Final Project Submission

Please fill out:

- Student name: Maria Kuzmin
- Student pace: self paced
- Scheduled project review date/time:
- Instructor name: Morgan Jones
- Blog post URL: tbd

Don't Roam Buy a Home

The real estate startup “Don’t Roam Buy a Home” (DRBH for short) has contacted us as they are trying to create an app targeted at those who do not feel like they can compete in the current brutal real estate market. DRBH’s application is meant for the normal average person that does not have much knowledge about the real estate market but is trying to understand what would be the best investment given their resources and needs.

In our business case, DRBH has hired us in order to assist with analyzing raw real estate data and breaking down the trends of the housing market in King County. Their end product centers around having users enter the desired number of bedrooms, bathrooms, the overall residence’s square feet or lot size as well as neighborhood, budget in terms of savings and possible monthly mortgage. With this information the app will provide the user key information to help make the most educated choice and have the most profitable investment with the available funds.

Don't Roam Buy a Home will help users answer some questions like:

- What is the best neighborhood for me to look into buying a house, given my budget?
- What are the most important factors to look at, when evaluating a house and my needs in a house?
- Can I afford an extra bedroom/bathroom or should I save up and add one later on?
- Would it be better to buy a new property or a fixer upper and use the extra money to renovate?

These and more are the information that we will be able to provide to the users of the app, starting from our analysis of the King's County Housing Market.

How are we going to get there

Here is a roadmap of the steps that we are going to take:

- Data Preparation:
    - Looking at the Data
    - Cleaning the Data
    - Removing Outliers
    - Preparing data: One Hot Encoding

Imports

Let's import all the libraries that we are going to need for our analysis.

In [1]:
```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns
from scipy import stats as stats
import statsmodels.api as sm
from statsmodels.formula.api import ols
import math

from sklearn.preprocessing import OneHotEncoder, StandardScaler, MinMaxScal
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import LabelBinarizer
from sklearn.preprocessing import PolynomialFeatures
import sklearn.metrics as metrics
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_validate

import datetime
import calendar
from datetime import datetime
from colorama import Fore
from colorama import Style

%matplotlib inline
plt.style.use('seaborn-notebook')
```

The Data

This project uses the King County House Sales dataset, which can be found in `kc_house_data.csv`. Here is a brief description of the meaning of each column:

Column Names and Descriptions for King County Data Set

- `id` - Unique identifier for a house
- `date` - Date house was sold
- `price` - Sale price (prediction target)
- `bedrooms` - Number of bedrooms
- `bathrooms` - Number of bathrooms
- `sqft_living` - Square footage of living space in the home
- `sqft_lot` - Square footage of the lot
- `floors` - Number of floors (levels) in house
- `waterfront` - Whether the house is on a waterfront
  - Includes Duwamish, Elliott Bay, Puget Sound, Lake Union, Ship Canal, Lake Washington, Lake Sammamish, other lake, and river/slough waterfronts
- `view` - Quality of view from house
  - Includes views of Mt. Rainier, Olympics, Cascades, Territorial, Seattle Skyline, Puget Sound, Lake Washington, Lake Sammamish, small lake / river / creek, and other

- `condition` - How good the overall condition of the house is. Related to maintenance of house.
  - See the King County Assessor Website (https://info.kingcounty.gov/assessor/esales/Glossary.aspx?type=r) for further explanation of each condition code
- `grade` - Overall grade of the house. Related to the construction and design of the house.
  - See the King County Assessor Website (https://info.kingcounty.gov/assessor/esales/Glossary.aspx?type=r) for further explanation of each building grade code
- `sqft_above` - Square footage of house apart from basement
- `sqft_basement` - Square footage of the basement
- `yr_built` - Year when house was built
- `yr_renovated` - Year when house was renovated
- `zipcode` - ZIP Code used by the United States Postal Service
- `lat` - Latitude coordinate
- `long` - Longitude coordinate
- `sqft_living15` - The square footage of interior housing living space for the nearest 15 neighbors
- `sqft_lot15` - The square footage of the land lots of the nearest 15 neighbors

Let us proceed by loading the data and taking a look.

In [2]:
```
df=pd.read_csv('Data/kc_house_data.csv')
```

In [3]:
```
df.head()
```

Out[3]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront
0	7129300520	10/13/2014	221900.0	3	1.00	1180	5650	1.0	NaN
1	6414100192	12/9/2014	538000.0	3	2.25	2570	7242	2.0	NO
2	5631500400	2/25/2015	180000.0	2	1.00	770	10000	1.0	NO
3	2487200875	12/9/2014	604000.0	4	3.00	1960	5000	1.0	NO
4	1954400510	2/18/2015	510000.0	3	2.00	1680	8080	1.0	NO

5 rows × 21 columns

In [4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   id             21597 non-null  int64  
 1   date           21597 non-null  object 
 2   price          21597 non-null  float64
 3   bedrooms       21597 non-null  int64  
 4   bathrooms      21597 non-null  float64
 5   sqft_living    21597 non-null  int64  
 6   sqft_lot       21597 non-null  int64  
 7   floors         21597 non-null  float64
 8   waterfront     19221 non-null  object 
 9   view           21534 non-null  object 
 10  condition      21597 non-null  object 
 11  grade          21597 non-null  object 
 12  sqft_above     21597 non-null  int64  
 13  sqft_basement  21597 non-null  object 
 14  yr_built       21597 non-null  int64  
 15  yr_renovated   17755 non-null  float64
 16  zipcode        21597 non-null  int64  
 17  lat            21597 non-null  float64
 18  long           21597 non-null  float64
 19  sqft_living15  21597 non-null  int64  
 20  sqft_lot15     21597 non-null  int64  
dtypes: float64(6), int64(9), object(6)
memory usage: 3.5+ MB
```

Exploring the columns with null values:

In [5]:
```
df['waterfront'].value_counts()
```

Out[5]:
```
NO     19075
YES      146
Name: waterfront, dtype: int64
```

In [6]:
```
df['view'].value_counts()
```

Out[6]:
```
NONE         19422
AVERAGE        957
GOOD           508
FAIR           330
EXCELLENT      317
Name: view, dtype: int64
```

In [7]:
```
df['yr_renovated'].value_counts()
```

Out[7]:
```
0.0       17011
2014.0       73
2003.0       31
2013.0       31
2007.0       30
          ...
1946.0        1
1959.0        1
1971.0        1
1951.0        1
1954.0        1
Name: yr_renovated, Length: 70, dtype: int64
```

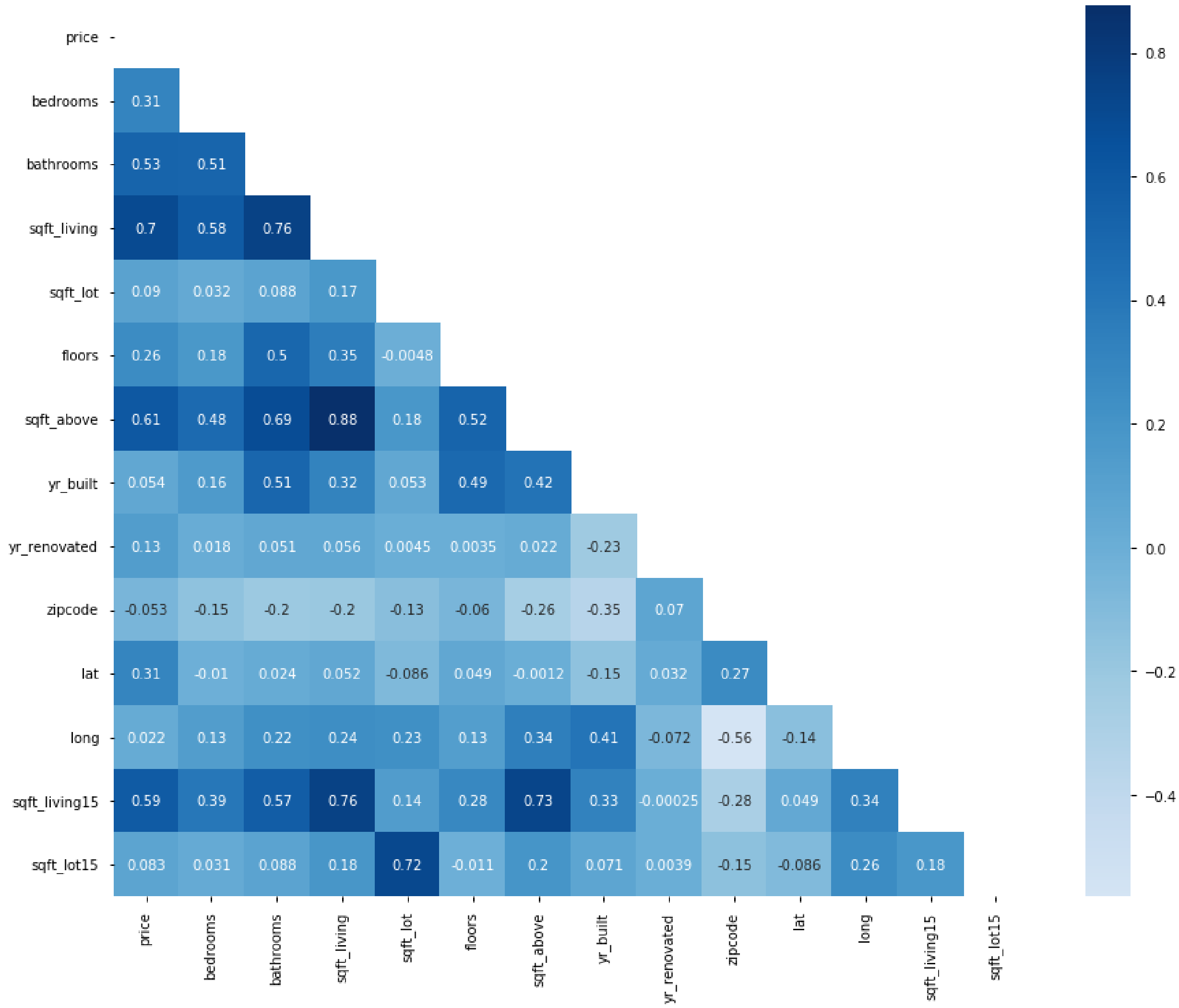
ID is a unique identifier therefore not really relevant to our analysis. Also waterfront and view have very few entries that are not NaN.
Year renovated also has a few entries but I think the information it carries can be very interesting so we are going to keep it for now.

In [8]:
```
df.drop(['id'], axis=1, inplace=True)
```

Heatmap

To get a first sense of what type of correlations the variables have with each other we can generate a heatmap:

In [9]:
```
fig, ax = plt.subplots(figsize=(15, 12))
sns.heatmap(df.corr(), center=0, ax=ax, annot=True, mask=np.triu(np.ones_li
```

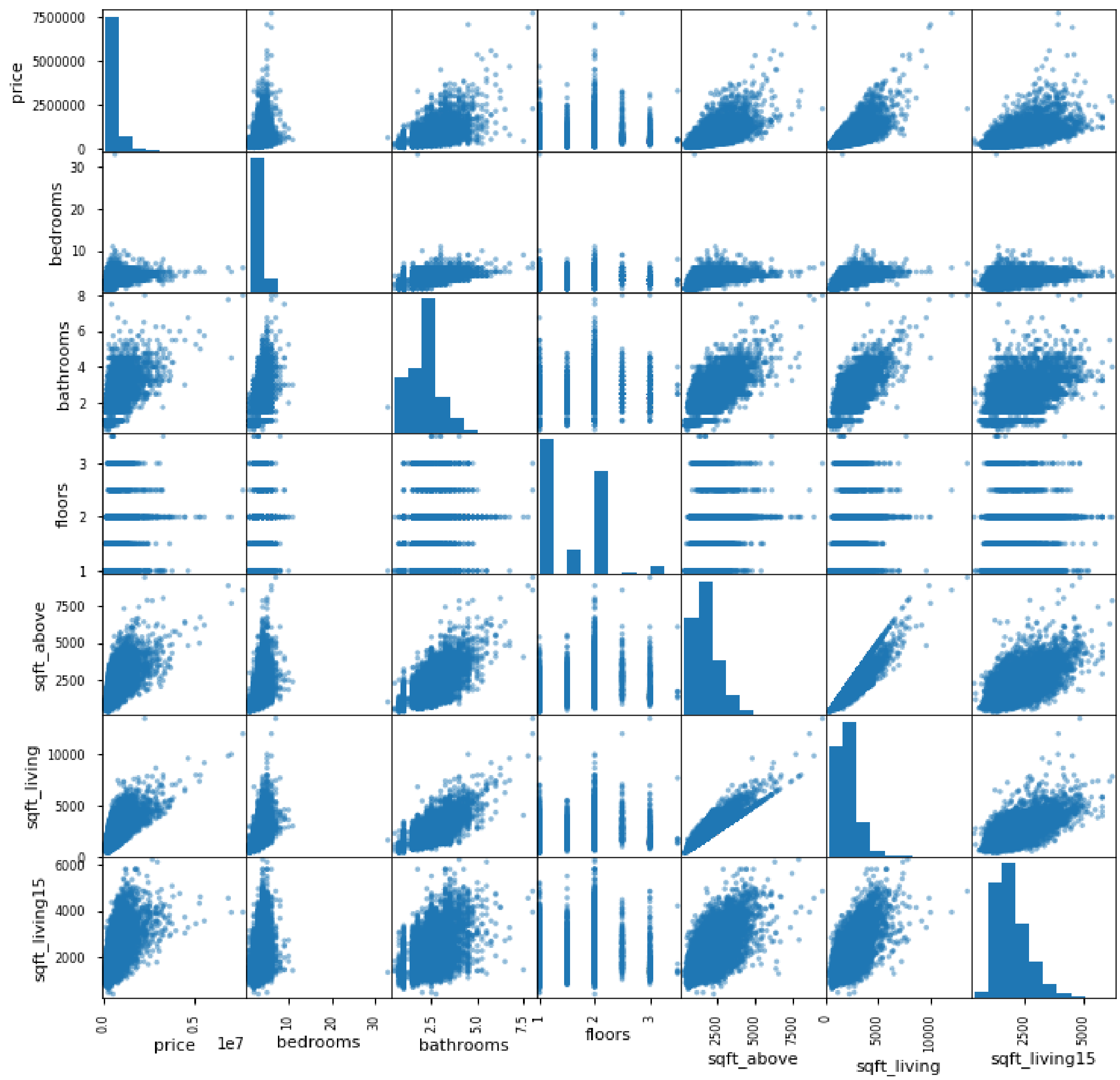


We can see that some of the strongest correlations with price (over 0.5) are: number of bathrooms, sqft_living, sqft_above,sqft_living15 which tells us something about the area where the house is.

Scatterplot

We can also generate a scatterplot that shows us the different variables plotted vs one another to reveal underlying correlations.

In [10]:
```
main=pd.DataFrame(df, columns = ['price',"bedrooms","bathrooms", "floors","
pd.plotting.scatter_matrix(main,figsize  = [12, 12]);
```



From this very first visualization we start to see some linear correlations and we can spot some outliers. Number of bedrooms seems to have an outlier, and there seems to be some correlation of price with price of the houses, in particular with number of bathrooms, sqft_above, sqft_living and

sqft_living 15.
We can run simple linear regression models for bathrooms and squarefeet living, to start to get a sense of the correlation of price with these two variables.

In [11]:
```
# build the formula
f='bathrooms~price'
# create a fitted model in one line
model = ols(formula=f, data=df).fit()
```

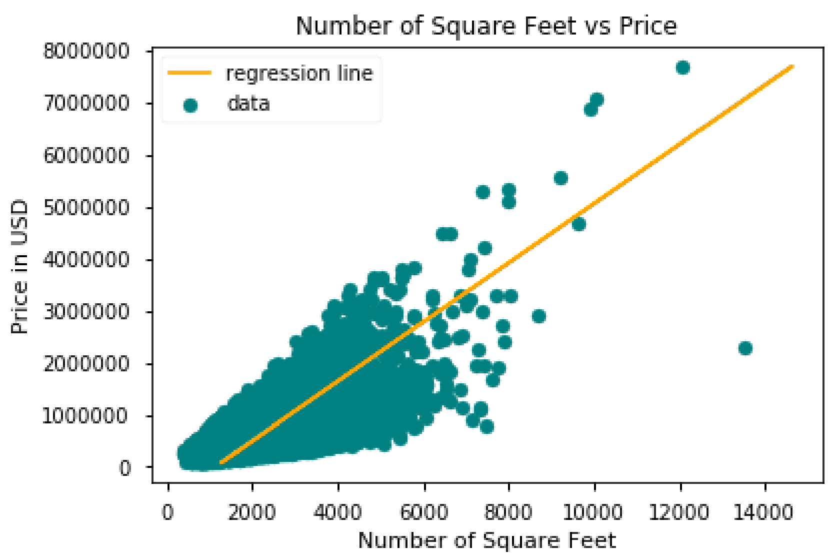
In [12]:
```
regression_line=np.array(model.predict())
plt.scatter(df['bathrooms'], (df['price']), color='green')
plt.plot(regression_line,df['price'] ,color='blue')
plt.legend(labels=('regression line','data'))
plt.title("Number of Bathrooms vs Price")
plt.xlabel("Number of bathrooms")
plt.ylabel("Price in USD")
plt.show;
```

```
/Users/vi/opt/anaconda3/envs/learn-env/lib/python3.6/site-packages/matplo
tlib/cbook/__init__.py:1377: FutureWarning: Support for multi-dimensional
indexing (e.g. `obj[:, None]`) is deprecated and will be removed in a fut
ure version.  Convert to a numpy array before indexing instead.
  x[:, None]
/Users/vi/opt/anaconda3/envs/learn-env/lib/python3.6/site-packages/matplo
tlib/axes/_base.py:239: FutureWarning: Support for multi-dimensional inde
xing (e.g. `obj[:, None]`) is deprecated and will be removed in a future
version.  Convert to a numpy array before indexing instead.
  y = y[:, np.newaxis]
```

In [13]:
```
# build the formula
f='sqft_living~price'
# create a fitted model in one line
model = ols(formula=f, data=df).fit()
```

In [14]:
```
regression_line=np.array(model.predict())
plt.scatter(df['sqft_living'], df['price'], color='teal')
plt.plot(regression_line,df['price'] ,color='orange')
plt.legend(labels=('regression line','data'))
plt.title("Number of Square Feet vs Price")
plt.xlabel("Number of Square Feet")
plt.ylabel("Price in USD")
plt.show;
```

