SELECTING THE BEST MODEL TO PREDICT HOUSING PURCHASING AMOUNT USING REGRESSION

BUDI SALEH, Created on 24 Sep 20

By using the boston dataset from Kagle and we would like to develop a model to predict the total price housing amount that customers are willing to pay given the following attributes:

- No of bedrooms
- square foot (living room,lot, above, and basement)
- City
- View
- Condition
- Year built and renovated
- Bathrooms

The model should predict:

- Price of House

This exercise to find the best regression model based on the

- Mean square error
- Mean absolute error
- r2 score

Linear Model are using as follow :

1. Linear regression as baseline
2. Ridge
3. Lasso
4. KNN
5. Decission Tree
6. Random Forrest
7. Support Vector

Import the library

In [1]:
```
import matplotlib as mp
import sklearn
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LinearRegression, Ridge, Lasso
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import cross_val_score, train_test_split, GridSea
rchCV
from sklearn.metrics import mean_squared_error as mse
from sklearn.metrics import r2_score as rs
from sklearn.metrics import mean_absolute_error as mae
import category_encoders as ce
from math import sqrt
from sklearn.svm import SVR # Support Vector Regressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeRegressor
from sklearn.svm import LinearSVR
# to perform hyperparameter tuning
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
```

In [2]:
```
%matplotlib inline
```

Using data download from Kagle for Boston Housing dataset - Read using panda

In [3]:
```
df = pd.read_csv('data.csv')
```

In [4]:
```
df
```

Out[4]:

	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	vie
0	2014-05-02 00:00:00	3.130000e+05	3.0	1.50	1340	7912	1.5	0	
1	2014-05-02 00:00:00	2.384000e+06	5.0	2.50	3650	9050	2.0	0	
2	2014-05-02 00:00:00	3.420000e+05	3.0	2.00	1930	11947	1.0	0	
3	2014-05-02 00:00:00	4.200000e+05	3.0	2.25	2000	8030	1.0	0	
4	2014-05-02 00:00:00	5.500000e+05	4.0	2.50	1940	10500	1.0	0	
...	...	...	...	...	...	...	...	...	
4595	2014-07-09 00:00:00	3.081667e+05	3.0	1.75	1510	6360	1.0	0	
4596	2014-07-09 00:00:00	5.343333e+05	3.0	2.50	1460	7573	2.0	0	
4597	2014-07-09 00:00:00	4.169042e+05	3.0	2.50	3010	7014	2.0	0	
4598	2014-07-10 00:00:00	2.034000e+05	4.0	2.00	2090	6630	1.0	0	
4599	2014-07-10 00:00:00	2.206000e+05	3.0	2.50	1490	8102	2.0	0	

4600 rows × 18 columns

Explarotary Data Analysis

In [5]:
```
df.describe()
```

Out[5]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wate
count	4.600000e+03	4600.000000	4600.000000	4600.000000	4.600000e+03	4600.000000	4600.0
mean	5.519630e+05	3.400870	2.160815	2139.346957	1.485252e+04	1.512065	0.0
std	5.638347e+05	0.908848	0.783781	963.206916	3.588444e+04	0.538288	0.0
min	0.000000e+00	0.000000	0.000000	370.000000	6.380000e+02	1.000000	0.0
25%	3.228750e+05	3.000000	1.750000	1460.000000	5.000750e+03	1.000000	0.0
50%	4.609435e+05	3.000000	2.250000	1980.000000	7.683000e+03	1.500000	0.0
75%	6.549625e+05	4.000000	2.500000	2620.000000	1.100125e+04	2.000000	0.0
max	2.659000e+07	9.000000	8.000000	13540.000000	1.074218e+06	3.500000	1.0

