FLIP ROBO

Car Price Prediction Project

Submitted By:
Khushboo Khatri

Acknowledgement

I would like to appreciate FlipRobo to provide us the opportunity to work on this project, our SME Shwetank Mishra to help me whenever I needed.

I won’t also like to extend my thanks to Datatrained academy to taught us the basic required to complete these projects.

I would also like to acknowledge that the data which is used here is collected from car24 website.

Introduction:

With the covid 19 impact in the market, we have seen lot of changes in the car market. Now some cars are in demand hence making them costly and some are not in demand hence cheaper. One of our clients works with small traders, who sell used cars. With the change in market due to covid 19 impact, our client is facing problems with their previous car price valuation machine learning models. So, they are looking for new machine learning models from new data. We have to make car price valuation model. This project contains two phase-

Data Collection Phase

Here we scrape used cars data from car24.com, by selecting Delhi NCR location. We did not put any filter for car type or budget, so that we can include all types of cars for example: SUV, Sedan, Coupe, minivan, Hatchback.

The number of columns for data doesn’t have limit. However, we choose to have these columns: Brand, model, variant, manufacturing year, driven kilometre’s, fuel, number of owners, Transmission and at last target variable Price of the car.

For data collection we use selenium tool in python. We use car24/Delhi, and scrape around 5660 cars details. We made 9 columns excel sheet out of it, which we use further in Model Building.

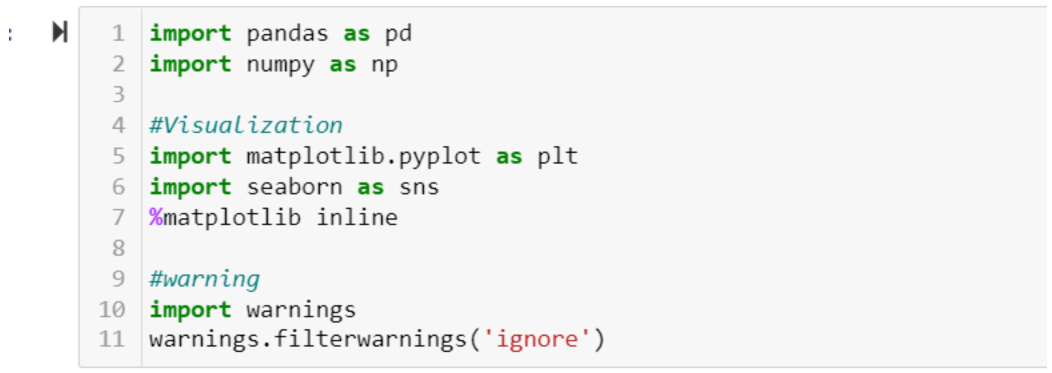
Model Building Phase

After data collection, we start to build our ML model. Before model building, we perform all the required data pre-processing steps.

Let’s start how we approach the Model Building Phase:

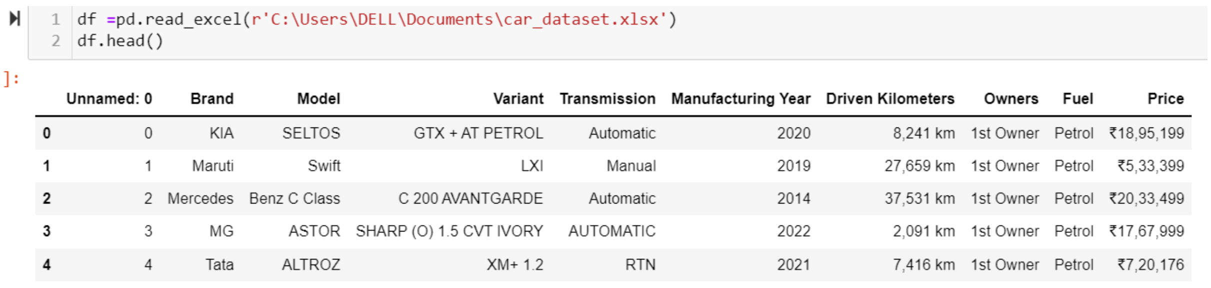
Data Source and their Format:

The data which we collected from car24.com, we saved it in excel form. Now, for model building we need to get then saved data. For that first we need to import basic library in jupyter notebook.



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

#Visualization
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

#warning
import warnings
warnings.filterwarnings('ignore')
```

Once we import all the basic libraries , it time to get the data. As our data is in form of excel sheet, we use pd.read_excel () to fetch the data.



```python
df =pd.read_excel(r'C:\Users\DELL\Documents\car_dataset.xlsx')
df.head()
```

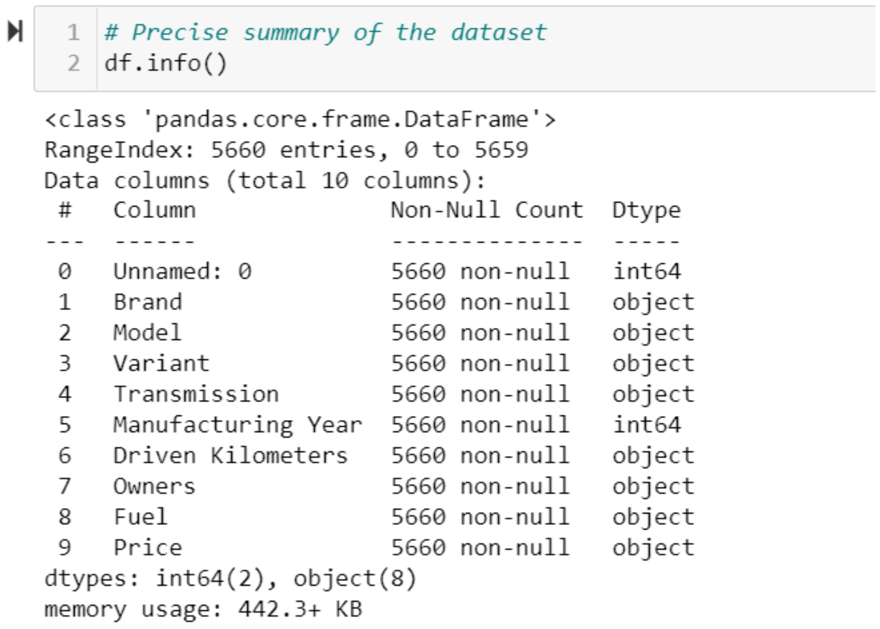
]:

	Unnamed: 0	Brand	Model	Variant	Transmission	Manufacturing Year	Driven Kilometers	Owners	Fuel	Price
0	0	KIA	SELTOS	GTX + AT PETROL	Automatic	2020	8,241 km	1st Owner	Petrol	₹18,95,199
1	1	Maruti	Swift	LXI	Manual	2019	27,659 km	1st Owner	Petrol	₹5,33,399
2	2	Mercedes	Benz C Class	C 200 AVANTGARDE	Automatic	2014	37,531 km	1st Owner	Petrol	₹20,33,499
3	3	MG	ASTOR	SHARP (O) 1.5 CVT IVORY	AUTOMATIC	2022	2,091 km	1st Owner	Petrol	₹17,67,999
4	4	Tata	ALTROZ	XM+ 1.2	RTN	2021	7,416 km	1st Owner	Petrol	₹7,20,176

From here we can Observe that our dataset consists of 10 columns, out of which column Unnamed: 0 is indexing columns, which will be of no use in our machine learning model building purpose. So, we can drop it.

Other columns are Brand, Model, Variant, Transmission gives us complete detail about type of car and its brand. Manufacturing Year denotes the year in which the car has been built or manufactured. Driven Kilometers define, the kilometre's driven by the car so far. Owners denotes number of owners the car had. Fuel denotes the type of fuel used by the car. Price which is our target column, denotes the price of the car.

Let's move ahead and get the precise summary of our data set. For that we use info().



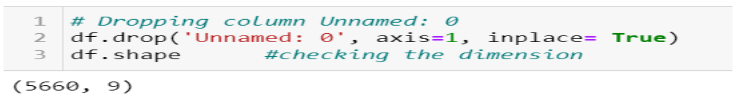
```python
# Precise summary of the dataset
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5660 entries, 0 to 5659
Data columns (total 10 columns):
 #   Column              Non-Null Count  Dtype 
---  ------              --------------  ----- 
 0   Unnamed: 0          5660 non-null   int64 
 1   Brand               5660 non-null   object
 2   Model               5660 non-null   object
 3   Variant             5660 non-null   object
 4   Transmission        5660 non-null   object
 5   Manufacturing Year  5660 non-null   int64 
 6   Driven Kilometers   5660 non-null   object
 7   Owners              5660 non-null   object
 8   Fuel                5660 non-null   object
 9   Price               5660 non-null   object
dtypes: int64(2), object(8)
memory usage: 442.3+ KB
```

We can observe that , in our dataset we don't have any null values. The total number of rows are 5660 & columns are 10. Except Unnamed: 0 and Manufacturing Year all other columns are of object type, these two are of integer type columns.

Price and Driven Kilometers are two columns which has informative numerical values which we need to fetch as it is removing the km from Driven Kilometers column and ₹ from Price column.

For remaining columns, we need to do some kind of encoding or mapping to fit them in the machine learning model. Let's drop Unnamed:0 column.



```python
# Dropping column Unnamed: 0
df.drop('Unnamed: 0', axis=1, inplace= True)
df.shape          #checking the dimension
```

```
(5660, 9)
```

Dropping any column is part of Data Cleaning, Let’s go ahead and see how many unique value is contained by each column.

```
df.nunique()
```

```
]: Brand                  18
   Model                  70
   Variant               152
   Transmission            8
   Manufacturing Year     14
   Driven Kilometers     269
   Owners                  3
   Fuel                    3
   Price                 264
   dtype: int64
```

We can see that our nominal data has high number of cardinality, so to encode them we can use either label encoder or Binary encoder. As one hot encoder increases the number of columns to a very great extend and hence affects the efficiency and time consumed and over all cost of production.

Let’s go ahead and do some feature engineering followed by visualization.

Feature Engineering and EDA:

Feature Engineering is a preprocessing technique in machine learning that, create features from raw data. Here in our data set we can see that Driven Kilometers and Price are some thing which we need to featured engineered.

Let’s start with Driven Kilometer, our amin objective to keep the integer value in it as it is, and making it a numerical column by getting rid of categorical part. We don’t want any data loss, here.

```
# Driven Kilometers column
df['Driven Kilometers']=df['Driven Kilometers'].apply(lambda x:x.split(' ')[0])
df['Driven Kilometers'].head()
```

```
]: 0      8,241
   1     27,659
   2     37,531
   3      2,091
   4      7,416
   Name: Driven Kilometers, dtype: object
```

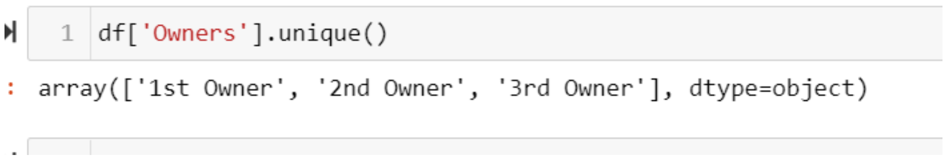
```
for i in range(len(df['Driven Kilometers'])):
    df['Driven Kilometer'][i]= float(df['Driven Kilometers'][i].replace(',',''))
df['Driven Kilometer']
```

```
]: 0        8241.0
   1       27659.0
   2       37531.0
   3        2091.0
   4        7416.0
             ...
   5655    19919.0
   5656    25068.0
   5657    69575.0
   5658      649.0
   5659    16179.0
   Name: Driven Kilometer, Length: 5660, dtype: float64
```

```
df['Driven Kilometers']=df['Driven Kilometer'].astype(int)
```

In the first step we separate km from our numerical data, in the second step as we can see ‘,’ is removed from it in order to convert this string value to numerical. And in the last step we convert the entire column to integer. So that we can use them as it is in our Model building.

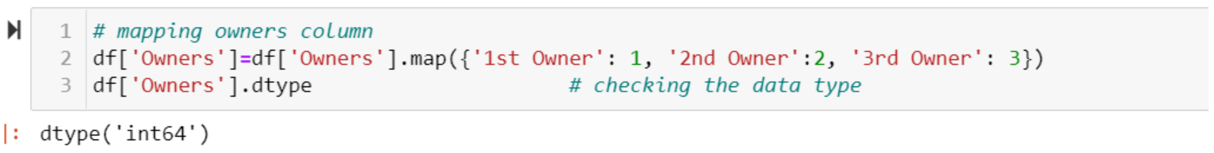
Some thing similar need to be done for Price as well. However, before that let’s have a look at our Owners column.



```
df['Owners'].unique()
```

```
array(['1st Owner', '2nd Owner', '3rd Owner'], dtype=object)
```

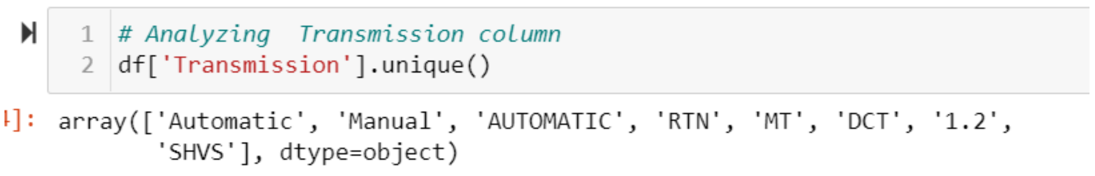
We can observe that instead of number 1 , 2, 3 we got 1st owner, 2nd owner and 3rd owner, we can do mapping and convert them into meaningful numerical column.



```
# mapping owners column
df['Owners']=df['Owners'].map({'1st Owner': 1, '2nd Owner':2, '3rd Owner': 3})
df['Owners'].dtype                          # checking the data type
```

```
dtype('int64')
```

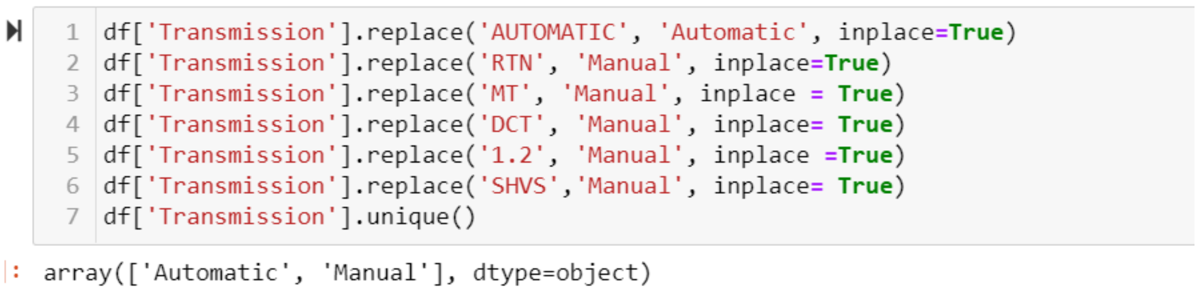
Since, our Owners column is engineered properly let, move ahead with Transmission column



```
# Analyzing  Transmission column
df['Transmission'].unique()
```

```
array(['Automatic', 'Manual', 'AUTOMATIC', 'RTN', 'MT', 'DCT', '1.2',
       'SHVS'], dtype=object)
```

We can observe that, Automatic is written in two different format, we can make it one. Also Transmission of any car can either be manual or Automatic, so rest RTN, MT, DCT, 1.2, SHVS can be replaced to Manual.



```
df['Transmission'].replace('AUTOMATIC', 'Automatic', inplace=True)
df['Transmission'].replace('RTN', 'Manual', inplace=True)
df['Transmission'].replace('MT', 'Manual', inplace = True)
df['Transmission'].replace('DCT', 'Manual', inplace= True)
df['Transmission'].replace('1.2', 'Manual', inplace =True)
df['Transmission'].replace('SHVS','Manual', inplace= True)
df['Transmission'].unique()
```

```
array(['Automatic', 'Manual'], dtype=object)
```

Now our Transmission data looks good and clean. Let’s check out the Brand columns in order to see if we have any duplicates there as well.

