Final Project Submission

Please fill out:

- Student name: Deztany Jackson
- Student pace: self paced
- Scheduled project review date/time: Friday Jan 6, 2023 11AM
- Instructor name: Morgan Jones
- Blog post URL: http://dmvinedata.com/learning-to-learn/

Home Improvement House Predictions

Author: Deztany Jackson

Business Understanding

Real Estate agents in King County, Seattle are evaluating the neighborhoods to encourage current home owners of he benefits of improving and upgrading their property value. Housing data from King County was used to develop linear regressions models to support future price prediction.

As a new data scientist I want to build up my clientele. I am working with the real estate agents in the area to build my netowork as they increase theirs.

The primary stakeholders are real estate agents because of their wide use cases, network, domain knowledge and their incentive for home owners to increase their property value. They can also use this for getting a jump start on marketing to potential home buyers. The same predictions could be useful for the homeowners, potential buyers and even those in the remodeling and construction business. Because of their connection and real estate agents are able to influence a larger community's property value which as greater impact than convinving individual homeowners. The area attracts new implants from tech jobs. A great number of these people (as singles or families) may be looking to buy or rent.

This model is used as an intial model supporting course predictions. The main attributes used to support model prediciton are: Condition and Grade. The main attributes used to support model creation are:

- 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors',
- 'waterfront', 'yr_built', 'zipcode', 'cond_num', 'grade_num' I would not guarantee the price predicions are 100% accurate, but will be useful to support general predictions.
  The model used accounts for 67% variability. This means there are things that the model still doesn account for in making predictions.

  Phase 1 Project Description, 2022

Data Understanding

This project uses the King County House Sales dataset (from GitHub project repo). This can be found in several locations: Git Hub Data & Kaggle.

The data is in the format of "csv". The initial data used in the modeling will start with 20 of possible attributes. As modeling progressing certain attributes (features) will be processed, transformed and possibly removed.

In [1]:
```
#Imports necessary intial libraries

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

#Allowing for tables to have longer scroll capability
pd.set_option('display.max_rows', 500)
```

The imported data is the entire initial dataset. This will be scoped down before the intial understanding to support the the core modeling needs with the limited time available. Starting with the entire dataset could potentially support a more accurate model with the increase of the attributes to choose from. We will settle for good enough with the dataset we have.

In [2]:
```
#Import of data to explore, make the id the index column
#The full entire dataset.
df = pd.read_csv('data/kc_house_data.csv', index_col=0)
#displaying the intial top five
df.head()
```

Out[2]:

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

In [3]:
```
#Describes the columns(features) and types of the dataset
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21597 entries, 7129300520 to 1523300157
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  object
 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(6), int64(8), object(6)
memory usage: 3.5+ MB
```

The start of the intial dataset used for the modeling. Certain features will be removed before the data understanding and analysis begins. This is an initial scoping to simplifying modeling only, no statstical or other reason.

In [4]:
```
#Ignore these column at a minimum level
#Ignore data, view, sqft_above, sqft_basement, yr_renovated,lat ,long ,sqft_livi

# The new dataset without the extra attributes. Use copy to keep old dataset int
kc_df = df.copy()

#Columns to drop
drop_col = ["date","view", "sqft_above", "sqft_basement", "yr_renovated","lat" ,
kc_df = kc_df.drop(drop_col, axis = 1)
kc_df.head()
```

Out[4]:

id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	condition
7129300520	221900.0	3	1.00	1180	5650	1.0	NaN	Average
6414100192	538000.0	3	2.25	2570	7242	2.0	NO	Average
5631500400	180000.0	2	1.00	770	10000	1.0	NO	Average
2487200875	604000.0	4	3.00	1960	5000	1.0	NO	Very Good
1954400510	510000.0	3	2.00	1680	8080	1.0	NO	Average

Looking at the dataset to see the size, any null values and the data types of the original dataset we will work with

In [5]:
```
#Viewing information about new dataset. Keep an eye on features with null values
kc_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21597 entries, 7129300520 to 1523300157
Data columns (total 11 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   price        21597 non-null  float64
 1   bedrooms     21597 non-null  int64  
 2   bathrooms    21597 non-null  float64
 3   sqft_living  21597 non-null  int64  
 4   sqft_lot     21597 non-null  int64  
 5   floors       21597 non-null  float64
 6   waterfront   19221 non-null  object 
 7   condition    21597 non-null  object 
 8   grade        21597 non-null  object 
 9   yr_built     21597 non-null  int64  
 10  zipcode      21597 non-null  int64  
dtypes: float64(3), int64(5), object(3)
memory usage: 2.0+ MB
```

Using shape to check size of dataset, rows and columns. Column names and amount are a key to track of through data preparation.

In [6]:
```
#Check size of the data set
kc_df.shape
```

Out[6]:
```
(21597, 11)
```

The following features are all numerical and categorical. The describe function only gives out for numerical numbers, not categorical (objects). The features of "yr_built", "bathroom", "bedroom" and "zipcode" are numbers and used to classify and count values. The rest are numerical cardinal values. The names of the features are generally self explanatory and will stay the same.

The features (not pictured below) of "condition" and "grade" picture are categorical strings, but will potential change to numerical values for ease of use.

Ref: Numerical Numbers, Rod Pierce

In [7]:
```
# Numerical descriptions and statistics
kc_df.describe()
```

Out[7]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	2
count	2.159700e+04	21597.000000	21597.000000	21597.000000	2.159700e+04	21597.000000	
mean	5.402966e+05	3.373200	2.115826	2080.321850	1.509941e+04	1.494096	
std	3.673681e+05	0.926299	0.768984	918.106125	4.141264e+04	0.539683	
min	7.800000e+04	1.000000	0.500000	370.000000	5.200000e+02	1.000000	
25%	3.220000e+05	3.000000	1.750000	1430.000000	5.040000e+03	1.000000	
50%	4.500000e+05	3.000000	2.250000	1910.000000	7.618000e+03	1.500000	
75%	6.450000e+05	4.000000	2.500000	2550.000000	1.068500e+04	2.000000	
max	7.700000e+06	33.000000	8.000000	13540.000000	1.651359e+06	3.500000	

Initial Features Used

- **King County Table**
  - Rows: 21597
  - Features: 11
  - "id" is not a feature used in the modeling

Cardinal Numbers

- ***price - (Target Variable)***
  - Description: Price is the amount of the house in context of the current attributes.
  - Type: Float Number
  - Expectation/Comment: Price will be out target variable. We will want to see after developing a solid model the varying of attributes would effect price. The price difference after an "upgrading" the home is also needed.
- **bedrooms**
  - Description: Number of bedrooms for the given home
  - Type: Int Number
  - Expectation/Comment:Will evaluate how well this impacts the model.
- **bathrooms**
  - Description: Number of bathrooms for the given home
  - Type: Int Number
  - Expectation/Comment: It has the second highest correlation value against price. These are.25, .5 and .75 are use in addition to whole numbers. .25-Sink; .5-Sink and Toilet; .75-Sink, Toilet and Shower,/Bath; 1 - Everything
- **sqft_living**
  - Description: The size of the livable space in the house
  - Type: Int Number
  - Expectation/Comment:We will assume larger will amount gather more money.
- **sqft_lot**
  - Description: The size of the lot
  - Type: Int Number
  - Expectation/Comment: We will assume larger will gather more money.
- **floors**
  - Description:'he Number of Floors.
  - Type:float Number
  - Expectation/Comment: We are not using the other attributes that compliment the number of floors.Sometimes one floor could be desirable. It would be hard to understand the house is architected with floors. It could be one floor and a basement, or two floors and no basement.

Nominal Numbers

- **yr_built**
  - Description: Year when house was built
  - Type: Int Number

    - Expectation/Comment: It could give context to the grade and condition. This may be useful for changing variables for prediction purposes
- **zipcode**
    - Description: ZIP Code used by the United States Postal Service
    - Type: Int Number
    - Expectation/Comment: The Zipcode will be there to help add to the business case. It can be helpful in creating collection of houses to focus on.

Categorical Objects

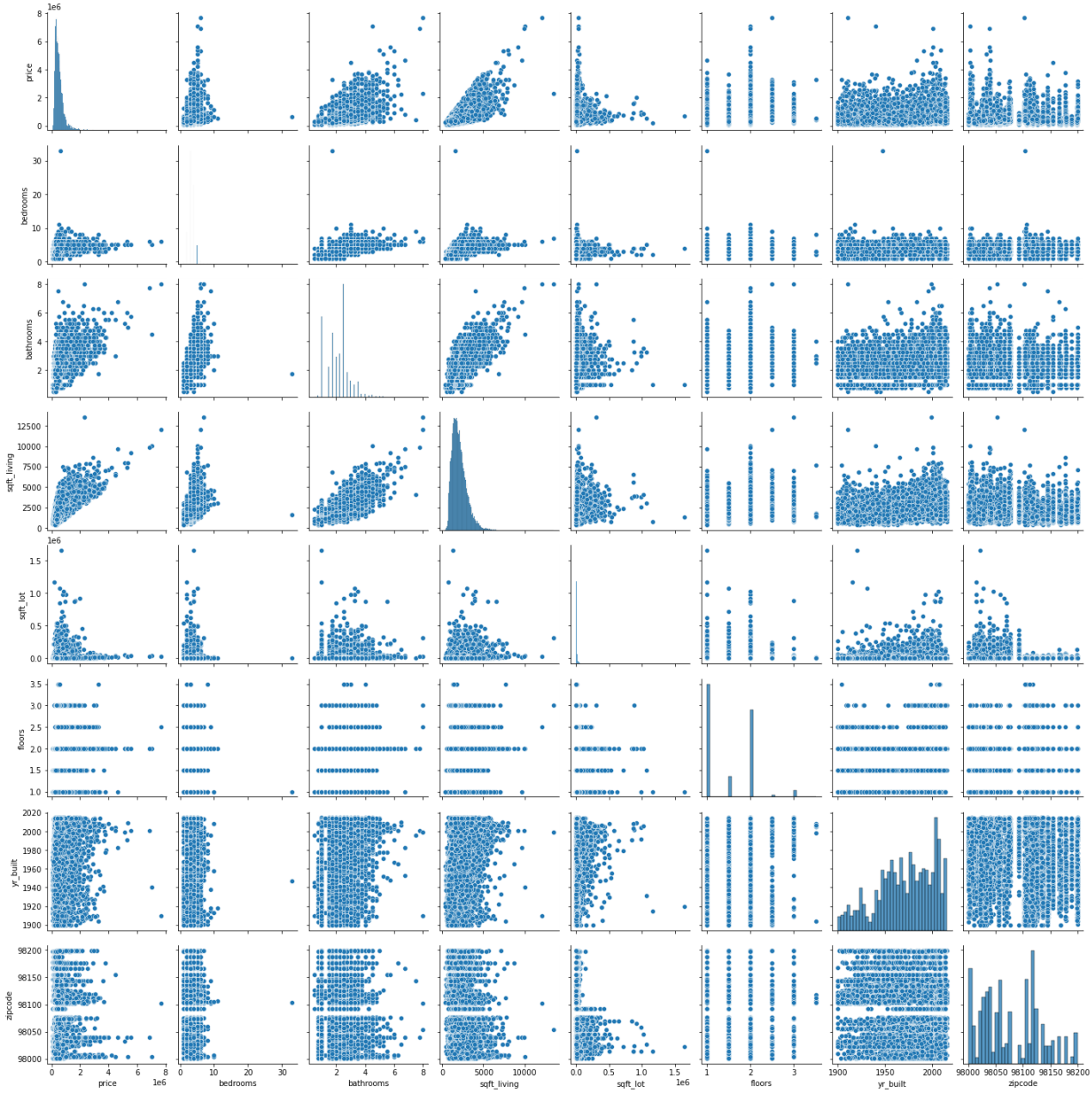
- **waterfront**
    - Description: Whether the house is on a waterfront
    - Type: Object String
    - Expectation/Comment: There isn't any information on what type of water. For those that are Null, we will fill Null with Unknown
- **condition**
    - Description: How good the overall condition of the house is. Related to maintenance of house.
    - Type: Intial Object String (Transformed to Int) - 5 Values
    - Expectation/Comment: This paired with the grade may the the top useful in changing for prediction. Will be transformed into number from string later in model.
- **grade**
    - Description: Overall grade of the house. Related to the construction and design of the house.
    - Type: Intial Object String (Transformed to Float) - 13 Vaues
    - Expectation/Comment: This paired with the condition may the the top useful in changing for prediction. Will be transformed into number from string later in model.

Viewing and counting the values of the categorical features. We can assess what the best route to work with these is. Do we change to ordinal numbers or process this in the future using techniques like One Hot Encoding? For the "condition" and "grade" mapping numbers to the values will be used.

A broad view of the distribution of the features help understand the possible transformation needs of the data. Looking to see if the data is linear and/or normal distributed. The sqft_living looks the most linear against price. Most of the data is skewed and not normally distributed. Log-transformation and normalization should help the continous data during the modeling.

In [8]:
```
#Pair plot for quick view of the datasets distribution and linearity
sns.pairplot(kc_df)
```

Out[8]:
```
<seaborn.axisgrid.PairGrid at 0x7fd324614b20>
```



Data Preparation

Data preparation happens throughout the modeling process in iteration as new information is known. There will be some preparation (e.g.data transformation and scaling) that will happen as we split the training and test data from one another. This is to protect against data leakage.

Looking further into the categorical data, to assess future processing needs.

In [9]:
```
#Viewing the values and their count for a feature
kc_df[["condition"]].value_counts()
```

Out[9]:
```
condition
Average      14020
Good          5677
Very Good     1701
Fair           170
Poor            29
dtype: int64
```

Using the "Column Names" file description and King County Clossary, Building Conditions the string and values mapping was found and used. Even though OHE can be performed on the string values in the future. Having numbers makes it easier for data processing and manipulation (on new data or modified test data) for predicitons.

In [10]:
```
#Creating a dictionary to map to the string values for condition
cond_num = {'Very Good':5, "Good": 4,'Average': 3, "Fair": 2, "Poor": 1}
```

In [11]:
```
#Applying the dictionary to map to the values of the original values
# New column is added
kc_df["cond_num"] = kc_df["condition"].map(cond_num)
```

In [12]:
```
#Checking the information on the new column
kc_df[["cond_num"]].info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21597 entries, 7129300520 to 1523300157
Data columns (total 1 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   cond_num  21597 non-null  int64
dtypes: int64(1)
memory usage: 337.5 KB
```

In [13]:
```
#Checking to see if the value counts of the original and the new column match
kc_df[["cond_num","condition"]].value_counts()
```

Out[13]:
```
cond_num  condition
3         Average      14020
4         Good          5677
5         Very Good     1701
2         Fair           170
1         Poor            29
dtype: int64
```

The original and new column for "condition" have a matching number of values The same process that was done to the "condition" attribute is done to the grade attribute below.

In [14]:
```
#Viewing the values and their count for a feature
kc_df[["grade"]].value_counts()
```

Out[14]:
```
grade
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
dtype: int64
```

In [15]:
```
#Creating a dictionary to map to the string values for grade
grade_num = {'13 Mansion':13, "12 Luxury": 12,'11 Excellent': 11,
             "10 Very Good": 10,"9 Better": 9,"8 Good": 8,
             "7 Average": 7,"6 Low Average": 6, "5 Fair": 5,
             "4 Low": 4, "3 Poor": 3}
```

In [16]:
```
#Applying the dictionary to map to the values of the original values
# New column is added
kc_df["grade_num"] = kc_df["grade"].map(grade_num)
```

In [17]:
```
#Checking to see if the value counts of the original and the new column
kc_df[["grade_num","grade"]].value_counts()
```

Out[17]:
```
grade_num  grade
7          7 Average        8974
8          8 Good           6065
9          9 Better         2615
6          6 Low Average    2038
10         10 Very Good     1134
11         11 Excellent      399
5          5 Fair            242
12         12 Luxury          89
4          4 Low              27
13         13 Mansion         13
3          3 Poor              1
dtype: int64
```

Dropping the original columns of condition and grade. This is essentially duplicate information as the cond_num and grade_num.

In [18]:
```
#drop the original condition and grade columns
kc_df = kc_df.drop(["condition","grade"], axis = 1)
kc_df.head()
```

Out[18]:

id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
7129300520	221900.0	3	1.00	1180	5650	1.0	NaN	1955
6414100192	538000.0	3	2.25	2570	7242	2.0	NO	1951
5631500400	180000.0	2	1.00	770	10000	1.0	NO	1933
2487200875	604000.0	4	3.00	1960	5000	1.0	NO	1965
1954400510	510000.0	3	2.00	1680	8080	1.0	NO	1987

The "waterfront" feature is the only feature that has "NaN". These values will be imputed with a value of "Unknown" than deleted or imputed with "No". The Null values make up over ~10% of the data set. This could impact training of the model adversly because the limited data. Less than 1% of the known homes have a waterfront view "No" may seem like a conservative choice to inpute, but using "Unknown" allows future data exploration if more information becomes available.

In [19]:
```
#Checking for waterfront Null counts
kc_df[["waterfront"]].isna().value_counts()
```

Out[19]:
```
waterfront
False         19221
True           2376
dtype: int64
```

In [20]:
```
#Checking waterfront current value counts
kc_df[["waterfront"]].value_counts()
```

Out[20]:
```
waterfront
NO            19075
YES             146
dtype: int64
```

If waterfront becomes a main feature it is good to understand which home types the Null is associated with. It seems the "average" and "good" grade home are traced to much of the Null values. If they have space for home repair, having a more accurate value of "waterfront" could turn out helpful for real estate agents to market.

In [21]:
```
#Checking to see which house types these Null values affect
kc_df[kc_df["waterfront"].isna()]["grade_num"].value_counts()
```

Out[21]:
```
7     1000
8      656
9      295
6      228
10     116
11      42
5       30
12       7
4        2
Name: grade_num, dtype: int64
```

Usually it is better to fill in "NaN" after train and test split to prevent data leakage. Because filling in unknown as no effect on the other features, decided to do it before.

In [22]:
```
#Replaces "NaN" with "Unk"
kc_df["waterfront"].fillna("Unk", inplace = True)

#Check NaN Count after replacing "NaN"with "Unk"
print("Values: ",kc_df[["waterfront"]].value_counts())

print("Waterfront Null Values:", kc_df[["waterfront"]].isna().sum())
```

```
Values:  waterfront
NO            19075
Unk            2376
YES             146
dtype: int64
Waterfront Null Values: waterfront    0
dtype: int64
```

Remove duplicate values from the dataset. This could be from clerical issues multiple data entries.

In [23]:
```
#Remove the duplicate rows from data set
kc_df = kc_df.drop_duplicates()
print (kc_df.duplicated().sum(),kc_df.value_counts().sum())
```

```
0 21590
```

Training and Testing & Cross Validation Approach

Predicting new home prices comes after training and testing the model. We will split our data set into 80% training set and 20% testing. The train/test split is used for intial model validation.

Kfold cross validation will also be used. This allows the dataset to be split into "train" and "test" and then when the training data is used with cross validation it will be split into "training" and "validation" data.

The target value is the "price" value. This will be set to "y" and the rest of the data will be in "X". This is then used to do the initial train/test split.

We do not want the test data to be trained with the training data. This is data leakage and can distort the training process.