In [6]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4600 entries, 0 to 4599
Data columns (total 18 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   date           4600 non-null   object 
 1   price          4600 non-null   float64
 2   bedrooms       4600 non-null   float64
 3   bathrooms      4600 non-null   float64
 4   sqft_living    4600 non-null   int64  
 5   sqft_lot       4600 non-null   int64  
 6   floors         4600 non-null   float64
 7   waterfront     4600 non-null   int64  
 8   view           4600 non-null   int64  
 9   condition      4600 non-null   int64  
 10  sqft_above     4600 non-null   int64  
 11  sqft_basement  4600 non-null   int64  
 12  yr_built       4600 non-null   int64  
 13  yr_renovated   4600 non-null   int64  
 14  street         4600 non-null   object 
 15  city           4600 non-null   object 
 16  statezip       4600 non-null   object 
 17  country        4600 non-null   object 
dtypes: float64(4), int64(9), object(5)
memory usage: 647.0+ KB
```

In [7]:
```
df.isnull().sum()
```

Out[7]:
```
date             0
price            0
bedrooms         0
bathrooms        0
sqft_living      0
sqft_lot         0
floors           0
waterfront       0
view             0
condition        0
sqft_above       0
sqft_basement    0
yr_built         0
yr_renovated     0
street           0
city             0
statezip         0
country          0
dtype: int64
```

Drop unnecessary column

In [8]:
```
df2 = df.drop(['date','street','statezip', 'country'], axis =1)
```

In [9]:
```
df2
```

Out[9]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	conditi
0	3.130000e+05	3.0	1.50	1340	7912	1.5	0	0	
1	2.384000e+06	5.0	2.50	3650	9050	2.0	0	4	
2	3.420000e+05	3.0	2.00	1930	11947	1.0	0	0	
3	4.200000e+05	3.0	2.25	2000	8030	1.0	0	0	
4	5.500000e+05	4.0	2.50	1940	10500	1.0	0	0	
...	...	...	...	...	...	...	...	...	
4595	3.081667e+05	3.0	1.75	1510	6360	1.0	0	0	
4596	5.343333e+05	3.0	2.50	1460	7573	2.0	0	0	
4597	4.169042e+05	3.0	2.50	3010	7014	2.0	0	0	
4598	2.034000e+05	4.0	2.00	2090	6630	1.0	0	0	
4599	2.206000e+05	3.0	2.50	1490	8102	2.0	0	0	

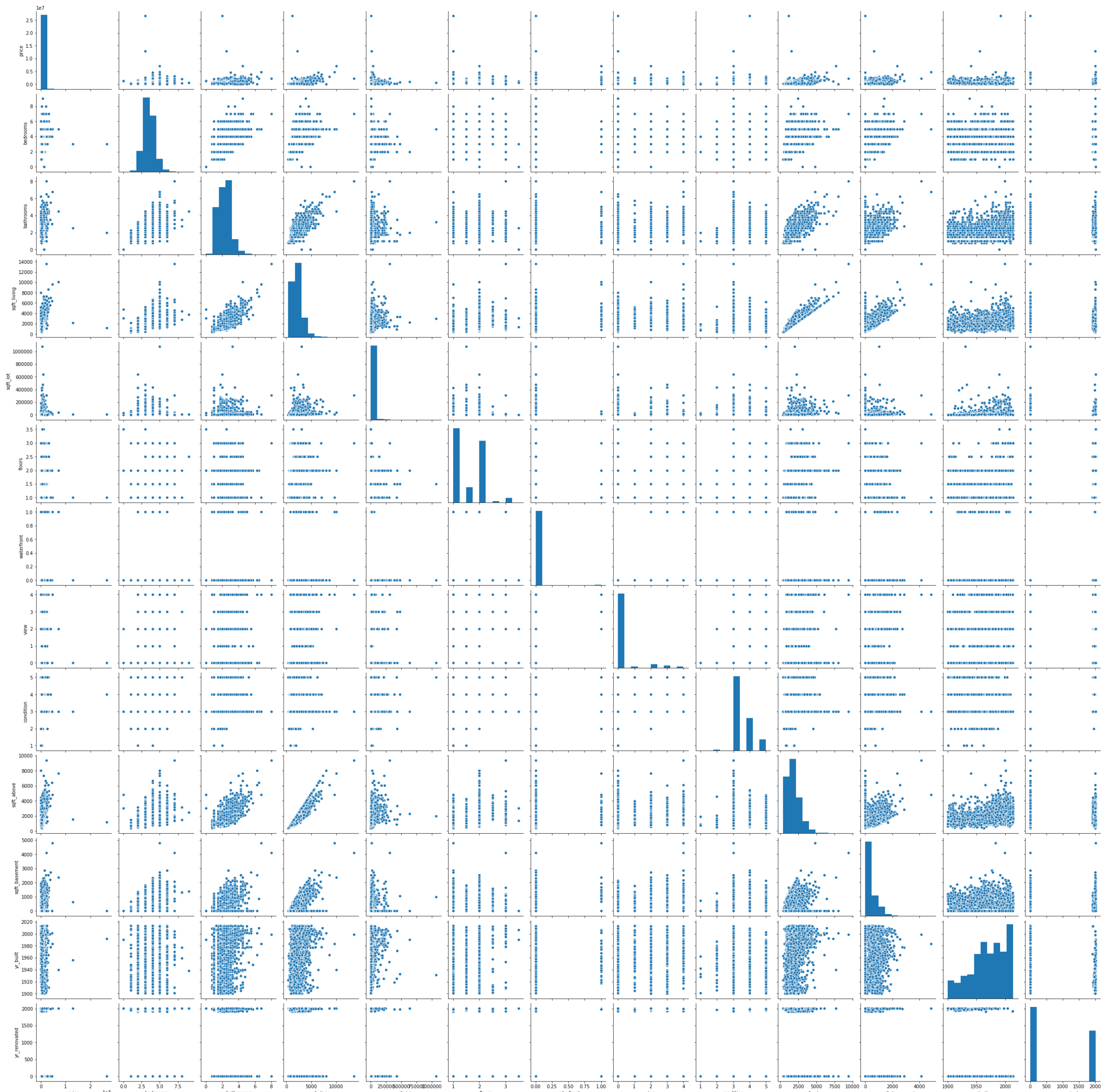
4600 rows × 14 columns

Visual the data

Using pairplot to see the corelation

In [10]:
```
sns.pairplot(df2)
```

Out[10]: <seaborn.axisgrid.PairGrid at 0x5493dc8>

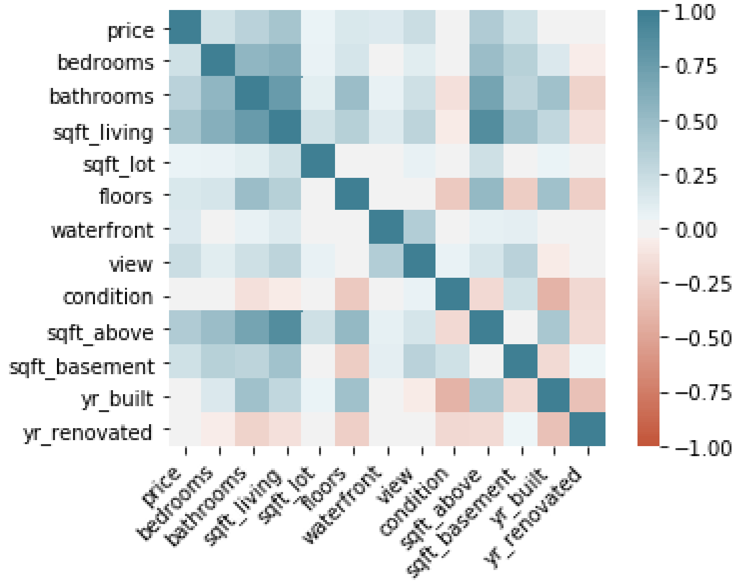


From the graph, it can seen that there are relation with regression

Create a correlation heatmap

In [11]:
```
corr = df2.corr()
```

In [12]:
```
ax = sns.heatmap(
    corr,
    vmin=-1, vmax=1, center=0,
    cmap=sns.diverging_palette(20, 220, n=200),
    square=True
)
ax.set_xticklabels(
    ax.get_xticklabels(),
    rotation=45,
    horizontalalignment='right'
);
```



Correlation table

In [13]:

