FLIP ROBO

Car Price Prediction

Submitted by:

Harshitha K.S

Introduction

The price of the new car in the industry is fixed by the manufacturer with some additional costs incurred by the Government in the form of taxes. So, customers buying a new car can be assured of the money they invest to be worthy. But, due to the Covid19 impact and increased prices of the new cars and the financial incapability of the customers to buy them, used car sales on a global increase. Therefore, there is an urgent need for a used car prediction system which effectively determines the worthiness of the car using variety of features. Existing system includes a process where seller decides a price randomly and buyer has no idea about the car and its value in the present day scenario. In fact, seller also has no idea about the car existing value or the price he should be selling the car at. To overcome this problem, we have developed a model which will be highly effective. Regression Algorithm are used because they provide us with continuos value as output and not a categorized value. Because of which it will be possible to predict the actual price of a car rather than price range of a car. User interface has also been developed which acquires input from any user and displays the price of a car according to the user's inputs

Problem Statement

With the Covid19 impact in the market, we have seen lot of changes in the market. Now some cars in the demand hence making them costly and some are not in demand hence cheaper. One of the clients work with small traders, who sell used cars. With the change in market due to covid19 impact, The client is facing problems with the previous car price valuation. So they are looking for new machine learning models from new data

Objective

The main objective of this project is to predict the car price.

EDA STEPS

1. Importing Libraries
2. Loading the dataset
3. Checking the missing value
4. Checking the d-type of the dataset
5. Checking the information of the dataset
6. Checking the distribution of the categorical variable

1.Importing Libraries

```
import numpy as np#for Data Analysis
import pandas as pd#for scientific computataion
import matplotlib.pyplot as plt#for Data Visualization
import seaborn as sns#for Data Visualization
```

2.Loading the dataset

```
df=pd.read_csv(r'F:\ucar2.csv')
```

3.Checking the missing values

```
df.isnull().sum()
```

```
Brand           0
fueltype        0
mileage         0
model           0
price           0
transmission    0
variant         0
year            0
dtype: int64
```

4.Checking the d-types of the dataset

```
df.dtypes
```

```
Brand           object
fueltype        object
mileage          int64
model           object
price            int64
transmission    object
variant         object
year             int64
dtype: object
```

5.Checking the information of the dataset

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 199 entries, 0 to 198
Data columns (total 8 columns):
 #   Column        Non-Null Count  Dtype 
---  ------        --------------  ----- 
 0   Brand         199 non-null    object
 1   fueltype      199 non-null    object
 2   mileage       199 non-null    int64 
 3   model         199 non-null    object
 4   price         199 non-null    int64 
 5   transmission  199 non-null    object
 6   variant       199 non-null    object
 7   year          199 non-null    int64 
dtypes: int64(3), object(5)
memory usage: 12.6+ KB
```

6.checking the columns

```
df.columns
```

```
Index(['Brand', 'fueltype', 'mileage', 'model', 'price', 'transmission',
       'variant', 'year'],
      dtype='object')
```

7.Checking the distribution of the categorical variables

Checking the value count of fuel type

```
df.fueltype.value_counts()
```

```
Diesel            72
0                 71
Petrol            54
CNG & Hybrids      2
Name: fueltype, dtype: int64
```

Checking the value count of transmission

```
df.transmission.value_counts()
```

```
0            106
Manual        70
Automatic     23
Name: transmission, dtype: int64
```

Checking the value count of brand

```
df.Brand.value_counts()
```

```
NO rating        71
Maruti Suzuki    29
Hyundai          27
Mahindra          8
Ford              8
BMW               8
Toyota            8
Volkswagen        6
Mercedes-Benz     6
Tata              5
Renault           5
Honda             5
Audi              3
Chevrolet         2
Fiat              2
Land Rover        2
Kia               1
Nissan            1
Mitsubishi        1
Skoda             1
Name: Brand, dtype: int64
```

Checking the value count of model

```
df.model.value_counts()
```

```
0              71
Grand i10       6
Swift Dzire     6
i20             5
Innova          4
               ..
Q3              1
Bolt            1
Venue           1
Tigor           1
Rapid           1
Name: model, Length: 72, dtype: int64
```

EDA STEPS

1.Checking the missing values

```
#1. Checking the Missing Values

missing_value=[feature for feature in df.columns if df[feature].isnull().sum()>1]
```

```
missing_value
```

```
[]
```

2.checking for numerical columns

```
#Checking the number of numerical features
numerical_feature=[feature for feature in df.columns if df[feature].dtypes!="O"]
```

