Machine Learning 1

DATA PREPROCESSING

The libraries used in python for data science in ML

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

whereas in R dont have to import anything

set the Machine learning folder as working Directory to access .csv file

to read data from csv file use var_name = pd.read_csv('File_name')

```
dataset = pd.read_csv('filename')
x = dataset.iloc[: , :-1].values
```

the above line create a array of dataset but we not inluding the last column as we know iloc[row,column]

: means include all or traverse from 0 to last, putting -1 will means it will traverse to last-1 and .values convert it into array if we don't put .values it will return data frame not array

dataset.iloc[,].values — create a array of dataset

[: , :] — it means all the column and rows

[: ,3] — it means all the rows and 3rd column

[:,-1] — it means all the rows and all the column expect last

1-Missing Data -SB

now to take care of missing values we are going to replace the missing values with the mean of the column by using imputer class

```
from sklearn.preprocessing import Imputer
imputer = Imputer(missing_values='NaN',strategy="mean",axis=0)
```

above line is imp and we have to do it because it is a class so we have to create a object first

then

```
imputer = imputer.fit(x[ : ,1:3])
x[ : ,1:3] = imputer.transform(x[ : ,1:3])
```

it will replace missing value ,NaN by the mean of the column of column 2 , 3 as index start at 0 we are not including 1st column and we use 3 as it is not included because 3 is the upper bound

here .fit is used as it will fit the imputer on dataset and compute the mean of column

we can also use imputer.fit_tranform as it will do the same thing in one line

```
x[:,1:3] = imputer.fit_transform(x[:,1:3])
```

2-Categorical variable -SB

```
from sklearn.preprocessing import LabelEncoder
labelencoder_X = LabelEncoder()
x[ : ,0] = labelencoder_X.fit_transform(x[ : , 0])
```

#above line change the categorical data to mathematical data like if a column contain only three value - 'a,b,c' the this will replace a =0 , b=1, c=2 and thus transform the text categorical data into mathematical number above is just a eg of how to do it

but it has a problem that while assigning value it assign a=0 and b=1 and so on but these value are not comparable as we can't say a is greater then b or not but with assign value we can so to do so we have to import another class

```
from sklearn.preprocessing import  OneHotEncoder
onehotencoder = OneHotEncoder(categorical_feature = [0])
x = onehotencoder.fit_transform(x).toarray()
```

u have to do the first thing too

it will create different column for different category as one column for category a and if it is present then 1 else 0 and so on

In categorical_feature should be equal to column u want to expand

now u have to do the same for y as for we can use only labelencoder as in our case it only consist of two categories

3-Splitting the data to into training set and test set

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

here random state = 0 just mean that every time u run the programme these sets will be same and test size is a ratio of total/test and other then that should be remain same

4-Feature Scaling - SB

Scaling the value to same scale like from -1 to 1 this using of section depends on data ur working with as scaling the dummy variable will just improve the accuracy little bit.

as there are some column with very large values compared to others or vise versa this will bring the values of all column to the same scale of -1 to 1.

while using algoritm if we don't do scaling they run for very long time or algo will diverge

```
from sklearn.preprocessing import  StandardScaler
sc_x = StandardScaler()
x_train = sc_x.fit_transform(x_train)
x_test = sc_x.transform(x_test)
sc_y = StandardScaler()
y_train = sc_y.fit_transform(y_train)
```

scaling need not to apply on dependent variable in most case

we need to inverse the scaling to get the actual values

```
y_pred=sc_y.inverse_transform(regressor.predict(x_test))
```

Template

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# to read the file.
dataset = pd.read_csv('Data.csv')
x = dataset.iloc[:,:-1].values
y = dataset.iloc[:,-1].values
```

```
# for removing missing data with mean
"""from sklearn.preprocessing import Imputer
imputer = Imputer(missing_values="NaN", strategy="mean", axis=0)
x[:,1:3] = imputer.fit_transform(x[:,1:3])"""
# for categorical variable
"""from sklearn.preprocessing import LabelEncoder,OneHotEncoder
labelencoder_X = LabelEncoder()
x[ : ,0] = labelencoder_X.fit_transform(x[ : , 0])
onehotencoder = OneHotEncoder(categorical_features = [0])
x = onehotencoder.fit_transform(x).toarray()
labelencoder_Y = LabelEncoder()
y = labelencoder_Y.fit_transform(y)"""
# for splitting the data set
from sklearn.model_selection import train_test_split
x_train , x_test , y_train , y_test = train_test_split(x , y , test_size=0.2 , random_state = 0)
# for feature scaling
"""from sklearn.preprocessing import  StandardScaler
sc_x = StandardScaler()
x_train = sc_x.fit_transform(x_train)
x_test = sc_x.transform(x_test)"""
```

Note - To check the versions of all the dependencies

```
# Python version
import sys
print('Python: {}'.format(sys.version))
# scipy
import scipy
print('scipy: {}'.format(scipy.__version__))
# numpy
import numpy
print('numpy: {}'.format(numpy.__version__))
# matplotlib
import matplotlib
print('matplotlib: {}'.format(matplotlib.__version__))
# pandas
import pandas
print('pandas: {}'.format(pandas.__version__))
# scikit-learn
import sklearn
print('sklearn: {}'.format(sklearn.__version__))
```

Note - To run code in google colab

- Open https://colab.research.google.com, click Sign in in the upper right corner, use your Google credentials to sign in.
- Click File -> Save a copy in Drive... to save your progress in Google Drive
- Click Runtime -> Change runtime type and select GPU in Hardware accelerator box
- Execute the following code in the first cell that downloads dependencies

Note - This will create scatter plot with every combination of rows, good for visualing data at initial step

```
from pandas.plotting import scatter_matrix
scatter_matrix(data)
plt.show()
#print the 20 entries
print(data.head(20))
```

REGRESSION

We have a data type whose y_value are continues

SIMPLE LINEAR REGRESSION

In simple linear Regression we create a linear line (y=ax+c) best suited for the values plotted on x - y graph of the training set value .

Distance is calculated using square root of difference of squares of value of y(real value) and y^(value lies on line corresponding to the same x) and the line which is at min distance from all points is best suited

So in simple Linear Regression coding we first read the csv file split it into dependent variables and independent variable and then into training set then we use a new library scykit learn

1-Fitting Simple Linear Regression To The Training Set

```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(x_train,y_train)
```

now we have train (find the best suited line) the model regressor so to predict the data

2-Predicting test set result

```
y_pred = regressor.predict(x_test)
```

now the y_pred contain the predicted data so if we compare y_pred and y_test we see the result is very similar there is a little deviation but that's ok as it can't predict the exact value

3-Plotting the data

```
#For training set
plt.scatter(X_train,y_train,color = 'red')
plt.plot(x_train,regression.predict(x_train),color = 'blue')
plt.title('Salary cs experince (Training set)')
plt.xlabel('Years of Experince')
plt.ylabel('Salary')
plt.show()


#For test set
plt.scatter(X_test,y_test,color = 'red')
plt.plot(x_test,y_pred,color = 'blue')
```

```
plt.title('Salary cs experince (Testing set)')
plt.xlabel('Years of Experince')
plt.ylabel('Salary')
plt.show()
```

MULTIPLE LINEAR REGRESSION

Multiple linear regression model is same as linear but it has multiple independent variable so the equation seems as —

y=ax1+bx2+cx3 ……. and so on + constant.

In linear regression model we assume some assumption to be true so when working on regression model first check these assumption are true for the dataset

-Linearity, homoscedasticity, Multivarient normality, Independence of error, Lack of multicollinearity

Dummy variable — while dealing with categorical variable let say we have two different categories of data in the column and using onehotencoder we created two column for each then in the model we don't have to use both column as this will create problems let say in the equation ax1+bx2 variable are added because of dummy variable then we know that x1+x2=1 then a and b will become same and nullyfy the effect of both so normally include one variable less then total (general approach other wise it dependent how many column of dummy variable u have to use)

p values — it denotes how much is our result differ from the hypothetical one

Now, there are lots of variable in data set but we don't have use all of them only those which create a accurate model as if there is lots of garbage in model then result will also be garbage so we have to determine which one to use and don't for that there are many approach -:

1- all in

2- Backward elimination

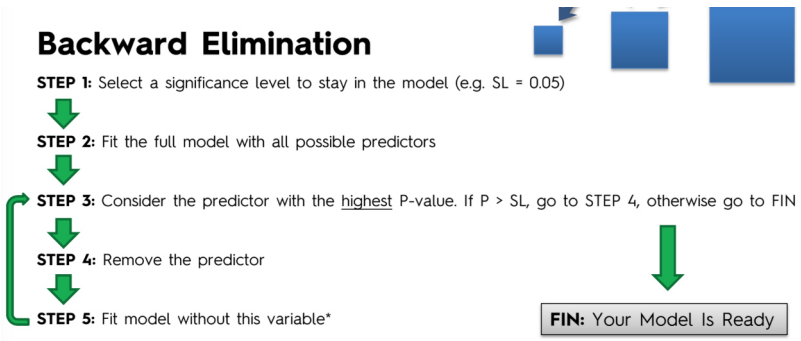
3- Forward selection

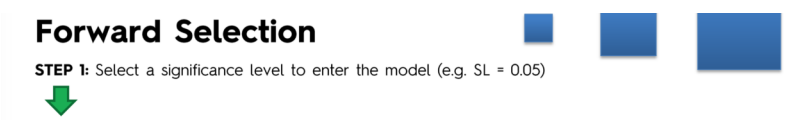
4- Bidirectional elimination — using both 2 and 3 at same time

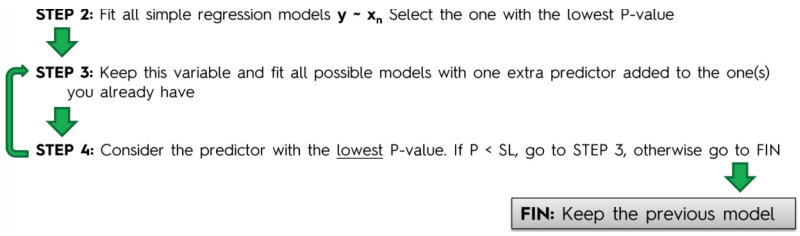
5- score comparison

"All-in" – cases:

- Prior knowledge; OR
- You have to; OR
- Preparing for Backward Elimination







other we will study later

First read the data and create test data and train data using template

1-Avoiding Dummy Variable Trap

when we use encoder to convert categorical values to dummy variable the new column is created in the start of array so if we remove the first column we are good to go -