```python
cmap = cmap=sns.diverging_palette(5, 250, as_cmap=True)

def magnify():
    return [dict(selector="th",
                 props=[("font-size", "7pt")]),
            dict(selector="td",
                 props=[('padding', "0em 0em")]),
            dict(selector="th:hover",
                 props=[("font-size", "12pt")]),
            dict(selector="tr:hover td:hover",
                 props=[('max-width', '200px'),
                        ('font-size', '12pt')])
]

corr.style.background_gradient(cmap, axis=1)\
    .set_properties(**{'max-width': '80px', 'font-size': '10pt'})\
    .set_caption("Correlation Table")\
    .set_precision(2)\
    .set_table_styles(magnify())
```

Out[13]:

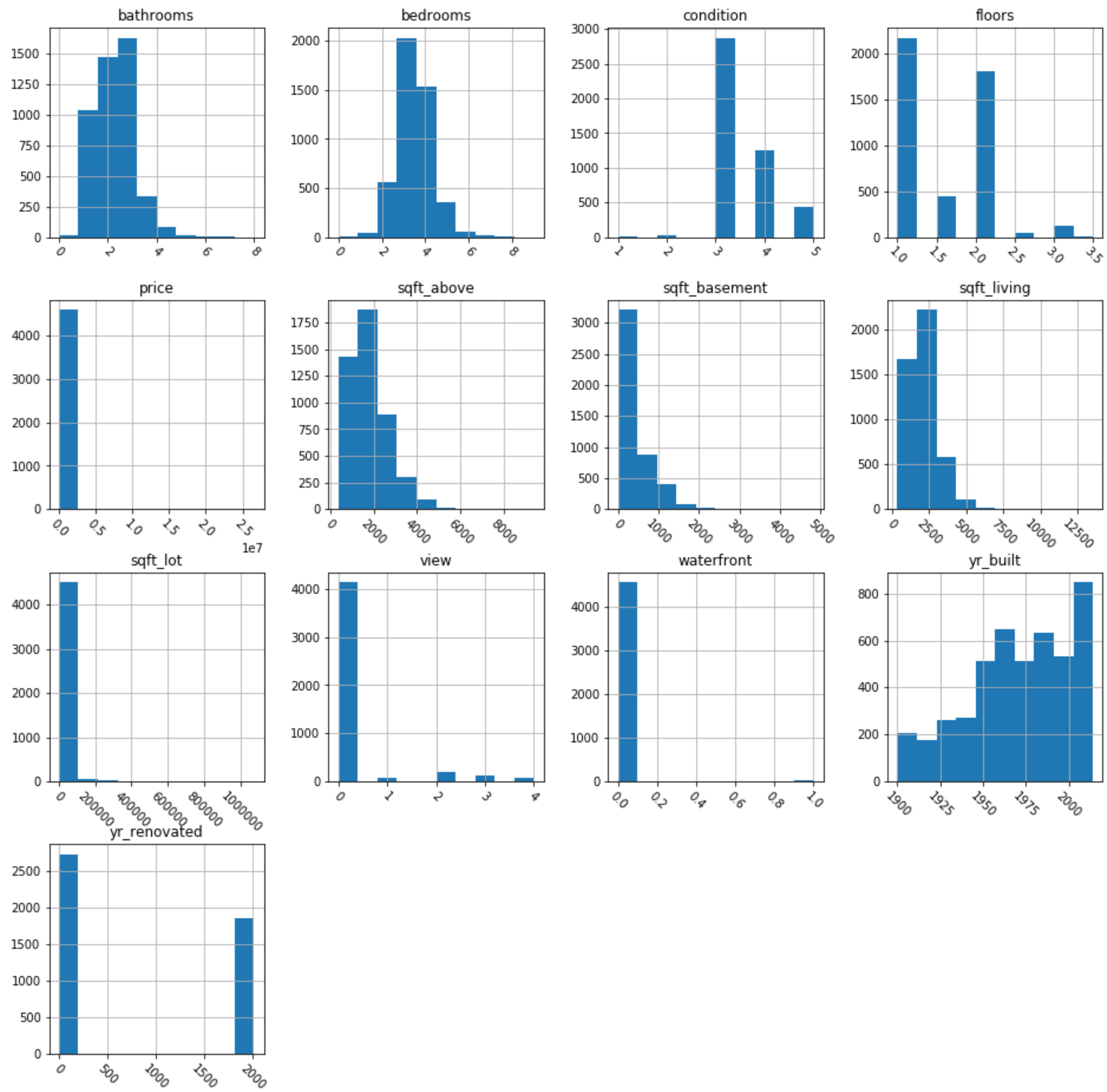
Correlation Table

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	sqft_above	sqft_base
price	1.00	0.20	0.33	0.43	0.05	0.15	0.14	0.23	0.03	0.37	
bedrooms	0.20	1.00	0.55	0.59	0.07	0.18	-0.00	0.11	0.03	0.48	
bathrooms	0.33	0.55	1.00	0.76	0.11	0.49	0.08	0.21	-0.12	0.69	
sqft_living	0.43	0.59	0.76	1.00	0.21	0.34	0.12	0.31	-0.06	0.88	
sqft_lot	0.05	0.07	0.11	0.21	1.00	0.00	0.02	0.07	0.00	0.22	
floors	0.15	0.18	0.49	0.34	0.00	1.00	0.02	0.03	-0.28	0.52	-
waterfront	0.14	-0.00	0.08	0.12	0.02	0.02	1.00	0.36	0.00	0.08	
view	0.23	0.11	0.21	0.31	0.07	0.03	0.36	1.00	0.06	0.17	
condition	0.03	0.03	-0.12	-0.06	0.00	-0.28	0.00	0.06	1.00	-0.18	
sqft_above	0.37	0.48	0.69	0.88	0.22	0.52	0.08	0.17	-0.18	1.00	-
sqft_basement	0.21	0.33	0.30	0.45	0.03	-0.26	0.10	0.32	0.20	-0.04	
yr_built	0.02	0.14	0.46	0.29	0.05	0.47	-0.02	-0.06	-0.40	0.41	-
yr_renovated	-0.03	-0.06	-0.22	-0.12	-0.02	-0.23	0.01	0.02	-0.19	-0.16	

There are correlation for price to all columns with positive correlation and negative with year renovated

Plot histogram

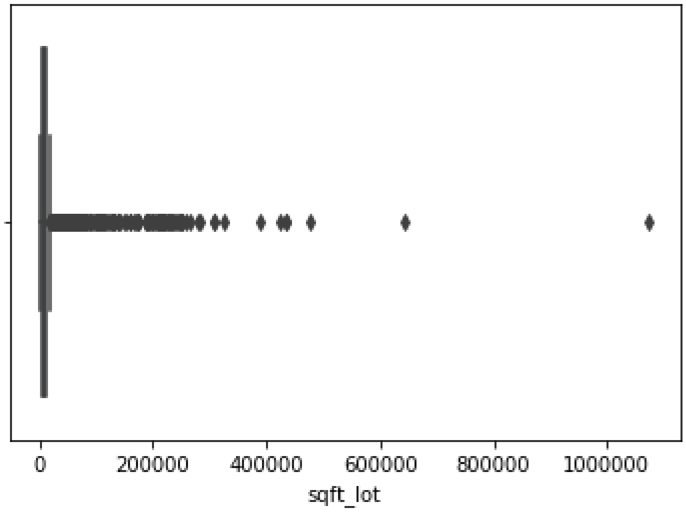
In [14]:
```
# Plot histogram grid
df2.hist(figsize=(16,16), xrot=-45) ## Display the labels rotated by 45 degres
s