```
/Users/vi/opt/anaconda3/envs/learn-env/lib/python3.6/site-packages/matplo
tlib/cbook/__init__.py:1377: FutureWarning: Support for multi-dimensional
indexing (e.g. `obj[:, None]`) is deprecated and will be removed in a fut
ure version.  Convert to a numpy array before indexing instead.
  x[:, None]
/Users/vi/opt/anaconda3/envs/learn-env/lib/python3.6/site-packages/matplo
tlib/axes/_base.py:239: FutureWarning: Support for multi-dimensional inde
xing (e.g. `obj[:, None]`) is deprecated and will be removed in a future
version.  Convert to a numpy array before indexing instead.
  y = y[:, np.newaxis]
```



These visualizations confirm the positive correlation we suspected between these two variables and price.

Correlation matrix

We are going to need this more in detail later but a correlation matrix can also give us a better sense of what are the correlations between the variables.

In [15]:
```
df.corr()
```

Out[15]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	sqft_above	yr_bu
price	1.000000	0.308787	0.525906	0.701917	0.089876	0.256804	0.605368	0.0539
bedrooms	0.308787	1.000000	0.514508	0.578212	0.032471	0.177944	0.479386	0.1556
bathrooms	0.525906	0.514508	1.000000	0.755758	0.088373	0.502582	0.686668	0.5071
sqft_living	0.701917	0.578212	0.755758	1.000000	0.173453	0.353953	0.876448	0.3181
sqft_lot	0.089876	0.032471	0.088373	0.173453	1.000000	-0.004814	0.184139	0.0529
floors	0.256804	0.177944	0.502582	0.353953	-0.004814	1.000000	0.523989	0.4891
sqft_above	0.605368	0.479386	0.686668	0.876448	0.184139	0.523989	1.000000	0.4240
yr_built	0.053953	0.155670	0.507173	0.318152	0.052946	0.489193	0.424037	1.0000
yr_renovated	0.129599	0.018495	0.051050	0.055660	0.004513	0.003535	0.022137	-0.2252
zipcode	-0.053402	-0.154092	-0.204786	-0.199802	-0.129586	-0.059541	-0.261570	-0.3472
lat	0.306692	-0.009951	0.024280	0.052155	-0.085514	0.049239	-0.001199	-0.1483
long	0.022036	0.132054	0.224903	0.241214	0.230227	0.125943	0.344842	0.4099
sqft_living15	0.585241	0.393406	0.569884	0.756402	0.144763	0.280102	0.731767	0.3263
sqft_lot15	0.082845	0.030690	0.088303	0.184342	0.718204	-0.010722	0.195077	0.0707

There are some interesting correlations here. As we expected especially for number of bathrooms, living sqfootage and sqft above.
To better study and understand these correlations we are going to do some data preparation, using one hot encoding and then running a linear regression model and then we will proceed to try to improve our model and then studying the correlation coefficients.

Cleaning the data

Changing formats

Some of the data is in the format 'object' and therefore cannot be included in the calculations.
In one case, basement sqft it is probably a matter of just changing the format, while other entries like condition and grade are string values that need to be translated into something numerical that our model can work with.
Consequently we are going to use the method "One Hot Encoding" to change the format of the categorical variables to be able to include them in our calculation.

In [16]:
```
df['sqft_basement'].value_counts()
```

Out[16]:
```
0.0       12826
?           454
600.0       217
500.0       209
700.0       208
          ...  
1281.0        1
1275.0        1
225.0         1
2050.0        1
1548.0        1
Name: sqft_basement, Length: 304, dtype: int64
```

Clearly most of the values are numerical, and can be easily turned into a numerical format, while we are going to change the value '?' into zero, as we are not aware of what is the square footage of the basement, it is safer to do this compared to assuming some other value for it.

In [17]:
```
df['sqft_basement'] = df['sqft_basement'].replace('?','0.0')
```

In [18]:
```
df['sqft_basement'] = df['sqft_basement'].astype(float)
```

In [19]:
```
df['sqft_basement'].value_counts()
```

Out[19]:
```
0.0       13280
600.0       217
500.0       209
700.0       208
800.0       201
          ...  
915.0         1
295.0         1
1281.0        1
2130.0        1
906.0         1
Name: sqft_basement, Length: 303, dtype: int64
```

In [20]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 20 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   date           21597 non-null  object 
 1   price          21597 non-null  float64
 2   bedrooms       21597 non-null  int64  
 3   bathrooms      21597 non-null  float64
 4   sqft_living    21597 non-null  int64  
 5   sqft_lot       21597 non-null  int64  
 6   floors         21597 non-null  float64
 7   waterfront     19221 non-null  object 
 8   view           21534 non-null  object 
 9   condition      21597 non-null  object 
 10  grade          21597 non-null  object 
 11  sqft_above     21597 non-null  int64  
 12  sqft_basement  21597 non-null  float64
 13  yr_built       21597 non-null  int64  
 14  yr_renovated   17755 non-null  float64
 15  zipcode        21597 non-null  int64  
 16  lat            21597 non-null  float64
 17  long           21597 non-null  float64
 18  sqft_living15  21597 non-null  int64  
 19  sqft_lot15     21597 non-null  int64  
dtypes: float64(7), int64(8), object(5)
memory usage: 3.3+ MB
```

Some other columns are not numerical but can be converted. For example "grade" has actually a number representing it but because it is followed also by a string its format is object. Let us take care of that.

In [21]:
```
df['grade'].value_counts()
```

Out[21]:
```
7 Average        8974
8 Good           6065
9 Better         2615
6 Low Average    2038
10 Very Good     1134
11 Excellent      399
5 Fair            242
12 Luxury          89
4 Low              27
13 Mansion         13
3 Poor              1
Name: grade, dtype: int64
```

In [22]:
```
# Creating a dictionary using which we will remap the values
dict = {'7 Average' : 7,'8 Good': 8,'9 Better':9,'6 Low Average':6,'10 Very
        '12 Luxury':12,'4 Low':4,'13 Mansion':13 ,'3 Poor':3}

# Remap the values of the dataframe
df=df.replace({"grade": dict})
```

In [23]:
```
df['grade'] = df['grade'].astype(float)
```

Now let us explore the column 'condition'. Maybe we can also replace that easily with numerical values.

In [24]:
```
df['condition'].value_counts()
```

Out[24]:
```
Average      14020
Good          5677
Very Good     1701
Fair           170
Poor            29
Name: condition, dtype: int64
```

From the reference we have (https://info.kingcounty.gov/assessor/esales/Glossary.aspx?type=r (https://info.kingcounty.gov/assessor/esales/Glossary.aspx?type=r)) it is clear that the values for condition are to be interepreted as follows:

1 = Poor
2 = Fair
3 = Average
4 = Good
5= Very Good

We will then translate those values in the same way

In [25]:
```
dict = {'Poor':1, 'Fair':2, 'Average': 3,'Good': 4,'Very Good':5}

# Remap the values of the dataframe
df=df.replace({"condition": dict})
```

In [26]:
```
df['condition'].value_counts()
```

Out[26]:
```
3    14020
4     5677
5     1701
2      170
1       29
Name: condition, dtype: int64
```

In [27]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 20 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   date           21597 non-null  object 
 1   price          21597 non-null  float64
 2   bedrooms       21597 non-null  int64  
 3   bathrooms      21597 non-null  float64
 4   sqft_living    21597 non-null  int64  
 5   sqft_lot       21597 non-null  int64  
 6   floors         21597 non-null  float64
 7   waterfront     19221 non-null  object 
 8   view           21534 non-null  object 
 9   condition      21597 non-null  int64  
 10  grade          21597 non-null  float64
 11  sqft_above     21597 non-null  int64  
 12  sqft_basement  21597 non-null  float64
 13  yr_built       21597 non-null  int64  
 14  yr_renovated   17755 non-null  float64
 15  zipcode        21597 non-null  int64  
 16  lat            21597 non-null  float64
 17  long           21597 non-null  float64
 18  sqft_living15  21597 non-null  int64  
 19  sqft_lot15     21597 non-null  int64  
dtypes: float64(8), int64(9), object(3)
memory usage: 3.3+ MB
```

Handling missing values

In [28]:
```
df.isnull().sum()
```

Out[28]:
```
date                0
price               0
bedrooms            0
bathrooms           0
sqft_living         0
sqft_lot            0
floors              0
waterfront       2376
view               63
condition           0
grade               0
sqft_above          0
sqft_basement       0
yr_built            0
yr_renovated     3842
zipcode             0
lat                 0
long                0
sqft_living15       0
sqft_lot15          0
dtype: int64
```

As we can see the columns that have missing values are 'waterfront', 'view' and 'year renovated'.
There are clear reasons for some values to be missing as not all of the houses have a waterfront or information about the view, and not all the houses were renovated.
We will use that information later and separately, but for now we are not going to include these variables in our model so there is not need to worry about them.

Preliminary Regression Model

Running a linear regression here before one hot encoding. This is very early on and super preliminary but we can do this just to get a first sense.
Because of the large amount of null values in waterfront and view I have to drop them, or the model won't work.

In [29]:
```
df.drop(['view', 'waterfront'], axis=1, inplace=True)
```

In [30]:
```
y=df['price']
X=df.drop(['price', 'date', 'yr_renovated'], axis=1)

linreg = LinearRegression()
linreg.fit(X, y)
```

Out[30]:
```
LinearRegression()
```

In [31]:
```
R2=metrics.r2_score(y,linreg.predict(X))
print(f"{Fore.RED} The R squared value for this Preliminary Regression mode
```

```
 The R squared value for this Preliminary Regression model is  0.66087962
77809059
```

As a very preliminary regression model we got an R squared of 0.66. That means that our first draft model explain about 66% of the data. It is not terrible but there is definitely room for improvement.

Running the model in statsmodel to be able to see also the coefficients.

In [32]:
```
X = sm.add_constant(X)
model = sm.OLS(y,X).fit()
model.summary()
```

Out[32]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.661
Model:	OLS	**Adj. R-squared:**	0.661
Method:	Least Squares	**F-statistic:**	2804.
Date:	Fri, 11 Nov 2022	**Prob (F-statistic):**	0.00
Time:	18:58:04	**Log-Likelihood:**	-2.9571e+05
No. Observations:	21597	**AIC:**	5.915e+05
Df Residuals:	21581	**BIC:**	5.916e+05
Df Model:	15		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	-3.721e+06	3.09e+06	-1.203	0.229	-9.78e+06	2.34e+06
bedrooms	-4.664e+04	2008.144	-23.224	0.000	-5.06e+04	-4.27e+04
bathrooms	4.741e+04	3443.208	13.769	0.000	4.07e+04	5.42e+04
sqft_living	125.8798	19.208	6.553	0.000	88.230	163.529
sqft_lot	0.1511	0.051	2.966	0.003	0.051	0.251
floors	1.265e+04	3823.290	3.308	0.001	5155.376	2.01e+04
condition	2.589e+04	2464.791	10.503	0.000	2.11e+04	3.07e+04
grade	1.031e+05	2287.988	45.067	0.000	9.86e+04	1.08e+05
sqft_above	61.2191	19.197	3.189	0.001	23.592	98.846
sqft_basement	54.8663	19.039	2.882	0.004	17.549	92.184
yr_built	-3066.2645	72.865	-42.082	0.000	-3209.085	-2923.444
zipcode	-484.7039	34.925	-13.879	0.000	-553.159	-416.249
lat	5.503e+05	1.13e+04	48.493	0.000	5.28e+05	5.73e+05
long	-2.485e+05	1.4e+04	-17.800	0.000	-2.76e+05	-2.21e+05
sqft_living15	38.0698	3.619	10.521	0.000	30.977	45.162
sqft_lot15	-0.3213	0.078	-4.120	0.000	-0.474	-0.168

Omnibus:	19469.874	**Durbin-Watson:**	1.993
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	1929787.031
Skew:	3.950	**Prob(JB):**	0.00
Kurtosis:	48.630	**Cond. No.**	2.14e+08

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 2.14e+08. This might indicate that there are
strong multicollinearity or other numerical problems.

Let us look at which ones are the variables which have the highest coefficients.

In [33]:
```
coefficients=model.params
sort_coef=coefficients.sort_values(ascending=False)
sort_coef[0:10]
```

Out[33]:
```
lat              550309.302063
grade            103112.329822
bathrooms         47409.251943
condition         25888.462328
floors            12649.307510
sqft_living         125.879803
sqft_above           61.219132
sqft_basement        54.866329
sqft_living15        38.069783
sqft_lot              0.151083
dtype: float64
```

Latitude gives us a hint that there is a strong influence of the location of the house.
But we will understand this better once we do some scaling and more in depth analysis. The other factors that are most influential in determining the price of the house are the condition and grade, together with number of bathrooms, number of floors and squarefeet of living space. What the coefficients are telling us is given a one-unit change in the feature variable when the other features are unchanged, how much is the dependent variable changed.

Outliers

We should to inspect the bedroom variable since we spotted a possible outlier before.

In [34]:
```
df['bedrooms'].value_counts()
```

Out[34]:
```
3     9824
4     6882
2     2760
5     1601
6      272
1      196
7       38
8       13
9        6
10       3
11       1
33       1
Name: bedrooms, dtype: int64
```

Yes, 33 is definitely an outlier so we will remove that.

In [35]:
```
df.drop(df.loc[df['bedrooms']==33].index, inplace=True)
```

Let us also drop the outliers for the sale price. Instead of dropping the outliers for every single feature, which would be time consuming and lead us to discard a big chunk of the data, this seems like a reasonable solution.

In [36]:
```
"""Removing outliers in the main database """
Q95=np.percentile(df['price'],95)
Q5=np.percentile(df['price'],5)
df.drop(df.index[df['price']>Q95], inplace=True)
df.drop(df.index[df['price']<Q5], inplace=True)
```

One Hot Encoding

One Hot Encoding is a method that allows us to transform categorical variables into numerical ones to be able to better include them in our model.
Some variables that can be considered categorical are the number of bedrooms and bathrooms and floors.
So let us start from there.

In [37]:
```
df
```

Out[37]:

	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	condition	grade	s
0	10/13/2014	221900.0	3	1.00	1180	5650	1.0	3	7.0	
1	12/9/2014	538000.0	3	2.25	2570	7242	2.0	3	7.0	
3	12/9/2014	604000.0	4	3.00	1960	5000	1.0	5	7.0	
4	2/18/2015	510000.0	3	2.00	1680	8080	1.0	3	8.0	
6	6/27/2014	257500.0	3	2.25	1715	6819	2.0	3	7.0	
...	...	...	...	...	...	...	...	...	...	
21592	5/21/2014	360000.0	3	2.50	1530	1131	3.0	3	8.0	
21593	2/23/2015	400000.0	4	2.50	2310	5813	2.0	3	8.0	
21594	6/23/2014	402101.0	2	0.75	1020	1350	2.0	3	7.0	
21595	1/16/2015	400000.0	3	2.50	1600	2388	2.0	3	8.0	
21596	10/15/2014	325000.0	2	0.75	1020	1076	2.0	3	7.0	

19512 rows × 18 columns

The numbers of bathrooms are not integers, but categorized in different ways whether there is a toilet or a sink or a shower etc. This with one hot encoding would lead us to have A LOT of columns and also correlation will be more confusing to interpret. So to make this a little simpler we are going to round them up.

In [38]:
```
#Rounding up number of bathrooms
df['bathrooms']=df['bathrooms'].round(0)
df['bathrooms'].value_counts()
```

Out[38]:
```
2.0    13222
1.0     3244
3.0     2211
4.0      810
5.0       17
6.0        3
0.0        3
7.0        1
8.0        1
Name: bathrooms, dtype: int64
```

One Hot Encoding number of bathrooms, bedrooms and floors.

In [39]:
```
ohe = OneHotEncoder(handle_unknown='ignore', sparse=False)

# Categorical columns
cat_columns = ['bedrooms', 'bathrooms', 'floors']

# Fit encoder on training set
ohe.fit(df[cat_columns])

# Get new column names
new_cat_columns = ohe.get_feature_names(input_features=cat_columns)

# Transform training set
df_cat = pd.DataFrame(ohe.fit_transform(df[cat_columns]),
                      columns=new_cat_columns, index=df.index)

# Replace training columns with transformed versions
ohe_df = pd.concat([df.drop(cat_columns, axis=1), df_cat], axis=1)
ohe_df
```

Out[39]:

	date	price	sqft_living	sqft_lot	condition	grade	sqft_above	sqft_basement	yr_b
0	10/13/2014	221900.0	1180	5650	3	7.0	1180	0.0	1(
1	12/9/2014	538000.0	2570	7242	3	7.0	2170	400.0	1(
3	12/9/2014	604000.0	1960	5000	5	7.0	1050	910.0	1(
4	2/18/2015	510000.0	1680	8080	3	8.0	1680	0.0	1(
6	6/27/2014	257500.0	1715	6819	3	7.0	1715	0.0	1(
...	...	...	...	...	...	...	...	...	
21592	5/21/2014	360000.0	1530	1131	3	8.0	1530	0.0	2(
21593	2/23/2015	400000.0	2310	5813	3	8.0	2310	0.0	2(
21594	6/23/2014	402101.0	1020	1350	3	7.0	1020	0.0	2(
21595	1/16/2015	400000.0	1600	2388	3	8.0	1600	0.0	2(
21596	10/15/2014	325000.0	1020	1076	3	7.0	1020	0.0	2(

19512 rows × 41 columns

So these are now our columns:

In [40]:
```
ohe_df.columns
```

Out[40]:
```
Index(['date', 'price', 'sqft_living', 'sqft_lot', 'condition', 'grade',
       'sqft_above', 'sqft_basement', 'yr_built', 'yr_renovated', 'zipcod
e',
       'lat', 'long', 'sqft_living15', 'sqft_lot15', 'bedrooms_1',
       'bedrooms_2', 'bedrooms_3', 'bedrooms_4', 'bedrooms_5', 'bedrooms_
6',
       'bedrooms_7', 'bedrooms_8', 'bedrooms_9', 'bedrooms_10', 'bedrooms
_11',
       'bathrooms_0.0', 'bathrooms_1.0', 'bathrooms_2.0', 'bathrooms_3.
0',
       'bathrooms_4.0', 'bathrooms_5.0', 'bathrooms_6.0', 'bathrooms_7.
0',
       'bathrooms_8.0', 'floors_1.0', 'floors_1.5', 'floors_2.0', 'floors
_2.5',
       'floors_3.0', 'floors_3.5'],
      dtype='object')
```

OHE Linear Regression Model

We are ready now for our first linear regression model!
Let us run it, and we are going to keep all the possible variables, with only the essential exceptions. We are going to be excluding view and waterfront, since they have a lot of missing values therefore they describe a small part of the data anyway.
The variables date and year renovated are numerical but because of their particular meaning we are going to treat them separately later on to do a more in depth analysis about them.

In [41]:
```
y=ohe_df['price']
X=ohe_df.drop(['price','date', 'yr_renovated'], axis=1)

linreg= LinearRegression()
linreg.fit(X,y)
```

Out[41]:
```
LinearRegression()
```

In [42]:
```
R2= metrics.r2_score(y,linreg.predict(X))
print(f"{Fore.BLUE} The R squared value for this model done after One-Hot-E
```

