$: tells us that for given input $\mathbf{x_i}$ what is the probability that **y** is $\mathbf{c_1}$.

$P(y=c_2 \mid x=x_i)$: tells us that for given input $\mathbf{x_i}$ what is the probability that **y** is $\mathbf{c_2}$.

and so on till $\mathbf{c_j}$

Out of all these probabilities calculations, **y** belongs to that particular class which has maximum probability.

We will be using Bayes theorem for doing these probability calculations. The formula is given as:

$$P(y = c_j \mid x = x_i) = \frac{P(x = x_i \mid y = c_j) * P(y = c_j)}{P(x = x_i)}$$

This gives us the probability that the output belongs to **j**$^{\mathbf{th}}$ class for the current values of data point **x**$_{\mathbf{i}}$.

Since for all the classes **1, 2,..., j** the denominator will have the same value, so we can ignore this while doing comparison (ultimately, we have to find the max). Hence, we obtain the given formula to calculate probabilities.

Naive Assumption

The estimate for probability **P(y=c$_j$)**, can be done directly from the number of training points. Suppose there are 100 training points and 3 output classes, **10 belong to class C$_1$**, **30 belong to class C$_2$** and remaining **60 belong to class C$_3$**.
The estimate values of class probabilities will be:
P(y = C$_1$) = 10/100 = 0.1
P(y = C$_2$) = 30/100 = 0.3
P(y = C$_3$) = 60/100 = 0.6

To make the probability estimate for **P(x=x$_i$ | y=c$_j$)**, naive bayes classification algorithm assumes all the features to be independent.

So, we can calculate this by individually multiplying the probabilities obtained for all these features (assuming features to be independent), for the output of **j**$^{\mathbf{th}}$ class

$$P(x = x_i | y = c_j) = P(x = x_i^1 \mid y = c_j) * P(x = x_i^2 \mid y = c_j) * .... * P(x = x_i^n \mid y = c_j)$$

Here, x_i^1 denotes the value of 1st feature of i^{th} data point and $x = x_i^n$ denotes the value n^{th} feature of the i^{th} data point.

After taking up the naive assumption, we can easily calculate the individual probabilites and then by simply multiplying the result calculate the final probability P'.

$$P'(y = c_j \mid x = x_i) = \prod_k P(x = x_i^k \mid y = c_j) * P(y = c_j)$$

.
Finding the second term in the above formula is pretty straight forward, we shall simply divide the total training data belonging to class $c_j$, with total training data. Let's have a look at the first term.

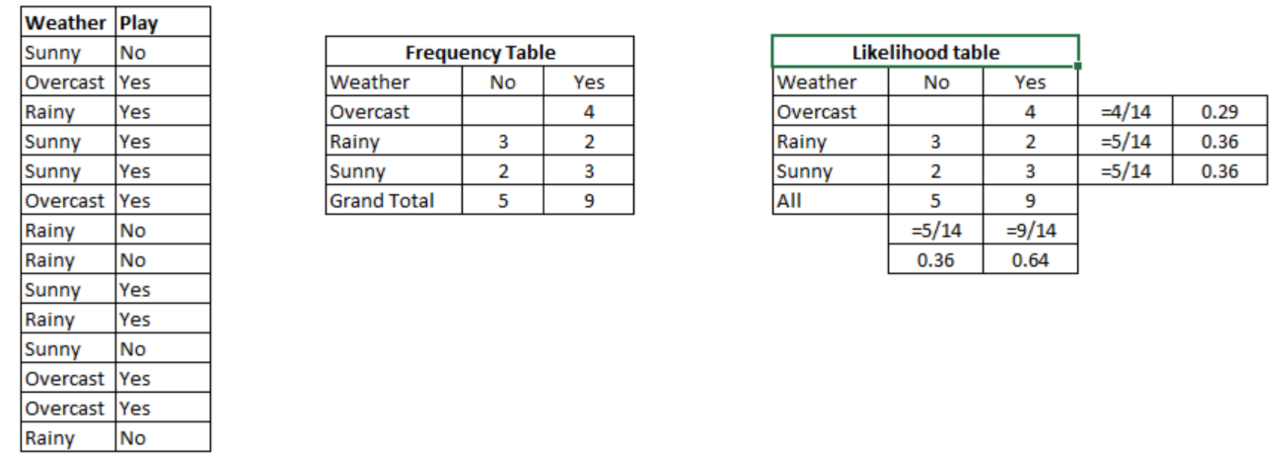
Implementation with Discrete Data

Now, let's see how we can find $\mathbf{P(x = x_i^k \mid y = c_j)}$ for any one class.

Let's assume our selected class to be **c**, and the value of the input feature to be $\mathbf{x^j}$, so the probability $\mathbf{P(x = x^j \mid y = c)}$ is given by

$$P(x = x^j \mid y = c) = \frac{Count\ of\ Training\ Data(x = x^j\ and\ y = c)}{Count\ of\ Training\ Data(y = c)}$$

Let's look at the previous example.



Weather	Play
Sunny	No
Overcast	Yes
Rainy	Yes
Sunny	Yes
Sunny	Yes
Overcast	Yes
Rainy	No
Rainy	No
Sunny	Yes
Rainy	Yes
Sunny	No
Overcast	Yes
Overcast	Yes
Rainy	No

Frequency Table		
Weather	No	Yes
Overcast		4
Rainy	3	2
Sunny	2	3
Grand Total	5	9

Likelihood table				
Weather	No	Yes		
Overcast		4	=4/14	0.29
Rainy	3	2	=5/14	0.36
Sunny	2	3	=5/14	0.36
All	5	9		
	=5/14	=9/14		
	0.36	0.64		

Here, let's find the probability $P(x = Sunny \mid y = No)$

$$P(x = Sunny \mid y = No) = \frac{Count\ of\ Training\ Data(x = Sunny\ and\ y = c)}{Count\ of\ Training\ Data(y = No)}$$

$$= \frac{2}{5}$$

The above example considers only one feature for our input. To maintain computations for multi featured input, we shall use dictionaries while we write the code. Let us look at the dictionary structure.