```
# Let's check brand columns
df['Brand'].value_counts()
```

```
]: Maruti         2300
   Hyundai         980
   Honda           540
   Ford            300
   Tata            260
   Mahindra        260
   Renault         200
   KIA             160
   MG              160
   Toyota           80
   Audi             80
   Jeep             80
   BMW              60
   Volkswagen       60
   Skoda            40
   Mercedes         40
   Datsun           40
   Nissan           20
   Name: Brand, dtype: int64
```

We can observe that, all the elements are completely unique. Let's move ahead with Price, as weed to perform some feature engineering in Price column as well, we can see that it is Object type column as the numbers there are complied with rupee symbol, we need to keep the Integers as it is as it very valuable data.

```
# Analyzing Price column, as it is also object type, we need to convert it into numerical column.
df['Price']=df['Price'].apply(lambda x : x.split('₹')[1])
```

```
for i in range(len(df['Price'])):
    df['Price'][i]= float(df['Price'][i].replace(',',''))
df['Price']
```

```
3]: 0       1895199.0
    1        533399.0
    2       2033499.0
    3       1767999.0
    4        720176.0
              ...
    5655     566799.0
    5656     398999.0
    5657     381299.0
    5658     731399.0
    5659     431599.0
    Name: Price, Length: 5660, dtype: object
```

```
df['Price']= df['Price'].astype(int)
```

Now our target column is ready, let's perform some visualizations. Let's check the datatype of every column first before proceeding.

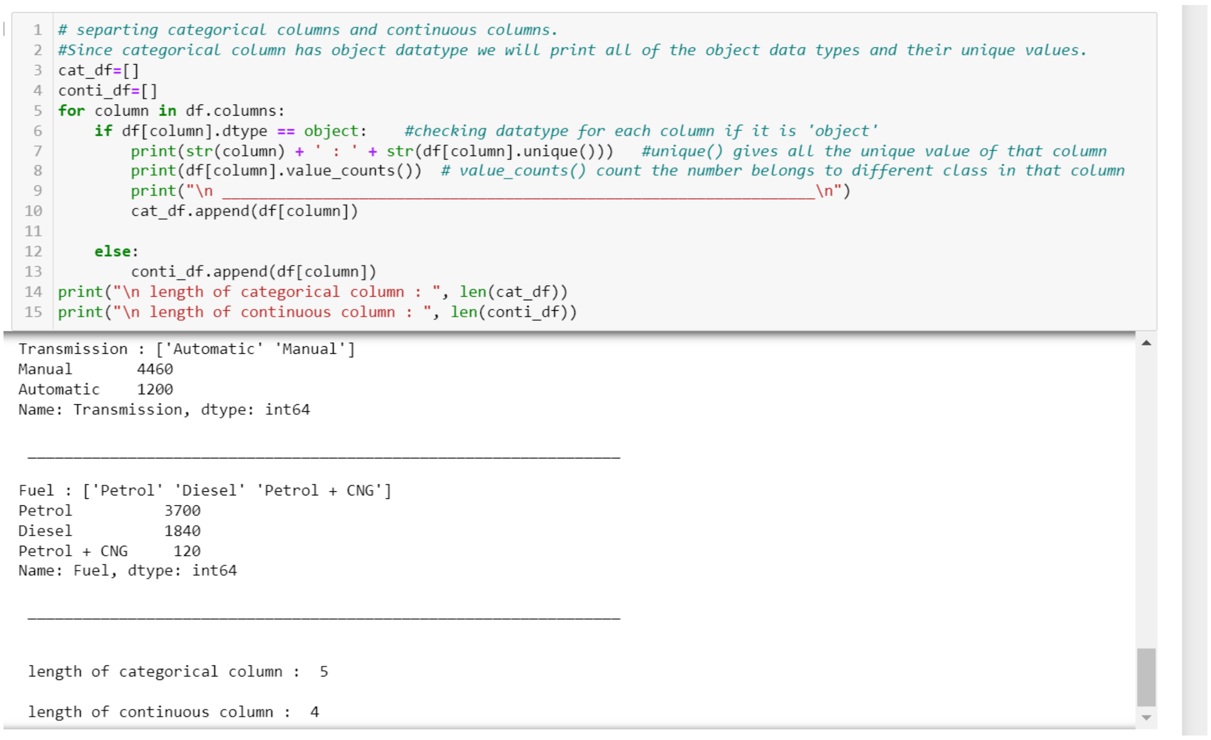
```
# Let's chcek datatype of all the columns
df.dtypes
```

```
]: Brand                      object
   Model                      object
   Variant                    object
   Transmission               object
   Manufacturing Year          int64
   Driven Kilometers           int32
   Owners                      int64
   Fuel                       object
   Price                       int32
   Driven Kilometer          float64
   dtype: object
```

```
# drop Driven Kilometer
df.drop(columns=['Driven Kilometer'], axis=1, inplace=True)
df.shape
```

```
]: (5660, 9)
```

We can observe that Driven Kilometer and Driven Kilometers consist of similar value so we drop one of them. Now, lets go ahead and separate categorical columns and continuous columns.



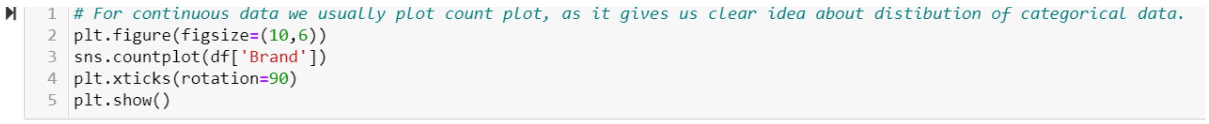
```
# separting categorical columns and continuous columns.
#Since categorical column has object datatype we will print all of the object data types and their unique values.
cat_df=[]
conti_df=[]
for column in df.columns:
    if df[column].dtype == object:    #checking datatype for each column if it is 'object'
        print(str(column) + ' : ' + str(df[column].unique()))   #unique() gives all the unique value of that column
        print(df[column].value_counts())  # value_counts() count the number belongs to different class in that column
        print("\n _________________________________________________________________\n")
        cat_df.append(df[column])

    else:
        conti_df.append(df[column])
print("\n length of categorical column : ", len(cat_df))
print("\n length of continuous column : ", len(conti_df))
```

```
Transmission : ['Automatic' 'Manual']
Manual       4460
Automatic    1200
Name: Transmission, dtype: int64

 _________________________________________________________________


Fuel : ['Petrol' 'Diesel' 'Petrol + CNG']
Petrol          3700
Diesel          1840
Petrol + CNG     120
Name: Fuel, dtype: int64

 _________________________________________________________________


 length of categorical column :  5

 length of continuous column :  4
```

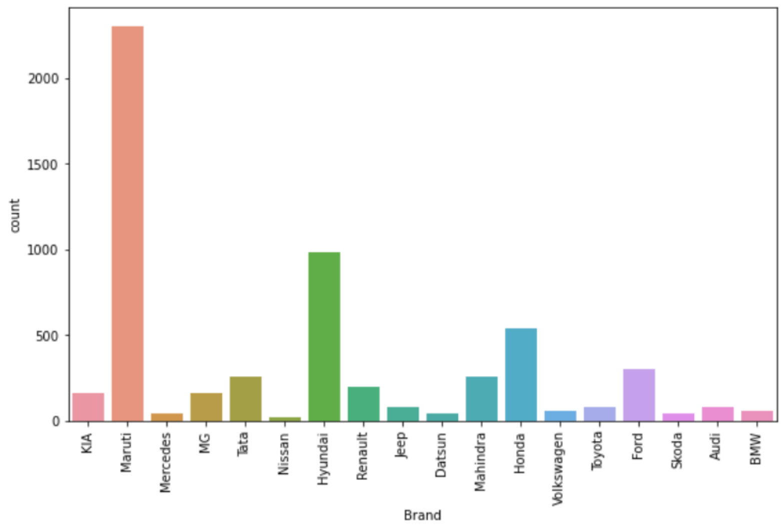
We can observe that we have 5 categorical columns and 4 continuous columns.

And looking at the elements of categorical columns makes me sure that it is nominal data, and when we encode. We will encode accordingly.

Let's begin with Univariate Analysis:

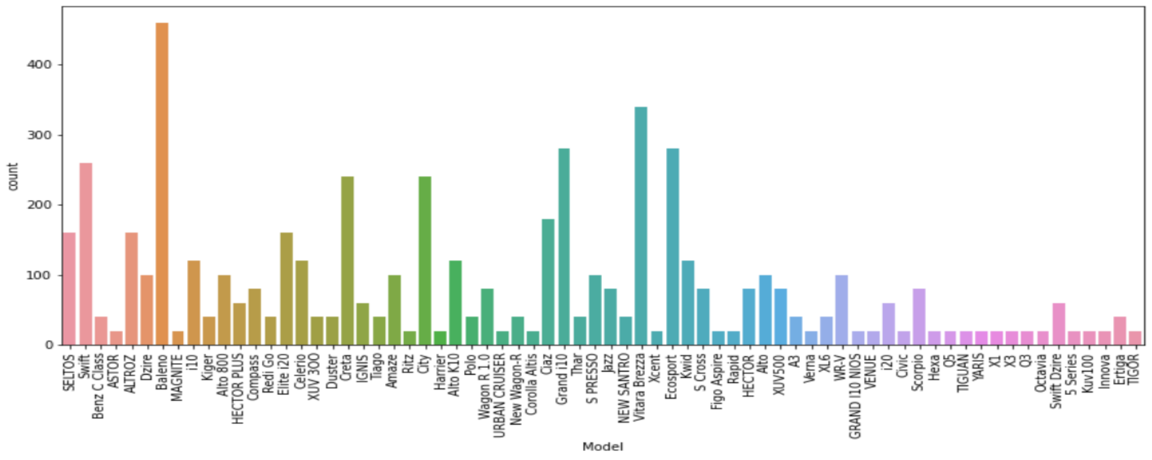


```
# For continuous data we usually plot count plot, as it gives us clear idea about distibution of categorical data.
plt.figure(figsize=(10,6))
sns.countplot(df['Brand'])
plt.xticks(rotation=90)
plt.show()
```



We can observe that Brand Maruti has the highest record for cars while Nissan has lowest.

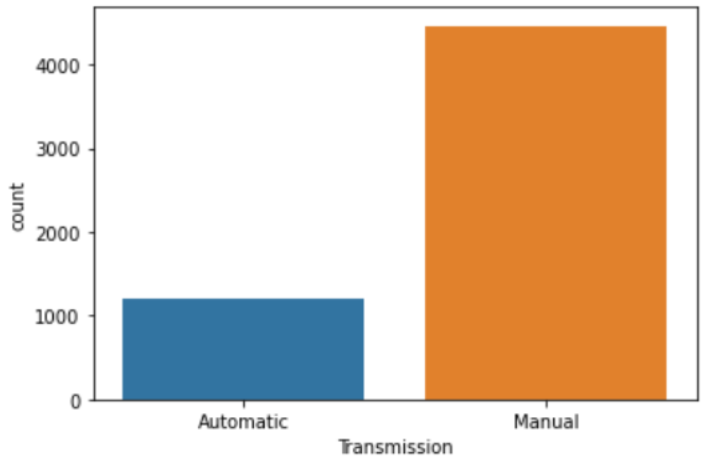
```
plt.figure(figsize=(14,6))
sns.countplot(df['Model'])
plt.xticks(rotation=90)
plt.show()
```



From above observation we can clearly see how Models of various cars present in the marketplace in order to sell.

```
sns.countplot(df['Transmission'])
```

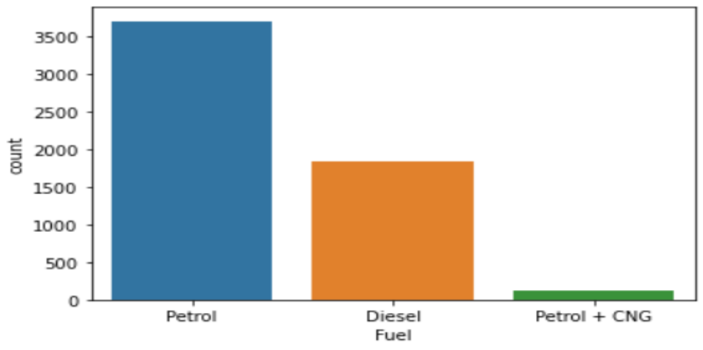
```
5]: <AxesSubplot:xlabel='Transmission', ylabel='count'>
```



There are a greater number of manual cars ten automatic cars available in Indian market.

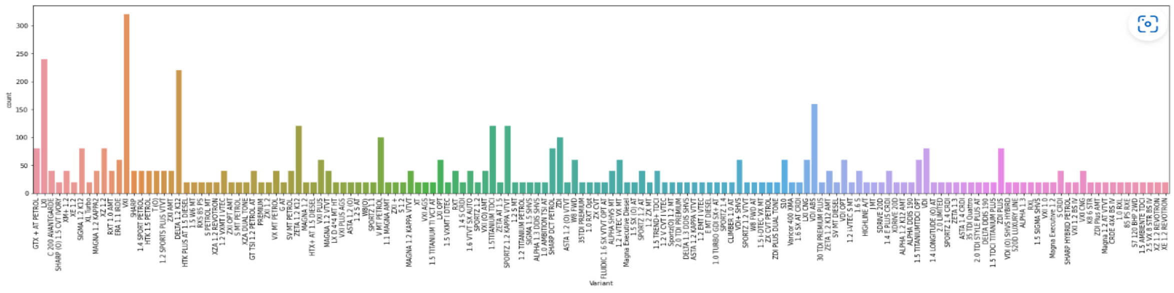
```
sns.countplot(df['Fuel'])
```

```
]: <AxesSubplot:xlabel='Fuel', ylabel='count'>
```



We can observe that more number of Petrol cars are in the market. Even there are few cars which run on Petrol + CNG. I am assuming they would be cabs, as cabs in Delhi mostly runs on Petrol + CNG.

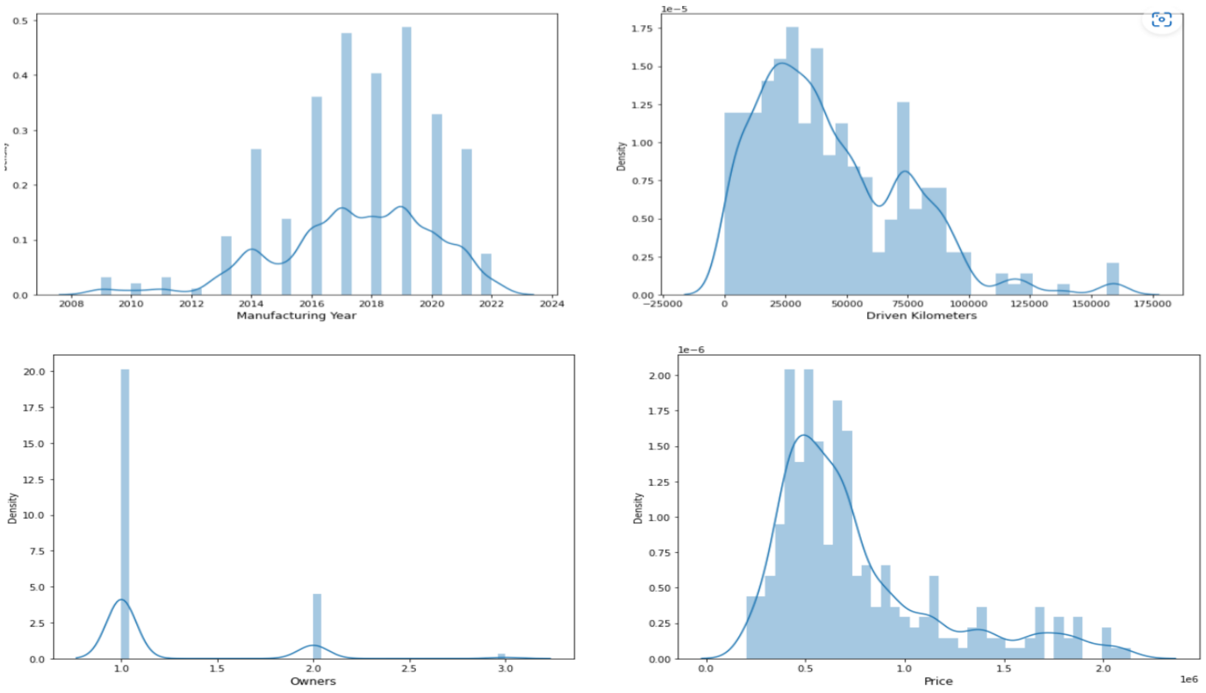
```python
plt.figure(figsize=(30,6))
sns.countplot(df['Variant'])
plt.xticks(rotation=90)
plt.show()
```



We can observe that, there are some variant which are high on the market like VXI model followed by LXI than other variant of car.

As our categorical features are covered let’s plot Distribution plot for continuous variables.

