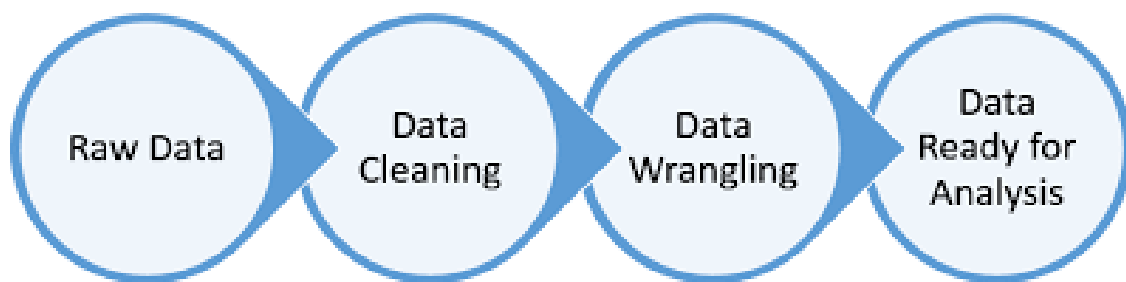


Data Preprocessing in Python

Pre-processing refers to the transformations applied to our data before feeding it to the algorithm. Data Preprocessing is a technique that is used to convert the raw data into a clean data set. In other words, whenever the data is gathered from different sources it is collected in raw format which is not feasible for the analysis.



Need of Data Preprocessing

- For achieving better results from the applied model in Machine Learning projects the format of the data has to be in a proper manner. Some specified Machine Learning model needs information in a specified format, for example, Random Forest algorithm does not support null values, therefore to execute random forest algorithm null values have to be managed from the original raw data set.
- Another aspect is that the data set should be formatted in such a way that more than one Machine Learning and Deep Learning algorithm are executed in one data set, and best out of them is chosen.

Steps involved in data preprocessing :

1. *Importing the required Libraries*
2. *Importing the data set*
3. *Handling the Missing Data.*
4. *Encoding Categorical Data.*
5. Splitting the data set into test set and training set.
6. Feature Scaling.

Step 1: Importing the required Libraries

Every time we make a new model, we will require to import Numpy and Pandas. Numpy is a Library which contains Mathematical functions and is used for scientific computing while Pandas is used to import and manage the data sets.

```
import pandas as pd
import numpy as np
```

Here we are importing the pandas and Numpy library and assigning a shortcut “pd” and “np” respectively.

Step 2: Importing the Dataset

Data sets are available in .csv format. A CSV file stores tabular data in plain text. Each line of the file is a data record. We use the read_csv method of the pandas library to read a local CSV file as a **dataframe**.

```
dataset = pd.read_csv('Data.csv')
```

After carefully inspecting our dataset, we are going to create a matrix of features in our dataset (X) and create a dependent vector (Y) with their respective observations. To read the columns, we will use iloc of pandas (used to fix the indexes for selection) which takes two parameters — [row selection, column selection].

```
X = dataset.iloc[:, :-1].values  
y = dataset.iloc[:, 3].values
```

Step 3: Handling the Missing Data

An example of Missing data and Imputation

```
import numpy as np  
# Importing the SimpleImputer class  
from sklearn.impute import SimpleImputer  
# Imputer object using the mean strategy and  
# missing_values type for imputation  
imputer = SimpleImputer(missing_values = np.nan, strategy = 'mean')  
data = [[12, np.nan, 34], [10, 32, np.nan], [np.nan, 11, 20]]  
print("Original Data : \n", data)  
# Fitting the data to the imputer object  
imputer = imputer.fit(data)  
# Imputing the data  
data = imputer.transform(data)  
print("Imputed Data : \n", data)
```

The data we get is rarely homogenous. Sometimes data can be missing and it needs to be handled so that it does not reduce the performance of our machine learning model.

To do this we need to replace the missing data by the Mean or Median of the entire column. For this we will be using the sklearn.preprocessing Library which contains a class called Imputer which will help us in taking care of our missing data.

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

Our object name is **imputer**. The Imputer class can take parameters like :

1. **missing_values** : It is the placeholder for the missing values. All occurrences of missing_values will be imputed. We can give it an integer or “NaN” for it to find missing values.
2. **strategy** : It is the imputation strategy — If “mean”, then replace missing values using the mean along the axis (Column). Other strategies include “median” and “most_frequent”.
3. **axis** : It can be assigned 0 or 1, 0 to impute along columns and 1 to impute along rows.

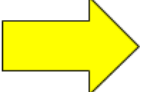
Now we fit the imputer object to our data.

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

Now replacing the missing values with the mean of the column by using transform method.

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

Step 4: Encoding categorical data

Color		Red	Yellow	Green
Red		1	0	0
Red		1	0	0
Yellow		0	1	0
Green		0	0	1
Yellow		0	0	1

Converting Categorical data into dummy variables

Any variable that is not quantitative is categorical. Examples include Hair color, gender, field of study, college attended, political affiliation, status of disease infection.

One-Hot Encoding

One hot encoding transforms categorical features to a format that works better with classification and regression algorithms.

```
from sklearn.preprocessing import OneHotEncoder
```

```
from sklearn.compose import ColumnTransformer
```

```
ct = ColumnTransformer([("Country", OneHotEncoder(), [0])], remainder = 'passthrough')
```

```
X = ct.fit_transform(X)
```

```
labelencoder_y = LabelEncoder()
```

```
y = labelencoder_y.fit_transform(y)
```

Step 5: Splitting the Data set into Training set and Test Set

Now we divide our data into two sets, one for training our model called the **training set** and the other for testing the performance of our model called the **test set**. The split is generally 80/20. To do this we import the “train_test_split” method of “sklearn.model_selection” library.

```
from sklearn.model_selection import train_test_split
```

Now to build our training and test sets, we will create 4 sets —

1. **X_train** (training part of the matrix of features),
2. **X_test** (test part of the matrix of features),
3. **Y_train** (training part of the dependent variables associated with
4. the X train sets, and therefore also the same indices) ,
5. **Y_test** (test part of the dependent variables associated with the X test sets, and therefore also the same indices).

We will assign to them the test_train_split, which takes the parameters — arrays (X and Y),

test_size (Specifies the ratio in which to split the data set).

```
X_train, X_test, Y_train, Y_test = train_test_split( X, Y, test_size = 0.2, random_state = 0)
```

Step 6: Feature Scaling

Most of the machine learning algorithms use the **Euclidean distance** between two data points in their computations. Because of this, **high magnitudes features will weigh more** in the distance calculations **than features with low magnitudes**. To avoid this Feature standardization or Z-score normalization is used. This is done by using “StandardScaler” class of “sklearn.preprocessing”.

```
from sklearn.preprocessing import StandardScaler  
sc_X = StandardScaler()
```

Further we will transform our X_test set while we will need to fit as well as transform our X_train set.

The transform function will transform all the data to a same standardized scale.

```
X_train = sc_X.fit_transform(X_train)  
X_test = sc_X.transform(X_test)
```

Regression

Regression analysis is a statistical method to model the relationship between a dependent (target) and independent (predictor) variables with one or more independent variables. More specifically, Regression analysis helps us to understand how the value of the dependent variable is changing corresponding to an independent variable when other independent variables are held fixed. It predicts continuous/real values such as **temperature, age, salary, price**, etc.

We can understand the concept of regression analysis using the below example:

Example: Suppose there is a marketing company A, who does various advertisement every year and get sales on that. The below list shows the advertisement made by the company in the last 5 years and the corresponding sales:

Advertisement	Sales
\$90	\$1000
\$120	\$1300
\$150	\$1800
\$100	\$1200
\$130	\$1380
\$200	??

Now, the company wants to do the advertisement of \$200 in the year 2019 **and wants to know the prediction about the sales for this year**. So to solve such type of prediction problems in machine learning, we need regression analysis.

