Final Project Submission

Please fill out:

- Student name: David Schenck
- Student pace: Flex
- Scheduled project review date/time: June 27, 2023, 2 pm
- Instructor name: Morgan Jones
- Blog post URL: https://daviddata24.wordpress.com/2023/06/10/a-predictive-soccer-model-part-1/ (https://daviddata24.wordpress.com/2023/06/10/a-predictive-soccer-model-part-1/)

Project Description

Data: The data used in this project provides details on houses sold in King County, WA between June 10, 2021 and June 9,2022. The data includes information about the price, features of the home (such as square footage, number of rooms, condition, environoment).

Stakeholder: The stakeholder is a real estate agency in the King County area.

Business Problem: The real estate agency wants to be able to set realistic prices on homes being put on the market. Setting prices too low means they and the seller get less money and setting the prices too high means the houses will be more difficult to sell. The real estate agency also wants to be able to provide guidance to sellers about how they can raise the value of their home.

Method: I will create a model that will be able to estimate the value of a house based on what features it has. The model will be created by using linear regression.

Data Exploration

In [297]:
```
# Modules
import numpy as np
import pandas as pd
import scipy.stats as stats
import statsmodels.api as sm
import matplotlib.pyplot as plt
import seaborn as sns
plt.style.use('ggplot')
```

In [298]:

```
# Read in data
df = pd.read_csv('data/kc_house_data.csv')
df
```

Out[298]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	fl
0	7399300360	5/24/2022	675000.0	4	1.0	1180	7140	
1	8910500230	12/13/2021	920000.0	5	2.5	2770	6703	
2	1180000275	9/29/2021	311000.0	6	2.0	2880	6156	
3	1604601802	12/14/2021	775000.0	3	3.0	2160	1400	
4	8562780790	8/24/2021	592500.0	2	2.0	1120	758	
...	...	...	...	...	...	...	...	...
30150	7834800180	11/30/2021	1555000.0	5	2.0	1910	4000	
30151	194000695	6/16/2021	1313000.0	3	2.0	2020	5800	
30152	7960100080	5/27/2022	800000.0	3	2.0	1620	3600	
30153	2781280080	2/24/2022	775000.0	3	2.5	2570	2889	
30154	9557800100	4/29/2022	500000.0	3	1.5	1200	11058	

30155 rows × 25 columns

The data includes 30,155 records.

There is a mix of continuous numeric, discrete numeric, and categorical data.

The target variable is going to be the price.

In [299]:
```
# Look at columns and dtypes
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30155 entries, 0 to 30154
Data columns (total 25 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   id             30155 non-null  int64  
 1   date           30155 non-null  object 
 2   price          30155 non-null  float64
 3   bedrooms       30155 non-null  int64  
 4   bathrooms      30155 non-null  float64
 5   sqft_living    30155 non-null  int64  
 6   sqft_lot       30155 non-null  int64  
 7   floors         30155 non-null  float64
 8   waterfront     30155 non-null  object 
 9   greenbelt      30155 non-null  object 
 10  nuisance       30155 non-null  object 
 11  view           30155 non-null  object 
 12  condition      30155 non-null  object 
 13  grade          30155 non-null  object 
 14  heat_source    30123 non-null  object 
 15  sewer_system   30141 non-null  object 
 16  sqft_above     30155 non-null  int64  
 17  sqft_basement  30155 non-null  int64  
 18  sqft_garage    30155 non-null  int64  
 19  sqft_patio     30155 non-null  int64  
 20  yr_built       30155 non-null  int64  
 21  yr_renovated   30155 non-null  int64  
 22  address        30155 non-null  object 
 23  lat            30155 non-null  float64
 24  long           30155 non-null  float64
dtypes: float64(5), int64(10), object(10)
memory usage: 5.8+ MB
```

There are some columns with missing data:
heat_source: Missing 32
sewer_system: Missing 14

In [300]:
```
# Convert date to datetime
df.date = pd.to_datetime(df.date)
```

In [301]:
```
print(f"Earliest date: {df.date.min()}   Latest date: {df.date.max()}")
```

```
Earliest date: 2021-06-10 00:00:00   Latest date: 2022-06-09 00:00:00
```

The data represents one year from June 10, 2021 to June 9, 2022

In [302]:
```
# Look at value counts for each column
for col in df.columns:
    print('COLUMN:',col)
    print(df[col].value_counts())
    print()
```

```
COLUMN: id
1233100736    2
2026059180    1
1423089049    1
3892500020    1
7321900035    1
             ..
1946000070    1
1250201175    1
5422570190    1
691000015     1
9430110210    1
Name: id, Length: 30154, dtype: int64

COLUMN: date
2021-07-01    196
2021-08-02    186
2021-07-06    176
2021-06-23    176
2021-06-16    174
```

All of the column IDs are unique EXCEPT for 1233100736. Let's look those two records.

In [305]:
```
df[df['id'] == 1233100736]
```

Out[305]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors
4845	1233100736	2021-09-28	2600000.0	3	4.0	3500	8455	2.0
4846	1233100736	2021-09-28	2600000.0	3	4.0	3500	8455	2.0

2 rows × 25 columns

This is a repeat, so I will remove record 4846.

In [306]:
```
df.drop(index = 4846, inplace = True)
```

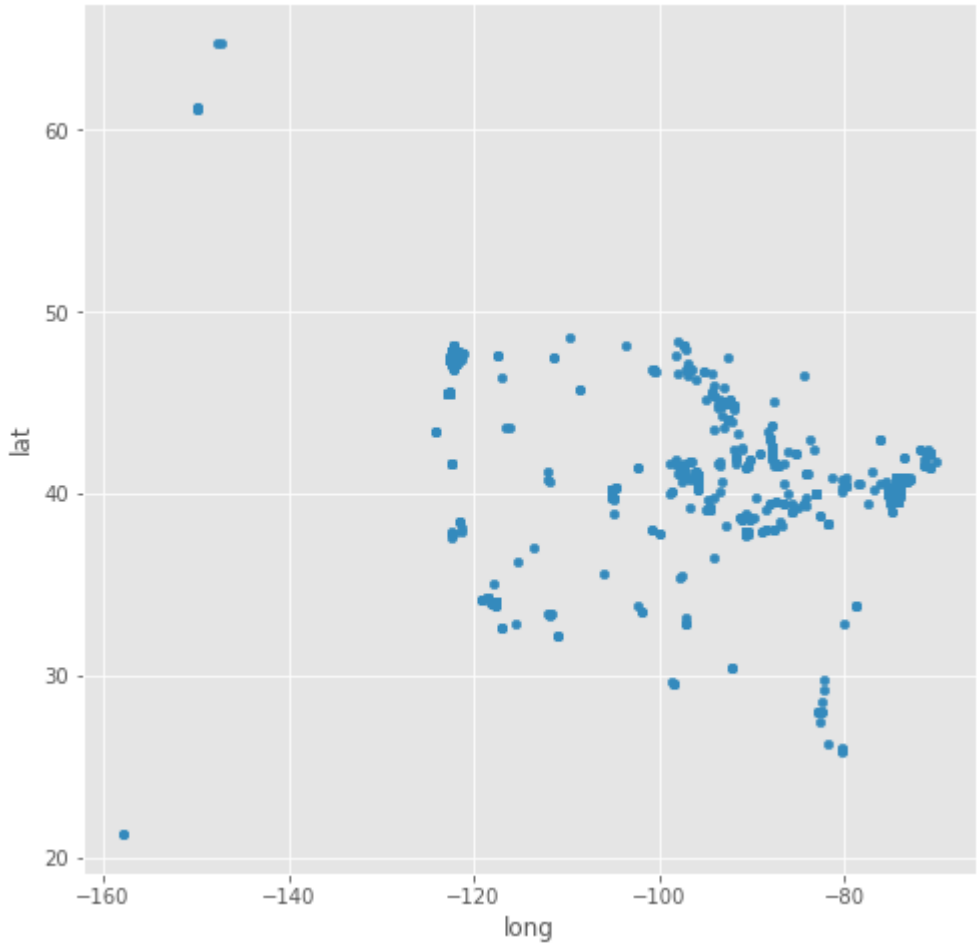
Location Data

Below is a plot of latitude and longitude. You can see a vague impression of the shape of the United States (with Alaska in the top left and Hawaii in the bottom left)

The data description says that all of the data is actually from King County in Washington state. All of the addresses, latitudes, and longitudes that are not in King County are errors. It is okay to keep the data in if I am not going to use geography for my analysis, but I should remove anything that isn't in Washington if I do want to use the location data.

In [315]:
```
# Look at locations of the homes in the data
fig, ax = plt.subplots( figsize = (8,8) )
df.plot.scatter(x = 'long', y = 'lat', ax = ax)
```

Out[315]: <AxesSubplot:xlabel='long', ylabel='lat'>



Below, I use the addresses to find the state and zip code of each record.

Again, the data is actually all from Washington state, but there are errors in finding latitude, longitude, and state for about 3% of the data.

In [316]:

```
def find_state(x):
    """
    This function takes the addresses in the data and finds the state.
    This is made straightforward because the last word of the state is
    always in position -4 after splitting the data.
    """
    last_word = x.split()[-4]
    word = x.split()[-4]
    if last_word == 'Jersey': word = 'New Jersey'
    if last_word == 'York': word = 'New York'
    if last_word == 'Island': word = 'Rhode Island'
    if last_word == 'Mexico': word = 'New Mexico'
    if last_word == 'Dakota':
        first_word = x.split()[-5]
        word = first_word+' '+last_word
    if last_word == 'Carolina':
        first_word = x.split()[-5]
        word = first_word+' '+last_word
    if last_word == 'Virginia':
        first_word = x.split()[-5]
        if first_word == 'West':
            word = 'West Virginia'
        else:
            word = 'Virginia'
    return word

def find_zip(x):
    """
    This function finds the zip code from the addresses in the data.
    """
    zip_code = x.split()[-3][:-1]
    return int(zip_code)

df['state'] = df['address'].apply(find_state)
df['zip'] = df['address'].apply(find_zip)
```

In [9]:
```
df['state'].value_counts()
```

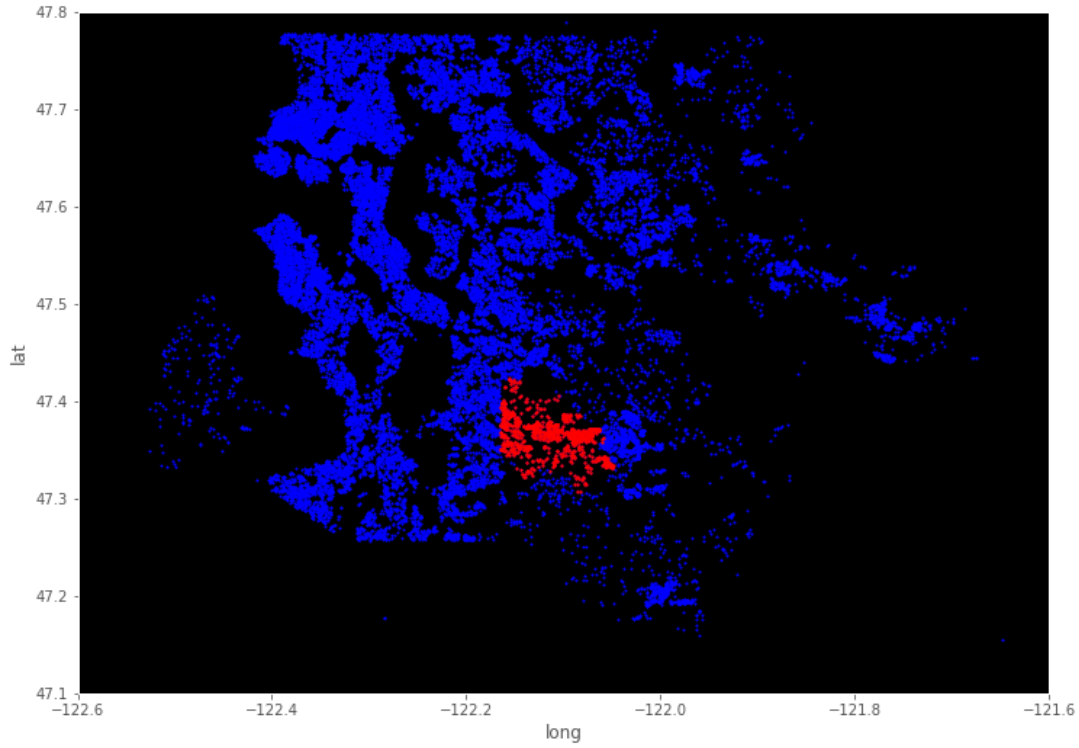
Out[9]:
```
Washington        29245
Nebraska            159
New Jersey           79
California           77
New York             66
Minnesota            64
Missouri             61
Wisconsin            51
Illinois             36
Pennsylvania         34
Massachusetts        30
Indiana              30
Oregon               29
Colorado             28
Ohio                 27
North Dakota         21
Iowa                 18
Florida              18
Texas                12
Kansas               11
Arizona               8
Michigan              8
Alaska                7
West Virginia         6
Montana               5
South Carolina        5
Utah                  4
Rhode Island          3
Louisiana             3
Hawaii                2
Idaho                 2
Oklahoma              2
New Mexico            1
Arkansas              1
Nevada                1
Maryland              1
Name: state, dtype: int64
```

In [317]:
```
df[df['state']=='Washington']['zip'].value_counts()[0:10]
```

Out[317]:
```
98042    992
98038    858
98103    761
98115    761
98117    748
98023    695
98034    691
98058    683
98133    664
98001    623
Name: zip, dtype: int64
```

I used the cell below to help figure out which zip codes have the correct geographical information

In [318]:
```
fig, ax = plt.subplots(figsize = (12,9))
df.plot.scatter(x = 'long', y = 'lat', ax = ax, color = (0.0,0.0,1.0,1.(
df[df['zip'] == 98042].plot.scatter(x = 'long', y = 'lat', ax = ax, col(
ax.set_xbound([-122.6,-121.6])
ax.set_ybound([47.1,47.8])
ax.set_facecolor((0.0, 0.0, 0.0))
plt.gca().set_aspect('equal')
plt.grid(False)
```



Below, I assign each zip code to a unique integer ID. All of the records with incorrect geographic information will be assigned the same ID. By plotting the points on the map, I found that any zip code with fewer than 20 records is an incorrect zip code. There are also some incorrect ZIP codes with more than 20 records, but those are easy to find because they have the wrong state (not Washington)

It is easiest to have the records with incorrect zip codes assigned to 0. I will then ignore these ZIP codes when creating the final dataset for the model.

In [322]:
```
# Convert each zip code to an integer ID
zip_ids = {}
n = 1
for zip_code, count in zip(df['zip'].value_counts().index,df['zip'].valu
    if count < 20:
        zip_ids[zip_code] = 0
    else:
        zip_ids[zip_code] = n
        n += 1

df['zip_id'] = df['zip'].replace(zip_ids)

#The lines below were used before I decided to remove data with incorrec
"""
# Shift the zip_id by one
df['zip_id'] = df['zip_id'] - 1
# Put incorrect records at the end
df['zip_id'] = df['zip_id'].apply(lambda x: df['zip_id'].max() + 1 if x
"""
```

Out[322]:
```
"\n# Shift the zip_id by one\ndf['zip_id'] = df['zip_id'] - 1\n# Put i
ncorrect records at the end\ndf['zip_id'] = df['zip_id'].apply(lambda
x: df['zip_id'].max() + 1 if x == -1 else x)\n"
```

Below, I remove any data that includes incorrect geographic information.

In [325]:
```
# Leave out incorrect geographic information
df = df[(df['state'] == 'Washington') & (df['zip_id'] != 0)]
len(df)
```

Out[325]:
```
29186
```

The dataset still includes 29,186 records

In [329]:
```
# Look at repeat addresses still in the data
df['address'].value_counts()
```

Out[329]:
```
Delridge Way Southwest, Seattle, Washington 98106, United States
24
Northeast 201st Street, Woodinville, Washington 98072, United States
11
Interlake Avenue North, Seattle, Washington 98103, United States
11
26th Avenue, Seattle, Washington 98122, United States
9
12006 31st Ave NE, Seattle, Washington 98125, United States
7

..
12512 Southeast 17th Street, Bellevue, Washington 98005, United States
1
1393 Northeast Hickory Lane, Issaquah, Washington 98029, United States
1
2552 Northeast 83rd Street, Seattle, Washington 98115, United States
1
14309 Southeast 38th Street, Bellevue, Washington 98006, United States
1
13321 Southeast 148th Street, Renton, Washington 98058, United States
1
Name: address, Length: 29026, dtype: int64
```

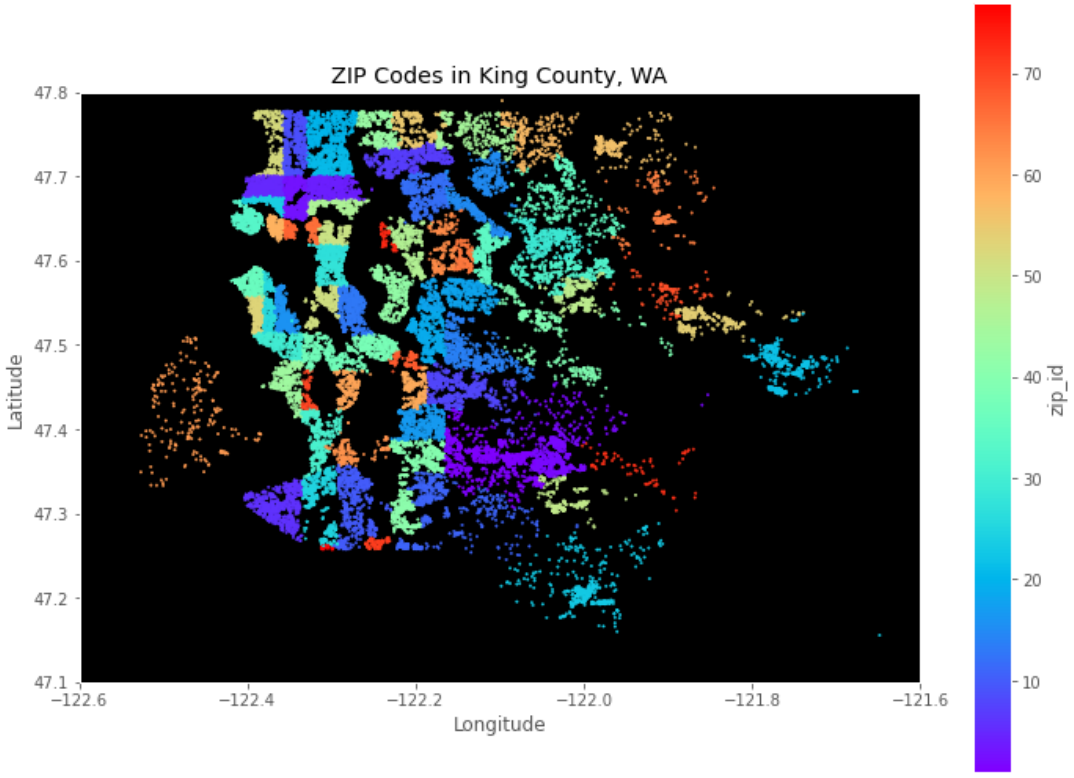
There are still address repeated in the data. After taking a look at these, it appears that there are two reasons for these repeats:

1. They correspond to different homes on the same street. Many of the repeated addresses do not have a street number, hence why they aren't unique.
2. They correspond to buildings with different units (like a townhouse or condominium). If the unit number was provided, they would be unique.

