3. LINEAR REGRESSION

In the below example, we will be working on a housing dataset trying to create a model to predict housing prices based upon the existing features

```
from google.colab import drive
 drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
import pandas as pd
 import numpy as np

df=pd.read_csv('/content/drive/MyDrive/ML Lab/USA_Housing.csv')
df.head()
```

	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price	
0	79545.458574	5.682861	7.009188	4.09	23086.800503	1.059034e+06	208 Michael 674\nLaur

```
df.describe()  #---gives an account of statistical numbers of the dataframe
```

	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price
count	5000.000000	5000.000000	5000.000000	5000.000000	5000.000000	5.000000e+03
mean	68583.108984	5.977222	6.987792	3.981330	36163.516039	1.232073e+06
std	10657.991214	0.991456	1.005833	1.234137	9925.650114	3.531176e+05
min	17796.631190	2.644304	3.236194	2.000000	172.610686	1.593866e+04
25%	61480.562388	5.322283	6.299250	3.140000	29403.928702	9.975771e+05
50%	68804.286404	5.970429	7.002902	4.050000	36199.406689	1.232669e+06
75%	75783.338666	6.650808	7.665871	4.490000	42861.290769	1.471210e+06
max	107701.748378	9.519088	10.759588	6.500000	69621.713378	2.469066e+06

Divide the dataset into data and labels

```
X=df[['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number of Rooms',
       'Avg. Area Number of Bedrooms', 'Area Population']]
y=df['Price']
```

Divide the dataset into train and test dataset

```
 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=10)
#---random_state gives us the randomness in the split of the dataset
```

Create and Train the Model

```
 from sklearn.linear_model import LinearRegression

lm=LinearRegression()
lm.fit(X_train,y_train)
```

```
LinearRegression()
```

Evaluating the Model

```
print(lm.intercept_)
#---returns the intercept of the model
```

```
-2627104.6624616296
```

```
Lm.coef_
#---returns the coefficient of each feature
```

```
array([2.15815476e+01, 1.65387561e+05, 1.19480881e+05, 1.26155976e+03,
       1.52330337e+01])
```

```
X_train.columns
```

```
Index(['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number of Rooms',
       'Avg. Area Number of Bedrooms', 'Area Population'],
      dtype='object')
```

create the dataframe of the coefficients and their corresponding features

```
cdf=pd.DataFrame(lm.coef_,X.columns,columns=['Coeff'])
cdf
```

	Coeff
Avg. Area Income	21.581548
Avg. Area House Age	165387.561302
Avg. Area Number of Rooms	119480.880523
Avg. Area Number of Bedrooms	1261.559755
Area Population	15.233034