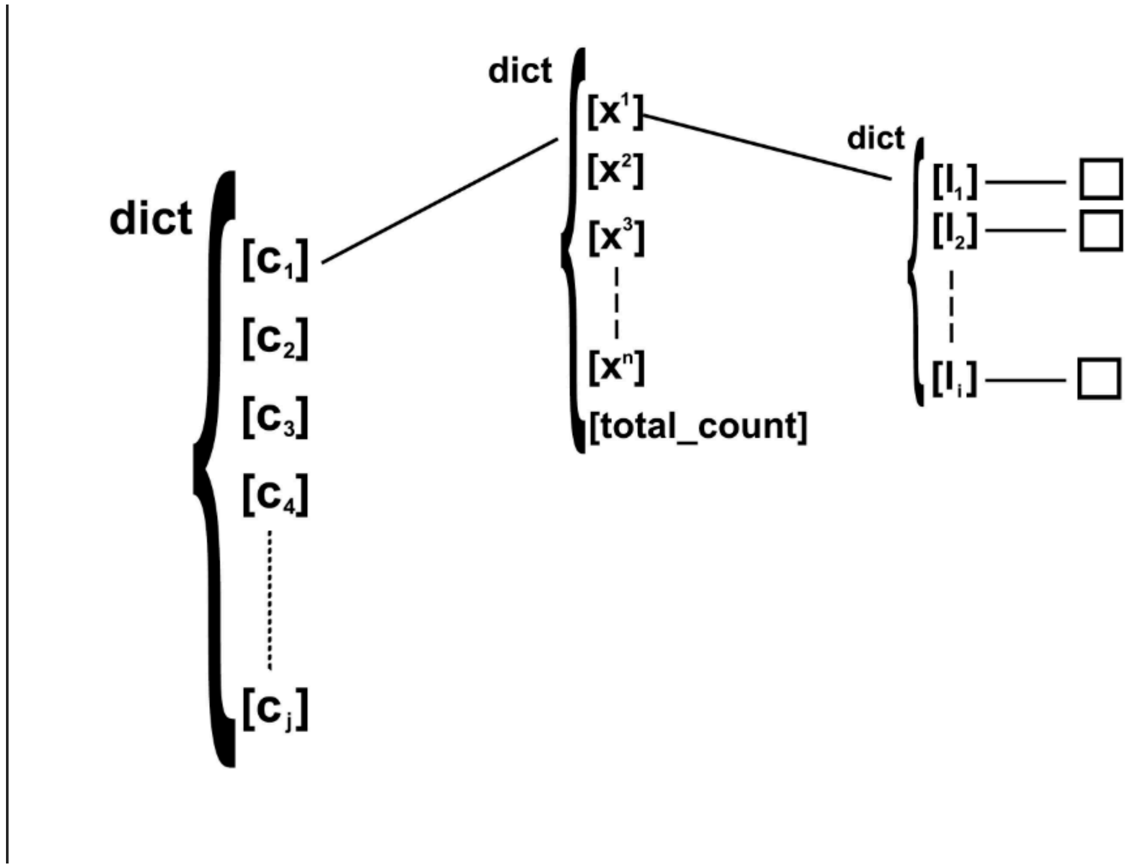
We will implement a multi level dictionary. At the first level, we will store the classes $\mathbf{c_1, c_2, c_3, \ldots, c_j}$ as keys, to which the data belongs to.

For each class key we will store another dictionary (second level), where the keys will be the features, $\mathbf{x_j^1, x_j^2, x_j^3, \ldots, x_n^j}$, where n is the number of features.

For each feature, we will store another dictionary (third level), where the keys will be all the possible values that feature can take. The keys of this dictionary will store the corresponding count.

Note: Apart from storing the feature dictionaries, the top level dictionary of each class will store one extra key, where the value would be the frequency of occurrence (total count) of that particular class.

Below is the diagrammatic structure of the same.



where $\mathbf{c_1, c_2, ..., c_j}$ represent the classes, $\mathbf{x^1, x^2, x^3...., x^n}$ represent the features and $\mathbf{l_1, l_2, l_3, ..., l_i}$ represent the possible labels of each feature.

Laplace Correction

Let's consider the following situation: you've trained a Naive Bayes algorithm to differentiate between spam and not spam mails. What happens if the word "Casino" doesn't show up in your training data set, but appears in a test sample?

Well, your algorithm has never seen it before, so it sets the probability that "Casino" appears in a spam document to 0. So every time this word appears in the test data , you will try hard **(it has P = 0)** to mark it as not spam just because you have not seen that word in the spam part of training data.This will make the model very less efficient and thus we want to minimise it. We want to keep in mind the possibility of any word we have not seen (or for that matter seen in the not-spam part of training data), may have a above-zero probability of being a word used in spam mails. The same is true for each word to be a part of not-spam mails.

To avoid such issues with unseen values for features, as well as to combat overfitting to the data set, we pretend as if we've seen each word 1 (or k, if you're smoothing by k) time more than we've actually seen it, and adjust the denominator of our frequency divisions by the size of the overall vocabulary to account for the "pretence", which actually works well in practice.

If you take smoothing factor k equal to 1, it becomes Laplace correction.
The equations below show Laplace correction for the example taken.

Without correction :

$$P(w_i \mid c_j) = \frac{count(w_i, c_j)}{\sum_w count(w, c_j)}$$

With correction :

$$P(w_i \mid c) = \frac{count(w_i, c_j) + 1}{(\sum_w count(w, c_j)) + |V|}$$

where V is the number of all possible words in mails with topic c_j.

Self Implementation of Naive Bayes

```
import numpy as np
from sklearn import datasets
from sklearn import model_selection
```

import all the required models.

```
def fit(X_train, Y_train):
    result = {}
    class_values = set(Y_train)
    for current_class in class_values:
        result[current_class] = {}
        result["total_data"] = len(Y_train)
        current_class_rows = (Y_train == current_class)
        X_train_current = X_train[current_class_rows]
        Y_train_current = Y_train[current_class_rows]
        num_features = X_train.shape[1]
        result[current_class]["total_count"] = len(Y_train_current)
        for j in range(1, num_features + 1):
            result[current_class][j] = {}
            all_possible_values = set(X_train[:, j - 1])
            for current_value in all_possible_values:
                result[current_class][j][current_value] = (X_train_current[:, j -
1] == current_value).sum()
    return result
```

fit() function turns the data into the dictionary with probabilities.

```
def probability(dictionary, x, current_class):
    output = np.log(dictionary[current_class]["total_count"]) -
np.log(dictionary["total_data"])
    num_features = len(dictionary[current_class].keys()) - 1
    for j in range(1, num_features + 1):
        xj = x[j - 1]
        count_current_class_with_value_xj = dictionary[current_class][j][xj] + 1
        count_current_class = dictionary[current_class]["total_count"] +
len(dictionary[current_class][j].keys())
        current_xj_probablity = np.log(count_current_class_with_value_xj) -
np.log(count_current_class)
        output = output + current_xj_probablity
    return output
```