```
df[numerical_feature]
```

	mileage	price	year
0	70500	2650000	2015
1	0	2295000	2018
2	0	530000	2016
3	0	32000	2013
4	30808	140999	2014
...	...	...	...
194	0	500000	2011
195	0	345000	2013
196	0	625000	2018
197	0	410000	4036
198	0	420000	2014

199 rows × 3 columns

```
print('Number of numerical variables', len(numerical_feature))
```

```
Number of numerical variables 3
```

3.checking for the distribution of numerical variables

```
#Checking the number of numerical features
numerical_feature=[feature for feature in df.columns if df[feature].dtypes!="O"]
```

```
df[numerical_feature]
```

	mileage	price	year
0	70500	2650000	2015
1	0	2295000	2018
2	0	530000	2016
3	0	32000	2013
4	30808	140999	2014
...	...	...	...
194	0	500000	2011
195	0	345000	2013
196	0	625000	2018
197	0	410000	4036
198	0	420000	2014

199 rows × 3 columns

```
print('Number of numerical variables', len(numerical_feature))
```

```
Number of numerical variables 3
```

```
#checking the number of unique values present in numerical column

print("Number of unique values in numeric column:", df['price'].nunique())
print("The unique value in the numerical column: \n",df['price'].unique())
```

```
Number of unique values in numeric column: 154
The unique value in the numerical column: 
 [ 2650000  2295000   530000    32000   140999   400000   355000   640000
  1845000  1025000   599000   199999   225000   265000   499599   285000
   250000   841000   300000   275000   860000  3450000   560000  1021000
  1735000  2241000   824000   800000   361000   710000  1600000   396000
  1075000  4650000   590000   484000   385000   675000  5200000   215000
   470000    40000  1530000   145000   535000  1300000  1100000   780000
   450000  1750000   199000   980000  3700000  1900000   245000   750000
   120000   550000   375000  1350000  1709999   161000   420000   990000
   545000   525000   820000   730000   350000   865000   340000   330000
   561000    35000   415000   380000    95000   660000   655555   575000
   490000   570000   440000   495000  1650000  1397000   251000   211000
   125000  1050000   625000 26510297   210000   200000   465000   425000
   320000  4900000   155000  2250000    68000   240000   430000   150000
   975000   370000   485000  2992000    82000  1625000   540000    94000
   160000  2775000   220000   230000 14771998   175000   195000  1890000
   850000   235000  5500000   585000   295000   345000   581000   140000
   655000  2372000   915000   650000    79000  1256000   100000  3350000
    60002  1085000  1250000   565000   799733   139999  1175000    99000
   725000   311000   390000   855000  1270000   711000  2850000   130000
   500000   410000]
```

4.Checking for categorical variables

```
#checking the categorical feature
discrete_feature=[feature for feature in df.columns if feature not in numerical_feature]
```

```
df[discrete_feature]
```

	Brand	fueltype	model	transmission	variant
0	Audi	Diesel	A4	Automatic	35 TDI Premium + Sunroof
1	Audi	Diesel	A6	0	0
2	Audi	Diesel	Q3	0	0
3	BMW	Diesel	3 Series	0	2.5 GX (Diesel) 8 Seater BS IV
4	BMW	Diesel	3 Series	Manual	2.5 GX (Diesel) 8 Seater
...	...	...	...	...	...
194	Volkswagen	Diesel	Vento	0	0
195	Volkswagen	Diesel	Vento	0	V
196	Volkswagen	Petrol	Ameo	0	2002-2013 SLE BS IV
197	Volkswagen	Petrol	Polo	0	0
198	Volkswagen	Petrol	Polo	0	Others

199 rows × 5 columns

```
print("Count of discrete columns:", len(discrete_feature))
```

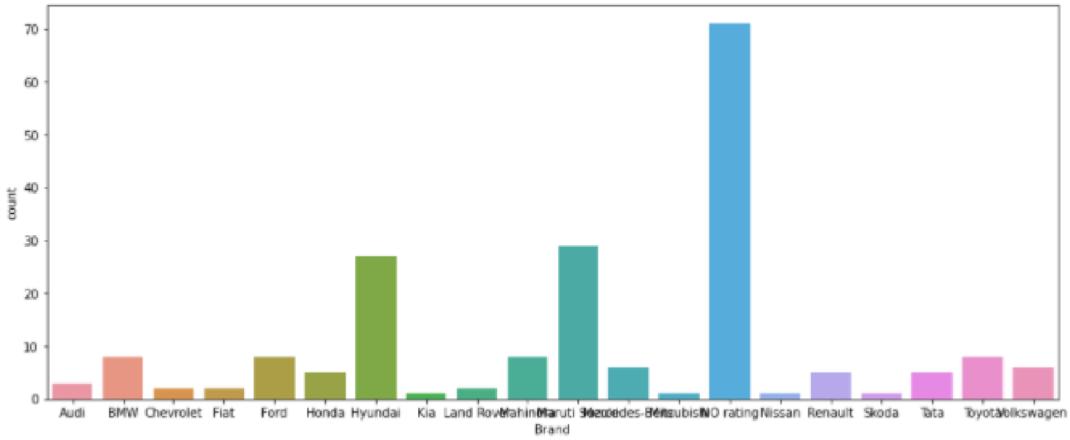
```
Count of discrete columns: 5
```

5.Types of categorical variables

