Real Estate Price Prediction Using Regression Based Predictive Modelling

Dataset:

House Sales in King County, USA

Predict house price using regression

k https://www.kaggle.com/harlfoxem/housesalesprediction?utm_medium=Exinfluencer&utm_source=Exinfluencer&utm_content=000026UJ&utm_term=10006555&utm_id=NA-SkillsNetwork-wwwcourseraorg-SkillsNetworkCoursesIBMDeveloperSkillsNetworkDA0101ENSkillsNetwork20235326-2022-01-01

```
"""
@author: Aditi
"""

import pandas as pd
import numpy as np
from scipy import stats

#for visualisation
import seaborn as sns
import matplotlib.pyplot as plt

#for model development
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler,PolynomialFeatures

#for model eveluation and refinement
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV
```

Importing Dataset and basic insights from data

```
file_name='kc_house_data_NaN.csv'
df=pd.read_csv(file_name)
```

```
df.head()
```

```
   Unnamed: 0          id             date  ...     long  sqft_living15  sqft_lot15
0           0  7129300520  20141013T000000  ... -122.257           1340        5650
1           1  6414100192  20141209T000000  ... -122.319           1690        7639
2           2  5631500400  20150225T000000  ... -122.233           2720        8062
3           3  2487200875  20141209T000000  ... -122.393           1360        5000
4           4  1954400510  20150218T000000  ... -122.045           1800        7503
```

```
#concise summary of dataframe
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 22 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Unnamed: 0     21613 non-null  int64
 1   id             21613 non-null  int64
 2   date           21613 non-null  object
 3   price          21613 non-null  float64
 4   bedrooms       21600 non-null  float64
 5   bathrooms      21603 non-null  float64
 6   sqft_living    21613 non-null  int64
 7   sqft_lot       21613 non-null  int64
 8   floors         21613 non-null  float64
 9   waterfront     21613 non-null  int64
 10  view           21613 non-null  int64
 11  condition      21613 non-null  int64
 12  grade          21613 non-null  int64
 13  sqft_above     21613 non-null  int64
 14  sqft_basement  21613 non-null  int64
 15  yr_built       21613 non-null  int64
 16  yr_renovated   21613 non-null  int64
 17  zipcode        21613 non-null  int64
 18  lat            21613 non-null  float64
 19  long           21613 non-null  float64
 20  sqft_living15  21613 non-null  int64
 21  sqft_lot15     21613 non-null  int64
dtypes: float64(6), int64(15), object(1)
memory usage: 3.6+ MB
```

```
#datatypes present
df.dtypes
#statistical summaryof dataframe
df.describe()
```

```
       Unnamed: 0            id  ...  sqft_living15     sqft_lot15
count   21613.00000  2.161300e+04  ...   21613.000000   21613.000000
mean    10806.00000  4.580302e+09  ...    1986.552492   12768.455652
std      6239.28002  2.876566e+09  ...     685.391304   27304.179631
min         0.00000  1.000102e+06  ...     399.000000     651.000000
25%      5403.00000  2.123049e+09  ...    1490.000000    5100.000000
50%     10806.00000  3.904930e+09  ...    1840.000000    7620.000000
75%     16209.00000  7.308900e+09  ...    2360.000000   10083.000000
max     21612.00000  9.900000e+09  ...    6210.000000  871200.000000

[8 rows x 21 columns]
```

Data Wrangling

```
#dropping columns that are not required
df.drop(['id', 'Unnamed: 0'], axis=1, inplace=True)

#handling missing values
#checking if there are any missing values present and in which columns
nan_columns = df.columns[df.isnull().any()].tolist()
print('Bedroom null count: ',df['bedrooms'].isnull().sum(),' Bathroom null count: ',df
['bathrooms'].isnull().sum())

#replacing null values with average value of the column
for i in nan_columns:
    df[i].replace(np.nan, df[i].mean(), inplace=True)

#binning price into groups: low, medium, high
bins = np.linspace(min(df['price']), max(df['price']),4)
group_names = ['low','medium','high']
df['binned-price']=pd.cut(df['price'], bins, labels=group_names, include_lowest=True)
```

EDA

```
#EXPLORATORY DATA ANALYSIS
#count of houses by price category
df['binned-price'].value_counts()
#count of houses by number of floors
df['floors'].value_counts()
```

```
1.0    10680
2.0     8241
1.5     1910
3.0      613
2.5      161
3.5        8
Name: floors, dtype: int64
```

```
low       21531
medium       76
high          6
Name: binned-price, dtype: int64
```

```
#analysing price for houses with or without waterfront view
df['waterfront'].value_counts()
df_group_1 = df[['waterfront','price']].groupby(['waterfront'],as_index=True).mean()
new_index = ['No','Yes']
df_group_1.index=new_index
df_group_1 = df_group_1.rename(columns={'price': 'average price'})
```

```
     average price
No    5.315636e+05
Yes   1.661876e+06
```

```
#checking for outliers for waterfront view
sns.boxplot(x='waterfront', y='price', data=df)
```

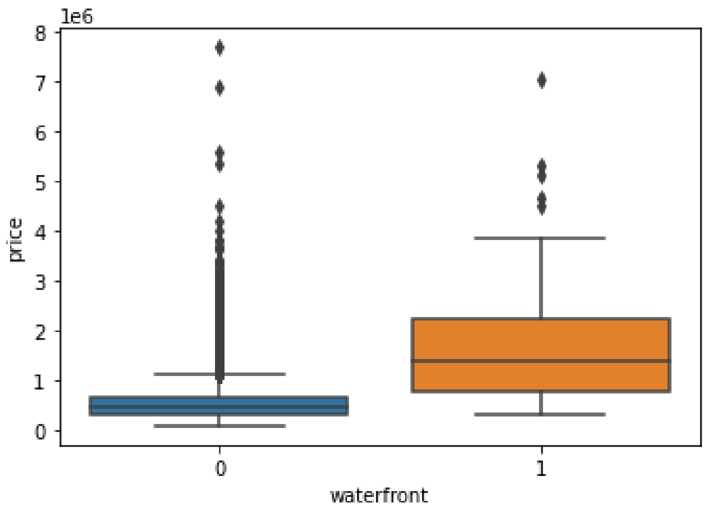


***Figure*: Price outliers for presence and absence of waterfront view**

```
#correlation of each variable with one another using heatmap
corr_heatmap_plot = sns.heatmap(df.corr(),cmap="YlGnBu", annot=False)
plt.show()
```

