Final Project Submission

Please fill out:

- Student name: Harriet Joseph
- Student pace: part time
- Scheduled project review date/time:
- Instructor name: Samuel Jane
- Blog post URL:

1. BUSINESS UNDERSTANDING

Stakeholder:

The stakeholder for this dataset could be a real estate agency or a property management company that deals with buying, selling, and renting houses in King County. They might be interested in analyzing this dataset to gain insights into the housing market of the county and improve their business decisions and also give accurate advice to homeowners on how to increase the value of their homes and by what amount

Business problem:

The business problem that this stakeholder might face is determining the factors that influence house prices in the county. By understanding these factors, they could price their properties more accurately, invest in the right locations, and negotiate better deals with buyers and sellers. The stakeholder might also be interested in identifying the most desirable neighborhoods and property features that attract buyers and renters, so that they can focus their marketing efforts and increase their sales and revenue. In summary, the stakeholder wants to use regression modeling to predict house prices and gain insights into the factors that affect house values in King County.

This project uses the King County House Sales dataset, which can be found in kc_house_data.csv and description of the column names can be found in column_names.md.

Column Names and descriptions for Kings County Data Set

- **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

2.DATA UNDERSTANDING

loading data

In [1]:

```
#import needed libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

import warnings
warnings.filterwarnings('ignore')
```

In [2]:

```
#load the king county house sales dataset
df = pd.read_csv('kc_house_data.csv', index_col = 0)
df
```

Out[2]:

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

21597 rows × 20 columns

In [3]:

```
df.columns
```

Out[3]:

```
Index(['date', 'price', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lo
t',
       'floors', 'waterfront', 'view', 'condition', 'grade', 'sqft_above',
       'sqft_basement', 'yr_built', 'yr_renovated', 'zipcode', 'lat', 'lon
g',
       'sqft_living15', 'sqft_lot15'],
      dtype='object')
```

In [4]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21597 entries, 7129300520 to 1523300157
Data columns (total 20 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   date           21597 non-null  object 
 1   price          21597 non-null  float64
 2   bedrooms       21597 non-null  int64  
 3   bathrooms      21597 non-null  float64
 4   sqft_living    21597 non-null  int64  
 5   sqft_lot       21597 non-null  int64  
 6   floors         21597 non-null  float64
 7   waterfront     19221 non-null  float64
 8   view           21534 non-null  float64
 9   condition      21597 non-null  int64  
 10  grade          21597 non-null  int64  
 11  sqft_above     21597 non-null  int64  
 12  sqft_basement  21597 non-null  object 
 13  yr_built       21597 non-null  int64  
 14  yr_renovated   17755 non-null  float64
 15  zipcode        21597 non-null  int64  
 16  lat            21597 non-null  float64
 17  long           21597 non-null  float64
 18  sqft_living15  21597 non-null  int64  
 19  sqft_lot15     21597 non-null  int64  
dtypes: float64(8), int64(10), object(2)
memory usage: 3.5+ MB
```

In [5]:

```
print(df.describe())
```

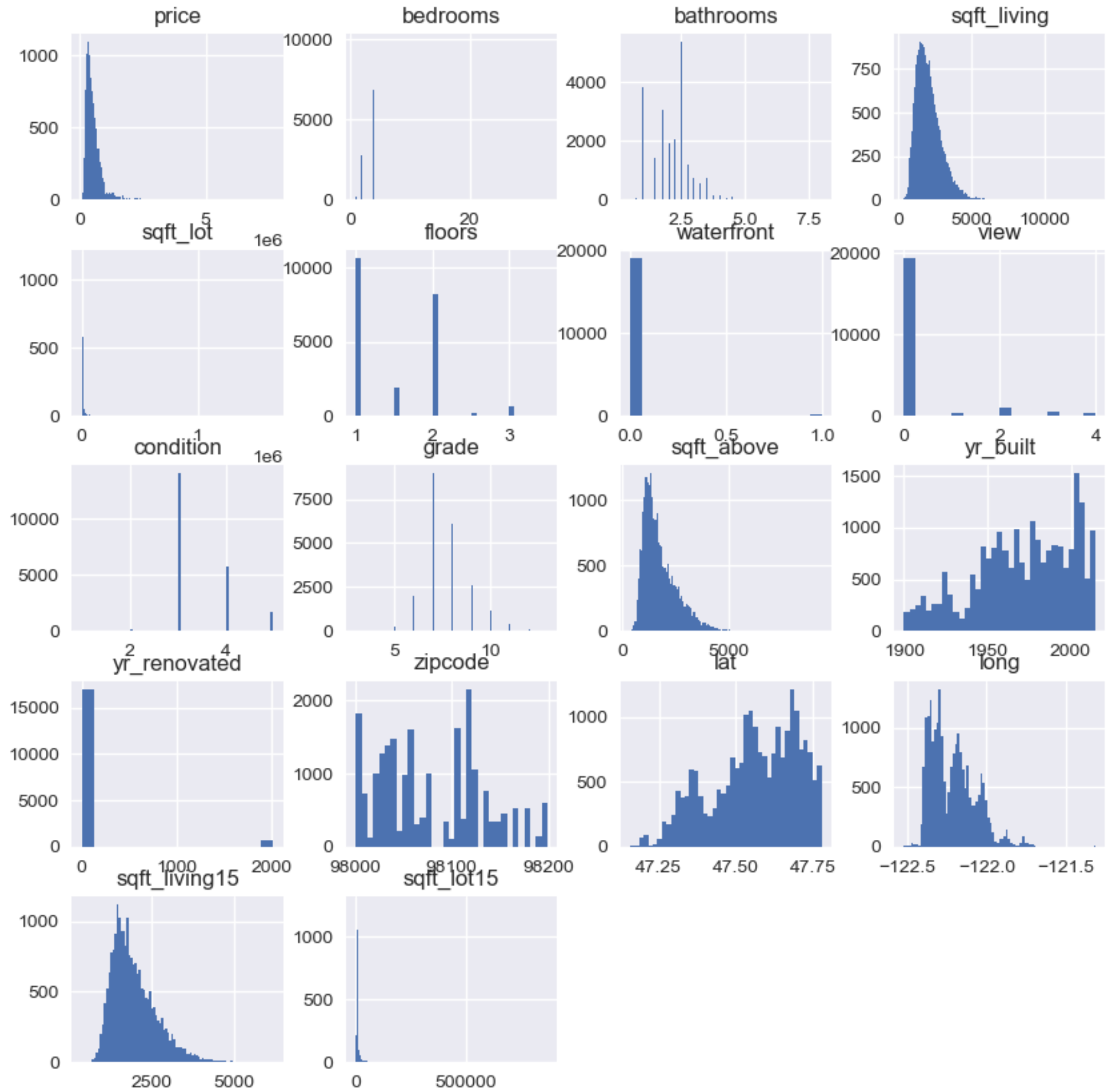
```
              price      bedrooms     bathrooms   sqft_living      sqft_lo
t  \
count  2.159700e+04  21597.000000  21597.000000  21597.000000  2.159700e+0
4
mean   5.402966e+05      3.373200      2.115826   2080.321850  1.509941e+0
4
std    3.673681e+05      0.926299      0.768984    918.106125  4.141264e+0
4
min    7.800000e+04      1.000000      0.500000    370.000000  5.200000e+0
2
25%    3.220000e+05      3.000000      1.750000   1430.000000  5.040000e+0
3
50%    4.500000e+05      3.000000      2.250000   1910.000000  7.618000e+0
3
75%    6.450000e+05      4.000000      2.500000   2550.000000  1.068500e+0
4
max    7.700000e+06     33.000000      8.000000  13540.000000  1.651359e+0
6

             floors    waterfront          view     condition         grad
e  \
count  21597.000000  19221.000000  21534.000000  21597.000000  21597.00000
0
mean       1.494096      0.007596      0.233863      3.409825      7.65791
5
std        0.539683      0.086825      0.765686      0.650546      1.17320
0
min        1.000000      0.000000      0.000000      1.000000      3.00000
0
25%        1.000000      0.000000      0.000000      3.000000      7.00000
0
50%        1.500000      0.000000      0.000000      3.000000      7.00000
0
75%        2.000000      0.000000      0.000000      4.000000      8.00000
0
max        3.500000      1.000000      4.000000      5.000000     13.00000
0

         sqft_above      yr_built  yr_renovated       zipcode           la
t  \
count  21597.000000  21597.000000  17755.000000  21597.000000  21597.00000
0
mean    1788.596842   1970.999676     83.636778  98077.951845     47.56009
3
std      827.759761     29.375234    399.946414     53.513072      0.13855
2
min      370.000000   1900.000000      0.000000  98001.000000     47.15590
0
25%     1190.000000   1951.000000      0.000000  98033.000000     47.47110
0
50%     1560.000000   1975.000000      0.000000  98065.000000     47.57180
0
75%     2210.000000   1997.000000      0.000000  98118.000000     47.67800
0
max     9410.000000   2015.000000   2015.000000  98199.000000     47.77760
0

               long  sqft_living15     sqft_lot15
count  21597.000000   21597.000000   21597.000000
mean    -122.213982    1986.620318   12758.283512
std        0.140724     685.230472   27274.441950
```

```
min     -122.519000    399.000000     651.000000
25%     -122.328000   1490.000000    5100.000000
50%     -122.231000   1840.000000    7620.000000
75%     -122.125000   2360.000000   10083.000000
max     -121.315000   6210.000000  871200.000000
```

- to understand the distribbution of the data plot histogram for all columns

In [6]:

```python
plt.style.use('seaborn')
# Create histograms for all variables
df.hist(figsize=(10,10), bins= 'auto')
plt.show()
```



In [7]:

```
#our target variable is the price
#The dataset contains information about house sales in King County, Washington state, USA
#There are 21,597 entries (rows) and 20 columns.
#Each row represents a different house sale and each column represents a different attrib

#with over 20,000 observations, we likely have enough data to build a reasonably complex 
#The distribution of the data is not very well specified for all the predictors at this s
#summary statistics provided that the price has a wide range of values,
#with a mean of $540,296 and a standard deviation of $367,368.

#comprises of  are continuous,discrete and categorical data as shown in the plot above.
```

3.DATA PREPARATION

- **DROP IRRELEVANT COLUMNS**

In [8]:

```
# Declare relevant columns
relevant_columns = [
    'price',   #price of houses
    'bedrooms',   #number of bedrooms
    'bathrooms',   #number of bathrooms
    'sqft_living', #square - footage of the home
    'sqft_lot',     #square - footage of the lot
    'floors',      #floors(level) of the house
    'waterfront',  #House which has a view to a waterfront
    'condition',#How good the condition is ( Overall )
    'grade', #overall grade given to the housing unit, based on King County grading syste
    'view',        #has it been viewed
    'yr_built',     #Built Year
    'yr_renovated', # Year when house was renovated
    'zipcode',      #zip
    'sqft_above',   #square footage of house apart from basement
    'sqft_basement' #square footage of the basement
]


# Reassign dataframe so that it only contains relevant columns
df = df.loc[:, relevant_columns]

# Visually inspect new dataframe
df
```

Out[8]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	conditio
id								
7129300520	221900.0	3	1.00	1180	5650	1.0	NaN	
6414100192	538000.0	3	2.25	2570	7242	2.0	0.0	
5631500400	180000.0	2	1.00	770	10000	1.0	0.0	
2487200875	604000.0	4	3.00	1960	5000	1.0	0.0	
1954400510	510000.0	3	2.00	1680	8080	1.0	0.0	
...	...	...	...	...	...	...	...	.
263000018	360000.0	3	2.50	1530	1131	3.0	0.0	
6600060120	400000.0	4	2.50	2310	5813	2.0	0.0	
1523300141	402101.0	2	0.75	1020	1350	2.0	0.0	
291310100	400000.0	3	2.50	1600	2388	2.0	NaN	
1523300157	325000.0	2	0.75	1020	1076	2.0	0.0	

21597 rows × 15 columns

In [9]:

```
#check that the shape is correct
# X_train should have the same number of rows as before
assert df.shape[0] == 21597

# Now X_train should only have as many columns as relevant_columns
assert df.shape[1] == len(relevant_columns)
```

- **HANDLING MISSING VALUES**

In [10]:

```
df.isna().sum()
```

Out[10]:

```
price              0
bedrooms           0
bathrooms          0
sqft_living        0
sqft_lot           0
floors             0
waterfront      2376
condition          0
grade              0
view              63
yr_built           0
yr_renovated    3842
zipcode            0
sqft_above         0
sqft_basement      0
dtype: int64
```

Ok, it looks like we have some NaNs in waterfront, view and yr_renovated, do these NaNs actually represent *missing* values, or is there some real value/category being represented by NaN?

- begin with yr_renovated

In [11]:

```
#yr_renovated is not categorical ,therefore we can assume that zero represents houses tha
#therefore fill missing values with zeroes
df['yr_renovated'].fillna(0, inplace=True)
```

- then views we realise that the view is categorical ranges from 0 to 4, since it represents small percentage we can drop thr rows with NANs

In [12]:

```
df.dropna(subset=['view'], inplace=True)
```

- lastly check on waterfront.

- *waterfront is a categorical data, where o represents, house has no view to a waterfront and 1 represents a house with a view to a waterfront**

In [13]:

```
# inspecting the waterfront column
print(df['waterfront'].value_counts())
print(df['waterfront'].unique())
```

```
0.0    19019
1.0      145
Name: waterfront, dtype: int64
[nan  0.  1.]
```

In [14]:

```
# Calculate the mode of the waterfront column
waterfront_mode = df['waterfront'].mode()[0]

# Fill in the missing values with the mode
df['waterfront'] = df['waterfront'].fillna(waterfront_mode)
```

In [15]:

```
print(df['waterfront'].unique())
```

```
[0. 1.]
```

In [16]:

```
#check again if the changes were made
df.isna().sum()
```

Out[16]:

```
price            0
bedrooms         0
bathrooms        0
sqft_living      0
sqft_lot         0
floors           0
waterfront       0
condition        0
grade            0
view             0
yr_built         0
yr_renovated     0
zipcode          0
sqft_above       0
sqft_basement    0
dtype: int64
```

- **CONVERT CATEGORICAL VALUES TO NUMBERS**

In [17]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21534 entries, 7129300520 to 1523300157
Data columns (total 15 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   price          21534 non-null  float64
 1   bedrooms       21534 non-null  int64  
 2   bathrooms      21534 non-null  float64
 3   sqft_living    21534 non-null  int64  
 4   sqft_lot       21534 non-null  int64  
 5   floors         21534 non-null  float64
 6   waterfront     21534 non-null  float64
 7   condition      21534 non-null  int64  
 8   grade          21534 non-null  int64  
 9   view           21534 non-null  float64
 10  yr_built       21534 non-null  int64  
 11  yr_renovated   21534 non-null  float64
 12  zipcode        21534 non-null  int64  
 13  sqft_above     21534 non-null  int64  
 14  sqft_basement  21534 non-null  object 
dtypes: float64(6), int64(8), object(1)
memory usage: 2.6+ MB
```

In [18]:

```
print(df['sqft_basement'].value_counts())
```

```
0.0       12798
?           452
600.0       216
500.0       209
700.0       207
          ...  
3480.0        1
1840.0        1
2730.0        1
2720.0        1
248.0         1
Name: sqft_basement, Length: 302, dtype: int64
```

- **data type conversion**

In [19]:

```
#sqft_basement and date are in object data type
#convert the sqft_basement type to float64
#the column has a special character ? remove that first
df = df[df['sqft_basement'] != '?']
df['sqft_basement'] = df['sqft_basement'].astype('float64')
```

In [20]:

```
#check for datatypes
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21082 entries, 7129300520 to 1523300157
Data columns (total 15 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   price          21082 non-null  float64
 1   bedrooms       21082 non-null  int64  
 2   bathrooms      21082 non-null  float64
 3   sqft_living    21082 non-null  int64  
 4   sqft_lot       21082 non-null  int64  
 5   floors         21082 non-null  float64
 6   waterfront     21082 non-null  float64
 7   condition      21082 non-null  int64  
 8   grade          21082 non-null  int64  
 9   view           21082 non-null  float64
 10  yr_built       21082 non-null  int64  
 11  yr_renovated   21082 non-null  float64
 12  zipcode        21082 non-null  int64  
 13  sqft_above     21082 non-null  int64  
 14  sqft_basement  21082 non-null  float64
dtypes: float64(7), int64(8)
memory usage: 2.6 MB
```

In [21]:

```
# Print a summary of the dataset
df
```

Out[21]:

id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	conditio
7129300520	221900.0	3	1.00	1180	5650	1.0	0.0	
6414100192	538000.0	3	2.25	2570	7242	2.0	0.0	
5631500400	180000.0	2	1.00	770	10000	1.0	0.0	
2487200875	604000.0	4	3.00	1960	5000	1.0	0.0	
1954400510	510000.0	3	2.00	1680	8080	1.0	0.0	
...	...	...	...	...	...	...	...	
263000018	360000.0	3	2.50	1530	1131	3.0	0.0	
6600060120	400000.0	4	2.50	2310	5813	2.0	0.0	
1523300141	402101.0	2	0.75	1020	1350	2.0	0.0	
291310100	400000.0	3	2.50	1600	2388	2.0	0.0	
1523300157	325000.0	2	0.75	1020	1076	2.0	0.0	

21082 rows × 15 columns

- **CHECK FOR MULTICOLLINEARITY**

In [22]:

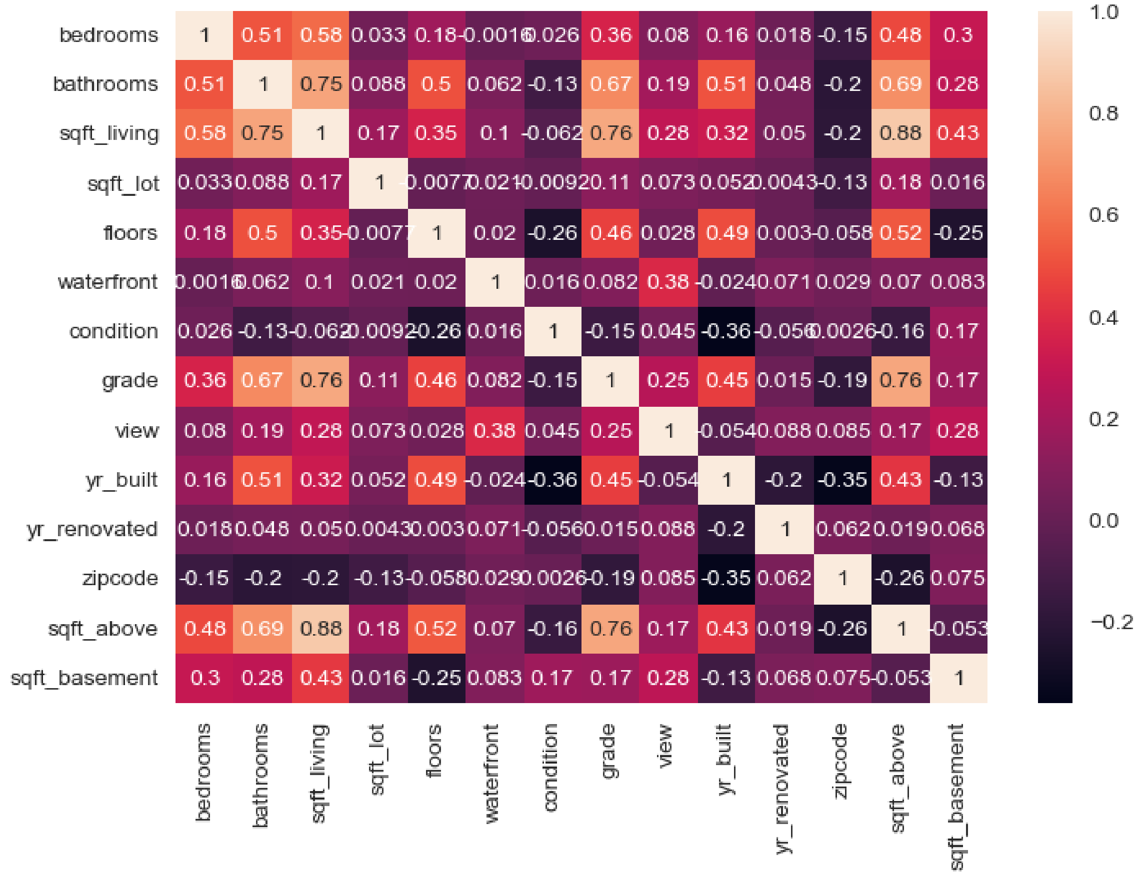
```
relevant_columns
```

Out[22]:

```
['price',
 'bedrooms',
 'bathrooms',
 'sqft_living',
 'sqft_lot',
 'floors',
 'waterfront',
 'condition',
 'grade',
 'view',
 'yr_built',
 'yr_renovated',
 'zipcode',
 'sqft_above',
 'sqft_basement']
```

In [23]:

```
# checking for multicollinearity
# we can use a heatmap to visualize how the variables are correlated

# create a DataFrame from the relevant columns
relevant_df = df[['price', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors',
                  'view', 'yr_built', 'yr_renovated', 'zipcode', 'sqft_above', 'sqft_base

# select the features to include in the correlation matrix
features = relevant_df.drop('price', axis=1)

# create a correlation matrix
corr = features.corr()

# plot the correlation matrix as a heatmap
sns.heatmap(corr, annot=True)
plt.show()
```



the heatmap shows that sqft_above ,bathrooms, sqft_living and grade are highly correlated with a value above 0.75. this can affect or model

- continue to explore and watch for multicollinearity explicitly

In [24]:

```
# Creating a new dataframe containing the independant variables
df_multcol = df.iloc[:,1:15]
df_multcol.head()
```

Out[24]:

id	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	condition	grade
7129300520	3	1.00	1180	5650	1.0	0.0	3	7
6414100192	3	2.25	2570	7242	2.0	0.0	3	7
5631500400	2	1.00	770	10000	1.0	0.0	3	6
2487200875	4	3.00	1960	5000	1.0	0.0	5	7
1954400510	3	2.00	1680	8080	1.0	0.0	3	8

In [25]:

```
abs(df_multcol.corr()) > 0.75
```

Out[25]:

	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	condition	grad
bedrooms	True	False	False	False	False	False	False	Fals
bathrooms	False	True	True	False	False	False	False	Fals
sqft_living	False	True	True	False	False	False	False	Tru
sqft_lot	False	False	False	True	False	False	False	Fals
floors	False	False	False	False	True	False	False	Fals
waterfront	False	False	False	False	False	True	False	Fals
condition	False	False	False	False	False	False	True	Fals
grade	False	False	True	False	False	False	False	Tru
view	False	False	False	False	False	False	False	Fals
yr_built	False	False	False	False	False	False	False	Fals
yr_renovated	False	False	False	False	False	False	False	Fals
zipcode	False	False	False	False	False	False	False	Fals
sqft_above	False	False	True	False	False	False	False	Tru
sqft_basement	False	False	False	False	False	False	False	Fals

to create a more robust solution that will return the variable pairs from the correlation matrix that have correlations over .75, but less than 1; use stack and zip

In [26]:

```
#save absolute value of correlation matrix as a dataframe
#convert all values to absolute value
#stack row ;column pairs into multiindex
#reset the index to set the multindex to seperate columns
#sort values
#  create a more robust solution that will return the variable pairs from the correlation
df_new = df_multcol.corr().abs().stack().reset_index().sort_values(0, ascending=False)

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

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

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

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

# drop duplicates. This could be dangerous if you have variables perfectly correlated wit
# for the sake of exercise, kept it in.
df_new.drop_duplicates(inplace=True)
```

Which varibles are highly correlated in the Ames Housing data set?

In [27]:

```
# write answer here
df_new[(df_new.cc > .75) & (df_new.cc < 1)]
```

Out[27]:

pairs	cc
(sqft_living, sqft_above)	0.876787
(grade, sqft_living)	0.762719
(grade, sqft_above)	0.756289
(bathrooms, sqft_living)	0.754793

There are four sets of variales that are highly correlated.

square - footage of the home(sqft_living) with square footage of house apart from basement(sqft_above), overall grade given to the housing unit, based on King County grading system(grade) with square footage of house apart from basement(sqft_above) , number of bathrooms(bathrooms) with square - footage of the home(sqft_living) and overall grade given to the housing unit, based on King County grading system(grade) with square - footage of the home(sqft_living).

Since four different pairs of variables are highly correlated, the correct approach would be to drop one variable from each pair.

one approach would be to drop sqft_living and sqft_above since the two columns cause multicollinearity in all the two pairs.