dictionary will be result returned by fit() function and one thing to note is we take log values instead of original values specified in formulas above due to which we perform addition wherever there is multiplication and subtraction wherever there is division we do this to avoid multiplication with a 0 probability

```
def probability(dictionary, x, current_class):
    output = dictionary[current_class]["total_count"] /  dictionary["total_data"]
    num_features = len(dictionary[current_class].keys()) - 1
    for j in range(1, num_features + 1):
        xj = x[j - 1]
        count_current_class_with_value_xj = dictionary[current_class][j][xj] + 1
        count_current_class = dictionary[current_class]["total_count"] +
len(dictionary[current_class][j].keys())
        current_xj_probablity = count_current_class_with_value_xj /
count_current_class
        output = output * current_xj_probablity
    return output
```

this is how our function looks like if we don't take log values.

```python
def predict_single_point(dictionary,x):
    classes = dictionary.keys()
    best_p = -1000
    best_class = -1
    first_run = True
    for current_class in classes:
        if (current_class == "total_data"):
            continue
        p_current_class = probability(dictionary, x, current_class)
        if (first_run or p_current_class > best_p):
            best_p = p_current_class
            best_class = current_class
        first_run = False
    return best_class
```

predicts the probability of best class for single point,

```python
def predict(result,x_test):
    y_pred=[]
    for x in x_test:
        x_class=predict_single_point(result,x)
        y_pred.append(x_class)
    return y_pred
```

just loops over every point and calls predict() function.

```python
def makeLabelled(column):
    second_limit = column.mean()
    first_limit = 0.5 * second_limit
    third_limit = 1.5 * second_limit
    for i in range (0,len(column)):
        if (column[i] < first_limit):
            column[i] = 0
        elif (column[i] < second_limit):
            column[i] = 1
        elif(column[i] < third_limit):
            column[i] = 2
        else:
            column[i] = 3
    return column
```

we use this function to handle numeric values in a feature,

```
iris = datasets.load_iris()
X = iris.data
Y = iris.target
for i in range(0,X.shape[-1]):
    X[:,i]=makeLabelled(X[:,i])
```

we load iris dataset and convert numeric to class based values.

```
X_train, x_test, Y_train, Y_test = model_selection.train_test_split(X,
Y,test_size=0.25, random_state=0)

result=fit(X_train,Y_train)
Y_pred = predict(dictionary, X_test)
```

we now have our own naive byes classifier.

```
# Various metrics for understanding how well our model has performed.
from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score

print("Classification Report")
print(classification_report(Y_test, Y_pred))
print("Confusion Matrix")
print(confusion_matrix(Y_test, Y_pred))
print()
print("Accuracy Score")
print(accuracy_score(Y_test, Y_pred) * 100, "%", sep="")
```

after we run this code we get

```
Classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        13
           1       0.94      1.00      0.97        16
           2       1.00      0.89      0.94         9

    accuracy                           0.97        38
   macro avg       0.98      0.96      0.97        38
weighted avg       0.98      0.97      0.97        38

Confusion Matrix
[[13  0  0]
 [ 0 16  0]
 [ 0  1  8]]

Accuracy Score
97.36842105263158%
```

implementation using sk-learn

```
from sklearn.naive_bayes import CategoricalNB
clf = naive_bayes.GaussianNB()
clf.fit(X_train, Y_train)
Y_pred_CategoricalNB = clf.predict(X_test)
```

its as simple as this.

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

print("Classification Report")
print(classification_report(Y_test,y_pred))
print("Confusion Matrix")
print(confusion_matrix(Y_test,y_pred))
print()
print("Accuracy Score")
print(accuracy_score(Y_test,y_pred) * 100, "%", sep="")
```

```
Classification Report
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        13
           1       1.00      1.00      1.00        16
           2       1.00      1.00      1.00         9

    accuracy                           1.00        38
   macro avg       1.00      1.00      1.00        38
weighted avg       1.00      1.00      1.00        38

Confusion Matrix
[[13  0  0]
 [ 0 16  0]
 [ 0  0  9]]