```
plt.figure(figsize=(15,6))
print(df['Brand'].value_counts())
print("-"*70)
sns.countplot(df['Brand'].sort_values())
```

```
NO rating        71
Maruti Suzuki    29
Hyundai          27
Mahindra          8
Ford              8
BMW               8
Toyota            8
Volkswagen        6
Mercedes-Benz     6
Tata              5
Renault           5
Honda             5
Audi              3
Chevrolet         2
Fiat              2
Land Rover        2
Kia               1
Nissan            1
Mitsubishi        1
Skoda             1
Name: Brand, dtype: int64
```

```
<AxesSubplot:xlabel='Brand', ylabel='count'>
```

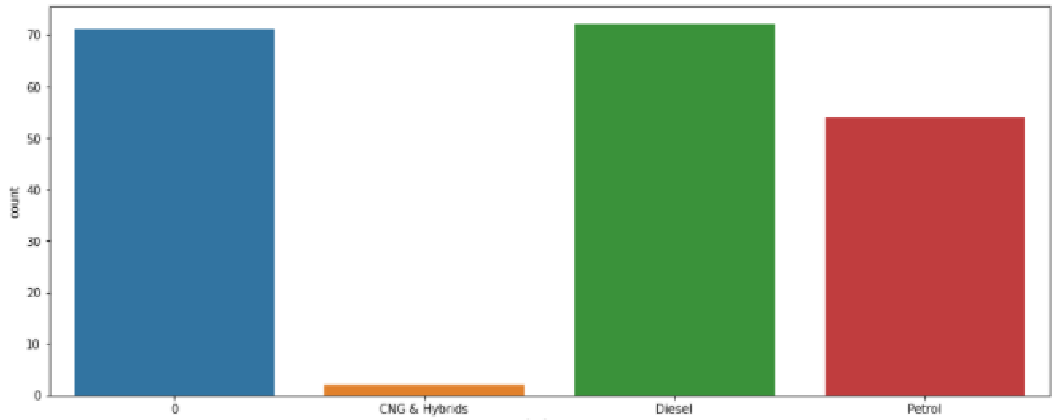



```python
plt.figure(figsize=(15,6))
print(df['fueltype'].value_counts())
print("-"*70)
sns.countplot(df['fueltype'].sort_values())
```

```
Diesel           72
0                71
Petrol           54
CNG & Hybrids     2
Name: fueltype, dtype: int64
----------------------------------------------------------------------
```

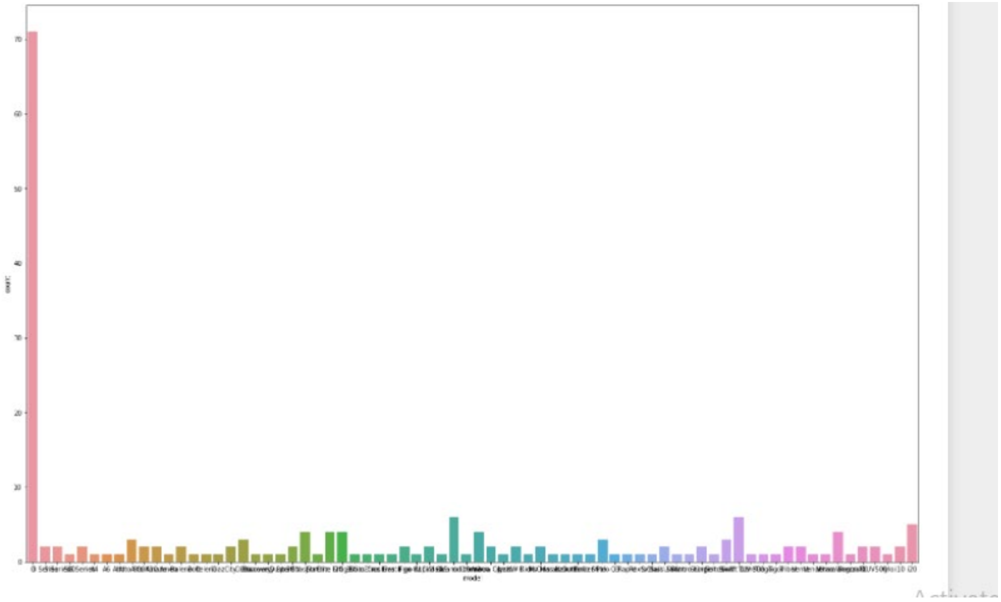
```
c:\users\admin\appdata\local\programs\python\python37\lib\site-packages\seaborn\_decorators.py:43: FutureWarning: Pass the foll
owing variable as a keyword arg: x. From version 0.12, the only valid positional argument will be `data`, and passing other arg
uments without an explicit keyword will result in an error or misinterpretation.
  FutureWarning
```

