CIS_4321-01 Group 11 Project - Patrick Garrido and Brandon Kang

April 27, 2022

[1]:
```
#Brandon Kang and Patrick Garrido
#CIS 4321
#Group 11 Final Project


#Import package
import pandas as pd
from pandas.plotting import parallel_coordinates

import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

import statsmodels.api as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor

from dmba import regressionSummary, exhaustive_search, plotDecisionTree
from dmba import adjusted_r2_score, AIC_score, BIC_score
from dmba import backward_elimination, forward_selection, stepwise_selection
from dmba import classificationSummary, gainsChart

from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn import preprocessing
from sklearn.neighbors import NearestNeighbors, KNeighborsRegressor
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.metrics import pairwise
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.cluster import KMeans

from scipy.cluster.hierarchy import dendrogram, linkage, fcluster
```

[2]:
```
#LOADING DATA
df = pd.read_csv('Updated_df.csv')
```

[86]:
```
# For cleaning, we cleaned our data on another notebook. The cleaning that we␣
↪conducted consisted of removing one outlier
# that had 33 bedrooms, to fix this mistake we changed the value to 3␣
↪considering the price of the property and
# size of the property. We also got rid of rows that had 0 bedrooms and 0␣
↪bathrooms due to the main focus of this project
# was to create a model that can accurately predict house prices, not land. The␣
↪data above is the final product of our cleaning.
# We also binned certain features, but we will provide the code in the models␣
↪that
# required binning.
```

[87]:
```
#DESCRITPTIVE ANALYSIS
pd.set_option('display.float_format', lambda x: '%.3f' % x)
desc = df.describe()

desc.loc['+3_std'] = desc.loc['mean'] + (desc.loc['std'] * 3)
desc.loc['-3_std'] = desc.loc['mean'] - (desc.loc['std'] * 3)

desc
```

[87]:
```
                  id         date       price  bedrooms  bathrooms  \
count      21580.000    21580.000   21580.000 21580.000  21580.000
mean  4580157459.841 20143903.326  540381.237     3.368      2.115
std   2876924031.984     4436.932  367488.431     0.895      0.768
min      1000102.000 20140502.000   78000.000     1.000      0.500
25%   2123049166.750 20140722.000  322000.000     3.000      1.750
50%   3904921185.000 20141016.000  450000.000     3.000      2.250
75%   7309100120.000 20150217.000  645000.000     4.000      2.500
max   9900000190.000 20150527.000 7700000.000    10.000      8.000
+3_std 13210929555.794 20157214.120 1642846.531     6.052      4.418
-3_std -4050614636.111 20130592.531 -562084.057     0.684     -0.188

        sqft_living    sqft_lot    floors  waterfront      view  …  \
count     21580.000   21580.000 21580.000   21580.000 21580.000  …
mean       2079.542   15103.815     1.494       0.008     0.234  …
std         917.876   41428.314     0.540       0.087     0.767  …
min         370.000     520.000     1.000       0.000     0.000  …
25%        1424.250    5040.000     1.000       0.000     0.000  …
50%        1910.000    7617.000     1.500       0.000     0.000  …
75%        2550.000   10685.500     2.000       0.000     0.000  …
max       13540.000 1651359.000     3.500       1.000     4.000  …
+3_std     4833.170  139388.758     3.113       0.267     2.534  …
-3_std     -674.086 -109181.127    -0.125      -0.252    -2.065  …

           grade  sqft_above  sqft_basement  yr_built  yr_renovated   zipcode  \
count  21580.000   21580.000      21580.000 21580.000     21580.000 21580.000
```

```
mean       7.658    1788.217        291.325  1971.001        84.439 98077.935
std        1.174     827.729        442.298    29.378       401.762    53.520
min        3.000     370.000          0.000  1900.000         0.000 98001.000
25%        7.000    1190.000          0.000  1951.000         0.000 98033.000
50%        7.000    1560.000          0.000  1975.000         0.000 98065.000
75%        8.000    2210.000        560.000  1997.000         0.000 98118.000
max       13.000    9410.000       4820.000  2015.000      2015.000 98199.000
+3_std    11.179    4271.403       1618.219  2059.136      1289.724 98238.494
-3_std     4.138    -694.970      -1035.569  1882.867     -1120.847 97917.375

             lat      long  sqft_living15  sqft_lot15
count  21580.000 21580.000      21580.000   21580.000
mean      47.560  -122.214       1986.833   12762.549
std        0.139     0.141        685.432   27284.620
min       47.156  -122.519        399.000     651.000
25%       47.471  -122.328       1490.000    5100.000
50%       47.572  -122.230       1840.000    7620.000
75%       47.678  -122.125       2360.000   10083.000
max       47.778  -121.315       6210.000  871200.000
+3_std    47.976  -121.792       4043.128   94616.410
-3_std    47.144  -122.636        -69.462  -69091.312

[10 rows x 21 columns]
```