# Clear the text "residue"
plt.show()
```



Using boxplot to see the outliers

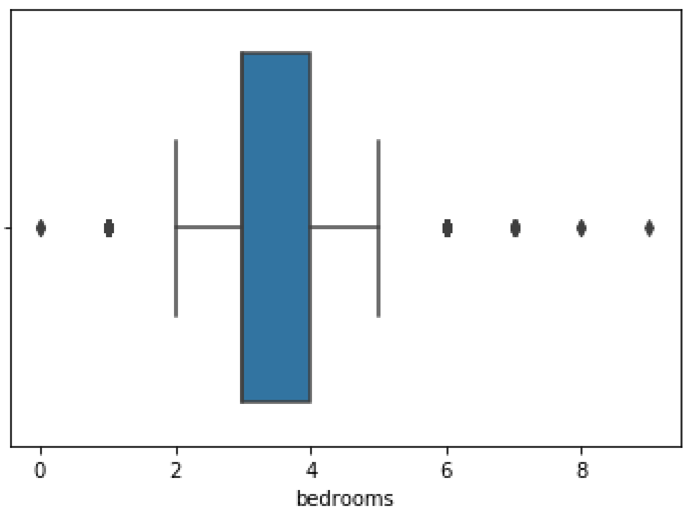
In [15]:
```
sns.boxplot(df2.sqft_lot)
```

Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x15f6c348>



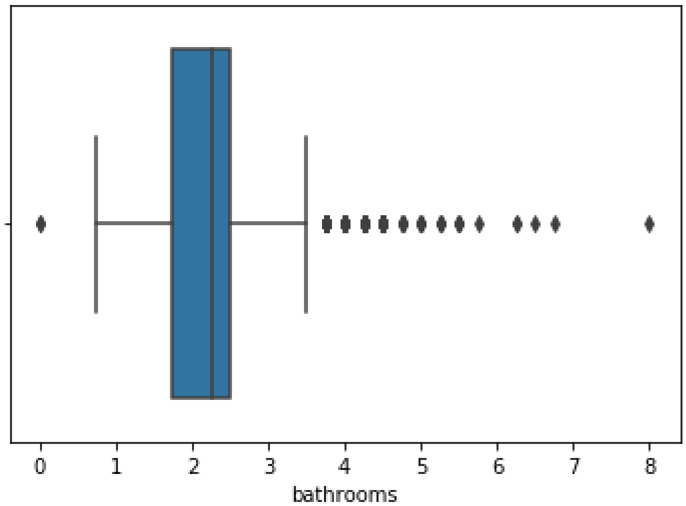
In [16]:
```
sns.boxplot(df2.bedrooms)
```

Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0x16e7c688>



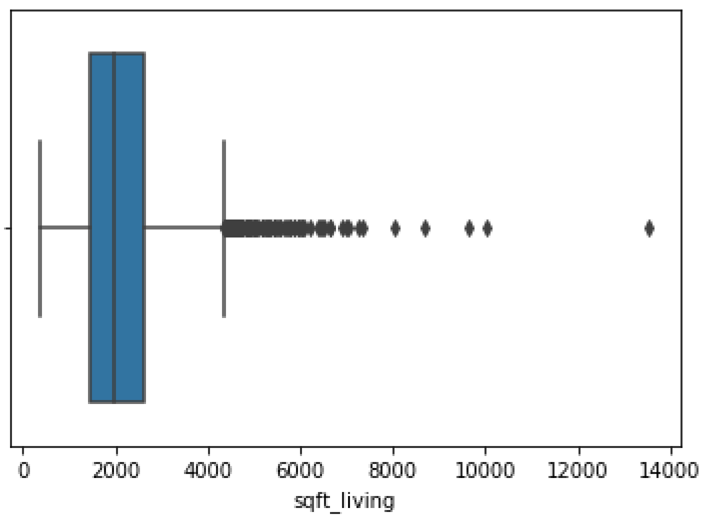
In [17]:
```
sns.boxplot(df2.bathrooms)
```

Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x16ed2848>



In [18]:
```
sns.boxplot(df2.sqft_living)
```

Out[18]: <matplotlib.axes._subplots.AxesSubplot at 0x167b54c8>



From the above it was seen that the sqft lot have so many outliers, so we need to do the cleaning by using filtering the data

Filtering the data and select the sqf lot below 500,000

In [19]:
```
df2.sqft_lot.sort_values(ascending=False).head()
```

Out[19]:
```
1078    1074218
2480     641203
3487     478288
375      435600
879      435600
Name: sqft_lot, dtype: int64
```

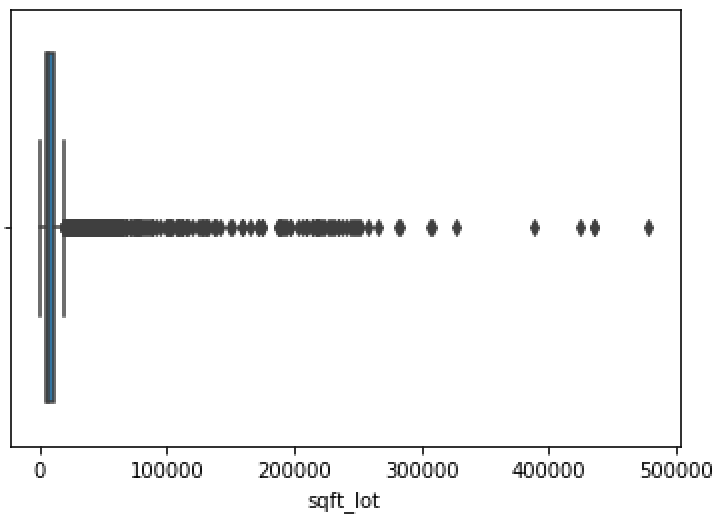
It was seen that there are range so high

In [20]:
```
df2 = df2[df.sqft_lot <= 500000]
df2.shape
```

Out[20]:
```
(4598, 14)
```

In [21]:
```
sns.boxplot(df2.sqft_lot)
```

Out[21]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x16bd3948>
```



