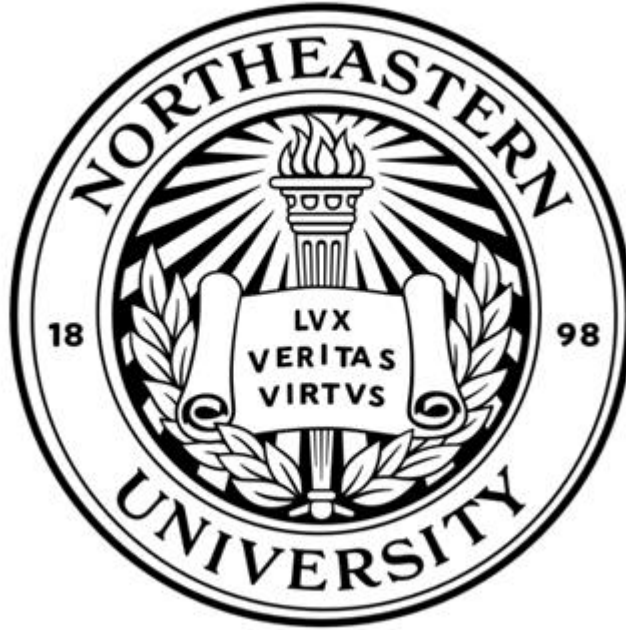


Housing Rate Prediction Model



Deepti Sonth - 001448813

Naqiyah Lakdawala - 001449938

Priyanshi Agrawal - 001840704

Rajavi Mehta – 001057845

Data Science Engineering Methods and Tools

Professor Handan Liu

Table of Contents:

1.0	Introduction.....	2
2.0	Methodology.....	3
3.0	Specifications or Description of our dataset.....	4
4.0	Exploration data Analysis.....	6
5.0	Clustering.....	20
6.0	Linear Regression for Multiple Variables.....	25
7.0	Ridge and Lasso Regression.....	34
8.0	Random Forest Regressor.....	42
9.0	Gradient Boosting Regressor.....	53
10.0	Hypertuning of Parameters.....	64
11.0	Conclusion.....	64
12.0	Future Scope.....	65
13.0	Reference.....	66

1.0 Introduction:

Overview

To predict the average price of a house in a block based on several features in real - life scenarios. In all model building assumptions are made, and certain conditions also required to be approximately met for purposes of estimation and meeting certain accuracy levels..

This dataset was derived from the 1990 U.S. census, using one row per census block group. A block group is the smallest geographical unit for which the U.S.Census Bureau publishes sample data

There are 34858 instances and 20 numeric, predictive attributes and 1 target variable

The target variable is the house price that is to be predicted.

Motivations/ Objectives:

The United States housing bubble was a real estate bubble affecting over half of the U.S. states. Housing prices peaked in early 2006, started to decline in 2006 and 2007, and reached new lows in 2012. Affecting the world's real estate conditions to worsen. Having a predictive model in which the housing rate of the houses is determined by overall variables, surrounding the region provides a sense of stability in the market and overall financial stability for an individual. Thus the objectives for the project is to provide top most accuracy for the target variable i.e the median house rate value according to the surrounding parameters such as the population of the region, the age factor of the population, the region and house properties.

2.0 Methodology:

- 1. Exploratory Data Analysis:** The Dataset we are using is a housing rate data set which allows us to predict the accuracy of the median of house rates value according to the parameters available to us in the data set. In EDA , we clean and explore each feature used in the dataset and detect anomalies
- 2. Feature Engineering:** Transforming the columns and handling null values
- 3. Feature Selection:** Selecting only the relevant variables for our prediction
- 4. Correlation of data variables:** Finding correlation between different variables in a dataset
- 5. Supervised Learning Data Algorithms** we plan to use are - Decision Tree Regression and regression techniques like random forest and gradient boosting
- 6. For Unsupervised Learning Data** we plan to use are-Clustering or Dimension Structure Reduction
- 7. Visualization:** Plotting and analysing of Descriptive and target variables using matplotlib and Seaborn
- 8. Hypertuning of Parameters:** To increase the accuracy of model predictions
- 9. Analyze Residuals:** for the dataset - To analyze if the model residuals well- behaved

3.0 Specifications/Description of our dataset:

The data pertains to the houses found in a given district and some summary statistics about them based on the 1990 census data. The columns are as follows:

The dataset consists of the following columns:

- **Suburb:** Suburb where the house is located
- **Address:** Address of the house
- **Rooms:** Number of rooms in the house
- **Price:** Price of the house in dollars
- **Method:**
 - S - property sold;
 - SP - property sold prior;
 - PI - property passed in;
 - PN - sold prior not disclosed;
 - SN - sold not disclosed;
 - NB - no bid;
 - VB - vendor bid;
 - W - withdrawn prior to auction;
 - SA - sold after auction;
 - SS - sold after auction price not disclosed.
 - N/A - price or highest bid not available.
- **Type:**
 - br - bedroom(s);
 - h - house,cottage,villa, semi,terrace;
 - u - unit, duplex;
 - t - townhouse; dev site - development site;
 - o res - other residential.
- **SellerG:** Real Estate Agent
- **Date:** Date the house was sold
- **Distance:** Distance from Central Business District
- **Regionname:** General Region (West, North West, North, North east ...etc)
- **Propertycount:** Number of properties that exist in the suburb.
- **Bedroom2 :** Number of Bedrooms in the house
- **Bathroom:** Number of Bathrooms in the house
- **Car:** Number of carspots in the building available to the residents
- **Landsize:** Size of the Land
- **BuildingArea:** Size of the Building

- **YearBuilt:** Year the house was built
- **CouncilArea:** Governing council for the area
- **Latitude:** Location of the building latitude
- **Longitude:** Location of the building in longitude
- **Postcode :** Postcode of the location of the houses

4.0 Exploration data Analysis

The objective of this project is to apply exploratory analysis and regression techniques to identify which features affect home prices the most in the Housing Market.

The first step is to load the data and gain a better understanding of the information each column contains.

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline
```

```
dataset = pd.read_csv('C:\\Users\\naqiy\\Desktop\\Data science\\Melbourne_housing_FULL.csv')
```

dataset

Out[172]:

	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distance	Postcode	...	Bathroom	Car	Landsize	BuildingArea
0	Abbotsford	68 Studley St	2	h	NaN	SS	Jellis	3/09/2016	2.5	3067.0	...	1.0	1.0	126.0	NaN
1	Abbotsford	85 Turner St	2	h	1480000.0	S	Biggin	3/12/2016	2.5	3067.0	...	1.0	1.0	202.0	NaN
2	Abbotsford	25 Bloomburg St	2	h	1035000.0	S	Biggin	4/02/2016	2.5	3067.0	...	1.0	0.0	156.0	78
3	Abbotsford	18/659 Victoria St	3	u	NaN	VB	Rounds	4/02/2016	2.5	3067.0	...	2.0	1.0	0.0	NaN
4	Abbotsford	5 Charles St	3	h	1465000.0	SP	Biggin	4/03/2017	2.5	3067.0	...	2.0	0.0	134.0	150
...
34852	Yarraville	13 Burns St	4	h	1480000.0	PI	Jas	24/02/2018	6.3	3013.0	...	1.0	3.0	593.0	NaN
34853	Yarraville	29A Murray St	2	h	888000.0	SP	Sweeney	24/02/2018	6.3	3013.0	...	2.0	1.0	98.0	104
34854	Yarraville	147A Severn St	2	t	705000.0	S	Jas	24/02/2018	6.3	3013.0	...	1.0	2.0	220.0	120
34855	Yarraville	12/37 Stephen St	3	h	1140000.0	SP	hockingstuart	24/02/2018	6.3	3013.0	...	NaN	NaN	NaN	NaN
34856	Yarraville	3 Tarrengower St	2	h	1020000.0	PI	RW	24/02/2018	6.3	3013.0	...	1.0	0.0	250.0	103

34857 rows × 21 columns



▶ `dataset.info()`

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 34857 entries, 0 to 34856
Data columns (total 21 columns):
Suburb          34857 non-null object
Address         34857 non-null object
Rooms           34857 non-null int64
Type            34857 non-null object
Price           27247 non-null float64
Method          34857 non-null object
SellerG         34857 non-null object
Date            34857 non-null object
Distance        34856 non-null float64
Postcode        34856 non-null float64
Bedroom2        26640 non-null float64
Bathroom        26631 non-null float64
Car             26129 non-null float64
Landsize        23047 non-null float64
BuildingArea    13742 non-null float64
YearBuilt       15551 non-null float64
CouncilArea     34854 non-null object
Lattitude       26881 non-null float64
Longitude       26881 non-null float64
Regionname      34854 non-null object
Propertycount   34854 non-null float64
dtypes: float64(12), int64(1), object(8)
memory usage: 5.6+ MB
```

Now are primary variable is the price of the housing so noticing the data we cannot afford to have any null values in the main category so we eliminate the null/ missing values of the variable

```
#cleaning predictive variable "Y" by taking non-null values only
df=dataset[dataset['Price'].notna()]
```

```
| df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 27247 entries, 1 to 34856
Data columns (total 21 columns):
Suburb          27247 non-null object
Address         27247 non-null object
Rooms           27247 non-null int64
Type            27247 non-null object
Price           27247 non-null float64
Method          27247 non-null object
SellerG         27247 non-null object
Date            27247 non-null object
Distance        27246 non-null float64
Postcode        27246 non-null float64
Bedroom2        20806 non-null float64
Bathroom        20800 non-null float64
Car             20423 non-null float64
Landsize        17982 non-null float64
BuildingArea    10656 non-null float64
YearBuilt       12084 non-null float64
CouncilArea     27244 non-null object
Lattitude       20993 non-null float64
Longtitude      20993 non-null float64
Regionname      27244 non-null object
Propertycount   27244 non-null float64
dtypes: float64(12), int64(1), object(8)
memory usage: 4.6+ MB
```

Now, Looking at the data we can see that the price of the house always depends on the numbers of bedrooms and as the number of rooms it has so we tried to derive to a conclusion and analysed that unique number of rooms can be determined by calculating Rooms - bedrooms but it showed us negative numerical numbers which is not possible making Bedroom2 a faulty and nondeterministic columns which compelled us to drop these values as well for fault tolerant

predictions of the data.

```
df['Rooms_without_Bedrooms'] = df['Rooms'] - df['Bedroom2']  
df
```

Rooms_without_Bedrooms	
	0.0
	0.0
	0.0
	0.0
	1.0
	...
	0.0
	0.0
	0.0
	NaN
	0.0

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

```
array([ 0.,  1., -1., nan,  2., -2., -3.,  3.,  4., -5., -6.,  
       -17., -7.])
```

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 27247 entries, 1 to 34856
Data columns (total 22 columns):
Suburb                27247 non-null object
Address              27247 non-null object
Rooms                27247 non-null int64
Type                 27247 non-null object
Price                27247 non-null float64
Method               27247 non-null object
SellerG              27247 non-null object
Date                 27247 non-null object
Distance             27246 non-null float64
Postcode             27246 non-null float64
Bedroom2             20806 non-null float64
Bathroom             20800 non-null float64
Car                  20423 non-null float64
Landsize             17982 non-null float64
BuildingArea         10656 non-null float64
YearBuilt            12084 non-null float64
CouncilArea          27244 non-null object
Lattitude            20993 non-null float64
Longtitude           20993 non-null float64
Regionname           27244 non-null object
Propertycount        27244 non-null float64
Rooms_without_Bedrooms 20806 non-null float64
dtypes: float64(13), int64(1), object(8)
memory usage: 4.8+ MB
```

```
df = df.drop(['Bedroom2', 'Rooms_without_Bedrooms'],1)
```

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

```
)]: array([ 1.,  2., nan,  3.,  4.,  0.,  7.,  5.,  6.,  8.,  9.])
```

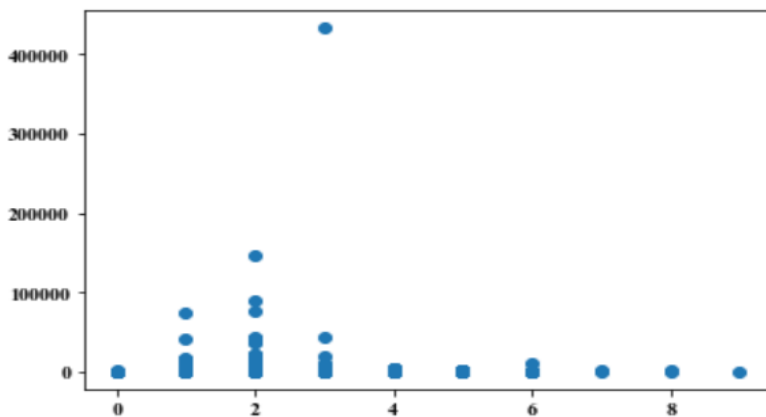
```
df['Bathroom'].value_counts()
```

```
)]: 1.0    10527  
     2.0    8464  
     3.0    1537  
     4.0     176  
     5.0      46  
     0.0      34  
     6.0      10  
     8.0        3  
     7.0        2  
     9.0        1  
     Name: Bathroom, dtype: int64
```

After Eliminating the bedroom , we concluded that the bathroom column is important as some instances consisted of more bathrooms than bedrooms which contributed to the building area and landsize of the houses

```
plt.scatter(df['Bathroom'],df['Landsize'])
```

```
<matplotlib.collections.PathCollection at 0x1c55bd863c8>
```



The next analysis we made that the land size and the building area of the houses had some discrepancies which led us to make changes to the dataset, we derived a column to eliminate the null values and brought down the dataset . We also observed that in some rows the building area was greater than the land size which is not possible and had false values which forced us to eliminate these values and thus making the dataset cleaner we calculated these values and eliminated those.

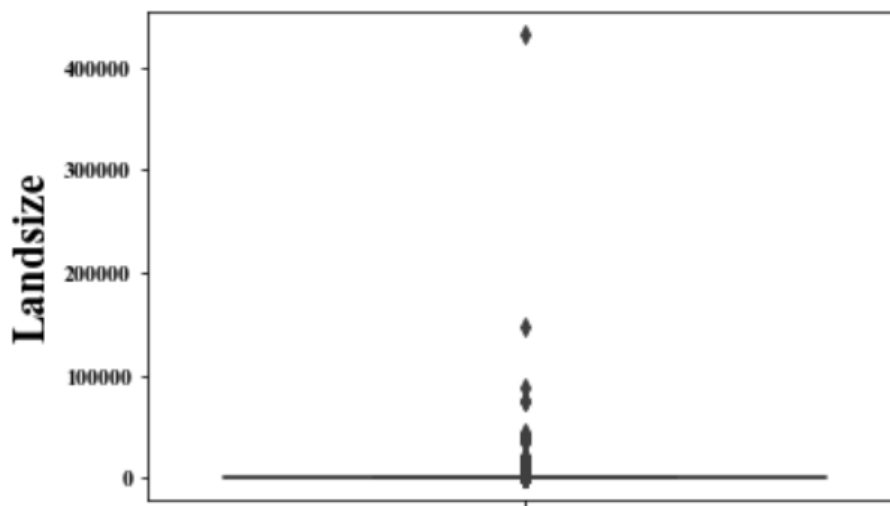
```
df['BuildingArea']=df['BuildingArea'].fillna(0)
```

```
df['BuildingArea'].loc[df.BuildingArea<1].count()
```

```
4]: 16652
```

```
sns.boxplot(data = df, y = 'Landsize')
```

```
5]: <matplotlib.axes._subplots.AxesSubplot at 0x1c559e05f48>
```



```
df = df[~(df['BuildingArea'] < 1)]  
#check the deletion  
df['BuildingArea'].loc[df.BuildingArea<1].count()
```

```
: 0
```

▶ `df.info()`

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10595 entries, 0 to 10594
Data columns (total 21 columns):
index          10595 non-null int64
Suburb         10595 non-null object
Address        10595 non-null object
Rooms          10595 non-null int64
Type           10595 non-null object
Price          10595 non-null float64
Method         10595 non-null object
SellerG        10595 non-null object
Date           10595 non-null object
Distance       10595 non-null float64
Postcode       10595 non-null float64
Bathroom       10595 non-null float64
Car            10420 non-null float64
Landsize       9334 non-null float64
BuildingArea   10595 non-null float64
YearBuilt      10189 non-null float64
CouncilArea    10595 non-null object
Lattitude      10586 non-null float64
Longitude      10586 non-null float64
Regionname     10595 non-null object
Propertycount  10595 non-null float64
dtypes: float64(11), int64(2), object(8)
```

| `df['BA-LS'] = (df['BuildingArea'] - df['Landsize'])`

```
Date          10595 non-null object
Distance      10595 non-null float64
Postcode      10595 non-null float64
Bathroom      10595 non-null float64
Car           10420 non-null float64
Landsize      9334 non-null float64
BuildingArea  10595 non-null float64
YearBuilt     10189 non-null float64
CouncilArea   10595 non-null object
Latitude      10586 non-null float64
Longitude     10586 non-null float64
Regionname    10595 non-null object
Propertycount 10595 non-null float64
BA-LS         9334 non-null float64
dtypes: float64(12), int64(2), object(8)
memory usage: 1.8+ MB
```

```
] :  ► df=df.drop(['BA-LS'], axis=1)
```

```
df1= df[df['Landsize'].notna()]
df1
```

```
df1['SUB']= df1['Landsize']-df1['BuildingArea']
df1
```

```
▶ df1 = df1[~(df1['SUB'] < 0)]  
#check the deletion  
df1['SUB'].loc[df1.SUB<0].count()  
df1=df1.reset_index()
```



```
df1.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7961 entries, 0 to 7960
Data columns (total 23 columns):
level_0      7961 non-null int64
index        7961 non-null int64
Suburb       7961 non-null object
Address      7961 non-null object
Rooms        7961 non-null int64
Type         7961 non-null object
Price        7961 non-null float64
Method       7961 non-null object
SellerG      7961 non-null object
Date         7961 non-null object
Distance     7961 non-null float64
Postcode     7961 non-null float64
Bathroom     7961 non-null float64
Car          7843 non-null float64
Landsize     7961 non-null float64
BuildingArea 7961 non-null float64
YearBuilt    7687 non-null float64
CouncilArea  7961 non-null object
Latitude     7954 non-null float64
Longitude    7954 non-null float64
Regionname   7961 non-null object
Propertycount 7961 non-null float64
SUB          7961 non-null float64
```

Here we added the extra SUB , substitute column which gives us no null values and the correct values of the Landsize as the building area cannot be bigger than the size of the land so only valid values remain.

