housing_price_regression

November 11, 2022

0.1 House Price Regression Model

Please fill out: * **Student name**: David Boyd * **Student pace**: self paced * **Scheduled project review date/time**: N/A * **Instructor name**: Abhineet Kulkarni

[45]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import datetime

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import sklearn.metrics as metrics
from sklearn.metrics import r2_score
from sklearn.metrics import mean_squared_error

from scipy import stats
from scipy.stats import skew, boxcox_normmax
from scipy.special import boxcox1p, inv_boxcox1p

import statsmodels.api as sm

%matplotlib inline
```

0.2 Useful Functions

Below are some functions that will be used later on, just grouping them at the top for readability purposes

[2]:
```
def diagnostic_plots(df, feature):
        '''
        Plots Histogram, Quartile Plot(Q-Q) and Boxplot
        '''
        # define figure size
        plt.figure(figsize=(16, 4))
```

```
        # histogram
        plt.subplot(1, 3, 1)
        sns.histplot(df[feature], bins=25)
        plt.title('Histogram')

        # Q-Q plot
        plt.subplot(1, 3, 2)
        stats.probplot(df[feature], dist="norm", plot=plt)
        plt.ylabel('Feature quantiles')

        # boxplot
        plt.subplot(1, 3, 3)
        sns.boxplot(y=df[feature])
        plt.title('Boxplot')

        plt.show()
```

[3]:
```
def adjust_skewness(df):
        """
        Function takes in a dataframe and returns a dataframe which isn't␣
 ↪impacted by skewness anymore
        """
        ## Getting all the data that are not of "object" type.
        numeric = df.dtypes[df.dtypes != "object"].index

        # Check the skew of all numerical features
        skewed_feats = df[numeric].apply(lambda x: skew(x)).
 ↪sort_values(ascending=False)
        high_skew = skewed_feats[abs(skewed_feats) > 0.5]
        skewed_features = high_skew.index

        for feat in skewed_features:
            df[feat] = boxcox1p(df[feat], boxcox_normmax(df[feat] + 1))
            print(f'for {feat} the lambda value is {boxcox_normmax(df[feat] +␣
 ↪1)}')
```

0.3 EDA Phase

First things first, we want to explore the dataset that we have been given, this means: - Understanding the features we've been given, the amount and their datatype - Identifying what features are categorical and whether we need to OHE them - Handling any null values - Identifying and handling any outliers - Understanding what are the options for each categorical variable

[4]:
```
df = pd.read_csv('data/kc_house_data.csv')
df.info()
```

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

[5]:
```
df.head()
```

[5]:
```
           id        date   price  bedrooms  bathrooms  sqft_living  sqft_lot  \
0  7129300520  10/13/2014  221900         3       1.00         1180      5650   
1  6414100192   12/9/2014  538000         3       2.25         2570      7242   
2  5631500400   2/25/2015  180000         2       1.00          770     10000   
3  2487200875   12/9/2014  604000         4       3.00         1960      5000   
4  1954400510   2/18/2015  510000         3       2.00         1680      8080   

   floors waterfront  view  …          grade sqft_above  sqft_basement  \
0     1.0        NaN  NONE  …      7 Average       1180              0   
1     2.0         NO  NONE  …      7 Average       2170            400   
2     1.0         NO  NONE  …  6 Low Average        770              0   
3     1.0         NO  NONE  …      7 Average       1050            910   
4     1.0         NO  NONE  …         8 Good       1680              0   

  yr_built  yr_renovated  zipcode      lat     long  sqft_living15  sqft_lot15  
0     1955           0.0    98178  47.5112 -122.257           1340        5650  
```

```
1      1951         1991.0    98125  47.7210 -122.319            1690         7639
2      1933            NaN    98028  47.7379 -122.233            2720         8062
3      1965            0.0    98136  47.5208 -122.393            1360         5000
4      1987            0.0    98074  47.6168 -122.045            1800         7503

[5 rows x 21 columns]
```

[6]:
```python
# counting the number of null values per column
df.isnull().sum()
```

[6]:
```
id                   0
date                 0
price                0
bedrooms             0
bathrooms            0
sqft_living          0
sqft_lot             0
floors               0
waterfront        2376
view                63
condition            0
grade                0
sqft_above           0
sqft_basement        0
yr_built             0
yr_renovated      3842
zipcode              0
lat                  0
long                 0
sqft_living15        0
sqft_lot15           0
dtype: int64
```

Looking at the above, we can see there are null values in yr_renovated, view, waterfront which need to be handled before potentially using them in a model. It is also clear that certain fields won't be useful to the model, these are lat, long, date and id

[7]:
```python
df.describe()
```

[7]:
```
                 id         price      bedrooms     bathrooms   sqft_living  \
count  2.159700e+04  2.159700e+04  21597.000000  21597.000000  21597.000000
mean   4.580474e+09  5.402966e+05      3.373200      2.115826   2080.321850
std    2.876736e+09  3.673681e+05      0.926299      0.768984    918.106125
min    1.000102e+06  7.800000e+04      1.000000      0.500000    370.000000
25%    2.123049e+09  3.220000e+05      3.000000      1.750000   1430.000000
50%    3.904930e+09  4.500000e+05      3.000000      2.250000   1910.000000
75%    7.308900e+09  6.450000e+05      4.000000      2.500000   2550.000000
```

```
max    9.900000e+09  7.700000e+06     33.000000      8.000000  13540.000000

           sqft_lot        floors    sqft_above      yr_built  yr_renovated  \
count  2.159700e+04  21597.000000  21597.000000  21597.000000  17755.000000
mean   1.509941e+04      1.494096   1788.596842   1970.999676     83.636778
std    4.141264e+04      0.539683    827.759761     29.375234    399.946414
min    5.200000e+02      1.000000    370.000000   1900.000000      0.000000
25%    5.040000e+03      1.000000   1190.000000   1951.000000      0.000000
50%    7.618000e+03      1.500000   1560.000000   1975.000000      0.000000
75%    1.068500e+04      2.000000   2210.000000   1997.000000      0.000000
max    1.651359e+06      3.500000   9410.000000   2015.000000   2015.000000

            zipcode           lat          long  sqft_living15     sqft_lot15
count  21597.000000  21597.000000  21597.000000   21597.000000   21597.000000
mean   98077.951845     47.560093   -122.213982    1986.620318   12758.283512
std       53.513072      0.138552      0.140724     685.230472   27274.441950
min    98001.000000     47.155900   -122.519000     399.000000     651.000000
25%    98033.000000     47.471100   -122.328000    1490.000000    5100.000000
50%    98065.000000     47.571800   -122.231000    1840.000000    7620.000000
75%    98118.000000     47.678000   -122.125000    2360.000000   10083.000000
max    98199.000000     47.777600   -121.315000    6210.000000  871200.000000
```

Looking at the describe function above, we can see some clear outliers across several features, which will need handling before building an effective model

[8]:
```
df_cat = df[['waterfront', 'view', 'condition', 'grade', 'yr_renovated',␣
 ↪'sqft_basement']]
df_cat.head()
```

[8]:
```
  waterfront  view  condition          grade  yr_renovated sqft_basement
0        NaN  NONE    Average      7 Average           0.0             0
1         NO  NONE    Average      7 Average        1991.0           400
2         NO  NONE    Average  6 Low Average           NaN             0
3         NO  NONE  Very Good      7 Average           0.0           910
4         NO  NONE    Average         8 Good           0.0             0
```

[9]:
```
for col in df_cat.columns:
    print(df_cat[col].unique()) # to print categories name only
    print(df_cat[col].value_counts()) # to print count of every category
```

```
[nan 'NO' 'YES']
NO     19075
YES      146
Name: waterfront, dtype: int64
['NONE' nan 'GOOD' 'EXCELLENT' 'AVERAGE' 'FAIR']
NONE         19422
AVERAGE        957
GOOD           508
```

```
FAIR            330
EXCELLENT       317
Name: view, dtype: int64
['Average' 'Very Good' 'Good' 'Poor' 'Fair']
Average      14020
Good          5677
Very Good     1701
Fair           170
Poor            29
Name: condition, dtype: int64
['7 Average' '6 Low Average' '8 Good' '11 Excellent' '9 Better' '5 Fair'
 '10 Very Good' '12 Luxury' '4 Low' '3 Poor' '13 Mansion']
7 Average        8974
8 Good           6065
9 Better         2615
6 Low Average    2038
10 Very Good     1134
11 Excellent      399
5 Fair            242
12 Luxury          89
4 Low              27
13 Mansion         13
3 Poor              1
Name: grade, dtype: int64
[   0. 1991.   nan 2002. 2010. 1992. 2013. 1994. 1978. 2005. 2003. 1984.
 1954. 2014. 2011. 1983. 1945. 1990. 1988. 1977. 1981. 1995. 2000. 1999.
 1998. 1970. 1989. 2004. 1986. 2007. 1987. 2006. 1985. 2001. 1980. 1971.
 1979. 1997. 1950. 1969. 1948. 2009. 2015. 1974. 2008. 1968. 2012. 1963.
 1951. 1962. 1953. 1993. 1996. 1955. 1982. 1956. 1940. 1976. 1946. 1975.
 1964. 1973. 1957. 1959. 1960. 1967. 1965. 1934. 1972. 1944. 1958.]
0.0       17011
2014.0       73
2003.0       31
2013.0       31
2007.0       30
          …
1946.0        1
1959.0        1
1971.0        1
1951.0        1
1954.0        1
Name: yr_renovated, Length: 70, dtype: int64
['0' '400' '910' '1530' '?' '730' '1700' '300' '970' '760' '720' '700'
 '820' '780' '790' '330' '1620' '360' '588' '1510' '410' '990' '600' '560'
 '550' '1000' '1600' '500' '1040' '880' '1010' '240' '265' '290' '800'
 '540' '710' '840' '380' '770' '480' '570' '1490' '620' '1250' '1270'
 '120' '650' '180' '1130' '450' '1640' '1460' '1020' '1030' '750' '640'
 '1070' '490' '1310' '630' '2000' '390' '430' '850' '210' '1430' '1950'
```

