Housing Analysis

Author: Eric Wehmueller

Overview

This project is the second project for Flatiron School's bootcamp program in Data Science. We are being placed into a hypothetical situation as a Data Scientist and hoping to provide value to our business for the scenario we are given.

Business Problem

I have been hired by a real estate agency that helps homeowners sell homes. For this project, I am to provide expected/estimated home prices to homeowners based on the logistics of their home. This can also give insight on how home renovations might increase the estimated value of their homes, and what type of potential renovations are best.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
import scipy.stats as stats
%matplotlib inline
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from statsmodels.formula.api import ols
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_squared_error
from statsmodels.stats.outliers_influence import varianc
```

Data Investigation and Cleaning

To start, we have access to the King County House Sales dataset. Let's take a look at this to get a feel for what our starting point is and what raw data we have to work with.

```
df_original = pd.read_csv("data\kc_house_data.csv")
```

```
df_original.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  float64
 9   view           21534 non-null  float64
 10  condition      21597 non-null  int64  
 11  grade          21597 non-null  int64  
 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(8), int64(11), object(2)
memory usage: 3.5+ MB
```

```
df_original.head(10)
```

	id	date	price	bedrooms	bathroo
0	7129300520	10/13/2014	221900.0	3	1.00
1	6414100192	12/9/2014	538000.0	3	2.25
2	5631500400	2/25/2015	180000.0	2	1.00
3	2487200875	12/9/2014	604000.0	4	3.00
4	1954400510	2/18/2015	510000.0	3	2.00
5	7237550310	5/12/2014	1230000.0	4	4.50
6	1321400060	6/27/2014	257500.0	3	2.25
7	2008000270	1/15/2015	291850.0	3	1.50
8	2414600126	4/15/2015	229500.0	3	1.00
9	3793500160	3/12/2015	323000.0	3	2.50

10 rows × 21 columns

Per the project description, I will be ignoring the following features: date, view, sqft_above, sqft_basement, yr_renovated, zipcode, lat, long, sqft_living15, sqft_lot15. For the time being, I am trying to make my modeling phase in this project as simple as possible.

```
df_col_drops = df_original.drop(columns=['id', 'date', '
display(df_col_drops)
```

	price	bedrooms	bathrooms	sqft_living	sq
0	221900.0	3	1.00	1180	565
1	538000.0	3	2.25	2570	724
2	180000.0	2	1.00	770	100
3	604000.0	4	3.00	1960	500
4	510000.0	3	2.00	1680	808
...	...	...	...	...	...
21592	360000.0	3	2.50	1530	113
21593	400000.0	4	2.50	2310	581
21594	402101.0	2	0.75	1020	135
21595	400000.0	3	2.50	1600	238
21596	325000.0	2	0.75	1020	107

21597 rows × 12 columns

```
df_col_drops.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   price        21597 non-null  float64
 1   bedrooms     21597 non-null  int64  
 2   bathrooms    21597 non-null  float64
 3   sqft_living  21597 non-null  int64  
 4   sqft_lot     21597 non-null  int64  
 5   floors       21597 non-null  float64
 6   waterfront   19221 non-null  float64
 7   condition    21597 non-null  int64  
 8   grade        21597 non-null  int64  
 9   yr_built     21597 non-null  int64  
 10  lat          21597 non-null  float64
 11  long         21597 non-null  float64
dtypes: float64(6), int64(6)
memory usage: 2.0 MB
```

Waterfront appears to have ~2000 null values. Let's investigate what values are in this column to see what we can do about the null values.

Which ones are the most important features?

```
df_col_drops.waterfront.value_counts()
```

```
0.0    19075
1.0      146
Name: waterfront, dtype: int64
```

Only 146 have a waterfront view. Since this is a binary-filled column, I believe we can fill in all NaNs with a zero value. This makes sense, as NaNs almost certainly denotes the absence of a waterfront view.

```
df_col_drops.waterfront.fillna(0, inplace=True)
display(df_col_drops.head())
```

	price	bedrooms	bathrooms	sqft_living	sqft_lo
0	221900.0	3	1.00	1180	5650
1	538000.0	3	2.25	2570	7242
2	180000.0	2	1.00	770	10000
3	604000.0	4	3.00	1960	5000
4	510000.0	3	2.00	1680	8080

```
df_col_drops.describe()
```

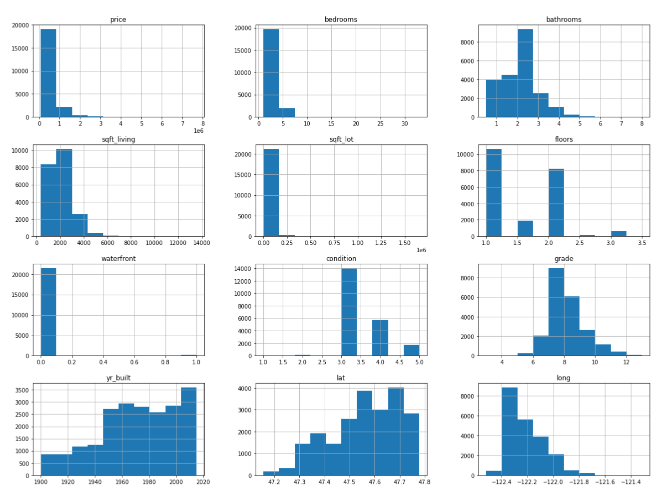
	price	bedrooms	bathrooms	sqft_liv
count	2.159700e+04	21597.000000	21597.000000	21597.00(
mean	5.402966e+05	3.373200	2.115826	2080.321{
std	3.673681e+05	0.926299	0.768984	918.1061z
min	7.800000e+04	1.000000	0.500000	370.0000(
25%	3.220000e+05	3.000000	1.750000	1430.000(
50%	4.500000e+05	3.000000	2.250000	1910.000(
75%	6.450000e+05	4.000000	2.500000	2550.000(
max	7.700000e+06	33.000000	8.000000	13540.00(

```
df_col_drops.columns
```

```
Index(['price', 'bedrooms', 'bathrooms', 'sqft_living', 's
qft_lot', 'floors',
       'waterfront', 'condition', 'grade', 'yr_built', 'la
t', 'long'],
      dtype='object')
```

