Final Project Submission

Please fill out:

- Student name: Shayan Abdul Karim Khan
- Student pace: self paced
- Scheduled project review date/time: Monday 21st November 2022
- Instructor name: Abhineet Kulkarni
- Blog post URL:

Problem Overview

Paragon Real Estate is a real estate agency with licensed operations in the King County Area. They help homeowners buy/sell homes.

Paragon wants to be able to advice it's clients about how home renovations might increase the estimated value of their homes, and approximately by what amount.

They want to help homeowners make smarter choices about investing in their propreties so that homeowners can understand whether a renovation will be helpful for their proprety valuable.

Business Questions

The business questions that will be explored in this analysis are as follows:

1. What kind of renovations increases the value?

```
 - This will give homeowners an understanding of which renovatio
ns to focus on to increase the value of their homes and which re
novations to avoid.
```

2. What is the impact to the value of the kind of renovations identified in the first question?

```
 - This will provide insight to a homeowner to understand what k
ind of renovations to prioritise and what kind of change can the
y expect.
```

Data Sources

We will be using the official King County House Sales dataset to conduct analysis and answer Paragon's **Business Questions.**

Data Understanding

Data Understanding

The dataset listed above has following characteristics discussed in this section:

- Contents and Features of the dataset
- Relevance of the Features to the Business Questions
- Relevant features of the datasets that will be used for analysis
- Limitations of the dataset
- Avenues of analysis that will be pursued

We will start by importing the appropriate python libraries to explore the datasets.

In [616]:
```
import pandas as pd #imports the pandas library as pd to work on databa
import sqlite3 as sql # imports the sqlite3 library to leverage sql wit
from pandasql import sqldf # imports pandas sql library
import matplotlib.pyplot as plt # importing matplotlib for visualizatic
%matplotlib inline
import numpy as np # imports the numpy library
import datetime as dt #import datetime module
import seaborn as sns #import seaborn
from collections import Counter #import Counter
import statsmodels.api as sm #import stats models
from statsmodels.stats.outliers_influence import variance_inflation_fac

#import scikit library functions
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.datasets import make_regression
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split, cross_validate, S
from sklearn.feature_selection import RFECV


#import scipy libraries
from scipy import stats as stats

#import plotly
import plotly.express as px
import plotly.graph_objects as go
```

Let's import the dataset and take an intial look at it. The King County House Sales dataset is stored as `kc_house_data.csv` in the `data` folder.

In [617]:

```
#import the dataset
init_data = pd.read_csv('data/kc_house_data.csv')

#take an initial look at the dataset
init_data.head()
```

Out[617]:

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

5 rows × 21 columns

Lets look at the overview of the data frame using the `.info()` function

In [618]:
```
1 init_data.info() # getting the overview info of the dataframe records
```

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

There are **twenty-one** columns in this dataset.

We will use the `columns_names.md` file in the `data` folder to get insights into what information these columns provide us.

We will go through the columns and try to deduce whether the information is useful enough to include as a feature in our analysis.

- `id` : These are the unique identifiers of the houses. We can use this as the index to our dataframe.

- `date` : This column contains the date that the specific house was sold on. This column might be important depending on how much variation it has. Lets take a look at the range of years that we have available.

In [619]:
```
#convert string object to datetime
dt = pd.to_datetime(init_data['date'])

#convert to an array of only years
dt_yr = pd.DatetimeIndex(dt).year

#extract min and max year values
print('The oldest data we have available is for', dt_yr.min())
print(' ')
print('The newest data we have available is for', dt_yr.max())
```

```
The oldest data we have available is for 2014

The newest data we have available is for 2015
```

Looks like we have data only for **2014** and **2015**.

This means that our predictions will be lacking data from recent years and predictions will not be taking into account inflation and recent real estate market impacts.

We do not have ample data from varying years to give us a confidence on how house prices vary by months or years. Therefore, we will drop this column also. Also, This doen't tell us anything about what kind of renovations to pursue therefore we will **ignore** this feature.

- `price` : This is the sale price of the house. This is the most important data for analysis. This data is **float** type.

- `bedrooms` : This is the number of bedrooms in a house. Intuitively, we know that bedrooms are an important factor in the value of the house. We will use this data to understand how changing the numbeer of beedroms impacts the house value.

- `bathrooms` : Similar to bedrooms, bathrooms are also an integral part of a house and intuitively, we know that they can play a part in the buying of a house therefore we will use this column for analysis. Interestingly, this dataset is a float. We know that there are 4 kinds of bathrooms: quarter bath, half bath, three-quarter bath, and a full bath. Lets make sure that the float data we have here is only in increments of 0.25.

In [620]:
```
#get the unique values and value counts in the bathroom columns
init_data['bathrooms'].value_counts()
```

Out[620]:
```
2.50    5377
1.00    3851
1.75    3048
2.25    2047
2.00    1930
1.50    1445
2.75    1185
3.00     753
3.50     731
3.25     589
3.75     155
4.00     136
4.50     100
4.25      79
0.75      71
4.75      23
5.00      21
5.25      13
5.50      10
1.25       9
6.00       6
0.50       4
5.75       4
6.75       2
8.00       2
6.25       2
6.50       2
7.50       1
7.75       1
Name: bathrooms, dtype: int64
```

Through visual inspectioin, it is clear that the float data meets our knowlede of the 4 types of bathrooms. Lets move onto the other columns.

- `sqft_living` : This is the square footage of living space in the house. The size of the habitable area can play a big part in the price of a house. It will be insteresting to deevelop insights into how changing the living space square footage can impact the value of a house.

- `sqft_lot` : This columns contains the total siz of the property in square footage. Similar to the habitable area, the total size of the property is also an impoortant factor to consider when deducing house prices. Nonetheless, it is not something that can be changed through renovations. Therefore, we will **ignore** this column.

- `floors` : This is data on how many levels a house has. Generally, floors are also an important factor in a house price. Intuitively, this feature is tied together with `sqft_living` but it will be important to investigate how changes in the number of floors impacts the value of the house. Interestingly, this data is **float** type. We know from domain knowledge that there are full floors and half floors. Lets investigate to make sure that the data meets our domain knowledge assumptions.

In [621]:
```
#extract unique value counts
init_data['floors'].value_counts()
```

Out[621]:
```
1.0    10673
2.0     8235
1.5     1910
3.0      611
2.5      161
3.5        7
Name: floors, dtype: int64
```

Looks like we either have full floors and half floors therefore this data should be good. We can move onto investigating the remaining columns.

- `waterfront` : This column contains information on whether a house is on a waterfront. Watrefront properties are generally known to be more valuable but this isn't something that can be changed through renovations therefore it won't help us answer the business questions. We will **ignore** this column in our analysis.

- `view` : This is the quality of view from the house. The quality of view is highly subjective to individual preferences and we don't have information on the basis for the quality. Also, the quality of view isn't something that can be renovated therefore we will **ignore** this column.

- `condition` : This data shows how good the overall condition of the house is. It is related to maintenance of the house. Intuitively, we know that the condition of the house plays a big part in the house value. This is also something that can be improved through renovations. Lets investigate what kind of data this columns contains.

In [622]:
```
#extract unique value counts
init_data['condition'].value_counts()
```

Out[622]:
```
Average      14020
Good          5677
Very Good     1701
Fair           170
Poor            29
Name: condition, dtype: int64
```

Looks like these are categories stored in string format. We can process these later and convert them to integers to make them easire to work with. The codes respective to these descriptions listed on the King Coounty Assessor website is as follows:

1 = Poor- Worn out. Repair and overhaul needed on painted surfaces, roofing, plumbing, heating and numerous functional inadequacies. Excessive deferred maintenance and abuse, limited value-in-use, approaching abandonment or major reconstruction; reuse or change in occupancy is imminent. Effective age is near the end of the scale regardless of the actual chronological age.

2 = Fair- Badly worn. Much repair needed. Many items need refinishing or overhauling, deferred maintenance obvious, inadequate building utility and systems all shortening the life expectancy and increasing the effective age.

3 = Average- Some evidence of deferred maintenance and normal obsolescence with age in that a few minor repairs are needed, along with some refinishing. All major components still functional and contributing toward an extended life expectancy. Effective age and utility is standard for like properties of its class and usage.

4 = Good- No obvious maintenance required but neither is everything new. Appearance and utility are above the standard and the overall effective age will be lower than the typical property.

5= Very Good- All items well maintained, many having been overhauled and repaired as they have shown signs of wear, increasing the life expectancy and lowering the effective age with little deterioration or obsolescence evident with a high degree of utility.

- `grade` : This is the overall grade of the house which is related to the construction and design of the house. The Kings County Assessor provides official information on these grades which is listed below:
  - Grades run from grade 1 to 13. Generally defined as:
    - 1-3 Falls short of minimum building standards. Normally cabin or inferior structure.
    - 4 Generally older, low quality construction. Does not meet code.
    - 5 Low construction costs and workmanship. Small, simple design.
    - 6 Lowest grade currently meeting building code. Low quality materials and simple designs.
    - 7 Average grade of construction and design. Commonly seen in plats and older sub-divisions.
    - 8 Just above average in construction and design. Usually better materials in both the exterior and interior finish work.
    - 9 Better architectural design with extra interior and exterior design and quality.
    - 10 Homes of this quality generally have high quality features. Finish work is better and more design quality is seen in the floor plans. Generally have a larger square footage.
    - 11 Custom design and higher quality finish work with added amenities of solid woods, bathroom fixtures and more luxurious options.
    - 12 Custom design and excellent builders. All materials are of the highest quality and all conveniences are present.
    - 13 Generally custom designed and built. Mansion level. Large amount of highest quality cabinet work, wood trim, marble, entry ways etc.

These grade descriptions tell us that it is possible to jump into higher grades through reenovations and house improvements. This would be an interesting feature to investigate to develop insights for the user.