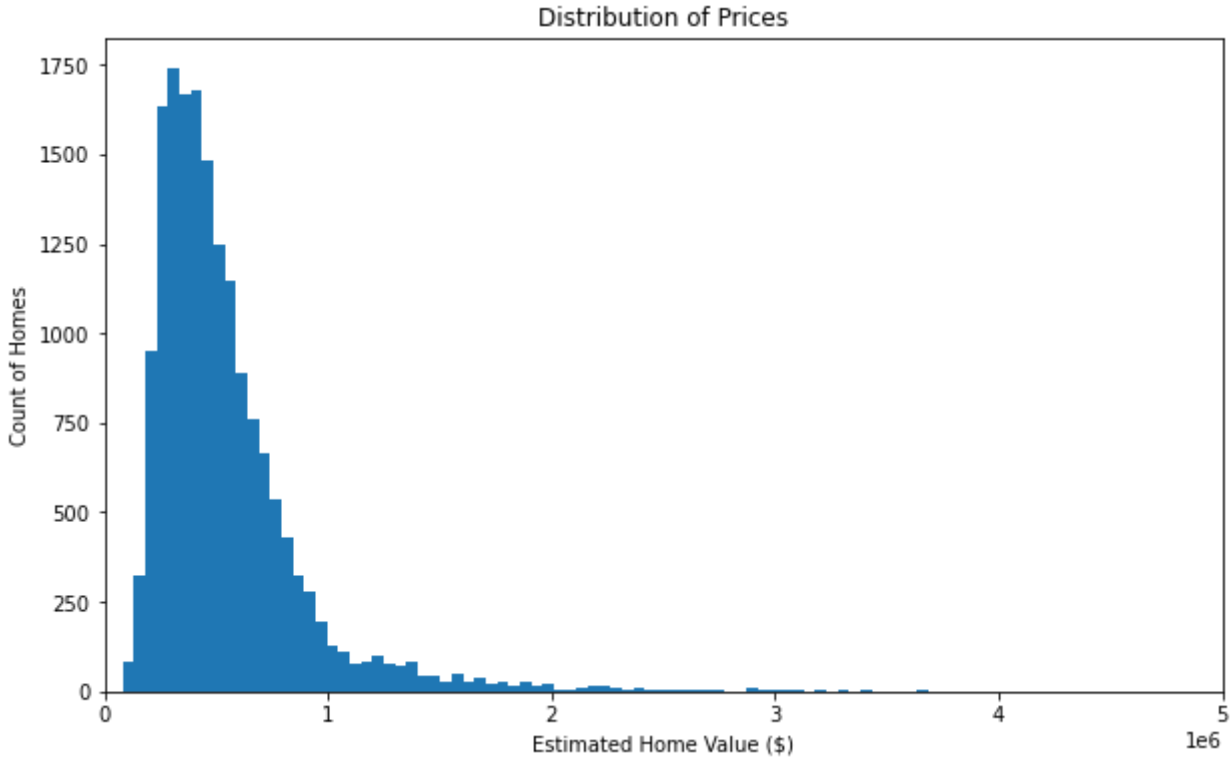
In [24]:
```
# Importing library to split the data into training and test data for model vali
from sklearn.model_selection import train_test_split

#dropping the target variable from the dataset features
X = kc_df.drop("price", axis=1)
# Setting the target variable
y = kc_df["price"]

#Setting the test size .
test_size = .2

#Using the basic train/test split. Adding a random state to product the random s
X_train , X_test, y_train, y_test = train_test_split(X,y, test_size= test_size,
```

In [25]:
```
#Looking at the split's shape
print(X_train.shape , X_test.shape, y_train.shape, y_test.shape)
```

```
(17272, 10) (4318, 10) (17272,) (4318,)
```

Visualization of target variable: Ref: Linear Reg Lab, #20 Most of the housing prices are less than 1 million dollars.

In [26]:
```
#Setting the frame and figsize
fig, ax = plt.subplots(figsize=(10, 6))


#Plotting the training portion of the target variables
ax.hist(y_train, bins=150)

#Setting labels and titles
ax.set_xlabel("Estimated Home Value ($)")
ax.set_ylabel("Count of Homes")
ax.set_title("Distribution of Prices")

#Setting limit on the x axis
plt.xlim([0, 5000000])

plt.savefig("images/Initial_DistPrices_1.png", dpi=99)
```



The target variable in the training data set is positively skewed. A log transformation may support this model effort and normalize the target variables.

In [27]:
```
#Looking at the distribution of the price
y_train.describe()
```

Out[27]:
```
count    1.727200e+04
mean     5.415971e+05
std      3.646503e+05
min      8.000000e+04
25%      3.230000e+05
50%      4.520000e+05
75%      6.450000e+05
max      7.700000e+06
Name: price, dtype: float64
```

In [28]:
```
#Reviewing the y test "price" stats
y_test.describe()
```

Out[28]:
```
count    4.318000e+03
mean     5.351554e+05
std      3.782946e+05
min      7.800000e+04
25%      3.200000e+05
50%      4.450000e+05
75%      6.394875e+05
max      7.060000e+06
Name: price, dtype: float64
```

Reviewing the statistics on the y_train and y_test value for current understanding and future comparison

---

Baseline Modeling (1)

Initial Correlation

The initial linear regression model will be done with the highest correlated feature. This will be considered our baseline model. From here we will do several iterations to see if we can improve the model's performance with different techniques.

Check for the highest correlated values to the target variable "price". Correlation works on numerical values, not categorical ones.

In [29]:
```
#Print out correlation values in dataframe
#This is done with the "price" value in the dataset.
corr = kc_df.corr()
corr
```

Out[29]:

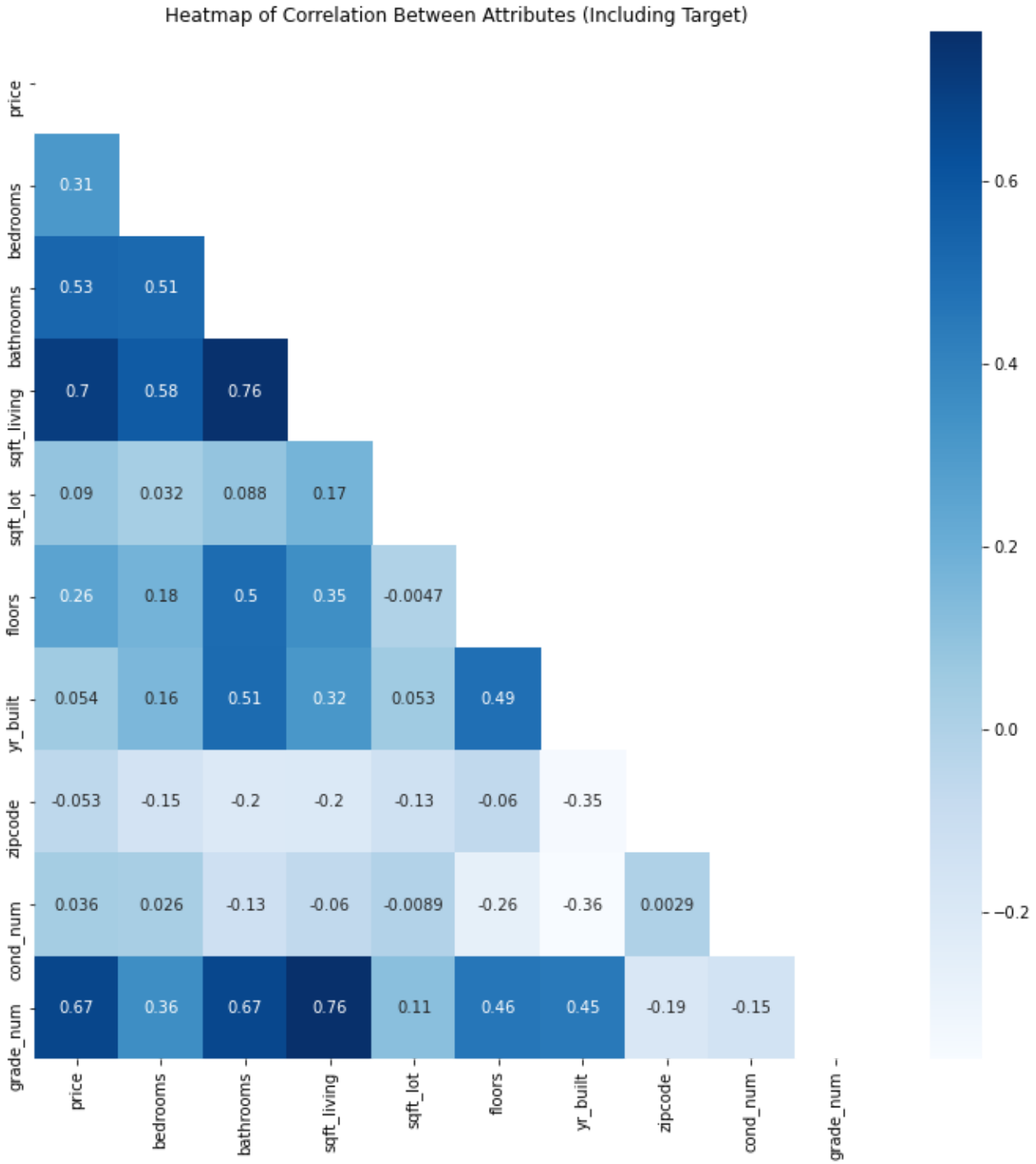
	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	yr_built	zip
price	1.000000	0.308835	0.525933	0.701938	0.089868	0.256948	0.054015	-0.05
bedrooms	0.308835	1.000000	0.514567	0.578211	0.032448	0.178196	0.155831	-0.15
bathrooms	0.525933	0.514567	1.000000	0.755813	0.088396	0.502788	0.507247	-0.20
sqft_living	0.701938	0.578211	0.755813	1.000000	0.173423	0.354350	0.318431	-0.1
sqft_lot	0.089868	0.032448	0.088396	0.173423	1.000000	-0.004664	0.053100	-0.12
floors	0.256948	0.178196	0.502788	0.354350	-0.004664	1.000000	0.488904	-0.0
yr_built	0.054015	0.155831	0.507247	0.318431	0.053100	0.488904	1.000000	-0.34
zipcode	-0.053381	-0.154143	-0.204782	-0.199751	-0.129583	-0.059711	-0.347430	1.00
cond_num	0.036039	0.026450	-0.126446	-0.059526	-0.008894	-0.263915	-0.361447	0.00
grade_num	0.668077	0.356788	0.665881	0.763031	0.114826	0.458705	0.447723	-0.18

The most correlated is the sqft_living with (.7), the least is the zipcode with (-.05). Below this correlation table will be visualized in a heatmap.

Heatmap: Ref: Phase 2, #20 Linear Reg Lab

In [30]:
```
# Set up figure and axes
fig, ax = plt.subplots(figsize=(12,12))

# Plot a heatmap of the correlation matrix, with both
# numbers and colors indicating the correlations
sns.heatmap(
    # Specifies the data to be plotted
    data=corr,
    # The mask means we only show half the values,
    # instead of showing duplicates. It's optional.
    mask=np.triu(np.ones_like(corr, dtype=bool)),
    # Specifies that we should use the existing axes
    ax=ax,
    #Color of the heatmap
    cmap="Blues",
    # Specifies that we want labels, not just colors
    annot=True,

)
```

```
# Customize the plot appearance
ax.set_title("Heatmap of Correlation Between Attributes (Including Target)");

plt.savefig("images/Int_Corr_2.png", dpi=99)
```

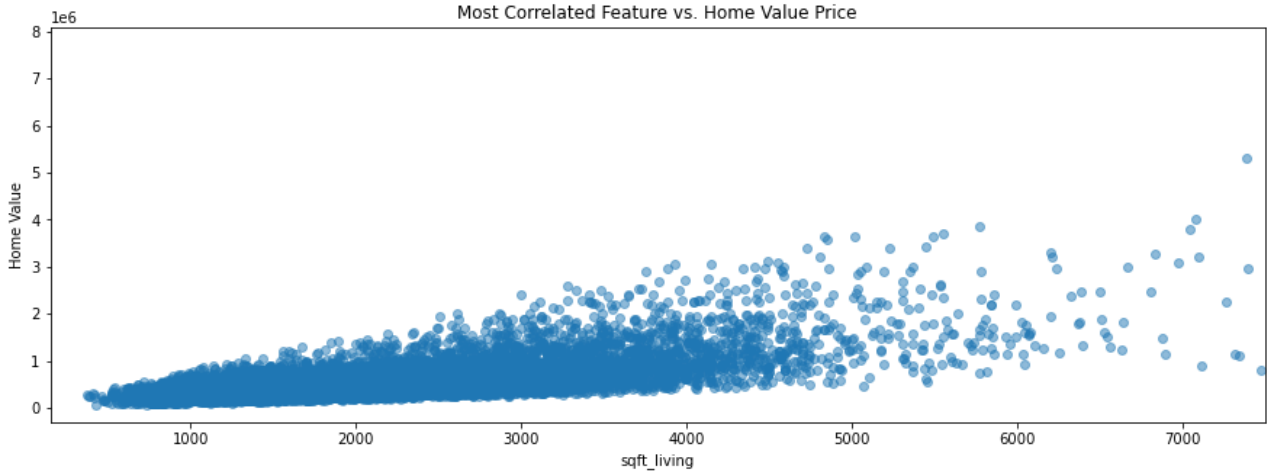


The "sqft_living" feature has the highest correlation of .7 with the "price". The "grad_num" feature is the second highest correlated with the "price". This is good to know because this will be one of the attributes changed during predictions. The zipcode" feature has a negative correlation value. "Bathrooms" and "sqft_living are highly correlated as well. These correlations are the only ones above .7. This dataset doesn't have a really high correlation with the "price" feature or with each other.

Below we will plot the most correlated feature against price. This will let us see if this distribution is linear for our baseline model.

In [31]:
```
#Strongest numerical feature from the heatmap
most_correlated_feature = "sqft_living"
```

In [32]:
```
#Plot a scatter plot "Most Correlated Feature vs. Home Value"
fig, ax = plt.subplots(figsize = (15,5))

ax.scatter(X_train[most_correlated_feature], y_train, alpha=0.5)
ax.set_xlabel(most_correlated_feature)
ax.set_ylabel("Home Value")
ax.set_title("Most Correlated Feature vs. Home Value Price");
plt.xlim([150, 7500])
plt.savefig("images/Linear_Price_3.png", dpi=99)
```



The graph shows a fairly linear data plot. The larger the sqft gets with the increase of price the more outliers arise.

Baseline Model - Basic Train/Test Split

Modeling the baseline feature using the basic train/test split first then a kfold validation method.

Modeling using linear regression approach. This looks at the relationship between dependent (y) and independent (x) values.

- The "fit" method learns something about the data.
- The "transform" method uses what it learned to transform the data.

The train and test data are separated but, the same process needs to be applied on each.

- During processing, fitting the data is done on the training data to learn. It is applied on the testing data using transform.

In [33]:
```
#Reshaping from a 1D series to a multidimensional array that the transformers su

#Reshape training data to support modeling without error
X_train[most_correlated_feature]
X_array = np.array(X_train[most_correlated_feature])
newarr = X_array.reshape(-1,1)
```

```
#Reshape the test data to support modeling without error for single arrays
X_test[most_correlated_feature]
X_array_test = np.array(X_test[most_correlated_feature])
newarr_test = X_array_test.reshape(-1,1)
```

In [34]:
```
#Showing the data prior and post reshaping
print(X_train[most_correlated_feature].shape,newarr.shape)
```

```
(17272,) (17272, 1)
```

In [35]:
```
#Importing Linear Regression library
from sklearn.linear_model import LinearRegression

#Initializing model
linreg = LinearRegression()
#Fitting training data
linreg.fit(newarr, y_train)
linreg.fit(newarr_test, y_test)
LinearRegression()
```

Out[35]:
```
LinearRegression()
```

After we train the model. The train and test data is applied (.predict) to the trained linear regression model to predict the estimated y value ("price"). From here a R2 score and RMSE are calculated. The R2 allows us to see how well the test and train model are compared to each other and how much of the variation the model covers. The higher the R2 score the better. We would like the test R2 score to be higher than the trained model.

RMSE is used to measure distance between predicted and actual values. It measures how well the model predicts. The lower value the better (closer to zero is best). Usually the training data is higher than the test data, a lower test value supports the accuracy of the model.

- RMSE Kaggle, 2020
- David Dalpiaz, 2020

In [36]:
```
#Importing library to look at errors
from sklearn.metrics import mean_squared_error

#Y price Predictions for training and testing features
y_hat_train = linreg.predict(newarr)
y_hat_test = linreg.predict(newarr_test)

#Root Mean Square Error
train_rmse = mean_squared_error(y_train, y_hat_train, squared = False)
test_rmse = mean_squared_error(y_test, y_hat_test, squared = False)

#r2 Score
Model_train_score = linreg.score(newarr,y_train)
Model_score = linreg.score(newarr_test,y_test)

print("Baseline Train/Test:")
print('Train RMSE ', train_rmse)
print('Test RMSE: ', test_rmse)
print()
print('Train Model Score: ', Model_train_score)
print('Test Model Score: ', Model_score)
```

```
Baseline Train/Test:
Train RMSE  260172.0361161922
Test RMSE:  268864.35998011974

Train Model Score:  0.49091149233831743
Test Model Score:  0.494749423259338
```

The test RMSE is currently high and worse than the training Error. They both are far from 0. The model will not generalize well for future test data. The reason is unclear,it may be irreducible noise. Will look into applying log transforming to certain continous attributes in the dataset to see. The base model R2 scores are both round only .49, the test R2 is higher than the trained data. This base model isn't strong enough to support predicting prices. This tells us how we the model is at predicting the variance in dataset. There may be better features or a combination of a few that hopefully will increase the score.

Baseline Model- Kfold Validation

The original dataset is split into "train" and "test". We are applying the kfold validation to our training set which splits the training dataset into "train and "validation". The amount of splits is set in the cv. The Kfold score will be a mean of the various splits.

In [37]:
```
#Importing cross validate and shuffle split.
#Shuffle split creates different values for splits and test size
from sklearn.model_selection import cross_validate, ShuffleSplit

#Splitting the dataset twith a .3 test size
splitter = ShuffleSplit(n_splits=3, test_size=0.3, random_state=42)

#Initalize as used estimator
baseline_model = LinearRegression(normalize= True)

#Baseline scores using kfold cross validation
#Most correlated feature - sqft_living
baseline_scores = cross_validate(
    estimator=baseline_model,
    X=X_train[[most_correlated_feature]],
    y=y_train,
    return_train_score=True,
    cv=splitter
)


#Mean of the train and validation scores
print("Baseline Models:")
print("Current Kfold Train mean score:     ", baseline_scores["train_score"].mea
print("Current Kfold Validation mean score:", baseline_scores["test_score"].mean
print()
#Training and validation Scores
print('Previous Train/Test Split Train Model Score: ' ,Model_train_score)
print('Previous Train/Test Split Validation Model Score: ', Model_score)
```

```
Baseline Models:
Current Kfold Train mean score:      0.4884772214299433
Current Kfold Validation mean score: 0.5000072841051805

Previous Train/Test Split Train Model Score:  0.49091149233831743
Previous Train/Test Split Validation Model Score:  0.494749423259338
```

The Kfold method offers a higher validation mean score (.50) than the previous Train/Test score. Both offer a higher score than their associated training score. The kfold validation will be used in the future iterations of the model training.

Second Model with Categories and Numerical Features (2)

We will improve the baseline model by adding more features training more features(numerical and categorical). Additional features should support an increase a R2 score because it will help describe the dataset more. More processing of the data will occur across the training and testing data to support an improved model.