```
x=x[ : ,1: ]
```

this will remove the first column

2-Linear Regression

we are still using the same model as in simple LR just number of variable are changed so

```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(x_train,y_train)
y_pred = regressor.predict(x_test)
```

as there is no graph because multidimension graph is not possible to draw in 2D space

3-Backward Elimination

now one thing important that we have a constant in the equation which can be written as constant*x0 where x0 is always 1 as stats model don't deals with constant so to do so we have to include a column of 1 at the start of x array

```
x = np.append(arr=np.ones((50,1)).astype(int), values=x, axis=1)
```

""" if we put arr=x and values=bla then column of 1 will be added at the end of x so we interchange it , astype is used as default it will return float but we need int as arr is of int"""

```
import statsmodel.formula.api as sm
x_opt = x[ : , [0, 1, 2, 3, 4, 5]]
regressor_OLS = sm.OLS(endog = y, exog = x_opt).fit()
regressor_OLS.summary()
```

""" the above code will return detail summary which include p values of each column so in tutorial we consider significance level of 0.05 so we remove the column with highest significance level and repeat the above code until only column with pvalue less than 0.05 left thus we obtain a optimised x values """

Automatic Backward Elimination — although no need we can do manually unless x matrix has large number of column. (it is user defined)

```
import statsmodels.formula.api as sm
         def backwardElimination(x, sl):
             numVars = len(x[0])
```

```
        for i in range(0, numVars):
            regressor_OLS = sm.OLS(y, x).fit()
            maxVar = max(regressor_OLS.pvalues).astype(float)
            if maxVar > sl:
                for j in range(0, numVars - i):
                    if (regressor_OLS.pvalues[j].astype(float) == maxVar):
                        x = np.delete(x, j, 1)
        regressor_OLS.summary()
        return x

    SL = 0.05
    X_opt = X[:, [0, 1, 2, 3, 4, 5]]
    X_Modeled = backwardElimination(X_opt, SL)
```

POLYNOMIAL LINEAR REGRESSION

in polynomial linear regression y=b+a1*x^1+a2*x^2+a3*x^3 ……. so on

here line formed is not straight but a polynomial but it still called linear because constant are linear. linear keyword is denoting the linearity of constant

for this regression we first create multiple linear regression the fit it to polynomial

NOTE - x variable must be 2D array so if x contain one column the use .iloc[: ,0:1] as in 0:1 1 column is not included because it the upper bound but 0 column is and it will form 2D array if we include only 0 then a 1D array will be created which we don't want

NOTE - if we don't have enough data then we don't split data into training and test because if we do model don't have enough data to get train.

Step -1

for polynomial regression first we create a polynomial dataset - x_poly then fit it into multiple regression which create a polynomial regression

```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import PolynomialFeatures
poly_reg = PolynomialRegression(degree = 2)
# degree is only parameter imp for now and it denote degree of polynomial regression
x_poly = poly_reg.fit_transform(x)
lin_reg = LinearRegression()
lin_reg.fit(x_poly,y)
y_pred = lin_reg.predict(x_test)
```

it will create a new dataframe x_poly which has no of column equal to degree of polynomial +1 where column 0 = 1(x^0) , column 1 = x^1 ,column 2 = x^2 , and so on then we create a multiple regression and pass the x_poly instead of x

To improve accuracy of our model we just need to adjust the degree of polynomial

increasing the degree generally improve accuracy

STEP-2 plotting the data

if we plot the data as it is. It will create a step graph let say we have 10 values in x then graph will look like the 10 dot join using the line so to this happen when we have less data so to make it look like the real poly_reg we slip the data into more #just for bakwaas no real use for project

plotting in higher resolution

```
x_grid = np.range(min(x),max(x),0.1)         # 0.1 will determine the step size
x_grid = x_grid.reshape((len(x_grid),1)      # convert it from array to vector
ply.plot(x_grid,lin_reg.predict(poly_reg.fit_transform(x_grid)))
plt.show()
```

Template

```
# Regression Template

# Importing the libraries

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

# Importing the dataset

dataset = pd.read_csv('Position_Salaries.csv')

X = dataset.iloc[:, 1:2].values

y = dataset.iloc[:, 2].values

# Splitting the dataset into the Training set and Test set

"""from sklearn.cross_validation import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)"""

# Feature Scaling

"""from sklearn.preprocessing import StandardScaler

sc_X = StandardScaler()

X_train = sc_X.fit_transform(X_train)

X_test = sc_X.transform(X_test)

sc_y = StandardScaler()

y_train = sc_y.fit_transform(y_train)"""

# Fitting the Regression Model to the dataset

# Create your regressor here

# Predicting a new result

y_pred = regressor.predict(6.5)

# Visualising the Regression results

plt.scatter(X, y, color = 'red')

plt.plot(X, regressor.predict(X), color = 'blue')

plt.title('Truth or Bluff (Regression Model)')

plt.xlabel('Position level')

plt.ylabel('Salary')

plt.show()

# Visualising the Regression results (for higher resolution and smoother curve)

X_grid = np.arange(min(X), max(X), 0.1)

X_grid = X_grid.reshape((len(X_grid), 1))

plt.scatter(X, y, color = 'red')

plt.plot(X_grid, regressor.predict(X_grid), color = 'blue')
```

```
plt.title('Truth or Bluff (Regression Model)')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()
```

Note - Confusion matrix help in calculating accuracy of model

	Actual Positive	Actual Negative
Predicted Positive	True Positive(TP)	False Positive(FP) (Type 1 Error)
Predicted Negative	False Negative(FN) (Type 2 Error)	True Negative(TN)

$$\text{Accuracy} = \frac{\text{True Positive + True Negative}}{\text{Total Population}}$$

$$\text{Error Rate/Misclassification rate} = \frac{\text{False Positive + False Negative}}{\text{Total Population}}$$

$$\text{Precision} = \frac{\text{True Positive}}{\text{Predicted Positive(TP+FP)}}$$

$$\text{Sensitivity/Recall} = \frac{\text{True Positive}}{\text{Actual Positive(TP+FN)}}$$

$$\text{Specificity} = \frac{\text{True Negative}}{\text{Actual Negative(FP+TN)}}$$

$$\text{F1 Score} = \frac{2 * (\text{Recall} * \text{Precision})}{\text{Recall + Precision}}$$