Now the outliers has reduced

One hot encoder for category data

In [22]:
```
import category_encoders as ce
# create an object of the OneHotEncoder
OHE = ce.OneHotEncoder(cols=['city'],use_cat_names=True)
# encode the categorical variables
df3 = OHE.fit_transform(df2)
```

In [23]:
```
df3.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4598 entries, 0 to 4599
Data columns (total 57 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   price                     4598 non-null   float64
 1   bedrooms                  4598 non-null   float64
 2   bathrooms                 4598 non-null   float64
 3   sqft_living               4598 non-null   int64  
 4   sqft_lot                  4598 non-null   int64  
 5   floors                    4598 non-null   float64
 6   waterfront                4598 non-null   int64  
 7   view                      4598 non-null   int64  
 8   condition                 4598 non-null   int64  
 9   sqft_above                4598 non-null   int64  
 10  sqft_basement             4598 non-null   int64  
 11  yr_built                  4598 non-null   int64  
 12  yr_renovated              4598 non-null   int64  
 13  city_Shoreline            4598 non-null   int64  
 14  city_Seattle              4598 non-null   int64  
 15  city_Kent                 4598 non-null   int64  
 16  city_Bellevue             4598 non-null   int64  
 17  city_Redmond              4598 non-null   int64  
 18  city_Maple Valley         4598 non-null   int64  
 19  city_North Bend           4598 non-null   int64  
 20  city_Lake Forest Park     4598 non-null   int64  
 21  city_Sammamish            4598 non-null   int64  
 22  city_Auburn               4598 non-null   int64  
 23  city_Des Moines           4598 non-null   int64  
 24  city_Bothell              4598 non-null   int64  
 25  city_Federal Way          4598 non-null   int64  
 26  city_Kirkland             4598 non-null   int64  
 27  city_Issaquah             4598 non-null   int64  
 28  city_Woodinville          4598 non-null   int64  
 29  city_Normandy Park        4598 non-null   int64  
 30  city_Fall City            4598 non-null   int64  
 31  city_Renton               4598 non-null   int64  
 32  city_Carnation            4598 non-null   int64  
 33  city_Snoqualmie           4598 non-null   int64  
 34  city_Duvall               4598 non-null   int64  
 35  city_Burien               4598 non-null   int64  
 36  city_Covington            4598 non-null   int64  
 37  city_Inglewood-Finn Hill  4598 non-null   int64  
 38  city_Kenmore              4598 non-null   int64  
 39  city_Newcastle            4598 non-null   int64  
 40  city_Mercer Island        4598 non-null   int64  
 41  city_Black Diamond        4598 non-null   int64  
 42  city_Ravensdale           4598 non-null   int64  
 43  city_Clyde Hill           4598 non-null   int64  
 44  city_Algona               4598 non-null   int64  
 45  city_Skykomish            4598 non-null   int64  
 46  city_Tukwila              4598 non-null   int64  
 47  city_Vashon               4598 non-null   int64  
 48  city_Yarrow Point         4598 non-null   int64  
 49  city_SeaTac               4598 non-null   int64  
 50  city_Medina               4598 non-null   int64  
 51  city_Enumclaw             4598 non-null   int64  
```

```
 52  city_Snoqualmie Pass      4598 non-null   int64
 53  city_Pacific              4598 non-null   int64
 54  city_Beaux Arts Village   4598 non-null   int64
 55  city_Preston              4598 non-null   int64
 56  city_Milton               4598 non-null   int64
dtypes: float64(4), int64(53)
memory usage: 2.0 MB
```

Select the X and Y

In [24]:
```
df3.iloc[:,:12].describe()
```

Out[24]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	wa
count	4.598000e+03	4598.000000	4598.000000	4598.000000	4598.000000	4598.000000	4598.
mean	5.519002e+05	3.400826	2.160613	2139.127012	14485.896694	1.511962	0.
std	5.639402e+05	0.908505	0.783783	963.328580	30962.062220	0.538357	0.
min	0.000000e+00	0.000000	0.000000	370.000000	638.000000	1.000000	0.
25%	3.226250e+05	3.000000	1.750000	1460.000000	5000.250000	1.000000	0.
50%	4.604435e+05	3.000000	2.250000	1980.000000	7683.000000	1.500000	0.
75%	6.547125e+05	4.000000	2.500000	2620.000000	11000.000000	2.000000	0.
max	2.659000e+07	9.000000	8.000000	13540.000000	478288.000000	3.500000	1.

Need to do scaling the data using data scaler due to different in standard deviation and mean

In [25]:
```
y = df3['price']
```

In [26]:
```
X=df3.iloc[:, 1:57]
```

In [27]:
```
X.shape
```

Out[27]: (4598, 56)

In [28]:
```
y.shape
```

Out[28]: (4598,)

Use the datascaler

In [29]:
```
from sklearn.preprocessing import StandardScaler
scaler_x  = StandardScaler()
X_scaled = scaler_x.fit_transform(X)
```

In [30]:
```
print(X_scaled)
```

```
[[-0.44124149 -0.84294362 -0.82963793 ... -0.01474901 -0.02086051
  -0.02086051]
 [ 1.76041708  0.43305795  1.56855858 ... -0.01474901 -0.02086051
  -0.02086051]
 [-0.44124149 -0.20494284 -0.21711155 ... -0.01474901 -0.02086051
  -0.02086051]
 ...
 [-0.44124149  0.43305795  0.90412319 ... -0.01474901 -0.02086051
  -0.02086051]
 [ 0.6595878  -0.20494284 -0.0510027  ... -0.01474901 -0.02086051
  -0.02086051]
 [-0.44124149  0.43305795 -0.67391088 ... -0.01474901 -0.02086051
  -0.02086051]]
```

In [31]:
```
X_scaled.shape
```

Out[31]:
```
(4598, 56)
```

In [32]:
```
y.shape
```

Out[32]:
```
(4598,)
```

In [33]:
```
y = y.values.reshape(-1,1)
```

In [34]:
```
scaler_y = StandardScaler()

y_scaled = scaler_y.fit_transform(y)
```

Split the dataset

In [35]:
```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_scaled, test_s
ize = .25, random_state = 3)
```

Regression Model

1. Linear Regression as baseline

In [36]:
```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

Out[36]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=Fals
e)
```