```
 The R squared value for this model done after One-Hot-Encoding is  0.665
8783543447049
```

We see our R squared value already improving compared to the basic first regression.

Running the same model but with statsmodel to have more information

In [43]:
```
model = sm.OLS(y, X).fit()
model.summary()
```

Out[43]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.666
Model:	OLS	**Adj. R-squared:**	0.665
Method:	Least Squares	**F-statistic:**	1109.
Date:	Fri, 11 Nov 2022	**Prob (F-statistic):**	0.00
Time:	18:58:04	**Log-Likelihood:**	-2.5567e+05
No. Observations:	19512	**AIC:**	5.114e+05
Df Residuals:	19476	**BIC:**	5.117e+05
Df Model:	35		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
sqft_living	68.5748	11.753	5.835	0.000	45.537	91.612
sqft_lot	0.2634	0.031	8.464	0.000	0.202	0.324
condition	2.358e+04	1480.746	15.922	0.000	2.07e+04	2.65e+04
grade	7.617e+04	1391.676	54.730	0.000	7.34e+04	7.89e+04
sqft_above	2.8334	11.755	0.241	0.810	-20.207	25.874
sqft_basement	12.4466	11.647	1.069	0.285	-10.383	35.276
yr_built	-1940.6084	47.147	-41.161	0.000	-2033.020	-1848.197
zipcode	-177.0983	21.065	-8.407	0.000	-218.388	-135.809
lat	5.104e+05	6658.844	76.647	0.000	4.97e+05	5.23e+05
long	-5.135e+04	8322.522	-6.170	0.000	-6.77e+04	-3.5e+04
sqft_living15	57.9546	2.288	25.333	0.000	53.470	62.439
sqft_lot15	-0.1512	0.047	-3.221	0.001	-0.243	-0.059
bedrooms_1	-2.38e+06	4.62e+05	-5.155	0.000	-3.29e+06	-1.48e+06
bedrooms_2	-2.388e+06	4.62e+05	-5.172	0.000	-3.29e+06	-1.48e+06
bedrooms_3	-2.411e+06	4.61e+05	-5.226	0.000	-3.32e+06	-1.51e+06
bedrooms_4	-2.421e+06	4.61e+05	-5.247	0.000	-3.33e+06	-1.52e+06
bedrooms_5	-2.436e+06	4.61e+05	-5.279	0.000	-3.34e+06	-1.53e+06
bedrooms_6	-2.437e+06	4.61e+05	-5.283	0.000	-3.34e+06	-1.53e+06
bedrooms_7	-2.471e+06	4.62e+05	-5.350	0.000	-3.38e+06	-1.57e+06
bedrooms_8	-2.458e+06	4.63e+05	-5.313	0.000	-3.36e+06	-1.55e+06
bedrooms_9	-2.434e+06	4.65e+05	-5.236	0.000	-3.34e+06	-1.52e+06
bedrooms_10	-2.314e+06	4.65e+05	-4.978	0.000	-3.23e+06	-1.4e+06

bedrooms_11	-2.522e+06	4.74e+05	-5.324	0.000	-3.45e+06	-1.59e+06
bathrooms_0.0	-3.012e+06	5.67e+05	-5.311	0.000	-4.12e+06	-1.9e+06
bathrooms_1.0	-2.911e+06	5.63e+05	-5.168	0.000	-4.02e+06	-1.81e+06
bathrooms_2.0	-2.897e+06	5.63e+05	-5.143	0.000	-4e+06	-1.79e+06
bathrooms_3.0	-2.865e+06	5.63e+05	-5.085	0.000	-3.97e+06	-1.76e+06
bathrooms_4.0	-2.835e+06	5.64e+05	-5.032	0.000	-3.94e+06	-1.73e+06
bathrooms_5.0	-2.872e+06	5.64e+05	-5.095	0.000	-3.98e+06	-1.77e+06
bathrooms_6.0	-3.088e+06	5.67e+05	-5.445	0.000	-4.2e+06	-1.98e+06
bathrooms_7.0	-3.239e+06	5.75e+05	-5.634	0.000	-4.37e+06	-2.11e+06
bathrooms_8.0	-2.952e+06	5.79e+05	-5.095	0.000	-4.09e+06	-1.82e+06
floors_1.0	-4.495e+06	8.45e+05	-5.321	0.000	-6.15e+06	-2.84e+06
floors_1.5	-4.468e+06	8.45e+05	-5.288	0.000	-6.12e+06	-2.81e+06
floors_2.0	-4.458e+06	8.45e+05	-5.275	0.000	-6.11e+06	-2.8e+06
floors_2.5	-4.424e+06	8.45e+05	-5.234	0.000	-6.08e+06	-2.77e+06
floors_3.0	-4.418e+06	8.46e+05	-5.224	0.000	-6.08e+06	-2.76e+06
floors_3.5	-4.409e+06	8.47e+05	-5.207	0.000	-6.07e+06	-2.75e+06

Omnibus:	1945.504	**Durbin-Watson:**	1.989
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	3575.346
Skew:	0.681	**Prob(JB):**	0.00
Kurtosis:	4.594	**Cond. No.**	1.07e+16

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The smallest eigenvalue is 1.72e-18. This might indicate that there are
strong multicollinearity problems or that the design matrix is singular.

Next, we have to make sure that our model satisfies the assumptions for linear regression.

Assumptions of Linear Regression

There are some necessary assumptions to use linear regression and obtain trustworthy results.
Now are are going to manipulate our data to make sure that the variables we are working with fulfill those assumptions.
We already took care of the categorical variables with one hot encoding.
Now it is time to work on the continuous ones, and we will first normalize them and then scale them, whenever necessary, to satify the normality assumption for running this type of model.

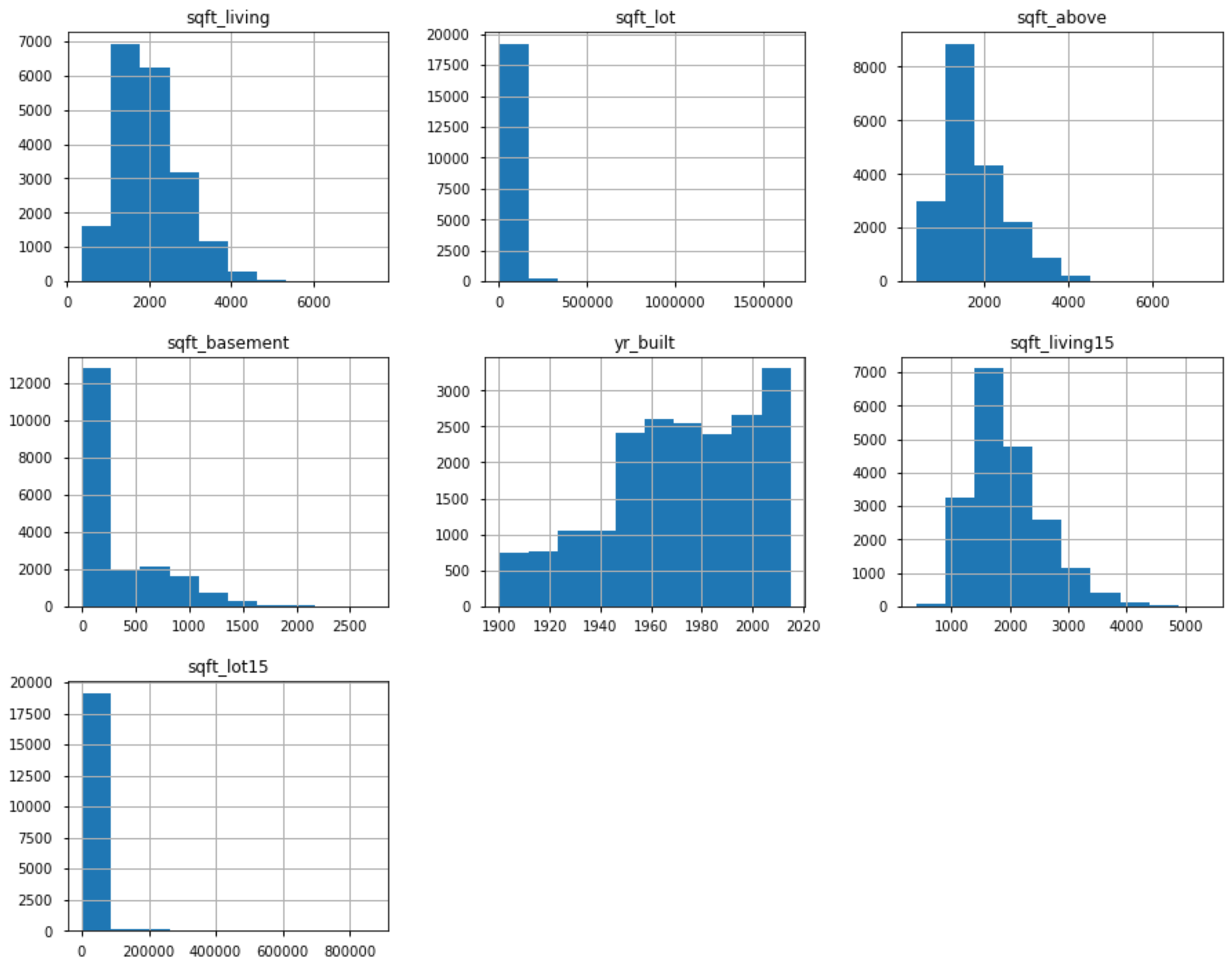
Transforming the variables, normalizing and scaling them

The easiest way to see if a variable is normally distributed is by plotting histograms.

The continuous variables on which we chose to focus on so far are: sqft_living, sqft_lot, sqft_above, sqft_basement, yr_built, sqftliving_15 and sqft_lot_15.

In [44]:
```
cols=['sqft_living', 'sqft_lot', 'sqft_above','sqft_basement', 'yr_built',
       'sqft_lot15']
```

In [45]:
```
df[cols].hist(figsize  = [15,12]);
```



From what we can observe, squarefoot living, sqft above and sqft living 15 could probably beneficiate from logarythmic normalization.
Sqft basement, Sqft lot and Sqft lot15 are also not following a normal distribution but they seem to be zero-inflated variables which is complicated to work with, so we might just leave them as they are.

In [46]:
```
df['sqft_basement'].value_counts()
```

Out[46]:
```
0.0       11988
600.0       208
500.0       198
700.0       191
800.0       187
          ...
1930.0        1
602.0         1
172.0         1
225.0         1
2200.0        1
Name: sqft_basement, Length: 270, dtype: int64
```

As expected, there are a lot of zeros for these variables.
It was probably classified this way when there is simply no basement in the house.
But we also don't want to drop all the columns with no basement, since 13280 is a consistent number and we don't want to lose all of that information.
We will leave it as is for now and select the features which don't have this characteristic.

Transformations

A simple way to normalize a variable is by transforming it, taking the logarythm of its value, since the log is a monotonically icreasing function this is not going to change the overall trend of our variable but is going to help us satisfy the requirements for the model, to have normally distributed variables.
We just need to remember when we are trying to draw some conclusions about these variables, that now we are working with the logarythm and not the original value itself.

In [47]:
```
df_log=pd.DataFrame()
non_normal = ['sqft_living', 'sqft_above', 'sqft_living15']
for feat in non_normal:
    df_log[feat] = df[feat].map(lambda x: np.log(x))
df_log = df_log.add_suffix('_log')
```

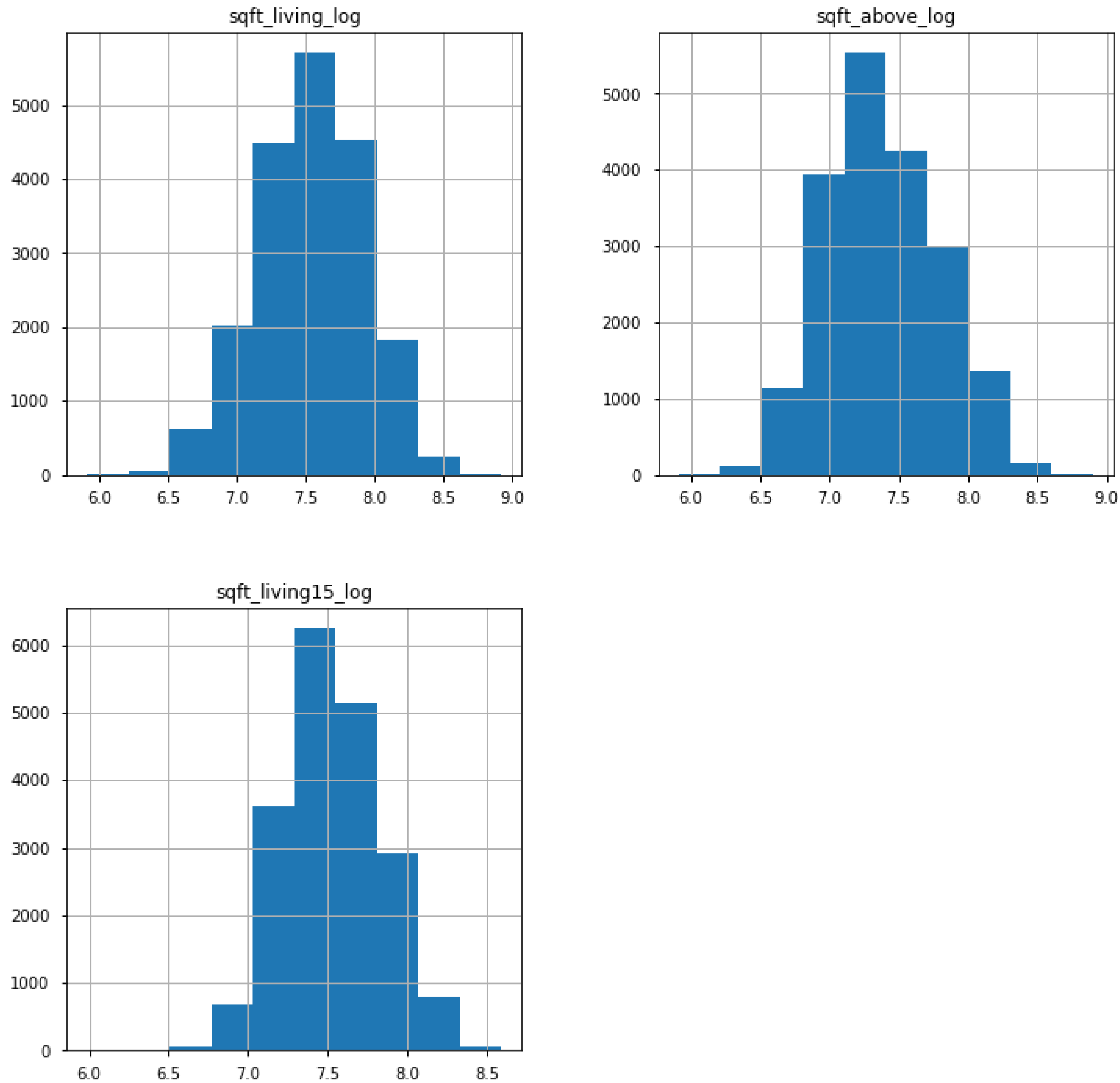
In [48]:
```
df_log
```

Out[48]:

	sqft_living_log	sqft_above_log	sqft_living15_log
0	7.073270	7.073270	7.200425
1	7.851661	7.682482	7.432484
3	7.580700	6.956545	7.215240
4	7.426549	7.426549	7.495542
6	7.447168	7.447168	7.713338
...	...	...	...
21592	7.333023	7.333023	7.333023
21593	7.745003	7.745003	7.512071
21594	6.927558	6.927558	6.927558
21595	7.377759	7.377759	7.251345
21596	6.927558	6.927558	6.927558

19512 rows × 3 columns

In [49]:
```
df_log.hist(figsize = [12,12]);
```



The values are a little more normal now.

Scaling

The process of scaling helps us putting all the different variables on the same scale, so that they can be compared in a more equal way when considering the coefficients.
There are several way of scaling, we are going to transform all the variables with both Min-Max_scaling and Normalization, and see which one brigns the better results.

Min Max Scaling

In the min max scaling each value gets the min subtracted and is divided by the difference between the max and the min of that variable. In this way all the values fall between 0 and 1. Here is the formula for it:

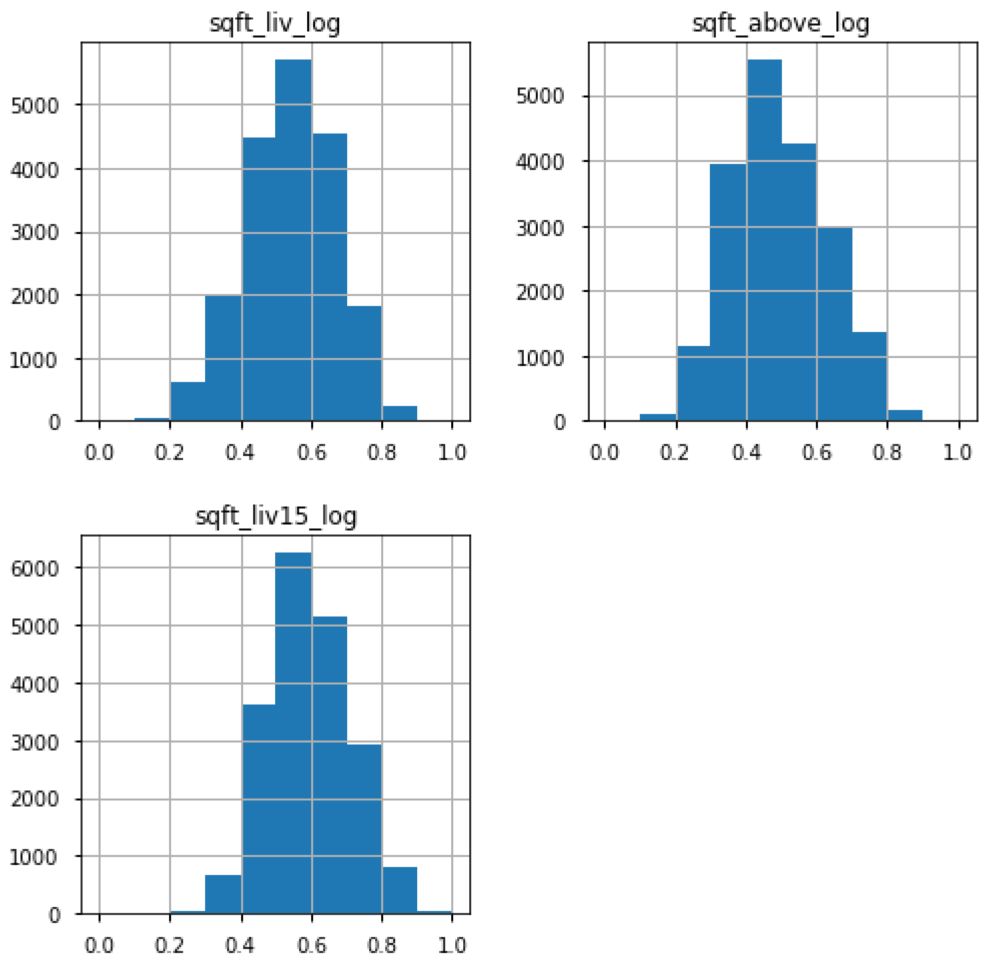
$$x' = \frac{x - min(x)}{(max(x) - min(x))}$$

In [50]:
```
sqll = df_log['sqft_living_log']
sqal  = df_log['sqft_above_log']
sq1v15 = df_log['sqft_living15_log']