```
<AxesSubplot:xlabel='fueltype', ylabel='count'>
```




```python
plt.figure(figsize=(25,16))
print(df['model'].value_counts())
print("-"*70)
sns.countplot(df['model'].sort_values())
```

```
0              71
Grand i10       6
Swift Dzire     6
i20             5
Innova          4
               ..
Q3              1
Bolt            1
Venue           1
Tigor           1
Rapid           1
Name: model, Length: 72, dtype: int64
```

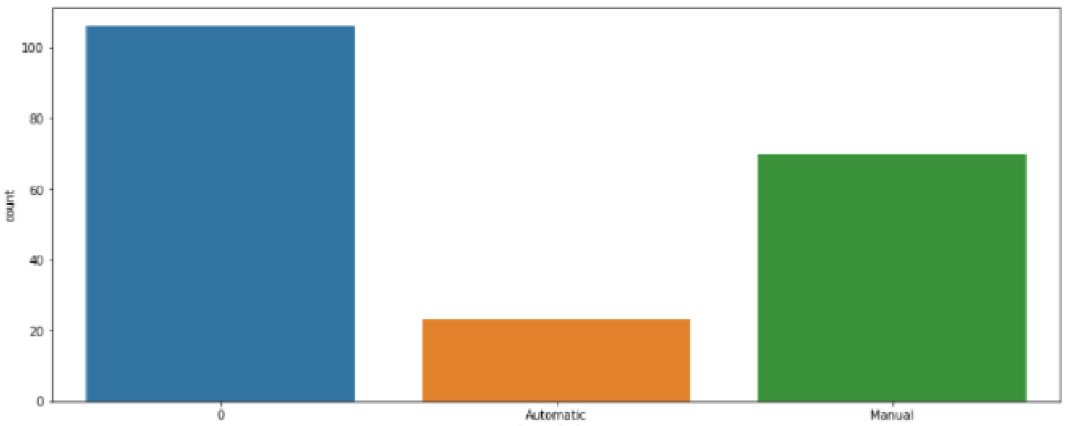


```
plt.figure(figsize=(15,6))
print(df['transmission'].value_counts())
print("-"*70)
sns.countplot(df['transmission'].sort_values())
```

```
0            106
Manual        70
Automatic     23
Name: transmission, dtype: int64
----------------------------------------------------------------------
```

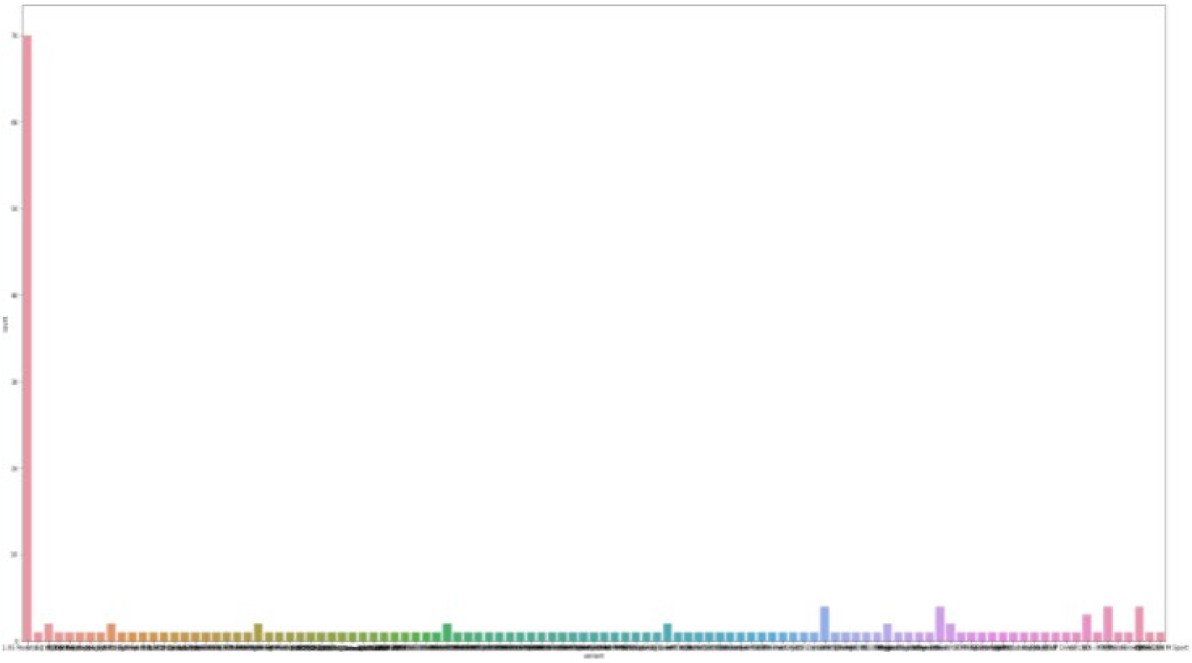
```
c:\users\admin\appdata\local\programs\python\python37\lib\site-packages\seaborn\_decorators.py:43: FutureWarning: Pass the foll
owing variable as a keyword arg: x. From version 0.12, the only valid positional argument will be `data`, and passing other arg
uments without an explicit keyword will result in an error or misinterpretation.
  FutureWarning
```