Regression is a [supervised learning technique](#)

which helps in finding the correlation between variables and enables us to predict the continuous output variable based on the one or more predictor variables. It is mainly used for **prediction, forecasting, time series modeling, and determining the causal-effect relationship between variables**.

In Regression, we plot a graph between the variables which best fits the given datapoints, using this plot, the machine learning model can make predictions about the data. In simple words, ***"Regression shows a line or curve that passes through all the datapoints on target-predictor graph in such a way that the vertical distance between the datapoints and the regression line is minimum."*** The

distance between datapoints and line tells whether a model has captured a strong relationship or not.

Some examples of regression can be as:

- Prediction of rain using temperature and other factors
- Determining Market trends
- Prediction of road accidents due to rash driving.

Terminologies Related to the Regression Analysis:

- **Dependent Variable:** The main factor in Regression analysis which we want to predict or understand is called the dependent variable. It is also called **target variable**.
- **Independent Variable:** The factors which affect the dependent variables or which are used to predict the values of the dependent variables are called independent variable, also called as a **predictor**.
- **Outliers:** Outlier is an observation which contains either very low value or very high value in comparison to other observed values. An outlier may hamper the result, so it should be avoided.
- **Multicollinearity:** If the independent variables are highly correlated with each other than other variables, then such condition is called Multicollinearity. It should not be present in the dataset, because it creates problem while ranking the most affecting variable.
- **Underfitting and Overfitting:** If our algorithm works well with the training dataset but not well with test dataset, then such problem is called **Overfitting**. And if our algorithm does not perform well even with training dataset, then such problem is called **underfitting**.

Why do we use Regression Analysis?

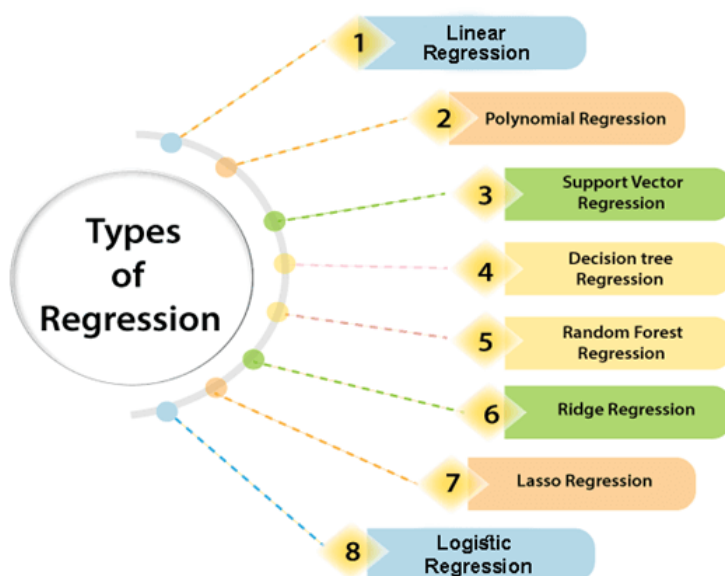
As mentioned above, Regression analysis helps in the prediction of a continuous variable. There are various scenarios in the real world where we need some future predictions such as weather condition, sales prediction, marketing trends, etc., for such case we need some technology which can make predictions more accurately. So for such case we need Regression analysis which is a statistical method and used in machine learning and data science. Below are some other reasons for using Regression analysis:

- Regression estimates the relationship between the target and the independent variable.
- It is used to find the trends in data.
- It helps to predict real/continuous values.
- By performing the regression, we can confidently determine the **most important factor, the least important factor, and how each factor is affecting the other factors.**

Types of Regression

There are various types of regressions which are used in data science and machine learning. Each type has its own importance on different scenarios, but at the core, all the regression methods analyze the effect of the independent variable on dependent variables. Here we are discussing some important types of regression which are given below:

- **Linear Regression**
- **Logistic Regression**
- **Polynomial Regression**
- **Support Vector Regression**
- **Decision Tree Regression**
- **Random Forest Regression**
- **Ridge Regression**
- **Lasso Regression:**



Simple Linear Regression

Simple Linear Regression is a type of Regression algorithms that models the relationship between a dependent variable and a single independent variable. The relationship shown by a Simple Linear Regression model is linear or a sloped straight line, hence it is called Simple Linear Regression.

The key point in Simple Linear Regression is that the ***dependent variable must be a continuous/real value***. However, the independent variable can be measured on continuous or categorical values.

Simple Linear regression algorithm has mainly two objectives:

- **Model the relationship between the two variables.** Such as the relationship between Income and expenditure, experience and Salary, etc.
- **Forecasting new observations.** Such as Weather forecasting according to temperature, Revenue of a company according to the investments in a year, etc.

Simple Linear Regression Model:

The Simple Linear Regression model can be represented using the below equation:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \dots + \beta_r x_r + \varepsilon$$

Where,

This equation is the **regression equation**. $\beta_0, \beta_1, \dots, \beta_r$ are the **regression coefficients**, and ε is the **random error**.

Implementation of Simple Linear Regression Algorithm using Python

Problem Statement example for Simple Linear Regression:

Here we are taking a dataset that has two variables: salary (dependent variable) and experience (Independent variable). The goals of this problem is:

- **We want to find out if there is any correlation between these two variables**
- **We will find the best fit line for the dataset.**
- **How the dependent variable is changing by changing the independent variable.**

we will create a Simple Linear Regression model to find out the best fitting line for representing the relationship between these two variables.

To implement the Simple Linear regression model in machine learning using Python, we need to follow the below steps:

Step-1: Data Pre-processing

The first step for creating the Simple Linear Regression model is **data pre-processing**

First, we will import the three important libraries, which will help us for loading the dataset, plotting the graphs, and creating the Simple Linear Regression model.

```
import numpy as nm
```

```
import matplotlib.pyplot as mtp
```

```
import pandas as pd
```

Next, we will load the dataset into our code:

```
data_set= pd.read_csv('Salary_Data.csv')
```

The above output shows the dataset, which has two variables: Salary and Experience.

After that, we need to extract the dependent and independent variables from the given dataset. The independent variable is years of experience, and the dependent variable is salary. Below is code for it:

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

```
y= data_set.iloc[:, 1].values
```

In the above lines of code, for x variable, we have taken -1 value since we want to remove the last column from the dataset. For y variable, we have taken 1 value as a parameter, since we want to extract the second column and indexing starts from the zero.

In the above output image, we can see the X (independent) variable and Y (dependent) variable has been extracted from the given dataset.

Next, we will split both variables into the test set and training set. We have 30 observations, so we will take 20 observations for the training set and 10 observations for the test set. We are splitting our dataset so that we can train our model using a

training dataset and then test the model using a test dataset. The code for this is given below:

Splitting the dataset into training and test set.

from sklearn.model_selection **import** train_test_split

x_train, x_test, y_train, y_test= train_test_split(x, y, test_size= $\frac{1}{3}$, random_state=0)

By executing the above code, we will get x-test, x-train and y-test, y-train dataset. Consider the below images:

Test-dataset:

	0
0	1.3
1	10.3
2	4.1
3	3.9
4	9.5
5	8.7
6	9.6
7	4
8	5.3
9	7.9

	0
0	39751
1	122391
2	57081
3	63218
4	116969
5	109431
6	112635
7	55794
8	83088
9	101302

Training Dataset:

	0
0	2.7
1	5.1
2	3.2
3	4.5
4	8.2
5	6.8
6	1.1
7	10.5
8	3
9	2.2
10	5.8
11	6
12	3.7

	0
0	56642
1	66029
2	64445
3	61111
4	113812
5	91738
6	45207
7	121872
8	60150
9	39891
10	81363
11	93940
12	57189

- For simple linear Regression, we will not use Feature Scaling. Because Python libraries take care of it for some cases, so we don't need to perform it here.

Now, our dataset is well prepared to work on it and we are going to start building a Simple Linear Regression model for the given problem.