In [37]:
```
y_pred_lin = regressor.predict(X_test)
```

In [38]:
```
y_train_lin = regressor.predict(X_train)
```

In [39]:
```
y_predict_lin = scaler_y.inverse_transform(y_pred_lin)
```

In [40]:
```
y_train_linear = scaler_y.inverse_transform(y_train_lin)
```

In [41]:
```
y_test_orig = scaler_y.inverse_transform(y_test)
```

In [42]:
```
y_train_orig = scaler_y.inverse_transform(y_train)
```

In [43]:
```
print("Train Results for Linear regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train_orig, y_train_linear)))
print("R-squared: ", rs(y_train_orig, y_train_linear))
print("Mean Absolute Error: ", mae(y_train_orig, y_train_linear))
```

```
Train Results for Linear regression:
*******************************
Root mean squared error:  543776.6926778575
R-squared:  0.23306523863088413
Mean Absolute Error:  143356.0830687553
```

In [92]:
```
RMSE_lin = sqrt(mse(y_train_orig, y_train_linear))
R_square_lin = rs(y_train_orig, y_train_linear)
MAE_lin = mae(y_train_orig, y_train_linear)
```

In [44]:
```
print("Test Results for Linear regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_test_orig, y_predict_lin)))
print("R-squared: ", rs(y_test_orig, y_predict_lin))
print("Mean Absolute Error: ", mae(y_test_orig, y_predict_lin))
```

```
Test Results for Linear regression:
*******************************
Root mean squared error:  217349.7619004028
R-squared:  0.5892605775944486
Mean Absolute Error:  129180.659933383
```

Ridge Regression

In [45]:
```
tuned_params = {'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 1
00000]}
model_Ridge = GridSearchCV(Ridge(), tuned_params, scoring = 'neg_mean_absolute
_error', cv=10, n_jobs=-1)
model_Ridge.fit(X_train, y_train)
```

Out[45]:
```
GridSearchCV(cv=10, error_score=nan,
             estimator=Ridge(alpha=1.0, copy_X=True, fit_intercept=True,
                             max_iter=None, normalize=False, random_state=Non
e,
                             solver='auto', tol=0.001),
             iid='deprecated', n_jobs=-1,
             param_grid={'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 100
0,
                                   10000, 100000]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='neg_mean_absolute_error', verbose=0)
```

In [46]:
```
model_Ridge.best_estimator_
```

Out[46]:
```
Ridge(alpha=1000, copy_X=True, fit_intercept=True, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001)
```

In [47]:
```
## Predict Train results
y_train_ridge = model_Ridge.predict(X_train)
```

In [48]:
```
## Predict Test results
y_pred_ridge = model_Ridge.predict(X_test)
```

In [49]:
```
y_predict_ridge = scaler_y.inverse_transform(y_pred_ridge)
```

In [50]:
```
y_train_ridg = scaler_y.inverse_transform(y_train_ridge)
```

In [51]:
```
print("Train Results for Ridge regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train_orig, y_train_ridg)))
print("R-squared: ", rs(y_train_orig, y_train_ridg))
print("Mean Absolute Error: ", mae(y_train_orig, y_train_ridg))
```

```
Train Results for Ridge regression:
*******************************
Root mean squared error:  545922.5975553852
R-squared:  0.22700018871270244
Mean Absolute Error:  140999.8893371432
```

In [93]:
```
RMSE_ridge = sqrt(mse(y_train_orig, y_train_ridg))
R_square_ridge = rs(y_train_orig, y_train_ridg)
MAE_ridge = mae(y_train_orig, y_train_ridg)
```

In [52]:
```
print("Test Results for Ridge regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_test_orig, y_predict_ridge)))
print("R-squared: ", rs(y_test_orig, y_predict_ridge))
print("Mean Absolute Error: ", mae(y_test_orig, y_predict_ridge))
```

```
Test Results for Ridge regression:
*******************************
Root mean squared error:  216476.82764744837
R-squared:  0.5925532289854684
Mean Absolute Error:  126089.03185500484
```

Suport Vector

In [53]:
```
## Building the model again with the best hyperparameters
model_SVR = SVR(kernel='linear')
model_SVR.fit(X_train, y_train)
```

```
C:\Users\BIrawan\Anaconda3\lib\site-packages\sklearn\utils\validation.py:760:
DataConversionWarning: A column-vector y was passed when a 1d array was expec
ted. Please change the shape of y to (n_samples, ), for example using ravel
().
  y = column_or_1d(y, warn=True)
```

Out[53]:
```
SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma='scale',
    kernel='linear', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

In [54]:
```
## Predict Train results
y_train_SVR = model_SVR.predict(X_train)
```

In [55]:
```
## Predict Test results
y_pred_SVR = model_SVR.predict(X_test)
```

In [56]:
```
y_predict_SVR= scaler_y.inverse_transform(y_pred_SVR)
```

In [57]:
```
y_trainSVR = scaler_y.inverse_transform(y_train_SVR)
```

In [58]:
```
print("Train Results for Support Vector regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train_orig, y_trainSVR)))
print("R-squared: ", rs(y_train_orig, y_trainSVR))
print("Mean Absolute Error: ", mae(y_train_orig, y_trainSVR))
```

```
Train Results for Support Vector regression:
*******************************
Root mean squared error:  548026.8147132694
R-squared:  0.22102976653869932
Mean Absolute Error:  131408.60806086232
```

In [94]:
```
RMSE_svr = sqrt(mse(y_train_orig, y_trainSVR))
R_square_svr = rs(y_train_orig, y_trainSVR)
MAE_svr = mae(y_train_orig, y_trainSVR)
```

In [59]:
```
print("Test Results for Support Vector regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_test_orig, y_predict_SVR)))
print("R-squared: ", rs(y_test_orig, y_predict_SVR))
print("Mean Absolute Error: ", mae(y_test_orig, y_predict_SVR))
```

```
Test Results for Support Vector regression:
*******************************
Root mean squared error:  214217.2718139181
R-squared:  0.6010145865804548
Mean Absolute Error:  116611.49934353186
```

Random Forest

In [60]:
```
tuned_params = {'n_estimators': [100, 200, 300, 400, 500], 'min_samples_split'
: [2, 5, 10], 'min_samples_leaf': [1, 2, 4]}
model_RF = RandomizedSearchCV(RandomForestRegressor(), tuned_params, n_iter=20
, scoring = 'neg_mean_absolute_error', cv=5, n_jobs=-1)
model_RF.fit(X_train, y_train)
```