Accuracy Score
100.0%
```

we get 100% accuracy score mainly because we use make labelled function but inbuilt naive bayes uses gaussian kernel resulting in better accuracy.

How to improve naive bayes ?

1. If continuous features do not have normal distribution, we should use transformation or different methods to convert it in normal distribution.

2. If test data set has zero frequency issue, apply smoothing techniques “Laplace Correction” to predict the class of test data set.

3. Remove correlated features, as the highly correlated features are voted twice in the model and it can lead to over inflating importance.

4. You might think to apply some classifier combination technique like ensembling, bagging and boosting but these methods would not help. Actually, “ensembling, boosting, bagging” won’t help since their purpose is to reduce variance. Naive Bayes has no variance to minimize.

Unsupervised Learning

Unsupervised learning is a learning method in which a machine learns without any supervision. The training is provided to the machine with the set of data that has not been labeled, classified, or categorized, and the algorithm needs to act on that data without any supervision. The goal of unsupervised learning is to restructure the input data into new features or a group of objects with similar patterns.

Examples:

1. Feature selection
2. Anomaly detection
3. Pattern recognition
4. Data Mining

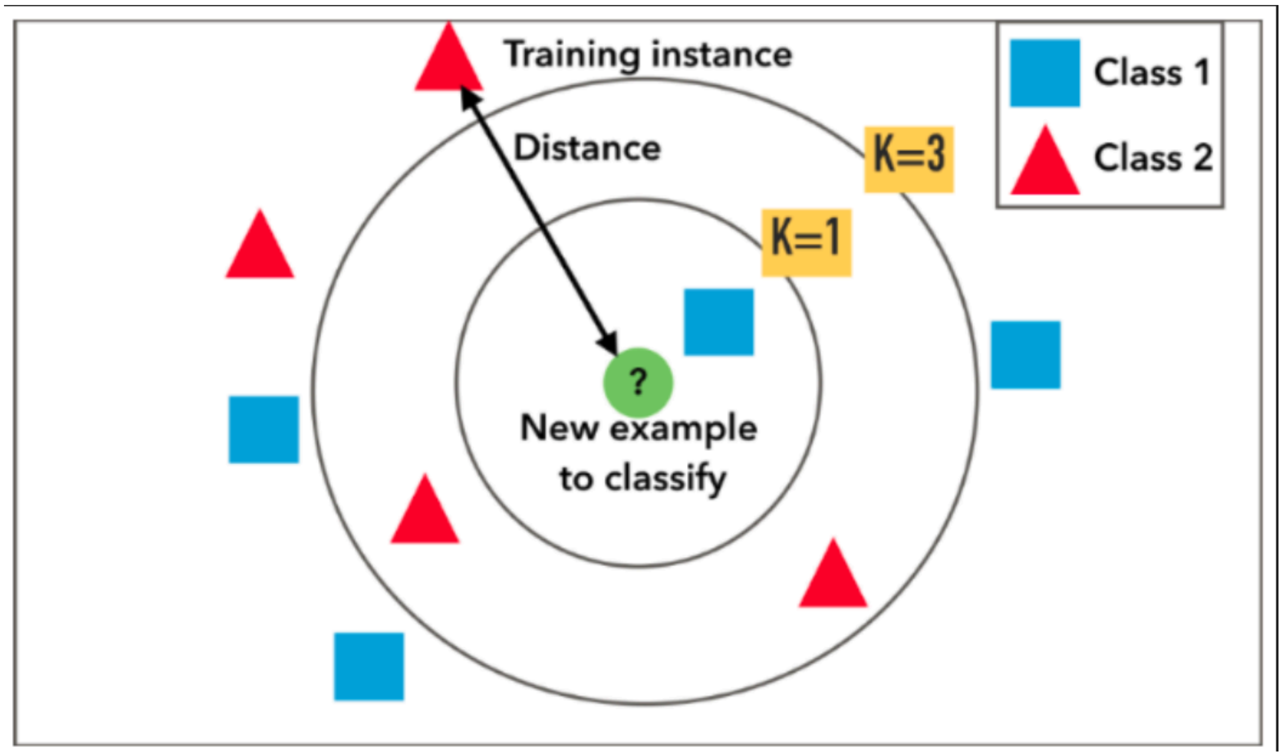
KNN

KNN stands for K-Nearest Neighbours. KNN is simple classification algorithm and it is generally used for datasets in which data points are separated into several classes and we have to predict the class for the new sample point.
KNN is non-parametric and lazy learning algorithm.

Non-parametric means that the algorithm does not make any assumptions on the given data distribution. Non-parametric covers technique that do not rely on data belonging to particular distribution and do not assume the structure of model to be fixed. So, KNN is used as classification algorithm in cases where we do not have much information about the distribution of the data.

KNN is referred to asLazy algorithm since is does not use training points to do any generalization, which means that there is no separate training phase. KNN keeps all the training data and uses most of the training data during the testing phase. Thus, KNN does not learn any model, it make predictions on the fly, computing similarity between testing point and each training data point.

KNN algorithm is based on **feature similarity**. We can classify the testing data point on the basis of resemblance of its features with that of the training data set.



The test sample (green circle) should either be classified into Class 1(blue squares) or Class 2 (red triangles). The class for the sample testing point is decided on the basis of majority vote-out.

How to choose the correct value of 'K'?

'K' in KNN is a parameter that refers to the number of nearest neighbours to include in the majority of the voting process.

If value of K is 1, the nearest training point belongs to Class 1, so we will say that the testing sample belongs to Class 1. Now, take the value of K to be 3, again using the methodology of majority vote, we will say that testing sample belongs to Class 2 (red triangles), since out of three nearest training data points, two belongs to Class 2 and one belong to Class 1.