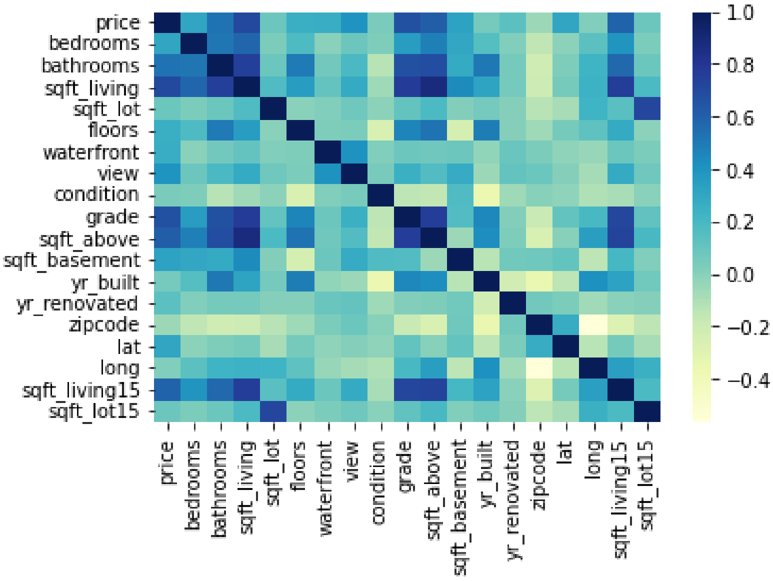
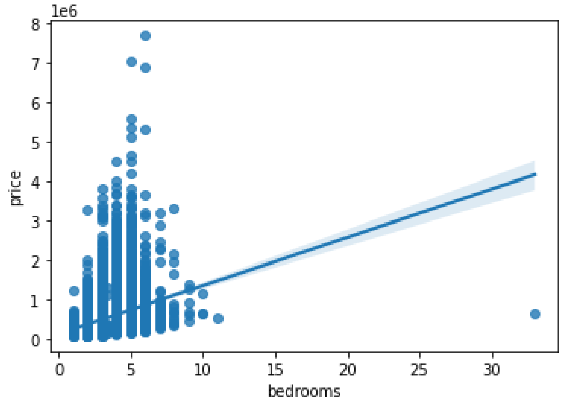


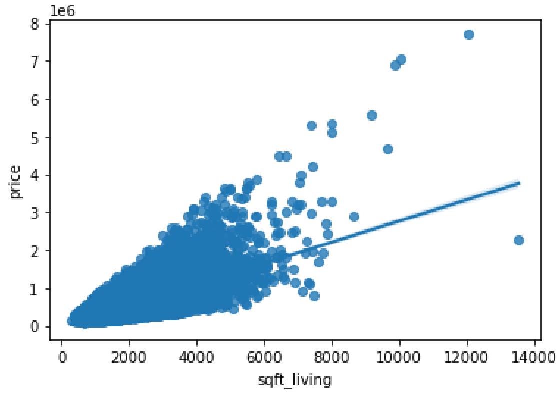
Figure*: *Correlation Heat Map

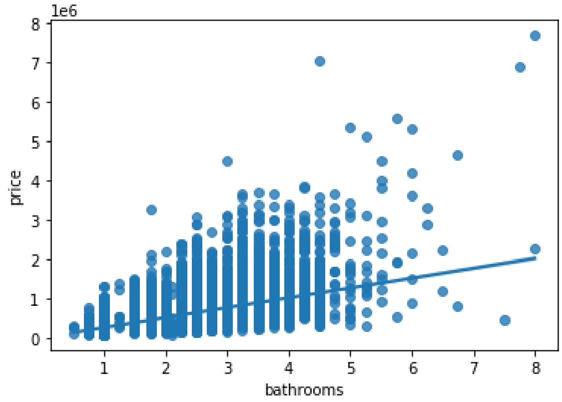
```
#identifying features that are highly correlated with price
df.corr()['price'].sort_values()
```

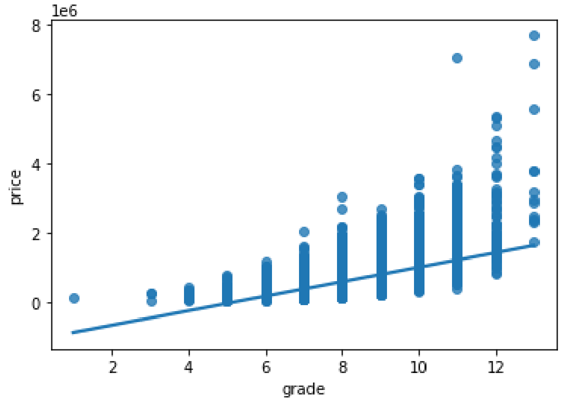
```
zipcode          -0.053203
long              0.021626
condition         0.036362
yr_built          0.054012
sqft_lot15        0.082447
sqft_lot          0.089661
yr_renovated      0.126434
floors            0.256794
waterfront        0.266369
lat               0.307003
bedrooms          0.308797
sqft_basement     0.323816
view              0.397293
bathrooms         0.525738
sqft_living15     0.585379
sqft_above        0.605567
grade             0.667434
sqft_living       0.702035
price             1.000000
Name: price, dtype: float64
```

```
#determining how different features are correlated with price
sns.regplot(x='bedrooms',y='price',data=df)
sns.regplot(x='sqft_living',y='price',data=df)
sns.regplot(x='bathrooms',y='price',data=df)
sns.regplot(x='grade',y='price',data=df)
```










```
#pearson correlation for identifying features that are highly correlated with price
int_float_col = list(df.select_dtypes(include=['int64','float64']).columns)
for col in int_float_col:
    pearson_coef, p_value = stats.pearsonr(df[col], df['price'])
    if p_value<0.05 and (pearson_coef>=0.5 or pearson_coef<=-0.5):
        print(col, ' ', pearson_coef, ' ', p_value)
```

```
price   1.0   0.0
bathrooms   0.5257375111242718   0.0
sqft_living   0.7020350546118   0.0
grade   0.667434256020237   0.0
sqft_above   0.6055672983560781   0.0
sqft_living15   0.585378903579568   0.0
```

