1 Phase 2 Project

Author: Freddy Abrahamson
Date created: 12-29-2021
Discipline: Data Science

1.1 Overview

For this project, I will use multiple linear regression modeling to analyze house sales in King County, in Washington state.

1.2 Business Problem

The goal of this project is to to provide advice to homeowners about how home renovations can increase the value of their homes, and by what amount. The information for this project is derived from information comprised of the different characteristics of over 20,000 homes in King County,which is located in Washington State. I will use this information gain a better understanding about how different remodels, or renovations to the homes listed, impact their price.

1.3 Data Understanding

Describe the data being used for this project.

The data comes from the King County House Sales dataset, in the form of a 'csv' file. The file will be converted into a pandas dataframe. It contains information about the different characteristics of the homes in the King County area,including the number of bedrooms, building grades, square footage, and price. King County is located in Washington State, and has a size of approximately 2300 square miles, per the U.S Census Bureau:

kc_house_data.csv

I will be giving this dataframe a brief overview of its different characteristics, with a view toward using its columns as variables in a regression model. These include:

- dataframe shape: the number of rows and columns in the dataframe
- any missing/null values
- continuous variables
- categorical variables
- binary variables
- zero inflated variables
- outliers

Since the goal is to try to gain insights, as to how much much a particular upgrade or remodel can the impact the price of the house, as opposed to predicting home prices, I will be placing an emphasis on choosing features with the least explanatory overlap. To that end, for instance, I would favor a feature such as a bedroom, or a bathroom over square footage.

In [1]:
```
# Import standard packages
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings('ignore')
import matplotlib.pyplot as plt
plt.style.use('seaborn')
import seaborn as sns
from scipy import stats
import statsmodels.api as sm
import statsmodels.stats.api as sms
%matplotlib inline
```

In [2]:
```
#Importing dataframe
df = pd.read_csv('data/kc_house_data.csv')
df = df.sort_values(by=['price'])
```

I will use the following visualizations, and set of descriptive statistics, to familiarize myself: with the data.

In [3]:
```
#Dropping the 'id' column from the dataframe since it is of no use in the r
kc_df = df.drop('id',axis=1).copy()
kc_df.info()
```

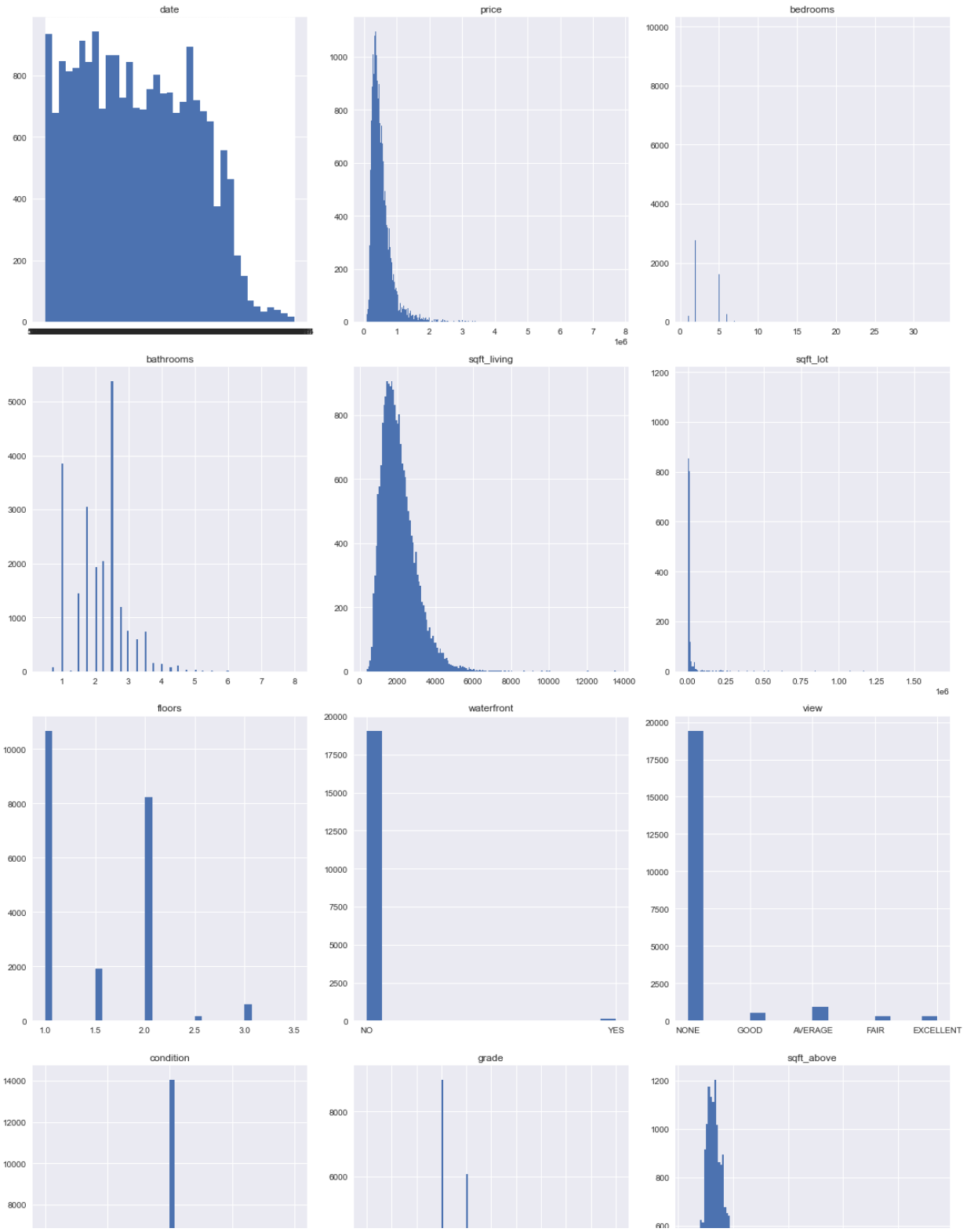
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21597 entries, 15279 to 7245
Data columns (total 20 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   date           21597 non-null  object 
 1   price          21597 non-null  float64
 2   bedrooms       21597 non-null  int64  
 3   bathrooms      21597 non-null  float64
 4   sqft_living    21597 non-null  int64  
 5   sqft_lot       21597 non-null  int64  
 6   floors         21597 non-null  float64
 7   waterfront     19221 non-null  object 
 8   view           21534 non-null  object 
 9   condition      21597 non-null  object 
 10  grade          21597 non-null  object 
 11  sqft_above     21597 non-null  int64  
 12  sqft_basement  21597 non-null  object 
 13  yr_built       21597 non-null  int64  
 14  yr_renovated   17755 non-null  float64
 15  zipcode        21597 non-null  int64  
 16  lat            21597 non-null  float64
 17  long           21597 non-null  float64
 18  sqft_living15  21597 non-null  int64  
 19  sqft_lot15     21597 non-null  int64  
dtypes: float64(6), int64(8), object(6)
memory usage: 3.5+ MB
```

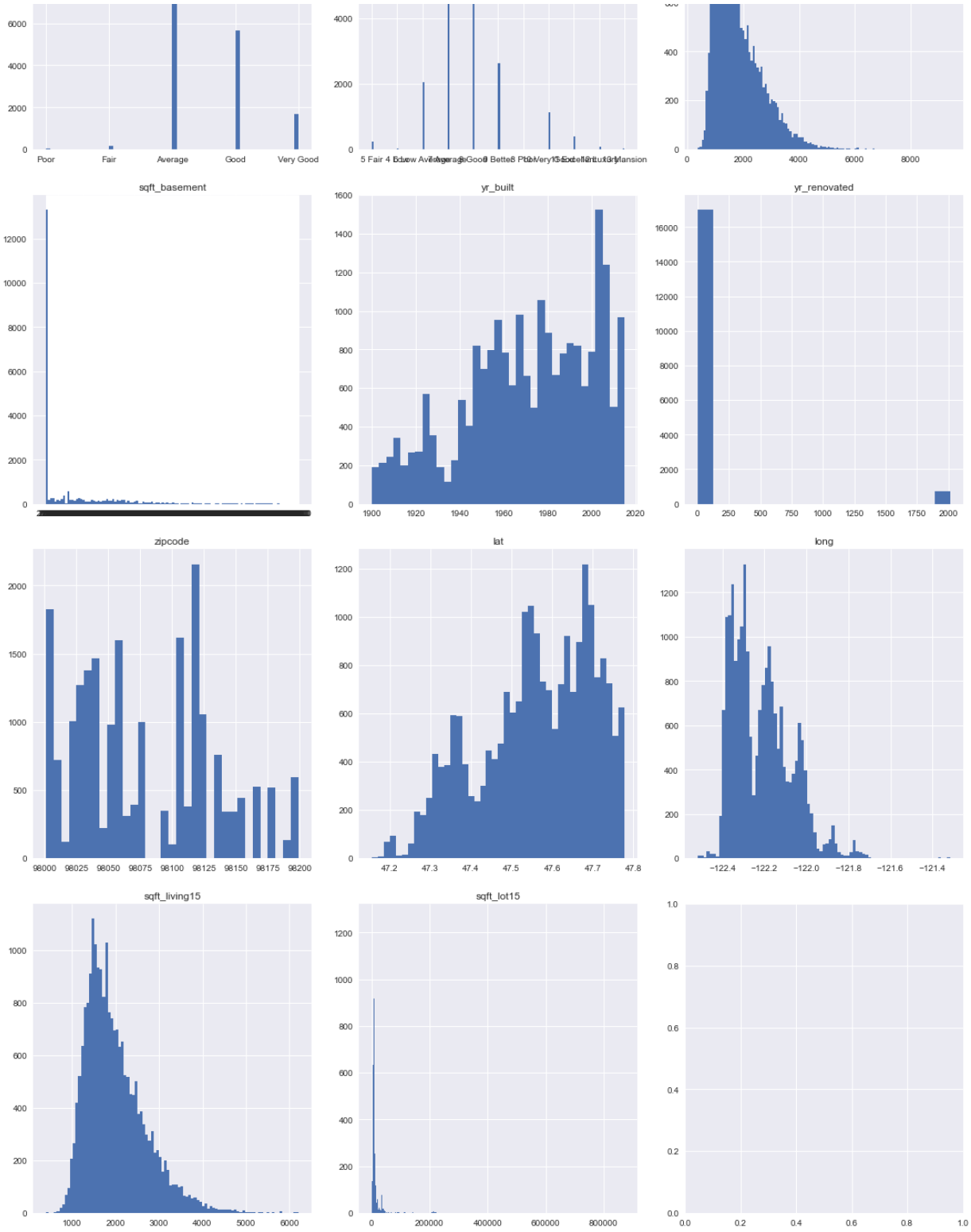
In [4]:
```
kc_df.describe()
```

Out[4]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	sqft
count	2.159700e+04	21597.000000	21597.000000	21597.000000	2.159700e+04	21597.000000	21597.
mean	5.402966e+05	3.373200	2.115826	2080.321850	1.509941e+04	1.494096	1788.
std	3.673681e+05	0.926299	0.768984	918.106125	4.141264e+04	0.539683	827.
min	7.800000e+04	1.000000	0.500000	370.000000	5.200000e+02	1.000000	370.
25%	3.220000e+05	3.000000	1.750000	1430.000000	5.040000e+03	1.000000	1190.
50%	4.500000e+05	3.000000	2.250000	1910.000000	7.618000e+03	1.500000	1560.
75%	6.450000e+05	4.000000	2.500000	2550.000000	1.068500e+04	2.000000	2210.
max	7.700000e+06	33.000000	8.000000	13540.000000	1.651359e+06	3.500000	9410.

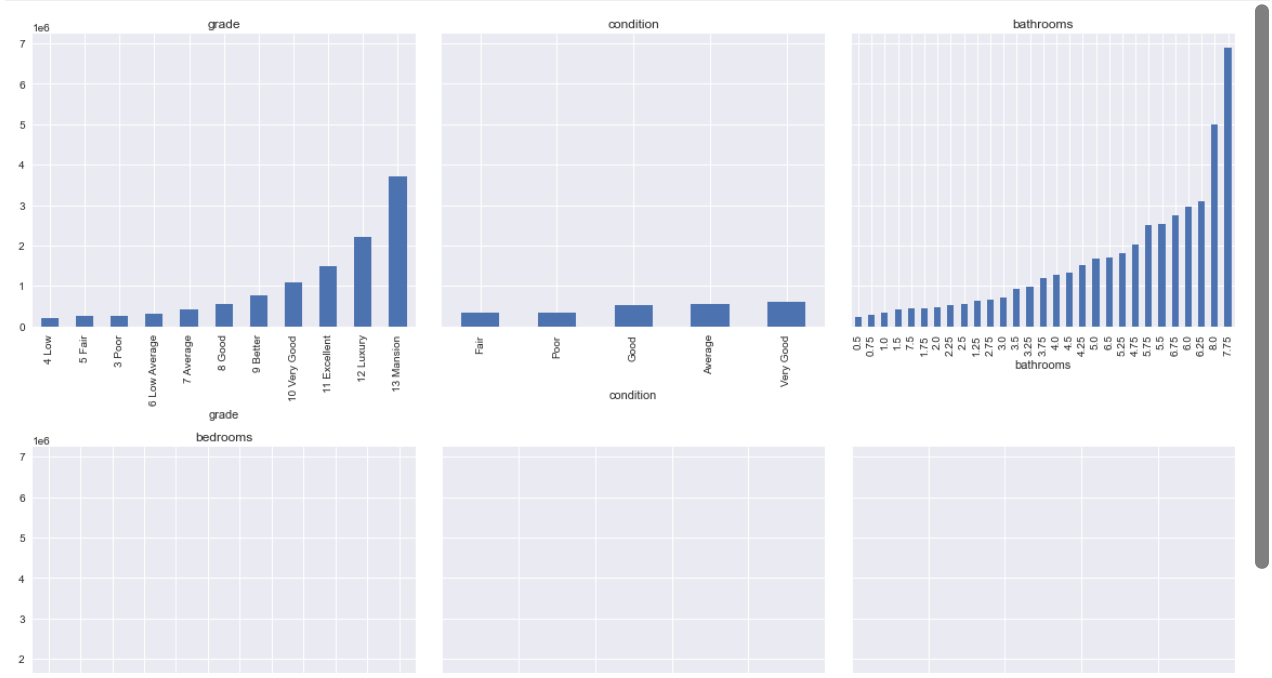
In [5]:
```
#Creating a histogram of each column of the dataframe to visually inspect e