Lets investigate whether the values in this column match up with the official information.

In [623]:
```
#extract unique value counts
init_data['grade'].value_counts()
```

Out[623]:
```
7 Average        8974
8 Good           6065
9 Better         2615
6 Low Average    2038
10 Very Good     1134
11 Excellent      399
5 Fair            242
12 Luxury          89
4 Low              27
13 Mansion         13
3 Poor              1
Name: grade, dtype: int64
```

The data matches up with the official information. We can do some pre-processing to make it easier to work with this data which we'll explore later.

One thing to note is that from the King County assessor website, we know that `condition` is relative to `grade`. Lets use the groupby function to inspect the descriptive values of the two features to understand how they are related. Since the `condition` column is more generalized, we'll use that as the primary grouper and see what range of `grade` values does it cover.

In [624]:
```
#setting up the pivot table
init_data.groupby(['condition','grade'])['grade'].count()
```

Out[624]:
```
condition  grade
Average    10 Very Good      921
           11 Excellent      332
           12 Luxury          73
           13 Mansion         11
           4 Low              12
           5 Fair            100
           6 Low Average    1035
           7 Average        5229
           8 Good           4266
           9 Better         2041
Fair       10 Very Good        2
           4 Low               4
           5 Fair             15
           6 Low Average      59
           7 Average          75
           8 Good             13
           9 Better            2
Good       10 Very Good      156
           11 Excellent       56
           12 Luxury          13
           13 Mansion          2
           4 Low              10
           5 Fair             84
           6 Low Average     685
           7 Average        2831
           8 Good           1394
           9 Better          446
Poor       4 Low               1
           5 Fair              9
           6 Low Average      11
           7 Average           6
           8 Good              2
Very Good  10 Very Good       55
           11 Excellent       11
           12 Luxury           3
           3 Poor              1
           5 Fair             34
           6 Low Average     248
           7 Average         833
           8 Good            390
           9 Better          126
Name: grade, dtype: int64
```

We can see that every level of `condition` has multiple `grade` types. This tells us that a higher grade doeesn't necessarily mean a better condition. For instance, we see that there are **mansions** that are in **Average** conditions while there are also buildings likee cabins that are categorized as **poor** grade but are in **Very Good** condition. Therefore, we have to look at both of these variables during analysis.

As a general understanding, `condition` is related to the maintenance of the house while `grade` is more concerned with the architectural aspects of a house. Renovations to improve on the lack of maintenance is a lot easier financially as compared to architectural changes.

- `sqft_above` : This is square footage of house apart from basement. With the total square feet already available, it would not be extra insightful to look at the separate square footage. Therefore, we will **ignore** this column.

- `sqft_basement` : This is square footage of the basement. Similiar to the previous column, this won't add major insights for us with the total square footage already available. Therefore, we will **ignore** this column.

- `yr_built` : This column contains the year when the house was built. This is important information for estimating the house value because the age of a house can both be a pro or a con depending on the type of buyer and type of property. Nonetheless, this is not something that we can change or renovate therefore it will not providee valuable insights to answer the business questions. Therefore, we will **ignore** this column.

- `yr_renovated` : This is the year that the house was renovated in. Without more information on what kind of renovation took place, this data can be highly misleading because reenovations can be major overhaul or smaller improvements. For this reason, we will **ignore** this data.

- `zipcode` : This is the Zip Code that the property is located in which is used by the United States Postal Service. Zipcodes play a big part in establishing the value of a house because of different factors like rate of crime, school quality, etc. Nonetheless, this isn't something that can be changed. This can be something that is looked at separately in conjunction with the other factors at play in a neighbourhood to identify areas of most return on investment. But for the purpose of answering our business questions, wee will **ignore** this feature.

- `lat` : This is the latitude coordinates of the house. This is the same case as zipcode therefore we will **ignore** this feature also.

- `long` : This is the longitude coordinates of the house. Same casee as latitude, we will **ignore** this feature.

- `sqft_living15` : This is the square footage of interior housing living space for the nearest 15 neighbors. This isn't something that a homeowner looking to renovate has control over therefore we will **ignore** this data.

- `sqft_lot15` : This is the square footage of the land lots of the nearest 15 neighbors. Similar to the previous column, we will **ignore** this data because a homeowner doesn't have control over this feature.

Summary

Out of the **21 features** that this dataset has, we were able to identify **7 features** which can be helpful in answering the business questions of our client. Before we review th features that we will be using, lets look at the features we will be ignoring.

Columns/Features to be ignored

Allthough these features can be important in determining th value of the house, they are being ignored because of one or both of the reasons listed below:

1. They can't be changed through renovation
2. They are linked to other more insightful features which we have chosen instead.

The features to be ignored are:

- `date`
- `sqft_lot`
- `waterfront`
- `view`
- `sqft_above`
- `sqft_basement`
- `yr_built`
- `yr_renovated`
- `zipcode`
- `lat`
- `long`
- `sqft_living15`
- `sqft_lot15`

Columns/Features carried forward

The features we are carrying forward and howw they will help us answer the business questions are the following:

- `price` : This is the main target that we will bee investigating and the only sourcee to find out the house price

- `bedrooms` : This will help us identify whether increasing thee bedrooms increases the house value. This will also inform a homeowner of how important the number of bedrooms are for their house price.

- `bathrooms` : Similar to number of bedrooms, this will help homeowners identify whether adding bathrooms increases their house value.

- `sqft_living` : This will provide insights to the homeowner if increasing the living space by adding another floor or building an extension is a valuable renovation or not.

- `conditions` : This provides a qualitative and quantitative assessment of categorically grouping the condition of the house. This can provide valuable insights into how changing th condition can affect the value of the house. This can help homeowners identify exactly what kind of requirements they need to meet to jump to a better condition and help plan for more impactful renovations.

- `grade` : This is the more thorough categories of the quality of construction and the house. The condition data is related to it and as shown above, it will be important to investigate the relation of this in conjunction to to other variables in determining types and impacts of renovations.

- `floor` : This is the number of levels in a house. This is important to take forward since large architectural changes such as adding more floors can improve the gradee of the house and make way for more bedrooms, bathrooms and licing square footage.

Data Limitations

no way of identifying the cost effectiveness, detailed kind of renovation, difficulty of renovation, equity of improvment, location specific improvements. inflation impact, housing market trends, lack of more recent data

Our dataset has many limitations that are not being solved as a part of this analysis. Some of them are listed below:

- There is no data on how much it cost to make a new floor, bedroom, bathroom, etc. We also don't have information on how lavish the interior of the house is. Therefore there is no way to do a cost-benefit analysis to understand the return on investment and how long it would take for the increase in value to actualize.

- We also don't have definite information on how difficult the different types of renovations would be and how long they would take to accomplish. A quicker renovation can have a quick turn-around for customers who are looking to flip or sell houses quick.

- We also don't have detailed information on the zipcodes. Zipcodes and neighbourhoods play a big role during appraisals, and even minor renovations can have a major impact because of location. Also, bad neighbourhoods with high crime can have a ceiling to how much a housee's value can go up.

- We also don't have housing market and inflation trends to gauge how renovations are impacted by features that are outside of the customer's control.

- There is also a lack of data on the type of houses that we are looking at. Renovations like more bedrooms on a rental multi-family property in a university area would have a different impact on the house price as compared to other types of propeerties. The lack of this kind of data also severly hampers our ability to confidently explain the trend and correlation.

Avenues of Analysis

We are going to use multiple linear regression to understand how the different features impact house prices. The reason we want to use linear regression is because it allows us to reliably predict values and understand the impact of multiple variables on the target variable.

Linear Regression also gives us the power of infeerring the relationships of multiple variables to the target variable. Wee will use that insight to infer which kind of renovations would improve the house prices. Since we don't have a plethora of relevant features, we will avoid building a predictivee model.

Data Preparation

The following steps will be followed in preparing the data:

- Data Cleaning
- Data Processing for Regression

Data Cleaning

Lets start with selecting the columns we will be using.

In [625]:

```python
#copy dataframe to avoid changing the original data
data_cln = init_data.copy()

#note the columns to keep
col_kp = ['price','bedrooms','bathrooms','floors','sqft_living','condit

#seleect the column to use
data_cln = data_cln[col_kp]

#preview the info
print(data_cln.info())

#preview the data
data_cln.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 7 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   floors       21597 non-null  float64
 4   sqft_living  21597 non-null  int64  
 5   condition    21597 non-null  object 
 6   grade        21597 non-null  object 
dtypes: float64(3), int64(2), object(2)
memory usage: 1.2+ MB
None
```

Out[625]:

	price	bedrooms	bathrooms	floors	sqft_living	condition	grade
0	221900.0	3	1.00	1.0	1180	Average	7 Average
1	538000.0	3	2.25	2.0	2570	Average	7 Average
2	180000.0	2	1.00	1.0	770	Average	6 Low Average
3	604000.0	4	3.00	1.0	1960	Very Good	7 Average
4	510000.0	3	2.00	1.0	1680	Average	8 Good

Lets seperate grade descriptions and numbers so that we can process them seperately if we need to.

In [626]:

```
#split the grade column into 2 new columns
data_cln[['grade#', 'grade_desc']] = data_cln['grade'].str.split(' ', 1

#drop the grade column
data_cln.drop(['grade'],axis=1, inplace=True)

#convert grade# column to int64
data_cln['grade#'] = data_cln['grade#'].astype('int64')
#preview the data
print(data_cln.info())
data_cln.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 8 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   floors       21597 non-null  float64
 4   sqft_living  21597 non-null  int64
 5   condition    21597 non-null  object
 6   grade#       21597 non-null  int64
 7   grade_desc   21597 non-null  object
dtypes: float64(3), int64(3), object(2)
memory usage: 1.3+ MB
None
```

Out[626]:

	price	bedrooms	bathrooms	floors	sqft_living	condition	grade#	grade_desc

Using the `column_names.md` file and the King County Assessor website, lets add a column with the respective condition numbers for the condition descriptions in the `condition` column.

