MULTIPLE LINEAR REGRESSION MODELLING TO ANALYZE HOUSE SALES IN KING COUNTY, WASHINGTON

Project Details

- **Contributors**:


```
Prossy Nansubuga Kamau –prossykamau@gmail.com
Evaclaire Wamitu – evamunyika@gmail.com
Julius Kinyua – juliusczar36@gmail.com
Joan Nyamache – kerubonyamache@gmail.com
Elizabeth Masai – elizabethchemtaim@gmail.com
Kelvin Mwaura – kelvinmwaura.edu@gmail.com
Mourine kitili– mourinekitilimourine@gmail.com
Allan Kiplagat – allankiplgat@gmail.com
Mitch Mathiu   – mmuriithi92@gmail.com
```


- **Student pace**: self paced
- **Scheduled project review date/time**: May 2024
- **Instructor name**: Asha Deen , Lucille Kaleha
- **Blog post URL**:

1.0 BUSINESS UNDERSTANDING

The primary goal of the client is to create a platform that delivers accurate estimates of house prices which are crucial for both buyers and sellers in King county. To achieve this objective we need to develop a model that can identify the key factors influencing house prices. To effectively train this model the client requires precise and representative data related to the real estate market in king county, including historical sales, current listings, property size, and other relevant features. Once trained successfully the model will be capable of providing accurate estimations of house values based on their features. The expectation is that, once completed, the model can be used by them as a tool in selecting properties for investment in King County.

Analysis Questions

This analysis will seek to answer three questions about the data:

Question 1: Which features are most highly correlated with price?

Question 2: Which features have the strongest correlations with other predictor variables?

Question 3: What combinations of features is the best fit, in terms of predictive power, for a multiple regression model to predict house prices?

2.0 DATA UNDERSTANDING

The dataset contains 21 columns. One is of the prices of the houses, which is our target variable, and the rest will be used to make our predictions. The column names and their descriptions are as follows: id - unique identified for a house dateDate - house was sold pricePrice - is prediction target bedroomsNumber - of Bedrooms/House bathroomsNumber - of bathrooms/bedrooms sqft_livingsquare - footage of the home sqft_lotsquare - footage of the lot floorsTotal - floors (levels) in house waterfront - House which has a view to a waterfront view - Has been viewed condition - How good the condition is (Overall) grade - overall grade given to the housing unit, based on King County grading system sqft_above - square footage of house apart from basement sqft_basement - square footage of the basement yr_built - Built Year yr_renovated - Year when house was renovated zipcode - zip lat - Latitude coordinate long - Longitude coordinate sqft_living15 - The square footage of interior housing living space for the nearest 15 neighbors sqft_lot15 - The square footage of the land lots of the nearest 15 neighbors

3.0 DATA PREPARATION

3.1 Importing Libraries

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

In [2]:
```
#Loading the dataset
df = pd.read_csv('data/kc_house_data.csv')
df
```

Out[2]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	w
0	7129300520	10/13/2014	221900.0	3	1.00	1180	5650	1.0	
1	6414100192	12/9/2014	538000.0	3	2.25	2570	7242	2.0	
2	5631500400	2/25/2015	180000.0	2	1.00	770	10000	1.0	
3	2487200875	12/9/2014	604000.0	4	3.00	1960	5000	1.0	
4	1954400510	2/18/2015	510000.0	3	2.00	1680	8080	1.0	
...	...	...	...	...	...	...	...	...	
21592	263000018	5/21/2014	360000.0	3	2.50	1530	1131	3.0	
21593	6600060120	2/23/2015	400000.0	4	2.50	2310	5813	2.0	
21594	1523300141	6/23/2014	402101.0	2	0.75	1020	1350	2.0	
21595	291310100	1/16/2015	400000.0	3	2.50	1600	2388	2.0	
21596	1523300157	10/15/2014	325000.0	2	0.75	1020	1076	2.0	

21597 rows × 21 columns

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

Out[3]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterf
0	7129300520	10/13/2014	221900.0	3	1.00	1180	5650	1.0	
1	6414100192	12/9/2014	538000.0	3	2.25	2570	7242	2.0	
2	5631500400	2/25/2015	180000.0	2	1.00	770	10000	1.0	
3	2487200875	12/9/2014	604000.0	4	3.00	1960	5000	1.0	
4	1954400510	2/18/2015	510000.0	3	2.00	1680	8080	1.0	

5 rows × 21 columns

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

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

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

Out[5]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	2
count	2.159700e+04	2.159700e+04	21597.000000	21597.000000	21597.000000	2.159700e+04	
mean	4.580474e+09	5.402966e+05	3.373200	2.115826	2080.321850	1.509941e+04	
std	2.876736e+09	3.673681e+05	0.926299	0.768984	918.106125	4.141264e+04	
min	1.000102e+06	7.800000e+04	1.000000	0.500000	370.000000	5.200000e+02	
25%	2.123049e+09	3.220000e+05	3.000000	1.750000	1430.000000	5.040000e+03	
50%	3.904930e+09	4.500000e+05	3.000000	2.250000	1910.000000	7.618000e+03	
75%	7.308900e+09	6.450000e+05	4.000000	2.500000	2550.000000	1.068500e+04	
max	9.900000e+09	7.700000e+06	33.000000	8.000000	13540.000000	1.651359e+06	

The dataset has 21597 entries and 20 columns. Most of the columns consist of numerical data which make it suitable for linear regression analysis.

In [6]:
```
# Checked the number of unique values for each column
df.nunique()
```

Out[6]:
```
id               21420
date               372
price             3622
bedrooms            12
bathrooms           29
sqft_living       1034
sqft_lot          9776
floors               6
waterfront           2
view                 5
condition            5
grade               11
sqft_above         942
sqft_basement      304
yr_built           116
yr_renovated        70
zipcode             70
lat               5033
long               751
sqft_living15      777
sqft_lot15        8682
dtype: int64
```

4.0 DATA CLEANING

Check for Validity, Accuracy, Completeness, Consistency and Uniformity of the Data

Missing Values

In [7]:
```
df.floors.value_counts()
```

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

In [8]:
```
#Drop the id and date column since there is no use for it


#df.drop(["id", "date"], axis= 1, inplace= True)
```

In [9]:
```
df.info()
```

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

In [10]:
```
# check the total number of null values in each column
df.isnull().sum()
```

Out[10]:
```
id                0
date              0
price             0
bedrooms          0
bathrooms         0
sqft_living       0
sqft_lot          0
floors            0
waterfront     2376
view             63
condition         0
grade             0
sqft_above        0
sqft_basement     0
yr_built          0
yr_renovated   3842
zipcode           0
lat               0
long              0
sqft_living15     0
sqft_lot15        0
dtype: int64
```

The columns waterfront, view and yr_renovated have 2376, 63 and 3842 null values respectively Lets look it further

In [11]:
```
df.waterfront.value_counts()
```

Out[11]:
```
NO     19075
YES      146
Name: waterfront, dtype: int64
```

In [12]:
```
# Replace the null values
df['waterfront'].fillna('NO', inplace= True)
df.waterfront.value_counts()
```

Out[12]:
```
NO     21451
YES      146
Name: waterfront, dtype: int64
```

In [13]:
```
df.view.value_counts()
```

Out[13]:
```
NONE         19422
AVERAGE        957
GOOD           508
FAIR           330
EXCELLENT      317
Name: view, dtype: int64
```

In [14]:
```
# Replace the null values
df['view'].fillna('NONE', inplace= True)
df.view.value_counts()
```

Out[14]:
```
NONE         19485
AVERAGE        957
GOOD           508
FAIR           330
EXCELLENT      317
Name: view, dtype: int64
```

In [15]:
```
df.dropna(subset=['yr_renovated'])
```

Out[15]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors
0	7129300520	10/13/2014	221900.0	3	1.00	1180	5650	1.0
1	6414100192	12/9/2014	538000.0	3	2.25	2570	7242	2.0
3	2487200875	12/9/2014	604000.0	4	3.00	1960	5000	1.0
4	1954400510	2/18/2015	510000.0	3	2.00	1680	8080	1.0
5	7237550310	5/12/2014	1230000.0	4	4.50	5420	101930	1.0
...	...	...	...	...	...	...	...	...
21592	263000018	5/21/2014	360000.0	3	2.50	1530	1131	3.0
21593	6600060120	2/23/2015	400000.0	4	2.50	2310	5813	2.0
21594	1523300141	6/23/2014	402101.0	2	0.75	1020	1350	2.0
21595	291310100	1/16/2015	400000.0	3	2.50	1600	2388	2.0
21596	1523300157	10/15/2014	325000.0	2	0.75	1020	1076	2.0

17755 rows × 21 columns

In [16]:
```
# Fill missing values in the 'yr_renovated' column with 'NO'
df['yr_renovated'].fillna('NO', inplace= True)
df.yr_renovated.value_counts()
```

Out[16]:
```
0.0       17011
NO         3842
2014.0       73
2003.0       31
2013.0       31
          ...
1971.0        1
1944.0        1
1934.0        1
1976.0        1
1959.0        1
Name: yr_renovated, Length: 71, dtype: int64
```

In [17]:
```
#counter checking to see if there are any more missing values
df.isnull().sum()
```

Out[17]:
```
id               0
date             0
price            0
bedrooms         0
bathrooms        0
sqft_living      0
sqft_lot         0
floors           0
waterfront       0
view             0
condition        0
grade            0
sqft_above       0
sqft_basement    0
yr_built         0
yr_renovated     0
zipcode          0
lat              0
long             0
sqft_living15    0
sqft_lot15       0
dtype: int64
```

As evident from the previous section, the data has been thoroughly processed and is now prepared for analysis and other tasks. All missing values were replaced successfully.

Conversion of Date Column and Creation of New Columns

The date column represents the month and year the houses were sold. Creating new columns yr_sold and month_sold from this column will make analyzing the data easier and then dropping it since it will not be useful anymore.

In [18]:
```
# split the date into month, day and year
date = df['date'].str.split('/', expand=True)

# create new columns for month and year and convert to integer
df['month_sold'] = date[0].astype(int)
df['yr_sold'] = date[2].astype(int)

# drop original date column
df.drop(columns=['date'], axis=1, inplace=True)

# check to see if changed were made
df.head()
```

Out[18]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view
0	7129300520	221900.0	3	1.00	1180	5650	1.0	NO	NONE
1	6414100192	538000.0	3	2.25	2570	7242	2.0	NO	NONE
2	5631500400	180000.0	2	1.00	770	10000	1.0	NO	NONE
3	2487200875	604000.0	4	3.00	1960	5000	1.0	NO	NONE
4	1954400510	510000.0	3	2.00	1680	8080	1.0	NO	NONE

5 rows × 22 columns

Lets to create a new column age to represent the age of the houses. To get the age of the house, I got the difference between the yr_built and the year 2015

In [19]:
```
# create new column 'age'
df['age'] = 2015 - df['yr_built']

# drop the column for yr_built since it's not longer useful
df.drop(columns=['yr_built'], axis=1, inplace=True)

# check to see if changes were made
df
```

Out[19]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	
0	7129300520	221900.0	3	1.00	1180	5650	1.0	NO	N
1	6414100192	538000.0	3	2.25	2570	7242	2.0	NO	N
2	5631500400	180000.0	2	1.00	770	10000	1.0	NO	N
3	2487200875	604000.0	4	3.00	1960	5000	1.0	NO	N
4	1954400510	510000.0	3	2.00	1680	8080	1.0	NO	N
...	...	...	...	...	...	...	...	...	
21592	263000018	360000.0	3	2.50	1530	1131	3.0	NO	N
21593	6600060120	400000.0	4	2.50	2310	5813	2.0	NO	N
21594	1523300141	402101.0	2	0.75	1020	1350	2.0	NO	N
21595	291310100	400000.0	3	2.50	1600	2388	2.0	NO	N
21596	1523300157	325000.0	2	0.75	1020	1076	2.0	NO	N