```
#iterating over all columns except id to see general dis

df_col_drops.hist(figsize = (20,15));
```



It appears that we have some outliers in this data, so it's a little difficult to get a sense for what some the distrubutions actually are.

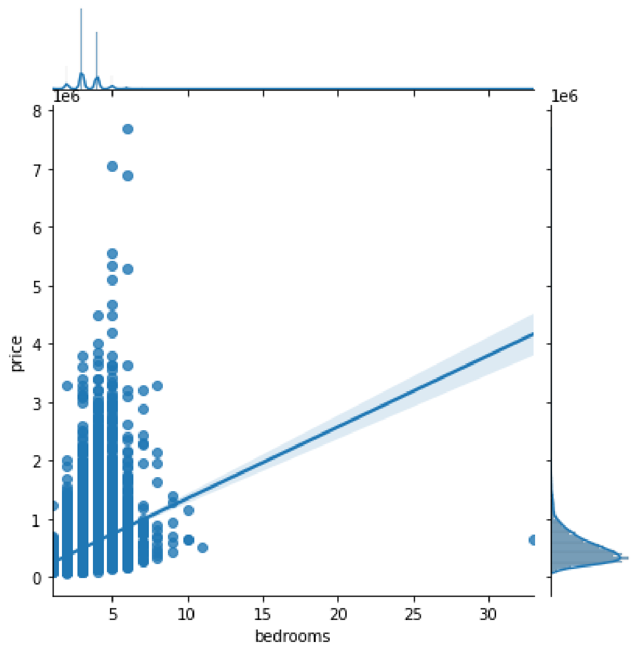
Specifically, I'm seeing a single entry priced at 7.7 million.

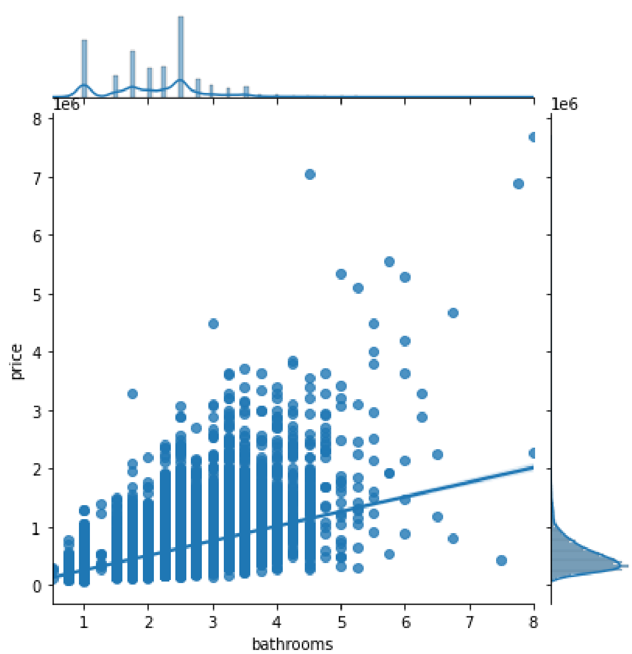
I also can't really tell what the bedroom distribution is with an outlier of 33.

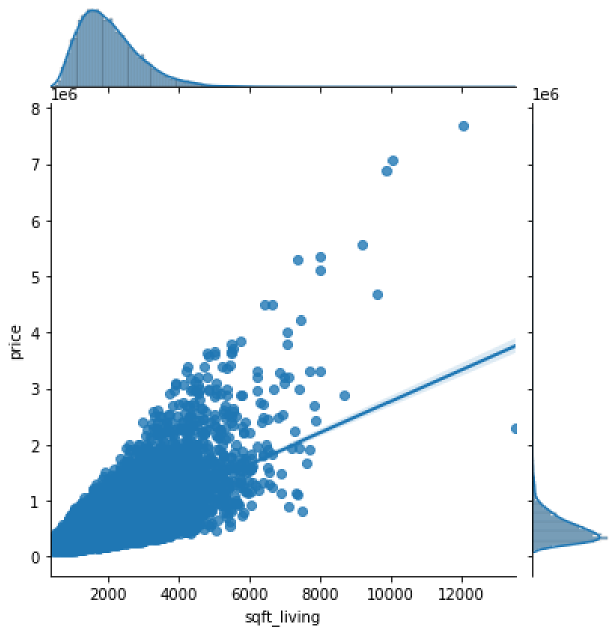
sqft_lot has only a single column in this view and the mean is vastly different from the median. We will need to take a closer look at this as well.

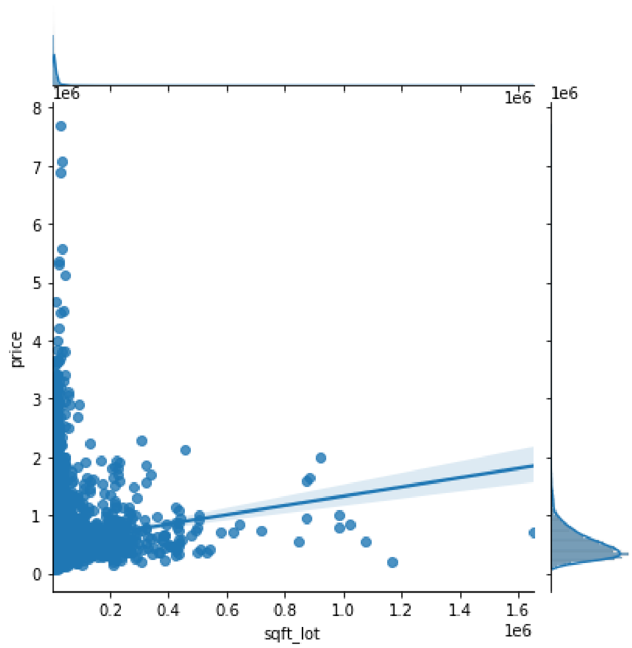
Condition and grade seem to be relatively normal.

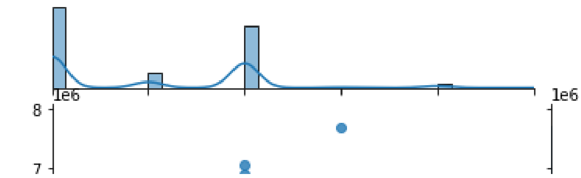
```
#Check for linearity via jointplots
for col_name in df_col_drops.columns[1:]:
    ax = sns.jointplot(x=col_name, y='price', data=df_co
    ax.
```

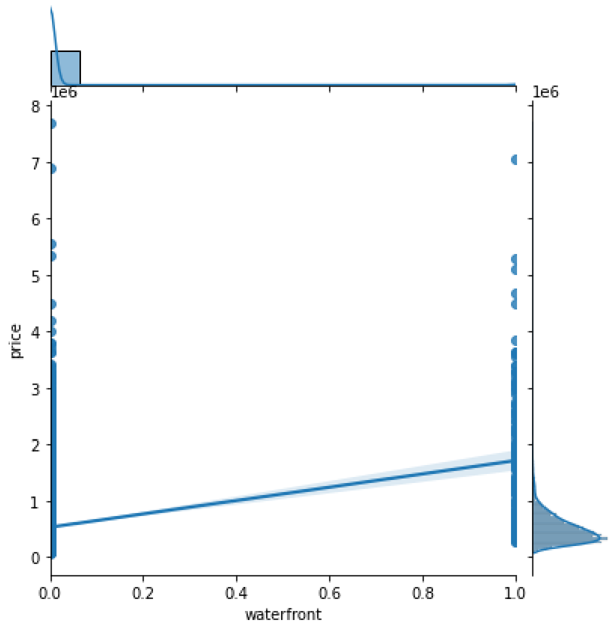


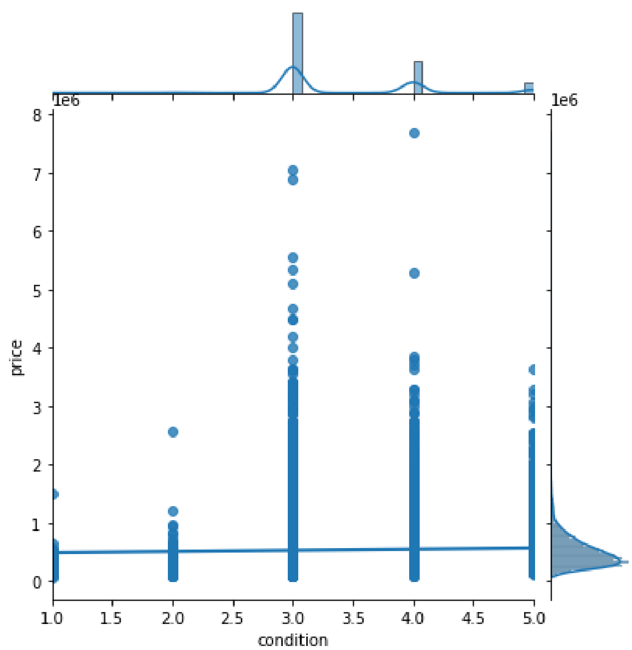


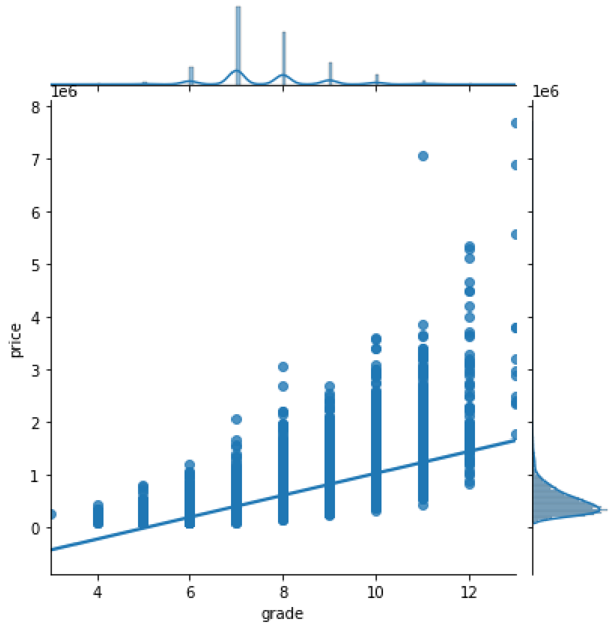

1e6
1e6
price
sqft_living
0
1
2
3
4
5
6
7
8
2000
4000
6000
8000
10000
12000

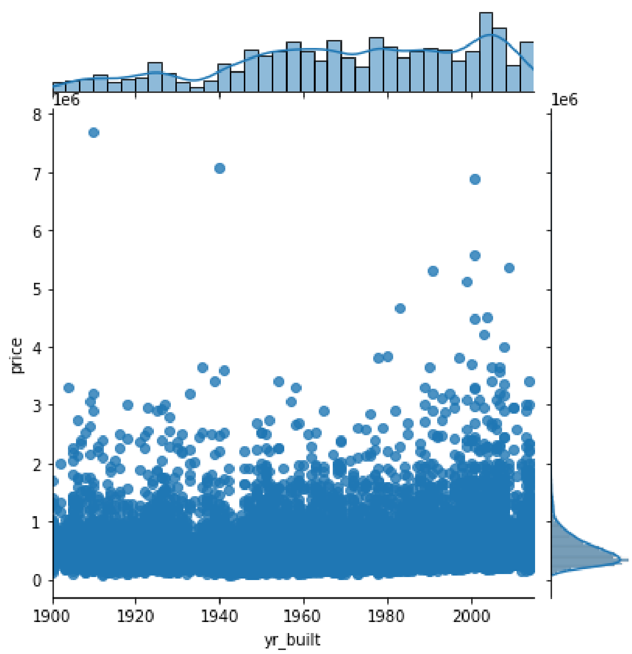

1e6
1e6
1e6
1e6
price
sqft_lot
0
1
2
3
4
5
6
7
8
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6

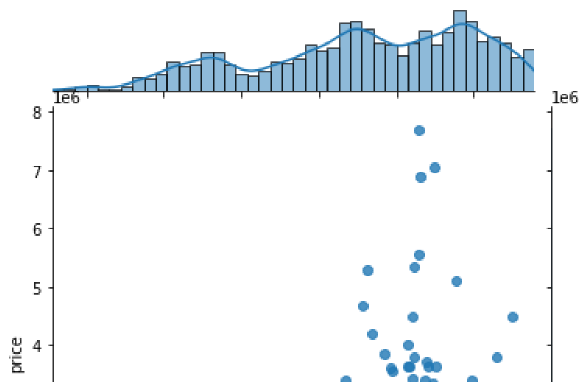


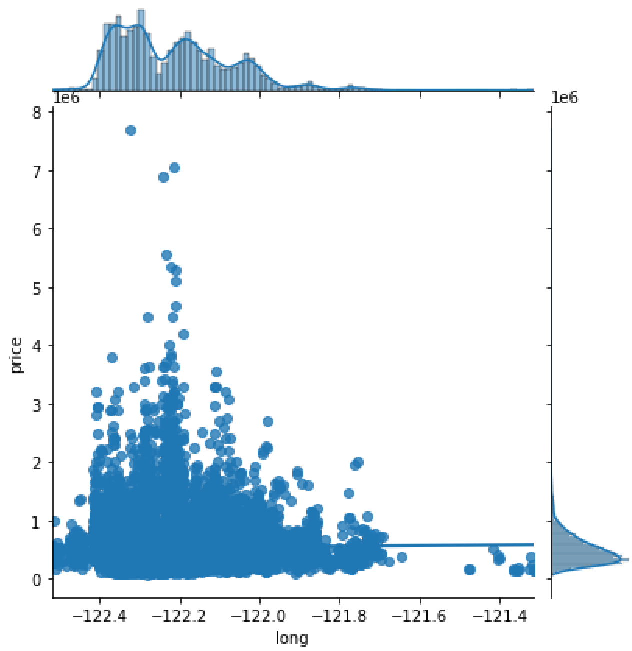

1e6
1e6
price
waterfront


1e6
1e6
price
condition


1e6
1e6
price
grade
0
1
2
3
4
5
6
7
8
4
6
8
10
12


1e6
1e6
price
yr_built
0
1
2
3
4
5
6
7
8
1900
1920
1940
1960
1980
2000





It worth noting that these jointplots reveal several of these columns to have linear relations with price.

Strong Linear Relation: sqft_living, grade

Somehwat Linear: bathrooms, sqft_lot, waterfront