```
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
accuracy = ((cm[0][0]+cm[1][1])/(cm[0][0]+cm[0][1]+cm[1][0]+cm[1][1]))*100
print(accuracy)
```

SUPPORT VECTOR REGRESSION (SVM)

THEORY

The terms that we are going to be using frequently

1-Kernel: The function used to map a lower dimensional data into a higher dimensional data.

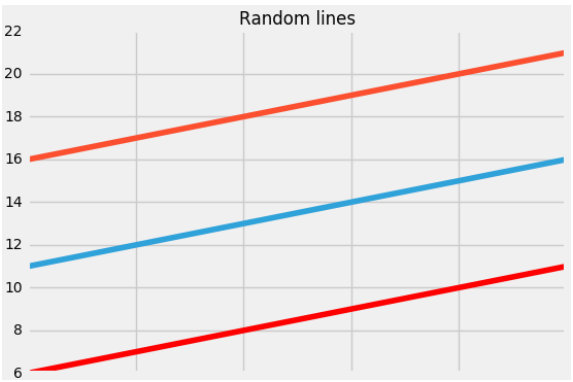
2-Hyper Plane: In SVM this is basically the separation line between the data classes. Although in SVR we are going to define it as the line that will will help us predict the continuous value or target value

3-Boundary line: In SVM there are two lines other than Hyper Plane which creates a margin . The support vectors can be on the Boundary lines or outside it. This boundary line separates the two classes. In SVR the concept is same.

4-Support vectors: This are the data points which are closest to the boundary. The distance of the points is minimum or least.

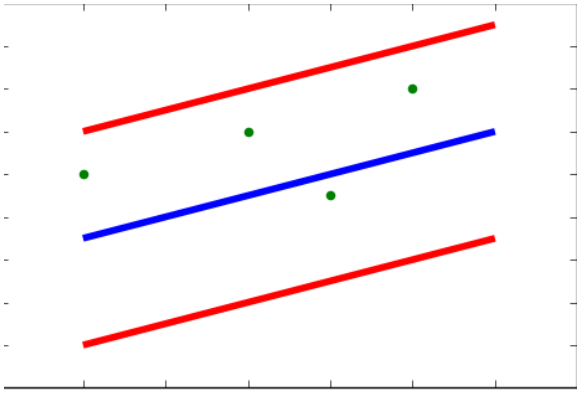
Why SVR ? Whats the main difference between SVR and a simple regression model?

In simple regression we try to minimise the error rate. While in SVR we try to fit the error within a certain threshold. This might be a bit confusing but let me explain.



```
1 2 3 4 5 6
```

Blue line: Hyper Plane; Red Line: Boundary Line



See fig 2 see how all the points are within the boundary line(Red Line). Our objective when we are moving on with SVR is to basically consider the points that are within the boundary line. Our best fit line is the line hyperplane that has maximum number of points.

So Let's Start:

So the first thing we have to understand is *what is this boundary line ?(yes! that red line).* Think of it as to lines which are at a distance of 'e' (though not e its basically epsilon) but for simplicity lets say its 'e'.

So the lines that we draw are at '+e' and '-e ' distance from Hyper Plane.

Assuming our hyper plane is a straight line going through the Y axis

We can say that the Equation of the hyper plane is

wx+b=0

So we can state that the two the equation of the boundary lines are

Wx+b=+e

Wx+b=-e

respectively

Thus coming in terms with the fact that for any linear hyper plane the equation that satisfy our SVR is:

e≤y-Wx-b≤+e

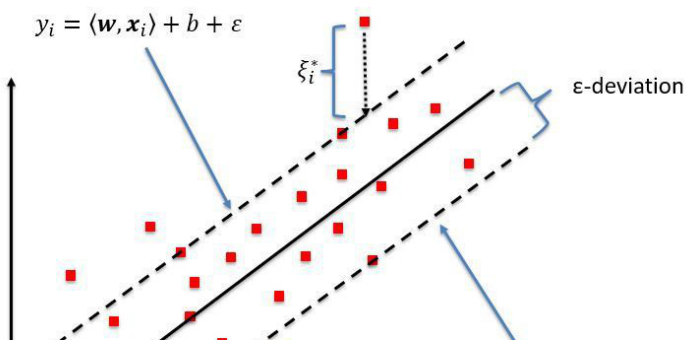
stating the fact that y=Wx+b

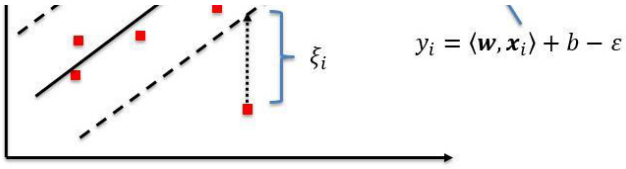
y-Wx-b=0

This applies for all other type of regression (non-linear, polynomial)

RECAP

What we are trying to do here is basically trying to decide a decision boundary at 'e' distance from the original hyper plane such that data points closest to the hyper plane or the support vectors are within that boundary line





Thus the decision boundary is our Margin of tolerance that is We are going to take only those points who are within this boundary.

Or in simple terms that we are going to take only those those points which have least error rate. Thus giving us a better fitting model.

Building a SVR

1. Collect a training set $\tau = \{\vec{X}, \vec{Y}\}$
2. Choose a kernel and it's parameters as well as any regularization needed.
3. Form the correlation matrix, $\vec{K}$
4. Train your machine, exactly or approximately, to get contraction coefficients $\vec{\alpha} = \{\alpha_i\}$
5. Use those coefficients, create your estimator $f(\vec{X}, \vec{\alpha}, x^\star) = y^\star$

first do the usual like importing data set and all

in SVR there are two argument that are important kernel and degree

kernel option — linear, poly, rbf, sigmoid, precomputed, callable

for now we are going to use rbf kernel and not gonna use degree

WE NEED FEATURE SCALING ITS IMPORTANT FOR THIS REGRESSOR

```
sc_x = StandardScaler()
sc_y = StandardScaler()
x = sc_x.fit_transform(x)
y = sc_y.fit_transform(y)
#----------
from sklearn.svm import SVR
regressor = SVR(kernal='rbf')
regressor.fit(x, y)
y_pred=sc_y.inverse_transform(regressor.predict(sc_x.transform(np.array([[6.5]]))))
```

DECISION TREE REGRESSION

In this regression model the data set is slip into different leaves or section based on similarities in results then average is taken of each leaves and when we want to predict the value it just look for on which leave the x value lie and return the avg of y on that leave

```
from sklearn.tree import DecisionTreeRegressor
regressor = DecisionTreeRegressor(random_state = 0)
regressor.fit(x,y)
y_pred = regressor.predict(x_test)
```

RANDOM FOREST REGRESSION

In random forest regression we take k data point from our data set and use decision tree regression on the k dataset and we repeat this couple 100 of times so we have like couple 100 decision tree regression model so while predicting value we take average of all

the pred result from different trees and this give us a very accurate result because it is possible on tree has high error but averaging 1000 of then will not.

```
from sklearn.ensemble import RandomForestRegressor
regressor = RandomForestRegressor(n_estimators = 100, random_state = 0)
regressor.fit(x_train,y_train)
y_pred = regressor.predict(x_test)
```

only n_estimator argument is imp as it will determine the number of tree used in the forest

EVALUATING REGRESSION MODEL PERFORMANCE

It is used for evaluating which column to remove or keep in the dataset mainly used in multiple regression it help us see the effect of each column on the result

SSrev = SUM(Yi - Y^i)^2

SStot = SUM(Yi - Yavg)^2

R^2 = 1 - SSrev/SStot

here R is called squared Intuition and its value should be maximum for better result for better result we use Adj R^2

Adj R^2 = 1 - (1 - R^2)(n-1)/(n-p-1)

p -> no of regression

n -> Sample size

so according to above formula if we increase the no of regression p will increase which decreases the Adj R^2 but R^2 also decrease so if we are adding a new column its effect on the model should be significant for accuracy to get better other wise it will decrease the accuracy

Using the same code of backward elimination which give the statistic summary of the dataset it also give r^2 and adj r^2 in the summary so while we are eliminating columns on the basis of p-value we have to look for adj r^2 that it does not decrease if it does then removing that column is not good idea

```
Call:
lm(formula = Profit ~ R.D.Spend + Administration + Marketing.Spend,
    data = dataset)

Residuals:
   Min     1Q Median     3Q    Max
-33534  -4795     63   6606  17275

Coefficients:
                 Estimate Std. Error t value Pr(>|t|)
(Intercept)      5.012e+04  6.572e+03   7.626 1.06e-09 ***
R.D.Spend        8.057e-01  4.515e-02  17.846  < 2e-16 ***
Administration  -2.682e-02  5.103e-02  -0.526    0.602
Marketing.Spend  2.723e-02  1.645e-02   1.655    0.105
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 9232 on 46 degrees of freedom
Multiple R-squared:  0.9507,    Adjusted R-squared:  0.9475
F-statistic:   296 on 3 and 46 DF,  p-value: < 2.2e-16
```

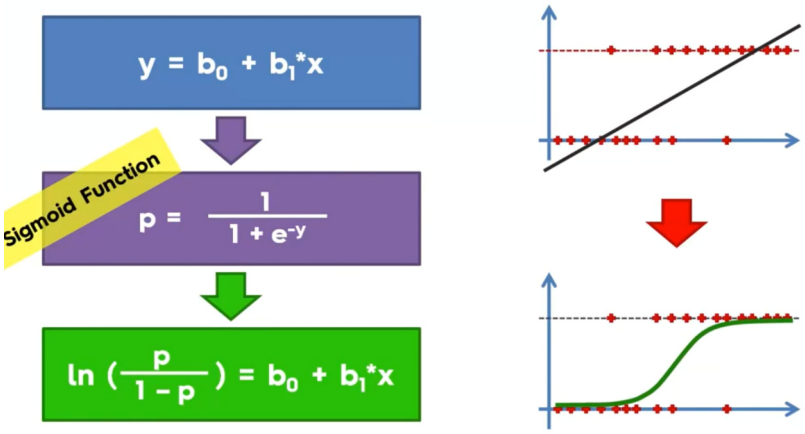
demo image

CLASSIFIERS

In classifier we have data set whose y_value is categorical type or like

LOGISTIC REGRESSION INTUITION

This type of regression is mainly used when u have discreet result like yes/no because in those type of situation above model will fail as they work on non-discreet results and instead of predicting what is the result it predict the probability of each discreet value eg it predict the chances of yes for certain x values if it has yes/no type y value



so basically instead of fitting linear line we fit the logistic line (see in fig graph 2) and it predict the probably of occurrence.

use feature scaling on x value and preprocess the data first