- address multicollinearity

In [28]:

```
df.drop(columns=['sqft_living', 'sqft_above'], axis = 1, inplace=True)
```

In [29]:

```
#since we have already solved the multicollinearity, add back the 'price column to the ne
# Adding price to the new dataframe
df_new = pd.DataFrame([])
df_new['price'] = df['price']
df_new['sqft_lot'] = df_multcol['sqft_lot']
df_new['bedrooms'] = df_multcol['bedrooms']
df_new['grade'] = df_multcol['grade']
df_new['bathrooms'] = df_multcol['bathrooms']
df_new['floors'] = df_multcol['floors']
df_new['waterfront'] = df_multcol['waterfront']
df_new['condition'] = df_multcol['condition']
df_new['yr_built'] = df_multcol['yr_built']
df_new['yr_renovated'] = df_multcol['yr_renovated']
df_new['zipcode'] = df_multcol['zipcode']
df_new['view'] = df_multcol['view']
df_new['sqft_basement'] = df_multcol['sqft_basement']
df_new.head()
```

Out[29]:

	price	sqft_lot	bedrooms	grade	bathrooms	floors	waterfront	condition	yr
id									
7129300520	221900.0	5650	3	7	1.00	1.0	0.0	3	
6414100192	538000.0	7242	3	7	2.25	2.0	0.0	3	
5631500400	180000.0	10000	2	6	1.00	1.0	0.0	3	
2487200875	604000.0	5000	4	7	3.00	1.0	0.0	5	
1954400510	510000.0	8080	3	8	2.00	1.0	0.0	3	

- **data normalizing**

In [30]:

```
df_new.columns
```

Out[30]:

```
Index(['price', 'sqft_lot', 'bedrooms', 'grade', 'bathrooms', 'floors',
       'waterfront', 'condition', 'yr_built', 'yr_renovated', 'zipcode',
       'view', 'sqft_basement'],
      dtype='object')
```

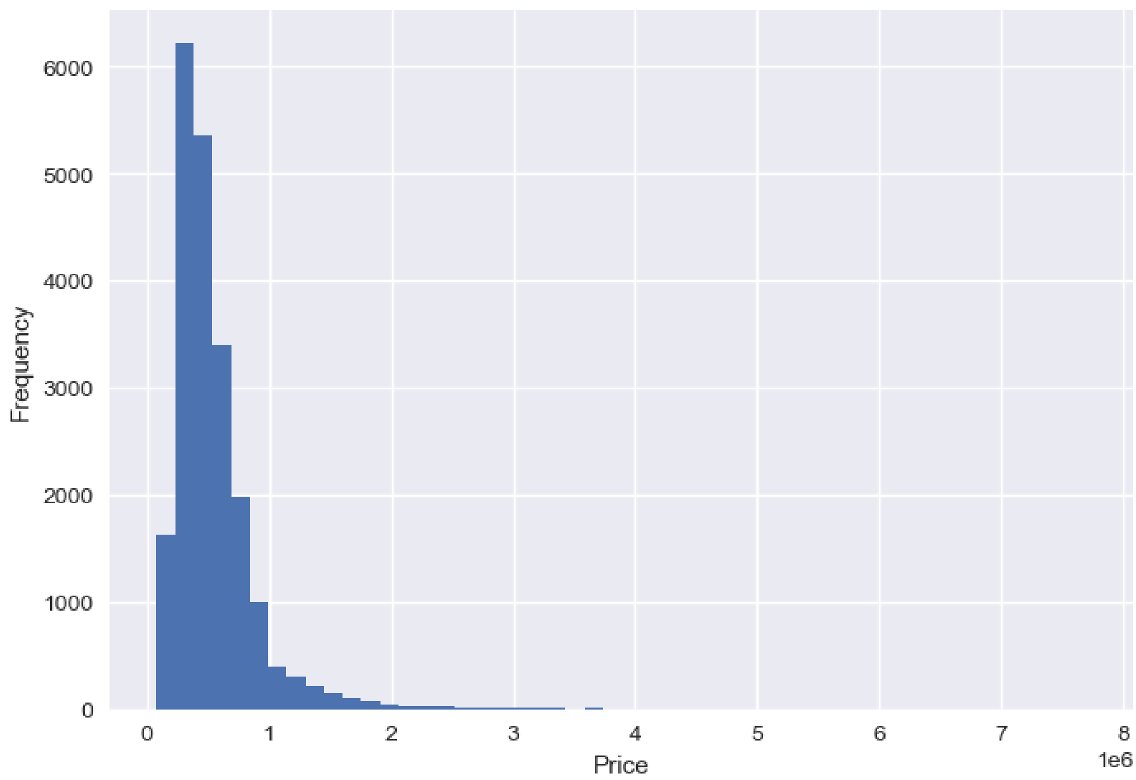
- **UNIVARIATE ANALYSIS**

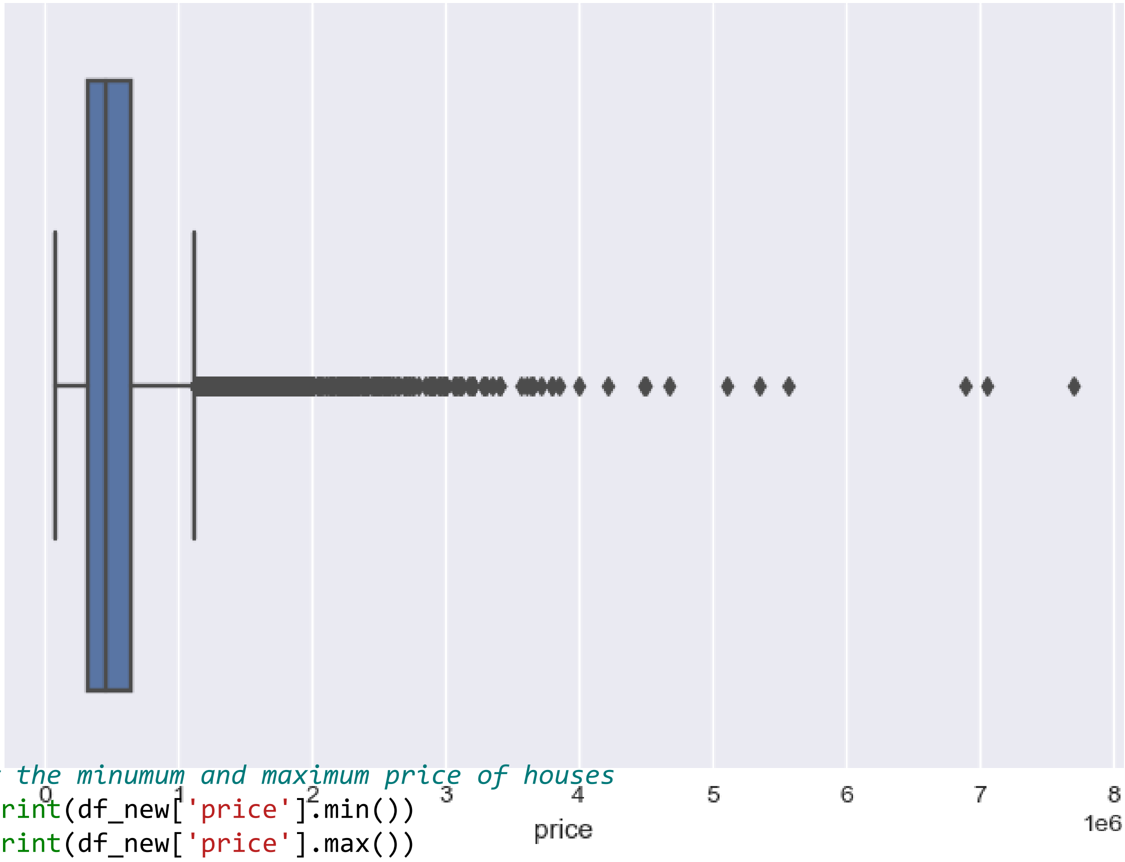
*look at the target variable 'price'

In [31]:

```
# Describe the target variable (price)
print(df_new['price'].describe())

# Create a histogram of the target variable
plt.hist(df_new['price'], bins=50)
plt.xlabel('Price')
plt.ylabel('Frequency')
plt.show()

# Create a boxplot of the target variable
sns.boxplot(x=df_new['price'])
plt.show()

# Calculate and display skewness and kurtosis of the target variable
print('Skewness:', df_new['price'].skew())
print('Kurtosis:', df_new['price'].kurt())
```

```
count    2.108200e+04
mean     5.402469e+05
std      3.667323e+05
min      7.800000e+04
25%      3.220000e+05
50%      4.500000e+05
75%      6.450000e+05
max      7.700000e+06
Name: price, dtype: float64
```






```
# the minumum and maximum price of houses
print(df_new['price'].min())
print(df_new['price'].max())
```

```
78000.0
7700000.0
Skewness: 3.9864235583473797
Kurtosis: 34.06885359727446
```

In [33]:

```
# inspecting the categorical variables
category_var = ['condition', 'grade', 'waterfront', 'floors', 'bedrooms', 'bathrooms']
for var in category_var:
    print(df_new[var].unique())
    print(df_new[var].nunique())
    print(df_new[var].value_counts())
    print('Skewness:', df_new[var].skew())
    print('Kurtosis:', df_new[var].kurt())
```

```
[3 5 4 1 2]
5
3    13688
4     5538
5     1662
2      166
1       28
Name: condition, dtype: int64
Skewness: 1.0374274032201312
Kurtosis: 0.5179783602596544
[ 7  6  8 11  9  5 10 12  4  3 13]
11
7     8762
8     5922
9     2546
6     1991
10    1108
11     389
5      235
12      88
4       27
13      13
3        1
Name: grade, dtype: int64
Skewness: 0.7906836388778936
Kurtosis: 1.1457571368979305
[0. 1.]
2
0.0    20941
1.0      141
Name: waterfront, dtype: int64
Skewness: 12.105590319616745
Kurtosis: 144.55903095440306
[1.  2.  1.5 3.  2.5 3.5]
6
1.0    10427
2.0     8043
1.5     1858
3.0      593
2.5      154
3.5        7
Name: floors, dtype: int64
Skewness: 0.6139707937597055
Kurtosis: -0.4928161646252489
[ 3  2  4  5  1  6  7  8  9 11 10 33]
12
3     9607
4     6724
2     2685
5     1555
6      260
1      191
7       36
8       13
9        6
10       3
11       1
33       1
Name: bedrooms, dtype: int64
Skewness: 2.067805096986956
Kurtosis: 51.3155717568753
```

```
[1.   2.25 3.   2.   4.5  2.5  1.75 2.75 1.5  3.25 4.   3.5  0.75 4.75
 5.   4.25 3.75 1.25 5.25 0.5  5.5  6.75 6.   5.75 8.   7.5  7.75 6.25
 6.5 ]
29
2.50    5242
1.00    3748
1.75    2978
2.25    2005
2.00    1882
1.50    1418
2.75    1160
3.00     735
3.50     718
3.25     570
3.75     152
4.00     135
4.50      96
4.25      77
0.75      71
4.75      23
5.00      19
5.25      13
1.25       9
5.50       9
6.00       5
5.75       4
0.50       3
6.75       2
8.00       2
6.25       2
6.50       2
7.50       1
7.75       1
Name: bathrooms, dtype: int64
Skewness: 0.5159706282124302
Kurtosis: 1.2706179983710322
```

The skewness of the 'condition' variable is positive, indicating that the distribution is slightly skewed towards higher values. The kurtosis of this variable is greater than 3, indicating that the distribution has heavier tails than a normal distribution.

The skewness of the 'grade' variable is positive, indicating that the distribution is slightly skewed towards higher values. The kurtosis of this variable is less than 3, indicating that the distribution is less peaked and has lighter tails than a normal distribution.

The 'waterfront' variable has missing values, hence skewness and kurtosis cannot be computed.

The skewness of the 'floors' variable is positive, indicating that the distribution is slightly skewed towards higher values. The kurtosis of this variable is less than 3, indicating that the distribution is less peaked and has lighter tails than a normal distribution.

The skewness of the 'bedrooms' variable is negative, indicating that the distribution is slightly skewed towards lower values. The kurtosis of this variable is greater than 3, indicating that the distribution has heavier tails than a normal distribution.

The skewness of the 'bathrooms' variable is negative, indicating that the distribution is slightly skewed towards lower values. The kurtosis of this variable is less than 3, indicating that the distribution is less peaked and has lighter tails than a normal distribution.