1. 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.

2. Inversely, as we increase the value of K, our predictions become more stable due to majority voting / averaging, and thus, more likely to make more accurate predictions (up to a certain point). Eventually, we begin to witness an increasing number of errors. It is at this point we know we have pushed the value of K too far.

3. In cases where we are taking a majority vote (e.g. picking the mode in a classification problem) among labels, we usually make K an odd number to have a tiebreaker.

KNN can be used for Classification as well as Regression.

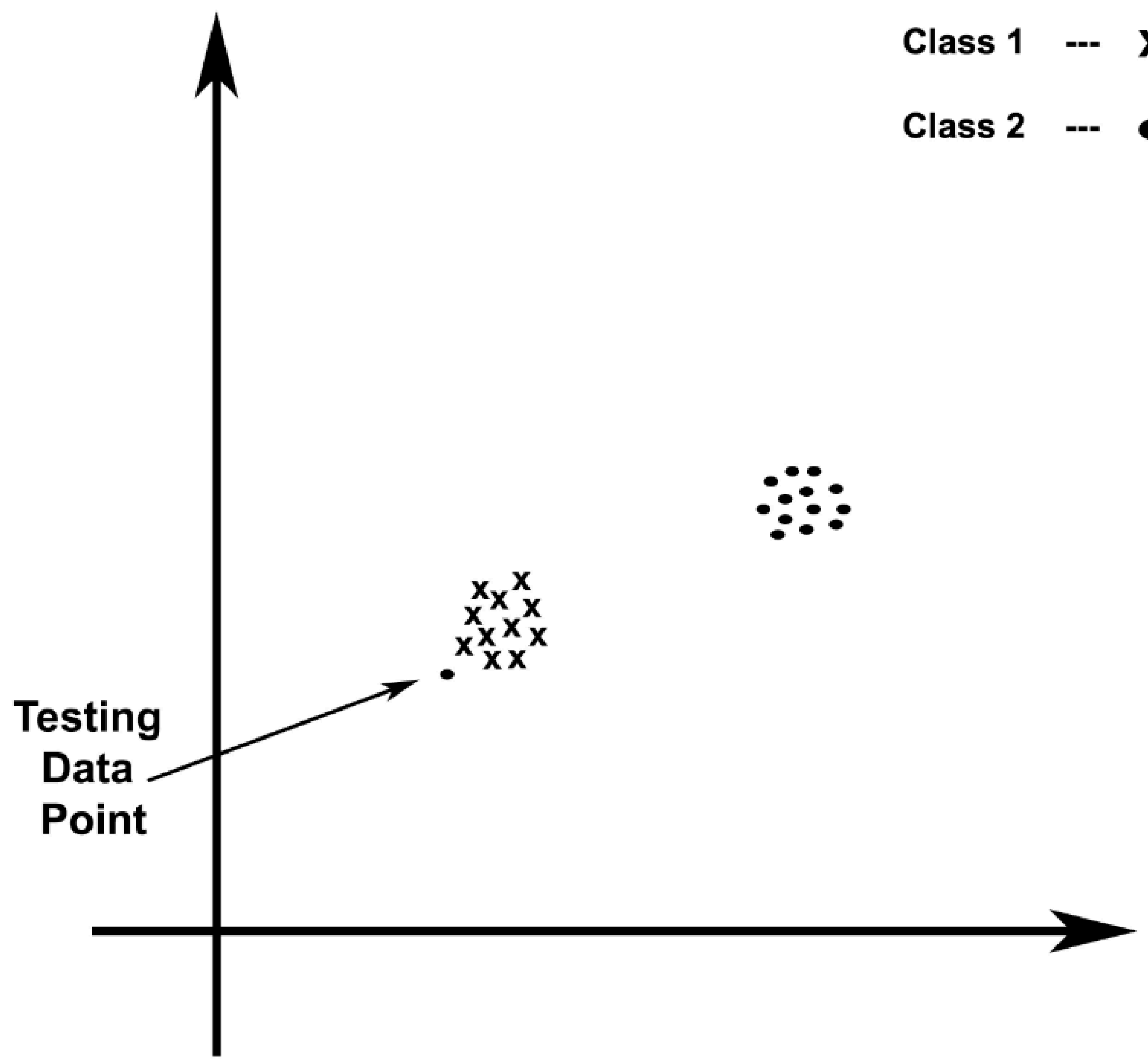
While using KNN for classification - output is a class membership, we will classify the sample point using the technique of majority vote among its neighbors and the most common class among its K nearest neighbours is assigned to the testing point. In regression, output is the property value of object, this value is average or median of the value of its K nearest neighbors.

In classification, KNN is used to predict a class which is a discrete value whereas in regression, KNN predicts continuous values

Why don't we choose value value of K to be 1?

Choosing the value of K to be 1 makes our model more prone to outliers and overfitting. Value of 1 means that we will consider only the closest (or nearest) neighbor to predict the class for our testing sample, and in majority of the cases it will lead to overfitting

---



Consider the case here, if we choose value of K to be 1, the given training sample will get classified as dot (Class 2), instead of being classified as a cross (Class 1). So, in such cases certain optimal value of K should be chosen to get good results. Choosing the value of K to be 1, leads to formation of complex decision boundaries and hence will lead to overfitting.

Since we will be using the majority vote technique, so value of K is taken to be odd, to obtain clear result about the class of the testing data sample.

Distance Metric for KNN

There are various distance metrices that can be chosen such as Manhattan Distance, Euclidian Distance, etc.

$$Manhattan\ Distance = |\sum_{i=1}^{n} X_1^i - X_2^i|$$

$$Euclidian\ Distance = \sqrt{(\sum_{i=1}^{n}(X_1^i - X_2^i)^2)}$$

Variations in KNN

Variations in KNN are possible on the basis of how neighbouring points are going to vote. The weight of the vote of the testing point(s) is inversely proportional to its distance from the testing point. We can go either with uniform voting or with weighted voting. In case of weighted voting, the point nearer to the testing sample will have a larger say in the vote as compared to the point which is farther away from the testing point

Feature Scaling before KNN

In KNN, we are looking for points which are closest to the testing point. In case we do not perform feature scaling, if we have value of one feature in thousands and value of other feature in smaller units, then the effect of first feature will completely overpower and dominate over the effect of second feature in the final output. So, it is of utmost importance to apply feature scaling before applying KNN so that all the features have equal contribution in the final predicted output for the given testing point.

KNN in Sklearn