fig, axes = plt.subplots(nrows=(7), ncols=3, figsize=(16,40))
kc_df_cols = kc_df.columns

for col, ax in zip(kc_df_cols, axes.flatten()):
    ax.hist(kc_df[col].dropna(), bins='auto')
    ax.set_title(col)

fig.tight_layout()
```




Poor
Fair
Average
Good
Very Good
sqft_basement
yr_built
yr_renovated
zipcode
lat
long
sqft_living15
sqft_lot15

In [6]:
```
df_nums_non_null_only = kc_df.iloc[:,np.r_[1:6,6,11,13:19,19]]
df_nums_non_null_only_cols_list = list(df_nums_non_null_only.columns)

count=1
plt.subplots(figsize=(20, 20))
for i in df_nums_non_null_only.columns:
    plt.subplot(5,3,count)
    sns.scatterplot(df_nums_non_null_only[i],df_nums_non_null_only["price"]
    count+=1

plt.show()
```

In [7]:
```
import matplotlib.pyplot as plt
%matplotlib inline

fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(16,10), sharey=True)

categoricals = ['grade', 'condition','bathrooms','bedrooms']

for col, ax in zip(categoricals, axes.flatten()):
    (df.groupby(col)                  # group values together by column of int
         .mean()['price']           # take the mean of the saleprice for each
         .sort_values()               # sort the groups in ascending order
         .plot
         .bar(ax=ax))                 # create a bar graph on the ax

    ax.set_title(col)                 # Make the title the name of the colum

fig.tight_layout()
```



1.3.1 Data Understanding Take-aways So Far:

1. There appear to be six categorical variables:

    - bedrooms
    - bathrooms
    - floors
    - view
    - condition
    - grade

2. There is atleast one binary variable:

- waterfront

3. There appear to be several potential zero inflated variables:
    - sqft_lot
    - sqft_basement
    - yr_renovated
    - sqft_lot15
4. There are also several right-skewed distributions, that may be normalized by a log transformation
5. Sqft_living appears to have the strongest correlation with price. Sqft_above and bathrooms, also appear to have a pretty strong linear relationship with price.
6. Since the relationship between the ordinal variables (grade,condition, bathrooms,bedrooms) are not repectively monotonic, some of the results given by the regression model may appear non-sensical.
7. The descriptive analysis, as well as the visualizations show that there are outliers. I will begin by removing these.

1.4 Removing Outliers

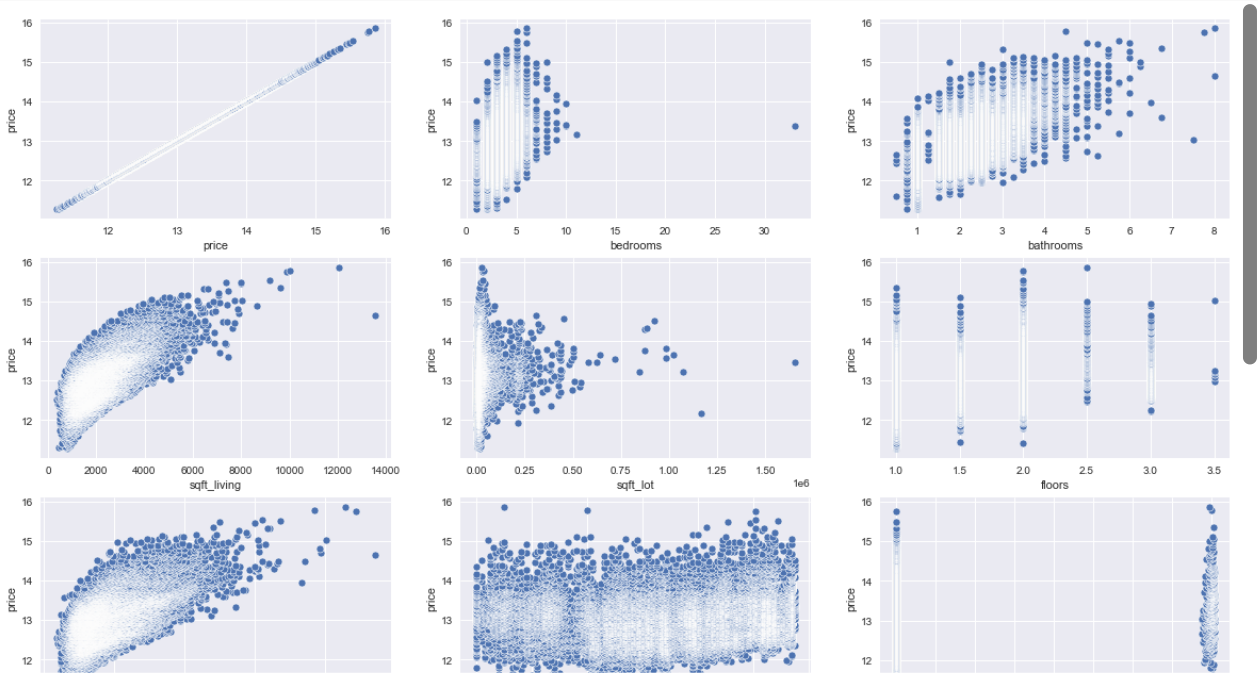
In [8]:
```
#Log transforming the price may help in more closely approximating a normal
kc_df['price'] = kc_df['price'].apply(lambda x: np.log(x))
```

In [9]:
```
#Using the Z-score to drop any 'price' values that are more than 3 standard
#Code taken from geeksforgeeks website:

# Z score
z = np.abs(stats.zscore(kc_df['price']))
```

In [10]:
```
print("Old Shape: ", kc_df.shape)

# Position of the outlier
drop_list = (np.where(z > 3))

''' Removing the Outliers '''
kc_df.drop(drop_list[0], inplace = True)


print("New Shape: ", kc_df.shape)
```

```
Old Shape:  (21597, 20)
New Shape:  (21428, 20)
```

In [11]:
```
#Reviewing the scatter plots after the outliers are dropped

df_nums_non_null_only = kc_df.iloc[:,np.r_[1:6,6,11,13:19,19]]
df_nums_non_null_only_cols_list = list(df_nums_non_null_only.columns)

count=1
plt.subplots(figsize=(20, 20))
for i in df_nums_non_null_only.columns:
    plt.subplot(5,3,count)
    sns.scatterplot(df_nums_non_null_only[i],df_nums_non_null_only["price"]
    count+=1

plt.show()
```



1.5 Split The Data

In [12]:
```
X_data = kc_df.drop('price',axis=1)
y_data = kc_df['price']
```

2 Preprocessing Data

In this section, I will deal with incorrect data types, as well as missing , or 'zero-inflated' data. I will also create the categorical 'dummy' columns , as well as perform transformations on continuous data if necessary.

In [13]:
```
kc_df.head()
```

Out[13]:

	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view
15279	5/6/2014	11.264464	2	1.00	780	16344	1.0	NO	NONE
465	5/23/2014	11.289782	1	0.75	430	5050	1.0	NaN	NONE
16184	3/24/2015	11.302204	2	1.00	730	9975	1.0	NaN	NONE
8267	11/5/2014	11.314475	3	1.00	860	10426	1.0	NO	NONE
2139	5/8/2014	11.320554	2	1.00	520	22334	1.0	NO	NONE

In [14]:
```
#Reviewing the features' correlations with price and each other:
X_data_corr_matrix = pd.concat([y_data,X_data],axis=1)
X_data_corr_matrix.corr()
```

Out[14]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	sqft_above	yr_bui
price	1.000000	0.342175	0.549168	0.693735	0.099157	0.311952	0.599718	0.07697
bedrooms	0.342175	1.000000	0.514700	0.577473	0.032125	0.180880	0.477602	0.15632
bathrooms	0.549168	0.514700	1.000000	0.755681	0.088137	0.503727	0.685987	0.50539
sqft_living	0.693735	0.577473	0.755681	1.000000	0.172687	0.358523	0.875408	0.31788
sqft_lot	0.099157	0.032125	0.088137	0.172687	1.000000	-0.003162	0.183642	0.05438
floors	0.311952	0.180880	0.503727	0.358523	-0.003162	1.000000	0.529531	0.48495
sqft_above	0.599718	0.477602	0.685987	0.875408	0.183642	0.529531	1.000000	0.42407
yr_built	0.076975	0.156323	0.505396	0.317881	0.054381	0.484957	0.424076	1.00000
yr_renovated	0.119082	0.018673	0.052167	0.056261	0.004389	0.004740	0.022725	-0.22492
zipcode	-0.036906	-0.152642	-0.205006	-0.197497	-0.129125	-0.062490	-0.259739	-0.35101
lat	0.450110	-0.009336	0.023668	0.052610	-0.085677	0.047639	-0.000769	-0.14992

In [15]:
```
# 'sqft_living' is the feature most highly correlated with 'price'. I will
#single feature. Although I do not plan to use it, it is important to have
baseline  = X_data['sqft_living']
```

In [16]:
```
#This model shows that 'sqft_living alone explains almost 48% of the varian
import statsmodels.api as sm
baseline_df = sm.add_constant(baseline)
baseline_model = sm.OLS(y_data,baseline_df).fit()
baseline_model.summary()
```

Out[16]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.481
Model:	OLS	**Adj. R-squared:**	0.481
Method:	Least Squares	**F-statistic:**	1.988e+04
Date:	Tue, 01 Feb 2022	**Prob (F-statistic):**	0.00
Time:	15:27:33	**Log-Likelihood:**	-9607.2
No. Observations:	21428	**AIC:**	1.922e+04
Df Residuals:	21426	**BIC:**	1.923e+04
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	12.2182	0.006	1902.668	0.000	12.206	12.231
sqft_living	0.0004	2.83e-06	140.991	0.000	0.000	0.000

Omnibus:	3.602	**Durbin-Watson:**	0.869
Prob(Omnibus):	0.165	**Jarque-Bera (JB):**	3.622
Skew:	0.028	**Prob(JB):**	0.163
Kurtosis:	2.971	**Cond. No.**	5.63e+03

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

2.1 Dealing with NaN(s) and/or Dropping Columns

In [17]:
```
# I will drop the date column, since there is a supposition that all the pr

X_data.drop('date',axis=1,inplace=True)
```

In [18]:
```
#Reviewing columns with missing data
yr_renovated_not_null = X_data['yr_renovated'].notnull().sum()
yr_renovated_null = len(X_data.loc[X_data['yr_renovated']==0])

missing_data = [x for x in X_data.columns if X_data[x].notnull().sum()!=214
for col in missing_data:
    percent_missing = 1-X_data[col].notnull().sum()/21428
    print("The percentage of data missing in the",col, "column is:",percent
print('The number of X_data.yr_renovated values equal to 0:',len(X_data.loc
print('The percentage of the [yr_renovated] non-missing values not equal to
       (yr_renovated_not_null - yr_renovated_null)/yr_renovated_not_null)
```

```
The percentage of data missing in the waterfront column is: 0.11004293447
825275
The percentage of data missing in the view column is: 0.00284674257980210
84
The percentage of data missing in the yr_renovated column is: 0.177618069
81519512
The number of X_data.yr_renovated values equal to 0: 16879
The percentage of the [yr_renovated] non-missing values not equal to 0:
0.0421632050845534
```

In [19]:
```
#I will drop the "yr_renovated" column since:
# 1.the percentage of missing data is approaching 20%
# 2.even within the 80% of non-null data, 96% is 0

#for the other two columns, I will replace the missing values with a string

X_data.drop('yr_renovated',axis=1,inplace=True)
X_data["waterfront"].fillna("missing", inplace = True)
X_data["view"].fillna("missing", inplace = True)
```

In [20]:
```
#The dataframe has no missing values:
X_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21428 entries, 15279 to 7245
Data columns (total 17 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   bedrooms       21428 non-null  int64  
 1   bathrooms      21428 non-null  float64
 2   sqft_living    21428 non-null  int64  
 3   sqft_lot       21428 non-null  int64  
 4   floors         21428 non-null  float64
 5   waterfront     21428 non-null  object 
 6   view           21428 non-null  object 
 7   condition      21428 non-null  object 
 8   grade          21428 non-null  object 
 9   sqft_above     21428 non-null  int64  
 10  sqft_basement  21428 non-null  object 
 11  yr_built       21428 non-null  int64  
 12  zipcode        21428 non-null  int64  
 13  lat            21428 non-null  float64
```

In [21]:
```
#I will create a new dataframe to differentiate between one with, and witho
X_data_w_cat = X_data.copy()
```

2.2 Creating Categorical Variables

In [22]:
```
# 1. I will apply one-hot encoding to the following features:
#    view
#    condition
#    grade
#    bedrooms
#    bathrooms
#    floors

# 2. I will do the same with the 'waterfront' since it now is split ino thr

#creating the new dataframes with the dummy columns:
view_dummies = pd.get_dummies(X_data['view'], prefix='view', drop_first=Tru
conditon_dummies = pd.get_dummies(X_data['condition'], prefix='cond', drop_
grade_dummies = pd.get_dummies(X_data['grade'], prefix='grd', drop_first=Tr
waterfront_dummies = pd.get_dummies(X_data['waterfront'], prefix='wtrfrnt',
bedroom_dummies = pd.get_dummies(X_data['bedrooms'], prefix='bdrm_', drop_f
bathroom_dummies = pd.get_dummies(X_data['bathrooms'], prefix='bthrm_', dro
floor_dummies = pd.get_dummies(X_data['floors'], prefix='flr_', drop_first=


#dropping the columns containing the categorical data from 'X_data' datafra
X_data_only_cont = X_data.drop(['view','condition','grade','waterfront','be
                                'bathrooms','floors'], axis=1)

#concatenating one-hot dataframes with 'X_data' dataframe:
X_data_w_cat = pd.concat([X_data_only_cont,view_dummies,conditon_dummies,gr
                          bedroom_dummies,bathroom_dummies,floor_dummies],