[88]:
```
# From price, the most noticeable feature is the max value being 7 million and
↪the third standard deviation from the
# mean is only one million this indicates that 7 million is an outlier and that
↪there will probably be more outliers since
# the difference between the max value and 3rd standard deviation from the mean
↪is quite large. The standard deviation is
# also quite small compared to the max value, telling us that the majority of
↪the values are closely related to eachother.

# Since this dataset contains values about property size, we can compare the
↪max value to the third standard deviation
# from the mean, and see that there are outliers, and these outliers make sense
↪because these outliers are most likely
# mansions that few people live in.
```

[89]:
```
corr = df.corr()
corr
```

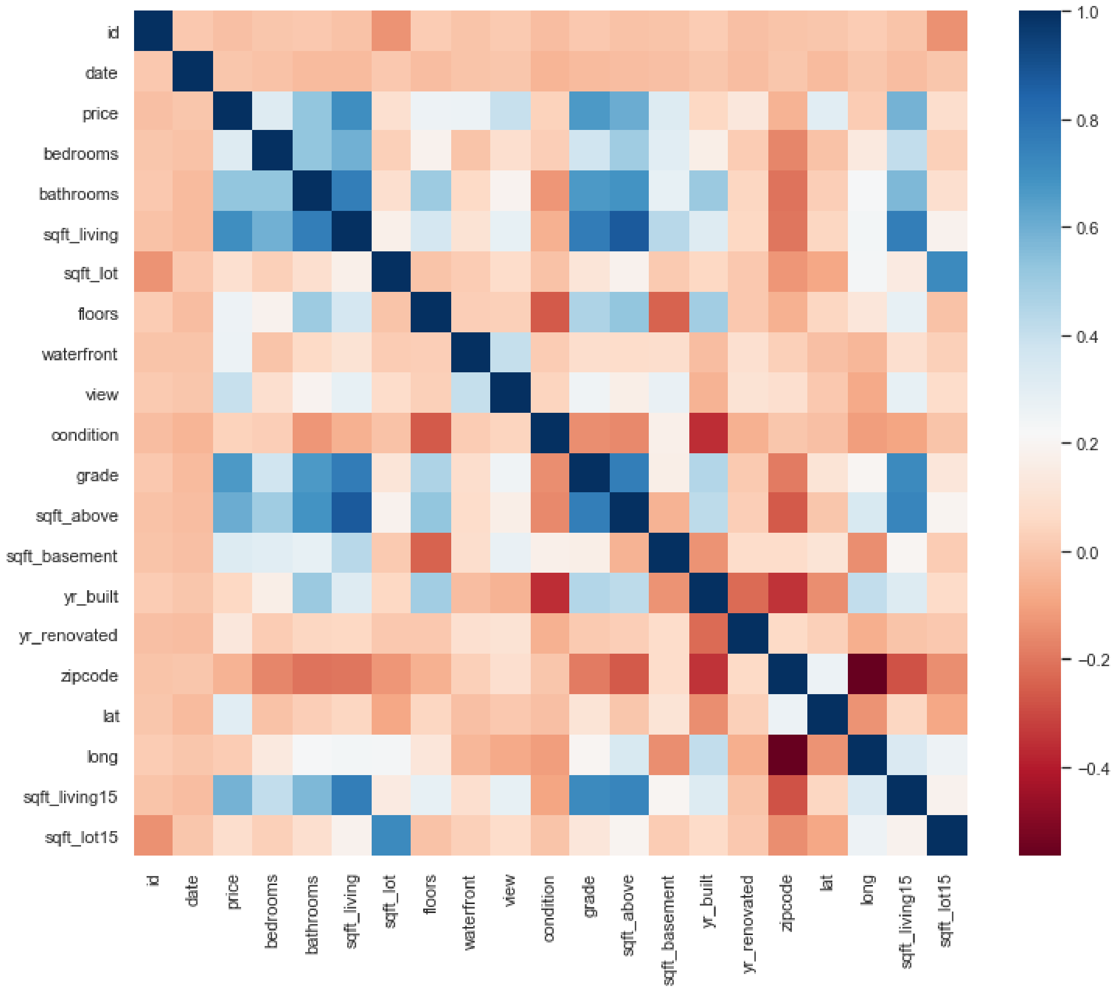
[89]:
```
                 id   date  price  bedrooms  bathrooms  sqft_living  \
id            1.000  0.010 -0.017     0.002      0.005       -0.012
date          0.010  1.000  0.003    -0.009     -0.027       -0.030
price        -0.017  0.003  1.000     0.320      0.527        0.703
bedrooms      0.002 -0.009  0.320     1.000      0.529        0.596
```

```
bathrooms      0.005 -0.027  0.527     0.529      1.000         0.756
sqft_living   -0.012 -0.030  0.703     0.596      0.756         1.000
sqft_lot      -0.132  0.006  0.090     0.035      0.089         0.174
floors         0.019 -0.022  0.257     0.185      0.503         0.354
waterfront    -0.003 -0.004  0.266    -0.007      0.064         0.104
view           0.011  0.001  0.397     0.084      0.189         0.285
condition     -0.024 -0.047  0.036     0.024     -0.126        -0.059
grade          0.008 -0.032  0.668     0.372      0.668         0.764
sqft_above    -0.011 -0.024  0.606     0.496      0.687         0.877
sqft_basement -0.006 -0.016  0.324     0.309      0.283         0.435
yr_built       0.022  0.003  0.054     0.163      0.508         0.318
yr_renovated  -0.017 -0.024  0.126     0.018      0.051         0.055
zipcode       -0.008  0.001 -0.053    -0.162     -0.206        -0.200
lat           -0.002 -0.030  0.307    -0.012      0.024         0.052