```
 '440' '220' '1160' '860' '580' '2060' '1820' '1180' '200' '1150' '1200'
 '680' '530' '1450' '1170' '1080' '960' '280' '870' '1100' '460' '1400'
 '660' '1220' '900' '420' '1580' '1380' '475' '690' '270' '350' '935'
 '1370' '980' '1470' '160' '950' '50' '740' '1780' '1900' '340' '470'
 '370' '140' '1760' '130' '520' '890' '1110' '150' '1720' '810' '190'
 '1290' '670' '1800' '1120' '1810' '60' '1050' '940' '310' '930' '1390'
 '610' '1830' '1300' '510' '1330' '1590' '920' '1320' '1420' '1240' '1960'
 '1560' '2020' '1190' '2110' '1280' '250' '2390' '1230' '170' '830' '1260'
 '1410' '1340' '590' '1500' '1140' '260' '100' '320' '1480' '1060' '1284'
 '1670' '1350' '2570' '1090' '110' '2500' '90' '1940' '1550' '2350' '2490'
 '1481' '1360' '1135' '1520' '1850' '1660' '2130' '2600' '1690' '243'
 '1210' '1024' '1798' '1610' '1440' '1570' '1650' '704' '1910' '1630'
 '2360' '1852' '2090' '2400' '1790' '2150' '230' '70' '1680' '2100' '3000'
 '1870' '1710' '2030' '875' '1540' '2850' '2170' '506' '906' '145' '2040'
 '784' '1750' '374' '518' '2720' '2730' '1840' '3480' '2160' '1920' '2330'
 '1860' '2050' '4820' '1913' '80' '2010' '3260' '2200' '415' '1730' '652'
 '2196' '1930' '515' '40' '2080' '2580' '1548' '1740' '235' '861' '1890'
 '2220' '792' '2070' '4130' '2250' '2240' '1990' '768' '2550' '435' '1008'
 '2300' '2610' '666' '3500' '172' '1816' '2190' '1245' '1525' '1880' '862'
 '946' '1281' '414' '2180' '276' '1248' '602' '516' '176' '225' '1275'
 '266' '283' '65' '2310' '10' '1770' '2120' '295' '207' '915' '556' '417'
 '143' '508' '2810' '20' '274' '248']
0       12826
?         454
600       217
500       209
700       208
        …
1275        1
225         1
243         1
274         1
875         1
Name: sqft_basement, Length: 304, dtype: int64
```

We can see multiple issues inside our categorical variables, these are: - sqft_basement having a ? option, these need removing - yr_renovated having a nan option - Extracting/converting the grade & condition features into numerics

0.4 Preprocessing Stage

Looking at the above tables there are several steps to complete in this section, which have been listed below: - Clean up the missing values on the yr_renovated column, assume is NAN then it hasn't been renovated, so 0 will be inputted. - Using the date, pull out the yr_sold to help calculate the age of the property at sale, years since renovation - Create booleans to identify if it has had a renovation, or if the property has a basement, to show roughly what extra value they add to a property (considered with the age/size of each feature later on)

Once the steps above have been completed we are able to look at the correlation between different features with the price. This will help us determine which features we are able to drop, then can begin on encoding any categorical columns left.

[10]:
```
df['date'] = pd.to_datetime(df['date'])
df['yr_sold'] = df['date'].dt.year

# calculating property age
df['prop_age'] = abs(df['yr_sold'] - df['yr_built'])
# filling missing renovated values with 0, as assumed they didn't get renovated
df['yr_renovated'].fillna(0, inplace=True)
```

[11]:
```
# calculating number of years since upgrade/renovation
df['yrs_since_upgrade'] = np.where(df['yr_renovated'] == 0, df['prop_age'],
                          np.where(df['yr_renovated'] != 0, abs(df['yr_sold'] -␣
 ↪df['yr_renovated']), 0))

# boolean of whether a renovation happened or not
df['had_upgrade'] = np.where(df['yr_renovated'] == 0, "No",
                          np.where(df['yr_renovated'] != 0, "Yes", "No"))

# boolean of whether it has a basement or not
df['has_basement'] = np.where(df['sqft_basement'].isin([0,'?']) , "No",
                          np.where(df['sqft_basement'] != 0, "Yes", "No"))

# calculating square footage of basement
df['sqft_basement'] = df['sqft_living'] - df['sqft_above']
```

[12]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 26 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   id                 21597 non-null  int64
 1   date               21597 non-null  datetime64[ns]
 2   price              21597 non-null  int64
 3   bedrooms           21597 non-null  int64
 4   bathrooms          21597 non-null  float64
 5   sqft_living        21597 non-null  int64
 6   sqft_lot           21597 non-null  int64
 7   floors             21597 non-null  float64
 8   waterfront         19221 non-null  object
 9   view               21534 non-null  object
 10  condition          21597 non-null  object
 11  grade              21597 non-null  object
 12  sqft_above         21597 non-null  int64
```

```
 13  sqft_basement      21597 non-null  int64
 14  yr_built           21597 non-null  int64
 15  yr_renovated       21597 non-null  float64
 16  zipcode            21597 non-null  int64
 17  lat                21597 non-null  float64
 18  long               21597 non-null  float64
 19  sqft_living15      21597 non-null  int64
 20  sqft_lot15         21597 non-null  int64
 21  yr_sold            21597 non-null  int64
 22  prop_age           21597 non-null  int64
 23  yrs_since_upgrade  21597 non-null  float64
 24  had_upgrade        21597 non-null  object
 25  has_basement       21597 non-null  object
dtypes: datetime64[ns](1), float64(6), int64(13), object(6)
memory usage: 4.3+ MB
```

[12]:
```
# Dropping all unneccesary columns
df = df.drop(['id','date', 'waterfront', 'view', 'zipcode', 'lat', 'long',␣
 ↪'yr_renovated', 'sqft_living', 'sqft_lot', 'yr_built', 'yr_sold'], axis=1)
```

[13]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 14 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   price              21597 non-null  int64
 1   bedrooms           21597 non-null  int64
 2   bathrooms          21597 non-null  float64
 3   floors             21597 non-null  float64
 4   condition          21597 non-null  object
 5   grade              21597 non-null  object
 6   sqft_above         21597 non-null  int64
 7   sqft_basement      21597 non-null  int64
 8   sqft_living15      21597 non-null  int64
 9   sqft_lot15         21597 non-null  int64
 10  prop_age           21597 non-null  int64
 11  yrs_since_upgrade  21597 non-null  float64
 12  had_upgrade        21597 non-null  object
 13  has_basement       21597 non-null  object
dtypes: float64(3), int64(7), object(4)
memory usage: 2.3+ MB
```

[14]:
```
cond_dict = {'Poor':1, 'Fair':2, 'Average':3, 'Good':4, 'Very Good':5}

# converting condition into a numerical column using a dict
```

```
df.replace({"condition": cond_dict}, inplace=True)
df['condition'] = df['condition'].astype(int)
```

[15]:
```
grade_dict = {'3 Poor':3,
              '4 Low':4,
              '5 Fair':5,
              '6 Low Average':6,
              '7 Average':7,
              '8 Good':8,
              '9 Better':9,
              '10 Very Good':10,
              '11 Excellent':11,
              '12 Luxury':12,
              '13 Mansion':13}