X_data_w_cat.describe()
```

Out[22]:

	sqft_living	sqft_lot	sqft_above	yr_built	zipcode	lat	
count	21428.000000	2.142800e+04	21428.000000	21428.000000	21428.000000	21428.000000	21428.
mean	2077.922718	1.513879e+04	1785.662918	1970.752147	98077.929438	47.560031	-122.
std	915.165858	4.152263e+04	824.815870	29.299933	53.539263	0.138733	0.
min	370.000000	5.200000e+02	370.000000	1900.000000	98001.000000	47.155900	-122.
25%	1429.250000	5.065000e+03	1190.000000	1951.000000	98033.000000	47.470200	-122.
50%	1910.000000	7.621500e+03	1560.000000	1974.000000	98065.000000	47.571800	-122.
75%	2550.000000	1.072000e+04	2210.000000	1996.000000	98118.000000	47.678200	-122.
max	13540.000000	1.651359e+06	9410.000000	2015.000000	98199.000000	47.777600	-121.

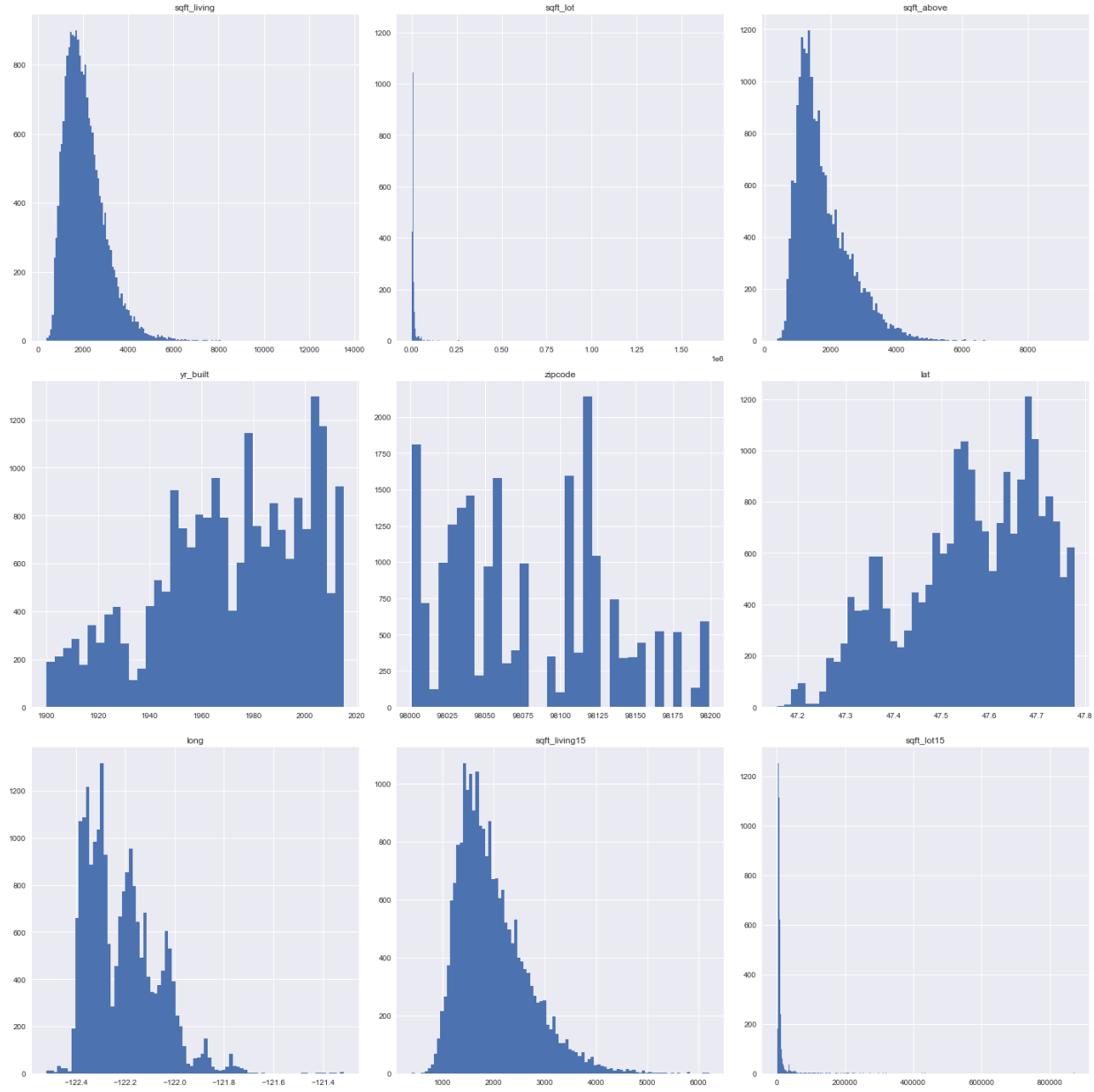
8 rows × 74 columns

In [23]:
```
#Given that 'sqft_basement' has missing data, and the vast majority of the

X_data_w_cat.drop(['sqft_basement'],axis=1, inplace=True)
X_data_w_cat.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21428 entries, 15279 to 7245
Data columns (total 74 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   sqft_living        21428 non-null  int64  
 1   sqft_lot           21428 non-null  int64  
 2   sqft_above         21428 non-null  int64  
 3   yr_built           21428 non-null  int64  
 4   zipcode            21428 non-null  int64  
 5   lat                21428 non-null  float64
 6   long               21428 non-null  float64
 7   sqft_living15      21428 non-null  int64  
 8   sqft_lot15         21428 non-null  int64  
 9   view_EXCELLENT     21428 non-null  uint8  
 10  view_FAIR          21428 non-null  uint8  
 11  view_GOOD          21428 non-null  uint8  
 12  view_NONE          21428 non-null  uint8  
 13  view_missing       21428 non-null  uint8  
```

2.3 Log Transforming the Continuous Features

In [24]:
```
#Creating a histogram of each column of the dataframe to visually inspect e
cont_preds_only = X_data_w_cat.iloc[:,0:9]


fig, axes = plt.subplots(nrows=(3), ncols=3, figsize=(20,20))
cont_preds_only_cols = cont_preds_only.columns

for col, ax in zip(cont_preds_only_cols, axes.flatten()):
    ax.hist(cont_preds_only[col].dropna(), bins='auto')
    ax.set_title(col)

fig.tight_layout()
```



In [25]:
```
#creating new dataframe that is log transformed
X_data_w_cat_log_trsfm = X_data_w_cat.copy()

#I will drop 'yr_built','zipcode', and 'long' since they have a low correla
#with several other predictors. I will keep "lat" for now, since it has a p
#and low correlations with the several other predictors:
X_data_w_cat_log_trsfm.drop(['yr_built','long','zipcode'],axis=1,inplace=Tr
```

In [26]:
```
#I will log transform the right skewed features:

continuous_features = ['sqft_living','sqft_above','sqft_lot','sqft_living15

for ftr in continuous_features:
    X_data_w_cat_log_trsfm['log_'+ftr] = np.log(X_data_w_cat_log_trsfm[ftr]
X_data_w_cat_log_trsfm.head()
```

Out[26]:

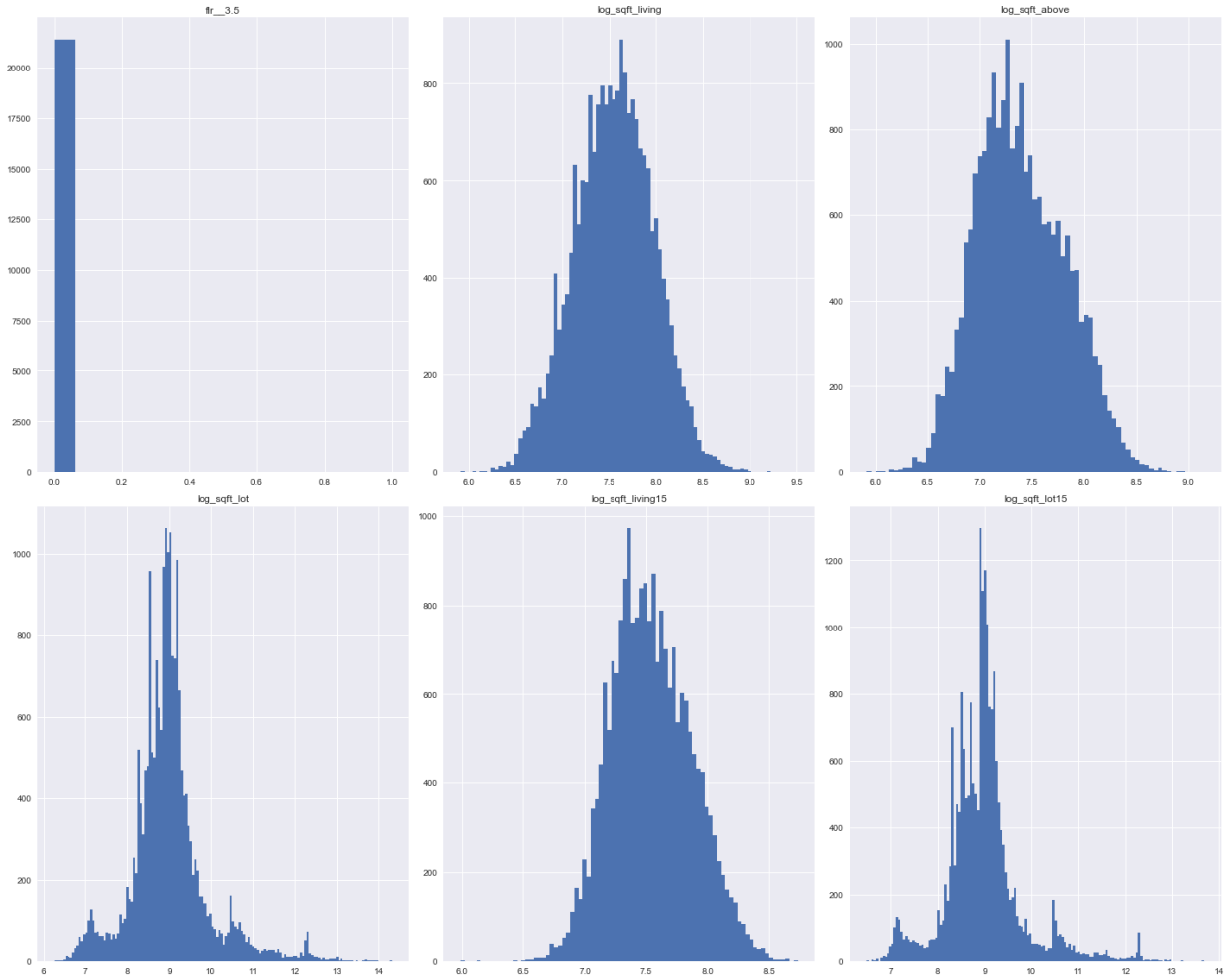
	sqft_living	sqft_lot	sqft_above	lat	sqft_living15	sqft_lot15	view_EXCELLENT	view_FA
15279	780	16344	780	47.4739	1700	10387	0	
465	430	5050	430	47.6499	1200	7500	0	
16184	730	9975	730	47.4808	860	9000	0	
8267	860	10426	860	47.4987	1140	11250	0	
2139	520	22334	520	47.4799	1572	10570	0	

5 rows × 76 columns

In [27]:
```
#I will drop the 'non-transformed' original columns:
X_data_w_cat_log_trsfm = X_data_w_cat_log_trsfm.drop(['sqft_living','sqft_l
                                                      'sqft_above'],axis=1)
X_data_w_cat_log_trsfm.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21428 entries, 15279 to 7245
Data columns (total 71 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   lat                 21428 non-null  float64
 1   view_EXCELLENT      21428 non-null  uint8
 2   view_FAIR           21428 non-null  uint8
 3   view_GOOD           21428 non-null  uint8
 4   view_NONE           21428 non-null  uint8
 5   view_missing        21428 non-null  uint8
 6   cond_Fair           21428 non-null  uint8
 7   cond_Good           21428 non-null  uint8
 8   cond_Poor           21428 non-null  uint8
 9   cond_Very Good      21428 non-null  uint8
 10  grd_11 Excellent    21428 non-null  uint8
 11  grd_12 Luxury       21428 non-null  uint8
 12  grd_13 Mansion      21428 non-null  uint8
 13  grd_3 Poor          21428 non-null  uint8
```

In [28]:
```
#Histogram of features after log transform:

#Creating a histogram of each column of the dataframe to visually inspect e
trnsfrmd_only = X_data_w_cat_log_trsfm.iloc[:,65:]


fig, axes = plt.subplots(nrows=(2), ncols=3, figsize=(20,16))
trnsfrmd_only_cols = trnsfrmd_only.columns

for col, ax in zip(trnsfrmd_only_cols, axes.flatten()):
    ax.hist(trnsfrmd_only[col].dropna(), bins='auto')
    ax.set_title(col)

fig.tight_layout()
```



2.4 Pre-processing is complete I will create two dataframes one scaled, and one not scaled

In [29]:
```
#I now have the complete dataframe. The data has all the categorical values
#all the interactions included. The data is also log transformed. Creating
#I will use to create my final dataframe (for easier interpretation) once m

#Completely pre-processed Dataframe with no scale
X_data_preproc_cmplt_no_scale = X_data_w_cat_log_trsfm.copy()

#1. Seperating the dataframe into seperate parts with continuous and catego
#these columns are non-consecutive I will divide into all the consecutive p
#concatenate in the order they were in the original dataframe:
cat_vars_1 =  X_data_w_cat_log_trsfm.iloc[:,1:66]
cont_vars_2 =  X_data_w_cat_log_trsfm.iloc[:,np.r_[0,66:70,70]]


#Scaling the dataframes with continuous data:
cont_vars_2_scaled = (cont_vars_2 - np.mean(cont_vars_2))/ np.std(cont_vars