long           0.021 -0.000  0.022     0.140      0.226         0.242
sqft_living15 -0.003 -0.023  0.585     0.411      0.572         0.757
sqft_lot15    -0.139  0.000  0.083     0.033      0.089         0.185

               sqft_lot  floors  waterfront   view  …  grade  sqft_above  \
id               -0.132   0.019      -0.003  0.011  …  0.008      -0.011
date              0.006  -0.022      -0.004  0.001  … -0.032      -0.024
price             0.090   0.257       0.266  0.397  …  0.668       0.606
bedrooms          0.035   0.185      -0.007  0.084  …  0.372       0.496
bathrooms         0.089   0.503       0.064  0.189  …  0.668       0.687
sqft_living       0.174   0.354       0.104  0.285  …  0.764       0.877
sqft_lot          1.000  -0.005       0.022  0.075  …  0.115       0.184
floors           -0.005   1.000       0.024  0.029  …  0.459       0.524
waterfront        0.022   0.024       1.000  0.402  …  0.083       0.072
view              0.075   0.029       0.402  1.000  …  0.252       0.168
condition        -0.009  -0.264       0.017  0.046  … -0.147      -0.159
grade             0.115   0.459       0.083  0.252  …  1.000       0.757
sqft_above        0.184   0.524       0.072  0.168  …  0.757       1.000
sqft_basement     0.016  -0.246       0.081  0.277  …  0.169      -0.052
yr_built          0.053   0.489      -0.026 -0.054  …  0.448       0.424
yr_renovated      0.008   0.006       0.093  0.104  …  0.014       0.023
zipcode          -0.130  -0.060       0.030  0.085  … -0.186      -0.262
lat              -0.086   0.050      -0.014  0.006  …  0.114      -0.001
long              0.230   0.126      -0.042 -0.078  …  0.200       0.345
sqft_living15     0.145   0.280       0.086  0.281  …  0.714       0.732
sqft_lot15        0.718  -0.011       0.031  0.073  …  0.121       0.195

               sqft_basement  yr_built  yr_renovated  zipcode    lat   long  \
id                    -0.006     0.022        -0.017   -0.008 -0.002  0.021
date                  -0.016     0.003        -0.024    0.001 -0.030 -0.000
price                  0.324     0.054         0.126   -0.053  0.307  0.022
bedrooms               0.309     0.163         0.018   -0.162 -0.012  0.140
