Importing Libraries:

Syntax: import L_b name as sub name

Ex: import numpy as np
import Pandas as pd
import matplotlib.pyplot as plt

Importing dataset:

① In supervise machine learning } Means: In which modil the programmer will know ~~bel~~ about the data set or we can say that the data set is given and the programmer knows about that the output].

→ import pandas as pd

dataset = pd.read_csv('file.csv')
X = dataset.iloc [a:b, c:d].values
Y = dataset.iloc [a1:b1, c1:d1].values

* pd.read_csv : csv = formats of data set
- clipboard ; - excel ; - feather ; - fwf, - gbq
- hdf ; - html ; - json ; - msgpack ; - parquet
- pickle ; - sas, - sql, ~~- sqlq~~
- sql_query ; - sql_table
- stata ; - table

* Now the whole data set is divided into two parts

X and Y :

In which X is the data set which contain the data ~~whose~~ which is used to train the model. OR in other words the data set X helps to find out result

And the other one Y is the data set which use to test the ~~data set~~ model. It means After training the model then check the result कितना सही है।

* Why do we create X and Y separately?
Because we want to work with Numpy arrays instead of pandas dataframe. Numpy arrays are the most convenient format to work with when doing data preprocessing and building Machine learning models. So we create two separate arrays

* iloc :-
It locates the column by its index

* .values :-
It return the values of the columns you are taking inside a Numpy arrays

Q Explain indixing in X, Y arrays?

: a, a1 ; b, b1 → denote lines

: c, c1 ; d, d1 → denote colums

Now a = denote we take all the line
b = denote we do not take this no 'b' line
similar in column
c = denote we take all the column
d :- denote we do not take "d" no of column

* Program file must store in that folder which contain the dataset
· Program's data set stored in the directory in which the program file save

→ we can change the directory on clicking file explorer which is present in the ~~right~~ right side of the spyder

Finding the missing value:-

* For finding missing value and fixed it by using sklearn library

```
→ from sklearn.preprocessing import Imputer
  imputer = Imputer(missing_values='NaN',
                    strategy='mean', axis=0
imputer = imputer.fit(X[:, 1:3])
  X[:, 1:3] = imputer.transform(X[:, 1:3])
```

→ Definition:

① first create the object for further use of Imputer library

② In Imputer we pass some important thing

a) missing_values = 'NaN'

→ integer or "NaN", optional (default = "NaN")
The place holder for the missing values. All the blank place replaced by "nan" by the help of this

b) Strategy = 'mean' {default = 'mean'}

by the help of strategy we find the missing value by appling 'mean', 'mode', 'median', 'most_frequent'

i) mean:- If replace missing values using the mean along the axis

ii) median:- ——————" —— median
——" ——

iii) most_frequent: ——————" —— most_frequent
——

c) axis = 0 {default = "0"}

If axis = 0 then strategy applied along columns
If axis = 1 the strategy applied along line

d) Verbose {default = 0}
* controls the verbosity of the imputer

e) Copy {default = True}

Q) What is the difference between fit and transform?

fit:-
It is used to extract some info of the data and on which the object is applied

transform:
It is used to apply some transform

Dealing with categorical data:-

* In this section: the machine learning knows only deals with integer values or (0 or 1) not string, character (Name etc) so

first convert the character data into integer then convert into the one form like 0 and 1

→ from sklearn.preprocessing import LabelEncoder
labelencoder = X = LabelEncoder()
① — X[:, 0] = labelencoder_X.fit_transform(X[:, 0])

After processing Eq-1: We get some digits like 0, 1, 2 and so on so (However since one is greater than Zero and two is greater than 1 the equation in the model will think (according to machine Example Spain has a higher value than Germany and France and Germany has a higher value than France) They are only a 3 categories not a relational Order between them

For solving this the programmer create Dummy Variable

→ from sklearn.preprocessing import OneHotEncoder
one hot Encoder = OneHotEncoder (categorical_feature = [0])
X = one hot Encoder.fit_transform(X).toarray()

→ LabelEncoder_Y = LabelEncoder()
Y = LabelEncoder_Y.fit_transform(Y)

Q What do the two 'fit_transform' methods do

→ When the 'fit_transform()' method is called from the 'LabelEncoder()' class, it transforms the categories strings in to integers.

Example:
It transform France, Spain, and Gurmany into 0, 1, and 2

→ When the 'fit_transform()' method is called from the 'OneHotEncoder()' class, it creates separate columns for each different label with binary values 0 and 1. Those separate columns are the dummy variables

* Categorical ~~data~~ feature:
on which column the fit_transform is applied

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 = 0.25, random_state = 0)