```
<AxesSubplot:xlabel='transmission', ylabel='count'>
```



```python
plt.figure(figsize=(35,20))
print(df['variant'].value_counts())
print("-"*70)
sns.countplot(df['variant'].sort_values())
```

```
0                          70
Others                      4
VXI                         4
LXI                         4
ZDI                         4
                           ..
2.4 ZX MT                   1
Sportz                      1
Magna Executive 1.2         1
V2 LS                       1
1.2 MPI Highline Plus       1
Name: variant, Length: 109, dtype: int64
```



Algorithm used:-

1.Linear Regression

2.Lasso Regression

3.Random Forest Regression

4.Decision Tree Regression

1.Linear Regression:-

```python
# Loading linear regression model
lin_reg_model=LinearRegression()
```

```python
lin_reg_model.fit(X_train,Y_train)
```

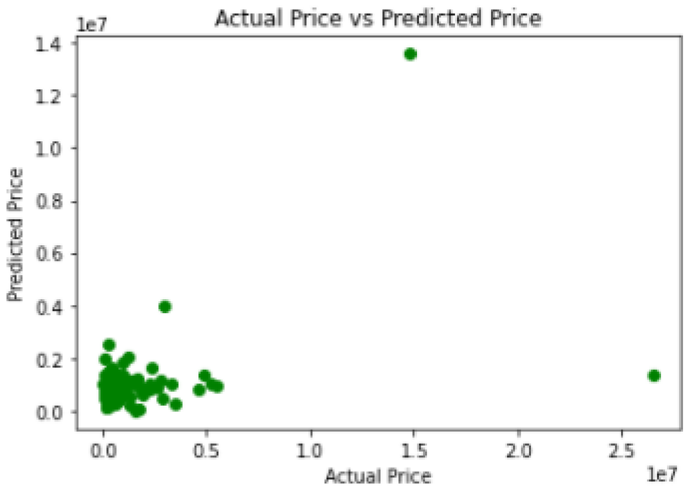
```
LinearRegression()
```

```python
predicted_values=lin_reg_model.predict(X_train)
```

```
# R square error
error_score=metrics.r2_score(Y_train,predicted_values)
print("R square error:",error_score)
```

```
R square error: 0.1912408054332838
```

```
plt.scatter(Y_train,predicted_values,c='g')
plt.xlabel('Actual Price')
plt.ylabel('Predicted Price')
plt.title('Actual Price vs Predicted Price')
plt.show()
```




```
predict=lin_reg_model.predict(X_test)
```

```
predict
```

```
array([1337021.75443433,  838895.63765552,  351463.48434735,
        838925.4380242 , 1024953.59087808, 1025013.19161544,
       1024834.38940336,  122874.66082815, 1337021.75443433,
        838806.23654948,   75270.80665244,   88388.94975967,
        655018.52077439, 1337021.75443433, 1265214.40123334,
        838925.4380242 , 1284791.54138327,  838836.03691816,
       1337021.75443433,  838687.03507476,  838925.4380242 ,
        899032.78165042,  838836.03691816, 1652775.38530487,
        475583.90173019,  892090.0135835 ,  401683.64187023,
        838985.03876156, 1024864.18977204, 1337021.75443433,
        839014.83913024,  599569.36223632, 1092799.66236881,
       1025013.19161544,  271630.77011906,  838895.63765552,
       1337021.75443433, 1024983.39124676, 1337021.75443433,
       1024744.98829732])
```