bathrooms              0.283     0.508         0.051   -0.206  0.024  0.226
```

```
sqft_living          0.435     0.318         0.055   -0.200  0.052  0.242
sqft_lot             0.016     0.053         0.008   -0.130 -0.086  0.230
floors              -0.246     0.489         0.006   -0.060  0.050  0.126
waterfront           0.081    -0.026         0.093    0.030 -0.014 -0.042
view                 0.277    -0.054         0.104    0.085  0.006 -0.078
condition            0.174    -0.362        -0.061    0.003 -0.015 -0.106
grade                0.169     0.448         0.014   -0.186  0.114  0.200
sqft_above          -0.052     0.424         0.023   -0.262 -0.001  0.345
sqft_basement        1.000    -0.133         0.071    0.074  0.110 -0.144
yr_built            -0.133     1.000        -0.225   -0.347 -0.148  0.410
yr_renovated         0.071    -0.225         1.000    0.064  0.029 -0.068
zipcode              0.074    -0.347         0.064    1.000  0.267 -0.564
lat                  0.110    -0.148         0.029    0.267  1.000 -0.135
long                -0.144     0.410        -0.068   -0.564 -0.135  1.000
sqft_living15        0.201     0.326        -0.003   -0.279  0.049  0.336
sqft_lot15           0.018     0.071         0.008   -0.147 -0.086  0.256

               sqft_living15  sqft_lot15
id                    -0.003      -0.139
date                  -0.023       0.000
price                  0.585       0.083
bedrooms               0.411       0.033
bathrooms              0.572       0.089
sqft_living            0.757       0.185
sqft_lot               0.145       0.718
floors                 0.280      -0.011
waterfront             0.086       0.031
view                   0.281       0.073
condition             -0.093      -0.003
grade                  0.714       0.121
sqft_above             0.732       0.195
sqft_basement          0.201       0.018
yr_built               0.326       0.071
yr_renovated          -0.003       0.008
zipcode               -0.279      -0.147
lat                    0.049      -0.086
long                   0.336       0.256
sqft_living15          1.000       0.183
sqft_lot15             0.183       1.000

[21 rows x 21 columns]
```