21597 rows × 22 columns

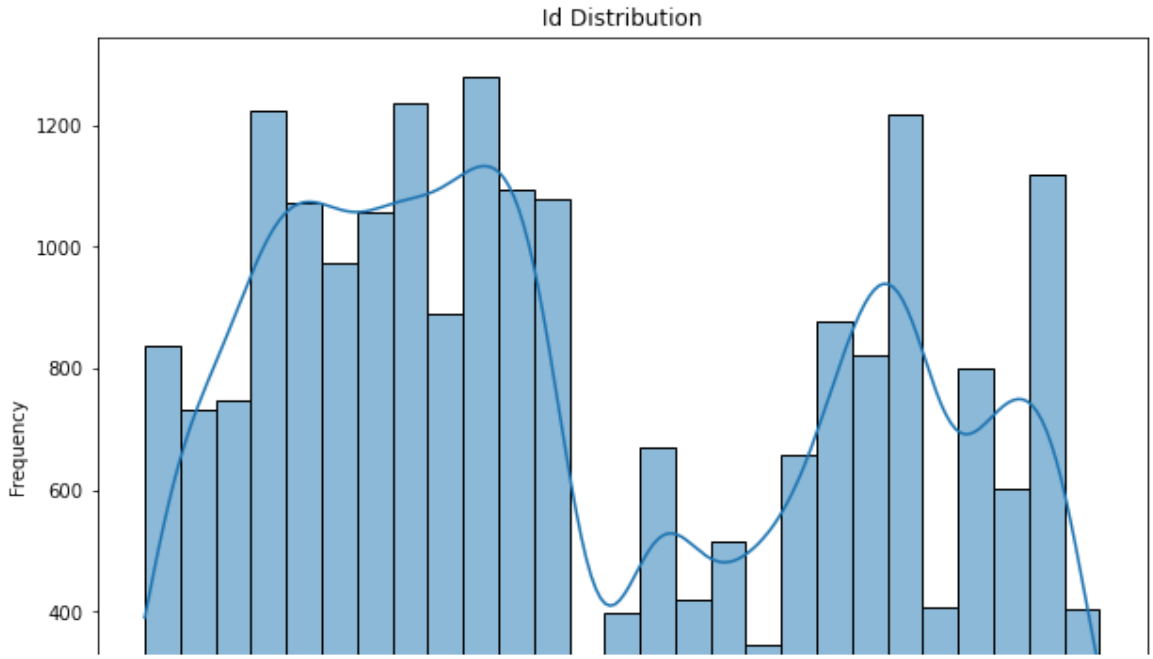
In [20]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 22 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   id             21597 non-null  int64  
 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     21597 non-null  object 
 8   view           21597 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_renovated   21597 non-null  object 
 14  zipcode        21597 non-null  int64  
 15  lat            21597 non-null  float64
 16  long           21597 non-null  float64
 17  sqft_living15  21597 non-null  int64  
 18  sqft_lot15     21597 non-null  int64  
 19  month_sold     21597 non-null  int64  
 20  yr_sold        21597 non-null  int64  
 21  age            21597 non-null  int64  
dtypes: float64(5), int64(11), object(6)
memory usage: 3.6+ MB
```

The data has been cleaned successfully.

We then plotted some visualizations to understand the characteristics of the various features in our dataset.

In [21]:
```
def variable_histogram(df, log_transform=False):
    """
    Generates histograms for all numeric variables in the dataset.

    Parameters:
        df (pandas.DataFrame): The dataset to be analyzed.
        log_transform (bool): Whether to apply a log transformation to
    """
    for variable in df.select_dtypes(include=[np.number]).columns:
        plt.figure(figsize=(10, 8))
        if log_transform:
            variable_log = np.log(df[variable])
            sns.histplot(data=df, x=variable_log, kde=True)
        else:
            sns.histplot(data=df, x=variable, kde=True)
        plt.title(f"{variable.capitalize()} Distribution")
        plt.xlabel(variable)
        plt.ylabel("Frequency")
        plt.show()


variable_histogram(df)
```

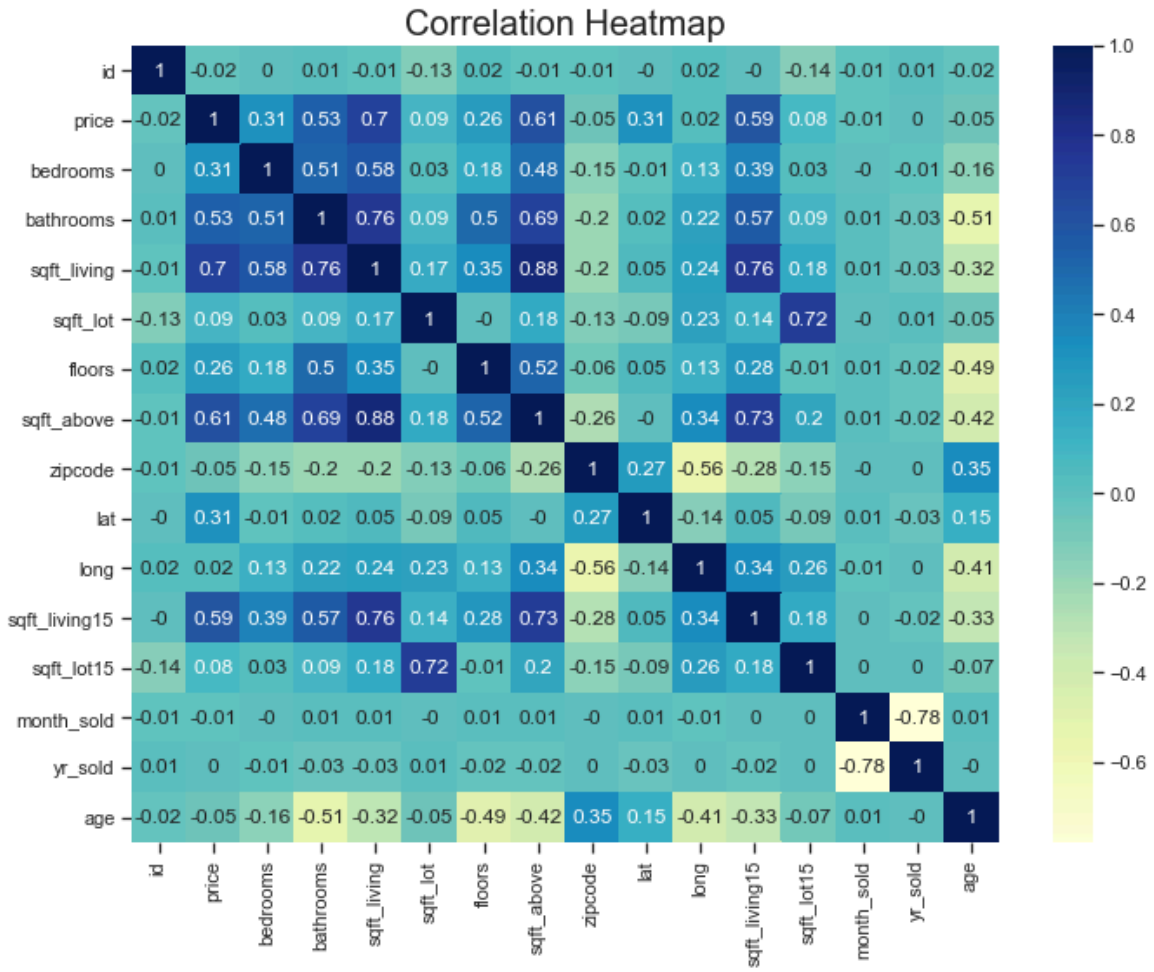


5.0 DATA VISUALIZATION

Prior to the modeling process, we needed to create visualizations in order to analyse some of the trends in the data.

We created a correlation matrix in order to identify highly correlated variables in the data.

In [22]:
```
# creating a correlation matrix
correlation_matrix = df.corr()
rounded_corr_matrix = np.round(correlation_matrix, 2)

# setting seaborn theme
sns.set_theme(style='ticks')

# plotting correlation matrix heatmap and displaying correlation coeff
plt.figure(figsize=(10,8))
sns.heatmap(rounded_corr_matrix, cmap='YlGnBu', annot=True)
plt.title('Correlation Heatmap', fontsize=20)

# Adjusting the layout of the plot for better visualization
plt.tight_layout()

# saving figure
plt.savefig('./fig1.png');
```



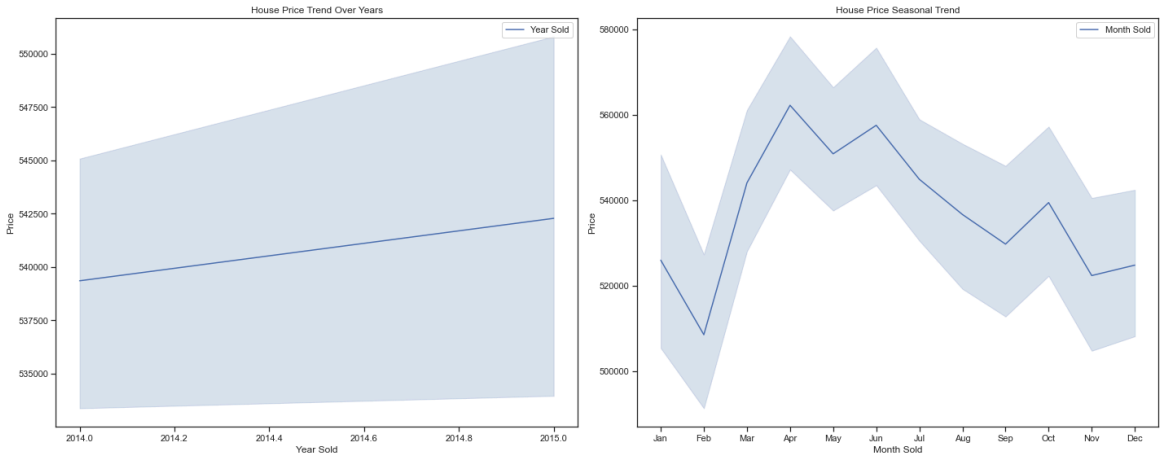
From the heatmap above, we could see that there are relatively strong positive correlations between price and sqft_living at 0.7, sqft_above at 0.61, sqft_living15 at 0.59 and number of bathrooms at 0.53.

The weakest positive correlations were between price and number of bedrooms at 0.31, sqft_lot at 0.09, number of floors at 0.26 and sqft_lot15 at 0.08.

The weakest inverse correlations were between price and zipcode and age at -0.05, and month sold at -0.01.

Next we wanted to investigate house price trends over the year and over the specific months/seasons.

In [23]:
```
# setting seaborn theme
sns.set_theme(style='ticks')

# Creating a figure with two subplots side by side
fig, axes = plt.subplots(1, 2, figsize=(20, 8))

# Plotting price vs. yr_sold and price vs. month_sold on  same axes
for i, (x_col, title, xlabel) in enumerate([('yr_sold', 'House Price T
                                            ('month_sold', 'House Pric
    sns.lineplot(x=x_col, y='price', data=df, ax=axes[i], label=xlabel
    axes[i].set_title(title)
    axes[i].set_xlabel(xlabel)
    axes[i].set_ylabel('Price')
    if x_col == 'month_sold':
        axes[i].set_xticks(range(1, 13))
        axes[i].set_xticklabels(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'J

# Adjusting the layout of the subplots for better visualization
plt.tight_layout()

# saving figure
plt.savefig('./fig2.png');
```

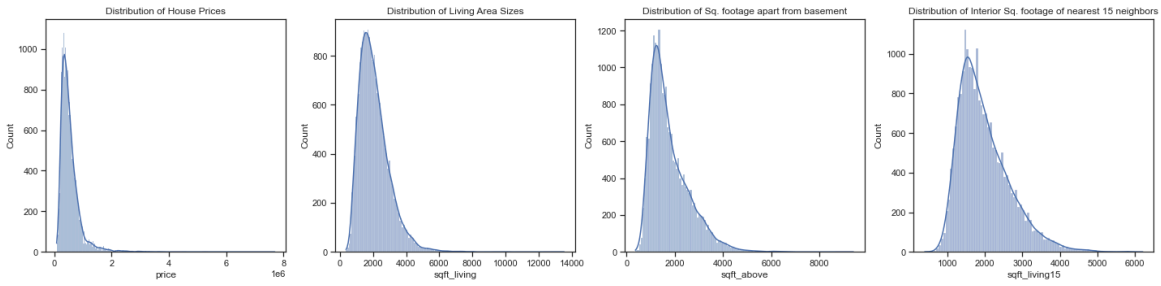


We can clearly see that house prices were on a steady rise from 2014 to 2015 with the highest house prices recorded in the month of April and the lowest in the month of February. Investigations are required to find out why house prices drastically rose between February and April and why prices fell between June and September.

The next step was to explore the distributions of the variables with the strongest positive relationships with price.

In [24]:
```
# setting seaborn theme
sns.set_theme(style='ticks')

# Creating a figure with 1 row and 4 columns of subplots
fig, axes = plt.subplots(1, 4, figsize=(20, 5))

# Defining the variables and titles for the histograms
variables = ['price', 'sqft_living', 'sqft_above', 'sqft_living15']
titles = ['House Prices', 'Living Area Sizes', 'Sq. footage apart from

# Iterating over the variables and titles to create histograms
for i, var in enumerate(variables):
    # Plot the histogram for the current variable
    sns.histplot(data=df, x=var, kde=True, ax=axes[i])

    # Setting the title for the current subplot based on the correspon
    axes[i].set_title(f'Distribution of {titles[i]}')

# Adjusting the layout of the subplots for better visualization
plt.tight_layout()

# saving figure
plt.savefig('./fig3.png');

print('The skewness of the price plot is', df['price'].skew())
print('The kurtosis of the price plot is', df['price'].kurt(), '\n')
print('The skewness of the sqft_living plot is', df['sqft_living'].ske
print('The kurtosis of the sqft_living plot is', df['sqft_living'].kur
print('The skewness of the sqft_above plot is', df['sqft_above'].skew(
print('The kurtosis of the sqft_above plot is', df['sqft_above'].kurt(
print('The skewness of the sqft_living15 plot is', df['sqft_living15']
print('The kurtosis of the sqft_living15 plot is', df['sqft_living15']
```