Little to No Linear Relation: bedrooms, floors, condition, yr_built, lat, long

It appears that the features that have the largest impact on the price of a home are the square footage of the home, as well as the Grade- this rating is given by the King County Housing System. I have copied this system below for more context.

1-3 Falls short of minimum building standards. Normally cabin or inferior structure.

4 Generally older, low quality construction. Does not meet code.

5 Low construction costs and workmanship. Small, simple design.

6 Lowest grade currently meeting building code. Low quality materials and simple designs.

7 Average grade of construction and design. Commonly seen in plats and older sub-divisions.

8 Just above average in construction and design. Usually better materials in both the exterior and interior finish work.

9 Better architectural design with extra interior and exterior design and quality.

10 Homes of this quality generally have high quality features. Finish work is better and more design quality is seen in the floor plans. Generally have a larger square footage.

11 Custom design and higher quality finish work with added amenities of solid woods, bathroom fixtures and more luxurious options.

12 Custom design and excellent builders. All materials are of the highest quality and all conveniences are present.

13 Generally custom designed and built. Mansion level. Large amount of highest quality cabinet work, wood trim, marble, entry ways etc.

Feature Engineering

Two fields jump out at me: latitude and longitude. As we already know, this data is taken from the King County Housing dataset, which includes the city of Seattle. Let's engineer a feature that determines the distance from "downtown" using lat and long.