```
lasso_reg_model=Lasso()
lasso_reg_model.fit(X_train,Y_train)
```

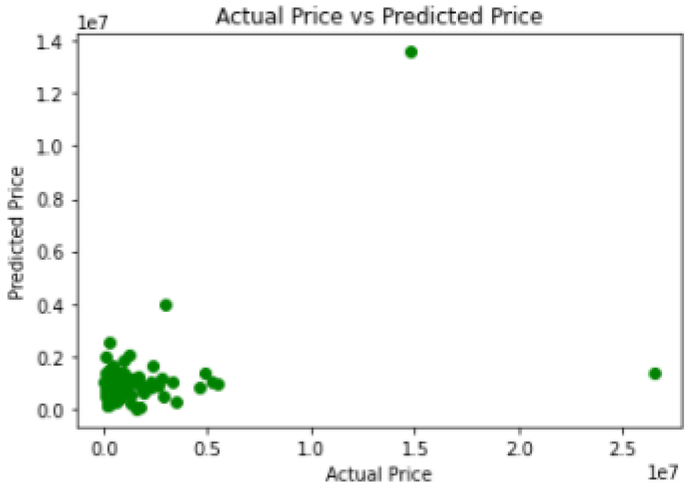
```
Lasso()
```

```python
predicted_values1=lasso_reg_model.predict(X_train)
```

```python
# R square error
error_score2=metrics.r2_score(Y_train,predicted_values1)
print("R square error:",error_score2)
```

```
R square error: 0.1912408054327901
```

```python
plt.scatter(Y_train,predicted_values1,c='g')
plt.xlabel('Actual Price')
plt.ylabel('Predicted Price')
plt.title('Actual Price vs Predicted Price')
plt.show()
```



```
predicted_values1
```

```
array([ 9.81428692e+05,  4.95386213e+05,  1.33701937e+06,  1.33701937e+06,
        1.33701937e+06,  1.20521055e+06,  1.02492271e+06,  1.98045487e+06,
        9.01506598e+05,  8.98673826e+05,  1.24517160e+06,  1.02507171e+06,
        8.50887492e+05,  8.38896135e+05,  8.98971821e+05,  1.02471411e+06,
        6.67604327e+05,  8.39015333e+05,  6.01461777e+05,  8.38866336e+05,
        1.10717584e+06,  9.31028523e+05,  1.14480589e+06,  1.02495251e+06,
        1.02492271e+06,  6.14863829e+05,  6.02660608e+05,  1.33701937e+06,
        1.33701937e+06,  1.35914315e+07,  6.73791272e+05,  9.21549763e+05,
        2.52798385e+06,  4.31508137e+05,  8.38985533e+05,  1.33701937e+06,
        1.09291913e+06,  6.61518620e+05,  8.49619878e+05,  5.51301880e+05,
        7.68163280e+05,  8.38836536e+05,  7.41575970e+05,  7.77468297e+05,
        8.99061219e+05,  9.18880870e+05,  5.63112789e+05,  5.18214924e+05,
        4.19278385e+05,  1.33701937e+06,  1.02492271e+06,  1.02544848e+06,
        8.41627669e+05,  1.48893399e+06,  8.38985533e+05,  8.38955734e+05,
        1.40907454e+06,  8.38657739e+05,  8.38866336e+05,  8.38955734e+05,
        1.33701937e+06,  1.33701937e+06,  1.15326119e+06,  8.89580925e+05,
        1.02510151e+06,  1.08938616e+06,  1.02495251e+06,  1.02510151e+06,
        4.11893086e+05,  7.53713365e+05,  1.02504191e+06,  8.38687539e+05,
        1.33701937e+06,  1.02504191e+06,  8.38925934e+05,  1.05533935e+06,
        1.13216846e+06,  6.81783481e+05,  8.38836536e+05,  3.61784010e+05,
        1.08499839e+06,  2.80785823e+05,  8.38776937e+05,  8.38866336e+05,
        5.53418239e+05,  1.61287588e+06,  1.06934300e+06,  5.51063484e+05,
        1.02504191e+06,  8.38896135e+05,  3.95053448e+06,  8.38985533e+05,
        8.38955734e+05,  5.97491546e+05,  1.84217781e+06,  8.38836536e+05,
        8.38866336e+05,  1.02498231e+06,  1.12528846e+06,  1.67735428e+05,
        9.82714141e+05,  1.02501211e+06,  8.38866336e+05,  8.38776937e+05,
        2.95670377e+05,  3.93633650e+05,  1.33701937e+06,  1.33701937e+06,
        8.38985533e+05,  1.33701937e+06,  1.63285640e+06,  7.37728947e+05,
        9.05565344e+05,  1.08517719e+06,  1.08532741e+06,  1.02504191e+06,
        1.21686102e+06,  1.07860289e+06,  1.42299027e+06,  5.83321508e+04,
        1.08487920e+06,  1.02486311e+06,  1.02501211e+06,  1.33701937e+06,
        1.21471486e+06,  1.77933395e+05,  1.02495251e+06,  7.70995197e+05,
        5.03378423e+05,  3.51556244e+05,  8.39015333e+05,  8.31118607e+05,
        1.02507171e+06,  1.33701937e+06,  2.25707855e+05,  1.33701937e+06,
        1.01782392e+06,  1.18158269e+05,  8.39045132e+05,  2.08841233e+06,
        8.38836536e+05,  1.33701937e+06,  1.33701937e+06,  8.65604297e+05,
       -7.11699534e+01,  1.25251697e+05,  1.02480351e+06,  8.38836536e+05,
        8.38955734e+05,  1.02507171e+06,  9.30852694e+05,  8.38836536e+05,
        1.33701937e+06,  1.31603408e+06,  1.02495251e+06,  2.49863280e+05,
        8.38866336e+05,  8.38955734e+05,  4.81978246e+05])
```