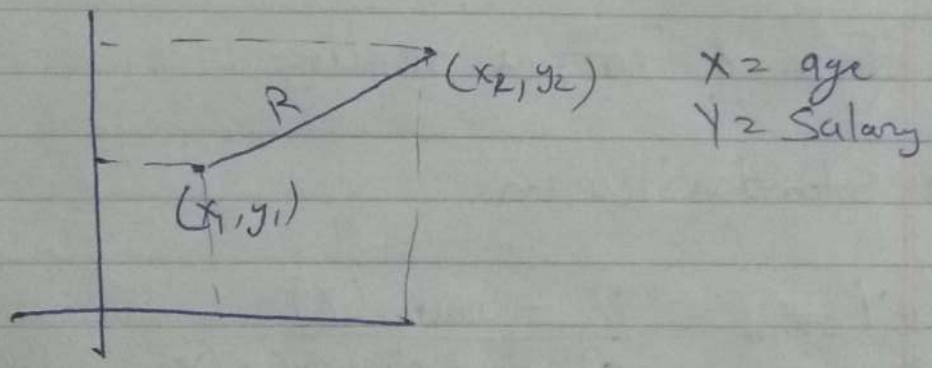
Feature scaling :-

Q Why we use Feature scaling :-

In given data set the age variable (27 to 50) and the salary variable (48k to 90k) are not on the same scale.

The machine learning model and a lot of machine learning models are based on Euclidean method

Euclidean method



R = Euclidean distance = $\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$

The Euclidean distance will be dominated by the salary

Example

(48, 79000) ; (27, 48000)

$R = \sqrt{x^2+y^2}$

$y = 31000$; $y^2 = 961000000$

$x = 21$; $x^2 = 441$

the square of salary (y^2) dominates the square difference of age

this is the reason we use feature scaling for remove the domination of one variable to another variable.

The feature scaling helps to optimize the accuracy of your model prediction

or The programmer does not apply feature scaling to keep the most interpretation as possible in your model

→ Types of Feature scaling

① Standardisation

$$X_{stand} = \frac{X - mean(X)}{standard\ deviation(X)}$$

② Normalisation

$$X_{norm} = \frac{X - min(X)}{max(X) - min(X)}$$

By applying feature scaling the speed of execution of program increases

→ from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler() (compute)
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)

Complete Data processing Code :-

```
# Data processing
# Import the libraries

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

# Importing the dataset
dataset = pd.read_csv('Data.csv')
X = dataset.iloc[:, :-1].values
Y = dataset.iloc[:, 3].values

# Taking care of missing data
from sklearn.preprocessing import Imputer
imputer = Imputer(missing_value='NaN', strategy='mean'
                  axis=0)
imputer = imputer.fit(X[:, 1:3])
X[:, 1:3] = imputer.transform(X[:, 1:3])

# Encoding Categorical data
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
labelencoder_X = LabelEncoder()
X[:, 0] = labelencoder_X.fit_transform(X[:, 0])
onehotencoder = OneHotEncoder(categorical_feature = [0])
X = onehotencoder.fit_transform(X).toarray()
labelEncoder_Y = LabelEncoder()
Y = labelencoder_Y.fit_transform(Y)

# 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 = 0.2, 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)
```

Part

Day :-02

Part :-2 Regression

① Simple linear regression :-

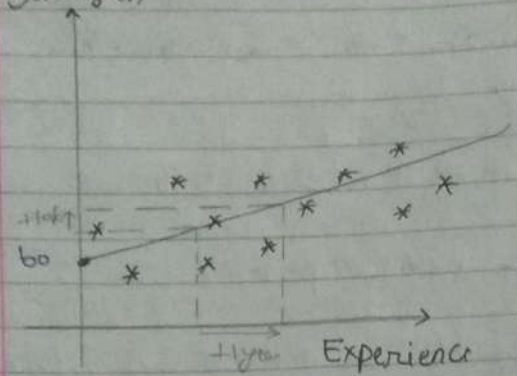
Q What are exactly the cofficients b_0 and b_1 in the SLR equation:

$$\text{Salary} = b_0 + b_1 \text{Experience}$$

Where b_0 and b_1 are the constant cofficient
Salary = dependent variable
Experience = independent variable

Q Why ~~did~~ didn't we apply feature scaling in our SLR model?
It's simply because since y is a linear combination of the independent variables, the cofficients can adapt their scale to put everything on the same scale.
From example if you have two independent variable x_1 and x_2 and if y takes values between 0 and 1 x_1 takes values between 1 and 10 and x_2 takes values between 10 and 100, then b_1 can be multiplied by 0.1 and b_2 can be multiplied by 0.01 so that y, $b_1 \cdot x_1$ and $b_2 \cdot x_2$ are all on the same scale

Q What does 'regressor.fit(X_train, Y_train)' do Exactly?
The fit method will take the values of X_train and Y_train and then will compute the ~~coef~~ coefficients b_0 and b_1 of the simple linear regression equation ($y = b_0 + b_1 x$) ~~as seen in the~~.