I do not see a compelling reason to remove any of these.

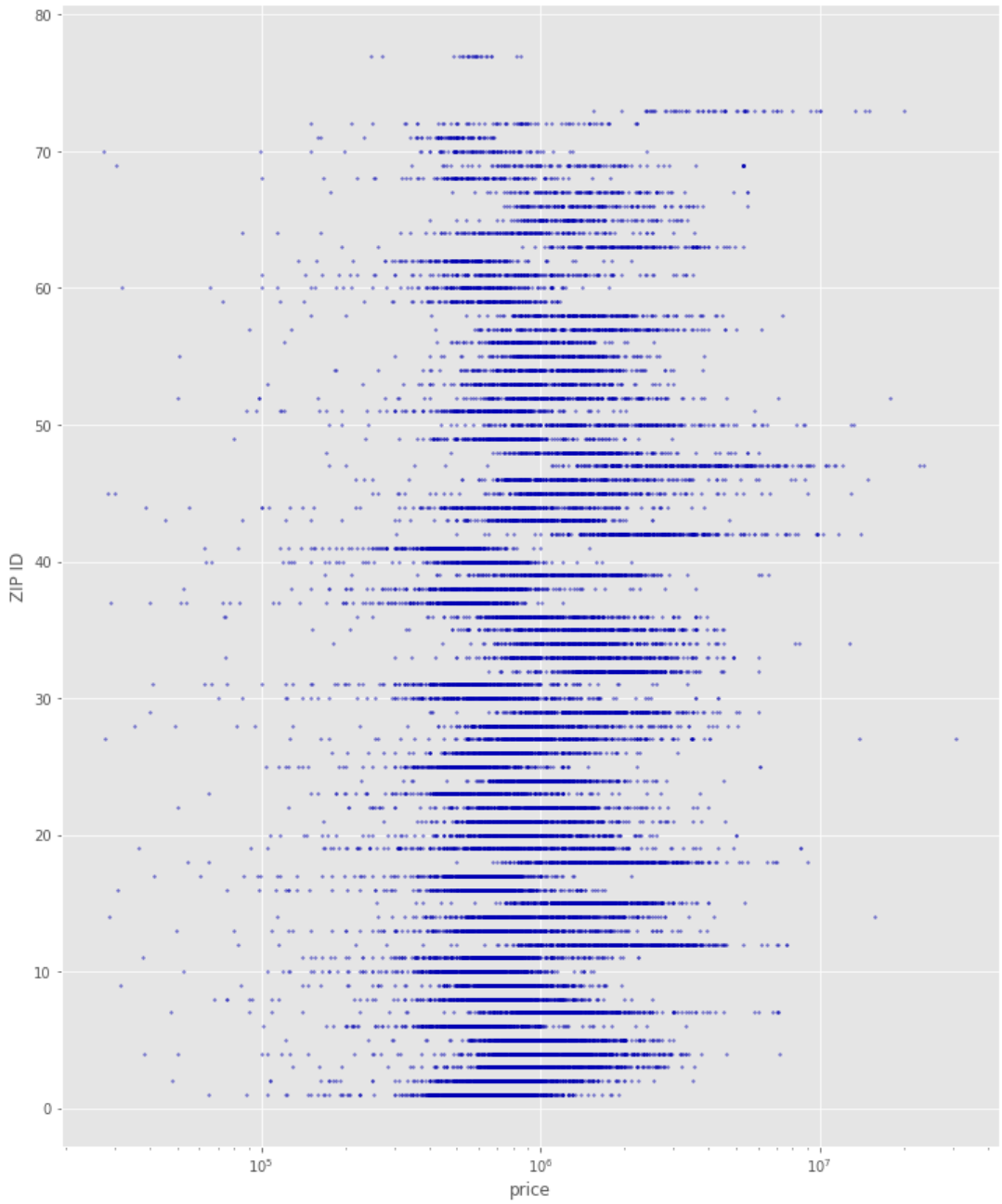
In [330]:

```
# Look at locations of the homes in the data
# Colored by zip code ID
fig, ax = plt.subplots(figsize = (12,9))
df.plot.scatter(x = 'long', y = 'lat', ax = ax, c = 'zip_id', cmap = 'r
ax.set_xbound([-122.6,-121.6])
ax.set_ybound([47.1,47.8])
ax.set_facecolor((0.0, 0.0, 0.0))
ax.set_title('ZIP Codes in King County, WA')
ax.set_xlabel('Longitude', fontsize = 12)
ax.set_ylabel('Latitude', fontsize = 12)
plt.gca().set_aspect('equal')
plt.grid(False)
```



In [331]:
```
# Plot of price in each zip code
fig, ax = plt.subplots( figsize = (12,15) )

ax.scatter(df['price'],df['zip_id'], s = 3, color = (0.0,0.0,0.7,0.5))
ax.set_xscale('log')
ax.set_xlabel('price')
ax.set_ylabel('ZIP ID')
```

Out[331]: Text(0, 0.5, 'ZIP ID')



It does look like some zip codes have significantly different price ranges than other zip codes

Preparing Data for Model

Below, I place each feature (other than the target variable, price) into one of three categories:
Continuous numeric, discrete numeric, string categorical

In [332]:
```
cont_num = ['sqft_living','sqft_lot','sqft_above','sqft_basement','sqft_

disc_num = ['bedrooms','bathrooms','floors','yr_built','yr_renovated','

str_cat = ['waterfront','greenbelt','nuisance','view','condition','heat_
```

I put grade in discrete numerical group. I want the values to actual be numbers. I do that below.

In [333]:
```
# Change grades from strings to integers
df['grade'] = df['grade'].replace({'1 Cabin': 1,'2 Substandard': 2,'3 P
                                   '6 Low Average': 6, '7 Average': 7,
                                   '11 Excellent': 11, '12 Luxury': 12,
df['grade'].value_counts()
```

```
<ipython-input-333-e635de71cd00>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas
-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
(https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html
#returning-a-view-versus-a-copy)
  df['grade'] = df['grade'].replace({'1 Cabin': 1,'2 Substandard':
2,'3 Poor': 3, '4 Low': 4, '5 Fair': 5,\
```

Out[333]:
```
7     11548
8      8860
9      3588
6      2837
10     1349
11      402
5       390
12      122
4        49
13       24
3        13
2         2
1         2
Name: grade, dtype: int64
```

In [334]:
```
# Look at statistical summary of numerical columns
df.describe()
```

Out[334]:

	id	price	bedrooms	bathrooms	sqft_living	sqft
count	2.918600e+04	2.918600e+04	29186.000000	29186.000000	29186.000000	2.918600e
mean	4.536334e+09	1.113084e+06	3.436065	2.332728	2131.210957	1.715816e
std	2.881814e+09	8.956271e+05	0.978179	0.895497	977.078832	6.103528e
min	1.000055e+06	2.736000e+04	0.000000	0.000000	3.000000	4.020000e
25%	2.087000e+09	6.450000e+05	3.000000	2.000000	1440.000000	5.000000e
50%	3.874010e+09	8.680000e+05	3.000000	2.500000	1940.000000	7.560000e
75%	7.286500e+09	1.310000e+06	4.000000	3.000000	2640.000000	1.077525e
max	9.904000e+09	3.075000e+07	13.000000	10.500000	15360.000000	3.253932e

Some of the data have mean values significantly larger than the median values. This is an indication that their distributions are skewed with a tail of large values. The columns with larger means than medians are price, sqft_living, sqft_lot, sqft_above.

For sqft_basement, sqft_garage, and sqft_patio, the minimum values are 0. For records with zeros, that indicates that it doesn't have a basement, garage, or patio. It might make sense to create a categorical variable for whether a home has these features.

The median of yr_renovated is zero. This means more than half of the houses in the dataset have never been renovated.

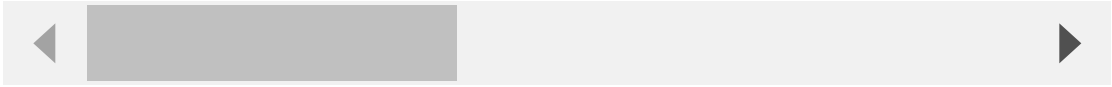
In [335]:

```
# Look at houses with smallest garages
df[df['sqft_garage'] > 0].sort_values(by = 'sqft_garage')
```

Out[335]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors
14977	1549500215	2021-12-17	1803000.0	4	4.0	3	326701	2.(
12936	7175300065	2021-07-07	560000.0	3	1.0	1270	6000	1.5
22305	1138000090	2021-12-30	885000.0	3	2.0	1180	7201	1.(
23451	7305300470	2022-05-20	1112200.0	4	3.0	1990	8409	1.(
1879	3387800200	2022-05-23	510000.0	3	1.0	1290	9052	1.(
...	...	...	...	...	...	...	...	..
7468	2746000022	2021-07-20	890000.0	4	3.0	2510	49658	1.(
3158	1196001840	2021-12-08	328000.0	4	3.0	2740	23800	1.(
2984	1824059176	2021-12-17	4700000.0	5	6.0	6620	30056	2.(
18100	3304700355	2021-12-12	17800000.0	5	7.0	12470	92345	2.(
1980	1424079032	2021-12-01	1150000.0	4	3.0	4500	62205	1.(

19747 rows × 28 columns



There is one record that claims to have the following:

3 square feet of living space

2 square feet of above grade space

1 square foot basement and 1 square foot garage

This could be a mistake and they thought they were listing rooms or someone is being cheeky. Either way, this record is not representative of a real home, so I will remove it.

In [336]:
```
# Remove the record with unrealistic data
df.drop(index = 14977, inplace = True)
df[df['sqft_garage'] > 0].sort_values(by = 'sqft_garage')
```

```
C:\Users\david\anaconda3\envs\learn-env\lib\site-packages\pandas\core
\frame.py:4163: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas
-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
(https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html
#returning-a-view-versus-a-copy)
  return super().drop(
```

Out[336]:

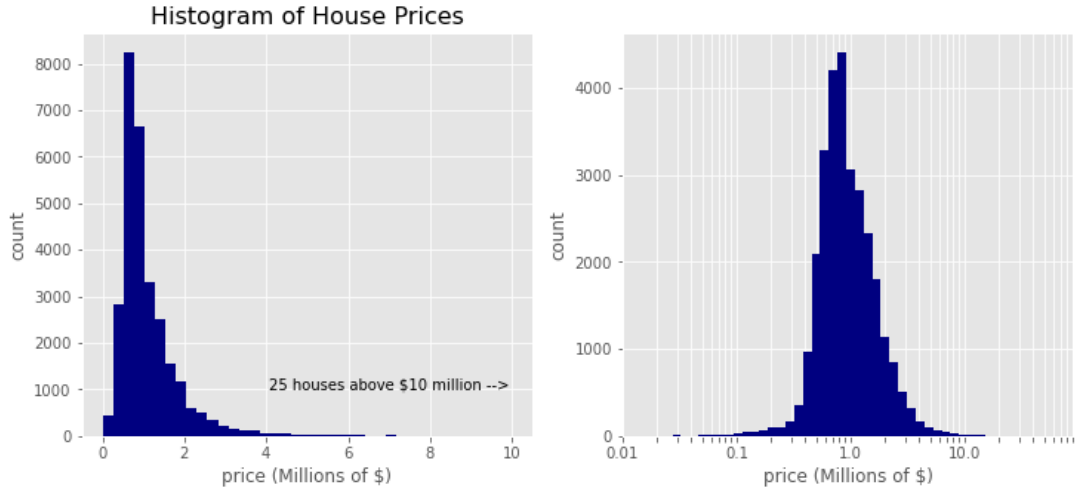
	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors
12936	7175300065	2021-07-07	560000.0	3	1.0	1270	6000	1.5
23451	7305300470	2022-05-20	1112200.0	4	3.0	1990	8409	1.0
22305	1138000090	2021-12-30	885000.0	3	2.0	1180	7201	1.0
18142	6145601500	2021-11-22	760000.0	3	1.0	1450	3936	1.0
16746	5379802853	2021-09-02	525000.0	3	2.0	1480	16875	1.0
...	...	...	...	...	...	...	...	..
7468	2746000022	2021-07-20	890000.0	4	3.0	2510	49658	1.0
3158	1196001840	2021-12-08	328000.0	4	3.0	2740	23800	1.0
2984	1824059176	2021-12-17	4700000.0	5	6.0	6620	30056	2.0
18100	3304700355	2021-12-12	17800000.0	5	7.0	12470	92345	2.0
1980	1424079032	2021-12-01	1150000.0	4	3.0	4500	62205	1.0

19746 rows × 28 columns

In [349]:
```
np.linspace(0,10**7,40)
```

Out[349]:
```
array([       0.        ,   256410.25641026,   512820.51282051,
        769230.76923077,  1025641.02564103,  1282051.28205128,
       1538461.53846154,  1794871.79487179,  2051282.05128205,
       2307692.30769231,  2564102.56410256,  2820512.82051282,
       3076923.07692308,  3333333.33333333,  3589743.58974359,
       3846153.84615385,  4102564.1025641 ,  4358974.35897436,
       4615384.61538462,  4871794.87179487,  5128205.12820513,
       5384615.38461538,  5641025.64102564,  5897435.8974359 ,
       6153846.15384615,  6410256.41025641,  6666666.66666667,
       6923076.92307692,  7179487.17948718,  7435897.43589744,
       7692307.69230769,  7948717.94871795,  8205128.20512821,
       8461538.46153846,  8717948.71794872,  8974358.97435897,
       9230769.23076923,  9487179.48717949,  9743589.74358974,
      10000000.        ])
```

In [355]:
```
# Histogram of price (log-scale on the right)
fig, ax = plt.subplots(ncols = 2, figsize = (12, 5))

ax[0].hist(df['price']/1000000, bins = np.linspace(0,10,40), color = (0
ax[0].set_ylabel('count')
ax[0].set_xlabel('price (Millions of $)')
ax[0].set_title('Histogram of House Prices',fontsize = 16)
ax[0].text(7, 1000, '25 houses above $10 million -->', horizontalalignm

ax[1].hist(np.log(df['price']), bins = 40, color = (0.0,0.0,0.5,1.0))
ax[1].set_ylabel('count')
ax[1].set_xlabel('price (Millions of $)')
log_ticks = np.append(\
                      np.append(\
                                np.append(\
                                          np.linspace(10**4,9*10**4,9),
                                          np.linspace(10**5,9*10**5,9))
                                np.linspace(10**6,9*10**6,9)),\
                      np.linspace(10**7,9*10**7,9)\
                     )
ax[1].set_xticks(np.log(log_ticks))
ax[1].set_xticklabels([x/10**6 if i%9 == 0 else '' for i,x in enumerate
```