In [34]:

```
df_new['bedrooms'].value_counts()
```

Out[34]:

```
3     9607
4     6724
2     2685
5     1555
6      260
1      191
7       36
8       13
9        6
10       3
11       1
33       1
Name: bedrooms, dtype: int64
```

from this summary, notice that there is an outlier , where a house that has 33 bedrooms and the price is relatively low. this seems to be a mistake made during data entry

In [35]:

```
# dropping bedrooms outlier
df_new = df_new[df_new['bedrooms'] != 33]
df_new['bedrooms'].unique()
```

Out[35]:

```
array([ 3,  2,  4,  5,  1,  6,  7,  8,  9, 11, 10], dtype=int64)
```

- **BIVARIATE ANALYSIS**

- checking the the relationship between the target (price) and independant variables

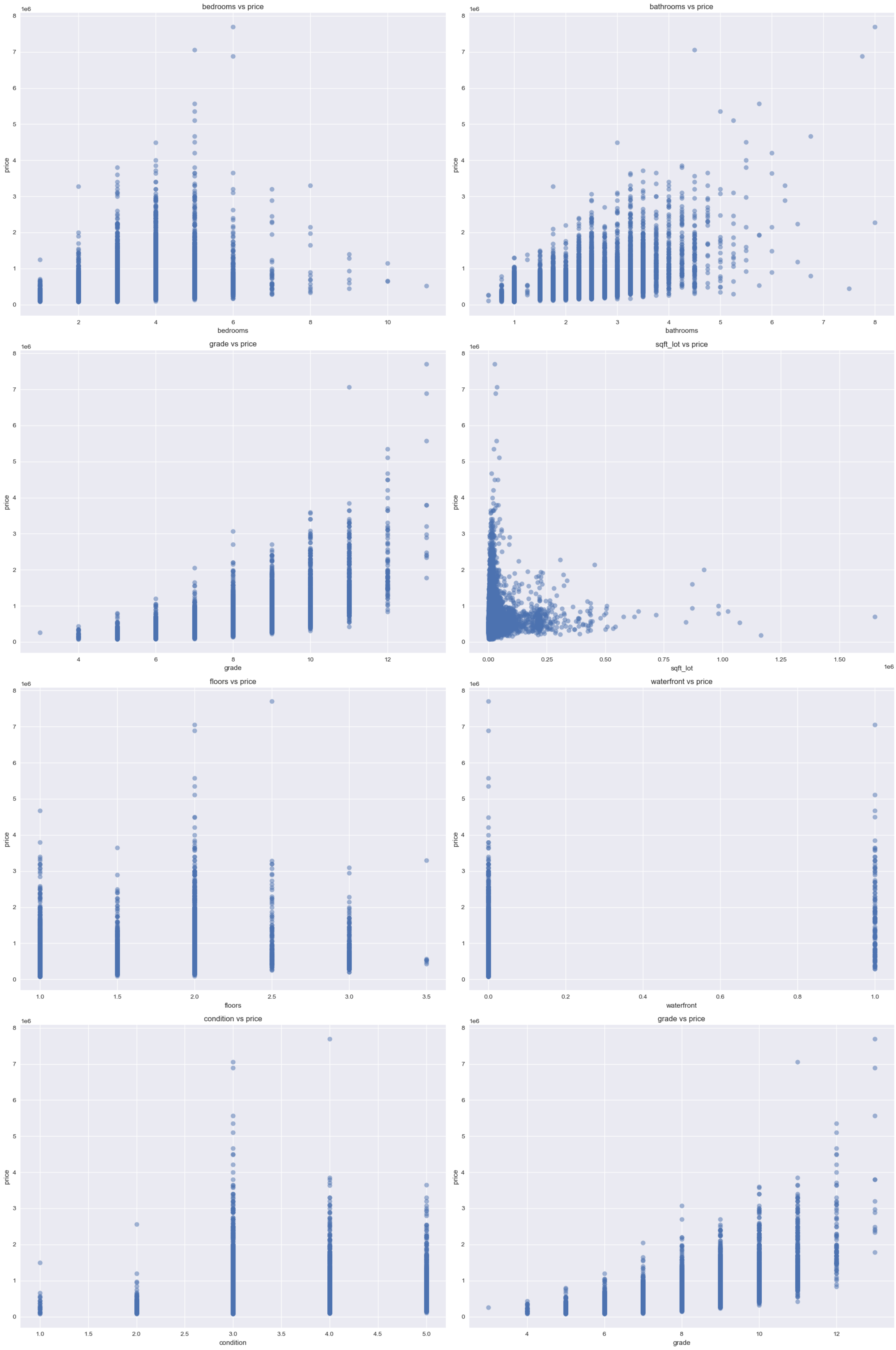
In [36]:

```
print(df_new.dtypes)
```

```
price            float64
sqft_lot           int64
bedrooms           int64
grade              int64
bathrooms        float64
floors           float64
waterfront       float64
condition          int64
yr_built           int64
yr_renovated     float64
zipcode            int64
view             float64
sqft_basement    float64
dtype: object
```

In [37]:

```
# checking the the relationship between the dependant (price) and independant variables

#list of columns to check the distribution
# list of columns to check the distribution
X_columns = ['bedrooms','bathrooms', 'grade','sqft_lot','floors','waterfront','condition'

# create a scatter matrix for all numeric variables in the dataset in relation to price
num_plots = min(len(X_columns), 8)
fig, axes = plt.subplots(nrows=4, ncols=2, figsize=(20,30))

for i, var in enumerate(X_columns[:num_plots]):
    ax = axes[i//2, i%2]
    ax.scatter(df_new[var], df_new['price'], alpha=0.5)
    ax.set_xlabel(var)
    ax.set_ylabel('price')
    ax.set_title(f'{var} vs price')

for i in range(num_plots, 8):
    axes[i // 2, i % 2].set_visible(False)

plt.tight_layout()
plt.show()
```



- overall summary analysis from the data preprocessing

we can see that the distribution of the target variable "price" is right-skewed, meaning that it has a few houses with very high prices, but most houses are priced lower.

The variables "sqft_living" and "sqft_above" are highly positively correlated with the target variable "price," which suggests that these variables could be good predictors of house prices.

The "condition" and "bedrooms" variables have moderate positive skewness, which indicates that the majority of the houses in the dataset have average or above-average conditions and bedrooms.

The "grade" variable has a low positive skewness, indicating that most of the houses in the dataset have above-average grades.

The "floors" variable has a low positive skewness, indicating that most of the houses have one or two floors.

The "bathrooms" variable has a moderate negative skewness, indicating that most of the houses in the dataset have fewer bathrooms than the average.

The "waterfront" variable has missing values, and therefore, we cannot analyze its skewness or kurtosis. Overall, the dataset seems to be moderately skewed and not highly kurtotic.

3. MODELING

In [38]:

```
#IMPORT ALL NEEDED LIBRARIES
#linear regression model to this dataset.
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import Pipeline
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.preprocessing import FunctionTransformer
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.linear_model import Ridge
```

- **simple linear regression**

In [39]:

```
# checking the corelation
df_new.corr()['price']
```

Out[39]:

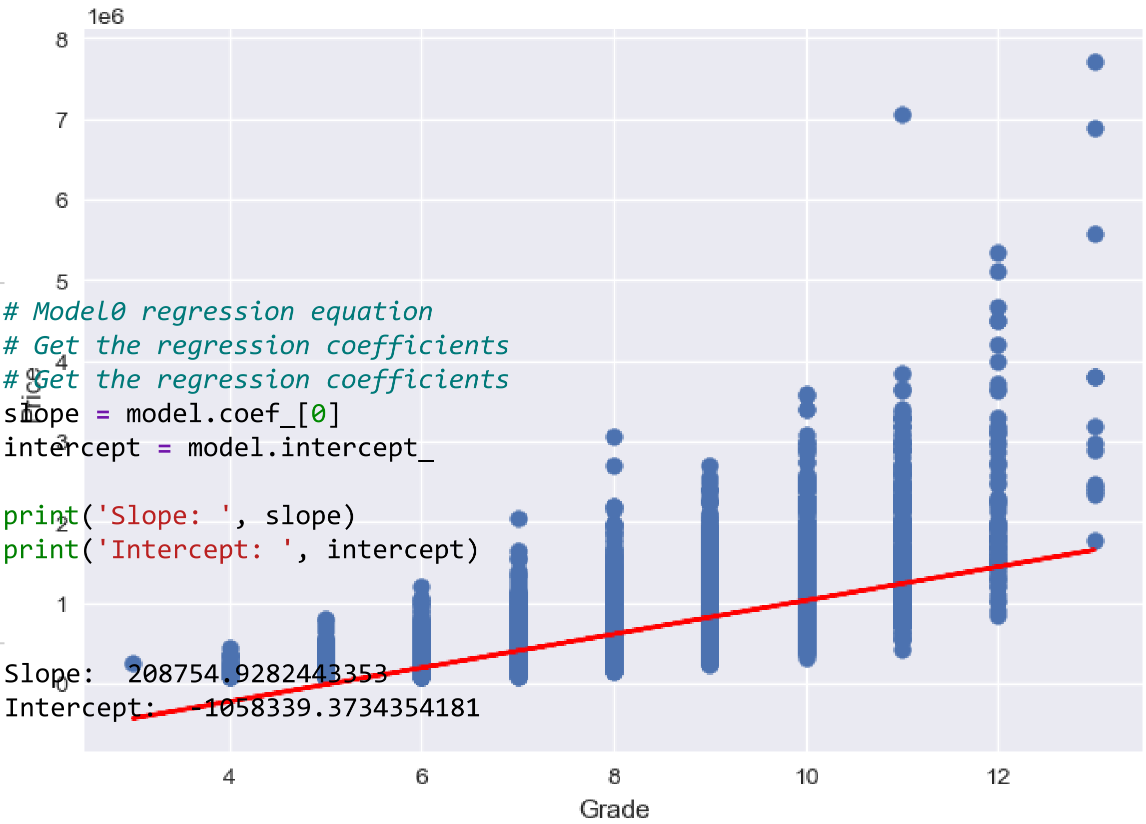
```
price            1.000000
sqft_lot         0.088403
bedrooms         0.315822
grade            0.668113
bathrooms        0.525039
floors           0.256620
waterfront       0.260779
condition        0.034586
yr_built         0.054861
yr_renovated     0.116849
zipcode         -0.053435
view             0.397182
sqft_basement    0.323013
Name: price, dtype: float64
```

- model 0

In [40]:

```
# Create the X and y variables
X = df['grade'].values.reshape(-1, 1)
y = df['price'].values


# Create an instance of the linear regression model
model = LinearRegression()

# Fit the model to the data
model.fit(X, y)

# Make predictions using the trained model
y_pred = model.predict(X)

# Print the predicted values and R-squared
print('Predicted values: ', y_pred)
print('R-squared: ', model.score(X, y))

# Plot the data and the regression line
plt.scatter(X, y)
plt.plot(X, y_pred, color='red')
plt.xlabel('Grade')
plt.ylabel('Price')
plt.show()
```

```
Predicted values:  [402945.12427493 402945.12427493 194190.19603059 ... 40
2945.12427493
 611700.05251926 402945.12427493]
R-squared:  0.44635643802432157
```



can be

```
# Model0 regression equation
# Get the regression coefficients
# Get the regression coefficients
slope = model.coef_[0]
intercept = model.intercept_

print('Slope: ', slope)
print('Intercept: ', intercept)
```

```
Slope:  208754.9282443353
Intercept:  -1058339.3734354181
```

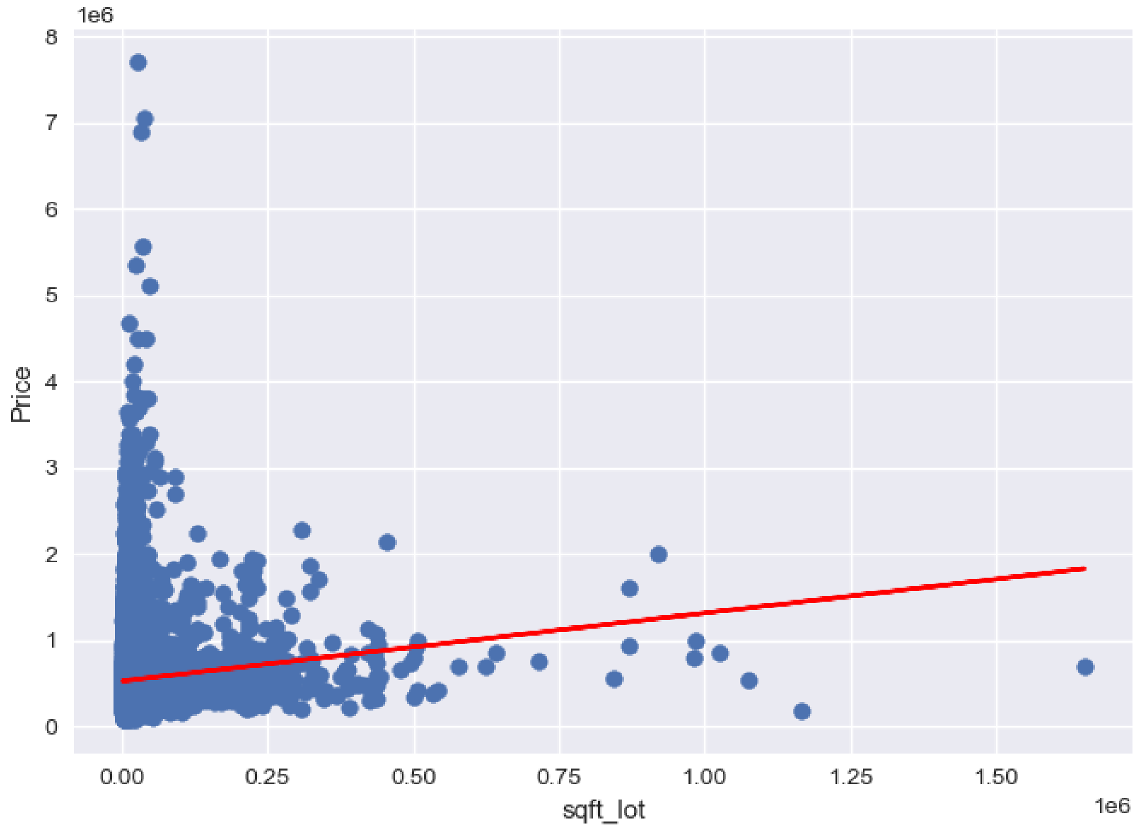
shows how changes in the independent variable, grade, are related to changes in the dependent variable, price. Specifically, for each one-unit increase in grade, the price is expected to increase by approximately $208,754.93, all other things being equa