scaled_sqll = (sqll - min(sqll)) / (max(sqll) - min(sqll))
scaled_sqal = (sqal - min(sqal)) / (max(sqal) - min(sqal))
scaled_sq1v15 = (sq1v15 - min(sq1v15)) / (max(sq1v15) - min(sq1v15))

data_cont_mmscaled = pd.DataFrame([])
data_cont_mmscaled['sqft_liv_log'] = scaled_sqll
data_cont_mmscaled['sqft_above_log'] = scaled_sqal
data_cont_mmscaled['sqft_liv15_log'] = scaled_sq1v15

data_cont_mmscaled.hist(figsize = [8,8]);
```



In [51]:
```
mmscaled_df=pd.concat([ohe_df,data_cont_mmscaled], axis=1)
```

In [52]:
```
mmscaled_df.head()
```

Out[52]:

	date	price	sqft_living	sqft_lot	condition	grade	sqft_above	sqft_basement	yr_built
0	10/13/2014	221900.0	1180	5650	3	7.0	1180	0.0	1955
1	12/9/2014	538000.0	2570	7242	3	7.0	2170	400.0	1951
3	12/9/2014	604000.0	1960	5000	5	7.0	1050	910.0	1965
4	2/18/2015	510000.0	1680	8080	3	8.0	1680	0.0	1987
6	6/27/2014	257500.0	1715	6819	3	7.0	1715	0.0	1995

5 rows × 44 columns

At this point I can drop the columns that I transformed and scaled

In [53]:
```
mmscaled_df.drop(['sqft_living', 'sqft_above', 'sqft_living15'], axis=1, in
mmscaled_df = mmscaled_df.dropna() # dropping null values that would give t
```

In [54]:
```
mmscaled_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 16057 entries, 0 to 21596
Data columns (total 41 columns):
 #   Column          Non-Null Count  Dtype  
---  ------          --------------  -----  
 0   date            16057 non-null  object 
 1   price           16057 non-null  float64
 2   sqft_lot        16057 non-null  int64  
 3   condition       16057 non-null  int64  
 4   grade           16057 non-null  float64
 5   sqft_basement   16057 non-null  float64
 6   yr_built        16057 non-null  int64  
 7   yr_renovated    16057 non-null  float64
 8   zipcode         16057 non-null  int64  
 9   lat             16057 non-null  float64
 10  long            16057 non-null  float64
 11  sqft_lot15      16057 non-null  int64  
 12  bedrooms_1      16057 non-null  float64
 13  bedrooms_2      16057 non-null  float64
 14  bedrooms_3      16057 non-null  float64
 15  bedrooms_4      16057 non-null  float64
 16  bedrooms_5      16057 non-null  float64
 17  bedrooms_6      16057 non-null  float64
 18  bedrooms_7      16057 non-null  float64
 19  bedrooms_8      16057 non-null  float64
 20  bedrooms_9      16057 non-null  float64
 21  bedrooms_10     16057 non-null  float64
 22  bedrooms_11     16057 non-null  float64
 23  bathrooms_0.0   16057 non-null  float64
 24  bathrooms_1.0   16057 non-null  float64
 25  bathrooms_2.0   16057 non-null  float64
 26  bathrooms_3.0   16057 non-null  float64
 27  bathrooms_4.0   16057 non-null  float64
 28  bathrooms_5.0   16057 non-null  float64
 29  bathrooms_6.0   16057 non-null  float64
 30  bathrooms_7.0   16057 non-null  float64
 31  bathrooms_8.0   16057 non-null  float64
 32  floors_1.0      16057 non-null  float64
 33  floors_1.5      16057 non-null  float64
 34  floors_2.0      16057 non-null  float64
 35  floors_2.5      16057 non-null  float64
 36  floors_3.0      16057 non-null  float64
 37  floors_3.5      16057 non-null  float64
 38  sqft_liv_log    16057 non-null  float64
 39  sqft_above_log  16057 non-null  float64
 40  sqft_liv15_log  16057 non-null  float64
dtypes: float64(35), int64(5), object(1)
memory usage: 5.1+ MB
```

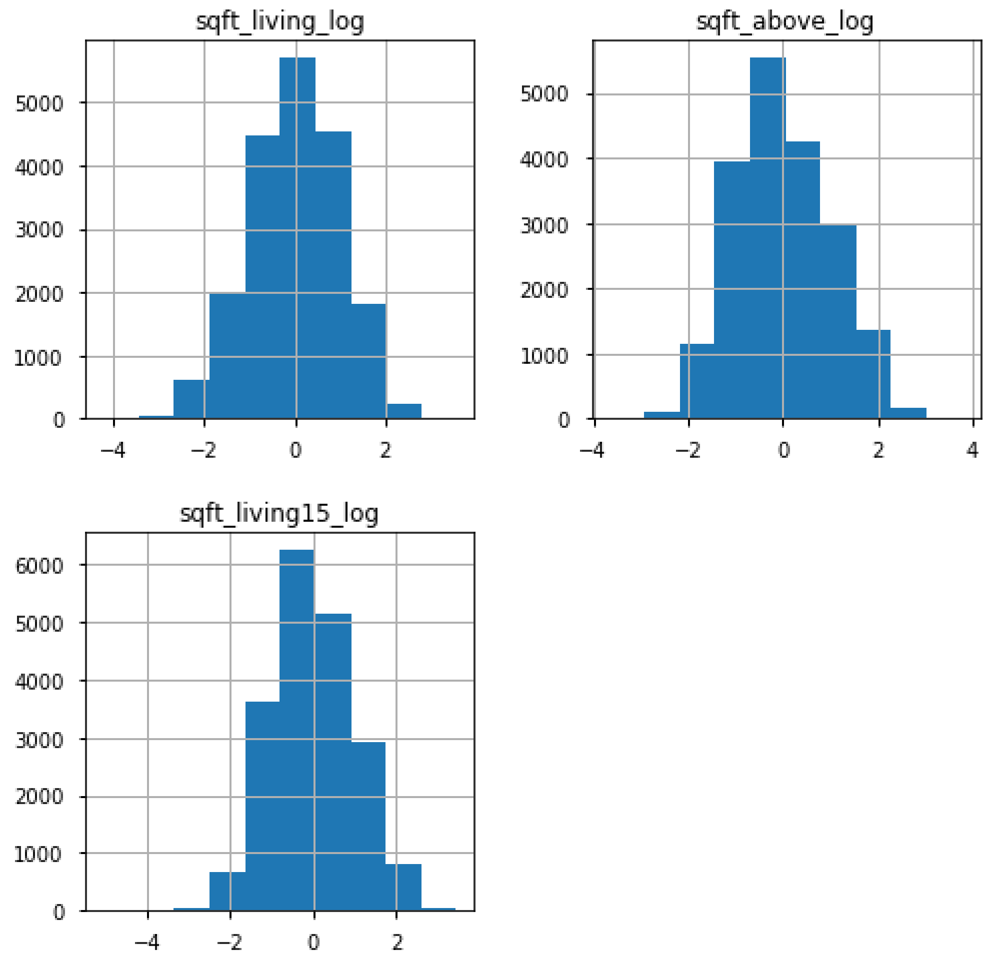
Normalizing instead of min max scaling

This time instead of using the MinMax scaler we are going to use a normalizer, which substracts the mean from the value and divides by the standard deviation. The features scaled this way are centered around zero and have a standard deviation of 1.

$$x' = \frac{x - \bar{x}}{\sigma}$$

In [55]:
```
def normalize(feature):
    return (feature - feature.mean()) / feature.std()

data_cont_norm = df_log.apply(normalize)
data_cont_norm.hist(figsize  = [8, 8]);
```



In [56]:
```
norm_df=pd.concat([ohe_df,data_cont_norm], axis=1)
```

In [57]:
```
norm_df.drop(['sqft_living', 'sqft_above', 'sqft_living15'], axis=1, inplac
norm_df = norm_df.dropna()
```

In [58]:
```
norm_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 16057 entries, 0 to 21596
Data columns (total 41 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   date               16057 non-null  object 
 1   price              16057 non-null  float64
 2   sqft_lot           16057 non-null  int64  
 3   condition          16057 non-null  int64  
 4   grade              16057 non-null  float64
 5   sqft_basement      16057 non-null  float64
 6   yr_built           16057 non-null  int64  
 7   yr_renovated       16057 non-null  float64
 8   zipcode            16057 non-null  int64  
 9   lat                16057 non-null  float64
 10  long               16057 non-null  float64
 11  sqft_lot15         16057 non-null  int64  
 12  bedrooms_1         16057 non-null  float64
 13  bedrooms_2         16057 non-null  float64
 14  bedrooms_3         16057 non-null  float64
 15  bedrooms_4         16057 non-null  float64
 16  bedrooms_5         16057 non-null  float64
 17  bedrooms_6         16057 non-null  float64
 18  bedrooms_7         16057 non-null  float64
 19  bedrooms_8         16057 non-null  float64
 20  bedrooms_9         16057 non-null  float64
 21  bedrooms_10        16057 non-null  float64
 22  bedrooms_11        16057 non-null  float64
 23  bathrooms_0.0      16057 non-null  float64
 24  bathrooms_1.0      16057 non-null  float64
 25  bathrooms_2.0      16057 non-null  float64
 26  bathrooms_3.0      16057 non-null  float64
 27  bathrooms_4.0      16057 non-null  float64
 28  bathrooms_5.0      16057 non-null  float64
 29  bathrooms_6.0      16057 non-null  float64
 30  bathrooms_7.0      16057 non-null  float64
 31  bathrooms_8.0      16057 non-null  float64
 32  floors_1.0         16057 non-null  float64
 33  floors_1.5         16057 non-null  float64
 34  floors_2.0         16057 non-null  float64
 35  floors_2.5         16057 non-null  float64
 36  floors_3.0         16057 non-null  float64
 37  floors_3.5         16057 non-null  float64
 38  sqft_living_log    16057 non-null  float64
 39  sqft_above_log     16057 non-null  float64
 40  sqft_living15_log  16057 non-null  float64
dtypes: float64(35), int64(5), object(1)
memory usage: 5.1+ MB
```

MinMax Scaler Linear Regression

Let us try a linear regression with the Min Max scaling, and then we will do it again with the normal scaling and we can compare the results.

In [59]:
```
y=mmscaled_df['price']
X=mmscaled_df.drop(['price', 'date', 'yr_renovated'], axis=1)

linreg= LinearRegression()
linreg.fit(X,y)
LinearRegression()
```

Out[59]: LinearRegression()

In [60]:
```
R2=metrics.r2_score(y,linreg.predict(X))
print(f"{Fore.MAGENTA}The R squared value for this model scaled with MinMax
{Style.RESET_ALL}", R2)
```

```
The R squared value for this model scaled with MinMaxScaling is  0.660987
1645016018
```

Normal Scaler Linear Regression

In [61]:
```
y=norm_df['price']
X=norm_df.drop(['price', 'date', 'yr_renovated'], axis=1)

linreg= LinearRegression()
linreg.fit(X,y)
```

Out[61]: LinearRegression()

In [62]:
```
R2=metrics.r2_score(y,linreg.predict(X))
print(f"{Fore.GREEN}The R squared value for this model scaled with Normal S
{Style.RESET_ALL}", R2)
```

```
The R squared value for this model scaled with Normal Scaler is  0.660987
1645016014
```

How interesting! The two different scaling gave us a very similar R squared value, almost exactly the same!
Also R squared has not really improved, it has actually slightly decreased from the scaling and normalization.
We didn't expect the transformations to increase considerably the model performance, but it was still important to do them to be able to run a linear regression model fulfilling all the assumptions.
But there is still more improving that we can do for the model!

Next we need to check for multicollinearity and try to drop the variables that have p>0.05.

Multicollinearity

Multicollinearity is when the assumptions that all the variables are independent is NOT fulfilled, which can cause problems because if one variable is strictly dependent on another then the model interpretation becomes hard.
This is so because in general we want to be able to tell which one of the specific variables is

influencing the target in which specific way, but if two variables influence also each other it is hard to interpret that.
Multicollinearity is also a direct consequence of One-Hot-Encoding, but we can fix that by dropping one variable for each category that we splitted with OHE.

In [63]:
```
norm_df.corr().head()
```

Out[63]:

	price	sqft_lot	condition	grade	sqft_basement	yr_built	yr_renovated	
price	1.000000	0.093164	0.032698	0.611630	0.213646	0.023411	0.091812	-(
sqft_lot	0.093164	1.000000	0.000768	0.096755	0.016532	0.042459	0.007583	-(
condition	0.032698	0.000768	1.000000	-0.187875	0.175993	-0.378006	-0.063469	(
grade	0.611630	0.096755	-0.187875	1.000000	0.049647	0.477778	-0.020651	-(
sqft_basement	0.213646	0.016532	0.175993	0.049647	1.000000	-0.163938	0.042148	(

5 rows × 40 columns

This matrix is hard to interpret with so many variables but we are going to filter only the correlations that are above 0.75, which are considered strong correlations.

In [64]:
```
corr_matr=abs(norm_df.corr()) > 0.75
corr_matr.head()
```

Out[64]:

	price	sqft_lot	condition	grade	sqft_basement	yr_built	yr_renovated	zipcode	
price	True	False	False	False	False	False	False	False	Fal
sqft_lot	False	True	False	False	False	False	False	False	Fal
condition	False	False	True	False	False	False	False	False	Fal
grade	False	False	False	True	False	False	False	False	Fal
sqft_basement	False	False	False	False	True	False	False	False	Fal

5 rows × 40 columns

Creating a dataframe with this information

In [65]:
```
df_cc=norm_df.corr().abs().stack().reset_index().sort_values(0, ascending=F

# zip the variable name columns (Which were only named level_0 and level_1
df_cc['pairs'] = list(zip(df_cc.level_0, df_cc.level_1))

# set index to pairs
df_cc.set_index(['pairs'], inplace = True)

#drop level columns
df_cc.drop(columns=['level_1', 'level_0'], inplace = True)