#putting the datframe back together:
X_data_preproc_cmplt_scaled = pd.concat([cat_vars_1,cont_vars_2_scaled],axi
X_data_preproc_cmplt_scaled.head()
```

Out[29]:

	view_EXCELLENT	view_FAIR	view_GOOD	view_NONE	view_missing	cond_Fair	cond_Good
15279	0	0	0	1	0	0	0
465	0	0	0	1	0	1	0
16184	0	0	0	1	0	0	0
8267	0	0	0	1	0	0	0
2139	0	0	0	1	0	1	0

5 rows × 71 columns

In [30]:
```
X_data_preproc_cmplt_scaled.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21428 entries, 15279 to 7245
Data columns (total 71 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   view_EXCELLENT      21428 non-null  uint8
 1   view_FAIR           21428 non-null  uint8
 2   view_GOOD           21428 non-null  uint8
 3   view_NONE           21428 non-null  uint8
 4   view_missing        21428 non-null  uint8
 5   cond_Fair           21428 non-null  uint8
 6   cond_Good           21428 non-null  uint8
 7   cond_Poor           21428 non-null  uint8
 8   cond_Very Good      21428 non-null  uint8
 9   grd_11 Excellent    21428 non-null  uint8
 10  grd_12 Luxury       21428 non-null  uint8
 11  grd_13 Mansion      21428 non-null  uint8
 12  grd_3 Poor          21428 non-null  uint8
 13  grd_4 Low           21428 non-null  uint8
 14  grd_5 Fair          21428 non-null  uint8
 15  grd_6 Low Average   21428 non-null  uint8
 16  grd_7 Average       21428 non-null  uint8
 17  grd_8 Good          21428 non-null  uint8
 18  grd_9 Better        21428 non-null  uint8
 19  wtrfrnt_YES         21428 non-null  uint8
 20  wtrfrnt_missing     21428 non-null  uint8
 21  bdrm__2             21428 non-null  uint8
 22  bdrm__3             21428 non-null  uint8
 23  bdrm__4             21428 non-null  uint8
 24  bdrm__5             21428 non-null  uint8
 25  bdrm__6             21428 non-null  uint8
 26  bdrm__7             21428 non-null  uint8
 27  bdrm__8             21428 non-null  uint8
 28  bdrm__9             21428 non-null  uint8
 29  bdrm__10            21428 non-null  uint8
 30  bdrm__11            21428 non-null  uint8
 31  bdrm__33            21428 non-null  uint8
 32  bthrm__0.75         21428 non-null  uint8
 33  bthrm__1.0          21428 non-null  uint8
 34  bthrm__1.25         21428 non-null  uint8
 35  bthrm__1.5          21428 non-null  uint8
 36  bthrm__1.75         21428 non-null  uint8
 37  bthrm__2.0          21428 non-null  uint8
 38  bthrm__2.25         21428 non-null  uint8
 39  bthrm__2.5          21428 non-null  uint8
 40  bthrm__2.75         21428 non-null  uint8
 41  bthrm__3.0          21428 non-null  uint8
 42  bthrm__3.25         21428 non-null  uint8
 43  bthrm__3.5          21428 non-null  uint8
 44  bthrm__3.75         21428 non-null  uint8
 45  bthrm__4.0          21428 non-null  uint8
 46  bthrm__4.25         21428 non-null  uint8
 47  bthrm__4.5          21428 non-null  uint8
 48  bthrm__4.75         21428 non-null  uint8
 49  bthrm__5.0          21428 non-null  uint8
```

```
 50  bthrm__5.25         21428 non-null  uint8
 51  bthrm__5.5          21428 non-null  uint8
 52  bthrm__5.75         21428 non-null  uint8
 53  bthrm__6.0          21428 non-null  uint8
 54  bthrm__6.25         21428 non-null  uint8
 55  bthrm__6.5          21428 non-null  uint8
 56  bthrm__6.75         21428 non-null  uint8
 57  bthrm__7.5          21428 non-null  uint8
 58  bthrm__7.75         21428 non-null  uint8
 59  bthrm__8.0          21428 non-null  uint8
 60  flr__1.5            21428 non-null  uint8
 61  flr__2.0            21428 non-null  uint8
 62  flr__2.5            21428 non-null  uint8
 63  flr__3.0            21428 non-null  uint8
 64  flr__3.5            21428 non-null  uint8
 65  lat                 21428 non-null  float64
 66  log_sqft_living     21428 non-null  float64
 67  log_sqft_above      21428 non-null  float64
 68  log_sqft_lot        21428 non-null  float64
 69  log_sqft_living15   21428 non-null  float64
 70  log_sqft_lot15      21428 non-null  float64
dtypes: float64(6), uint8(65)
memory usage: 2.5 MB
```

3 Feature Selection

3.1 Baseline models:

In [31]:
```
# I will begin by creating three different baseline models:

# 1. The feature with the highest corellation to the dependent variable (pr
# 2. The dataframe with all the features, as well as 'dummy' features, log
# 3. In this dataframe I have dropped some 'generalized' features with high
```

In [32]:
```
#check to see how features correlate with 'price':

pd.set_option('display.max_rows', None)
corr_train_data = pd.concat([y_data, X_data_preproc_cmplt_scaled], axis=1)
abs_corr_train_data_price = corr_train_data.corr().price.apply(lambda x: ab
abs_corr_train_data_price.sort_values(ascending=False).head(20)
```

Out[32]:
```
price                1.000000
log_sqft_living      0.673606
log_sqft_living15    0.607271
log_sqft_above       0.584432
lat                  0.450110
bthrm__1.0           0.339781
grd_7 Average        0.337364
view_NONE            0.327100
grd_9 Better         0.310535
grd_6 Low Average    0.309619
grd_11 Excellent     0.279746
flr__2.0             0.278656
bdrm__4              0.226270
view_EXCELLENT       0.223593
bthrm__3.5           0.208360
bdrm__3              0.191951
bthrm__3.25          0.190208
bdrm__5              0.180959
bdrm__2              0.180730
grd_12 Luxury        0.179048
Name: price, dtype: float64
```

In [33]:
```
#setting up baseline dataframes:

baseline_1a = X_data_preproc_cmplt_scaled[['log_sqft_living']]
baseline_2 = X_data_preproc_cmplt_scaled
baseline_3 = X_data_preproc_cmplt_scaled.drop(['log_sqft_living','log_sqft_
                                               'log_sqft_above'],axis =1)
```

In [34]:
```
baseline_1a_df = sm.add_constant(baseline_1a)
model_baseline_1a = sm.OLS(y_data,baseline_1a_df).fit()
model_baseline_1a.summary()
```

Out[34]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.454
Model:	OLS	**Adj. R-squared:**	0.454
Method:	Least Squares	**F-statistic:**	1.780e+04
Date:	Tue, 01 Feb 2022	**Prob (F-statistic):**	0.00
Time:	15:27:44	**Log-Likelihood:**	-10161.
No. Observations:	21428	**AIC:**	2.033e+04
Df Residuals:	21426	**BIC:**	2.034e+04
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	13.0468	0.003	4913.115	0.000	13.042	13.052

In [35]:
```
baseline_2_df = sm.add_constant(baseline_2)
model_baseline_2 = sm.OLS(y_data,baseline_2_df).fit()
model_baseline_2.summary()
```

Out[35]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.758
Model:	OLS	**Adj. R-squared:**	0.757
Method:	Least Squares	**F-statistic:**	943.6
Date:	Tue, 01 Feb 2022	**Prob (F-statistic):**	0.00
Time:	15:27:44	**Log-Likelihood:**	-1426.0
No. Observations:	21428	**AIC:**	2996.
Df Residuals:	21356	**BIC:**	3570.
Df Model:	71		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	13.0830	0.131	101.145	0.000	13.025	13.540

In [36]:
```
baseline_3_df = sm.add_constant(baseline_3)
model_baseline_3 = sm.OLS(y_data,baseline_3_df).fit()
model_baseline_3.summary()
```

Out[36]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.724
Model:	OLS	**Adj. R-squared:**	0.723
Method:	Least Squares	**F-statistic:**	835.1
Date:	Tue, 01 Feb 2022	**Prob (F-statistic):**	0.00
Time:	15:27:44	**Log-Likelihood:**	-2857.6
No. Observations:	21428	**AIC:**	5851.
Df Residuals:	21360	**BIC:**	6393.
Df Model:	67		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	13.2851	0.140	94.738	0.000	13.010	13.560

While removing several features that 'encompassed' much of the explanatory value of other features, drop the R-squared by about three points, I think it was a worthy trade-off because it makes for a more stable model, where we can more decisively isolate the explanatory power of each feature.

3.2 Manually removing features with high p-values

In [37]:
```
#Removing features with p=values over .2:
features_1 = baseline_3.drop(['wtrfrnt_missing','bdrm__11','bdrm__33','bthr
```

In [38]:
```
features_1_df = sm.add_constant(features_1)
model_features_1 = sm.OLS(y_data,features_1_df).fit()
model_features_1.summary()
```

Out[38]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.724
Model:	OLS	**Adj. R-squared:**	0.723
Method:	Least Squares	**F-statistic:**	917.3
Date:	Tue, 01 Feb 2022	**Prob (F-statistic):**	0.00
Time:	15:27:45	**Log-Likelihood:**	-2860.0
No. Observations:	21428	**AIC:**	5844.
Df Residuals:	21366	**BIC:**	6338.
Df Model:	61		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	13.4498	0.040	338.323	0.000	13.372	13.528

In [39]:
```
#Removing any features with p-values in the double digits:
features_2 = features_1.drop(['grd_3 Poor','bdrm__3','bdrm__7','bthrm__1.25
```

In [40]:
```
features_2_df = sm.add_constant(features_2)
model_features_2 = sm.OLS(y_data,features_2_df).fit()
model_features_2.summary()
```

Out[40]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.724
Model:	OLS	**Adj. R-squared:**	0.723
Method:	Least Squares	**F-statistic:**	998.9
Date:	Tue, 01 Feb 2022	**Prob (F-statistic):**	0.00
Time:	15:27:45	**Log-Likelihood:**	-2863.7
No. Observations:	21428	**AIC:**	5841.
Df Residuals:	21371	**BIC:**	6296.
Df Model:	56		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	13.4746	0.035	388.808	0.000	13.407	13.543

In [41]:
```
#Removing any features with p-values over .05:
features_3 = features_2.drop(['cond_Poor','bdrm__10'],axis=1)
```

In [42]:
```
features_3_df = sm.add_constant(features_3)
model_features_3 = sm.OLS(y_data,features_3_df).fit()
model_features_3.summary()
```

Out[42]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.723
Model:	OLS	**Adj. R-squared:**	0.723
Method:	Least Squares	**F-statistic:**	1036.
Date:	Tue, 01 Feb 2022	**Prob (F-statistic):**	0.00
Time:	15:27:45	**Log-Likelihood:**	-2866.9
No. Observations:	21428	**AIC:**	5844.
Df Residuals:	21373	**BIC:**	6282.
Df Model:	54		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	13.4750	0.035	388.783	0.000	13.407	13.543

3.3 Checking for multi-colinearity using the VIF

With all the p-values below .05, I will direct my attention toward reducing multi-colinearity:

In [43]:
```
# Using V.I.F to identify the features that are most highly correlated with

from statsmodels.stats.outliers_influence import variance_inflation_factor
vif = [variance_inflation_factor(features_3.values, i) for i in range(featu
pd.Series(vif, index=features_3.columns, name="Variance Inflation Factor")
```

Out[43]:
```
view_EXCELLENT         1.818826
view_FAIR              1.332951
view_GOOD              1.510708
view_NONE             20.956567
view_missing           1.067069
cond_Fair              1.035367
cond_Good              1.596238
cond_Very Good         1.203570
grd_11 Excellent       1.405275
grd_12 Luxury          1.206301
grd_13 Mansion         1.291930
grd_4 Low              1.026245
grd_5 Fair             1.340227
grd_6 Low Average      3.884111
grd_7 Average         11.561338
grd_8 Good             7.119254
grd_9 Better           3.325710
wtrfrnt_YES            1.512626
bdrm__2                1.519526
```

In [44]:
```
#I will begin by dropping features with a V.I.F of two digits:

vif_ser = pd.Series(vif, index=features_3.columns, name="Variance Inflation
elim_lst = list(vif_ser[vif_ser>=10].index)
features_4 = features_3.drop(elim_lst,axis=1)
```

In [45]:
```
features_4_df = sm.add_constant(features_4)
model_features_4 = sm.OLS(y_data,features_4_df).fit()
model_features_4.summary()
```

Out[45]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.657
Model:	OLS	**Adj. R-squared:**	0.656
Method:	Least Squares	**F-statistic:**	801.2
Date:	Tue, 01 Feb 2022	**Prob (F-statistic):**	0.00
Time:	15:27:48	**Log-Likelihood:**	-5189.6
No. Observations:	21428	**AIC:**	1.048e+04
Df Residuals:	21376	**BIC:**	1.090e+04
Df Model:	51		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	12.8735	0.007	1728.201	0.000	12.859	12.888

In [46]:
```
#Removing any additonal features that have shown up with p-values over .05:
features_5 = features_4.drop(['view_missing','bdrm__8','bdrm__9','bthrm__5.
                              'flr__3.0'],axis=1)
```

In [47]:
```
# Using V.I.F to identify the features that are most highly correlated with