Salary = $b_0 + b_1 * \text{Experience}$

Complete Simple Linear Regression code

```
#Simple Linear Regression

# Importing the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

#importing the data set
dataset = pd.read_csv('Salary_Data.csv')
X = dataset.iloc[:,:-1].values
Y = dataset.iloc[:, 1].values

#splitting the data set into the train_test_split
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 1/3, random_state =0)

#fitting Simple_linear regression model
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, Y_train)

# predicting the test set result
Y_pred = regressor.predict(X_test)

# ploting the training set result
plt.scatter(X_train, Y_train, color = 'red')
plt.plot(X_train, regressor.predict(X_train), color = 'blue')
plt.title('Salary vs Experience (test set)')
plt.xlabel('year of Experience')
plt.ylabel('Salary')
plt.show()

#ploting the test set result
plt.scatter(X_test, Y_test, color = 'red')
plt.plot(X_train, regressor.predict(X_train), color = 'blue')
plt.title('Salary vs Experience (test set)')
plt.xlabel('year of Experience')
plt.ylabel('Salary')
plt.show()
```

Simple linear Model

① **plt.scatter :-**
this helps to plot points on a graph like (2.2, 39891) or (2.9, 56642)

② **plt.plot :-**
this helps to draw a linear line which follows this equation $y = a_0 + a_1 x$

- Plt. title :-
  This use to provide the title

- Plt. (xlabel, ylabel) :
  helps to label the x and y cordinates respectively

Linear Regression function :-

Parameters :-

① fit_intercept :- Default :- True {only boolean value passed} (optional)

Whether to calculate the intercept for this model. If set false, no intercept will be used in calculation {eg. data is expected to be already centered}

② normalize :- Default : false {only boolean value is passed} (optional)

* parameter ignored when fit_intercept is set to false.
* If, true, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm.

③ Copy_X : boolean, optional, default = true

If True, X will be copied ; else, it may be overwritten

④ n_jobs :- int or None, optional (default = None)

The number of jobs to use for the computation. This will only provide speedup for n_targets > 1 and sufficient large problems

27/01/19

Day : 3

2: Multiple linear regression

What are the Multiple linear regression assumption in more details :-

Linearity :- There must be a linear relationship b/w the dependent variable and the independent variable. Scatterplot can show whether there is a linear or curvilinear relationship

Homoscadasiticity :- This assumption states that the variance of error terms is similar across the values of the independent variables. A plot of standardized residuals versus predicity values can show whether points are Equally distributed across all values of the independent variables

Multivariate Normality :- MLR assumes that the residuals (total difference b/w predicted value and observed value)

Q How is the coefficient b_0 related to the dummy variable trap?

Since $D_2 = 1 - D_1$ then if you include both D_1 and D_2 you get

$$y = b_0 + b_1 X_1 + b_2 X_2 + b_3 X_3 + b_4 D_1 + b_5 D_2$$
$$= b_0 + b_1 X_1 + b_2 X_2 + b_3 X_3 + b_4 D_1 + b_5 (1 - D_1)$$
$$= b_0 + b_1 X_1 + b_2 X_2 + b_3 X_3 + D_1 (b_4 - b_5)$$
$$= b_0^* + b_1 X_1 + b_2 X_2 + b_3 X_3 + b_4^* D_1$$

with $\boxed{b_0^* = b_0 + b_5 \quad b_4^* = b_4 - b_5}$

Q How can we select the important variable which helps to form best model and lock risk

* In MLR many independent variable are present in dataset; so we have to find the important independent variables which really helps to predict the good result in less the time taken for predicting the result.

If the programmers takes the complete dataset independ variable then the Program looks like a garbage and the model takes time to predict the result

Must watch vidio no: 44 no: 67

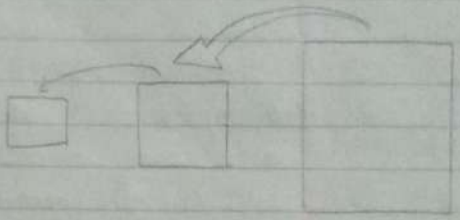
Q Name the method of building models of MLR

5 Methods of building models

1. All in
2. Backward Elimination
3. Forward Elimination
4. Bidirectional Elimination
5. score comparison

(2), (3), (4) — Stepwise regression

Backward Elimination :-



Step

1. Select a significance level to stay in the model (eg: SL = 0.05)
2. Fit the full model with all possible predictors

Example

```
import statsmodels.formula.api as Sm
X = np.append(arr = np.ones((50,1)).astype(int),
              values = X, axis = 1)

X_opt = X[:, [0,1,2,3,4,5]]
re_OLS = Sm.OLS(endog = Y, exog = X_opt).fit()
re_OLS.summary
```

3. Consider the predictor with the highest P-value. If P > SL, go to ↓ Step 4, otherwise go to ↓ Step 5

4. Remove the predictor.

5. Fit model with out this variable.

Code:

```
# Importing the libraries

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

# importing the data set
dataset = pd.read_csv('50_Startups.csv')
X = dataset.iloc[:, :-1].values
y = dataset.iloc[:, -1].values

# imports dealing with categorical data
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
labelencoder_X = LabelEncoder()
X[:, 3] = labelencoder_X.fit_transform(X[:, 3])
onehotencoder = OneHotEncoder(categorical_features = [3])
X = onehotencoder.fit_transform(X).toarray()

# Avoiding the dummy variable trap
X = X[:, 1:]

# splitting the dataset into train_test_split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split
                    (X, y, test_size = 1/3, random_state = 0)

# fitting the multiple linear regression
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, Y_train)
```