```
# checking mean square error, RMSE

print("Mean square error",mean_squared_error(Y_test,predict))
print("RMSE",np.sqrt(mean_squared_error(Y_test,predict)))
```

```
Mean square error 630378656171.4229
RMSE 793963.8884555284
```

```
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_absolute_error, mean_squared_error,r2_score
from sklearn.model_selection import RandomizedSearchCV, train_test_split
```

```
# Implementing random forest regressor

#calling a object
rf=RandomForestRegressor()
#model fitting
rf.fit(X_train,Y_train)
#predicting the model
y_pred=rf.predict(X_test)
```

```
print("Train score",rf.score(X_train,Y_train))
```

```
Train score 0.26456305166898675
```

```
print("Mean square error",mean_squared_error(Y_test,y_pred))
print("RMSE",np.sqrt(mean_squared_error(Y_test,y_pred)))
```

```
Mean square error 962151488171.7551
RMSE 980893.2093616283
```

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

# create a regressor object
regressor = DecisionTreeRegressor()

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

```
DecisionTreeRegressor()
```

```
# predicting a new value
y_pred = regressor.predict(X_test)
```

```
# checking MSE and RMSE

print("Mean square error",mean_squared_error(Y_test,y_pred))
print("RMSE",np.sqrt(mean_squared_error(Y_test,y_pred)))
```

```
Mean square error 306531212443.76025
RMSE 553652.6098951943
```

1.N_estimators: The number of decision trees being built in the forest. Default values in sklearn are 100. N_estimators are mostly correlated to the size of data, to encapsulate the trends in the data, more number of DTs are needed.

2.Max_depth: The maximum levels allowed in a decision tree. If set to nothing, The decision tree will keep on splitting until purity is reached

3.Max_features: Maximum number of features used for a node split process. Types: sqrt, log2. If total features are n_features then: sqrt(n_features) or log2(n_features) can be selected as max features for node splitting

4.Min_samples_split: This parameter decides the minimum number of samples required to split an internal node. Default value =2. The problem with such a small value is that the condition is checked on the terminal node. If the data points in the node exceed the value 2, then further splitting takes place. Whereas if a more lenient value like 6 is set, then the splitting will stop early and the decision tree wont overfit on the data.

5.Min_sample_leaf: This parameter sets the minimum number of data point requirements in a node of the decision tree. It affects the terminal node and basically helps in controlling the depth of the tree. If after a split the data points in a node goes under the min_sample_leaf number, the split won't go through and will be stopped at the parent node.