# rename correlation column as cc rather than 0
df_cc.columns = ['cc']
```

In [66]:
```
df_cc.drop_duplicates(inplace=True)
df_cc[(df_cc.cc>.75) & (df_cc.cc <1)]
```

Out[66]:

	cc
pairs	
(sqft_above_log, sqft_living_log)	0.83786
(floors_1.0, floors_2.0)	0.77745

It looks like squarefoot living and squarefoot above are strongly correlated and that makes sense. So we will drop squarefoot living to avoid multicollinearity.
1 floor and 2 floors are highly correlated too, as we know from one hot encoding when we are supposed to drop the first column to avoid multicollineairty. We will drop one column for each categorical value in the following LRM.
But first we are going to check the ohe linear model we previously ran, but doing it with statsmodel to see the coefficients, so to drop the ones that have p>0.05

P values, F-statistic

Same model that we ran right after normalization, doing it with statsmodel to see the coefficients.

In [67]:
```
norm_df.drop('date', axis=1, inplace=True)
```

In [68]:
```
y=norm_df['price']
X=norm_df.drop(['price', 'yr_renovated'], axis=1)
X = sm.add_constant(X)
model = sm.OLS(y, X).fit()
model.summary()
```

Out[68]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.661
Model:	OLS	**Adj. R-squared:**	0.660
Method:	Least Squares	**F-statistic:**	918.8
Date:	Fri, 11 Nov 2022	**Prob (F-statistic):**	0.00
Time:	18:58:10	**Log-Likelihood:**	-2.1053e+05
No. Observations:	16057	**AIC:**	4.211e+05
Df Residuals:	16022	**BIC:**	4.214e+05
Df Model:	34		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	-7.281e+06	1.51e+06	-4.808	0.000	-1.02e+07	-4.31e+06
sqft_lot	0.2429	0.034	7.222	0.000	0.177	0.309
condition	2.335e+04	1641.365	14.224	0.000	2.01e+04	2.66e+04
grade	8.125e+04	1510.598	53.786	0.000	7.83e+04	8.42e+04
sqft_basement	36.0243	6.787	5.308	0.000	22.722	49.327
yr_built	-1970.3933	52.556	-37.492	0.000	-2073.408	-1867.378
zipcode	-162.7267	23.456	-6.937	0.000	-208.703	-116.750
lat	5.135e+05	7395.891	69.430	0.000	4.99e+05	5.28e+05
long	-4.31e+04	9189.560	-4.690	0.000	-6.11e+04	-2.51e+04
sqft_lot15	-0.1091	0.051	-2.139	0.032	-0.209	-0.009
bedrooms_1	-5.988e+05	1.39e+05	-4.305	0.000	-8.71e+05	-3.26e+05
bedrooms_2	-6.231e+05	1.39e+05	-4.486	0.000	-8.95e+05	-3.51e+05
bedrooms_3	-6.605e+05	1.39e+05	-4.765	0.000	-9.32e+05	-3.89e+05
bedrooms_4	-6.667e+05	1.39e+05	-4.810	0.000	-9.38e+05	-3.95e+05
bedrooms_5	-6.787e+05	1.39e+05	-4.896	0.000	-9.5e+05	-4.07e+05
bedrooms_6	-6.82e+05	1.39e+05	-4.917	0.000	-9.54e+05	-4.1e+05
bedrooms_7	-7.026e+05	1.41e+05	-4.989	0.000	-9.79e+05	-4.27e+05
bedrooms_8	-6.789e+05	1.44e+05	-4.700	0.000	-9.62e+05	-3.96e+05
bedrooms_9	-6.724e+05	1.51e+05	-4.439	0.000	-9.69e+05	-3.76e+05
bedrooms_10	-5.523e+05	1.52e+05	-3.640	0.000	-8.5e+05	-2.55e+05

bedrooms_11	-7.647e+05	1.76e+05	-4.346	0.000	-1.11e+06	-4.2e+05
bathrooms_0.0	-1.019e+06	2.04e+05	-4.988	0.000	-1.42e+06	-6.18e+05
bathrooms_1.0	-8.93e+05	1.9e+05	-4.695	0.000	-1.27e+06	-5.2e+05
bathrooms_2.0	-8.92e+05	1.9e+05	-4.693	0.000	-1.26e+06	-5.19e+05
bathrooms_3.0	-8.556e+05	1.9e+05	-4.498	0.000	-1.23e+06	-4.83e+05
bathrooms_4.0	-8.064e+05	1.9e+05	-4.234	0.000	-1.18e+06	-4.33e+05
bathrooms_5.0	-8.377e+05	1.92e+05	-4.367	0.000	-1.21e+06	-4.62e+05
bathrooms_6.0	-1.069e+06	2.05e+05	-5.216	0.000	-1.47e+06	-6.67e+05
bathrooms_7.0	-9.562e-10	2.11e-10	-4.540	0.000	-1.37e-09	-5.43e-10
bathrooms_8.0	-9.086e+05	2.28e+05	-3.992	0.000	-1.35e+06	-4.62e+05
floors_1.0	-1.253e+06	2.52e+05	-4.972	0.000	-1.75e+06	-7.59e+05
floors_1.5	-1.234e+06	2.52e+05	-4.893	0.000	-1.73e+06	-7.4e+05
floors_2.0	-1.219e+06	2.52e+05	-4.832	0.000	-1.71e+06	-7.25e+05
floors_2.5	-1.195e+06	2.53e+05	-4.729	0.000	-1.69e+06	-7e+05
floors_3.0	-1.183e+06	2.53e+05	-4.673	0.000	-1.68e+06	-6.87e+05
floors_3.5	-1.197e+06	2.57e+05	-4.657	0.000	-1.7e+06	-6.93e+05
sqft_living_log	3.17e+04	5317.143	5.961	0.000	2.13e+04	4.21e+04
sqft_above_log	2.217e+04	4903.911	4.522	0.000	1.26e+04	3.18e+04
sqft_living15_log	3.409e+04	1518.228	22.454	0.000	3.11e+04	3.71e+04

Omnibus:	1346.571	**Durbin-Watson:**	1.989
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	2350.573
Skew:	0.608	**Prob(JB):**	0.00
Kurtosis:	4.426	**Cond. No.**	1.01e+16

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The smallest eigenvalue is 1.6e-18. This might indicate that there are
strong multicollinearity problems or that the design matrix is singular.

Beside all the categorical values from one hot encoding, none of the other variables have p>0.05. So let us run another linear regression model dropping a column for each categorical variable, and squarefoot living.

P-values Linear Regression Model

In [69]:
```
y=norm_df['price']
X=norm_df.drop(['price', 'yr_renovated', 'bedrooms_1','bathrooms_1.0','floo
X = sm.add_constant(X)
model = sm.OLS(y, X).fit()
model.summary()
```

Out[69]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.660
Model:	OLS	**Adj. R-squared:**	0.660
Method:	Least Squares	**F-statistic:**	943.5
Date:	Fri, 11 Nov 2022	**Prob (F-statistic):**	0.00
Time:	18:58:10	**Log-Likelihood:**	-2.1055e+05
No. Observations:	16057	**AIC:**	4.212e+05
Df Residuals:	16023	**BIC:**	4.214e+05
Df Model:	33		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	-1.094e+07	2.09e+06	-5.234	0.000	-1.5e+07	-6.84e+06
sqft_lot	0.2430	0.034	7.218	0.000	0.177	0.309
condition	2.342e+04	1643.083	14.255	0.000	2.02e+04	2.66e+04
grade	8.181e+04	1509.251	54.208	0.000	7.89e+04	8.48e+04
sqft_basement	72.0932	3.077	23.432	0.000	66.063	78.124
yr_built	-1980.5793	52.584	-37.665	0.000	-2083.651	-1877.508
zipcode	-153.7007	23.432	-6.559	0.000	-199.631	-107.770
lat	5.137e+05	7403.759	69.387	0.000	4.99e+05	5.28e+05
long	-4.317e+04	9199.450	-4.692	0.000	-6.12e+04	-2.51e+04
sqft_lot15	-0.1090	0.051	-2.135	0.033	-0.209	-0.009
bedrooms_2	-2.004e+04	1.12e+04	-1.793	0.073	-4.2e+04	1868.482
bedrooms_3	-5.529e+04	1.12e+04	-4.945	0.000	-7.72e+04	-3.34e+04
bedrooms_4	-5.951e+04	1.14e+04	-5.205	0.000	-8.19e+04	-3.71e+04
bedrooms_5	-7.18e+04	1.2e+04	-5.970	0.000	-9.54e+04	-4.82e+04
bedrooms_6	-7.507e+04	1.44e+04	-5.204	0.000	-1.03e+05	-4.68e+04
bedrooms_7	-9.258e+04	3.04e+04	-3.049	0.002	-1.52e+05	-3.31e+04
bedrooms_8	-7.62e+04	4.71e+04	-1.616	0.106	-1.69e+05	1.62e+04
bedrooms_9	-7.224e+04	7.05e+04	-1.025	0.306	-2.1e+05	6.6e+04
bedrooms_10	4.569e+04	7.1e+04	0.643	0.520	-9.36e+04	1.85e+05
bedrooms_11	-1.593e+05	1.21e+05	-1.321	0.187	-3.96e+05	7.71e+04

bathrooms_0.0	-1.02e+05	8.5e+04	-1.200	0.230	-2.69e+05	6.46e+04
bathrooms_2.0	6106.3792	3333.214	1.832	0.067	-427.094	1.26e+04
bathrooms_3.0	4.225e+04	4752.387	8.890	0.000	3.29e+04	5.16e+04
bathrooms_4.0	8.984e+04	6532.049	13.753	0.000	7.7e+04	1.03e+05
bathrooms_5.0	5.578e+04	3.11e+04	1.795	0.073	-5130.448	1.17e+05
bathrooms_6.0	-1.834e+05	8.62e+04	-2.127	0.033	-3.52e+05	-1.44e+04
bathrooms_7.0	6.328e-09	7.92e-08	0.080	0.936	-1.49e-07	1.61e-07
bathrooms_8.0	6306.1015	1.39e+05	0.045	0.964	-2.66e+05	2.78e+05
floors_1.5	1.799e+04	3807.181	4.725	0.000	1.05e+04	2.55e+04
floors_2.0	3.351e+04	2971.551	11.277	0.000	2.77e+04	3.93e+04
floors_2.5	5.79e+04	1.32e+04	4.398	0.000	3.21e+04	8.37e+04
floors_3.0	6.623e+04	6622.140	10.001	0.000	5.32e+04	7.92e+04
floors_3.5	5.401e+04	5.38e+04	1.004	0.315	-5.14e+04	1.59e+05
sqft_above_log	4.878e+04	2034.961	23.969	0.000	4.48e+04	5.28e+04
sqft_living15_log	3.532e+04	1505.750	23.457	0.000	3.24e+04	3.83e+04

Omnibus:	1319.698	**Durbin-Watson:**	1.988
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	2294.337
Skew:	0.600	**Prob(JB):**	0.00
Kurtosis:	4.411	**Cond. No.**	1.01e+16

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The smallest eigenvalue is 1.6e-18. This might indicate that there are
strong multicollinearity problems or that the design matrix is singular.

With this linear model R is still around 6.66 but more values with high p have emerged but these are probably false positives for p, given the high number of variables we are dealing with.

We finished normalizing and scaling for our model to fit within the criteria of normality.
We also want to avoid multicollinearity if we want our coefficients from our model to be reliable and to help us give a better interpretation of the relationship between the variables.
Now that we took care of all this, it is time to do some feature engineering to see what other types of variables that are present in the data we have could help us improve our model.

Feature Engineering

Feature engineering is about finding new features that we can incorporate in our model and that

can improve the way it describes and fits the data.
We are going to look at three features that are already provided (renovations, zipcode and time of sale) but we are going to manipulate them so that they can tell us something more and the information that they carry can be better interpreted by the model.
Then after that we are going to try an unusual way to find other features that can help our model too.

Renovations

Let us explore the information that we have about houses that were renovated. Since these houses were already a small number to begin with, in this case we are going to take again the original database without the removal of the outliers, to have as much data as possible.

In [70]:
```
df_orig=pd.read_csv('Data/kc_house_data.csv')
```

In [71]:
```
df_orig.head()
```

Out[71]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront
0	7129300520	10/13/2014	221900.0	3	1.00	1180	5650	1.0	NaN
1	6414100192	12/9/2014	538000.0	3	2.25	2570	7242	2.0	NO
2	5631500400	2/25/2015	180000.0	2	1.00	770	10000	1.0	NO
3	2487200875	12/9/2014	604000.0	4	3.00	1960	5000	1.0	NO
4	1954400510	2/18/2015	510000.0	3	2.00	1680	8080	1.0	NO

5 rows × 21 columns

In [72]:
```
df_orig['yr_renovated'].value_counts()
```

Out[72]:
```
0.0       17011
2014.0       73
2003.0       31
2013.0       31
2007.0       30
          ...
1946.0        1
1959.0        1
1971.0        1
1951.0        1
1954.0        1
Name: yr_renovated, Length: 70, dtype: int64
```

In [73]:
```
df_orig['yr_renovated'].isna().sum()
```

Out[73]:
```
3842
```

In [74]:
```
df_orig['yr_renovated'] = df_orig['yr_renovated'].fillna(0)
```

Let us create a dataframe with only houses that were renovated so we can study them better

In [75]:
```
reno=pd.DataFrame()
reno=df_orig.loc[df_orig['yr_renovated']!=0]
```

In [76]:
```
reno
```

Out[76]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfr
1	6414100192	12/9/2014	538000.0	3	2.25	2570	7242	2.0	
35	9547205180	6/13/2014	696000.0	3	2.50	2300	3060	1.5	
95	1483300570	9/8/2014	905000.0	4	2.50	3300	10250	1.0	
103	2450000295	10/7/2014	1090000.0	3	2.50	2920	8113	2.0	
125	4389200955	3/2/2015	1450000.0	4	2.75	2750	17789	1.5	
...	...	...	...	...	...	...	...	...	
19602	6392000625	7/12/2014	451000.0	2	1.00	900	6000	1.0	
20041	126039256	9/4/2014	434900.0	3	2.00	1520	5040	2.0	
20428	4305600360	2/25/2015	500012.0	4	2.50	2400	9612	1.0	
20431	3319500628	2/12/2015	356999.0	3	1.50	1010	1546	2.0	N
20946	1278000210	3/11/2015	110000.0	2	1.00	828	4524	1.0	

744 rows × 21 columns

Ok so the houses that have actually been renovated are only 744. Let us run some statistics on them.

In [77]:
```
reno.describe()
```

Out[77]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floo
count	7.440000e+02	7.440000e+02	744.000000	744.000000	744.000000	744.000000	744.0000(
mean	4.418716e+09	7.689019e+05	3.459677	2.306116	2327.377688	16215.530914	1.5047(
std	2.908265e+09	6.271258e+05	1.068823	0.898233	1089.002040	38235.308760	0.4932(
min	3.600057e+06	1.100000e+05	1.000000	0.750000	520.000000	1024.000000	1.0000(
25%	1.922985e+09	4.122500e+05	3.000000	1.750000	1560.000000	5000.000000	1.0000(
50%	3.899100e+09	6.075020e+05	3.000000	2.250000	2200.000000	7375.000000	1.5000(
75%	7.014200e+09	9.000000e+05	4.000000	2.750000	2872.500000	12670.750000	2.0000(
max	9.829200e+09	7.700000e+06	11.000000	8.000000	12050.000000	478288.000000	3.0000(

This is already giving us a lot of information.
The houses that have been renovated have on average 3.46 bedrooms, 2.3 bathrooms, 2327 squarefeet of living space. These houses were on average built in 1939 and renovated around 1996 and sold on average for 769 thousand dollars.

Let us take a look at some information from the general database of houses that were NOT renovated.

In [78]:
```
noren=pd.DataFrame()
noren=df_orig.loc[df_orig['yr_renovated']==0]
```

In [79]:
```
noren.describe()
```

Out[79]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	
count	2.085300e+04	2.085300e+04	20853.000000	20853.000000	20853.000000	2.085300e+04	20853.
mean	4.586246e+09	5.321403e+05	3.370115	2.109037	2071.507313	1.505959e+04	1.
std	2.875507e+09	3.518947e+05	0.920686	0.763118	910.209733	4.152180e+04	0.
min	1.000102e+06	7.800000e+04	1.000000	0.500000	370.000000	5.200000e+02	1.
25%	2.125059e+09	3.200000e+05	3.000000	1.500000	1420.000000	5.050000e+03	1.
50%	3.904931e+09	4.490000e+05	3.000000	2.250000	1900.000000	7.620000e+03	1.
75%	7.326200e+09	6.350000e+05	4.000000	2.500000	2540.000000	1.062600e+04	2.
max	9.900000e+09	6.890000e+06	33.000000	8.000000	13540.000000	1.651359e+06	3.

The houses that have not been renovated have on average 3.37 bedrooms, 2.1 bathrooms, 2072 squarefeet of living space. These houses were on average built in 1972 and sold on average for $532,140.

We can already see a considerable difference, considering that the number of bedrooms and bathrooms is similar, the lot size is not that close but comparable, and the price difference is around $230,000.

But we can do this in a more precise way to get a more accurate sense of the difference.
We are going to scout in the non renovated DataFrame for houses exactly in the range of the renovated houses, take a sample from there and calculate the average values for even more accurate figures.

In [80]:
```
filtered_values=pd.DataFrame()
filtered_values = noren.loc[((noren['bedrooms']>=3) & (noren['bedrooms']<=4
                            (noren['bathrooms']>=2) & (noren['sqft_living'
```

In [81]:
```
filtered_values.describe()
```

Out[81]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	fl
count	3.665000e+03	3.665000e+03	3665.000000	3665.000000	3665.000000	3665.000000	3665.00(
mean	4.710393e+09	5.379679e+05	3.586357	2.453615	2325.689495	15680.863302	1.65
std	2.895765e+09	2.072246e+05	0.492553	0.229057	198.732436	35649.527682	0.47
min	3.600072e+06	1.942500e+05	3.000000	2.000000	2005.000000	1159.000000	1.00(
25%	2.206700e+09	3.760000e+05	3.000000	2.250000	2150.000000	5400.000000	1.00(
50%	4.027700e+09	5.100000e+05	4.000000	2.500000	2310.000000	7857.000000	2.00(
75%	7.524951e+09	6.499500e+05	4.000000	2.500000	2490.000000	11100.000000	2.00(
max	9.839301e+09	1.950000e+06	4.000000	3.000000	2700.000000	715690.000000	3.00(

In this way we obtained values even closer to the ones for the renovated houses.
For a house that was not renovated, that has on average 3.5 bedrooms, 2.4 bathrooms and 2325 squarefeet of living space, the average selling price is
538 thousand dollars.
So given the same characteristics on average for a house that was renovated and one that wasn't, the house that was renovated got sold for roughly on average $230 thousand more.

In [82]:
```
reno.insert(1, 'reno', 1)
filtered_values.insert(1,'reno',0)
```

In [83]:
```
filtered_values
```

Out[83]:

	id	reno	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors
24	3814700200	0	11/20/2014	329000.0	3	2.25	2450	6500	2.0
29	1873100390	0	3/2/2015	719000.0	4	2.50	2570	7173	2.0
30	8562750320	0	11/10/2014	580500.0	3	2.50	2320	3980	2.0
34	7955080270	0	12/3/2014	322500.0	4	2.75	2060	6659	1.0
37	2768000400	0	12/30/2014	640000.0	4	2.00	2360	6000	2.0
...	...	...	...	...	...	...	...	...	...
21573	7570050450	0	9/10/2014	347500.0	3	2.50	2540	4760	2.0
21578	5087900040	0	10/17/2014	350000.0	4	2.75	2500	5995	2.0
21587	7852140040	0	8/25/2014	507250.0	3	2.50	2270	5536	2.0
21589	3448900210	0	10/14/2014	610685.0	4	2.50	2520	6023	2.0
21593	6600060120	0	2/23/2015	400000.0	4	2.50	2310	5813	2.0

3665 rows × 22 columns

Creating one database with the houses of comparable characteristic, where some of them renovated and some of them not.

In [84]:
```
compar=pd.concat([reno,filtered_values], axis=0)
```

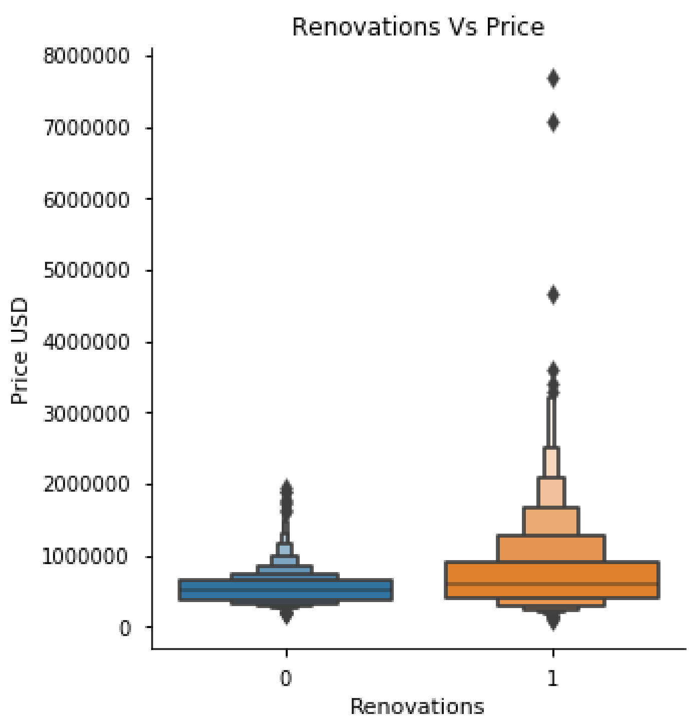
In [85]:
```
compar
```

Out[85]:

	id	reno	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors
1	6414100192	1	12/9/2014	538000.0	3	2.25	2570	7242	2.0
35	9547205180	1	6/13/2014	696000.0	3	2.50	2300	3060	1.5
95	1483300570	1	9/8/2014	905000.0	4	2.50	3300	10250	1.0
103	2450000295	1	10/7/2014	1090000.0	3	2.50	2920	8113	2.0
125	4389200955	1	3/2/2015	1450000.0	4	2.75	2750	17789	1.5
...	...	...	...	...	...	...	...	...	...
21573	7570050450	0	9/10/2014	347500.0	3	2.50	2540	4760	2.0
21578	5087900040	0	10/17/2014	350000.0	4	2.75	2500	5995	2.0
21587	7852140040	0	8/25/2014	507250.0	3	2.50	2270	5536	2.0
21589	3448900210	0	10/14/2014	610685.0	4	2.50	2520	6023	2.0
21593	6600060120	0	2/23/2015	400000.0	4	2.50	2310	5813	2.0

4409 rows × 22 columns

In [86]:
```
ax=sns.catplot(data=compar, x="reno", y="price", kind="boxen")
ax.set(ylabel="Price USD",
       xlabel="Renovations", title='Renovations Vs Price');
```



This gives us a sense also visually of how valuable it is to renovate a house vs not.
In general it seems like buying a house for a lower price (in worse conditions or with worse grading) keeping some money aside to put toward renovations could be a good idea. The value of the

houses that underwent renovations is clearly higher compared to the ones that didn't.

At this point to be able to add this information in our linear regression we are going to transform the information about renovation in a binary variable in our originial DataFrame.

In [87]:
```
df['reno']=0
```

In [88]:
```
df.loc[df['yr_renovated']!=0, 'reno']='1'
```

In [89]:
```
df['reno'].value_counts()
```

Out[89]:
```
0    15446
1     4066
Name: reno, dtype: int64
```

At this point we can drop the 'yr_renovated' column and add this new column to the one hot encoding DataFrame and run the model again, with the renovation information.

In [90]:
```
ohe_df['reno']=df['reno']
ohe_df.drop('yr_renovated', axis=1, inplace=True)
```

In [91]:
```
ohe_df.dropna(inplace=True)
```

In [92]:
```
y=ohe_df['price']
X=ohe_df.drop(['price','date'], axis=1)