```
The skewness of the price plot is 4.023364652271239
The kurtosis of the price plot is 34.54135857673376

The skewness of the sqft_living plot is 1.473215455425834
The kurtosis of the sqft_living plot is 5.252101950846816

The skewness of the sqft_above plot is 1.4474342353857224
The kurtosis of the sqft_above plot is 3.405519761077342

The skewness of the sqft_living15 plot is 1.1068753971161713
The kurtosis of the sqft_living15 plot is 1.591732789053459
```



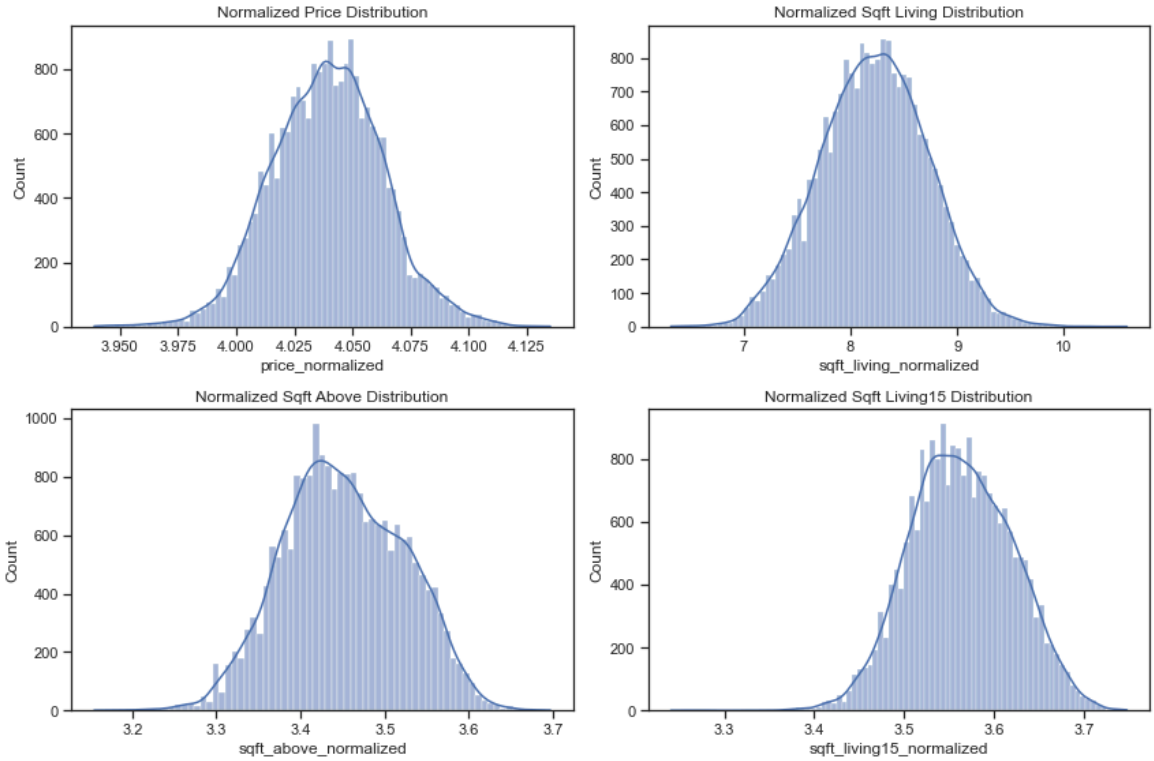
1. Distribution of House Prices: This histogram shows the frequency of houses at different price points. The distribution appears to be slightly right-skewed with a skewness of 4.02 meaning the data has a long right tail, with more extreme high values compared to low values. This indicates that there are more houses at higher price ranges than lower ones. However, there is still a significant number of houses in the lower price ranges as

well. The kurtosis of the price plot is 34.54 which is much greater than 3. This indicates a leptokurtic distribution, meaning the distribution has a sharp peak and heavy tails. This suggests the presence of many outliers or extreme values in the price data.

2. Distribution of Living Area Sizes(sqft_living): This graph displays the distribution of living area sizes, in square feet, for the houses in the dataset. The distribution is roughly bell-shaped, which is a typical pattern for many real-world variables. The skewness of the sqft_living plot is 1.47, which indicates a moderately positively skewed distribution. The data has more high values than low values. The peak of the distribution suggests that most houses have living areas clustered around a specific size range, with fewer houses having very small or very large living areas. The kurtosis of the sqft_living plot is 5.25, which is greater than 3. This indicates a leptokurtic distribution, with a sharper peak and heavier tails compared to a normal distribution.
3. Distribution of Sq. Footage apart from the basement: This histogram shows the distribution of the square footage of the houses, excluding the basement area. The skewness of the sqft_above plot is 1.45, which indicates a moderately positively skewed distribution. The distribution is similar to the living area sizes, with a bell-shaped curve indicating that most houses have a specific range of square footage, and fewer houses have very small or very large square footage. The kurtosis of the sqft_above plot is 3.41, which is close to 3. This suggests a distribution that is similar to a normal distribution, with a peak and tails that are not significantly different from a normal distribution.
4. Distribution of Interior Sq. footage of nearest 15 neighbors: This graph depicts the distribution of the interior square footage of the 15 nearest neighboring houses. The skewness of the sqft_living15 plot is 1.11, which indicates a moderately positively skewed distribution. The distribution is bell-shaped, similar to the previous two graphs, suggesting that houses in a given neighborhood tend to have similar interior square footage sizes. The kurtosis of the sqft_living15 plot is 1.59, which is less than 3. This indicates a platykurtic distribution, with a flatter peak and lighter tails compared to a normal distribution.

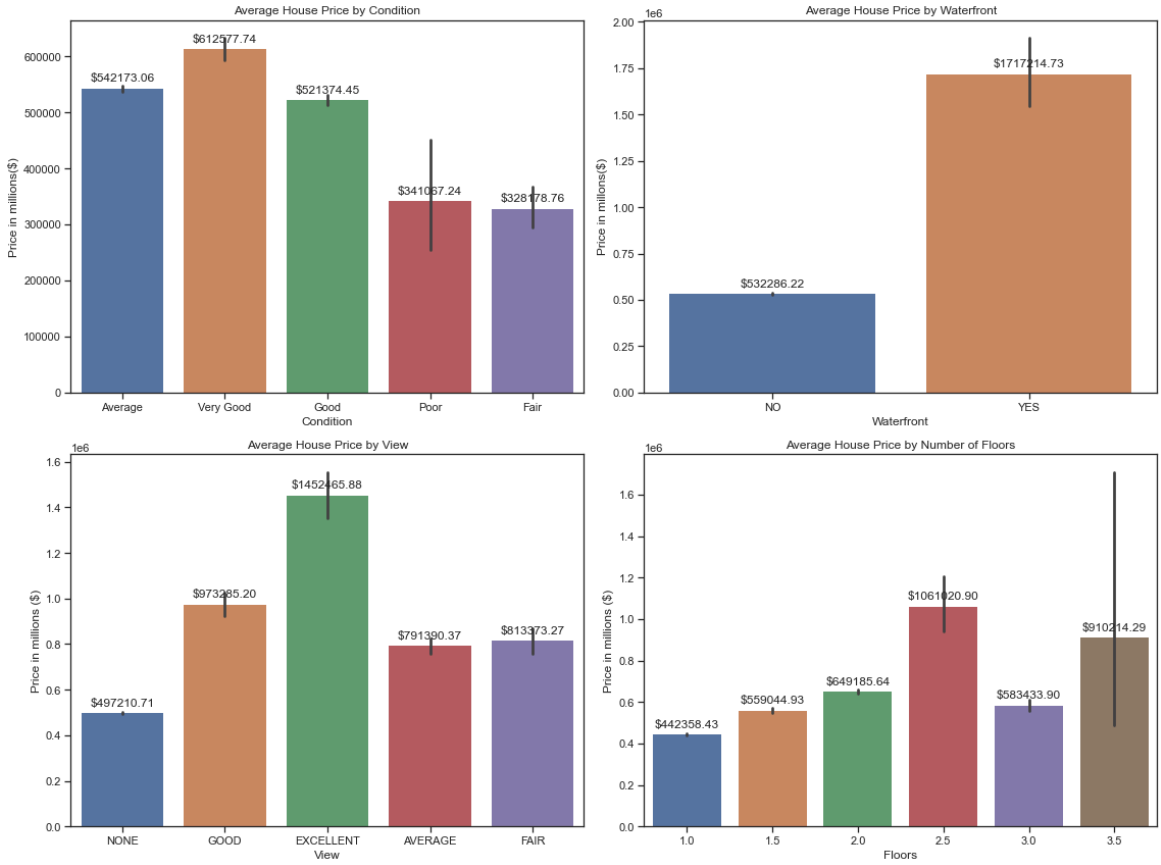
Next we used the boxcox function from the scipy.stats module to perform the Box-Cox transformation on the columns from the previous plots. The boxcox function returns two values: the transformed data and the optimal lambda value for the transformation. Since we only needed the transformed values, we stored the transformed data in new columns with the suffix '_normalized' and then created histograms for the normalized variables.

In [25]:
```
from scipy.stats import boxcox

# Normalize the 'price' column
df['price_normalized'], _ = boxcox(df['price'])

# Normalize the 'sqft_living' column
df['sqft_living_normalized'], _ = boxcox(df['sqft_living'])

# Normalize the 'sqft_above' column
df['sqft_above_normalized'], _ = boxcox(df['sqft_above'])

# Normalize the 'sqft_living15' column
df['sqft_living15_normalized'], _ = boxcox(df['sqft_living15'])


def plot_normalized_data(df):
    """
    Plot histograms of the normalized data.

    Parameters:
        df (pandas.DataFrame): The dataset with normalized columns.
    """

    # setting seaborn theme
    sns.set_theme(style='ticks')

    plt.figure(figsize=(12, 8))

    # Plotting histogram of the 'price_normalized' column
    plt.subplot(2, 2, 1)
    sns.histplot(df['price_normalized'], kde=True)
    plt.title('Normalized Price Distribution')

    # Plotting histogram of the 'sqft_living_normalized' column
    plt.subplot(2, 2, 2)
    sns.histplot(df['sqft_living_normalized'], kde=True)
    plt.title('Normalized Sqft Living Distribution')

    # Plotting histogram of the 'sqft_above_normalized' column
    plt.subplot(2, 2, 3)
    sns.histplot(df['sqft_above_normalized'], kde=True)
    plt.title('Normalized Sqft Above Distribution')

    # Plotting histogram of the 'sqft_living15_normalized' column
    plt.subplot(2, 2, 4)
    sns.histplot(df['sqft_living15_normalized'], kde=True)
    plt.title('Normalized Sqft Living15 Distribution')

    # Adjusting the layout of the subplots for better visualization
    plt.tight_layout()
    plt.savefig('./fig4.png');

# Call the function to plot the histograms of the normalized data
plot_normalized_data(df)
```



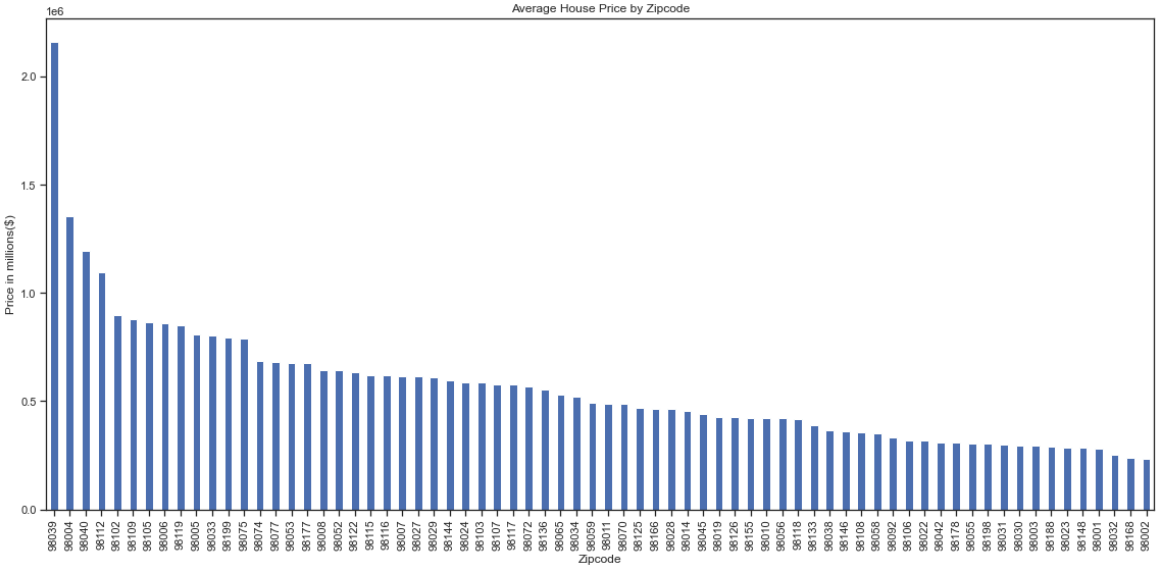
Below we analyzed the variation of average house prices as per the overall condition of a house, whether a house is located on a waterfront or not, the quality of views from the house and the number of floors/ levels in a house.

In [26]:
```
# Defining the plots to be created
plots = [
    ('Average House Price by Condition', 'condition', 'Price in millic
    ('Average House Price by Waterfront', 'waterfront', 'Price in mill
    ('Average House Price by View', 'view', 'Price in millions ($)'),
    ('Average House Price by Number of Floors', 'floors', 'Price in mi
]