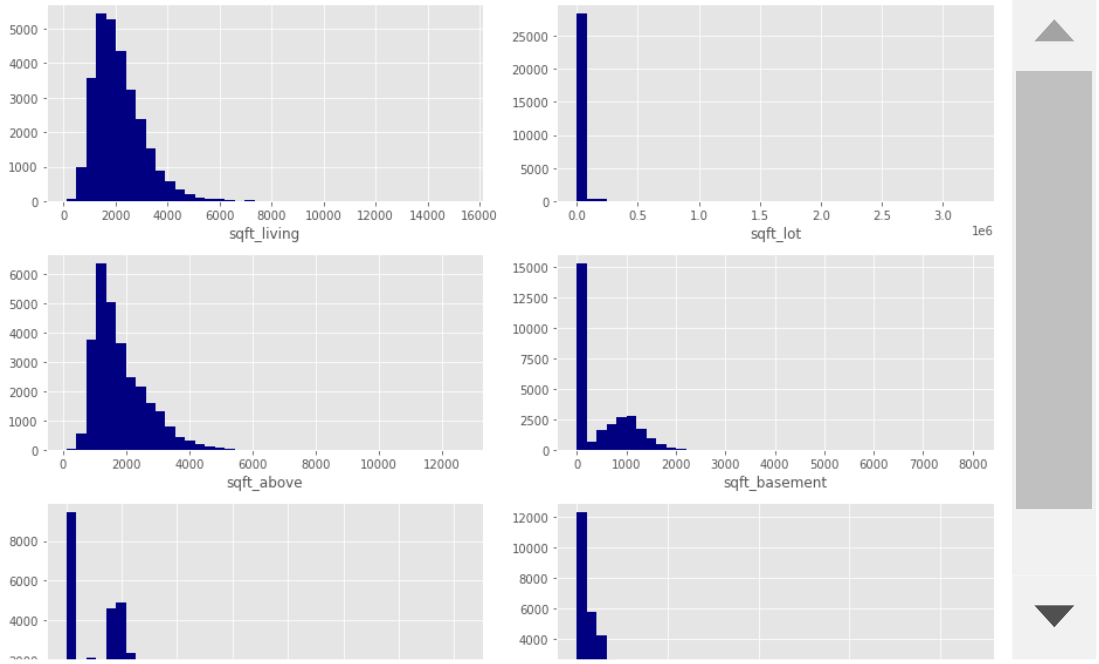
The histogram on the left is highly skewed. After taking a natural log of the prices, the histogram turns into the one on the right which is more symmetric and normal. There is a substantial tail to the left which is kind of surprising. The minimum prices are as low as 27,000, very low for a house. Looking up some of these properties manually on Zillow shows that the prices were not anywhere near that low in reality. I think these might be typos or possibly they were just the down payments on the homes. One example is record 7577 which claims to have a price of 28,307. However, on Zillow, the price was actually 1,500,000. Source: [https://www.zillow.com/homedetails/15708-124th-Ave-NE-Woodinville-WA-98072/48747253_zpid/ (https://www.zillow.com/homedetails/15708-124th-Ave-NE-Woodinville-WA-98072/48747253_zpid/)](https://www.zillow.com/homedetails/15708-124th-Ave-NE-Woodinville-WA-98072/48747253_zpid/)

I think this might just be the down payment on the sale for two reasons:

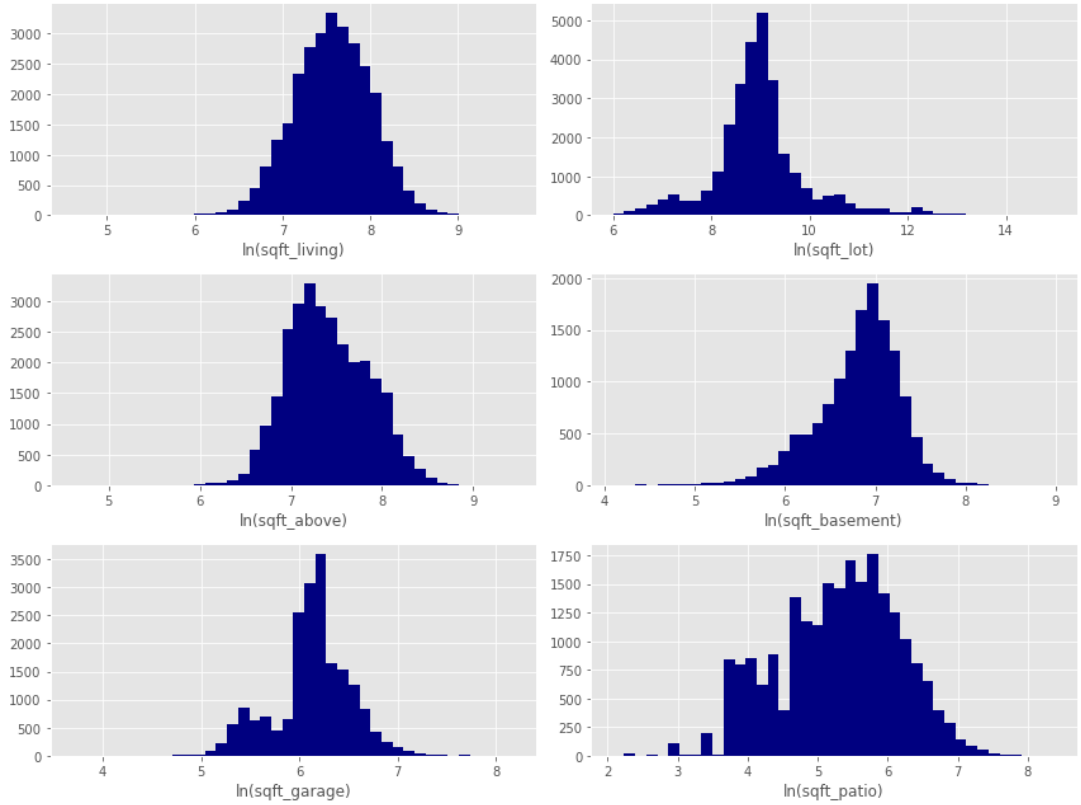
1. 28,307 is about 19% of 150,000. A typical down payment on a home should be about 20% of the price.
2. The sale date is listed as November 4, 2021 in this dataset, but Zillow says it was sold on December 6, 2021. I think this might mean the house was officially sold in December after a down payment was made in November.

This is not the only home like this. There are other examples of homes that sold for over 1

In [356]:
```
# Histograms of the continuous numerical data
fig, ax = plt.subplots(ncols = 2, nrows = 3, figsize = (12,9) )
for n, feat in enumerate(cont_num):
    i = n // 2
    j = n % 2
    ax[i,j].hist(df[feat], bins = 40, color = (0.0,0.0,0.5,1.0))
    ax[i,j].set_xlabel(feat)
fig.tight_layout()
```



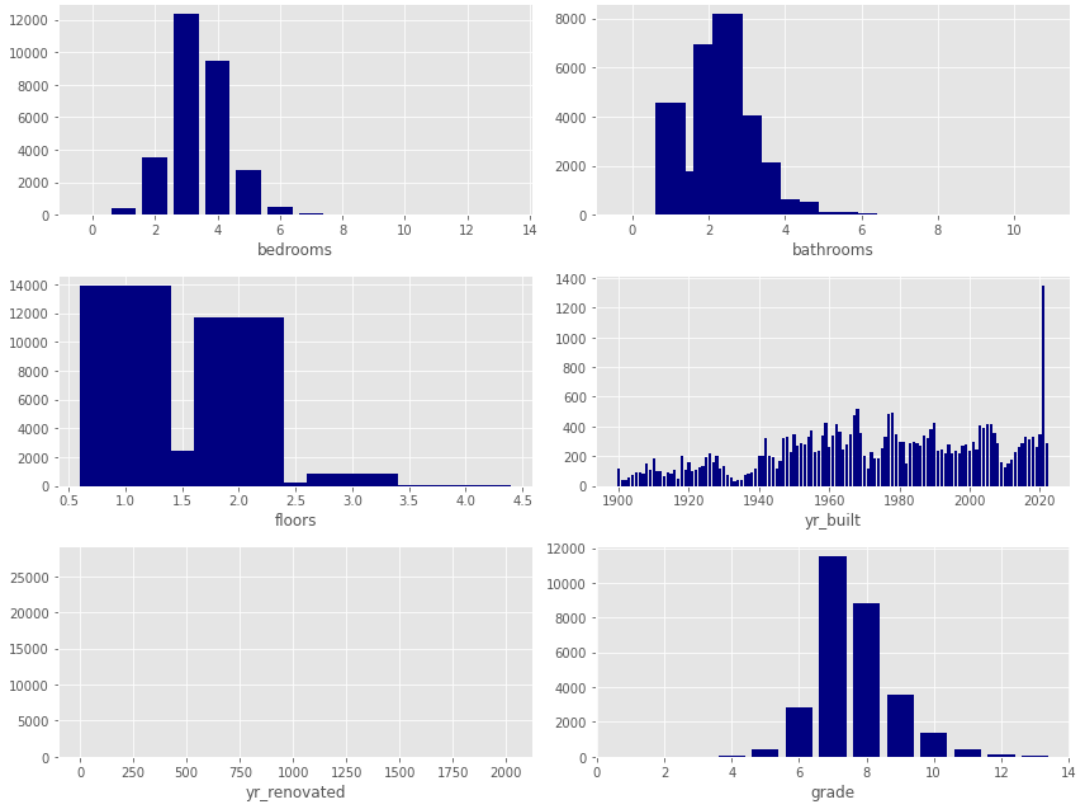
In [357]:
```
# Histograms of the continuous numerical data
fig, ax = plt.subplots(ncols = 2, nrows = 3, figsize = (12,9) )
for n, feat in enumerate(cont_num):
    i = n // 2
    j = n % 2
    ax[i,j].hist(np.log(df[df[feat] != 0][feat]), bins = 40, color = (0
    ax[i,j].set_xlabel(f"ln({feat})")
fig.tight_layout()
```



The distributions are look more symmetric now. Most of the continuous numeric features have highly skewed distributions. It might make sense to take a log of these.

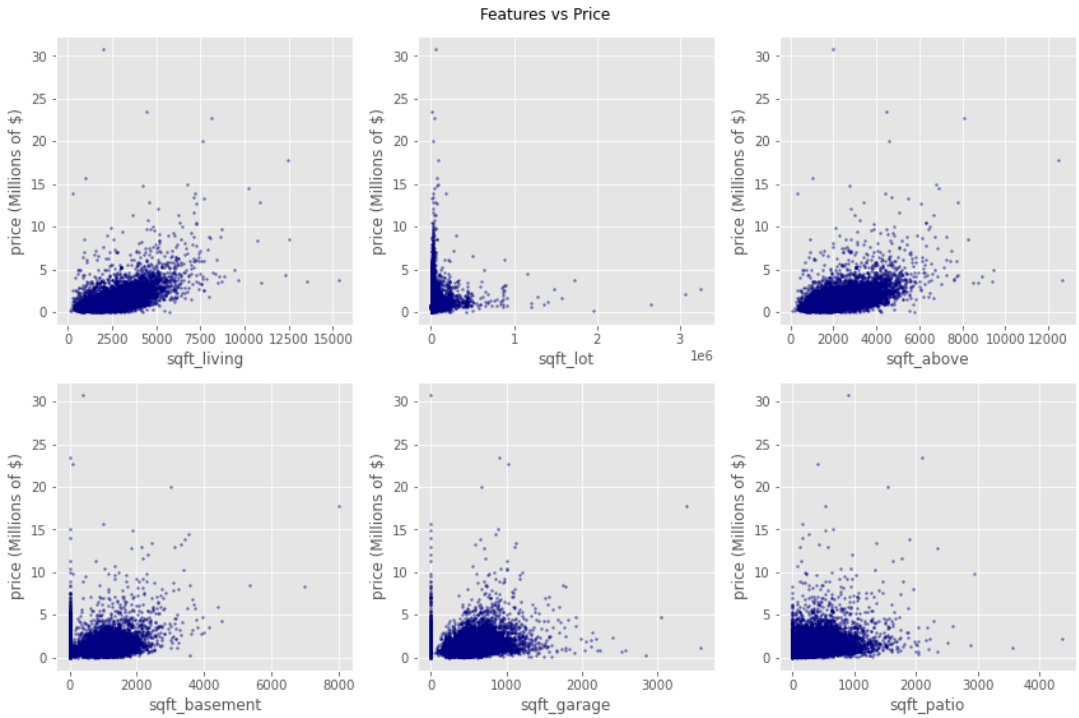
In [358]:

```
# Histograms of the discrete numerical data
fig, ax = plt.subplots(ncols = 2, nrows = 3, figsize = (12,9) )
for n, feat in enumerate(disc_num):
    i = n // 2
    j = n % 2
    x = df[feat].value_counts().index
    y = df[feat].value_counts()
    ax[i,j].bar(x,y, color = (0.0,0.0,0.5,1.0))
    ax[i,j].set_xlabel(feat)
fig.tight_layout()
```



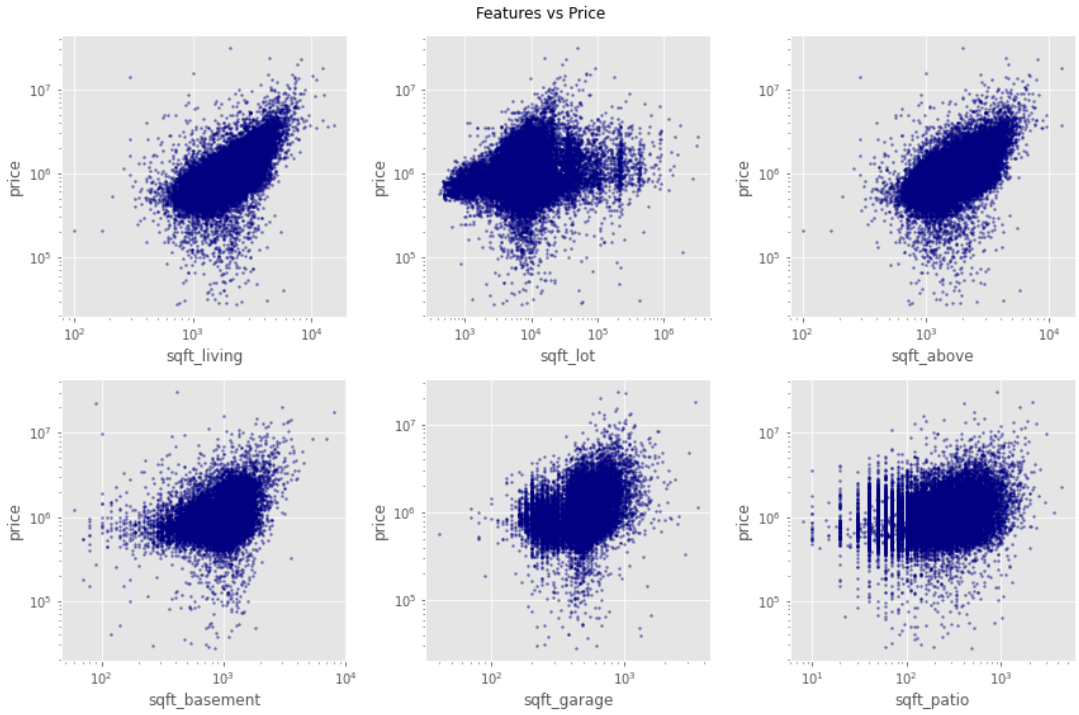
In [359]:

```
# Plots of continuous numeric features vs. price
fig, ax = plt.subplots(ncols = 3, nrows = 2, figsize = (12,8))

for n, feat in enumerate(cont_num):
    i = n // 3
    j = n % 3
    df_good = df[df[feat].isna() == False]
    ax[i,j].scatter(df_good[feat],df_good['price']/1000000, alpha=0.5,
    ax[i,j].set_xlabel(feat)
    ax[i,j].set_ylabel('price (Millions of $)')
fig.suptitle('Features vs Price')
fig.tight_layout()
```



These plots really reinforce the need to take logs of some of these variables. For most of the scatterplots above, the data is concentrated in the bottom left corner. The same plots, but with logs taken, are below.

In [360]:
```
# Same as above, but now log-log plots
fig, ax = plt.subplots(ncols = 3, nrows = 2, figsize = (12,8))

for n, feat in enumerate(cont_num):
    i = n // 3
    j = n % 3
    df_good = df[(df[feat].isna() == False) & (df[feat] != 0)]
    ax[i,j].scatter(df_good[feat],df_good['price'], alpha=0.5, s = 3, c
    ax[i,j].set_yscale('log')
    ax[i,j].set_xscale('log')
    ax[i,j].set_xlabel(feat)
    ax[i,j].set_ylabel('price')
fig.suptitle('Features vs Price')
fig.tight_layout()
```



Notes about plots above:
It seems appropriate to take logs of most of these variables since their distributions are pretty skewed before taking logs. After taking logs, the scatterplots have a much more normal distributions and linear trends are easier to see.
It is not appropriate to take logs of latitude and longitude. This is fine because that is not how I plan to use those anyway.

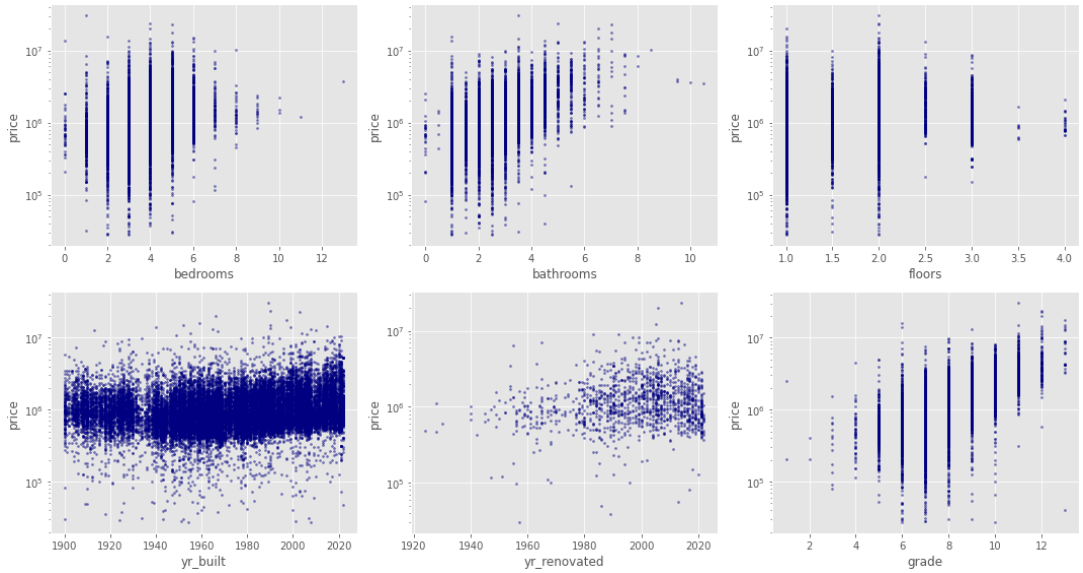
Visible trends:
sqft_living and sqft_above both show a positive trend with price.
sqft_basement, sqft_garage, and sqft_patio show weaker positive trends with price.

For some of these features, there are zeros that effectively become negative infinity after taking a log. A way around this is to create categorical variables that indicate whether the feature is 0 or not. For example, sqft_garage is 0 if a home does not have a garage. I would

create a categorical variable called has_garage which would either be 0 or 1. In the model, I would use has_garage x ln(garage) instead of just using ln(garage). I think it would also make sense to use has_garage by itself in addition to this interaction term.