```
#using 47.605° N, 122.334° W as the exact point for down
dtwn_lat = 47.605
dtwn_long = -122.334
dtwn_coords = (dtwn_lat, dtwn_long)
print(type(dtwn_coords))

second_coords = (df_col_drops['lat'][0], df_col_drops['l
print(second_coords)
```

```
<class 'tuple'>
(47.5112, -122.257)
```

```
import haversine as hs

#solving for a single location, in kilometers
hs.haversine(dtwn_coords, second_coords)
```

```
11.923605090619347
```

```
#creating feature column
df_col_drops['dist_to_dtwn'] = df_col_drops.lat
for index, row in df_col_drops.iterrows():
    df_col_drops['dist_to_dtwn'][index] = hs.haversine(d
```

```
<ipython-input-15-edcbe8011a22>:4: SettingWithCopyWarnin
g:
A value is trying to be set on a copy of a slice from a D
ataFrame

See the caveats in the documentation: https://pandas.pydata.o
rg/pandas-docs/stable/user_guide/indexing.html#returning-a-view-vers
us-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/i
ndexing.html#returning-a-view-versus-a-copy)
  df_col_drops['dist_to_dtwn'][index] = hs.haversine(dtw
n_coords, point2=(df_col_drops['lat'][index], df_col_dro
ps['long'][index]) )
```

```
df_col_drops.head(10)
```

	price	bedrooms	bathrooms	sqft_living	sqft_l
0	221900.0	3	1.00	1180	5650
1	538000.0	3	2.25	2570	7242
2	180000.0	2	1.00	770	10000
3	604000.0	4	3.00	1960	5000
4	510000.0	3	2.00	1680	8080
5	1230000.0	4	4.50	5420	10193(
6	257500.0	3	2.25	1715	6819
7	291850.0	3	1.50	1060	9711
8	229500.0	3	1.00	1780	7470
9	323000.0	3	2.50	1890	6560

```
#dropping these so we don't confuse our model- dist_to_d
df_col_drops = df_col_drops.drop(['lat', 'long'], axis=1
```

Modeling

Model 1

```
outcome = 'price'
x_cols = list(df_col_drops.columns)
x_cols.remove(outcome)
print(x_cols)
```

```
['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floo
rs', 'waterfront', 'condition', 'grade', 'yr_built', 'dist
_to_dtwn']
```

```
price_log = np.log(df_col_drops.price)
price_log = pd.DataFrame(price_log)

X1= df_col_drops.drop('price', 1)
y1= price_log
```

```
X_train, X_test, y_train, y_test = train_test_split(X1,
```

```
#normalization
for col in x_cols:
    X_train[col] = (X_train[col] - X_train[col].mean())/
display(X_train.head())
print(len(X_train), len(X_test))
```

```
<ipython-input-21-c934d189158c>:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a Da
taFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.or
g/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-
a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guide/indexi
ng.html#returning-a-view-versus-a-copy)
  X_train[col] = (X_train[col] - X_train[col].mean())/X_tr
ain[col].std()
```

	bedrooms	bathrooms	sqft_living	sqft_lot	
11744	0.672806	0.508126	0.397234	-0.230020	0.9
12492	-0.397156	0.834553	-0.736429	-0.169504	-0.
13866	-0.397156	-1.450430	-1.096150	-0.105329	-0.
16645	-0.397156	0.508126	-0.125995	-0.155416	0.9
11548	0.672806	0.508126	-0.060591	-0.183641	0.9

```
17277 4320
```

```python
# predictors = '+'.join(x_cols)
# formula = outcome + '~' + predictors
# model = ols(formula=formula, data=train).fit()
# model.summary()

predictors = sm.add_constant(X_train)
model_1 = sm.OLS(y_train, predictors).fit()
model_1.summary()
```

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.730
Model:	OLS	**Adj. R-squared:**	0.729
Method:	Least Squares	**F-statistic:**	4658.
Date:	Mon, 29 Mar 2021	**Prob (F-statistic):**	0.00
Time:	14:04:14	**Log-Likelihood:**	-2160.4
No. Observations:	17277	**AIC:**	4343.
Df Residuals:	17266	**BIC:**	4428.
Df Model:	10		
Covariance Type:	nonrobust		

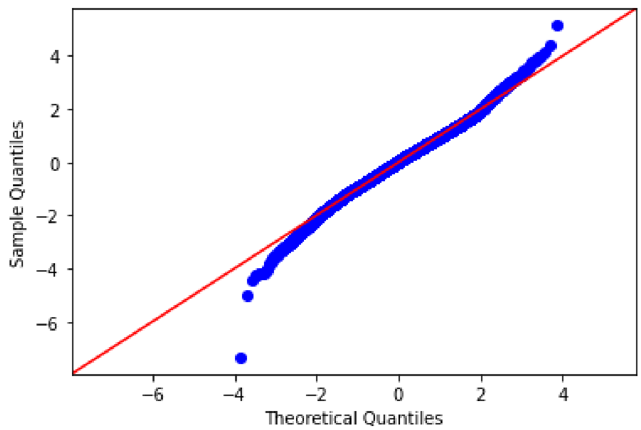
	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.97**
const	13.0448	0.002	6251.211	0.000	13.041	13.(
bedrooms	-0.0192	0.003	-7.252	0.000	-0.024	-0.0
bathrooms	0.0434	0.004	11.617	0.000	0.036	0.05
sqft_living	0.1983	0.004	46.479	0.000	0.190	0.2(
sqft_lot	0.0371	0.002	16.938	0.000	0.033	0.04
floors	0.0139	0.003	5.261	0.000	0.009	0.01
waterfront	0.0423	0.002	20.053	0.000	0.038	0.04
condition	0.0380	0.002	16.695	0.000	0.034	0.04
grade	0.2158	0.004	59.532	0.000	0.209	0.22
yr_built	-0.0564	0.003	-18.141	0.000	-0.063	-0.0
dist_to_dtwn	-0.1869	0.002	-76.113	0.000	-0.192	-0.1

Omnibus:	327.427	**Durbin-Watson:**	1.995
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	586.147
Skew:	-0.144	**Prob(JB):**	5.25e-128
Kurtosis:	3.855	**Cond. No.**	4.89

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

The p-values are less than 0.05 for our selected columns. Let's take a look at our residuals for normality.

```
fig = sm.graphics.qqplot(model_1.resid, dist=stats.norm,
```



This doesn't look great, as our QQ plot looks incorrect and we have a pronounced funnel shape on our check for homoscedasticity. We are going to need to make some changes.