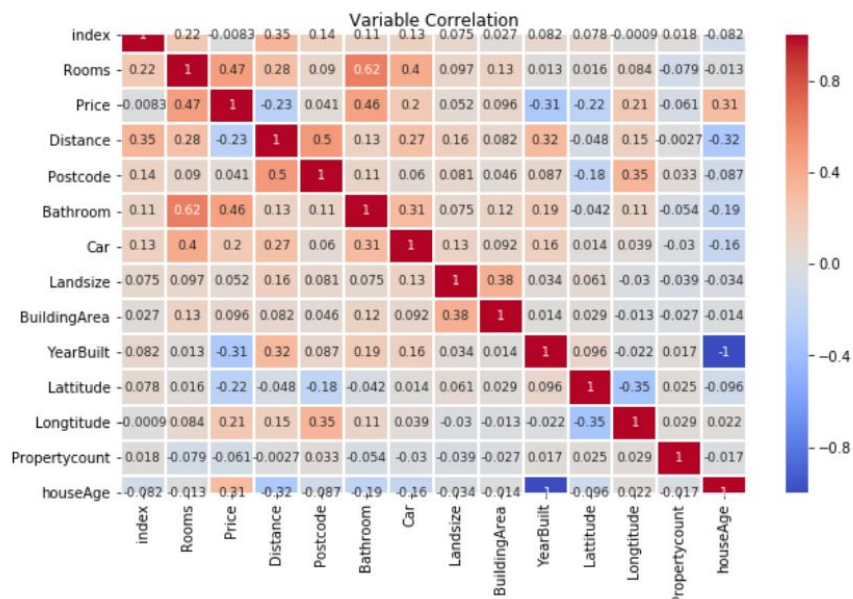
In the end to determine the price of the house , we have to determine how old any property is so for that we derived a columns which tells us the age of the house by calculating the present year - the year it was built-in so here it is,

```
df1['houseAge'] = 2020-df1['YearBuilt']
```

After Analysing the columns who are most relative to the pricing variable we plotted a heat map to see if we can draw more conclusions

```
In [65]: plt.figure(figsize=(10,6))
sns.heatmap(df1.corr(),cmap = 'coolwarm',linewidth = 1,annot= True, annot_kws={"size": 9})
plt.title('Variable Correlation')
```

Out[65]: Text(0.5, 1, 'Variable Correlation')



Here we realized that all the values contributing to the significance of the pricing of the house are all numerical , and we cannot only depend on numerical values for correlation , therefore to draw more narratives in the analysis we decided to include some categorical values also which are in different types of datatypes.

```
In [210]: df1['Regionname'].unique()

Out[210]: array(['Northern Metropolitan', 'Western Metropolitan',
                'Southern Metropolitan', 'Eastern Metropolitan',
                'South-Eastern Metropolitan', 'Northern Victoria',
                'Eastern Victoria', 'Western Victoria'], dtype=object)

In [211]: df1['Method'].unique()

Out[211]: array(['S', 'VB', 'PI', 'SP', 'SA'], dtype=object)
```

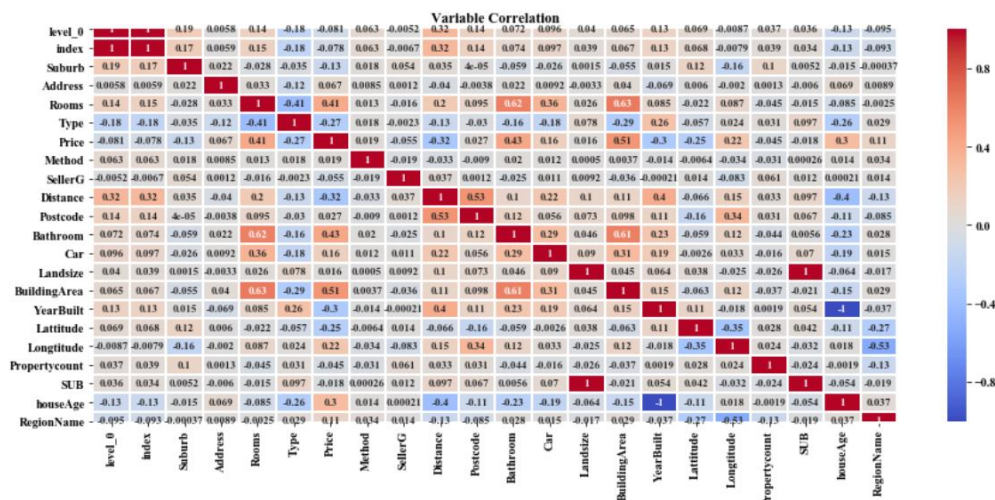
So To correlate the significance we converted the categorical values using catcode

```
df2=df1
df2['RegionName']=df2['Regionname'].astype('category').cat.codes
df2['Method']=df2['Method'].astype('category').cat.codes
df2['SellerG']=df2['SellerG'].astype('category').cat.codes
df2['Type']=df2['Type'].astype('category').cat.codes
df2['Suburb']=df2['Suburb'].astype('category').cat.codes
df2['Address']=df2['Address'].astype('category').cat.codes
df2.corr()
```

And After the categorical values were able to contribute to the correlation of the pricing we removed the correlation matrix heat map again and saw....

```
In [246]: plt.figure(figsize=(15,6))
sns.heatmap(df2.corr(), cmap = 'coolwarm', linewidth = 1, annot= True, annot_kws={"size": 9})
plt.title('Variable Correlation')

Out[246]: Text(0.5, 1, 'Variable Correlation')
```



Here, we finalized our prime variables that contributed to the variables and decided that the

range could be -0.3 to $+0.3 < 1$ for correlation . And we had a new data set prepared which included the following columns for prime predictions in the model

```
In [217]: dataset = df2[['Rooms', 'Bathroom', 'BuildingArea', 'houseAge', 'YearBuilt', 'Distance', 'Price', 'Type']]
dataset
```

```
Out[217]:
```

	Rooms	Bathroom	BuildingArea	houseAge	YearBuilt	Distance	Price	Type
0	2	1.0	79.0	120.0	1900.0	2.5	1035000.0	0
1	3	2.0	210.0	110.0	1910.0	2.5	1876000.0	0
2	2	1.0	107.0	130.0	1890.0	2.5	1636000.0	0
3	2	1.0	75.0	120.0	1900.0	2.5	1097000.0	0
4	3	2.0	190.0	15.0	2005.0	2.5	1350000.0	0
...
7956	3	2.0	158.0	25.0	1995.0	6.8	1400000.0	0
7957	3	2.0	118.0	4.0	2016.0	25.5	500000.0	0
7958	3	2.0	158.0	8.0	2012.0	25.5	570000.0	0
7959	2	1.0	120.0	20.0	2000.0	6.3	705000.0	1
7960	2	1.0	103.0	90.0	1930.0	6.3	1020000.0	0

7961 rows × 8 columns

5.0 Clustering

The next step would be to treat every row of the data as vectors in pursuit of finding properties which are similar to each other. The algorithm used to achieve this feat is K means Clustering. For each cluster, you will place a point(a centroid) in space and the vectors are grouped based on their proximity to their nearest centroid. The following steps are used in clustering-

1. **Cleaning Data:** For clustering, your data must be indeed integers. Moreover, since k-means is using Euclidean distance, having categorical columns is not a good idea. I normalized my dataset which would have a certain set of variables chosen. All the none values were either imputed or removed in the previous step. The variable 'Date' in its standard format will not be fed well in the model so I made a new column which encodes the month of the date present and dropped the former.
2. **Choose Number of Clusters and Standardize Data -** For choosing the number of clusters k, I use the Elbow method. The Elbow method tries different values of k and plot the average distance of data points from their respective centroid (average within-cluster sum of squares) as a function of k. We look for the "elbow" point where increasing the k value drastically lowers the average distance of each data point to its respective centroid, but as you continue increasing k the improvements begin to decrease asymptotically. The ideal k value is found at the elbow of such a plot. I have also standardized the data to counter the high influence by variables having unusual scales then the rest.

```

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from scipy import stats
from sklearn.cluster import KMeans
from sklearn import metrics
from sklearn.preprocessing import scale
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Ridge, RidgeCV, Lasso, LassoCV
from sklearn.metrics import mean_squared_error
%matplotlib inline

```

```

dataset_c = dataset[['Type', 'Price']]
from sklearn import preprocessing
x = dataset_c #returns a numpy array
min_max_scaler = preprocessing.MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(x)
df_c = pd.DataFrame(x_scaled)
df_c.head()

```

```

1]:

```

	0	1
0	0.0	0.101928
1	0.0	0.196753
2	0.0	0.169692
3	0.0	0.108919
4	0.0	0.137445

```

# Choosing the optimal k
from scipy.spatial.distance import cdist, pdist

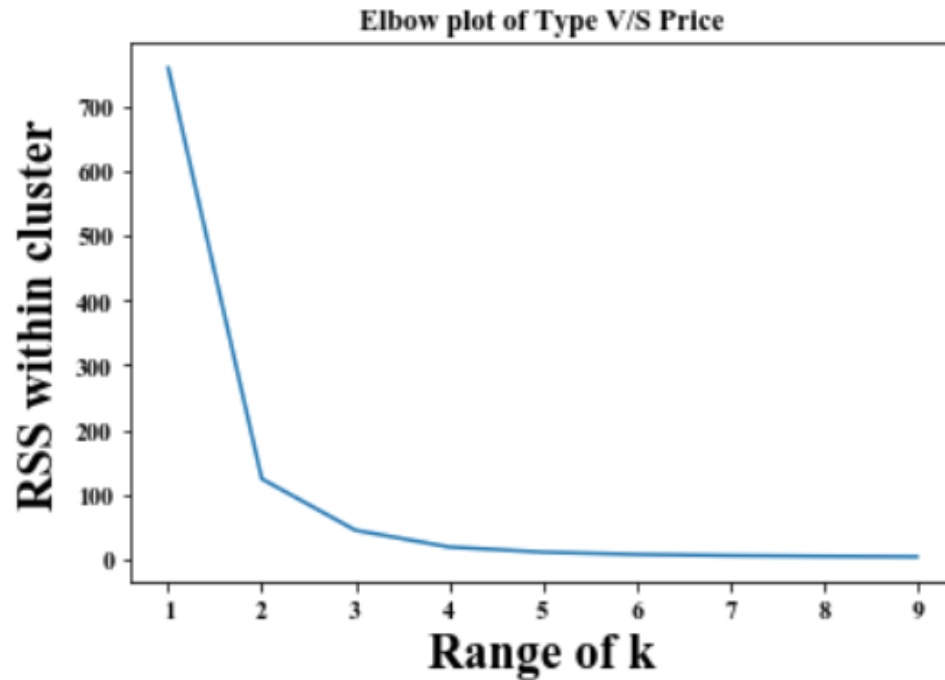
k_range = range(1,10)
# Try clustering the data for k values ranging 1 to 10
k_means_var = [KMeans(n_clusters = k).fit(df_c) for k in k_range]
centroids = [X.cluster_centers_ for X in k_means_var]

k_euclid = [cdist(df_c, cent, 'euclidean') for cent in centroids]
dist = [np.min(ke, axis=1) for ke in k_euclid]

# Calculate within-cluster sum of squares
wcsc = [sum(d**2) for d in dist]

# Visualize the elbow method for determining k
import matplotlib.pyplot as plt
plt.plot(k_range, wcsc)
plt.xlabel('Range of k')
plt.ylabel('RSS within cluster')
plt.title('Elbow plot of Type V/S Price')
# plt.savefig('Images/Elbow_5')
plt.show()

```



```

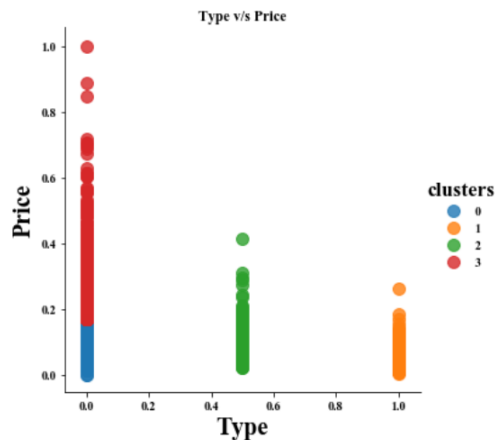
kmeans = KMeans(n_clusters=4, random_state=0).fit(df_c)
labels = kmeans.labels_
#Glue back to originaal data
df_c['clusters'] = labels
df2_C = df_c.rename(columns = {0 : 'Type', 1: 'Price'})

```

```

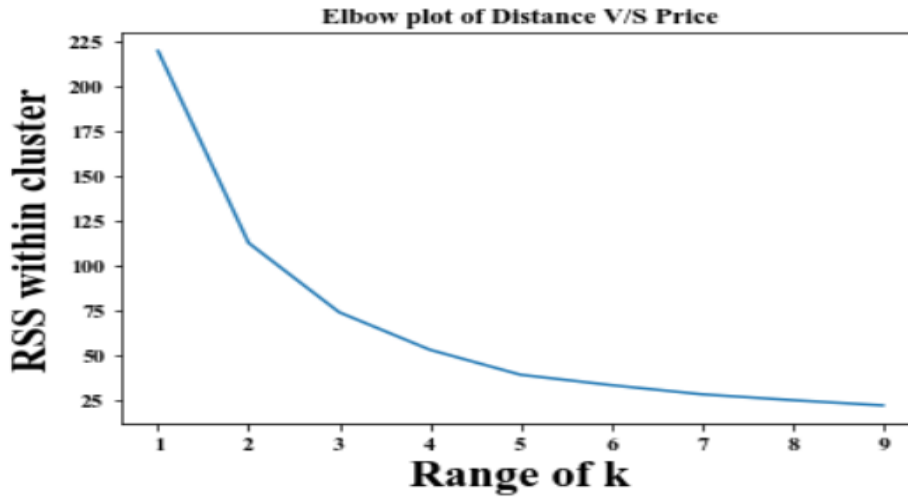
sns.lmplot('Type', 'Price', data = df2_C, fit_reg=False, hue="clusters", scatter_kws={"marker": "D", "s": 100})
plt.title('Type v/s Price')
plt.xlabel('Type')
plt.ylabel('Price')
#plt.savefig('Images/cluster_5.png')
plt.show()

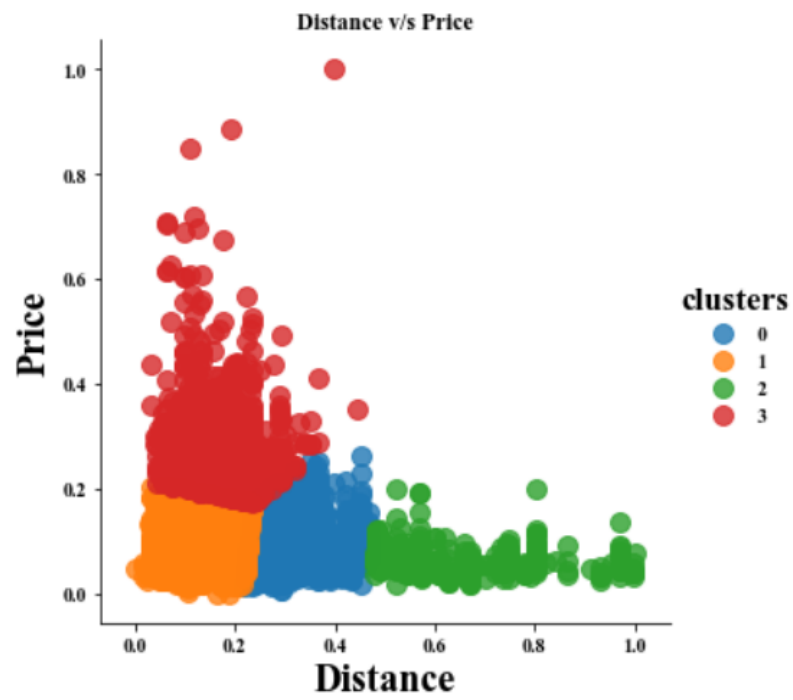
```