Predictions

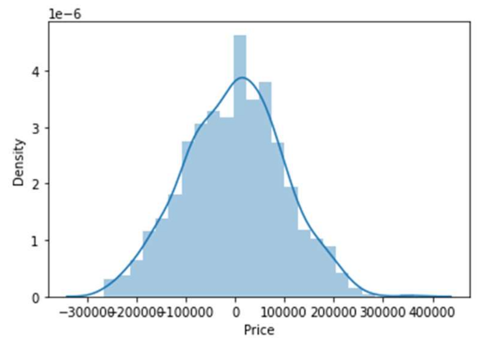
```
predictions=lm.predict(X_test)
print(predictions)
#----prints the predicted prices of the test data
```

```
[1195058.57686777 1185265.81533394 1046786.18912141  827779.10087163
 1485163.51852817  542138.67656906 2473734.91051554 1447749.45072346
 1322766.64693212 1137661.55592712 1373281.5332969  1719966.99319518
 1372477.12886964  521453.39951913 1412349.85322239 1074289.43772196
  959129.69791297 1125293.74910385 1281045.09627646 1009354.62576307
  788527.61237036 1070710.32466421 1286070.3134668  1054795.95470599
 1248708.19974997 1446840.1040186   376036.04699424 1275418.65003585
 1204407.28849144 1500191.62082191 1634136.01083338 1579737.86341595
 1492533.30221774  380169.03592358 1813542.15699057 1216232.59480279
 1556216.5663798  1070938.6827649  1231003.47995495 1076791.63249564
 1005529.40900257 1788883.43344541 1037652.18880185 1435240.71184002
  833401.69258938  888585.17957254 1135122.87341006  906362.57346584
 1315708.06079081 1504142.03564301 1039185.53538879 1100211.37049307
 1384869.6553268  1200247.79396012 1434283.21881892 1033550.96298737
 1308054.92681456 1040966.53077915 1248892.81897506 1688966.68947409
 1294565.23743182 1154012.07633326 1564396.41306413 1066515.06839952
 1063853.95040258 1171627.9585449  1330809.39741521 1396718.44425596
  914942.29440672  333028.97528029 1237403.44772102 1510837.76176718
  644750.15151063  890879.21651868  561199.30038835 1413413.82346402
  472760.61534687  856001.16826853 2003467.44151865 1610624.07784381
 1298855.40895054 1563258.7483035  1218826.13829269 1364902.47502827
 1493072.15464443 1262157.34687643 1080617.08335814  760089.58707337
 1592897.06834013 1126811.03584491  861770.6090358  1308696.32249411
  815552.13446182 1503897.13579057 1147240.98912237 1223929.46857118
 1724150.52782489 1330634.35787643 1432688.8622638  1427608.45516375
  779452.86423463  995360.44072375 1261800.00071935 1486224.23072818
 1218782.07854204  755852.73325812 1352259.10424676 1185127.84174571
 1324217.88846749 1010631.87458897 1000582.6347852  1778881.85154317
 1427335.6893585  1271756.09625649 1412394.42094484  897892.66292713
  929347.71354236 1331906.86763277  731693.85549888 1107208.23008267
 1017453.35228256  762051.68849156 1217293.80108105  747625.61786101
 1201468.75392815 1757166.74580624 1432014.58743352  984187.06090841
 1375938.1885741   609299.48157038 1036197.19617444 1736354.90875664
 1361471.78344618 1382330.9036085  1568087.78658757 1270445.82840212
  885080.67812009 1040058.30656772 1102976.25213826 1118835.90845265
 1489732.89562434 1128255.00324122 1646505.20318664 1440165.690414
 1084741.99909048  955969.23595788 1192497.25155405 1414903.31064779
 1845356.37914496  698862.39398656 1233742.00398621 1511835.46427449
 1384891.62036136 1059491.0734903  1539483.13746397 1767648.51854636
 1151210.60691603 1218701.18492515 1374614.04015617  966751.09234745
 1347387.24249331 1062852.33147057  621044.98096609 1290526.94127037
 1123553.33935298  993429.6251153  1072347.8721411  1181495.30695054
 1102252.08139149  629646.78418936 1495388.09203582 1217162.64767695
 1114602.21444291  976992.69072691 1700640.94690627 1391595.79239828
 1285197.26439415  729329.83946897 1157455.80621576 1249597.9848707
  491984.92936959  745846.74620137 1204018.49724276 1292494.85489027
 1252895.20506425  977031.60192752  908042.95682114 1394565.15269751
  487958.43548616 1571770.63530504 1478145.40758478 1457616.35153603
 1318327.99356959 1325392.08057379 1481471.74655303 1460608.13910621
 1414373.63824735 1359773.67803581  528380.68186885 1060647.64533498
 1289435.4830723  1823929.44265541 1421593.27319543 1794401.02847611
  669908.43087636 1270861.37336913  988002.49538576 1541541.58363554
  467777.72697182 1771003.64753243 1424864.17752307 1683483.08623522
 1184964.84964056 1111399.92039405 1213277.96438038 1106829.68779565
 1153682.50921279 1076046.46724822 1751807.27769856  665213.58646646
 1450168.34176578  820891.89450544 1349511.8948598  1550570.64756427
 1164113.01637004 1274478.96601775 2047647.6704336  1245398.7452038
 1084428.25231552  969761.79124351 1373249.70752477 1540610.42229149
```

```python
import seaborn as sns

sns.distplot((y_test-predictions))
#---Plotting the residuals i.e, |Original values -predicted values|
#---If you have normally distributed residuals then the model is correct choice
```

```
/usr/local/lib/python3.8/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a d
  warnings.warn(msg, FutureWarning)
<matplotlib.axes._subplots.AxesSubplot at 0x7faa355acb50>
```



Evaluating Metrics

Regression Evaluation Metrics

Here are three common evaluation metrics for regression problems:

Mean Absolute Error (MAE) is the mean of the absolute value of the errors:

$$\frac{1}{n}\sum_{i=1}^{n}|y_i - y_i|$$

Mean Squared Error (MSE) is the mean of the squared errors:

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - y_i)^2$$

Root Mean Squared Error (RMSE) is the square root of the mean of the squared errors:

$$\sqrt{\frac{1}{n}\sum_{i=1}(y_i - y_i)^2}$$

Comparing these metrics:

- **MAE** is the easiest to understand, because it's the average error.
- **MSE** is more popular than MAE, because MSE "punishes" larger errors, which tends to be useful in the real world.
- **RMSE** is even more popular than MSE, because RMSE is interpretable in the "y" units.