```
regression = LinearRegression()
regression.fit(X_train, y_train)

#use the regression for the train and test data
y_hat_train = regression.predict(X_train)
y_hat_test = regression.predict(X_test)

#Root Mean Square Error
train_rmse = np.sqrt(mean_squared_error(y_train, y_hat_t
test_rmse = np.sqrt(mean_squared_error(y_test, y_hat_tes

print(f'Train Root Mean Square Error: {train_rmse}')
print(f'Test Root Mean Square Error: {test_rmse}')
```

```
Train Root Mean Square Error: 0.2742010910106044
Test Root Mean Square Error: 1764.4910958009116
```

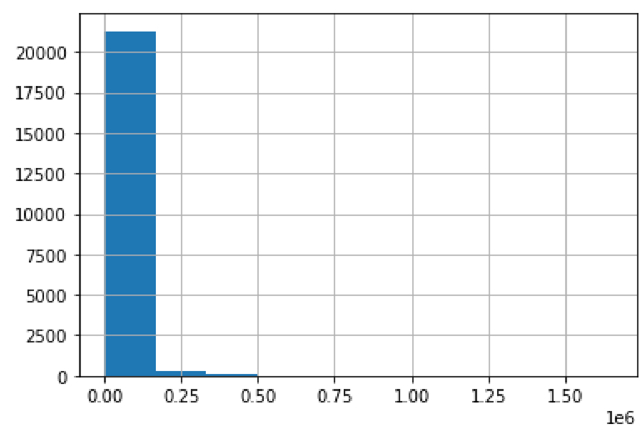
Models Addressing Multicollinearity

For this iteration, I'm going to remove some outliers. (log transformation?)

I recall having the most issues determining the normal distributions of sqft_lot and bedrooms, so I'm going to filter on both.

```
df_col_drops.sqft_lot.hist()
```

```
<AxesSubplot:>
```



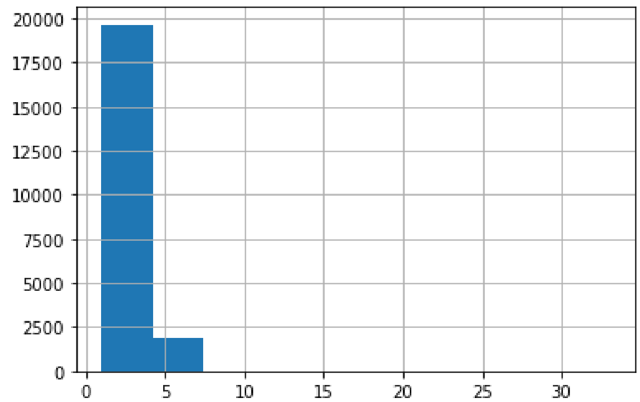

```
for i in range(80,100):
    q = i/100
    print("{} percentile: {}".format(q,df_col_drops.sqft_
```

```
0.8 percentile: 12182.399999999998
0.81 percentile: 12558.0
0.82 percentile: 13055.439999999995
0.83 percentile: 13503.68
0.84 percentile: 14197.0
0.85 percentile: 15000.0
0.86 percentile: 15716.040000000012
0.87 percentile: 16646.640000000003
0.88 percentile: 18000.0
0.89 percentile: 19550.0
0.9 percentile: 21371.600000000006
0.91 percentile: 24149.360000000015
0.92 percentile: 28505.119999999995
0.93 percentile: 34848.0
0.94 percentile: 37643.19999999999
0.95 percentile: 43307.200000000026
0.96 percentile: 50655.28
0.97 percentile: 67381.7199999999
0.98 percentile: 107157.0
0.99 percentile: 213008.0
```

I think filtering out homes with greater than 100k sqaure feet is acceptable here.

```
df_col_drops.bedrooms.hist()
```

```
<AxesSubplot:>
```




```
for i in range(80,100):
    q = i/100
    print("{} percentile: {}".format(q,df_col_drops.bedr
```

```
0.8 percentile: 4.0
0.81 percentile: 4.0
0.82 percentile: 4.0
0.83 percentile: 4.0
0.84 percentile: 4.0
0.85 percentile: 4.0
0.86 percentile: 4.0
0.87 percentile: 4.0
0.88 percentile: 4.0
0.89 percentile: 4.0
0.9 percentile: 4.0
0.91 percentile: 4.0
0.92 percentile: 5.0
0.93 percentile: 5.0
0.94 percentile: 5.0
0.95 percentile: 5.0
0.96 percentile: 5.0
0.97 percentile: 5.0
0.98 percentile: 5.0
0.99 percentile: 6.0
```

```
df_col_drops.bedrooms.value_counts()
```

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

I will also be filtering out all houses with more than 6 bedrooms, removing about 2% of the total entries. (may overlap with sq footage)

I will also include a log transformation to the price feature, as this may help fix our QQplot from Model 1.

```
orig_tot = len(df_col_drops)
df_outlier_filter = df_col_drops.copy()
df_outlier_filters = df_outlier_filter[df_outlier_filter
print('Percent removed:', (orig_tot -len(df_outlier_filt

df_outlier_filters = df_outlier_filters[df_outlier_filte
print('Percent removed:', (orig_tot -len(df_outlier_filt

#applying a log transformation to the price, which is ri
df_outlier_filter['price'] = np.log(df_outlier_filter['p

#train2, test2 = train_test_split(df_outlier_filters)
```

```
Percent removed: 0.021530768162244755
Percent removed: 0.024355234523313424
```

```
X2 = df_outlier_filter.drop('price', 1)
y2 = df_outlier_filter['price']
X_train2, X_test2, y_train2, y_test2 = train_test_split(

# Refit model with subset features
predictors = sm.add_constant(X_train2)
model_2 = sm.OLS(y_train2, predictors).fit()
model_2.summary()
```

OLS Regression Results

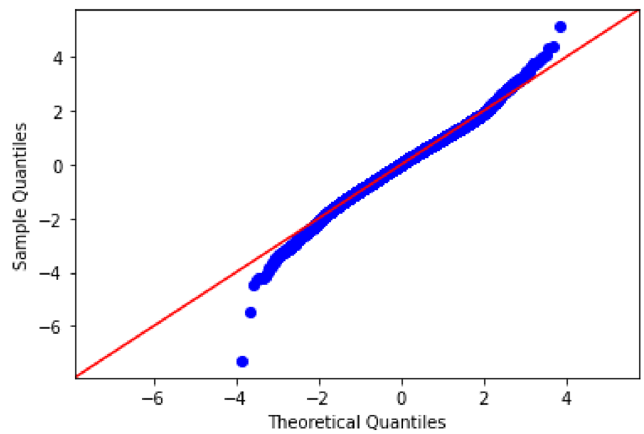
Dep. Variable:	price	**R-squared:**	0.731
Model:	OLS	**Adj. R-squared:**	0.730
Method:	Least Squares	**F-statistic:**	4682.
Date:	Mon, 29 Mar 2021	**Prob (F-statistic):**	0.00
Time:	14:04:15	**Log-Likelihood:**	-2144.8
No. Observations:	17277	**AIC:**	4312.
Df Residuals:	17266	**BIC:**	4397.
Df Model:	10		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975**
const	15.1190	0.204	74.281	0.000	14.720	15.51
bedrooms	-0.0202	0.003	-6.768	0.000	-0.026	-0.01
bathrooms	0.0559	0.005	11.456	0.000	0.046	0.065
sqft_living	0.0002	4.66e-06	46.015	0.000	0.000	0.000
sqft_lot	9.24e-07	5.21e-08	17.726	0.000	8.22e-07	1.03e-06
floors	0.0265	0.005	5.429	0.000	0.017	0.036
waterfront	0.5214	0.026	20.186	0.000	0.471	0.572
condition	0.0593	0.004	16.905	0.000	0.052	0.066
grade	0.1835	0.003	59.469	0.000	0.177	0.190
yr_built	-0.0020	0.000	-18.661	0.000	-0.002	-0.00
dist_to_dtwn	-0.0174	0.000	-74.853	0.000	-0.018	-0.01

Omnibus:	305.749	**Durbin-Watson:**	1.998
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	533.927
Skew:	-0.140	**Prob(JB):**	1.15e-116
Kurtosis:	3.815	**Cond. No.**	4.39e+06