Step-2: Fitting the Simple Linear Regression to the Training Set:

Now the second step is to fit our model to the training dataset. To do so, we will import the **LinearRegression** class of the **linear_model** library from the **scikit learn**. After importing the class, we are going to create an object of the class named as a **regressor**. The code for this is given below:

```
#Fitting the Simple Linear Regression model to the training dataset
from sklearn.linear_model import LinearRegression
regressor= LinearRegression()
regressor.fit(x_train, y_train)
```

In the above code, we have used a **fit()** method to fit our Simple Linear Regression object to the training set. In the fit() function, we have passed the x_train and y_train, which is our training dataset for the dependent and an independent variable. We have fitted our regressor object to the training set so that the model can easily learn the correlations between the predictor and target variables. After executing the above lines of code, we will get the below output.

Output:

```
Out[7]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
normalize=False)
```

Step: 3. Prediction of test set result:

dependent (salary) and an independent variable (Experience). So, now, our model is ready to predict the output for the new observations. In this step, we will provide the test dataset (new observations) to the model to check whether it can predict the correct output or not.

We will create a prediction vector **y_pred**, and **x_pred**, which will contain predictions of test dataset, and prediction of training set respectively.

```
#Prediction of Test and Training set result
y_pred= regressor.predict(x_test)
x_pred= regressor.predict(x_train)
```

On executing the above lines of code, two variables named y_pred and x_pred will generate in the variable explorer options that contain salary predictions for the training set and test set.

Output:

You can check the variable by clicking on the variable explorer option in the IDE, and also compare the result by comparing values from `y_pred` and `y_test`. By comparing these values, we can check how good our model is performing.

Step: 4. visualizing the Training set results:

Now in this step, we will visualize the training set result. To do so, we will use the `scatter()` function of the `pyplot` library, which we have already imported in the pre-processing step. The **`scatter ()` function** will create a scatter plot of observations.

In the x-axis, we will plot the Years of Experience of employees and on the y-axis, salary of employees. In the function, we will pass the real values of training set, which means a year of experience `x_train`, training set of Salaries `y_train`, and color of the observations. Here we are taking a green color for the observation, but it can be any color as per the choice.

Now, we need to plot the regression line, so for this, we will use the **`plot()` function** of the `pyplot` library. In this function, we will pass the years of experience for training set, predicted salary for training set `x_pred`, and color of the line.

Next, we will give the title for the plot. So here, we will use the **`title()` function** of the **`pyplot`** library and pass the name ("Salary vs Experience (Training Dataset)").

After that, we will assign labels for x-axis and y-axis using **`xlabel()` and `ylabel()` function**.

Finally, we will represent all above things in a graph using `show()`. The code is given below:

```
mtp.scatter(x_train, y_train, color="green")
mtp.plot(x_train, x_pred, color="red")
mtp.title("Salary vs Experience (Training Dataset)")
mtp.xlabel("Years of Experience")
mtp.ylabel("Salary(In Rupees)")
mtp.show()
```

Output:

By executing the above lines of code, we will get the below graph plot as an output.



In the above plot, we can see the real values observations in green dots and predicted values are covered by the red regression line. The regression line shows a correlation between the dependent and independent variable.

The good fit of the line can be observed by calculating the difference between actual values and predicted values. But as we can see in the above **plot, most of the observations are close to the regression line, hence our model is good for the training set.**

Step: 5. visualizing the Test set results:

In the previous step, we have visualized the performance of our model on the training set. Now, we will do the same for the Test set. The complete code will remain the same as the above code, except in this, we will use `x_test`, and `y_test` instead of `x_train` and `y_train`.

Here we are also changing the color of observations and regression line to differentiate between the two plots, but it is optional.

```
#visualizing the Test set results

mtp.scatter(x_test, y_test, color="blue")
mtp.plot(x_train, x_pred, color="red")
mtp.title("Salary vs Experience (Test Dataset)")
mtp.xlabel("Years of Experience")
```

```
mtp.ylabel("Salary(In Rupees)")  
mtp.show()
```

Output:

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



In the above plot, there are observations given by the blue color, and prediction is given by the red regression line. As we can see, most of the observations are close to the regression line, hence we can say our Simple Linear Regression is a good model and able to make good predictions.

Multiple Linear Regression

Multiple Linear Regression is an extension of Simple Linear regression as it takes more than one predictor variable to predict the response variable.

We can define it as:

Multiple Linear Regression is one of the important regression algorithms which models the linear relationship between a single dependent continuous variable and more than one independent variable.

Example:

Prediction of CO₂ emission based on engine size and number of cylinders in a car.

Some key points about MLR:

- For MLR, the dependent or target variable(Y) must be the continuous/real, but the predictor or independent variable may be of continuous or categorical form.
- Each feature variable must model the linear relationship with the dependent variable.
- MLR tries to fit a regression line through a multidimensional space of data-points.

MLR equation:

In Multiple Linear Regression, the target variable(Y) is a linear combination of multiple predictor variables $x_1, x_2, x_3, \dots, x_n$. Since it is an enhancement of Simple Linear Regression, so the same is applied for the multiple linear regression equation, the equation becomes:

$$y = \beta_0 + \beta_1 X_1 + \dots + \beta_n X_n + \epsilon$$

Where,

Y= Output/Response variable

$b_0, b_1, b_2, b_3, b_n, \dots$ = Coefficients of the model.

$x_1, x_2, x_3, x_4, \dots$ = Various Independent/feature variable

Assumptions for Multiple Linear Regression:

- A **linear relationship** should exist between the Target and predictor variables.

- The regression residuals must be **normally distributed**.
- MLR assumes little or **no multicollinearity** (correlation between the independent variable) in data.

Implementation of Multiple Linear Regression model using Python:

To implement MLR using Python, we have below problem:

We have a dataset of **50 start-up companies**. This dataset contains five main information: **R&D Spend, Administration Spend, Marketing Spend, State, and Profit for a financial year**. Our goal is to create a model that can easily determine which company has a maximum profit, and which is the most affecting factor for the profit of a company.

Since we need to find the Profit, so it is the dependent variable, and the other four variables are independent variables. Below are the main steps of deploying the MLR model:

1. Data Pre-processing Steps
2. Fitting the MLR model to the training set
3. Predicting the result of the test set

Step-1: Data Pre-processing Step:

The very first step is [data pre-processing](#)

, which we have already discussed in this tutorial. This process contains the below steps:

- **Importing libraries:** Firstly, we will import the library which will help in building the model. Below is the code for it:

```
# importing libraries
```

```
import numpy as nm
```

```
import matplotlib.pyplot as mtp
```

```
import pandas as pd
```

- **Importing dataset:** Now we will import the dataset(50_CompList), which contains all the variables. Below is the code for it:


```
#importing datasets
data_set= pd.read_csv('50_CompList.csv')
```

Output: We will get the dataset as:

In above output, we can clearly see that there are five variables, in which four variables are continuous and one is categorical variable.

- **Extracting dependent and independent Variables:**

```
#Extracting Independent and dependent Variable
```

```
x= data_set.iloc[:, :-1].values
y= data_set.iloc[:, 4].values
```

Out[5]:

As we can see in the above output, the last column contains categorical variables which are not suitable to apply directly for fitting the model. So we need to encode this variable.

Encoding Dummy Variables:

As we have one categorical variable (State), which cannot be directly applied to the model, so we will encode it. To encode the categorical variable into numbers, we will use the **LabelEncoder** class. But it is not sufficient because it still has some relational order, which may create a wrong model. So in order to remove this problem, we will use **OneHotEncoder**, which will create the dummy variables. Below is code for it:

```
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.compose import ColumnTransformer

# Country column
ct = ColumnTransformer([("State", OneHotEncoder(), [3])], remainder = 'passthrough')
x = ct.fit_transform(x)
```

Here we are only encoding one independent variable, which is state as other variables are continuous.

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

The above code will split our dataset into a training set and test set.