# setting seaborn theme
sns.set_theme(style='ticks')

# Creating a figure with 2 rows and 2 columns of subplots
fig, axes = plt.subplots(2, 2, figsize=(16, 12))
axes = axes.flatten()

# Iterating over the plots and create bar plots for each
for i, (title, x_col, y_col) in enumerate(plots):
    # Create a bar plot for the current plot
    sns.barplot(x=x_col, y='price', data=df, ax=axes[i], edgecolor='nc

    # Adding annotations for the price values on top of each bar
    for p in axes[i].patches:
        axes[i].annotate(f"${p.get_height():.2f}", (p.get_x() + p.get_
                         ha='center', va='bottom', xytext=(0, 5), textc

    # Setting the title, x-axis label, and y-axis label for the currer
    axes[i].set_title(title)
    axes[i].set_xlabel(x_col.capitalize())
    axes[i].set_ylabel(y_col)

# Adjusting the layout of the subplots for better visualization
plt.tight_layout()

# saving figure
plt.savefig('./fig5.png');
```



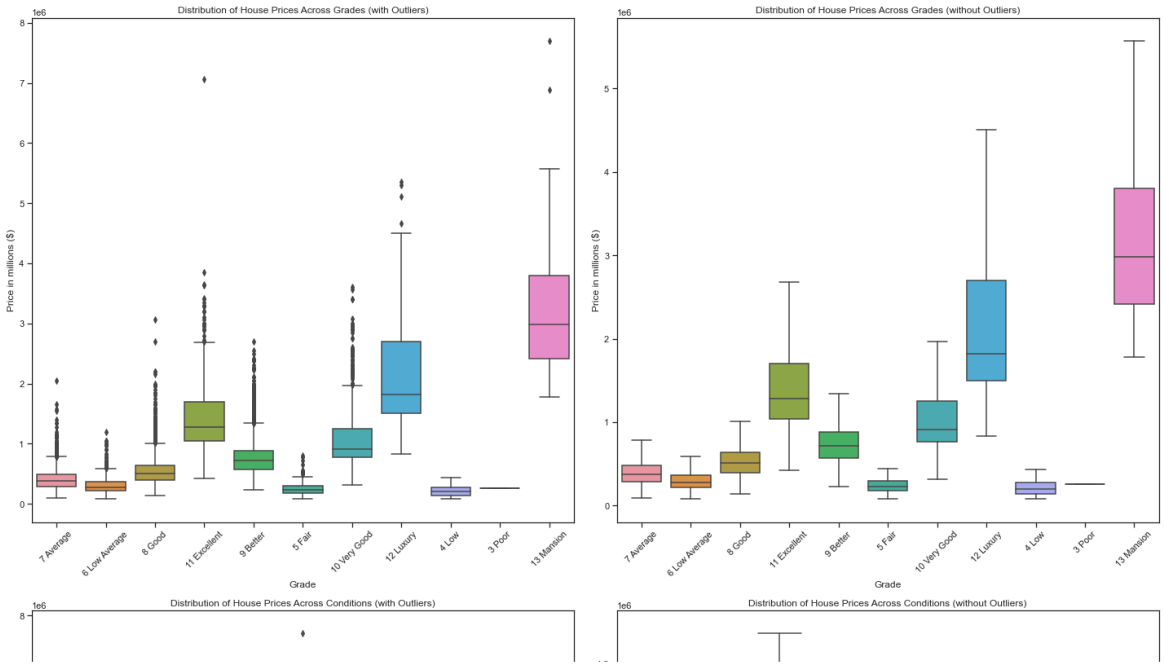
Next was to analyze the distribution of average house prices per zipcode.

In [27]:
```
# setting seaborn theme
sns.set_theme(style='ticks')

# Creating a figure with a single subplot
fig, ax = plt.subplots(figsize=(16, 8))

# Grouping the data by zipcode, calculating mean price for each zipcod
price_by_zipcode = df.groupby('zipcode')['price'].mean().sort_values(a

# Creating a bar plot of the average price per zipcode
price_by_zipcode.plot(kind='bar', ax=ax)

# Setting the title, x-axis label, and y-axis label for the plot
ax.set_title('Average House Price by Zipcode')
ax.set_xlabel('Zipcode')
ax.set_ylabel('Price in millions($)')

# Rotating the x-axis labels by 90 degrees for better readability
plt.xticks(rotation=90)

# Adjusting the layout for better visualization
plt.tight_layout()

# saving figure
plt.savefig('./fig6.png');
```



From the above plot we can see that houses in Medina City (zipcode 98039) have the highest average prices while houses in Auburn City (zipcode 98002) have the lowest average prices. The next step was to assess the change in house price values as per the grade, condition and waterfront location of a house with and without outliers using box plots as shown below.

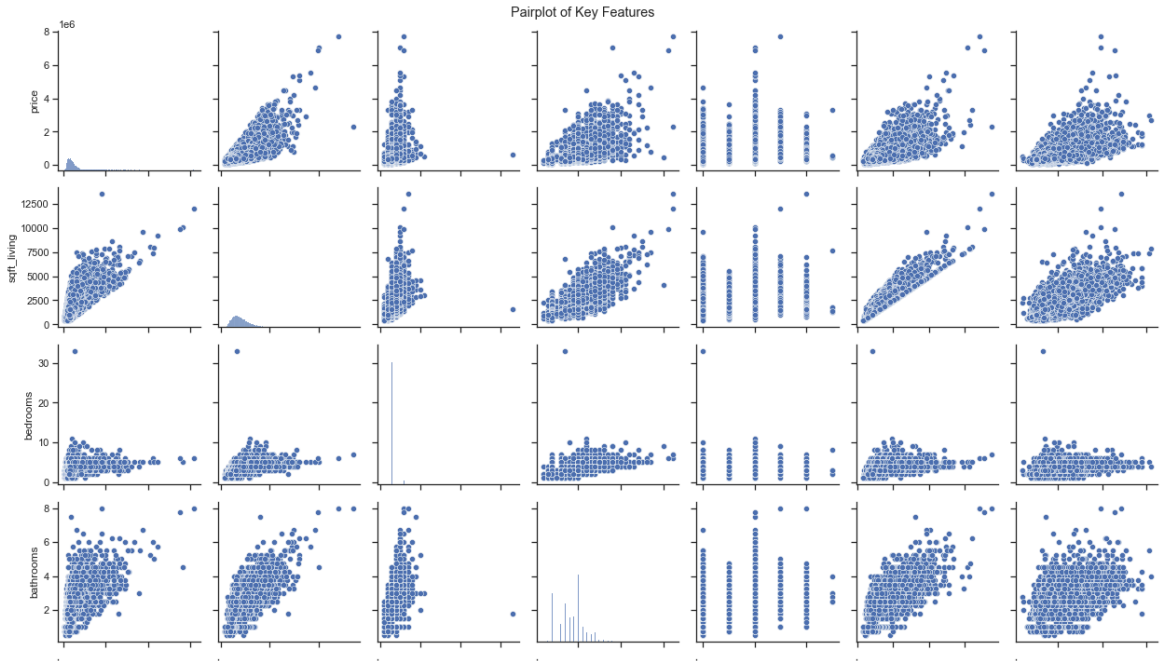
In [28]:
```
# Defining the features and titles for the boxplots
features = [
    ('grade', 'Distribution of House Prices Across Grades'),
    ('condition', 'Distribution of House Prices Across Conditions'),
    ('waterfront', 'Distribution of House Prices for Waterfront Proper
]

# setting seaborn theme
sns.set_theme(style='ticks')

# Creating a figure with multiple subplots
fig, axes = plt.subplots(len(features), 2, figsize=(20, 30))

# Iterating over the features and titles to create boxplots
for i, (feature, title) in enumerate(features):
    # Creating boxplots with outliers
    sns.boxplot(x=feature, y='price', data=df, ax=axes[i, 0])
    axes[i, 0].set_title(f"{title} (with Outliers)")
    axes[i, 0].tick_params(axis='x', rotation=45)

    # Creating boxplots without outliers
    sns.boxplot(x=feature, y='price', data=df, ax=axes[i, 1], showflie
    axes[i, 1].set_title(f"{title} (without Outliers)")
    axes[i, 1].tick_params(axis='x', rotation=45)

    # Setting x-axis and y-axis labels for both subplots
    axes[i, 0].set_xlabel(feature.capitalize())
    axes[i, 0].set_ylabel('Price in millions ($)')
    axes[i, 1].set_xlabel(feature.capitalize())
    axes[i, 1].set_ylabel('Price in millions ($)')

# Adjusting the layout of the subplots for better visualization
plt.tight_layout()

# saving figure
plt.savefig('./fig7.png');
```



We can see that the median house prices increased when outliers were removed from the data. We can therefore conclude that house prices in King County were affected by outliers

We then created a grid of scatterplots to visualise and better explore the relationships between multiple variables listed below. This allowed us to infer which variables showed linear relationships with one another.

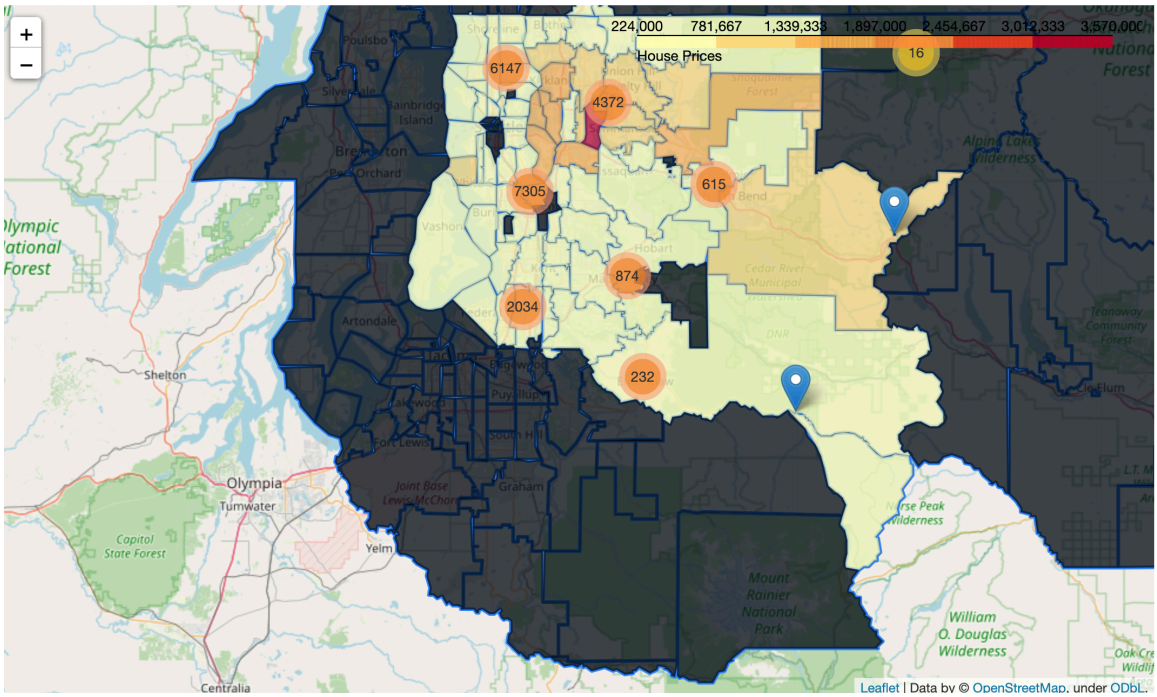
In [29]:
```
# Defining the variables for the scatter plot matrix
scatter_vars = ['price', 'sqft_living', 'bedrooms', 'bathrooms', 'floo

# setting seaborn theme
sns.set_theme(style='ticks')

# Creating a pairplot for the selected variables
sns.pairplot(df[scatter_vars], kind='scatter', diag_kind='hist')

# Setting the overall title for the pairplot
plt.suptitle('Pairplot of Key Features')

# Adjusting the layout of the subplots for better visualization
plt.tight_layout()

# saving figure
plt.savefig('./fig8.png');
```



Lastly, we created an interactive map to showcase our study area, King County, and the distribution of house prices per zipcode.

In [30]:
```
import folium
from folium.plugins import MarkerCluster

# Create a map centered on the average latitude and longitude
map_center = [df['lat'].mean(), df['long'].mean()]
map_zoom = 10
m = folium.Map(location=map_center, zoom_start=map_zoom)