# predict the value:
y_pred = re.predict(x_test)

# building the optimal model using backward Elimination
import statsmodels.formula.api as sm
X = np.append(arr = np.ones((50,1)).astype(int), values = X, axis = 1)

① X_opt = X[:,[0,1,2,3,4,5]]
re_OLS = sm.OLS(endog = y, exog = X_opt).fit()
re_OLS.summary()

② X_opt = X[:,[0,1,3,4,5]]
re_OLS = sm.OLS(endog = y, exog = X_opt).fit()
re_OLS.summary()

③ X_opt = X[:,[0,1,4,5]]
re_OLS = sm.OLS(endog = y, exog = X_opt).fit()
re_OLS.summary()

④ X_opt = X[:,[0,4,5]]
re_OLS = sm.OLS(endog = y, exog = X_opt).fit()
re_OLS.summary()

⑤ X_opt = X[:,[4,5]]
re_OLS = sm.OLS(endog = y, exog = X_opt).fit()
re_OLS.summary()

* Only 4 and 5 attribute have 'p' value is less than SL

* toarray() :- Return a dense ndarray representation of this matrix

Polynomial Regression:

① Is Polynomial Regression a linear or non linear model?
That depends on what you are referring to. Polynomial Regression is linear on the coefficients since we have any power of the coefficients (all the coff have the power 1 : $b_0, b_1, \dots b_n$). However, polynomial regression is a non-linear function of the input X, since we have the inputs raised to several powers : $X, X^2, \dots$
That is how we can also see the Polynomial regression as a non linear model.

② Why didn't we apply feature scaling in our P model?
It's simply because, since y is a linear combination of X and X^2, the coff. can adapt their scale to put everything on the same scale. For Example if y takes values between 0 and 1, X takes values between 1 and 10 and X^2 takes values between 1 and 100, then b_1 can be multiplied by 0.01 and b_2 can be multiplied by 0.01 so that y, $b_1 X_1$ and $b_2 X_2$ are all on the same scale

3. How do we find the best degree?
Test several degrees and you see which one gives you the best fit

4 What's the reason behind to create a second linear regression 'lin_reg_2'?

Because 'lin_reg' is already fitted to X and y and now we want to fit a new linear model to X_poly and y. So we have to create a new regressor object

Q Why we create linear_regression model in polynomial linear regression.?
This is because we can create / compair the outcomes which comes from both different model

Q Why we create 'X_poly' variable?
Because we ~~trans~~ transform the variable 'x' into the x^n where n is the degree, which is feeded by the programmer according to the best fitting
Often appling the fit_transform method
index 0 is a dummy variable, index 1 is the a X variable, and index 2 is the X^n variable

* Feature scaling apply only when the algorithm ~~is~~ must based on Euclidian distance

Raw Code:

```
# Polynomial regression

# importing the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# importing the dataset
data = pd.read_csv('Position_salaries.csv')
X = data.iloc[:, 1:2].values
y = data.iloc[:, 2].values

# Fitting the linear regression to the dataset
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)

# Fitting the Polynomial Regression to the dataset
from sklearn.preprocessing import PolynomialFeatures
poly_reg = PolynomialFeatures(degree = 4)
X_poly = poly_reg.fit_transform(X)
lin_reg_2 = LinearRegression()
lin_reg_2.fit(X_poly, y)

# Visualising the Linear Regression result
plt.scatter(X, y, color = 'red')
plt.plot(X, lin_reg.predict(X), color = 'blue')
plt.title('')
plt.xlabel('')
plt.ylabel('')
plt.show()

# Visualising the polynomial regression result
plt.scatter(X, y, color = 'red')
plt.plot(X, lin_reg_2.predict(poly_reg.fit_transform(X)), color = 'blue')
plt.title('')
plt.xlabel('')
plt.ylabel('')
plt.show()

# Visualising the polynomial regression result (for high quality)
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, lin_reg_2.predict(poly_reg.fit_transform(X_grid)), color = 'blue')
plt.title('')
plt.xlabel('')
plt.ylabel('')
plt.show()

# predicting the result by linear regression
lin_reg.predict(6.5)

# predicting the result by Polynomial regression
lin_reg_2.predict(poly_reg.fit_transform(6.5))
```

Polynomial Feature function:

① Generate X_poly matrix

Parameters:

① degree : integer, Default = 2
The degree of the PF

② interaction_only : boolean, Default = False
If true only interaction features are produced: features that are produced of at most degree distinct input features

SVR : Support vector regression

1. When should i use SVR?
You should use SVR if a linear model like linear regression doesn't fit very well your data. This would mean you are dealing with a non-linear problem

Q Why do we need to 'sc_Y.inverse_transform'?
We need the inverse_transform method to go back to the original scale. Indeed we applied feature scaling so we get this scale around '0' and if we make a prediction without inversing the scale we will get the scaled predicted salary. And of course we want the real salary, not the scaled one, so we have to use inverse transform method