```python
# Let's see distribution of continuous data
conti_feature= ['Manufacturing Year','Driven Kilometers','Owners','Price']
plt.figure(figsize=(20,15))
graph=1
for column in df[conti_feature]:
    if graph<=4:       #4 denotes total number of columns
        plt.subplot(2,2,graph)
        ax=sns.distplot(df[column])
        plt.xlabel(column, fontsize=12)
    graph+=1
plt.show()
```



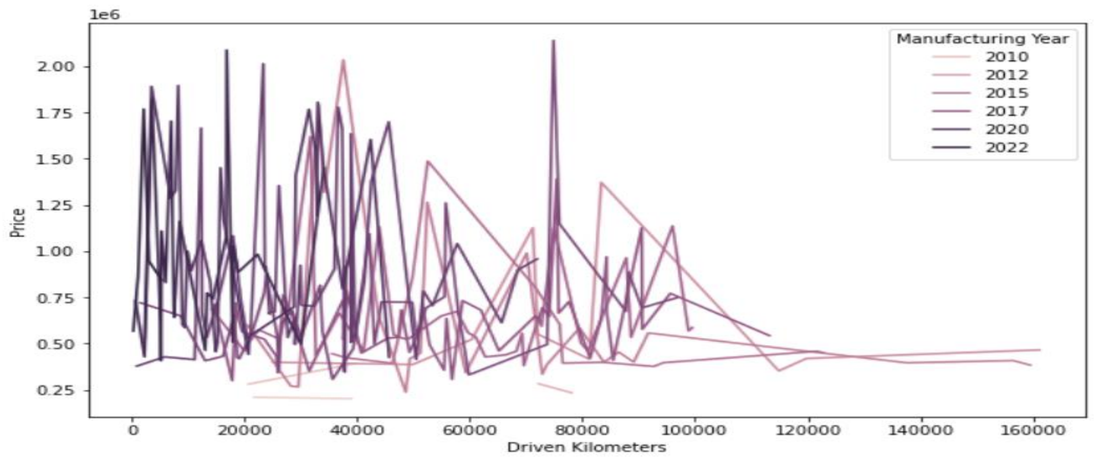
From above observation we can sense that Manufacturing year and Owners are discrete columns, having discrete values. In case of owner number of single owner cars are more while we have least number of 3rd owner car.

Similarly, we observe that cars with manufacturing year 2019 are highest in number, the dataset very less number of cars which are older than 2012.

Driven Kilometers and Price are continuous columns, we can see clear bell shape curve in the distribution pattern.

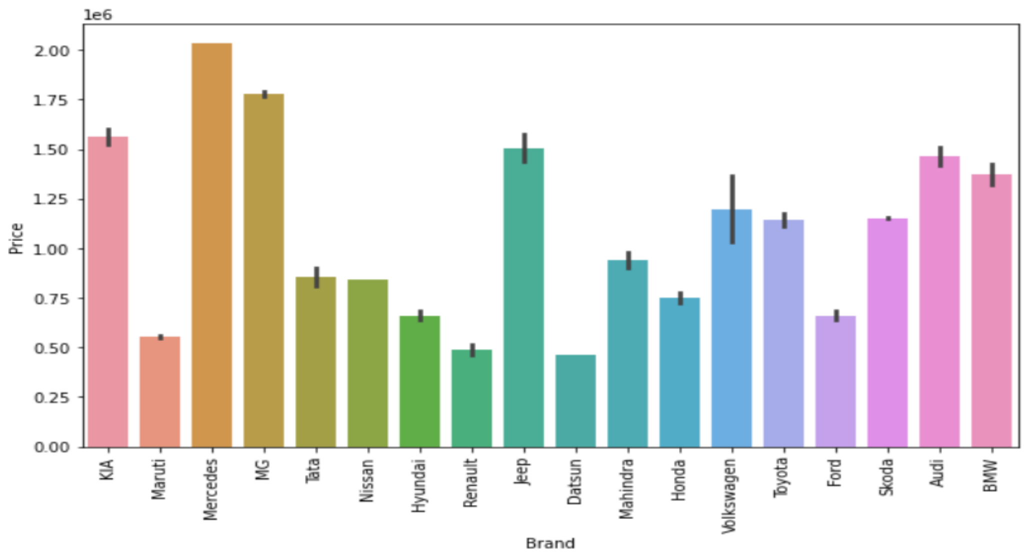
Let's starts with Multivariate Analysis:

```
plt.figure(figsize=(10,6))
sns.lineplot(x='Driven Kilometers', y='Price', data= df, hue= 'Manufacturing Year
plt.show()
```



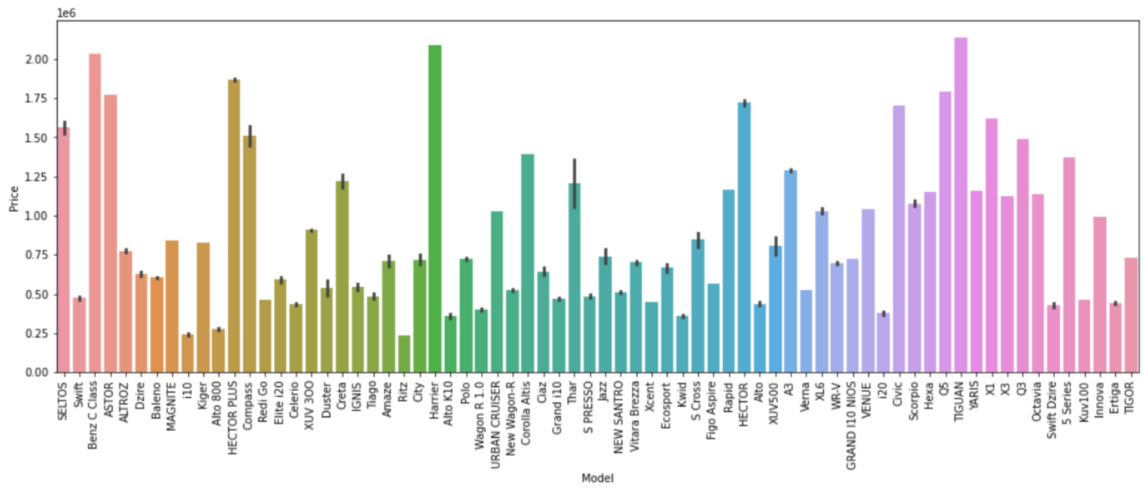
We can observe that as the car gets older the Price decreases. Also, the Price also depends on Brand and other features also, that is the reason we can see both high and low even for newer cars.

```
plt.figure(figsize=(10,6))
sns.barplot(x='Brand', y='Price', data=df)
plt.xticks(rotation= 90)
plt.show()
```



We can observe that Mercedes, MG are comparatively expensive Brands in car, while Datsun, Maruti are cheaper than rest.

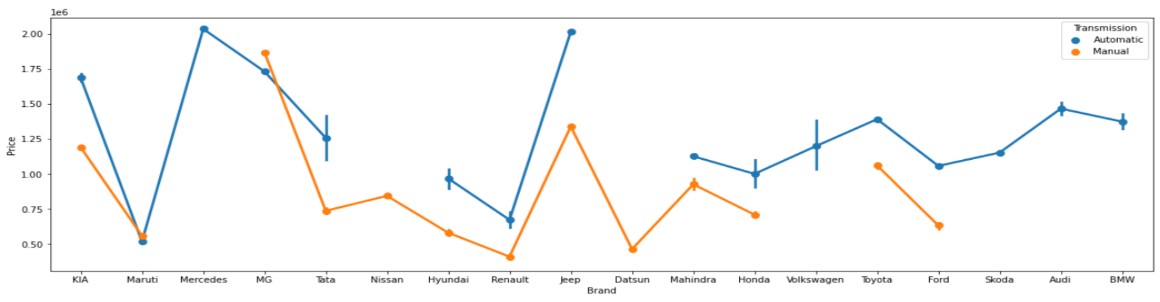
```
plt.figure(figsize=(18,6))
sns.barplot(x='Model', y='Price', data=df)
plt.xticks(rotation= 90)
plt.show()
```



We can observe that Tiguan, Harrier are the expensive cars followed by Benz C class, while Ritz, i10 , auto 800 are compact under lower budget vehicles.

```
plt.figure(figsize=(20,6))
sns.pointplot(x='Brand', y='Price', hue='Transmission', data=df)
```

```
]: <AxesSubplot:xlabel='Brand', ylabel='Price'>
```

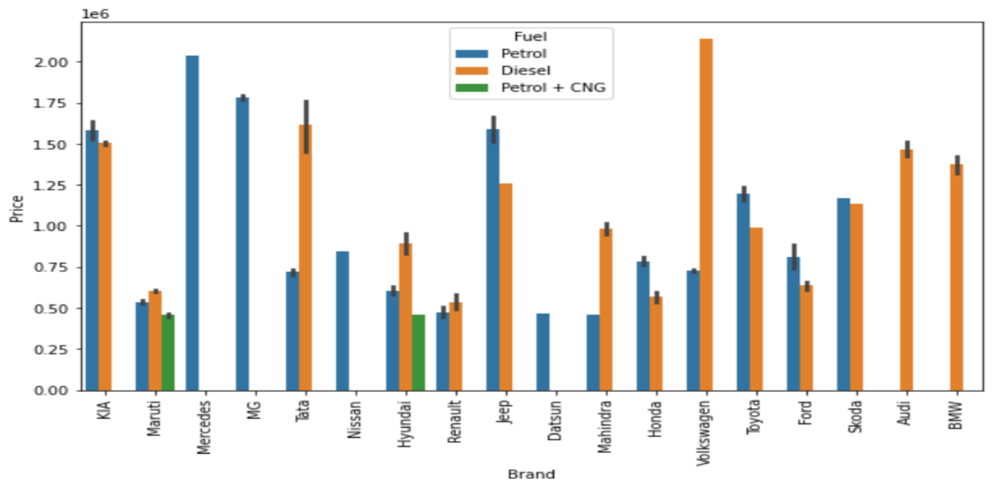


We can observe that automatic vehicles are comparatively expensive then manual transmission.

Lets see now brand is connected with Price and type of Fuel.

```
plt.figure(figsize=(10,6))
sns.barplot('Brand', 'Price', hue= 'Fuel', data= df)
plt.xticks(rotation= 90)
plt.show
```

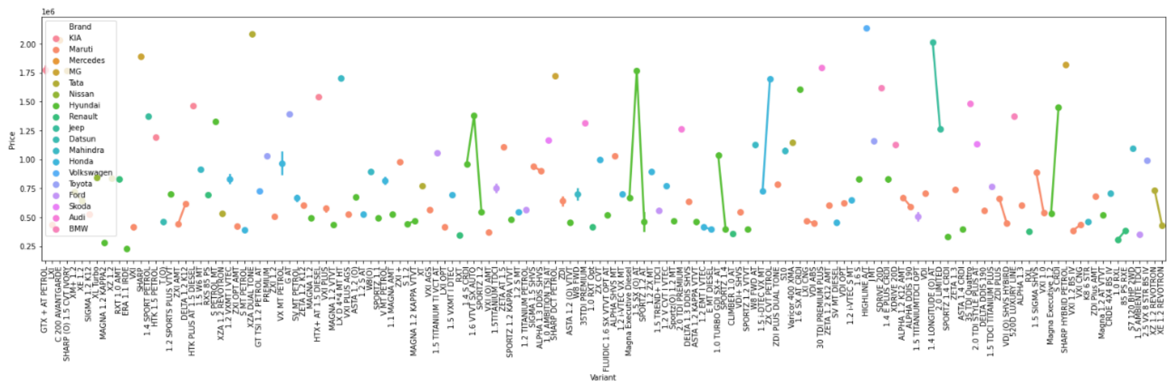
```
]: <function matplotlib.pyplot.show(close=None, block=None)>
```



Maruti and Hyundai are the Brands which has all three types of fuel engine. Audi and BMW has Diesel cars only while Mercedes, MG, Nissan, Datsun Brands has Petrol cars only in our dataset.

The Price point of diesel or petrol cars mostly depends on brand.

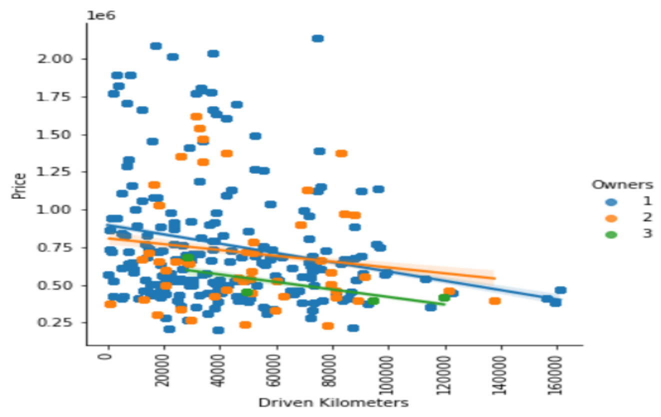
```python
plt.figure(figsize=(27, 6))
sns.pointplot('Variant', 'Price', hue= 'Brand', data=df)
plt.xticks(rotation=90)
plt.show()
```



From here we can observe that variant plays an important role in determining the price of car. Brands themselves consist of certain variant in cars which are expensive.

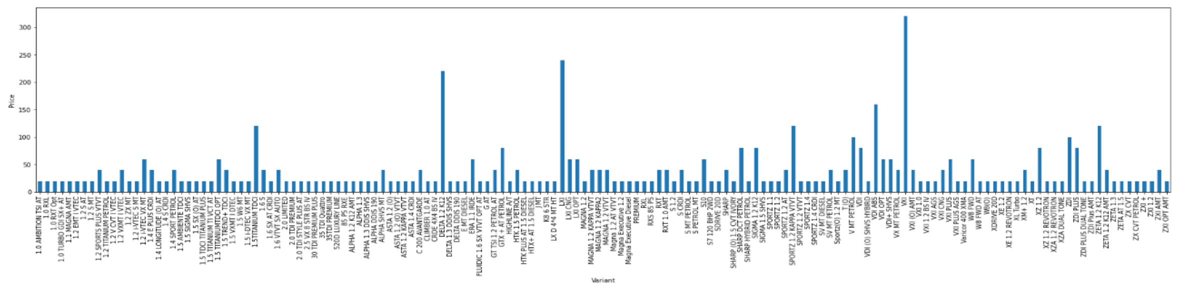
```python
plt.figure(figsize=(18,6))
sns.lmplot(data=df, x='Driven Kilometers', y= 'Price', hue= 'Owners', fit_reg= True)
plt.xticks(rotation=90)
plt.show()
```