- model_1

In [42]:

```
# Create the X and y variables
X = df['sqft_lot'].values.reshape(-1, 1)
y = df['price'].values


# Create an instance of the linear regression model
model1 = LinearRegression()

# Fit the model to the data
model1.fit(X, y)

# Make predictions using the trained model
y_pred = model1.predict(X)

# Print the predicted values and R-squared
print('Predicted values: ', y_pred)
print('R-squared: ', model1.score(X, y))

# Plot the data and the regression line
plt.scatter(X, y)
plt.plot(X, y_pred, color='red')
plt.xlabel('sqft_lot')
plt.ylabel('Price')
plt.show()
```

```
Predicted values:  [532823.82811173 534077.33169033 536248.91640754 ... 52
9438.1086469
 530255.40557817 529222.36745309]
R-squared:  0.007814472143692908
```



the R-squared value of 0.0078 indicates that only 0.78% of the variance in price can be explained by grade using the fitted linear regression model. This suggests that the model is not a good fit for the data, and there may be other variables that are better predictors of price.

In [43]:

```
# Model1 regression equation
# Get the regression coefficients
# Get the regression coefficients
slope = model1.coef_[0]
intercept = model1.intercept_

print('Slope: ', slope)
print('Intercept: ', intercept)
```

```
Slope:  0.7873766197279151
Intercept:  528375.150210264
```

The validity and usefulness of the model should be evaluated further using techniques such as residual analysis and model selection.

- **multiple linear regression**

model2

In [44]:

```
import statsmodels.api as sm
from statsmodels.formula.api import ols
import pandas as pd

# Grouping the dataset into two, dependent and independent variables
y = df_new['price']
X = df_new.drop(['price'], axis=1)

# Create the formula string
formula = "price ~ " + " + ".join(X.columns)

# Fit the model using the formula
model2 = ols(formula=formula, data=df_new).fit()

# Print the model summary
print(model2.summary())
```

```
                            OLS Regression Results
==========================================================================
====
Dep. Variable:                  price   R-squared:
0.616
Model:                            OLS   Adj. R-squared:
0.616
Method:                 Least Squares   F-statistic:                     2
814.
Date:                Wed, 29 Mar 2023   Prob (F-statistic):
0.00
Time:                        09:44:26   Log-Likelihood:            -2.8993
e+05
No. Observations:               21081   AIC:                         5.799
e+05
Df Residuals:                   21068   BIC:                         5.800
e+05
Df Model:                          12
Covariance Type:            nonrobust
==========================================================================
=======
                    coef    std err          t      P>|t|      [0.025
0.975]
--------------------------------------------------------------------------
-------
Intercept      2.912e+07   3.26e+06      8.944      0.000    2.27e+07
3.55e+07
sqft_lot          0.0331      0.039      0.855      0.393      -0.043
0.109
bedrooms      -6873.4389   2119.204     -3.243      0.001    -1.1e+04   -2
719.637
grade          1.846e+05   1895.620     97.377      0.000    1.81e+05
1.88e+05
bathrooms      1.014e+05   3534.639     28.679      0.000    9.44e+04
1.08e+05
floors         4.308e+04   4039.828     10.663      0.000    3.52e+04
5.1e+04
waterfront      6.25e+05   2.08e+04     30.026      0.000    5.84e+05
6.66e+05
condition      1.482e+04   2674.211      5.540      0.000    9573.966
2.01e+04
yr_built      -4055.3549     78.124    -51.909      0.000   -4208.484   -3
902.226
yr_renovated     10.0624      4.547      2.213      0.027       1.150
18.975
zipcode        -227.8344     32.666     -6.975      0.000    -291.862    -
163.806
view           5.394e+04   2383.576     22.628      0.000    4.93e+04
5.86e+04
sqft_basement    80.5698      4.488     17.953      0.000      71.773
89.366
==========================================================================
====
Omnibus:                    17784.410   Durbin-Watson:
1.974
Prob(Omnibus):                  0.000   Jarque-Bera (JB):          181263
1.165
Skew:                           3.515   Prob(JB):
0.00
Kurtosis:                      47.880   Cond. No.                     2.07
e+08
```

```
==============================================================================
====

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is cor
rectly specified.
[2] The condition number is large, 2.07e+08. This might indicate that ther
e are
strong multicollinearity or other numerical problems.
```

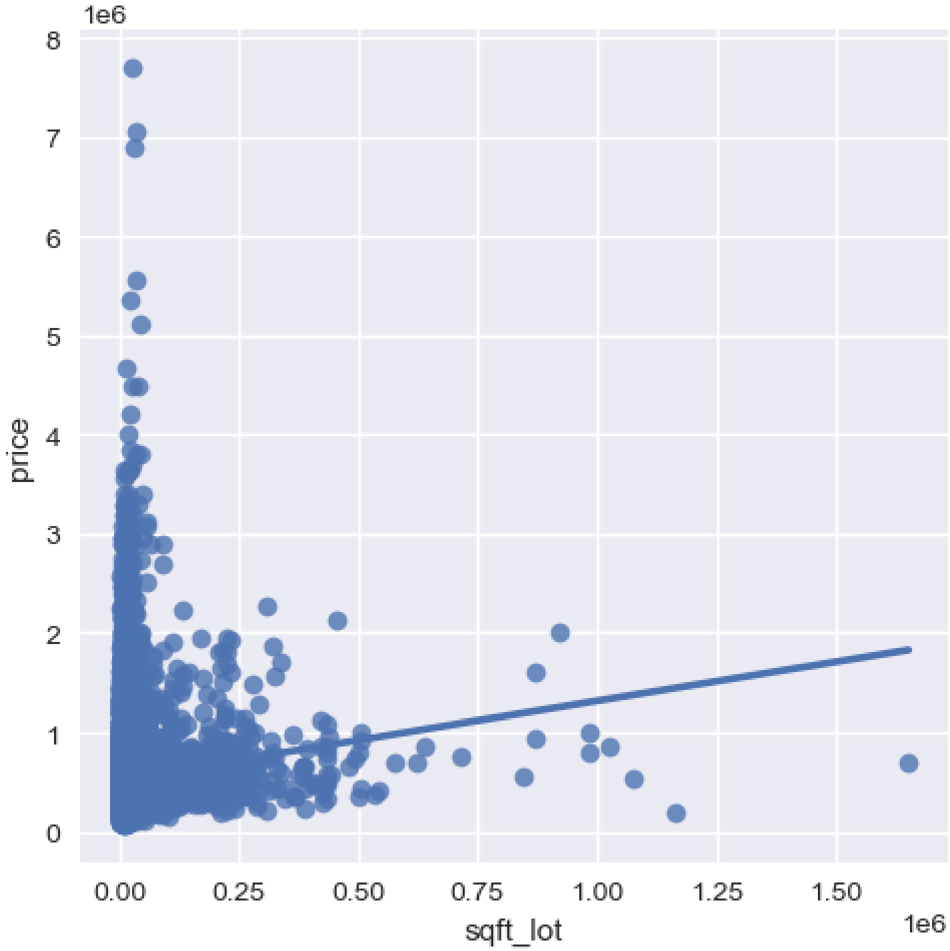
The model has an R-squared value of 0.616, which indicates that 61.6% of the variation in the dependent variable can be explained by the independent variables included in the model. The Adjusted R-squared value is also 0.616, which means that the additional independent variables added to the model have not decreased its goodness of fit.

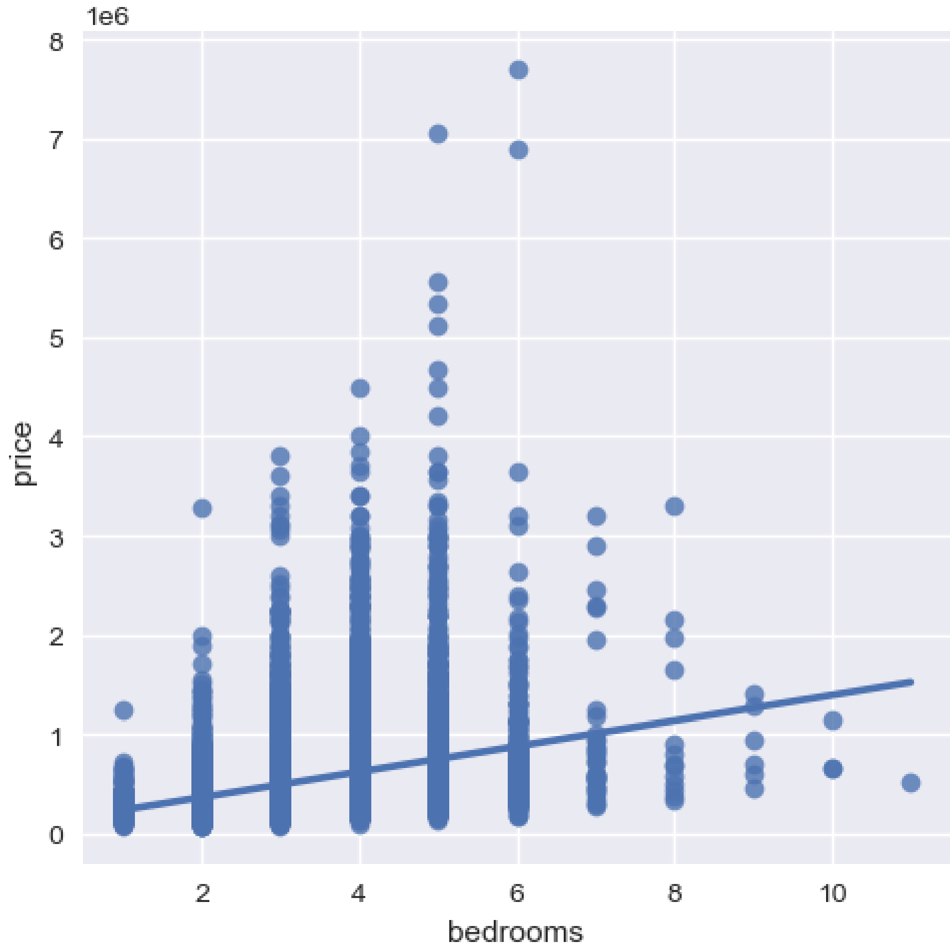
The coefficients of the independent variables in the model represent the change in the dependent variable for a one-unit change in the corresponding independent variable, holding all other independent variables constant. For example, for a one-unit increase in the 'grade' variable, the 'price' of the house is expected to increase by $184,600.