* Support vector machines (SVM) support linear and non-linear regression that we can refer to as SVR

* SVR performs linear regression in a higher (dimensional space)

* In SVR each point in train set represent its own dimension

* In SVR we can also apply limit in the margin of e which is distance between the original support vector regression line and the line pass from the element of two different cluster's element. The Element is the nearest element of both the cluster. We can limit the margin by Epsilon

Building SVR Model

Step:

① (optional) Collect a training set $\tau = \{\vec{X}, \vec{Y}\}$

② Choose a kernel and its 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 these coefficients, create your estimator $f(\vec{X}, \vec{\alpha}, x^*) \approx y^*$

* Kernel are
① Gaussian
② Regularization
③ Noise

Raw code:-

```
# SVR
# importing the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# importing the dataset
dataset = pd.read_csv('Position_Salaries.csv')
X = dataset.iloc[:, 1:2].values
y = dataset.iloc[:, 2].values

# Feature Scaling
from sklearn.preprocessing import StandardScaler
sc_X = StandardScaler()
sc_y = StandardScaler()
X = sc_X.fit_transform(X)
y = sc_y.fit_transform(y)

# Fitting SVR model
from sklearn.svm import SVR
regressor = SVR(kernel = 'rbf')
regressor.fit(X, y)

# predicting a new result
y_pred = regressor.predict(6.5)
y_pred = sc_y.inverse_transform(y_pred)

# visualising the graph
```

Decision Tree regression

Q How does the algo split the data points?
It uses reduction of standard deviation of the predictions. In other words, the standard deviation is decreased right after a split. Hence building a decision tree is all about finding the attribute that returns the highest standard deviation reduction

Q What is the Information Gain and how it does it work in Decision tree?
The information gain in Decision tree is exactly the standard Deviation Reduction we are looking to reach. We calculate by how much the standard Deviation decreases after each split. Because the more the standard Deviation is decreased after a split; the more homogeneous the child node will be

Q What is the Entropy and how does it work in Decision trees?
The Entropy measures the disorder in a set, here in a part resulting from a split. So the more homogeneous is your data in a part, the lower will be the Entropy The more you have splits, the more you have chance to find parts in which your data is homogeneous, and therefore the lower will be the entropy (close to 0) in these parts. However you might still find ~~more~~ some nodes where the data is not homogeneous, and therefore the entropy would not be that small.

* Decision tree is not much sense in 1D.

* In Decision tree & data can be divide by applying the splits

Code :

```
# importing the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# importing the dataset
dataset = pd.read_csv('Position_Salaries.csv')
X = dataset.iloc[:, 1:2].values
y = dataset.iloc[:, 2].values

# Implementing the Decision tree Regression algo
from sklearn.tree import DecisionTreeRegressor
regressor = DecisionTreeRegressor(random_state=0)
regressor.fit(X, y)

# predicting the result
y_pred = regressor.predict(6.5)

# Visualising the Regression result - HD
plt.scatter(X, y, color = 'red')
plt.plot(X, regressor.predict(X), color = 'blue')
plt.title('')
plt.xlabel('')
plt.ylabel('')
plt.show()
"""

# Visualising the result (for higher resolution and s
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('')
plt.xlabel('')
plt.ylabel('')
plt.show()
```

* In X_grid ; 0.1. Explain the resolution. Resolution can be increase by increasing the 0 in value like 0.01.

Random Forest Regression

Q What is the advantage and drawback of Random Forests compared to Decision Trees?

Advantage :- Random Forests can give you a better predict power than Decision Trees.

Drawback - Decision tree will give you more interpretability than Random forest, because you can plot the graph of a Decision tree to see the different splits that lead to the prediction, as seen in the Intuition Lecture. Thats something you can't do with Random forest.

Q When to use Random Forest and when to use the other models?

* The best answer to that question is: try them all
* First you need to figure out whether your problem is linear or non-linear.

a) If the problem is linear, you should go for Simple Linear Regression if you only have one feature and Multiple Linear Regression if you have several features.

b) If the problem is non linear, you should go for Polynomial Regression, SVR, Decision tree, and Random ~~tree~~ Forest

The method consists of using a very relevant technique that evaluates your models performance, called k-Fold Cross Validation, and then picking the model that shows the best result.

Feel free to

d How do I know how many trees I should use?
First, I would recommend to choose the number of trees by experimenting. It usually takes less time than we think to figure out a best value by tweaking and tuning your model manually. That's actually what we do in general when we build a Machine learning model! we do it in several shots, by experimenting several values of hyperparmeters like the no of trees.

k-Fold Cross validation and Grid search which are powerful techniques that you can use to find the optimal value of a hyperparameter, like here the no of trees.