# converting grade into a numerical column using a dict
df.replace({"grade": grade_dict}, inplace=True)
df['grade'] = df['grade'].astype(int)
df.info()
```

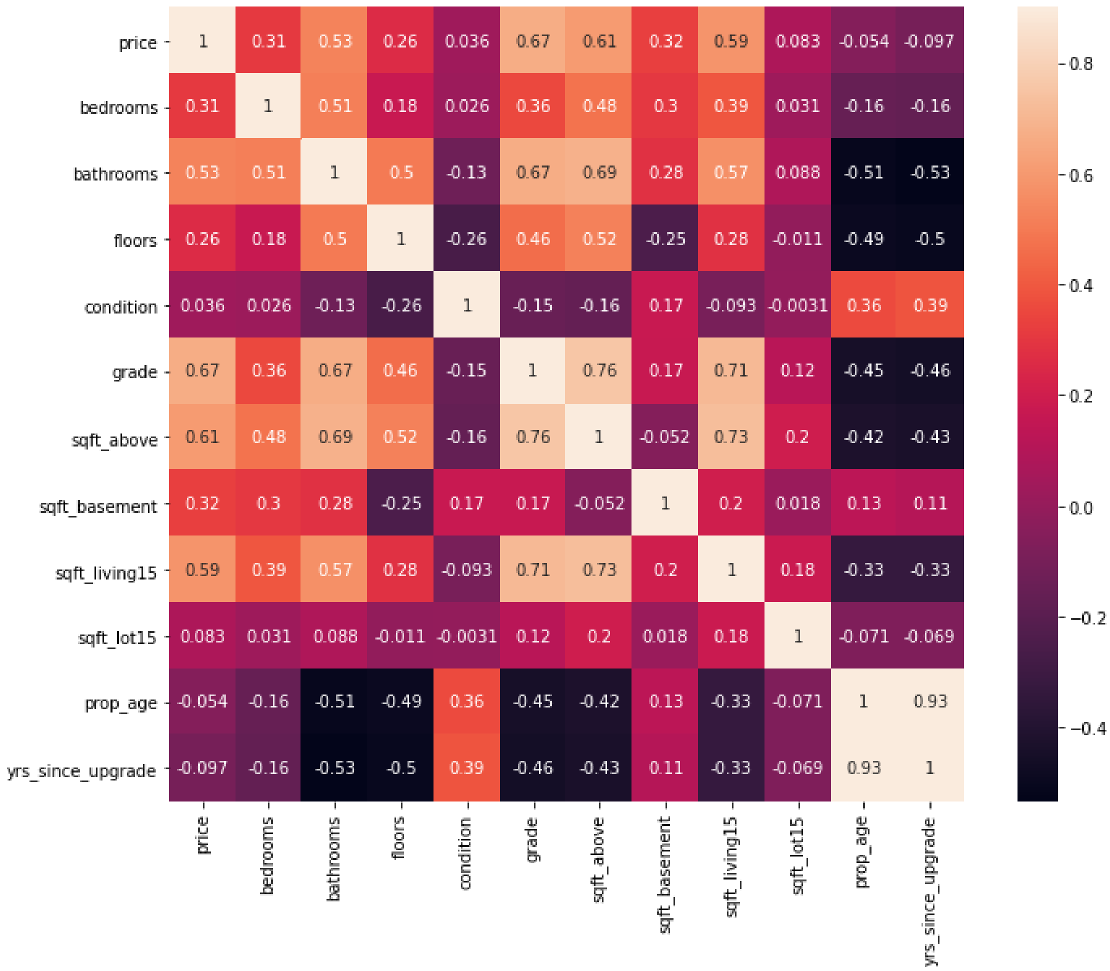
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 14 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   price              21597 non-null  int64
 1   bedrooms           21597 non-null  int64
 2   bathrooms          21597 non-null  float64
 3   floors             21597 non-null  float64
 4   condition          21597 non-null  int64
 5   grade              21597 non-null  int64
 6   sqft_above         21597 non-null  int64
 7   sqft_basement      21597 non-null  int64
 8   sqft_living15      21597 non-null  int64
 9   sqft_lot15         21597 non-null  int64
 10  prop_age           21597 non-null  int64
 11  yrs_since_upgrade  21597 non-null  float64
 12  had_upgrade        21597 non-null  object
 13  has_basement       21597 non-null  object
dtypes: float64(3), int64(9), object(2)
memory usage: 2.3+ MB
```

Time to visualise each of the numeric columns left to understand whether they are normally distributed or not.

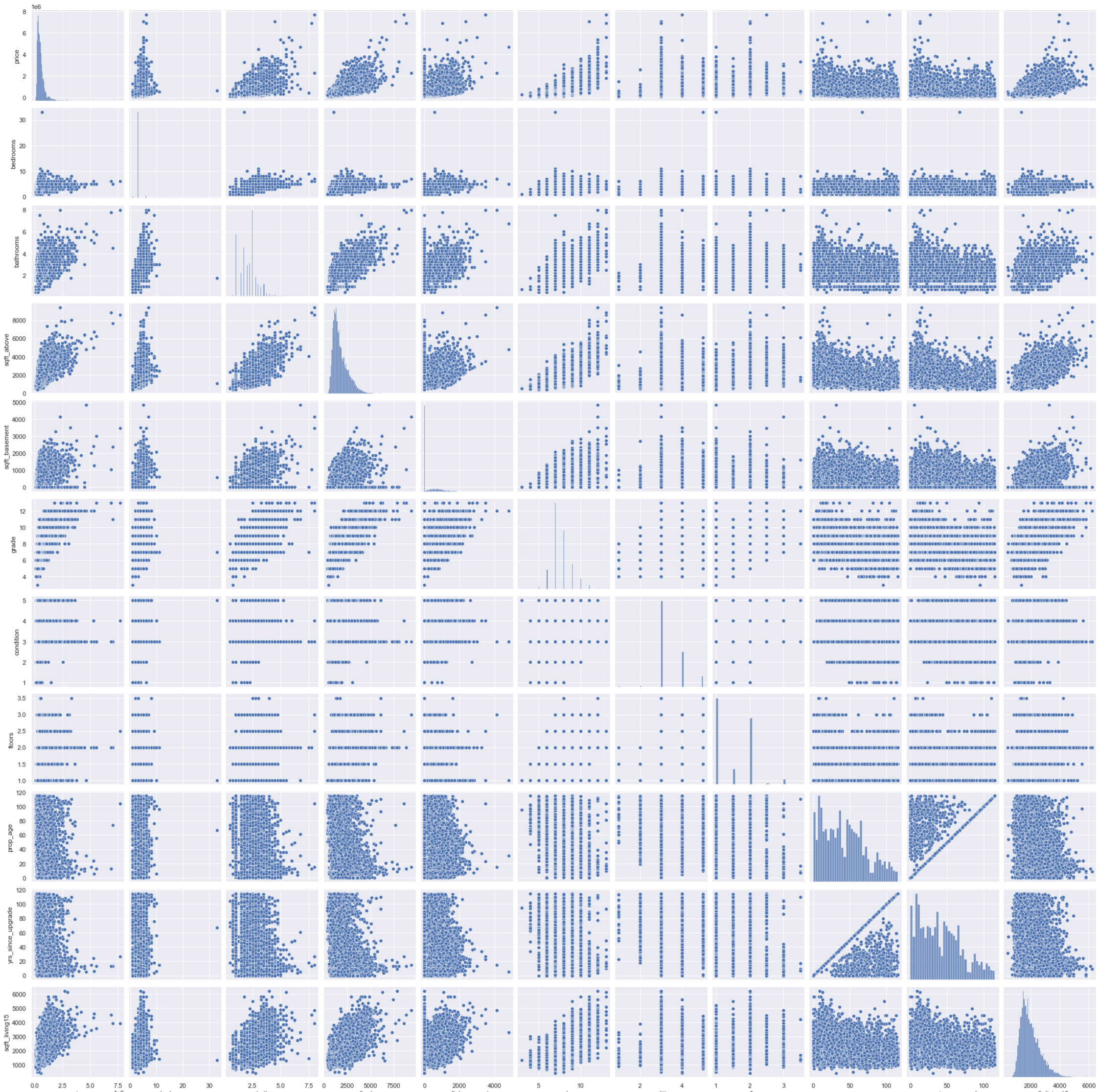
[16]:
```
# Plot the Correlation map to see how features are correlated with target: Price
corr_matrix = df.corr()
plt.subplots(figsize=(12,9))
```

```python
sns.heatmap(corr_matrix, vmax=0.9, square=True, annot=True)
```

[16]: <AxesSubplot:>



[17]:
```python
sns.set()
cols = ['price', 'bedrooms', 'bathrooms','sqft_above', 'sqft_basement'␣
↪,'grade','condition',
        'floors', 'prop_age', 'yrs_since_upgrade', 'sqft_living15']

# quickly visualising what relationship features have with each other
sns.pairplot(df[cols], height = 2.5)
plt.show();
```



0.5 Handling Outliers

After handling the outliers, we need to handle the following aspects: - Numeric data needs to be standardised - Categorical data needs to be one hot encoded/binary encoded - Transform the price to remove skewness

[18]:
```
df_numeric = df.select_dtypes(exclude = 'object')

# calculting the z-score for all numeric columns to exclude outliers
z_score = np.abs(stats.zscore(df_numeric))
no_outliers = (z_score < 3).all(axis = 1)
df_filtered = df_numeric[no_outliers]
```