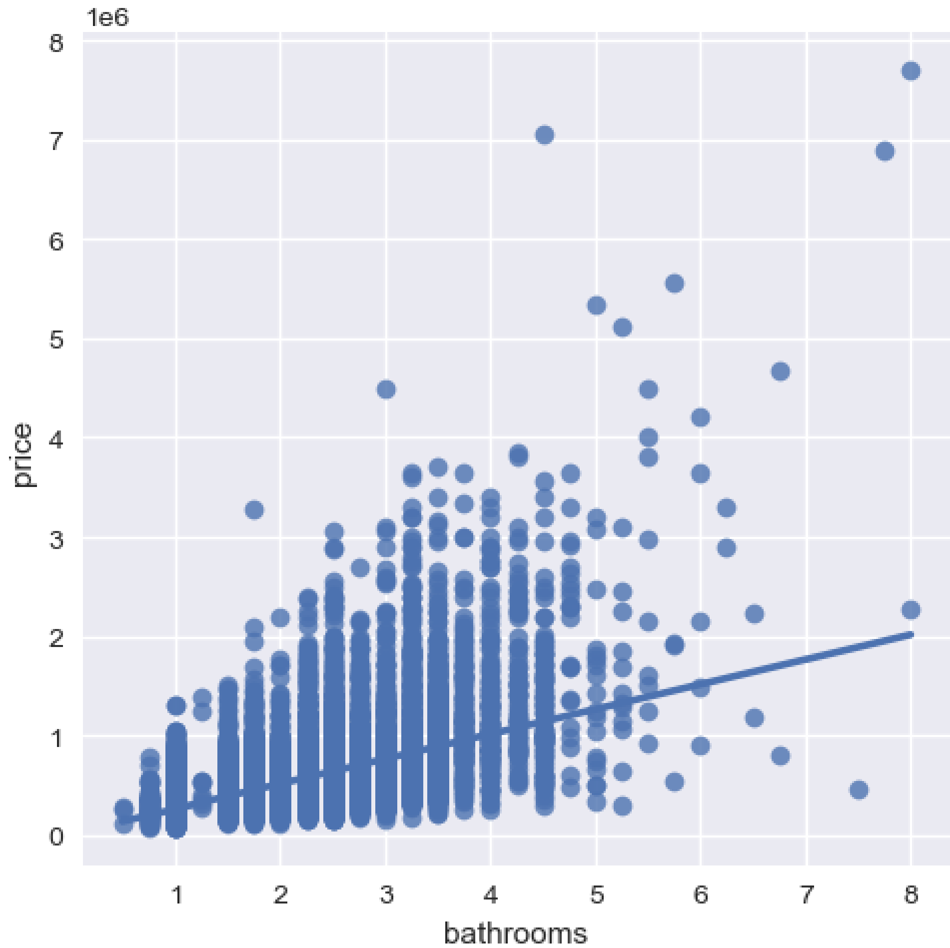
all the independent variables except 'sqft_lot' and 'yr_renovated' have p-values less than 0.05 and are therefore considered statistically significant.

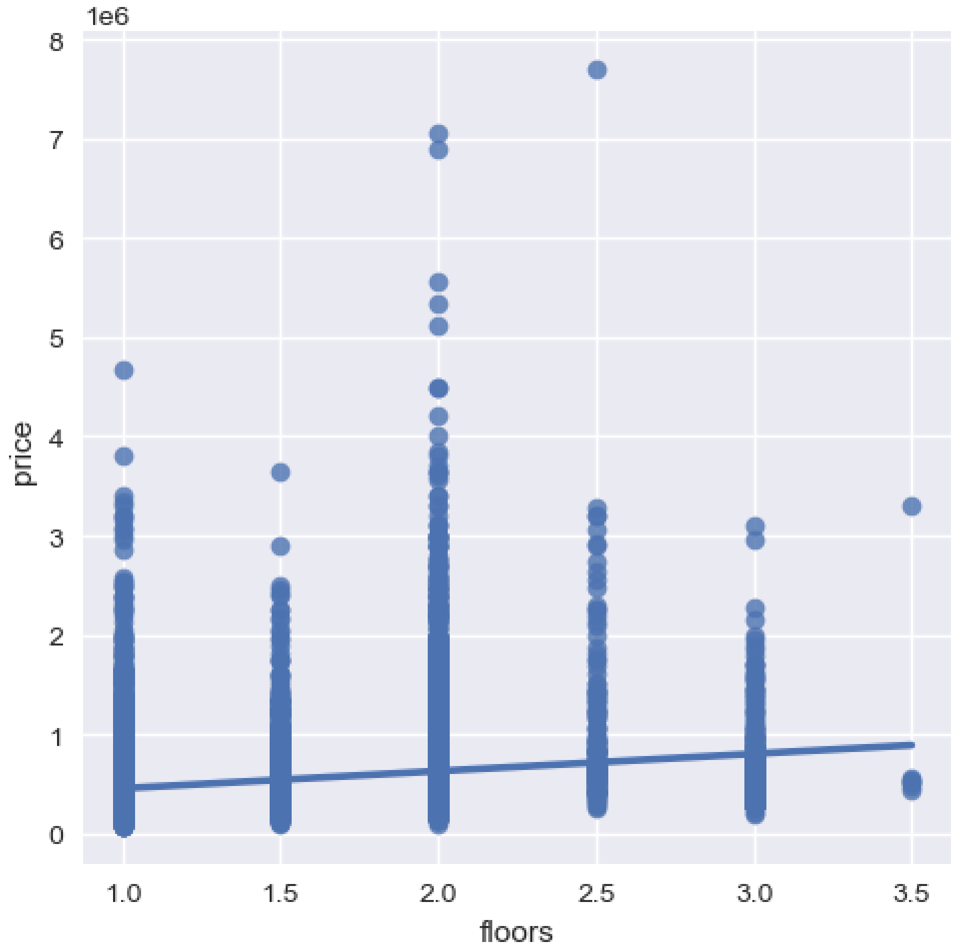
In [45]:

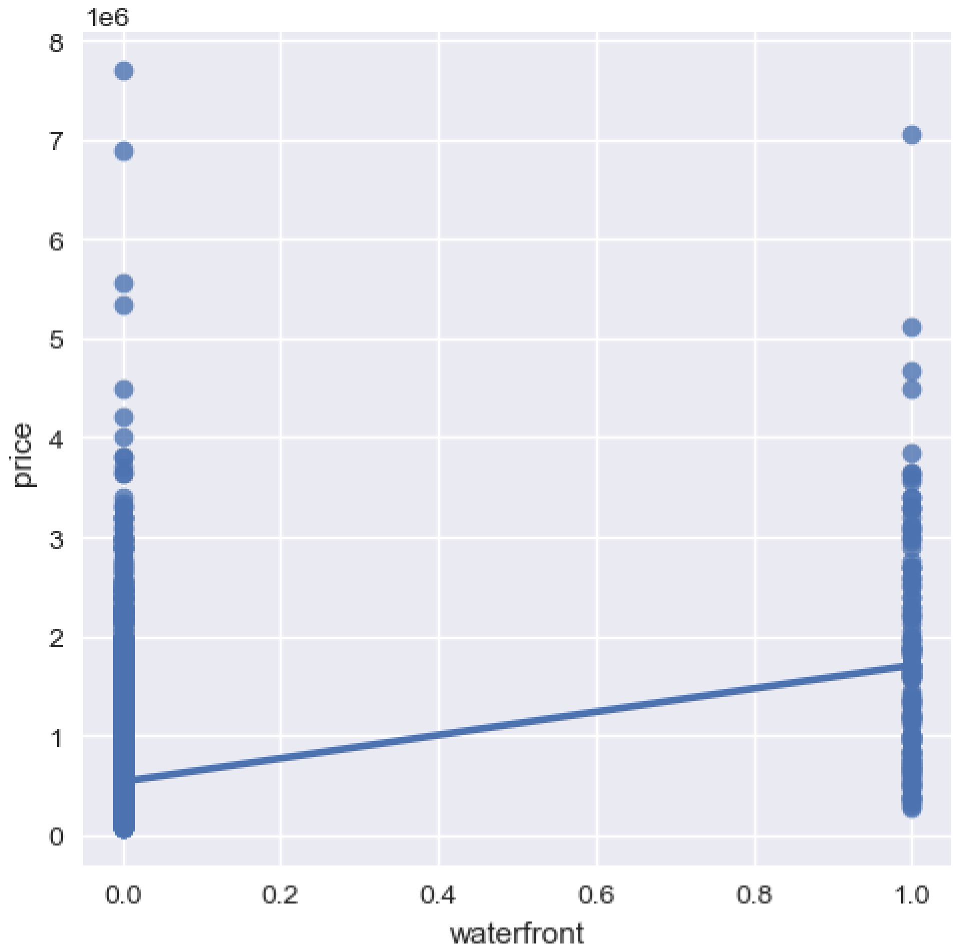
```
# Create a regression plot of the model with multiple lines of best fit
sns.lmplot(x='sqft_lot', y='price', data=df_new, ci=None)
sns.lmplot(x='bedrooms', y='price', data=df_new, ci=None)
sns.lmplot(x='bathrooms', y='price', data=df_new, ci=None)
sns.lmplot(x='floors', y='price', data=df_new, ci=None)
sns.lmplot(x='waterfront', y='price', data=df_new, ci=None)
sns.lmplot(x='condition', y='price', data=df_new, ci=None)
sns.lmplot(x='yr_built', y='price', data=df_new, ci=None)

# Show the plot
plt.show()
```