```
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression(random_state = 0)
classifier.fit(x_train,y_train)
y_pred = classifier.predict(x_test)
```

Now to check model performance we are going to use confusion matrix which tell how many prediction are correct

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test,y_pred)
```

in console type cm it return a 2x2 square matrix sum of (1,1) and (2,2) gives the total no of correct pred and sum of other two give no of incorrect prediction

for plotting the graph (it is different from above used)

```
from matplotlib.colors import ListedColormap
x_set, y_set = x_train, y_train
x1, x2 = np.meshgrid(np.arange(start = x_set[ : , 0].min -1, stop = x_set[ : ,0].max() +1 , step
= 0.01),
         np.arange(start = x_set[ : , 1].min -1, stop = x_set[ : ,1].max() +1 , step = 0.01))
plt.contourf(x1, x2, classifier.predict(np.arange([x1.ravel(), x2.ravel()]).T).reshape(x1.shape),
               alpha = 0.75, cmap = ListedColormap((‘red’, ‘green’)))
plt.xlim(x1.min(), x1.max())
plt.ylim(x2.min(), x2.max())


for i, j in enumerate(np.unique(y_set)):
     plt.scatter(x_set[y_set == j,0], x_set[y_set == j, 1],
                 c = ListedColormap((‘red’, ‘green’ ))(i), label =j)
plt.title(‘Logistic Regrssion (Training set)’)
plt.xlabel(‘Age’)
plt.ylabel(‘Estimated Salary’)
```

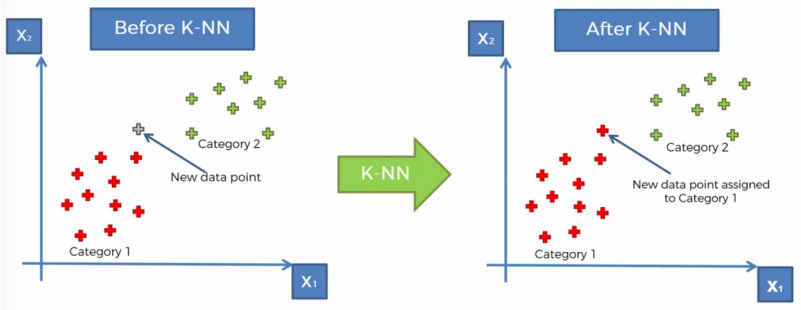
```
plt.legend()
plt.show()
```

Template

```
# Classification template
# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
# Importing the dataset
dataset = pd.read_csv('Social_Network_Ads.csv')
X = dataset.iloc[:, [2, 3]].values
y = dataset.iloc[:, 4].values
# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, random_state = 0)
# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X_train = sc.fit_transform(X_train)
X_test = sc.transform(X_test)
# Fitting classifier to the Training set
# Create your classifier here
# Predicting the Test set results
y_pred = classifier.predict(X_test)
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
# Visualising the Training set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_train, y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
```

```
plt.title('Classifier (Training set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
# Visualising the Test set results
from matplotlib.colors import ListedColormap
X_set, y_set = X_test, y_test
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 1, stop = X_set[:, 0].max() + 1, step = 0.01),
                     np.arange(start = X_set[:, 1].min() - 1, stop = X_set[:, 1].max() + 1, step = 0.01))
plt.contourf(X1, X2, classifier.predict(np.array([X1.ravel(), X2.ravel()]).T).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1],
                c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('Classifier (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

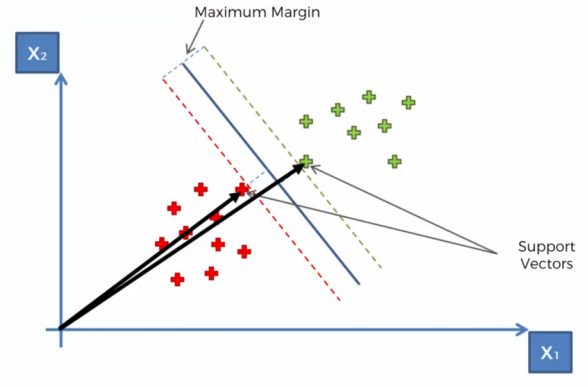
K-NEAREST NEIGHBOR INTUITION



In this algo it create group just like regression tree but draw back is if we enter a data point which is not in the range of any group then it give inaccurate prediction so in this algo if some data point is enter which don't belong to any group then it use Euclidean distance to find k nearest neighbour and the group with max neighbour will include the datapoint.

```
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors = 5)
classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)
```

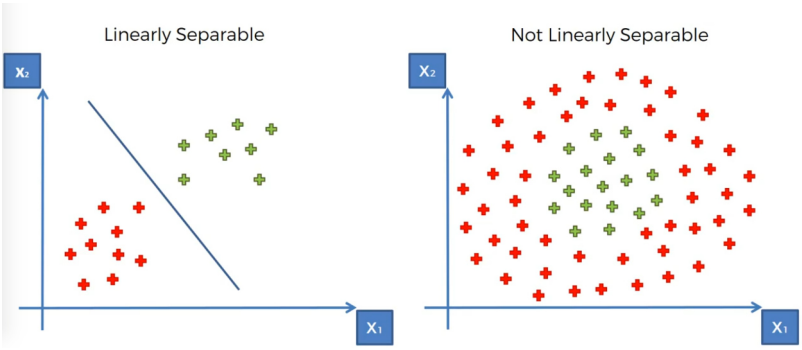
SVM INTUITION



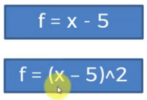

```
from sklearn.preprocessing import SVC
classifier = SVC(kernel = 'linear', random_state = 0)
classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)
```

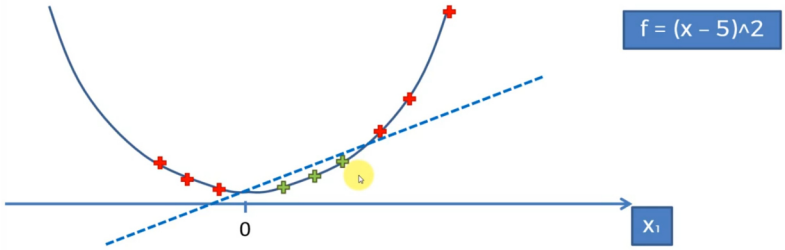
KERNEL SVM INTUITION

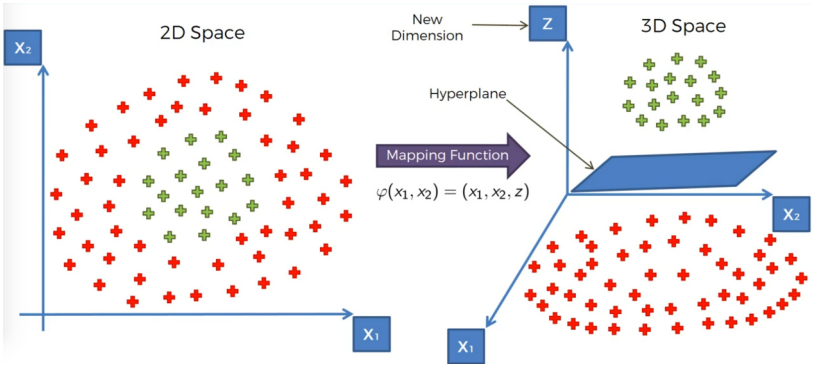
In SVM intuition the data is separable by a line or curve means there exist a line like in above three our classifier is creating a line which separate the data set what if your data is not separable by a line or curve like in fig below



in this case we use kernel svm intuition which create a line or hyperplane by increasing the dimension our data

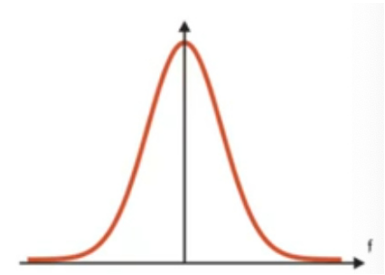




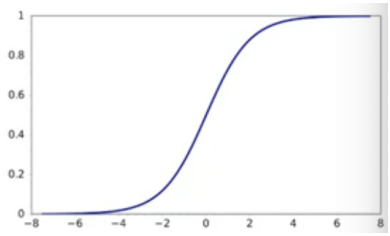


types of Kernal

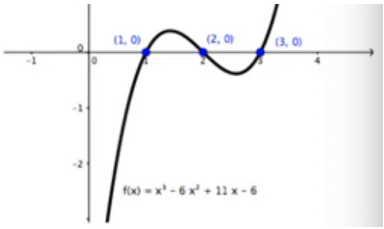
GAUSSIAN RBF KERNAL



SIGMOID KERNEL



POLYNOMIAL KERNEL