I also like the idea of shifting each so the median is at 0. This will make the constant easier

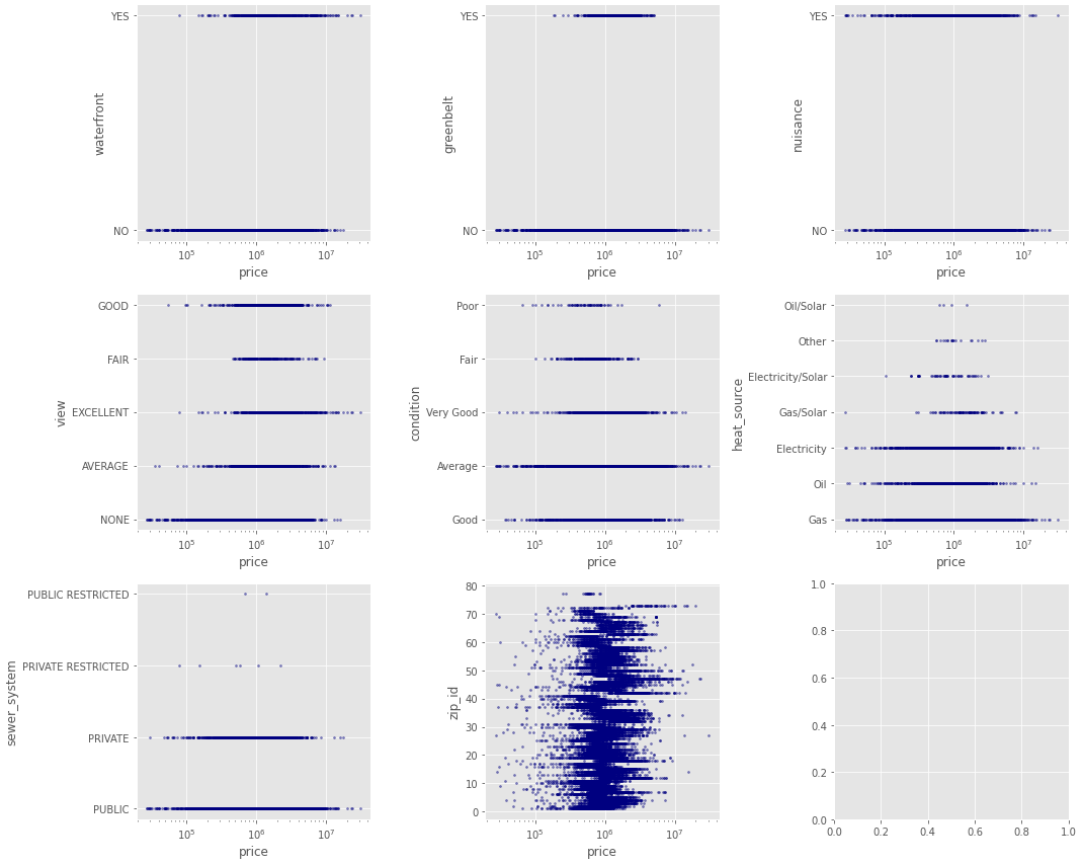
In [361]:
```
# Plots of discrete numeric features with price on log scale
fig, ax = plt.subplots(ncols = 3, nrows = 2, figsize = (15,8))

for n, feat in enumerate(disc_num):
    i = n // 3
    j = n % 3
    if feat == 'yr_renovated':
        df_good = df[(df[feat].isna() == False) & (df[feat] != 0)]
    else:
        df_good = df[df[feat].isna() == False]
    ax[i,j].scatter(df_good[feat],df_good['price'], alpha=0.5, s=4, col
    ax[i,j].set_yscale('log')
    ax[i,j].set_xlabel(feat)
    ax[i,j].set_ylabel('price')
fig.tight_layout()
```



The discrete numeric features are fine as is. It would not make much sense to take logs of these.

Visible trends:
The strongest trend is grade vs. price. Higher grades tend to go with higher prices.
It also looks like more bathrooms tends to lead to higher prices.
There might be a slight upward trend in price for more recently built houses.

In [362]:
```
# Scatterplots of categorical features vs price in log scale
fig, ax = plt.subplots(ncols = 3, nrows = 3, figsize = (15,12))

for n, feat in enumerate(str_cat):
    i = n // 3
    j = n % 3
    df_good = df[df[feat].isna() == False]
    ax[i,j].scatter(df_good['price'], df_good[feat], s=4, color = (0.0,(
    ax[i,j].set_xscale('log')
    ax[i,j].set_ylabel(feat)
    ax[i,j].set_xlabel('price')
fig.tight_layout()
```



Visible trends:

It appears that having waterfront property tends to correlate with higher prices.
There may be a trend with price vs. view, but it is not as obvious.

Linear Models with Logarithmic Data

I am going to use a linear regression model to relate the features of the houses to their prices.

A linear model means that some target variable (price) is equal to a linear combination of many independent variables.

$$\hat{y} = C + B_1 \cdot x_1 + B_2 \cdot x_2 + B_3 \cdot x_3 \ldots$$

I could hypothetically use price as my target variable, but its distribution is highly skewed (lots of low values, only a few high values). The linear regression model does not fit data like this very well because it is assuming the data is more or less normally distributed. I fixed this by instead using ln(price/median(price)) as my target variable. I also took the ln of some of the independent variables because their distributions were also heavily skewed. At the end of the day, my model will look more like this:

$$ln\left(\frac{\hat{y}}{y_{med}}\right) = C + B_1 \cdot x_1 + B_2 \cdot (x_2 - x_{2,med}) + B_3 \cdot ln\left(\frac{x_3}{x_{3,med}}\right)$$

When I feed the data to statsmodels to do the linear regression, it doesn't know that some of the data has been passed through a natural log or shifted by the median (or both). It just knows it has data and it will create a matrix to solve for the parameter values. It is up to me to turn the model back into something useful (it probably wouldn't work out if I told my stakeholder's all my recommendations in terms of ln(price)). If I take e to the power of each side of the equation, I get this:

$$\frac{\hat{y}}{y_{med}} = e^C \cdot \left(e^{B_1}\right)^{x_1} \cdot \left(e^{B_2}\right)^{x_2 - x_{2,med}} \cdot \left(\frac{x_3}{x_{3,med}}\right)^{B_3}$$

Now the model is not a sum of different contributions, but a product. While the model did not contain any interaction terms, the terms all behave in a similar way to interaction terms. The larger all the terms contributions are, the bigger effect each other term has if it changes. This means that none of the parameters effects are truly independent in this model. For example, this model says that building a garage will not add a set dollar amount to the price. Instead, it will multiply the price it had before the garage was built by some amount. A house that was worth a lot will get a bigger price increase than one that is worth a little, even though they made the same change. Whether or not this is actually how it works in the real world, I don't know, but it could affect the success of the model I make.

It is possible to create a model with both additive and multiplicative terms instead of them all being multiplicative like the model here. However, this would not be a linear regression and a more advanced technique would be needed to fit the model.

The terms e^C term represents the baseline multiplier. If we multiply the median price by this, we get the price of a home where $x_1 = 0$, $x_2 = x_{2,med}$, and $x_3 = x_{3,med}$.

The terms corresponding to B_1 and B_2 have similar interpretation. If x_1 increases by 1, then the predicted price is multiplied by e^{B_1} (same with x_2 and e^{B_2}).

Since the linear regression model was fed the natural log of x_3, it changes how B_3 is interpreted compared to the other parameters. Instead of thinking about how much we add to x_3, we need to think by what do we multiply x_3. If x_3 is multiplied by a, then the predicted price is multiplied by a^{B_3}.

In the cell below, I make some adjustments to the data to make it better for performing linear regression.

1. Create categories to indicate whether the house has a basement, garage, or patio

2. Take ln of continuous numerical data (except lat and long) then shift so the median is at 0. This is the same as dividing all the data by the median, THEN taking the ln.
3. Shift the discrete numerical data so it has the median at zero.

In [363]:

```
to_ln = ['price','sqft_living','sqft_lot','sqft_above','sqft_basement',
to_shift = ['bedrooms','bathrooms','floors','yr_built','grade']

df_adj = df.copy()

for feat in to_ln:
    # If 0, set ln(feature) to -10
    df_adj['ln_'+feat] = df[feat].apply(lambda x: -10 if x == 0 else np
    feat_median = np.median(df_adj[df_adj['ln_'+feat] != -10]['ln_'+fea
    print(f"Median of ln_{feat} is {feat_median}")
    df_adj['ln_'+feat] = df_adj['ln_'+feat].apply(lambda x: -10 if x ==
    df_adj.drop(columns = [feat], inplace=True)

for feat in to_shift:
    df_adj[feat] = df[feat] - np.median(df[feat])
    print(f"Median of {feat} is {np.median(df[feat])}")

feat_median = np.median(df[df['yr_renovated'] != 0]['yr_renovated'])
# Houses that haven't been renovated get -100 assigned
df_adj['yr_renovated'] = df['yr_renovated'].apply(lambda x: -100 if x =
print(f"Median of yr_renovated is {np.median(df[df['yr_renovated'] != 0

df_adj
```

```
Median of ln_price is 13.673946993642486
Median of ln_sqft_living is 7.570443252057374
Median of ln_sqft_lot is 8.930626469173578
Median of ln_sqft_above is 7.365180126021013
Median of ln_sqft_basement is 6.887552571664617
Median of ln_sqft_garage is 6.173786103901937
Median of ln_sqft_patio is 5.3471075307174685
Median of bedrooms is 3.0
Median of bathrooms is 2.5
Median of floors is 1.5
Median of yr_built is 1976.0
Median of grade is 7.0
Median of yr_renovated is 2002.0
```

Out[363]:

	id	date	bedrooms	bathrooms	floors	waterfront	greenbelt	nuisance
0	7399300360	2022-05-24	1.0	-1.5	-0.5	NO	NO	NC
1	8910500230	2021-12-13	2.0	0.0	-0.5	NO	NO	YES
2	1180000275	2021-09-29	3.0	-0.5	-0.5	NO	NO	NC
3	1604601802	2021-12-14	0.0	0.5	0.5	NO	NO	NC
4	8562780790	2021-08-24	-1.0	-0.5	0.5	NO	NO	YES
...	...	...	...	...	...	...	...	..
30150	7834800180	2021-11-30	2.0	-0.5	0.0	NO	NO	NC
30151	194000695	2021-06-16	0.0	-0.5	0.5	NO	NO	NC
30152	7960100080	2022-05-27	0.0	-0.5	-0.5	NO	NO	YES
30153	2781280080	2022-02-24	0.0	0.0	0.5	NO	NO	NC
30154	9557800100	2022-04-29	0.0	-1.0	-0.5	NO	NO	NC

29185 rows × 28 columns

Below, I use the correlation coefficients of the data to see which pairs of features are highly colinear. I would not want to include two features in my model that are highly correlated.

In [364]:
```
# Correlation coefficients
df_adj.corr()
```

Out[364]:

	id	bedrooms	bathrooms	floors	grade	yr_built	yr_r
id	1.000000	-0.005132	-0.011802	0.034913	0.003357	0.022604	
bedrooms	-0.005132	1.000000	0.594506	0.196438	0.390804	0.183311	
bathrooms	-0.011802	0.594506	1.000000	0.428055	0.653244	0.455973	
floors	0.034913	0.196438	0.428055	1.000000	0.471861	0.521337	
grade	0.003357	0.390804	0.653244	0.471861	1.000000	0.481290	
yr_built	0.022604	0.183311	0.455973	0.521337	0.481290	1.000000	
yr_renovated	-0.028852	0.015476	0.050879	-0.015956	-0.001556	-0.223357	
lat	-0.000386	-0.017786	0.047748	0.048355	0.132624	-0.152693	
long	0.008604	0.137690	0.183340	0.089025	0.183695	0.359332	
zip	-0.003302	-0.159394	-0.175896	-0.033864	-0.149544	-0.298547	
zip_id	0.001484	0.028806	0.059083	0.003264	0.087654	-0.061679	
ln_price	-0.022020	0.345827	0.519643	0.256414	0.620478	0.130574	
ln_sqft_living	-0.013912	0.678669	0.775099	0.370362	0.725346	0.352809	
ln_sqft_lot	-0.168031	0.157808	0.095564	-0.286061	0.103149	-0.100794	
ln_sqft_above	-0.010699	0.577865	0.677219	0.525950	0.710707	0.445021	
ln_sqft_basement	0.003329	0.104196	0.140970	-0.203287	0.034925	-0.277505	
ln_sqft_garage	0.030892	0.243693	0.362191	0.148715	0.380867	0.503528	
ln_sqft_patio	-0.006242	0.196271	0.322247	0.203612	0.293574	0.197915	

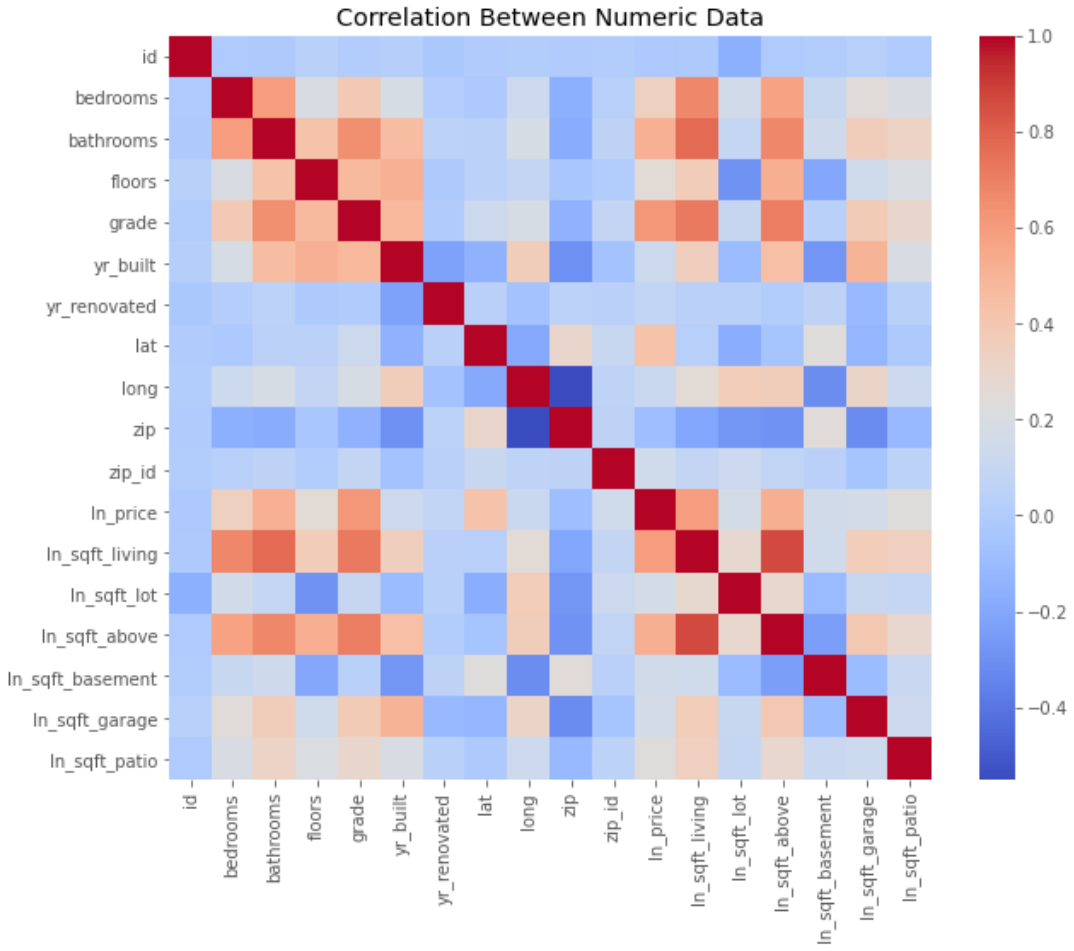
In [365]:
```
# Correlation between living space and basement space
np.corrcoef(df_adj[df_adj['ln_sqft_basement'] != -10]['ln_sqft_living'],
            df_adj[df_adj['ln_sqft_basement'] != -10]['ln_sqft_basement
```

Out[365]:
```
array([[1.        , 0.59684334],
       [0.59684334, 1.        ]])
```

The DataFrame of correlation coefficients underestimates some of the coefficients, particular for the ln_sqft_basement, ln_sqft_garage, and ln_sqft_patio features because it is counting the values even when it doesn't have that feature. If I remove those, like I have above, the correlation coefficient goes up significantly. This means that it might not make sense to include parameters for living area and basement/garage/patio area because they are fairly colinear.