from statsmodels.stats.outliers_influence import variance_inflation_factor
vif = [variance_inflation_factor(features_5.values, i) for i in range(featu
pd.Series(vif, index=features_5.columns, name="Variance Inflation Factor")
```

Out[47]:
```
view_EXCELLENT          1.549463
view_FAIR               1.026085
view_GOOD               1.062907
cond_Fair               1.033617
cond_Good               1.521771
cond_Very Good          1.172310
grd_11 Excellent        1.248634
grd_12 Luxury           1.154111
grd_13 Mansion          1.254129
grd_4 Low               1.008858
grd_5 Fair              1.074543
grd_6 Low Average       1.462768
grd_8 Good              1.577017
grd_9 Better            1.449332
wtrfrnt_YES             1.507551
bdrm__2                 1.511366
bdrm__4                 1.770995
bdrm__5                 1.285252
bdrm__6                 1.101650
```

In [48]:
```
features_5_df = sm.add_constant(features_5)
model_features_5 = sm.OLS(y_data,features_5_df).fit()
model_features_5.summary()
```

Out[48]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.656
Model:	OLS	**Adj. R-squared:**	0.656
Method:	Least Squares	**F-statistic:**	907.4
Date:	Tue, 01 Feb 2022	**Prob (F-statistic):**	0.00
Time:	15:27:50	**Log-Likelihood:**	-5196.8
No. Observations:	21428	**AIC:**	1.049e+04
Df Residuals:	21382	**BIC:**	1.085e+04
Df Model:	45		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	12.8777	0.007	1789.974	0.000	12.864	12.892
view_EXCELLENT	0.4491	0.022	20.731	0.000	0.407	0.492
view_FAIR	0.2748	0.017	15.958	0.000	0.241	0.309
view_GOOD	0.3290	0.014	23.187	0.000	0.301	0.357
cond_Fair	-0.1126	0.024	-4.674	0.000	-0.160	-0.065
cond_Good	0.0935	0.005	18.122	0.000	0.083	0.104
cond_Very Good	0.1801	0.008	22.008	0.000	0.164	0.196
grd_11 Excellent	0.5794	0.018	33.051	0.000	0.545	0.614
grd_12 Luxury	0.7869	0.035	22.248	0.000	0.718	0.856
grd_13 Mansion	1.1282	0.096	11.765	0.000	0.940	1.316
grd_4 Low	-0.5897	0.060	-9.868	0.000	-0.707	-0.473
grd_5 Fair	-0.4066	0.021	-19.716	0.000	-0.447	-0.366
grd_6 Low Average	-0.2241	0.008	-27.016	0.000	-0.240	-0.208
grd_8 Good	0.1112	0.005	20.630	0.000	0.101	0.122
grd_9 Better	0.3141	0.008	41.622	0.000	0.299	0.329
wtrfrnt_YES	0.4002	0.031	12.714	0.000	0.339	0.462
bdrm__2	0.0261	0.007	3.594	0.000	0.012	0.040
bdrm__4	0.0760	0.005	14.613	0.000	0.066	0.086
bdrm__5	0.0922	0.009	10.324	0.000	0.075	0.110
bdrm__6	0.0434	0.020	2.197	0.028	0.005	0.082
bthrm__1.0	-0.2089	0.009	-23.280	0.000	-0.227	-0.191

bthrm__1.5	-0.1609	0.010	-16.041	0.000	-0.181	-0.141
bthrm__1.75	-0.0880	0.008	-10.821	0.000	-0.104	-0.072
bthrm__2.0	-0.0753	0.009	-8.343	0.000	-0.093	-0.058
bthrm__2.25	-0.0343	0.008	-4.161	0.000	-0.050	-0.018
bthrm__2.75	0.0756	0.010	7.418	0.000	0.056	0.096
bthrm__3.0	0.1147	0.012	9.328	0.000	0.091	0.139
bthrm__3.25	0.2693	0.014	19.554	0.000	0.242	0.296
bthrm__3.5	0.2830	0.013	22.360	0.000	0.258	0.308
bthrm__3.75	0.4121	0.026	15.811	0.000	0.361	0.463
bthrm__4.0	0.4193	0.028	15.008	0.000	0.365	0.474
bthrm__4.25	0.5334	0.036	14.814	0.000	0.463	0.604
bthrm__4.5	0.4171	0.033	12.751	0.000	0.353	0.481
bthrm__4.75	0.5971	0.066	9.108	0.000	0.469	0.726
bthrm__5.0	0.4942	0.068	7.216	0.000	0.360	0.628
bthrm__5.25	0.5476	0.087	6.324	0.000	0.378	0.717
bthrm__5.5	0.6505	0.100	6.476	0.000	0.454	0.847
bthrm__6.0	0.7073	0.130	5.456	0.000	0.453	0.961
bthrm__6.25	0.5680	0.224	2.534	0.011	0.129	1.007
bthrm__7.75	0.9592	0.324	2.957	0.003	0.323	1.595
bthrm__8.0	0.5088	0.225	2.261	0.024	0.068	0.950
flr__1.5	0.1560	0.008	19.960	0.000	0.141	0.171
flr__2.0	0.0954	0.006	16.180	0.000	0.084	0.107
flr__2.5	0.3229	0.025	12.884	0.000	0.274	0.372
lat	0.2210	0.002	101.018	0.000	0.217	0.225
log_sqft_lot	0.0531	0.002	23.346	0.000	0.049	0.058

Omnibus:	762.776	**Durbin-Watson:**	1.280
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	1154.282
Skew:	0.345	**Prob(JB):**	2.24e-251
Kurtosis:	3.903	**Cond. No.**	196.

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

Given that all the V.I.F(s) are under five, I will look at high multi-colinearity pair-wise; paying particular attention to those with the highest correlations with price,since multi- colinearity between these would create much greater (unexpected) fluctuations in the dependent

variable

In [49]:
```
#check to see how features correlate with 'price':

pd.set_option('display.max_rows', None)
features_5_corr_mtx = pd.concat([y_data,features_5],axis=1)
features_5_corr_mtx_abs = features_5_corr_mtx.corr().price.apply(lambda x:
features_5_corr_mtx_abs.sort_values(ascending=False).head(20)
```

Out[49]:
```
price                1.000000
lat                  0.450110
bthrm__1.0           0.339781
grd_9 Better         0.310535
grd_6 Low Average    0.309619
grd_11 Excellent     0.279746
flr__2.0             0.278656
bdrm__4              0.226270
view_EXCELLENT       0.223593
bthrm__3.5           0.208360
bthrm__3.25          0.190208
bdrm__5              0.180959
bdrm__2              0.180730
grd_12 Luxury        0.179048
view_GOOD            0.173134
wtrfrnt_YES          0.170171
grd_5 Fair           0.146765
log_sqft_lot         0.136703
bthrm__3.75          0.131357
```

In [50]:
```
pr_ws = features_5.corr().abs().stack().reset_index().sort_values(0, ascend

pr_ws['pairs'] = list(zip(pr_ws.level_0, pr_ws.level_1))

pr_ws.set_index(['pairs'], inplace = True)

pr_ws.drop(columns=['level_1', 'level_0'], inplace = True)

# cc for correlation coefficient
pr_ws.columns = ['cc']

pr_ws.drop_duplicates(inplace=True)

pr_ws[(pr_ws.cc>.60) & (pr_ws.cc<1)]
```

Out[50]:

pairs	cc

In [51]:
```
#I created an alternate model 5a, since I noticed that 'lat' has a high imp
#the normality of the residual distribution:

features_5a = features_5.drop('lat',axis=1)
features_5a_df = sm.add_constant(features_5a)
model_features_5a = sm.OLS(y_data,features_5a_df).fit()
model_features_5a.summary()
```

Out[51]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.492
Model:	OLS	**Adj. R-squared:**	0.491
Method:	Least Squares	**F-statistic:**	471.2
Date:	Tue, 01 Feb 2022	**Prob (F-statistic):**	0.00
Time:	15:27:50	**Log-Likelihood:**	-9377.2
No. Observations:	21428	**AIC:**	1.884e+04
Df Residuals:	21383	**BIC:**	1.920e+04
Df Model:	44		
Covariance Type:	nonrobust		

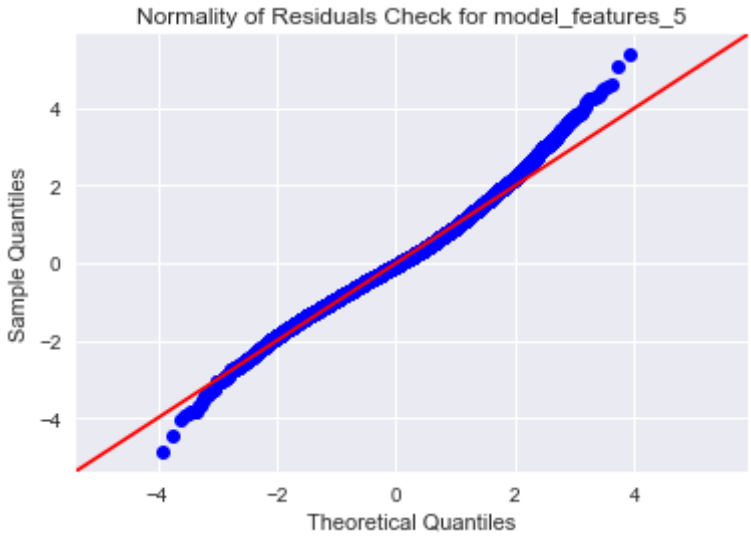
	coef	std err	t	P>|t|	[0.025	0.975]
const	12.8555	0.009	1470.806	0.000	12.838	12.873

The JB score drops greatly once the 'lat' feature is removed, but so does the R-squared(about 15 points). I will try to see if anything can be done to more closely approximate a normal residual distribution, without removing this feature.

4 Assumptions of Multiple Linear Regression

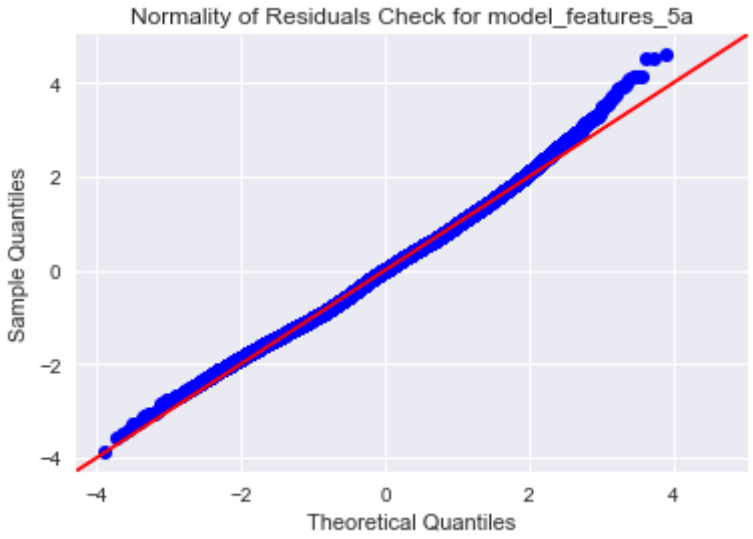
4.1 I will use some visulizations to review residual normality, and homosdeskacity :

In [52]:
```
fig = sm.graphics.qqplot(model_features_5.resid, dist=stats.norm, line='45'
plt.title('Normality of Residuals Check for model_features_5');
```



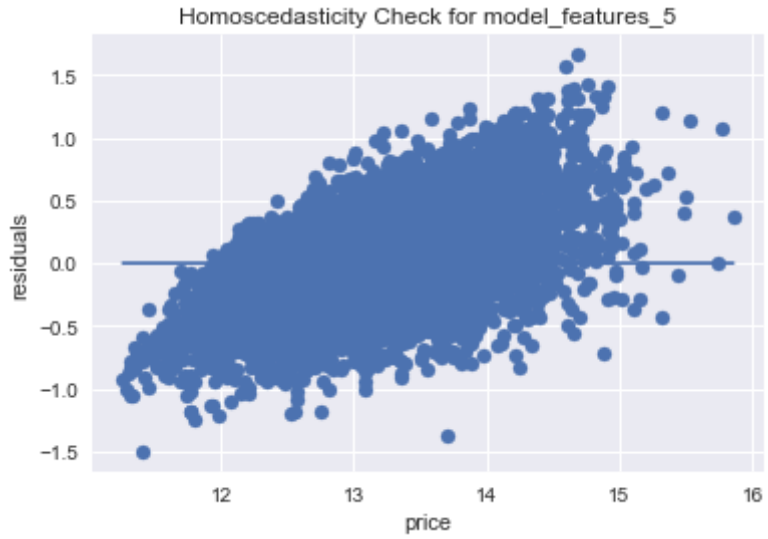
Above is a qq plot of comparing the distribution of the residuals in 'model_features_5' to a normal distribution. This model includes the 'cb_rt_lat' feature.

In [53]:
```
fig = sm.graphics.qqplot(model_features_5a.resid, dist=stats.norm, line='45
plt.title('Normality of Residuals Check for model_features_5a');
```



Above is a qq plot of comparing the distribution of the residuals in 'model_features_5a' to a normal distribution. This model dropped the 'cb_rt_lat' feature.

In [54]:
```
plt.scatter(y_data, model_features_5.resid)
plt.plot(y_data, [0 for i in range (len(features_5))]);
plt.xlabel("price")
plt.ylabel("residuals")
plt.title('Homoscedasticity Check for model_features_5');
```



Above is a scatter plot used to visually inspect for heteroscedasticity.

I see that there is some change in the variance on both extremes of the plot. The goal is to trim the ends of the plot, and retrain the model on the new data set. I will check to see if this also helps with the normality of the residual distribution. To this end, I will create a new dataframe that includes, price, residuals,predicted prices, as well as their corresponding z-scores.

4.2 Creating New Dataframe with the latest feature set (full_feature_set):

Dataframe includes price, residuals, predicted variables, and their respective z-scores

In [55]:
```
#creating a 'predicted_list' to add to the prices predicted by the model to
predicted_list = list(model_features_5.predict(features_5_df))

#joining 'price' with the independent variables:
full_feature_set = pd.concat([y_data,features_5_df],axis=1)

#adding predicted prices and residuals to the dataframe:
full_feature_set['preds'] = predicted_list
full_feature_set['resids'] = model_features_5.resid
full_feature_set.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21428 entries, 15279 to 7245
Data columns (total 49 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   price               21428 non-null  float64
 1   const               21428 non-null  float64
 2   view_EXCELLENT      21428 non-null  uint8
 3   view_FAIR           21428 non-null  uint8
 4   view_GOOD           21428 non-null  uint8
 5   cond_Fair           21428 non-null  uint8
 6   cond_Good           21428 non-null  uint8
 7   cond_Very Good      21428 non-null  uint8
 8   grd_11 Excellent    21428 non-null  uint8
 9   grd_12 Luxury       21428 non-null  uint8
 10  grd_13 Mansion      21428 non-null  uint8
 11  grd_4 Low           21428 non-null  uint8
 12  grd_5 Fair          21428 non-null  uint8
 13  grd_6 Low Average   21428 non-null  uint8
```

In [56]:
```
#Creating the z-score columns, and initially assigning them the values of t

full_feature_set['z_score_resids'] = full_feature_set['resids']
full_feature_set['z_score_price'] = full_feature_set['price']
full_feature_set['z_score_preds'] = full_feature_set['preds']
```

In [57]:
```
#Creating variables representing the mean and standard deviation:
resids_mean = full_feature_set.resids.mean()
resids_std = full_feature_set.resids.std()
price_mean = full_feature_set.price.mean()
price_std = full_feature_set.price.std()
preds_mean = full_feature_set.preds.mean()
preds_std = full_feature_set.preds.std()


#Applying their respective z-scores to the z_score columns:
full_feature_set['z_score_resids']=  full_feature_set['z_score_resids'].app
full_feature_set['z_score_price'] =  full_feature_set['z_score_price'].appl
full_feature_set['z_score_preds'] =  full_feature_set['z_score_preds'].appl

full_feature_set.head()
```

Out[57]:

	price	const	view_EXCELLENT	view_FAIR	view_GOOD	cond_Fair	cond_Good	cond_V Gc
15279	11.264464	1.0	0	0	0	0	0	
465	11.289782	1.0	0	0	0	1	0	
16184	11.302204	1.0	0	0	0	0	0	
8267	11.314475	1.0	0	0	0	0	0	
2139	11.320554	1.0	0	0	0	1	0	

5 rows × 52 columns

4.2.1 Removing some residuals to improve Heteroscedasticity:

In [58]:
```
#Creating a sub-dataframe with the 'price' greater than 12 , and less than
full_feature_set = full_feature_set.loc[(full_feature_set['price'] > 12) &