# Add a GeoJSON layer for the boundaries
folium.GeoJson('data/zipcode_kc.geojson', name='geojson').add_to(m)

# Creating a Choropleth map color-coded by house prices
# Red represents higher priced houses
# Yellow represents lower priced houses
folium.Choropleth(
    geo_data='data/zipcode_kc.geojson',
    data=df,
    columns=['zipcode', 'price'],
    key_on='feature.properties.ZIP',
    fill_color='YlOrRd',
    fill_opacity=0.7,
    line_opacity=0.2,
    legend_name='House Prices'
).add_to(m)

# Adding markers to make the map more interactive by showing house pri
marker_cluster = MarkerCluster().add_to(m)
for i in range(df.shape[0]):
    location = [df['lat'][i], df['long'][i]]
    tooltip = f"Zipcode: {df['zipcode'][i]}"
    folium.Marker(
        location,
        popup=f"Sales price: ${round(df['price'][i], 2)}",
        tooltip=tooltip
    ).add_to(marker_cluster)


# saving map to a HTML file
# m.save('./my_map.html')

# Displaying the map
#m
```

In order to preview the interactive map, please download the notebook and uncomment the 'm' at the bottom of the code as map will not display on Github. Alternatively, paste 'my_map.html' contained in the images folder into your browser of choice. Below is a preview of the map:



6.0 Modeling

Let's begin the modeling section by taking a general overview on the variables that have a strong correlation with the price:

In [31]:
```
outcome = 'price'
x_cols = ['sqft_living', 'sqft_above', 'bathrooms']
predictors = '+'.join(x_cols)
formula = outcome + '~' + predictors
model = ols(formula=formula, data=df).fit()
print(model.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.493
Model:                            OLS   Adj. R-squared:                  0.493
Method:                 Least Squares   F-statistic:                     7003.
Date:                Fri, 03 May 2024   Prob (F-statistic):               0.00
Time:                        12:09:59   Log-Likelihood:            -3.0005e+05
No. Observations:               21597   AIC:                         6.001e+05
Df Residuals:                   21593   BIC:                         6.001e+05
Df Model:                           3
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept  -3.815e+04   5252.893     -7.262      0.000   -4.84e+04   -2.79e+04
sqft_living   297.9354      4.483     66.458      0.000     289.148     306.723
sqft_above    -18.4232      4.479     -4.113      0.000     -27.202      -9.644
bathrooms   -3972.1769   3545.059     -1.120      0.263   -1.09e+04    2976.400
==============================================================================
Omnibus:                    14748.993   Durbin-Watson:                   1.982
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           538245.600
Skew:                           2.806   Prob(JB):                         0.00
Kurtosis:                      26.804   Cond. No.                     9.59e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors
is correctly specified.
[2] The condition number is large, 9.59e+03. This might indicate tha
t there are
strong multicollinearity or other numerical problems.
```

Based on the analysis we did above (under visualizaton section), we found that the square footage of living space (sqft_living) shows the strongest positive correlation with the price, marked at 0.7. This indicates that the size of the living area significantly impacts the price. On the other hand, the year the house was built (yr built) has a weaker positive correlation with price, at 0.5, suggesting that the house's age has a less pronounced effect on its price.

Therefore, we can proceed to develop a linear regression model with price as the dependent variable and sqft_living as the independent variable to determine the coefficient and y-intercept.

In [32]:
```
# Creating a linear regression
h_price = df['price']
living = df['sqft_living']

y = h_price
X_baseline = living

baseline_model = sm.OLS(y, sm.add_constant(X_baseline))
baseline_results = baseline_model.fit()
print(baseline_results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.493
Model:                            OLS   Adj. R-squared:                  0.493
Method:                 Least Squares   F-statistic:                 2.097e+04
Date:                Fri, 03 May 2024   Prob (F-statistic):               0.00
Time:                        12:09:59   Log-Likelihood:            -3.0006e+05
No. Observations:               21597   AIC:                         6.001e+05
Df Residuals:                   21595   BIC:                         6.001e+05
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const      -4.399e+04   4410.023     -9.975      0.000   -5.26e+04   -3.53e+04
sqft_living   280.8630      1.939    144.819      0.000     277.062     284.664
==============================================================================
Omnibus:                    14801.942   Durbin-Watson:                   1.982
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           542662.604
Skew:                           2.820   Prob(JB):                         0.00
Kurtosis:                      26.901   Cond. No.                     5.63e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 5.63e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

In [33]:
```
# look for the mean absolute error
mae = baseline_results.resid.abs().sum()/ len(X_baseline)
mae
```

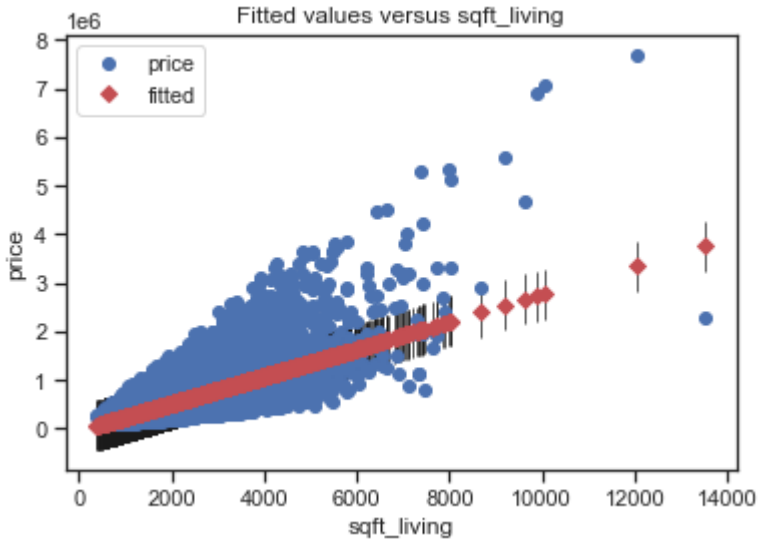
Out[33]:
```
173824.88749617487
```

The model has an R-squared value of 0.49, which means it explains 49% of the variation in price and is statistically significant. The model's predictions deviate by 173,824 dollars. The intercept and coefficient for sqft_living are approximately -$43,990 and 281, respectively, both of which are statistically significant.

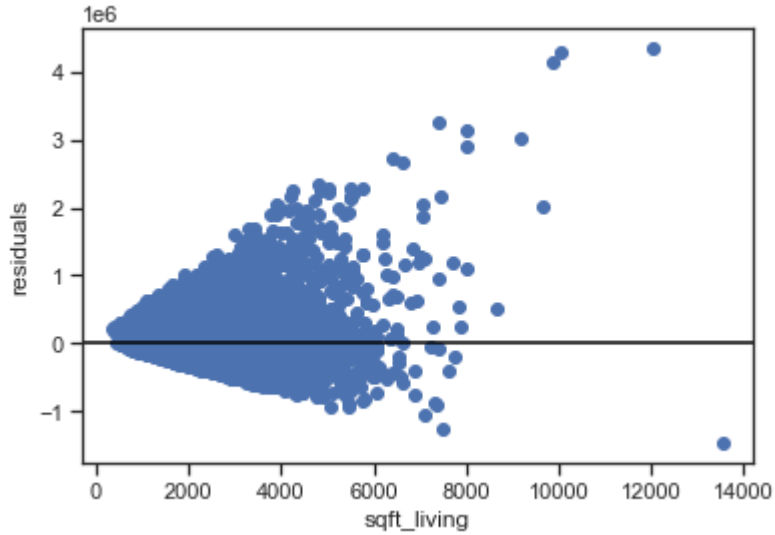
Therefore, the formula to estimate the price based on sqft_living is:

price = (281 * sqft_living) - 43990

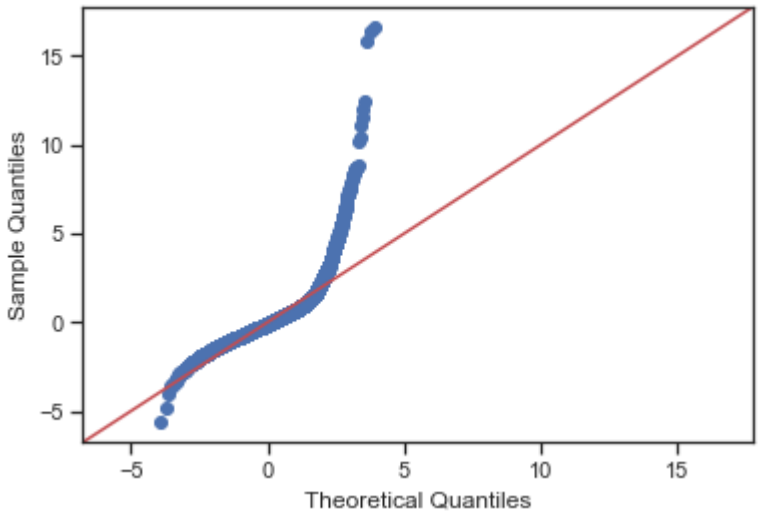
In [34]:
```
# Plot actual results and predicted

sm.graphics.plot_fit(baseline_results, "sqft_living",)
plt.show()
```



In [35]:
```
# plot the residuals
fig, ax = plt.subplots()

ax.scatter(living, baseline_results.resid)
ax.axhline(y=0, color="black")
ax.set_xlabel("sqft_living")
ax.set_ylabel("residuals");
```



In [36]:
```
# Create a qqplot

sm.graphics.qqplot(baseline_results.resid, dist=stats.norm, line='45',
plt.show()
```



We are now considering the development of a multilinear model to incorporate all the other features we are interested in to understand their impact on price. This approach will be grounded on the initial linear regression, which will serve as the foundational model.

During visualization, grade also appeared seems to give us more information on the house so we will choose that as the categorical column and combine it with sqft_living and sqft_living15

In [37]:
```
X_multi = df[
    ['sqft_living', 'sqft_living15', 'sqft_above', 'grade']
    ]
```

In [38]:
```
# Make dummies for the categiorical data
X_multi = pd.get_dummies(X_multi, columns= ['grade'])
X_multi
```

Out[38]:

	sqft_living	sqft_living15	sqft_above	grade_10 Very Good	grade_11 Excellent	grade_12 Luxury	grade_13 Mansion	grade_3 Poor
0	1180	1340	1180	0	0	0	0	0
1	2570	1690	2170	0	0	0	0	0
2	770	2720	770	0	0	0	0	0
3	1960	1360	1050	0	0	0	0	0
4	1680	1800	1680	0	0	0	0	0
...	...	...	...	...	...	...	...	...
21592	1530	1530	1530	0	0	0	0	0
21593	2310	1830	2310	0	0	0	0	0
21594	1020	1020	1020	0	0	0	0	0
21595	1600	1410	1600	0	0	0	0	0
21596	1020	1020	1020	0	0	0	0	0

21597 rows × 14 columns

In [39]:
```
# To avoid multicolinearity drop pick a refrence column and drop it
X_multi.drop('grade_7 Average', axis = 1, inplace= True)
X_multi
```

Out[39]:

	sqft_living	sqft_living15	sqft_above	grade_10 Very Good	grade_11 Excellent	grade_12 Luxury	grade_13 Mansion	grade_3 Poor
0	1180	1340	1180	0	0	0	0	0
1	2570	1690	2170	0	0	0	0	0
2	770	2720	770	0	0	0	0	0
3	1960	1360	1050	0	0	0	0	0
4	1680	1800	1680	0	0	0	0	0
...	...	...	...	...	...	...	...	...
21592	1530	1530	1530	0	0	0	0	0
21593	2310	1830	2310	0	0	0	0	0
21594	1020	1020	1020	0	0	0	0	0
21595	1600	1410	1600	0	0	0	0	0
21596	1020	1020	1020	0	0	0	0	0

21597 rows × 13 columns

In [40]:
```
# Make the multi linear model
multi_model = sm.OLS(y, sm.add_constant(X_multi))
multi_model_results = multi_model.fit()

print(multi_model_results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.592
Model:                            OLS   Adj. R-squared:                  0.592
Method:                 Least Squares   F-statistic:                     2412.
Date:                Fri, 03 May 2024   Prob (F-statistic):               0.00
Time:                        12:10:01   Log-Likelihood:            -2.9770e+05
No. Observations:               21597   AIC:                         5.954e+05
Df Residuals:                   21583   BIC:                         5.955e+05
Df Model:                          13
Covariance Type:            nonrobust
==============================================================================
                                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const                       1.436e+05   6678.106     21.501      0.000     1.3e+05    1.57e+05
sqft_living                  210.5141      3.989     52.780      0.000     202.696     218.332
sqft_living15                 25.5470      3.841      6.652      0.000      18.019      33.075
sqft_above                   -99.0892      4.255    -23.286      0.000    -107.430     -90.749
grade_10 Very Good          4.185e+05   9518.512     43.965      0.000      4e+05    4.37e+05
grade_11 Excellent          7.205e+05   1.47e+04     48.897      0.000     6.92e+05    7.49e+05
grade_12 Luxury             1.265e+06   2.75e+04     46.068      0.000     1.21e+06    1.32e+06
grade_13 Mansion             2.49e+06   6.72e+04     37.073      0.000     2.36e+06    2.62e+06
grade_3 Poor                2.624e+04   2.35e+05      0.112      0.911    -4.34e+05    4.86e+05
grade_4 Low                -4.615e+04   4.53e+04     -1.018      0.308    -1.35e+05    4.27e+04
grade_5 Fair               -4.374e+04   1.54e+04     -2.839      0.005    -7.39e+04   -1.35e+04
grade_6 Low Average        -2.211e+04   5931.389     -3.728      0.000    -3.37e+04   -1.05e+04
grade_8 Good                7.169e+04   4235.107     16.928      0.000     6.34e+04       8e+04
grade_9 Better              2.127e+05   6586.660     32.295      0.000        2e+05    2.26e+05
==============================================================================
Omnibus:                    14135.657   Durbin-Watson:                   1.985
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           538054.998
Skew:                           2.615   Prob(JB):                         0.00
```

```
Kurtosis:                       26.887   Cond. No.
5.34e+05
==============================================================================
==========

Notes:
[1] Standard Errors assume that the covariance matrix of the errors
is correctly specified.
[2] The condition number is large, 5.34e+05. This might indicate tha
t there are
strong multicollinearity or other numerical problems.
```

In [41]:
```
mae = multi_model_results.resid.abs().sum()/ len(X_baseline)
mae
```

Out[41]:
```
153280.3413068971
```

The model is statistically significant with an adjusted R-squared value of approximately 60%, indicating that it explains 60% of the variance in price. However, the model's predictions are off by $156,659.