In [366]:
```
fig, ax = plt.subplots ( figsize = (10,8))
sns.heatmap(df_adj.corr(), ax = ax, cmap = 'coolwarm')
ax.set_title('Correlation Between Numeric Data')
```

Out[366]: Text(0.5, 1.0, 'Correlation Between Numeric Data')



In [367]:
```
# Find the largest correlation coefficients
df_corr = df_adj.corr().stack().reset_index().sort_values(0, ascending=
df_corr['pairs'] = list(zip(df_corr.level_0, df_corr.level_1))
df_corr.drop(columns = ['level_0','level_1'], inplace=True)
df_corr.set_index(['pairs'], inplace=True)
df_corr.rename(columns = {0:'corr'}, inplace=True)
df_corr
```

Out[367]:

pairs	corr
(id, id)	1.000000
(zip, zip)	1.000000
(floors, floors)	1.000000
(grade, grade)	1.000000
(yr_built, yr_built)	1.000000
...	...
(ln_sqft_basement, long)	-0.311144
(zip, ln_sqft_garage)	-0.317496
(ln_sqft_garage, zip)	-0.317496
(zip, long)	-0.550053
(long, zip)	-0.550053

324 rows × 1 columns

In [368]:
```
#Find pairs of features with strong correlation
df_corr[(df_corr['corr'] > 0.75) | (df_corr['corr'] < -0.75)]
```

Out[368]:

pairs	corr
(id, id)	1.000000
(zip, zip)	1.000000
(floors, floors)	1.000000
(grade, grade)	1.000000
(yr_built, yr_built)	1.000000
(yr_renovated, yr_renovated)	1.000000
(lat, lat)	1.000000
(long, long)	1.000000
(zip_id, zip_id)	1.000000
(bedrooms, bedrooms)	1.000000
(ln_price, ln_price)	1.000000
(ln_sqft_living, ln_sqft_living)	1.000000
(ln_sqft_lot, ln_sqft_lot)	1.000000
(ln_sqft_above, ln_sqft_above)	1.000000
(ln_sqft_basement, ln_sqft_basement)	1.000000
(ln_sqft_garage, ln_sqft_garage)	1.000000
(bathrooms, bathrooms)	1.000000
(ln_sqft_patio, ln_sqft_patio)	1.000000
(ln_sqft_above, ln_sqft_living)	0.871877
(ln_sqft_living, ln_sqft_above)	0.871877
(bathrooms, ln_sqft_living)	0.775099
(ln_sqft_living, bathrooms)	0.775099

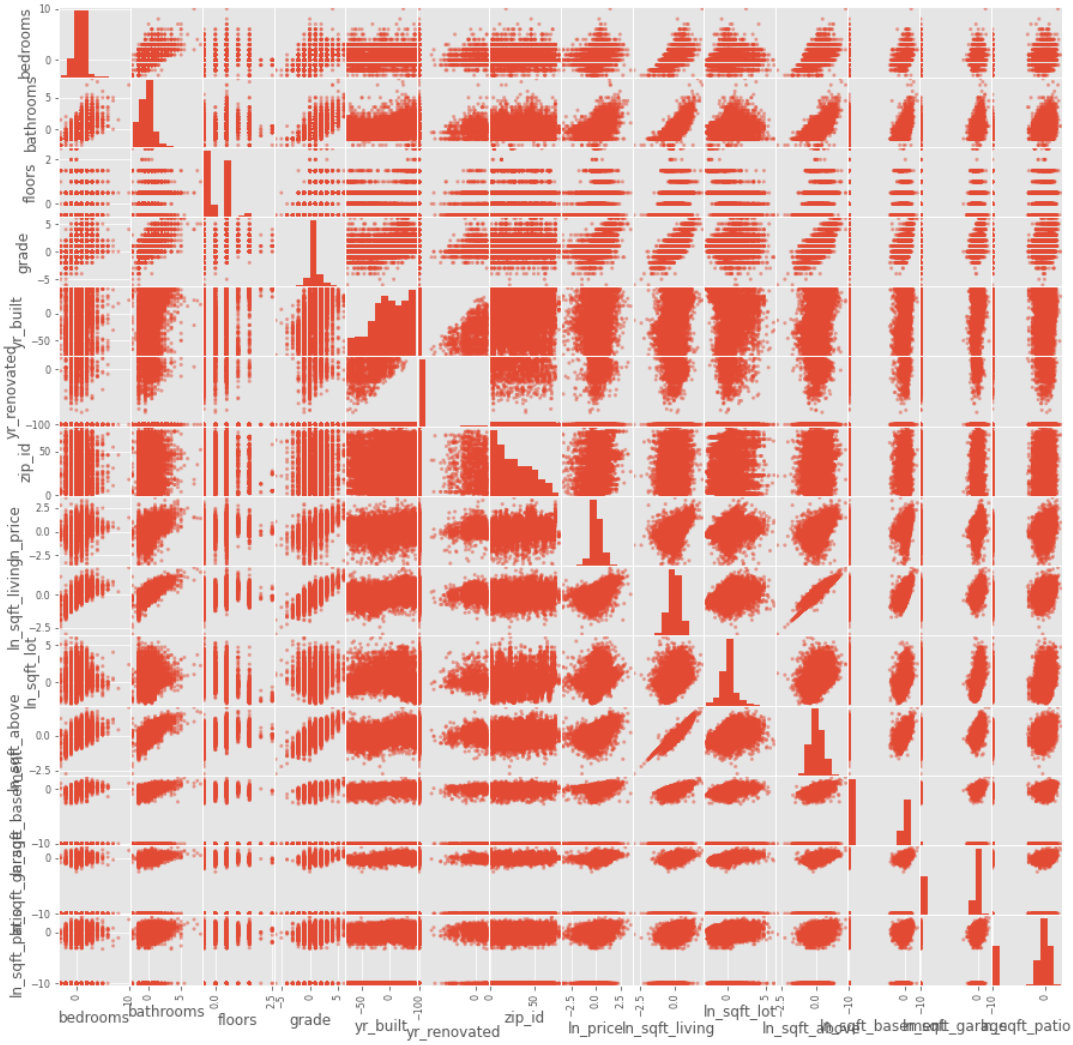
Strong correlations:

ln_sqft_living and ln_sqft_above: 0.871877

ln_sqft_living and bathrooms: 0.775099

It would not make sense to include both ln_sqft_living and ln_sqft_above in the model since they are highly correlated and basically represent the same information.

In [369]:
```
# Scatter matrix again, but this time with logs of continuous numeric v
for_scatter_matrix = ['bedrooms','bathrooms','floors','grade','yr_built
                      'ln_sqft_living','ln_sqft_lot','ln_sqft_above','l
                      'ln_sqft_patio']
pd.plotting.scatter_matrix(df_adj[for_scatter_matrix], figsize = (15,15
```



The scatter matrix above shows strong correlation between ln_sqft_living and ln_sqft_above. In fact, ln_sqft_living has linear trends with a few other features (bedrooms, bathrooms, grade, ln_sqft_basement, ln_sqft_garage).

Iteratively making a model

I use the cell below to create a model. The cells below it are used to evaluate the model.

In [446]:

```
# Choose features to include
# 1 is included, 0 is excluded

features = [['ln_sqft_living','numerical',1], \
            ['ln_sqft_lot','numerical',0], \
            ['ln_sqft_basement','num-cat',0], \
            ['ln_sqft_garage','num-cat',0], \
            ['ln_sqft_patio','num-cat',0], \
            ['bedrooms','numerical',0], \
            ['bathrooms','numerical',0], \
            ['floors','numerical',0], \
            ['yr_built','numerical',0], \
            ['condition','categorical',1], \
            ['grade','numerical',1], \
            ['waterfront','YN',1], \
            ['greenbelt','YN',0], \
            ['nuisance','YN',0], \
            ['view','categorical',0], \
            ['zip_id','categorical',1], \
            ['yr_renovated','num-cat2',1]]

y = df_adj['ln_price']

in_fit = df_adj[['ln_price']]
for feat in features:
    if feat[2] == 1:

        if feat[1] == 'numerical': in_fit[feat[0]] = df_adj[feat[0]]

        if feat[1] == 'num-cat':
            in_fit['has_'+feat[0]] = df_adj[feat[0]].apply(lambda x: 0 
            in_fit[feat[0]] = df_adj[feat[0]] * in_fit['has_'+feat[0]]

        if feat[1] == 'num-cat2':
            in_fit['has_'+feat[0]] = df_adj[feat[0]].apply(lambda x: 0 
            #in_fit[feat[0]] = df_adj[feat[0]] * in_fit['has_'+feat[0]]

        if feat[1] == 'categorical':
            in_fit[feat[0]] = df_adj[feat[0]]
            in_fit = pd.get_dummies(in_fit, columns=[feat[0]], drop_fir

        if feat[1] == 'YN': in_fit['has_'+feat[0]] = df_adj[feat[0]].ap

X = in_fit.drop(columns = 'ln_price')
results = sm.OLS(y,sm.add_constant(X)).fit()
results.summary()
```

```
<ipython-input-446-6d6fda21f6da>:28: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFram
e.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/p
andas-docs/stable/user_guide/indexing.html#returning-a-view-versu
s-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guid
e/indexing.html#returning-a-view-versus-a-copy)
  if feat[1] == 'numerical': in_fit[feat[0]] = df_adj[feat[0]]
<ipython-input-446-6d6fda21f6da>:39: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFram
e.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/p
andas-docs/stable/user_guide/indexing.html#returning-a-view-versu
s-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guid
e/indexing.html#returning-a-view-versus-a-copy)
```

The cell below turns all of the parameters into multipliers (the result of exponentiating both sides of the equation).

mult_10percent represents the amount by which the price is multiplied if the independent variable is multiplied by 1.1 (gets 10% larger). Only variables that were passed through a natural log before creating the model have this kind of multiplier.

mult_add1 represents the amount by which the price is multiplied if the independent variable is increased by 1. A variable only has this kind of multiplier if it was NOT passed through a natural log before fitting.

In [468]:
```
df_results = pd.DataFrame({'parameter': [par for par in results.params.
                           'mult_10percent': [1.1**results.params[par]
                                              for par in results.params
                           'mult_add1': [np.exp(results.params[par]) if
                                         for par in results.params.inde
                           'CI95_10percent_low':[1.1**(results.params[pa
                                                 if par[0:3] == 'ln_' el
                           'CI95_10percent_high':[1.1**(results.params[p
                                                  if par[0:3] == 'ln_' (
                           'CI95_add1_low':[np.exp(results.params[par]-1
                                                if par[0:3] != 'ln_' else 1
                           'CI95_add1_high':[np.exp(results.params[par]+
                                                if par[0:3] != 'ln_' else 1
                           'p-value':[results.pvalues[par] for par in re
                          )
df_results.set_index('parameter', inplace=True)
df_results[0:10]
```

Out[468]:

parameter	mult_10percent	mult_add1	CI95_10percent_low	CI95_10percent_high	C
const	1.000000	0.654999	1.000000	1.000000	
ln_sqft_living	1.043119	1.000000	1.041804	1.044437	
condition_Fair	1.000000	0.952979	1.000000	1.000000	
condition_Good	1.000000	1.068652	1.000000	1.000000	
condition_Poor	1.000000	0.941340	1.000000	1.000000	
condition_Very Good	1.000000	1.133831	1.000000	1.000000	
grade	1.000000	1.122272	1.000000	1.000000	
has_waterfront	1.000000	1.615156	1.000000	1.000000	
zip_id_2	1.000000	1.155256	1.000000	1.000000	
zip_id_3	1.000000	1.783765	1.000000	1.000000	

In [448]:
```
# Get baseline price

base_price = np.exp( results.params['const'] + np.log(np.median(df['pri
print(f"Baseline price = {round(base_price,2)}")
print()
```

```
Baseline price = 568539.04
```

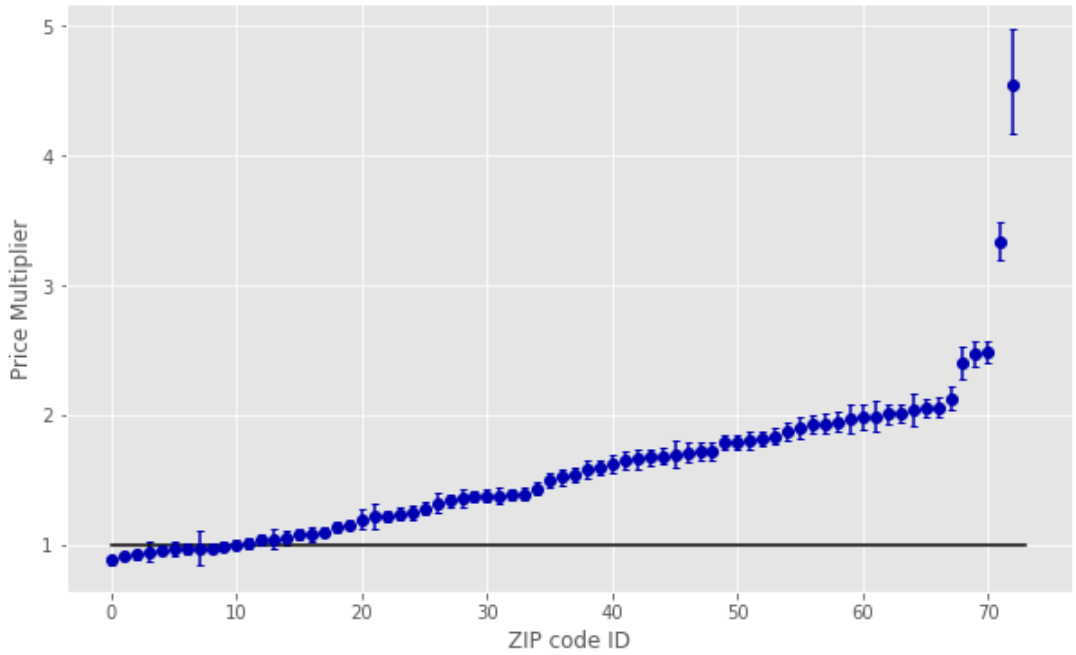
In [482]:
```
np.exp( -2.16 + np.log(np.median(df['price'])) )
```

Out[482]:
```
100102.20506103818
```

In [438]:
```
# Plot 95% confidence intervals for ZIP code parameters
fig, ax = plt.subplots( figsize = (10,6) )

df_zip = df_results[df_results.index.str.contains('zip')]
df_zip.sort_values(by = 'mult_add1',inplace=True)

yerrors = np.array([[m-el,eh-m] for m,el,eh in zip(df_zip['mult_add1'],
                                                   df_zip['CI95_add1_low'],\
                                                   df_zip['CI95_add1_high'])])

ax.errorbar(x=np.arange(len(df_zip)),y=df_zip['mult_add1'],\
            yerr=yerrors.T, fmt='o', color = (0.0,0.0,0.7,1.0), capsize
ax.plot([0,len(df_zip)],[1,1], color = (0.0,0.0,0.0,1.0))
ax.set_ylabel('Price Multiplier')
ax.set_xlabel('ZIP code ID')
```

```
<ipython-input-438-ac9d4fbcc347>:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas
-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
(https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html
#returning-a-view-versus-a-copy)
  df_zip.sort_values(by = 'mult_add1',inplace=True)
```

Out[438]:
```
Text(0.5, 0, 'ZIP code ID')
```



In [439]:
```
sum(df_zip['p-value']< 0.05/73)
```

Out[439]: 60

In [440]:
```
sum(df_zip['CI95_add1_low'] > 1.1)
```

Out[440]: 54

In [441]:
```
sum(df_zip['CI95_add1_high'] < 0.9)
```

Out[441]: 0

In [374]:
```
# Convert the covariance matrix to a correlation matrix
corr_params = results.cov_params().copy()
for param in corr_params.index:
    corr_params[param] = corr_params[param] / np.sqrt(results.cov_param
    corr_params[corr_params.index == param] = corr_params[corr_params.i
                                              np.sqrt(results.cov_par
corr_params
```

Out[374]:

	const	ln_sqft_living	condition_Fair	condition_Good	condi
const	1.000000	0.082526	-0.037504	-0.176134	
ln_sqft_living	0.082526	1.000000	0.009978	-0.048187	
condition_Fair	-0.037504	0.009978	1.000000	0.084234	
condition_Good	-0.176134	-0.048187	0.084234	1.000000	
condition_Poor	-0.021886	-0.004889	0.019020	0.049572	
...	...	...	...	...	
zip_id_71	-0.271677	0.004685	0.009781	0.026506	
zip_id_72	-0.245858	-0.011284	0.002902	0.009071	
zip_id_73	-0.220297	0.004250	-0.003615	-0.009589	

Evaluating a model

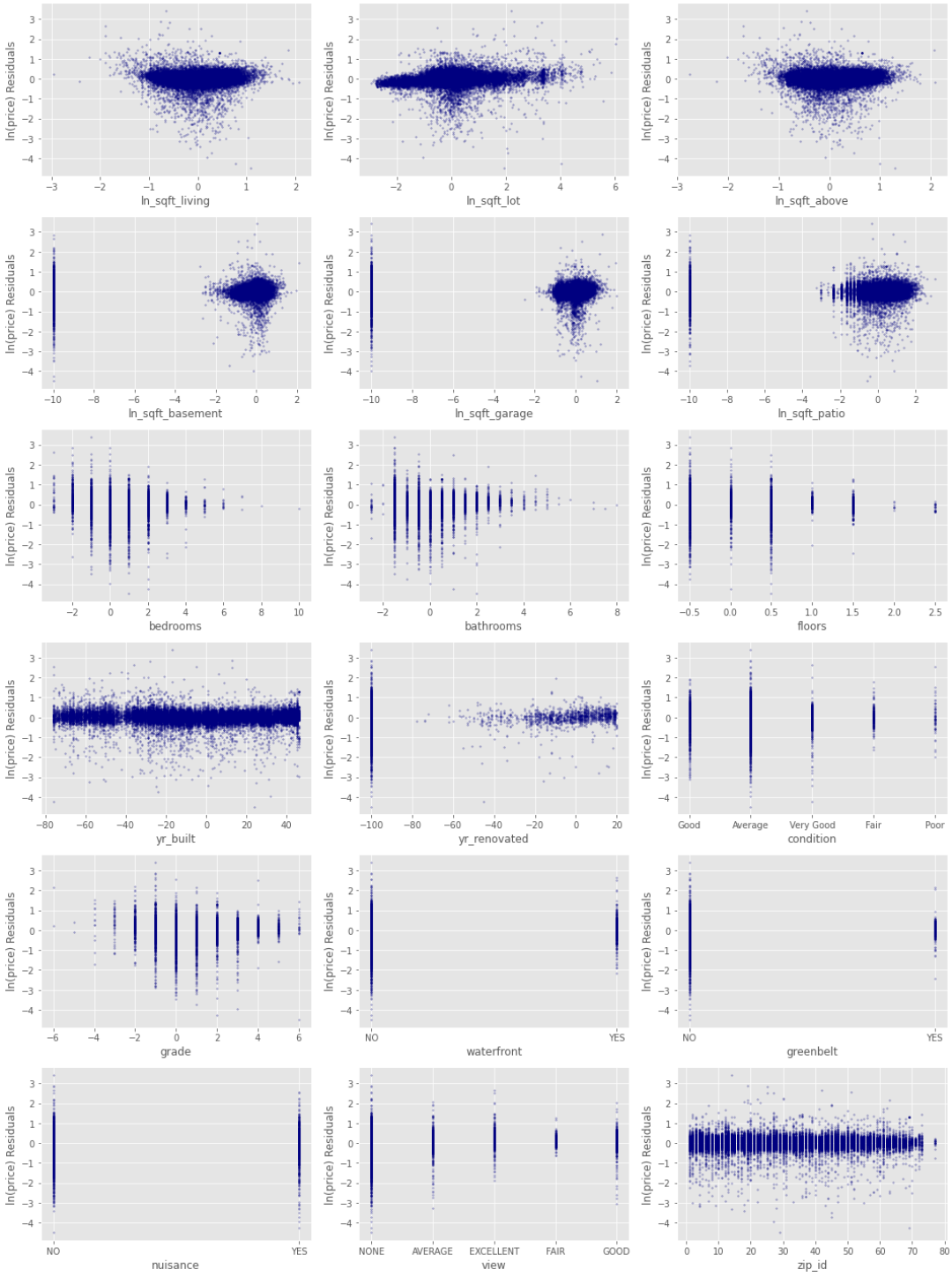
Each time I created a model, I would use the cells below to determine how good the model is.