#splitting data set so I can create a new model:
y_data_II = full_feature_set['price']
new_df_II = full_feature_set.drop(['price','preds','resids','z_score_price'
new_df_II.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 20931 entries, 16893 to 19513
Data columns (total 46 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   const               20931 non-null  float64
 1   view_EXCELLENT      20931 non-null  uint8
 2   view_FAIR           20931 non-null  uint8
 3   view_GOOD           20931 non-null  uint8
 4   cond_Fair           20931 non-null  uint8
 5   cond_Good           20931 non-null  uint8
 6   cond_Very Good      20931 non-null  uint8
 7   grd_11 Excellent    20931 non-null  uint8
 8   grd_12 Luxury       20931 non-null  uint8
 9   grd_13 Mansion      20931 non-null  uint8
 10  grd_4 Low           20931 non-null  uint8
 11  grd_5 Fair          20931 non-null  uint8
 12  grd_6 Low Average   20931 non-null  uint8
 13  grd_8 Good          20931 non-null  uint8
```

In [59]:
```
model_features_6 = sm.OLS(y_data_II,new_df_II).fit()
model_features_6.summary()
```

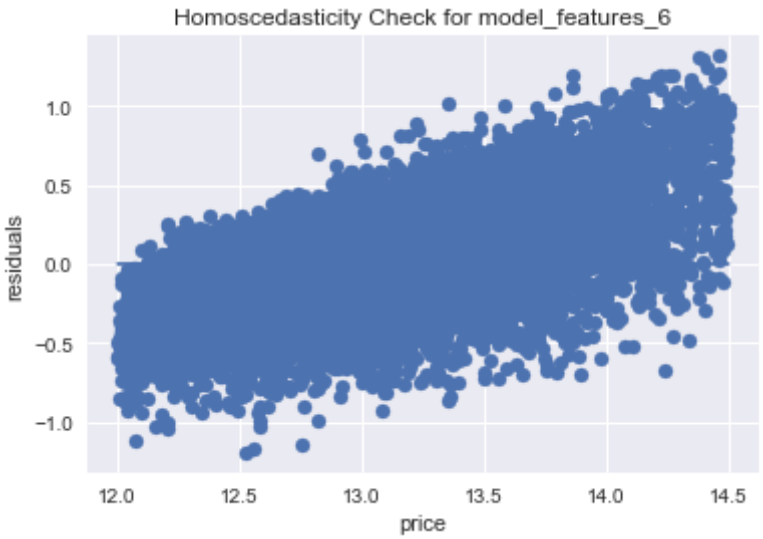
Out[59]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.624
Model:	OLS	**Adj. R-squared:**	0.623
Method:	Least Squares	**F-statistic:**	824.2
Date:	Tue, 01 Feb 2022	**Prob (F-statistic):**	0.00
Time:	15:27:50	**Log-Likelihood:**	-4143.3
No. Observations:	20931	**AIC:**	8373.
Df Residuals:	20888	**BIC:**	8714.
Df Model:	42		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	12.8784	0.007	1858.275	0.000	12.865	12.892

In [60]:
```
plt.scatter(y_data_II,model_features_6.resid)
plt.plot(y_data_II, [0 for i in range (len(new_df_II))]);
plt.xlabel("price")
plt.ylabel("residuals")
plt.title('Homoscedasticity Check for model_features_6');
```



The variance in the scatter plot above looks pretty uniform.

4.2.2 Rechecking for Multi-colinearity

After I removed some of the residuals, I noticed that the condition number went up very sharply. I will recheck the V.I.F scores, and the correlation matrix for this new dataframe.

In [61]:
```
# Using V.I.F to identify the features that are most highly correlated with

from statsmodels.stats.outliers_influence import variance_inflation_factor
vif = [variance_inflation_factor(new_df_II.values, i) for i in range(new_df
pd.Series(vif, index=new_df_II.columns, name="Variance Inflation Factor")
```

Out[61]:
```
const                11.532904
view_EXCELLENT        1.398076
view_FAIR             1.009699
view_GOOD             1.037636
cond_Fair             1.019516
cond_Good             1.169195
cond_Very Good        1.101667
grd_11 Excellent      1.178669
grd_12 Luxury         1.096675
grd_13 Mansion        1.002153
grd_4 Low             1.005117
grd_5 Fair            1.046502
grd_6 Low Average     1.304920
grd_8 Good            1.313048
grd_9 Better          1.361722
wtrfrnt_YES           1.385942
bdrm__2               1.318744
bdrm__4               1.313722
bdrm__5               1.212367
```

In [62]:
```
pd.set_option('display.max_columns', None)
full_feature_set.corr()
```

Out[62]:

	price	const	view_EXCELLENT	view_FAIR	view_GOOD	cond_Fair	cond_Goo
price	1.000000	NaN	0.169607	0.093125	0.167505	-0.067977	-0.04343
const	NaN	NaN	NaN	NaN	NaN	NaN	Na
view_EXCELLENT	0.169607	NaN	1.000000	-0.013607	-0.016738	-0.009164	0.01495
view_FAIR	0.093125	NaN	-0.013607	1.000000	-0.018917	-0.005641	0.01767
view_GOOD	0.167505	NaN	-0.016738	-0.018917	1.000000	-0.005013	0.01363
cond_Fair	-0.067977	NaN	-0.009164	-0.005641	-0.005013	1.000000	-0.05017
cond_Good	-0.043439	NaN	0.014950	0.017676	0.013638	-0.050171	1.00000
cond_Very Good	0.055165	NaN	0.020080	0.010029	0.002779	-0.024481	-0.17607
grd_11 Excellent	0.239948	NaN	0.044235	0.019936	0.075727	-0.010341	-0.04153
grd_12 Luxury	0.126447	NaN	0.055228	-0.006307	0.049101	-0.004248	-0.01774

4.2.3 Dropping some more features, and creating a new model:

Trimming the residuals appears to have produced some features with V.I.F values of 'Nan', and some with p-values over .05 .

In [63]:
```
# Removing features with pearson coefficient equal to Nan, and feature with
y_data_III = y_data_II
new_df_III = new_df_II.drop(['bthrm__6.0','bthrm__6.25','bthrm__7.75','bthr
new_df_III.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 20931 entries, 16893 to 19513
Data columns (total 42 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   const               20931 non-null  float64
 1   view_EXCELLENT      20931 non-null  uint8
 2   view_FAIR           20931 non-null  uint8
 3   view_GOOD           20931 non-null  uint8
 4   cond_Fair           20931 non-null  uint8
 5   cond_Good           20931 non-null  uint8
 6   cond_Very Good      20931 non-null  uint8
 7   grd_11 Excellent    20931 non-null  uint8
 8   grd_12 Luxury       20931 non-null  uint8
 9   grd_13 Mansion      20931 non-null  uint8
 10  grd_4 Low           20931 non-null  uint8
 11  grd_5 Fair          20931 non-null  uint8
 12  grd_6 Low Average   20931 non-null  uint8
 13  grd_8 Good          20931 non-null  uint8
```

In [64]:
```
model_features_7 = sm.OLS(y_data_III,new_df_III).fit()
model_features_7.summary()
```

Out[64]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.624
Model:	OLS	**Adj. R-squared:**	0.623
Method:	Least Squares	**F-statistic:**	844.3
Date:	Tue, 01 Feb 2022	**Prob (F-statistic):**	0.00
Time:	15:27:52	**Log-Likelihood:**	-4143.9
No. Observations:	20931	**AIC:**	8372.
Df Residuals:	20889	**BIC:**	8706.
Df Model:	41		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	12.8784	0.007	1858.060	0.000	12.865	12.892

4.2.4 Re-checking for normality of the distribtion of the residuals:

Removing the features with 'Nan' correlations brought the Cond. No. back down, but the Jarque-Bera score from the new model is still pretty elevated. I will try to improve the kurtosis and skew, so that they more closely resemble that of a normal distribution.

In [65]:
```
fig = sm.graphics.qqplot(model_features_7.resid, dist=stats.norm, line='45'
plt.title('Normality of Residuals Check for model_features_7');
```

Above is a qq plot of comparing the distribution of the residuals in 'model_features_7' to a normal distribution. The JB score is still pretty high. I will create a new feature set, with the intention of trimming some of the residuals at the uppper end, to more closely approximate a normal distribution.

4.3 Creating New Dataframe with the latest feature set (full_feature_set_II):

Dataframe includes price, residuals, predicted variables, and their respective z-scores

I will create a new feature set ,and sort it based on the residuals so that it coincides with the qq-plot.

In [66]:
```
#creating a 'predicted_list' to add to the prices predicted by the model to
predicted_list_II = list(model_features_7.predict(new_df_III))

#joining 'price' with the independent variables:
full_feature_set_II = pd.concat([y_data_III,new_df_III],axis=1)

#adding predicted prices and residuals to the dataframe:
full_feature_set_II['preds'] = predicted_list_II
full_feature_set_II['resids'] = model_features_7.resid
full_feature_set_II.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 20931 entries, 16893 to 19513
Data columns (total 45 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   price              20931 non-null  float64
 1   const              20931 non-null  float64
 2   view_EXCELLENT     20931 non-null  uint8
 3   view_FAIR          20931 non-null  uint8
 4   view_GOOD          20931 non-null  uint8
 5   cond_Fair          20931 non-null  uint8
 6   cond_Good          20931 non-null  uint8
 7   cond_Very Good     20931 non-null  uint8
 8   grd_11 Excellent   20931 non-null  uint8
 9   grd_12 Luxury      20931 non-null  uint8
 10  grd_13 Mansion     20931 non-null  uint8
 11  grd_4 Low          20931 non-null  uint8
 12  grd_5 Fair         20931 non-null  uint8
 13  grd_6 Low Average  20931 non-null  uint8
```

In [67]:
```
#Creating the z-score columns, and initially assigning them the values of t

full_feature_set_II['z_score_resids'] = full_feature_set_II['resids']
full_feature_set_II['z_score_price'] = full_feature_set_II['price']
full_feature_set_II['z_score_preds'] = full_feature_set_II['preds']
```

In [68]:
```
#Creating variables representing the mean and standard deviation:
resids_mean = full_feature_set_II.resids.mean()
resids_std = full_feature_set_II.resids.std()
price_mean = full_feature_set_II.price.mean()
price_std = full_feature_set_II.price.std()
preds_mean = full_feature_set_II.preds.mean()
preds_std = full_feature_set_II.preds.std()


#Applying their respective z-scores to the z_score columns:
full_feature_set_II['z_score_resids']=  full_feature_set_II['z_score_resids
full_feature_set_II['z_score_price'] =  full_feature_set_II['z_score_price'
full_feature_set_II['z_score_preds'] =  full_feature_set_II['z_score_preds'
```

In [69]:
```
#sorting dataframe based on residuals
full_feature_set_sorted = full_feature_set_II.sort_values(by=['resids'])

#Removing some residuals on the right end of the plot:
full_feature_set_sorted = full_feature_set_sorted.iloc[0:20590,:]