```
from sklearn.svm import SVC
classifier = SVC(kernel = 'rbf', random_state = 0)
classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)
```

NAIVE BAYES

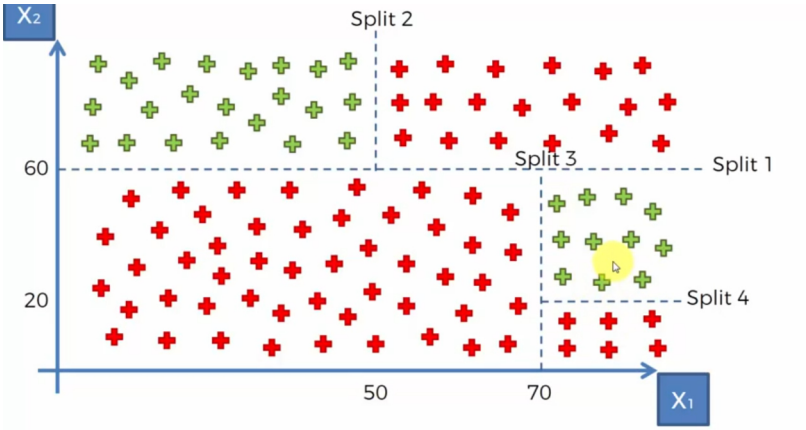
It is probablic classifier. We use bayes theorm from probab

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

```
from sklearn.naive_bayes import GaussianNB
classifier = GaussianNB()
classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)
```

DECISION TREE CLASSIFICATION

Same as concept as above decision tree but different type. It split the data into different leaves of same type




```
from sklearn.tree import DecisionTreeClassifier
classifier = DecisionTreeClassifier(criterion='entropy', random_state=0)
classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)
```

RANDOM FOREST CLASSIFICATION

same theory as random forest regression - use lots of decision tree model to predict

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators=10, criterion='entropy', random_state=0)
classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)
```

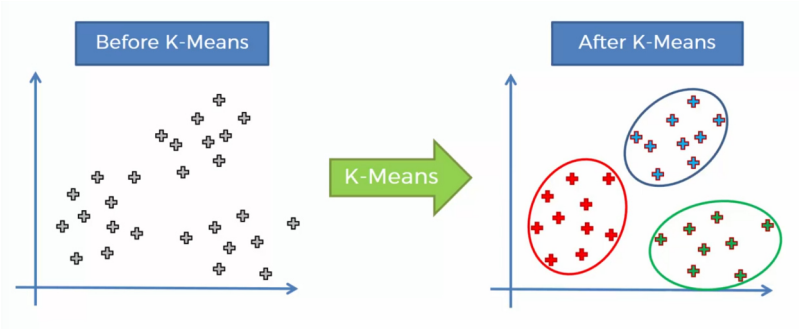
CLUSTERING

Clustering is similar to the classification, but the basic is different. In clustering you don't know what you are looking for, and you are trying to identify some segments or clusters in your data.

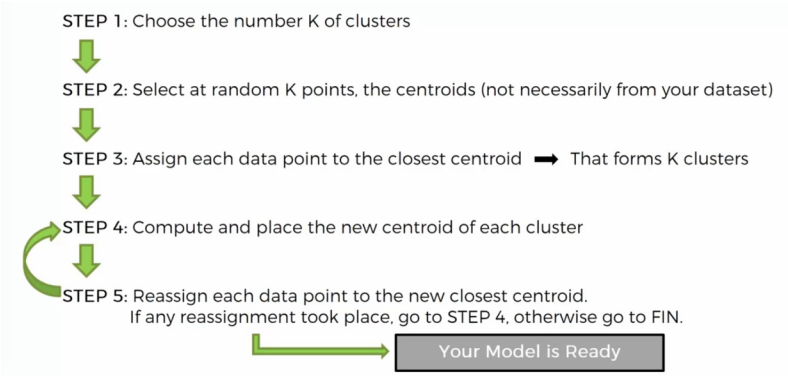
Means you don't have the yvalue u just the data and u have to classify it eg - u have the data of customer of mall and u have to determine who are the targeted audience should be so that mall can determine what they are buying and organise and purchase things in order to increase their sale

K-MEANS CLUSTERING

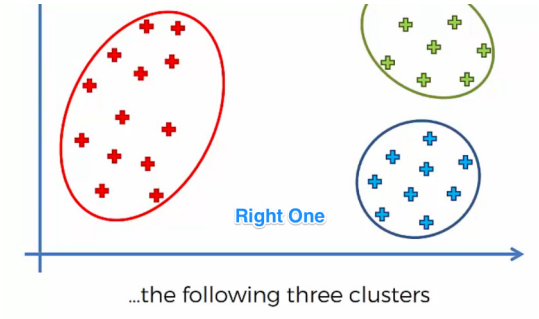
It takes out complexity from the dataset and allows us to identify the data as cluster

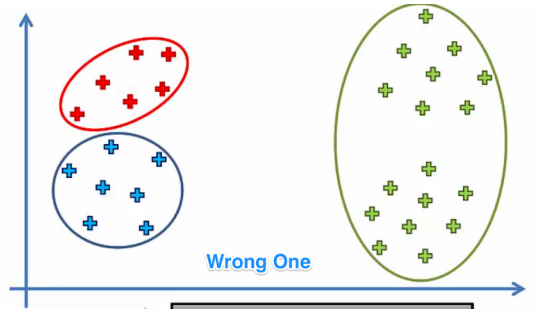


Step for algo



K-means trap when u select your centroid wrong





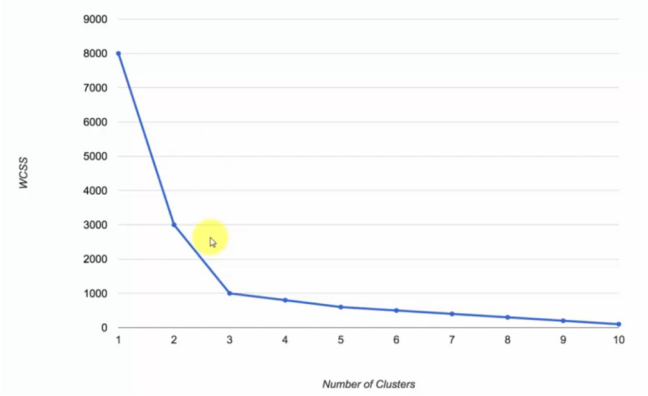
so how we solve the problem we use modify the formula to Kmeans++ which is actually used

in this algo so no need to worry about this one

But the real problem is how we determine the number of cluster we need for our dataset

$$\text{WCSS} = \sum_{P_i \text{ in Cluster 1}} \text{distance}(P_i, C_1)^2 + \sum_{P_i \text{ in Cluster 2}} \text{distance}(P_i, C_2)^2 + \sum_{P_i \text{ in Cluster 3}} \text{distance}(P_i, C_3)^2$$

we use the above formula where we compute the sum of euclidean distance of each point from its centroid and lower the value of WCSS better the result but it has a trap too. If no of cluster is equal to no of centroid then the within distance will be zero and that should be best right so on increasing no of centroid WCSS will decrease no matter what so how we determine it. We plot a curve for every value of K(no of cluster) and we will get a curve like this



point where the relative decrease become low that the optimal no of cluster in above image its 3

so we apply algo for every k value and plot a graph and determine it and this method is called elbow method as curve look like elbow

```
from sklearn.cluster import KMeans
wcss=[]
for i in range(1,11):
kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10, random_state=0)
kmeans.fit(x)
wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.show()
```

parameter of KMeans —>

n_cluster - no of cluster used

init - method used for determining the position of centroid as we have discussed above we will use k-means++

max_iter - max iteration used for correctly positioning the centroid default is 300 we will stick to it but in case u change it don't change it too much as after certain point iternation start yeilding same value again and again and that generally come at 300 and after that point u are just making your model slow

n_init - no of time the algo will run with different position of centroid and take avg of then default is 10 and we will use that only

random_state - usual

kmeans.inertia_ = it calculate wcss value for model and we will save it in the array name wcss[]

we plot the graph and determine the optimum value of k and again make the kmeans object with that optimum value of cluster

then we apply the kmeans algo actually

```
kmeans = KMeans(n_clusters=5, init='k-means++', max_iter=300, n_init=10, random_state=0)
y_kmeans = kmeans.fit_predict(x)
# above line will make an array telling to which cluster each client belong
Now plot the Graph
plt.scatter(x[y_kmeans == 0, 0], x[y_kmeans == 0, 1], s=100, c='red', label='Cluster 1')
plt.scatter(x[y_kmeans == 1, 0], x[y_kmeans == 1, 1], s=100, c='blue, label='Cluster 2')
plt.scatter(x[y_kmeans == 2, 0], x[y_kmeans == 2, 1], s=100, c='green', label='Cluster 3')
plt.scatter(x[y_kmeans == 3, 0], x[y_kmeans == 3, 1], s=100, c='cyan', label='Cluster 4')
plt.scatter(x[y_kmeans == 4, 0], x[y_kmeans == 4, 1], s=100, c='magenta', label='Cluster 5')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=200, c='yellow',
label='centr')
plt.legend()
plt.show()
# we add legend because there are lots of scatter plot and we want to combine them
```