In [38]:
```
#Viewing the current columns in the dataset to use
X_train.columns
```

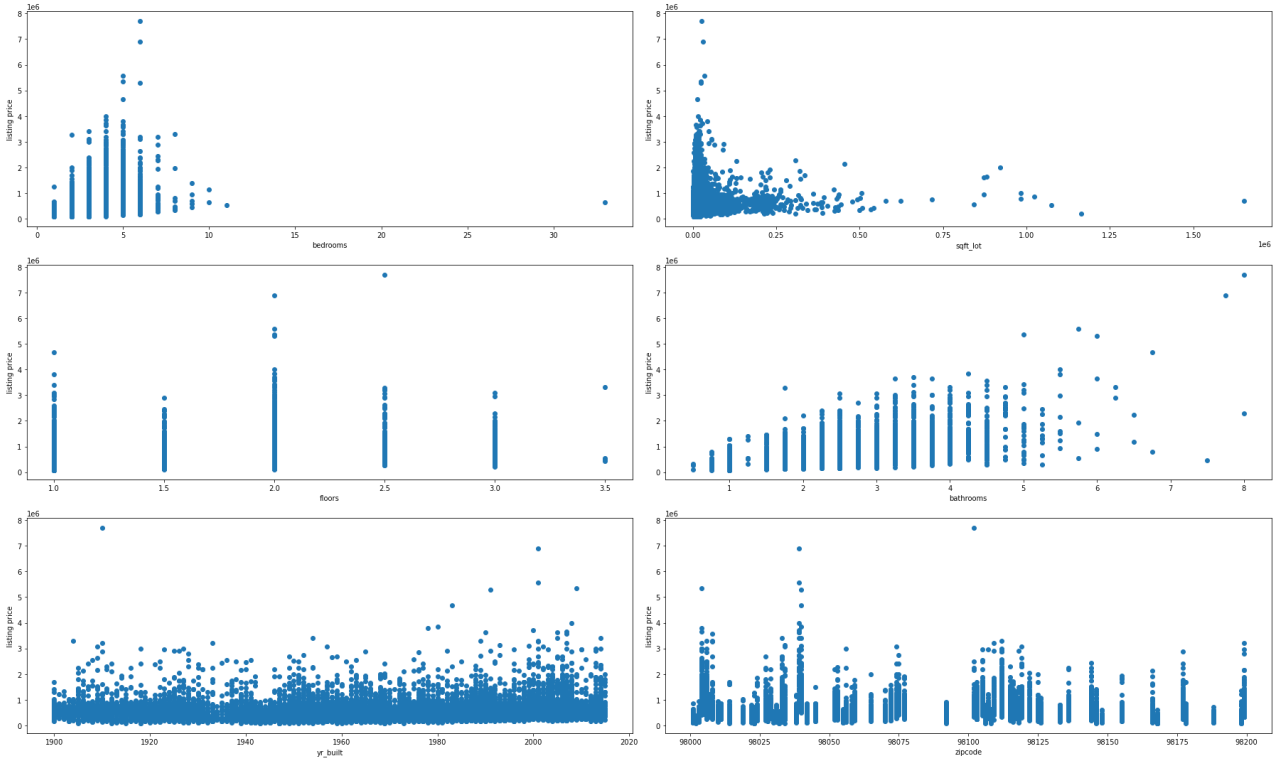
Out[38]:
```
Index(['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors',
       'waterfront', 'yr_built', 'zipcode', 'cond_num', 'grade_num'],
      dtype='object')
```

In [39]:
```
#Specify the groups of features in list for specific plotting uses
#sqft_living was plot above, therfore it is not in this numerical list
#Not showing sqft living, cond_num and grade_num
numerical = ['bedrooms','sqft_lot','floors","bathrooms","yr_built",  "zipcode"]
categoricals = ['waterfront']
```

Linear Reg Code Ref: Phase 2, Topic 20 Linear Reg Lab

Plotting scatter plots to look which features would need further processing and transformaiton.

In [40]:
```
#Plotting scatter plots of numerical values exept sqft_living
X_train_plot = X_train[numerical]

fig, axes = plt.subplots(ncols=2, nrows=3, figsize=(25, 15))
fig.set_tight_layout(True)

for ide, c in enumerate(X_train_plot.columns):
    ax = axes[ide//2][ide%2]
    ax.scatter(X_train_plot[c], y_train)
    ax.set_xlabel(c)
    ax.set_ylabel("listing price")

plt.savefig("images/Pairwise_4", dpi=99)
```



Checking for assumptions of model for all the numerical features .

From the continous features sqft_living is the most linear, but skewed. The other skewed continous features is the sqft_lot.

Log transformations will be performed on the sqft_living and sqft_lot because of their skewness. We will treat bathrooms,bedrooms, floors, zipcode , cond_num, grade_num, and yr_built as discrete variables and

"A heuristic you might use to select continous variables might be a combination of features that are not object datatypes and have at least a certain amount of unique values." Ref: Phase 2 Module 20, Feature Scaling and Norm

---

Data processing and transformation

Certain features from the dataset will be transformed. Log transformations will be performed on the sqft features and the "price" to normalize their data, due to their skewness. Because we already transformed the "condition" and "grade" features to numbers, we will apply the One Hot Encoding transformation to the "Waterfront" categorical feature.

The process of transforming the features. Ref Naming Categories after OHE: Codementor

In [41]:
```
#Setting column names for one hot encoded variables
categoricals
cat_cols_encoded = []
for col in categoricals:
    cat_cols_encoded += [f"{col[0:2]}_{cat}" for cat in list(X[col].unique())]
```

In [42]:
```
cat_cols_encoded
```

Out[42]: ['wa_Unk', 'wa_NO', 'wa_YES']

Reminder, only some of the features will be transformed based on their type.

In [43]:
```
#Creating a list of the features to transform

#Features going through log-transformaiton
log_feat = ["sqft_lot", "sqft_living"]
#Features that will not be transformed
discrete_col= ["bathrooms","bedrooms","yr_built","zipcode","floors", "cond_num",
#OHE features
categoricals = ['waterfront']
```

In [44]:
```
#Reshape Y training data to support modeling without error for single arrays

Y_array_train = np.array(y_train)
y_trainarr = Y_array_train.reshape(-1,1)

#Reshape the test data to support modeling without error for single arrays
Y_array_test = np.array(y_test)
y_testarr = Y_array_test.reshape(-1,1)
```

The below code fits and transforms the data. The datasets are then fitting and transforming data. The X and y train and test datasets are being transformed. Because the X and y are different sets they have different transformers applied.

In [45]:
```
#  import OneHotEncoder from sklearn.preprocessing
from sklearn.preprocessing import OneHotEncoder, FunctionTransformer

# Instantiate transformers
log_transformer = FunctionTransformer(np.log, validate=True)
log_transformer_y = FunctionTransformer(np.log, validate=True)
# OHE instantiation. Will drop a value after the concatenation
ohe = OneHotEncoder(sparse=False)

#Fitting data
#log_feat = ["sqft_lot", "sqft_living"]
log_transformer.fit(X_train[log_feat])
#categoricals = ['waterfront']
ohe.fit(X_train[categoricals])
# ["price"]
log_transformer_y.fit(y_testarr  )

#Train Transformations
X_train_log = log_transformer.transform(X_train[log_feat])
X_train_ohe = ohe.transform(X_train[categoricals])
y_train_log = log_transformer_y.transform(y_trainarr)

#Test Transformation
X_test_log = log_transformer.transform(X_test[log_feat])
X_test_ohe = ohe.transform(X_test[categoricals])
y_test_log = log_transformer_y.transform(y_testarr)


#Concatenate transformed trained data into one dataframe. Keeping the same index

X_train_sec = pd.concat([
        pd.DataFrame(X_train_log, columns= log_feat, index=X_train.index),
```

```
        pd.DataFrame(X_train[discrete_col], columns= discrete_col, index=X_train
        pd.DataFrame(X_train_ohe, columns = cat_cols_encoded, index=X_train.inde
        ], axis=1)

#Concatenate transformed test data.
X_test_sec = pd.concat([
        pd.DataFrame(X_test_log, columns=log_feat, index=X_test.index),
        pd.DataFrame(X_test[discrete_col], columns= discrete_col, index=X_test.i
        pd.DataFrame(X_test_ohe, columns = cat_cols_encoded, index=X_test.index)

    ], axis=1)

#Converting the original y log array to a series with the original shape and an
y_train_log = pd.Series(data=y_train_log.reshape((y_train.shape)), index=X_train
y_test_log = pd.Series(data=y_test_log.reshape((y_test.shape)), index=X_test.ind
```

In [46]:
```
#checking data shapes columns added after OHE
print( X_train_sec.shape, X_test_sec.shape)
```

```
(17272, 12) (4318, 12)
```

In [47]:
```
#Drop the first col of each ohe CAT to reduce multicollinarity
X_train_sec= X_train_sec.drop(['wa_Unk'], axis=1)
X_test_sec= X_test_sec.drop(['wa_Unk'], axis=1)
```

In [48]:
```
#Checking shape
print( X_train_sec.shape,X_test_sec.shape)
```

```
(17272, 11) (4318, 11)
```

Scoring and Predicting Cat and Numericals

Scoring and predicting after transformations occurred.

In [49]:
```
#Initializing model
linreg_sec = LinearRegression(normalize= True)

#Fitting training data
linreg_sec.fit(X_train_sec, y_train_log)
linreg_sec.fit(X_test_sec, y_test_log)
LinearRegression()
```

Out[49]:
```
LinearRegression()
```

In [50]:
```
#Splitting the dataset in three sections with a .3 test size
splitter = ShuffleSplit(n_splits=3, test_size=0.3, random_state=42)

#Modeling second model
second_scores = cross_validate(
    estimator=linreg_sec,
    X=X_train_sec,
    y=y_train_log,
    return_train_score=True,
    cv=splitter
)
```

In [51]:
```
#Y price Predictions for training and testing features
y_hat_train_sec = linreg_sec.predict(X_train_sec)
y_hat_test_sec = linreg_sec.predict(X_test_sec)
```

```
#Root Mean Square Error (if squared is set to False)
train_rmse_sec = mean_squared_error(y_train_log, y_hat_train_sec, squared = Fals
test_rmse_sec = mean_squared_error(y_test_log, y_hat_test_sec, squared = False)
```

In [52]:
```
#Print out of errors and scores for current and previous models

print('Current Train RMSE ', train_rmse_sec)
print('Current Test RMSE: ', test_rmse_sec)
print()
#Mean of the Current kfold validation scores
print("Current Model Kfold Train score:     ", second_scores["train_score"].mean
print("Current Model Validation score:", second_scores["test_score"].mean())
print()
#Mean of the traing validation scores
print("Baseline Models:")
print("Previous Kfold Train mean score:     ", baseline_scores["train_score"].me
print("Previous Kfold Validation mean score:", baseline_scores["test_score"].mea
print()
#Train and validation Scores
print('Previous Train/Test Split Train Model Score: ', Model_train_score)
print('Previous Train/Test Split Validation Model Score: ', Model_score)
```

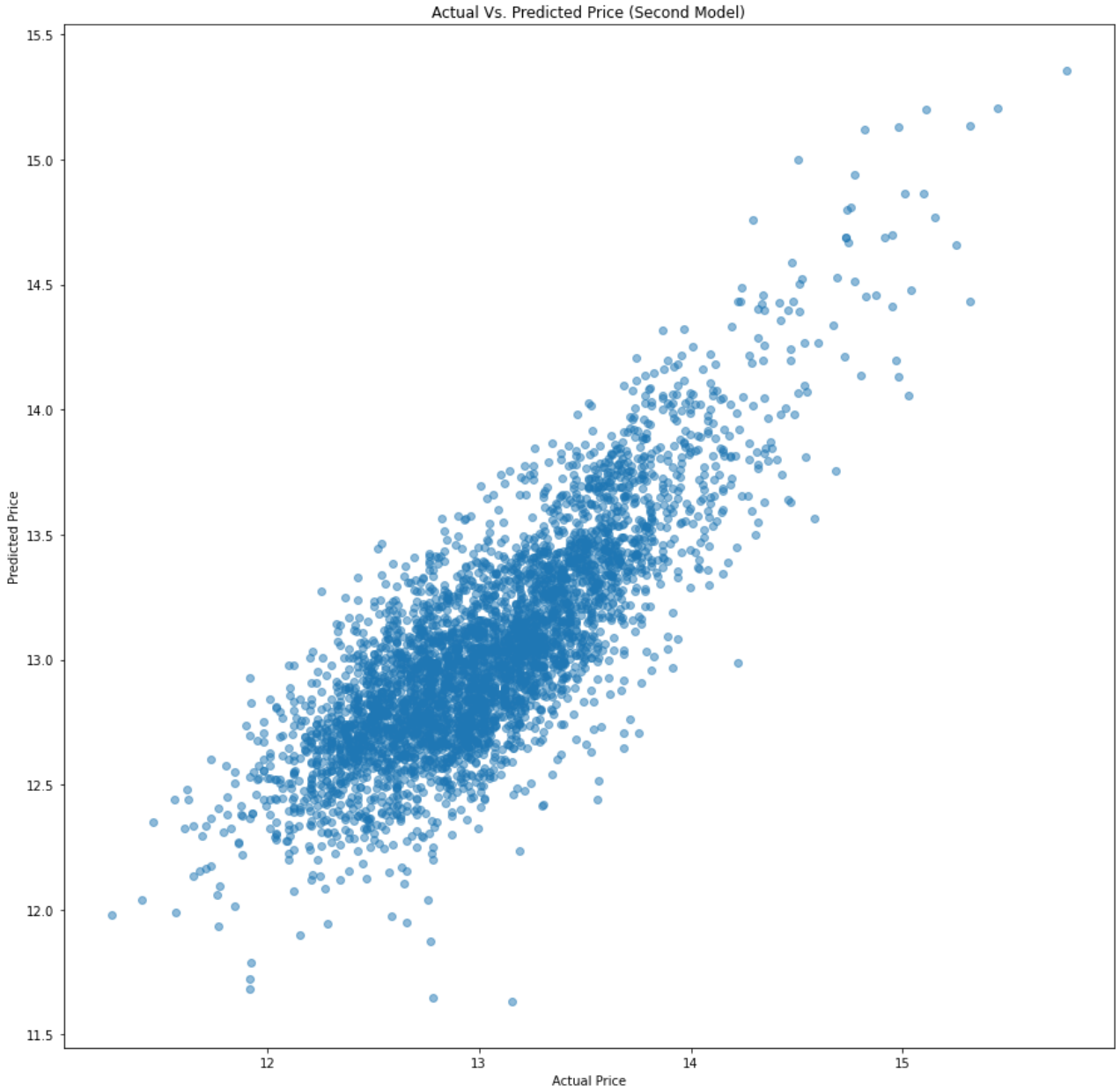
```
Current Train RMSE  0.3148336380868248
Current Test RMSE:  0.30755351160737365

Current Model Kfold Train score:      0.6446907194384797
Current Model Validation score: 0.6414405396108452

Baseline Models:
Previous Kfold Train mean score:      0.4884772214299433
Previous Kfold Validation mean score: 0.5000072841051805

Previous Train/Test Split Train Model Score:  0.49091149233831743
Previous Train/Test Split Validation Model Score:  0.494749423259338
```

The current RMSE of the test data is less than that of the training data. This means the model is more generalized than before. The values are still higher than 0, but have decreased substantially with the transformations.

The current model has increase the R2 score for the validation model by ~(.14). This means we increased the ability for our model to understand the variance in the model by 14%. The validation model is not better than the train model score, there might be a slight overfitting if we use this.

So far we will keep this current model and improve.

In [53]:
```
#plotting the linearity zoomed in
fig, ax = plt.subplots(figsize = (15,15))

ax.scatter(y_test_log, y_hat_test_sec, alpha=0.5)
ax.set_title("Actual Vs. Predicted Price (Second Model)")
ax.set_xlabel("Actual Price")
ax.set_ylabel("Predicted Price")
#plt.xlim([0, 2500000])

plt.savefig("images/Act_Pred_5.png", dpi=99)
```



The predicted data is being plotted for a perfect fitted line. The graph shows linearity up until ~1 Million dollars, but a high number of outliers after this point.

Investigating Multicollinearity

Looking into further multicollinearity of the data. We want the features to be independent. The less they are, the more they can alter the model's fit.

In [54]:
```
#Reviewing current columns in model
X_train_sec.columns
```

Out[54]:
```
Index(['sqft_lot', 'sqft_living', 'bathrooms', 'bedrooms', 'yr_built',
       'zipcode', 'floors', 'cond_num', 'grade_num', 'wa_NO', 'wa_YES'],
      dtype='object')
```

In [55]:
```
#Using statsmodel to view statstics on the model
import statsmodels.api as sm

#Build model
```

```
stats_model = sm.OLS(y_train_log, sm.add_constant(X_train_sec)).fit()

#View model
stats_model.summary()
```

Out[55]:

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.644
Model:	OLS	**Adj. R-squared:**	0.644
Method:	Least Squares	**F-statistic:**	2837.
Date:	Mon, 02 Jan 2023	**Prob (F-statistic):**	0.00
Time:	21:51:05	**Log-Likelihood:**	-4523.4
No. Observations:	17272	**AIC:**	9071.
Df Residuals:	17260	**BIC:**	9164.
Df Model:	11		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.3025	5.076	0.060	0.952	-9.646	10.251
sqft_lot	-0.0397	0.003	-12.423	0.000	-0.046	-0.033
sqft_living	0.4395	0.012	36.579	0.000	0.416	0.463
bathrooms	0.0848	0.006	15.413	0.000	0.074	0.096
bedrooms	-0.0389	0.003	-11.436	0.000	-0.046	-0.032
yr_built	-0.0059	0.000	-52.006	0.000	-0.006	-0.006
zipcode	0.0002	5.08e-05	3.878	0.000	9.75e-05	0.000
floors	0.0450	0.006	7.626	0.000	0.033	0.057
cond_num	0.0370	0.004	9.129	0.000	0.029	0.045
grade_num	0.2348	0.003	70.474	0.000	0.228	0.241
wa_NO	-0.0053	0.008	-0.688	0.491	-0.020	0.010
wa_YES	0.5477	0.031	17.856	0.000	0.488	0.608

Omnibus:	75.352	**Durbin-Watson:**	1.989
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	104.717
Skew:	-0.014	**Prob(JB):**	1.82e-23
Kurtosis:	3.380	**Cond. No.**	2.08e+08

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 2.08e+08. This might indicate that there are strong multicollinearity or other numerical problems.

Which ones features should be eliminated based on p value? We are assuming alpha level (significance level) of .05. If the p-value is above this, this let's us know we should reject these features. This is based on a hypothesis that these features are meaningful to the model.

Looks like there is a lot of multicolinearity in the model. The features that are above the needed p-value are:

- **"wa_NO"** .

Also, both "wa_NO" coefficient and the "zipcode" are outside of the confidence intervals, meaning they are outliers. The two most significant coefficients are the "grad_num" and "sqft_living based on their std_err and high t value

Ref Statsmodel InterpretationTim McAleer, 2020

We will run Recursive Feature Evaluation (RFE) to see if some of the same values are chosen to be removed.

Feature Recommendations with RFECV

Feature selection method that elimates the weakest features depening on settings.

In [56]:
```
from sklearn.feature_selection import RFECV
#Features must be scaled to be used
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import cross_validate, ShuffleSplit

splitter = ShuffleSplit(n_splits=3, test_size=0.3, random_state=42)


X_train_for_RFECV = StandardScaler().fit_transform(X_train_sec)

#Initiate linear Refression model
model_for_RFECV = LinearRegression()


# Instantiate and fit the selector
selector = RFECV(model_for_RFECV, cv=splitter)
selector.fit(X_train_for_RFECV, y_train_log)


# Print the results Ref: [Phase 2, Topic 20, Linear Regression Lab]
print("Non Recommended Features")
#Creates a fist for Reocmmended dropped columns
#The recommended dropped features will be False. If False, add to list RFECV_rec
RFECV_rec_drop_col = []
for index, col in enumerate(X_train_sec.columns):
    if selector.support_[index] == False:
        print(f"{col}: {selector.support_[index]}")
        RFECV_rec_drop_col.append(col)
```

```
Non Recommended Features
wa_NO: False
```

In [57]:
```
print("Rec P-value based drop: " "co_Poor","wa_NO" )
```

```
print("Rec RFECV based drop: ",  RFECV_rec_drop_col )
```

```
Rec P-value based drop: co_Poor wa_NO
Rec RFECV based drop:  ['wa_NO']
```

The RFECV method methods agrees on dropping all the same features as the p-value test. On the next model all of the recommended features will be dropped.