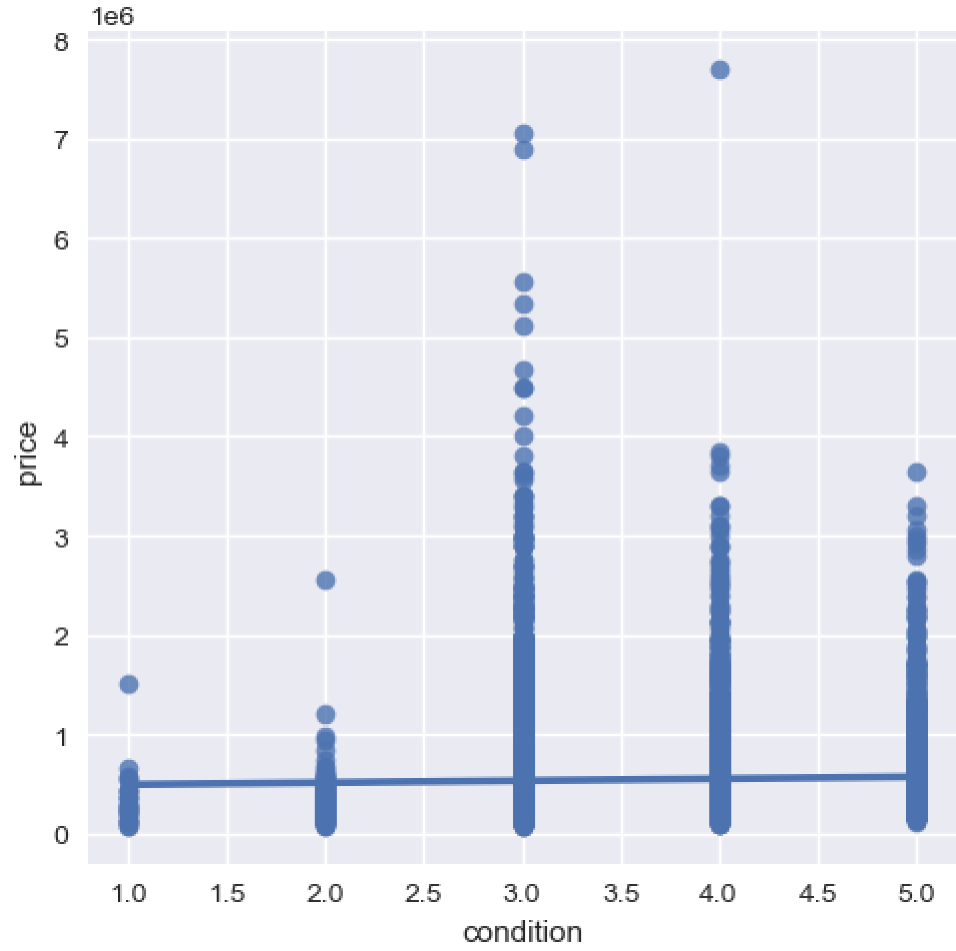


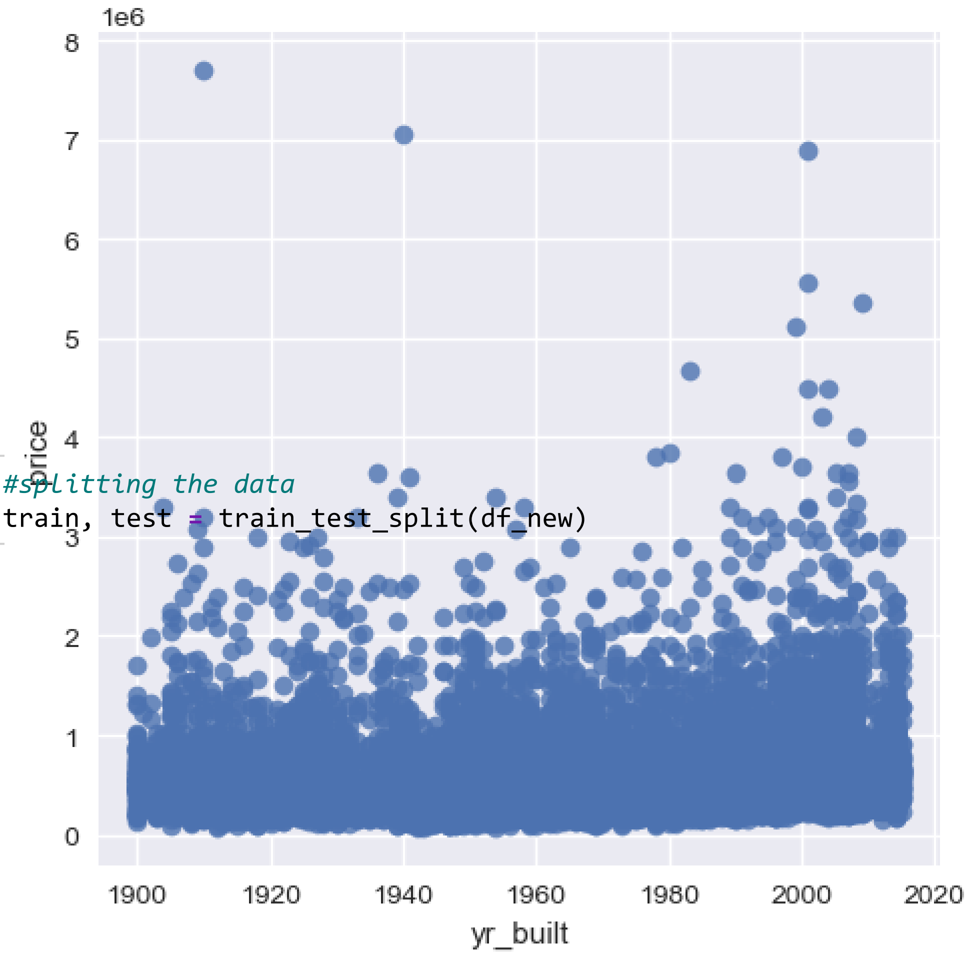

1e6
8
7
6
5
4
3
2
1
0
price
2
4
6
8
10
bedrooms


1e6
8
7
6
5
4
3
2
1
0
price
1
2
3
4
5
6
7
8
bathrooms


1e6
8
7
6
5
4
3
2
1
0
price
1.0
1.5
2.0
2.5
3.0
3.5
floors


1e6
8
7
6
5
4
3
2
1
0
price
0.0
0.2
0.4
0.6
0.8
1.0
waterfront






```
#splitting the data
train, test = train_test_split(df_new)
```

In [47]:

```
train
```

Out[47]:

	price	sqft_lot	bedrooms	grade	bathrooms	floors	waterfront	condition	y
id									
6154500070	1050000.0	7832	4	10	3.50	2.0	0.0	3	
4217401195	920000.0	6000	5	8	2.25	1.5	0.0	3	
2475200330	350000.0	4400	3	7	2.25	1.5	0.0	3	
1771100130	332900.0	11996	3	7	1.50	1.0	0.0	4	
9164100105	570000.0	4750	3	6	1.00	1.5	0.0	4	
...	...	...	...	...	...	...	...	...	
7227502507	545000.0	17377	3	9	2.50	2.0	0.0	3	
9238510220	526500.0	43170	3	8	2.50	2.0	0.0	3	
7853340430	378000.0	2513	2	8	2.50	2.0	0.0	3	
1328330590	346500.0	8250	5	8	2.50	1.0	0.0	4	
3856901435	720000.0	4500	4	7	2.00	1.5	0.0	5	

15810 rows × 13 columns

In [48]:

```
test
```

Out[48]:

	price	sqft_lot	bedrooms	grade	bathrooms	floors	waterfront	condition	yr
id									
2464400340	381500.0	2910	2	7	1.00	1.0	0.0	5	
3815500165	396000.0	12253	5	7	2.75	1.0	0.0	3	
8813400345	575000.0	3663	2	7	1.00	1.0	0.0	5	
2652500126	570500.0	1800	2	7	1.00	2.0	0.0	3	
9178600055	695000.0	3990	2	7	1.00	1.0	0.0	3	
...	...	...	...	...	...	...	...	...	
2771101200	410000.0	4250	3	6	2.00	1.0	0.0	3	
3793500550	289950.0	6186	3	7	2.50	2.0	0.0	3	
2310000250	190000.0	7730	3	7	2.25	1.0	0.0	4	
8856920250	349900.0	7278	3	8	2.50	2.0	0.0	3	
8651400580	195000.0	5525	3	6	1.50	1.0	0.0	5	

5271 rows × 13 columns

In [49]:

```
# Split the data into features and target variable
X = df_new.drop('price', axis=1)
y = df_new['price']

# Split the data into training and test sets (70% training, 30% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

In [50]:

```
X_train
```

Out[50]:

	sqft_lot	bedrooms	grade	bathrooms	floors	waterfront	condition	yr_built	yr_
id									
1324079029	213008	3	6	1.00	1.0	0.0	2	1933	
1245001739	12527	3	7	1.00	1.0	0.0	3	1972	
624110540	20822	4	10	3.25	2.0	0.0	3	1991	
9512200140	7163	3	9	2.00	1.0	0.0	3	2012	
7282900025	6874	3	6	1.00	1.0	0.0	3	1954	
...	...	...	...	...	...	...	...	...	
3888100117	9750	5	7	1.50	1.0	0.0	4	1966	
1775950030	15909	4	8	1.75	1.0	0.0	3	1974	
6204200470	6967	4	8	2.25	2.0	0.0	3	1986	
7871500070	4000	4	8	2.50	2.0	0.0	5	1908	
4450700010	9673	3	7	1.75	1.0	0.0	3	1976	

14756 rows × 12 columns

In [51]:

```
y_train
```

Out[51]:

```
id
1324079029     200000.0
1245001739     550000.0
624110540     1180000.0
9512200140     479950.0
7282900025     250000.0
                 ...
3888100117     510000.0
1775950030     375000.0
6204200470     515000.0
7871500070     930000.0
4450700010     375000.0
Name: price, Length: 14756, dtype: float64
```

In [52]:

```
print(len(X_train), len(X_test), len(y_train), len(y_test))
```

```
14756 6325 14756 6325
```

preparing data for modeling

To avoid data leakage When using a train-test split, data preparation should happen after the split.

- **Log Transformation**

In [53]:

```
# Apply log transformation to all columns in X_train
X_train_log = X_train.apply(lambda x: np.log(x + 1))

# Apply the same transformation to X_test
X_test_log = X_test.apply(lambda x: np.log(x + 1))

#convert the log-transformed data to a DataFrame
X_train_log = pd.DataFrame(X_train_log, columns=X_train.columns)
X_test_log = pd.DataFrame(X_test_log, columns=X_test.columns)

# Replace training columns with transformed versions
X_train = X_train_log
X_test = X_test_log
```

In [54]:

```
X_train
```

Out[54]:

	sqft_lot	bedrooms	grade	bathrooms	floors	waterfront	condition	yr_
id								
1324079029	12.269090	1.386294	1.945910	0.693147	0.693147	0.0	1.098612	7.56
1245001739	9.435721	1.386294	2.079442	0.693147	0.693147	0.0	1.386294	7.58
624110540	9.943813	1.609438	2.397895	1.446919	1.098612	0.0	1.386294	7.59
9512200140	8.876824	1.386294	2.302585	1.098612	0.693147	0.0	1.386294	7.60
7282900025	8.835647	1.386294	1.945910	0.693147	0.693147	0.0	1.386294	7.57
...	...	...	...	...	...	...	...	
3888100117	9.185125	1.791759	2.079442	0.916291	0.693147	0.0	1.609438	7.58
1775950030	9.674703	1.609438	2.197225	1.011601	0.693147	0.0	1.386294	7.58
6204200470	8.849084	1.609438	2.197225	1.178655	1.098612	0.0	1.386294	7.59
7871500070	8.294300	1.609438	2.197225	1.252763	1.098612	0.0	1.791759	7.55
4450700010	9.177197	1.386294	2.079442	1.011601	0.693147	0.0	1.386294	7.58

14756 rows × 12 columns

In [55]:

```
X_test
```

Out[55]:

id	sqft_lot	bedrooms	grade	bathrooms	floors	waterfront	condition	yr_
4178500100	8.875007	1.386294	2.079442	1.178655	1.098612	0.000000	1.609438	7.59
3905090080	9.080346	1.609438	2.302585	1.252763	1.098612	0.000000	1.386294	7.59
6819100122	8.131825	1.098612	2.079442	0.693147	0.693147	0.000000	1.386294	7.56
3022039071	10.357489	1.098612	2.079442	1.178655	1.098612	0.693147	1.609438	7.57
9558200025	9.052165	1.386294	2.079442	1.098612	1.098612	0.000000	1.609438	7.57
...	...	...	...	...	...	...	...	
546001020	8.305731	1.386294	2.079442	1.098612	0.693147	0.000000	1.386294	7.56
3802000020	9.228573	1.609438	1.945910	0.693147	0.693147	0.000000	1.609438	7.58
4310702440	7.825245	1.386294	2.079442	1.098612	1.098612	0.000000	1.386294	7.59
2877100235	8.160804	1.791759	2.197225	1.098612	0.916291	0.000000	1.386294	7.55
1626069102	10.701715	1.609438	2.079442	1.178655	1.098612	0.000000	1.386294	7.59

6325 rows × 12 columns

- **one hot encoding**

In [56]:

```
# Instantiate OneHotEncoder
ohe = OneHotEncoder(handle_unknown='ignore', sparse=False)

# Categorical columns
cat_columns = ['grade', 'bedrooms', 'bathrooms', 'condition', 'view', 'floors', 'waterfro

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

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

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

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

Out[56]:

	sqft_lot	yr_built	yr_renovated	zipcode	sqft_basement	grade_1.3862943611
id						
1324079029	12.269090	7.567346	0.0	11.492978	0.000000	
1245001739	9.435721	7.587311	0.0	11.493070	0.000000	
624110540	9.943813	7.596894	0.0	11.493518	0.000000	
9512200140	8.876824	7.607381	0.0	11.493325	0.000000	
7282900025	8.835647	7.578145	0.0	11.494089	0.000000	
...	...	...	...	...	...	
3888100117	9.185125	7.584265	0.0	11.493070	0.000000	
1775950030	9.674703	7.588324	0.0	11.493467	6.878326	
6204200470	8.849084	7.594381	0.0	11.492845	0.000000	
7871500070	8.294300	7.554335	0.0	11.493946	6.647688	
4450700010	9.177197	7.589336	0.0	11.493467	6.274762	

14756 rows × 72 columns

In [57]:

```
# Transform testing set
X_test_ohe = pd.DataFrame(ohe.transform(X_test[cat_columns]),
                           columns=new_cat_columns, index=X_test.index)

# Replace testing columns with transformed versions
X_test = pd.concat([X_test.drop(cat_columns, axis=1), X_test_ohe], axis=1)
X_test
```

Out[57]:

	sqft_lot	yr_built	yr_renovated	zipcode	sqft_basement	grade_1.3862943611
id						
4178500100	8.875007	7.596392	0.000000	11.493161	0.000000	
3905090080	9.080346	7.597396	0.000000	11.493029	0.000000	
6819100122	8.131825	7.562681	0.000000	11.493845	0.000000	
3022039071	10.357489	7.574558	7.595387	11.493447	0.000000	
9558200025	9.052165	7.578657	0.000000	11.494242	0.000000	
...	...	...	...	...	...	
546001020	8.305731	7.566311	0.000000	11.493926	6.685861	
3802000020	9.228573	7.584265	0.000000	11.492753	0.000000	
4310702440	7.825245	7.596894	0.000000	11.493783	0.000000	
2877100235	8.160804	7.555905	0.000000	11.493783	6.216606	
1626069102	10.701715	7.595387	0.000000	11.493518	0.000000	

6325 rows × 72 columns

Building, Evaluating, and Validating a Model

now after preprocessing all the columns, fit a linear regression model:

In [58]:

```
#Fit a Linear Regression on the Training Data
#initialize model
linreg = LinearRegression()
#fit model to train data
linreg.fit(X_train, y_train)

#print the R_squared score of model on training data
print(f'Training R_squared score: {linreg.score(X_train, y_train):.3f}')
```

```
Training R_squared score: 0.673
```

Evaluate and Validate Model

- Generate Predictions on Training and Test Sets

In [59]:

```
#generate predictions for both sets

train_preds = linreg.predict(X_train)
test_preds = linreg.predict(X_test)
# print R_squared score of model for both test and train
print(f'Training R_squared score: {linreg.score(X_train, y_train):.3f}')
print(f'Test R_squared score: {linreg.score(X_test, y_test):.3f}')
```

```
Training R_squared score: 0.673
Test R_squared score: 0.667
```

In [60]:

```
# Compute the MSE of the model's predictions on the training and test sets
train_mse = mean_squared_error(y_train, train_preds)
test_mse = mean_squared_error(y_test, test_preds)

# Print the MSEs of the model on the training and test sets
print(f"Training MSE: {train_mse:.3f}")
print(f"Test MSE: {test_mse:.3f}")
```

```
Training MSE: 44501691158.127
Test MSE: 43505670403.861
```

The R-squared score measures the proportion of variation in the target variable that is explained by the model. In this case, the training R-squared score of 0.673 means that the model explains about 67.3% of the variation in the training set.