```
<Figure size 1296x432 with 0 Axes>
```



We can observe that the Price point for 3rd owner is always low. Less driven single owner cars get good sales Price

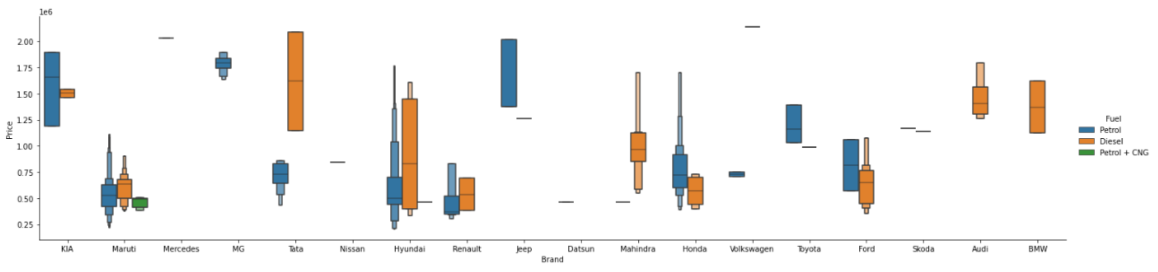
```python
plt.figure(figsize=(30,6))
df.groupby('Variant').Price.count().plot.bar(ylim=0)
plt.ylabel('Price')
plt.show()
```



This plot helps us to get a clear view of how the car price varies with the variant of the car, for example : VXI models are higher in sales data..

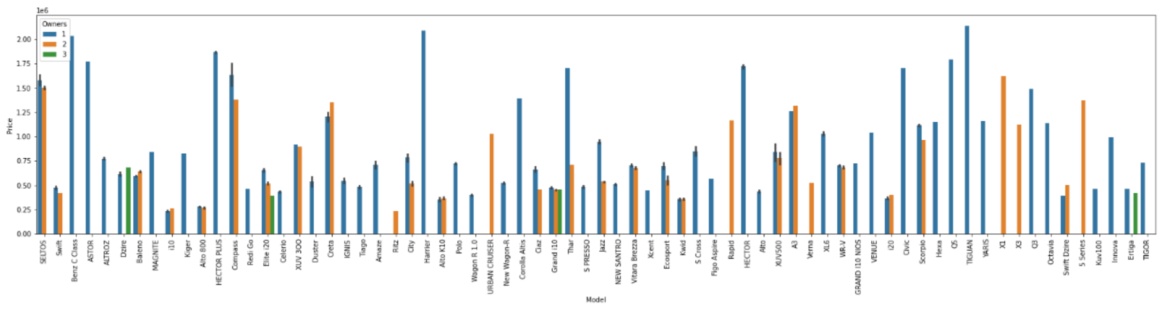
```
sns.catplot(x='Brand', y='Price', data=df, hue='Fuel', kind='boxen', height=5, aspect=4)
```

```
]: <seaborn.axisgrid.FacetGrid at 0x229fb4a11f0>
```



From above plot, we tried to see the mean Price value of each brand, depending on their Fuel. For example : Mercedes and Volkswagen cars has higher means for Price.

```
plt.figure(figsize=(30,6))
sns.barplot(x='Model',y='Price',data=df,hue='Owners')
plt.xticks(rotation=90)
plt.show()
```



We can observe that compact cars like Dzire, Elite i20, Grand i10, Ertiga offen have 3rd owner. As these normally used for cab purpose or purchased by new driver, in order to learn driving.

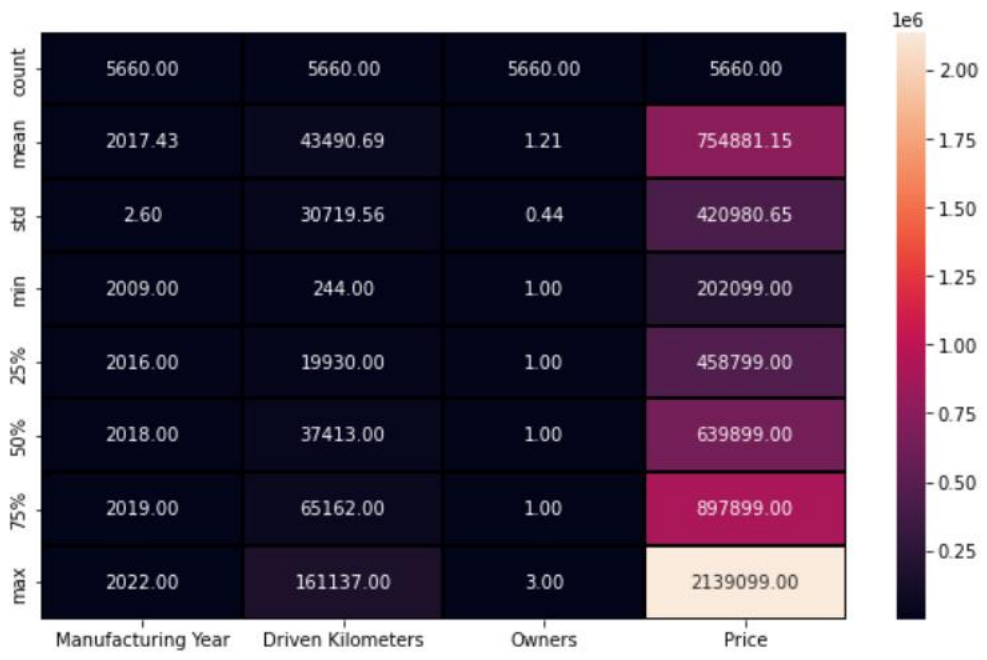
Statistical Analysis of Dataset:

```
# Statistical analysis of the Dataset
df.describe()
```

]:

	Manufacturing Year	Driven Kilometers	Owners	Price
count	5660.000000	5660.000000	5660.000000	5.660000e+03
mean	2017.431095	43490.685512	1.208481	7.548812e+05
std	2.603168	30719.562442	0.439681	4.209806e+05
min	2009.000000	244.000000	1.000000	2.020990e+05
25%	2016.000000	19930.000000	1.000000	4.587990e+05
50%	2018.000000	37413.000000	1.000000	6.398990e+05
75%	2019.000000	65162.000000	1.000000	8.978990e+05
max	2022.000000	161137.000000	3.000000	2.139099e+06

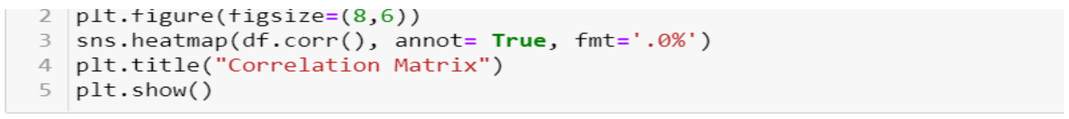
We can observe that the price of cars is in lakhs. Which is represent in this way. Statistical description basically plots the table for continuous columns. Let's visualize the entire summary.



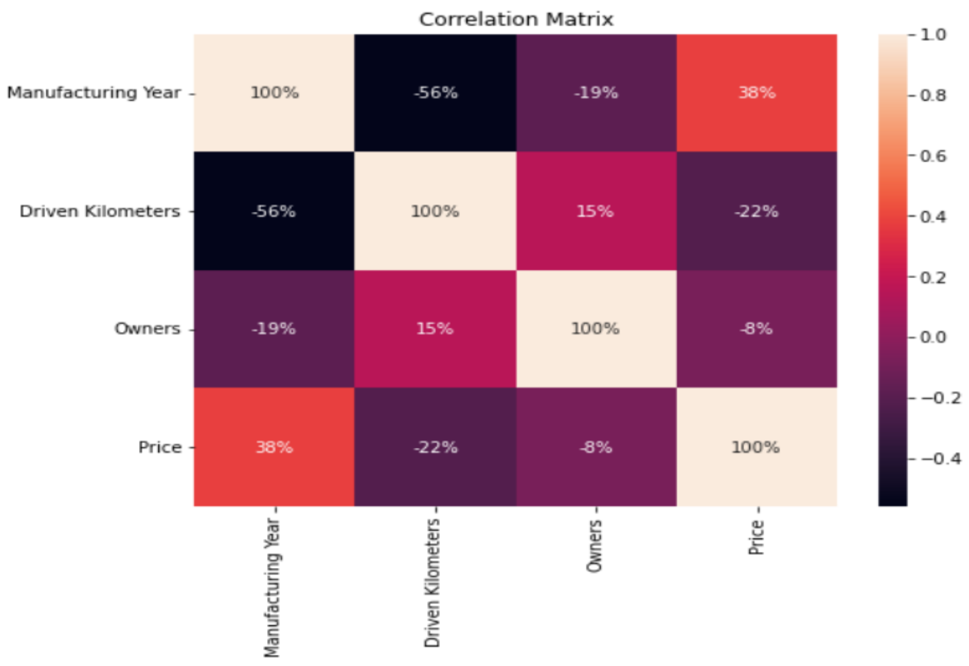
	Manufacturing Year	Driven Kilometers	Owners	Price
count	5660.00	5660.00	5660.00	5660.00
mean	2017.43	43490.69	1.21	754881.15
std	2.60	30719.56	0.44	420980.65
min	2009.00	244.00	1.00	202099.00
25%	2016.00	19930.00	1.00	458799.00
50%	2018.00	37413.00	1.00	639899.00
75%	2019.00	65162.00	1.00	897899.00
max	2022.00	161137.00	3.00	2139099.00

From above observation we can say that, the mean price of a car is around 7.5Lakh, and mean year of manufacturing is 2017 and driven kilometers is about 43490. The highest price for car in the dataset is 2139099.

As of now the dataset looks good. Let's go ahead and visualize the correlation matrix, before encoding categorical columns.



```
plt.figure(figsize=(8,6))
sns.heatmap(df.corr(), annot= True, fmt='.0%')
plt.title("Correlation Matrix")
plt.show()
```



	Manufacturing Year	Driven Kilometers	Owners	Price
Manufacturing Year	100%	-56%	-19%	38%
Driven Kilometers	-56%	100%	15%	-22%
Owners	-19%	15%	100%	-8%
Price	38%	-22%	-8%	100%

We can observe that Manufacturing Year is highest correlated feature so far. and Owners is lesser correlation comparatively.

Both Driven kilometers and Owners are negatively correlated to Price, which we observe early also. Negative correlation means if the value of One Increases the value of other variable decreases. Here in our case as owner or driven kilometer increases, there is sure depreciation of Car Price.

Let's Encode our categorical columns.

Encoding:

Encoding is a technique of converting categorical variables into numerical values so that it could be easily fitted to a machine learning model.

In our dataset, let's first check out how many categorical columns we need to encode.

```
df.dtypes
```

```
Brand                  object
Model                  object
Variant                object
Transmission           object
Manufacturing Year      int64
Driven Kilometers       int32
Owners                  int64
Fuel                   object
Price                   int32
dtype: object
```

We can observe that 'Brand', 'Model', 'Variant','Transmission', 'Fuel' are our categorical columns. And to encode them we can use label encoder as they have Nominal data.

As they are nominal data, we can encode them using Label Encoder, as we don't want to increase the number of columns.

```
# Encoding using LabelEncoder
from sklearn.preprocessing import LabelEncoder
lab_enc= LabelEncoder()

#Transform
df['Brand']= lab_enc.fit_transform(df['Brand'])
df['Model']= lab_enc.fit_transform(df['Model'])
df['Variant']= lab_enc.fit_transform(df['Variant'])
df['Transmission']= lab_enc.fit_transform(df['Transmission'])
df['Fuel']= lab_enc.fit_transform(df['Fuel'])
```

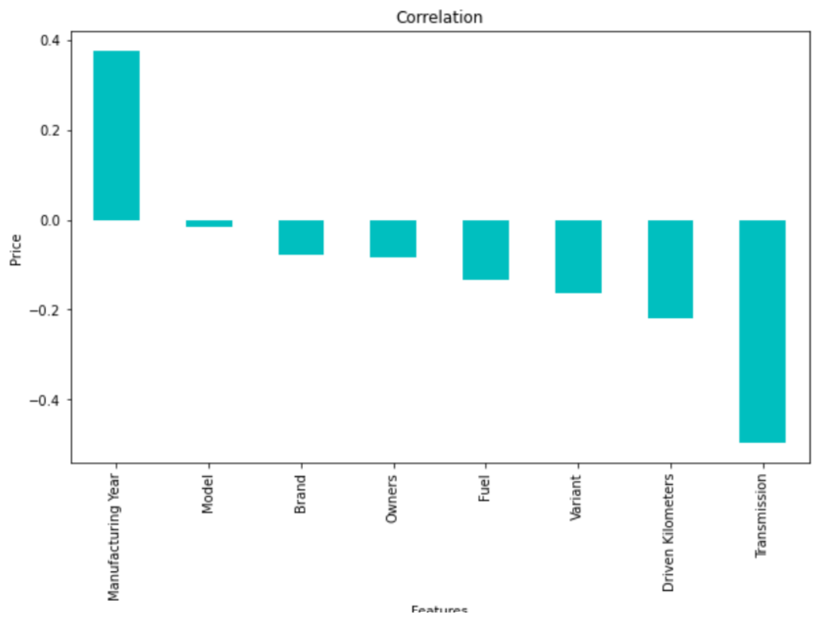
Let's have a look at our encoded dataset using .head().

```
df.head()    # checking our dataset
```

	Brand	Model	Variant	Transmission	Manufacturing Year	Driven Kilometers	Owners	Fuel	Price
0	7	47	62	0	2020	8241	1	1	1895199
1	10	49	70	1	2019	27659	1	1	533399
2	11	9	51	0	2014	37531	1	1	2033499
3	8	3	93	0	2022	2091	1	1	1767999
4	15	2	132	1	2021	7416	1	1	720176

We can observe that all our data is properly encoded, we are not left with any Object column. It's safe to proceed further in ML model Building. However, before that, let's check for correlation one more time.

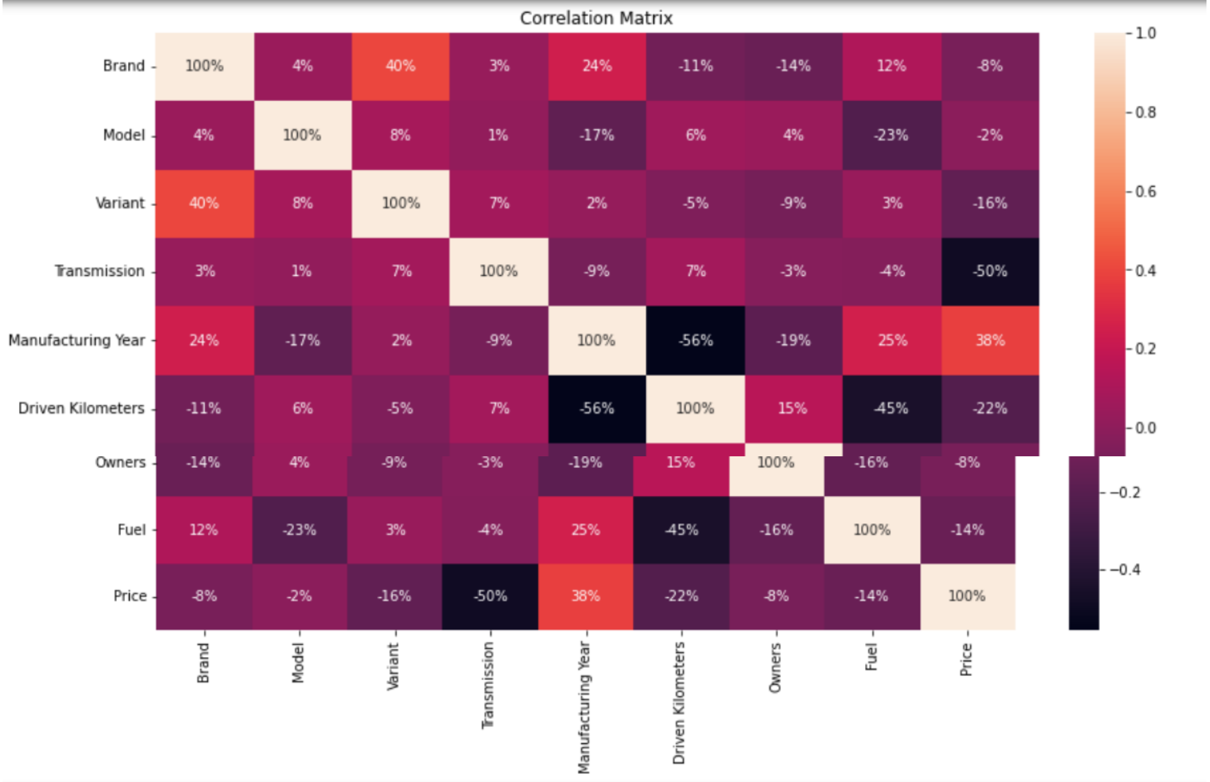
```python
# correlation with target variable
plt.figure(figsize=(10,6))
df.corr()['Price'].sort_values(ascending= False).drop(['Price']).plot(kind= 'bar', color='c')
plt.xlabel("Features")
plt.ylabel("Price")
plt.title("Correlation")
plt.show()
```



From here We can observe that most of the features are negatively correlated with our target variable i.e Price. Manufacturing Year shows strong correlation with Price. Model alone don't show strong correlation with Price, but we can see that Brand, Variant is comparatively stronger features.

Let's plot correlation matrix one more time for entire dataset. This will also helps us to observe decree of correlation between our independent Features.