#splitting the data to create a new model:
y_data_IV = full_feature_set_sorted['price']
new_df_IV = full_feature_set_sorted.drop(['price','preds','resids','z_score
                                          'z_score_preds'],axis=1)
```

In [70]:
```
model_features_8 = sm.OLS(y_data_IV,new_df_IV).fit()
model_features_8.summary()
```

Out[70]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.653
Model:	OLS	**Adj. R-squared:**	0.652
Method:	Least Squares	**F-statistic:**	943.7
Date:	Tue, 01 Feb 2022	**Prob (F-statistic):**	0.00
Time:	15:27:52	**Log-Likelihood:**	-2586.6
No. Observations:	20590	**AIC:**	5257.
Df Residuals:	20548	**BIC:**	5590.
Df Model:	41		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	12.8553	0.007	1971.557	0.000	12.843	12.868
view_EXCELLENT	0.4225	0.021	20.154	0.000	0.381	0.464
view_FAIR	0.2550	0.016	16.168	0.000	0.224	0.286
view_GOOD	0.3205	0.013	24.471	0.000	0.295	0.346
cond_Fair	-0.0772	0.023	-3.305	0.001	-0.123	-0.031
cond_Good	0.0855	0.005	18.244	0.000	0.076	0.095
cond_Very Good	0.1658	0.007	22.293	0.000	0.151	0.180
grd_11 Excellent	0.5802	0.017	34.108	0.000	0.547	0.614
grd_12 Luxury	0.7530	0.039	19.542	0.000	0.677	0.828
grd_13 Mansion	1.1681	0.275	4.248	0.000	0.629	1.707
grd_4 Low	-0.3119	0.071	-4.385	0.000	-0.451	-0.172
grd_5 Fair	-0.2908	0.021	-13.877	0.000	-0.332	-0.250
grd_6 Low Average	-0.1990	0.008	-26.198	0.000	-0.214	-0.184
grd_8 Good	0.1329	0.005	27.317	0.000	0.123	0.142
grd_9 Better	0.3420	0.007	49.922	0.000	0.329	0.355
wtrfrnt_YES	0.3045	0.033	9.291	0.000	0.240	0.369
bdrm__2	0.0343	0.007	5.178	0.000	0.021	0.047
bdrm__4	0.0758	0.005	16.135	0.000	0.067	0.085
bdrm__5	0.0854	0.008	10.476	0.000	0.069	0.101
bdrm__6	0.0496	0.018	2.731	0.006	0.014	0.085
bthrm__1.0	-0.1775	0.008	-21.764	0.000	-0.194	-0.162
bthrm__1.5	-0.1482	0.009	-16.336	0.000	-0.166	-0.130

bthrm__1.75	-0.0742	0.007	-10.116	0.000	-0.089	-0.060
bthrm__2.0	-0.0651	0.008	-8.016	0.000	-0.081	-0.049
bthrm__2.25	-0.0408	0.007	-5.484	0.000	-0.055	-0.026
bthrm__2.75	0.0765	0.009	8.331	0.000	0.058	0.094
bthrm__3.0	0.1013	0.011	9.071	0.000	0.079	0.123
bthrm__3.25	0.2245	0.013	17.537	0.000	0.199	0.250
bthrm__3.5	0.2534	0.012	21.774	0.000	0.231	0.276
bthrm__3.75	0.3793	0.025	15.397	0.000	0.331	0.428
bthrm__4.0	0.3416	0.027	12.566	0.000	0.288	0.395
bthrm__4.25	0.4654	0.037	12.443	0.000	0.392	0.539
bthrm__4.5	0.3499	0.031	11.147	0.000	0.288	0.411
bthrm__4.75	0.2176	0.092	2.357	0.018	0.037	0.398
bthrm__5.0	0.2973	0.074	4.002	0.000	0.152	0.443
bthrm__5.25	0.4710	0.092	5.112	0.000	0.290	0.652
bthrm__5.5	0.5602	0.138	4.060	0.000	0.290	0.831
flr__1.5	0.1595	0.007	22.606	0.000	0.146	0.173
flr__2.0	0.0864	0.005	16.151	0.000	0.076	0.097
flr__2.5	0.2707	0.024	11.314	0.000	0.224	0.318
lat	0.2084	0.002	105.639	0.000	0.205	0.212
log_sqft_lot	0.0472	0.002	23.018	0.000	0.043	0.051

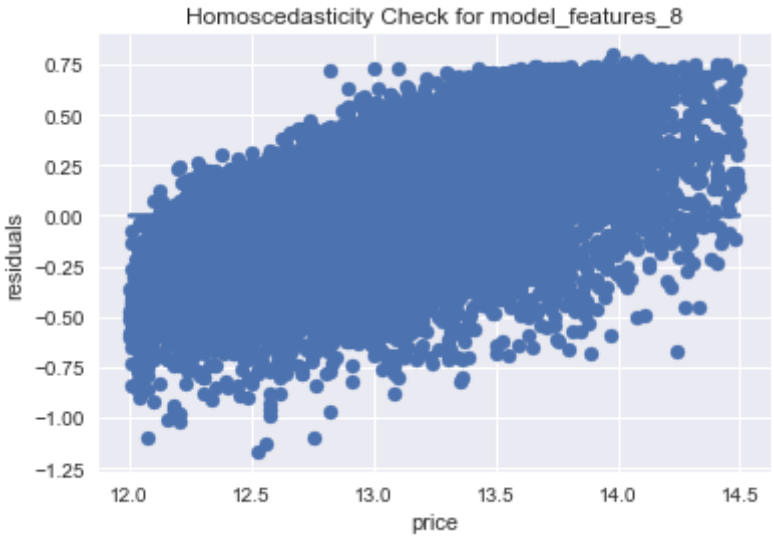
Omnibus:	64.244	**Durbin-Watson:**	0.005
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	64.770
Skew:	0.136	**Prob(JB):**	8.62e-15
Kurtosis:	3.042	**Cond. No.**	181.

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

With a slight skew, and the kurtosis close to 3, the distribution of the residuals is now much closer to that of a normal distribution.

4.3.1 Rechecking Homoscedasticity holds after removing residuals to improve normality:

In [71]:
```
plt.scatter(y_data_IV,model_features_8.resid)
plt.plot(y_data_IV, [0 for i in range (len(new_df_IV))]);
plt.xlabel("price")
plt.ylabel("residuals")
plt.title('Homoscedasticity Check for model_features_8');
```



Given that the variance in residuals is still stable, I will make this my final model, and formally confirm the four assumptions

4.4 Creating Final Model That is Scaled

In [72]:
```
final_features_scaled = full_feature_set_sorted
model_final_scaled = model_features_8
y_data_final_scaled = y_data_IV
X_data_final_scaled = new_df_IV
final_features_scaled.head()
```

Out[72]:

	price	const	view_EXCELLENT	view_FAIR	view_GOOD	cond_Fair	cond_Good	cond_V Gc
326	12.524435	1.0	0	0	0	0	0	
7090	12.560244	1.0	0	0	0	0	0	
19173	12.753037	1.0	0	0	0	0	0	
5861	12.072541	1.0	0	0	0	0	1	
10084	12.206073	1.0	0	0	0	0	0	

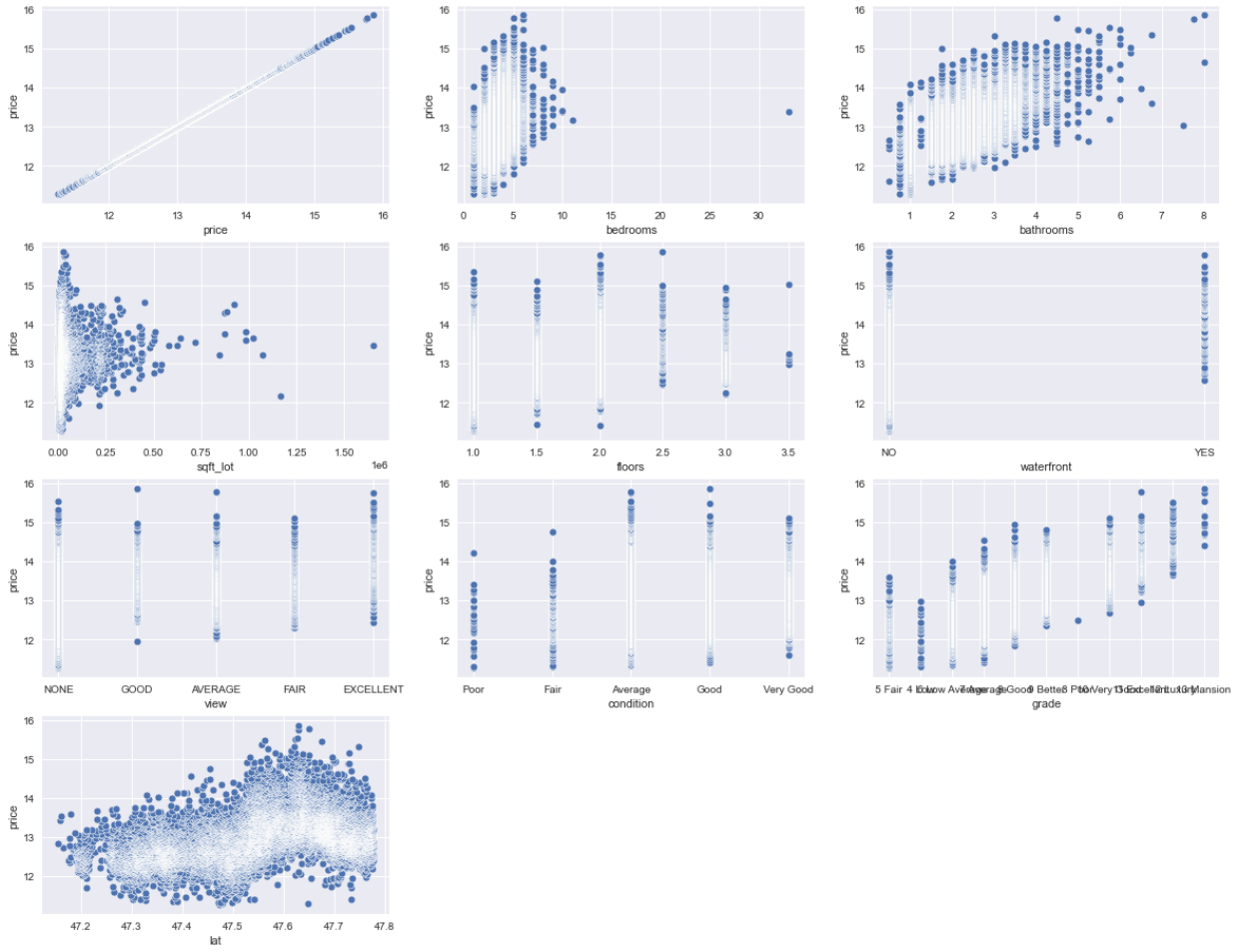
4.5 Formally Confirming the Four Assumptions.

4.5.1 Assumption of Variable linearity:

In [73]:
```
kc_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21428 entries, 15279 to 7245
Data columns (total 20 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   date           21428 non-null  object 
 1   price          21428 non-null  float64
 2   bedrooms       21428 non-null  int64  
 3   bathrooms      21428 non-null  float64
 4   sqft_living    21428 non-null  int64  
 5   sqft_lot       21428 non-null  int64  
 6   floors         21428 non-null  float64
 7   waterfront     19070 non-null  object 
 8   view           21367 non-null  object 
 9   condition      21428 non-null  object 
 10  grade          21428 non-null  object 
 11  sqft_above     21428 non-null  int64  
 12  sqft_basement  21428 non-null  object 
 13  yr_built       21428 non-null  int64  
 14  yr_renovated   17622 non-null  float64
 15  zipcode        21428 non-null  int64  
 16  lat            21428 non-null  float64
 17  long           21428 non-null  float64
 18  sqft_living15  21428 non-null  int64  
 19  sqft_lot15     21428 non-null  int64  
dtypes: float64(6), int64(8), object(6)
memory usage: 3.4+ MB
```

In [74]:
```
#Reviewing the scatter plots after the outliers are dropped

df_nums_non_null_only = kc_df.iloc[:,np.r_[1:4,5:11,16]]
df_nums_non_null_only_cols_list = list(df_nums_non_null_only.columns)

count=1
plt.subplots(figsize=(20, 20))
for i in df_nums_non_null_only.columns:
    plt.subplot(5,3,count)
    sns.scatterplot(df_nums_non_null_only[i],df_nums_non_null_only["price"]
    count+=1

plt.show()
```



These scatter plots are based on the original dataframe, after removing the outliers. Viewing the features that were ultimately chosen, including the discrete variables before they were separated, we can see the linear realtionship between the dependent (price), and the independent variables. Although, some of these correlations, are not perfectly linear, such as with the 'lat' feature, that has a downward trend toward the end of the plot, there is still a strong linear relationship that can be well explained by a linear model. This should satisfy the assumption of linearity.

4.5.2 Assumption of Variable Independence:

In [75]:
```
# Using V.I.F to identify the features that are most highly correlated with

from statsmodels.stats.outliers_influence import variance_inflation_factor
vif = [variance_inflation_factor(X_data_final_scaled.values, i) for i in ra
pd.Series(vif, index=X_data_final_scaled.columns, name="Variance Inflation
```

Out[75]:
```
const                11.605723
view_EXCELLENT        1.399282
view_FAIR             1.009735
view_GOOD             1.038161
cond_Fair             1.019606
cond_Good             1.169798
cond_Very Good        1.103227
grd_11 Excellent      1.190076
grd_12 Luxury         1.060081
grd_13 Mansion        1.002254
grd_4 Low             1.005205
grd_5 Fair            1.044900
grd_6 Low Average     1.302194
grd_8 Good            1.325622
grd_9 Better          1.380311
wtrfrnt_YES           1.388714
bdrm__2               1.320837
bdrm__4               1.312291
bdrm__5               1.207761
```

In [76]:
```
pr_ws = X_data_final_scaled.corr().abs().stack().reset_index().sort_values(

pr_ws['pairs'] = list(zip(pr_ws.level_0, pr_ws.level_1))

pr_ws.set_index(['pairs'], inplace = True)

pr_ws.drop(columns=['level_1', 'level_0'], inplace = True)

# cc for correlation coefficient
pr_ws.columns = ['cc']

pr_ws.drop_duplicates(inplace=True)

pr_ws[(pr_ws.cc>.50) & (pr_ws.cc<1)]
```

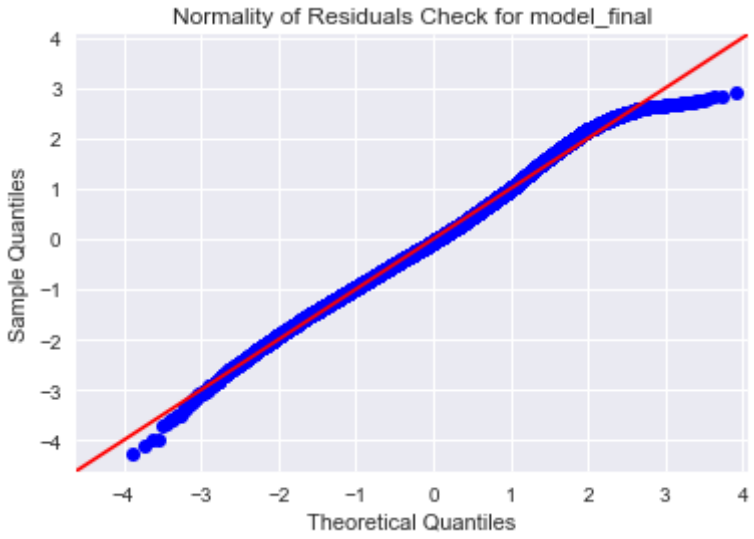
Out[76]:

	cc
pairs	
(wtrfrnt_YES, view_EXCELLENT)	0.521719

Given that all the V.I.F scores (with the exception of const) are well under 5, and the highest pairwise correlation is at .52, I think that the variables have an acceptable level of indepence amongst themselves, and that the assumption of variable indepence is satisfied.

4.5.3 Assumption of the Normal Distribution of Residuals:

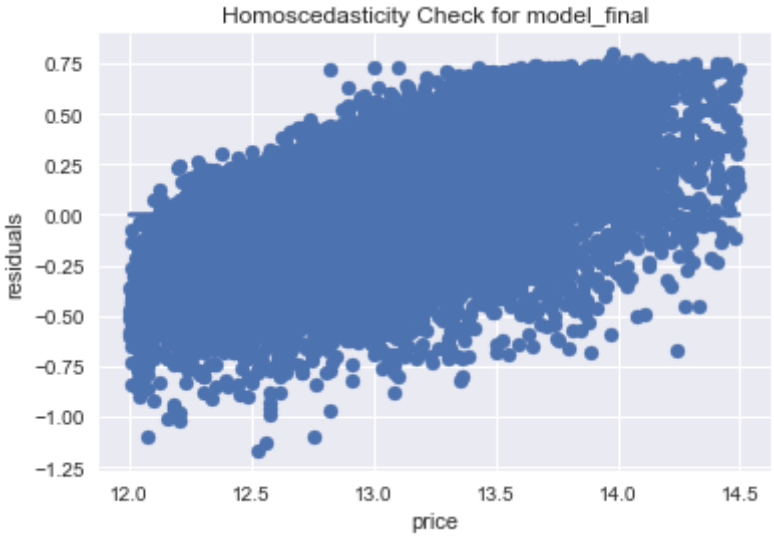
In [77]:
```
fig = sm.graphics.qqplot(model_final_scaled.resid, dist=stats.norm, line='4
plt.title('Normality of Residuals Check for model_final');
```



With a Jarque Bera score of 64.7, a slight skew of .136, and the Kurtosis nearly at 3, with a score of 3.033, I think this satisfies the assumption of normality.

4.5.4 Assumption of Homoscedasticity:

In [78]:
```
plt.scatter(y_data_final_scaled,model_final_scaled.resid)
plt.plot(y_data_final_scaled, [0 for i in range (len(X_data_final_scaled))]
plt.xlabel("price")
plt.ylabel("residuals")
plt.title('Homoscedasticity Check for model_final');
```



Although the value of the residuals increase with price, the variance at each price point appears pretty consistent. This should satisfy the assumption of Homoscedasticity.

With these four assumptions satisfied, I will create one more model based on my final model, but with no scale, for easier interpretation. To that end, I will create a full feature dataframe, including the observed price, predicted price, residuals, and their respective z-scores.

4.6 Creating a Final Model That Is Not Scaled

In [79]:
```
#creating dataframe with no scale from the dependent and independent variab
final_features_no_scale = pd.concat([y_data,X_data_preproc_cmplt_no_scale],