```python
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt

cancer = datasets.load_breast_cancer()
x_train, x_test, y_train, y_test = train_test_split(cancer.data, cancer.target,
test_size = 0.2, random_state = 0)
```

we load brest cancer dataset and split it into training and testing.

```python
clf = KNeighborsClassifier()
clf.fit(x_train, y_train)
clf.score(x_test, y_test)
```

the score comes out to be 0.9385964912280702

Default value of K (number of neighbors) is equal to 5 in Sklearn. By default, Sklearn implements Minkowski distance metric. General form of Minkowski distance is :

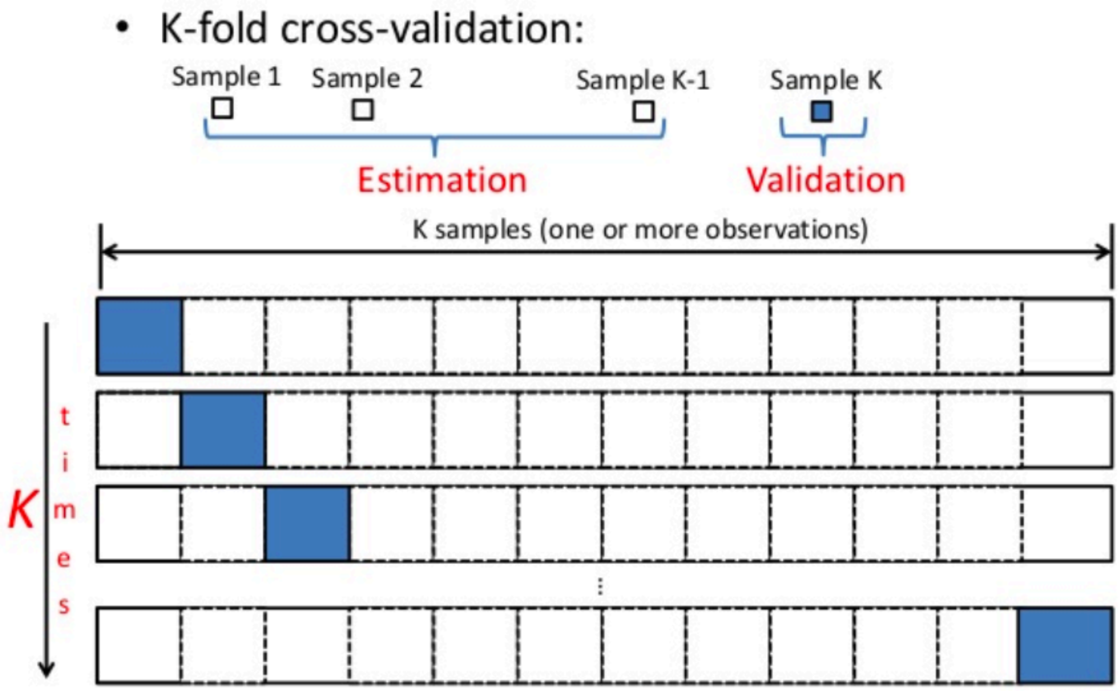
$$Minkowski\ Distance = |\sum_{i=1}^{n}(X_1^i - X_2^i)^p|^{\frac{1}{p}}$$

If p = 1, that means we are using Manhattan Distance and if p = 2, that means we are using Euclidian Distance

Cross Validation

The best value of K is the one for which we get lowest error on testing data. So, what we can do is to repeatedly train our model using both training and testing data, for different values of parameter K and then finally choose value of K which results in minimum error. But in this process, we are using the testing data as a part of our training process to obtain the optimal value of K. Hence, this process is not to be used. On the other hand if we use only training data and tune the value of parameter K, it will lead to overfitting. This will result in lower value of error on training data, but comparatively higher value of error on the testing data.

So, to obtain the optimal value of K, we use a method known as CROSS VALIDATION. Cross Validation basically means taking out the subset from the training data and not using this subset in the training process. This subset of training data is called the 'validation set'. There are various techniques available for cross validation, we will be using the most general one, known as K-fold cross validation.

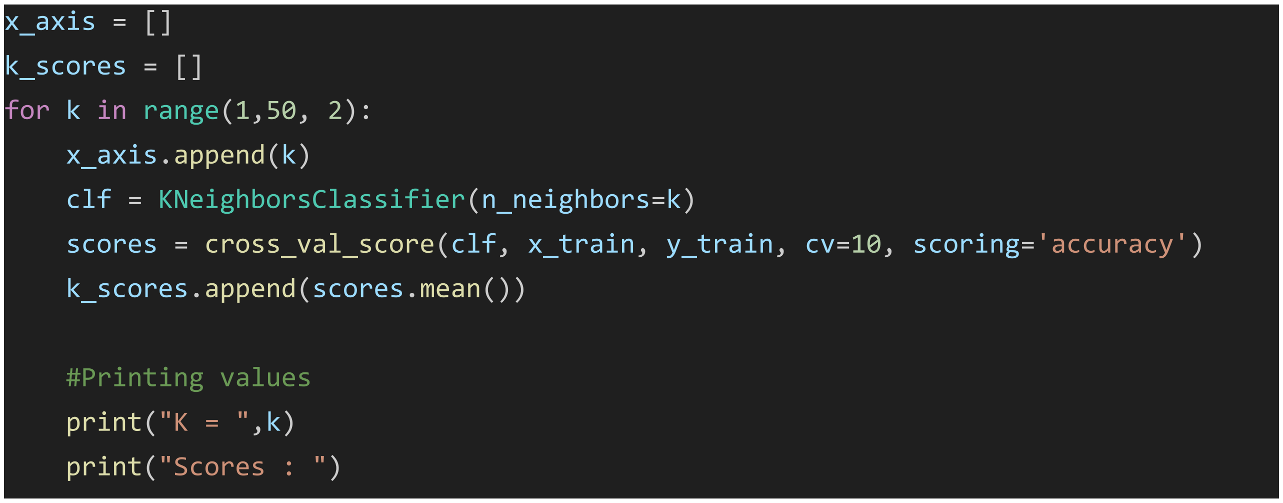


In K-fold cross validation, the training data is randomly split into K different samples (or folds). One of the sample is taken to be the validation set and the model is fitted on the remaining (K - 1) samples. The accuracy of the model is then computed. The same process is repeated K times, each time taking a different sample of points to be in the validation set. This results in K values for test error and these values are averaged out to obtain the overall result.

Cross Validation is used to estimate the test error and generate more robust models

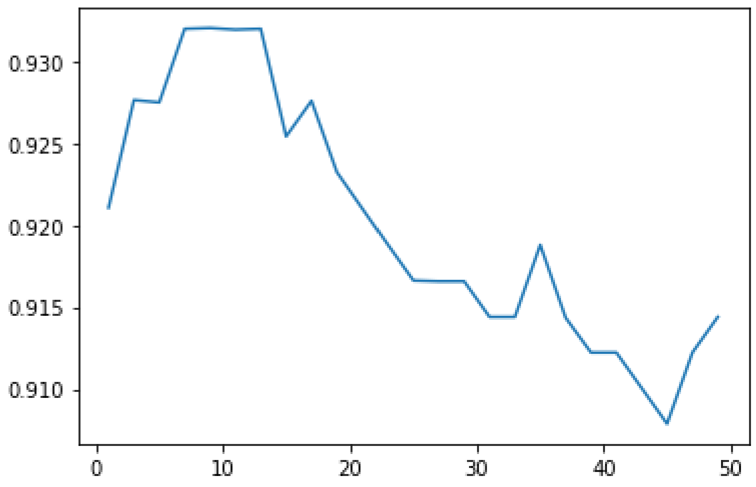
Cross Validation in Sklearn

The simplest way to use cross-validation is to call the cross_val_score helper function on the estimator and the dataset.