[90]:
```python
sns.set(rc = {'figure.figsize': (12,10)})
sns.heatmap(corr, xticklabels = corr.columns, yticklabels = corr.columns, cmap= 'RdBu')
```

[90]:
```
<AxesSubplot:>
```



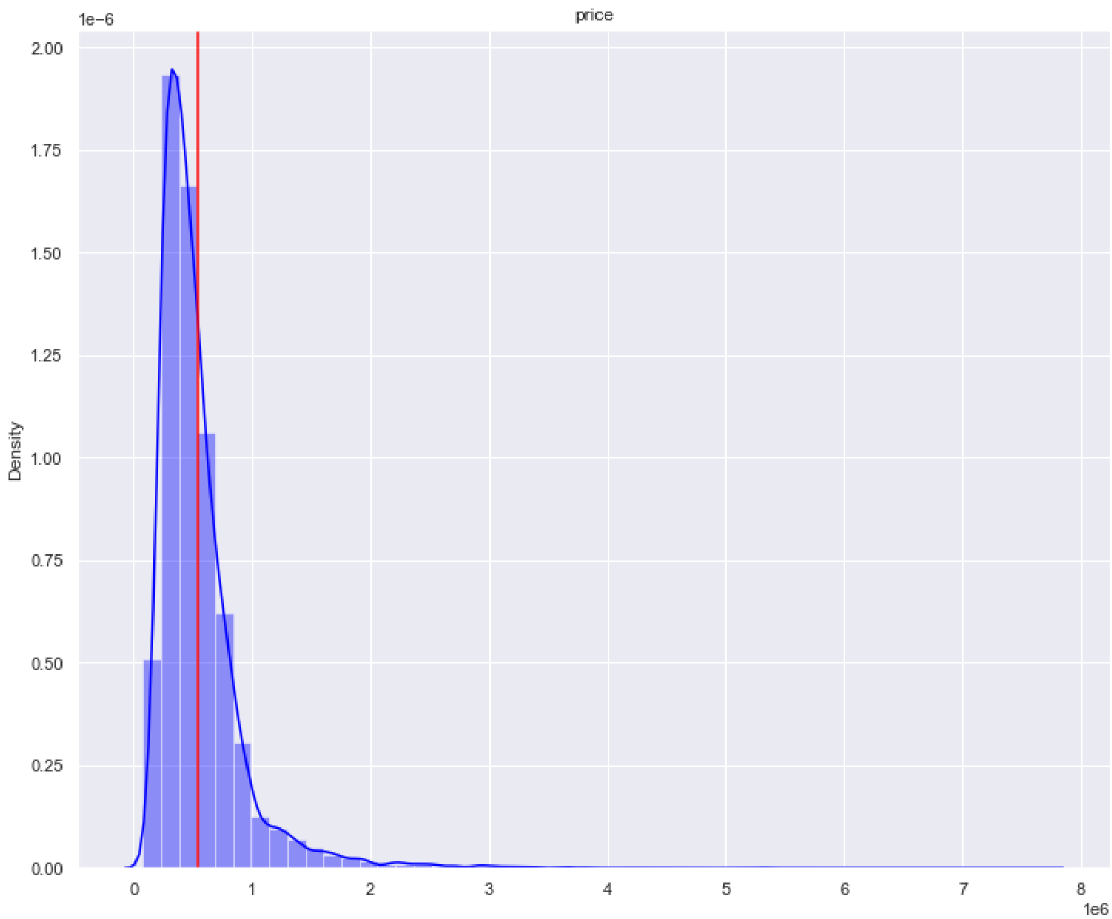
[91]:
```
# From this output we can see that this dataset does not have any negative␣
↪values, so there
# will be no variable where one will affect the other in a negative way.␣
↪However, we do see
# positive values and multicollinearity between those variables. Ex. The top␣
↪left of the map
# we see sqft_living showing multicollinearity between the variables: price,␣
↪bedrooms,and bathrooms
# with a correlation somewhere between .6 - .8. We also see other areas in the␣
↪map where variables
# show sign of collinearity, this is indicated by the regions in dark blue.
```

[92]:
```
# DATA VISUALIZATION
o = df['price'].values
mean = df['price'].mean()
```

```
sns.distplot(o, color = 'blue').set_title('price')
plt.axvline(mean,0,1, color = 'red')
mean
```

```
C:\Users\kangb\anaconda3\lib\site-packages\seaborn\distributions.py:2551:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
```