I looked at the residuals of the model, both compared to each feature in the data, but also the distribution of residuals.

I also looked at partial regression plots and CCPR plots to tell if the fits were significant

In [469]:

```
# Plot residuals against each numeric feature
fig, ax = plt.subplots(ncols = 3, nrows = 6, figsize = (15,20))

features = ['ln_sqft_living','ln_sqft_lot','ln_sqft_above','ln_sqft_bas
            'bedrooms','bathrooms','floors','yr_built','yr_renovated','
            'nuisance','view','zip_id']

for n, feat in enumerate(features):
    i = n // 3
    j = n % 3
    ax[i,j].scatter(df_adj[feat], results.resid, s=3, color = (0.0,0.0,
    ax[i,j].set_ylabel('ln(price) Residuals')
    ax[i,j].set_xlabel(feat)
fig.tight_layout()
```



In [507]:

```
# Root mean square of residuals
print(f"RMSE: {np.sqrt(results.mse_resid)}")
print(f"One-Sigma Confidence Interval: [{np.exp(-np.sqrt(results.mse_re
```

```
RMSE: 0.3285535351215288
One-Sigma Confidence Interval: [0.719964383813306 - 1.388957596351474
6]
```

The RMSE is 0.329. This value is still in terms of ln(price). After exponentiating both -RMSE and +RMSE, I get a confidence interval of 0.72 - 1.39. These values represent a confidence interval of multipliers. If the residuals are close to being normally distributed, then this would represent a 68% confidence interval. Let's find the actual 68.3% confidence interval.

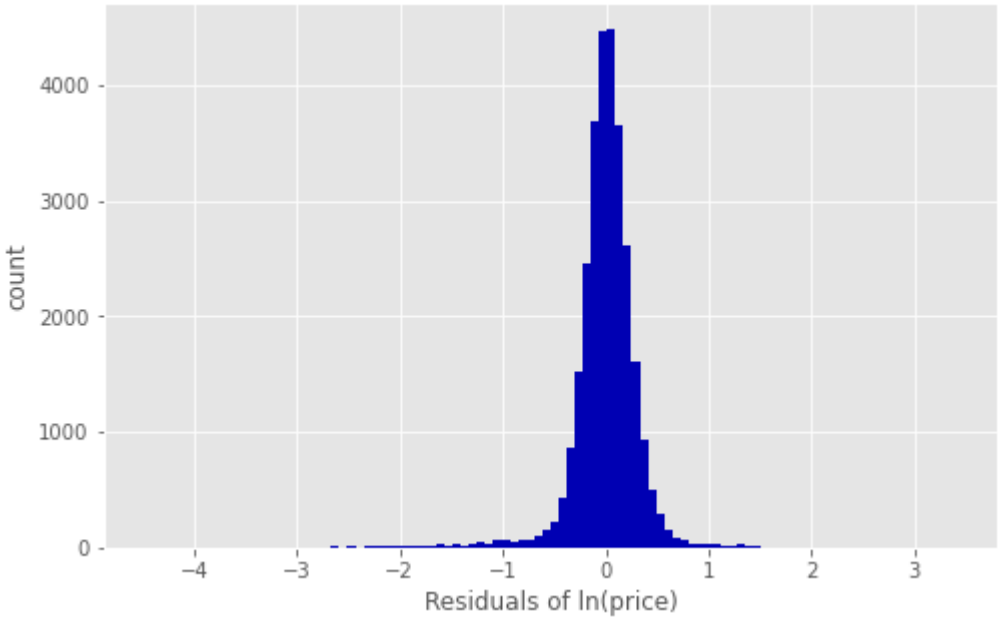
In [509]:
```
np.exp(stats.mstats.mquantiles(results.resid, prob=[0.1585,0.8415]))
```

Out[509]:
```
array([0.82084677, 1.25026589])
```

Since the distribution is more centrally concentrated than a normal distribution, the 68.3% confidence interval is not necessarily just -RMSE to +RMSE. The actual quantiles give a confidence interval from 0.82 to 1.25, which is narrower than what the RMSE gave. This is because the distribution of the residuals is not normal, which is demonstrated below.

In [499]:
```
# Plot histogram of residuals
fig, ax = plt.subplots( figsize = (8,5) )

ax.hist(results.resid, bins = 100, color = (0.0,0.0,0.7,1.0))
ax.set_xlabel('Residuals of ln(price)')
ax.set_ylabel('count')
```

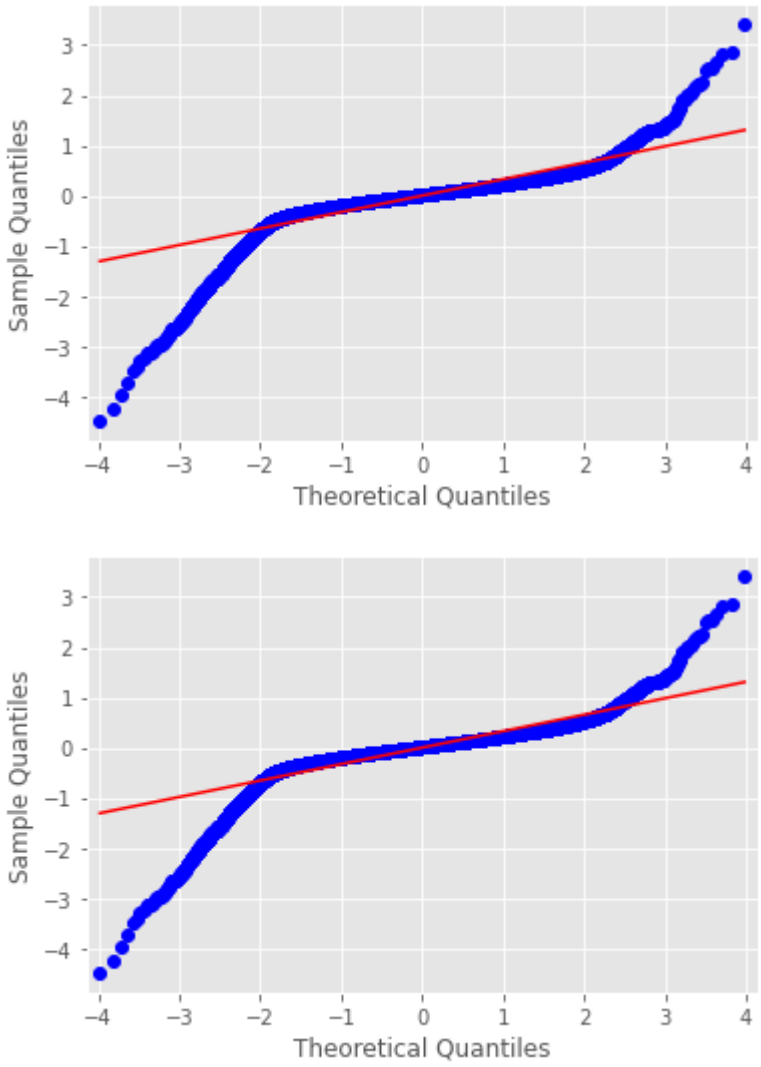
Out[499]:
```
Text(0, 0.5, 'count')
```



The distribution above is symmetric and single-peaked like a normal distribution, but it does appear to be more centrally concentrated. A qq plot will help make that more clear.

In [497]:
```
# Make Quantile-Quantile plot to compare to normal
sm.qqplot(results.resid,line='s')
```

Out[497]:



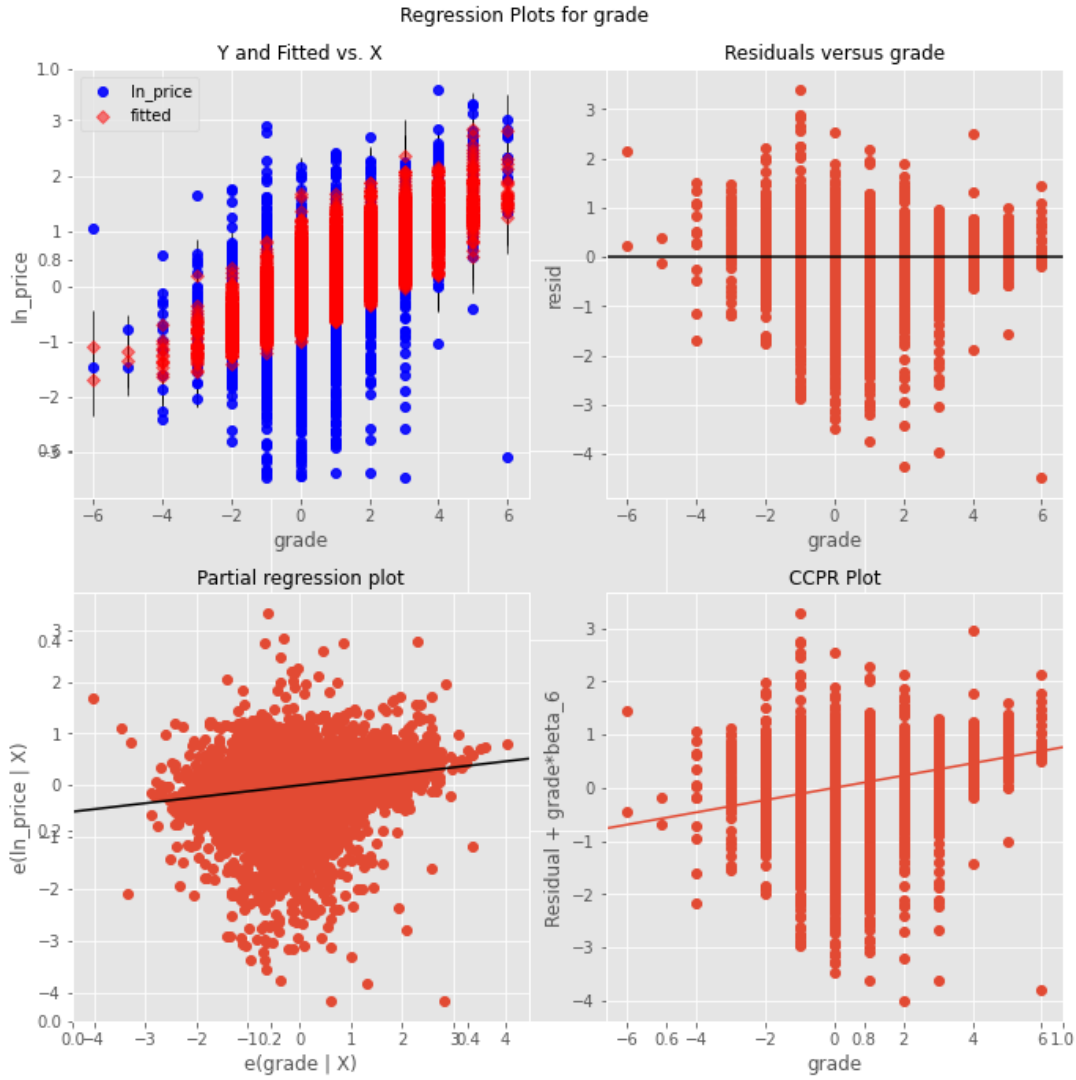
The residuals match normal quantiles near the center, but the tails do not match.

In [378]:
```
stats.kurtosis(results.resid)
```

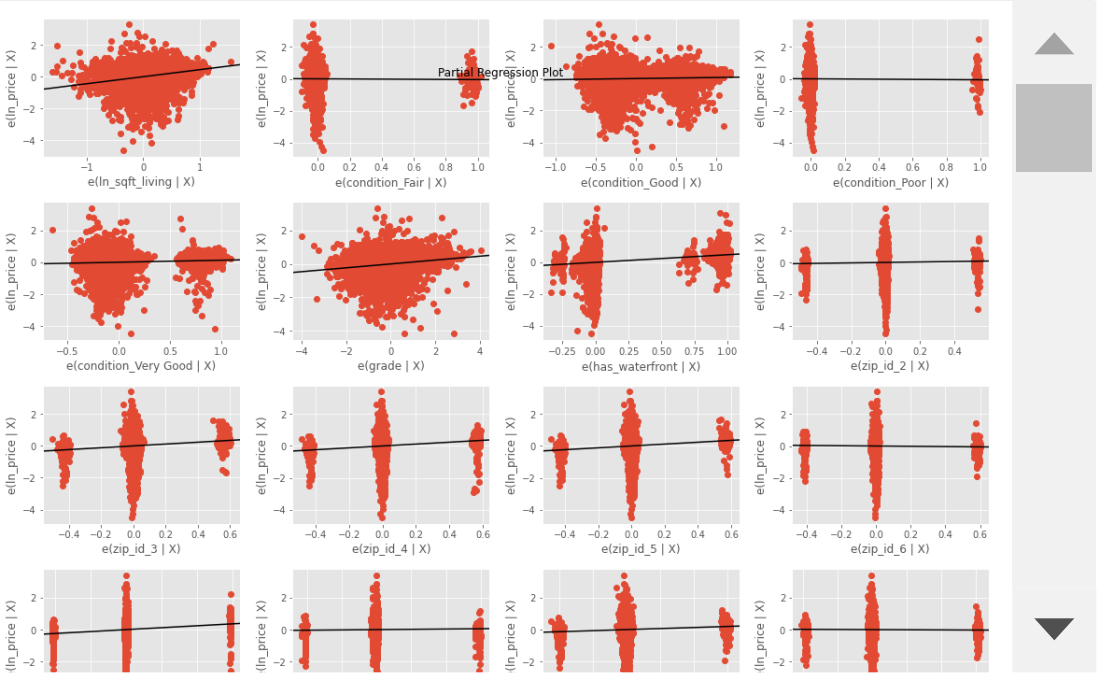
Out[378]:
```
20.5189652658584
```

The histogram of the residuals are symmetric (no skew), and more centrally peaked than a normal distribution (kurtosis equals 20). This means the distribution is more concentrated near the center and less in the tails.

In [510]:
```
# Plots for evaluating the fit to specific parameters
fig,ax = plt.subplots(figsize=(10,10))
sm.graphics.plot_regress_exog(results,'grade',fig=fig)
plt.tight_layout()
```



In [382]:
```
# Partial regression plots for the model parameters
fig = plt.figure(figsize=(15,60))
sm.graphics.plot_partregress_grid(
    results,
    exog_idx=list(X.columns),
    grid=(22,4),
    fig=fig)
plt.tight_layout()
```



Interpretation of Models

Version 1:

parameters:

1. const
2. ln_sqft_living

The price of a house should definitely be correlated with the size of the house, so square feet of living space is a good place to start.

Results:
Adjusted R-squared: 0.354
F p-value: 0.00

	coef	std err	t	P>|t|	[0.025	0.975]
const	0.0664	0.003	24.096	0.000	0.061	0.072
ln_sqft_living	0.7875	0.006	126.472	0.000	0.775	0.800

const meaning: Price is $927,560.62 for a home with median square feet of living space (1920 sq-ft)
ln_sqft_meaning: Price 7.8% larger if sqft_living gets 10% larger

Interpretation
As expected, there is a statistically signficant trend between price and sqft_living. Growth in the size of the home is predicted to raise the price.

Version 2:

parameters:

1. const
2. ln_sqft_living
3. grade

I added in grade because it looked like there was still a trend with grade when compared to the residuals of model Version 1.

Results:
Adjusted R-squared: 0.429
F p-value: 0.00

	coef	std err	t	P>|t|	[0.025	0.975]
const	-0.0633	0.003	-19.019	0.000	-0.070	-0.057
ln_sqft_living	0.4048	0.009	47.618	0.000	0.388	0.421
grade	0.2018	0.003	62.049	0.000	0.195	0.208

const meaning: Price is $814,783.68 for a home with:

1. median square feet of living space (1940 sq-ft).
2. a grade of 7.

ln_sqft_living meaning: Price 3.9% larger if sqft_living gets 10% larger
grade meaning: Price gets 22.4% larger if grade increases by 1

Interpretation
The adjusted R-squared rose to 0.429, meaning the model is accounting for more of the variance in price.
Both sqft_living and grade had statistically significant trends.
This model predicts that price goes up 3.9% when living area goes up by 10%.
According to this model, the grade has a big effect; increasing the grade by 1 raises the price by 22.4%.

Version 3:

parameters:

1. const
2. ln_sqft_living

3. grade
4. has_waterfront

This time, I introduced a categorical variable that says whether a house is a waterfront property.