In [58]:
```
#Checking current columns
X_train_sec.columns
```

Out[58]:
```
Index(['sqft_lot', 'sqft_living', 'bathrooms', 'bedrooms', 'yr_built',
       'zipcode', 'floors', 'cond_num', 'grade_num', 'wa_NO', 'wa_YES'],
      dtype='object')
```

In [59]:
```
#Dropping features from both training and test set separately
X_train_sec_drop = X_train_sec.drop( RFECV_rec_drop_col, axis =1)
X_test_sec_drop = X_test_sec.drop( RFECV_rec_drop_col, axis =1)
```

In [60]:
```
#Checking current shape
print( X_train_sec_drop.shape,X_test_sec_drop.shape)
```

```
(17272, 10) (4318, 10)
```

In [61]:
```
#Check the stats model with the dataset (with wa_NO removed)
model = sm.OLS(y_train_log, sm.add_constant(X_train_sec_drop)).fit()

#Model the stats model
model.summary()
```

Out[61]:

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.644
Model:	OLS	**Adj. R-squared:**	0.644
Method:	Least Squares	**F-statistic:**	3120.
Date:	Mon, 02 Jan 2023	**Prob (F-statistic):**	0.00
Time:	21:51:06	**Log-Likelihood:**	-4523.7
No. Observations:	17272	**AIC:**	9069.
Df Residuals:	17261	**BIC:**	9155.
Df Model:	10		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.3710	5.075	0.073	0.942	-9.576	10.318
sqft_lot	-0.0397	0.003	-12.424	0.000	-0.046	-0.033
sqft_living	0.4394	0.012	36.576	0.000	0.416	0.463
bathrooms	0.0848	0.006	15.413	0.000	0.074	0.096
bedrooms	-0.0389	0.003	-11.432	0.000	-0.046	-0.032
yr_built	-0.0059	0.000	-52.013	0.000	-0.006	-0.006
zipcode	0.0002	5.08e-05	3.865	0.000	9.68e-05	0.000

floors	0.0450	0.006	7.629	0.000	0.033	0.057
cond_num	0.0370	0.004	9.125	0.000	0.029	0.045
grade_num	0.2349	0.003	70.482	0.000	0.228	0.241
wa_YES	0.5483	0.031	17.883	0.000	0.488	0.608

Omnibus:	75.119	**Durbin-Watson:**	1.989
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	104.330
Skew:	-0.014	**Prob(JB):**	2.21e-23
Kurtosis:	3.380	**Cond. No.**	2.08e+08

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 2.08e+08. This might indicate that there are
strong multicollinearity or other numerical problems.

Even though features were dropped, the model still suggests a high multicollinearity and the same R2 score. We will be using the model for predictions and not specifically inferences on the features and their impacts. We can ignore this for now and continue on.

Best Features Model (3)

We will take the best features we have left and train the next model. The best features so far include:

- 'sqft_lot', 'sqft_living', 'bathrooms', 'bedrooms', 'yr_built',
- 'zipcode', 'floors', 'cond_num', 'grade_num', 'wa_YES'

In [62]:
```
#Setting new dataframe for third model
X_train_third = X_train_sec_drop
X_test_third = X_test_sec_drop
```

In [63]:
```
#Printing current columans
X_train_third.columns
```

Out[63]:
```
Index(['sqft_lot', 'sqft_living', 'bathrooms', 'bedrooms', 'yr_built',
       'zipcode', 'floors', 'cond_num', 'grade_num', 'wa_YES'],
      dtype='object')
```

In [64]:
```
#Split values for the cv
splitter = ShuffleSplit(n_splits=3, test_size=0.3, random_state=42)

#Initiating the Estimator
third_model = LinearRegression()

#Scoring using the esimator and training data
third_scores = cross_validate(
    estimator=third_model,
    X=X_train_third,
```

```
    y=y_train_log,
    return_train_score=True,
    cv=splitter
)
```

In [65]:
```
#Initializing model
#third_model = LinearRegression(normalize= True)

#Fitting training data
third_model.fit(X_train_third, y_train_log)
third_model.fit(X_test_third, y_test_log)
LinearRegression()
```

Out[65]:
```
LinearRegression()
```

In [66]:
```
#Y price Predictions for training and testing features
y_hat_train_third = third_model.predict(X_train_third)
y_hat_test_third = third_model.predict(X_test_third)

#Root Mean Square Error (if squared is set to False)
train_rmse_third = mean_squared_error(y_train_log, y_hat_train_third, squared =
test_rmse_third = mean_squared_error(y_test_log, y_hat_test_third, squared = Fal
```

In [67]:
```
#Print out of errors and scores for current and previous models

print('Current Train RMSE ', train_rmse_third)
print('Current Test RMSE: ', test_rmse_third)
print()
#Mean of the Third validation scores
print('Current Train Model Mean Score: ', third_scores["train_score"].mean())
print('Current Validation Model Mean Score: ', third_scores["test_score"].mean()
print()
#Mean of the Current kfold validation scores
print("Second Model Kfold Train Mean score:     ", second_scores["train_score"].
print("Second Model Validation Mean score:", second_scores["test_score"].mean())
print()
#Mean of the traing validation scores
print("Baseline Models:")
print("Previous Kfold Train Mean Score:     ", baseline_scores["train_score"].me
print("Previous Kfold Validation Mean Score:", baseline_scores["test_score"].mea
print()
#Train and validation Scores
print('Previous Train/Test Split Train Model Score: ', Model_train_score)
print('Previous Train/Test Split Validation Model Score: ', Model_score)
```

```
Current Train RMSE  0.3147718460539656
Current Test RMSE:  0.3075890371545622

Current Train Model Mean Score:  0.644673314060476
Current Validation Model Mean Score:  0.6414594822403191

Second Model Kfold Train Mean score:      0.6446907194384797
Second Model Validation Mean score: 0.6414405396108452

Baseline Models:
Previous Kfold Train Mean Score:      0.4884772214299433
Previous Kfold Validation Mean Score: 0.5000072841051805

Previous Train/Test Split Train Model Score:  0.49091149233831743
Previous Train/Test Split Validation Model Score:  0.494749423259338
```

In [68]:
```
print('Current Train Model Scorea: ', third_scores["train_score"])
print('Current Validation Model Scores: ', third_scores["test_score"])
```

```
Current Train Model Scorea:  [0.64160177 0.64360325 0.64881492]
Current Validation Model Scores:  [0.64847415 0.643949   0.6319553 ]
```

There isn't a big change in the third and second model R2 score even after feature selection. The test validation increased slightly as the train score decreased. They are vary smilar though. When looking at the specic fold splits, we can see there are runs where the validation score was larger than the training score.

Final model (4)

After modeling three iterations at similar score of ~.65 for the training and validation data, a final model will be fit and used for prediciton using the last dataset.

In [69]:
```
#Transform data set by changing the condition or grade and try to predict on fin
X_train_final = X_train_third
X_test_final = X_test_third
```

In [70]:
```
#Ref Linear Regression Model 20

#Initiate Linear Regression model
final_lin = LinearRegression()

# Fit the model on X_train_final and y_train
final_lin.fit(X_train_final, y_train_log)

# Score the model on X_test_final and y_test
train_final_score = final_lin.score(X_train_final, y_train_log)
test_final_score = final_lin.score(X_test_final, y_test_log)

#The train and test model predict the y predictors
y_hat_train_fin = final_lin.predict(X_train_final)
y_hat_test_fin = final_lin.predict(X_test_final)

#Mean Square erorr of the known y and predictors
train_rmse = mean_squared_error(y_train_log, y_hat_train_fin, squared= False)
test_rmse = mean_squared_error(y_test_log, y_hat_test_fin, squared = False)
```

In [71]:
```
#Final RMSE and R2 Scores
print('Final Train Mean Squared Error:', train_rmse)
print('Final Test Mean Squared Error: ', test_rmse)
print()
print('Final Train Model Score: ', train_final_score)
print('Final Test Model Score: ', test_final_score)
```

```
Final Train Mean Squared Error: 0.31441867034646886
Final Test Mean Squared Error:  0.3079357313675995

Final Train Model Score:  0.6438531698069864
Final Test Model Score:  0.6565129767577557
```

For the final model, the test .66 is higher than the the training model. This helps us know the model is not overfitted.

The current model's RMSE is .307.

Interpret Final Model (5)

Interpreting final model against linearity assumptions.

Ref Interpreation Code: Phase 2, Topic 20 Linear Reg Lab

In [72]:
```
print(pd.Series(final_lin.coef_, index=X_train_final.columns, name="Coefficients
print()
print("Intercept:", final_lin.intercept_)
```

```
sqft_lot      -0.039696
sqft_living    0.439416
bathrooms      0.084826
bedrooms      -0.038859
yr_built      -0.005880
zipcode        0.000196
floors         0.045027
cond_num       0.036963
grade_num      0.234864
wa_YES         0.548252
Name: Coefficients, dtype: float64

Intercept: 0.3709916228143566
```

If all the assumptions are met then these coefficiants and intercept will give us inferencial insight. The coefficiants in this model help us understand how the different features impact the target variable of price.

We will highlight a the base coefficents:

```
- The sqft_living feature saids that for each unit increase of the
living area the price will increase by .43
- The grade and condition are specific features that deal with home
quality metrics.
    - Every unit of increase of the grade increases the price by
.23.
    - Every unit of increase of the condition increases the price
by .03.
- The same pattern can be applied tho the other coefficients.
```

These values are still with log transformation values

In [73]:
```
y_pred_no_log = np.exp(final_lin.intercept_)
y_pred_no_log
```

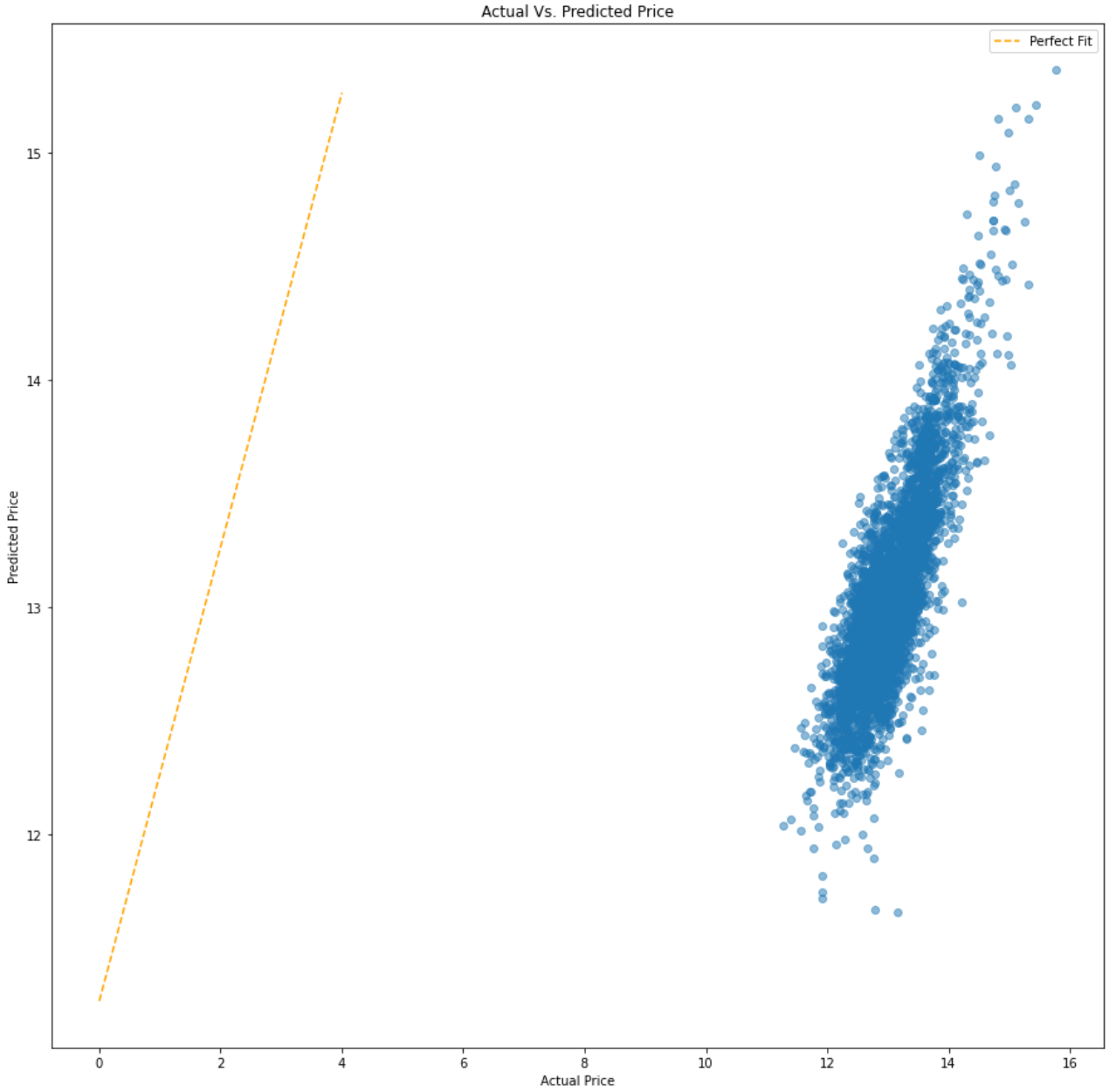
Out[73]:
```
1.4491709333618572
```

In [74]:
```
#For explanational purposes inversing the values with log apply
import math

#Y pred inverse log
y_pred_no_log = np.exp(y_hat_test_fin)
y_pred_no_log = y_pred_no_log.round(2)
y_pred_no_log
```

Out[74]:
```
array([807435.97, 239940.92, 377907.52, ..., 492818.19, 492192.68,
       406997.63])
```

Linear Regression Assumption

In [75]:
```
fig, ax = plt.subplots(figsize = (15,15))

perfect_line = np.arange(y_test_log.min(), y_test_log.max())
ax.plot(perfect_line, linestyle="--", color="orange", label="Perfect Fit")
ax.scatter(y_test_log, y_hat_test_fin, alpha=0.5)
ax.set_title("Actual Vs. Predicted Price")
ax.set_xlabel("Actual Price")
ax.set_ylabel("Predicted Price")

ax.legend();

plt.savefig("images/Lin_Assump6.png", dpi=99)
```



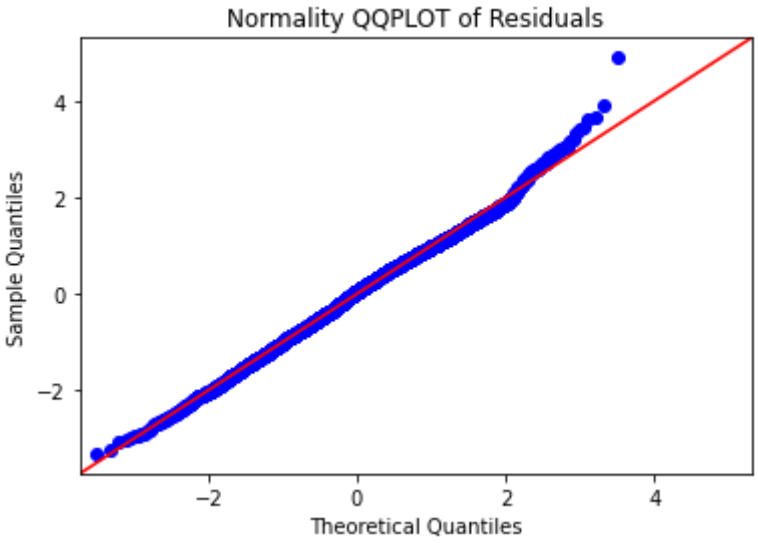
The perfect line is parallel to the scatter plot. The scatter plot has the same linear shape. It is

still considered linear.

Normality Assumption

Model RegressionRef: Phase 2, Model Reggression Lab Video

In [76]:
```
import scipy.stats as stats
residuals = (y_test_log - y_hat_test_fin)
#QQPLOT
sm.graphics.qqplot(residuals, dist=stats.norm, line='45', fit=True);
plt.title("Normality QQPLOT of Residuals")

plt.savefig("images/Normality_7.png", dpi=99)
```



Using the QQ plot shows normality with a slight offshoot at the end.

Multicollinearity

In [77]:
```
from statsmodels.stats.outliers_influence import variance_inflation_factor
#Variance Inflation Factor
vif = [variance_inflation_factor(X_train_final.values, i) for i in range(X_train
pd.Series(vif, index=X_train_final.columns, name="Variance Inflation Factor")
```

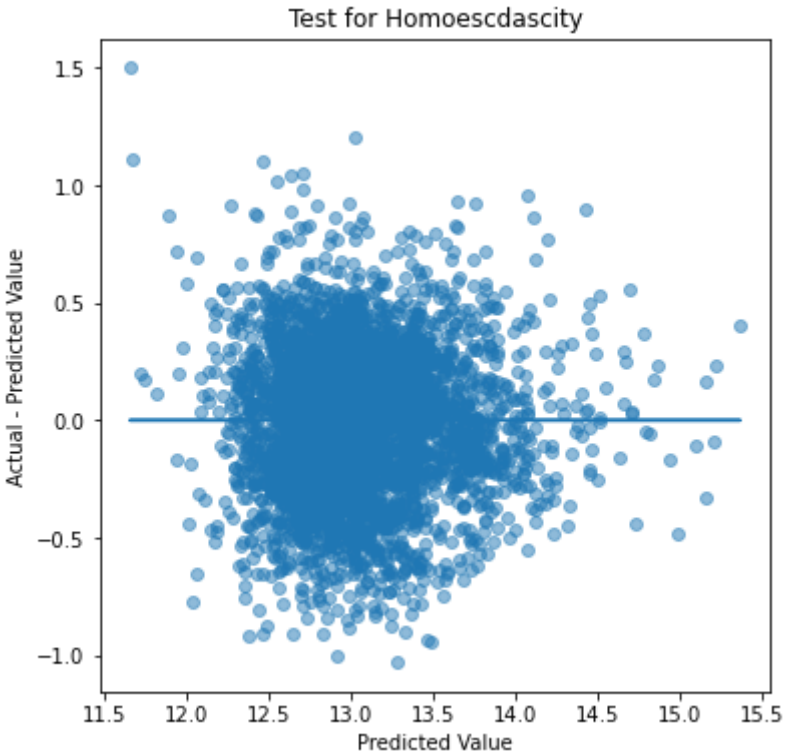
Out[77]:
```
sqft_lot         137.239768
sqft_living     1440.908390
bathrooms         26.860744
bedrooms          24.645905
yr_built        7382.348064
zipcode         8235.823316
floors            15.333726
cond_num          33.895878
grade_num        116.375693
wa_YES             1.024235
Name: Variance Inflation Factor, dtype: float64
```

Wow! All the numerical values and two grade categorical values have high VIF except for "wa_Yes". There is a strong multicollinarity between these features. Dropping some of these features will support the multicollinearity problem. It is not clear what is the best combination of features is the most effective for our problem. (In a different iteration, not pictured, some of these were dropped after these results; the multicollinearity dropped, but also the R2 score.)