```python
# Plotting correlation matrix using heat map
plt.figure(figsize=(14,8))
sns.heatmap(df.corr(), annot= True, fmt='.0%')
plt.title("Correlation Matrix")
plt.show()
```



From above observation it is clear that or dataset don't have multicollinearity. As the independent variables i.e. the feature columns are not correlated with one another.

We also observe that Transmission shows strong correlation with Price. While Model shows least correlation with Price which is similar to what we observe in the previous plot.

Skewness And Outliers.

For checking the skewness we use .skew()

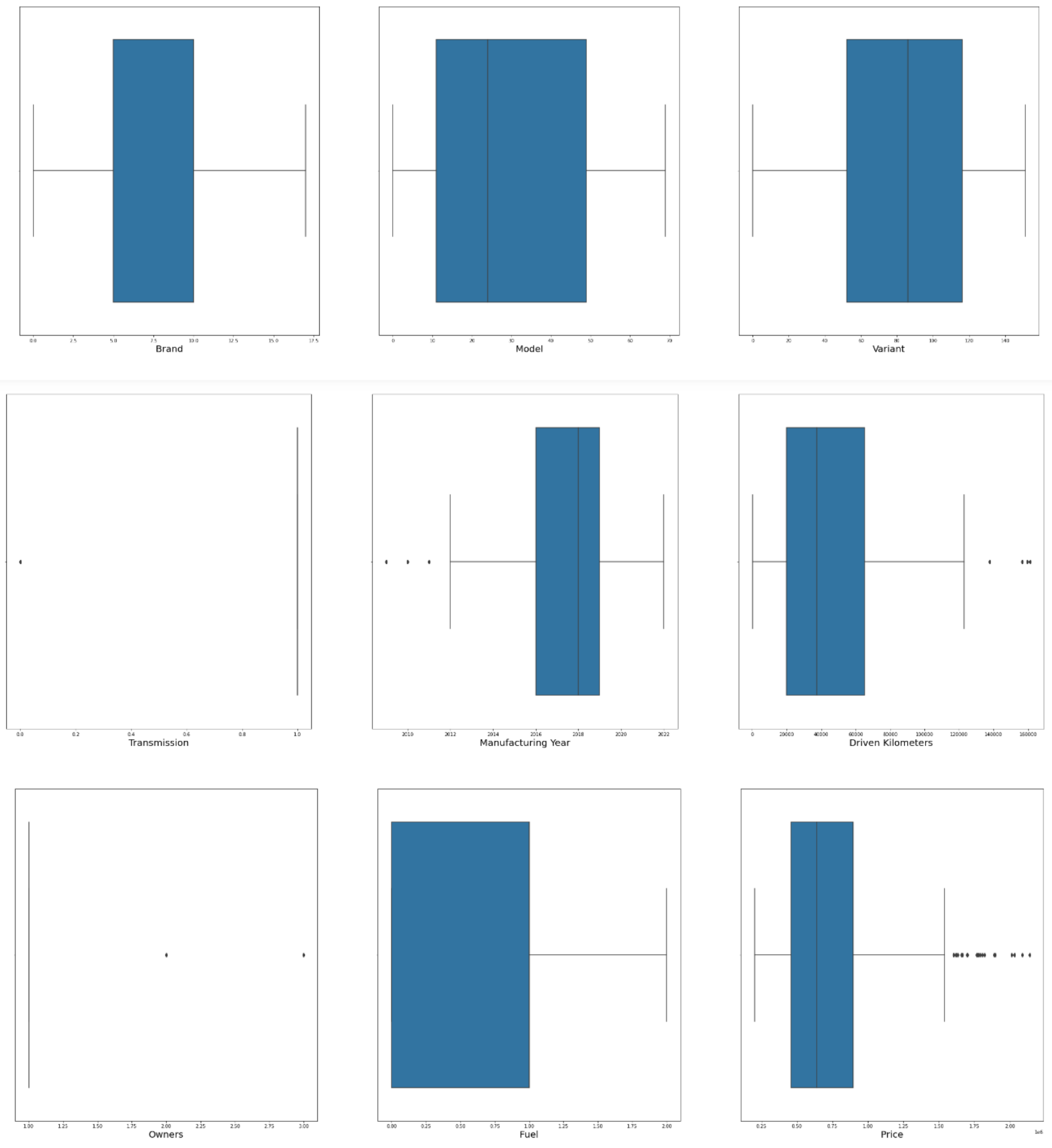
```python
df.skew()
```

```
]:  Brand                  0.062091
    Model                  0.395864
    Variant               -0.233537
    Transmission          -1.409531
    Manufacturing Year    -0.647837
    Driven Kilometers      0.995041
    Owners                 1.922652
    Fuel                  -0.346383
    Price                  1.403594
    dtype: float64
```

We can observe that, taking threshold value for skewness to be =/-1 . We don't observe any strong skewness in our dataset. And those columns which shows skewness are categorical columns or label and on both the cases we don't treat skewness in them.

Let’s visualise outliers using boxplot.

```python
plt.figure(figsize=(40,45))
plotnumber=1
for i in df:
    if plotnumber<=9:
        ax=plt.subplot(3,3,plotnumber)
        sns.boxplot(df[i])
        plt.xlabel(i,fontsize= 20)

    plotnumber+=1
plt.show()
```



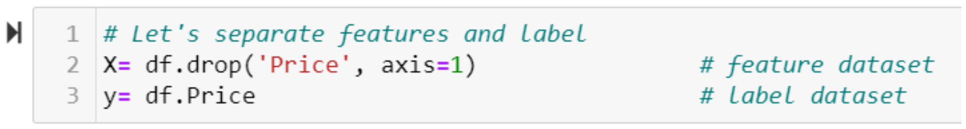
We can see couple outliers in Price, but is our Label column so we don't disturb that, rest dataset looks good.

We are safe to proceed further, with scaling and then finally building the ML model.

Scaling:

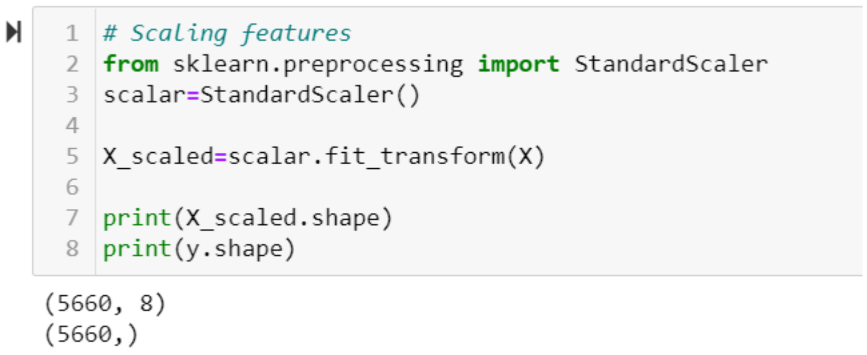
It is performed during the data pre-processing to handle highly varying magnitudes or values or units. If feature scaling is not done, then a machine learning algorithm tends to weigh greater values, higher and consider smaller values as the lower values, regardless of the unit of the values.

We will use standard scalar in order to scale our data, one thing to note here that, we only scale features not label. So, we need to separate, Features and Label.



```
# Let's separate features and label
X= df.drop('Price', axis=1)                    # feature dataset
y= df.Price                                    # label dataset
```

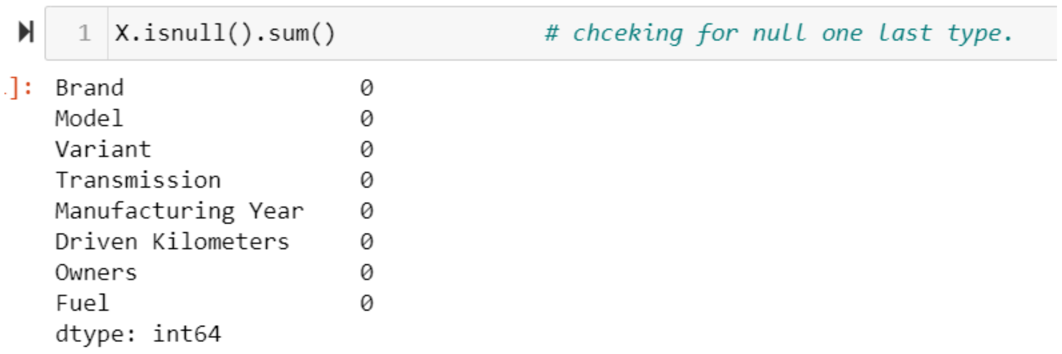
Let’s scale them using Standard Scalar techniques.



```
# Scaling features
from sklearn.preprocessing import StandardScaler
scalar=StandardScaler()

X_scaled=scalar.fit_transform(X)

print(X_scaled.shape)
print(y.shape)
```

```
(5660, 8)
(5660,)
```

Since we scaled our features, we are all set to build the machine learning model. Before that let’s check our Feature dataset, in order to see if we have any missing value or something.



```
X.isnull().sum()                       # chceking for null one last type.
```

```
]:  Brand                  0
    Model                  0
    Variant                0
    Transmission           0
    Manufacturing Year     0
    Driven Kilometers      0
    Owners                 0
    Fuel                   0
    dtype: int64
```

We can observe that the Feature dataset don’t have any null values, and is scaled properly. It is safe to go ahead with model Building.

Model Building:

Building an ML Model requires splitting of data into two sets, such as 'training set' and 'testing set' in the ratio of 80:20 or 75:25. Here we are using train test split in order to split the data.

```python
# importing necessary library
from sklearn.model_selection import train_test_split, GridSearchCV

#metrics
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error
```

```python
X_train,X_test, y_train,y_test = train_test_split(X_scaled, y, test_size=.20, random_state=42)
```

As we split our dataset into train and test set, (20% data for test) we can go ahead with different Models. As Price is our label, we know that it will be a regression model.

So, let's get started with Linear Regression.

Linear Regression:

Linear Regression is **a machine learning algorithm based on supervised learning**. It performs a regression task. Regression models a target prediction value based on independent variables. It is mostly used for finding out the relationship between variables and forecasting.

```python
from sklearn.linear_model import LinearRegression

# ML Model
LR= LinearRegression()

#fit
LR.fit(X_train,y_train)

#predict
y_pred= LR.predict(X_test)
pred=LR.predict(X_train)

print("--------- Train score---------")
LR_train_MAE= round(mean_absolute_error(y_train, pred), 2)
LR_train_avg_MAE= LR_train_MAE/(max(y)-min(y))
LR_train_R2 = round(r2_score(y_train, pred), 4)
LR_train_RMSE=(np.sqrt(mean_squared_error(y_train, pred))/(max(y)-min(y)))

print(f" R^2 Score : {LR_train_R2}\n")
print(f" MAE score avg : {LR_train_avg_MAE}\n")
print(f" RMSE score avg : {LR_train_RMSE}\n")


#score variables
LR_R2= round(r2_score(y_test, y_pred), 4);
LR_MAE=(mean_absolute_error(y_test, y_pred)/(max(y)-min(y)))
LR_RMSE=(np.sqrt(mean_squared_error(y_test,y_pred)))/(max(y)-min(y))


print("---------------Test Score-------------")
print(f" R^2 Score : {LR_R2}\n")
print(f" MAE score avg : {LR_MAE}\n")
print(f" RMSE score avg : {LR_RMSE}\n")
```

We are using r2 score , mean absolute error and root mean squared error as our scoring parameter, It will help us to determine the efficiency of the model. And at the end helps in model selection.

Also, we are taking the calculated average values of both the errors because normally it tells us the deviation from actual value, and our actual value are in 7 figures. So, it’s better to calculated the average of all the deviations. Our motive is to have the values near to zero. Let’s have a look at our scores.

```
--------- Train score---------
 R^2 Score : 0.4638

 MAE score avg : 0.11612123902942695

 RMSE score avg : 0.16132774138886954

---------------Test Score-------------
 R^2 Score : 0.4357

 MAE score avg : 0.11186514389644407

 RMSE score avg : 0.15385505780075992
```

```python
# cross validation
from sklearn.model_selection import cross_val_score
LR= LinearRegression()
scores= cross_val_score(LR, X_train, y_train, scoring='r2', cv=10)
LR_CS=scores.mean()
print("Cross validation score is : ", LR_CS)
```

```
Cross validation score is :  0.4581797936768311
```

We can observe that although our MAE. RMSE scores are better, we can still work on r2 score. After cross validation the score increase slightly. We can perform LASSO or Ridge to see if our Linear model works better. However, before regularization (lasso or ridge) let's perform an ensemble technique, and see how our model performs.

RandomForestRegressor:

Random Forest Regression is **a supervised learning algorithm that uses ensemble learning method for regression**. Ensemble learning method is a technique that combines predictions from multiple machine learning algorithms to make a more accurate prediction than a single model.

```python
from sklearn.ensemble import RandomForestRegressor

X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.20, random_state=42)
#Model
RFR=RandomForestRegressor()

#fit
RFR.fit(X_train,y_train)

#predict
y_pred= RFR.predict(X_test)
pred=RFR.predict(X_train)

print("--------- Train score---------")
RFR_train_R2= round(r2_score(y_train, pred), 4)
RFR_train_MAE=(mean_absolute_error(y_train,pred))/(max(y)-min(y))
RFR_train_RMSE=(np.sqrt(mean_squared_error(y_train,pred)))/(max(y)-min(y))

print(f" R^2 Score : {RFR_train_R2}\n")
print(f" MAE avg score : {RFR_train_MAE}\n")
print(f" RMSE avg score : {RFR_train_RMSE}\n")

#score variables
RFR_R2= round(r2_score(y_test, y_pred), 4)
RFR_MAE=(mean_absolute_error(y_test,y_pred))/(max(y)-min(y))
RFR_RMSE=(np.sqrt(mean_squared_error(y_test,y_pred)))/(max(y)-min(y))
print("---------------Test Score-------------")
print(f" R^2 Score : {RFR_R2}\n")
print(f" MAE avg score : {RFR_MAE}\n")
print(f" RMSE avg score : {RFR_RMSE}\n")

```

```
---------Train score---------
 R^2 Score : 1.0

 MAE avg score : 0.0

 RMSE avg score : 0.0

---------------Test Score-------------
 R^2 Score : 1.0

 MAE avg score : 0.0

 RMSE avg score : 0.0
```

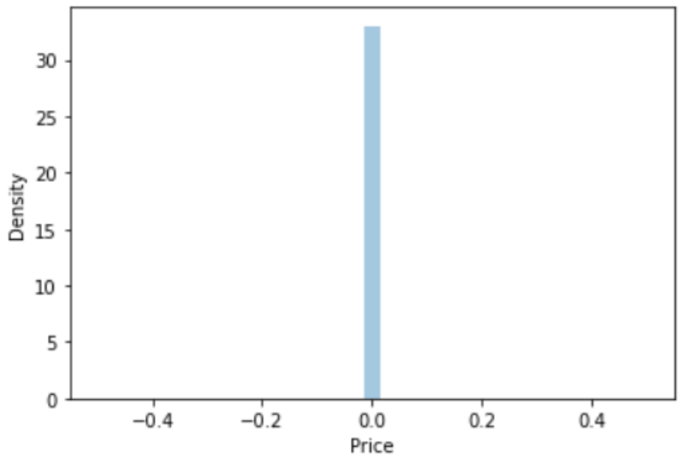
Interesting, Looks like it’s a perfect fit for the model. Anyways, it can be overfitting, let’s have a look at cross validation score.