linreg= LinearRegression()
linreg.fit(X,y)
```

Out[92]:
```
LinearRegression()
```

In [93]:
```
R2= metrics.r2_score(y,linreg.predict(X))
print(f"{Fore.BLUE}The R squared value for this model after One-Hot-Encodin
```

```
The R squared value for this model after One-Hot-Encoding the renovation
 factor is  0.66603
```

Slightly better than before, but let's see what else we can do.

Zipcodes

Another feature that we might want to consider and include more seriously in our model is the zipcode of the house, since it gives us an indication of the area and also since it does not really make sense to be interpreted as a numerical value, unless it is treated with one hot encoding.

In [94]:
```
norm_df
```

Out[94]:

	price	sqft_lot	condition	grade	sqft_basement	yr_built	yr_renovated	zipcode	lat
0	221900.0	5650	3	7.0	0.0	1955	0.0	98178	47.5112
1	538000.0	7242	3	7.0	400.0	1951	1991.0	98125	47.7210
3	604000.0	5000	5	7.0	910.0	1965	0.0	98136	47.5208
4	510000.0	8080	3	8.0	0.0	1987	0.0	98074	47.6168
6	257500.0	6819	3	7.0	0.0	1995	0.0	98003	47.3097
...	...	...	...	...	...	...	...	...	...
21592	360000.0	1131	3	8.0	0.0	2009	0.0	98103	47.6993
21593	400000.0	5813	3	8.0	0.0	2014	0.0	98146	47.5107
21594	402101.0	1350	3	7.0	0.0	2009	0.0	98144	47.5944
21595	400000.0	2388	3	8.0	0.0	2004	0.0	98027	47.5345
21596	325000.0	1076	3	7.0	0.0	2008	0.0	98144	47.5941

16057 rows × 40 columns

In [95]:
```
# Categorical columns
cat_columns = ['zipcode']

# Fit encoder on training set
ohe.fit(df[cat_columns])

# Get new column names
new_cat_columns = ohe.get_feature_names(input_features=cat_columns)

# Transform training set
df_zipcode = pd.DataFrame(ohe.fit_transform(df[cat_columns]),
                          columns=new_cat_columns, index=df.index)

# Replace training columns with transformed versions
normzip_df = pd.concat([norm_df.drop(cat_columns, axis=1), df_zipcode], axi
normzip_df.dropna(inplace=True)
normzip_df
```

Out[95]:

	price	sqft_lot	condition	grade	sqft_basement	yr_built	yr_renovated	lat	long
0	221900.0	5650.0	3.0	7.0	0.0	1955.0	0.0	47.5112	-122.257
1	538000.0	7242.0	3.0	7.0	400.0	1951.0	1991.0	47.7210	-122.319
3	604000.0	5000.0	5.0	7.0	910.0	1965.0	0.0	47.5208	-122.393
4	510000.0	8080.0	3.0	8.0	0.0	1987.0	0.0	47.6168	-122.045
6	257500.0	6819.0	3.0	7.0	0.0	1995.0	0.0	47.3097	-122.327
...	...	...	...	...	...	...	...	...	..
21592	360000.0	1131.0	3.0	8.0	0.0	2009.0	0.0	47.6993	-122.346
21593	400000.0	5813.0	3.0	8.0	0.0	2014.0	0.0	47.5107	-122.362
21594	402101.0	1350.0	3.0	7.0	0.0	2009.0	0.0	47.5944	-122.299
21595	400000.0	2388.0	3.0	8.0	0.0	2004.0	0.0	47.5345	-122.069
21596	325000.0	1076.0	3.0	7.0	0.0	2008.0	0.0	47.5941	-122.299

16057 rows × 109 columns

In [96]:
```
normzip_df.isna().sum()
```

Out[96]:
```
price            0
sqft_lot         0
condition        0
grade            0
sqft_basement    0
                ..
zipcode_98177    0
zipcode_98178    0
zipcode_98188    0
zipcode_98198    0
zipcode_98199    0
Length: 109, dtype: int64
```

In [97]:
```
y=normzip_df['price']
X=normzip_df.drop(['price'], axis=1)

linreg= LinearRegression()
linreg.fit(X,y)
```

Out[97]:
```
LinearRegression()
```

In [98]:
```
= metrics.r2_score(y,linreg.predict(X))
int(f"{Fore.GREEN} The R squared value for this model after One-Hot-Encodin
```

```
 The R squared value for this model after One-Hot-Encoding the Zipcode is
0.80601
```

WOW!!! Very relevant increase in the R squared value.
Clearly the zipcode contains some important information and has a lot of influence on the price of the houses.

Let us explore a little bit more this concept of the zipcode, which is ultimately hinting at the fact that the location of the house is a heavy factor on the price. We are going to group the data by zipcode and create some plots to visually explore the correlation

In [99]:
```
zipcode_df=df.groupby('zipcode').mean()
zipcode_df.sort_values(by='price', ascending=True, inplace=True)
```

In [100]:
```
zipcode_df.reset_index(inplace=True)
zipcode_df
```

Out[100]:

	zipcode	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	condition
0	98002	262413.368421	3.601504	2.127820	1860.909774	7721.947368	1.447368	3.631579
1	98032	274844.183673	3.551020	1.887755	1916.897959	10820.806122	1.239796	3.622449
2	98168	275974.888889	3.200000	1.638889	1645.333333	12054.338889	1.197222	3.294444
3	98031	305721.819923	3.528736	2.042146	1964.275862	12242.360153	1.463602	3.501916
4	98023	305722.699519	3.512019	2.127404	2133.954327	10660.939904	1.382212	3.370192
...	...	...	...	...	...	...	...	...
65	98112	751201.011494	3.224138	1.994253	1939.252874	3856.298851	1.718391	3.396552
66	98005	751926.896774	3.825806	2.264516	2532.774194	17975.451613	1.238710	3.722581
67	98004	847738.156250	3.468750	1.987500	2020.625000	10869.462500	1.265625	3.625000
68	98040	848716.464286	3.785714	2.267857	2497.833333	12320.797619	1.309524	3.857143
69	98039	937857.142857	3.285714	1.714286	1692.857143	10905.285714	1.142857	3.714286

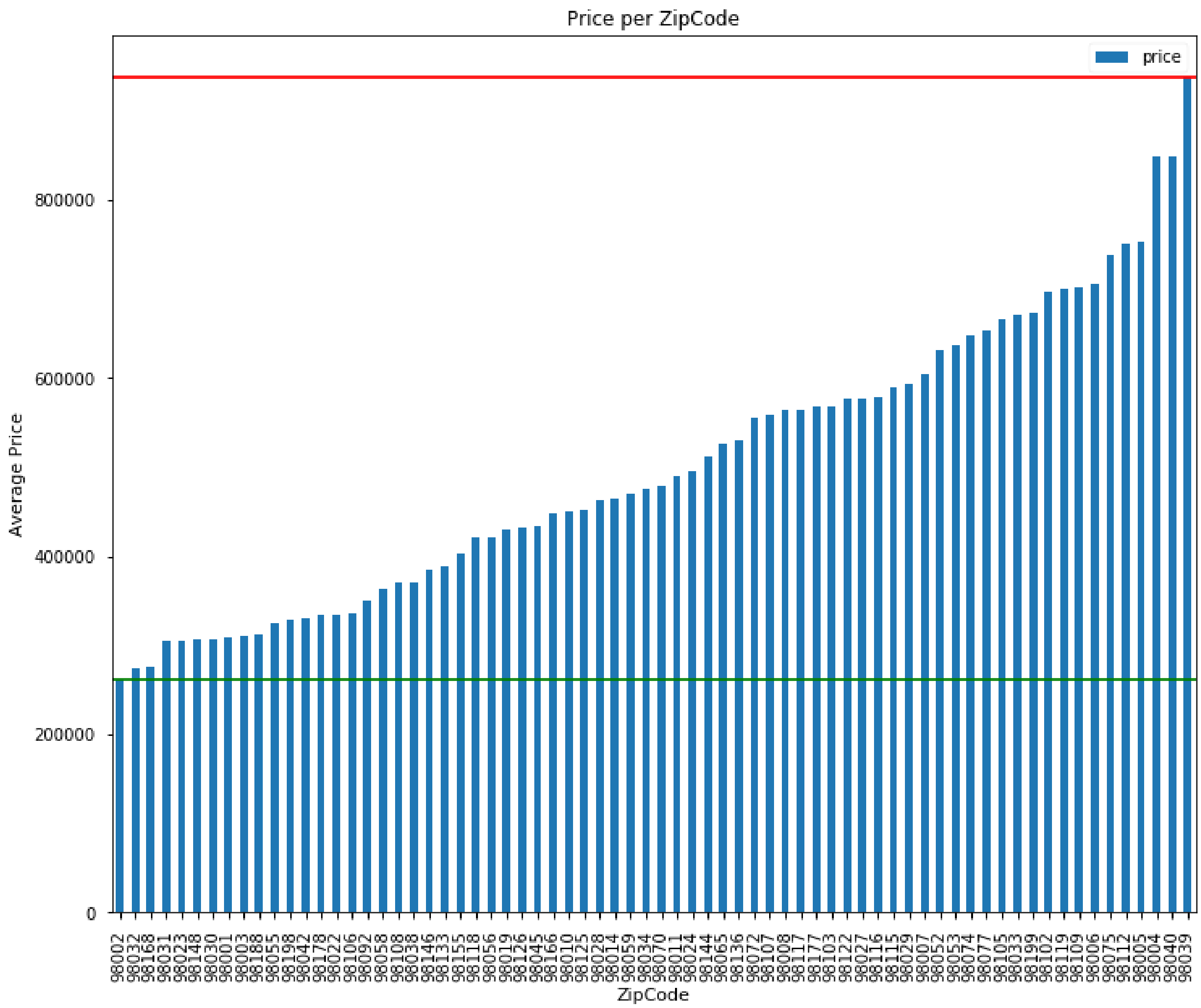
70 rows × 17 columns

In [101]:
```
cheapest=(zipcode_df.iloc[0]['price']).round()
priciest=(zipcode_df.iloc[-1]['price']).round()
print(f"{Fore.BLUE} The  cheapest zipcode on average has the house price of
print(f"{Fore.RED} while the most expansive one is{Style.RESET_ALL}", prici
```