In this spherical variant of MAP-DP, as with K -means, the Euclidean metric $\frac{1}{2} \|\cdot\|_2^2$ is used to compute distances to cluster centroids. However, in MAP-DP, the log of N_k^{-i} is subtracted from this distance when updating assignments. Also, the composite variance $\sigma_k^{-i} + \hat{\sigma}^2$ features in the distance calculations such that the smaller $\sigma_k^{-i} + \hat{\sigma}^2$ becomes, the less important the number of data points in the cluster N_k^{-i} becomes the assignment. In that case, the algorithm behaves much like K -means. But, if $\sigma_k^{-i} + \hat{\sigma}^2$ becomes large, then, if a cluster already has many data points assigned to it, it is more likely that the current data point is assigned to that cluster (in other words, clusters exhibit a “*rich-get-richer*” effect). MAP-DP thereby takes into account the density of clusters, unlike K -means. We can see $\sigma_k^{-i} + \hat{\sigma}^2$ as controlling the “balance” between geometry and density

Similarly when we Compared the price with Distance the results were something like this:





6.0 Linear Regression for Multiple Variables

Multiple linear regression (MLR), also known as multiple regression, is a statistical technique that uses several independent variables to predict the outcome of a dependent variable. The goal of multiple linear regression (MLR) is to model the linear relationship between the independent variables and dependent variable.

The multiple regression model is based on the following assumptions:

- There is a linear relationship between the dependent variables and the independent variables.
- The independent variables are not too highly correlated with each other.
- y_i observations are selected independently and randomly from the population.

The Formula for Multiple Linear Regression Is:

$$y = m_1x_1 + m_2x_2 + m_3x_3 + \dots + m_ix_i + b + r$$

Where,

- y = the dependent variable of the regression
- m_i = slope of the regression for that variable
- x_i = independent variable of the regression
- B = constant
- r = residual

6.0.1 R squared

The coefficient of determination (R-squared) is a statistical metric that is used to measure how much of the variation in outcome can be explained by the variation in the independent variables. R^2 always increases as more predictors are added to the MLR model even though the predictors may not be related to the outcome variable. R^2 can only be between 0 and 1, where 0 indicates that the outcome cannot be predicted by any of the independent variables and 1 indicates that the outcome can be predicted without error from the independent variables.

Calculating R-Squared

The actual calculation of R-squared requires several steps. This includes taking the data points (observations) of dependent and independent variables and finding the line of best fit, often from a regression model. From there you calculate predicted values, subtract actual values and square the results. This yields a list of errors squared, which is then summed and equals the unexplained variance.

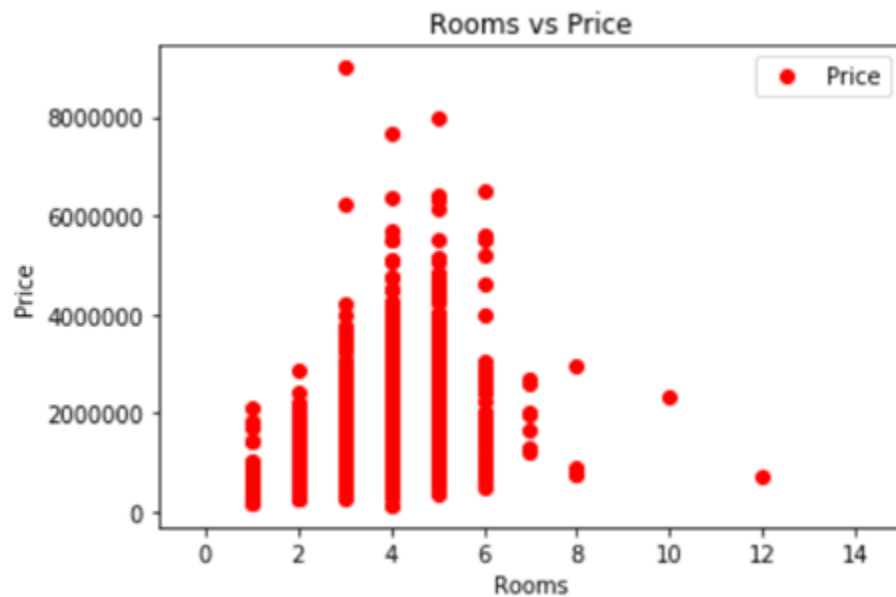
Limitations of R-Squared

R-squared will give you an estimate of the relationship between movements of a dependent variable based on an independent variable's movements. It doesn't tell you whether your chosen model is good or bad, nor will it tell you whether the data and predictions are biased. A high or low R-square isn't necessarily good or bad, as it doesn't convey the reliability of the model, nor whether you've chosen the right regression. You can get a low R-squared for a good model, or a high R-square for a poorly fitted model, and vice versa.

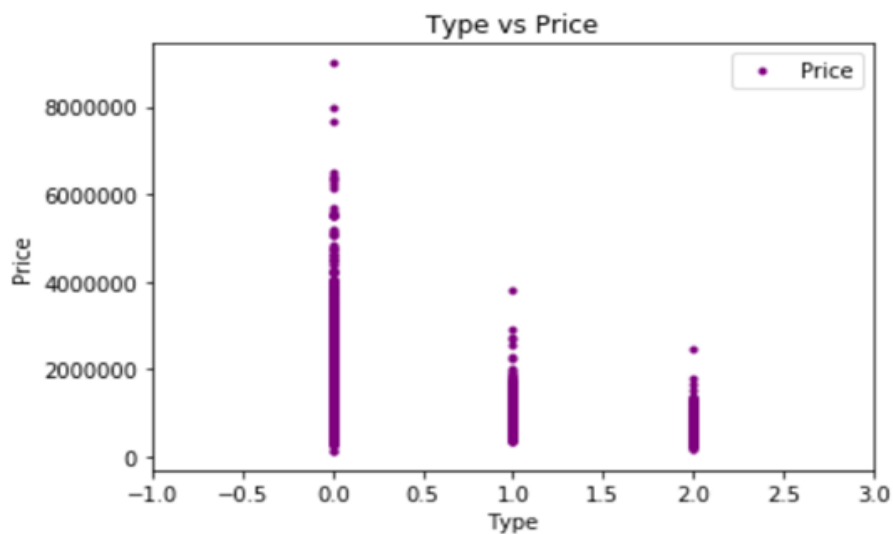
R^2 by itself can't be used to identify which predictors should be included in a model and which should be excluded.

6.0.2 Dependent variable versus Independent variables

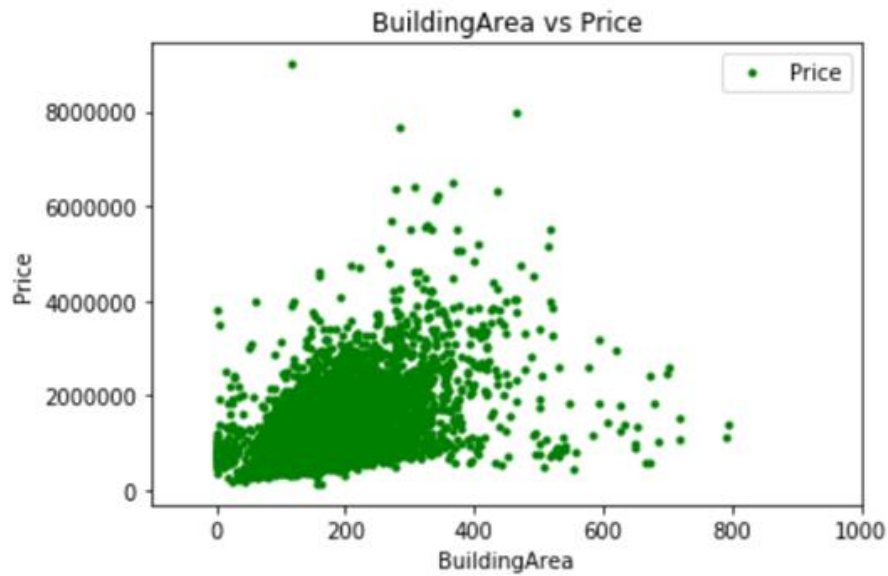
Before beginning the Linear Progression, we will see the graphs between each independent variable versus the dependent variable to understand the behavior of how the dependent variable, here Price, is dependent on each independent variable



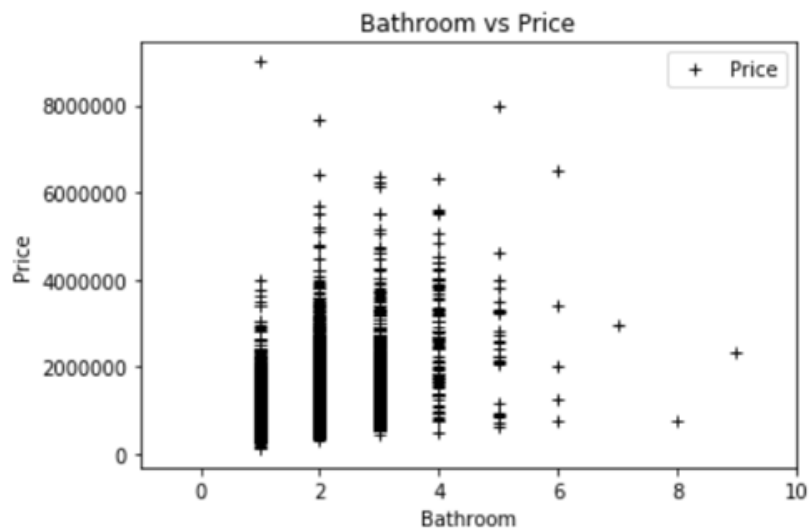
Here, we can see that the number of rooms in a house are majorly between 1 to 6 and that the houses with 5 rooms cost the highest.



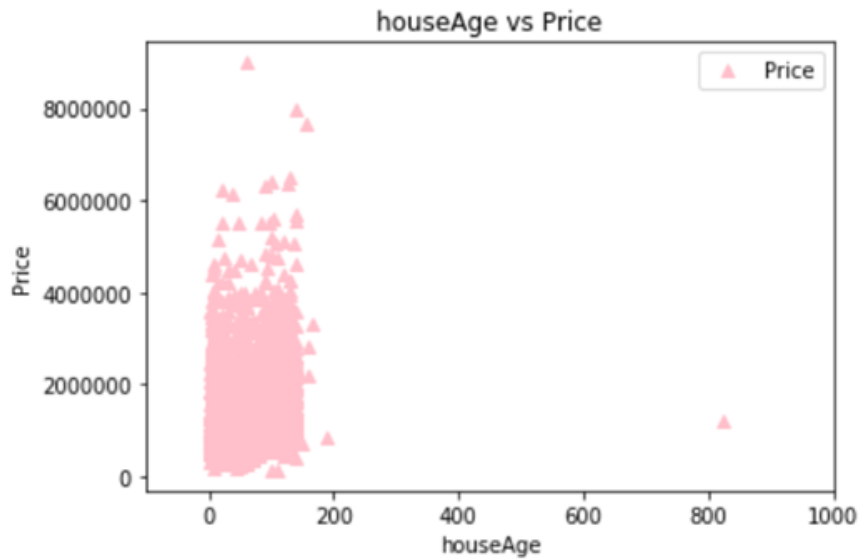
Here, 0 is type house, villa, etc, 1 is type unit and 2 is townhouse. The most expensive house is type villa.



In this graph, the majority of the building area is up to 400m.



In this graph, we can see that the majority of the houses have 2-3 bathrooms and they cost around the same price.



As we can see in the above graph that the most houses were built in the range of 0 - ~180yrs.



This graph describes the distance of the house from the central business district and most of them lie between 0-20 miles

6.0.3 Steps to perform Linear Regression

1. Choose a class of the model
2. Choose model hyperparameters

```
In [8]: # 1. Choose a class of model-
from sklearn.linear_model import LinearRegression

In [9]: # 2. Choose model hyperparameters
model = LinearRegression(fit_intercept=True)
model

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

3. Declare the feature set and the target variable
4. Split the data into training set and testing set

Here we are keeping training data = 80% and testing data = 20%

```
In [11]: #Features set
X = mel[['Rooms', 'Bathroom', 'houseAge', 'BuildingArea', 'YearBuilt', 'Distance', 'Type']]

# target variable(Dependent variable)
y = mel['Price']

# Training, test, split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = .20, random_state= 0)
```

5. Fit the model

```
In [12]: # Fit
reg = LinearRegression()

# Fit model to training data
reg.fit(X_train, y_train)

Out[12]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

6. Evaluate:

Calculate the metrics of the training set

```
In [61]: #metrics of training set
from sklearn import metrics

print('Metrics of training set:')
print('\nIntercept: {:>0.04f}'.format(reg.intercept_))

print('\nRoot Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_train, reg.predict(X_train))))
print('\nR^2: {:>0.04f}'.format(reg.score(X_train, y_train)))

Metrics of training set:

Intercept: 5511136.0556

Root Mean Squared Error: 469667.4732229277

R^2: 0.5298
```

Calculate the metrics of the testing set

```
In [59]: # Predicting test set results
y_pred = reg.predict(X_test)

print('Metrics of testing set:')
print('\nIntercept: {:>0.04f}'.format(reg.intercept_))
print('\nRoot Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
print('\nR^2: {:>0.04f}'.format(reg.score(X_test, y_test)))

Metrics of testing set:

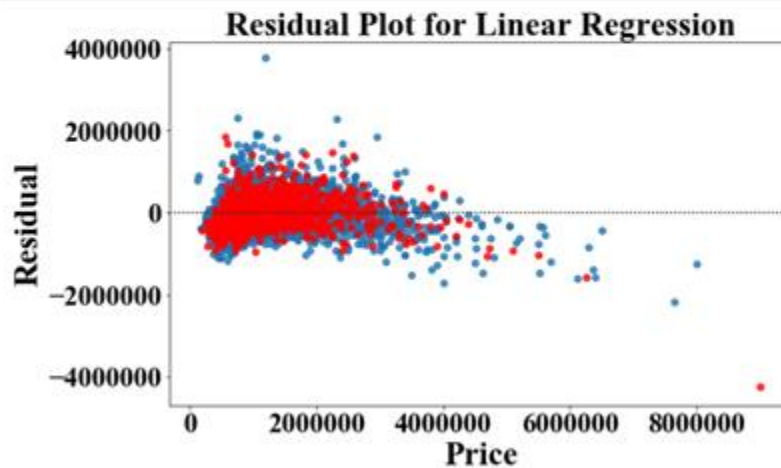
Intercept: 5511136.0556

Root Mean Squared Error: 511331.4195059427

R^2: 0.4745
```

Residuals: Residuals are the difference between the dependent variable (y) and the predicted variable (y_predicted). A residual plot is a scatter plot of the independent variables and the residual.

```
In [63]: plt.figure(figsize = (10, 6))
plt.rcParams["axes.labelsize"] = 30
plt.rcParams["font.family"] = "Times New Roman"
plt.title('Residual Plot for Linear Regression', fontsize=30)
plt.ylabel('Residual')
plt.xlabel('Price')
residuals = sns.residplot(y_train, reg.predict(X_train))
residuals_test = sns.residplot(y_test, reg.predict(X_test), color='r')
residuals_test.tick_params(labelsize=25)
plt.show()
```