Model Development

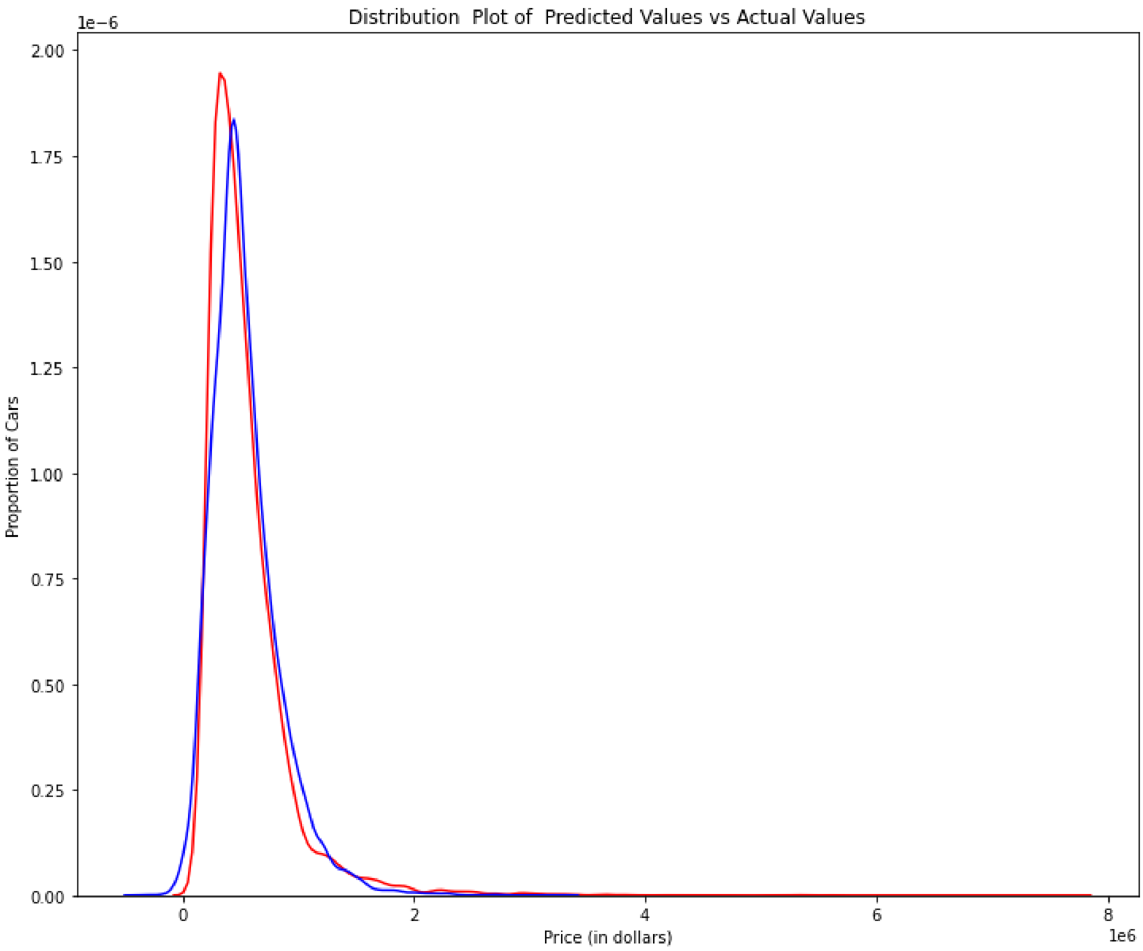
```
#multiple linear regression
features =["floors", "waterfront","lat" ,"bedrooms" ,"sqft_basement" ,"view" ,"bathroo
ms","sqft_living15","sqft_above","grade","sqft_living"]

X = df[features]
Y= df['price']
lm = LinearRegression()
lm.fit(X, Y)
lm.score(X, Y)
```

```
lm.score(X, Y)
0.6576951666037504
```

The default scoring method used is R-square. Multiple linear regression model shows an R-square value of 0.6579

```
#model evaluation using visualisation
Title = 'Distribution  Plot of  Predicted Values vs Actual Values'
DistributionPlot(Y, yhat_m, "Actual Values (Train)", "Predicted Values(Train)", Title)
```



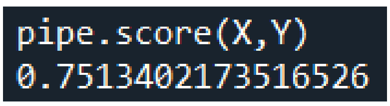
Red curve represents actual values and the blue curve represents predicted values

From the above distribution plot, slight under-fitting is observed and the model could be improved.