The building conditions are coded from 1-5. They are coded as follows:

1 = Poor

2 = Fair

3 = Average

4 = Good

5= Very Good

In [627]:
```
#check the values in the condition column
data_cln['condition'].value_counts()
```

Out[627]:
```
Average      14020
Good          5677
Very Good     1701
Fair           170
Poor            29
Name: condition, dtype: int64
```

In [628]:
```
#create a dictionary of the description and codes
cond_cd = {'Poor':1, 'Fair':2, 'Average':3, 'Good':4, 'Very Good':5}

#map evey condition record and compare with the the dictionary to creat
data_cln['cond_code'] = data_cln['condition'].map(lambda x: cond_cd[x])

#preview the data
print(data_cln.info())
data_cln.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 9 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   floors       21597 non-null  float64
 4   sqft_living  21597 non-null  int64
 5   condition    21597 non-null  object
 6   grade#       21597 non-null  int64
 7   grade_desc   21597 non-null  object
 8   cond_code    21597 non-null  int64
dtypes: float64(3), int64(4), object(2)
memory usage: 1.5+ MB
None
```

Out[628]:

	price	bedrooms	bathrooms	floors	sqft_living	condition	grade#	grade_desc	cond_code
0	221900.0	3	1.00	1.0	1180	Average	7	Average	3
1	538000.0	3	2.25	2.0	2570	Average	7	Average	3
2	180000.0	2	1.00	1.0	770	Average	6	Low Average	3
3	604000.0	4	3.00	1.0	1960	Very Good	7	Average	5
4	510000.0	3	2.00	1.0	1680	Average	8	Good	3

We will make the `price` column easier to work with. Lets change the scale to **$100,000**.

In [629]:
```
#divide the price column by 100000
data_cln['price'] = data_cln['price']/100000
```

In [630]:
```
#rename the price column
data_cln.rename(columns={'price':'price($100,000)'},inplace=True)

#preview the dataframe
data_cln.head()
```

Out[630]:

	price($100,000)	bedrooms	bathrooms	floors	sqft_living	condition	grade#	grade_desc	cond_c
0	2.219	3	1.00	1.0	1180	Average	7	Average	
1	5.380	3	2.25	2.0	2570	Average	7	Average	
2	1.800	2	1.00	1.0	770	Average	6	Low Average	
3	6.040	4	3.00	1.0	1960	Very Good	7	Average	
4	5.100	3	2.00	1.0	1680	Average	8	Good	

That should end the data cleaning process. We will now process this data to prepare it for regression analysis.

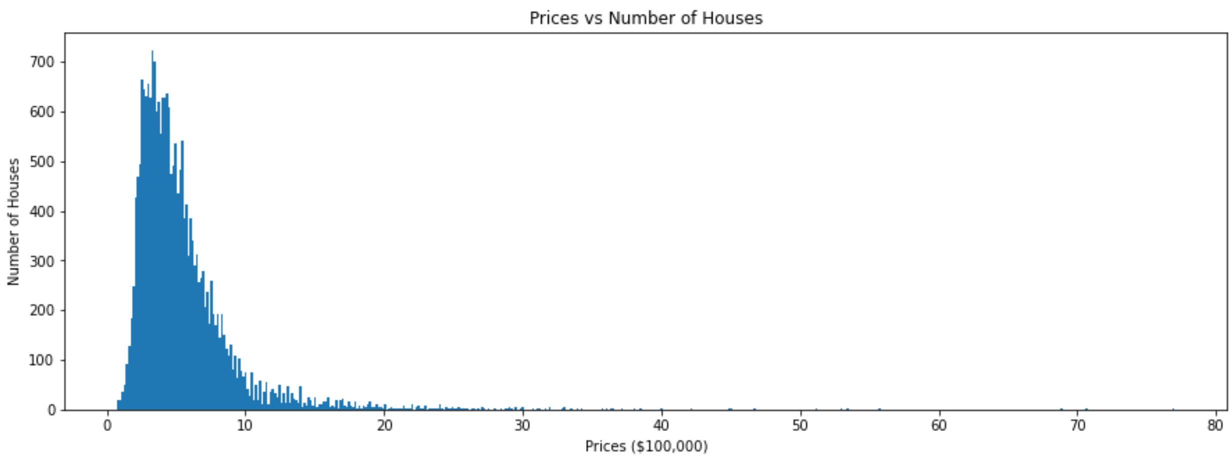
Data Processing for Regression Analysis

Lets start with separating the target variable, i.e `price` .

Lets take a look at the distribution of the target variable to get an understanding of how the data is distributed.

In [631]:

```
#defining the skeletion of the graph
fig, ax = plt.subplots(figsize=(15,5))

#plotting the histogram
ax.hist(data_cln['price($100,000)'],bins=500);

#seetting titles and axis labels
ax.set_title('Prices vs Number of Houses');
ax.set_xlabel('Prices ($100,000)');
ax.set_ylabel('Number of Houses');
```



We can see a right-skewed normal distribution with very few houses above **$1 million**.

Lets take a quick look at how many these are.

In [632]:

```
#filter for >1 million and measure the length
print('Number of houses sold for more than $1 million:', len(data_cln[d

print('The percentage of houses sold for than $1 million:',
      len(data_cln[data_cln['price($100,000)'] > 10])*100/len(data_cln)
```

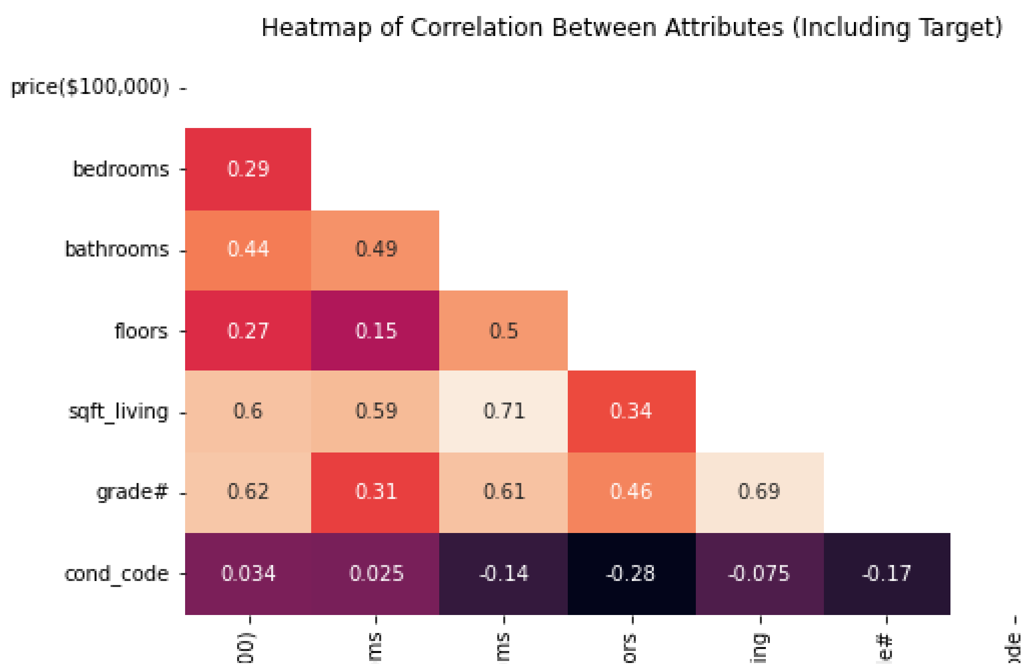
```
Number of houses sold for more than $1 million: 1458
The percentage of houses sold for than $1 million: 6.75093763022642
```

Approximately 6.75% is a considerably small number which wew wil drop so as not tp sleew oour model.

Lets take a quick look at how the correlation between the different variables and the target variable of price changes as we go over **$1 million** as compared to if wee stay under that.