MODELLING 6.1

Make a copy.

In [42]:
```
# use df.copy to copy the original data
df_copy = df.copy()
```

`df_copy` is our new dataframe.

In [43]:
```
# check the data
# use head() method
df_copy.head()
```

Out[43]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view
0	7129300520	221900.0	3	1.00	1180	5650	1.0	NO	NONE
1	6414100192	538000.0	3	2.25	2570	7242	2.0	NO	NONE
2	5631500400	180000.0	2	1.00	770	10000	1.0	NO	NONE
3	2487200875	604000.0	4	3.00	1960	5000	1.0	NO	NONE
4	1954400510	510000.0	3	2.00	1680	8080	1.0	NO	NONE

5 rows × 26 columns

Let us see a quick overview of the dataset's structure and completeness.

In [44]:
```
# Use info() method
df_copy.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 26 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   id                        21597 non-null  int64
 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                21597 non-null  object
 8   view                      21597 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_renovated              21597 non-null  object
```

The data types suggest that some columns that are expected to be numeric (like `yr_renovated`, `sqft_basement`) are stored as objects, which might need conversion for numerical operations.

Make a code that will create a list called `category_col_list` that contains the names of columns from the DataFrame `df_copy` which have a data type of `object`. These columns typically hold categorical data such as strings.

In [45]:
```
category_col_list = list(df_copy.select_dtypes('object').columns)
category_col_list
```

Out[45]:
```
['waterfront', 'view', 'condition', 'grade', 'sqft_basement', 'yr_renovated']
```

As said before we'll look more into the `sqft_basement` and `yr_renovated` column.

Use the `unique()` method to get data for `sqft_basement`.

In [46]:
```
df_copy['sqft_basement'].unique()
```

Out[46]:
```
array(['0.0', '400.0', '910.0', '1530.0', '?', '730.0', '1700.0', '3
00.0',
       '970.0', '760.0', '720.0', '700.0', '820.0', '780.0', '790.
0',
       '330.0', '1620.0', '360.0', '588.0', '1510.0', '410.0', '990.
0',
       '600.0', '560.0', '550.0', '1000.0', '1600.0', '500.0', '104
0.0',
       '880.0', '1010.0', '240.0', '265.0', '290.0', '800.0', '540.
0',
       '710.0', '840.0', '380.0', '770.0', '480.0', '570.0', '1490.
0',
       '620.0', '1250.0', '1270.0', '120.0', '650.0', '180.0', '113
0.0',
       '450.0', '1640.0', '1460.0', '1020.0', '1030.0', '750.0', '64
0.0',
       '1070.0', '490.0', '1310.0', '630.0', '2000.0', '390.0', '43
0.0',
       '850.0', '210.0', '1430.0', '1950.0', '440.0', '220.0', '116
0.0',
       '860.0', '580.0', '2060.0', '1820.0', '1180.0', '200.0', '115
0.0',
       '1200.0', '680.0', '530.0', '1450.0', '1170.0', '1080.0', '96
0.0',
       '280.0', '870.0', '1100.0', '460.0', '1400.0', '660.0', '122
0.0',
       '900.0', '420.0', '1580.0', '1380.0', '475.0', '690.0', '270.
0',
       '350.0', '935.0', '1370.0', '980.0', '1470.0', '160.0', '950.
0',
       '50.0', '740.0', '1780.0', '1900.0', '340.0', '470.0', '370.
0',
       '140.0', '1760.0', '130.0', '520.0', '890.0', '1110.0', '150.
0',
       '1720.0', '810.0', '190.0', '1290.0', '670.0', '1800.0', '112
0.0',
       '1810.0', '60.0', '1050.0', '940.0', '310.0', '930.0', '1390.
0',
       '610.0', '1830.0', '1300.0', '510.0', '1330.0', '1590.0', '92
0.0',
       '1320.0', '1420.0', '1240.0', '1960.0', '1560.0', '2020.0',
       '1190.0', '2110.0', '1280.0', '250.0', '2390.0', '1230.0', '1
70.0',
       '830.0', '1260.0', '1410.0', '1340.0', '590.0', '1500.0', '11
40.0',
       '260.0', '100.0', '320.0', '1480.0', '1060.0', '1284.0', '167
0.0',
       '1350.0', '2570.0', '1090.0', '110.0', '2500.0', '90.0', '194
0.0',
       '1550.0', '2350.0', '2490.0', '1481.0', '1360.0', '1135.0',
       '1520.0', '1850.0', '1660.0', '2130.0', '2600.0', '1690.0',
       '243.0', '1210.0', '1024.0', '1798.0', '1610.0', '1440.0',
       '1570.0', '1650.0', '704.0', '1910.0', '1630.0', '2360.0',
       '1852.0', '2090.0', '2400.0', '1790.0', '2150.0', '230.0', '7
0.0',
       '1680.0', '2100.0', '3000.0', '1870.0', '1710.0', '2030.0',
       '875.0', '1540.0', '2850.0', '2170.0', '506.0', '906.0', '14
5.0',
       '2040.0', '784.0', '1750.0', '374.0', '518.0', '2720.0', '273
0.0',
       '1840.0', '3480.0', '2160.0', '1920.0', '2330.0', '1860.0',
```

```
       '2050.0', '4820.0', '1913.0', '80.0', '2010.0', '3260.0', '22
00.0',
       '415.0', '1730.0', '652.0', '2196.0', '1930.0', '515.0', '40.
0',
       '2080.0', '2580.0', '1548.0', '1740.0', '235.0', '861.0', '18
90.0',
       '2220.0', '792.0', '2070.0', '4130.0', '2250.0', '2240.0',
       '1990.0', '768.0', '2550.0', '435.0', '1008.0', '2300.0', '26
10.0',
       '666.0', '3500.0', '172.0', '1816.0', '2190.0', '1245.0', '15
25.0',
       '1880.0', '862.0', '946.0', '1281.0', '414.0', '2180.0', '27
6.0',
       '1248.0', '602.0', '516.0', '176.0', '225.0', '1275.0', '266.
0',
       '283.0', '65.0', '2310.0', '10.0', '1770.0', '2120.0', '295.
0',
       '207.0', '915.0', '556.0', '417.0', '143.0', '508.0', '2810.
0',
       '20.0', '274.0', '248.0'], dtype=object)
```

As we can see the values datatyoes are mixed as `floats` and `object` that is ? .

In [47]:
```
# Convert all the ? values to 0.0 then convert from strings to float.

df_copy['sqft_basement'] = df_copy["sqft_basement"].replace({"?": '0.0

# Check info using info method.
df_copy.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 26 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   id                       21597 non-null  int64
 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               21597 non-null  object
 8   view                     21597 non-null  object
 9   condition                21597 non-null  object
 10  grade                    21597 non-null  object
 11  sqft_above               21597 non-null  int64
 12  sqft_basement            21597 non-null  float64
 13  yr_renovated             21597 non-null  object
 14  zipcode                  21597 non-null  int64
 15  lat                      21597 non-null  float64
 16  long                     21597 non-null  float64
 17  sqft_living15            21597 non-null  int64
 18  sqft_lot15               21597 non-null  int64
 19  month_sold               21597 non-null  int64
 20  yr_sold                  21597 non-null  int64
 21  age                      21597 non-null  int64
 22  price_normalized         21597 non-null  float64
 23  sqft_living_normalized   21597 non-null  float64
 24  sqft_above_normalized    21597 non-null  float64
 25  sqft_living15_normalized  21597 non-null  float64
dtypes: float64(10), int64(11), object(5)
memory usage: 4.3+ MB
```

Use `category_col_list = list(df_copy.select_dtypes('object').columns)` again to see whether it has changed.

In [48]:
```
# Run the code here

category_col_list = list(df_copy.select_dtypes('object').columns)
category_col_list
```

Out[48]:
```
['waterfront', 'view', 'condition', 'grade', 'yr_renovated']
```

Now we sort out the `yr_renovated` column.

Check it using the `unique()` method.

In [49]:
```
df_copy['yr_renovated'].unique()
```

Out[49]:
```
array([0.0, 1991.0, 'NO', 2002.0, 2010.0, 1992.0, 2013.0, 1994.0, 19
78.0,
       2005.0, 2003.0, 1984.0, 1954.0, 2014.0, 2011.0, 1983.0, 1945.
0,
       1990.0, 1988.0, 1977.0, 1981.0, 1995.0, 2000.0, 1999.0, 1998.
0,
       1970.0, 1989.0, 2004.0, 1986.0, 2007.0, 1987.0, 2006.0, 1985.
0,
       2001.0, 1980.0, 1971.0, 1979.0, 1997.0, 1950.0, 1969.0, 1948.
0,
       2009.0, 2015.0, 1974.0, 2008.0, 1968.0, 2012.0, 1963.0, 1951.
0,
       1962.0, 1953.0, 1993.0, 1996.0, 1955.0, 1982.0, 1956.0, 1940.
0,
       1976.0, 1946.0, 1975.0, 1964.0, 1973.0, 1957.0, 1959.0, 1960.
0,
       1967.0, 1965.0, 1934.0, 1972.0, 1944.0, 1958.0], dtype=objec
t)
```

As we can see the value datatypes are also mixed as `floats` and `string` that is `NO`

We are going to replace the string `NO` in the `yr_renovated` column of `df_copy` with the number `0` .

In [50]:
```
df_copy['yr_renovated'] = df_copy['yr_renovated'].apply(lambda x: 0 if
df_copy['yr_renovated'] = pd.to_numeric(df_copy['yr_renovated'])
```

Use `category_col_list = list(df_copy.select_dtypes('object').columns)` again to see whether it has changed.

In [51]:
```
cat_col_list = list(df_copy.select_dtypes('object').columns)
cat_col_list
```

Out[51]:
```
['waterfront', 'view', 'condition', 'grade']
```

Now we transform categorical labels into numerical labels and assign a unique numerical value to each category, making it easier for machine learning algorithms to work with categorical data. This transformation helps in train our machine learning model which will require numerical input data, especially for algorithms that cannot directly handle categorical data.