The whole code look something like this

---

from sklearn.cluster import KMeans

wcss=[]

for i in range(1,11):

kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10, random_state=0)

kmeans.fit(x)

wcss.append(kmeans.inertia_)

plt.plot(range(1, 11), wcss)

plt.show()

kmeans = KMeans(n_clusters=5, init='k-means++', max_iter=300, n_init=10, random_state=0)

y_kmeans = kmeans.fit_predict(x)

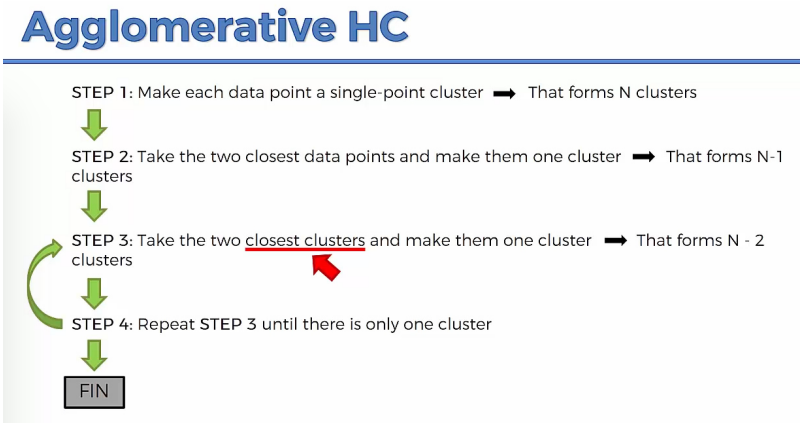
```
plt.scatter(x[y_kmeans == 0, 0], x[y_kmeans == 0, 1], s=100, c='red', label='Cluster 1')
plt.scatter(x[y_kmeans == 1, 0], x[y_kmeans == 1, 1], s=100, c='blue, label='Cluster 2')
plt.scatter(x[y_kmeans == 2, 0], x[y_kmeans == 2, 1], s=100, c='green', label='Cluster 3')
plt.scatter(x[y_kmeans == 3, 0], x[y_kmeans == 3, 1], s=100, c='cyan', label='Cluster 4')
plt.scatter(x[y_kmeans == 4, 0], x[y_kmeans == 4, 1], s=100, c='magenta', label='Cluster 5')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=200, c='yellow', label='centr')
plt.legend()
plt.show()
```

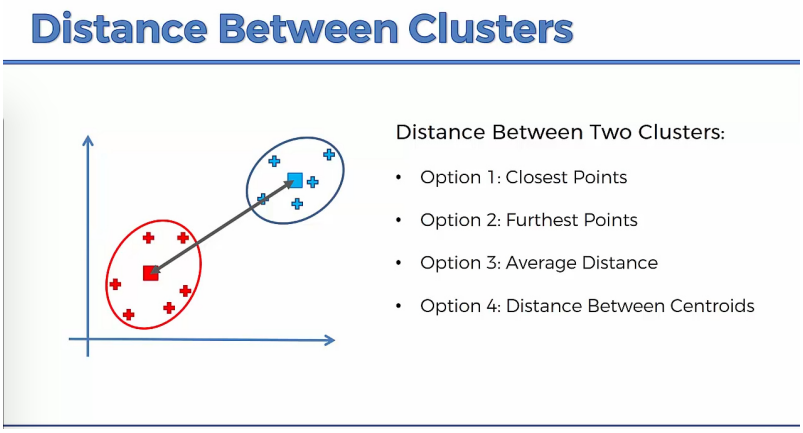
HIERARCHICAL CLUSTERING

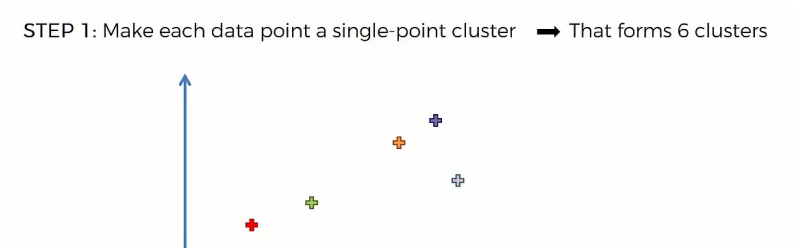
Same result will be yielded with different accuracy depend on the data set. Approach is diff

there are two type of hierarchical clustering - Agglomerative and Divisive

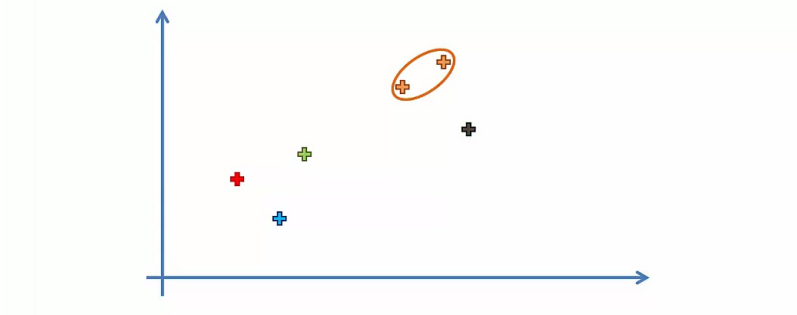
we going to use agglomerative as both will give same result



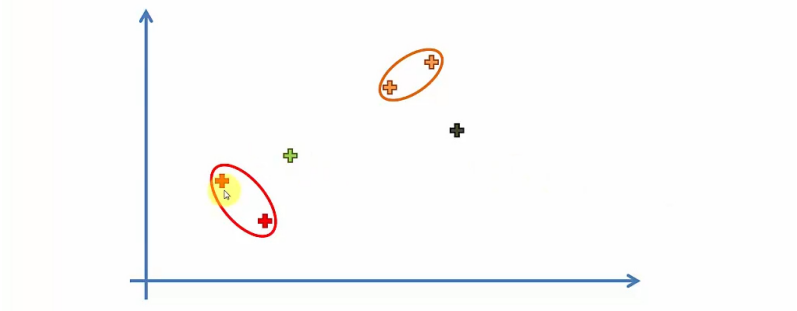




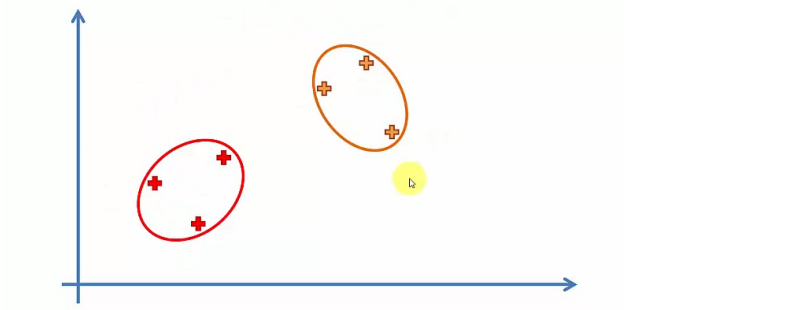
STEP 2: Take the two closest data points and make them one cluster
➡ That forms 5 clusters



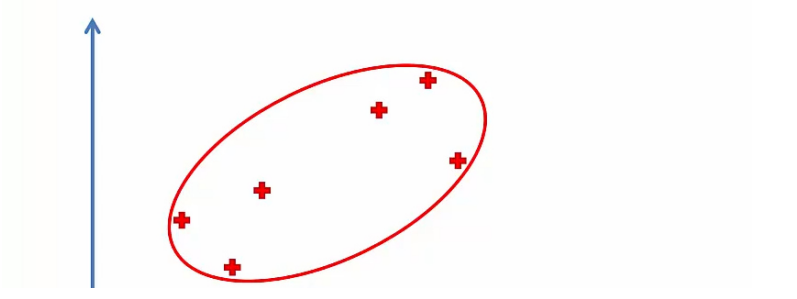
STEP 3: Take the two closest clusters and make them one cluster
➡ That forms 4 clusters