#dropping the columns that don't match up with the scaled feature set:
for col in final_features_no_scale.columns:
    if col not in final_features_scaled.columns:
        final_features_no_scale.drop(col,axis=1,inplace=True)


#matching up the rows with the scaled feature set:
final_features_no_scale = final_features_no_scale[final_features_no_scale.i


#Given the 'no_scale' dataframe is missing the 'const' column , but include
#opposite case, these two should have the same shape:
final_features_no_scale.shape == X_data_final_scaled.shape
```

Out[79]:
```
True
```

In [80]:
```
final_features_no_scale.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 20590 entries, 16893 to 16511
Data columns (total 42 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   price               20590 non-null  float64
 1   lat                 20590 non-null  float64
 2   view_EXCELLENT      20590 non-null  uint8
 3   view_FAIR           20590 non-null  uint8
 4   view_GOOD           20590 non-null  uint8
 5   cond_Fair           20590 non-null  uint8
 6   cond_Good           20590 non-null  uint8
 7   cond_Very Good      20590 non-null  uint8
 8   grd_11 Excellent    20590 non-null  uint8
 9   grd_12 Luxury       20590 non-null  uint8
 10  grd_13 Mansion      20590 non-null  uint8
 11  grd_4 Low           20590 non-null  uint8
 12  grd_5 Fair          20590 non-null  uint8
 13  grd_6 Low Average   20590 non-null  uint8
```

In [81]:
```
#splitting 'no_scale' dataframe:
X_data_final_no_scale = final_features_no_scale.drop('price',axis=1)
y_data_final_no_scale = final_features_no_scale['price']

#adding const:
X_data_final_no_scale    = sm.add_constant(X_data_final_no_scale)
final_features_no_scale = sm.add_constant(final_features_no_scale)

#creating model:
model_final_no_scale = sm.OLS(y_data_final_no_scale,X_data_final_no_scale).
model_final_no_scale.summary()
```

Out[81]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.653
Model:	OLS	**Adj. R-squared:**	0.652
Method:	Least Squares	**F-statistic:**	943.7
Date:	Tue, 01 Feb 2022	**Prob (F-statistic):**	0.00
Time:	15:27:55	**Log-Likelihood:**	-2586.6
No. Observations:	20590	**AIC:**	5257.
Df Residuals:	20548	**BIC:**	5590.
Df Model:	41		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	50.0728	0.680	86.021	0.000	60.405	57.741

In [82]:
```
final_features_no_scale_corr_mtx = final_features_no_scale.corr().price
final_features_no_scale_corr_mtx.sort_values(ascending=False)
#final_features_no_scale_corr_mtx
```

Out[82]:
```
price                1.000000
lat                  0.464374
grd_9 Better         0.351629
flr__2.0             0.269100
grd_11 Excellent     0.253280
bdrm__4              0.231661
bthrm__3.5           0.215018
bthrm__3.25          0.179699
view_EXCELLENT       0.174553
view_GOOD            0.173844
bdrm__5              0.162807
grd_12 Luxury        0.133379
grd_8 Good           0.131498
bthrm__2.75          0.130650
bthrm__3.75          0.127192
log_sqft_lot         0.119955
wtrfrnt_YES          0.111437
bthrm__4.0           0.109916
bthrm__3.0           0.105404
```

Everything in the summary matches that of the 'scaled' model, with the exception of the condition number, which I think is simply due to removing the scaling.

In [83]:
```
#creating a 'predicted_list' to add to the prices predicted by the model to
predicted_list_no_scale = list(model_final_no_scale.predict(X_data_final_no


#adding predicted prices and residuals to the dataframe:
final_features_no_scale['preds'] = predicted_list_no_scale
final_features_no_scale['resids'] = model_final_no_scale.resid


#Creating the z-score columns, and initially assigning them the values of t
final_features_no_scale['z_score_resids'] = final_features_no_scale['resids
final_features_no_scale['z_score_preds']  = final_features_no_scale['preds'
final_features_no_scale['z_score_price']  = final_features_no_scale['price'


#Creating variables representing the mean and standard deviation:
resids_mean = full_feature_set_II.resids.mean()
resids_std = full_feature_set_II.resids.std()
price_mean = full_feature_set_II.price.mean()
price_std = full_feature_set_II.price.std()
preds_mean = full_feature_set_II.preds.mean()
preds_std = full_feature_set_II.preds.std()


#Applying their respective z-scores to the z_score columns:
final_features_no_scale['z_score_resids'] = final_features_no_scale['z_scor
final_features_no_scale['z_score_price']  = final_features_no_scale['z_scor
final_features_no_scale['z_score_preds']  = final_features_no_scale['z_scor


#Sorting based on Residuals to align index with the 'scaled' feature set:
final_features_no_scale = final_features_no_scale.sort_values(by=['resids']


#The two feature sets should have the same shape:
final_features_scaled.shape == final_features_no_scale.shape
```

Out[83]:
```
True
```

In [84]:
```
#adding a column with the absolute bvalues of the residuals to sort from le
final_features_no_scale['abs_resids'] = final_features_no_scale['resids']
final_features_no_scale['abs_resids'] = final_features_no_scale['abs_resids
final_features_no_scale = final_features_no_scale.sort_values(by=['abs_resi
final_features_no_scale.head()
```

Out[84]:

	const	price	lat	view_EXCELLENT	view_FAIR	view_GOOD	cond_Fair	cond_Good
5446	1.0	14.392124	47.5303	0	0	0	0	0
17389	1.0	12.472276	47.3585	0	0	0	0	0
2835	1.0	13.102161	47.6601	0	0	0	0	0
21353	1.0	13.030138	47.4917	0	0	0	0	0
7284	1.0	12.774223	47.5033	0	0	0	0	1

5 Renovation and Remodel Recommendations

5.1 Baseline Features:

Baseline model:
'const': 1
'lat': 47.56
'view': 'AVERAGE' (default)
'condition': 'AVERAGE' (default)
'grade': '10 Very Good' (default)
'waterfront': 'NO' (default)
'bedrooms': 1 (default)
'bathrooms': 0.5 (default)
'floors': 1 (default)
'log_sqft_lot': 9
'price' : 12.855537576059533 aprox.: 382903.231151

5.2 1. Add a Second Floor:

In [85]:
```
#Created a dataframe to a see how changes in a certain feature impact the p

house_features = [{'const':1, 'lat':47.56, 'view_EXCELLENT':0, 'view_FAIR':
       'cond_Fair':0, 'cond_Good':0, 'cond_Very Good':0, 'grd_11 Excellent'
       'grd_12 Luxury':0, 'grd_13 Mansion':0, 'grd_4 Low':0, 'grd_5 Fair':0
       'grd_6 Low Average':0, 'grd_8 Good':0, 'grd_9 Better':0, 'wtrfrnt_YE
       'bdrm__2':0, 'bdrm__4':0, 'bdrm__5':0, 'bdrm__6':0, 'bthrm__1.0':0,
       'bthrm__1.75':0, 'bthrm__2.0':0, 'bthrm__2.25':0, 'bthrm__2.75':0, '
       'bthrm__3.25':0, 'bthrm__3.5':0, 'bthrm__3.75':0, 'bthrm__4.0':0, 'b
       'bthrm__4.5':0, 'bthrm__4.75':0, 'bthrm__5.0':0, 'bthrm__5.25':0, 'b
       'flr__1.5':0, 'flr__2.0':1, 'flr__2.5':0, 'log_sqft_lot':9}]
```

In [86]:
```
house_features = pd.DataFrame(house_features)
house_features
```

Out[86]:

	const	lat	view_EXCELLENT	view_FAIR	view_GOOD	cond_Fair	cond_Good	cond_Very Good	gı Exc
0	1	47.56	0	0	0	0	0	0	

In [87]:
```
model_final_no_scale.predict(house_features)[0]
```

Out[87]: 12.941941308413988

In this case, we have a log-level regression, where the dependent variable 'price', is log transformed, and the independent variable 'flr__2.0' is not. It takes the following form:

ln y = b0 + b1(x) + E

where b0 is the constant, b1 is the slope coefficient, and E is the error term.

We can calculate the change in price, based on the change in x with the following formula:

%(change in y) = 100 * ((e^b1) - 1)

Based on this formula, with b1 = 0.0864, the change in y would be equal to 9.02423%. If we multiply 382,903.231151 by 1.0902423, we get 417,457.299407. If we compare this to the price predicted by the model in terms of an exponent, e^12.941941308413988 = 417458.872072. The log-level interpretation matches the predicted value with a precision of 99.99962328%.

5.3 2. Upgrading Building Grade to 'grd_11 Excellent:

In [88]:
```
#Created a dataframe to a see how changes in a certain feature impact the p

house_features = [{'const':1, 'lat':47.56, 'view_EXCELLENT':0, 'view_FAIR':
       'cond_Fair':0, 'cond_Good':0, 'cond_Very Good':0, 'grd_11 Excellent'
       'grd_12 Luxury':0, 'grd_13 Mansion':0, 'grd_4 Low':0, 'grd_5 Fair':0
       'grd_6 Low Average':0, 'grd_8 Good':0, 'grd_9 Better':0, 'wtrfrnt_YE
       'bdrm__2':0, 'bdrm__4':0, 'bdrm__5':0, 'bdrm__6':0, 'bthrm__1.0':0,
       'bthrm__1.75':0, 'bthrm__2.0':0, 'bthrm__2.25':0, 'bthrm__2.75':0, '
       'bthrm__3.25':0, 'bthrm__3.5':0, 'bthrm__3.75':0, 'bthrm__4.0':0, 'b
       'bthrm__4.5':0, 'bthrm__4.75':0, 'bthrm__5.0':0, 'bthrm__5.25':0, 'b
       'flr__1.5':0, 'flr__2.0':0, 'flr__2.5':0, 'log_sqft_lot':9}]
```

In [89]:
```
house_features = pd.DataFrame(house_features)
house_features
```

Out[89]:

	const	lat	view_EXCELLENT	view_FAIR	view_GOOD	cond_Fair	cond_Good	cond_Very Good	gr Exc
0	1	47.56	0	0	0	0	0	0	

In [90]:
```
model_final_no_scale.predict(house_features)[0]
```

Out[90]: 13.43576963632787

In this case, we have a log-level regression, where the dependent variable 'price', is log transformed, and the independent variable 'grd_11 Excellent' is not. It takes the following form:

ln y = b0 + b1(x) + E

where b0 is the constant, b1 is the slope coefficient, and E is the error term.

We can calculate the change in price, based on the change in x with the following formula:

%(change in y) = 100 * ((e^b1) - 1)

Based on this formula, with b1 = 0.5802, the change in y would be equal to 78.639567%. If we multiply 382,903.231151 by 1.78639567, we get 684,016.674157. If we compare this to the price predicted by the model in terms of an exponent, e^13.43576963632787 = 684,038.60586. The log-level interpretation matches the predicted value with a precision of 99.996793791%.

5.4 3. Upgrading to Four Bedrooms:

In [91]:
```
#Created a dataframe to a see how changes in a certain feature impact the p

house_features = [{'const':1, 'lat':47.56, 'view_EXCELLENT':0, 'view_FAIR':
       'cond_Fair':0, 'cond_Good':0, 'cond_Very Good':0, 'grd_11 Excellent'
       'grd_12 Luxury':0, 'grd_13 Mansion':0, 'grd_4 Low':0, 'grd_5 Fair':0
       'grd_6 Low Average':0, 'grd_8 Good':0, 'grd_9 Better':0, 'wtrfrnt_YE
       'bdrm__2':0, 'bdrm__4':1, 'bdrm__5':0, 'bdrm__6':0, 'bthrm__1.0':0,
       'bthrm__1.75':0, 'bthrm__2.0':0, 'bthrm__2.25':0, 'bthrm__2.75':0, '
       'bthrm__3.25':0, 'bthrm__3.5':0, 'bthrm__3.75':0, 'bthrm__4.0':0, 'b
       'bthrm__4.5':0, 'bthrm__4.75':0, 'bthrm__5.0':0, 'bthrm__5.25':0, 'b
       'flr__1.5':0, 'flr__2.0':0, 'flr__2.5':0, 'log_sqft_lot':9}]
```

In [92]:
```
house_features = pd.DataFrame(house_features)
house_features
```

Out[92]:

	const	lat	view_EXCELLENT	view_FAIR	view_GOOD	cond_Fair	cond_Good	cond_Very Good	gı Exc
0	1	47.56	0	0	0	0	0	0	

In [93]:
```
model_final_no_scale.predict(house_features)[0]
```

Out[93]: 12.931293218903669

In this case, we have a log-level regression, where the dependent variable 'price', is log transformed, and the independent variable 'bdrm__4' is not. It takes the following form:

ln y = b0 + b1(x) + E

where b0 is the constant, b1 is the slope coefficient, and E is the error term.

We can calculate the change in price, based on the change in x with the following formula:

%(change in y) = 100 * ((e^b1) - 1)

Based on this formula, with b1 = 0.0758, the change in y would be equal to 7.87468%. If we multiply 382,903.231151 by 1.0787468, we get 413,055.635314. If we compare this to the price predicted by the model in terms of an exponent, e^12.931293218903669 = 413,037.31498. The log-level interpretation matches the predicted value with a precision of 99.995564681%.

6 Project Conclusion: Main Take-aways

1. It is critical to have some understanding of the subject matter one is dealing with, in order to be able to select features, more effectively, and root out non-sensical results.
2. If the ordinal data in one's data set does not have a monotonic relationship, one may wind up with non-sensical results. For example, recommending a downgrade from 'grd_10 Very Good' to 'grd_9 Better', in order to increase the price of the home.

3. High multi-colinearity between features, may lead to unexpected results. For example, one may have two features that are highly correlated, where both have positive Pearson correlations, but one has a negative regression coefficient.
4. The features with the higher Perason correlation coefficient, do not necessarily have a higher regression coefficient. I found this particularly in the 'dummy' variables. I think this may have something to do with these variables not having an equal number of occurences, since only one can be chosen at a time, for each row of data.
5. I understand that the distribution of the residuals do not necessarily have to be perfectly normal, but I am not clear as to what is defined as 'normal enough'.
6. I may have been able to further improve the R-squared score by adding 'sqft_bsmnt' as a binary variable.
7. I originally planned to associate the zip codes with their corresponding average household incomes (per the U.S census bureau),then binning by income groups, and creating interactions, but ultimately decicded against it because it generated too many columns, and didn't lend itself to learning.
8. Log transforming the dependent variable can help approximate a normal residual distribution.
9. Remove rows with outliers in the dependent variable can help approximate a normal residual distribution.