Polynomial Regression Model

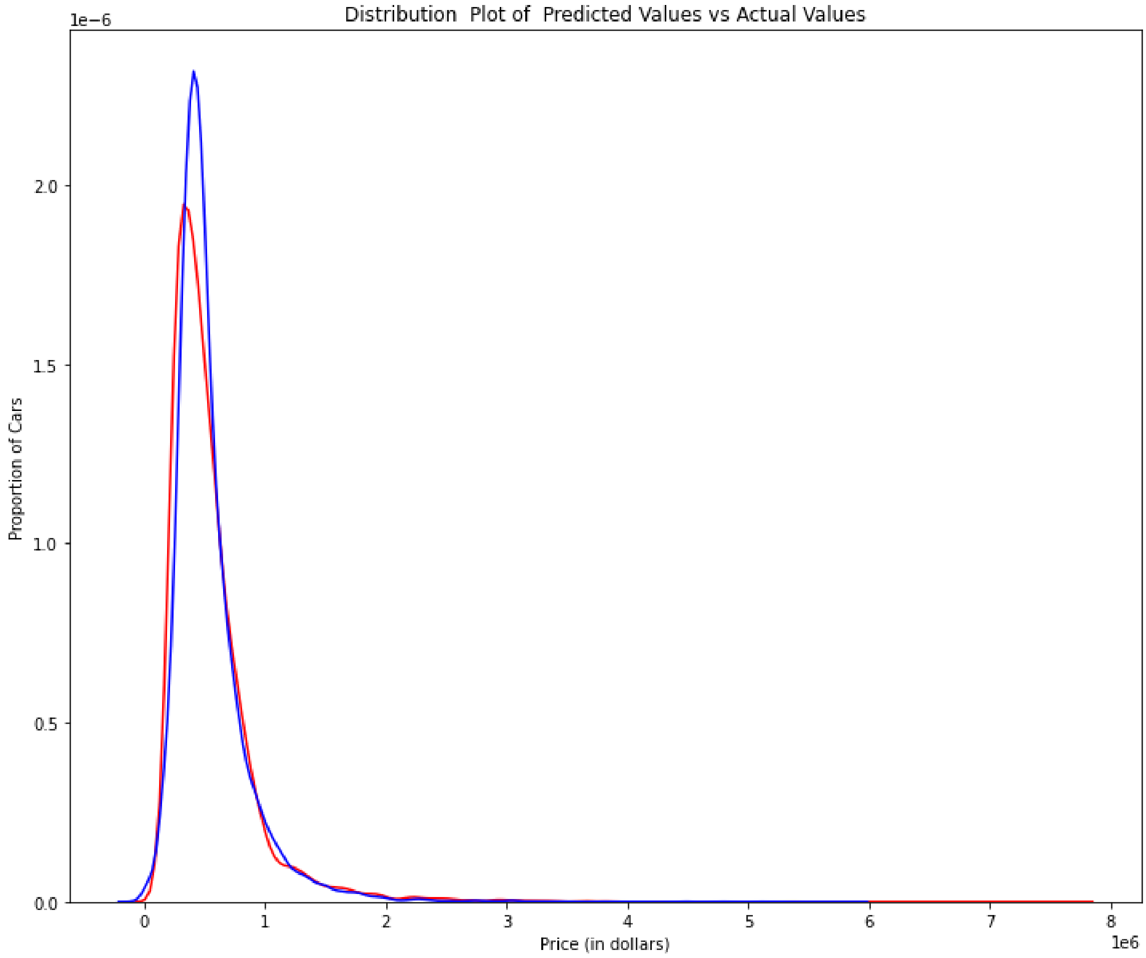
```
Input=[('scale',StandardScaler()),('polynomial',PolynomialFeatures(include_bias=Fals
e)),('model',LinearRegression())]
pipe=Pipeline(Input)
pipe.fit(X,Y)
yhat = pipe.predict(X)

#in-sample evaluation using R^2
pipe.score(X,Y)
```



The polynomial regression shows a better R-square value as compared to multiple linear regression model.

```
#model evaluation using visualisation
Title = 'Distribution  Plot of  Predicted Values vs Actual Values'
DistributionPlot(Y, yhat, "Actual Values (Train)", "Predicted Values(Train)", Title)
```



Red curve represents actual values and the blue curve represents predicted values

However, from the above distribution plot, we observe slight over-fitting of the model.

Model Evaluation and Refinement

We define training and testing datasets.

```
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.15,random_state=
1)
print("number of test samples :", x_test.shape[0])
print("number of training samples:",x_train.shape[0])
```

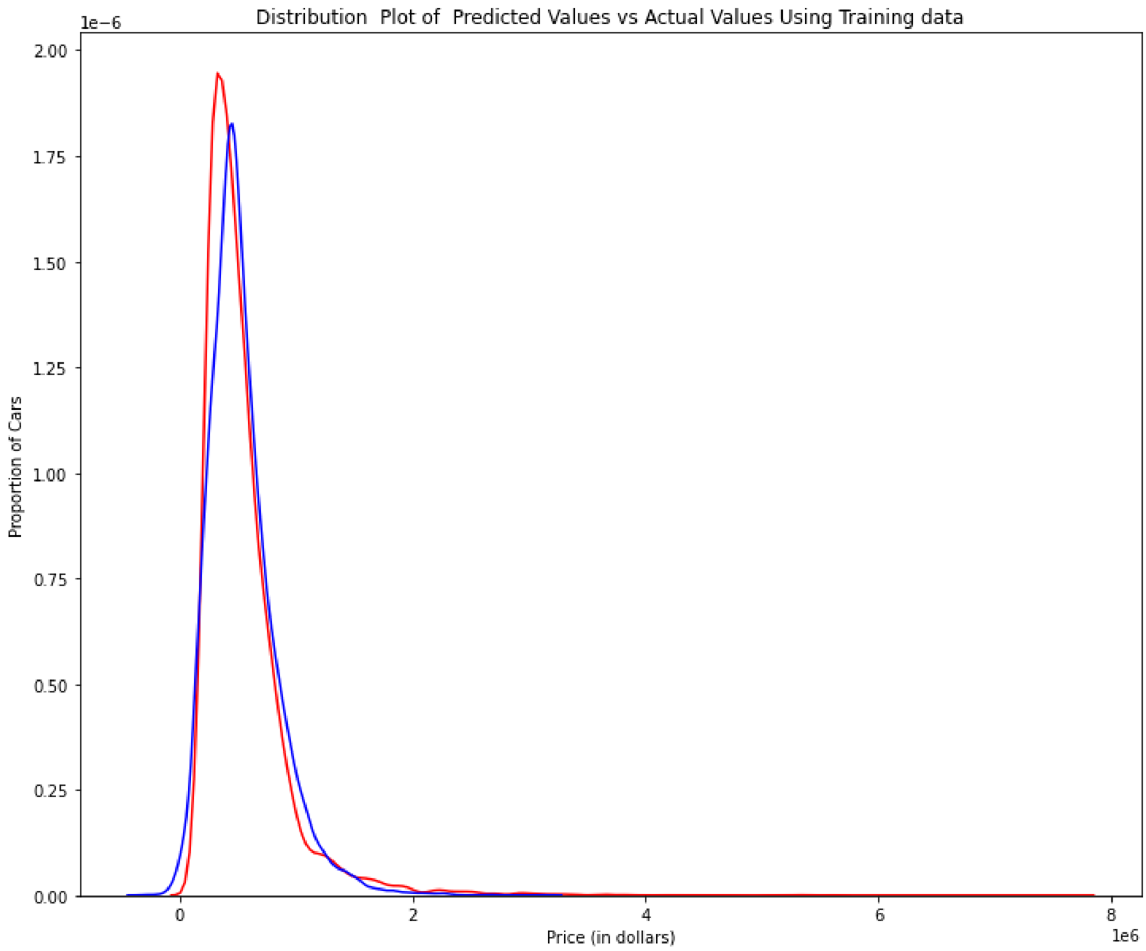
Multiple Regression Model

```
#Multiple regression model
lre = LinearRegression()
lre.fit(x_train, y_train)
lre.score(x_train, y_train)
lre.score(x_test, y_test)
yhat_train = lre.predict(x_train)
```

```
lre.score(x_test, y_test)
0.6478834184390381
```