Output: The above code will split the dataset into training set and test set. You can check the output by clicking on the variable explorer option given in Spyder IDE. The test set and training set will look like the below image:

Step: 2- Fitting our MLR model to the Training set:

Now, we have well prepared our dataset in order to provide training, which means we will fit our regression model to the training set. It will be similar to as we did in [Simple Linear Regression](#)

model. The code for this will be:

```
#Fitting the MLR model to the training set:
from sklearn.linear_model import LinearRegression
regressor= LinearRegression()
regressor.fit(x_train, y_train)
```

Output:

```
Out[9]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
normalize=False)
```

Now, we have successfully trained our model using the training dataset. In the next step, we will test the performance of the model using the test dataset.

Step: 3- Prediction of Test set results:

The last step for our model is checking the performance of the model. We will do it by predicting the test set result. For prediction, we will create a **y_pred** vector. Below is the code for it:

```
#Predicting the Test set result;
y_pred= regressor.predict(x_test)
```

By executing the above lines of code, a new vector will be generated under the variable explorer option. We can test our model by comparing the predicted values and test set values.

In the above output, we have predicted result set and test set. We can check model performance by comparing these two value index by index. For example, the first index has a predicted value of **103015\$** profit and test/real value of **103282\$** profit. The difference is only of **267\$**, which is a good prediction, so, finally, our model is completed here.

- We can also check the score for training dataset and test dataset. Below is the code for it:

```
print('Train Score: ', regressor.score(x_train, y_train))  
print('Test Score: ', regressor.score(x_test, y_test))
```

Output: The score is:

```
Train Score:  0.9501847627493607  
Test Score:  0.9347068473282446
```

The above score tells that our model is 95% accurate with the training dataset and 93% accurate with the test dataset.

Applications of Multiple Linear Regression:

There are mainly two applications of Multiple Linear Regression:

- Effectiveness of Independent variable on prediction:
- Predicting the impact of changes:

Polynomial Regression

- Polynomial Regression is a regression algorithm that models the relationship between a dependent(y) and independent variable(x) as nth degree polynomial. The Polynomial Regression equation is given below:

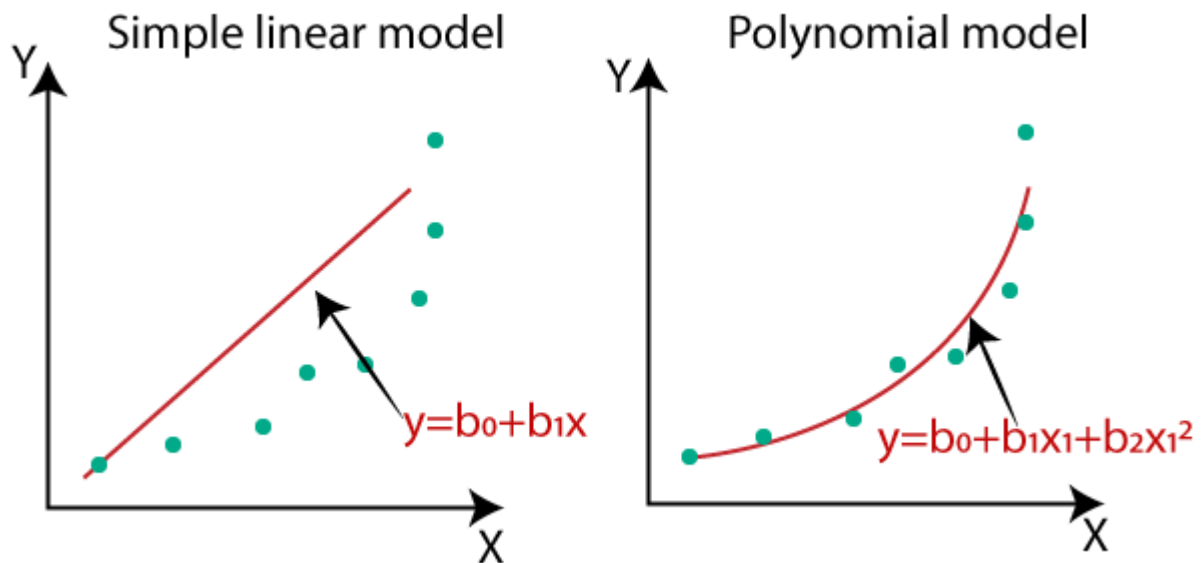
$$y = b_0 + b_1x_1 + b_2x_1^2 + b_3x_1^3 + \dots + b_nx_1^n$$

- It is also called the special case of Multiple Linear Regression in ML. Because we add some polynomial terms to the Multiple Linear regression equation to convert it into Polynomial Regression.
- It is a linear model with some modification in order to increase the accuracy.
- The dataset used in Polynomial regression for training is of non-linear nature.
- It makes use of a linear regression model to fit the complicated and non-linear functions and datasets.
- **Hence, "In Polynomial regression, the original features are converted into Polynomial features of required degree (2,3,...,n) and then modelled using a linear model."**

Need for Polynomial Regression:

The need of Polynomial Regression in ML can be understood in the below points:

- If we apply a linear model on a **linear dataset**, then it provides us a good result as we have seen in Simple Linear Regression, but if we apply the same model without any modification on a **non-linear dataset**, then it will produce a drastic output. Due to which loss function will increase, the error rate will be high, and accuracy will be decreased.
- So for such cases, **where data points are arranged in a non-linear fashion, we need the Polynomial Regression model**. We can understand it in a better way using the below comparison diagram of the linear dataset and non-linear dataset.



- In the above image, we have taken a dataset which is arranged non-linearly. So if we try to cover it with a linear model, then we can clearly see that it hardly covers any data point. On the other hand, a curve is suitable to cover most of the data points, which is of the Polynomial model.
- Hence, *if the datasets are arranged in a non-linear fashion, then we should use the Polynomial Regression model instead of Simple Linear Regression.*

Note: A Polynomial Regression algorithm is also called Polynomial Linear Regression because it does not depend on the variables, instead, it depends on the coefficients, which are arranged in a linear fashion.

Equation of the Polynomial Regression Model:

Simple Linear Regression equation:

$$y = b_0 + b_1x \quad \text{.....(a)}$$

Multiple Linear Regression equation:

$$y = b_0 + b_1x + b_2x_2 + b_3x_3 + \dots + b_nx_n \quad \text{.....(b)}$$

Polynomial Regression equation:

$$y = b_0 + b_1x + b_2x^2 + b_3x^3 + \dots + b_nx^n \quad \text{.....(c)}$$

When we compare the above three equations, we can clearly see that all three equations are Polynomial equations but differ by the degree of variables. The Simple and Multiple Linear equations are also Polynomial equations with a single degree, and the Polynomial regression equation is Linear equation with the nth degree. So if we

add a degree to our linear equations, then it will be converted into Polynomial Linear equations.

Implementation of Polynomial Regression using Python:

Here we will implement the Polynomial Regression using Python. We will understand it by comparing Polynomial Regression model with the Simple Linear Regression model. So first, let's understand the problem for which we are going to build the model.

Problem Description: There is a Human Resource company, which is going to hire a new candidate. The candidate has told his previous salary 160K per annum, and the HR have to check whether he is telling the truth or bluff. So to identify this, they only have a dataset of his previous company in which the salaries of the top 10 positions are mentioned with their levels. By checking the dataset available, we have found that there is a **non-linear relationship between the Position levels and the salaries**. Our goal is to build a **Bluffing detector regression** model, so HR can hire an honest candidate. Below are the steps to build such a model.

Position	Level(X-variable)	Salary(Y-Variable)
Business Analyst	1	45000
Junior Consultant	2	50000
Senior Consultant	3	60000
Manager	4	80000
Country Manager	5	110000
Region Manager	6	150000
Partner	7	200000
Senior Partner	8	300000
C-level	9	500000
CEO	10	1000000

Steps for Polynomial Regression:

The main steps involved in Polynomial Regression are given below:

- Data Pre-processing
- Build a Linear Regression model and fit it to the dataset

- Build a Polynomial Regression model and fit it to the dataset
- Visualize the result for Linear Regression and Polynomial Regression model.
- Predicting the output.