Coefficients for each independent variable

```
In [17]: cdf = pd.DataFrame(data = reg.coef_, index = X.columns, columns = ['Coefficients'])
cdf
```

Out[17]:

Coefficients	
Rooms	87675.727142
Bathroom	212995.732722
houseAge	2701.194360
BuildingArea	3179.158927
YearBuilt	-2701.194360
Distance	-28838.753965
Type	-42449.042015

Intercept:

```
In [18]: reg.intercept_ #intercept
```

Out[18]: 5511136.055626783

Actual Values vs Predicted Values for the testing set:

```
In [22]: #Compare the actual value and the predicted value
df = pd.DataFrame({'Actual': y_test_res.reshape(-1), 'Predicted': y_pred_res.reshape(-1)})
df
```

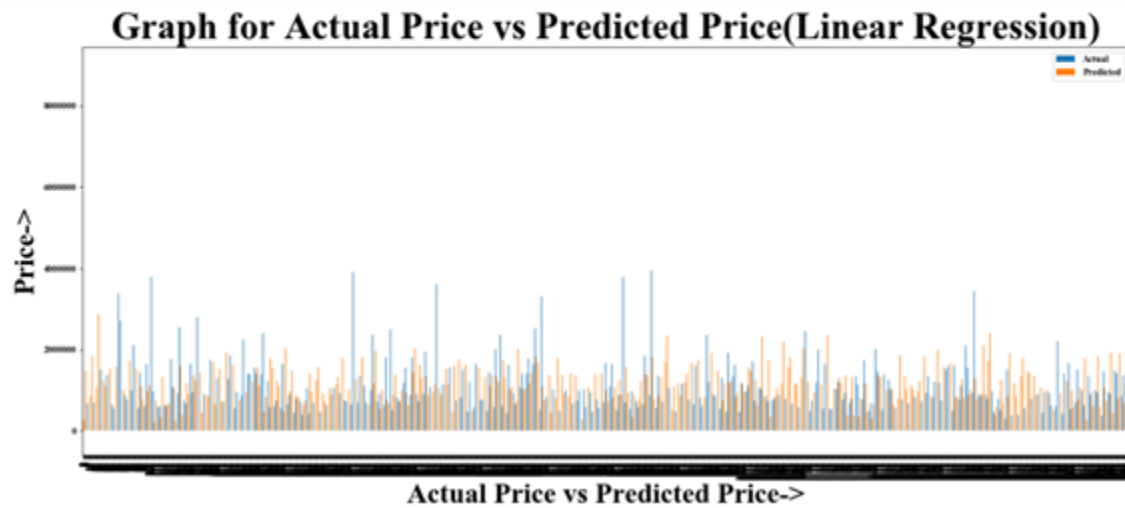
Out[22]:

	Actual	Predicted
0	960500.0	7.756996e+05
1	505000.0	2.730029e+05
2	510000.0	7.507878e+05
3	621000.0	8.798352e+05
4	3401000.0	1.477962e+06
...
1588	950000.0	1.400258e+06
1589	830000.0	5.299495e+05
1590	660000.0	8.632729e+05
1591	1400000.0	1.755731e+06
1592	1616000.0	1.710726e+06

1593 rows × 2 columns

Plot of Actual data(testing set) vs Predicted data:

```
In [48]: df.plot(kind='bar',figsize=(20,8))
plt.plot(which='major', linewidth='0.3')
plt.plot(which='minor', linewidth='0.3')
plt.ylabel('Price->')
plt.xlabel('Actual Price vs Predicted Price->')
plt.title('Graph for Actual Price vs Predicted Price(Linear Regression)', fontsize=40)
plt.show()
```



7.0 Ridge Regression

The Ridge regression is a technique which is specialized to analyze multiple regression data which is multicollinearity in nature.

The term multicollinearity refers to the collinearity concept in statistics. In this phenomenon, one predicted value in multiple regression models is linearly predicted with others to attain a certain level of accuracy.

The concept of multicollinearity occurs when there are high correlations between more than two predicted variables.

Ridge regression is used to create a parsimonious model in the following scenarios:

- The number of predictor variables in a given set exceeds the number of observations.
- The dataset has multicollinearity (correlations between predictor variables).

The regularization techniques are as follows:

- Penalize the magnitude of coefficients of features
- Minimize the error between the actual and predicted observations

Formula

$$\text{Cost}(W) = \text{RSS}(W) + \lambda * (\text{sum of squares of weights})$$

$$= \sum_{i=1}^N \left\{ y_i - \sum_{j=0}^M w_j x_{ij} \right\}^2 + \lambda \sum_{j=0}^M w_j^2$$

Lambda is the hyperparameter.

RSS = sum of squared residuals

7.1 Steps to Perform Ridge Regression

1. Import Ridge from sklearn.linear_model and split the dataset into training and testing set (80-20)

```
In [24]: from sklearn.linear_model import Ridge #import Ridge

In [94]: a = mel.drop('Price',1)
b = mel['Price']

from sklearn.model_selection import train_test_split
a_train, a_test, b_train, b_test = train_test_split(X,mel.Price, test_size = 0.20, random_state=42)
```

2. Select an arbitrary value of alpha (preferable close to 1-5%) and fir the model for training

```
In [113]: ridgeReg = Ridge(alpha=0.05, normalize=True)

#Training the model
ridgeReg.fit(a_train,b_train)

Out[113]: Ridge(alpha=0.05, copy_X=True, fit_intercept=True, max_iter=None,
              normalize=True, random_state=None, solver='auto', tol=0.001)
```

3. Perform prediction on the testing dataset and evaluate

```
In [114]: b_pred = ridgeReg.predict(a_test)

In [115]: #metrics of training set
from sklearn import metrics
print('Metrics of training set')
print('\nIntercept: {:>0.04f}'.format(ridgeReg.intercept_))
print('\nRoot Mean Squared Error:',np.sqrt(metrics.mean_squared_error(b_train,ridgeReg.predict(a_train))))
print('\nR^2: {:>0.04f}'.format(ridgeReg.score(a_train, b_train)))

Metrics of training set

Intercept: 5107318.9713

Root Mean Squared Error: 483990.5560503056

R^2: 0.5110
```

4. Evaluation

```
In [116]: # Predicting test set results
y_pred = reg.predict(a_test)

print('Metrics of testing set:')
print('\nIntercept: {:>0.04f}'.format(reg.intercept_))

print('\nRoot Mean Squared Error:',np.sqrt(metrics.mean_squared_error(b_test,b_pred)))
print('\nR^2: {:>0.04f}'.format(reg.score(a_test, b_test)))

Metrics of testing set:

Intercept: 5511136.0556

Root Mean Squared Error: 457244.03428469005

R^2: 0.5413
```

Coefficients:

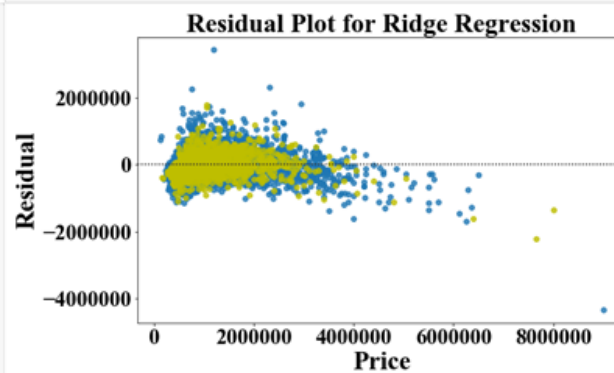
```
In [118]: cdf = pd.DataFrame(data = ridgeReg.coef_, index = X.columns, columns = ['Coefficients'])
cdf
```

Out[118]:

Coefficients	
Rooms	94496.054019
Bathroom	214745.336661
BuildingArea	2849.335085
houseAge	2470.527811
YearBuilt	-2470.527811
Distance	-28964.079607
Type	-61049.480413

Residual Plot:

```
In [117]: plt.figure(figsize = (10, 6))
plt.rcParams["axes.labelsize"] = 30
plt.rcParams["font.family"] = "Times New Roman"
plt.title('Residual Plot for Ridge Regression', fontsize=30)
plt.ylabel('Residual')
plt.xlabel('Price')
residuals = sns.residplot(b_train, ridgeReg.predict(a_train))
residuals_test = sns.residplot(b_test, ridgeReg.predict(a_test), color='y')
residuals_test.tick_params(labelsize=25)
plt.show()
```



Actual vs Predicted data for testing set:

```
In [121]: df1 = pd.DataFrame({'Actual': b_test_res.reshape(-1), 'Predicted': b_pred_res.reshape(-1)})
df1
```

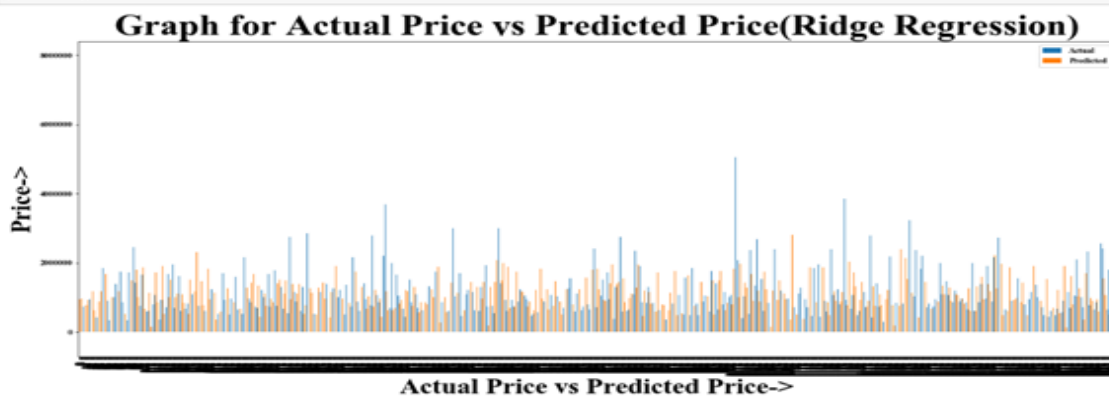
Out[121]:

	Actual	Predicted
0	1210000.0	1.054845e+06
1	680000.0	9.455428e+05
2	1225000.0	1.201379e+06
3	841000.0	1.232623e+06
4	801000.0	9.642360e+05
...
1588	1053000.0	5.697115e+05
1589	1035000.0	1.266204e+06
1590	994000.0	1.418655e+06
1591	1185000.0	1.090334e+06
1592	1000000.0	1.341191e+06

1593 rows x 2 columns

Graph of actual values vs predicted values of testing data

```
In [122]: df1.plot(kind='bar',figsize=(20,8))
plt.plot(which='major', linewidth='0.3')
plt.plot(which='minor', linewidth='0.3')
plt.ylabel('Price->')
plt.xlabel('Actual Price vs Predicted Price->')
plt.title('Graph for Actual Price vs Predicted Price(Ridge Regression)', fontsize=40)
plt.show()
```



7.2 Lasso Regression

What is Lasso Regression?

Lasso regression is a type of **linear regression** that uses shrinkage. Shrinkage is where data values are shrunk towards a central point, like the mean. The lasso procedure encourages simple, sparse models (i.e. models with fewer parameters). This particular type of regression is well-suited for models showing high levels of multicollinearity or when you want to automate certain parts of model selection, like variable selection/parameter elimination.

The acronym “LASSO” stands for **L**east **A**bsolute **S**hrinkage and **S**election **O**perator.

L1 Regularization:

Lasso regression performs L1 regularization, which adds a penalty equal to the absolute value of the magnitude of coefficients. This type of regularization can result in sparse models with few coefficients; Some coefficients can become zero and eliminated from the model. Larger penalties result in coefficient values closer to zero, which is the ideal for producing simpler models. On the other hand, L2 regularization (e.g. Ridge regression) *doesn't* result in elimination of coefficients or sparse models. This makes the Lasso far easier to interpret than the Ridge.

Lasso solutions are quadratic programming problems, which are best solved with software (like Matlab). The goal of the algorithm is to minimize:

$$\sum_{i=1}^n (y_i - \sum_j x_{ij} \beta_j)^2 + \lambda \sum_{j=1}^p |\beta_j|$$

Which is the same as minimizing the sum of squares with constraint $\sum |\beta_j| \leq s$. Some of the β s are shrunk to exactly zero, resulting in a regression model that's easier to interpret.

A **tuning parameter**, λ controls the strength of the L1 penalty. λ is basically the amount of shrinkage:

- When $\lambda = 0$, no parameters are eliminated. The estimate is equal to the one found with linear regression.
- As λ increases, more and more coefficients are set to zero and eliminated (theoretically, when $\lambda = \infty$, *all* coefficients are eliminated).
- As λ increases, bias increases.
- As λ decreases, variance increases.

If an intercept is included in the model, it is usually left unchanged.

7.2.1 Steps To perform Lasso Regression :

Here in this model , we have split the testing and training data subsets and Removed our prime aspect attribute to predict it: Prices

- The testing model : 33%
- The training model : 77%

```
attributes= dataset.drop(['Price'],axis=1)

X_train, X_test, Y_train, Y_test = train_test_split(attributes, dataset['Price'],
test_size=0.33, random_state=5)
```

Select an arbitrary value of alpha (preferable close to 1-5%) and fir the model for training

```
import sklearn
from sklearn import linear_model
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV, train_test_split
lasso_para = {'alpha':[1, .1, .02, .01, .001]}

print(GridSearchCV(linear_model.Lasso(), param_grid=lasso_para).fit(X_train, Y_train).best_estimator_,)

Lasso(alpha=0.001, copy_X=True, fit_intercept=True, max_iter=1000,
      normalize=False, positive=False, precompute=False, random_state=None,
      selection='cyclic', tol=0.0001, warm_start=False)
```

Performing on the testing Dataset

```
predictions_test = lasso.predict(X_test)

print('Coefficients:\n {}'.format(str(lasso.coef_)))
print('\nIntercept: {:.>0.04f}'.format(lasso.intercept_))
print('\n' + 'RMSE: {:.>0.04f}'.format(np.sqrt(mean_squared_error(Y_test, lasso.predict(X_test)))))
print('\nR^2: {:.>0.04f}'.format(lasso.score(X_test, Y_test)))

Coefficients:
[ 8.44303785e+04  1.72242257e+05  3.39726361e+03  4.99436108e+03
 -9.54265784e-12 -2.95480524e+04 -5.35075648e+04]

Intercept: 138067.8057

RMSE: 468087.6284

R^2: 0.5475
```


Performing on the Training Dataset

```
import sklearn
from sklearn import linear_model
lasso = linear_model.Lasso(alpha = .001, max_iter = 1000, random_state=5)

lasso.fit(X_train, Y_train)

predictions = pd.DataFrame(data=lasso.predict(X_train).flatten())

predtransformed = predictions.apply(np.exp)

# train set
from sklearn.metrics import mean_squared_error
# train set
print('Coefficients:\n {}'.format(str(lasso.coef_)))
print('\nIntercept: {:.>0.04f}'.format(lasso.intercept_))
print('\n' + 'RMSE: {:.>0.04f}'.format(np.sqrt(mean_squared_error(Y_train, lasso.predict(X_train)))))
print('\nR^2: {:.>0.04f}'.format(lasso.score(X_train, Y_train)))

Coefficients:
[ 8.44303785e+04  1.72242257e+05  3.39726361e+03  4.99436108e+03
 -9.54265784e-12 -2.95480524e+04 -5.35075648e+04]

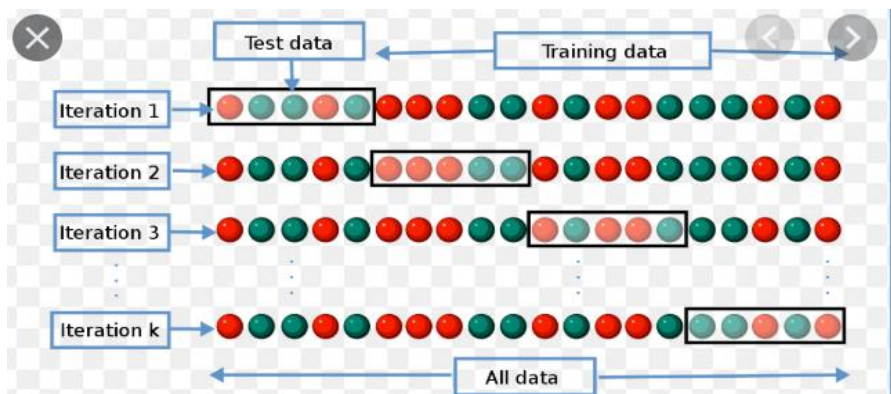
Intercept: 138067.8057

RMSE: 484147.7306

R^2: 0.5015
```

We have used the K-fold cross validation so that it ensures that every observation from the original dataset has the chance of appearing in training and test set.