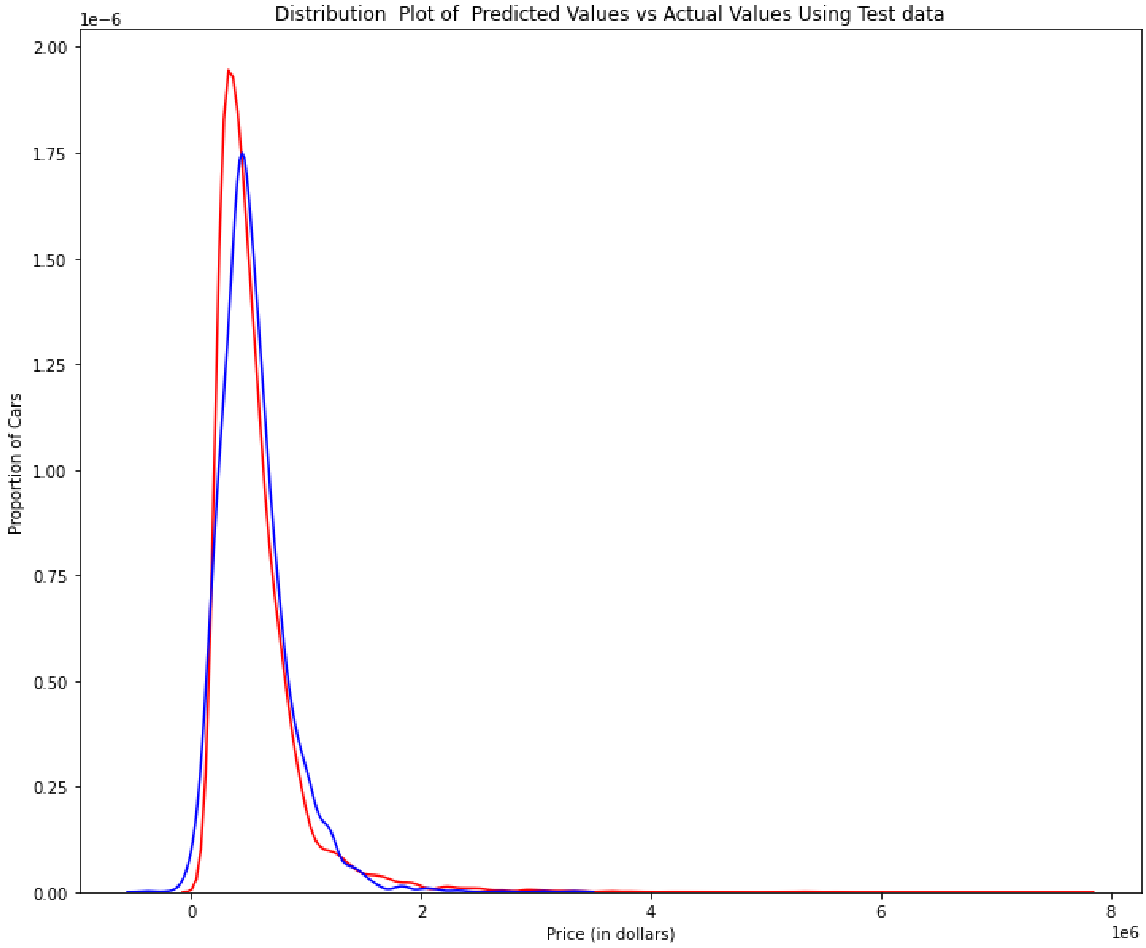
Using the test data, the multiple regression model gives an R-square value of 0.647

```
#plotting distribution of actual training data vs predicted training data
Title = 'Distribution  Plot of  Predicted Values vs Actual Values Using Training data'
DistributionPlot(df['price'], yhat_train, "Actual Values (Train)", "Predicted Values(T
rain)", Title)
```



Red curve represents actual values and the blue curve represents predicted values

```
#plotting distribution of actual test data vs predcited test data
yhat_tt = lre.predict(x_test)
Title = 'Distribution  Plot of  Predicted Values vs Actual Values Using Test data'
DistributionPlot(df['price'], yhat_tt, "Actual Values (Test)", "Predicted Values(Tes
t)", Title)
```