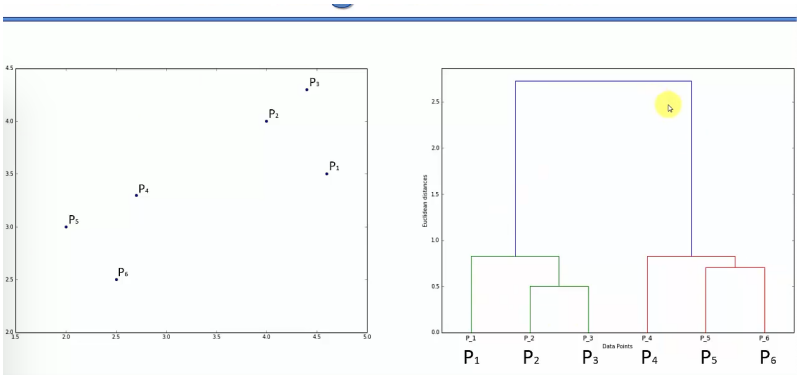
STEP 4: Repeat STEP 3 until there is only one cluster



STEP 4: Repeat STEP 3 until there is only one cluster

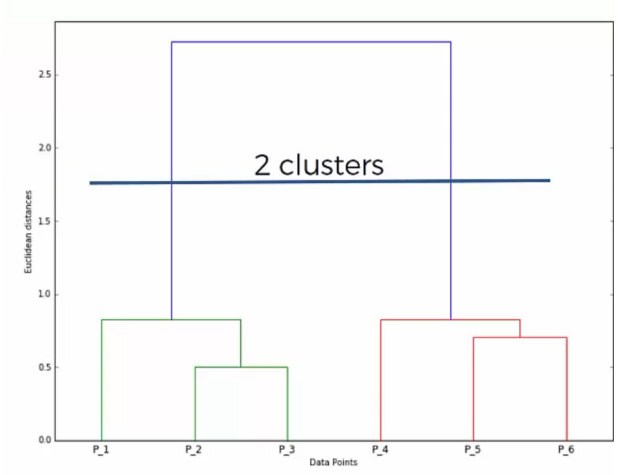


It first consider each point as seperate cluster then it conbine the two nearest cluster into one (distance b/t two cluster is calculated as explaned in second fig) and keep on combining then untill one cluster left and it saves all the info each step into Dendograms



The left graph in above fig is actually a dendogram where height of line signifies the distance b/w cluster

that way we determine a threshold and if the distance b/w the cluster is larger than that then we stop at that point like in below fig black line is the threshold



so it give 2 cluster. No of line cross threshold give no of cluster

Now the code

```
import scipy.cluster.hierarchy as sch
dendrogram = sch.dendrogram(sch.linkage(x, method='ward'))
plt.show
# this will show the dendrogram now we can determine the no of cluster from graph
from sklearn.cluster import AgglomerativeClustering
hc = AgglomerativeClustering(n_clusters=5, affinity='euclidean', linkage='ward')
y_hc = hc.fit_predict(x)
plt.scatter(x[y_kmeans == 0, 0], x[y_kmeans == 0, 1], s=100, c='red', label='Cluster 1')
plt.scatter(x[y_kmeans == 1, 0], x[y_kmeans == 1, 1], s=100, c='blue, label='Cluster 2')
plt.scatter(x[y_kmeans == 2, 0], x[y_kmeans == 2, 1], s=100, c='green', label='Cluster 3')
plt.scatter(x[y_kmeans == 3, 0], x[y_kmeans == 3, 1], s=100, c='cyan', label='Cluster 4')
plt.scatter(x[y_kmeans == 4, 0], x[y_kmeans == 4, 1], s=100, c='magenta', label='Cluster 5')
```

```
plt.legend()
plt.show()
```

ASSOCIATION RULE LEARNING

APRIORI

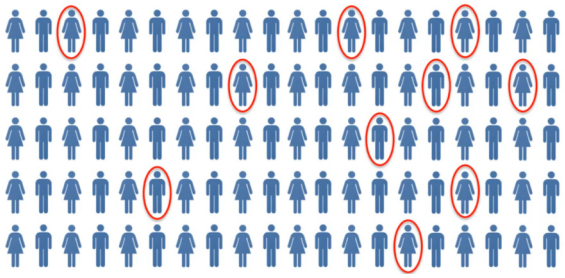
This ML intuition used for analysing condition where, when one thing happen other thing happen too. Eg - when customer buy bread they also buy butter, also like netflix recommendation system

apriori algo has three part to it - support, confidence and the lift

Movie Recommendation: $\text{support}(M) = \frac{\#\text{ user watchlists containing } M}{\#\text{ user watchlists}}$

Market Basket Optimisation: $\text{support}(I) = \frac{\#\text{ transactions containing } I}{\#\text{ transactions}}$

Apriori - Support

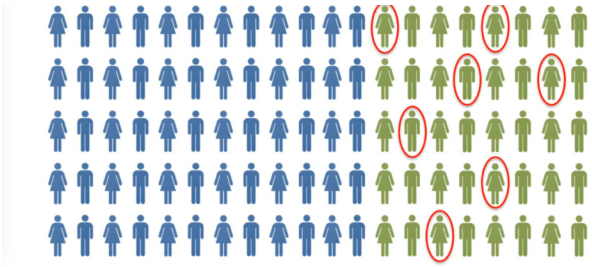


Apriori - Confidence

Movie Recommendation: $\text{confidence}(M_1 \rightarrow M_2) = \frac{\#\text{ user watchlists containing } M_1 \text{ and } M_2}{\#\text{ user watchlists containing } M_1}$

Market Basket Optimisation: $\text{confidence}(I_1 \rightarrow I_2) = \frac{\#\text{ transactions containing } I_1 \text{ and } I_2}{\#\text{ transactions containing } I_1}$

Apriori - Confidence

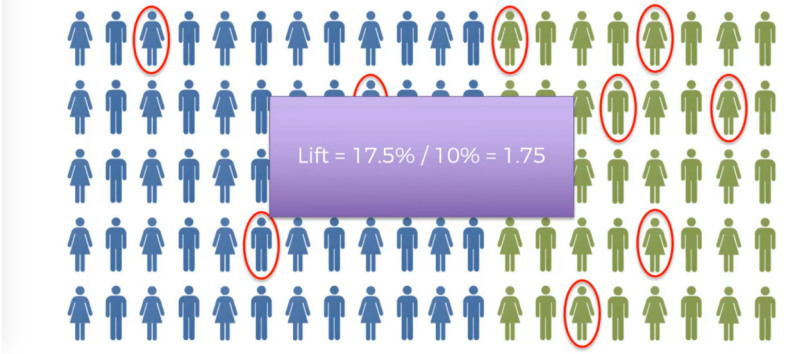


Apriori - Lift

Movie Recommendation: $\text{lift}(M_1 \rightarrow M_2) = \frac{\text{confidence}(M_1 \rightarrow M_2)}{\text{support}(M_2)}$

Market Basket Optimisation: $\text{lift}(I_1 \rightarrow I_2) = \frac{\text{confidence}(I_1 \rightarrow I_2)}{\text{support}(I_2)}$

Apriori - Lift



Apriori - Algorithm

Step 1: Set a minimum support and confidence

Step 2: Take all the subsets in transactions having higher support than minimum support

Step 3: Take all the rules of these subsets having higher confidence than minimum confidence

Step 4: Sort the rules by decreasing lift

apyori.py 15 kB

In Apriori algo we don't have the apyori library so we are going to use the above file which contain apriori algo and import it

After importing data we need convert the data from dataframe to list of list

because apriori function take list as argument

```
transaction = []
for i in range(0,7501):
transaction.append([str(dataset.values[i, j]) for j in range(0, 20)])
from apyori import apriori
rules = apriori(transaction, min_support=0.003, min_confidence=0.2, min_lift=3, max_lenght=2)
# min_support,min_confidence,min_lift these parameter depend the dataset on rows affect them
# max_lenght is max no of element u want in association then there is min_lenght also
# above data is entered on account of 7500 entries
results = list(rules)
# above line save all the rules or output into result variable which is a list
```

REINFORCEMENT LEARNING

Reinforcement Learning is a branch of Machine Learning, also called Online Learning. It is used to solve interacting problems where the data observed up to time t is considered to decide which action to take at time t + 1. It is also used for Artificial Intelligence when training machines to perform tasks such as walking. Desired outcomes provide the AI with reward, undesired with punishment. Machines learn through trial and error.