Code:-

```
#importing the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

#importing the dataset
data = pd.read_csv('Position_Salaries.csv')
X = data.iloc[:, 1:2].values
y = data.iloc[:, 2].values

#implementing the Random forest model.
from sklearn.ensemble import RandomForestRegressor
r = RandomForestRegressor(n_estimators=100,
                          random_state=0)
r.fit(X, y)

#predicting the result
y_pred = r.predict(6.5)

#visualising the result (High resolution)
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, r.predict(X_grid), color
plt.title('')
plt.xlabel('')
plt.ylabel('')
plt.show()
```

* n_estimators = define the no. of trees
  * No. of trees ↑ the prediction & result will more accurate

R-squared:-

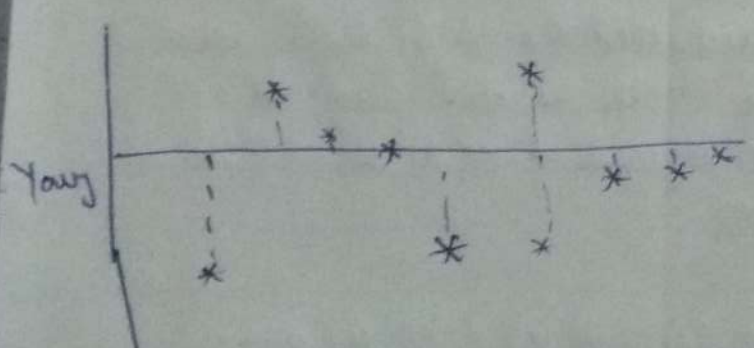
R-squared Explained that how good is your line compared to the you know average line in simple linear regression

$SS_{res} = SUM(y_i - \hat{y_i})^2 = SS\ residual$

$SS_{tot} = SUM(y_i - y_{avg})^2 = SS_{total}$

$$R_2^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

* R^2 can be (-ve) that means direction is wrong and model is broken
* $R^2 = 1$ best case, but not happen. Ideal case



* R^2 is closer to 1 it is good for model

"If you can dream it, you can achieve it."

Endeavor

Hierarchical clustering

* Two types of Hierarchical clustering

① Agglomerative { bottom up approch }

② Divisive { Top to bottom approch }

① Agglomerative Hierarchical clustering:-

Step:-

① Make each data point a single-point cluster ⇒ That forms N cluster

② Takes the two closest data points and make them one cluster → That forms N-1 clusters

③ Takes the two ***closest clusters*** and make them one cluster → That forms N-2 clusters

④ Repeat step 3 until there is only one cluster

Fin

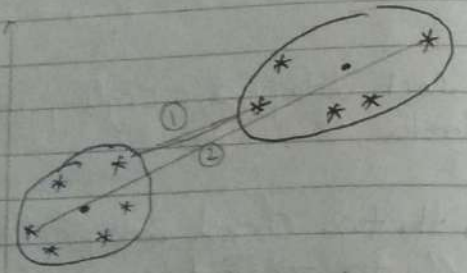
* At the end of all these steps only a large cluster were form.

* In Each repeatation only single combine cluster formed and at the end of this N-n {when 'n' is the no. of steps repeated}.

* Ways to calculate distance b/w two clusters

① closest points
② Furthest Points
③ Average Distance ? sum of all the distance of points lied in two different clusters ?

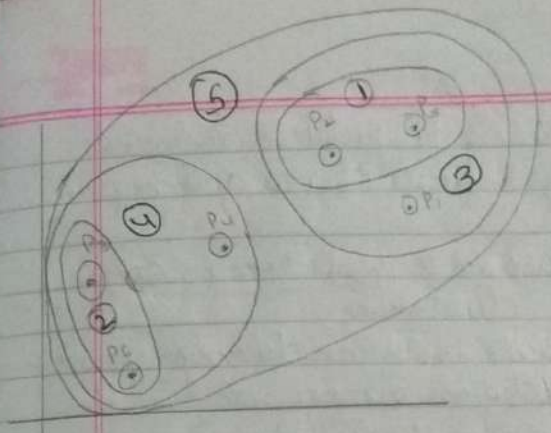
④ Distance B/w centroids



* Distance b/w two cluster is important

Dendograms:-

* Dendogram is kind of like memory of the AC algo
* In Dendogram
  a) Horizontal line ⇒ represent connecting one point to one cluster
  b) vertical line ⇒ shows difference of property of two different connecting cluster



* Dendogram can be formed by algo automatically

* Some Imp. Questions:-

① What is the point of Hierarchical clustering if it always leads to one cluster per observation point?
The main point of HC is to make the dendrogram, because you need to start with one single cluster then work your way down to see the different combination of clusters until having a number of clusters equal to the number of observations. And it's the dendrograms itself that allows to find the best clustering configuration

② When you are comparing the distance b/w two clusters or a cluster and a point, how exactly is it measured? Are you taking the centroid in the cluster and measuring the distance?