```
 The  cheapest zipcode on average has the house price of  262413.0
 while the most expansive one is 937857.0
```

Let us plot out our results to get a sense also visually of what we found out:

In [102]:
```
ax.set_xticklabels(labels=zipcode_df['zipcode'], rotation=90)
zipcode_df.plot.bar(x='zipcode', y='price',title='Price per ZipCode',\
                        xlabel='ZipCode', ylabel='Average Price', figsize=(12,
plt.axhline(y = cheapest, color = 'g', linestyle = '-')
plt.axhline(y = priciest, color = 'r', linestyle = '-');
```



As we can see there is a very considerable difference in average price of the houses in different zipcodes, ranging from about 230 thousand dollars to around 2 millions. Also choosing carefully the area to buy a house is going to have a great impact on the final pricetag we are going to get.

Seasons

Another variable that we can categorize a little bit better is the date.
It is probably hard for the model to categorize based on all the different dates of the sales, but maybe there is a trend there that we can explore.
Therefore we are going to extract from the sale date the month, and categorize the sales based on their month to see if that also has an influence on the price of the houses.

In [103]:
```
#extracting month from the dates of the sales
months=[]
for i in df['date']:
    dates=datetime.strptime(i, '%m/%d/%Y').date()
    mon=dates.month
    months.append(mon)
```

In [104]:
```
#creating a month column
df['months']=months
df
```

Out[104]:

	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	condition	grade	s
0	10/13/2014	221900.0	3	1.0	1180	5650	1.0	3	7.0	
1	12/9/2014	538000.0	3	2.0	2570	7242	2.0	3	7.0	
3	12/9/2014	604000.0	4	3.0	1960	5000	1.0	5	7.0	
4	2/18/2015	510000.0	3	2.0	1680	8080	1.0	3	8.0	
6	6/27/2014	257500.0	3	2.0	1715	6819	2.0	3	7.0	
...	...	...	...	...	...	...	...	...	...	
21592	5/21/2014	360000.0	3	2.0	1530	1131	3.0	3	8.0	
21593	2/23/2015	400000.0	4	2.0	2310	5813	2.0	3	8.0	
21594	6/23/2014	402101.0	2	1.0	1020	1350	2.0	3	7.0	
21595	1/16/2015	400000.0	3	2.0	1600	2388	2.0	3	8.0	
21596	10/15/2014	325000.0	2	1.0	1020	1076	2.0	3	7.0	

19512 rows × 20 columns

Now we can see the trend of the sales price based on the months of the sale, doing a group by.

In [105]:
```
months_df=df.groupby('months').mean()
months_df.reset_index(inplace=True)
months_df.sort_values(by='months', ascending=True, inplace=True)
months_df
```

Out[105]:

	months	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	condition
0	1	480951.812933	3.406467	2.066975	2035.106236	16949.896074	1.483256	3.360277
1	2	477202.199285	3.346738	2.014298	1985.775693	13917.984808	1.473637	3.394996
2	3	498740.626045	3.362007	1.982079	1979.032258	14971.445639	1.470131	3.350657
3	4	511064.674817	3.352078	2.005868	1989.225428	13378.982885	1.490220	3.358435
4	5	498311.252984	3.344353	2.023875	2019.639118	16376.844812	1.486915	3.415978
5	6	507974.386018	3.391591	2.077001	2066.659574	14224.773556	1.511905	3.457447
6	7	505746.858367	3.394153	2.059476	2081.888609	13798.805948	1.527722	3.442036
7	8	491403.859887	3.349718	2.018644	2027.585876	14415.706215	1.497458	3.438418
8	9	491742.391010	3.355911	2.056650	2035.057266	15616.770320	1.490764	3.431034
9	10	491742.410089	3.351335	2.043917	2028.411276	15309.508605	1.503561	3.405935
10	11	484299.368379	3.358103	2.021344	2010.656126	14815.472727	1.509091	3.397628
11	12	480756.378012	3.379518	2.038404	2037.604669	15582.225904	1.488328	3.399096

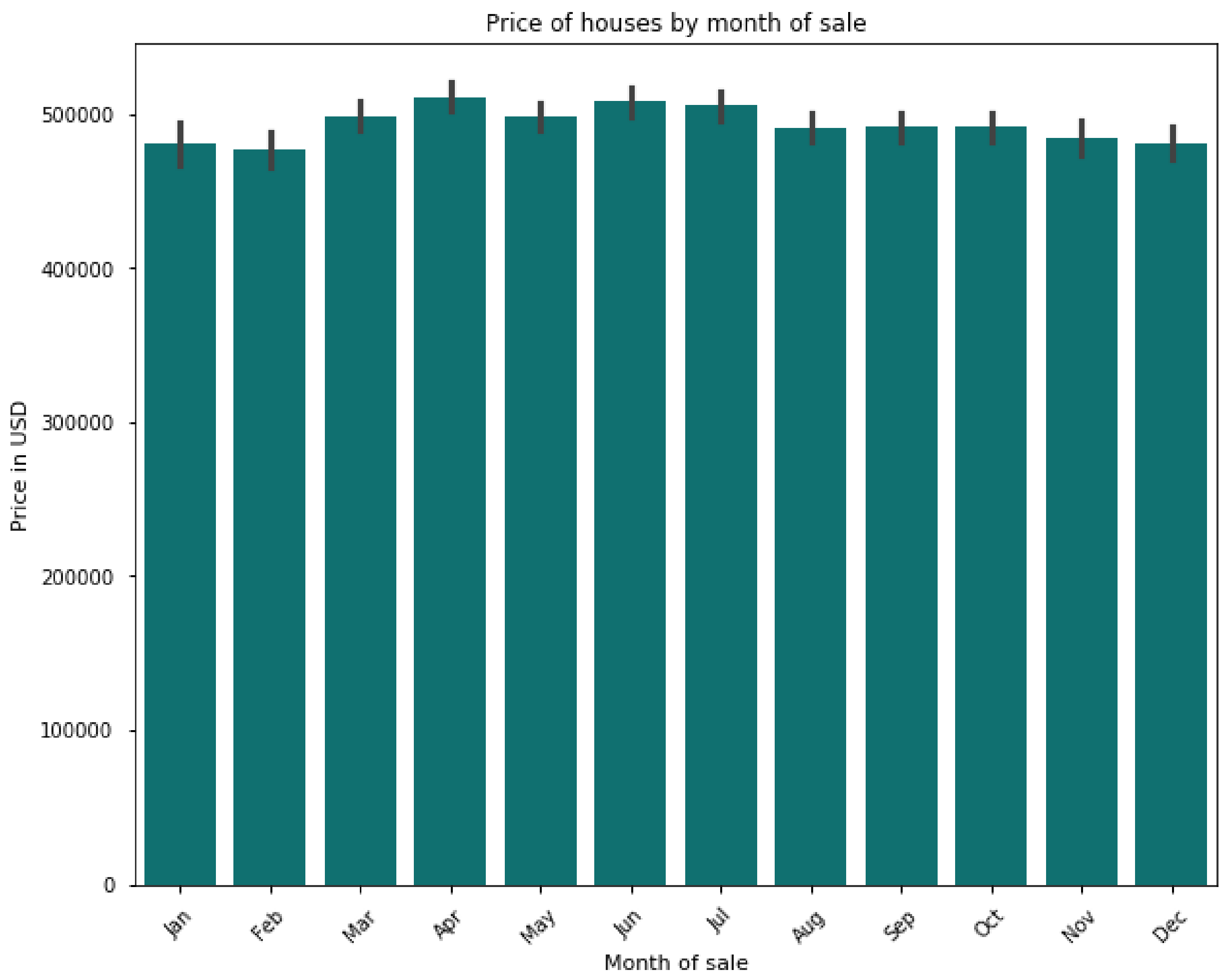
Let us note that the average price of a house sold in the month of February is 477 K while the one for a house sold in foo April is 511 K.

In [106]:
```
#Turning the months from numbers into abbreviations on the month names
months_df['months_names'] = months_df['months'].apply(lambda x: calendar.mo
```

In [107]:
```
months_df['seasons']=months_df['months']
```

In [108]:
```
df.reset_index(inplace=True)
df.sort_values(by='months', ascending=True, inplace=True)
```

In [109]:
```
fig, ax = plt.subplots(figsize=(10, 8))
sns.barplot(data=df,x='months', y='price', color='teal') #or palette='rocke
ax.set_xticklabels(labels=months_df['months_names'], rotation=45)
ax.set_title('Price of houses by month of sale')
ax.set_xlabel('Month of sale')
ax.set_ylabel('Price in USD')
plt.show()
```



As we can see there are some months like January February and December in which the houses tend to sell for less, on average, and other months like April May and June where the average sales are higher.

Now we are going to do a one hot encoding based on the months, since we saw it is a very effective way to include categorical values to the model.

In [110]:
```
origin_series = pd.Series(df['months'])
cat_origin = origin_series.astype('category')
monthsohe_df=pd.get_dummies(cat_origin)
```

Now we can merge the two data frames to run one more linear regression model

In [111]:
```
# Putting the two dataframes together, so we have normalized, zipcodes and
nzs_df = pd.concat([normzip_df, monthsohe_df], axis=1)
nzs_df.dropna(inplace=True)
nzs_df.head()
```

Out[111]:

	price	sqft_lot	condition	grade	sqft_basement	yr_built	yr_renovated	lat	long	sq
0	221900.0	5650.0	3.0	7.0	0.0	1955.0	0.0	47.5112	-122.257	
1	538000.0	7242.0	3.0	7.0	400.0	1951.0	1991.0	47.7210	-122.319	
3	604000.0	5000.0	5.0	7.0	910.0	1965.0	0.0	47.5208	-122.393	
4	510000.0	8080.0	3.0	8.0	0.0	1987.0	0.0	47.6168	-122.045	
6	257500.0	6819.0	3.0	7.0	0.0	1995.0	0.0	47.3097	-122.327	

5 rows × 121 columns

In [112]:
```
y=nzs_df['price']
X=nzs_df.drop(['price'], axis=1)

linreg= LinearRegression()
linreg.fit(X,y)
```

Out[112]:
```
LinearRegression()
```

In [113]:
```
R2= metrics.r2_score(y,linreg.predict(X))
print(f"{Fore.BLUE} The R squared value after One-Hot-Encoding the months i
```

```
 The R squared value after One-Hot-Encoding the months is  0.80453
```

The R squared value decreased a little. So actually encoding this information is not useful for our model, but we will keep in mind our considerations about the months of sale. But will keep pur best performing model, after One Hot Encoding the ZipCode. Now on this finalized model we are going to perform a cross validation and run more serious statistics.

Cross Validation

Cross validation is one of the methods we can do to evaluate the goodness of our model.

With cross validation a section of the data gets used to train the linear model on, and another section is used to test the model. This is repeated for several times, to reduce the amount of errore due to randomness of the selection of the data.

In [114]:
```
# So let us define one more time X and y like we did in our last - best per
y=normzip_df['price']
X=normzip_df.drop(['price'], axis=1)

results=cross_validate(linreg, X, y, cv=10, return_train_score=True, scorin
```

This test produces as many results as the splits we specified, which in this case is cv=10.
So to get the final result we will calculate the average of the several results.

In [115]:
```
test_R2=results['test_r2'].mean()
train_R2=results['train_r2'].mean()
test_MSRE=np.sqrt(-results['test_neg_mean_squared_error'].mean())
train_MSRE=np.sqrt(-results['train_neg_mean_squared_error'].mean())
print("train MSRE:",train_MSRE, "\ntest MSRE:", test_MSRE)
print("train R2:",train_R2, "\ntest R2:", test_R2)
```

```
train MSRE: 90477.4237704816
test MSRE: 91539.7559736034
train R2: 0.8062172460203941
test R2: 0.8008242898610437
```

As we can see the MSRE is pretty high, but not too bad considering that the unit is in dollars like price, so it means that when we are trying to make a prediction on price of the house we could be off of about $90.000.
On the other hand the R2 results are very good and the test performed almost as well as the train, which tells us that we are not overfitting.

To have a concrete proof that all of our work on the data has been useful we are going to run a model with the raw data, just a few one hot encoding to be able to compare the two and see the fruit of our work.

In [116]:
```
y=ohe_df['price']
X=ohe_df.drop(['price','date',], axis=1)

results_raw=cross_validate(linreg, X, y, cv=10, return_train_score=True, \
                           scoring=["r2", "neg_mean_squared_error"])
```

In [117]:
```
test_R2=results_raw['test_r2'].mean()
train_R2=results_raw['train_r2'].mean()
test_MSRE=np.sqrt(-results_raw['test_neg_mean_squared_error'].mean())
train_MSRE=np.sqrt(-results_raw['train_neg_mean_squared_error'].mean())
print("Raw Data train MSRE :",train_MSRE, "\ntest MSRE:", test_MSRE)
print("Raw Data train R2:",train_R2, "\ntest R2:", test_R2)
```

```
Raw Data train MSRE : 118661.73344041024
test MSRE: 119376.49785175962
Raw Data train R2: 0.6661842185304221
test R2: 0.660788455949015
```

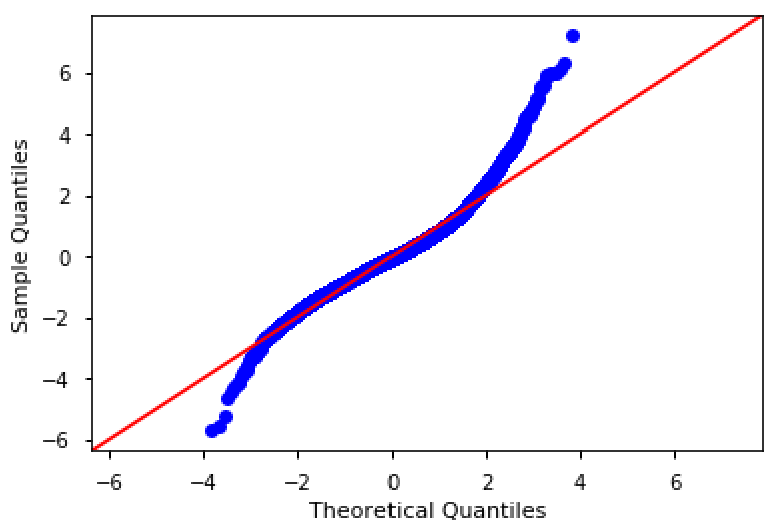
With all our manipulations on the data we were able to increase R2 by almost 15%, and brought down the MSRE by roughly 30 thousand dollars!

Checking for normality

Next we are going to inspect the residues from our test to check for normality.
One of the requirements that we mentioned before is the normal distribution of residues from our model, where the residues are the difference between the expected value and the real value.
We check this with a QQ plot.

In [118]:
```
y=normzip_df['price']
y=list(y)
X=normzip_df.drop(['price'], axis=1)
X = sm.add_constant(X)
model = sm.OLS(y, X).fit()
```

In [119]:
```
residuals=model.resid
fig = sm.graphics.qqplot(residuals, dist=stats.norm, line='45', fit=True)
```



In [120]:
```
y_hat = model.predict(X)
```

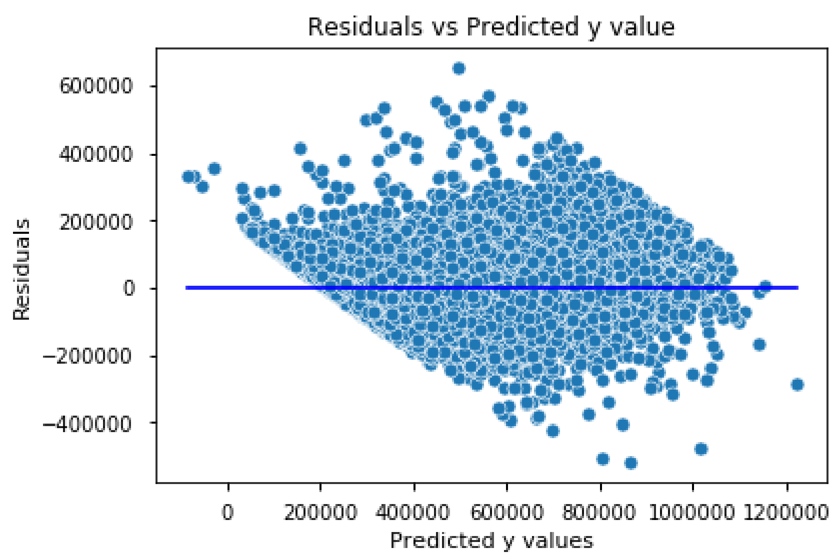
This QQ plot shows us that our sample has pretty heavy tails. But at the same time the central part seems to follow a straight line, there isn't strong evidence to think of a non linear relationship but we will verify that later on.

Checking for Homoscedasticity

Homoscedasticity indicates that a dependent variable's variability is equal across values of the independent variables. What we are looking to see is if the residuals (which again is the difference between the real value and the expected value) are equally distributed across the regression line or if they increase with the regression line, or if there is some sort of pattern. What we want ideally is for them to be equally randomly distributed along the x axis, without any particular pattern.

In [121]:
```
p = sns.scatterplot(y_hat,residuals)
plt.xlabel('Predicted y values')
plt.ylabel('Residuals')
p = sns.lineplot([y_hat.min(),y_hat.max()],[0,0],color='blue')
p = plt.title('Residuals vs Predicted y value')
```

```
/Users/vi/opt/anaconda3/envs/learn-env/lib/python3.6/site-packages/seabor
n/_decorators.py:43: FutureWarning: Pass the following variables as keywo
rd args: x, y. From version 0.12, the only valid positional argument will
be `data`, and passing other arguments without an explicit keyword will r
esult in an error or misinterpretation.
  FutureWarning
/Users/vi/opt/anaconda3/envs/learn-env/lib/python3.6/site-packages/seabor
n/_decorators.py:43: FutureWarning: Pass the following variables as keywo
rd args: x, y. From version 0.12, the only valid positional argument will
be `data`, and passing other arguments without an explicit keyword will r
esult in an error or misinterpretation.
  FutureWarning
```



There seems to be some sort of pattern here, so this would need further studying but overall it is an acceptable result in terms of the distribution of the residues.

Train-Test split instead of Cross Validation

Another way that we can run our model is with a train-test split. It's the same logic as the cross validation but instead of repeating it with different splits I do it only one time.
It is less reliable because doing only one split makes the test more subject to randomness, but there are things that we can study with this setting like the mean squared error and root mean squared error.

In [122]:
```
y=normzip_df['price']
X=normzip_df.drop(['price'], axis=1)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=19)
```

In [123]:
```
linreg = LinearRegression()
linreg.fit(X_train, y_train)

y_hat_train = np.array(linreg.predict(X_train))
y_hat_test = np.array(linreg.predict(X_test))
```

In [124]:
```
y_train=np.array(y_train)
y_test=np.array(y_test)
train_residuals = y_hat_train - y_train
test_residuals = y_hat_test - y_test
```

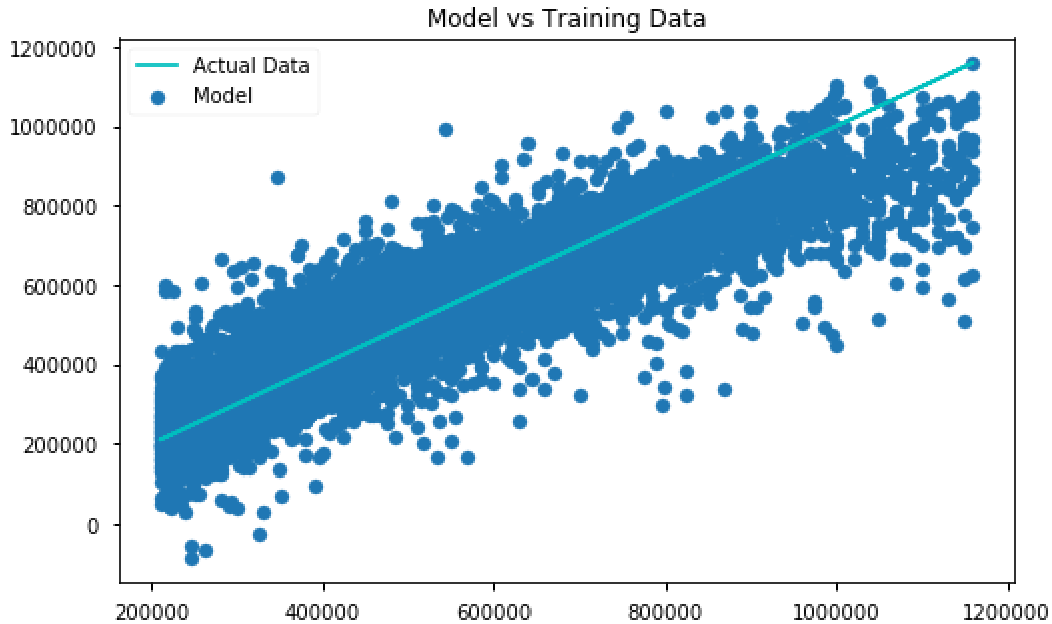
In [125]:
```
train_mse = np.sum((y_train-y_hat_train)**2)/len(y_train)
test_mse = np.sum((y_test-y_hat_test)**2)/len(y_test)
train_msre=np.sqrt(train_mse)
test_msre=np.sqrt(test_mse)
R2train= metrics.r2_score(y_train,linreg.predict(X_train))
R2test= metrics.r2_score(y_test,linreg.predict(X_test))

print('Train Root Mean Squared Error:', train_msre)
print('Train R squared value:', R2train)
print('Test Root Mean Squared Error:', test_msre)
print('Test R squared value:', R2test)
```