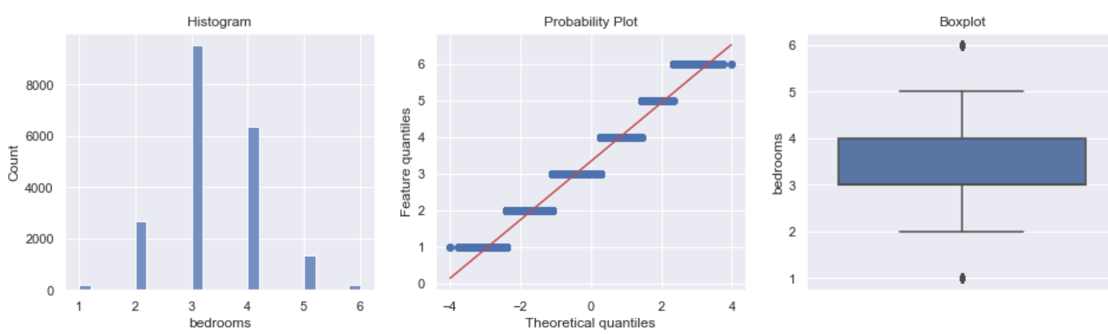
```
df_filtered.head()
```

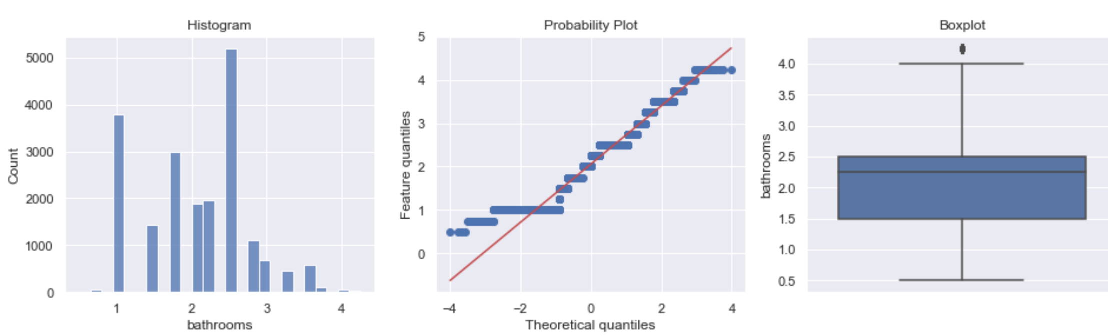
[18]:

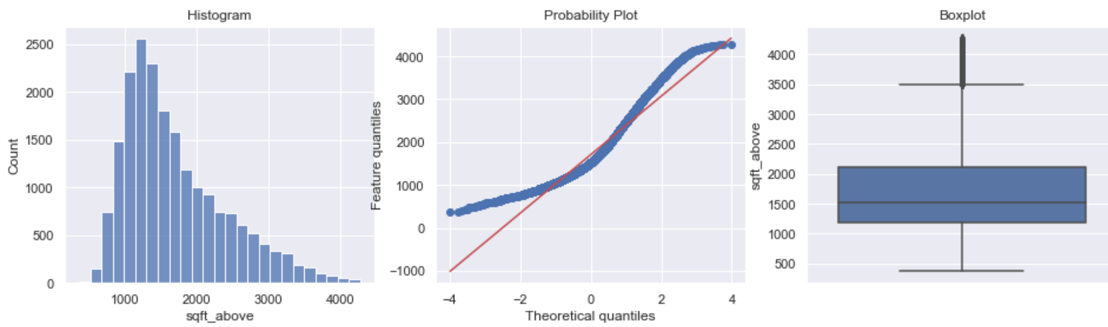
```
    price  bedrooms  bathrooms  floors  condition  grade  sqft_above  \
0  221900         3       1.00     1.0          3      7        1180
1  538000         3       2.25     2.0          3      7        2170
2  180000         2       1.00     1.0          3      6         770
3  604000         4       3.00     1.0          5      7        1050
4  510000         3       2.00     1.0          3      8        1680

   sqft_basement  sqft_living15  sqft_lot15  prop_age  yrs_since_upgrade
0              0           1340        5650        59               59.0
1            400           1690        7639        63               23.0
2              0           2720        8062        82               82.0
3            910           1360        5000        49               49.0
4              0           1800        7503        28               28.0
```

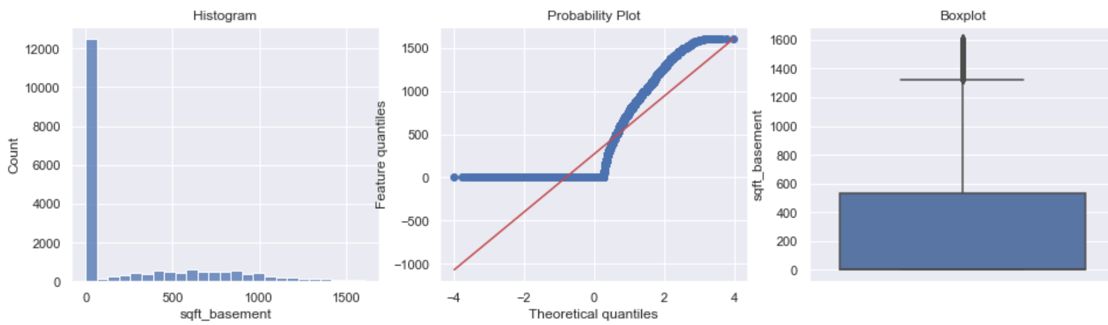
[19]:

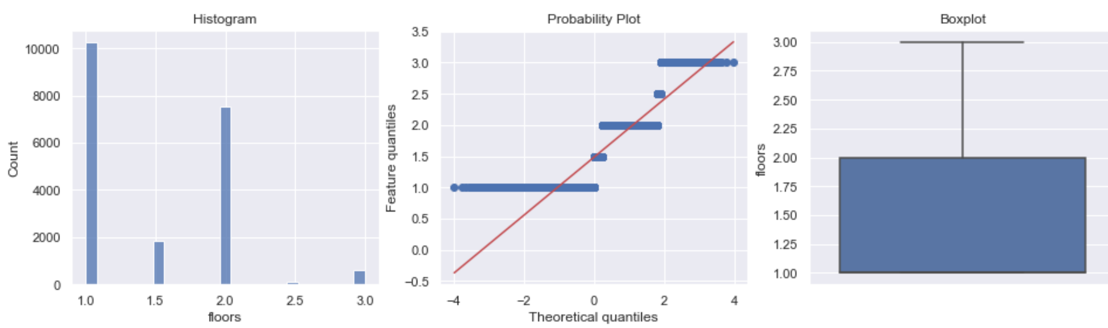
```
features = ['bedrooms', 'bathrooms', 'sqft_above', 'sqft_basement', 'floors',
       'condition', 'grade', 'prop_age', 'yrs_since_upgrade', 'sqft_living15']

# checking for whether features are linear or not and whether features are␣
 ↪skewed
for feat in features:
    diagnostic_plots(df_filtered,feat)
```

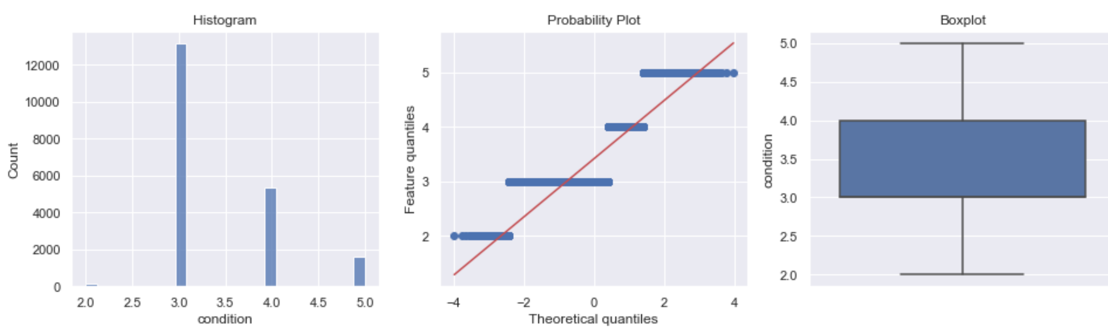


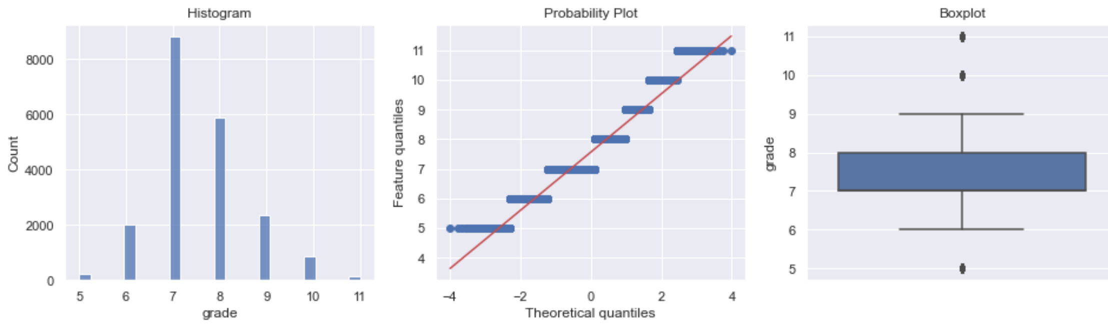


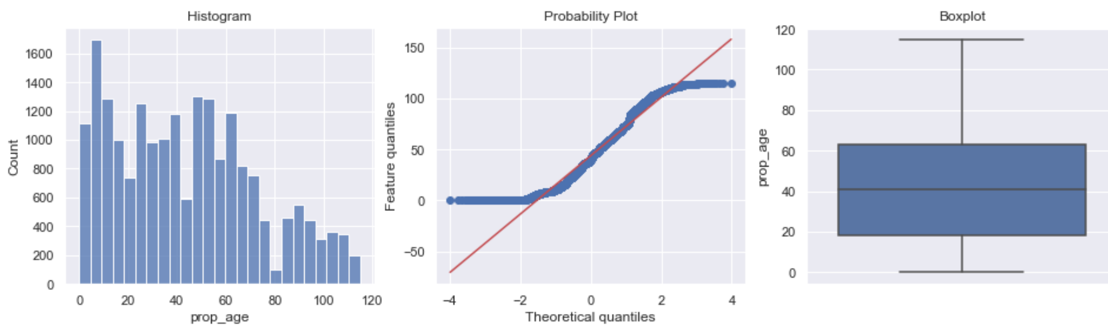

Histogram
Count
sqft_above
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
sqft_above