```
# cross validation
RFR= RandomForestRegressor()
scores= cross_val_score(RFR, X_train, y_train, scoring='r2', cv=10)
RFR_CS=scores.mean()
print("Cross validation score is : ", RFR_CS)
```

```
Cross validation score is :  1.0
```

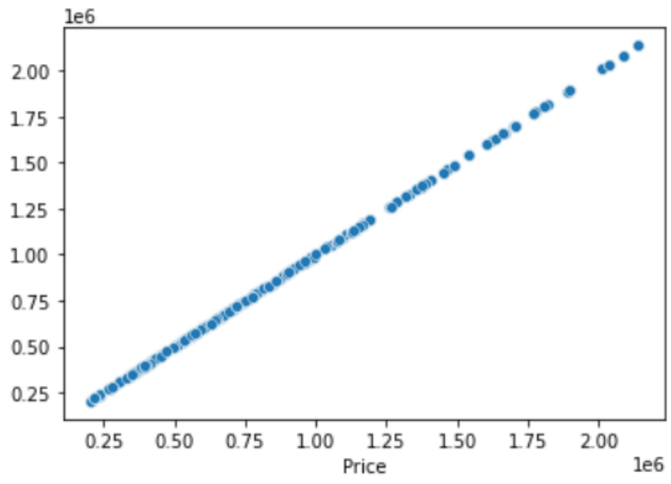
Great, Let’s do some visualization.

```
sns.distplot(y_test-y_pred)
plt.show()
```



```
sns.scatterplot(y_test,y_pred)
```

```
]: <AxesSubplot:xlabel='Price'>
```



Interesting!! It gives perfect straight line. Let's perform hyper parameter tuning, in order to rule out any confusion.

```
# we are performing hyper parameter tuning in order to chcek is over model overfitting or not.
# Hyper Parameter Tunning
# Create the random grid
from sklearn.model_selection import RandomizedSearchCV
random_grid = {'n_estimators': range(100,1200,100),
               'max_features':['auto', 'sqrt'] ,
               'max_depth': range(5,30,5),
               'min_samples_split': [2, 5, 10, 15, 100],
               'min_samples_leaf':[1, 2, 5, 10] }

#grid_search=GridSearchCV(estimator=RFR, param_grid= random_grid, cv=5)

#grid_search.fit(X_train,y_train)

#grid_search.best_estimator_

rnd_srch=RandomizedSearchCV(RandomForestRegressor(), cv=5, param_distributions= random_grid)

rnd_srch.fit(X_train,y_train)

rnd_srch.best_estimator_
```

```
]: RandomForestRegressor(max_depth=25, max_features='auto', min_samples_leaf=5,
                         min_samples_split=5, n_estimators=300)
```

```
#prediction= grid_search.best_estimator_.predict(X_test)

prediction=rnd_srch.best_estimator_.predict(X_test)
print("Post tuning scores")

#score variables
R2= round(r2_score(y_test, prediction), 4)
MAE=(mean_absolute_error(y_test,prediction))
RMSE=(np.sqrt(mean_squared_error(y_test,prediction)))
print("---------------Test Score-------------")
print(f" R^2 Score : {R2}\n")
print(f" MAE score : {MAE}\n")
print(f" RMSE score : {RMSE}\n")


```

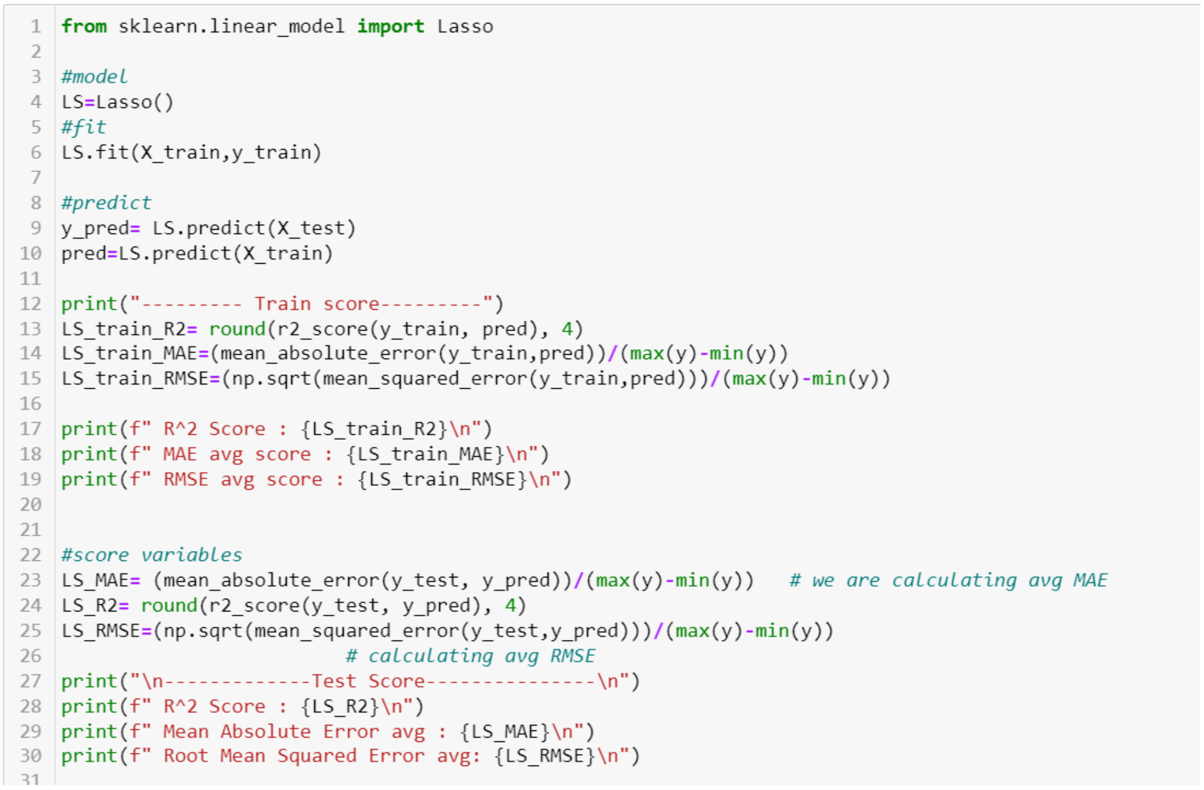
```
Post tuning scores
---------------Test Score-------------
 R^2 Score : 1.0

 MAE score : 398.5076471118441

 RMSE score : 931.1578852281531
```

These errors represent the deviation from actual value, It means if the price is 6789345 then the got a deviation in Predicted price of +/- 398.51 for mean absolute error which is very minimal. Same goes for RMSE also. Which is still pretty good score. We can say that this might be our final model.

Before deciding, let's build 2-3 more algorithm then will summarize every model to get the best one.

LASSO:

In statistics and machine learning, lasso (least absolute shrinkage and selection operator; also Lasso or LASSO) is a regression analysis method that performs both variable selection and regularization in order to enhance the prediction accuracy and interpretability of the resulting statistical model.



```python
from sklearn.linear_model import Lasso

#model
LS=Lasso()
#fit
LS.fit(X_train,y_train)

#predict
y_pred= LS.predict(X_test)
pred=LS.predict(X_train)

print("--------- Train score---------")
LS_train_R2= round(r2_score(y_train, pred), 4)
LS_train_MAE=(mean_absolute_error(y_train,pred))/(max(y)-min(y))
LS_train_RMSE=(np.sqrt(mean_squared_error(y_train,pred)))/(max(y)-min(y))

print(f" R^2 Score : {LS_train_R2}\n")
print(f" MAE avg score : {LS_train_MAE}\n")
print(f" RMSE avg score : {LS_train_RMSE}\n")


#score variables
LS_MAE= (mean_absolute_error(y_test, y_pred))/(max(y)-min(y))   # we are calculating avg MAE
LS_R2= round(r2_score(y_test, y_pred), 4)
LS_RMSE=(np.sqrt(mean_squared_error(y_test,y_pred)))/(max(y)-min(y))
                         # calculating avg RMSE
print("\n------------Test Score--------------\n")
print(f" R^2 Score : {LS_R2}\n")
print(f" Mean Absolute Error avg : {LS_MAE}\n")
print(f" Root Mean Squared Error avg: {LS_RMSE}\n")
```

```
--------- Train score---------
 R^2 Score : 0.4638

 MAE avg score : 0.11612108865403853

 RMSE avg score : 0.16132774140374892


-------------Test Score--------------

 R^2 Score : 0.4357

 Mean Absolute Error avg : 0.11186489554534768

 Root Mean Squared Error avg: 0.15385485058941437
```

```python
# cross validation
LS= Lasso()
scores= cross_val_score(LS, X_train, y_train, scoring='r2', cv=10)
LS_CS=scores.mean()
print("Cross validation score is : ", LS_CS)
```

```
Cross validation score is :  0.45817985092121793
```

It did perform better than Linear Regression, however its not as suitable as ensemble technique. Let’s do some hyper parameter tuning to see if it performs any better. We are performing RandomizedSearchCV as it is faster than Grid SearchCV.

```python
# Hyper parameter tuning

param_grid= {'selection' : ['cyclic', 'random'],
             'max_iter': range(500,1200,100),
             'alpha': [0.01, 0.1, 0.5, 1.0],
             'random_state': range(0,100, 5)}


rnd_srch=RandomizedSearchCV(Lasso(), cv=5, param_distributions= param_grid)

rnd_srch.fit(X_train,y_train)

rnd_srch.best_estimator_

```

```
]: Lasso(max_iter=900, random_state=80, selection='random')
```

```python
rnd_pred= rnd_srch.best_estimator_.predict(X_test)   # predicting with best para meters
print("Accuracy post tuning \n")
print(r2_score(y_test, rnd_pred))
print("\nPost tuning MAE\n")
print(mean_absolute_error(y_test,rnd_pred)/(max(y)-min(y)))
```

```
Accuracy post tuning

0.43570129038388783

Post tuning MAE

0.11186489554202898
```

We can observe that Linear model is not working well with our Dataset, while the ensemble technique works amazingly well. Let's do another ensemble boosting model before finalizing.

AdaBoost Regressor:

AdaBoost also called Adaptive Boosting is a technique in Machine Learning used as an Ensemble Method. An AdaBoost regressor is a meta-estimator that begins by fitting a regressor on the original dataset and then fits additional copies of the regressor on the same dataset but where the weights of instances are adjusted according to the error of the current prediction.

```python
from sklearn.ensemble import AdaBoostRegressor

#  Model
ADA= AdaBoostRegressor()

#fit
ADA.fit(X_train,y_train)

#predict
y_pred=ADA.predict(X_test)
pred=ADA.predict(X_train)

print("--------- Train score---------")
ADA_train_R2= round(r2_score(y_train, pred), 4)
ADA_train_MAE=(mean_absolute_error(y_train,pred))/(max(y)-min(y))
ADA_train_RMSE=(np.sqrt(mean_squared_error(y_train,pred)))/(max(y)-min(y))

print(f" R^2 Score : {ADA_train_R2}\n")
print(f" MAE avg score : {ADA_train_MAE}\n")
print(f" RMSE avg score : {ADA_train_RMSE}\n")


#score variables
ADA_MAE= (mean_absolute_error(y_test, y_pred))/(max(y)-min(y))   # we are calculating avg MAE
ADA_R2= round(r2_score(y_test, y_pred), 4)
ADA_RMSE=(np.sqrt(mean_squared_error(y_test,y_pred)))/(max(y)-min(y))
                                          # calculating avg RMSE
print("\n-------------Test Score---------------\n")
print(f" R^2 Score : {ADA_R2}\n")
print(f" Mean Absolute Error avg : {ADA_MAE}\n")
print(f" Root Mean Squared Error avg: {ADA_RMSE}\n")

```

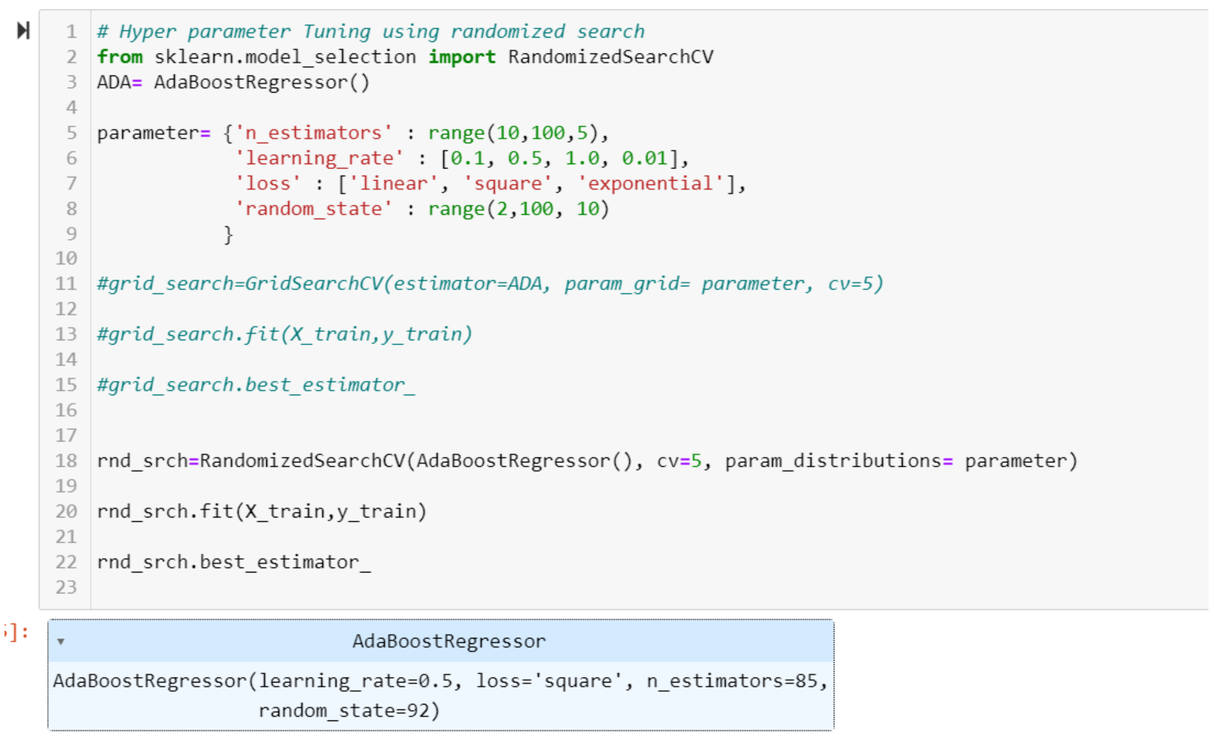
```
--------- Train score---------
 R^2 Score : 0.7586

 MAE avg score : 0.08914066893280381

 RMSE avg score : 0.10823587110919702


-------------Test Score---------------

 R^2 Score : 0.7382

 Mean Absolute Error avg : 0.08665084577045254

 Root Mean Squared Error avg: 0.10480272901776572
```

We can observe that, this technique also performs better in our model than other linear model techniques. Let’s check for cross validation score.