In [633]:

```

#compute the correlation matrix
corr = data_cln[data_cln['price($100,000)']<=10].corr()

# Set up figure and axes
fig, ax = plt.subplots(figsize=(8, 8))

# 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,
    # Specifies that we want labels, not just colors
    annot=True,
    # Customizes colorbar appearance
    cbar_kws={"label": "Correlation", "orientation": "horizontal", "pad
)

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



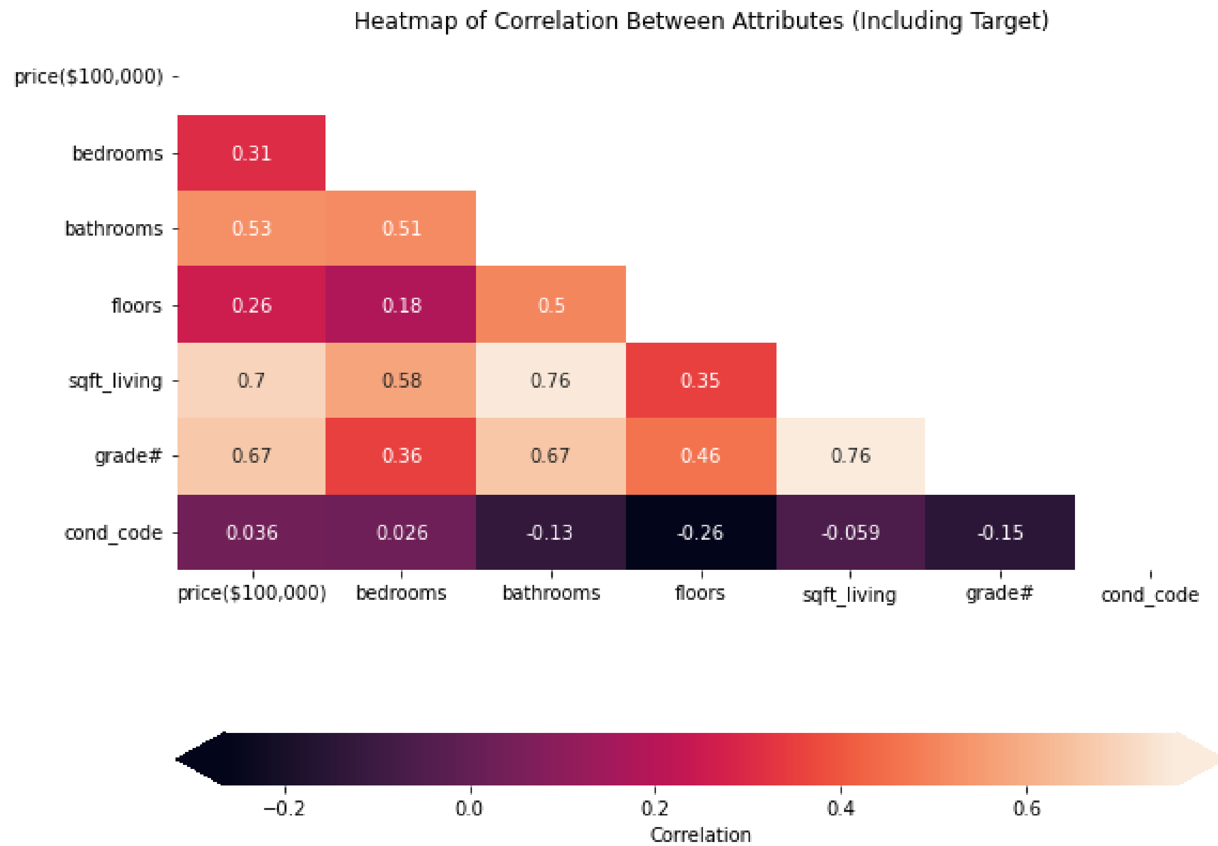
Interestingly, there is not a lot of difference except for correlatin of `price` and `grade#` .

The other trends remain the same. We can't accept to create a prefect model for all scenarios and we will try to fit our regression model so that it has the minimum error when evaluating prices for all price scenarios.

Now lets take a look at the correlation of our complete dataset.

In [634]:

```
# heatmap_data = pd.concat([y_train, X_train], axis=1)
corr = data_cln.corr()

# Set up figure and axes
fig, ax = plt.subplots(figsize=(10, 8))

# 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,
    # Specifies that we want labels, not just colors
    annot=True,
    # Customizes colorbar appearance
    cbar_kws={"label": "Correlation", "orientation": "horizontal", "pad
)

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



Based on thee plot abovee, `sqft_living` is the most correlated feature with `price`. We will use this feature to create the baseline model beforee iterating on it.

`grade#` is the second highest correlated feature.

`cond_code` has vrey little correlation with the `price` of a house which shows that this feature should be ignored.

Another interesting observation is the high correlation between `grade#`, `sqft_living`, and `bathrooms`. To avoid having a highlly collinear model, we will drop `bathrooms`, `bedrooms`.

We will investigate models with combinations of all these variables to generate the most appropriate regression model.

Before we move onto creating the model, we still need to investigate these variables for other propreties and transform them if needed.

Lets start with checking which variables are **categorical** variables after dropping `cond_code` and `grade#`. Thees ewere etwo columns we added eaerliree but even if these two features ar ecateegooricals, we can work with them using the descriptive analyses.

In [635]:
```
#drop cond_code and condition
data_cln = data_cln.drop(['cond_code','grade#','bathrooms','bedrooms'],
```

In [636]:
```
#preview data
data_cln
```

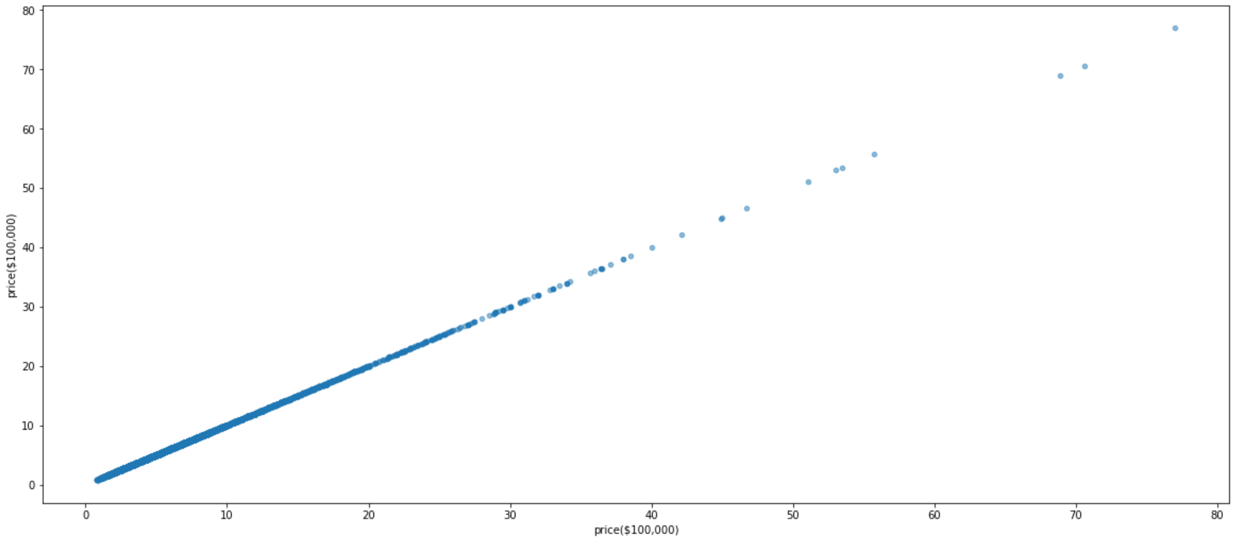
Out[636]:

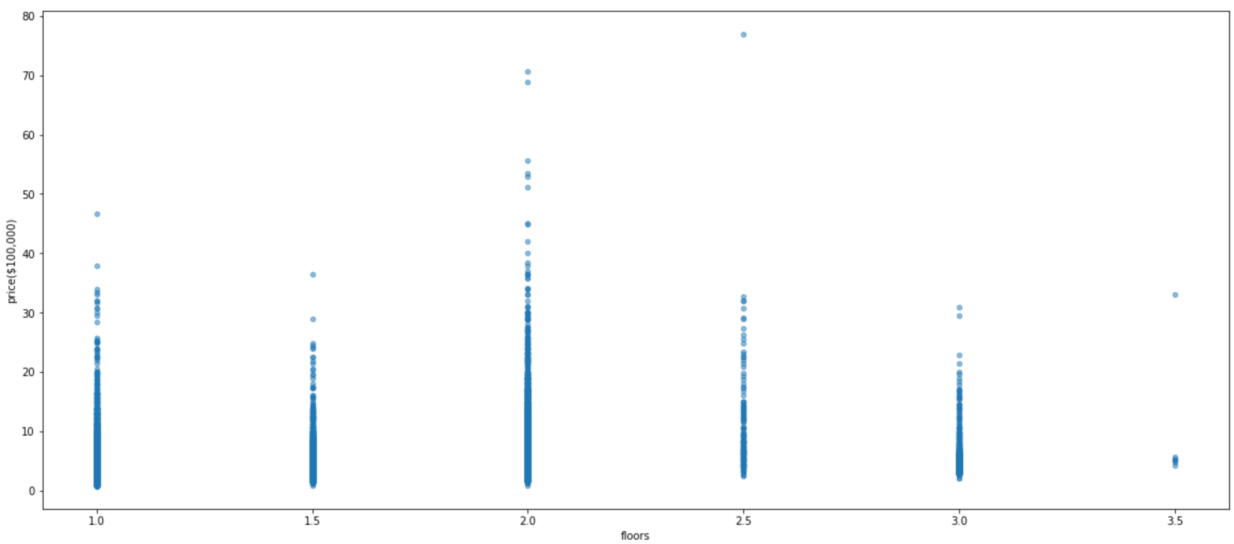
	price($100,000)	floors	sqft_living	condition	grade_desc
0	2.21900	1.0	1180	Average	Average
1	5.38000	2.0	2570	Average	Average
2	1.80000	1.0	770	Average	Low Average
3	6.04000	1.0	1960	Very Good	Average
4	5.10000	1.0	1680	Average	Good
...	...	...	...	...	...
21592	3.60000	3.0	1530	Average	Good
21593	4.00000	2.0	2310	Average	Good
21594	4.02101	2.0	1020	Average	Average
21595	4.00000	2.0	1600	Average	Good
21596	3.25000	2.0	1020	Average	Average