```
x_axis = []
k_scores = []
for k in range(1,50, 2):
    x_axis.append(k)
    clf = KNeighborsClassifier(n_neighbors=k)
    scores = cross_val_score(clf, x_train, y_train, cv=10, scoring='accuracy')
    k_scores.append(scores.mean())

    #Printing values
    print("K = ",k)
    print("Scores : ")
    print(scores)
    print("Mean Score = ",scores.mean())
```

```
plt.plot(x_axis, k_scores)
plt.show()
```



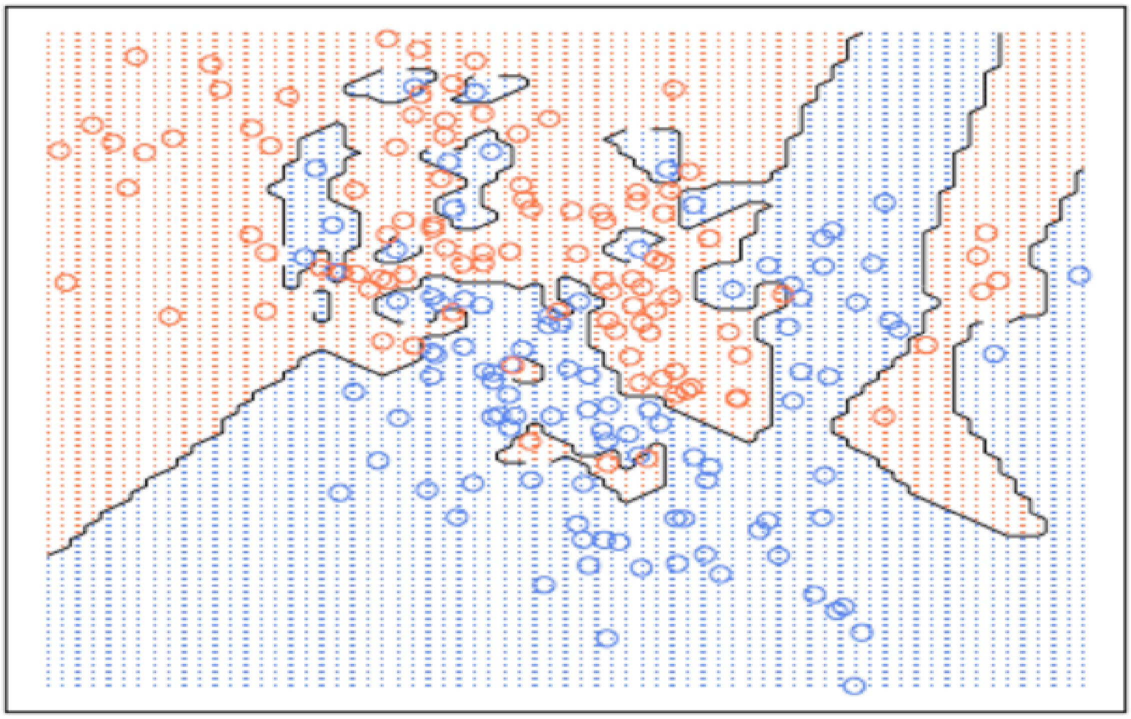
```
optimal_k = x_axis[k_scores.index(max(k_scores))]
optimal_k
```

this code outputs 9 which is our best k.

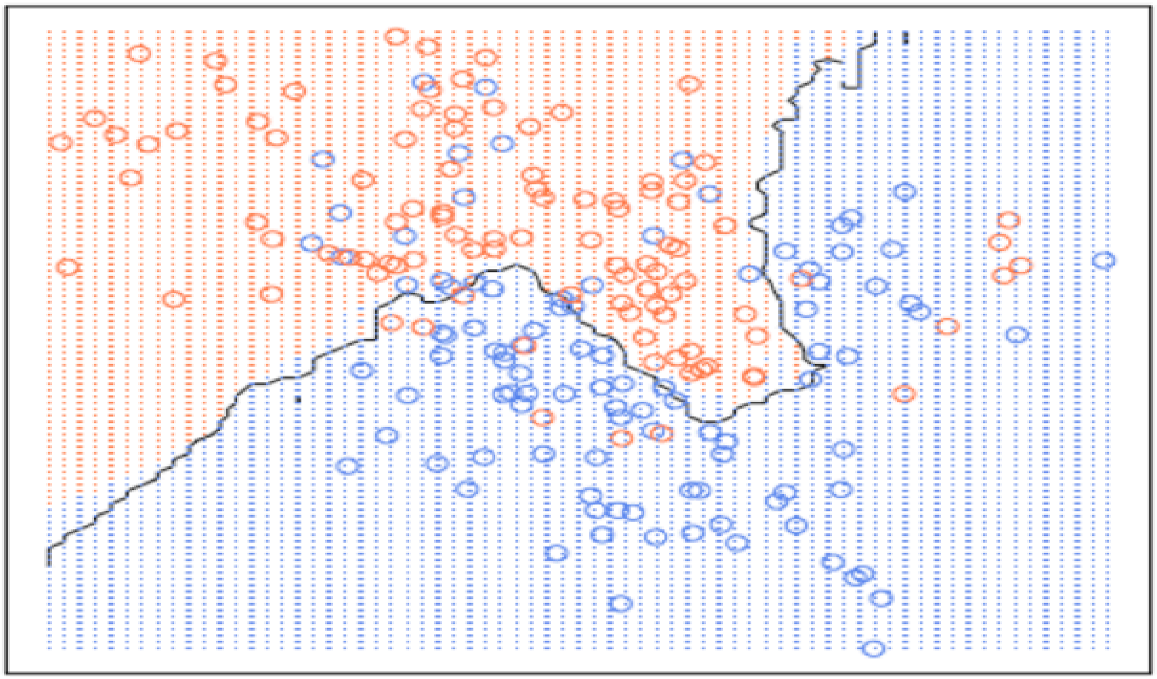
Using this way, we can choose the optimal value of K using cross validation. If the value of K is very less, for eg, say 1, it will lead to overfitting and will result in formation of complex decision boundaries. On the the hand, if the value of K is very high, we are basically underfitting and not actually taking class of neighbours into consideration. In this case, we predict the class of the testing sample from the majority class of the overall training data rather than from the majority class given by the neighbors.

Lower values of K corresponds to low bias but high variance and result in jagged and complex decision boundaries, while higher value of K corresponds to lower variance but increased bias and leads to formation of smoother decision boundaries.

nearest neighbour (k = 1)



20-nearest neighbour



Self Implementation of KNN

```
def predict_one(x_train, y_train, x_test, k):
    distances = []
    for i in range(len(x_train)):
        distance = ((x_train[i,:] - x_test)**2).sum()
        distances.append([distance,i])
    distances = sorted(distances)
    targets = []
    for i in range(k):
        index_of_training_data = distances[i][1]
        targets.append(y_train[index_of_training_data])
    return Counter(targets).most_common(1)[0][0]
```

```
def predict(x_train, y_train, x_test_data, k):
    predictions = []
    for x_test in x_test_data:
        predictions.append(predict_one(x_train, y_train, x_test, k))
    return predictions
```

calls predict one which classifies the data point to every point abd append it to y predicted.

```
y_pred = predict(X_train, Y_train, X_test, 7)
accuracy_score(Y_test, y_pred)
```

0.9473684210526315 will be our custom trained knn accuracy,

```
class KNN:
    def __init__(self,k=3):
        self.k=k

    def fit(self,x_train,y_train):
        self.x_train = np.array(x_train)
        self.y_train = np.array(y_train)

    def predict(self,x_test):
        y_preds = []
        x_test = np.array(x_test)
        for testing_point in x_test:
            distances = [[((x_train[i,:]-testing_point)**2).sum() , i] for i in
range(len(self.x_train))]
            distances = sorted(distances)
            mp = {cls:0 for cls in np.unique(self.y_train)}
            for k in range(self.k):
                mp[self.y_train[distances[k][1]]] += 1
            point_pred = -1000000
            for key in mp:
                if point_pred < mp[key]:
                    point_pred = key
            y_preds.append(point_pred)
        return y_preds
```

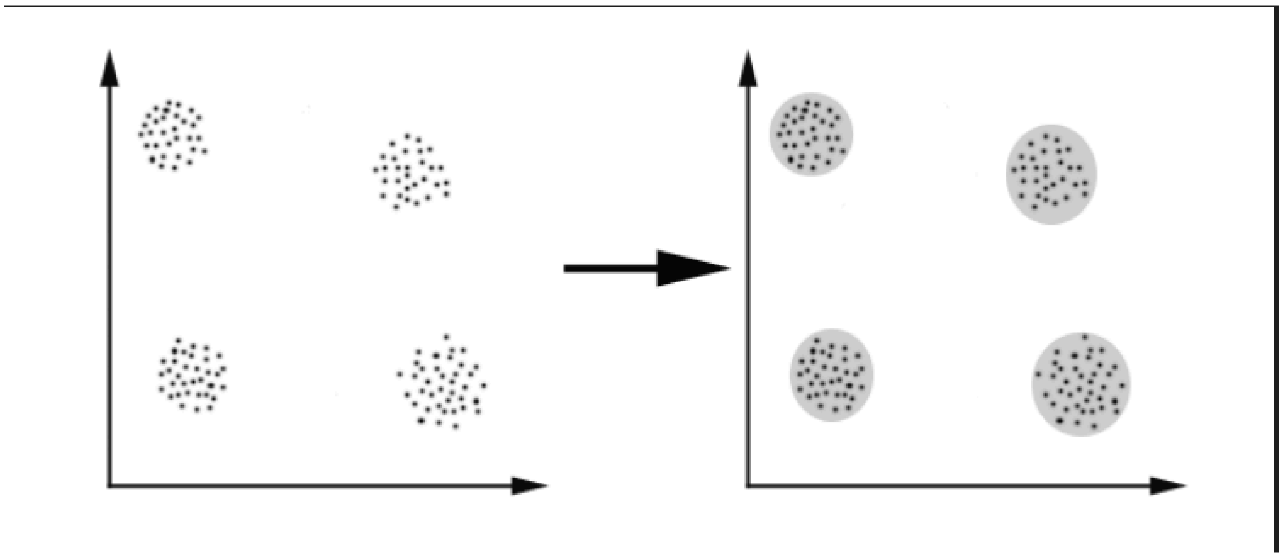
after putting everything together in a single class.