Because we are using this model for prediction and not inference we can keep using the trained fitted model. If the goal is to understand specific individual features and how they impact the model, then multicollinearity is a major issue that needs to be solved.

Homoescdasticity

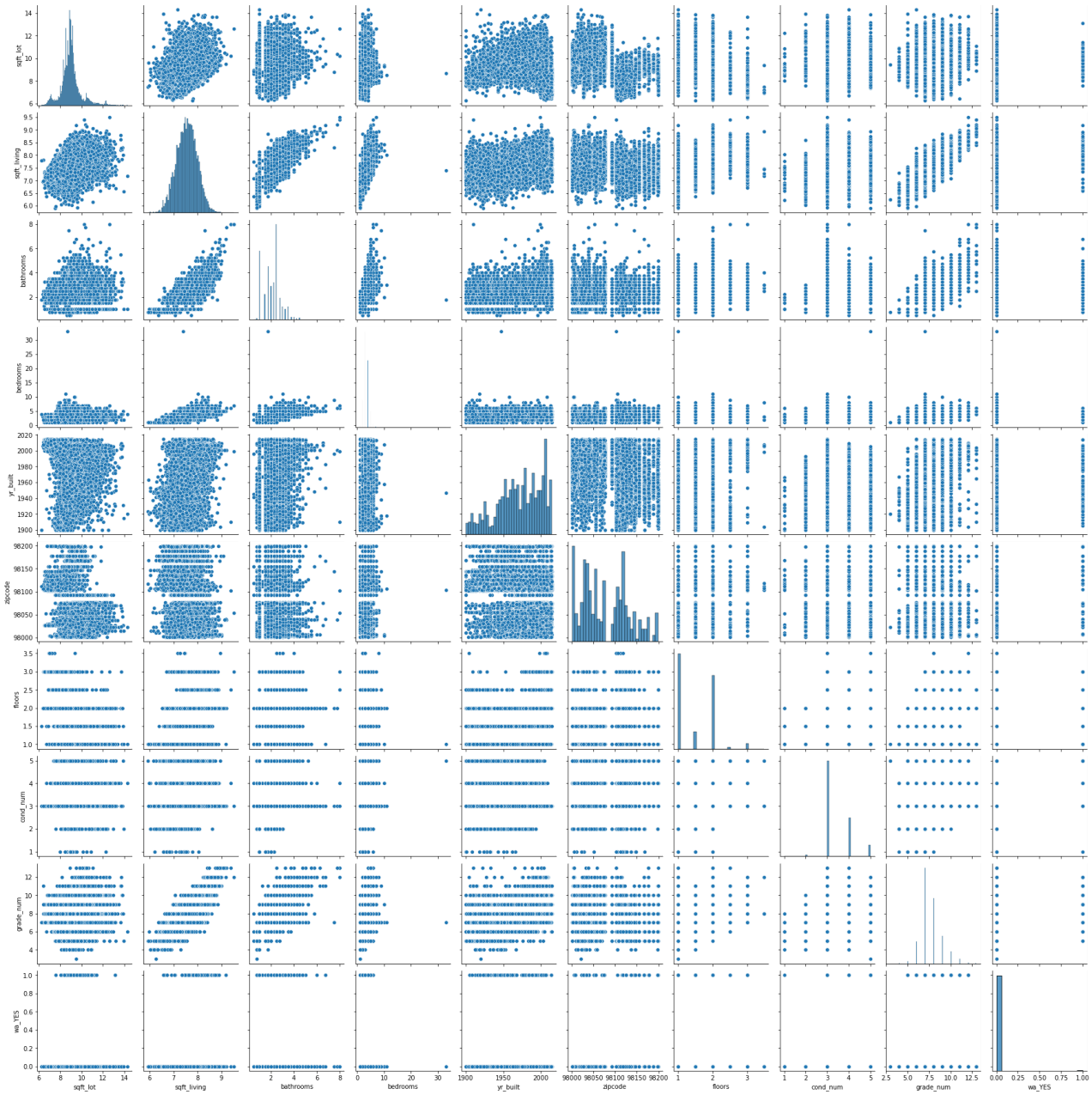
In [78]:
```
fig, ax = plt.subplots(figsize = (6,6))
#Scatter plot of predictions and residuals
ax.scatter(y_hat_test_fin, residuals, alpha=0.5)
ax.plot(y_hat_test_fin, [0 for i in range(len(X_test_final))])
ax.set_xlabel("Predicted Value")
ax.set_ylabel("Actual - Predicted Value");
ax.set_title("Test for Homoescdascity")

plt.savefig("images/Homoesc_8.png", dpi=99)
```



Most of the residuals are homoescadastic. There are still plenty of outliers that can be drop or dealt with. Even with the outliers it generally passess the test.

Model Feedback

In [79]:
```
#Looking at the final dataset in the pairplot
sns.pairplot(X_train_final)
plt.savefig("images/Pair_10.png", dpi=99)
```



Model Prediction

The best fit model will be used to make predictions for homes that have home improvements. In this case the "condition" and "grade" features for a specific group of homes in certain zipcodes. The top five zipcodes that have the most homes with a "**condition" of 3-Good or less & a "grade" of 6-Low Average or less**". The chosen homes will all be modified to have a condition of "4-Good and a grade of 8-Good

These predictions will serve as samples for the real estate agents to understand the benefit of real estate data analysis. Because home value can depend on the surrounding community, grouping the homes for predictions by zipcodes helps the agent target a specific community for an extra benefit.

Dataframe Evaluation and Manipulation

In [80]:
```
#Copy the original dataset (with waterfront, condition and grade processed)
imp_df = kc_df.copy()
imp_df.head()
```

Out[80]:

id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
7129300520	221900.0	3	1.00	1180	5650	1.0	Unk	1955
6414100192	538000.0	3	2.25	2570	7242	2.0	NO	1951
5631500400	180000.0	2	1.00	770	10000	1.0	NO	1933
2487200875	604000.0	4	3.00	1960	5000	1.0	NO	1965
1954400510	510000.0	3	2.00	1680	8080	1.0	NO	1987

Filter the data based on certain condition and grade criteria. The grade and condition values are mapped above and explained on King County Glossary

In [81]:
```
#Filter the grades meeting the criteria
re_build = imp_df[(imp_df.grade_num <= 6) & (imp_df.cond_num <= 3) ].sort_values
```

In [82]:
```
#Filtering the zipcodes and choose the highest amount of homes
re_build.groupby(["zipcode"]).count().sort_values(["cond_num"], ascending= False
```

In [83]:
```
#Creating a list from the top amount of homes
#Use isin to find the specific zipcodes in the list
filt_zcode = [98118,98168,98146,98106,98126]
re_filt = re_build[re_build.zipcode.isin(filt_zcode)]
re_filt.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 438 entries, 3395800155 to 2114700090
Data columns (total 11 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   price        438 non-null    float64
 1   bedrooms     438 non-null    int64  
 2   bathrooms    438 non-null    float64
 3   sqft_living  438 non-null    int64  
 4   sqft_lot     438 non-null    int64  
 5   floors       438 non-null    float64
 6   waterfront   438 non-null    object 
 7   yr_built     438 non-null    int64  
 8   zipcode      438 non-null    int64  
 9   cond_num     438 non-null    int64  
 10  grade_num    438 non-null    int64  
dtypes: float64(3), int64(7), object(1)
memory usage: 41.1+ KB
```

There are 438 entries that fit the criteria from the top five zipcode.

In [84]:
```
#Reviewing the grade values
re_filt["grade_num"].value_counts()
```

Out[84]:
```
6    396
5     37
```

```
4      5
Name: grade_num, dtype: int64
```

In [85]:
```
#Reviewing the condition values
re_filt["cond_num"].value_counts()
```

Out[85]:
```
3    399
2     33
1      6
Name: cond_num, dtype: int64
```

In [86]:
```
#Look at how the attributes for the columns changes
re_filt.describe()
```

Out[86]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	yr
count	438.000000	438.000000	438.000000	438.000000	438.000000	438.000000	438.0
mean	253603.541096	2.600457	1.147260	1104.963470	7541.929224	1.098174	1937.4
std	96721.581601	0.916515	0.356379	385.561368	4602.033460	0.215416	16.4
min	78000.000000	1.000000	0.500000	380.000000	1642.000000	1.000000	1900.0
25%	190000.000000	2.000000	1.000000	830.000000	5085.000000	1.000000	1925.0
50%	244250.000000	2.000000	1.000000	1025.000000	6333.500000	1.000000	1942.0
75%	300000.000000	3.000000	1.000000	1307.500000	8221.000000	1.000000	1948.0
max	795000.000000	6.000000	3.000000	2710.000000	49658.000000	2.000000	2000.0

In [87]:
```
#Checking which features are the most correlated with price
re_filt.corr()
```

Out[87]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	yr_built	zip
price	1.000000	0.167822	0.201698	0.307478	-0.172134	0.176807	-0.237924	-0.33
bedrooms	0.167822	1.000000	0.374962	0.627679	-0.007657	0.448324	0.032831	0.00
bathrooms	0.201698	0.374962	1.000000	0.525424	-0.040053	0.224834	-0.029989	-0.04
sqft_living	0.307478	0.627679	0.525424	1.000000	0.049063	0.440787	-0.039116	0.06
sqft_lot	-0.172134	-0.007657	-0.040053	0.049063	1.000000	0.043095	0.120375	0.3
floors	0.176807	0.448324	0.224834	0.440787	0.043095	1.000000	-0.130731	0.02
yr_built	-0.237924	0.032831	-0.029989	-0.039116	0.120375	-0.130731	1.000000	0.24
zipcode	-0.339209	0.006204	-0.048096	0.062236	0.391710	0.022898	0.247452	1.00
cond_num	0.133077	0.129963	0.025534	0.129319	-0.128576	0.104872	0.022373	-0.0
grade_num	0.248845	0.247798	0.142854	0.296250	-0.041478	0.080536	-0.021847	-0.0

We want to change the values of all the condition and grade features in the scoped dataset.

In [88]:
```
#Create a new dataframe as a baseline for changing the condition and grade value
re_new = re_filt.copy()
# Change the grade and conditions to the goal values
re_new["grade_num"] = 8
re_new["cond_num"] =  4
```

In [89]:
```
#Review the dataframe
re_new.head()
```

Out[89]:

id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
3395800155	250000.0	3	1.00	990	8100	1.0	NO	1949
8122600145	452000.0	4	2.00	1660	6150	1.0	NO	1945
8019200925	315000.0	5	1.75	1850	14800	1.5	NO	1937
1443500120	310000.0	2	1.00	750	5379	1.0	NO	1919
798000145	244500.0	2	1.75	1300	14500	1.0	NO	1939

Data Transformation

In [90]:
```
#Split the data between dependent and the independent variables.
#The target variable will only be used to compare the predictor
X_re_new = re_new.drop("price", axis = 1)
y_re_new = re_new["price"]
```

In [91]:
```
#Creating a lists for the features to transform
#Features going through log-transformaiton
log_feat = ["sqft_lot", "sqft_living"]
#Features that will not be transformed
discrete_col= ["bathrooms","bedrooms","yr_built","zipcode","floors", "cond_num",
#OHE features
categoricals = ['waterfront']
```

In [92]:
```
#Creating new column labels
#The X_train is used because it has the entire set of values for Waterfront.
#The dataset used for prediction has no "Yes" values
categoricals
cat_cols_re = []
for col in categoricals:
    cat_cols_re += [f"{col[0:2]}_{cat}" for cat in list(X_train[col].unique())]
```

In [93]:
```
#Display the column names
cat_cols_re
```

Out[93]:
```
['wa_NO', 'wa_Unk', 'wa_YES']
```

In [94]:
```
#Reshape Y training data to support modeling without error for single arrays
y_array_re = np.array(y_re_new)
y_re = y_array_re.reshape(-1,1)
```

Below we are transforming the data like the previous trained and test data. Because we are using a previously created model, we do not need to initiate and fit the models again. It has already been done.

In [95]:
```
#  import OneHotEncoder from sklearn.preprocessing
from sklearn.preprocessing import OneHotEncoder, FunctionTransformer

#The transformers wer
```

```
#Test Transformation
X_re_log = log_transformer.transform(X_re_new[log_feat])
X_re_ohe = ohe.transform(X_re_new[categoricals])
y_re_log = log_transformer_y.transform(y_re)

#Concatenate transformed train data. Keeping the same index as the original trai

X_re_pred = pd.concat([
        pd.DataFrame(X_re_log , columns= log_feat, index=X_re_new.index),
        pd.DataFrame(X_re_new[discrete_col], columns= discrete_col, index=X_re_n
        pd.DataFrame(X_re_ohe, columns = cat_cols_re, index=X_re_new.index),
        ], axis=1)


#Converting the original y log array to a series with the original shape and an
y_re_act_log = pd.Series(data=y_re_log.reshape((y_re_new.shape)), index=y_re_new
```

Using the same data we are initiating and fitting a linearegression model just like the final model above.

Prediction

In [96]:
```
#Ref Linear Regression Model 20
pred_lin = LinearRegression(normalize= True)

# Fit the model on X_train_final and y_train
pred_lin.fit(X_train_final, y_train)
```

Out[96]:
```
LinearRegression(normalize=True)
```

In [97]:
```
#Viewing the columns
X_re_pred.columns
```

Out[97]:
```
Index(['sqft_lot', 'sqft_living', 'bathrooms', 'bedrooms', 'yr_built',
       'zipcode', 'floors', 'cond_num', 'grade_num', 'wa_NO', 'wa_Unk',
       'wa_YES'],
      dtype='object')
```

In [98]:
```
#Reviewing the columns from the final training set
X_train_final.columns
```

Out[98]:
```
Index(['sqft_lot', 'sqft_living', 'bathrooms', 'bedrooms', 'yr_built',
       'zipcode', 'floors', 'cond_num', 'grade_num', 'wa_YES'],
      dtype='object')
```

In [99]:
```
#Dropping the features not used
X_re_pred.drop(['wa_NO', 'wa_Unk'], inplace= True, axis = 1)
```

Using the fitted model we will predict the new housing prices.

In [100…
```
#Newly predicted Home Values
re_pred = pred_lin.predict(X_re_pred)
#Rounding the values of the new predictions
re_pred = re_pred.round(-3)
re_pred
```

Out[100…
```
array([482000., 669000., 662000., 596000., 660000., 614000., 508000.,
```

```
497000., 490000., 708000., 621000., 641000., 471000., 557000.,
463000., 494000., 707000., 551000., 834000., 557000., 667000.,
570000., 468000., 572000., 618000., 605000., 686000., 604000.,
460000., 643000., 498000., 691000., 691000., 744000., 550000.,
657000., 461000., 555000., 466000., 456000., 375000., 620000.,
593000., 688000., 656000., 635000., 421000., 533000., 633000.,
684000., 652000., 478000., 621000., 519000., 483000., 444000.,
596000., 551000., 618000., 510000., 489000., 605000., 594000.,
502000., 593000., 635000., 522000., 535000., 635000., 499000.,
627000., 495000., 676000., 477000., 477000., 754000., 487000.,
573000., 505000., 465000., 422000., 592000., 714000., 639000.,
532000., 522000., 645000., 422000., 439000., 558000., 558000.,
480000., 488000., 512000., 500000., 583000., 585000., 585000.,
529000., 549000., 624000., 496000., 649000., 532000., 710000.,
845000., 559000., 576000., 524000., 648000., 490000., 498000.,
680000., 531000., 629000., 517000., 483000., 431000., 547000.,
512000., 575000., 494000., 574000., 653000., 450000., 530000.,
866000., 479000., 611000., 665000., 469000., 691000., 526000.,
527000., 470000., 542000., 390000., 586000., 440000., 785000.,
558000., 547000., 497000., 497000., 660000., 850000., 383000.,
746000., 746000., 545000., 518000., 674000., 610000., 610000.,
496000., 523000., 692000., 791000., 456000., 489000., 698000.,
503000., 556000., 538000., 617000., 617000., 563000., 682000.,
532000., 710000., 571000., 533000., 696000., 619000., 949000.,
484000., 707000., 755000., 527000., 599000., 367000., 611000.,
550000., 583000., 513000., 613000., 611000., 581000., 548000.,
750000., 886000., 665000., 495000., 638000., 477000., 453000.,
458000., 529000., 542000., 461000., 489000., 618000., 467000.,
218000., 769000., 460000., 726000., 579000., 584000., 448000.,
653000., 484000., 625000., 478000., 586000., 332000., 472000.,
602000., 519000., 434000., 561000., 561000., 579000., 563000.,
546000., 472000., 604000., 537000., 517000., 669000., 433000.,
530000., 526000., 858000., 693000., 692000., 555000., 496000.,
592000., 501000., 582000., 505000., 818000., 624000., 518000.,
490000., 563000., 542000., 732000., 734000., 591000., 534000.,
591000., 544000., 544000., 451000., 698000., 629000., 592000.,
512000., 659000., 463000., 589000., 531000., 503000., 580000.,
518000., 404000., 516000., 433000., 497000., 590000., 442000.,
794000., 794000., 605000., 577000., 468000., 537000., 889000.,
489000., 519000., 582000., 629000., 654000., 654000., 495000.,
607000., 736000., 591000., 476000., 597000., 490000., 638000.,
681000., 505000., 500000., 470000., 705000., 474000., 438000.,
484000., 479000., 470000., 568000., 636000., 555000., 491000.,
475000., 679000., 588000., 598000., 475000., 531000., 497000.,
455000., 509000., 673000., 638000., 796000., 466000., 564000.,
540000., 602000., 609000., 529000., 652000., 570000., 595000.,
598000., 514000., 582000., 479000., 565000., 728000., 471000.,
452000., 473000., 547000., 592000., 485000., 664000., 642000.,
576000., 640000., 519000., 513000., 784000., 500000., 773000.,
496000., 828000., 490000., 645000., 433000., 688000., 707000.,
659000., 465000., 607000., 717000., 539000., 518000., 433000.,
502000., 602000., 723000., 434000., 533000., 584000., 697000.,
636000., 636000., 687000., 578000., 529000., 626000., 661000.,
534000., 637000., 489000., 441000., 589000., 554000., 544000.,
499000., 728000., 781000., 640000., 621000., 610000., 803000.,
600000., 690000., 507000., 691000., 653000., 475000., 369000.,
482000., 492000., 450000., 762000., 596000., 569000., 272000.,
536000., 509000., 404000., 450000., 622000., 476000., 551000.,
554000., 494000., 475000., 500000., 482000., 495000., 495000.,
504000., 494000., 723000., 446000., 512000., 704000., 485000.,
494000., 591000., 509000., 374000., 579000., 588000., 383000.,
337000., 452000., 440000., 435000.])
```

New Dataframe Creation

Using new (modified condition, grade and new predictions) and old columns to create a new table

In [101…
```
#Replacing value names
re_new = re_new.rename(columns={'cond_num': "new_cond",'grade_num': "new_grade"
re_new.head()
```

Out[101…

id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
3395800155	250000.0	3	1.00	990	8100	1.0	NO	1949
8122600145	452000.0	4	2.00	1660	6150	1.0	NO	1945
8019200925	315000.0	5	1.75	1850	14800	1.5	NO	1937
1443500120	310000.0	2	1.00	750	5379	1.0	NO	1919
798000145	244500.0	2	1.75	1300	14500	1.0	NO	1939

In [102…
```
#Concatenating the dataframe to show original and new predictions
X_comp = pd.concat([
        re_filt,
        pd.DataFrame(re_pred ,columns= ["pred_value"], index=re_new.index),
        re_new[["new_cond","new_grade"]]], axis=1)
X_comp.head()
```