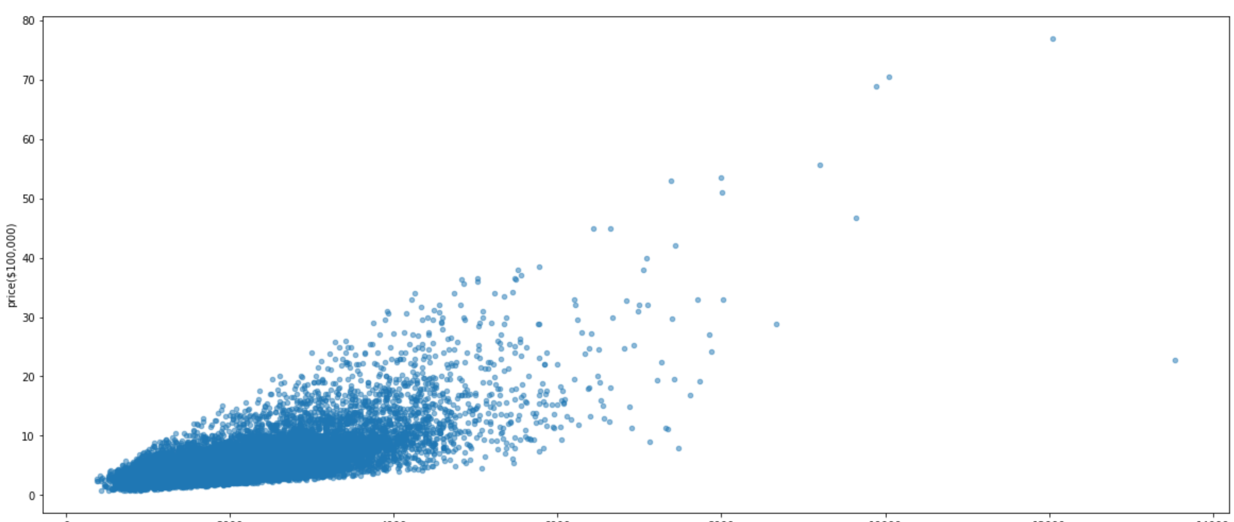
21597 rows × 5 columns

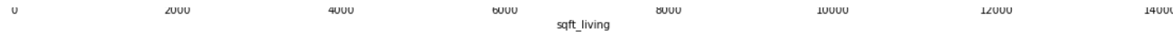
In [637]:

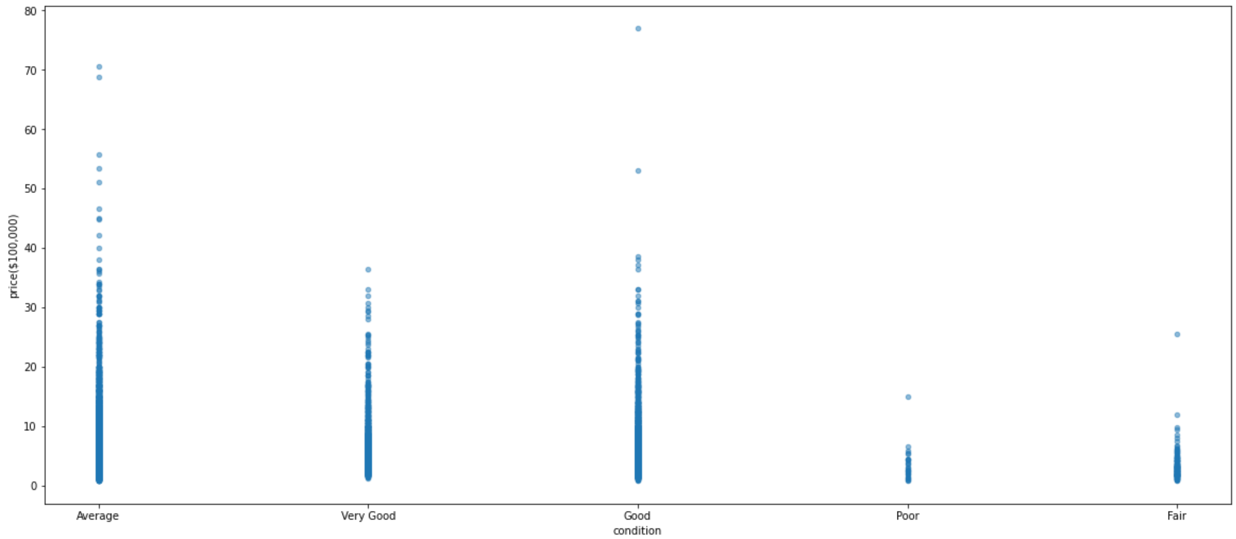
```
#Set up figure and axes
fig, axes = plt.subplots(nrows=5, ncols=1, figsize=(20, 50))

#plot the variables to check which ones are categoricals
for xcol, ax in zip(np.array(data_cln.columns),axes):
    data_cln.plot(kind='scatter', x=xcol, y='price($100,000)', ax=ax, a
```

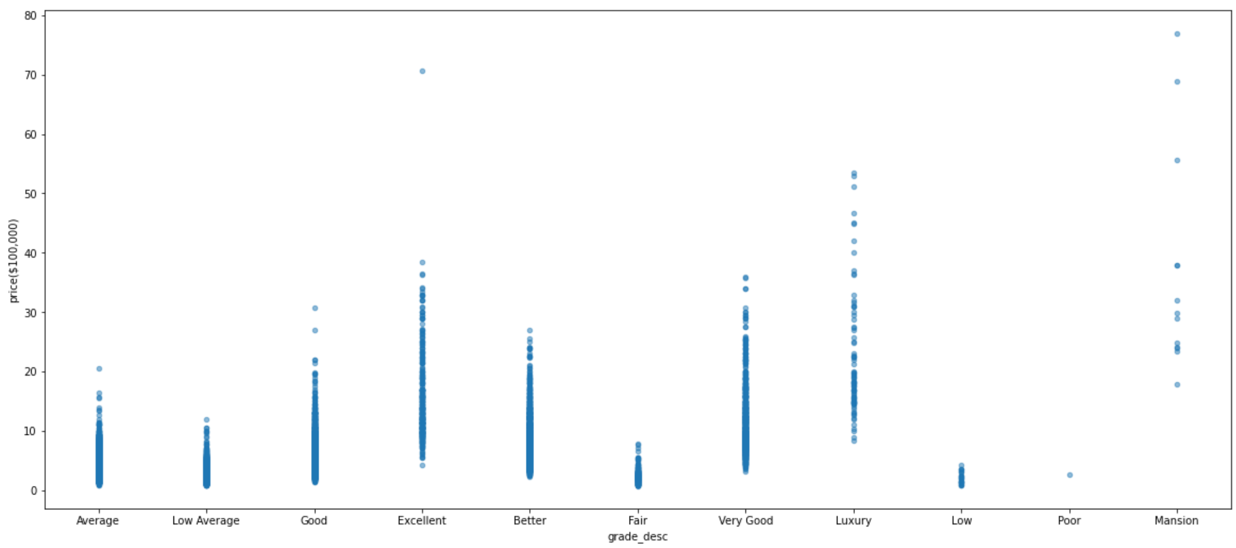












From visual inspection, we see that the features forming vertical plots are categorical variables. They are:

1. floors
2. grade
3. condition

The only continuous feature is `sqft_living` .

Since the values for `floors` is numeric, we won't have to encode them for our regression analysis but `grade` and `condition` has be to be encoded to use in our regression beause reegression modeels can only work with numeeric valuees.

Lets take a look at `sqft_living` to check if it will need any transformations.

Before we move onto investigating transformations, we should take a closer look at the graphs above. We can see that their are a lot of outliers in the `sqft_living` and `price` columns. We deduced earlier that we can ignore houses with prices above $1 million.

We can also see that there is very little data for sqft_living greeater than 6000 sqft. Including the houses with sqft areas bigger than 6000 sqft can potentially skew our data. Therefore we will ignore houses with sqft_living area greater than 6000 sqft.

In [638]:

```
data_x = data_cln[(data_cln['sqft_living']<=6000)]

data_x = data_cln[(data_cln['price($100,000)']<=10)]

```

In [639]:

```
#preview the data
data_x
```

Out[639]:

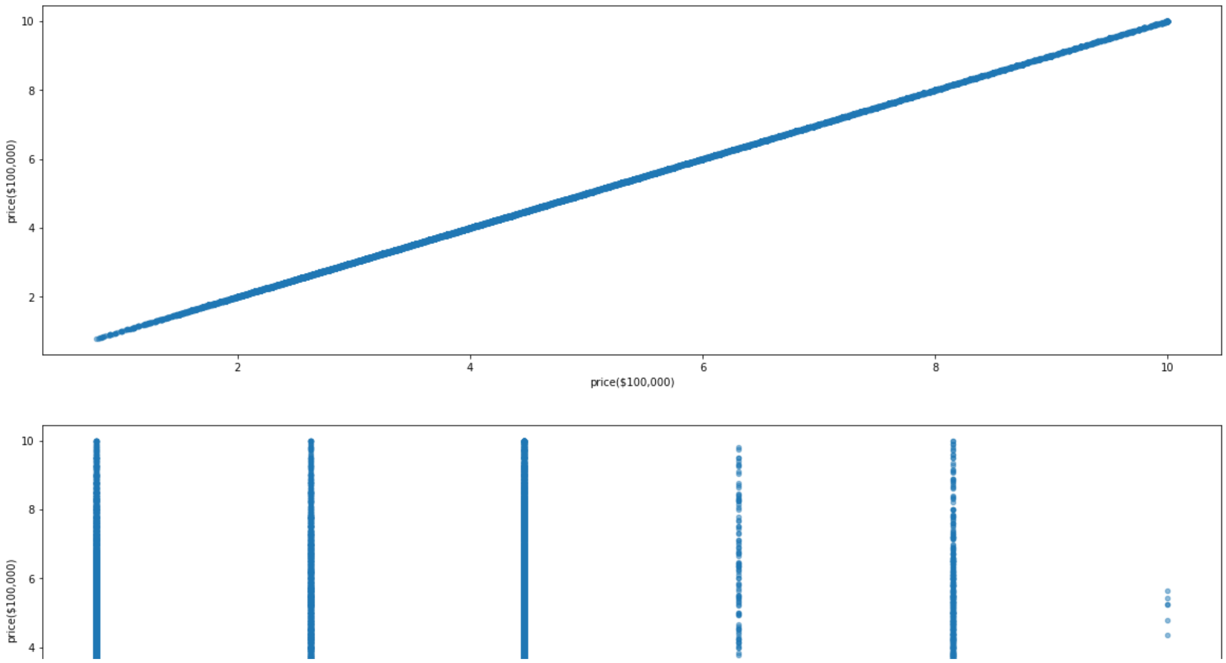
	price($100,000)	floors	sqft_living	condition	grade_desc
0	2.21900	1.0	1180	Average	Average
1	5.38000	2.0	2570	Average	Average
2	1.80000	1.0	770	Average	Low Average
3	6.04000	1.0	1960	Very Good	Average
4	5.10000	1.0	1680	Average	Good
...	...	...	...	...	...
21592	3.60000	3.0	1530	Average	Good
21593	4.00000	2.0	2310	Average	Good
21594	4.02101	2.0	1020	Average	Average
21595	4.00000	2.0	1600	Average	Good
21596	3.25000	2.0	1020	Average	Average

20139 rows × 5 columns

Filtering the two variables above basically means that according to our data source, it is rare to have houses that have living spacee areas greater than 6000 sqft or prices greater than a million dollars.