[92]: 540381.2371640408

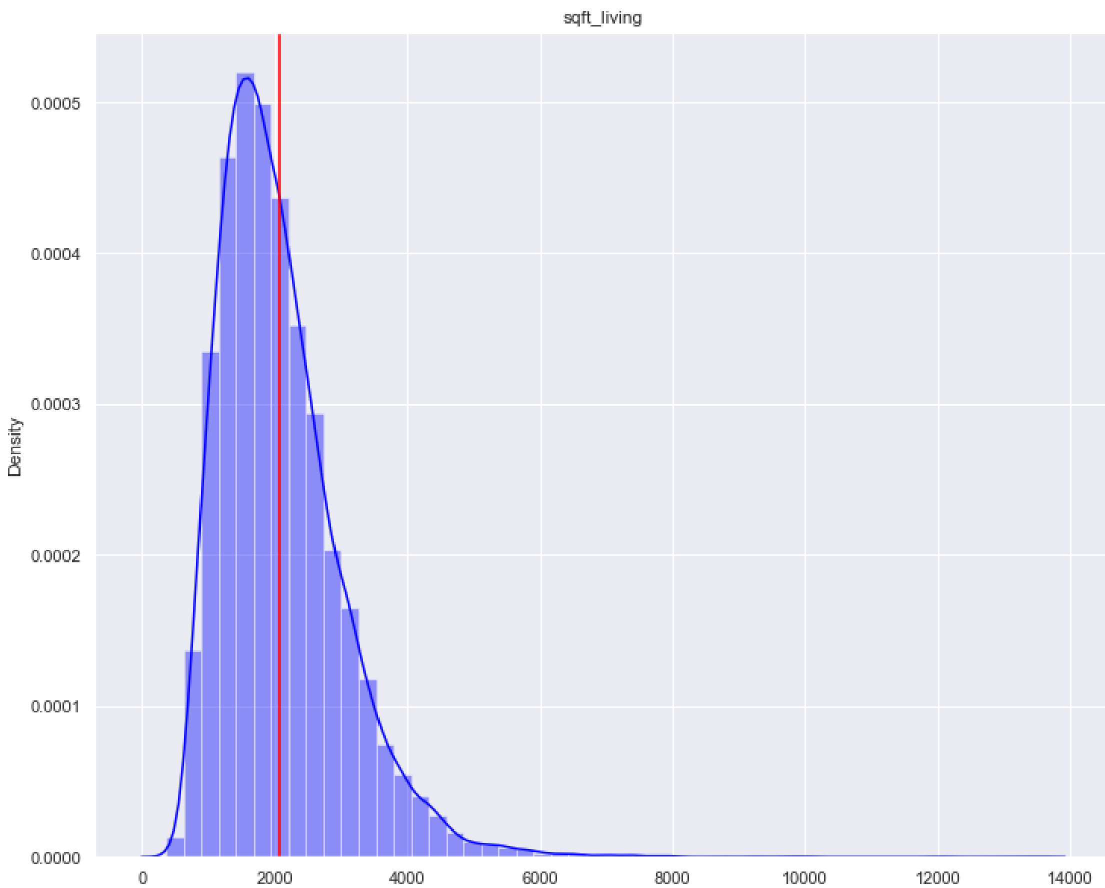


[93]:
```
# Price will be our dependent variable, and from this histogram we see
# it is skewed to the right indicating that there are outliers.
# The red line is the mean for price, and we can see that the mean is
# $540,381 based off of the output above the graph. We also see that most
# house prices cost less than the mean.
```

```
[94]: o = df['sqft_living'].values
mean = df['sqft_living'].mean()

sns.distplot(o, color = 'blue').set_title('sqft_living')
plt.axvline(mean,0,1, color = 'red')
mean
```

```
C:\Users\kangb\anaconda3\lib\site-packages\seaborn\distributions.py:2551:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
```

```
[94]: 2079.541751621872
```