```
C:\Users\BIrawan\Anaconda3\lib\site-packages\sklearn\model_selection\_search.
py:739: DataConversionWarning: A column-vector y was passed when a 1d array w
as expected. Please change the shape of y to (n_samples,), for example using
ravel().
  self.best_estimator_.fit(X, y, **fit_params)
```

Out[60]:
```
RandomizedSearchCV(cv=5, error_score=nan,
                   estimator=RandomForestRegressor(bootstrap=True,
                                                   ccp_alpha=0.0,
                                                   criterion='mse',
                                                   max_depth=None,
                                                   max_features='auto',
                                                   max_leaf_nodes=None,
                                                   max_samples=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_jobs=None, oob_score=Fal
se,
                                                   random_state=None, verbose
=0,
                                                   warm_start=False),
                   iid='deprecated', n_iter=20, n_jobs=-1,
                   param_distributions={'min_samples_leaf': [1, 2, 4],
                                        'min_samples_split': [2, 5, 10],
                                        'n_estimators': [100, 200, 300, 400,
                                                         500]},
                   pre_dispatch='2*n_jobs', random_state=None, refit=True,
                   return_train_score=False, scoring='neg_mean_absolute_erro
r',
                   verbose=0)
```

In [61]:
```
model_RF.best_estimator_
```

Out[61]:
```
RandomForestRegressor(bootstrap=True, ccp_alpha=0.0, criterion='mse',
                      max_depth=None, max_features='auto', max_leaf_nodes=Non
e,
                      max_samples=None, min_impurity_decrease=0.0,
                      min_impurity_split=None, min_samples_leaf=1,
                      min_samples_split=10, min_weight_fraction_leaf=0.0,
                      n_estimators=100, n_jobs=None, oob_score=False,
                      random_state=None, verbose=0, warm_start=False)
```

In [62]:
```
## Predict Train results
y_train_RF = model_RF.predict(X_train)
```

In [63]:
```
## Predict Test results
y_pred_RF = model_RF.predict(X_test)
```

In [64]:
```
y_predict_RF= scaler_y.inverse_transform(y_pred_RF)
```

In [65]:
```
y_trainRF = scaler_y.inverse_transform(y_train_RF)
```

In [66]:
```
print("Train Results for Random Forrest regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train_orig, y_trainRF)))
print("R-squared: ", rs(y_train_orig, y_trainRF))
print("Mean Absolute Error: ", mae(y_train_orig, y_trainRF))
```

```
Train Results for Random Forrest regression:
*******************************
Root mean squared error:  359409.76897367387
R-squared:  0.6649597381787381
Mean Absolute Error:  83788.20857374588
```

In [95]:
```
RMSE_rf = sqrt(mse(y_train_orig, y_trainRF))
R_square_rf = rs(y_train_orig, y_trainRF)
MAE_rf = mae(y_train_orig, y_trainRF)
```

In [67]:
```
print("Test Results for Random Forrest regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_test_orig, y_predict_RF)))
print("R-squared: ", rs(y_test_orig, y_predict_RF))
print("Mean Absolute Error: ", mae(y_test_orig, y_predict_RF))
```

```
Test Results for Random Forrest regression:
*******************************
Root mean squared error:  301957.37700281135
R-squared:  0.20724436301074567
Mean Absolute Error:  130711.47366073588
```

Decission Tree

In [68]:
```
tuned_params = {'min_samples_split': [2, 3, 4, 5, 7], 'min_samples_leaf': [1,
2, 3, 4, 6], 'max_depth': [2, 3, 4, 5, 6, 7]}
model_DT = RandomizedSearchCV(DecisionTreeRegressor(), tuned_params, n_iter=20
, scoring = 'neg_mean_absolute_error', cv=10, n_jobs=-1)
model_DT.fit(X_train, y_train)
```

Out[68]:
```
RandomizedSearchCV(cv=10, error_score=nan,
                   estimator=DecisionTreeRegressor(ccp_alpha=0.0,
                                                   criterion='mse',
                                                   max_depth=None,
                                                   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,
                                                   presort='deprecated',
                                                   random_state=None,
                                                   splitter='best'),
                   iid='deprecated', n_iter=20, n_jobs=-1,
                   param_distributions={'max_depth': [2, 3, 4, 5, 6, 7],
                                        'min_samples_leaf': [1, 2, 3, 4, 6],
                                        'min_samples_split': [2, 3, 4, 5,
7]},
                   pre_dispatch='2*n_jobs', random_state=None, refit=True,
                   return_train_score=False, scoring='neg_mean_absolute_erro
r',
                   verbose=0)
```

In [69]:
```
model_DT.best_estimator_
```

Out[69]:
```
DecisionTreeRegressor(ccp_alpha=0.0, criterion='mse', max_depth=7,
                      max_features=None, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=6, min_samples_split=5,
                      min_weight_fraction_leaf=0.0, presort='deprecated',
                      random_state=None, splitter='best')
```

In [70]:
```
## Predict Train results
y_train_DT = model_DT.predict(X_train)
```

In [71]:
```
## Predict Test results
y_pred_DT = model_DT.predict(X_test)
```

In [72]:
```
y_predict_DT= scaler_y.inverse_transform(y_pred_DT)
```

In [73]:
```
y_trainDT = scaler_y.inverse_transform(y_train_DT)
```

In [74]:
```
print("Train Results for Random Forrest regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train_orig, y_trainDT)))
print("R-squared: ", rs(y_train_orig, y_trainDT))
print("Mean Absolute Error: ", mae(y_train_orig, y_trainDT))
```

```
Train Results for Random Forrest regression:
*******************************
Root mean squared error:  503441.21157148166
R-squared:  0.3426225951880927
Mean Absolute Error:  155599.49303935387
```

In [96]:
```
RMSE_dt = sqrt(mse(y_train_orig, y_trainDT))
R_square_dt = rs(y_train_orig, y_trainDT)
MAE_dt = mae(y_train_orig, y_trainDT)
```

In [75]:
```
print("Test Results for Support Vector regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_test_orig, y_predict_DT)))
print("R-squared: ", rs(y_test_orig, y_predict_DT))
print("Mean Absolute Error: ", mae(y_test_orig, y_predict_DT))
```

```
Test Results for Support Vector regression:
*******************************
Root mean squared error:  367929.44616630883
R-squared:  -0.17700184590625723
Mean Absolute Error:  169146.10620063593
```

KNN

In [76]:
```
# creating odd list of K for KNN
neighbors = list(range(1,50,2))
# empty list that will hold cv scores
cv_scores = []

# perform 10-fold cross validation
for k in neighbors:
    knn = KNeighborsRegressor(n_neighbors=k)
    scores = cross_val_score(knn, X_train, y_train, cv=10, scoring='neg_mean_a
bsolute_error')
    cv_scores.append(scores.mean())