Note: Here, we will build the Linear regression model as well as Polynomial Regression to see the results between the predictions. And Linear regression model is for reference.

Data Pre-processing Step:

The data pre-processing step will remain the same as in previous regression models, except for some changes. In the Polynomial Regression model, we will not use feature scaling, and also we will not split our dataset into training and test set. It has two reasons:

- The dataset contains very less information which is not suitable to divide it into a test and training set, else our model will not be able to find the correlations between the salaries and levels.
- In this model, we want very accurate predictions for salary, so the model should have enough information.

The code for pre-processing step is given below:

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

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

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

Explanation:

- In the above lines of code, we have imported the important Python libraries to import dataset and operate on it.

- Next, we have imported the dataset '**Position_Salaries.csv**', which contains three columns (Position, Levels, and Salary), but we will consider only two columns (Salary and Levels).
- After that, we have extracted the dependent(Y) and independent variable(X) from the dataset. For x-variable, we have taken parameters as `[:,1:2]`, because we want 1 index(levels), and included `:2` to make it as a matrix.

Output:

By executing the above code, we can read our dataset as:

Index	Position	Level	Salary
0	Business Analyst	1	45000
1	Junior Consultant	2	50000
2	Senior Consultant	3	60000
3	Manager	4	80000
4	Country Manager	5	110000
5	Region Manager	6	150000
6	Partner	7	200000
7	Senior Partner	8	300000
8	C-level	9	500000
9	CEO	10	1000000

As we can see in the above output, there are three columns present (Positions, Levels, and Salaries). But we are only considering two columns because Positions are equivalent to the levels or may be seen as the encoded form of Positions.

Here we will predict the output for level **6.5** because the candidate has 4+ years' experience as a regional manager, so he must be somewhere between levels 7 and 6.

Building the Linear regression model:

Now, we will build and fit the Linear regression model to the dataset. In building polynomial regression, we will take the Linear regression model as reference and compare both the results. The code is given below:

```
#Fitting the Linear Regression to the dataset
from sklearn.linear_model import LinearRegression
```



```
lin_regs= LinearRegression()  
lin_regs.fit(x,y)
```

In the above code, we have created the Simple Linear model using **lin_regs** object of **LinearRegression** class and fitted it to the dataset variables (x and y).

Output:

```
Out[5]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,  
normalize=False)
```

Building the Polynomial regression model:

Now we will build the Polynomial Regression model, but it will be a little different from the Simple Linear model. Because here we will use **PolynomialFeatures** class of **preprocessing** library. We are using this class to add some extra features to our dataset.

```
#Fitting the Polynomial regression to the dataset  
from sklearn.preprocessing import PolynomialFeatures  
poly_regs= PolynomialFeatures(degree= 2)  
x_poly= poly_regs.fit_transform(x)  
lin_reg_2 =LinearRegression()  
lin_reg_2.fit(x_poly, y)
```

In the above lines of code, we have used **poly_regs.fit_transform(x)**, because first we are converting our feature matrix into polynomial feature matrix, and then fitting it to the Polynomial regression model. The parameter value(degree= 2) depends on our choice. We can choose it according to our Polynomial features.

After executing the code, we will get another matrix **x_poly**, which can be seen under the variable explorer option:

	0	1	2
0	1	1	1
1	1	2	4
2	1	3	9
3	1	4	16
4	1	5	25
5	1	6	36
6	1	7	49
7	1	8	64
8	1	9	81
9	1	10	100

Next, we have used another LinearRegression object, namely **lin_reg_2**, to fit our **x_poly** vector to the linear model.

Output:

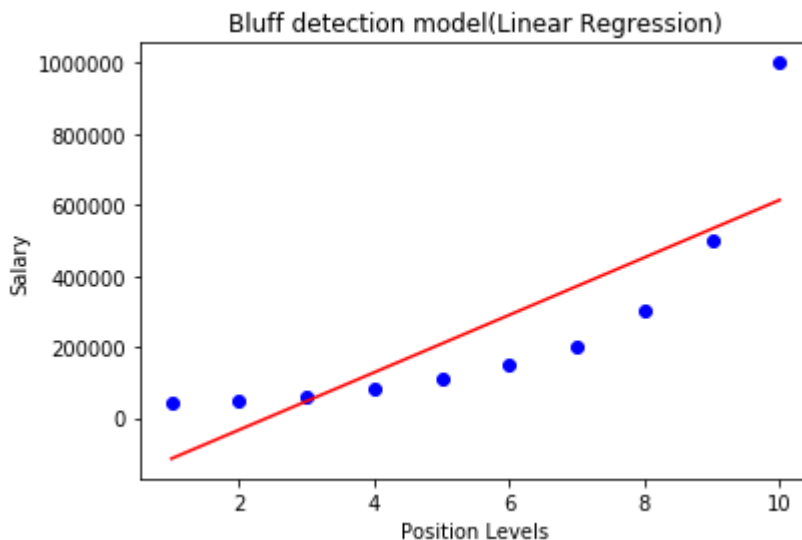
```
Out[11]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
normalize=False)
```

Visualizing the result for Linear regression:

Now we will visualize the result for Linear regression model as we did in Simple Linear Regression. Below is the code for it:

```
#Visualizing the result for Linear Regression model
mtp.scatter(x,y,color="blue")
mtp.plot(x,lin_regs.predict(x), color="red")
mtp.title("Bluff detection model(Linear Regression)")
mtp.xlabel("Position Levels")
mtp.ylabel("Salary")
mtp.show()
```

Output:



In the above output image, we can clearly see that the regression line is so far from the datasets. Predictions are in a red straight line, and blue points are actual values. If we consider this output to predict the value of CEO, it will give a salary of approx. 600000\$, which is far away from the real value.

So we need a curved model to fit the dataset other than a straight line.

Visualizing the result for Polynomial Regression

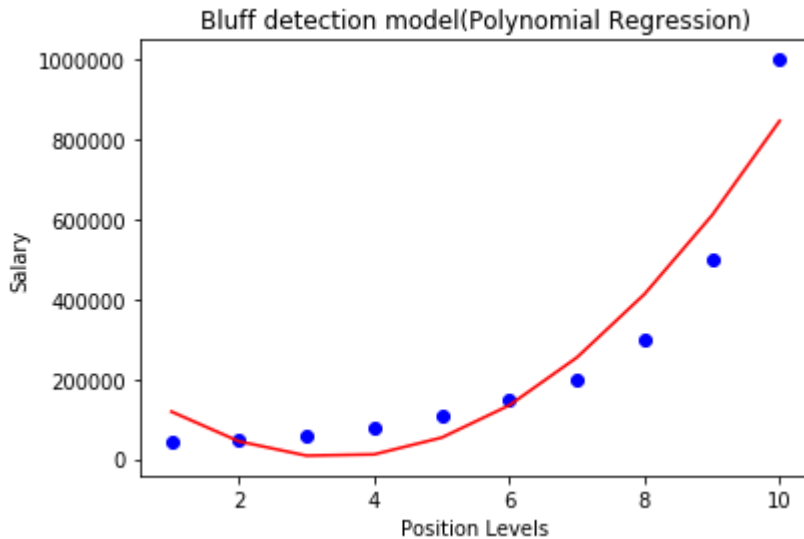
Here we will visualize the result of Polynomial regression model, code for which is little different from the above model.

Code for this is given below:

```
#Visulaizing the result for Polynomial Regression
mtp.scatter(x,y,color="blue")
mtp.plot(x, lin_reg_2.predict(poly_regs.fit_transform(x)), color="red")
mtp.title("Bluff detection model(Polynomial Regression)")
mtp.xlabel("Position Levels")
mtp.ylabel("Salary")
mtp.show()
```

In the above code, we have taken `lin_reg_2.predict(poly_regs.fit_transform(x))`, instead of `x_poly`, because we want a Linear regressor object to predict the polynomial features matrix.