Red curve represents actual values and the blue curve represents predicted values

```
print("Predicted values:", yhat_tt[0:5])
print("True values:", y_test[0:5].values)
```

```
Predicted values: [651758.04355904 514998.00433762 794352.01994903 702660.38974092
 213485.919915   ]
True values: [ 459000.   445000. 1057000.   732350.   235000.]
```

If we use **fewer data points to train** the model & **more to test** the model ⇒ **less accuracy** of generalization performance but **good precision.** If we use **more data points to train** & **less to test** ⇒ **good accuracy** but **poor precision**

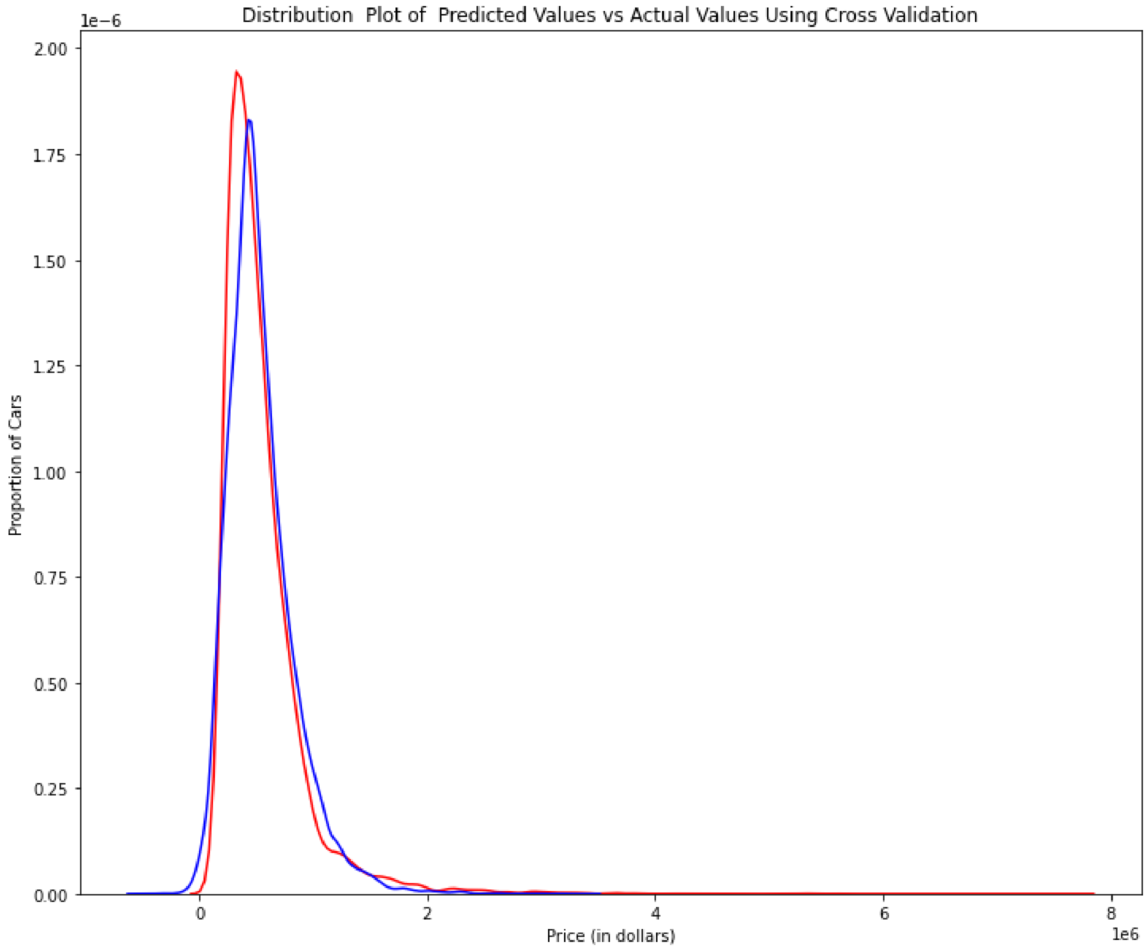
To overcome this, we will use cross-validation.

```
#Cross validation
lrc = LinearRegression()
Rcross = cross_val_score(lrc, X,Y, cv=4)
```

```
print("The mean of the folds are", Rcross.mean())
print("The standard deviation is" , Rcross.std())
yhat_cross = cross_val_predict(lrc, X,Y, cv=4)
```

```
The mean of the folds are 0.6543411661541467
The standard deviation is 0.009211527642024875
```

```
Title = 'Distribution  Plot of Predicted Values vs Actual Values Using Cross Validatio
n'
DistributionPlot(df['price'], yhat_cross, "Actual Values", "Predicted Values", Title)
```



Red curve represents actual values and the blue curve represents predicted values

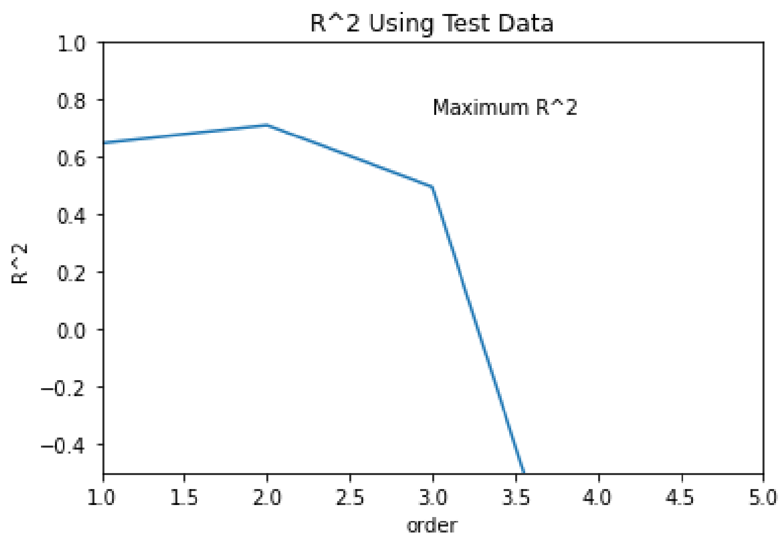
The model shows slightly better fit using cross-validation.