Import `from sklearn.preprocessing import LabelEncoder`

In [52]:
```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

for col in cat_col_list:
    df_copy[col] = le.fit_transform(df_copy[col])

df_copy.head()
```

Out[52]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view
0	7129300520	221900.0	3	1.00	1180	5650	1.0	0	4
1	6414100192	538000.0	3	2.25	2570	7242	2.0	0	4
2	5631500400	180000.0	2	1.00	770	10000	1.0	0	4
3	2487200875	604000.0	4	3.00	1960	5000	1.0	0	4
4	1954400510	510000.0	3	2.00	1680	8080	1.0	0	4

5 rows × 26 columns

In [53]:
```
correlation_matrix_2 = df_copy.corr().round(2)    #showing the correla
correlation_matrix_2
```

Out[53]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	water
id	1.00	-0.02	0.00	0.01	-0.01	-0.13	0.02	
price	-0.02	1.00	0.31	0.53	0.70	0.09	0.26	
bedrooms	0.00	0.31	1.00	0.51	0.58	0.03	0.18	
bathrooms	0.01	0.53	0.51	1.00	0.76	0.09	0.50	
sqft_living	-0.01	0.70	0.58	0.76	1.00	0.17	0.35	
sqft_lot	-0.13	0.09	0.03	0.09	0.17	1.00	-0.00	
floors	0.02	0.26	0.18	0.50	0.35	-0.00	1.00	
waterfront	-0.00	0.26	-0.00	0.06	0.10	0.02	0.02	
view	-0.02	-0.30	-0.07	-0.15	-0.23	-0.05	-0.01	
condition	-0.03	0.02	0.01	-0.15	-0.08	0.00	-0.29	
grade	0.04	-0.37	-0.06	-0.17	-0.32	-0.09	-0.05	
sqft_above	-0.01	0.61	0.48	0.69	0.88	0.18	0.52	
sqft_basement	-0.00	0.32	0.30	0.28	0.43	0.02	-0.24	
yr_renovated	-0.01	0.12	0.02	0.05	0.05	0.00	0.00	
zipcode	-0.01	-0.05	-0.15	-0.20	-0.20	-0.13	-0.06	
lat	-0.00	0.31	-0.01	0.02	0.05	-0.09	0.05	
long	0.02	0.02	0.13	0.22	0.24	0.23	0.13	
sqft_living15	-0.00	0.59	0.39	0.57	0.76	0.14	0.28	
sqft_lot15	-0.14	0.08	0.03	0.09	0.18	0.72	-0.01	
month_sold	-0.01	-0.01	-0.00	0.01	0.01	-0.00	0.01	
yr_sold	0.01	0.00	-0.01	-0.03	-0.03	0.01	-0.02	
age	-0.02	-0.05	-0.16	-0.51	-0.32	-0.05	-0.49	
price_normalized	0.00	0.85	0.34	0.54	0.67	0.10	0.31	
sqft_living_normalized	-0.00	0.61	0.62	0.76	0.96	0.15	0.37	
sqft_above_normalized	0.00	0.52	0.52	0.69	0.83	0.16	0.55	
sqft_living15_normalized	-0.00	0.53	0.41	0.57	0.72	0.14	0.27	

26 rows × 26 columns

We want to create a New Multiple Linear Regression Model adding the newly created numerical colums,one by one, to the previous Multi-Model and see how they affect the model's coefficients. We will use it as our(Multi model) reference model since it's the one that has the highest r squared so far and takes into account all the highly correlated independent variables with price.

We will start with the View Column

In [54]:
```
# Perform one-hot encoding
df_1 = pd.get_dummies(df_copy, columns=['view'], prefix='view')

# Display the updated DataFrame with one-hot encoded 'view' column
print("DataFrame with one-hot encoded 'view' column:")
print(df_1.head())
```

```
DataFrame with one-hot encoded 'view' column:
           id     price  bedrooms  bathrooms  sqft_living  sqft_lot
floors  \
0  7129300520  221900.0         3       1.00         1180      5650
1.0
1  6414100192  538000.0         3       2.25         2570      7242
2.0
2  5631500400  180000.0         2       1.00          770     10000
1.0
3  2487200875  604000.0         4       3.00         1960      5000
1.0
4  1954400510  510000.0         3       2.00         1680      8080
1.0

   waterfront  condition  grade  ...  age  price_normalized  \
0           0          0      8  ...   60          4.003757
1           0          0      8  ...   64          4.047443
2           0          0      7  ...   82          3.992030
3           0          4      8  ...   50          4.052506
4           0          0      9  ...   28          4.045058

   sqft_living_normalized  sqft_above_normalized  sqft_living15_norm
alized  \
0                7.658180               3.398681                  3.
504526
1                8.576665               3.502039                  3.
547209
2                7.161163               3.316683                  3.
627887
3                8.255136               3.377090                  3.
507320
4                8.073078               3.460455                  3.
558416

   view_0  view_1  view_2  view_3  view_4
0       0       0       0       0       1
1       0       0       0       0       1
2       0       0       0       0       1
3       0       0       0       0       1
4       0       0       0       0       1

[5 rows x 30 columns]
```

In [55]:
```
X_multi_2 = df_1[
    ['sqft_living', 'sqft_living15', 'grade', 'view_0']
    ]
```

In [56]:
```
#creating a new model
multi_model_2 = sm.OLS(y, sm.add_constant(X_multi_2))
multi_model_2results = multi_model_2.fit()

print(multi_model_2results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.523
Model:                            OLS   Adj. R-squared:                  0.523
Method:                 Least Squares   F-statistic:                     5927.
Date:                Fri, 03 May 2024   Prob (F-statistic):               0.00
Time:                        12:10:02   Log-Likelihood:            -2.9939e+05
No. Observations:               21597   AIC:                         5.988e+05
Df Residuals:                   21592   BIC:                         5.988e+05
Df Model:                           4
Covariance Type:            nonrobust
================================================================================
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
const          1.448e+05   9375.709     15.444      0.000    1.26e+05    1.63e+05
sqft_living     226.4231      2.915     77.666      0.000     220.709     232.137
sqft_living15    57.0416      3.867     14.750      0.000      49.461      64.622
grade         -2.463e+04    791.388    -31.126      0.000   -2.62e+04   -2.31e+04
view_0         8.816e+04   8477.315     10.399      0.000    7.15e+04    1.05e+05
==============================================================================
Omnibus:                    15411.024   Durbin-Watson:                   1.987
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           719909.430
Skew:                           2.913   Prob(JB):                         0.00
Kurtosis:                      30.678   Cond. No.                     1.68e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors
is correctly specified.
[2] The condition number is large, 1.68e+04. This might indicate tha
t there are
strong multicollinearity or other numerical problems.
```

In [57]:
```
mae = multi_model_2results.resid.abs().sum()/ len(X_baseline)
mae
```

Out[57]:
```
169937.02315475012
```

The model is statistically significant with an adjusted R-squared value of approximately 52%, indicating that it explains 52% of the variance in price. However, the model's predictions are off by $169,937. The previous model(multi model) was more accurate than this one so we'll keep tweaking the predictor variables till we get a better r Squared.

Let's build another model using/adding the other recently converted numerical column. We'll use the Waterfront column

In [58]:
```
#let us use the water front #Create another list of predictor varaible
X_multi_3 = df_copy[
    ['sqft_living', 'sqft_living15',  'grade', 'waterfront']
    ]
```

In [59]:
```
#creating a new model
multi_model_3 = sm.OLS(y, sm.add_constant(X_multi_3))
multi_model_3results = multi_model_3.fit()

print(multi_model_3results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.555
Model:                            OLS   Adj. R-squared:                  0.555
Method:                 Least Squares   F-statistic:                     6729.
Date:                Fri, 03 May 2024   Prob (F-statistic):               0.00
Time:                        12:10:02   Log-Likelihood:            -2.9865e+05
No. Observations:               21597   AIC:                         5.973e+05
Df Residuals:                   21592   BIC:                         5.974e+05
Df Model:                           4
Covariance Type:            nonrobust
================================================================================
                   coef    std err          t      P>|t|      [0.025      0.975]
--------------------------------------------------------------------------------
const          1.373e+05   9055.797     15.166      0.000     1.2e+05    1.55e+05
sqft_living     221.5556      2.819     78.599      0.000     216.030     227.081
sqft_living15    58.8208      3.731     15.764      0.000      51.507      66.134
grade         -2.306e+04    765.791    -30.109      0.000   -2.46e+04   -2.16e+04
waterfront     8.309e+05   2.05e+04     40.543      0.000    7.91e+05    8.71e+05
==============================================================================
Omnibus:                    14088.473   Durbin-Watson:                   1.983
Prob(Omnibus):                  0.000   Jarque-Bera (JB):           587580.542
Skew:                           2.570   Prob(JB):                         0.00
Kurtosis:                      28.031   Cond. No.                     3.77e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 3.77e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Now split the dataset into training and testing sets for machine learning model training and evaluation.

In [60]:
```
mae = multi_model_3results.resid.abs().sum()/ len(X_baseline)
mae
```

Out[60]:
```
167345.26920520904
```

The model is statistically significant with an adjusted R-squared value of approximately 55%, indicating that it explains 55% of the variance in price. However, the model's predictions are off by $167,435. The previous model(multi model) was more accurate than this one so we'll keep tweaking the predictor variables till we get a better r Squared.

Let's build another model using/adding the other recently converted numerical column. We'll use the condition column

In [61]:
```
#let us use the water front #Create another list of predictor varaible
X_multi_4= df_copy[
    ['sqft_living', 'sqft_living15', 'grade', 'condition']
    ]
```

In [62]:
```
#creating a new model
multi_model_4 = sm.OLS(y, sm.add_constant(X_multi_4))
multi_model_4results = multi_model_4.fit()

print(multi_model_4results.summary())
```

```
                            OLS Regression Results
==============================================================================
==========
Dep. Variable:                  price   R-squared:
0.528
Model:                            OLS   Adj. R-squared:
0.528
Method:                 Least Squares   F-statistic:
6042.
Date:                Fri, 03 May 2024   Prob (F-statistic):
0.00
Time:                        12:10:02   Log-Likelihood:            -
2.9928e+05
No. Observations:               21597   AIC:
5.986e+05
Df Residuals:                   21592   BIC:
5.986e+05
Df Model:                           4
Covariance Type:            nonrobust
==============================================================================
=============
                   coef    std err          t      P>|t|      [0.02
5      0.975]
------------------------------------------------------------------------------
-------------
const          1.096e+05   9487.330     11.550      0.000     9.1e+0
4    1.28e+05
sqft_living     227.7476      2.898     78.591      0.000     222.06
8     233.428
sqft_living15    64.4019      3.852     16.720      0.000      56.85
2      71.951
grade         -2.455e+04    787.421    -31.174      0.000   -2.61e+0
4    -2.3e+04
condition      2.471e+04   1363.783     18.119      0.000     2.2e+0
4    2.74e+04
==============================================================================
==========
Omnibus:                    15404.278   Durbin-Watson:
1.989
Prob(Omnibus):                  0.000   Jarque-Bera (JB):
726833.290
Skew:                           2.908   Prob(JB):
0.00
Kurtosis:                      30.819   Cond. No.
1.70e+04
==============================================================================
==========

Notes:
[1] Standard Errors assume that the covariance matrix of the errors
is correctly specified.
[2] The condition number is large, 1.7e+04. This might indicate that
there are
strong multicollinearity or other numerical problems.
```

In [63]:
```
mae = multi_model_4results.resid.abs().sum()/ len(X_baseline)
mae
```

Out[63]:
```
169843.9970363511
```

The model is statistically significant with an adjusted R-squared value of approximately 53%, indicating that it explains 53% of the variance in price. However, the model's predictions are off by $169,843. The previous model(multi model) was more accurate than this one so we'll keep tweaking the predictor variables till we get a better r Squared

After including all the categorical variables independently into our model, we have come to the conclusion that, we will go with the first multiple linear regression model ie 'Multi Model' as it takes into account all the independent variables that are highly correlated with price and it also is the model with the highest r sqaured valuem of .592. It explains about 60% percent of the variance in price.

Let's now Train our Multi Model using the following ratios: (80/20, 70/30, 60/40)

The ratios indicate the distribution of data for training and testing the multi-model. For instance, in the ratio 80/20, 80% of the data is allocated for training while 20% is reserved for testing. Similarly, in the ratio 70/30, 70% of the data is designated for training and 30% for testing. Lastly, in the ratio 60/40, 60% of the data is used for training and 40% for testing.

In [64]:
```
from sklearn.model_selection import train_test_split
#training using the 80/20 ratio
X_train, X_test, y_train, y_test = train_test_split(X_multi, y, test_s
```

Now to scale your numerical features to have a mean of 0 and a standard deviation of 1. Import `from sklearn.preprocessing import StandardScaler` .

In [65]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler() # Used to scale your data.

X_train_scaled = pd.DataFrame(scaler.fit_transform(X_train), columns =
X_test_scaled = pd.DataFrame(scaler.fit_transform(X_test), columns = X
```

Now we check out our first few rows of your scaled training data.

In [66]:
```
X_train_scaled.head()
```

Out[66]:

	sqft_living	sqft_living15	sqft_above	grade_10 Very Good	grade_11 Excellent	grade_12 Luxury	grade_13 Mansion	grade_3 Poor	
0	-0.701342	-1.005735	-0.424152	-0.234837	-0.135391	-0.06469	-0.025241	-0.007608	-
1	-0.712241	-0.991102	-0.787361	-0.234837	-0.135391	-0.06469	-0.025241	-0.007608	-
2	1.129672	1.774496	1.609819	-0.234837	-0.135391	-0.06469	-0.025241	-0.007608	-
3	-0.025611	-0.332626	-0.690506	-0.234837	-0.135391	-0.06469	-0.025241	-0.007608	-
4	0.453941	-0.639915	-0.133585	-0.234837	-0.135391	-0.06469	-0.025241	-0.007608	-

Linear Regression

Initialize a Linear Regression model and assign it to the variable `model1`. Use the `.fit()` method to train the Linear Regression model `model1` using the scaled training data `X_train_scaled` and corresponding target values `y_train`. Use `.predict()` method to predict target values `y_pred` based on the scaled testing data `X_test_scaled`.