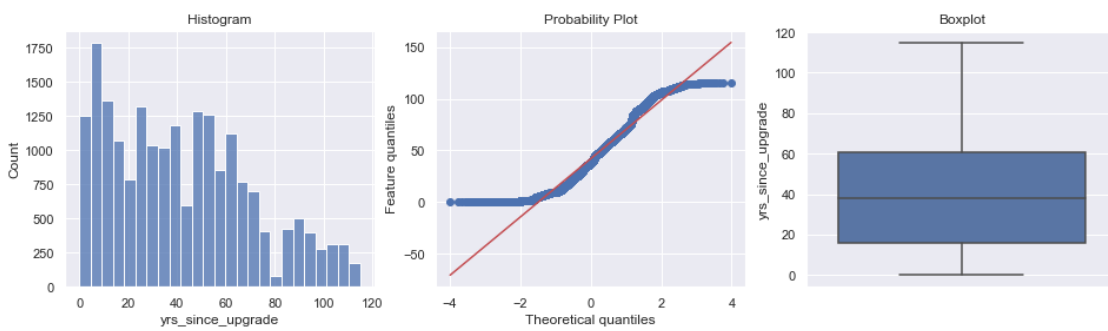

Histogram
Count
sqft_basement
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
sqft_basement

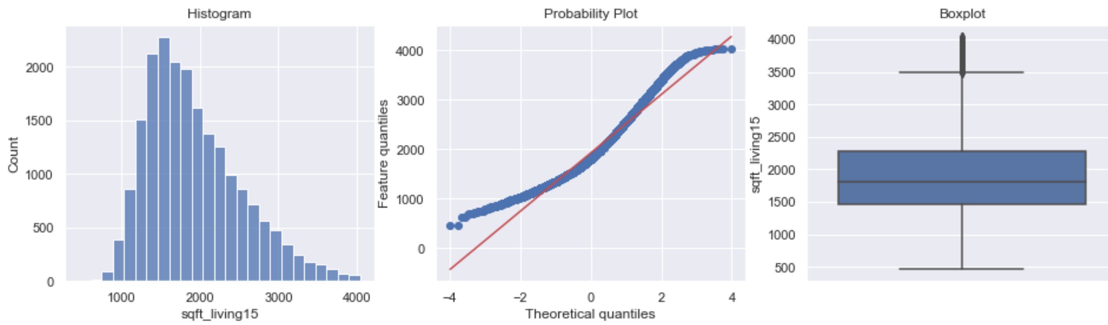

Histogram
Count
floors
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
floors


Histogram
Count
condition
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
condition


Histogram
Count
grade
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
grade


Histogram
Count
prop_age
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
prop_age


Histogram
Count
yrs_since_upgrade
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
yrs_since_upgrade



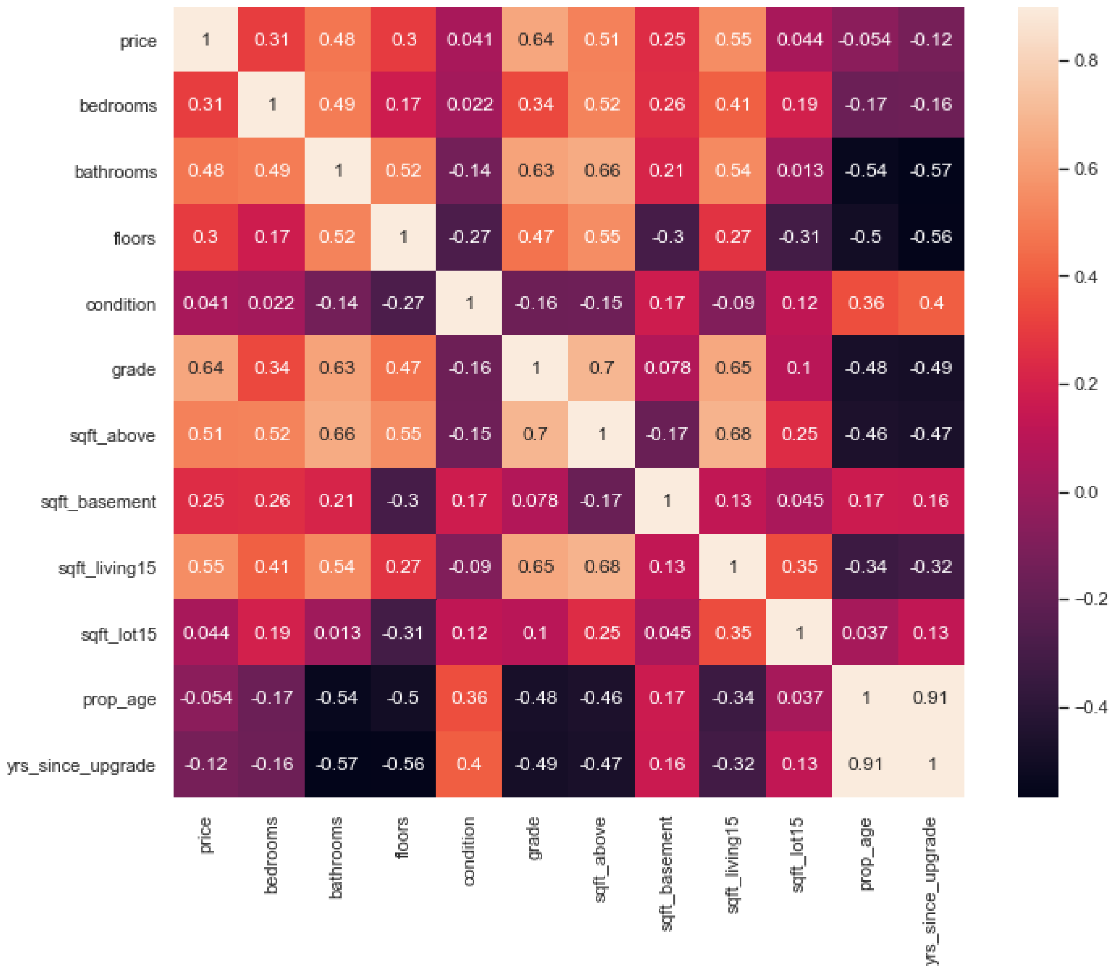

```
[20]: # removing skewness of features
      adjust_skewness(df_filtered)
```

```
/Users/davidboyd/opt/anaconda3/envs/learn-env/lib/python3.8/site-
packages/scipy/stats/stats.py:3845: PearsonRConstantInputWarning: An input array
is constant; the correlation coefficent is not defined.
  warnings.warn(PearsonRConstantInputWarning())
<ipython-input-3-f3db2a709ae3>:14: 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
  df[feat] = boxcox1p(df[feat], boxcox_normmax(df[feat] + 1))
```

```
for sqft_lot15 the lambda value is 1.1371715742807809
for price the lambda value is 0.9784635941282755
for sqft_basement the lambda value is 0.9987136051377098
for condition the lambda value is 0.9177283015573858
for sqft_above the lambda value is 0.7422674506937296
for sqft_living15 the lambda value is 0.9169760112006394
for floors the lambda value is 0.9978639576035712
for grade the lambda value is 1.0354208023687301
for yrs_since_upgrade the lambda value is 0.9889168982012952
```

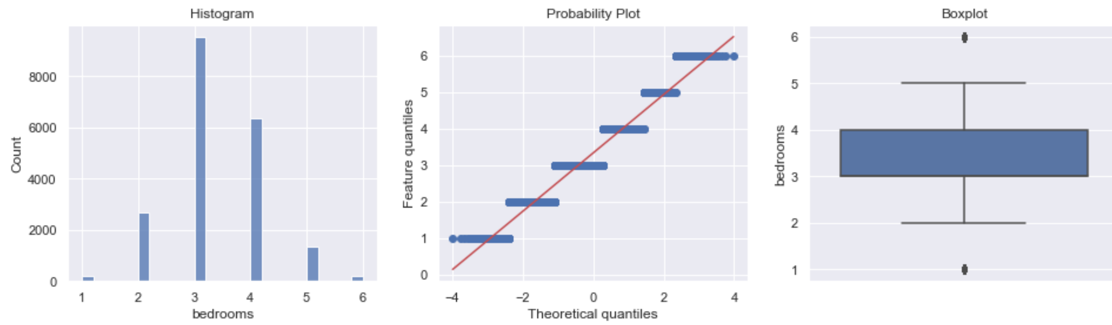
```
[21]: corr_matrix = df_filtered.corr()
      plt.subplots(figsize=(12,9))
      sns.heatmap(corr_matrix, vmax=0.9, square=True, annot=True)
```