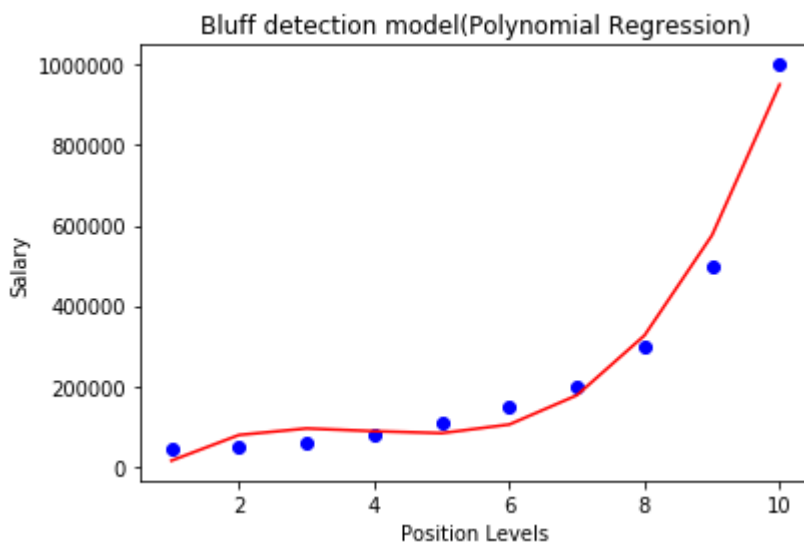
Output:



As we can see in the above output image, the predictions are close to the real values. The above plot will vary as we will change the degree.

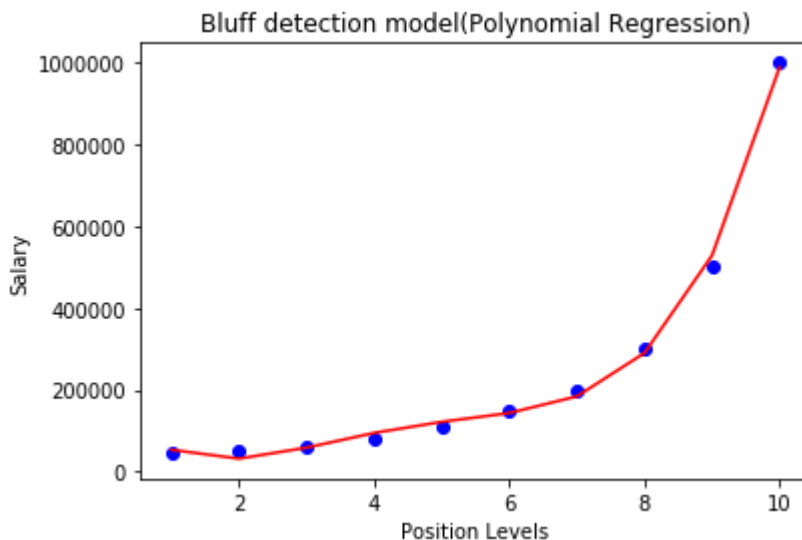
For degree= 3:

If we change the degree=3, then we will give a more accurate plot, as shown in the below image.



SO as we can see here in the above output image, the predicted salary for level 6.5 is near to 170K\$-190k\$, which seems that future employee is saying the truth about his salary.

Degree= 4: Let's again change the degree to 4, and now will get the most accurate plot. Hence we can get more accurate results by increasing the degree of Polynomial.



Predicting the final result with the Linear Regression model:

Now, we will predict the final output using the Linear regression model to see whether an employee is saying truth or bluff. So, for this, we will use the **predict()** method and will pass the value 6.5. Below is the code for it:

```
lin_pred = lin_regs.predict([[6.5]])  
print(lin_pred)
```

Output:

```
[330378.78787879]
```

Predicting the final result with the Polynomial Regression model:

Now, we will predict the final output using the Polynomial Regression model to compare with Linear model. Below is the code for it:

```
poly_pred = lin_reg_2.predict(poly_regs.fit_transform([[6.5]]))  
print(poly_pred)
```

Output:

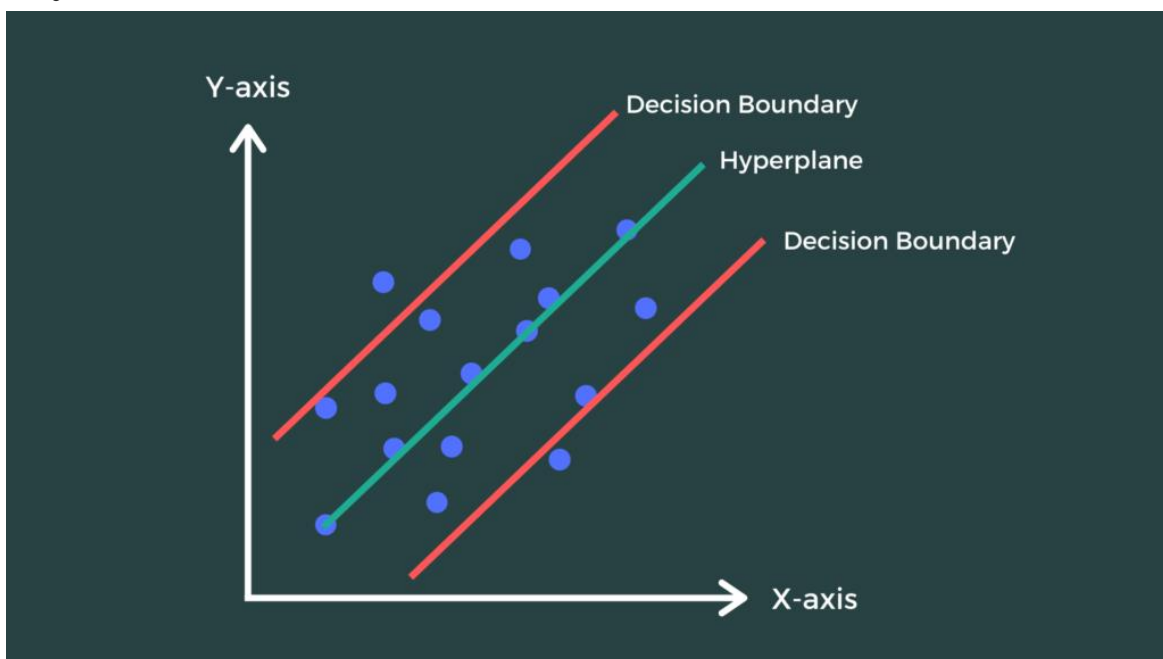
```
[158862.45265153]
```

As we can see, the predicted output for the Polynomial Regression is [158862.45265153], which is much closer to real value hence, we can say that future employee is saying true.

Support Vector Regression

Support Vector Regression (SVR) is a supervised learning model that can be used to perform both linear and nonlinear regressions. The goal of applying Support Vector Regression to a data set is to make sure that the errors do not exceed the threshold. In SVR, we fit as many instances as possible between the lines while limiting the margin violation. An SVR model uses the following hyper parameters in its model that determine the performance of the model.

- **Kernel:** The function used to map lower-dimensional data into higher dimensional data.
- **Hyper Plane:** The separation line between the data classes. For a Support Vector Regression problem, a hyperplane is a line that will help us predict the continuous value or target value.
- **Decision Boundary line:** The boundary lines are essentially the decision boundaries of the hyperplane. The support vectors can be on the Boundary lines or outside it. The best fit line is determined on the basis of the hyperplane having the maximum number of points inside its boundary line.
- **Support Vectors** are the data points that are closest to the decision boundary. The distance of the points is minimum or least.



Implementing SVR in Python

Data preprocessing

Imports these libraries:

get the libraries

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

get the dataset

```
dataset = pd.read_csv('/content/drive/MyDrive/Position_Salaries.csv')
# our dataset in this implementation is small, and thus we can print it all instead of viewing
only the end
print(dataset)
```

Output:

The above dataset contains ten instances. The significant feature in this dataset is the Level column. The Position column is just a description of the Level column, and therefore, it adds no value to our analysis. Therefore, we will separate the dataset into a set of features and study variables.