K-fold cross validation considers training on all but the kth part, and then validating on the kth part, iterating over $k = 1, \dots, K$ (When $K = n$, we call this leave-one-out cross-validation, because we leave out one data point at a time)



```

# K-fold cross validation
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
CVSlasso = cross_val_score(lasso, attributes, dataset['Price'], cv=10)
print('Cross validated R^2: {}'.format(CVSlasso))
print('Mean cross validated R^2: {:.4f}'.format(CVSlasso.mean()))

```

Cross validated R²: [0.47708514 0.58857423 0.55653538 0.50075474 0.54457127 0.43667971
0.3562498 0.50675244 0.51358776 0.52554188]

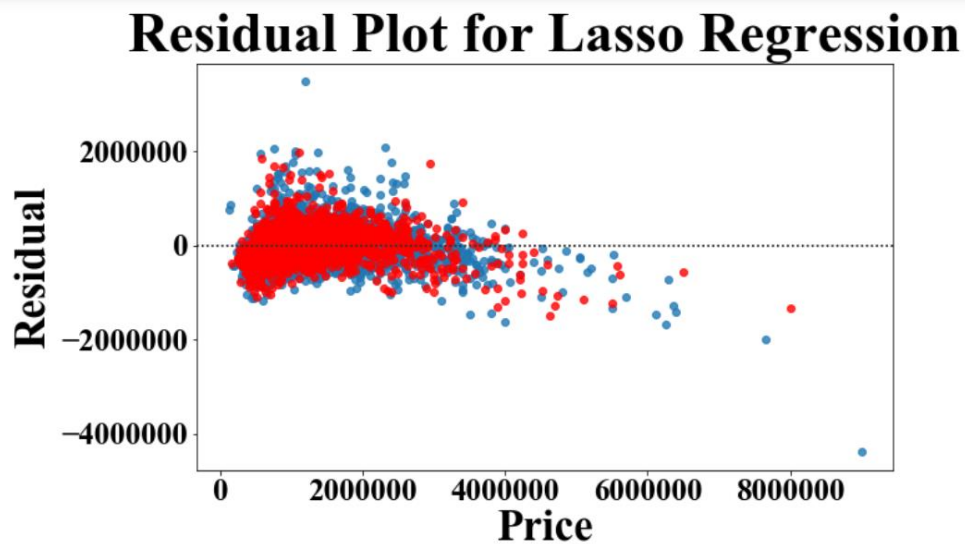
Mean cross validated R²: 0.5006

Residual Plot:

```

plt.figure(figsize = (10, 6))
plt.rcParams["axes.labelsize"] = 35
plt.rcParams["font.family"] = "Times New Roman"
plt.title('Residual Plot for Lasso Regression', fontsize=45)
plt.ylabel('Residual')
plt.xlabel('Price')
residuals = sns.residplot(Y_train, lasso.predict(X_train))
residuals_test = sns.residplot(Y_test, lasso.predict(X_test), color='r')
residuals_test.tick_params(labelsize=25)
plt.show()

```



8.0 Random Forest Regressor

Analysis using Random Forest Regressor

The random forest is a model made up of many decision trees. Rather than just simply averaging the prediction of trees (which we could call a “forest”), this model uses two key concepts that gives it the name random:

- Random sampling of training data points when building trees
- Random subsets of features considered when splitting nodes
- Random sampling of training observations

When training, each tree in a random forest learns from a random sample of the data points. The samples are drawn with replacement, known as bootstrapping, which means that some samples will be used multiple times in a single tree. The idea is that by training each tree on different samples, although each tree might have high variance with respect to a particular set of the training data, overall, the entire forest will have lower variance but not at the cost of increasing the bias.

At test time, predictions are made by averaging the predictions of each decision tree. This procedure of training each individual learner on different bootstrapped subsets of the data and then averaging the predictions is known as bagging, short for bootstrap aggregating.

A Random Forest is an ensemble technique capable of performing both regression and classification tasks with the use of multiple decision trees and a technique called Bootstrap Aggregation, commonly known as bagging. The basic idea behind this is to combine multiple decision trees in determining the final output rather than relying on individual decision trees.

Approach:

1. Pick at random K data points from the training set.
2. Build the decision tree associated with those K data points.
3. Choose the number Ntree of trees you want to build and repeat step 1 & 2.
4. For a new data point, make each one of your Ntree trees predict the value of Y for the data point, and assign the new data point the average across all of the predicted Y values.

The estimator to use for this project is the RandomForestRegressor

Random Forest Regressor-

A random forest is a meta estimator that fits a number of classifying decision trees on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. Some of the constant parameters through are 0

'Bootstrap'	TRUE
Loss	MSE
'Max_depth'	None
'Warm_start'	FALSE
'Min_sample_left'	1
Package	RandomForestRegressor SkLearn

```
jupyter Random Forest Regression Code Last Checkpoint: an hour ago (unsaved changes) Logout
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3 C
In [1]: 1 import numpy as np
        2 import pandas as pd
        3 from sklearn.model_selection import train_test_split
In [2]: 1 import warnings
        2 warnings.filterwarnings("ignore")
In [3]: 1 df=pd.read_csv("data_regression.csv")
In [4]: 1 X = df.drop(['Price'], axis = 1)
        2 y = df.Price
        3 X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2, random_state=101)
In [5]: 1 from sklearn import metrics
        2 from sklearn.metrics import mean_squared_error
        3 from sklearn.ensemble import RandomForestRegressor
        4 rf = RandomForestRegressor(random_state = 42).fit(X_train, y_train)
In [6]: 1 rf.get_params
Out[6]: <bound method BaseEstimator.get_params of RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
max_features='auto', max_leaf_nodes=None,
min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=None,
oob_score=False, random_state=42, verbose=0, warm_start=False)>
In [ ]: 1
```

Parameter Tuning the Model to improve Prediction-

Using RandomizedSearchCV for parameter tuning:

Step 1: Number of trees in random forest

Step 2: Maximum number of levels in tree

Step 3: Create the random grid

```
jupyter Random Forest Regression Code Last Checkpoint: an hour ago (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
In [7]: 1 pred = rf.predict(X_test)
        2 print('Root Mean Square Error:', np.sqrt(metrics.mean_squared_error(y_test, pred)))

Root Mean Square Error: 320603.9505345671

In [8]: 1 from tabulate import tabulate
        2 headers = ["name", "score"]
        3 values = sorted(zip(X_train.columns, rf.feature_importances_), key=lambda x: x[1] * -1)
        4 print(tabulate(values, headers, tablefmt="pretty"))

+-----+-----+
| name | score |
+-----+-----+
| Regionname_Southern Metropolitan | 0.22260231322021623 |
| Rooms | 0.17146510559407865 |
| Distance | 0.15062931463541437 |
| Landsize | 0.08633786196087781 |
| Longitude | 0.07641770621268287 |
| Latitude | 0.05805556522667351 |
| Type_u | 0.050980764938591105 |
| month | 0.02809422705656143 |
| Regionname_Eastern Metropolitan | 0.026808105821875594 |
| Type_h | 0.02590558246253618 |
| Bathroom | 0.025564252167309225 |
| Propertycount | 0.022007025904982484 |
| Car | 0.01191913072951989 |
| CouncilArea_Stonnington City Council | 0.007618933431596493 |
| Method_S | 0.006439841948103644 |
| Method_VB | 0.004428392709433375 |
| Method_PI | 0.0036635613150618196 |
| Type_t | 0.003432370704588083 |
| CouncilArea_Kingston City Council | 0.002953283467047477 |
| Method_SP | 0.002499301789349128 |
| CouncilArea_Boroondara City Council | 0.002143533800613426 |
| CouncilArea_Monash City Council | 0.0020245570560313174 |
| CouncilArea_Moonsee Valley City Council | 0.0011212043906372307 |
| CouncilArea_Port Phillip City Council | 0.0010782528319779982 |
+-----+-----+
```

jupyter Random Forest Regression Code Last Checkpoint: an hour ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted | Python 3 C

Run Code

CouncilArea_Moonee Valley City Council	0.0011212043906372307
CouncilArea_Port Phillip City Council	0.0010782528319779982
CouncilArea_Glen Eira City Council	0.0008278224564006474
Method_SA	0.0005834000513951935
CouncilArea_Melbourne City Council	0.0005325295197429353
CouncilArea_Maribyrnong City Council	0.0005062418863395945
CouncilArea_Hobsons Bay City Council	0.00044794901817962204
CouncilArea_Banyule City Council	0.0003895911553645951
CouncilArea_Manningham City Council	0.00034556527506041015
CouncilArea_Darebin City Council	0.00030299736657704395
CouncilArea_Yarra City Council	0.00028445119747070436
Regionname_Western Metropolitan	0.0002498379507172544
CouncilArea_Bayside City Council	0.00024977418092199844
CouncilArea_Moreland City Council	0.00024866021047532786
Regionname_Northern Metropolitan	0.0001560601033850044
CouncilArea_Whitehorse City Council	0.00014287268870530787
CouncilArea_Knox City Council	0.00011621509118527976
CouncilArea_Melton City Council	9.924820089624641e-05
CouncilArea_Brimbank City Council	7.075209169130046e-05
CouncilArea_Hume City Council	6.483485983616738e-05
CouncilArea_Greater Dandenong City Council	3.553637813032711e-05
CouncilArea_Casey City Council	3.0231347796140865e-05
CouncilArea_Whittlesea City Council	2.23892677595212e-05
CouncilArea_Millumbik Shire Council	1.88915622991845e-05
CouncilArea_Macedon Ranges Shire Council	1.8161207092482355e-05
Regionname_Eastern Victoria	1.2238067930077335e-05
Regionname_South-Eastern Metropolitan	1.1738069656373819e-05
CouncilArea_Yarra Ranges Shire Council	9.645629838249e-06
CouncilArea_Maroondah City Council	8.231778393515074e-06
Regionname_Northern Victoria	6.989585261587811e-06
CouncilArea_Frankston City Council	4.948121889588019e-06
Regionname_Western Victoria	4.73331740482186e-06
CouncilArea_Cardinia Shire Council	2.8926323419619477e-06
CouncilArea_Wyndham City Council	2.3841860186524105e-06
CouncilArea_Mitchell Shire Council	1.2601351373002549e-06
CouncilArea_Moorabool Shire Council	7.360329453444928e-07

jupyter Random Forest Regression Code Last Checkpoint: an hour ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted | Python 3 C

Run Code

```

In [9]: 1 #Parameter Tuning
        2
        3 from sklearn.ensemble import RandomForestRegressor
        4 rf = RandomForestRegressor(random_state = 42)
        5 rf.get_params()

Out[9]: {'bootstrap': True,
         'criterion': 'mse',
         'max_depth': None,
         'max_features': 'auto',
         'max_leaf_nodes': None,
         'min_impurity_decrease': 0.0,
         'min_impurity_split': None,
         'min_samples_leaf': 1,
         'min_samples_split': 2,
         'min_weight_fraction_leaf': 0.0,
         'n_estimators': 'warn',
         'n_jobs': None,
         'oob_score': False,
         'random_state': 42,
         'verbose': 0,
         'warm_start': False}

In [10]: 1 from sklearn.model_selection import RandomizedSearchCV
         2 n_estimators = [int(x) for x in np.linspace(start = 10, stop = 400, num = 5)]
         3 max_depth = [int(x) for x in np.linspace(10, 500, num = 5)]
         4 random_grid = {'n_estimators': n_estimators,
         5                  'max_depth': max_depth}
         6 random_grid

Out[10]: {'n_estimators': [10, 107, 205, 302, 400],
         'max_depth': [10, 132, 255, 377, 500]}

```

'**max_depth**': 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.

The range of this is [10,132,255,377,500].

'**n_estimators**': The number of trees in the forest. The range is [10, 107, 205, 302, 400]

```
jupyter Random Forest Regression Code Last Checkpoint: an hour ago (autosaved)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
In [ ]: 1
In [11]: 1 error_depth = []
2 from sklearn.ensemble import RandomForestRegressor
3 for d in max_depth:
4     #for d in n_estimators:
5     print(d)
6     rf_best = RandomForestRegressor(random_state = 42,max_depth=d ) #For max_depth
7     #rf_best = RandomForestRegressor(random_state = 42,n_estimators = d ) #For n_estimators
8     rf_best.fit(X_train, y_train)
9     predictions_best=rf_best.predict(X_test)
10    error_depth.append( metrics.mean_squared_error(y_test, predictions_best))

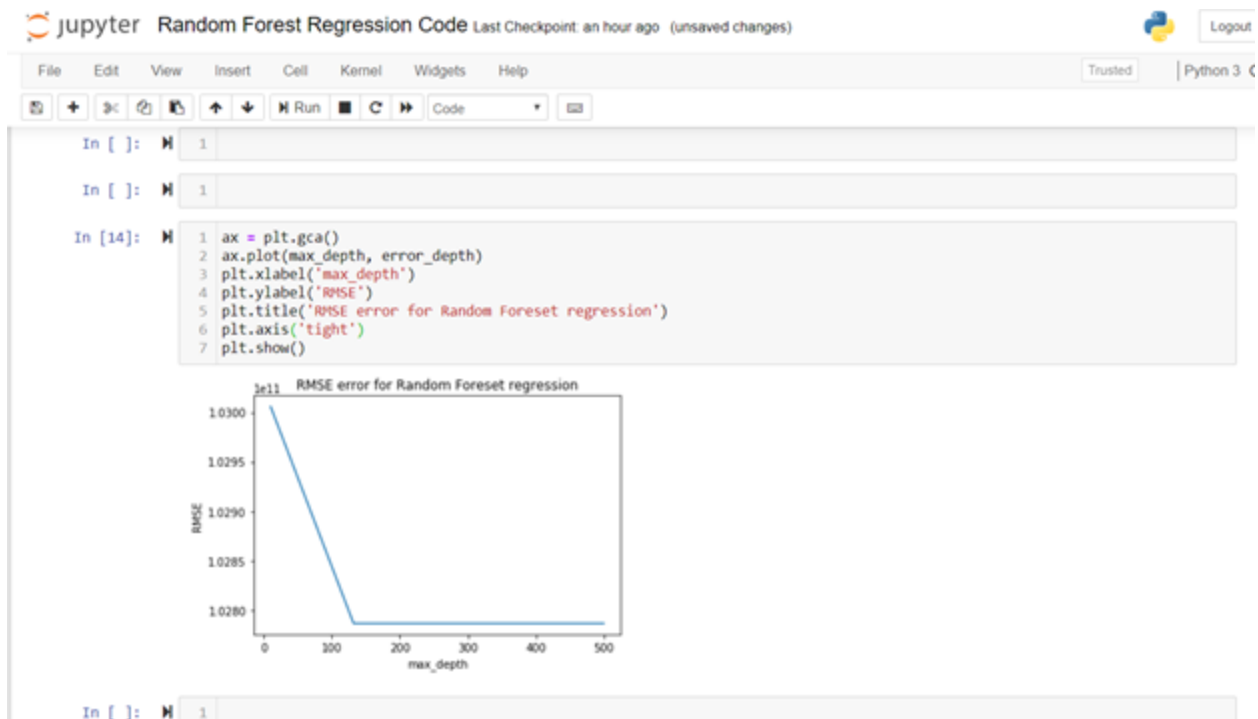
10
132
255
377
500

In [ ]: 1
```

```
jupyter Random Forest Regression Code Last Checkpoint: an hour ago (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help Trusted Python 3
In [12]: 1 import matplotlib.pyplot as plt
2 from scipy import stats

In [13]: 1 ax = plt.gca()
2 ax.plot(n_estimators, error_depth)
3 plt.xlabel('n_estimators')
4 plt.ylabel('RMSE')
5 plt.title('RMSE error for Random Foreset regression')
6 plt.axis('tight')
7 plt.show()

RMSE error for Random Foreset regression
1e11
10300
10295
10290
10285
10280
0 50 100 150 200 250 300 350 400
n_estimators
```



RandomizedSearchCV:

I used RandomizedSearchCV to perform parameter tuning with $cv = 3$. The parameters of the estimator used to apply these methods are optimized by cross-validated search over parameter settings.