Polynomial Regression Model

```
#Polynomial Regression
#R square test to select the best order
Rsq_test = []
order = [1,2,3,4,5]
for n in order:
    pr = PolynomialFeatures(degree=n)
    x_train_pr = pr.fit_transform(x_train)
    x_test_pr = pr.fit_transform(x_test)
    poly = LinearRegression()
    poly.fit(x_train_pr, y_train)
    Rsq_test.append(poly.score(x_test_pr,y_test))
print(Rsq_test)
```

```
[0.6478834184390174, 0.7117278177317132, 0.4969288880708065, -1.302063386319508,
-310.6969893989896]
```

```
plt.plot(order, Rsq_test)
plt.xlim(1, 5)
plt.ylim(-0.50,1)
plt.xlabel('order')
plt.ylabel('R^2')
plt.title('R^2 Using Test Data')
plt.text(3, 0.75, 'Maximum R^2 ')
```



We get the best value of R-square corresponding to polynomial degree=2.

```
pr = PolynomialFeatures(degree=2, include_bias=False)
x_train_pr = pr.fit_transform(x_train)
x_test_pr = pr.fit_transform(x_test)
```

```
poly = LinearRegression()
poly.fit(x_train_pr, y_train)
poly.score(x_train_pr,y_train)
poly.score(x_test_pr,y_test)
```

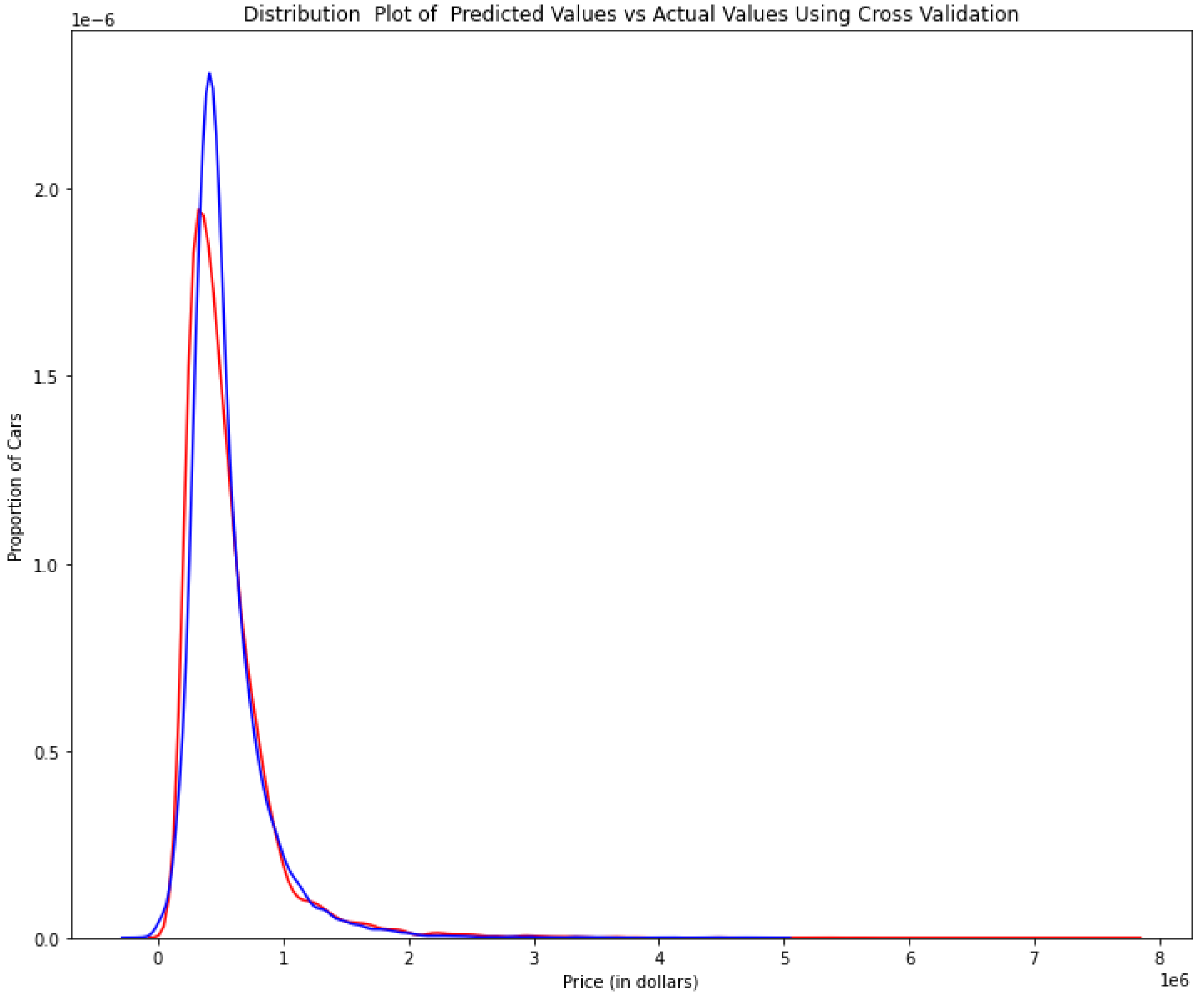
```
poly.score(x_test_pr,y_test)
0.7117277163998375
```

Using Cross Validation

```
X_pr = pr.fit_transform(X)
poly_cross = LinearRegression()
Rcross_poly = cross_val_score(poly_cross,X_pr,Y,cv=3)
print("The mean of the folds are", Rcross_poly.mean())
print("The standard deviation is" , Rcross_poly.std())
```

```
The mean of the folds are 0.7349150299840215
The standard deviation is 0.011823156938641794
```

```
yhat_polycross = cross_val_predict(poly_cross,X_pr,Y,cv=3)
yhat_polycross = cross_val_predict(poly_cross,X_pr,Y,cv=3)
Title = 'Distribution  Plot of Predicted Values vs Actual Values Using Cross Validation'
DistributionPlot(df['price'], yhat_polycross, "Actual Values", "Predicted Values",Title)
```



Red curve represents actual values and the blue curve represents predicted values

```
print("Predicted values:", yhat_polycross[0:4])
print("True values:", y_test[0:4].values)
```

```
Predicted values: [347866.87904453 553843.59248352 461727.5476017  404924.84692764]
True values: [ 459000.   445000. 1057000.   732350.]
```

The value of mean square of the folds is better however, we observe over-fitting in the model. Ro overcome this, we use Ridge Regression.