Variable separation:

```
# split the data into features and target variable separately
X_l = dataset.iloc[:, 1:-1].values # features set
y_p = dataset.iloc[:, -1].values # set of study variable
```

It's seen from the output above that the y_p variable is a vector, i.e., a 1D array.

Therefore, if we implement a model on this data, the study variable will dominate the feature variable, such that its contribution to the model will be neglected.

Due to this, we will have to scale this study variable to the same range as the scaled study variable.

Due to this, we have to reshape our y_p variable from 1D to 2D. The code below does this for us:

```
y_p = y_p.reshape(-1,1)
```

Output:

```
[[ 45000]
 [ 50000]
 [ 60000]
 [ 80000]
 [110000]
 [150000]
 [200000]
 [300000]
 [500000]
[1000000]]
```

From the above output, `y_p` was successfully reshaped into a 2D array.

Now, import the `StandardScaler` class and scale up the `X_l` and `y_p` variables separately as shown:

```
from sklearn.preprocessing import StandardScaler
StdS_X = StandardScaler()
StdS_y = StandardScaler()
X_l = StdS_X.fit_transform(X_l)
y_p = StdS_y.fit_transform(y_p)
```

Let's simultaneously print and check if our two variables were scaled.

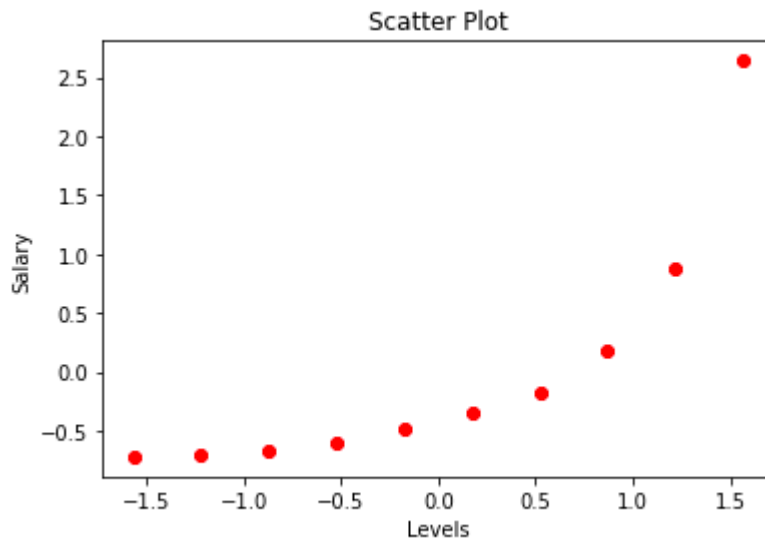
```
print("Scaled X_l:")
print(X_l)
print("Scaled y_p:")
print(y_p)
```

As we can see from the obtained output, both variables were scaled within the range -3 and +3.

Our data is now ready to implement our SVR model.

However, before we can do so, we will first visualize the data to know the nature of the SVR model that best fits it. So, let us create a scatter plot of our two variables.

```
plt.scatter(X_l, y_p, color = 'red') # plotting the training set
plt.title('Scatter Plot') # adding a title to our plot
plt.xlabel('Levels') # adds a label to the x-axis
plt.ylabel('Salary') # adds a label to the y-axis
plt.show() # prints
```

The plot shows a non-linear relationship between the Levels and Salary.

Due to this, we cannot use the linear SVR to model this data. Therefore, to capture this relationship better, we will use the SVR with the [kernel functions](#).

Implementing SVR

To implement our model, first, we need to import it from the scikit-learn and create an object to itself.

Since we declared our data to be non-linear, we will pass it to a kernel called the [Radial Basis function](#) (RBF) kernel.

After declaring the kernel function, we will fit our data on the object. The following program performs these rules:

```
# import the model
from sklearn.svm import SVR
# create the model object
regressor = SVR(kernel = 'rbf')
#RBF(Radial Basis Function)      linear/poly/rbf
# fit the model on the data
regressor.fit(X_l, y_p)
```

Since the model is now ready, we can use it and make predictions as shown:

```
A=regressor.predict(StdS_X.transform([[6.5]]))
print(A)
```

Output:

```
array([-0.27861589])
```

As we can see, the model prediction values are for the scaled study variable. But, the required value for the business is the output of the unscaled data. So, we need to get back to the real scale of the study variable.

So, for any predicted value to fit within such a new dimension of the study variable, it must be transformed from 1D to 2D; otherwise, we will get an error.

```
# Convert A to 2D
A = A.reshape(-1,1)
print(A)
```

Output:

```
array([[ -0.27861589]])
```

It is clear from the output above is a 2D array. Using the `inverse_transform()` function, we can convert it to an unscaled value in the original dataset as shown:

```
# Taking the inverse of the scaled value
A_pred = StdS_y.inverse_transform(A)
print(A_pred)
```

Output:

```
array([[170370.0204065]])
```

Here is the result, and it falls within the expected range.

However, if we were to run a polynomial regression on this data and predict the same values, we would have obtained the predicted values as 158862.45265155, which is only fixed on the curve. With the Support Vector regression, this is not the case. So there is that allowance given to the model to make the best prediction.

Code optimization

We can optimize the above operation into a single line of code as below.

```
B_pred = StdS_y.inverse_transform(regressor.predict(StdS_X.transform([[6.5]])).reshape(-1,1))
print(B_pred)
```

Output:

```
array([[170370.0204065]])
```

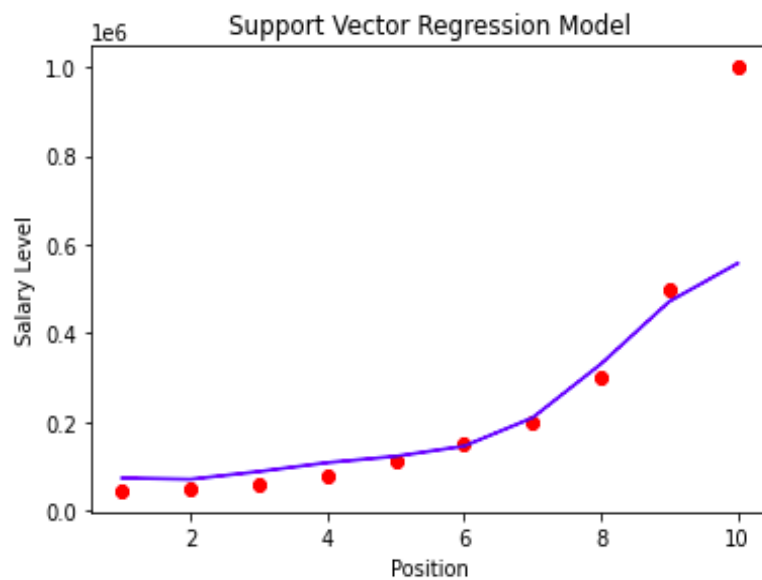
Since we now know how to implement and make predictions using the SVR model, the final thing we will do is to visualize our model.