```
#Randomizedsearchcv

random_parameters={'n_estimators':[int(x) for x in np.linspace(100,400,num=12)],
                   'max_features':['auto','sqrt','log2'],
                   'max_depth':[int(x) for x in np.linspace(5,30,num=6)],
                   'min_samples_split':[2,5,10,15,100],
                   'min_samples_leaf':[1,2,5,10]}
```

```
random_rf=RandomizedSearchCV(estimator=rf,param_distributions=random_parameters,n_iter=10,scoring="neg_mean_squared_error",
                            cv=10,verbose=2,random_state=42,n_jobs=1)
random_rf.fit(X_train,Y_train)
```

```
random_rf.best_params_
```

```
{'n_estimators': 127,
 'min_samples_split': 100,
 'min_samples_leaf': 1,
 'max_features': 'auto',
 'max_depth': 10}
```

```
pred_y=random_rf.predict(X_test)
pred_y
```

```
array([1079134.07430144,  930250.55388736,  862257.59595561,
        930250.55388736,  963977.46522513,  963977.46522513,
        953836.36085038,  919186.80626237, 1079134.07430144,
        920109.44951261,  868158.67801555,  910578.43190793,
        915121.80142748, 1079134.07430144, 1074146.42621632,
        930250.55388736,  969041.11815618,  920109.44951261,
       1079134.07430144,  920109.44951261,  930250.55388736,
        938858.9282418 ,  920109.44951261, 1084197.7272325 ,
       1025735.04098425, 1014744.24122039,  899587.63214408,
        938858.9282418 ,  953836.36085038, 1079134.07430144,
        938858.9282418 ,  925262.90580223,  899497.82878551,
        963977.46522513,  865860.7208063 ,  930250.55388736,
       1079134.07430144,  963977.46522513, 1079134.07430144,
        953836.36085038])
```

```
print("Mean square error",mean_squared_error(Y_test,pred_y))
print("RMSE",np.sqrt(mean_squared_error(Y_test,pred_y)))
```

```
Mean square error 549386585946.4761
RMSE 741206.1696629866
```

```
parameters={"splitter":["best","random"],
            "max_depth" : [1,3,5,7,9,11,12],
           "min_samples_leaf":[1,2,3,4,5,6,7,8,9,10],
           "min_weight_fraction_leaf":[0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9],
           "max_features":["auto","log2","sqrt",None],
           "max_leaf_nodes":[None,10,20,30,40,50,60,70,80,90] }
```

```
from sklearn.model_selection import GridSearchCV
```

max_features: int, float, string or None, optional (default=None)

The number of features to consider when looking for the best split:

If int, then consider max_features features at each split.

If float, then max_features is a fraction and int(max_features * n_features) features are considered at each split.

If “auto”, then max_features=sqrt(n_features).

If “sqrt”, then max_features=sqrt(n_features).

If “log2”, then max_features=log2(n_features).

If None, then max_features=n_features.

splitter: string, optional (default=”best”)

The strategy used to choose the split at each node. Supported strategies are “best” to choose the best split and “random” to choose the best random split.

max_depth: int or None, optional (default=None)

The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split: int, float, optional (default=2)

The minimum number of samples required to split an internal node:

If int, then consider min_samples_split as the minimum number.

If float, then min_samples_split is a fraction and ceil(min_samples_split * n_samples) are the minimum number of samples for each split.

min_samples_leaf: int, float, optional (default=1)

The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least min_samples_leaf training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

If int, then consider min_samples_leaf as the minimum number.

If float, then min_samples_leaf is a fraction and ceil(min_samples_leaf * n_samples) are the minimum number of samples for each node.

```
tuning_model=GridSearchCV(regressor,param_grid=parameters,scoring='neg_mean_squared_error',cv=3,verbose=3)
```

```
regressor.fit(X_train,Y_train)
```

```
DecisionTreeRegressor()
```

```
regressor.score(X_train,Y_train)
```

```
0.30034695214732665
```

```
tuned_pred=regressor.predict(X_test)
```

```
tuned_pred
```

```
array([1959752.47368421,  475000.        ,  140999.        ,
        340000.        ,  717500.        , 1548000.        ,
       1100000.        ,  841577.66666667, 1959752.47368421,
        670500.        ,  161000.        , 1750000.        ,
       1350000.        , 1959752.47368421,  285000.        ,
        340000.        ,  565000.        ,  366500.        ,
       1959752.47368421,  125000.        ,  340000.        ,
       1709999.        ,  366500.        ,  230000.        ,
        100000.        , 1100000.        ,  550000.        ,
       1117750.        ,  380000.        , 1959752.47368421,
       1197500.        ,  565000.        ,  375000.        ,
       1548000.        ,  311000.        ,  475000.        ,
       1959752.47368421,  215000.        , 1959752.47368421,
        560000.        ])
```

```
print("Mean square error",mean_squared_error(Y_test,tuned_pred))
print("RMSE",np.sqrt(mean_squared_error(Y_test,tuned_pred)))
```

```
Mean square error 989538414315.5891
RMSE 994755.4545291968
```

Conclusion

The best fit model is Random Forest which has less mean Squared error

future scope:- here we have only less features and the large amount of data is not available so the accuracy is low