```
Notes:
[1] Standard Errors assume that the covariance matrix of
the errors is correctly specified.
[2] The condition number is large, 4.39e+06. This might
indicate that there are
strong multicollinearity or other numerical problems.
```

```python
fig = sm.graphics.qqplot(model_2.resid, dist=stats.norm,
```




```python
regression2 = LinearRegression()
regression2.fit(X_train2, y_train2)

#use the regression for the train and test data
y_hat_train2 = regression2.predict(X_train2)
y_hat_test2 = regression2.predict(X_test2)

#Root Mean Square Error
train_rmse2 = np.sqrt(mean_squared_error(y_train2, y_hat
test_rmse2 = np.sqrt(mean_squared_error(y_test2, y_hat_t

print(f'Train Root Mean Square Error: {train_rmse2}')
print(f'Test Root Mean Square Error: {test_rmse2}')
```

```
Train Root Mean Square Error: 0.27395412776536393
Test Root Mean Square Error: 0.274568980060669
```

Similar problems as last time, but our OLS has alerted us that there is strong collinearity. Let's investigate what we should remove.

```
X = df_col_drops[x_cols]
X['constant'] = np.ones(X.shape[0])
vif = [variance_inflation_factor(X.values, i) for i in r
list(zip(x_cols, vif))
```

```
[('bedrooms', 1.6311136630472653),
 ('bathrooms', 3.215688966233134),
 ('sqft_living', 4.211173170919058),
 ('sqft_lot', 1.1150457494029746),
 ('floors', 1.6055072567823685),
 ('waterfront', 1.0219826889310346),
 ('condition', 1.1874067595264461),
 ('grade', 3.0015991231227797),
 ('yr_built', 2.240108337334204),
 ('dist_to_dtwn', 1.4004289073409446)]
```

You usually want to remove variables with a cif of 5~10 or greater, indicating that they are displaying multicollinearity with other variables in the feature set. None of these values are really in that range.

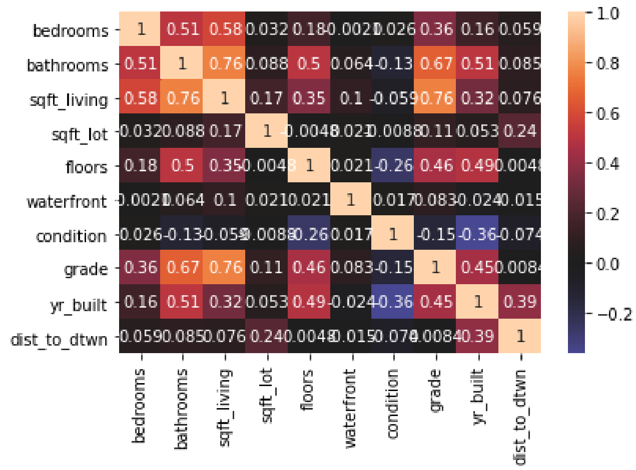
Going back to the drawing board, let's look at a multicolinearity heatmap to determine the columns to remove from our model.

```
first_features = ['bedrooms', 'bathrooms', 'sqft_living'
corr = df_col_drops[first_features].corr()
corr
```

	bedrooms	bathrooms	sqft_living	sqft_l
bedrooms	1.000000	0.514508	0.578212	0.03247
bathrooms	0.514508	1.000000	0.755758	0.08837
sqft_living	0.578212	0.755758	1.000000	0.17345
sqft_lot	0.032471	0.088373	0.173453	1.0000(
floors	0.177944	0.502582	0.353953	-0.0048
waterfront	-0.002127	0.063629	0.104637	0.0214&
condition	0.026496	-0.126479	-0.059445	-0.0088
grade	0.356563	0.665838	0.762779	0.11473
yr_built	0.155670	0.507173	0.318152	0.05294
dist_to_dtwn	0.058718	0.084731	0.076442	0.24347

```
sns.heatmap(corr, center=0, annot=True)
```

```
<AxesSubplot:>
```



sqft_living and grade = 0.76

sqft_living and bathrooms = 0.76

grade and bathrooms = 0.67

Let's remove grade and bathrooms for this model. We will also use our previous outlier filter, as this seems to be a step in the right direction.

```
# train3, test3 = train_test_split(df_outlier_filter)

# x_cols = ['bedrooms', 'sqft_living', 'sqft_lot', 'floo
# predictors = '+'.join(x_cols)
# formula = outcome + '~' + predictors
# model3 = ols(formula=formula, data=train3).fit()
# model3.summary()
X3 = df_outlier_filter.drop(columns=['price','grade','ba
y3 = df_outlier_filter['price']
X_train3, X_test3, y_train3, y_test3 = train_test_split(

# Refit model with subset features
predictors = sm.add_constant(X_train3)
model_3 = sm.OLS(y_train3, predictors).fit()
model_3.summary()
```

OLS Regression Results

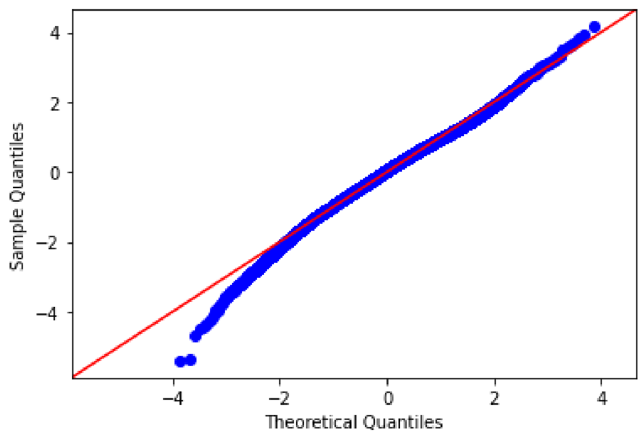
Dep. Variable:	price	**R-squared:**	0.672
Model:	OLS	**Adj. R-squared:**	0.672
Method:	Least Squares	**F-statistic:**	4430.
Date:	Mon, 29 Mar 2021	**Prob (F-statistic):**	0.00
Time:	14:04:17	**Log-Likelihood:**	-3788.4
No. Observations:	17277	**AIC:**	7595.
Df Residuals:	17268	**BIC:**	7665.
Df Model:	8		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.97
const	11.9934	0.208	57.584	0.000	11.585	12.4
bedrooms	-0.0413	0.003	-13.012	0.000	-0.047	-0.0
sqft_living	0.0004	3.42e-06	120.315	0.000	0.000	0.00
sqft_lot	1.108e-06	6.23e-08	17.801	0.000	9.86e-07	1.23 06
floors	0.0855	0.005	16.478	0.000	0.075	0.09
waterfront	0.5349	0.028	18.923	0.000	0.480	0.59
condition	0.0659	0.004	17.150	0.000	0.058	0.07
yr_built	0.0002	0.000	1.656	0.098	-3.24e-05	0.00
dist_to_dtwn	-0.0207	0.000	-82.778	0.000	-0.021	-0.0

Omnibus:	366.331	**Durbin-Watson:**	2.013
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	496.038
Skew:	-0.261	**Prob(JB):**	1.94e-108
Kurtosis:	3.645	**Cond. No.**	3.79e+06