```python
# Cross validation
scores= cross_val_score(ADA, X_train, y_train, scoring='r2', cv=10)
ADA_CS=scores.mean()
print("Cross validation score is : ", ADA_CS)
```

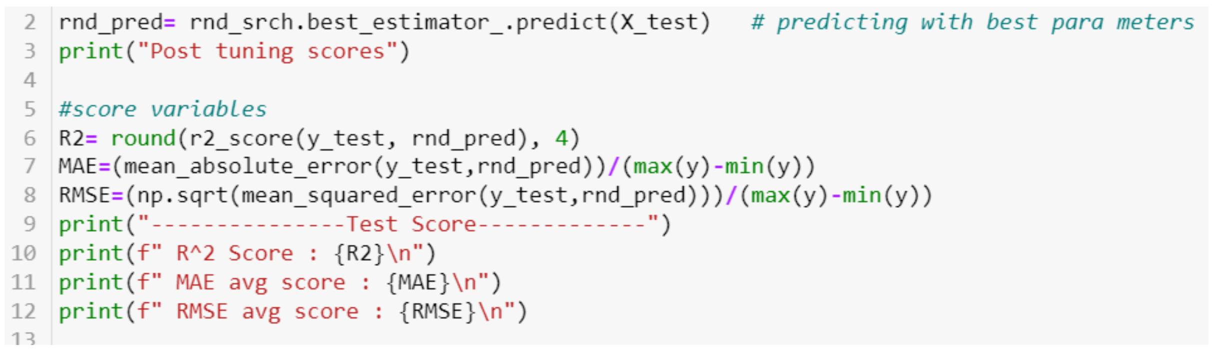
```
Cross validation score is :  0.7589167045100096
```

We can observe that the score increases slightly, we can go ahead and do hyper parameter tuning, and see how our model works.

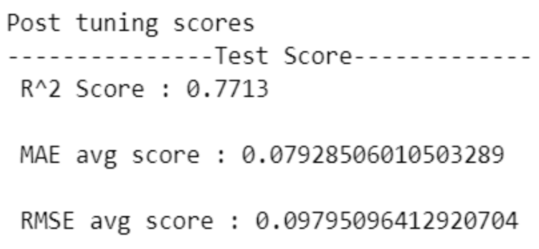


```python
# Hyper parameter Tuning using randomized search
from sklearn.model_selection import RandomizedSearchCV
ADA= AdaBoostRegressor()

parameter= {'n_estimators' : range(10,100,5),
            'learning_rate' : [0.1, 0.5, 1.0, 0.01],
            'loss' : ['linear', 'square', 'exponential'],
            'random_state' : range(2,100, 10)
           }

#grid_search=GridSearchCV(estimator=ADA, param_grid= parameter, cv=5)

#grid_search.fit(X_train,y_train)

#grid_search.best_estimator_


rnd_srch=RandomizedSearchCV(AdaBoostRegressor(), cv=5, param_distributions= parameter)

rnd_srch.fit(X_train,y_train)

rnd_srch.best_estimator_

```

```
]:
                        AdaBoostRegressor
AdaBoostRegressor(learning_rate=0.5, loss='square', n_estimators=85,
                  random_state=92)
```



```python
rnd_pred= rnd_srch.best_estimator_.predict(X_test)   # predicting with best para meters
print("Post tuning scores")

#score variables
R2= round(r2_score(y_test, rnd_pred), 4)
MAE=(mean_absolute_error(y_test,rnd_pred))/(max(y)-min(y))
RMSE=(np.sqrt(mean_squared_error(y_test,rnd_pred)))/(max(y)-min(y))
print("---------------Test Score------------")
print(f" R^2 Score : {R2}\n")
print(f" MAE avg score : {MAE}\n")
print(f" RMSE avg score : {RMSE}\n")
```

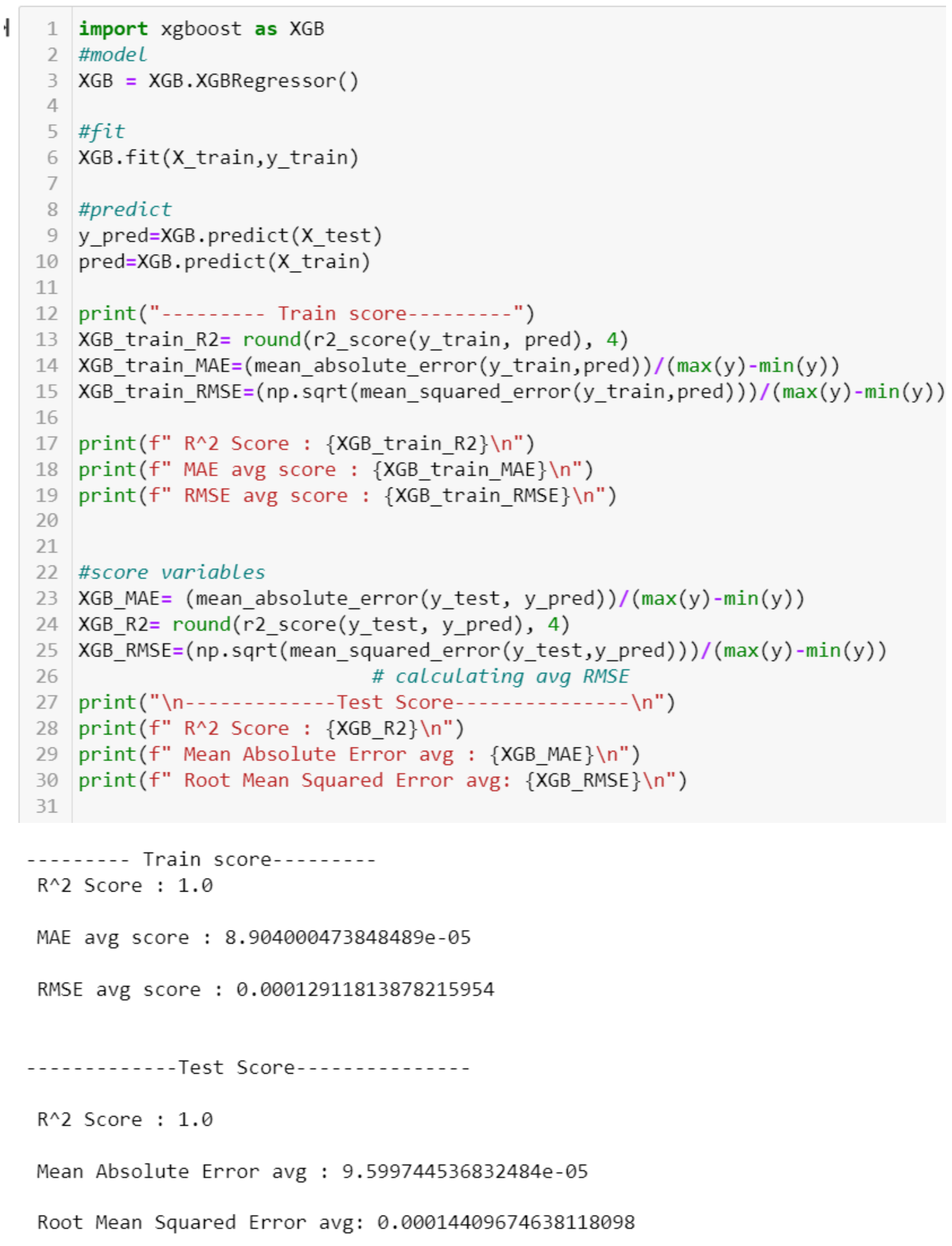


```
Post tuning scores
---------------Test Score------------
 R^2 Score : 0.7713

 MAE avg score : 0.07928506010503289

 RMSE avg score : 0.09795096412920704
```

We can see that Adaboost Regressor Model also works better with our dataset. We can now go ahead and perform XGBOOST Regressor

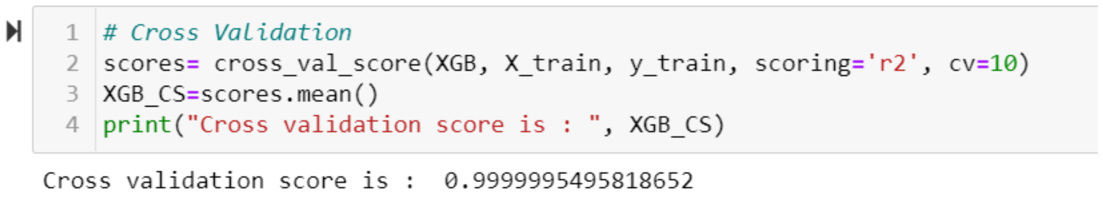
XGBoost Regressor:

XGBoost, which stands for Extreme Gradient Boosting, is **a scalable, distributed gradient-boosted decision tree (GBDT) machine learning library**. It provides parallel tree boosting and is the leading machine learning library for regression.



```python
import xgboost as XGB
#model
XGB = XGB.XGBRegressor()

#fit
XGB.fit(X_train,y_train)

#predict
y_pred=XGB.predict(X_test)
pred=XGB.predict(X_train)

print("--------- Train score---------")
XGB_train_R2= round(r2_score(y_train, pred), 4)
XGB_train_MAE=(mean_absolute_error(y_train,pred))/(max(y)-min(y))
XGB_train_RMSE=(np.sqrt(mean_squared_error(y_train,pred)))/(max(y)-min(y))

print(f" R^2 Score : {XGB_train_R2}\n")
print(f" MAE avg score : {XGB_train_MAE}\n")
print(f" RMSE avg score : {XGB_train_RMSE}\n")


#score variables
XGB_MAE= (mean_absolute_error(y_test, y_pred))/(max(y)-min(y))
XGB_R2= round(r2_score(y_test, y_pred), 4)
XGB_RMSE=(np.sqrt(mean_squared_error(y_test,y_pred)))/(max(y)-min(y))
                         # calculating avg RMSE
print("\n-------------Test Score--------------\n")
print(f" R^2 Score : {XGB_R2}\n")
print(f" Mean Absolute Error avg : {XGB_MAE}\n")
print(f" Root Mean Squared Error avg: {XGB_RMSE}\n")

```

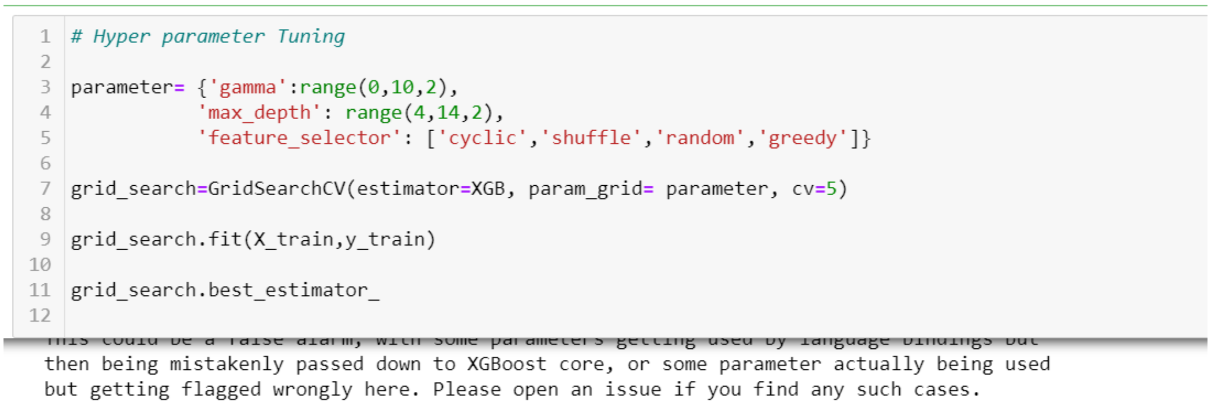
```
--------- Train score---------
 R^2 Score : 1.0

 MAE avg score : 8.904000473848489e-05

 RMSE avg score : 0.00012911813878215954


-------------Test Score--------------

 R^2 Score : 1.0

 Mean Absolute Error avg : 9.599744536832484e-05

 Root Mean Squared Error avg: 0.00014409674638118098
```



```python
# Cross Validation
scores= cross_val_score(XGB, X_train, y_train, scoring='r2', cv=10)
XGB_CS=scores.mean()
print("Cross validation score is : ", XGB_CS)
```

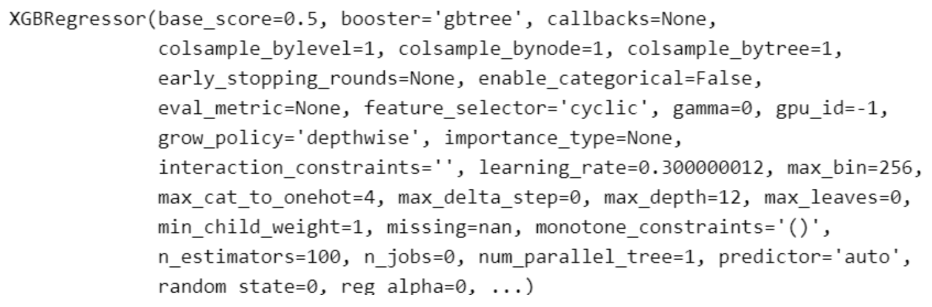
```
Cross validation score is :  0.9999995495818652
```

This Model also works well with our dataset. The score is good, the errors are almost zero. We can do some hyper parameter tuning.

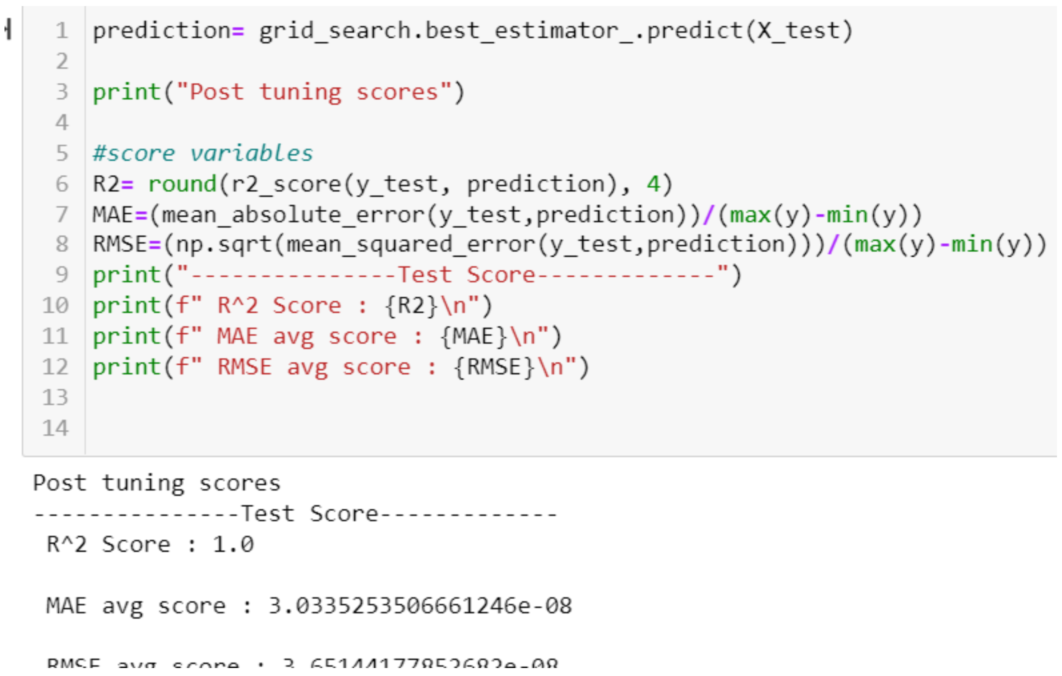


```python
# Hyper parameter Tuning

parameter= {'gamma':range(0,10,2),
            'max_depth': range(4,14,2),
            'feature_selector': ['cyclic','shuffle','random','greedy']}

grid_search=GridSearchCV(estimator=XGB, param_grid= parameter, cv=5)

grid_search.fit(X_train,y_train)

grid_search.best_estimator_
```

```
This could be a false alarm, with some parameters getting used by language bindings but
then being mistakenly passed down to XGBoost core, or some parameter actually being used
but getting flagged wrongly here. Please open an issue if you find any such cases.
```



```
XGBRegressor(base_score=0.5, booster='gbtree', callbacks=None,
             colsample_bylevel=1, colsample_bynode=1, colsample_bytree=1,
             early_stopping_rounds=None, enable_categorical=False,
             eval_metric=None, feature_selector='cyclic', gamma=0, gpu_id=-1,
             grow_policy='depthwise', importance_type=None,
             interaction_constraints='', learning_rate=0.300000012, max_bin=256,
             max_cat_to_onehot=4, max_delta_step=0, max_depth=12, max_leaves=0,
             min_child_weight=1, missing=nan, monotone_constraints='()',
             n_estimators=100, n_jobs=0, num_parallel_tree=1, predictor='auto',
             random_state=0, reg_alpha=0, ...)
```