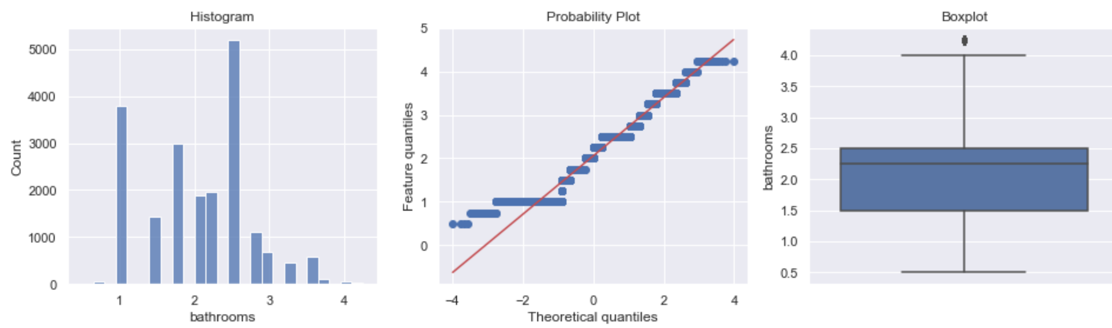
```
[21]: <AxesSubplot:>
```

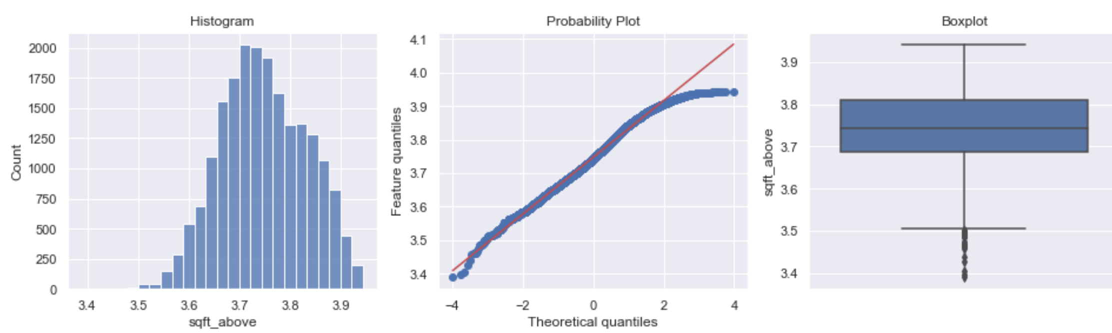


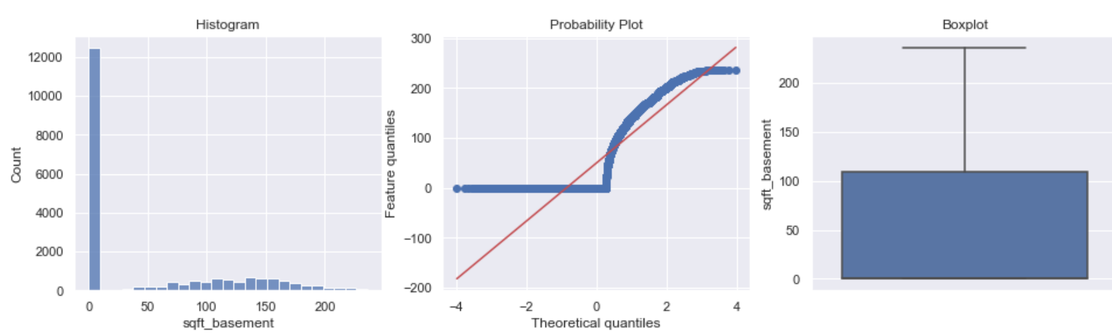
[22]:
```
X = df_filtered.drop('price', axis=1)
y = df_filtered['price']
```

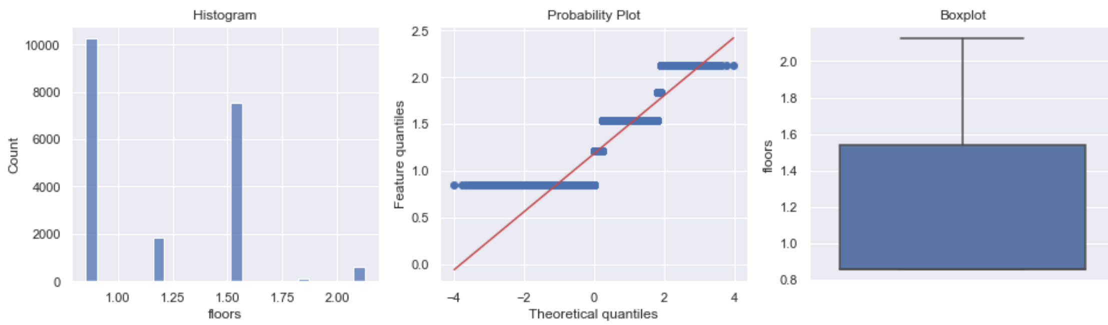
[23]:
```
features = ['bedrooms', 'bathrooms', 'sqft_above', 'sqft_basement', 'floors',
       'condition', 'grade', 'prop_age', 'yrs_since_upgrade', 'sqft_living15']

# checking features skew and linearity violations have been fixed via␣
↪transformations
for feat in features:
    diagnostic_plots(X,feat)
```

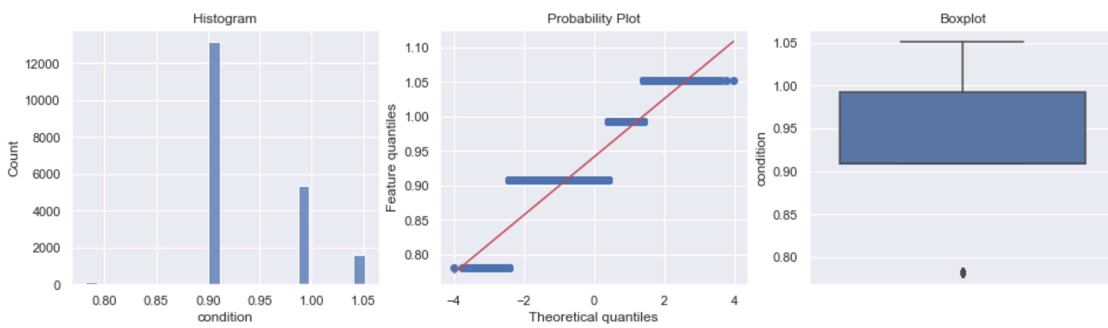

Histogram
Count
bedrooms
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
bedrooms

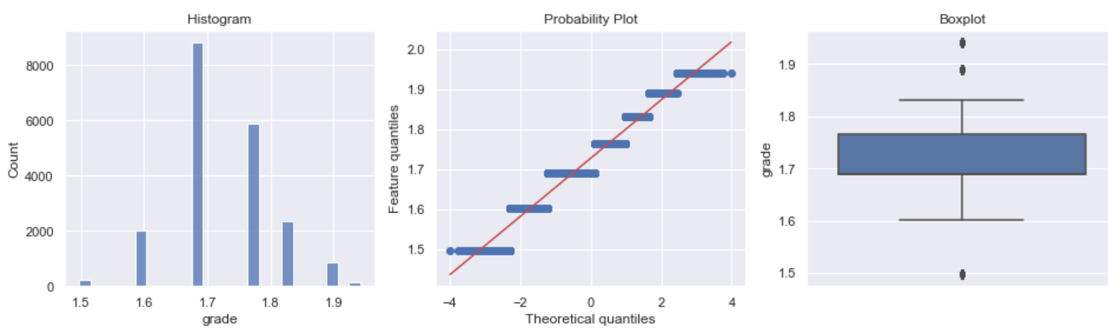

Histogram
Count
bathrooms
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
bathrooms

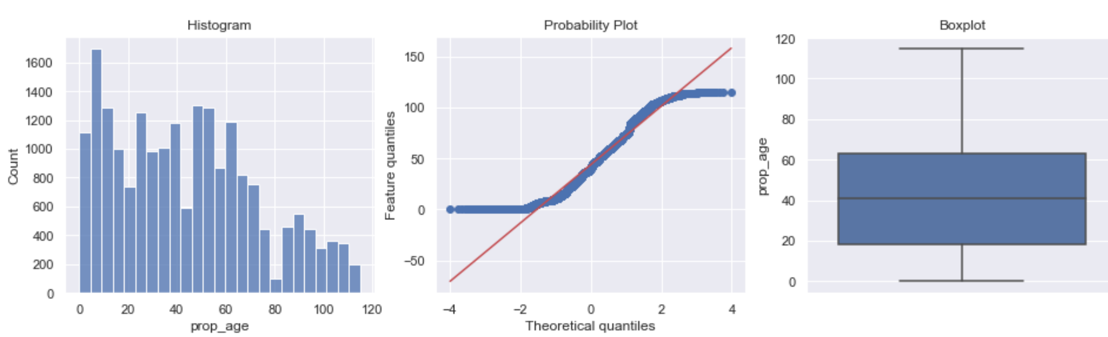

Histogram
Count
sqft_above
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
sqft_above

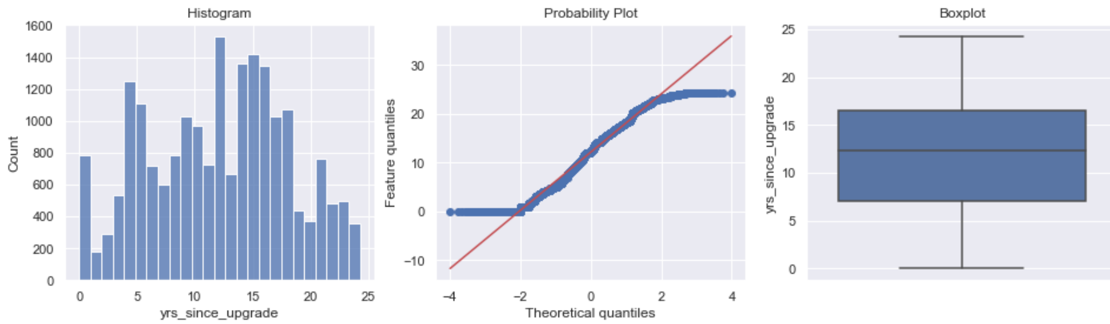

Histogram
Count
sqft_basement
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
sqft_basement