All of these are **loss functions**, because we want to minimize them.

```
from sklearn import metrics

MAE=metrics.mean_absolute_error(y_test,predictions)
MSE=metrics.mean_squared_error(y_test,predictions)
RMSE=np.sqrt(MSE)
print('MAE=',MAE)
print('MSE=',MSE)
print('RMSE=',RMSE)
```

```
MAE= 80530.8689222398
MSE= 10074785208.854475
RMSE= 100373.22954281422
```

1) Polynomial Regression

Comparing Linear Vs polynomial

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

dataset = pd.read_csv('/content/drive/MyDrive/ML Lab/Position_Salaries.csv')
print(dataset)
X = dataset.iloc[:, 1:2].values   #1:-1 is same
y = dataset.iloc[:, -1].values
print('______________________________')
print(X)
print(y)
```

```
            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
______________________________
[[ 1]
 [ 2]
 [ 3]
 [ 4]
 [ 5]
 [ 6]
 [ 7]
 [ 8]
 [ 9]
 [10]]
[  45000   50000   60000   80000  110000  150000  200000  300000  500000
 1000000]
```

Training the Linear Regression model on the whole dataset

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X, y)
```

```
LinearRegression()
```

Training the Polynomial Regression model on the whole dataset

```
from sklearn.preprocessing import PolynomialFeatures

poly_reg = PolynomialFeatures(degree = 1)       #b0 + b1x + b2x^2 + b3x^3 + ....
X_poly = poly_reg.fit_transform(X)
lin_reg_2 = LinearRegression()
lin_reg_2.fit(X_poly, y)
```

```
LinearRegression()
```

-----Arguments----

PolynomialFeatures(degree=2 (-- degree of polynomial features),

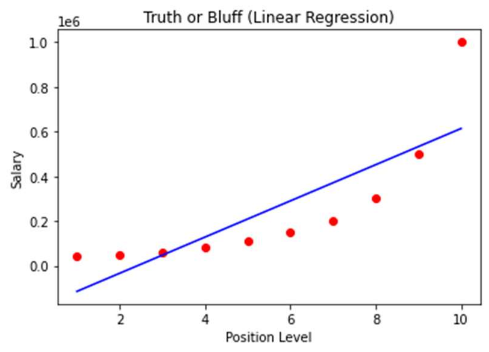
```
interaction_only=False(produces interaction features if true, ignores x[1] ^2 or x[1]^3),

include_bias=True( ----If True (default),then include a bias column, the feature in which all polynomial powers are zero),
```

)

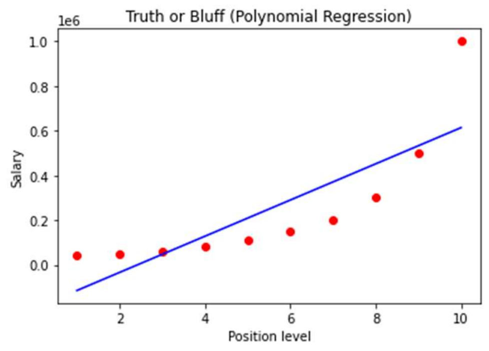
Visualising the Linear Regression results

```
plt.scatter(X, y, color = 'red')
plt.plot(X, lin_reg.predict(X), color = 'blue')
plt.title('Truth or Bluff (Linear Regression)')
plt.xlabel('Position Level')
plt.ylabel('Salary')
plt.show()
```



Visualising the Polynomial Regression results

```
plt.scatter(X, y, color = 'red')
plt.plot(X, lin_reg_2.predict(poly_reg.fit_transform(X)), color = 'blue')
plt.title('Truth or Bluff (Polynomial Regression)')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()
```



Support Vector Regression (SVR)

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

dataset = pd.read_csv('/content/drive/MyDrive/ML Lab/Position_Salaries.csv')
X = dataset.iloc[:, 1:-1].values
y = dataset.iloc[:, -1].values

from sklearn.svm import SVR
```

```
regressor = SVR(kernel = 'rbf')
regressor.fit(X, y)
regressor.predict([[6.5]])
```

```
array([130001.82883924])
```

-----Arguments----

SVR(kernel='rbf' (--'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable.default='rbf'),

```
degree=3(--- If we are using 'poly' kernel then it has to be specified, otherwise ignored by other kernels),

gamma='scale'(---If gamma is large, then variance is small implying the support vector does not have wide-spread influence. large gamma leads to h
{'scale', 'auto'} ,Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.
If gamma='scale' then it uses 1 / (n_features * X.var()) and gamma='auto the it uses 1 / n_features),

coef0=0.0 (---Independent term in kernel function.Significant for 'poly' and 'sigmoid'),

tol=0.001 (---Tolerance for stopping criterion.default=1e-3),

C=1.0 (---C is the parameter for the soft margin cost function, which controls the influence of each individual support vector; this process invol

epsilon=0.1 (---It controls the width of the ε-insensitive zone, used to fit the training data. The value of ε can affect the number of support ve

shrinking=True,

cache_size=200 (---Specify the size of the kernel cache (in MB)),

verbose=False,

max_iter=-1 (---Hard limit on iterations within solver, or -1 for no limit.default=-1),
```

)