Out[102…

id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
3395800155	250000.0	3	1.00	990	8100	1.0	NO	1949
8122600145	452000.0	4	2.00	1660	6150	1.0	NO	1945
8019200925	315000.0	5	1.75	1850	14800	1.5	NO	1937
1443500120	310000.0	2	1.00	750	5379	1.0	NO	1919
798000145	244500.0	2	1.75	1300	14500	1.0	NO	1939

In [103…
```
#Replacing names for better clarity
X_comp = X_comp.rename(columns={"price":"org_value",'cond_num': "org_cond",'grad
# Adding a column for difference in price
X_comp["value_diff"] = X_comp["pred_value"] - X_comp["org_value"]
#Adding a column for percent difference
X_comp["perc_diff"] =  (((X_comp["pred_value"] - X_comp["org_value"])/X_comp["or
X_comp.head()
```

Out[103…

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
3395800155	250000.0	3	1.00	990	8100	1.0	NO	1949
8122600145	452000.0	4	2.00	1660	6150	1.0	NO	1945
8019200925	315000.0	5	1.75	1850	14800	1.5	NO	1937
1443500120	310000.0	2	1.00	750	5379	1.0	NO	1919

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
798000145	244500.0	2	1.75	1300	14500	1.0	NO	1939

Data Filtering

Filtering data by zipcode

In [104…
```
#Create function to make table for specific zipcode
def filt_zcode(zipcode):
    zcode_filt = X_comp[X_comp["zipcode"] == zipcode]
    return zcode_filt
```

In [105…
```
#Viewing the table
spec_zip = filt_zcode(98118)
spec_zip.head()
```

Out[105…

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
1443500120	310000.0	2	1.00	750	5379	1.0	NO	1919
1443500925	455000.0	2	1.00	1140	11480	1.0	NO	1907
4356200120	248000.0	1	1.00	790	12000	1.0	NO	1918
4006000281	227000.0	3	1.75	2380	12681	1.0	NO	1918
9331800580	257000.0	2	1.00	1000	3700	1.0	NO	1929

In [106…
```
#Evaluating the difference in price between the original and predicted values
#Zipcode 98188
spec_zip[["value_diff"]].describe()
```

Out[106…

	value_diff
count	106.000000
mean	323466.584906
std	136648.067604
min	-35000.000000
25%	242000.000000
50%	328000.000000
75%	430625.000000
max	624000.000000

In [107…
```
#Viewing the change in values from lowest to highest
#Looking for outliers

X_comp.sort_values(by = "value_diff", ascending=True)
```

Out[107…

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
4364700805	315000.0	1	1.00	580	7200	1.0	Unk	2000
8121100395	645000.0	4	1.50	1600	6180	1.5	NO	1946
9349900105	795000.0	2	1.00	1380	5000	1.5	NO	1905
6303400395	325000.0	1	0.75	410	8636	1.0	NO	1953
1604601855	360500.0	3	1.00	970	6180	1.0	NO	1974
5297200089	664000.0	2	1.75	1720	5785	1.0	NO	1948
1723049033	245000.0	1	0.75	380	15000	1.0	NO	1963
3331500650	356000.0	3	1.00	920	3863	1.0	NO	1970
3277801450	390000.0	4	1.00	1140	6250	1.5	NO	1958
5249804560	510000.0	4	1.00	1060	7200	1.0	NO	1925
5649600266	386000.0	3	1.50	1550	8000	1.0	NO	1980
952003480	445000.0	4	1.00	1460	4600	1.0	NO	1946
6083000037	230000.0	2	1.00	930	7550	1.0	NO	1986
6888900115	555750.0	3	1.00	1060	4880	1.0	NO	1913
3438501860	385000.0	3	1.00	1020	5950	1.0	NO	1950
1005000250	350000.0	2	1.00	840	5551	1.0	NO	1952
3331500940	342000.0	2	1.00	740	6180	1.0	NO	1948
3624039074	430000.0	3	1.00	1210	5200	1.0	NO	1941
4365200860	385200.0	4	1.00	1550	7740	1.5	Unk	1954
9485700136	330000.0	3	1.00	1140	7316	1.0	Unk	1959
5550300205	338000.0	2	1.00	690	6400	1.0	NO	1943
5249803645	452000.0	2	1.00	1220	6000	1.0	Unk	1938
5132000140	415000.0	6	1.00	1370	5080	1.5	Unk	1931
3812400070	435000.0	5	1.00	1410	6750	1.5	NO	1929
8862000075	285000.0	2	1.00	790	6555	1.0	NO	1956
9285800735	406650.0	2	1.00	1070	6100	1.0	NO	1940
6303400475	227000.0	4	1.00	1120	8763	1.0	Unk	1971
5249804760	479500.0	2	1.00	930	5760	1.0	NO	1917
4364700990	335000.0	3	1.00	1030	7200	1.0	NO	1948
7454001405	387500.0	4	1.00	1370	7140	2.0	NO	1942
3298700946	340000.0	2	1.00	1090	6771	1.0	NO	1954
7454001090	307000.0	3	1.00	770	6552	1.0	Unk	1942
3438500880	325000.0	2	1.00	810	6827	1.0	NO	1944
8121100147	390000.0	3	2.25	1640	2875	2.0	NO	1983

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
7452500365	310000.0	2	1.00	870	5400	1.0	NO	1950
7129304540	440000.0	5	2.00	1430	5600	1.5	NO	1947
4006000251	226000.0	3	1.00	970	5000	1.0	NO	1968
9269200650	314000.0	2	1.00	720	4920	1.0	NO	1941
2586800270	425000.0	4	1.00	1260	7645	1.5	NO	1925
7211400760	277000.0	4	1.00	1450	6250	1.0	NO	1964
4364700165	249900.0	2	1.00	560	7560	1.0	NO	1944
8121100395	425000.0	4	1.50	1600	6180	1.5	NO	1946
7899800045	232900.0	3	1.50	910	5120	1.0	NO	1973
7950304065	260000.0	2	1.00	690	6000	1.0	NO	1949
2489200165	435000.0	3	1.00	1050	5500	1.0	NO	1920
3438500981	280000.0	2	1.00	790	13170	1.0	NO	1947
952004745	400800.0	4	1.00	1070	5750	1.0	NO	1923
7454000315	299500.0	2	1.00	740	6300	1.0	NO	1942
1446401190	175000.0	2	1.00	620	6600	1.0	NO	1963
5347200070	339000.0	3	1.00	1150	2496	1.0	NO	1947
3333000655	334000.0	2	1.00	890	6000	1.0	NO	1941
9485700150	275000.0	2	1.00	920	7688	1.0	NO	1955
9297301520	410000.0	2	1.75	870	4000	1.0	NO	1941
7454000585	289000.0	2	1.00	710	6300	1.0	NO	1942
123039364	300000.0	2	1.00	970	13700	1.0	NO	1949
3550800040	223000.0	3	1.00	940	7980	1.0	NO	1961
4109600306	475000.0	2	1.00	920	5157	1.0	NO	1909
4364700600	390000.0	3	1.00	1010	7920	1.0	NO	1925
3333002450	490000.0	1	1.00	850	8050	1.0	NO	1906
8150100240	265000.0	2	1.00	620	4760	1.0	NO	1941
2624049103	449000.0	2	1.00	1250	4576	1.0	Unk	1925
7387500335	280000.0	3	1.00	980	7480	1.0	NO	1948
1105000015	417000.0	2	1.00	920	6600	1.0	NO	1919
7211400850	229000.0	3	1.50	1200	5000	1.0	NO	1979
4302200790	248500.0	2	1.00	720	5160	1.0	NO	1949
7387500195	367000.0	4	1.00	1820	5500	1.5	NO	1947
3883800011	219900.0	3	1.00	860	10426	1.0	NO	1954
4154300505	315000.0	2	1.00	780	7200	1.0	NO	1935

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
8150100265	255000.0	2	1.00	620	4760	1.0	NO	1941
8122600145	452000.0	4	2.00	1660	6150	1.0	NO	1945
1446401540	243000.0	3	1.00	1500	6600	1.0	NO	1970
7262200150	315000.0	2	1.00	970	18557	1.0	NO	1939
9285800755	515000.0	3	2.50	1540	6100	1.0	NO	1944
2742100016	260000.0	3	1.00	940	5650	1.0	NO	1949
7453000070	275000.0	2	1.00	940	5000	1.0	NO	1951
2123049086	210000.0	2	0.75	840	49658	1.0	NO	1948
623049093	219900.0	3	1.00	910	6000	1.0	NO	1956
1623049062	210000.0	2	1.00	750	34133	1.0	NO	1950
7452500530	250000.0	2	1.00	850	6370	1.0	NO	1951
2113701200	250000.0	2	1.00	670	4640	1.0	NO	1943
7454001075	240000.0	2	1.00	670	10920	1.0	Unk	1942
795002450	270950.0	2	1.00	780	6250	1.0	NO	1942
9285800801	364500.0	3	1.00	1600	4489	1.0	NO	1944
795001600	340000.0	3	1.00	1710	10190	1.0	NO	1949
8088600080	274950.0	3	1.00	1450	8820	1.0	NO	1958
4302201130	205000.0	2	1.00	720	5040	1.0	NO	1955
2641800060	239000.0	3	1.00	940	8571	1.0	NO	1950
3395800295	250000.0	2	1.00	1030	8786	1.0	NO	1956
952003340	380000.0	2	1.00	780	3910	1.0	NO	1918
3395800155	250000.0	3	1.00	990	8100	1.0	NO	1949
1601600195	299000.0	3	1.00	1510	6200	1.0	NO	1955
4303200555	265000.0	2	1.00	770	5160	1.0	NO	1943
1446401220	226950.0	2	1.00	930	6600	1.0	NO	1957
7452500340	265000.0	3	1.00	1080	4930	1.0	Unk	1950
3298700840	263500.0	2	1.00	750	4515	1.0	NO	1942
1773101530	275000.0	1	1.00	520	4800	1.0	NO	1930
7452500730	264950.0	2	1.00	1000	6000	1.0	NO	1951
745000005	145000.0	1	0.75	480	9750	1.0	NO	1948
9269200786	399950.0	4	1.50	1850	6125	1.5	NO	1945
7454000295	245000.0	2	1.00	710	6322	1.0	NO	1942
7896300070	265000.0	4	1.00	1290	6034	1.0	NO	1950
8122100650	316000.0	2	1.00	730	5040	1.0	NO	1927

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
2624049073	360000.0	2	1.00	780	4200	1.0	NO	1920
7129304375	202000.0	1	0.75	590	5650	1.0	NO	1944
7899800905	475000.0	3	1.75	1150	10240	1.0	NO	1918
5347200220	225000.0	2	1.00	720	4758	1.0	Unk	1947
7972603931	240000.0	2	1.00	720	6345	1.0	NO	1943
5249803745	367500.0	2	1.00	810	4800	1.0	NO	1919
5347200165	265000.0	3	1.00	1070	4800	1.0	NO	1947
1446800511	249950.0	4	1.00	1330	7980	1.5	NO	1952
40001065	250000.0	2	1.00	1110	26051	1.0	NO	1951
402000260	190000.0	2	1.00	700	9500	1.0	NO	1951
1148000005	346000.0	3	1.75	1270	8100	1.0	NO	1950
123039336	244900.0	1	1.00	620	8261	1.0	NO	1939
1604600227	441000.0	2	1.00	1150	3000	1.0	NO	1915
6453300055	188000.0	1	1.00	550	16345	1.0	NO	1945
7452500315	285000.0	2	1.00	1210	4895	1.0	NO	1951
7972603950	238000.0	2	1.00	750	6480	1.0	NO	1943
7454000470	412500.0	3	1.75	1530	6300	1.0	Unk	1942
3598600049	224000.0	1	0.75	840	7203	1.5	NO	1949
1443500925	455000.0	2	1.00	1140	11480	1.0	NO	1907
795000405	285950.0	2	1.00	1170	6000	1.0	NO	1948
952004725	280000.0	2	1.00	880	5750	1.0	NO	1939
623049232	115000.0	2	0.75	550	7980	1.0	NO	1952
66000070	315000.0	2	1.00	630	6550	1.0	NO	1918
3361401210	209000.0	2	1.00	1070	6120	1.0	NO	1962
7452500045	235000.0	2	1.00	870	5000	1.0	NO	1949
3172600151	250000.0	4	1.00	1550	7296	1.5	NO	1957
6303400460	197000.0	2	1.00	770	8636	1.0	NO	1951
2112701165	285000.0	4	1.00	1430	3600	1.0	Unk	1947
923049323	239000.0	4	1.00	1280	8316	1.0	NO	1950
3333002790	243500.0	2	1.00	900	5016	1.0	Unk	1948
9382200025	220000.0	3	1.00	1090	6320	1.0	Unk	1954
795000865	235000.0	3	1.00	1020	6173	1.0	NO	1948
3298700941	260000.0	3	1.00	1200	4592	1.0	NO	1950
723049156	284700.0	3	1.00	1700	8645	1.0	NO	1955

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
3438501081	315000.0	3	1.00	970	6828	1.0	NO	1928
2341300115	235000.0	2	1.00	720	6321	1.0	NO	1940
4302200695	270000.0	2	1.00	1000	10320	1.0	NO	1943
952003350	507250.0	3	1.75	1400	5750	1.0	Unk	1915
4154302045	376000.0	2	1.00	880	2400	1.0	Unk	1918
795002375	280000.0	3	1.00	1200	6250	1.0	NO	1943
5147600105	178500.0	2	1.00	740	6460	1.0	NO	1953
985001321	291000.0	4	1.00	1590	24330	1.5	NO	1942
8151600610	235750.0	2	1.00	740	11250	1.0	NO	1938
4302200535	219000.0	2	1.00	900	5160	1.0	NO	1952
8864000440	225000.0	3	1.00	900	6099	1.0	Unk	1944
3336001360	254000.0	2	1.00	910	6000	1.0	NO	1943
1895000045	195000.0	2	1.00	820	5100	1.0	NO	1953
795002455	261000.0	2	1.00	970	12500	1.0	NO	1941
8113100150	210000.0	3	1.00	920	6612	1.0	NO	1948
7454000110	202000.0	2	1.00	670	7844	1.0	NO	1942
792000006	187000.0	2	1.00	840	11600	1.0	NO	1952
3331001995	509990.0	3	2.00	1440	4859	2.0	NO	1921
3624039111	215000.0	3	1.00	980	5600	1.0	NO	1949
3192000080	205000.0	3	1.00	1210	10185	1.0	NO	1957
1231001090	362362.0	2	1.00	710	4000	1.0	NO	1909
1231001110	380000.0	3	1.00	920	3532	1.0	NO	1910
7750500120	300000.0	3	1.00	950	4760	1.5	NO	1929
7950302150	385000.0	1	1.00	660	3570	1.0	NO	1906
1105000233	255000.0	2	1.00	940	9330	1.0	NO	1941
8148600055	225000.0	3	1.00	1040	6535	1.0	NO	1947
3192000085	180000.0	3	1.00	1010	10215	1.0	NO	1955
7889600685	205000.0	3	0.75	1080	5025	1.0	NO	1948
798000535	308000.0	3	1.00	1640	18144	1.5	NO	1942
723049219	210000.0	3	1.00	880	10800	1.0	NO	1942
7972603385	245000.0	2	1.00	870	6150	1.0	NO	1941
5347200060	280000.0	2	1.00	1260	4800	1.0	NO	1947
402000145	217000.0	2	1.00	970	5600	1.0	NO	1951
2789000120	335000.0	2	1.00	1800	8900	1.0	NO	1945

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
9352901085	256000.0	3	1.00	1290	4720	1.0	NO	1948
8151600701	234000.0	2	1.00	870	11100	1.0	NO	1940
1231000510	510000.0	3	1.75	1490	3800	1.0	NO	1913
2114700090	151000.0	2	0.75	720	5040	1.0	NO	1949
7972602080	312000.0	4	1.00	1190	7620	1.5	NO	1926
723049307	210000.0	3	1.00	1070	8179	1.0	NO	1949
7452500815	212625.0	2	1.00	960	5000	1.0	NO	1951
3330500085	366000.0	2	1.00	1210	3090	1.0	NO	1926
9269200520	310000.0	1	1.00	670	4920	1.0	NO	1920
1443500120	310000.0	2	1.00	750	5379	1.0	NO	1919
7454001210	239000.0	3	1.00	1040	6860	2.0	NO	1942
1721801280	230000.0	2	0.75	900	3527	1.0	NO	1939
40000228	221900.0	2	1.00	780	6727	1.0	Unk	1939
5347200175	299800.0	2	1.00	1310	2814	1.0	NO	1944
3332000091	320000.0	3	1.00	1190	4120	1.0	Unk	1929
133000070	179900.0	2	1.00	680	6400	1.0	NO	1943
3298700426	226550.0	3	1.00	990	4440	1.0	NO	1943
1623049041	82500.0	2	1.00	520	22334	1.0	NO	1951
1443501020	163250.0	2	1.00	770	8150	1.0	Unk	1951
4363700200	190000.0	4	1.00	1190	7920	1.0	NO	1951
6083000050	235000.0	3	1.75	1900	8540	1.0	Unk	1980
7387500185	249900.0	2	1.00	1140	5500	1.0	NO	1947
7129800036	109000.0	2	0.50	580	6900	1.0	NO	1941
9275200080	295000.0	3	1.50	720	7450	1.0	NO	1924
293000180	370000.0	2	1.00	910	5525	1.0	NO	1910
3826000280	272000.0	3	1.00	1130	8100	1.5	NO	1934
1231001225	385000.0	2	1.00	1010	4000	1.0	NO	1911
1121000095	320000.0	2	1.00	1120	5329	1.0	NO	1929
123039279	165000.0	2	1.00	640	7768	1.0	NO	1942
1446403835	189000.0	2	1.00	790	7128	1.0	NO	1944
8122600165	273000.0	3	1.00	1500	6250	1.0	NO	1945
2407000405	228500.0	3	1.00	1080	7486	1.5	NO	1942
257000138	280000.0	2	1.00	850	16400	1.0	NO	1923
1446400715	280000.0	2	1.00	1310	6600	1.0	NO	1942

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
7831800460	235000.0	2	1.00	1210	9400	1.0	NO	1949
7520000020	244000.0	4	1.00	1450	8960	1.0	NO	1943
1446400648	203000.0	2	1.00	1080	9067	1.0	NO	1951
1446401290	214950.0	3	1.00	1400	6600	1.0	NO	1954
1446400785	228950.0	3	1.00	1120	6625	1.0	NO	1942
7454001280	220000.0	3	1.00	1050	6300	1.0	NO	1942
3438500625	210000.0	3	1.00	1080	21043	1.0	Unk	1942
5269200050	175000.0	2	1.00	700	8174	1.0	NO	1941
7899800890	181000.0	2	1.50	720	5120	1.0	NO	1954
6620400205	200000.0	2	1.00	1000	6227	1.0	NO	1949
723049596	255000.0	2	1.00	810	7980	1.0	NO	1928
3438500430	270000.0	3	1.75	1390	10905	1.0	NO	1957
985001275	250000.0	1	1.00	800	16306	1.0	Unk	1931
8150100045	210000.0	2	1.00	830	6000	1.0	NO	1940
2113700510	315000.0	3	1.75	1170	4000	1.0	NO	1943
1670400090	182000.0	3	1.00	1160	18055	1.0	NO	1950
3438501700	300000.0	3	1.00	1300	20812	1.0	NO	1927
7889601300	268000.0	3	1.00	1420	6000	1.0	NO	1941
2641800015	158800.0	3	1.00	960	8291	1.0	NO	1950
7950300775	350000.0	1	1.00	790	4590	1.0	NO	1911
3438503045	165000.0	2	1.00	780	6380	1.0	NO	1947
3361400980	135000.0	2	1.00	600	6120	1.0	NO	1943
316000145	235000.0	4	1.00	1360	7132	1.5	NO	1941
7899800045	107000.0	3	1.50	910	5120	1.0	NO	1973
6303401050	132500.0	3	0.75	850	8573	1.0	NO	1945
8148600020	170000.0	2	1.00	870	6537	1.0	NO	1948
7972604345	137000.0	3	1.00	950	7620	1.0	NO	1954
1721800190	300000.0	2	1.50	1300	6120	1.0	NO	1945
2112700895	276000.0	2	1.00	720	4000	1.0	NO	1918
6099400053	145000.0	3	1.00	1010	5490	1.0	NO	1954
402000110	175000.0	2	1.00	960	5508	1.0	NO	1951
3826000470	232000.0	2	1.00	960	8100	1.0	NO	1936
9358000780	275000.0	2	1.00	830	5610	1.0	NO	1922
3361400190	190000.0	3	1.00	1040	8910	1.0	NO	1943