```python
prediction= grid_search.best_estimator_.predict(X_test)

print("Post tuning scores")

#score variables
R2= round(r2_score(y_test, prediction), 4)
MAE=(mean_absolute_error(y_test,prediction))/(max(y)-min(y))
RMSE=(np.sqrt(mean_squared_error(y_test,prediction)))/(max(y)-min(y))
print("---------------Test Score-------------")
print(f" R^2 Score : {R2}\n")
print(f" MAE avg score : {MAE}\n")
print(f" RMSE avg score : {RMSE}\n")
```

```
Post tuning scores
---------------Test Score-------------
 R^2 Score : 1.0

 MAE avg score : 3.0335253506661246e-08

 RMSE avg score : 3.65144177852682e-08
```

We can observe that this XGBoost also works amazingly well with our dataset, giving high r2 score and nearly zero error.

Let’s check on training score after tuning:

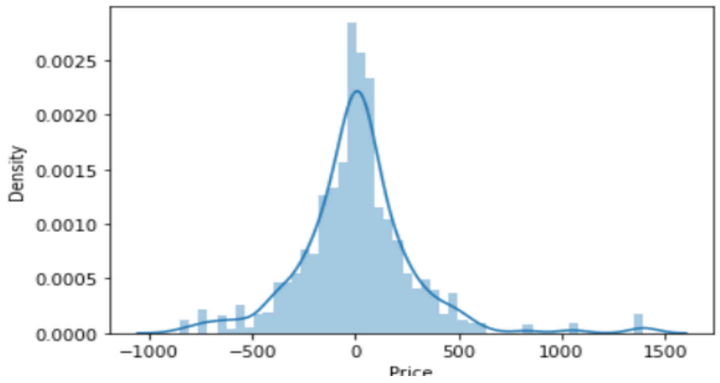
```
train_pred=grid_search.best_estimator_.predict(X_train)
print("Post Tuning score for train data")
r2_score(y_train,train_pred)
```

```
Post Tuning score for train data
```

```
]: 0.9999999999999726
```

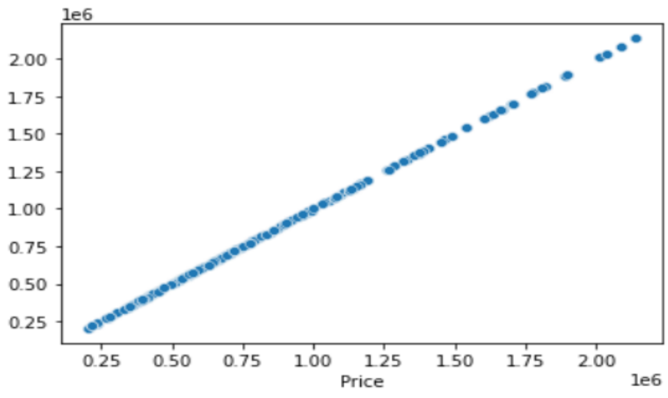
Let’s visualize the scores for XGBOOST Regressor Model.

```
sns.distplot(y_test-y_pred)
plt.show()
```



```
sns.scatterplot(y_test,prediction)
```

```
9]: <AxesSubplot:xlabel='Price'>
```



We can Observe that XGBOOST Regressor model , works equally well. Whatever error we are getting is also very low, After hyper parameter tuning, those error values decreases further almost tend to zero.

Also from the graph we plot, we can clearly see that there is minor deviation from true_y and predicted_y. which is not present in randomforrest

Let’s perform Ridge one last model to see if linear model has any score for our dataset.

Ridge :

Ridge regression is a model tuning method that is used to analyse any data that suffers from multicollinearity. This method performs L2 regularization. When the issue of multicollinearity occurs, least-squares are unbiased, and variances are large, this results in predicted values being far away from the actual values.

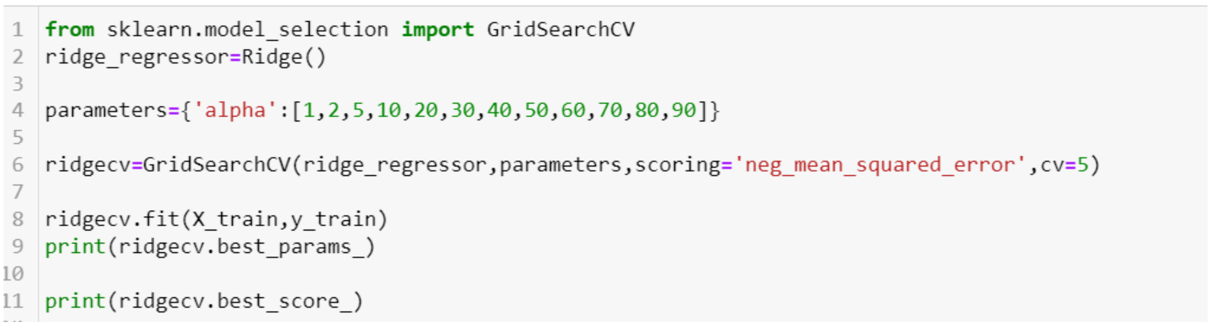
```
from sklearn.linear_model import Ridge
RG=Ridge()
#fit
RG.fit(X_train,y_train)

#predict
y_pred=RG.predict(X_test)
pred=RG.predict(X_train)

print("--------- Train score---------")
Rid_train_R2= round(r2_score(y_train, pred), 4)
Rid_train_MAE=(mean_absolute_error(y_train,pred))/(max(y)-min(y))
Rid_train_RMSE=(np.sqrt(mean_squared_error(y_train,pred)))/(max(y)-min(y))

print(f" R^2 Score : {Rid_train_R2}\n")
print(f" MAE avg score : {Rid_train_MAE}\n")
print(f" RMSE avg score : {Rid_train_RMSE}\n")


#score variables
Rid_MAE= (mean_absolute_error(y_test, y_pred))/(max(y)-min(y))
Rid_R2= round(r2_score(y_test, y_pred), 4)
Rid_RMSE=(np.sqrt(mean_squared_error(y_test,y_pred)))/(max(y)-min(y))
                         # calculating avg RMSE
print("\n------------Test Score--------------\n")
print(f" R^2 Score : {Rid_R2}\n")
print(f" Mean Absolute Error avg : {Rid_MAE}\n")
print(f" Root Mean Squared Error avg: {Rid_RMSE}\n")

```

```
--------- Train score---------
 R^2 Score : 0.4638

 MAE avg score : 0.11612177543533996

 RMSE avg score : 0.16132781765704865


------------Test Score--------------

 R^2 Score : 0.4359

 Mean Absolute Error avg : 0.11185680710409002

 Root Mean Squared Error avg: 0.15383394780599885
```

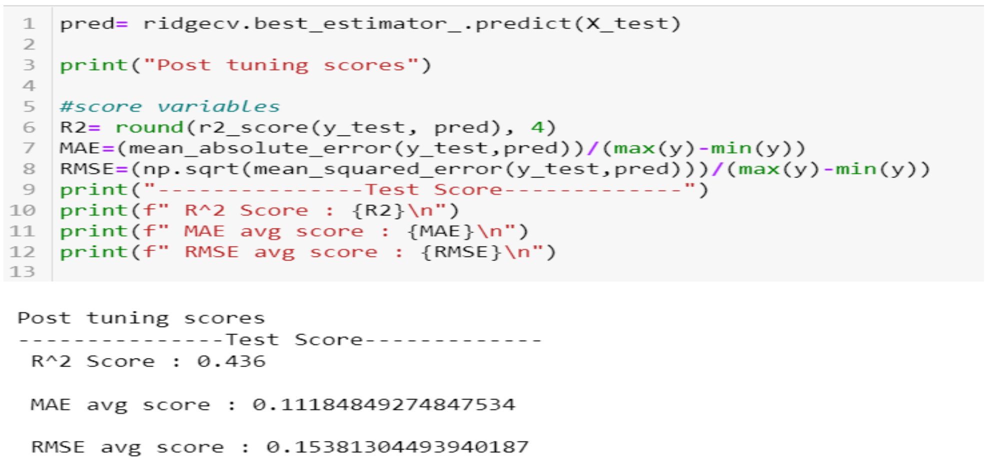
```
# Cross Validation
scores= cross_val_score(RG, X_train, y_train, scoring='r2', cv=10)
Rid_CS=scores.mean()
print("Cross validation score is : ", Rid_CS)
```

```
Cross validation score is :  0.4581878856037031
```

It definitely works better than linear regression and lasso, however the scores are still lower than ensemble techniques. Let’s do hyper parameter tuning in order to improve the performance.



```python
from sklearn.model_selection import GridSearchCV
ridge_regressor=Ridge()

parameters={'alpha':[1,2,5,10,20,30,40,50,60,70,80,90]}

ridgecv=GridSearchCV(ridge_regressor,parameters,scoring='neg_mean_squared_error',cv=5)

ridgecv.fit(X_train,y_train)
print(ridgecv.best_params_)

print(ridgecv.best_score_)
```

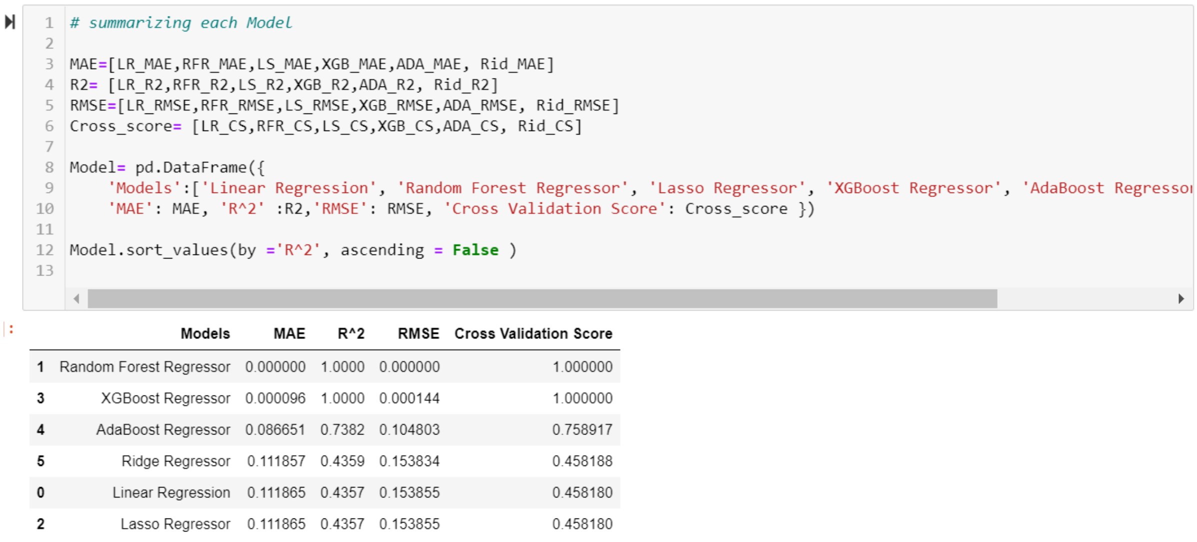


```python
pred= ridgecv.best_estimator_.predict(X_test)

print("Post tuning scores")

#score variables
R2= round(r2_score(y_test, pred), 4)
MAE=(mean_absolute_error(y_test,pred))/(max(y)-min(y))
RMSE=(np.sqrt(mean_squared_error(y_test,pred)))/(max(y)-min(y))
print("---------------Test Score-------------")
print(f" R^2 Score : {R2}\n")
print(f" MAE avg score : {MAE}\n")
print(f" RMSE avg score : {RMSE}\n")
```

```
Post tuning scores
---------------Test Score-------------
 R^2 Score : 0.436

 MAE avg score : 0.11184849274847534

 RMSE avg score : 0.15381304493940187
```

We can see that our Linear models performs not well as compared to ensemble models. Let’s summarize each model in order to get a clear picture of how the models are performing.

Summarizing:



```python
# summarizing each Model

MAE=[LR_MAE,RFR_MAE,LS_MAE,XGB_MAE,ADA_MAE, Rid_MAE]
R2= [LR_R2,RFR_R2,LS_R2,XGB_R2,ADA_R2, Rid_R2]
RMSE=[LR_RMSE,RFR_RMSE,LS_RMSE,XGB_RMSE,ADA_RMSE, Rid_RMSE]
Cross_score= [LR_CS,RFR_CS,LS_CS,XGB_CS,ADA_CS, Rid_CS]

Model= pd.DataFrame({
    'Models':['Linear Regression', 'Random Forest Regressor', 'Lasso Regressor', 'XGBoost Regressor', 'AdaBoost Regresso
    'MAE': MAE, 'R^2' :R2,'RMSE': RMSE, 'Cross Validation Score': Cross_score })

Model.sort_values(by ='R^2', ascending = False )
```

	Models	MAE	R^2	RMSE	Cross Validation Score
1	Random Forest Regressor	0.000000	1.0000	0.000000	1.000000
3	XGBoost Regressor	0.000096	1.0000	0.000144	1.000000
4	AdaBoost Regressor	0.086651	0.7382	0.104803	0.758917
5	Ridge Regressor	0.111857	0.4359	0.153834	0.458188
0	Linear Regression	0.111865	0.4357	0.153855	0.458180
2	Lasso Regressor	0.111865	0.4357	0.153855	0.458180

From the above table we can say that, Ensemble and Gradient Model works best with our dataset. However linear models show comparatively weaker performance. Also we can observe that Random Forest Regressor act like the perfect match for this particular dataset. One logical reason which I can think of is that, it might be possible since data is collected from only one website(car24.com) & one location(Delhi), there are chances that particular website is using RandomForest Algorithm to predict the prices of the Cars listed in there.

XGBoost Regressor works equally well with the dataset, with some errors or deviations here and there.

We can also Observe that Linear Regression and Lasso performs in the same manner, however Ridge shows some improvement in overall scores and hence we can say that it regularizes the linear regression model.

The performance of AdaBoost Regressor is pretty good, however we got two other models which performs better that it.

Conclusion:

In this Project, we perform two tasks:

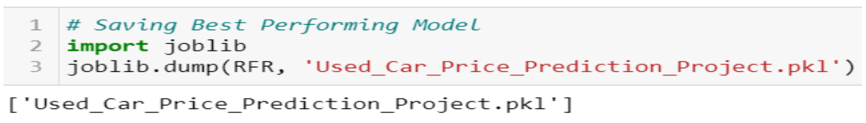
1. **Data Collection**: This is the very first part of our Project where we need to collect data from websites in order to perform a model which predict the Prices of Used car.
   For that we scrape the data from car24.com website. Here we choose the location Delhi and did not apply any other filter in order to collect data of all kinds of used car like Hatchback, SUV, Minivan, Sedan, Coupe.
   Here we collect a total number of 5660 cars data, and their important features are collected in the form 9 columns (Brand, Model, Variant, Transmission, Manufacturing Year, Driven Kilometers, Owners, Fuel, Price). We can clearly see that Price is our Target Variable. And others are our independent variable or features.
   We collected all this data in the form of Excel file.

2. **Model Building**: After collection of that, we use the same data to build our machine learning model. Here we perform all the required pre-processing steps to make the model suitable for Model Building algorithms.
   We in total build 6 Models in order to select on the best on, and after evaluating every single model we came to conclude that our dataset works best with ensemble techniques. So the Best suited Model we decide is RandomForest Regressor, as it gives us both high prediction scores and low error.

In the end, we will go ahead and save this model. For saving we will use pickle Model.

Saving The Model.

Pickle is a useful Python tool that allows you to save your models, to minimise lengthy re-training and allow you to share, commit, and re-load pre-trained machine learning models. Pickle is a generic object serialization module that can be used for serializing and deserializing objects.



```
# Saving Best Performing Model
import joblib
joblib.dump(RFR, 'Used_Car_Price_Prediction_Project.pkl')
```

```
['Used_Car_Price_Prediction_Project.pkl']
```