```
knn = KNN(7)
knn.fit(x_train,y_train)
y_pred = knn.predict(x_test)
accuracy_score(y_test,y_pred)
```

we can call every function just like we call for sklearn knn.

Clustering

Clustering can be considered the most important unsupervised learning problem; so, as every other problem of this kind, it deals with finding a structure in a collection of unlabeled data. A loose definition of clustering could be “the process of organizing objects into groups whose members are similar in some way”. A cluster is therefore a collection of objects which are “similar” between them and are “dissimilar” to the objects belonging to other clusters.



K-Means Clustering

The k-means clustering method is an unsupervised machine learning technique used to identify clusters of data objects in a dataset.

The basic idea is to define k centres, one for each cluster.The centroids are placed as much as possible far away from each other. The next step is to take each point belonging to a given data set and associate it to the nearest centroid. When no point is pending, the first step is completed and an early groupage is done. At this point we need to re-calculate k new centroids as barycenters of the clusters resulting from the previous step. After we have these k new centroids, a new binding has to be done between the same data set points and the nearest new centroid. A loop has been generated. As a result of this loop we may notice that the k centroids change their location step by step until no more changes are done.

Finally, this algorithm aims at minimizing an objective function, in this case a squared error function. The objective function

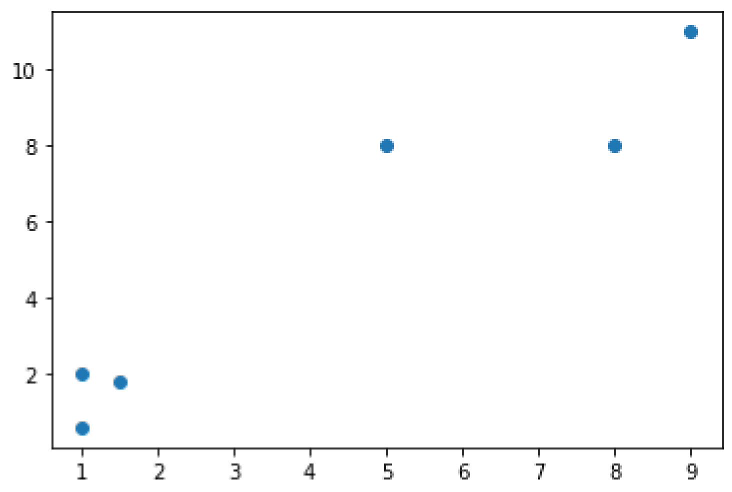
$$\sum_{j=1}^{k}\sum_{i=1}^{k}||x_i^j - c_j||^2$$

where $||\mathbf{x_i^j} - \mathbf{c_j}||$

chosen distance measure between a data point $\mathbf{x_i^j}$ and the cluster centre $\mathbf{c_j}$, is an indicator of the distance of the n data points from their respective cluster centres.

K-Means using Sklearn

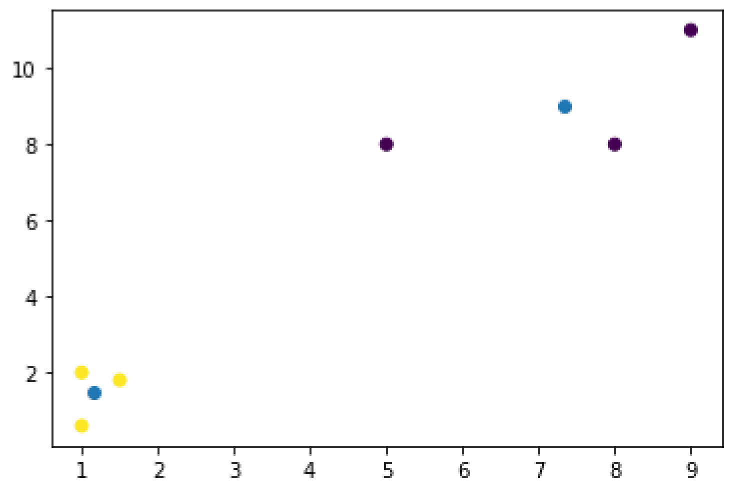
```
import numpy as np
import matplotlib.pyplot as plt
X = np.array([[1,2], [1.5,1.8], [5,8], [8,8], [1,0.6], [9,11]])
plt.scatter(X[:,0], X[:,1])
plt.show()
```



```
from sklearn.cluster import KMeans
k_means = KMeans(n_clusters = 2)
k_means.fit(X)
```

after we train the model.

```
plt.scatter(X[:,0], X[:,1], c=k_means.labels_)
plt.scatter(k_means.cluster_centers_[:,0],k_means.cluster_centers_[:,1])
plt.show()
```



we can clearly see 2 clusters in above image.

Self-Implementation of K-Means

```python
class K_Means:
    def __init__(self, k = 2, max_iter = 100):
        self.k = k
        self.max_iter = max_iter


    def fit (self, data):
        self.means = []
        for i in range(self.k):
            self.means.append(data[i])
        for i in range(self.max_iter):
            clusters = []
            for j in range(self.k):
                clusters.append([])
            for point in data:
                distances = [((point - m)**2).sum() for m in self.means]
                minDistance = min(distances)
                l = distances.index(minDistance)
                clusters[l].append(point)

            change = False
            for j in range(self.k):
                new_mean = np.average(clusters[j], axis=0)
                if not np.array_equal(self.means[j], new_mean):
                    change = True
                self.means[j] = new_mean
            if not change:
                break

    def predict(self, test_data):
        predictions = []
        for point in test_data:
            distances = [((point - m)**2).sum() for m in self.means]
            minDistance = min(distances)
            l = distances.index(minDistance)
            predictions.append(l)
        return predictions
```

we implement our own k–means class by applying our learning.

```
kmeans = K_Means(2, 10)
kmeans.fit(X)
kmeans.predict(X)
```

now we have our own k- means algorithm.

Applications of Clustering

Clustering has a large no. of applications spread across various domains. Some of the most popular applications of clustering are:
1. Recommendation engines
2. Market segmentation
3. Social network analysis
4. Search result grouping
5. Medical imaging
6. Image segmentation
7. Anomaly detection

conclusion

implementing ml algorithms from scratch might lead to better results than using pre built ml models from sk learn we miss out the opportunity to see the real benefit of implementing ml algorithms from scratch.