UPPER CONFIDENCE BOUND(UCB)

Step 1. At each round n, we consider two numbers for each ad i:

- $N_i(n)$ - the number of times the ad i was selected up to round n,
- $R_i(n)$ - the sum of rewards of the ad i up to round n.

Step 2. From these two numbers we compute:

- the average reward of ad i up to round n

$$\bar{r}_i(n) = \frac{R_i(n)}{N_i(n)}$$

- the confidence interval $[\bar{r}_i(n) - \Delta_i(n), \bar{r}_i(n) + \Delta_i(n)]$ at round n with

$$\Delta_i(n) = \sqrt{\frac{3}{2}\frac{\log(n)}{N_i(n)}}$$

Step 3. We select the ad i that has the maximum UCB $\bar{r}_i(n) + \Delta_i(n)$.

this type of learning is used when u have to make a ML model on the go like a company has 10 different type of ads and they want to know which one will give better result but they don't have any data. So u created a ML model which reward itself whenever user click on the ads and from that determine which ad is better.

lets discuss it further

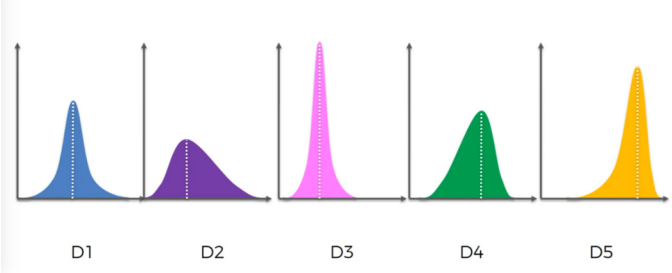
The multi-Armed Bandit problem

Let say u have five slot machine

u have 100 coins and u want to maximise ur profit so u have to analysis which one will give max profit by exploiting it on the go

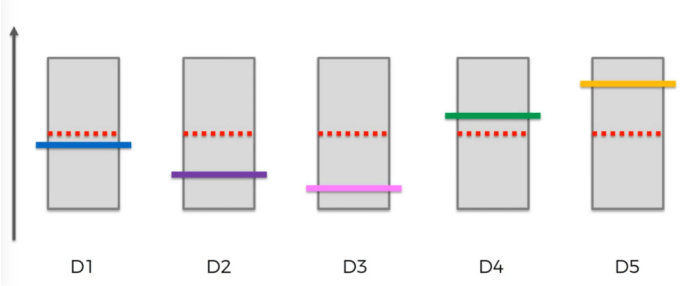
summary and correlation with ad problem

- We have d arms. For example, arms are ads that we display to users each time they connect to a web page.
- Each time a user connects to this web page, that makes a round.
- At each round n, we choose one ad to display to the user.
- At each round n, ad i gives reward $r_i(n) \in \{0,1\}$: $r_i(n) = 1$ if the user clicked on the ad i, 0 if the user didn't.
- Our goal is to maximize the total reward we get over many rounds.



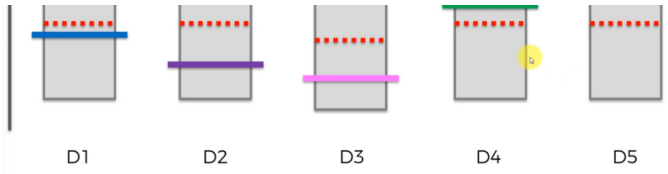
let say second fig give a outcome distribution of each machine but we don't know that other-wise we can predict the best by curve

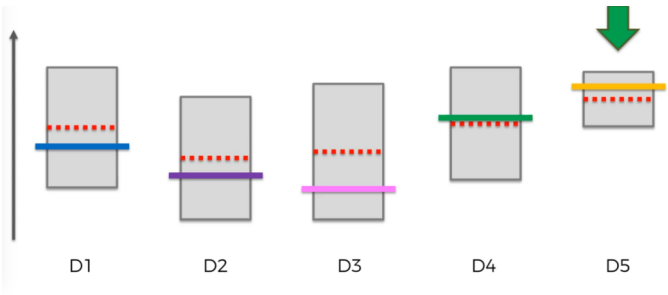
so this algo assume a default value same for each machine and a confidence band which include the actual result

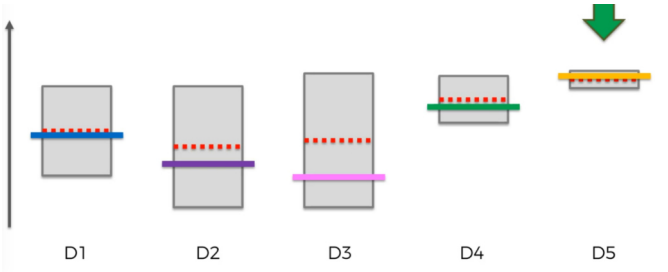


it will try machine which has upper limit as in first case all have same so it will try randomly

let say it try D3 so the red line decreases because event coincide with pink line, with this second thing happen that confidence bound also shrink because we have an add observation so this process will go on and it will find the best slot machine







Now the coding part after import the datafile

in course eg we have a data set which contain 10 ads and 10000 entries for user which tell which ads the user will click but we have to make it look like on the go

so we are using a random function to pick only one ad for each user

```
# Implementing random Selection
import random
N=10000
d=10
ads_selected =[]
total_reward =0
for n in range(0,N)
ad=random.randrange(d)
ads_selected.append(ad)
reward = dataset.values[n , ad]
total_reward=total_reward + reward
# Visualising the results
plt.hist(ads_selected)
plt.title('Histogram of ads selections')
plt.xlabel('Ads')
plt.ylabel('Number of times each ad was selected')
plt.show()
#
# Above code is not used in actuall algo
#
# Implementing UCB
import math
```

```
N = 10000
d = 10
ads_selected = []
numbers_of_selections = [0] * d
sums_of_rewards = [0] * d
total_reward = 0


for n in range(0, N):
    ad = 0
    max_upper_bound = 0
    for i in range(0, d):
        if (numbers_of_selections[i] > 0):
            average_reward = sums_of_rewards[i] / numbers_of_selections[i]
            delta_i = math.sqrt(3/2 * math.log(n + 1) / numbers_of_selections[i])
            upper_bound = average_reward + delta_i
        else:
            upper_bound = 1e400
        if upper_bound > max_upper_bound:
            max_upper_bound = upper_bound
            ad = i
    ads_selected.append(ad)
    numbers_of_selections[ad] = numbers_of_selections[ad] + 1
    reward = dataset.values[n, ad]
    sums_of_rewards[ad] = sums_of_rewards[ad] + reward
    total_reward = total_reward + reward


# Visualising the results
plt.hist(ads_selected)
plt.title('Histogram of ads selections')
plt.xlabel('Ads')
plt.ylabel('Number of times each ad was selected')
plt.show()
#
#
#Now to know the best ads look at the ad_selected vector and ads used most is the best ad and
the end u see it converge to one ad
# or look at the histogram
```

THOMPSON SAMPLING INTUITION

Bayesian Inference

- Ad i gets rewards $\mathbf{y}$ from Bernoulli distribution $p(\mathbf{y}|\theta_i) \sim \mathcal{B}(\theta_i)$.
- θ_i is unknown but we set its uncertainty by assuming it has a uniform distribution $p(\theta_i) \sim \mathcal{U}([0,1])$, which is the prior distribution.
- Bayes Rule: we approach θ_i by the posterior distribution

$$\underbrace{p(\theta_i|\mathbf{y})}_{\text{posterior distribution}} = \frac{p(\mathbf{y}|\theta_i)p(\theta_i)}{\int p(\mathbf{y}|\theta_i)p(\theta_i)d\theta_i} \propto \underbrace{p(\mathbf{y}|\theta_i)}_{\text{likelihood function}} \times \underbrace{p(\theta_i)}_{\text{prior distribution}}$$

- We get $p(\theta_i|\mathbf{y}) \sim \beta(\text{number of successes} + 1, \text{number of failures} + 1)$
- At each round n we take a random draw $\theta_i(n)$ from this posterior distribution $p(\theta_i|\mathbf{y})$, for each ad i.
- At each round n we select the ad i that has the highest $\theta_i(n)$.

Thompson Sampling Algorithm

Step 1. At each round n, we consider two numbers for each ad i:

- $N_i^1(n)$ - the number of times the ad i got reward 1 up to round n,
- $N_i^0(n)$ - the number of times the ad i got reward 0 up to round n.

Step 2. For each ad i, we take a random draw from the distribution below:

$$\theta_i(n) = \beta(N_i^1(n)+1, N_i^0(n)+1)$$

Step 3. We select the ad that has the highest $\theta_i(n)$.

Return

This is a probabilistic model we try few times and try to predict the avg value below img is for three img where the curve of one color shows the area of probability of values of each slot machine