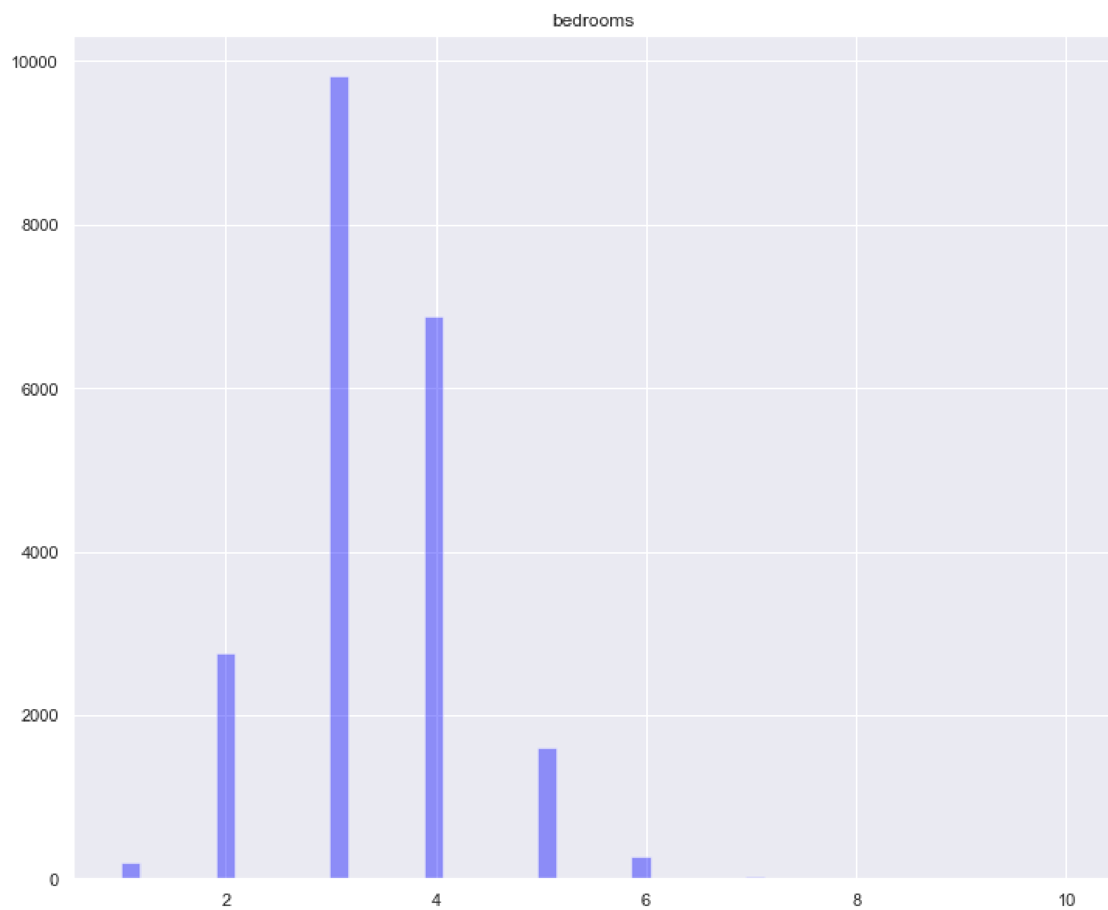

```
[95]: # From sqft_living we see that the distribution is skewed to the right showing␣
↪that there outliers, just like price.
```

```
# We see that the mean is 2079, and that the majority of the sqft of living in
↪this dataset are less than 2000.
```

[96]:
```
o = df['bedrooms'].values
mean = df['bedrooms'].mean()

sns.distplot(o, color = 'blue', kde = False).set_title('bedrooms')
```

```
C:\Users\kangb\anaconda3\lib\site-packages\seaborn\distributions.py:2551:
FutureWarning: `distplot` is a deprecated function and will be removed in a
future version. Please adapt your code to use either `displot` (a figure-level
function with similar flexibility) or `histplot` (an axes-level function for
histograms).
  warnings.warn(msg, FutureWarning)
```