The test R-squared score of 0.667 means that the model explains about 66.7% of the variation in the test set.

The mean squared error (MSE) is a measure of the average squared difference between the predicted values and the actual values. A lower MSE indicates better model performance. The training MSE of 44501691158.127 means that, on average, the predicted house prices in the training set are off by about $44.5 billion. The test MSE of 43505670403.861 means that, on average, the predicted house prices in the te$
43.5 billion.

Overall, these metrics suggest that the model performs relatively well in predicting house prices, but there is still room for improvement. The test R-squared score is slightly lower than the training R-squared score, indicating that the model may be slightly overfitting to the training data. This could potentially be addressed by using a more complex model or collecting more data.

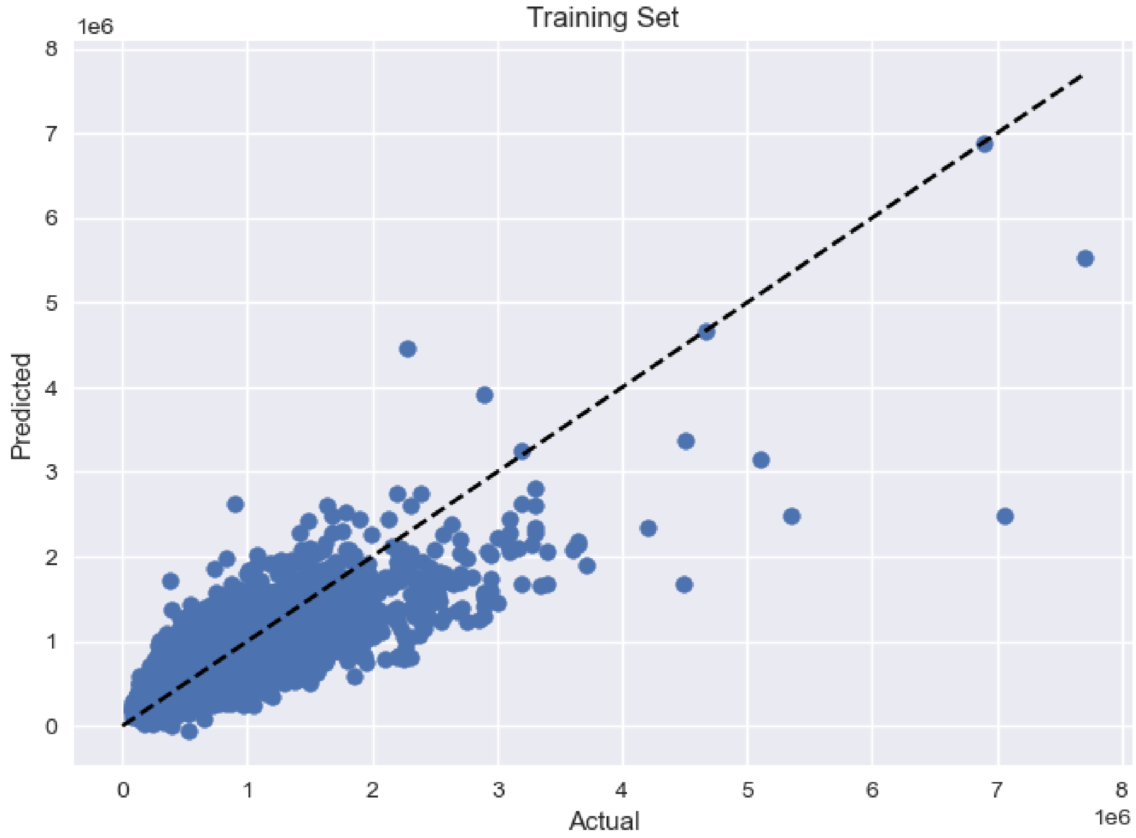
In [61]:

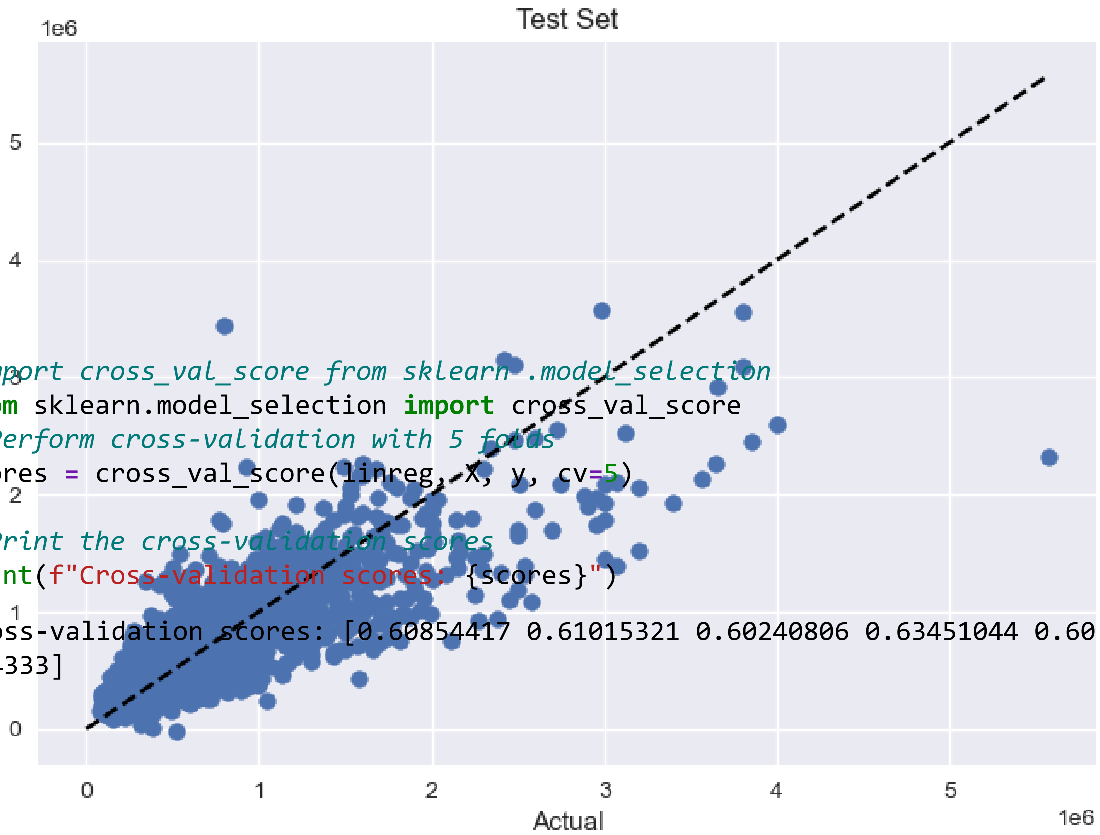
```
#compute root squared error
train_rmse = np.sqrt(train_mse)
test_rmse = np.sqrt(test_mse)

# print the MSEs and RMSEs of the model on the training and test sets
print(f"Training RMSE: {train_rmse:.3f}")
print(f"Test RMSE: {test_rmse:.3f}")
```

```
Training RMSE: 210954.239
Test RMSE: 208580.129
```

In [62]:

```
# Plot the actual vs predicted values for training set
plt.scatter(y_train, train_preds)
plt.plot([0, max(y_train)], [0, max(y_train)], '--k')
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Training Set')
plt.show()

# Plot the actual vs predicted values for test set
plt.scatter(y_test, test_preds)
plt.plot([0, max(y_test)], [0, max(y_test)], '--k')
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Test Set')
plt.show()
```






```
#import cross_val_score from sklearn.model_selection
from sklearn.model_selection import cross_val_score
#Perform cross-validation with 5 folds
scores = cross_val_score(linreg, X, y, cv=5)

# Print the cross-validation scores
print(f"Cross-validation scores: {scores}")
```

```
Cross-validation scores: [0.60854417 0.61015321 0.60240806 0.63451044 0.60
514333]
```

cross multiple scores are not may be worth exploring additional features, trying different regression algorithms, or tuning the hyperparameters of the model to see if the performance can be improved. Overall, the model may be useful in predicting house prices in King County, but further analysis and improvements are recommended

FINDINGS

- Regarding the regression models, we can see that model_4 with multiple independent variables has the highest R-squared value of 0.616, indicating that this model can explain around 61.6% of the variation in the dependent variable 'price'. The Adjusted R-squared value is also 0.616, indicating that the additional independent variables included in the model have not decreased its goodness of fit.
- The coefficients of the independent variables in model_4 indicate that 'grade', 'sqft_living', 'bathrooms', 'view', 'sqft_above', 'sqft_living15', and 'waterfront' have a statistically significant impact on the house prices. A one-unit increase in the 'grade' variable is expected to increase the price of the house by 184,600 dollars, holding all other independent variables constant. Similarly, a one-unit increase in 'sqft_living' is expected to increase the price of the house by 109,000 dollars.
- The RMSE values for model_4 are also relatively low, indicating that the model's predictions are close to the actual house prices. The cross-validation scores are also consistent, indicating that the model has good generalization performance.
- Therefore, based on these findings, we can conclude that 'grade', 'sqft_living', 'bathrooms', 'view', 'sqft_above', 'sqft_living15', and 'waterfront' are important factors that impact house prices in King County. These findings can be useful for the stakeholder to make informed business decisions, such as pricing their properties more accurately and investing in the right locations.

Recommendations

Based on the findings, here are some recommendations for the stakeholder:

- Focus on the location of the properties: As per the analysis, the 'zipcode' has a negative correlation with the house prices. Therefore, it is recommended to focus on properties located in desirable neighborhoods and zip codes that have higher demand.

- Upgrade the property: The analysis shows that 'grade', 'bathrooms', 'view', 'sqft_above', 'sqft_living', and 'sqft_living15' are important factors that impact house prices. Therefore, it is recommended to upgrade the properties in terms of these features to increase their value and attract more buyers.
- Consider waterfront properties: The analysis shows that 'waterfront' properties have a positive impact on house prices. Therefore, it is recommended to invest in waterfront properties to increase the value of the properties.
- Keep an eye on market trends: Real estate market trends can change quickly, so it is recommended to keep an eye on the market trends and adjust the business strategies accordingly. This can include analyzing the market demand for certain features and locations, and adjusting property prices and marketing strategies accordingly.
- Use multiple regression models: The multiple regression model (model_4) provides the best predictions for the house prices and can explain around 61.6% of the variation in the dependent variable 'price'. Therefore, it is recommended to use this model for predicting house prices and gaining insights into the factors that affect house values in King County.
- Overall, by following these recommendations, the stakeholder can make informed business decisions and increase their sales and revenue in the competitive real estate market of King County.

- **The potential benefits of following the above recommendations are:**

1. Increased revenue: By investing in properties located in desirable neighborhoods and zip codes, upgrading the properties with desirable features, and focusing on waterfront properties, the stakeholder can increase the value of their properties and attract more buyers. This can lead to increased revenue and profits.
2. Improved accuracy in property pricing: By using the multiple regression model to predict house prices, the stakeholder can make more accurate property pricing decisions, leading to better negotiation and sales strategies.
3. Improved customer satisfaction: By upgrading the properties with desirable features, the stakeholder can improve customer satisfaction and attract more buyers and renters, leading to increased revenue and better long-term customer relationships.
4. Competitive advantage: By keeping an eye on market trends and adjusting business strategies accordingly, the stakeholder can gain a competitive advantage in the real estate market of King County and improve their market position.
5. Improved decision-making: By gaining insights into the factors that affect house values in King County, the stakeholder can make more informed and data-driven business decisions, leading to better outcomes and improved business performance.

Overall, by following the recommendations, the stakeholder can benefit from increased revenue, improved customer satisfaction, a competitive advantage, and improved decision-making.

In [ ]:

```

```