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
7972601995	245000.0	2	1.00	1200	4880	1.0	Unk	1943
1773101020	307000.0	5	1.50	1310	4800	1.5	NO	1929
8019200845	245000.0	2	1.00	1020	15000	1.5	Unk	1933
1446403617	123000.0	2	1.00	1050	6600	1.5	Unk	1964
316000160	260000.0	3	1.00	1480	7469	1.5	NO	1940
2112700240	309000.0	3	1.00	1092	7500	1.5	NO	1918
2113200065	289000.0	2	1.00	1010	7740	1.0	Unk	1924
623049094	180000.0	3	1.00	1000	18513	1.0	NO	1940
2414600195	210000.0	3	1.00	1520	8600	1.0	NO	1951
8122600020	200000.0	4	1.00	1310	5200	1.5	NO	1945
1721801010	302100.0	3	1.00	1790	6120	1.0	NO	1937
7520000695	251000.0	3	1.00	840	4495	1.0	NO	1921
3336000296	250000.0	4	1.50	1220	4900	1.0	NO	1942
7831800505	200000.0	3	1.00	1230	4380	1.0	Unk	1947
795000620	157000.0	3	1.00	1080	6250	1.0	Unk	1950
2407000145	197200.0	3	1.00	1140	8775	1.0	NO	1942
7211402105	106000.0	1	1.00	560	5700	1.0	NO	1947
623049341	260000.0	3	2.00	1030	7260	1.0	NO	1947
3361401011	110000.0	2	1.00	600	6120	1.0	Unk	1943
8961800035	229000.0	2	1.00	1190	7408	1.0	NO	1941
1610000016	175000.0	4	1.00	1300	6030	1.5	Unk	1947
795002190	205000.0	2	1.00	1060	8000	1.0	NO	1941
3163600015	156000.0	2	1.00	600	4000	1.0	NO	1933
2123049194	199950.0	3	1.50	1370	10317	1.5	NO	1958
723049132	235000.0	2	1.00	1500	8015	1.0	Unk	1947
8019200925	315000.0	5	1.75	1850	14800	1.5	NO	1937
123039336	148000.0	1	1.00	620	8261	1.0	NO	1939
2124700015	345000.0	3	1.00	1120	10176	1.0	NO	1905
9331800580	257000.0	2	1.00	1000	3700	1.0	NO	1929
1523049188	84000.0	2	1.00	700	20130	1.0	NO	1949
254000735	329000.0	3	1.00	1140	5258	1.5	NO	1911
3883800011	82000.0	3	1.00	860	10426	1.0	NO	1954
723049197	195000.0	2	1.00	1020	8100	1.0	NO	1940
3812400202	156000.0	2	1.75	590	6138	1.0	NO	1947

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
3598600049	124000.0	1	0.75	840	7203	1.5	NO	1949
7950302345	345000.0	3	1.00	1010	3060	1.5	NO	1904
3336002215	319950.0	2	1.00	1240	5500	1.5	NO	1921
1773100765	229000.0	1	1.00	600	3720	1.0	NO	1920
1105000296	230000.0	2	1.00	720	5913	1.0	NO	1920
7973202225	154200.0	4	1.00	1310	8640	1.0	NO	1948
1231000640	290000.0	2	1.00	960	4000	1.0	NO	1918
1446401460	122000.0	2	1.00	760	5280	1.0	NO	1946
9358001590	340000.0	5	1.00	1880	3774	1.5	NO	1917
3332000615	389000.0	3	1.00	1330	3740	1.5	Unk	1903
2414600366	199900.0	1	1.00	720	7140	1.0	NO	1930
3812400657	160000.0	3	1.00	1200	8360	1.0	NO	1948
9382200121	187300.0	2	1.00	1310	7697	1.0	NO	1950
7899800120	294350.0	3	1.00	1410	5120	1.5	NO	1925
2114700500	90000.0	1	1.00	560	4120	1.0	NO	1947
2113701095	150000.0	2	1.00	830	4045	1.0	Unk	1943
7950302890	455000.0	4	2.00	2380	4500	1.5	NO	1926
8122100835	183000.0	2	1.00	670	5140	1.0	NO	1926
3256400051	210000.0	3	2.00	960	9380	1.0	NO	1949
6303400150	255000.0	3	1.00	1160	8636	1.0	NO	1923
623049185	185000.0	5	1.00	1590	6700	1.5	Unk	1942
2113701100	294010.0	3	1.75	1550	4057	1.0	NO	1945
8071000050	270000.0	2	1.00	1040	5700	1.0	NO	1922
3395800455	150000.0	2	1.00	890	8100	1.0	NO	1942
3365901435	165000.0	3	1.00	1200	13100	1.0	NO	1943
2586800140	135000.0	2	1.00	830	7609	1.0	NO	1943
111000190	146000.0	2	1.00	780	9750	1.0	NO	1937
1443500385	155000.0	2	1.00	910	6232	1.0	Unk	1943
7454000145	122000.0	2	1.00	740	6840	1.0	NO	1942
795000620	124000.0	3	1.00	1080	6250	1.0	NO	1950
723049533	271000.0	4	1.75	1490	9112	1.0	NO	1940
3826000550	245000.0	5	1.50	2000	8100	1.5	NO	1946
4356200120	248000.0	1	1.00	790	12000	1.0	NO	1918
6083000123	158000.0	3	1.00	1140	10477	1.0	Unk	1942

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
2114700615	148000.0	2	1.00	630	4200	1.0	NO	1930
8128700005	249000.0	4	1.00	1200	7552	1.0	NO	1919
7973202712	130000.0	2	1.00	780	5300	1.0	NO	1941
4364700600	216000.0	3	1.00	1010	7920	1.0	NO	1925
892000025	114975.0	2	1.00	740	6250	1.0	NO	1942
1909600115	420000.0	3	2.00	2330	6346	1.5	NO	1934
8151601190	180000.0	5	1.00	1460	11726	1.5	NO	1936
985001015	135000.0	1	1.00	790	13062	1.0	NO	1942
7899800915	216000.0	2	1.00	710	5120	1.0	NO	1918
8018600870	224000.0	2	1.00	1150	15000	1.0	NO	1930
1443500905	219950.0	3	1.00	1020	4960	1.5	NO	1926
795000620	115000.0	3	1.00	1080	6250	1.0	NO	1950
7211400990	256000.0	2	1.00	860	5000	1.0	NO	1915
5414100040	299950.0	2	1.00	800	3000	1.0	NO	1904
257000263	182200.0	4	1.00	1130	13927	1.5	NO	1929
6083000071	195000.0	3	2.00	1230	8235	1.5	NO	1959
1446801030	220000.0	4	1.75	1660	11664	1.0	NO	1952
8151600470	121800.0	2	1.00	940	8384	1.0	NO	1947
5132000140	175000.0	6	1.00	1370	5080	1.5	NO	1931
795000765	92000.0	2	1.00	760	5500	1.5	Unk	1947
859000110	125000.0	1	1.00	500	7440	1.0	NO	1928
1721801591	89950.0	1	1.00	570	4080	1.0	NO	1942
9200000050	109500.0	2	1.00	800	10625	1.0	NO	1942
952004570	320000.0	2	1.00	1140	3834	1.5	NO	1911
40000553	250000.0	2	1.00	1400	19570	1.5	NO	1929
7211402305	240000.0	3	1.75	1780	5000	1.5	NO	1957
123039333	240000.0	4	1.00	1910	16320	1.5	NO	1934
3330500335	325000.0	2	1.00	1010	6180	1.0	NO	1903
7972600670	339000.0	4	2.00	2470	5080	1.5	NO	1948
2113700500	250800.0	3	1.75	1290	4000	1.0	NO	1943
2407000110	275000.0	3	1.75	1580	8775	1.0	Unk	1942
7340601063	295500.0	3	1.75	1590	41550	1.5	Unk	1933
9297301015	277284.0	3	1.75	1030	4800	1.0	NO	1927
4006000401	140000.0	2	1.00	900	6400	1.0	NO	1940

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
7987400316	255000.0	1	0.50	880	1642	1.0	NO	1910
2113700205	220000.0	4	1.00	1200	6000	1.5	NO	1923
3826000070	185000.0	3	1.00	1150	8100	1.0	NO	1932
3028200080	81000.0	2	1.00	730	9975	1.0	Unk	1943
723049156	149000.0	3	1.00	1700	8645	1.0	NO	1955
1703401110	292000.0	2	1.00	880	5500	1.0	NO	1904
6303401395	245000.0	2	1.75	1220	8382	1.0	NO	1942
1604601155	180000.0	3	1.00	780	3540	1.0	Unk	1920
5147600095	152000.0	3	1.75	1070	7754	1.0	NO	1953
723049530	126500.0	3	1.00	1130	12212	1.0	NO	1942
423049067	160000.0	2	1.00	930	7742	1.0	NO	1933
257000105	192500.0	2	1.00	950	7692	1.0	NO	1926
4154302075	200000.0	2	1.00	830	7200	1.0	NO	1920
3298700820	160000.0	3	1.75	1010	5355	1.0	NO	1950
9352900695	170000.0	3	1.00	1480	5670	1.0	NO	1944
40000362	78000.0	2	1.00	780	16344	1.0	NO	1942
293000145	250000.0	4	1.00	1440	7404	1.0	NO	1918
1443500725	280000.0	3	1.00	1350	7553	1.5	Unk	1914
3331001910	312000.0	2	1.00	1170	5150	1.0	NO	1907
1721801010	225000.0	3	1.00	1790	6120	1.0	NO	1937
7889601320	115000.0	2	1.00	940	6000	1.0	NO	1943
6911700066	175000.0	2	1.00	670	2378	1.0	NO	1919
4365700330	275000.0	2	1.75	930	7080	1.0	NO	1923
7520000520	240500.0	2	1.00	1240	12092	1.0	NO	1922
84000245	190000.0	3	1.75	1100	9452	1.0	NO	1942
4365700450	193000.0	2	1.00	950	9000	1.0	NO	1924
3624039150	335000.0	3	2.00	1170	5360	1.0	NO	1919
798000145	244500.0	2	1.75	1300	14500	1.0	NO	1939
2724049185	175000.0	3	1.75	1430	4920	1.0	NO	1957
7891600260	175000.0	2	1.00	660	5000	1.0	NO	1915
3395800660	190000.0	3	1.00	1640	8100	1.0	NO	1939
6303400290	170000.0	2	1.00	860	8636	1.0	NO	1924
723049158	135000.0	4	1.00	1460	18599	1.5	NO	1940
3365900465	170000.0	3	1.50	1370	10176	1.0	NO	1947

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
7520000520	232000.0	2	1.00	1240	12092	1.0	Unk	1922
4006000423	230000.0	4	1.00	1870	14703	1.5	Unk	1928
723049326	104950.0	2	1.00	1170	8254	1.0	NO	1949
2690600005	162500.0	2	1.00	760	6141	1.0	NO	1920
3330501645	260000.0	3	1.00	1150	3090	1.0	NO	1910
3826000560	173000.0	2	1.00	1740	8100	1.0	NO	1947
123039604	102500.0	2	1.00	820	4320	1.0	NO	1937
3163600076	152275.0	1	1.00	1020	6871	1.0	NO	1937
7520000695	151100.0	3	1.00	840	4495	1.0	NO	1921
9505100035	200000.0	2	1.00	1250	8520	1.0	Unk	1928
8143600015	348140.0	2	1.50	2060	10880	1.0	NO	1924
8151600101	115000.0	2	1.00	790	7252	1.0	NO	1930
3332000615	310000.0	3	1.00	1330	3740	1.5	NO	1903
7889600230	114000.0	2	1.00	730	5200	1.0	Unk	1928
2658000115	190000.0	1	1.00	720	4800	1.0	NO	1914
8018600880	110000.0	2	1.00	800	15000	1.0	Unk	1927
4154300275	245000.0	2	1.00	990	4800	1.0	NO	1908
7129302555	260000.0	2	1.00	1410	5650	1.5	NO	1918
7950304075	225000.0	4	1.00	1150	6000	1.5	NO	1907
3336001316	160000.0	2	1.00	830	4500	1.0	NO	1920
4319200620	235000.0	2	1.00	1270	9182	1.5	NO	1917
1446400670	199950.0	3	1.50	1510	6600	1.0	NO	1938
3454800060	171800.0	4	2.00	1570	9600	1.0	Unk	1950
7129302095	265000.0	3	1.00	1122	6554	1.5	NO	1900
7135300026	160000.0	2	2.00	1040	4750	1.0	NO	1950
1721800470	230000.0	5	2.00	1930	6120	1.5	NO	1941
2114700540	366000.0	3	2.50	1320	4320	1.0	NO	1918
4364700730	280000.0	2	1.00	1880	7560	1.0	Unk	1919
3330500345	230000.0	2	1.00	1280	4635	1.0	NO	1917
2613200025	175000.0	2	1.00	1330	28270	1.5	NO	1925
7186800105	236500.0	4	1.00	2140	4217	1.5	NO	1925
1604600790	316000.0	2	2.00	860	3000	1.0	NO	1906
2853600155	110000.0	1	1.00	640	10280	1.0	NO	1920
7987400356	255000.0	2	1.00	1220	2500	1.0	NO	1910

id	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	yr_built
2414600400	210000.0	2	2.00	1190	7570	1.0	NO	1939
4302201045	150000.0	3	1.00	820	7680	1.5	NO	1910
7129301851	245000.0	2	1.00	1120	5650	1.0	NO	1904
7211401610	165000.0	3	1.00	1120	5000	1.0	NO	1917
7129304540	133000.0	5	2.00	1430	5600	1.5	NO	1947
9412900055	405000.0	3	1.75	2390	6000	1.0	NO	1908
6209000165	247500.0	4	1.75	2290	7765	1.0	NO	1936
3332000195	167500.0	3	1.00	760	3090	1.0	NO	1903
4356200210	153500.0	3	1.00	890	4810	1.0	NO	1910
1604602195	265000.0	5	1.75	1580	5292	1.0	NO	1913
7452500565	260000.0	3	2.00	2710	5000	2.0	NO	1951
2123049502	215000.0	3	2.00	1340	8505	1.0	Unk	1931
7243500015	275000.0	4	2.00	1720	5472	1.0	NO	1923
3330500705	197500.0	3	1.00	980	3090	1.5	NO	1903
1623049133	205000.0	4	2.00	2200	13320	1.0	NO	1944
985000900	198500.0	3	1.75	1520	7137	1.0	Unk	1932
8113101070	334900.0	4	1.75	2180	4066	1.5	NO	1911
3365900520	192500.0	3	1.00	1080	8580	1.5	NO	1900
3333002450	165000.0	1	1.00	850	8050	1.0	NO	1906
1895000260	207950.0	2	2.00	890	5000	1.0	NO	1917
1231000510	263000.0	3	1.75	1490	3800	1.0	NO	1913
7999600180	83000.0	2	1.00	900	8580	1.0	NO	1918
2114700115	291700.0	3	2.50	1970	4120	1.5	NO	1927
7972602490	220000.0	5	2.50	1760	10200	1.5	NO	1925
4006000281	227000.0	3	1.75	2380	12681	1.0	NO	1918
1443500305	194990.0	6	2.50	1560	7144	1.0	NO	1913
7347600490	245000.0	3	2.00	2040	13125	1.0	NO	1910
3365900462	265000.0	4	3.00	1730	7264	2.0	NO	1920
1703400585	325000.0	3	2.00	2330	4950	1.5	Unk	1900
3388110230	179000.0	4	1.75	1790	7175	1.5	NO	1900

Out of the homes in the 98188 zipcode, there was an average of 323,000 price increas, a minimum calculated value -35,000 loss and a max of 624,000 increase. The negative numbers are most likely prediction errors.

In [108…
```
X_comp.groupby(['zipcode',"org_grade"]).count()
```

Out[108…

zipcode	org_grade	org_value	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	y
98106	4	3	3	3	3	3	3	3	
	5	6	6	6	6	6	6	6	
	6	70	70	70	70	70	70	70	
98118	5	4	4	4	4	4	4	4	
	6	102	102	102	102	102	102	102	
98126	5	5	5	5	5	5	5	5	
	6	63	63	63	63	63	63	63	
98146	4	2	2	2	2	2	2	2	
	5	9	9	9	9	9	9	9	
	6	79	79	79	79	79	79	79	
98168	5	13	13	13	13	13	13	13	
	6	82	82	82	82	82	82	82	

In [109…
```
#Aggregating the table by the zipcodes
agg_zips = X_comp.groupby(['zipcode']).mean().round(-2)
agg_zips.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5 entries, 98106 to 98168
Data columns (total 14 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   org_value    5 non-null      float64
 1   bedrooms     5 non-null      float64
 2   bathrooms    5 non-null      float64
 3   sqft_living  5 non-null      float64
 4   sqft_lot     5 non-null      float64
 5   floors       5 non-null      float64
 6   yr_built     5 non-null      float64
 7   org_cond     5 non-null      float64
 8   org_grade    5 non-null      float64
 9   pred_value   5 non-null      float64
 10  new_cond     5 non-null      float64
 11  new_grade    5 non-null      float64
 12  value_diff   5 non-null      float64
 13  perc_diff    5 non-null      float64
dtypes: float64(14)
memory usage: 600.0 bytes
```

In [110…
```
#Looking at the sum of original and predicted values in the aggregating zip
agg_zips[["org_value", "pred_value"]].sum(axis = 1)
```

Out[110…
```
zipcode
98106    815300.0
98118    927500.0
98126    867000.0
98146    750200.0
98168    746400.0
dtype: float64
```

In [111…
```
#Average increase per zipcode
agg_zips[["value_diff","perc_diff"]]
```

Out[111…

zipcode	value_diff	perc_diff
98106	304100.0	100.0
98118	323500.0	100.0
98126	266500.0	100.0
98146	324300.0	200.0
98168	340300.0	200.0

In [112…
```
agg_zips[["org_value"]]
```

Out[112…

zipcode	org_value
98106	255600.0
98118	302000.0
98126	300300.0
98146	213000.0
98168	203100.0