# changing to misclassification error
MSE = [1 - x for x in cv_scores]

# determining best k
optimal_k = neighbors[MSE.index(min(MSE))]
print('\nThe optimal number of neighbors is %d.' % optimal_k)
```

```
The optimal number of neighbors is 5.
```

In [77]:
```
model_KNN = KNeighborsRegressor(n_neighbors = optimal_k)
model_KNN.fit(X_train, y_train)
```

Out[77]:
```
KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
                    metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                    weights='uniform')
```

In [78]:
```
## Predict Train results
y_train_KNN = model_KNN.predict(X_train)
```

In [79]:
```
## Predict Test results
y_pred_KNN = model_KNN.predict(X_test)
```

In [80]:
```
y_predict_KNN= scaler_y.inverse_transform(y_pred_KNN)
```

In [81]:
```
y_trainKNN = scaler_y.inverse_transform(y_train_KNN)
```

In [82]:
```
print("Train Results for KNN regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train_orig, y_trainKNN)))
print("R-squared: ", rs(y_train_orig, y_trainKNN))
print("Mean Absolute Error: ", mae(y_train_orig, y_trainKNN))
```

```
Train Results for KNN regression:
*******************************
Root mean squared error:  475448.93609199184
R-squared:  0.41369310089289824
Mean Absolute Error:  120781.00356891201
```

In [97]:
```
RMSE_knn = sqrt(mse(y_train_orig, y_trainKNN))
R_square_knn = rs(y_train_orig, y_trainKNN)
MAE_knn = mae(y_train_orig, y_trainKNN)
```

In [83]:
```
print("Test Results for KNN regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_test_orig, y_predict_KNN)))
print("R-squared: ", rs(y_test_orig, y_predict_KNN))
print("Mean Absolute Error: ", mae(y_test_orig, y_predict_KNN))
```

```
Test Results for KNN regression:
*******************************
Root mean squared error:  293743.8430326016
R-squared:  0.24978525557596376
Mean Absolute Error:  133455.36335048106
```

Lasso Regression

In [84]:
```
tuned_params = {'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000, 10000, 1
00000]}
model_LS = GridSearchCV(Lasso(), tuned_params, scoring = 'neg_mean_absolute_er
ror', cv=20, n_jobs=-1)
model_LS.fit(X_train, y_train)
```

Out[84]:
```
GridSearchCV(cv=20, error_score=nan,
             estimator=Lasso(alpha=1.0, 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=Fals
e),
             iid='deprecated', n_jobs=-1,
             param_grid={'alpha': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 100
0,
                                   10000, 100000]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='neg_mean_absolute_error', verbose=0)
```

In [85]:
```
model_LS.best_estimator_
```

Out[85]:
```
Lasso(alpha=0.01, 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)
```

In [86]:
```
## Predict Train results
y_train_LS = model_LS.predict(X_train)
```

In [87]:
```
## Predict Test results
y_pred_LS = model_LS.predict(X_test)
```

In [88]:
```
y_predict_LS= scaler_y.inverse_transform(y_pred_LS)
```

In [89]:
```
y_trainLS = scaler_y.inverse_transform(y_train_LS)
```

In [98]:
```
print("Train Results for Lasso regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_train_orig, y_trainLS)))
print("R-squared: ", rs(y_train_orig, y_trainLS))
print("Mean Absolute Error: ", mae(y_train_orig, y_trainLS))
```

```
Train Results for Lasso regression:
*******************************
Root mean squared error:  544980.8625046085
R-squared:  0.22966479040177779
Mean Absolute Error:  142639.46170476923
```

In [100]:
```
RMSE_ls = sqrt(mse(y_train_orig, y_trainLS))
R_square_ls = rs(y_train_orig, y_trainLS)
MAE_ls = mae(y_train_orig, y_trainLS)
```

In [99]:
```
print("Test Results for Lasso regression:")
print("*******************************")
print("Root mean squared error: ", sqrt(mse(y_test_orig, y_predict_LS)))
print("R-squared: ", rs(y_test_orig, y_predict_LS))
print("Mean Absolute Error: ", mae(y_test_orig, y_predict_LS))
```

```
Test Results for Lasso regression:
*******************************
Root mean squared error:  217825.14016566932
R-squared:  0.5874619087493652
Mean Absolute Error:  127931.71645607855
```

CREATE TABLE OF COMPARISSON FOR ALL MODELS

In [108]:
```
data_comp = {'Model Name':  ['Linear Regression', 'Ridge Regression','Support Vector','Random Forrest','KNN','Lasso'],
        'RMSE': [RMSE_lin, RMSE_ridge,RMSE_svr,RMSE_rf, RMSE_knn,RMSE_ls],
        'R-Squared': [R_square_lin,R_square_ridge,R_square_svr,R_square_rf,R_square_knn,R_square_ls],
        'MAE': [MAE_lin, MAE_ridge, MAE_svr, MAE_rf, MAE_knn, MAE_ls]
        }

df_comp  = pd.DataFrame (data_comp, columns = ['Model Name','RMSE','R-Squared','MAE'])

print (df_comp)
```

```
          Model Name           RMSE  R-Squared            MAE
0  Linear Regression  543776.692678   0.233065  143356.083069
1   Ridge Regression  545922.597555   0.227000  140999.889337
2     Support Vector  548026.814713   0.221030  131408.608061
3     Random Forrest  359409.768974   0.664960   83788.208574
4                KNN  475448.936092   0.413693  120781.003569
5              Lasso  544980.862505   0.229665  142639.461705
```

In [110]:
```
df_comp.sort_values(by='R-Squared', ascending=False )
```

Out[110]:

	Model Name	RMSE	R-Squared	MAE
3	Random Forrest	359409.768974	0.664960	83788.208574
4	KNN	475448.936092	0.413693	120781.003569
0	Linear Regression	543776.692678	0.233065	143356.083069
5	Lasso	544980.862505	0.229665	142639.461705
1	Ridge Regression	545922.597555	0.227000	140999.889337
2	Support Vector	548026.814713	0.221030	131408.608061

CONCLUSSION

From the above table it was shown that the Random Forrest is the best regression model on predicting the housing price in Boston dataset, due to :

1. highest score in R-squared
2. Lowest score in RMSE and MAE

Further improvement :

1. Using L1 and L2

Furthermore, the pipeline model can be made using random forest