Lets recheck the graphs we created from earlier to see if things have improveed.

In [640]:
```
#Set up figure and axes
fig, axes = plt.subplots(nrows=7, ncols=1, figsize=(20, 50))

#plot the variables
for xcol, ax in zip(np.array(data_x.columns),axes):
    data_x.plot(kind='scatter', x=xcol, y='price($100,000)', ax=ax, alp
```



Lets investigate thee distribution of our features now. We, idally want our features to be normally distributed without skeewness. If they are not, we will transform that feature for regression analysis.

In [641]:
```
#scattr plot matrix
pd.plotting.scatter_matrix(data_x, figsize=[12, 12]);
```

We will have to transform `sqft_living` . We can see that it is a skewed normal curve.

Linear regression models perform better with variables distributed normally. Therefore, it will be in our beenefit to transform this variable.

We can use some transformation techniques to improve the normal distribution characteristics for this variable but before can start transforming our data we need to split it into test and train datasets to ensure that we are not contaminating our test and train tests, we will split the data first with 25% of the dataset assigned for testing purposes.

Before we split our data, we need to check if the descriptive categorical variables at least have 2 records for each category. This is to eensure that we can stratify without issues.

In [642]:

```
#check grade_desc values
data_x['grade_desc'].value_counts()
```

Out[642]:

```
Average         8951
Good            5873
Better          2224
Low Average     2033
Very Good        694
Fair             242
Excellent         92
Low               27
Luxury             2
Poor               1
Name: grade_desc, dtype: int64
```

Looks like wee have 2 categories with only 1 record each. We will drop these records so that we can stratify propeerly.

In [643]:

```
data_x = data_x[(data_x['grade_desc']!='Luxury')] #drop luxury
data_x = data_x[(data_x['grade_desc']!='Poor')] # drop poor
```

In [644]:

```
data_x['grade_desc'].value_counts() #recheck values
```

Out[644]:

```
Average         8951
Good            5873
Better          2224
Low Average     2033
Very Good        694
Fair             242
Excellent         92
Low               27
Name: grade_desc, dtype: int64
```

Lets check `condition` for the same thing and make changes accordingly.

In [645]:
```
data_x['condition'].value_counts() #check condition column
```

Out[645]:
```
Average      13074
Good          5347
Very Good     1519
Fair           168
Poor            28
Name: condition, dtype: int64
```

The condition column should be good. Lets move on to splitting our data. We will have to separate the target variable and the features before splitting them up.

In [646]:
```
pr_x = data_x['price($100,000)'] # create target variable dataset
data_prep = data_x.drop(['price($100,000)'],axis=1) #create feature dat

data_prep #preview

```

Out[646]:

	floors	sqft_living	condition	grade_desc
0	1.0	1180	Average	Average
1	2.0	2570	Average	Average
2	1.0	770	Average	Low Average
3	1.0	1960	Very Good	Average
4	1.0	1680	Average	Good
...	...	...	...	...
21592	3.0	1530	Average	Good
21593	2.0	2310	Average	Good
21594	2.0	1020	Average	Average
21595	2.0	1600	Average	Good
21596	2.0	1020	Average	Average

Lets split our data into train and test sets.

In [647]:
```
#use train test split to separate the data
X_train, X_test, pr_train, pr_test = train_test_split(data_prep,
                                                      pr_x,
                                                      random_state=42,
                                                      test_size=0.25,
                                                     stratify=data_prep
```

Lets apply log transformation to the `sqft_living` data and check the distribution again.

In [648]:
```

#copy the datasets
X_train_log = X_train.copy()
X_test_log = X_test.copy()

#transform the X_train data
X_train_log['sqft_living_log'] = X_train['sqft_living'].map(lambda x: n

#transform the X_test data
X_test_log['sqft_living_log'] = X_test['sqft_living'].map(lambda x: np.

```

In [649]:
```
X_train_log #preview the data
```

Out[649]:

	floors	sqft_living	condition	grade_desc	sqft_living_log
12455	2.0	2650	Average	Better	7.882315
6914	1.0	3040	Good	Good	8.019613
12510	2.0	2040	Average	Average	7.620705
3518	1.5	1550	Average	Average	7.346010
6472	1.0	2060	Good	Average	7.630461
...	...	...	...	...	...
16348	2.0	3260	Average	Better	8.089482
19047	1.0	1240	Good	Average	7.122867
15157	2.0	2570	Average	Good	7.851661
14481	1.0	1150	Average	Average	7.047517
17902	1.0	2260	Average	Better	7.723120

15102 rows × 5 columns

In [650]:
```
data_tr_log = X_train_log.drop(['sqft_living'],axis=1) #drop the sqft_l
data_test_log = X_test_log.drop(['sqft_living'],axis=1) #drop the sqft_
```

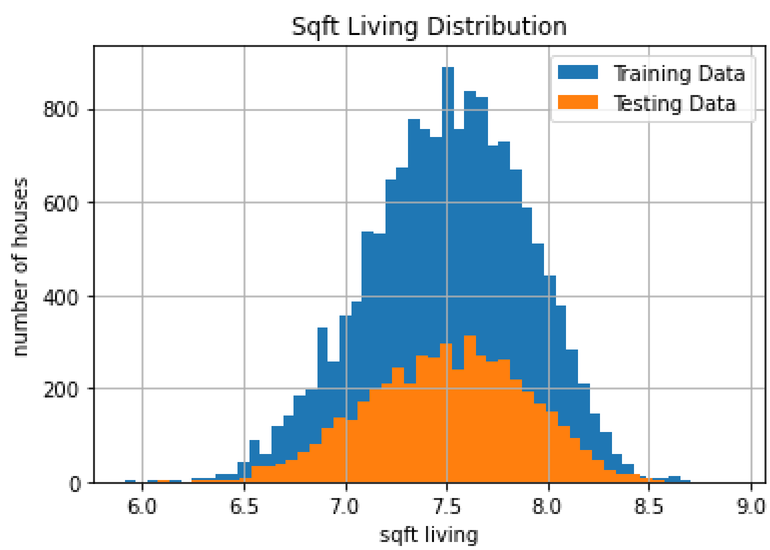
After log transformation lets scale the data also.

In [651]:
```
# scaler = StandardScaler() #set up scaler

# #scale train data
# data_tr_log['sqft_living_log_scaled'] = scaler.fit_transform(np.array

# #drop the log column
# data_tr_log = data_tr_log.drop(['sqft_living_log'],axis=1)


```

In [652]:
```
# #scale test data
# data_test_log['sqft_living_log_scaled'] = scaler.fit_transform(np.arr

# #drop the log column
# data_test_log = data_test_log.drop(['sqft_living_log'],axis=1)

```

Now lets check to make sure that the variable is normallly distributed

In [653]:
```
#plot the training and testing data

#set up figure and axis
fig,ax = plt.subplots()


#plot the data
data_tr_log['sqft_living_log'].hist(bins=50,label='Training Data');


data_test_log['sqft_living_log'].hist(bins=50,label='Testing Data');

#labels and titles
ax.set_title('Sqft Living Distribution');
ax.set_xlabel('sqft living');
ax.set_ylabel('number of houses');
ax.legend();
```



Now we need to encode the descriptive categorical variables

In [654]:
```
X_train_ohe = pd.get_dummies(data_tr_log[['condition','grade_desc']], d
X_test_ohe = pd.get_dummies(data_test_log[['condition','grade_desc']],
```

In [655]:
```
X_train_pro = pd.concat([data_tr_log, X_train_ohe], axis=1) #merege the
X_test_pro = pd.concat([data_test_log, X_test_ohe], axis=1) #merge the
X_train_pro #preview
```

Out[655]:

	floors	condition	grade_desc	sqft_living_log	condition_Fair	condition_Good	condition_Poor
12455	2.0	Average	Better	7.882315	0	0	0
6914	1.0	Good	Good	8.019613	0	1	0
12510	2.0	Average	Average	7.620705	0	0	0
3518	1.5	Average	Average	7.346010	0	0	0
6472	1.0	Good	Average	7.630461	0	1	0
...	...	...	...	...	...	...	...
16348	2.0	Average	Better	8.089482	0	0	0
19047	1.0	Good	Average	7.122867	0	1	0
15157	2.0	Average	Good	7.851661	0	0	0
14481	1.0	Average	Average	7.047517	0	0	0
17902	1.0	Average	Better	7.723120	0	0	0

15102 rows × 15 columns

In [656]:
```
X_train_mod = X_train_pro.drop(['condition','grade_desc'],axis=1) #drop
X_test_mod = X_test_pro.drop(['condition','grade_desc'],axis=1) #drop t
```

In [657]:
```
X_train_mod #preview
```

Out[657]:

	floors	sqft_living_log	condition_Fair	condition_Good	condition_Poor	condition_Very Good	grade_
12455	2.0	7.882315	0	0	0	0	
6914	1.0	8.019613	0	1	0	0	
12510	2.0	7.620705	0	0	0	0	
3518	1.5	7.346010	0	0	0	0	
6472	1.0	7.630461	0	1	0	0	
...	...	...	...	...	...	...	
16348	2.0	8.089482	0	0	0	0	
19047	1.0	7.122867	0	1	0	0	
15157	2.0	7.851661	0	0	0	0	
14481	1.0	7.047517	0	0	0	0	
17902	1.0	7.723120	0	0	0	0	

15102 rows × 13 columns

This brings our Data Preparation to an end. We will now move on to modelling the variables.