Using the random grid to search for best hyperparameters:

Step 1: First created the base model to tune

Step 2: Random search of parameters, using 3 fold cross validation, search across 100 different combinations, and use all available cores

Step 3: Fit the random search model

jupyter Random Forest Regression Code Last Checkpoint: an hour ago (unsaved changes) Python 3

```

In [ ]: 1

In [15]: 1 # Use the random grid to search for best hyperparameters
2 rf = RandomForestRegressor()
3 rf_random = RandomizedSearchCV(estimator = rf, param_distributions = random_grid, n_iter = 5, cv = 3, verbose=2, n_jobs
4 rf_random.fit(X_train,y_train)

Fitting 3 folds for each of 5 candidates, totalling 15 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done 15 out of 15 | elapsed: 1.6min finished

Out[15]: RandomizedSearchCV(cv=3, error_score='raise-deprecating',
        estimator=RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
        max_features='auto', max_leaf_nodes=None,
        min_impurity_decrease=0.0, min_impurity_split=None,
        min_samples_leaf=1, min_samples_split=2,
        min_weight_fraction_leaf=0.0, n_estimators='warn', n_jobs=None,
        oob_score=False, random_state=None, verbose=0, warm_start=False),
        fit_params=None, iid='warn', n_iter=5, n_jobs=-1,
        param_distributions={'n_estimators': [10, 107, 205, 302, 400], 'max_depth': [10, 132, 255, 377, 500]},
        pre_dispatch='2*n_jobs', random_state=None, refit=True,
        return_train_score='warn', scoring=None, verbose=2)

In [16]: 1 rf_random.best_params_

Out[16]: {'n_estimators': 107, 'max_depth': 255}

In [ ]: 1

```

jupyter Random Forest Regression Code Last Checkpoint: an hour ago (unsaved changes) Python 3

```

In [ ]: 1

In [ ]: 1

In [ ]: 1

In [17]: 1 from sklearn.ensemble import RandomForestRegressor
2 rf_best = RandomForestRegressor(max_depth = 500, n_estimators = 205)
3 rf_best.fit(X_train, y_train)
4 predictions_best=rf_best.predict(X_test)
5 np.sqrt(metrics.mean_squared_error(y_test, predictions_best))

Out[17]: 385924.8830873183

In [18]: 1 train_sizes = [1, 100, 500, 1000, 2500, 5000]
2 from sklearn.model_selection import learning_curve
3
4 train_sizes, train_scores, validation_scores = learning_curve(
5     estimator = RandomForestRegressor(max_depth = 500, n_estimators = 205),
6     X = X_train, y = y_train, train_sizes = train_sizes, cv = 5,
7     scoring = 'neg_mean_squared_error')

In [ ]: 1

```

Using the above I optimized my model to get the best parameters, I got the following results.
Before Tuning:

MSE Error	320603.9505345671
Max_depth	None
N_estimator	10

After Tuning:

MSE Error	305924.8830873183
Max_depth	255
N_estimator	107

OBSERVATIONS-

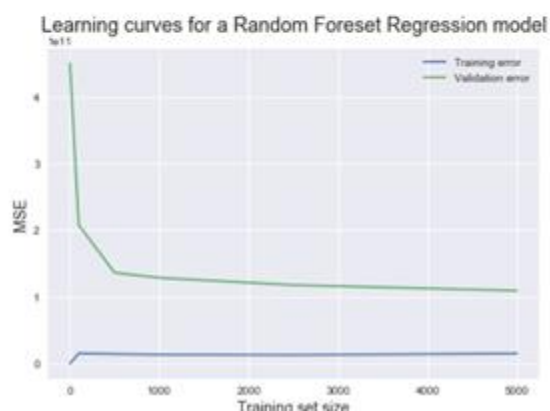
Learning Curve Observations-

- The validation curve could converge toward the training curve if more training instances were added
- This suggests that adding more training instances would help
- As the training error upon converging is low, the model has a low bias / high variance problem.
- As the gap between training error and validation error is large, it suggests high variance.

Also, due to low training error, we can conclude that the model over fits the training data.

```
In [21]: 1 train_scores_mean = -train_scores.mean(axis = 1)
2 validation_scores_mean = -validation_scores.mean(axis = 1)
3 import matplotlib.pyplot as plt
4 %matplotlib inline
5
6 plt.style.use('seaborn')
7
8 plt.plot(train_sizes, train_scores_mean, label = 'Training error')
9 plt.plot(train_sizes, validation_scores_mean, label = 'Validation error')
10
11 plt.ylabel('MSE', fontsize = 14)
12 plt.xlabel('Training set size', fontsize = 14)
13 plt.title('Learning curves for a Random Forest Regression model', fontsize = 18, y = 1.03)
14 plt.legend()
```

Out[21]: <matplotlib.legend.Legend at 0x2488202de10>



Feature Selection

- Random Forest is a tree based technique. A single random forest will unlike Penalized regression with lasso regularization completely ignore features.
- Decision tree splits by features are chosen by local criteria in any of the thousands or millions of nodes and cannot later be undone.
- As Random Forest does a more exhaustive decision making, it's best to input the entire dataset (without any prior feature selection) and henceforth upon using Random Forest achieve substantial increase in prediction performance (estimated by a repeated outer cross-validation) using its variable selection.

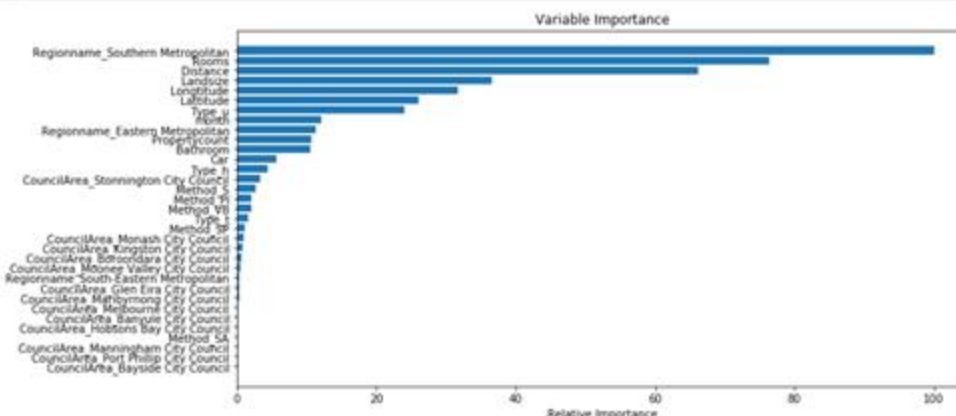
Methodology

- For feature selection, Random Forest provides: mean decrease impurity and mean decrease accuracy.
- Random forest consists of several decision trees. Every node in the decision trees is a condition on a single feature, designed to split the dataset into two so that similar response values end up in the same set.
- The measure based on which the (node) optimal condition is chosen is called impurity (variance).
- Using the training set, it computes how much each feature decreases the weighted impurity in a tree. Combining these decision trees (Forest) the impurity decreases from each feature which is averaged and ranked to give the most important features.
- As a result, I use top n features from Random Forest and input those in to the boosting model. The n feature becomes a hyperparameter and can be tuned. We haven't tuned it in this section, but this can be done to improve future results.

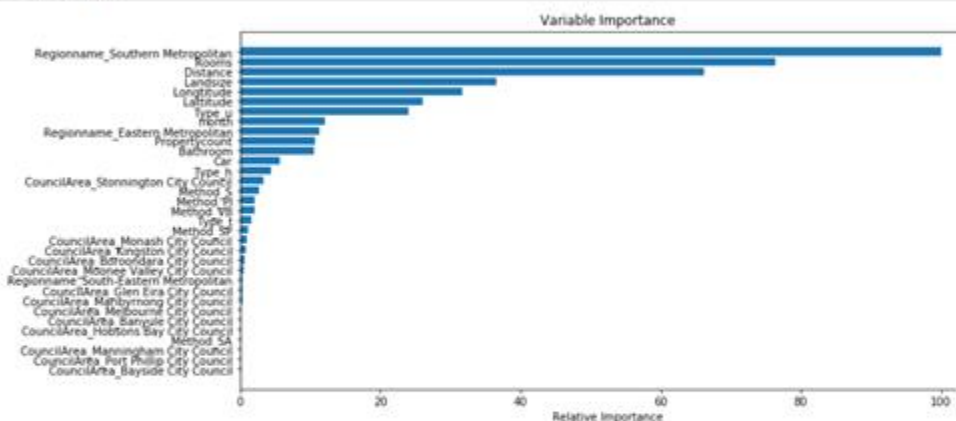
Observations-

- The following are the most important features.
- These will be useful in comparing models in Linear Models at the end.

```
In [19]: 1 feature_importance = rf.best.feature_importances_
2 # make importances relative to max importance
3 feature_importance = 100.0 * (feature_importance / feature_importance.max())
4 sorted_idx = np.argsort(feature_importance)
5 pos = np.arange(sorted_idx.shape[0]) * .5
6 plt.figure(figsize=(12, 6))
7 plt.subplot(1, 1, 1)
8 plt.barh(pos[25:], feature_importance[sorted_idx][25:], align='center')
9 plt.yticks(pos[25:], x.columns[sorted_idx][25:])
10 plt.xlabel('Relative Importance')
11 plt.title('Variable Importance')
12 plt.show()
```



```
12 plt.show()
```



```
In [20]: 1 pos
```

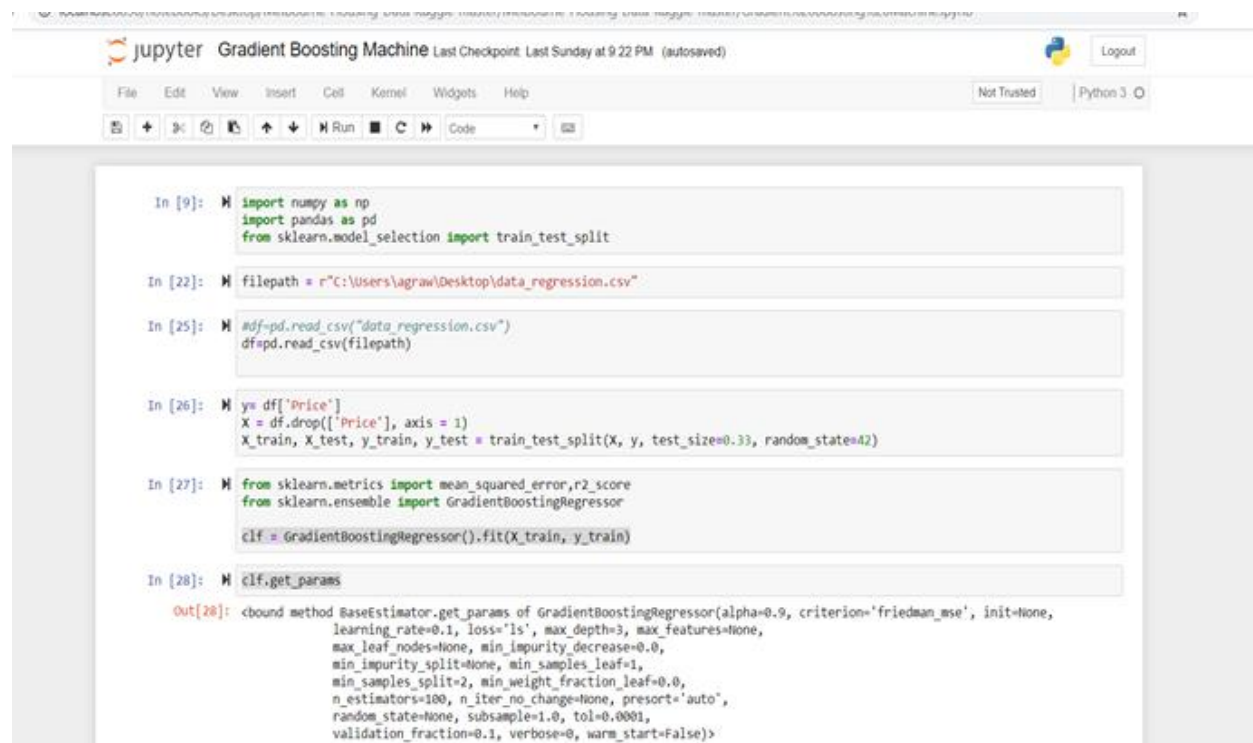
```
Out[20]: array([ 0.5,  1.5,  2.5,  3.5,  4.5,  5.5,  6.5,  7.5,  8.5,  9.5, 10.5,
11.5, 12.5, 13.5, 14.5, 15.5, 16.5, 17.5, 18.5, 19.5, 20.5, 21.5,
22.5, 23.5, 24.5, 25.5, 26.5, 27.5, 28.5, 29.5, 30.5, 31.5, 32.5,
33.5, 34.5, 35.5, 36.5, 37.5, 38.5, 39.5, 40.5, 41.5, 42.5, 43.5,
44.5, 45.5, 46.5, 47.5, 48.5, 49.5, 50.5, 51.5, 52.5, 53.5, 54.5,
55.5, 56.5, 57.5])
```

9.0 Gradient Boosting Regressor

Analysis using Gradient Boosting Regressor-

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage regression trees fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced. Parameters which are constant throughout Model

Loss	MSE
'max_depth'	None
'warm_start'	False
Min_samples_leaf	1
Package	GradientBoostingRegressor SKLearn



```
In [9]: import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split

In [22]: filepath = r"C:\Users\agrawa\Desktop\data_regression.csv"

In [25]: df=pd.read_csv("data_regression.csv")
df=pd.read_csv(filepath)

In [26]: y= df['Price']
X = df.drop(['Price'], axis = 1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)

In [27]: from sklearn.metrics import mean_squared_error, r2_score
from sklearn.ensemble import GradientBoostingRegressor

clf = GradientBoostingRegressor().fit(X_train, y_train)

In [28]: clf.get_params

Out[28]: <bound method BaseEstimator.get_params of GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
learning_rate=0.1, loss='ls', max_depth=3, max_features=None,
max_leaf_nodes=None, min_impurity_decrease=0.0,
min_impurity_split=None, min_samples_leaf=1,
min_samples_split=2, min_weight_fraction_leaf=0.0,
n_estimators=100, n_iter_no_change=None, presort='auto',
random_state=None, subsample=1.0, tol=0.0001,
validation_fraction=0.1, verbose=0, warm_start=False)>
```