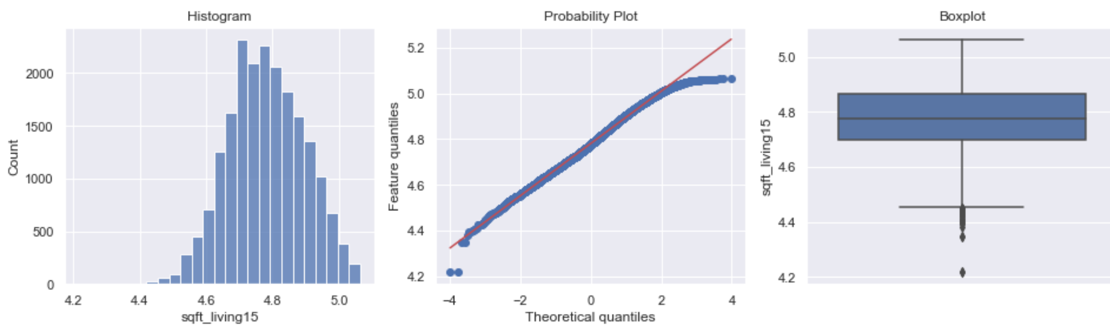

Histogram
Count
floors
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
floors


Histogram
Count
condition
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
condition

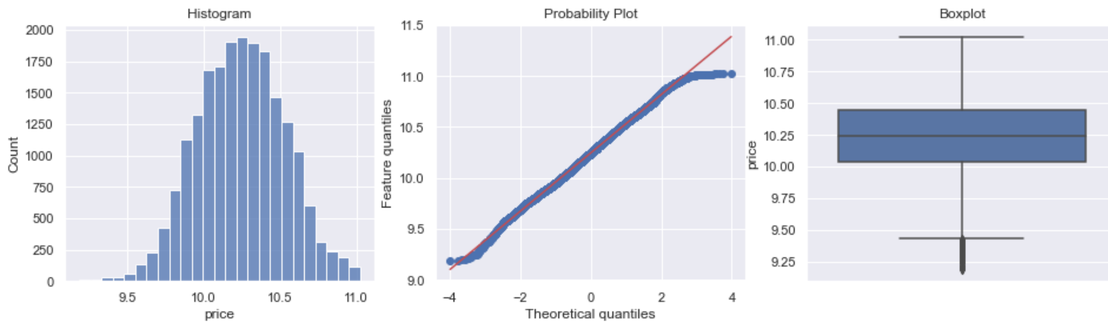

Histogram
Count
grade
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
grade


Histogram
Count
prop_age
Probability Plot
Feature quantiles
Theoretical quantiles
Boxplot
prop_age





[24]:
```
plt.figure(figsize=(16, 4))

        # histogram
plt.subplot(1, 3, 1)
sns.histplot(y, bins=25)
plt.title('Histogram')

        # Q-Q plot
plt.subplot(1, 3, 2)
stats.probplot(y, dist="norm", plot=plt)
plt.ylabel('Feature quantiles')

        # boxplot
plt.subplot(1, 3, 3)
sns.boxplot(y=y)
plt.title('Boxplot')

plt.show()
```




```
[25]: # dropping non-linear features
X = X.drop(['yrs_since_upgrade', 'prop_age', 'sqft_basement'], axis=1)
```

```
[26]: df_cat = df.loc[X.index]
df_cat = df_cat.select_dtypes(include = 'object')
df_cat.head()

from sklearn.preprocessing import LabelBinarizer

# Create a binarizer object for each binary categorical variable
upgrade_bin = LabelBinarizer()
basement_bin = LabelBinarizer()

# Fit and transform each respective binary cat variable to their respective␣
↪binarizer objects
df_cat['had_upgrade'] = upgrade_bin.fit_transform(df_cat['had_upgrade'])
df_cat['has_basement'] = basement_bin.fit_transform(df_cat['has_basement'])

df_cat.head()
```

[26]:

	had_upgrade	has_basement
0	0	1
1	1	1
2	0	1
3	0	1
4	0	1

```
[27]: # Merge numerical data and categorical data
X_prep = pd.merge(df_cat, X, left_index = True, right_index = True)
X_prep.head()
```

[27]:

	had_upgrade	has_basement	bedrooms	bathrooms	floors	condition \
0	0	1	3	1.00	0.853018	0.908772
1	1	1	3	2.25	1.535993	0.908772
2	0	1	2	1.00	0.853018	0.908772

```
3           0           1         4       3.00  0.853018    1.051467
4           0           1         3       2.00  0.853018    0.908772

      grade  sqft_above  sqft_living15  sqft_lot15
0  1.689392    3.686960       4.663428    6.828133
1  1.689392    3.816627       4.752436    7.011192
2  1.601396    3.585761       4.926768    7.043575
3  1.689392    3.660177       4.669191    6.753060
4  1.765015    3.764156       4.776161    7.000376
```

0.6 Regression Modelling

0.6.1 Baseline Model

[28]:
```python
X_train, X_test, y_train, y_test = train_test_split(X_prep['sqft_above'], y,␣
 ↪test_size=0.2, random_state=51)
```

[29]:
```python
from sklearn.linear_model import LinearRegression
regr = LinearRegression()
# Train the model using the training sets
regr.fit(X_train.values.reshape(-1,1), y_train)

# Make predictions using the testing set
y_test_pred = regr.predict(X_test.values.reshape(-1,1))
y_train_pred = regr.predict(X_train.values.reshape(-1,1))
```

[30]:
```python
print('MSE train: %.3f, test: %.3f' % (
        mean_squared_error(y_train, y_train_pred),
        mean_squared_error(y_test, y_test_pred)))
print('R^2 train: %.3f, test: %.3f' % (
        r2_score(y_train, y_train_pred),
        r2_score(y_test, y_test_pred)))
```

```
MSE train: 0.061, test: 0.059
R^2 train: 0.265, test: 0.249
```

[31]:
```python
# Create model intercept
feats_with_int = sm.add_constant(X_train)

# Fit model to data
model1 = sm.OLS(y_train,feats_with_int).fit()
model1.summary()
```

[31]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
```

```
==============================================================================
Dep. Variable:                  price   R-squared:                       0.265
Model:                            OLS   Adj. R-squared:                  0.265
Method:                 Least Squares   F-statistic:                     5848.
Date:                Wed, 09 Nov 2022   Prob (F-statistic):               0.00
Time:                        22:14:52   Log-Likelihood:                -352.54
No. Observations:               16236   AIC:                             709.1
Df Residuals:                   16234   BIC:                             724.5
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          3.7010      0.086     43.254      0.000       3.533       3.869
sqft_above     1.7460      0.023     76.474      0.000       1.701       1.791
==============================================================================
Omnibus:                      236.636   Durbin-Watson:                   1.965
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              140.295
Skew:                          -0.035   Prob(JB):                     3.43e-31
Kurtosis:                       2.550   Cond. No.                         177.
==============================================================================

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

0.6.2 Iteration 1 - All features

[32]:
```python
X_train1, X_test1, y_train1, y_test1 = train_test_split(X_prep, y, test_size=0.
 ↪2, random_state=51)
```

[33]:
```python
regr = LinearRegression()
# Train the model using the training sets
regr.fit(X_train1, y_train1)

# Make predictions using the testing set
y_test_pred1 = regr.predict(X_test1)
y_train_pred1 = regr.predict(X_train1)
```

[34]:
```python
print('MSE train: %.3f, test: %.3f' % (
        mean_squared_error(y_train1, y_train_pred1),
        mean_squared_error(y_test1, y_test_pred1)))
print('R^2 train: %.3f, test: %.3f' % (
        r2_score(y_train1, y_train_pred1),
        r2_score(y_test1, y_test_pred1)))
```

```
MSE train: 0.042, test: 0.042
R^2 train: 0.493, test: 0.474
```

[35]:
```python
# Create model intercept
feats_with_int1 = sm.add_constant(X_train1)

# Fit model to data
model2 = sm.OLS(y_train1,feats_with_int1).fit()
model2.summary()
```

[35]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.493
Model:                            OLS   Adj. R-squared:                  0.492
Method:                 Least Squares   F-statistic:                     1575.
Date:                Wed, 09 Nov 2022   Prob (F-statistic):               0.00
Time:                        22:14:53   Log-Likelihood:                 2657.2
No. Observations:               16236   AIC:                            -5292.
Df Residuals:                   16225   BIC:                            -5208.
Df Model:                          10
Covariance Type:            nonrobust
================================================================================
=
                    coef    std err          t      P>|t|      [0.025
0.975]
--------------------------------------------------------------------------------
-
const             3.0831      0.115     26.747      0.000       2.857
3.309
had_upgrade       0.1708      0.009     18.793      0.000       0.153
0.189
has_basement     -0.0014      0.011     -0.123      0.902      -0.024
0.021
bedrooms          0.0086      0.002      3.622      0.000       0.004
0.013
bathrooms         0.0183      0.004      5.131      0.000       0.011
0.025
floors           -0.0396      0.007     -5.828      0.000      -0.053
-0.026
condition         0.9698      0.034     28.782      0.000       0.904
1.036
grade             1.6700      0.033     50.745      0.000       1.606
1.735
sqft_above        0.2195      0.037      5.888      0.000       0.146
0.293
```