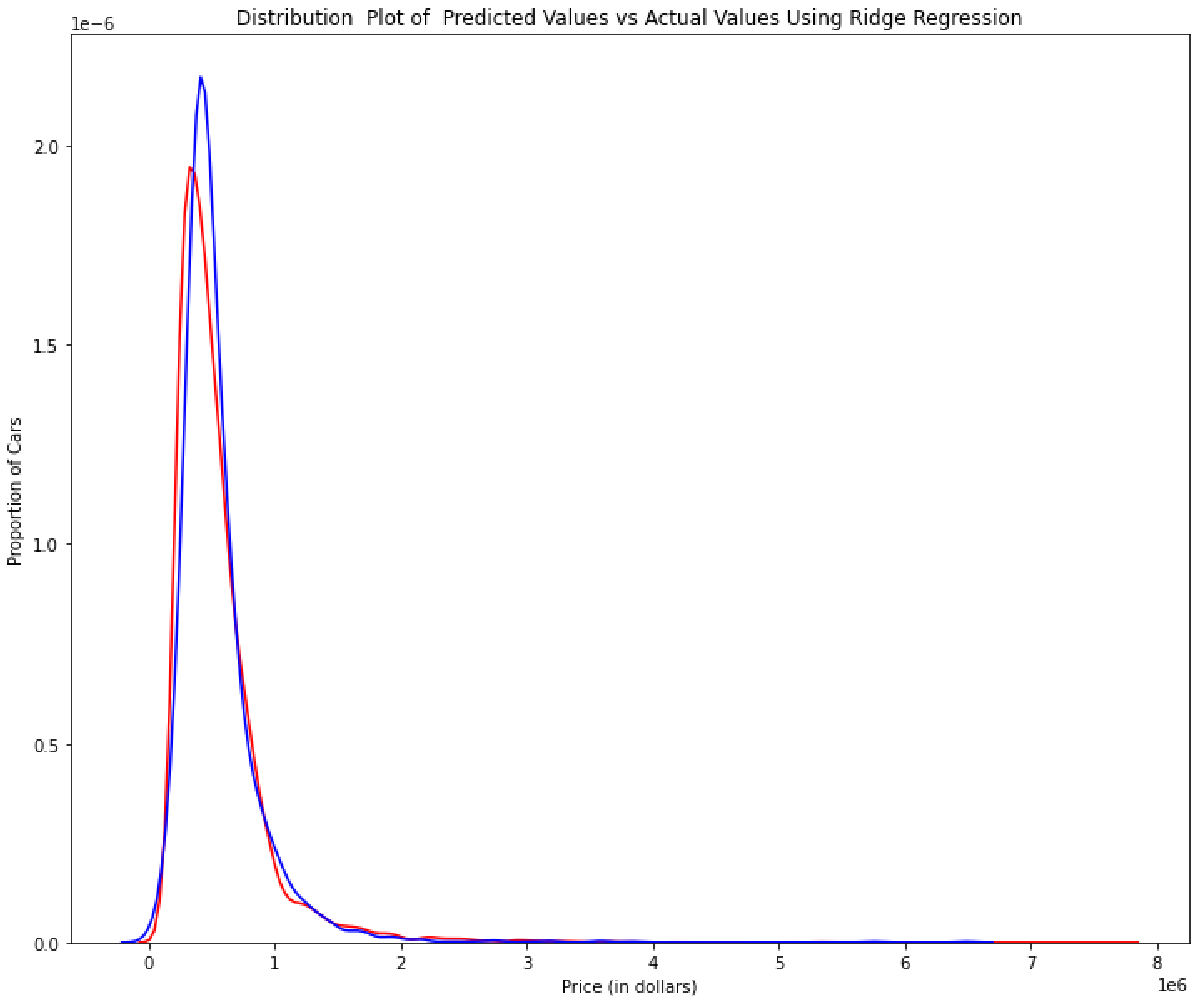
To select the best value of hyperparameter alpha we use GridSearchCV.

```
parameters1 = [{'alpha': [0.00001, 0.0001,0.001,0.01,0.1,1, 10]}]
RigeModel = Ridge()
Grid1 = GridSearchCV(RigeModel, parameters1,cv=3)
Grid1
Grid1.fit(x_train_pr,y_train)
```

```
BestRigeModel = Grid1.best_estimator_
BestRigeModel
```

```
BestRigeModel = Grid1.best_estimator_
BestRigeModel
Ridge(alpha=1e-05)
```

```
yhat_ridge = BestRigeModel.predict(x_test_pr)
Title = 'Distribution  Plot of Predicted Values vs Actual Values Using Ridge Regressio
n'
DistributionPlot(df['price'], yhat_ridge, "Actual Values", "Predicted Values", Title)
```




```
print("Predicted values:", yhat_ridge[0:4])
print("True values:", y_test[0:4].values)
```

```
Predicted values: [588772.49611092 452203.22577858 635608.21862602 701415.72738266]
True values: [ 459000.  445000. 1057000.  732350.]
```

```
BestRigeModel.score(x_test_pr,y_test)
```

```
BestRigeModel.score(x_test_pr,y_test)
0.7117044725650039
```

We observe that the model obtained by setting the value of hyperparameter alpha = 0.00001 gives more accurate prediction. It has been used to control the magnitude of estimated polynomial coefficients. The value of R-square obtained using ridge regression is as high as 0.7117.