Graphing Results

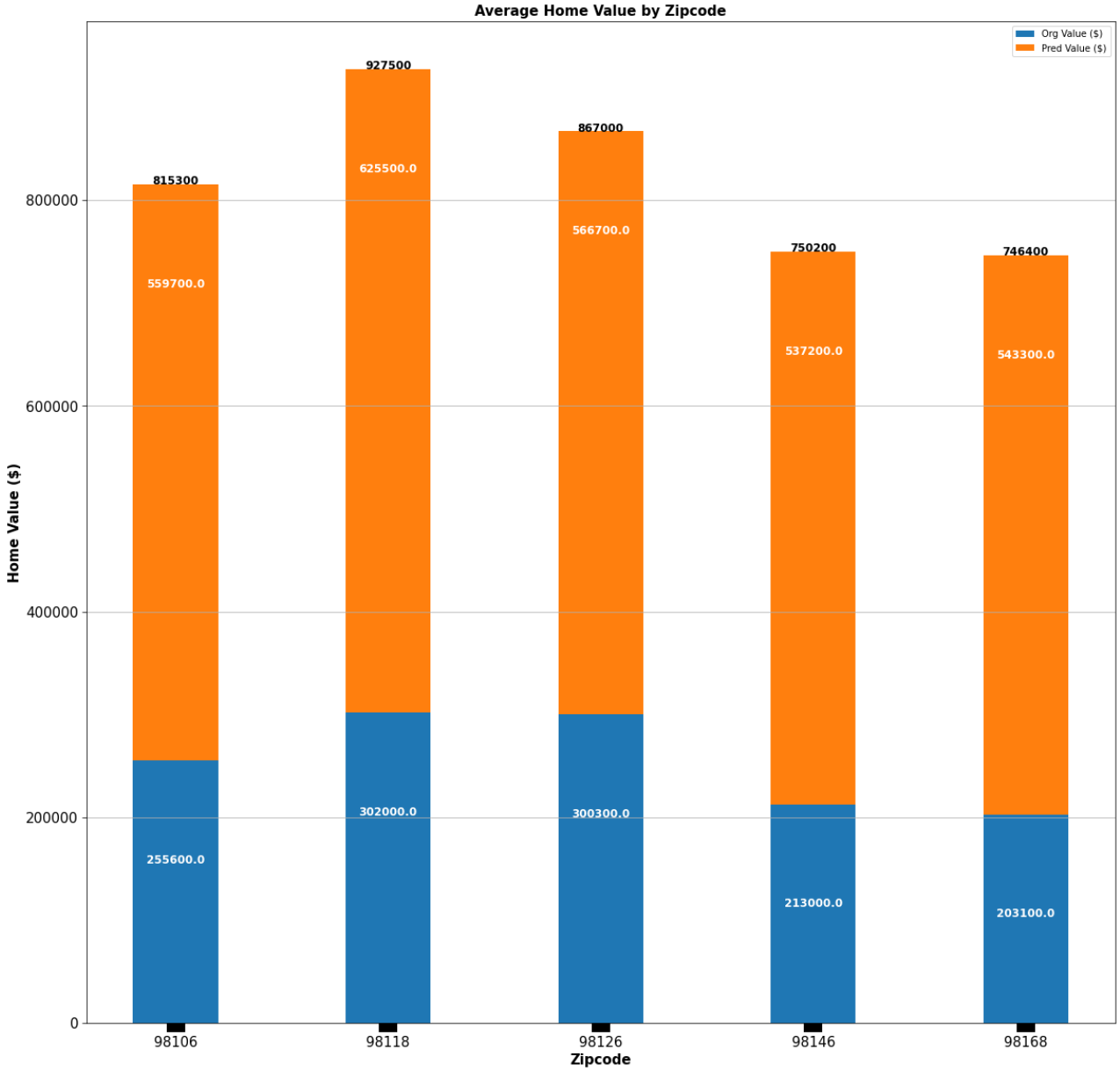
Stacked Bar Ref Stacked Bar Graph, Pythoncharts

Average Zipcode Home Value Comparison

Graphing the average home value by zipcodes

In [113…
```
#Graphing the stackedbars for Average Homevalue

#Import ploting library
from matplotlib import pyplot as plt

#Setting the plot frame, axis and size
fig, ax = plt.subplots(figsize = (20,20), facecolor = "white")

#Setting width of bars
width = .4
#Setting zipcodes as columns and strings.  Easier to have as strings than number
col = ["98106", "98118", "98126", "98146", "98168"]

#Creating two bar charts for bottom (pred value) and top stacks (original value)
p1 = ax.bar(col, agg_zips['org_value'],width = width,label='Org Value ($)')
p2 = ax.bar(col, agg_zips['pred_value'],width=width,  bottom=agg_zips['org_value
       label='Pred Value ($)')
```

```
#Setting labels on axis and titles
ax.tick_params(axis = "y",labelsize = 15, length=5)
ax.tick_params(axis = "x",labelsize = 15,  length=10, width = 20)
ax.set_ylabel("Home Value ($)", fontsize=15, fontweight = "bold")
ax.set_xlabel("Zipcode", fontsize=15, fontweight = "bold")
ax.set_title('Average Home Value by Zipcode', fontsize=15, fontweight = "bold")
ax.set_xticks(col)

#Adding gridlines on minor y axis
ax.grid("minor", axis ="y" )


#Calculating the total of old and pred value sum
#Seeting the offest where the label should be
y_offset = -4

#Calculation of total
zip_sums = agg_zips[["org_value", "pred_value"]].sum(axis = 1)
for i, zip_sum in enumerate(zip_sums):
    ax.text(col[i], zip_sum+ y_offset, round(zip_sum), ha='center',
          weight='bold',size=12)

#Putting value labels for each stack
y_offset = -100000
#For loop for evaluating each bar and placing the label there
#Patches are used to specify the bars
for bar in ax.patches:
  ax.text(
      # Put the text in the middle of each bar. get_x returns the start
      # so we add half the width to get to the middle.
      bar.get_x() + bar.get_width() / 2,
      # Vertically, add the height of the bar to the start of the bar,
      # along with the offset.
      bar.get_height() + bar.get_y() + y_offset,
      # This is actual value we'll show.
      round(bar.get_height()),
      # Center the labels and style them a bit.
      ha='center',
      color='white',
      weight='bold',
      size=12
  )

#Showing legend
ax.legend(prop={'size':10})


#Save image in folder
plt.savefig("images/ZipcodeAvg_HomeValue.png", dpi=99)
```

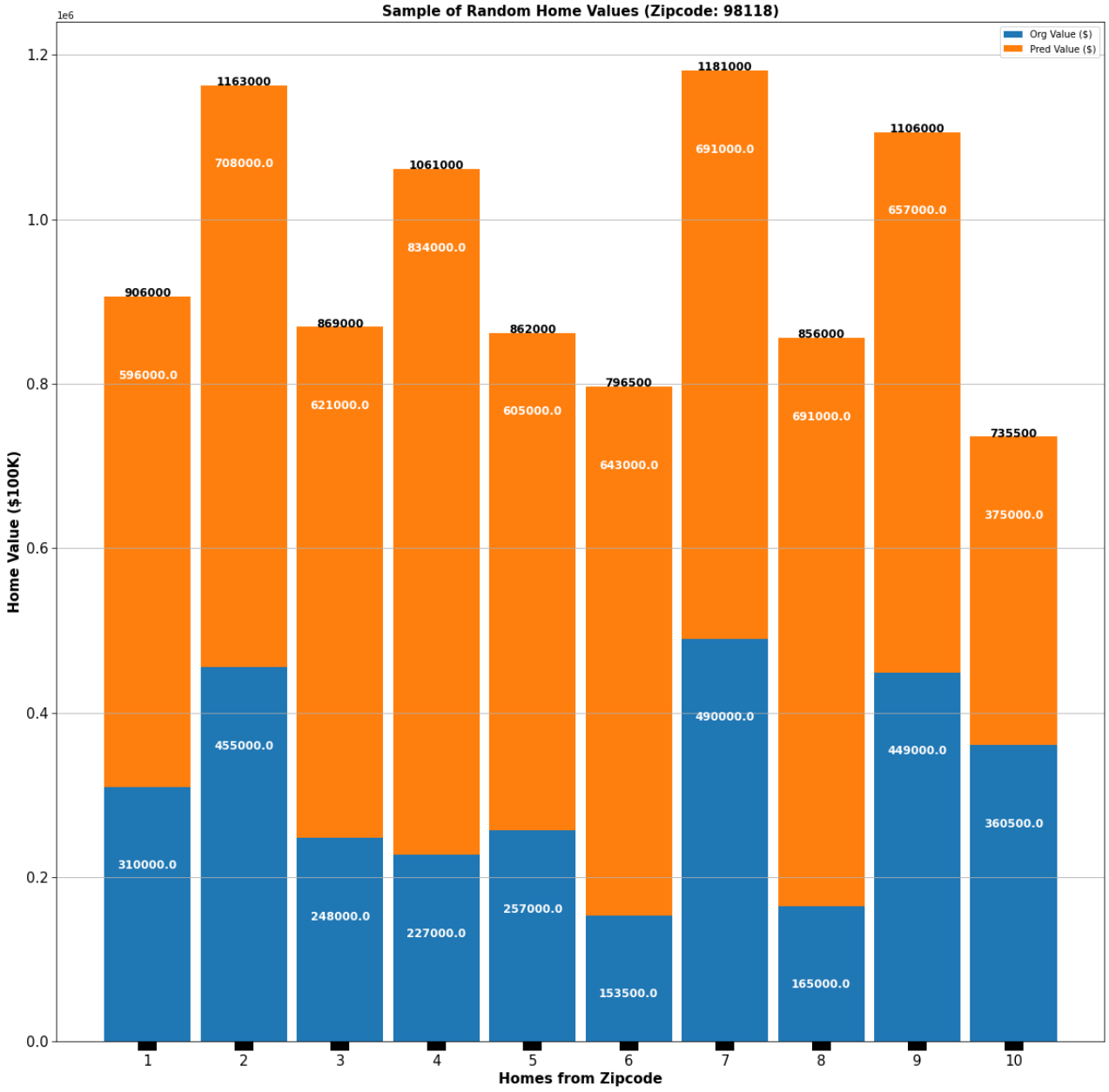


Zipcode 98118 Graphing Example

In [114…

```
#Graphing the stackedbars for a specific zipcode

#Import ploting library
from matplotlib import pyplot as plt

#Setting the plot frame, axis and size
fig, ax = plt.subplots(figsize = (20,20), facecolor = "white")

#Setting width of bars
width = .9

#Calling above function for zipcode 98118
gzip = filt_zcode(98118)

#Only using the top 10 (out of ~106) values
zip18 = gzip.head(10)

#Setting an arbitrary x-axis label
```

```
col = [1,2,3,4,5,6,7,8,9,10]

#Creating two bar charts for bottom (pred value) and top stacks (original value)
p1 = ax.bar(col, zip18['org_value'],width = width,label='Org Value ($)')
p2 = ax.bar(col, zip18['pred_value'],width=width,  bottom=zip18['org_value'],
       label='Pred Value ($)')

#Setting labels on axis and titles
ax.tick_params(axis = "y",labelsize = 15, length=5)
ax.tick_params(axis = "x",labelsize = 15,  length=10, width = 20)
ax.set_ylabel("Home Value ($100K)", fontsize=15, fontweight = "bold")
ax.set_xlabel("Homes from Zipcode", fontsize=15, fontweight = "bold")
ax.set_title('Sample of Random Home Values (Zipcode: 98118)',fontsize=15, fontwe
ax.set_xticks(col)

#Calculating the total of old and pred value sum
#Seeting the offest where the label should be
ax.grid("minor", axis ="y" )

#Total of old and pred values
y_offset = -4
zip_sums = zip18[["org_value", "pred_value"]].sum(axis = 1)
for i, zip_sum in enumerate(zip_sums):
    ax.text(col[i], zip_sum+ y_offset, round(zip_sum), ha='center',
          weight='bold',size=12)

#Putting value labels for each stack
#Seeting an offset of where to put the labels
y_offset = -100000
#For loop for evaluating each bar and placing the label there
#Patches are used to specify the bars
for bar in ax.patches:
    ax.text(
      # Put the text in the middle of each bar. get_x returns the start
      # so we add half the width to get to the middle.
      bar.get_x() + bar.get_width() / 2,
      # Vertically, add the height of the bar to the start of the bar,
      # along with the offset.
      bar.get_height() + bar.get_y() + y_offset,
      # This is actual value we'll show.
      round(bar.get_height()),
      # Center the labels and style them a bit.
      ha='center',
      color='white',
      weight='bold',
      size=12
  )

ax.legend(prop={'size':10})
#Save image in folder
plt.savefig("images/Zipcode18_HomeValue.png", dpi=99)
```



Regression Results

Model Results

The goal was to create a multilinear regression model that would be able to predict housing prices upon improvements. The model created to do this used an initial subset of data (from King County database) to process, train and test for this problem.

The final model was the fourth iteration. The intial model started as a base model linear regression model and increased in complexity with transformations (Log and One Hot Encoding) and filtering of features.

The final accuracy metrics are good enough to use the model for basic home value predictions.

RMSE Score Results:

The refinement of our model decreased the train and test RMSE scores by over 260K to ~.31. Applying the Log Transformer on our train and test target parameter "price" was the primary catalyst for this decrease. The price value was initial skewed and needed to be normalized.

The final model has a ~.31 RMSE. The closer RMSE to zero the better. The model increased its abilitity to accurately predict the target variable.

R2 Score Results:

The final model increased its R2 score by ~.15. The best base model score was .5 and the final test's model score ended with ~.66. As we refined the model, it increased its ability to account and explain for the variation. This was due primarily to the use of multiple features and the filtering of insignificant ones.

Linear Regression Model Assumptions:

The final model passed the linearity, normalization and homoescedasticity assumptions and failed the multicollinearity assumption.

For this particular problem we are using the model for predictions and inferencial uses. Therefore, the failure of multicollinearity was not a major roadblock this time. For inferencial use, a deeper evaluation on the most optimistic combinations of features to use for the model must be done.

Validation:

Our model development had a training, validation and test set. The test set that was intially split from the test set was segregated the entire model development. Using Kfold cross validation was a more robust validation method over the basic train/test split.

Model RMSE and R2 Scores:

Final Train Mean Squared Error: 0.31441867034646886
Final Test Mean Squared Error: 0.3079357313675995

Final Train Model Score: 0.6438531698069864
Final Test Model Score: 0.6565129767577557

Third Train RMSE: 0.3147718460539656
Third Test RMSE: 0.3075890371545622

Third Train Model Mean Score: 0.644673314060476
Third Validation Model Mean Score: 0.6414594822403191

Second Model Kfold Train Mean score: 0.6446907194384797
Second Model Validation Mean score: 0.6414405396108452

Baseline Models:

Train RMSE: 260172.0361161922
Test RMSE: 268864.35998011974

Kfold Train Mean Score: 0.4884772214299433
Kfold Validation Mean Score: 0.5000072841051805

Train/Test Split Train Model Score: 0.49091149233831743
Train/Test Split Validation Model Score: 0.494749423259338

Prediction Results

Using the the trained multi-linear regression model, home prices were able to be predicted for a subset of homes. Because this problem was to predict home values after improvements, homes were chosen that had a major space for improvement.

The intial dataset used was filtered by the top five zipcodes that have the most homes with a "**condition" of 3-Good or less & a "grade" of 6-Low Average or less**".

The chosen homes were then modified to have a condition of **4-Good** and a grade of **8-Good** while all the other features stayed the same. The modified condition and grades were chosen as an objective goal. The thought was to give home owner incentives to improve upon their property, even if they do not reach the desired criteria.

These predictions will serve as samples for the real estate agents to understand the benefit of real estate data analysis. Because home value can depend on the surrounding community, grouping the homes for predictions by zipcodes helps the agent target a specific community for an extra benefit.

Home Improvement Prediction Results

Across the five chosen zipcodes, they all resulted in several hundred thousand dollar increase and minimum 100% increase in value.

Average home value differences :

- Zipcode 98118: $ 323,500, %100 Increase
- Zipcode 98106: $ 304,100, %100 Increase
- Zipcode 98126: $ 266,500, %100 Increase
- Zipcode 98146: $ 324,300, %200 Increase
- Zipcode 98168: $ 340,300, %200 Increase

Specific zipcode example (98188) had the highest amount of homes (106) that met our criteria for needing improvement.

Zipcode 98188 home value differences:

- Average of $ 323,000 price increase
- Lowest change of $ 14,500 increase (35,000 loss was recorded, but most likely an outlier error).
- Highest change $624,000 increase.

Conclusion

Limitations

Stakeholders Audience

There were several major limitations. A lot of it stemmed from understanding the domain and making processing and analysis decisions from that knowledge.

- **Limited dataset:**
  - The full dataset was filtered and scoped due to limited resources. There were attributes (features) in the dataset that would have supported a more accurate and model.
  - Dataset does not take into account aesthietics. There are no features or pictures evaluating this aspect.
- **Unknown realistic "Condition" and "Grade" values:**
  - Lack of knowledge on knowing whata realistic increase in "condition" and "grade" would be from the baseline. All chosen instances in the dataset were increased to the same values. It may be unrealistic to have a home in poor condition and below minimum building stards actually increase to "good" in both.
- **Unknown affects on other variables:**
  - The "condition" and "grade" feeatures wree the only ones modified from the intial dataset. Home improvements would definitely affect those features. However, home improvements done on a home might also affect features like sqft_living, floors, bathrooms or bedrooms in some way.
- **Communal effects:**
  - The predictions were based on the modication of specific features for an individual home. The predicitons did not take into account how the home improvements would affect the surrounding comparible homes. This is common task done in real estate.
- **Model approach fit for specific problem:**
  - The model developed was for a specific problem. Understanding how home improvement how affect home values and the amount of the difference. Other usecases and problems were not taken into account. There may be insights to game from the model and predictions that
- **Time/Resources:**
  - Because of limited time and skillset a "good enough" model was delivered. The datasets, techniquesm model robustness had to be scoped.
- **Actual market culture:**

    - The dataset used wasn't the most current. These last two years have had major shifts in the supply and demand of homes. Therefore a shift in the home value estimates shift as well.
- **External effects:**
    - The model and predictions did not account for external effects that were not attributes of the home and its property.
- **Data scientist skillset:**
    - As a new data scientist the the model robustness is limited due to experience and skills.

Data Science Audience

- **Dataset limitations:**
    - A subset of the dataset was used. Using the whole dataset would have been would have given more choices to choose from in terms of correlation
    - The feature "waterfront" only had values of "Unknown", "No" and "Yes". It does not say what the waterfront view is, if the feature had a value of "Yes"
- **Linear Regression Assumptions:**
    - The multicollinearly assumption was the only assumption that clearly failed. Because of the resource and dataset scope limitations this aspect was not changed to fix this issue. If we wanted inferential analysis this would have to be fixed.
- **Limited in robustness:**
    - Not having the domian knowledge, a model that allowed for more flexibility in feature combinations and value valuations would have been beneficial.
- **Iterations:**
    - There were only 4 iterations done to create a prediction model. There were a lot of changes and approaches that would have required quite a lot more iterations to refine the model from the previous feedback received.
- **Feature combinations:**
    - The feature combinations were chosen from filtering the intial set down. Using the statsmodel p-value and the VIF. There wasn't a robust method that optimizes the best combinations for the model.
- **Model error and scores:**
    - It is not fully understood if the errors of a RMSE .3 R2 .66 are acceptable for the specific problem and solution approach. Also with these errors and score, it is still not understood what the cause of the of there errors is.

Recommendations:

- Choose and present incintives and vision of home improvement within a community. This incintivizes people and helps with accountability.
    - Choose a few zipcodes to try out and then offer feedback that would improve model accuracy or approach.

    - Focus on major areas with high needs to bring up. The communal effect will possibly increase prices even though the highest improvements may not have happened.
    - Present businessess(e.i. construction, remodeling) of the potential work to be done. Partnering with them and possibly offering discounts to the select communities would be a good incintive for communities to improve together
- Market to potential homebuyers (individuals and investors) of the potential return on investment. These homebuyers may potentially buy the homes before the improvements and then fix them up.
- Increase consultation with the data scientiest/analys to improve our domain knowledge. As both parties educate each other the model solution has a better chance at being more accurate and robust.
    - Feedback on realistic feature values after home improvement
    - Having examples and case studies of home improvements specifics would help give a realistic picture to all of the stakheholder supporting the predictions and work.