```
Notes:
[1] Standard Errors assume that the covariance matrix of
the errors is correctly specified.
[2] The condition number is large, 3.79e+06. This might
indicate that there are
strong multicollinearity or other numerical problems.
```

```python
fig = sm.graphics.qqplot(model_3.resid, dist=stats.norm,
```



```
regression3 = LinearRegression()
regression3.fit(X_train3, y_train3)

#use the regression for the train and test data
y_hat_train3 = regression3.predict(X_train3)
y_hat_test3 = regression3.predict(X_test3)

#Root Mean Square Error
train_rmse3 = np.sqrt(mean_squared_error(y_train3, y_hat
test_rmse3 = np.sqrt(mean_squared_error(y_test3, y_hat_t

print(f'Train Root Mean Square Error: {train_rmse3}')
print(f'Test Root Mean Square Error: {test_rmse3}')
```

```
Train Root Mean Square Error: 0.3012951581957556
Test Root Mean Square Error: 0.3076316639605217
```

This is a modeling choice. There are pros and cons to this approach versus the first model. Removing multiple components has substantially diminished the model's performance, as indicated by the r-squared value. However, multicollinearity between the features has been reduced.

Model 4

Our QQ plots are less than ideal in previous models. Let's see if we can fix that by using a transform on the appropriate features.

```
for col_name in df_outlier_filter.columns[1:]:
    print(col_name)
    print(df_outlier_filter[col_name].skew())
```

```
bedrooms
2.023641235344595
bathrooms
0.5197092816403838
sqft_living
1.473215455425834
sqft_lot
13.072603567136046
floors
0.6144969756263127
waterfront
12.039584643829357
condition
1.0360374245132955
grade
0.7882366363846076
yr_built
-0.4694499764949978
dist_to_dtwn
0.769367697269784
```

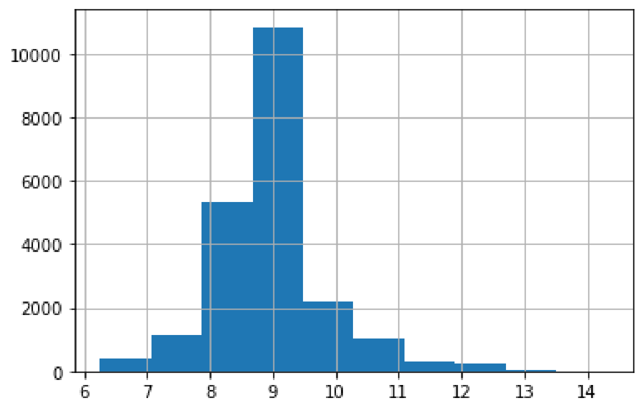
'sqft_lot' seems to be the main issue with the highest skew coefficient. I'm not sure if I should apply this to waterfront. We may need to use another method here, or look elsewhere for model improvements.

```
#only run once
df_outlier_filter['sqft_lot'] = np.log(df_outlier_filter
df_outlier_filter['sqft_lot'].skew()
```

```
0.9625003856495555
```

```
df_outlier_filter['sqft_lot'].hist()
```

```
<AxesSubplot:>
```




```
df_outlier_filter['bedrooms'] = np.log(df_outlier_filter
df_outlier_filter['bedrooms'].skew()
```

```
-0.6805637280656164
```

```
# x_cols = list(df_outlier_filter.columns)
# x_cols.remove(outcome)

# train4, test4 = train_test_split(df_outlier_filter)

# predictors = '+'.join(x_cols)
# formula = outcome + '~' + predictors
# model4 = ols(formula=formula, data=train4).fit()
# model4.summary()

X4 = df_outlier_filter.drop(columns=['price'], axis=1)
y4 = df_outlier_filter['price']
X_train4, X_test4, y_train4, y_test4 = train_test_split(

# Refit model with subset features
predictors = sm.add_constant(X_train4)
model_4 = sm.OLS(y_train4, predictors).fit()
model_4.summary()
```

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.730
Model:	OLS	**Adj. R-squared:**	0.730
Method:	Least Squares	**F-statistic:**	4678.
Date:	Mon, 29 Mar 2021	**Prob (F-statistic):**	0.00
Time:	14:04:17	**Log-Likelihood:**	-2162.6
No. Observations:	17277	**AIC:**	4347.
Df Residuals:	17266	**BIC:**	4433.
Df Model:	10		
Covariance Type:	nonrobust		

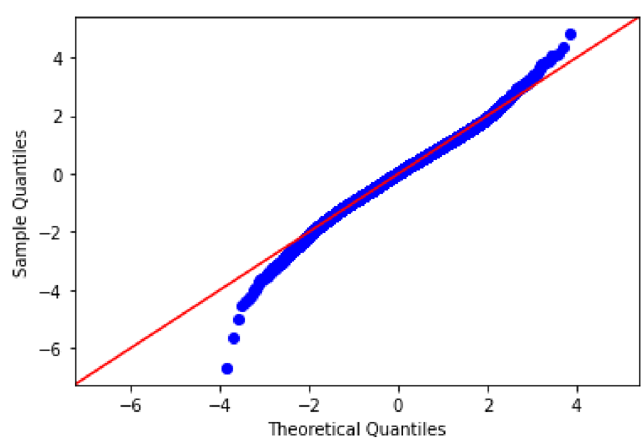
	coef	std err	t	P>|t|	[0.025	0.97
const	14.2244	0.210	67.717	0.000	13.813	14.6
bedrooms	-0.0612	0.009	-6.530	0.000	-0.080	-0.04
bathrooms	0.0602	0.005	12.300	0.000	0.051	0.07
sqft_living	0.0002	4.85e-06	40.562	0.000	0.000	0.00
sqft_lot	0.0609	0.003	19.535	0.000	0.055	0.06
floors	0.0602	0.005	11.676	0.000	0.050	0.07
waterfront	0.5191	0.026	20.178	0.000	0.469	0.57

condition	0.0618	0.004	17.621	0.000	0.055	0.06!
grade	0.1804	0.003	58.739	0.000	0.174	0.18(
yr_built	-0.0018	0.000	-16.730	0.000	-0.002	-0.0C
dist_to_dtwn	-0.0187	0.000	-73.247	0.000	-0.019	-0.01

Omnibus:	327.656	**Durbin-Watson:**	1.984
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	559.364
Skew:	-0.163	**Prob(JB):**	3.43e-122
Kurtosis:	3.819	**Cond. No.**	2.97e+05