Results:
Adjusted R-squared: 0.441
F p-value: 0.00

	coef	std err	t	P>|t|	[0.025	0.975]
const	-0.0707	0.003	-21.367	0.000	-0.077	-0.064
ln_sqft_living	0.4009	0.008	47.615	0.000	0.384	0.417
grade	0.2004	0.003	62.214	0.000	0.194	0.207
has_waterfront	0.4740	0.020	24.248	0.000	0.436	0.512

const meaning: Price is $808,767.80 for a home with:

1. median square feet of living space (1940 sq-ft).
2. a grade of 7.
3. not on a waterfront.

ln_sqft_living meaning: Price 3.9% larger if sqft_living gets 10% larger
grade meaning: Price gets 22.2% larger if grade increases by 1
has_waterfront meaning: A home on a waterfront will be worth 60.6% more than one that is not.

Interpretation
The adjusted R-squared rose to 0.441, a small increase from the last model.
All the parameters had p-values below 0.05.
The parameters for sqft_living and grade are nearly identical to the last model.
This model says there is a big difference between being on a waterfront and not. Being on a waterfront makes the price 60.6% larger on average.

Version 4:

parameters:
1 const
2 ln_sqft_living
3 grade
4 has_waterfront
5-77 zip_id_2 - zip_id_73, zip_id_77

The previous model had 4 parameters. This one has 77. This is because I introduced zip code as a categorical variable.

Results:
Adjusted R-squared: 0.679
F p-value: 0.00

	coef	std err	t	P>|t|	[0.025	0.975]
const	-0.3828	0.011	-36.113	0.000	-0.404	-0.362
ln_sqft_living	0.4588	0.007	69.133	0.000	0.446	0.472
grade	0.1034	0.003	39.576	0.000	0.098	0.108
has_waterfront	0.4969	0.015	32.885	0.000	0.467	0.527

+73 parameters for zip_id

const meaning: Price is $591,935.49 for a home with:

1. median square feet of living space (1940 sq-ft).
2. a grade of 7.
3. not on a waterfront.
4. home in zip code 98042.

ln_sqft_living meaning: Price 4.5% larger if sqft_living gets 10% larger
grade meaning: Price gets 10.9% larger if grade increases by 1
has_waterfront meaning: A home on a waterfront will be worth 64.4% more than one that is not.
zip_id meaning: Taking e to the power of the parameter says how much to multiply the baseline predictions by.

Interpretation
The adjusted R-squared rose to 0.679, a sizable increase from the last model.
All the parameters carried over from the last model had p-values below 0.05.
The parameters for sqft_living and has_waterfront only changed slightly.
The parameter for grade changed more drastically. Instead of predicting an increase of 22.2% when the grade increases by 1, it now predicts only an 10.9% increase.
Of the 73 parameters for zip_id, 63 of them had p-values below 0.05. Since there are 73 parameters for zip code, I could also compare to 0.05/73 to reduce the number of false positives. Even when the alpha value is divided by 73, 59 of the zip_id parameters had values below the threshold. This shows that there is a significant trend with zip code.

Many of the parameters are statistically significant, but how big of an actual effect is it? To determine this, I first took e to the power of the parameters. This turns the parameters into a value that you can multiply the baseline predictions by to get a predicted price. Since the baseline model assumes the zip code is 98042, these values essentially describe how much bigger the price of similar houses in other zip codes are. A value of 1.15 means the prices are predicted to be 15% larger.

I also used the estimated error in the parameters to create a 95% confidence interval for these multiplicative factors. For 53 of the zip codes, the entire 95% confidence interval was above 1.1, meaning the houses in these zip codes are predicted to be at least 10% more expensive than the houses in 98042 at a high significance.
34 of the zip codes have 95% confidence intervals entirely above 1.5, meaning these zip codes have houses at least 50% as expensive as those in 98042. Seven of the zip codes

Version 5:

parameters:

1 const
2 ln_sqft_living
3 grade
4 has_waterfront
5-8 condition parameters (Fair, Good, Poor, Very Good)
9-81 zip_id_2 - zip_id_73, zip_id_77

This model is the same as the last, except with 4 more parameters for the condition.

Results:
Adjusted R-squared: 0.684
F p-value: 0.00

	coef	std err	t	P>|t|	[0.025	0.975]
const	-0.4192	0.011	-39.130	0.000	-0.440	-0.398
ln_sqft_living	0.4483	0.007	67.871	0.000	0.435	0.461
condition_Fair	-0.0527	0.022	-2.377	0.017	-0.096	-0.009
condition_Good	0.0625	0.005	13.580	0.000	0.053	0.072
condition_Poor	-0.0661	0.041	-1.595	0.111	-0.147	0.015
condition_Very Good	0.1204	0.006	18.666	0.000	0.108	0.133
grade	0.1132	0.003	42.646	0.000	0.108	0.118
has_waterfront	0.4935	0.015	32.865	0.000	0.464	0.523

+73 parameters for zip_id

const meaning: Price is $570,764.95 for a home with:

1. median square feet of living space (1940 sq-ft).
2. a grade of 7.
3. not on a waterfront.
4. home in zip code 98042.
5. condition is Average.

ln_sqft_living meaning: Price 4.4% larger if sqft_living gets 10% larger
grade meaning: Price gets 12.0% larger if grade increases by 1
has_waterfront meaning: A home on a waterfront will be worth 63.8% more than one that is not.
zip_id meaning: Taking e to the power of the parameter says how much to multiply the baseline predictions by.
condition meaning: Taking e to the power of the parameter says how much to multiply the baseline predictions by.

Interpretation
The adjusted R-squared rose to 0.684, a small increase compared to Version 4.
All the parameters aside from those associated with zip code and condition have p-values below 0.05.
The parameters for sqft_living, has_waterfront, and grade only changed slightly.

The parameters for zip code are nearly identical as the last model. The number of parameters with p-values below 0.05/73 remained the same and the actual values of the parameters did not change substantially.

All four condition parameters were significant at a significance level of 0.05.

The price of a home with a Very Good condition will cost 12.8% more than an Average home (all else being equal).
For Good, Fair, and Poor, those numbers are 6.4% more, 5.1% less, and 6.4% less, respectively.

Version 6:

parameters:

1 const
2 ln_sqft_living
3 grade
4 has_waterfront
5-8 condition parameters (Fair, Good, Poor, Very Good)
9-81 zip_id_2 - zip_id_73, zip_id_77
82 has_yr_renovated
83 yr_renovated

This model is the same as the last, except with a parameter that indicates whether the house was renovated and another to scale with the year it was renovated.

Results:
Adjusted R-squared: 0.686
F p-value: 0.00

	coef	std err	t	P>|t|	[0.025	0.975]
const	-0.4246	0.011	-39.711	0.000	-0.446	-0.404
ln_sqft_living	0.4435	0.007	67.078	0.000	0.431	0.456
condition_Fair	-0.0443	0.022	-2.002	0.045	-0.088	-0.001
condition_Good	0.0687	0.005	14.892	0.000	0.060	0.078
condition_Poor	-0.0585	0.041	-1.415	0.157	-0.140	0.023
condition_Very Good	0.1280	0.006	19.805	0.000	0.115	0.141
grade	0.1150	0.003	43.269	0.000	0.110	0.120
has_waterfront	0.4847	0.015	32.183	0.000	0.455	0.514

+73 parameters for zip_id

has_yr_renovated	0.1036	0.009	10.931	0.000	0.085	0.122
yr_renovated	0.0043	0.000	8.638	0.000	0.003	0.005

const meaning: Price is $567,681.63 for a home with:

1. median square feet of living space (1940 sq-ft).
2. a grade of 7.
3. not on a waterfront.
4. home in zip code 98042.
5. condition is Average.
6. Not renovated.

ln_sqft_living meaning: Price 4.3% larger if sqft_living gets 10% larger
grade meaning: Price gets 12.2% larger if grade increases by 1
has_waterfront meaning: A home on a waterfront will be worth 62.4% more than one that is not.

zip_id meaning: Taking e to the power of the parameter says how much to multiply the baseline predictions by.
condition meaning: Taking e to the power of the parameter says how much to multiply the baseline predictions by.
has_yr_renovated meaning: Price gets 10.9% larger if the house has been renovated.
yr_renovated meaning: The more recent a renovation, the more it raises the price. The price is multiplied by 1.0043^(year-2002). The 2002 in the model is because yr_renovated had its median subtracted before doing the fit. It might seem like houses renovated prior to 2002 actually lower the price since 1.0043 to a negative power would be less than 0. However, the renovation still achieves the 10.9% increase because a renovation was done.

Interpretation
The adjusted R-squared rose to 0.686, barely a change from the previous model.
All the parameters aside from those associated with zip code and condition have p-values below 0.05.
The parameters for sqft_living, has_waterfront, and grade only changed slightly from Version 5.

The parameters for zip code were not affected by the addition of new parameters. The number of parameters with p-values below 0.05/73 remained the same and the actual values of the parameters did not change substantially.

All four condition parameters were significant at a significance level of 0.05.

The actual parameter values for condition did change slightly: The price of a home with a Very Good condition will cost 13.7% more than an Average home (all else being equal). For Good, Fair, and Poor, those numbers are 7.1% more, 4.3% less, and 5.6% less, respectively.

The has_yr_renovated parameter indicates that performing renovations tends to raise the price of a house by about 11%. The timing of the renovations also matters because more

Version 7:

parameters:

1 const
2 ln_sqft_living
3 grade
4 has_waterfront
5-8 condition parameters (Fair, Good, Poor, Very Good)
9-81 zip_id_1 - zip_id_78
82 has_yr_renovated

This model is the same as the last, except with the yr_renovated parameter removed (has_yr_renovated is still present).

Results:
Adjusted R-squared: 0.685
F p-value: 0.00

	coef	std err	t	P>|t|	[0.025	0.975]
const	-0.4231	0.011	-39.525	0.000	-0.444	-0.402
ln_sqft_living	0.4429	0.007	66.906	0.000	0.430	0.456
condition_Fair	-0.0482	0.022	-2.174	0.030	-0.092	-0.005
condition_Good	0.0664	0.005	14.393	0.000	0.057	0.075
condition_Poor	-0.0605	0.041	-1.460	0.144	-0.142	0.021
condition_Very Good	0.1256	0.006	19.433	0.000	0.113	0.138
grade	0.1154	0.003	43.362	0.000	0.110	0.121
has_waterfront	0.4794	0.015	31.817	0.000	0.450	0.509

+73 parameters for zip_id

has_yr_renovated	0.0882	0.009	9.460	0.000	0.070	0.106

const meaning: Price is $568,539.04 for a home with:

1. median square feet of living space (1940 sq-ft).
2. a grade of 7.
3. not on a waterfront.
4. home in zip code 98042.
5. condition is Average.
6. Not renovated.

ln_sqft_living meaning: Price 4.3% larger if sqft_living gets 10% larger
grade meaning: Price gets 12.2% larger if grade increases by 1
has_waterfront meaning: A home on a waterfront will be worth 61.5% more than one that is not.
zip_id meaning: Taking e to the power of the parameter says how much to multiply the baseline predictions by.
condition meaning: Taking e to the power of the parameter says how much to multiply the baseline predictions by.
has_yr_renovated meaning: Price gets 9.2% larger if the house has been renovated.

Interpretation
The adjusted R-squared rose to 0.685, the same as the previous model.
All the parameters aside from those associated with zip code and condition have p-values below 0.05.
The parameters for sqft_living, has_waterfront, and grade only changed slightly from Version 6.

The parameters for zip code were not affected by the addition of new parameters. The number of parameters with p-values below 0.05/78 remained the same and the actual values of the parameters did not change substantially.

All four condition parameters were significant at a significance level of 0.05.

The actual parameter values for condition did change slightly: The price of a home with a Very Good condition will cost 13.4% more than an Average home (all else being equal). For Good, Fair, and Poor, those numbers are 6.9% more, 4.7% less, and 5.8% less, respectively.

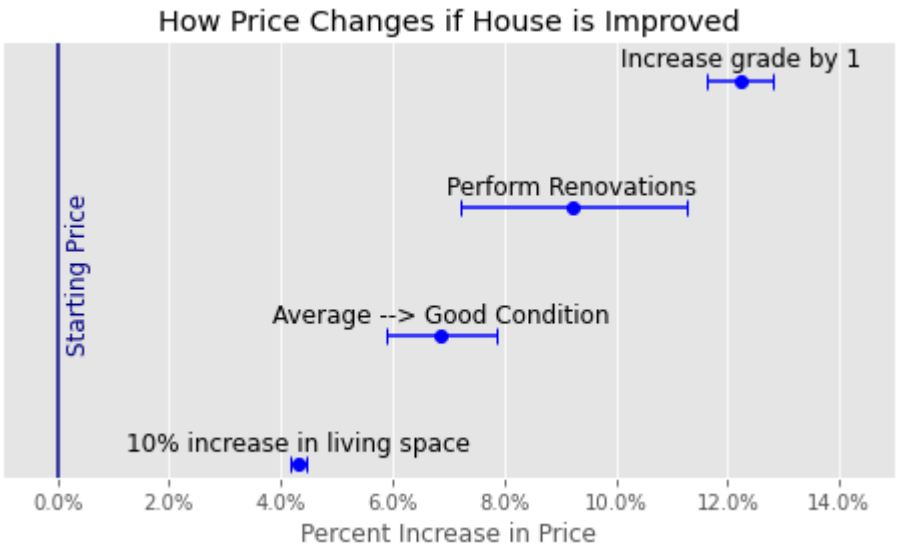
The has_yr_renovated parameter indicates that performing renovations tends to raise the price of a house by about 9%.
I removed the yr_renovated parameter because it made predictions that don't make sense for houses renovated a long time ago. One would expect that more recent renovations would raise the price more and the yr_renovated parameter did achieve that. However, if

Visualizing Results

Below, I create a visual that shows how certain factors affect the price

In [452]:
```
# Errorbar plot showing effect of changing certain features
fig, ax = plt.subplots( figsize = (8,4) )

plot_stuff = [['ln_sqft_living','mult_10percent','10% increase in livin
              ['condition_Good','mult_add1','Average --> Good Condition'
              ['has_yr_renovated','mult_add1','Perform Renovations','CI9
              ['grade','mult_add1','Increase grade by 1','CI95_add1']]

ax.axvline(x=1, color = (0.0,0.0,0.5,1.0))
ax.text(1.004, 1.5, 'Starting Price', rotation = 'vertical', fontsize =
        verticalalignment = 'center', horizontalalignment = 'center', c

n = 0
for stuff in plot_stuff:
    mean = df_results.loc[stuff[0]][stuff[1]]
    err_low = mean - df_results.loc[stuff[0]][stuff[3]+'_low']
    err_high = df_results.loc[stuff[0]][stuff[3]+'_high'] - mean
    ax.errorbar(x = df_results.loc[stuff[0]][stuff[1]], y = n, xerr=[[e
                color = (0.0,0.0,1.0,1.0))
    ax.text(df_results.loc[stuff[0]][stuff[1]], n + 0.17, stuff[2], fon
            verticalalignment = 'center', horizontalalignment = 'center
    n += 1

ax.set_yticks([])
ax.set_ybound([-0.1,3.3])
ax.set_xbound([0.99,1.15])
ax.set_xticks(np.linspace(1,1.14,8))
ax.set_xticklabels([f"{round(100*(x-1),0)}%" for x in np.linspace(1,1.1
ax.set_xlabel('Percent Increase in Price')
ax.set_title('How Price Changes if House is Improved')
```

Out[452]: Text(0.5, 1.0, 'How Price Changes if House is Improved')



I was originally going to use a horizontal bar plot rather than an errorbar plot. The reason was because I thought it would be more intuitive since you could compare the size of the baseline bar to the other bars. However, this meant that a lot of space on the plot was

Recommendations

If someone is going to sell their house in King County, WA, I would first recommend that they get their house appraised first so they have a solid idea how much it would cost if no renovations were done. If they can get specific information about anything that is harming the condition or grade of the house, that would be even better since they could address those specific issues.

Once the seller has an estimate of the value, they should next get an estimate for the cost of performing renovations. Whether or not they should do those renovations depends on whether it will cost more or less than the amount the house's price should increase.

If renovations are done and neither grade no condition is improved, the price is expected to go up by about 9%.
If renovations are done and the grade is improved by 1, the price is expected to go up by about 22%.
If renovations are done and the condition is improved from Average to Good, the price is expected to go up by about 20%.
If renovations are done and both grade and condition improve, the price is expected to up by about 31%.

The seller can compare the expected price increase to the price of the renovations to decide whether to pay for the renovations. The seller should also keep in mind that improving the grade or condition is not a guarantee and that their is uncertainty in the model. They also need to weigh non-monetary considerations like the hassle of getting the renovations done, especially if they plan to continue living in the house until it is sold.

Uncertainties

The model can make predictions, but I want to be able to provide a confidence interval rather than just one value.

The uncertainty comes from two sources: uncertainty in the parameters and variance not accounted for in the model.

Parameter uncertainty Along with values of the parameters, statsmodels also estimates uncertainties in the parameters in the form of a covariance matrix. It is important to use the entire covariance matrix and not just the variance of each individual parameter because some parameters might be correlated.

To find how the covariance of the parameters affects the price estimate, I need to propagate the errors. To propagate the errors, we need to take partial derivatives of each parameter of our linear model.

Linear Model: $\hat{y} = B_0 + \sum_{n=1}^{N} B_n x_n$

Partial Derivatives

$\frac{\partial \hat{y}}{\partial B_0} = 1$

$$\frac{\partial \hat{y}}{\partial B_i} = x_i$$

The variance in the predicted price is given by this formula:

$$\sigma^2_{\hat{y}} = \sum_{i=0}^{N} \sum_{j=0}^{N} \left(\frac{\partial \hat{y}}{\partial B_i} \right) \left(\frac{\partial \hat{y}}{\partial B_j} \right) C_{ij}$$

$$\sigma^2_{\hat{y}} = \sum_{i=0}^{N} \sum_{j=0}^{N} x_i x_j C_{ij}$$

where C is the covariance matrix. We can treat x_0 as being all 1's for this formula.