```
sqft_living15     0.6820       0.022      31.284       0.000        0.639
0.725
sqft_lot15       -0.1066       0.005     -21.671       0.000       -0.116
-0.097
==============================================================================
Omnibus:                       24.212   Durbin-Watson:                   1.973
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               22.730
Skew:                          -0.064   Prob(JB):                     1.16e-05
Kurtosis:                       2.869   Cond. No.                         766.
==============================================================================

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

[50]:
```python
log_transformed_coefs = model2.params
np.log(inv_boxcox1p(log_transformed_coefs[8], 0.7422674506937296))
```

[50]:
```
-1.4891978809749509
```

0.6.3 Iteration 2 - Top 4 correlated features only

[36]:
```python
X_train2, X_test2, y_train2, y_test2 = train_test_split(X_prep[['grade',␣
↪'sqft_above', 'bathrooms', 'bedrooms']], y, test_size=0.2, random_state=51)
```

[37]:
```python
regr = LinearRegression()
# Train the model using the training sets
regr.fit(X_train2, y_train2)

# Make predictions using the testing set
y_test_pred2 = regr.predict(X_test2)
y_train_pred2 = regr.predict(X_train2)
```

[38]:
```python
print('MSE train: %.3f, test: %.3f' % (
        mean_squared_error(y_train2, y_train_pred2),
        mean_squared_error(y_test2, y_test_pred2)))
print('R^2 train: %.3f, test: %.3f' % (
        r2_score(y_train2, y_train_pred2),
        r2_score(y_test2, y_test_pred2)))
```

```
MSE train: 0.048, test: 0.046
R^2 train: 0.423, test: 0.412
```

[39]:
```python
# Create model intercept
feats_with_int2 = sm.add_constant(X_train2)
```

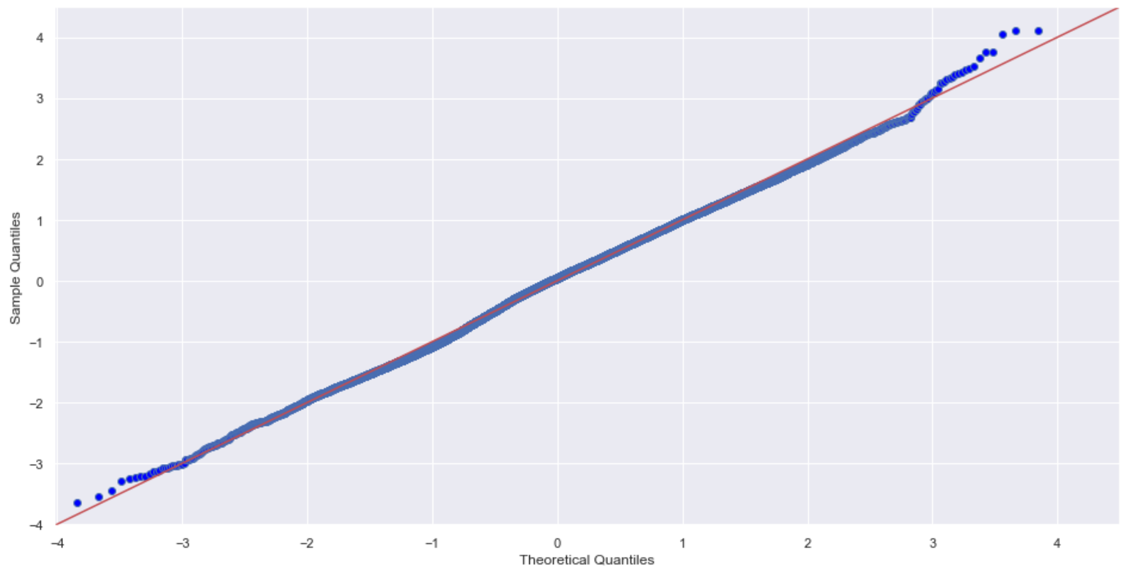
```python
# Fit model to data
model3 = sm.OLS(y_train2,feats_with_int2).fit()
model3.summary()
```

[39]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                  price   R-squared:                       0.423
Model:                            OLS   Adj. R-squared:                  0.423
Method:                 Least Squares   F-statistic:                     2970.
Date:                Wed, 09 Nov 2022   Prob (F-statistic):               0.00
Time:                        22:14:54   Log-Likelihood:                 1609.1
No. Observations:               16236   AIC:                            -3208.
Df Residuals:                   16231   BIC:                            -3170.
Df Model:                           4
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          5.8415      0.100     58.399      0.000       5.645       6.038
grade          1.9519      0.033     58.944      0.000       1.887       2.017
sqft_above     0.2405      0.033      7.396      0.000       0.177       0.304
bathrooms      0.0328      0.004      9.113      0.000       0.026       0.040
bedrooms       0.0189      0.002      7.670      0.000       0.014       0.024
==============================================================================
Omnibus:                       42.113   Durbin-Watson:                   1.971
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               34.505
Skew:                           0.038   Prob(JB):                     3.22e-08
Kurtosis:                       2.787   Cond. No.                         352.
==============================================================================

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

0.7 Testing Linear Assumptions

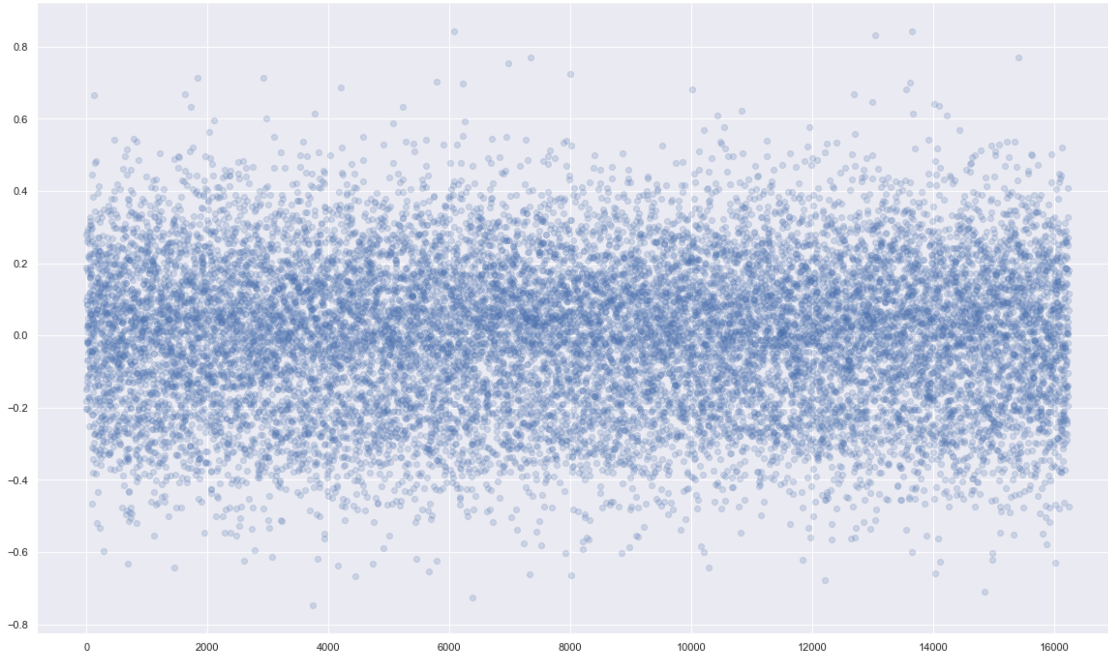
After building our multiple linear regression model, we need to check that the model meets the assumptions for linearity, the main ones are: - Checking the residuals of the model are normally distributed - Checking for homoskedascity of the residuals - There exists a linear relationship between the independent variable, x, and the dependent variable, y

[40]:
```
fig, ax = plt.subplots(figsize = (16,8))

res = model2.resid # residuals

sm.qqplot(res, fit=True, line='45', ax=ax);
```



[41]:
```
fig, ax = plt.subplots(figsize = (20,12))

resid = (y_train1 - y_train_pred1)
plt.scatter(x=range(y_train_pred1.shape[0]), y=resid, alpha=0.2)
```

[41]: <matplotlib.collections.PathCollection at 0x7f83701dbb50>



0.8 Conclusions

Looking at the differents models tried above, we can see when we only consider a simple linear regression model we account for 26.5% of the variance. We want to improve this, so with all the features included (excluding those with p-values greater than 0.05) we can see that we account for 49% of the variance. This means, when you are looking to improve the value of your home, if it's in the king's county, this is the impact each feature has:

- The value of your property increase by X for each square foot of above space added (doesn't include the basement)
- By adding another bathroom you increase the value by X
- By improving the grade of your house by 1 on the scale, your property value increases by Z

0.9 Next Steps

In order to better improve the accuracy of the model and better understand other impacting features, I suggest looking into the following data: - Stats around the local neighborhood (school quality, crime rate, etc) - Does the property have a garage and if so, how many cars can it fit inside - Proximity to local amenities