```
Notes:
[1] Standard Errors assume that the covariance matrix of
the errors is correctly specified.
[2] The condition number is large, 2.97e+05. This might
indicate that there are
strong multicollinearity or other numerical problems.
```

```
fig = sm.graphics.qqplot(model_4.resid, dist=stats.norm,
```



```
regression4 = LinearRegression()
regression4.fit(X_train4, y_train4)

#use the regression for the train and test data
y_hat_train4 = regression4.predict(X_train4)
y_hat_test4 = regression4.predict(X_test4)

#Root Mean Square Error
train_rmse4 = np.sqrt(mean_squared_error(y_train4, y_hat
test_rmse4 = np.sqrt(mean_squared_error(y_test4, y_hat_t

print(f'Train Root Mean Square Error: {train_rmse4}')
print(f'Test Root Mean Square Error: {test_rmse4}')
```

```
Train Root Mean Square Error: 0.27423613934382146
Test Root Mean Square Error: 0.26984535362250917
```

This is a nice improvement. This is our best model thus far. It passes the normality check from looking at the QQ plot and it is homoscedastic.

Interpreting this model:

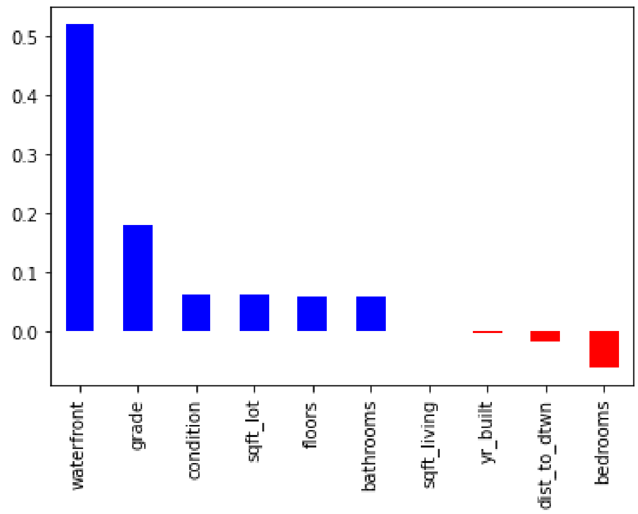
R-squared: 73.1% variation in the price can be explained by all of our feature columns.

Durbin-waton: A value preferred between 1-2 implies that the regression results are reliable from the side of homoscedasticity.

The highest coefficients belong to Grade and Waterfront: namely, what grade the home has been given by the King County Housing System. Additionally, having a waterfront view as a part of your home largely impacts the price.

When needed, we can now use this model to give us prediction values for an estimated price, given the values for the features of a home we are trying to sell. Obviously, someone would be unable to renovate their home to suddenly have a waterfront view, but doing something like adding a bathroom (the 3rd highest coefficient) seems to also have a significant impact of the expected price of a home for this model as well.

```
model_4.params[1:].sort_values(ascending=False).plot.bar
```



This is a visualization of our coefficients. To compare, I have taken the absolute value of each in the series, but made sure to indicate negative coefficients in red columns. I have also compared the Root Mean Squared Errors of our 4 models below.

```
print(f'Train Root Mean Square Error 1: {train_rmse}')
print(f'Test Root Mean Square Error 1: {test_rmse}')

print(f'Train Root Mean Square Error 2: {train_rmse2}')
print(f'Test Root Mean Square Error 2: {test_rmse2}')

print(f'Train Root Mean Square Error 3: {train_rmse3}')
print(f'Test Root Mean Square Error 3: {test_rmse3}')

print(f'Train Root Mean Square Error 4: {train_rmse4}')
print(f'Test Root Mean Square Error 4: {test_rmse4}')
```

```
Train Root Mean Square Error 1: 0.2742010910106044
Test Root Mean Square Error 1: 1764.4910958009116
Train Root Mean Square Error 2: 0.27395412776536393
Test Root Mean Square Error 2: 0.274568980060669
Train Root Mean Square Error 3: 0.3012951581957556
Test Root Mean Square Error 3: 0.3076316639605217
Train Root Mean Square Error 4: 0.27423613934382146
Test Root Mean Square Error 4: 0.26984535362250917
```

Just from glancing at this, I believe the best model to be Model 4. Although it may be slightly more overfitted than Model 2, Model 4 has the lowest Root Mean Squared Error on its test data. I'll now fit the model on our data, without a train test split.

```
X_final = df_outlier_filter.drop(columns=['price'], axis
y_final = df_outlier_filter['price']

predictors = sm.add_constant(X_final)
model_final = sm.OLS(y_final, predictors).fit()
model_final.summary()
```

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.731
Model:	OLS	**Adj. R-squared:**	0.730
Method:	Least Squares	**F-statistic:**	5851.
Date:	Mon, 29 Mar 2021	**Prob (F-statistic):**	0.00
Time:	14:21:05	**Log-Likelihood:**	-2632.6
No. Observations:	21597	**AIC:**	5287.
Df Residuals:	21586	**BIC:**	5375.
Df Model:	10		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	14.1879	0.187	75.701	0.000	13.821	14.555
bedrooms	-0.0610	0.008	-7.312	0.000	-0.077	-0.045
bathrooms	0.0631	0.004	14.468	0.000	0.055	0.072
sqft_living	0.0002	4.31e-06	44.623	0.000	0.000	0.000
sqft_lot	0.0629	0.003	22.683	0.000	0.057	0.068
floors	0.0590	0.005	12.825	0.000	0.050	0.068
waterfront	0.4995	0.023	21.736	0.000	0.454	0.545
condition	0.0595	0.003	19.086	0.000	0.053	0.066
grade	0.1812	0.003	66.004	0.000	0.176	0.187
yr_built	-0.0018	9.55e-05	-18.582	0.000	-0.002	-0.002
dist_to_dtwn	-0.0189	0.000	-82.640	0.000	-0.019	-0.018

Omnibus:	358.329	**Durbin-Watson:**	1.991
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	582.147
Skew:	-0.156	**Prob(JB):**	3.88e-127
Kurtosis:	3.741	**Cond. No.**	2.97e+05

Notes:

```
[1] Standard Errors assume that the covariance matrix of
the errors is correctly specified.
[2] The condition number is large, 2.97e+05. This might
indicate that there are
strong multicollinearity or other numerical problems.
```

```
regression_final = LinearRegression()
regression_final.fit(X_final, y_final)

y_hat_final = regression_final.predict(X_final)
rmse_final = np.sqrt(mean_squared_error(y_final, y_hat_f

print(f'Test Root Mean Square Error: {rmse_final}')
```

```
Test Root Mean Square Error: 0.2733395685228193
```

Conclusion

I believe the best model is Model 4, where the outliers have been filtered out and none of the features are removed . Although this suffers from multicollinearity, it has an r-squared value of ~0.73, which is the most accurate model in our analysis.

I believe this is acceptable within the context of this scenario. It affects the coefficients and p-values, but it does not influence the predicitons, precision of the predictions, and the statistics determining goodness of fit. Our primary goal is to have a model to make predictions for us.

To further improve this, I would use more of the columns included in the original dataset to try to increase my r-squared value and hopefully fix the QQplot issues I was having for all of my models.