```
Train Root Mean Squared Error: 91345.31719765035
Train R squared value: 0.804295603536762
Test Root Mean Squared Error: 88378.10912550481
Test R squared value: 0.8097102215361878
```

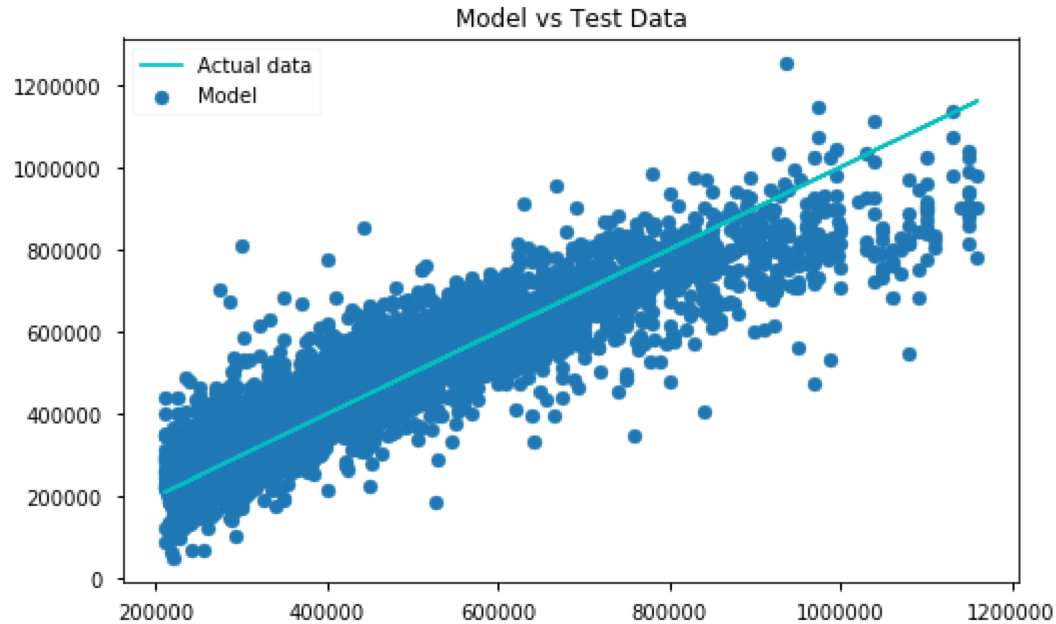
Let us take a look at how our the train sample of our model fits compared to the actual data:

In [126]:
```
plt.figure(figsize=(8, 5))
plt.scatter(y_train, y_hat_train, label='Model')
plt.plot(y_train, y_train, label='Actual Data', color='c')
plt.title('Model vs Training Data')
plt.legend();
```



This graph gives us a sense of what is the spread of the points predicted by my model compared to the actual data. The vertical distance between the points and the line denote the errors.
Next we will generate the same plot for the test data vs the model.

In [127]:
```
plt.figure(figsize=(8, 5))
plt.scatter(y_test, y_hat_test, label='Model')
plt.plot(y_train, y_train, label='Actual data', color='c')
plt.title('Model vs Test Data')
plt.legend();
```



In both cases it seems that most of the model data are pretty close to the actual data.

Polynomial Regression

Another thing that we can do to improve our model to fit our data is to include higher degree terms, using a polynomial fit. This includes products between the different independent variables, including higher powers of the single variables, up until a power that is set.

Third order Polynomial regression

In [128]:
```
df.isna().sum()
```

Out[128]:
```
index               0
date                0
price               0
bedrooms            0
bathrooms           0
sqft_living         0
sqft_lot            0
floors              0
condition           0
grade               0
sqft_above          0
sqft_basement       0
yr_built            0
yr_renovated     3455
zipcode             0
lat                 0
long                0
sqft_living15       0
sqft_lot15          0
reno                0
months              0
dtype: int64
```

In [129]:
```
pf = PolynomialFeatures(degree=3) # degree is the highest exponent in the p

y=df['price']
X=df.drop(['index','price', 'date', 'yr_renovated'], axis=1)

# Fitting the PolynomialFeatures object
pf.fit(X)
```

Out[129]:
```
PolynomialFeatures(degree=3)
```

In [130]:
```
pdf = pd.DataFrame(pf.transform(X), columns=pf.get_feature_names())
pdf
```

Out[130]:

	1	x0	x1	x2	x3	x4	x5	x6	x7	x8	...	x14^3	x14^2 x1
0	1.0	3.0	2.0	1250.0	1033.0	3.0	3.0	8.0	1250.0	0.0	...	1.520875e+09	0.000000e+0
1	1.0	3.0	1.0	1670.0	5200.0	1.0	5.0	7.0	1030.0	640.0	...	3.002246e+11	0.000000e+0
2	1.0	5.0	4.0	3500.0	101494.0	1.5	3.0	8.0	3500.0	0.0	...	5.767352e+13	1.492740e+0
3	1.0	3.0	2.0	2700.0	5040.0	1.0	3.0	8.0	1560.0	1140.0	...	1.250000e+11	0.000000e+0
4	1.0	3.0	2.0	1770.0	7667.0	1.0	3.0	8.0	1270.0	500.0	...	5.320317e+11	0.000000e+0
...	...	...	...	...	...	...	...	...	...	...	...	...	..
19507	1.0	3.0	2.0	1900.0	7076.0	1.0	3.0	7.0	1130.0	770.0	...	1.034285e+12	0.000000e+0
19508	1.0	4.0	2.0	2120.0	5293.0	2.0	3.0	7.0	2120.0	0.0	...	1.499752e+11	2.822797e+0
19509	1.0	3.0	2.0	1620.0	6415.0	2.0	4.0	7.0	1620.0	0.0	...	3.815514e+11	0.000000e+0
19510	1.0	3.0	2.0	2930.0	19900.0	1.5	3.0	9.0	2930.0	0.0	...	8.605043e+12	0.000000e+0
19511	1.0	2.0	1.0	770.0	7200.0	1.0	3.0	7.0	770.0	0.0	...	3.638414e+11	0.000000e+0

19512 rows × 1140 columns

Now fitting the linear regression model:

In [131]:
```
lr = LinearRegression()

lr.fit(pdf, y)
```

Out[131]:
```
LinearRegression()
```

In [132]:
```
lr.score(pdf, y)
```

Out[132]:
```
0.7642137995121101
```

This is a pretty good R squared, but what usually happens with the polynomial regression is that we are fitting the model on the train set so well, that we are actually overfitting and it will perform very poorly on the test set, because we are picking up not only the actual real trends of the relationship between the variables, but also some random noise, given by the random data. Let us see this in practice.

Train test split with poly to show the overfitting.

In [133]:
```
X_train, X_test, y_train, y_test = train_test_split(pdf, y,test_size=0.2,ra
lr_poly = LinearRegression()

# Always fit on the training set
lr_poly.fit(X_train, y_train)

train_R=lr_poly.score(X_train, y_train)
print("Train R squared:", train_R)
```

```
Train R squared: 0.7211632104593759
```

In [134]:
```
test_R=lr_poly.score(X_test, y_test)
print("Test R squared:", test_R)
```

```
Test R squared: -0.9352133941119369
```

The fact that R squared on the test is so bad shows us that we fitted very well the train, in fact too well picking up random noise. This model is not good because even if it can explain very well the train it cannot generalize to a randomly chosen test set.

Second Order Polynomial

What is usually recommended in this situation is to lower the order of the polynomial, which will reduce the number of variables, and overall give us a more simple curve for our model. Let's see what R squared will look like and if we can avoid the overfitting with a second degree polynomial.

In [135]:
```
pf = PolynomialFeatures(degree=2)

y=df['price']
X=df.drop(['price', 'date','yr_renovated' ], axis=1)

pf.fit(X)
```

Out[135]:
```
PolynomialFeatures()
```

In [136]:
```
pdf2 = pd.DataFrame(pf.transform(X), columns=pf.get_feature_names())
pdf2
```

Out[136]:

	1	x0	x1	x2	x3	x4	x5	x6	x7	x8	...	x14^2	x14 x15
0	1.0	20421.0	3.0	2.0	1250.0	1033.0	3.0	3.0	8.0	1250.0	...	1562500.0	1437500.0
1	1.0	2842.0	3.0	1.0	1670.0	5200.0	1.0	5.0	7.0	1030.0	...	2624400.0	10847520.0
2	1.0	6406.0	5.0	4.0	3500.0	101494.0	1.5	3.0	8.0	3500.0	...	10562500.0	125567000.0
3	1.0	8618.0	3.0	2.0	2700.0	5040.0	1.0	3.0	8.0	1560.0	...	9060100.0	15050000.0
4	1.0	5239.0	3.0	2.0	1770.0	7667.0	1.0	3.0	8.0	1270.0	...	4752400.0	17664540.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
19507	1.0	6917.0	3.0	2.0	1900.0	7076.0	1.0	3.0	7.0	1130.0	...	2371600.0	15574020.0
19508	1.0	6942.0	4.0	2.0	2120.0	5293.0	2.0	3.0	7.0	2120.0	...	3960100.0	10572870.0
19509	1.0	6968.0	3.0	2.0	1620.0	6415.0	2.0	4.0	7.0	1620.0	...	2689600.0	11894920.0
19510	1.0	6794.0	3.0	2.0	2930.0	19900.0	1.5	3.0	9.0	2930.0	...	9985600.0	64754720.0
19511	1.0	19479.0	2.0	1.0	770.0	7200.0	1.0	3.0	7.0	770.0	...	1742400.0	9423480.0

19512 rows × 190 columns

In [137]:
```
X_train, X_test, y_train, y_test = train_test_split(pdf2, y, test_size=0.2,
lr_poly = LinearRegression()

lr_poly.fit(X_train, y_train)

R2_train=lr_poly.score(X_train, y_train)
R2_test=lr_poly.score(X_test, y_test)

print("Train R squared:", R2_train)
print("Test R squared:", R2_test)
```

```
Train R squared: 0.7496610927470541
Test R squared: 0.7226804775485179
```

This is a pretty good result with polynomial fit, it doesn't give us the usual overfitting, but the R squared it produces is not really high, it is not giving us anything more than what we got with One Hot Encoding.
But there is something else that we can do, for which the Polynomial regression can help us.

Using Polynomial terms to create new variables

Usually one thing that is suggested to improve the model is trying to come up with new variables that can increase the correlations, by multiplying some of the independent variables we have.
But it is not necessary to try to multiply them at random, and the polynomial regression even

But it is not necessary to try to multiply them at random, and the polynomial regression even though it usually leads to overfitting, in this case can be very useful.
We are going to run again the model with the polynomial regression, and we are going to look at the factors that performed best (since the polynomial features contain also all the possible interactions between the variables, up until the power that we selected).
Then we are going to take only the terms that performed best, that have a higher coefficient, and make a model again fitting with just those, adding them to our best performing model so far.

In [138]:
```
pf = PolynomialFeatures(degree=3)

y=df['price']
X=df.drop(['index','price', 'date', 'yr_renovated'], axis=1)

pf.fit(X)
```

Out[138]:
```
PolynomialFeatures(degree=3)
```

In [139]:
```
y=df['price']
y = list(y)

model = sm.OLS(y, pdf).fit()
model.summary()
```

x13	-0.0528	0.005	-9.726	0.000	-0.063	-0.042
x14	-0.0193	0.002	-9.933	0.000	-0.023	-0.015
x15	0.0309	0.003	9.786	0.000	0.025	0.037
x16	0.0003	0.000	1.381	0.167	-0.000	0.001
x0^2	0.0056	0.001	9.475	0.000	0.004	0.007
x0 x1	-0.0052	0.001	-9.290	0.000	-0.006	-0.004
x0 x2	1.1884	0.119	9.955	0.000	0.954	1.422
x0 x3	6.6349	0.705	9.410	0.000	5.253	8.017
x0 x4	0.0180	0.002	9.786	0.000	0.014	0.022
x0 x5	-0.0091	0.001	-9.535	0.000	-0.011	-0.007
x0 x6	0.0013	0.000	8.895	0.000	0.001	0.002
x0 x7	0.9586	0.097	9.848	0.000	0.768	1.149

In [140]:
```
coefficients=model.params
sort_coef=coefficients.sort_values(ascending=False)
sort_coef[0:5]
```

Out[140]:
```
x4^2 x13      73.540660
x0 x10        54.759949
x5 x10        35.277357
x0 x8 x11     32.656419
x9^2 x11      30.144871
dtype: float64
```

We need to do a little interpretation here, looking at the name of the columns. Let us quickly create a dataframe to help us read into the x values a dataframe to help us read into the x values

In [141]:
```
cols=df.drop(['index','price', 'date', ], axis=1).columns
xlist=['x%d' % i for i in range(0, 18, 1)]
interp=pd.DataFrame(data=zip(cols,xlist),columns=['coe','xvalues'])
interp
```

Out[141]:

	coe	xvalues
0	bedrooms	x0
1	bathrooms	x1
2	sqft_living	x2
3	sqft_lot	x3
4	floors	x4
5	condition	x5
6	grade	x6
7	sqft_above	x7
8	sqft_basement	x8
9	yr_built	x9
10	yr_renovated	x10
11	zipcode	x11
12	lat	x12
13	long	x13
14	sqft_living15	x14
15	sqft_lot15	x15
16	reno	x16
17	months	x17

Perfect, with this we can interpret what are the top 5 coefficients that have the most influence on the price of the houses and include them in our model.

In [142]:
```
best_coefficients=['floors^2*long','bdrms*yr_reno','condition*yr_renovated'
                   'yr_built^2*zipcode']
```

A quick note: these coefficients don't need to make sense from a logical point of view, so we are transitioning from inferential statistic to predictive modeling where our top priority is not understanding the parameters that determin a change in our target and describe the relationship best, but to have a model that can work as well as possible in predicting the price of a house, even if not all the coefficients are logical.
Now let us create new columns with these features, and we will add them to the DataFrame and run the cross validation model again.

In [143]:
```
df_newfeatures=pd.DataFrame()
```

In [144]:
```
df_newfeatures['floors^2*long']=(df['floors']**2)*df['long']
df_newfeatures['bdrms*yr_reno']=(df['bedrooms'])*df['yr_renovated']
df_newfeatures['condition*yr_renovated']=df['condition']*df['yr_renovated']
df_newfeatures['bdrms*sqft_basement*zipcode']=df['bedrooms']*df['sqft_basem
df_newfeatures['yr_built^2*zipcode']=(df['yr_built']**2)*df['zipcode']
```

In [145]:
```
nz_nf_df=pd.concat([normzip_df,df_newfeatures], axis=1)
nz_nf_df.dropna(inplace=True)
```

In [146]:
```
nz_nf_df
```

Out[146]:

	price	sqft_lot	condition	grade	sqft_basement	yr_built	yr_renovated	lat	lon
0	221900.0	5650.0	3.0	7.0	0.0	1955.0	0.0	47.5112	-122.25
1	538000.0	7242.0	3.0	7.0	400.0	1951.0	1991.0	47.7210	-122.31
3	604000.0	5000.0	5.0	7.0	910.0	1965.0	0.0	47.5208	-122.39
4	510000.0	8080.0	3.0	8.0	0.0	1987.0	0.0	47.6168	-122.04
6	257500.0	6819.0	3.0	7.0	0.0	1995.0	0.0	47.3097	-122.32
...	...	...	...	...	...	...	...	...	
19505	425000.0	9680.0	4.0	7.0	0.0	1956.0	0.0	47.6340	-122.12
19506	268000.0	7510.0	5.0	8.0	0.0	1988.0	2013.0	47.2595	-122.21
19509	500000.0	7806.0	4.0	6.0	350.0	1949.0	0.0	47.6859	-122.16
19510	1010000.0	5400.0	5.0	8.0	1000.0	1910.0	0.0	47.6366	-122.36
19511	991500.0	26895.0	3.0	11.0	0.0	1995.0	0.0	47.7253	-122.09

11913 rows × 114 columns

Let us try again to run our cross validation

In [147]:
```
y=nz_nf_df['price']
X=nz_nf_df.drop(['price'], axis=1)

results=cross_validate(linreg,X, y, cv=10, return_train_score=True, scoring
```

In [148]:
```
test_R2=results['test_r2'].mean()
train_R2=results['train_r2'].mean()
test_MSRE=np.sqrt(-results['test_neg_mean_squared_error'].mean())
train_MSRE=np.sqrt(-results['train_neg_mean_squared_error'].mean())
print(" train MSRE:",train_MSRE, "\n test MSRE:", test_MSRE)
print(" train R2:",train_R2, "\n test R2:", test_R2)
```

```
 train MSRE: 90466.76806263774
 test MSRE: 91548.80493847934
 train R2: 0.8049255714067856
 test R2: 0.7999172610649521
```

We can see how in this way including only some of the terms generated by the polynomial fit, we still get a high R squared for the train set but we don't get the very low result for the test, R squared is high ALSO for the test which shows that we are not overfitting in this way.

Ultimately what we tried as a new technique helped improving the fit, is generating new features instead of trying randomly some products of variables, using the poly fit to our aid, but avoiding its usual issue of overfitting.

This technique could be used with however high polynomials we want, wihtout having the overfitting, since we are going to chose only the few best terms that describe the interaction and not all of them.

Predictions on price of the house

With this in depth analysis we were able to create different models, with polynomials and not, that interepret well the results of our data set, with an R squared as high as 80%.
Plugging into these models the information we have from a user about the area where they would like to buy a house, number of bedrooms bathrooms and floors, we should be able to predict the expected price of the house with an error of roughly 90K.
The features that we saw influence more heavily the price are: the area (in terms of zipcode), whether the house was renovated or not, the number of bathrooms, the sqft living area and the month in which the house was sold.
With these conclusions in mind let us see what concrete recommendations we can give to our users to be able to make the best choice for their dream home, in relation to their budget.

Recommendations:

Given everything that we have seen in this study, there are some concrete business recommendations that we can give to the users of "Don't Roam Buy a Home" to make the best educated choice in purchasing their home:

- Scout the areas which have houses in your price range. Depending on the zipcode the average sale price for a house can go from 260 thousand dollars to almost a million.
- Save some of your budget for renovations. In particular we recommend buying a house in worse condition but with more squarefootage and then improving it - renovations turned out to be a very important factor in the price of a house, and while adding a bathroom can cost as

little as 2500 dollars ([link (https://homeguide.com/costs/cost-to-add-a-bathroom)](https://homeguide.com/costs/cost-to-add-a-bathroom)) buying a house with an extra bathroom will increase your price by roughly 50K.
- Try to buy during low season, months like December January and February have the lowest average price sales, while in the spring and summer houses tend to sell for more.
- Keep an eye on number of bathrooms and squarefootage of the house, as factors that will influence the price of the property.

Next Steps:

One factor that turned out to be crucial is the renovations on the house. We could gather more data about that and offer better advice in terms of how much money to keep aside for renovations and what are the type of improvements that would be most beneficial, both for the user and to also increase the resell value of the house.

We could also do a more in depth study about the areas even within the zipcode, to produce targeted statistics and be able to provide an even more precise recommendation in terms of where to look for a house.

With the information we found, we could create and algorythm that takes into account not only the desired house features, but also the user's monthly salary and available cash for down payment. Inputing also the current interest rate and the usual taxes for their state we can determine what would be the closing costs, the monthly mortgage for the user and therefore the possible range of prices of houses that they can afford.
If they can't afford the house that they like, we could still help them out.
Considering that every month that passes, the user can save up the money that would have gone toward the mortgae to build their savings instead, we can also recommend the user to wait and purchase a house with a bigger down payment (hoping for not too high fluctuations of the market) and in how many months they could afford that house.
With the new technique we found for adding higher degree terms to the equation, without overfitting, we can improve the algorythm indefinitely, especially if we can input more variables, adding always more terms and increasing the grade of the polynomial to reach an ever more complex and precise model.

In [ ]:
```

```