Exactly, the metric is the euclidean distance b/w the centroid of the first cluster and the point

mf/3. Do we also need to perform feature scaling of Hierarchical clustering?
Yes because the equation of the clustering problem involve the euclidian distance. Anytime the model equations involve the Euclidean distance, you should apply feature scaling.

imp:- Should we use the dendrogram or the elbow method to find that optimal number of clusters
* You should use both, just to double check the optimal solution

* Priority order :- elbow method > dendrogram
  * If you really only have time for one
* Dendrogram is not always the easiest way to find the optimal number of clusters. But with the elbow method it is very easy, since the elbow is most of the time very obvious to spot

Dendrogram :-

```
import scipy.cluster.hierarchy as sch
d = sch.dendrogram(sch.linkage(X, method
plt.title('Dendrogram')
plt.xlabel('Customers')
plt.ylabel('Distance')
plt.show()
```

Ward method

* method = 'ward' :-

① For method 'single' an optimized algorithm based on minimum spanning tree is implimented. It has time complexity $O(n^2)$.

② Other method like 'complete', 'average', 'weighted', or 'ward' an algorithm called nearest-neighbours chain is implimented. It also has time complexity $O(n^2)$

③ Remaining methods a naive algorithm is implimented with $O(n^3)$ time complexity.

* ④ Method 'centroid', 'median', and 'ward' are correctly defined only if Euclidean pairwise metric is used.

If y is passed as precomputed pairwise distance, then it is used. If y is passed as precomputed pairwise distance, then it is a user responsibility to assure that these distance are in fact Euclidean, other wise the produced result will be incorrect

How the clusters are formed through the ward method of the at agglomerative clustering () python class?
HC is a general family of clustering algo. that build nested clusters by merging or splitting them successively. This hierarchy of clusters is ~~present~~ represented as a tree. The root of the tree is the unique cluster

that gather all the samples, the leaves being the clusters with only one sample. The agglomerative clustering() class performs HC using bottom up approach.

The ward method minimizes the sum of squared differences within all clusters. It is a variance - minimizing approach and in this sense is similar to the K-means ~~to the~~ objective function but tackled with an a AHC approach

* The only purpose of making first the dendrogram is to have an idea of the optimal number of clusters. Then when we apply AHC, we can input the optimal number of clusters that we found thanks to the dendrograms

* affinity:- It refers to the distance; which is ~~the~~ euclidian distance

Elbow method:-

```
from sklearn.cluster import KMeans
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init =
                    'k-means++', random_state
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
plt.plot(range(1, 11), wcss)
plt.title('The Elbow method')
plt.xlabel('No of clusters')
plt.ylabel('WCSS')
plt.show()
```

K-Means:

Where can we apply clustering alg. in real life!

1. Market Segmentation
2. Medicine with for example tumor detection
3. Fraud detection
4. to simply identify some clusters of your ~~Co~~ Customer in your company or business

Q How does the perpendicular line trick work when k = 3

* This trick is mainly used only in 2D or 3D space.

* In high dimensional data we use Euclidian distance to perform the clustering

* In very high dimensional data, we could do a trick, which is an "engulfing sphere" i.e starting at every centroid, start growing a spherical space outward radially. Until the sphere intersect everything the sphere engulfs belongs to the same clusters.

Q How does the elbow method work when more than 2 features are involved?

$$d(p,q) = \sqrt{(p-q)^2 + (p-q)^2 \cdots + (p-q)^2}$$

Q Name the another method/way to the Elbow method to find the ultimate no. of clusters?

Explain in next lecture

Can be done through Parameter tuning with grid search

Hierarchical clustering:-

```
# importing the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

dataset = pd.read_csv('Mall_Customers.csv')
X = dataset.iloc[:, [3, 4]].values

import scipy.cluster.hierarchy as sch
dendrogram = sch.dendrogram(sch.linkage(X, method = 'ward'))
plt.title('Dendrogram')
plt.xlabel('Customers')
plt.ylabel('Distance')

from sklearn.cluster import AgglomerativeClustering
hc = AgglomerativeClustering(n_clusters = 5, affinity
                             = 'euclidean', linkage = 'ward')
y_hc = hc.fit_predict(X)

plt.scatter(X[y_hc == 0, 0], X[y_hc == 0, 1], s = 100,
            c = 'red', label = 'Cluster 1')
plt.scatter(X[y_hc == 1, 0], X[y_hc == 1, 1], s = 100,
            c = 'blue', label = 'Cluster 2')
```

```
plt.title('Clusters of customers')
plt.xlabel('Annual Income(k$)')
plt.ylabel('Spending Score(1-100)')
plt.legend()
plt.show()
```

K 12 P

Par

A priori

What ar
support

Given tha
rulations

b/w s

Step-1 find support
step 2 find confidence
step 3 find lift

* Why we create "transaction" name list.

Part 5 : Association

Apriori :

Q What are three essential relation ship b/w the support, confidence and lift?

Given two movies M_1 and M_2 here are three essential relations to remember :

Relation b/w support and the confidence

$$\text{confidence}(M_1 \rightarrow M_2) = \frac{\text{support}(M_1, M_2)}{\text{support}(M_1)}$$

b/w lift and the support :-

$$\text{lift}(M_1 \rightarrow M_2) = \frac{\text{support}(M_1, M_2)}{\text{support}(M_1) * \text{support}(M_2)}$$

b/w lift and the confidence :

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

Code:

```
# importing the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

#importing the dataset
data = pd.read_csv('Market_Basket_Optimisa
                   .csv', header = None)

transactions = []
for i in range(0, 7501):
    transactions.append([data.values[i,j] for j in
                         range(0, 20)])

# Training the data
from apyori import apriori
rules = apriori(transactions, min_support=0.003, mi
                confidence=0.2, min_lift=3, min_leng

results = list(rules)
```

The Multi-Armed Bandit 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 add 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

Part :- 6 Reinforcement

① Upper confidence Bound:

Q Why does a single round can have multiple 1s for different ads?