For this project, the x values might be the natural log of a feature or be shifted by the median, but it doesn't matter. I just need to make sure that whatever I plug in for the x values has been processed in exactly the same way as the data that was fed to the model. The uncertainty will depend on the values I plug in. For example, I house with a high grade will have a higher uncertainty in the price than one with an average grade (all else being equal).

Once I have $\sigma_{\hat{y}}$, I can use it to get a confidence interval on ln(price), then I can exponentiate to turn the prediction and confidence interval into a price in dollars.

Residual Uncertainty Since the R-squared value is around 68%, that means that 32% of the variance in ln(price) is explained by the model. That means there is also random uncertainty on top of the uncertainty from the parameters. Thus, the final formula for estimating the uncertainty in ln(price) is given by:

$$\sigma^2_{\hat{y}} = \sigma^2_{res} + \sum_{i=0}^{N} \sum_{j=0}^{N} x_i x_j C_{ij}$$

where σ^2_{res} is the variance from the residuals. I could estimate this by simply calculating the sample variance of the residuals. However, since the distribution of the residuals was much more strongly peaked than a normal distribution, I could also estimate it by taking finding the 15.85% and 84.15% quantiles and taking half of the interval between them. This would give me a reasonable 68.3% confidence interval for the residuals.

Predictions

Below is a function that can be used to predict the price of a house in King County, WA. It can also produce a 95% confidence interval for the prediction which takes into account both the variance (and covariance) of the parameters and the variance that is not explained by the model.

In [453]:
```
def price_predictor(res, x):
    """
    Input:
    res: The results of a linear regression
    x: Values for the independent variables for which we want a predict
    x must be just one array of values. If you want multiple predictions

    Output:
    y: predicted value
    sigma_y = error of the predicted value
    """

    # Price prediction
    y = res.params[0] + sum(res.params[1:] * x)

    # Uncertainty from parameter covariance
    sigma_y = res.cov_params()['const']['const']
    sigma_y = sigma_y + sum(2 * res.cov_params()['const'][1:] * x)
    sigma_y = sigma_y + np.matmul( np.matmul( x.T , np.array(res.cov_pa

    # Uncertainty from residuals
    qvals = stats.mstats.mquantiles(res.resid, prob=[0.1585,0.8415])
    sigma_y = sigma_y + ( (qvals[1] - qvals[0])/2 )**2

    return y, np.sqrt(sigma_y)
```

In [454]:
```
# Use the function above to make predictions about prices and compare to
# Choose a record from the data
house = 361
lnprice, sigma_lnprice = price_predictor(results, np.array(X.iloc[house
#print(lnprice, sigma_lnprice)

price = np.exp( lnprice + np.log(np.median(df['price'])) )
price_95low = np.exp( lnprice + np.log(np.median(df['price'])) - 2*sigma
price_95high = np.exp( lnprice + np.log(np.median(df['price'])) + 2*sigu
price_predict = np.exp( y[house] + np.log(np.median(df['price'])) )

print(f"Predicted price: ${round(price,2)} \t Actual price: ${round(pri
print()
print(f"95% Confidence interval: [${round(price_95low,2)} - ${round(pri
```

```
Predicted price: $803843.33      Actual price: $1250000.0

95% Confidence interval: [$527344.94 - $1225315.8]
```

Jackknife test

Below, I choose a random sample comprising 90% of the data and I create a new model. Then, I use that model to predict the prices of the remaining 10% of the data. The comparisons between the predictions and real prices will give me a sense of how good the model is.

In [455]:
```
np.random.seed(123)
rando_numbers = np.random.random(len(df_adj))
rando_numbers
```

Out[455]:
```
array([0.69646919, 0.28613933, 0.22685145, ..., 0.67017802, 0.5808031
2,
       0.58616189])
```

In [456]:
```
df_adj_sub90 = df_adj[rando_numbers < 0.9]  # Used to make new model
df_adj_sub10 = df_adj[rando_numbers >= 0.9] # Used for testing the new
```

In [457]:

```
# Make new model using 90% of the data

features = [['ln_sqft_living','numerical',1], \
            ['ln_sqft_lot','numerical',0], \
            ['ln_sqft_basement','num-cat',0], \
            ['ln_sqft_garage','num-cat',0], \
            ['ln_sqft_patio','num-cat',0], \
            ['bedrooms','numerical',0], \
            ['bathrooms','numerical',0], \
            ['floors','numerical',0], \
            ['yr_built','numerical',0], \
            ['condition','categorical',1], \
            ['grade','numerical',1], \
            ['waterfront','YN',1], \
            ['greenbelt','YN',0], \
            ['nuisance','YN',0], \
            ['view','categorical',0], \
            ['zip_id','categorical',1], \
            ['yr_renovated','num-cat2',1]]

y_sub90 = df_adj_sub90['ln_price']

in_fit = df_adj_sub90[['ln_price']]
for feat in features:
    if feat[2] == 1:

        if feat[1] == 'numerical': in_fit[feat[0]] = df_adj_sub90[feat[

        if feat[1] == 'num-cat':
            in_fit['has_'+feat[0]] = df_adj_sub90[feat[0]].apply(lambda
            in_fit[feat[0]] = df_adj_sub90[feat[0]] * in_fit['has_'+fea

        if feat[1] == 'num-cat2':
            in_fit['has_'+feat[0]] = df_adj_sub90[feat[0]].apply(lambda
            #in_fit[feat[0]] = df_adj_sub90[feat[0]] * in_fit['has_'+fe

        if feat[1] == 'categorical':
            in_fit[feat[0]] = df_adj_sub90[feat[0]]
            in_fit = pd.get_dummies(in_fit, columns=[feat[0]], drop_fir

        if feat[1] == 'YN': in_fit['has_'+feat[0]] = df_adj_sub90[feat[

X_sub90 = in_fit.drop(columns = 'ln_price')
results_sub90 = sm.OLS(y_sub90,sm.add_constant(X_sub90)).fit()
results_sub90.summary()
```

```
<ipython-input-457-17ce1928e433>:27: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFram
e.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/p
andas-docs/stable/user_guide/indexing.html#returning-a-view-versu
s-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guid
e/indexing.html#returning-a-view-versus-a-copy)
  if feat[1] == 'numerical': in_fit[feat[0]] = df_adj_sub90[feat
[0]]
<ipython-input-457-17ce1928e433>:38: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFram
e.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/p
andas-docs/stable/user_guide/indexing.html#returning-a-view-versu
s-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guid
```

In [458]:
```
# Make new X DataFrame using 10% of the data
y_sub10 = df_adj_sub10['ln_price']

in_fit = df_adj_sub10[['ln_price']]
for feat in features:
    if feat[2] == 1:

        if feat[1] == 'numerical': in_fit[feat[0]] = df_adj_sub10[feat[

        if feat[1] == 'num-cat':
            in_fit['has_'+feat[0]] = df_adj_sub10[feat[0]].apply(lambda
            in_fit[feat[0]] = df_adj_sub10[feat[0]] * in_fit['has_'+fea

        if feat[1] == 'num-cat2':
            in_fit['has_'+feat[0]] = df_adj_sub10[feat[0]].apply(lambda
            #in_fit[feat[0]] = df_adj_sub10[feat[0]] * in_fit['has_'+fe

        if feat[1] == 'categorical':
            in_fit[feat[0]] = df_adj_sub10[feat[0]]
            in_fit = pd.get_dummies(in_fit, columns=[feat[0]], drop_fir

        if feat[1] == 'YN': in_fit['has_'+feat[0]] = df_adj_sub10[feat[

X_sub10 = in_fit.drop(columns = 'ln_price')
```

```
<ipython-input-458-3059a7946b60>:8: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas
-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
(https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html
#returning-a-view-versus-a-copy)
  if feat[1] == 'numerical': in_fit[feat[0]] = df_adj_sub10[feat[0]]
<ipython-input-458-3059a7946b60>:19: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas
-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
(https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html
#returning-a-view-versus-a-copy)
  in_fit[feat[0]] = df_adj_sub10[feat[0]]
```

In [460]:
```
# Compare predictions to actual prices for 10% of the data
price_ratio = np.array([price_predictor(results_sub90, np.array(X.loc[i
price_ratio = np.exp(price_ratio - y_sub10)
print(price_ratio)
```

```
6        0.746662
39       0.718372
48       1.071809
87       1.180133
94       1.266203
           ...
30094    0.707773
30096    1.011493
30101    0.867731
30145    1.487184
30151    0.756601
Name: ln_price, Length: 2828, dtype: float64
```

In [461]:
```
len(price_ratio[(price_ratio > 0.75) & (price_ratio < 1.25)]) / len(pri
```

Out[461]: 0.7507072135785007

About 75% of the predictions were within 75% to 125% of the actual price.

In [463]:
```
lnprice = np.array([price_predictor(results_sub90, np.array(X.loc[i]))[
+ np.log(np.median(df['price']))
```

In [464]:

```
# Scatter plot of predicted and real prices
fig, ax = plt.subplots( figsize = (9,9) )

ax.scatter(lnprice, y_sub10 + np.log(np.median(df['price'])), color = (

ax.plot([np.min(lnprice),np.max(lnprice)], [np.min(lnprice)+np.log(1.25
        color = (0.0,0.7,0.0,1.0), label='25% too high')

ax.plot([np.min(lnprice),np.max(lnprice)], [np.min(lnprice),np.max(lnpr
        label='exact match')

ax.plot([np.min(lnprice),np.max(lnprice)], [np.min(lnprice)+np.log(0.75
        color = (0.7,0.0,0.0,1.0), label='25% too low')

ax.set_xlabel('Predicted Price (Millions of $)', fontsize = 16)
ax.set_ylabel('Real Price (Millions of $)', fontsize = 16)
ax.set_title('Predicted Price vs. Actual Price')
ax.legend(loc='upper left')

log_ticks = np.append(\
                      np.append(\
                                np.append(\
                                          np.linspace(10**4,9*10**4,9),
                                         ),np.linspace(10**6,9*10**6,9)
                                ),np.linspace(10**7,9*10**7,9)\
                      )

ax.set_xticks(np.log(log_ticks))
ax.set_xticklabels([x/10**6 if i%1 == 0 else '' for i,x in enumerate(lo
ax.set_yticks(np.log(log_ticks))
ax.set_yticklabels([x/10**6 if i%1 == 0 else '' for i,x in enumerate(lo
ax.set_xbound([np.log(200000),np.log(6000000)])
ax.set_ybound([np.log(60000),np.log(20000000)])
```

The plot above shows that the model does do a solid job of estimating prices most of the time, but there are houses for which the predictions are off by a very large amount. This means that there are factors determining the values of houses that are not included in the data that can potentially be very important. The model is still a solid starting point for estimating the value of a house in King County, WA.

Test: remove suspiciously low price houses

Earlier, I pointed out that some of the house prices were unrealistically low and searching the addresses on Zillow showed that some were just plain wrong. Below, I rerun the fit without the suspiciously low prices.

In [486]:

```
# Choose features to include
# 1 is included, 0 is excluded

features = [['ln_sqft_living','numerical',1], \
            ['ln_sqft_lot','numerical',0], \
            ['ln_sqft_basement','num-cat',0], \
            ['ln_sqft_garage','num-cat',0], \
            ['ln_sqft_patio','num-cat',0], \
            ['bedrooms','numerical',0], \
            ['bathrooms','numerical',0], \
            ['floors','numerical',0], \
            ['yr_built','numerical',0], \
            ['condition','categorical',1], \
            ['grade','numerical',1], \
            ['waterfront','YN',1], \
            ['greenbelt','YN',0], \
            ['nuisance','YN',0], \
            ['view','categorical',0], \
            ['zip_id','categorical',1], \
            ['yr_renovated','num-cat2',1]]

# Use data excluding the very low prices
df_cut_low = df_adj[df_adj['ln_price'] > -2.16]  # Cut any price below

y_cut_low = df_cut_low['ln_price']

in_fit = df_cut_low[['ln_price']]
for feat in features:
    if feat[2] == 1:

        if feat[1] == 'numerical': in_fit[feat[0]] = df_cut_low[feat[0]

        if feat[1] == 'num-cat':
            in_fit['has_'+feat[0]] = df_cut_low[feat[0]].apply(lambda x
            in_fit[feat[0]] = df_cut_low[feat[0]] * in_fit['has_'+feat[

        if feat[1] == 'num-cat2':
            in_fit['has_'+feat[0]] = df_cut_low[feat[0]].apply(lambda x
            #in_fit[feat[0]] = df_cut_low[feat[0]] * in_fit['has_'+feat

        if feat[1] == 'categorical':
            in_fit[feat[0]] = df_cut_low[feat[0]]
            in_fit = pd.get_dummies(in_fit, columns=[feat[0]], drop_fir

        if feat[1] == 'YN': in_fit['has_'+feat[0]] = df_cut_low[feat[0]

X_cut_low = in_fit.drop(columns = 'ln_price')
results_cut_low = sm.OLS(y_cut_low,sm.add_constant(X_cut_low)).fit()
results_cut_low.summary()
```

```
<ipython-input-486-89a8155bc5bf>:31: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFram
e.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/p
andas-docs/stable/user_guide/indexing.html#returning-a-view-versu
s-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guid
e/indexing.html#returning-a-view-versus-a-copy)
  if feat[1] == 'numerical': in_fit[feat[0]] = df_cut_low[feat
[0]]
<ipython-input-486-89a8155bc5bf>:42: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFram
e.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/p
andas-docs/stable/user_guide/indexing.html#returning-a-view-versu
s-a-copy (https://pandas.pydata.org/pandas-docs/stable/user_guid
```

All of the parameters in this fit are similar to the parameters in the fit without cutting the low price data. They all fall within the 95% confidence intervals of the original fit.

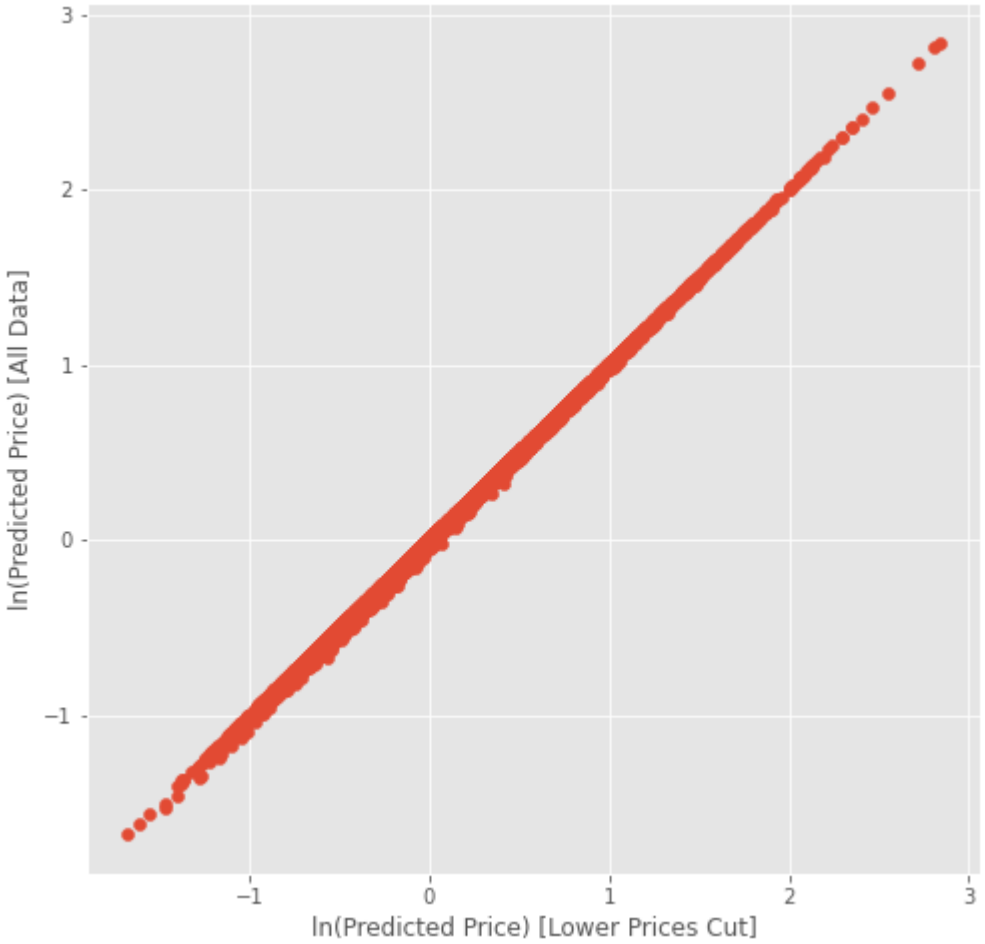
The adjusted R-squared did improve from 0.685 to 0.723, but this is not surprising or even a strong indicator that the fit is better. Removing the houses with very low prices removed the points that were contributing to the overall variance in price without reducing the variance explained by the model. The R-squared values had to increase.

This similarity between this model and the model using all the data shows that it is not necessary to remove these points as it will not have a statistically significant effect to do so.

In [492]:
```
pred_results_cut_low = np.array([price_predictor(results_cut_low, np.ar
pred_results = np.array([price_predictor(results, np.array(X.loc[i]))[0
```

In [495]:
```
fig, ax = plt.subplots(figsize = (8,8))

ax.scatter(pred_results_cut_low, pred_results)
ax.set_xlabel('ln(Predicted Price) [Lower Prices Cut]')
ax.set_ylabel('ln(Predicted Price) [All Data]')
```

Out[495]: Text(0, 0.5, 'ln(Predicted Price) [All Data]')



The predictions are super similar for both models (the one with all the data and the one cutting low price data). This means it is not worth it to sift through the data removing the incorrect prices because it won't make a significant difference anyway.