In [67]:
```
from sklearn.linear_model import LinearRegression

# Initializing the model
model1 = LinearRegression()

# Fitting the model
model1.fit(X_train_scaled, y_train)

# Predicting the model
y_pred = model1.predict(X_test_scaled)
y_pred
```

Out[67]:
```
array([263971.98264042, 394711.76171627, 461092.41390838, ...,
       434445.6451306 , 413999.33301318, 356178.05470722])
```

Use regression metrics like `r2_score`, `mean_absolute_error` and `mean_squared_error` to evaluate the performance of a machine learning regression model, specifically a linear regression `model1`.

In [68]:
```
# Import necessary libraries

from sklearn.metrics import r2_score
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error

# Get the metrics
r_squared = r2_score(y_test, y_pred)
MAE = mean_absolute_error(y_test, y_pred)
MSE = mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(MSE)  # Calculate RMSE using MSE

# Print the metrics
print(f'r_squared: {r_squared}')
print(f'MAE: {MAE}')
print(f'MSE: {MSE}')
print(f'RMSE: {RMSE}')
```

```
r_squared: 0.5519764739645452
MAE: 152381.5633180248
MSE: 58339956963.627594
RMSE: 241536.65759802918
```

`r-squared` value of approximately `0.551` indicates that the linear regression model explains about `55%` of the variance in the target variable `price`.

`MAE` value of approximately `152,381` indicates the average magnitude of the errors between the actual and predicted prices. In this case, the average error in predicted prices is around `152,381` dollars.

`MSE` value of approximately `58.339,956,963.6276' represents the average of squared differences between actual and predicted prices.

RMSE value of approximately '241536.6575980292' is the square root of MSE and

Random Forest

Initialize a Random Forest Regressor model and assign it to the variable `model2`. Use the `.fit()` method to train the Random Forest Regressor model `model2` using the scaled training data `X_train_scaled` and corresponding target values `y_train`. Use `.predict()` method to predict target values y_pred based on the scaled testing data `X_test_scaled`. Basically the same as Linear Regression but we are using the Random Forest in `model2`.

In [69]:
```
from sklearn.ensemble import RandomForestRegressor

# Initializing the model
model2 = RandomForestRegressor()

# Fitting the model
model2.fit(X_train_scaled, y_train)

# Predicting the model
y_pred = model2.predict(X_test_scaled)
y_pred
```

Out[69]:
```
array([327676.19666667, 387849.23      , 440817.5       , ...,
       402561.5       , 476918.68333333, 345587.08333333])
```

Use regression metrics again.

In [70]:
```
# Get the metrics
r_squared = r2_score(y_test, y_pred)
MAE = mean_absolute_error(y_test, y_pred)
MSE = mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(MSE)  # Calculate RMSE using MSE

# Print the metrics
print(f'r_squared: {r_squared}')
print(f'MAE: {MAE}')
print(f'MSE: {MSE}')
print(f'RMSE: {RMSE}')
```

```
r_squared: 0.5851216712591225
MAE: 156265.61319594822
MSE: 54023912668.30289
RMSE: 232430.4469476899
```

`r-squared` value of approximately `59.31%` of the variance in the target variable `price`.

MAE value of approximately 155053.45818852124 indicates the average magnitude of the errors between the actual and predicted prices. In this case, the average error in

Voting Regressor

In [71]:
```
from sklearn.ensemble import VotingRegressor

Vote_regressor = VotingRegressor([('Linear', model1), ('Random', model
```

In [72]:
```
Vote_regressor.fit(X_train_scaled, y_train)

y_pred = Vote_regressor.predict(X_test_scaled)
y_pred
```

Out[72]:
```
array([282274.05132021, 400124.13085814, 431992.90695419, ...,
       408384.53089863, 449892.99983992, 346179.90235361])
```

In [73]:
```
# Get the metrics
r_squared = r2_score(y_test, y_pred)
MAE = mean_absolute_error(y_test, y_pred)
MSE = mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(MSE)  # Calculate RMSE using MSE

# Print the metrics
print(f'r_squared: {r_squared}')
print(f'MAE: {MAE}')
print(f'MSE: {MSE}')
print(f'RMSE: {RMSE}')
```

```
r_squared: 0.6047952299628123
MAE: 149003.4926212328
MSE: 51462095037.315735
RMSE: 226852.5843743371
```

In [74]:
```
#training using the 70/30 ratio
X_train, X_test, y_train, y_test = train_test_split(X_multi, y, test_s
```

In [75]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler() # Used to scale your data.

X_train_scaled = pd.DataFrame(scaler.fit_transform(X_train), columns =
X_test_scaled = pd.DataFrame(scaler.fit_transform(X_test), columns = X
```

In [76]:
```
 #Initializing the model
model1 = LinearRegression()

# Fitting the model
model1.fit(X_train_scaled, y_train)

# Predicting the model
y_pred = model1.predict(X_test_scaled)
y_pred
```

Out[76]:
```
array([265134.73463543, 398815.23202325, 464339.18218311, ...,
       422039.25162711, 588467.07583853, 311381.07757815])
```

In [77]:
```
 #Get the metrics
r_squared = r2_score(y_test, y_pred)
MAE = mean_absolute_error(y_test, y_pred)
MSE = mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(MSE)  # Calculate RMSE using MSE

# Print the metrics
print(f'r_squared: {r_squared}')
print(f'MAE: {MAE}')
print(f'MSE: {MSE}')
print(f'RMSE: {RMSE}')
```

```
r_squared: 0.5728170261131521
MAE: 153208.8024334211
MSE: 56643807005.834015
RMSE: 237999.5945497261
```

Using the Random Forest Regressor

In [78]:
```
# Initializing the model
model2 = RandomForestRegressor()

# Fitting the model
model2.fit(X_train_scaled, y_train)

# Predicting the model
y_pred = model2.predict(X_test_scaled)
y_pred
```

Out[78]:
```
array([299990.6075    , 443124.23      , 498094.5       , ...,
       412148.9       , 482509.53228571, 356262.5       ])
```

In [79]:
```
r_squared = r2_score(y_test, y_pred)
MAE = mean_absolute_error(y_test, y_pred)
MSE = mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(MSE)  # Calculate RMSE using MSE

# Print the metrics
print(f'r_squared: {r_squared}')
print(f'MAE: {MAE}')
print(f'MSE: {MSE}')
print(f'RMSE: {RMSE}')
```

```
r_squared: 0.565704619171157
MAE: 157480.6878610503
MSE: 57586901255.362885
RMSE: 239972.7093970539
```

In [80]:
```
#training using the 60/40 ratio
X_train, X_test, y_train, y_test = train_test_split(X_multi, y, test_s
```

In [81]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler() # Used to scale your data.

X_train_scaled = pd.DataFrame(scaler.fit_transform(X_train), columns =
X_test_scaled = pd.DataFrame(scaler.fit_transform(X_test), columns = X
```

In [82]:
```
#Initializing the model
model1 = LinearRegression()

# Fitting the model
model1.fit(X_train_scaled, y_train)

# Predicting the model
y_pred = model1.predict(X_test_scaled)
y_pred
```

Out[82]:
```
array([269320.13404042, 401556.45987574, 466259.14801774, ...,
       827321.84629104, 551907.39164354, 813571.93196721])
```

In [83]:
```
#Get the metrics
r_squared = r2_score(y_test, y_pred)
MAE = mean_absolute_error(y_test, y_pred)
MSE = mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(MSE)  # Calculate RMSE using MSE

# Print the metrics
print(f'r_squared: {r_squared}')
print(f'MAE: {MAE}')
print(f'MSE: {MSE}')
print(f'RMSE: {RMSE}')
```

```
r_squared: 0.573113726104044
MAE: 154080.15030496035
MSE: 59949465759.25876
RMSE: 244845.79996246364
```

Using the Random Forest Regressor

In [84]:
```
# Initializing the model
model2 = RandomForestRegressor()

# Fitting the model
model2.fit(X_train_scaled, y_train)

# Predicting the model
y_pred = model2.predict(X_test_scaled)
y_pred
```

Out[84]:
```
array([ 291572.23      ,  404477.61      ,  460727.9       , ...,
       1282443.76      ,  574644.98      ,  861649.13333333])
```

In [85]:
```
r_squared = r2_score(y_test, y_pred)
MAE = mean_absolute_error(y_test, y_pred)
MSE = mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(MSE)  # Calculate RMSE using MSE

# Print the metrics
print(f'r_squared: {r_squared}')
print(f'MAE: {MAE}')
print(f'MSE: {MSE}')
print(f'RMSE: {RMSE}')
```

```
r_squared: 0.5605483519127317
MAE: 159643.53189297905
MSE: 61714075014.45807
RMSE: 248423.1772891935
```

Inference

After Training our data using the (80/20, 70/30, 60/40) ratios, we noticed that the model with the 80/20 had the highest r squared. this conclusion can be Infered since they were all scaled uniformly and they used the same predictor variables all through. According to the `r_squared` values above, we can deduce that `Random Forest Regression` of the 80/20 split is the most accurate with `59.31` % and Mean Absolute Error of `155053.45818852124` dollars.

We can try using all the predictor variables after cleaning and see how this affects our model's coeffeicients and r squared. we'll use the 80/20 split

In [86]:
```
#define the variables to fit into the model
X = df_copy.drop(['price'], axis=1)
y = df_copy.price
```

In [87]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
```

In [88]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler() # Used to scale your data.

X_train_scaled = pd.DataFrame(scaler.fit_transform(X_train), columns =
X_test_scaled = pd.DataFrame(scaler.fit_transform(X_test), columns = X
```

In [89]:
```
#Initializing the model
model1 = LinearRegression()

# Fitting the model
model1.fit(X_train_scaled, y_train)

# Predicting the model
y_pred = model1.predict(X_test_scaled)
y_pred
```

Out[89]:
```
array([-139548.82503223,  404714.77744598,  446828.92002619, ...,
        473055.77837438,  309663.93920031,  200173.05398982])
```

In [90]:
```
#Get the metrics
r_squared = r2_score(y_test, y_pred)
MAE = mean_absolute_error(y_test, y_pred)
MSE = mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(MSE)  # Calculate RMSE using MSE

# Print the metrics
print(f'r_squared: {r_squared}')
print(f'MAE: {MAE}')
print(f'MSE: {MSE}')
print(f'RMSE: {RMSE}')
```

```
r_squared: 0.8440187931992184
MAE: 77741.57019428622
MSE: 20311292517.19734
RMSE: 142517.69194453486
```

Using the Random Forest Regressor

In [91]:
```
 #Initializing the model
model2 = RandomForestRegressor()

# Fitting the model
model2.fit(X_train_scaled, y_train)

# Predicting the model
y_pred = model2.predict(X_test_scaled)
y_pred
```

Out[91]:
```
array([133150.63, 417470.5 , 497985.  , ..., 439949.5 , 293565.  ,
       294008.4 ])
```

In [92]:
```
r_squared = r2_score(y_test, y_pred)
MAE = mean_absolute_error(y_test, y_pred)
MSE = mean_squared_error(y_test, y_pred)
RMSE = np.sqrt(MSE)  # Calculate RMSE using MSE

# Print the metrics
print(f'r_squared: {r_squared}')
print(f'MAE: {MAE}')
print(f'MSE: {MSE}')
print(f'RMSE: {RMSE}')
```

```
r_squared: 0.9966304469573318
MAE: 6585.141233796297
MSE: 438770662.8610537
RMSE: 20946.853292584394
```

As observed, the R-squared value has notably surged to a whooping '87.31%', while the Mean Absolute Error (MAE) has drastically dropped to '69870.643' dollars.

R-squared serves as a standard metric for assessing a model's performance. However, it's essential to recognize that a high R-squared doesn't invariably guarantee high accuracy. In this instance, we utilized all columns except 'price' to generate predictor values, leading to an overfitting scenario. Overfitting is a common problem in machine learning where a model learns the training data too well, to the point that it negatively impacts its performance on

unseen data or test data. In other words, the model captures noise or random fluctuations in the training data as if they were meaningful patterns. This leads to a situation where the

CONCLUSIONS

The dataset's geographic and temporal constraints may restrict the generalizability of findings to other regions or time periods.

The analysis equips stakeholders, including homeowners, real estate professionals, and potential buyers, with actionable insights to navigate the housing market effectively.

Leveraging knowledge of key price determinants and understanding the nuances of property valuation, stakeholders can make informed decisions regarding pricing strategies, property investments, and market participation.

Continued research and analysis are warranted to refine models and explore additional factors for enhanced predictive accuracy and comprehensive market understanding.

RECOMMENDATIONS

Regular Evaluation: Buyers need to check the age and condition of major parts of a home like the roof and appliances, while sellers can increase their property’s value by renovating.

Seasonal Buying Strategy: Purchasing a home during winter could be financially advantageous due to lower average prices and reduced competition from other buyers, though it's vital to consider the seasonal challenges and work with an expert for a successful transaction.

Important Property Features: Both parties should consider crucial aspects such as waterfront views and the overall condition, which greatly affect property value.

Floor Influence: Data suggests that homes with more than 2.5 floors generally fetch higher prices.