Each round corresponds to a user connecting to a web page, and the user can only see one ad when he/she connects to the web page, the ads that is being shown to him/her. If he/she clicks on the ad, the reward is 1, otherwise it's 0. And there can be multiple ones because there are several ads that the user would love to click on. But only a crystal ball would tell us which ads the user would click on. And the data set is exactly that crystal ball.

Q In the first Intuition lecture, could you please explain Why D_5 is the best distribution? Why is it not D_3?

In this situation, 0 is loss and 1 is gain, or win. The D_5 is the best because it is skewed so we will have average ~~out~~ outcome close to 1, meaning that there we have more wins, or gain. And actually all casino machine nowdays are carefully programmed to have distribution like D_1 or D_3. But it is a good concrete example

Q Does the UCB strategy really come into effect during the first rounds

Not at the beginning, first we must have some first insights of the users response to the ads. That is just the

the beginning of the strategy. In real world for the first ten users connecting to the webpage, you would show ad 1 to the first user, ad 2 to the second user, ad 3 to the third user, - - -, ad 10 to the 10th user. Then the algo start.

The main purpose of this algo is not only to maximize the revenue but also to minimize the cost.

* Upper Confidence Bound algo :-

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 upto round n,
   - $R_i(n)$ - the sum of rewards of the ad i upto round n.
2. From these two numbers we compute
   - the average reward of ad i upto 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

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

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

Random_selection

```
# importing the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# importing the dataset
data = pd.read_csv('Ads_CTR_Optimisation.csv')

# 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 = data.values[n, ad]
    total_reward = total_reward + reward

#Visualising the result - Histogram
plt.hist(ads_selected)
plt.title('Histogram of ads selections')
plt.xlabel('Ads')
plt.ylabel('Numbers of times each ad was selected')
plt.show()
```

* Result = total_reward = 11993

UCB algo ? there is no package / libraries to direct implement UCB)

```
# importing the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# importing the dataset
data = pd.read_csv('Ads_CTR_Optimisation.csv')

# Implementing UCB
import math
N = 10000
d = 10
ad_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
```

ads_selected.append(ad)
numbers_of_selections[ad] = numbers_of_selections[ad] + 1
reward = data.values[n, ad]
sums_of_rewards[ad] = sums_of_rewards[ad] + reward
total_reward = total_reward + reward

#Visualising the result by histogram
plt.hist(ads_selected)
plt.title('')
plt.xlabel('')
plt.ylabel('')
plt.show()

* reward/result - total_reward = 2178

* Thompson sampling is better than UCB.

2) Thompson sampling:-

* Thompson sampling algo is probbistic or the UCB is deterministic

Q Why is the yellow mark the best choice, and not the grain mark?

The yellow mark is the best choice because it is the furthest from the origin on the x-axis which therefore means that it has the highest estimated return.

Q I don't understand how Thompson sampling can accept delayed feedback. Please Explain

When doing Thompson sampling, we can still perform updates in our algo. while we are waiting for the result of an experiment in the real world. This would not hinder our algo. from working. This is why it can accept delayed feedback.

Q Comparision b/w UCB & Thompson algo.

UCB	Thompson sampli
• Deterministic	Probabilistic
• Requires update at every round.	can accommodate de feedback.
	• Better empirical evid

Thompson sampling algo:-

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

- $N_i^1(n)$ - the number of time the ad i got reward 1 up to round n
- $N_i^0(n)$ - the number of time 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 $\theta_i(n)$

Thompson sampling algo:

```
# Importing the libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# importing the dataset
data = pd.read_csv('Ads_CTR_Optimisation.csv')

# Implementing Thompson sampling algo
import random
N = 10000
d = 10
ads_selected = []
numbers_of_rewards_1 = [0]*d
numbers_of_rewards_0 = [0]*d
total_reward = 0
for n in range(0,N):
    ad = 0
    max_random = 0
    for i in range(0,d):
        random_beta = random.betavariate(numbers_of_
            rewards_1[i]+1, numbers_of_rewards
            _0[i]+1)
        if random_beta > max_random:
            max_random = random_beta
            ad = i
    ads_selected.append(ad)
    reward = data.values[n, ad]
    if reward == 1:
```

numbers_of_rewards_1[ad] = numbers_of_rewards_1
[ad] + 1

else :
numbers_of_rewards_0[ad] = numbers_of_rewards
_0[ad] + 1
total_reward = total_reward + reward

# Visualising the result - Histogram
plt.hist(ads_selected)
plt.title(' ')
plt.xlabel(' ')
plt.ylabel(' ')
plt.show()

* result = total_reward = 2613