Parameter Tuning the Model to improve Prediction-

```
random_state=None, subsample=1.0, tol=0.0001,
validation_fraction=0.1, verbose=0, warm_start=False)>

In [33]: M p_test3 = {'learning_rate':[0.15,0.1,0.05,0.01,0.005,0.001], 'n_estimators':[100,250,500,750,1000,1250]}

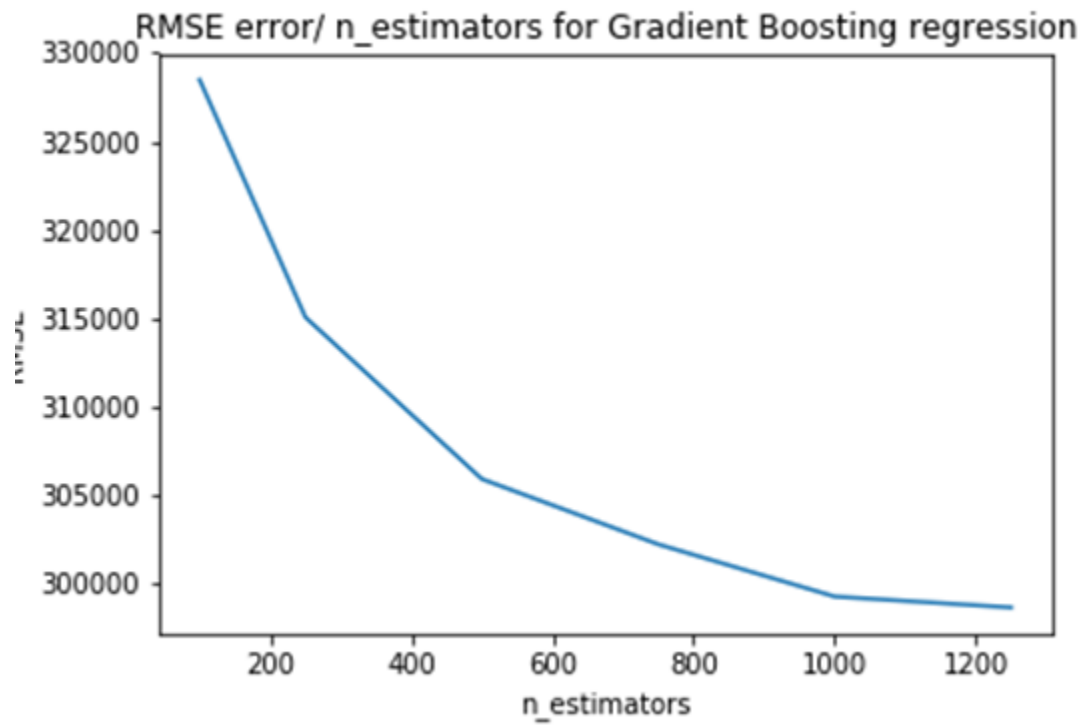
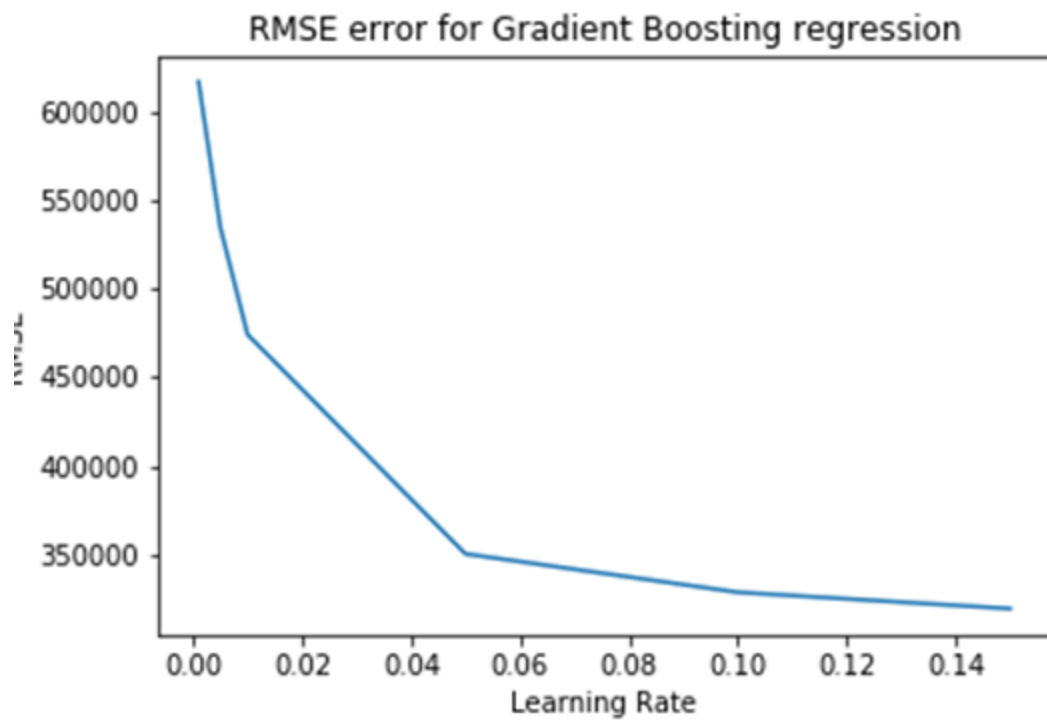
In [31]: M rmse = np.sqrt(mean_squared_error(y_test,clf.predict(X_test)))
rmse

Out[31]: 328497.3144657096

In [ ]: M error_depth = []
# for d in p_test3['learning_rate']:
for d in p_test3['n_estimators']:
    print(d)
    rf_best = GradientBoostingRegressor(n_estimators= d ) #For learning_rate
    #rf_best = RandomForestRegressor(random_state = 42,n_estimators = d ) #For n_estimators
    rf_best.fit(X_train, y_train)
    predictions_best=rf_best.predict(X_test)
    error_depth.append( np.sqrt(mean_squared_error(y_test,rf_best.predict(X_test))))
```

Learning Rate- learning rate shrinks the contribution of each tree by learning rate. There is a trade-off between learning rate and n_estimators. The range tested was [0.15,0.1,0.05,0.01,0.005,0.001]

No. Of estimators- The number of boosting stages to perform. Gradient boosting is fairly robust to overfitting, so a large number usually results in better performance.



GridSearch for parameters-

I have used the following parameters for Gradient Boosting changing only learning parameters and No. of estimators. {max_depth=4, min_samples_split=2, min_samples_leaf=1, subsample=1, max_features='sqrt', random_state=10), param_grid = p_test3, scoring='explained_variance', n_jobs=4, id=False, cv=5}

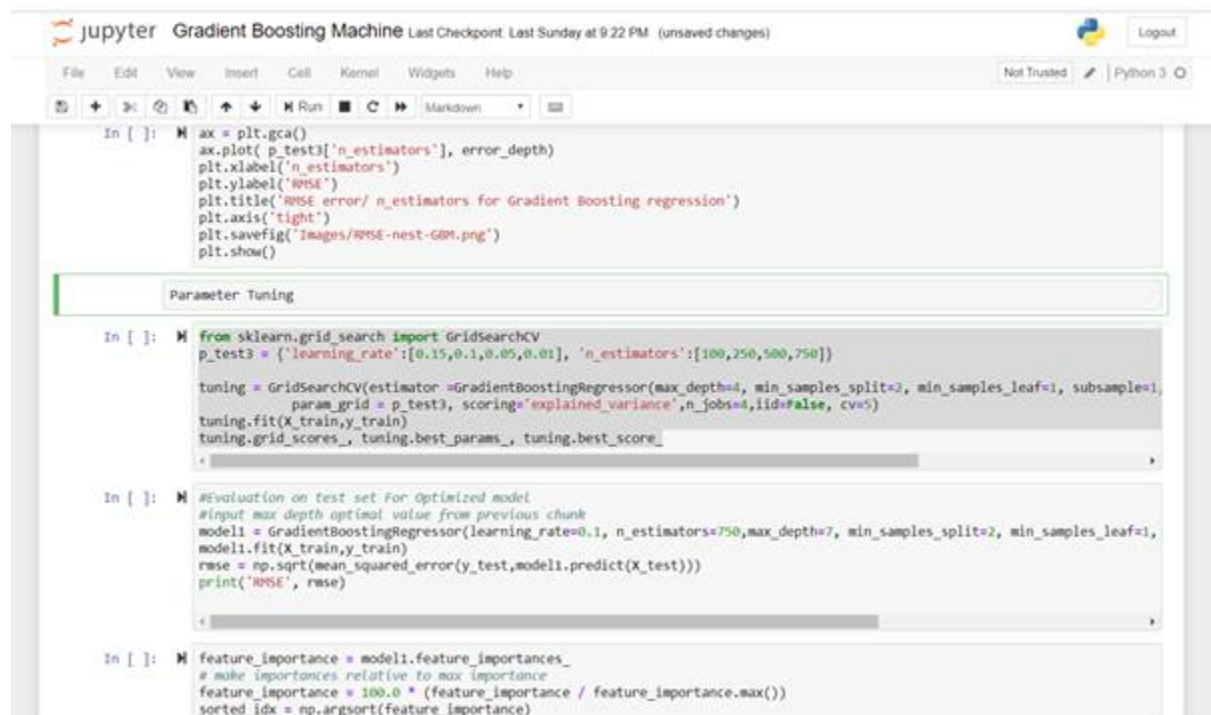
Also, I have used RandomizedSearchCV to perform parameter tuning with cv = 3. The parameters of the estimator used to apply these methods are optimized by cross-validated search over parameter settings.

Before Tuning

RMSE	328474.20397842064
Learning_rate	0.1
No. of estimators	100

After Tuning

RMSE	291257.119966
Learning_rate	0.1
No. of estimators	750



```
In [ ]: M ax = plt.gca()
ax.plot(p_test3['n_estimators'], error_depth)
plt.xlabel('n_estimators')
plt.ylabel('RMSE')
plt.title('RMSE error/ n_estimators for Gradient Boosting regression')
plt.axis('tight')
plt.savefig('images/RMSE-nest-GBM.png')
plt.show()

Parameter Tuning

In [ ]: M from sklearn.grid_search import GridSearchCV
p_test3 = {'learning_rate':[0.15,0.1,0.05,0.01], 'n_estimators':[100,250,500,750]}

tuning = GridSearchCV(estimator=GradientBoostingRegressor(max_depth=4, min_samples_split=2, min_samples_leaf=1, subsample=1,
param_grid = p_test3, scoring='explained_variance', n_jobs=4, iid=False, cv=5)
tuning.fit(X_train,y_train)
tuning.grid_scores_, tuning.best_params_, tuning.best_score_

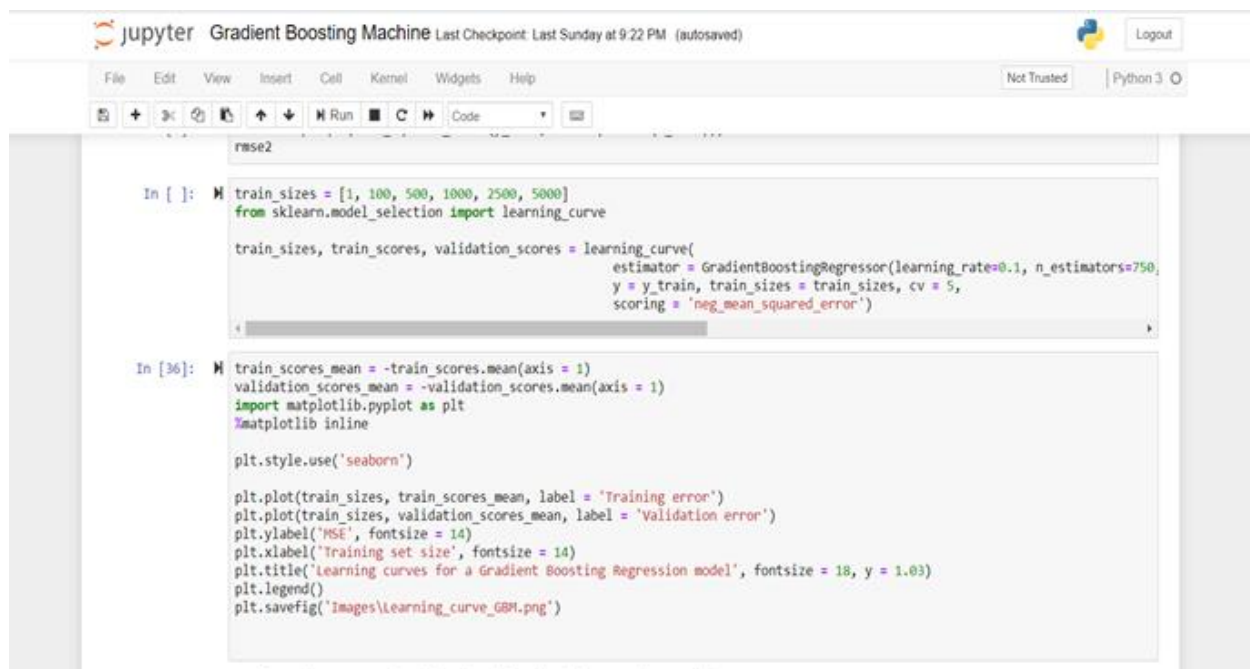
In [ ]: M #Evaluation on test set For Optimized model
#input max depth optimal value from previous chunk
model1 = GradientBoostingRegressor(learning_rate=0.1, n_estimators=750,max_depth=7, min_samples_split=2, min_samples_leaf=1,
model1.fit(X_train,y_train)
rmse = np.sqrt(mean_squared_error(y_test,model1.predict(X_test)))
print('RMSE', rmse)

In [ ]: M feature_importance = model1.feature_importances_
# make importance relative to max importance
feature_importance = 100.0 * (feature_importance / feature_importance.max())
sorted_idx = np.argsort(feature_importance)
```

OBSERVATIONS-

Learning Curve Observations-

1. The validation curve could converge toward the training curve if more training instances were added
2. As the training error upon converging is low, the model has a low bias / high variance problem.
3. As the gap between training error and validation error is large, it suggests high variance. Also, due to near zero training error, we can conclude that the model over fits the training data



```

ruse2

In [ ]: train_sizes = [1, 100, 500, 1000, 2500, 5000]
        from sklearn.model_selection import learning_curve

        train_sizes, train_scores, validation_scores = learning_curve(
            estimator = GradientBoostingRegressor(learning_rate=0.1, n_estimators=750,
            y = y_train, train_sizes = train_sizes, cv = 5,
            scoring = 'neg_mean_squared_error')

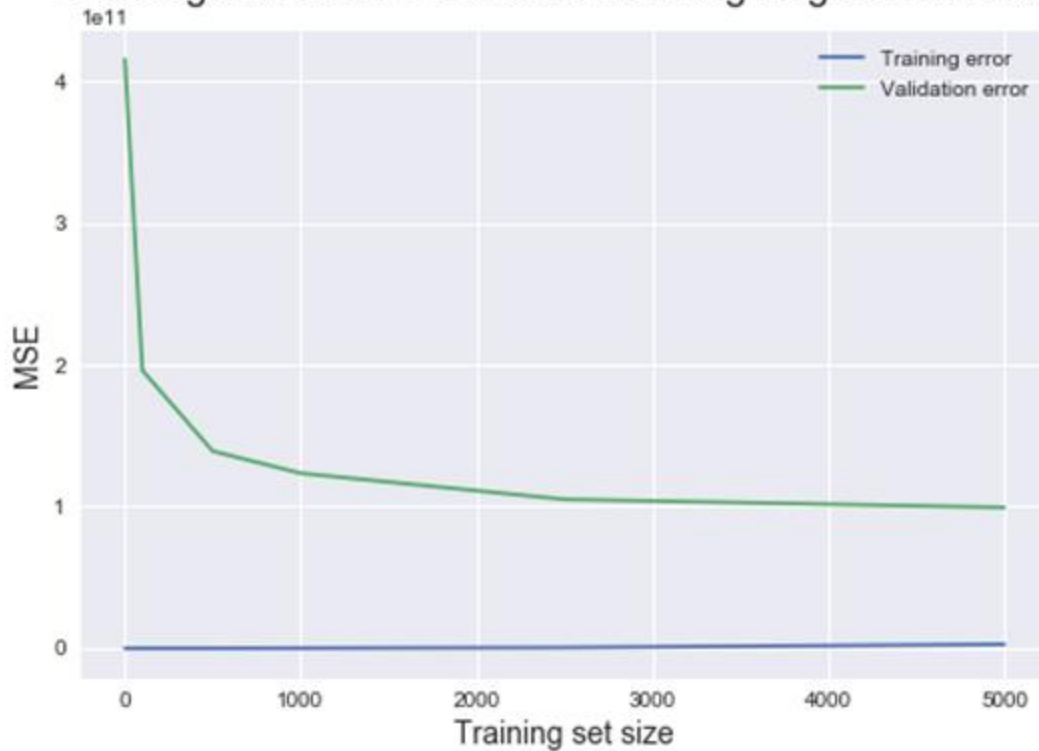
In [36]: train_scores_mean = -train_scores.mean(axis = 1)
        validation_scores_mean = -validation_scores.mean(axis = 1)
        import matplotlib.pyplot as plt
        %matplotlib inline

        plt.style.use('seaborn')

        plt.plot(train_sizes, train_scores_mean, label = 'Training error')
        plt.plot(train_sizes, validation_scores_mean, label = 'Validation error')
        plt.ylabel('MSE', fontsize = 14)
        plt.xlabel('Training set size', fontsize = 14)
        plt.title('Learning curves for a Gradient Boosting Regression model', fontsize = 18, y = 1.03)
        plt.legend()
        plt.savefig('images\\Learning_curve_GBM.png')