Modelling

We are going to start with a baseline model using the variable with the highest correlation to `price` .

During data processing, we found that `sqft_living` had the highest correlation. We will soleely use that variable to develop a baseline linear regression model.

A baseline model is used to compare against models that have multiple variables. We will use the most correlated featur eto the price to build it.

In [658]:
```
base_model = LinearRegression()
```

We will be using cross vaalidation to perform multiple separate train-test splits within the training daatasets.

This ensures that our Train dataset itself is spliced into different test sets. The r-squared value we get at th end is the mean threfore there is less llikelihood of getting an anomlous r-squared value.

In [659]:
```
#create a splitter function to slice the training set into test samples
splitter = ShuffleSplit(n_splits=10, test_size=0.2, random_state=0)

#Setting baseline scores
baseline_scores = cross_validate(
                                estimator=base_model,
                                X=np.array(X_train_mod['sqft_living_log
                                y=pr_train,
                                return_train_score=True,
                                cv=splitter
                                )

#print the etrain and validation scores
print('Train score:    ', baseline_scores['train_score'].mean())
print('Validation score:    ', baseline_scores['test_score'].mean())
```

```
Train score:     0.33965784294596224
Validation score:     0.34116912874681377
```

The values above are R-squared values. R-squared value is the proportion of the variation in the dependent variable that is predictable from the independent variable.

The Validation score is the more important of the two r-squared values because it tells us how our model actually performs.

Nonetheless, the Train and Validation Score are very similar for the baseline model but both of them are low.

This is a very weak model because of the low r-squared value. A value of 0.34 means that the model can only predict 34% of the variation in the dependent variable.

Therefore we have to try adding other features to the model to investigate if the preformamce improves.

We will add all of the relevant features we prepared and compare against the baselinee model.

In [660]:

```
#create the second model
s_model=LinearRegression()

#create th splitter function
splitter = ShuffleSplit(n_splits=10, test_size=0.2, random_state=0)

#run the model and record the scores
second_model_scores = cross_validate(
                                estimator=s_model,
                                X=np.array(X_train_mod),
                                y=pr_train,
                                return_train_score=True,
                                cv=splitter
                                )
print('Second Model including grade#')
print('Train score:   ', second_model_scores['train_score'].mean())
print('Validation score:   ', second_model_scores['test_score'].mean())
print()
print('Baseline Model with sqft_living_log')
print('Train score:   ', baseline_scores['train_score'].mean())
print('Validation score:   ', baseline_scores['test_score'].mean())
```

```
Second Model including grade#
Train score:    0.45429709503946525
Validation score:    0.45665411022858293

Baseline Model with sqft_living_log
Train score:    0.33965784294596224
Validation score:    0.34116912874681377
```

With a r-squared value of 0.45, this model can predict atleast 45% of the variation in the housing prices.

This is a much better result than the baseline model using all the features that we prepared. Nonetheless, the problem is that we are not optimizing our model to the most relevant features. In order to have the best resultls, we have the option of using the Recursive Feature Elimination process to improve hone in on the bese dfeatures to use.

In [661]:

```
#RFECV specific dataset
X_train_RFECV = StandardScaler().fit_transform(X_train_mod)

#set up model for RFECV
model_RFECV = LinearRegression()

#set up the selector and fit the model
selector = RFECV(model_RFECV, cv=splitter)
selector.fit(X_train_RFECV, pr_train)

# #print the rankings
# print(ft.ranking_)

#print which features does the model select
for index, col in enumerate(X_train_mod.columns):
    print(f"{col}:{selector.support_[index]}")

#setup an array with the best features
best_features = []
for index, col in enumerate(X_train_mod.columns):
    if selector.support_[index] == True:
        best_features.append(col)
print(best_features)

```

```
floors:False
sqft_living_log:True
condition_Fair:False
condition_Good:True
condition_Poor:False
condition_Very Good:True
grade_desc_Better:True
grade_desc_Excellent:True
grade_desc_Fair:True
grade_desc_Good:True
grade_desc_Low:False
grade_desc_Low Average:True
grade_desc_Very Good:True
['sqft_living_log', 'condition_Good', 'condition_Very Good', 'grade_desc_
Better', 'grade_desc_Excellent', 'grade_desc_Fair', 'grade_desc_Good', 'g
rade_desc_Low Average', 'grade_desc_Very Good']
```

In [662]:

```
#set up the final model
final_model = LinearRegression()

#fit the model using thee best features from RFECV analysis
final_model.fit(X_train_mod[best_features], pr_train)

#get the r-squared value
final_model.score(X_test_mod[best_features], pr_test)
```

Out[662]:

```
0.4639812796704821
```

Looks like if we ignore floors and a few other variables, we get a slight improvement in our model where we move from predicting 45% of the variation in the prices to 46%.

We will use this model as our finall because it is giving us the best r-squared results. Ideallly, we want the r-squared values to be as high as possible so that the confidence in the model predictions is high. Generally, an r-squareed value of 0.5 and higher is considered to be a strong model. With thee value of 0.46, we can say that oour model is close to being a strong model.

Lets set up a OLLS regression model and go throug the summary of the model to understand more about it.

In [663]:
```
1 sm.OLS(pr_train, sm.add_constant(X_train_mod[best_features])).fit().sum
```

Out[663]:

OLS Regression Results

Dep. Variable:	price($100,000)	**R-squared:**	0.455
Model:	OLS	**Adj. R-squared:**	0.454
Method:	Least Squares	**F-statistic:**	1398.
Date:	Fri, 18 Nov 2022	**Prob (F-statistic):**	0.00
Time:	16:14:35	**Log-Likelihood:**	-27029.
No. Observations:	15102	**AIC:**	5.408e+04
Df Residuals:	15092	**BIC:**	5.416e+04
Df Model:	9		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	-7.2455	0.307	-23.620	0.000	-7.847	-6.644
sqft_living_log	1.4976	0.042	36.048	0.000	1.416	1.579
condition_Good	0.3430	0.028	12.440	0.000	0.289	0.397
condition_Very Good	0.9390	0.046	20.383	0.000	0.849	1.029
grade_desc_Better	2.0459	0.046	44.621	0.000	1.956	2.136
grade_desc_Excellent	3.6396	0.179	20.308	0.000	3.288	3.991
grade_desc_Fair	-0.7042	0.111	-6.323	0.000	-0.923	-0.486
grade_desc_Good	0.8641	0.030	28.498	0.000	0.805	0.924
grade_desc_Low Average	-0.4958	0.044	-11.346	0.000	-0.581	-0.410
grade_desc_Very Good	2.8722	0.073	39.520	0.000	2.730	3.015

Omnibus:	590.217	**Durbin-Watson:**	1.993
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	662.094
Skew:	0.497	**Prob(JB):**	1.69e-144
Kurtosis:	3.253	**Cond. No.**	203.

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

These are the important observations to extract from the above data:

- All the **P>|t|** values are well under 0.05 therefore wee can be confident that the featurs selelcted are statistically significant.

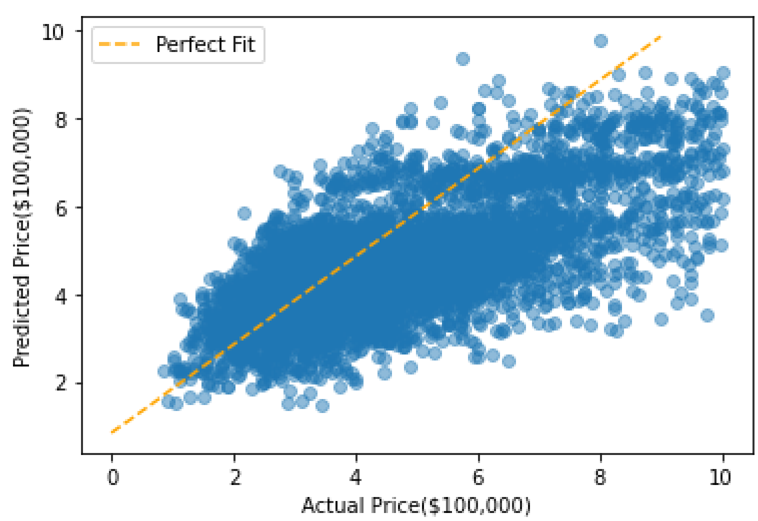
- The **Cond. No.** is an indication of multi-collinearity. This number should ideeally be bellow 3. We have to investigate the multi-colinearity assumption to see whether our model violates it or not.
- The **Skew** is 0 and **Kurtosis** is 3 for perfect Normal Distributions but we see a slight deviation so we will investigate the Normality Assumption too.

Lets check our assumptions for the model.