The following code carries out this task:

```
# inverse the transformation to go back to the initial scale
```

```
plt.scatter(StdS_X.inverse_transform(X_l), StdS_y.inverse_transform(y_p), color = 'red')
plt.plot(StdS_X.inverse_transform(X_l),
StdS_y.inverse_transform(regressor.predict(X_l).reshape(-1,1)), color = 'blue')
# add the title to the plot
plt.title('Support Vector Regression Model')
# label x axis
plt.xlabel('Position')
# label y axis
plt.ylabel('Salary Level')
# print the plot
plt.show()
```

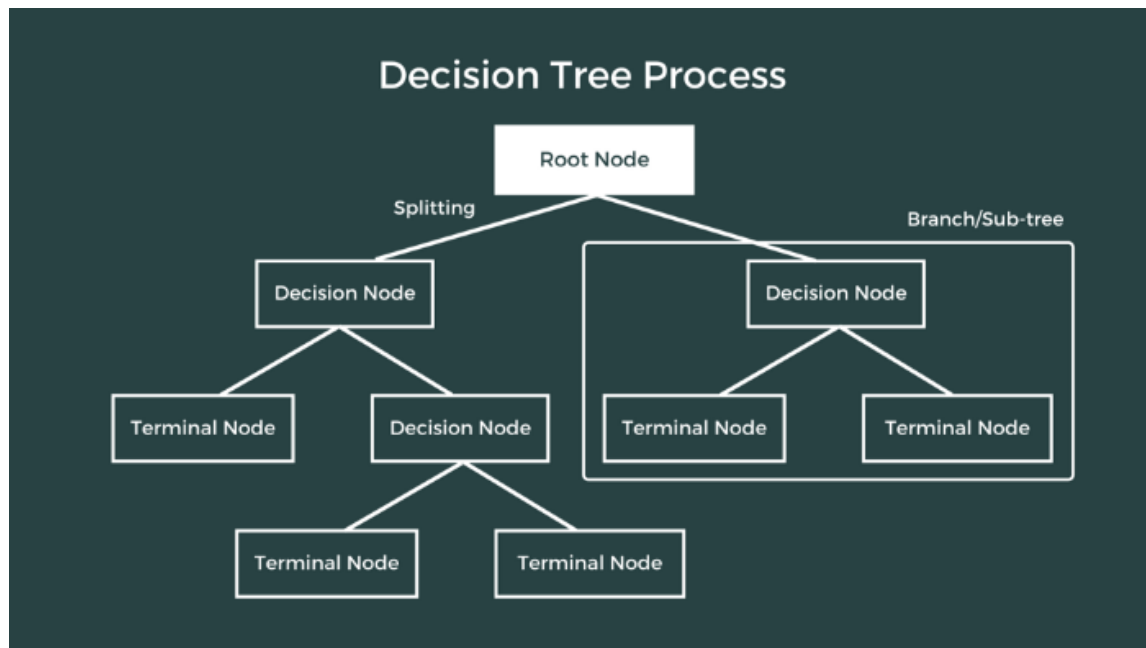
Output:



Decision Tree Regression

Decision Tree

A decision tree is one of the most frequently used Machine Learning algorithms for solving **regression** as well as **classification** problems. As the name suggests, the algorithm uses a tree-like model of decisions to either predict the target value (regression) or predict the target class (classification).



- **Root Node:** This represents the topmost node of the tree that represents the whole data points.
- **Splitting:** It refers to dividing a node into two or more sub-nodes.
- **Decision Node:** They are the nodes that are further split into sub-nodes, i.e., this node that is split is called a decision node.
- **Leaf / Terminal Node:** Nodes that do not split are called Leaf or Terminal nodes. These nodes are often the final result of the tree.
- **Branch / Sub-Tree:** A subsection of the entire tree is called branch or sub-tree.
- **Parent and Child Node:** A node, which is divided into sub-nodes is called a parent node of sub-nodes whereas sub-nodes are the child of the parent node. In the figure above, the decision node is the parent of the terminal nodes (child).

- **Pruning:** Removing sub-nodes of a decision node is called pruning. Pruning is often done in decision trees to prevent overfitting.

Decision Tree work

The process of splitting starts at the root node and is followed by a branched tree that finally leads to a leaf node (terminal node) that contains the prediction or the final outcome of the algorithm. Construction of decision trees usually works top-down, by choosing a variable at each step that best splits the set of items. Each sub-tree of the decision tree model can be represented as a binary tree where a decision node splits into two nodes based on the conditions.

Decision trees where the target variable or the terminal node can take continuous values (typically real numbers) are called **regression trees** which will be discussed in this lesson. If the target variable can take a discrete set of values these trees are called **classification trees**.

Step 1: Import the required libraries.

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

Step 2: Initialize and print the Dataset.

```
# import dataset
# dataset = pd.read_csv('Data.csv')
# alternatively open up .csv file to read data
```

```
dataset=np.array(
[['Asset Flip', 100, 1000],
['Text Based', 500, 3000],
['Visual Novel', 1500, 5000],
['2D Pixel Art', 3500, 8000],
['2D Vector Art', 5000, 6500],
['Strategy', 6000, 7000],
['First Person Shooter', 8000, 15000],
['Simulator', 9500, 20000],
['Racing', 12000, 21000],
['RPG', 14000, 25000],
['Sandbox', 15500, 27000],
['Open-World', 16500, 30000],
['MMOFPS', 25000, 52000],
['MMORPG', 30000, 80000]
])
```

```
# print the dataset
print(dataset)
```

Step 3: Select all the rows and column 1 from the dataset to “X”.

```
# select all rows by : and column 1 by 1:2 representing features
X=dataset.iloc[:, 1:2].astype(int)
print(X)
```

Step 4: Select all of the rows and column 2 from the dataset to “y”.

```
# select all rows by : and column 2
y =dataset.iloc[:, 2].astype(int)
print(y)
```

Output:

```
[ 1000  3000  5000  8000  6500  7000 15000 20000 21000 25000 27000 30000 52000 80000]
```

Step 5: Fit decision tree regressor to the dataset

```
# import the regressor
From sklearn.tree import DecisionTreeRegressor

# create a regressor object
regressor =DecisionTreeRegressor(random_state =0)

# fit the regressor with X and Y data
regressor.fit(X, y)
```

Output:

```
DecisionTreeRegressor(ccp_alpha=0.0, criterion='mse', max_depth=None,
max_features=None, max_leaf_nodes=None,
min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, presort='deprecated',
random_state=0, splitter='best')
```

Step 6: Predicting a new value

```
# predicting a new value
# test the output by changing values, like 3750
y_pred =regressor.predict([[3750]])
```

```
# print the predicted price
print("Predicted price: % d\n"%y_pred)
```

Output:

```
Predicted price: 8000
```

Step 7: Visualising the result

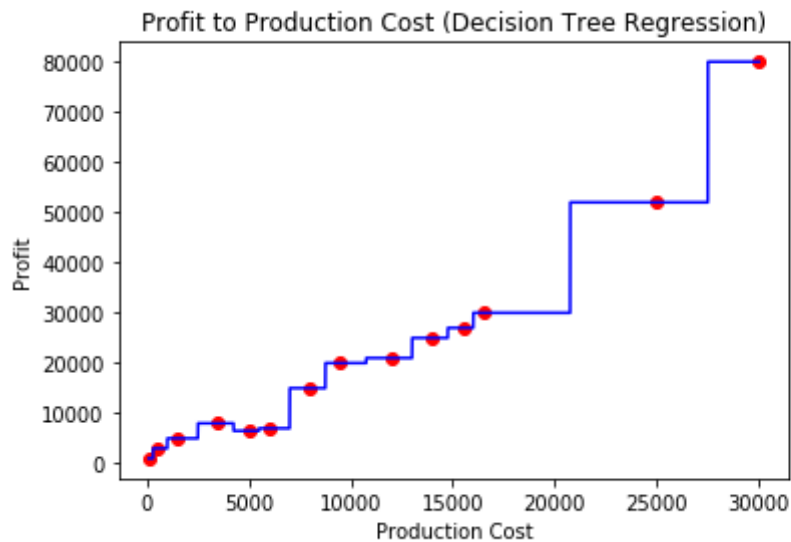
```
X_grid =np.arange(min(X), max(X), 0.01)
```

```
# reshape for reshaping the data into
```

```

# a len(X_grid)*1 array, i.e. to make
# a column out of the X_grid values
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('Profit to Production Cost (Decision Tree Regression)')
plt.xlabel('Production Cost')
plt.ylabel('Profit')
plt.show()

```



Step 8: The tree is finally exported and shown in the TREE STRUCTURE below, visualized

```

from sklearn.tree import plot_tree
from sklearn import tree
plt.subplots(figsize=(45, 40))
tree.plot_tree(regressor)
plt.show()Output (Decision Tree):

```