```

Learning curves for a Gradient Boosting Regression model



Feature Selection Observations-

1. The following are the most important features.
2. These will be useful in comparing models in building modified model at the end.

```

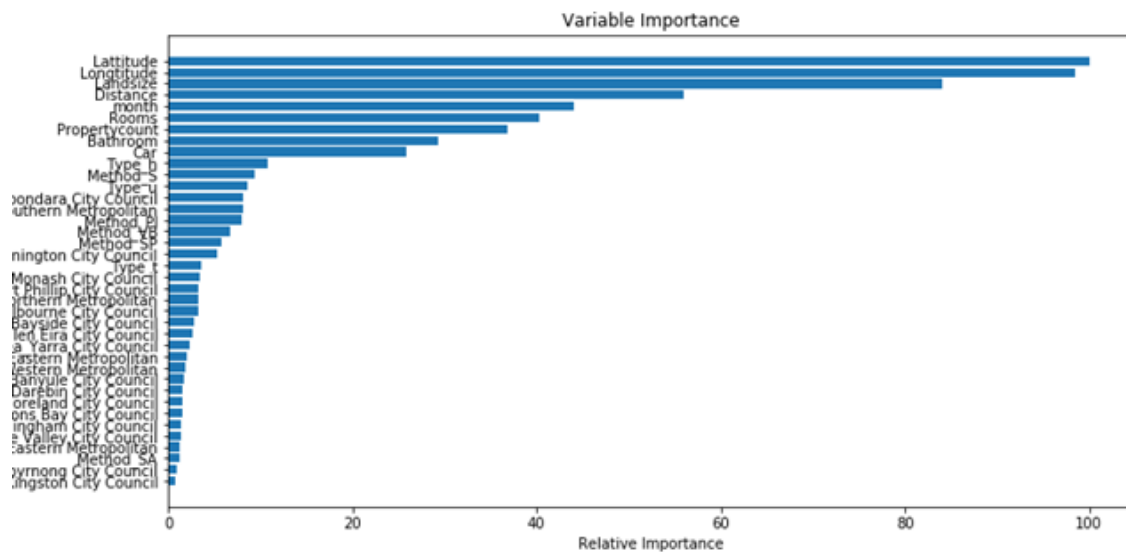
jupyter Gradient Boosting Machine Last Checkpoint: Last Sunday at 9:22 PM (autosaved)
File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3

In [ ]: #evaluation on test set for optimized model
#input max depth optimal value from previous chunk
model1 = GradientBoostingRegressor(learning_rate=0.1, n_estimators=750, max_depth=7, min_samples_split=2, min_samples_leaf=1,
model1.fit(X_train, y_train)
rmse = np.sqrt(mean_squared_error(y_test, model1.predict(X_test)))
print('RMSE', rmse)

In [ ]: feature_importance = model1.feature_importances_
# make importances relative to max importance
feature_importance = 100.0 * (feature_importance / feature_importance.max())
sorted_idx = np.argsort(feature_importance)
pos = np.arange(sorted_idx.shape[0]) + .5
plt.figure(figsize=(12, 6))
plt.subplot(1, 1, 1)
plt.barh(pos[20:], feature_importance[sorted_idx][20:], align='center')
plt.yticks(pos[20:], X.columns[sorted_idx][20:])
plt.xlabel('Relative Importance')
plt.title('Variable Importance')
plt.savefig('Images/Var_imp_gbm.png')
plt.show()

```

Output Plot for Feature Selection-




Optimization of GBM Model to improve accuracy-

Ensemble methods are used to use predictions of several weak estimators built to improve generalization/ robustness over a single estimator.

Boosting methods - Base estimators are built sequentially and one tries to reduce the bias of the combined estimator. It aims in improving the accuracy of the weak model. We apply base learning (ML) algorithms with a different distribution. Each time base learning algorithm is applied, it generates a new weak prediction rule. This is an iterative process. After many iterations, the boosting algorithm combines these weak rules into a single strong prediction rule.

Methodology Using GBM-

I have used the clusters created earlier and assigned them as features. As they are categorical variables, they get converted to One hot vectors. Rest of the features remain the same. Based on the best features predicted by Random Forest regressor we will train GBM and Tune it to improve prediction accuracy.

jupyter Housing_new method (autosaved)  Logout

File Edit View Insert Cell Kernel Widgets Help Not Trusted Python 3

```
-----
CouncilArea_Mitchell Shire Council    1.27793e-06
CouncilArea_Moorabool Shire Council  3.66143e-07
-----
```

```
In [16]: #best features based on decimal points judgement and include target variable
best_features=['Regionname_Southern Metropolitan','Rooms','Distance','Landsize','Longitude','Latitude','Type_u','month','Type_u','Regionname_Eastern Metropolitan','Bathroom','Propertycount','Car','Price','CouncilArea_Stonnington City Council','Method_S','Method_P1','Method_V6','CouncilArea_Kingston City Council','Method_S6','Type_t','CouncilArea_Boroondara','CouncilArea_Monash City Council','CouncilArea_Russekerry Valley City Council']
```

GBM MODEL with new features

```
In [17]: # df_modf_c[best_features]
y= df_m['Price']
x = df_m.drop(['Price'], axis = 1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
```

```
In [23]: #base case error without tuning
from sklearn.metrics import mean_squared_error,r2_score
from sklearn.ensemble import GradientBoostingRegressor

clf = GradientBoostingRegressor().fit(X_train, y_train)
```

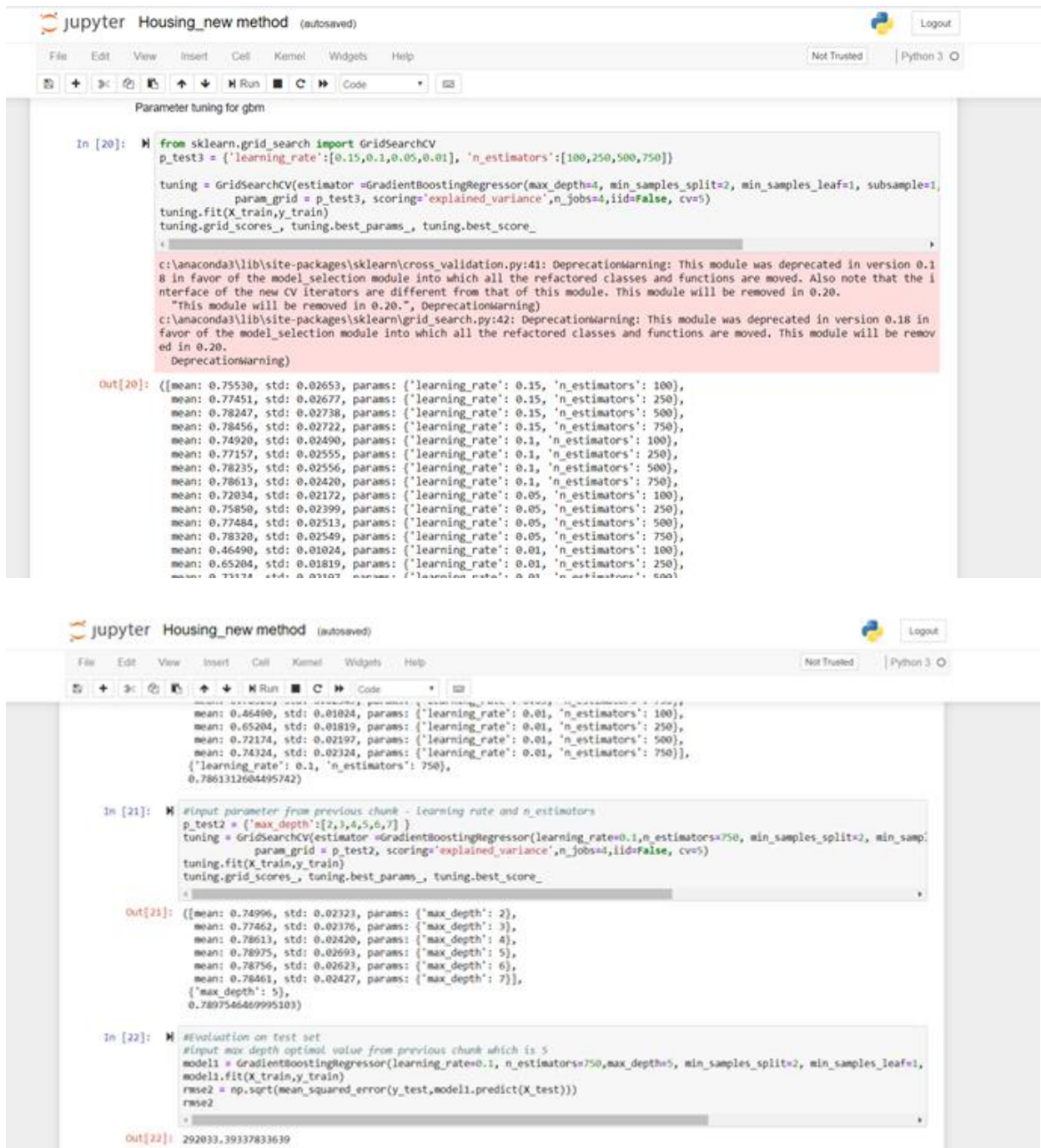
```
In [24]: # rmse = np.sqrt(mean_squared_error(y_test,clf.predict(X_test)))
rmse
```

```
Out[24]: 334115.86994414993
```

Parameter tuning for gbm

```
In [25]: # Best GBM model with cross-validation
```

Parameter Tuning GBM After feature Selection-



```
In [20]: from sklearn.grid_search import GridSearchCV
p_test3 = {'learning_rate':[0.15,0.1,0.05,0.01], 'n_estimators':[100,250,500,750]}

tuning = GridSearchCV(estimator=GradientBoostingRegressor(max_depth=4, min_samples_split=2, min_samples_leaf=1, subsample=1,
param_grid = p_test3, scoring='explained_variance',n_jobs=4,iid=False, cv=5)
tuning.fit(X_train,y_train)
tuning.grid_scores_, tuning.best_params_, tuning.best_score_

c:\anaconda3\lib\site-packages\sklearn\cross_validation.py:41: DeprecationWarning: This module was deprecated in version 0.18 in favor of the model_selection module into which all the refactored classes and functions are moved. Also note that the interface of the new CV iterators are different from that of this module. This module will be removed in 0.20.
"This module will be removed in 0.20."
c:\anaconda3\lib\site-packages\sklearn\grid_search.py:42: DeprecationWarning: This module was deprecated in version 0.18 in favor of the model_selection module into which all the refactored classes and functions are moved. This module will be removed in 0.20.
DeprecationWarning)

Out[20]: ([mean: 0.75530, std: 0.02653, params: {'learning_rate': 0.15, 'n_estimators': 100},
mean: 0.77451, std: 0.02677, params: {'learning_rate': 0.15, 'n_estimators': 250},
mean: 0.78247, std: 0.02738, params: {'learning_rate': 0.15, 'n_estimators': 500},
mean: 0.78456, std: 0.02722, params: {'learning_rate': 0.15, 'n_estimators': 750},
mean: 0.74920, std: 0.02490, params: {'learning_rate': 0.1, 'n_estimators': 100},
mean: 0.77157, std: 0.02555, params: {'learning_rate': 0.1, 'n_estimators': 250},
mean: 0.78235, std: 0.02556, params: {'learning_rate': 0.1, 'n_estimators': 500},
mean: 0.78613, std: 0.02420, params: {'learning_rate': 0.1, 'n_estimators': 750},
mean: 0.72034, std: 0.02172, params: {'learning_rate': 0.05, 'n_estimators': 100},
mean: 0.75850, std: 0.02399, params: {'learning_rate': 0.05, 'n_estimators': 250},
mean: 0.77484, std: 0.02513, params: {'learning_rate': 0.05, 'n_estimators': 500},
mean: 0.78320, std: 0.02549, params: {'learning_rate': 0.05, 'n_estimators': 750},
mean: 0.46490, std: 0.01024, params: {'learning_rate': 0.01, 'n_estimators': 100},
mean: 0.65204, std: 0.01819, params: {'learning_rate': 0.01, 'n_estimators': 250},
mean: 0.73174, std: 0.02107, params: {'learning_rate': 0.01, 'n_estimators': 500},
mean: 0.73174, std: 0.02107, params: {'learning_rate': 0.01, 'n_estimators': 750}],
{'learning_rate': 0.15, 'n_estimators': 750},
0.7845609995103)

In [21]: #Input parameter from previous chunk - learning rate and n_estimators
p_test2 = {'max_depth':[2,3,4,5,6,7]}
tuning = GridSearchCV(estimator=GradientBoostingRegressor(learning_rate=0.1,n_estimators=750, min_samples_split=2, min_samples_leaf=1,
param_grid = p_test2, scoring='explained_variance',n_jobs=4,iid=False, cv=5)
tuning.fit(X_train,y_train)
tuning.grid_scores_, tuning.best_params_, tuning.best_score_

Out[21]: ([mean: 0.74996, std: 0.02323, params: {'max_depth': 2},
mean: 0.77462, std: 0.02376, params: {'max_depth': 3},
mean: 0.78613, std: 0.02420, params: {'max_depth': 4},
mean: 0.78975, std: 0.02693, params: {'max_depth': 5},
mean: 0.78756, std: 0.02623, params: {'max_depth': 6},
mean: 0.78461, std: 0.02427, params: {'max_depth': 7}],
{'max_depth': 5},
0.789756669995103)

In [22]: #Evaluation on test set
#Input max depth optimal value from previous chunk which is 5
model1 = GradientBoostingRegressor(learning_rate=0.1, n_estimators=750,max_depth=5, min_samples_split=2, min_samples_leaf=1,
model1.fit(X_train,y_train)
rmse2 = np.sqrt(mean_squared_error(y_test,model1.predict(X_test)))
rmse2

Out[22]: 292033.39337833639
```

Results of the Tuned GBM Model –

Tuned the new GBM on the same set of parameters range as earlier (n_estimators, learning rate) and max depth of a single tree which is tuned over the range {'max_depth':[2,3,4,5,6,7] }.

Model	MSE
GBM (pre-feature selection, tuned	291257.119966
GBM (feature selection, not tuned	334113.86904414993
GBM (feature selection, tuned)	292033.3933783364

10.0 Hypertuning of Parameters

Comparison of Models based on Analysis-

1. **Ridge V/s Lasso-** Lasso doesn't significantly reduce the RMSE error significantly but Lasso is useful in feature selection which is used in the future models saving computation. Ridge doesn't do feature selection. Ridge is computationally less expensive than Lasso.
2. **Random Forest V/s Gradient Boosting Machine** – GBM is the best performing model after parameter tuning. There is not a huge effect on accuracy. However Random Forest Regressor is more computationally effective.
3. **Ridge/Lasso V/s Linear Regression V/s Subset Selection** – I chose Ridge and Lasso regression over other linear regression methods as they have a regularization effect which worked well with the high dimensional dataset I had. Linear model was too simplified to correctly model this data and subset selection was computationally expensive due to the large number of features.
4. **Random Forest/ Gradient Boosting Machine V/s KNN** – KNN would have been computationally expensive and I was not convinced with my cluster results in the previous results. Due to poor clusters, I anticipated KNN to perform poorly. On the other hand, Boosting and Random Forest are based on Decision Trees which can learn complex models.

Comparison Table:

Model	MSE (Tuned/Best Model)
Random Forest Regressor	305924.8830873183
Gradient Boosting Regressor	291257.119966
Lasso Regression	468087.6284
Ridge Regression	458294.7192598626
Linear Regression	511331.4195059427

11.0 Conclusion

Hence, Gradient Boosting Regressor gives best accuracy after tuning the Parameters out of all the other used models for the Housing Price Prediction.

Final Observation

- We observe that the selected feature selection method doesn't work well. Perhaps, tuning the number of features to include from Random Forest would improve the feature.
- Also, we haven't scaled any features, doing that in the future can improve accuracy
- The overall computation time of the modified model has decreased with considerable improvement in error.

12.0 Future Scope

In Future, we wish to use below methodology to further improve results by hyper tuning best selected features as hyper parameters predicted by random forest model using GBM -

For feature selection, Random Forest provides mean decrease impurity and mean decrease accuracy. Random forest consists of several decision trees. Every node in the decision trees is a condition on a single feature, designed to split the dataset into two so that similar response values end up in the same set. The measure based on which the (node) optimal condition is chosen is called impurity (variance). Using the training set, it computes how much each feature decreases the weighted impurity in a tree. Combining these decision trees (Forest) the impurity decreases from each feature which is averaged and ranked to give the most important features. As a result, we can use top n features from Random Forest and input those into the boosting model. The n feature becomes a hyperparameter and can be tuned. We haven't tuned it in this section, but this can be done to improve future results.

13.0 References

- <https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html>
- <https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0162259>
- <https://medium.com/datadriveninvestor/random-forest-regression-9871bc9a25eb>
- <https://www.statisticshowto.com/lasso-regression/>
- <https://towardsdatascience.com/an-implementation-and-explanation-of-the-random-forest-in-python-77bf308a9b76>
- <https://www.investopedia.com/terms/m/mlr.asp>
- <https://www.investopedia.com/terms/r/r-squared.asp>
- <https://mindmajix.com/ridge-regression>
- <https://www.udemy.com/course/machinelearning/learn/lecture/5732732#overview>
- <https://medium.com/all-things-ai/in-depth-parameter-tuning-for-gradient-boosting-3363992e9bae>