Linearity Assumption

In [664]:

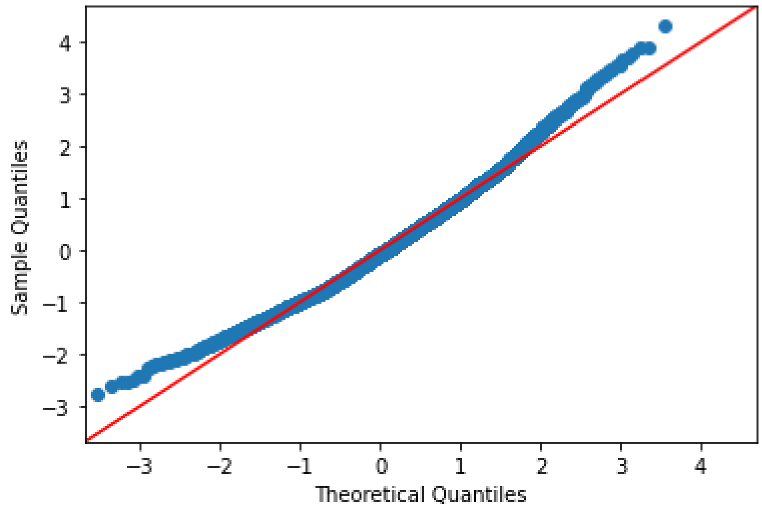
```
#predict the values for thee final model
preds = final_model.predict(X_test_mod[best_features])

#set up figure and axis
fig, ax = plt.subplots()

# #set perfecct linee
perfect_line = np.arange(pr_test.min(), pr_test.max())

# #plot the perfect line
ax.plot(perfect_line, linestyle="--", color="orange", label="Perfect Fi

#plot thee real vallues and the predicted values
ax.scatter(pr_test, preds, alpha=0.5)

#set the label
ax.set_xlabel("Actual Price($100,000)")
ax.set_ylabel("Predicted Price($100,000)")
ax.legend();
```



While our residuals are not prefectly linear, we can visually inspect that we are close to linearity. As the actual price tends to increase, our values are biased towards the lower end. Which means that the modele is under-predicting.

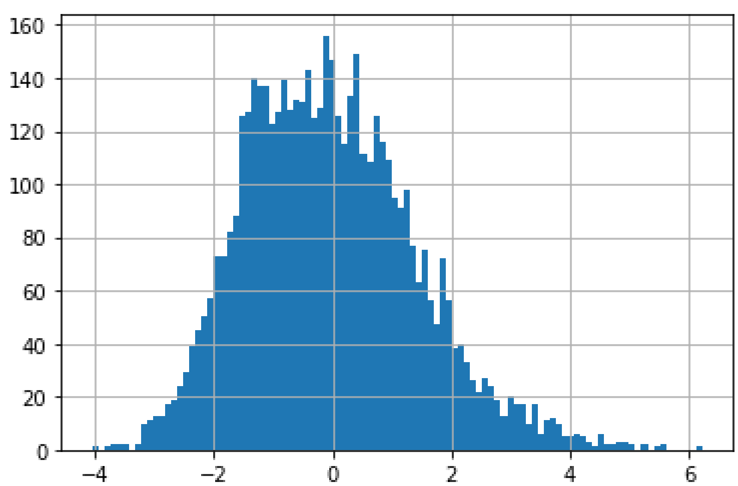
Normality Assumption:

In [665]:
```
import scipy.stats as stats
residuals = (pr_test - preds)
sm.graphics.qqplot(residuals, dist=stats.norm, line='45', fit=True);

```



Lets check a histogram distribution for better understanding.

In [666]:
```
residuals.hist(bins=100);

```



We have a bell shaped-curve which means the residuals are normally distributed with a slight right skewness as we saw from our OLS results. Wee know from this graph, that our model is under-predicting the prices, which is something we saw in the Linearity Assumption Test also.

Multi-Colinearity Assumption:

In [667]:
```
#multi-colinearity
X_f=X_train_log.copy()
a = X_train_mod[best_features]
vif = [variance_inflation_factor(a.values,i) for i in range(a.shape[1])]
pd.Series(vif, index=a.columns, name='VIF')
```

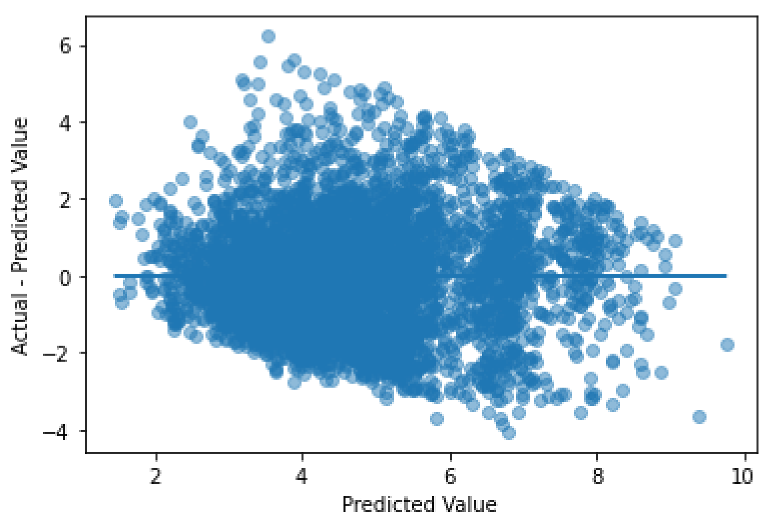
Out[667]:
```
sqft_living_log           3.123886
condition_Good            1.454836
condition_Very Good       1.138377
grade_desc_Better         1.319111
grade_desc_Excellent      1.016561
grade_desc_Fair           1.023719
grade_desc_Good           1.724535
grade_desc_Low Average    1.207339
grade_desc_Very Good      1.109697
Name: VIF, dtype: float64
```

Multi-Colineearity is important to test because it causes issues with the interpretation of the coefficients. Specifically, you can interpret a coefficient as “an increase of 1 in this predictor results in a change of (coefficient) in the response variable, holding all other predictors constant.” This becomes problematic when multicollinearity is present because we can’t hold correlated predictors constant. Additionally, it increases the standard error of the coefficients, which results in them potentially showing as statistically insignificant when they might actually be significant.

Generally VIF values lower than 5 indicate that there is low or no multi-collinearity.

Homoscedasticity:

In [668]:
```
fig, ax = plt.subplots()

ax.scatter(preds, residuals, alpha=0.5)
ax.plot(preds, [0 for i in range(len(X_test))])
ax.set_xlabel("Predicted Value")
ax.set_ylabel("Actual - Predicted Value");
```



Homoscedasticity is the is the same variance within our error terms. Homoscedasticity can impact significance tests for coefficients due to the standard errors being biased. Additionally, the confidence intervals can be either too wide or too narrow.

While, we do not have a perfect homoscedasticity, we can see that we have the same trend that we saw before which is that it is biased towards under-prediction.

Interpreting the Final Model:

Lets first take a quick look at the mean squareed error, which tells us how off our model is as compared to the real values.

In [669]:

```
from sklearn.metrics import mean_squared_error

mean_squared_error(pr_test, final_model.predict(X_test_mod[best_feature
```

Out[669]: 1.4436625399896426

This means that our model is off by almost $140,000 from the true prediction. This shows that this model should be refined a lot more before using it for preedictions. Nonetheless, our mmodel's main goal was for inference purposes and we can achieve that by evaluating the coefficients of our final model's variables.

In [670]:

```
#print cefficients
print(pd.Series(final_model.coef_, index=X_train_mod[best_features].col
print()
print("Intercept:", final_model.intercept_) #print intercept
```

```
sqft_living_log          1.497633
condition_Good           0.342965
condition_Very Good      0.938995
grade_desc_Better        2.045854
grade_desc_Excellent     3.639608
grade_desc_Fair         -0.704242
grade_desc_Good          0.864142
grade_desc_Low Average  -0.495792
grade_desc_Very Good     2.872165
Name: Coefficients, dtype: float64

Intercept: -7.245529345435795
```

The intercept being negative shows that out model's basic house price without any variables is in the negative which is not plausible in reality. Nonetheless, we are still able to use it for inference purposes.

In [671]:
```
#reference for undeerstanding grade categories
init_data['grade'].unique()
```

Out[671]:
```
array(['7 Average', '6 Low Average', '8 Good', '11 Excellent', '9 Bette
r',
       '5 Fair', '10 Very Good', '12 Luxury', '4 Low', '3 Poor',
       '13 Mansion'], dtype=object)
```

Our model is biasing towards under-predicting the prices towards the higher end.

Using the coefficients, the performance of certain renovations and how they impact the housing prices is given below. These can be used to inform homeowners what kind of renovations for these features would help them improve their house's value.

- `sqft living` : For every percentage increase in square footage of living space, the house price goes up by 1.5%
  - This means that improvements, such as increasing floors or extending the house to add extra rooms and living quarters, can increase the house prices significantly.
- `condition` :
  - Intuitively, it is clear that as the condition of the house improves, the house price also increases.
  - The modell showcases that as the condition shifts to 'Good' the house price starts to improvees by a mmultiple of 0.34, i.e. No obvious maintenance required but neither is everything new. Also, appearance and utility are above the standard.
  - This means that renovating houses to make sure that everything that the current house has is properly functional along with regular maintainenace can help a lot too.
  - Also, moving into the Very Good condition increases thee house value by a multiple 0f 0.94 but it also reequires that All items well maintained, many having been overhauled and repaired as they have shown signs of wear, increasing the life expectancy and lowering the effective age with little deterioration or obsolescence evident with a high degree of utility.
  - Complete remodelling project and flipping homes would see this kind of a change.
- `grade` :
  - We can see that having a Fair or Low Average grade for a house poses a negative penalty to a house's value but moving into just the Good Category starts having a positive impact on the house price. the Assessor describes Good (Grade 8) as 'Just above average in construction and design. Usually better materials in both the exterior and interior finish work'.
  - As the grades keep increasing, the price starts going up substantially and at Excellent, described as 'Custom design and higher quality finish work with added amenities of solid woods, bathroom fixtures and more luxurious options' increases the price the most by a factor of 3.64 which is very substantial.
  - This means that renovations geared towards adding extra luxurious amenities can increase the price of the house significantly.