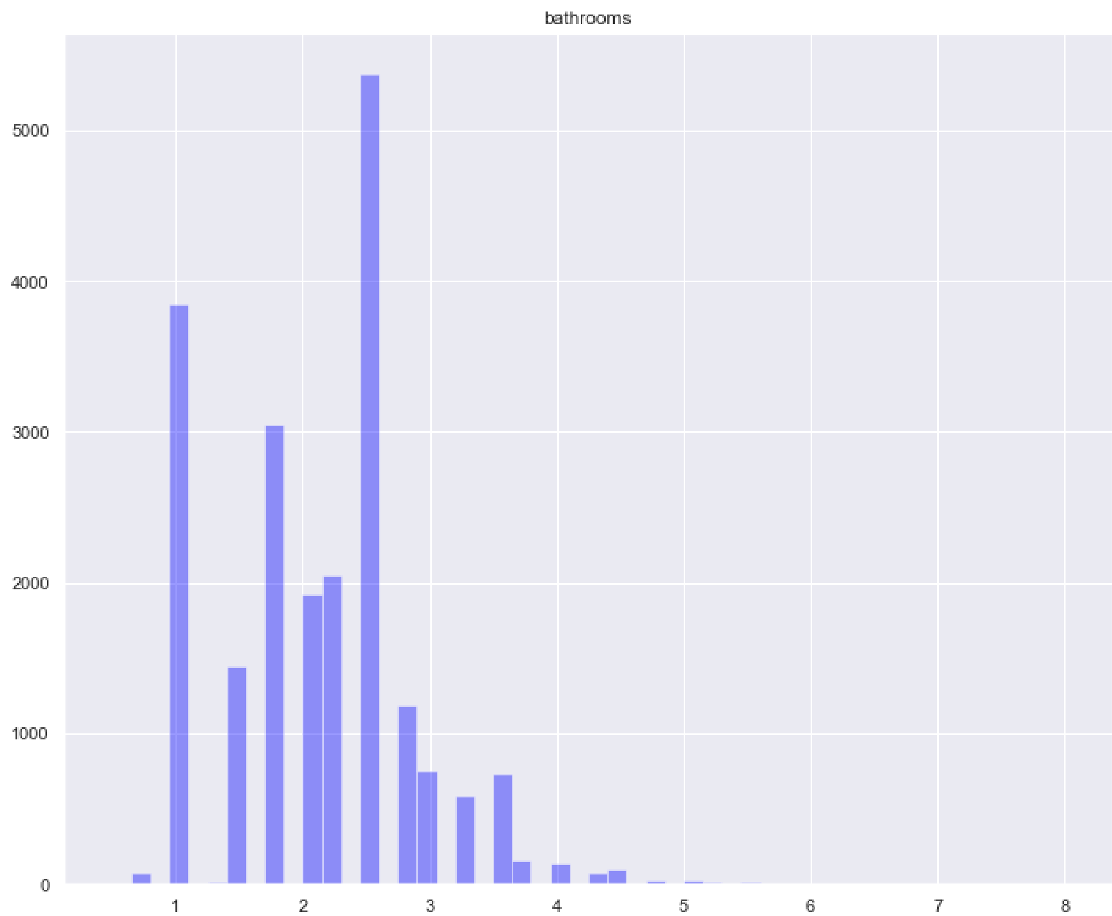
[96]: Text(0.5, 1.0, 'bedrooms')



[97]:
```
# From this graph we can see that majority of the houses are three bedroom
```

[98]:
```python
o = df['bathrooms'].values
mean = df['bathrooms'].mean()

sns.distplot(o, color = 'blue', kde = False).set_title('bathrooms')
```

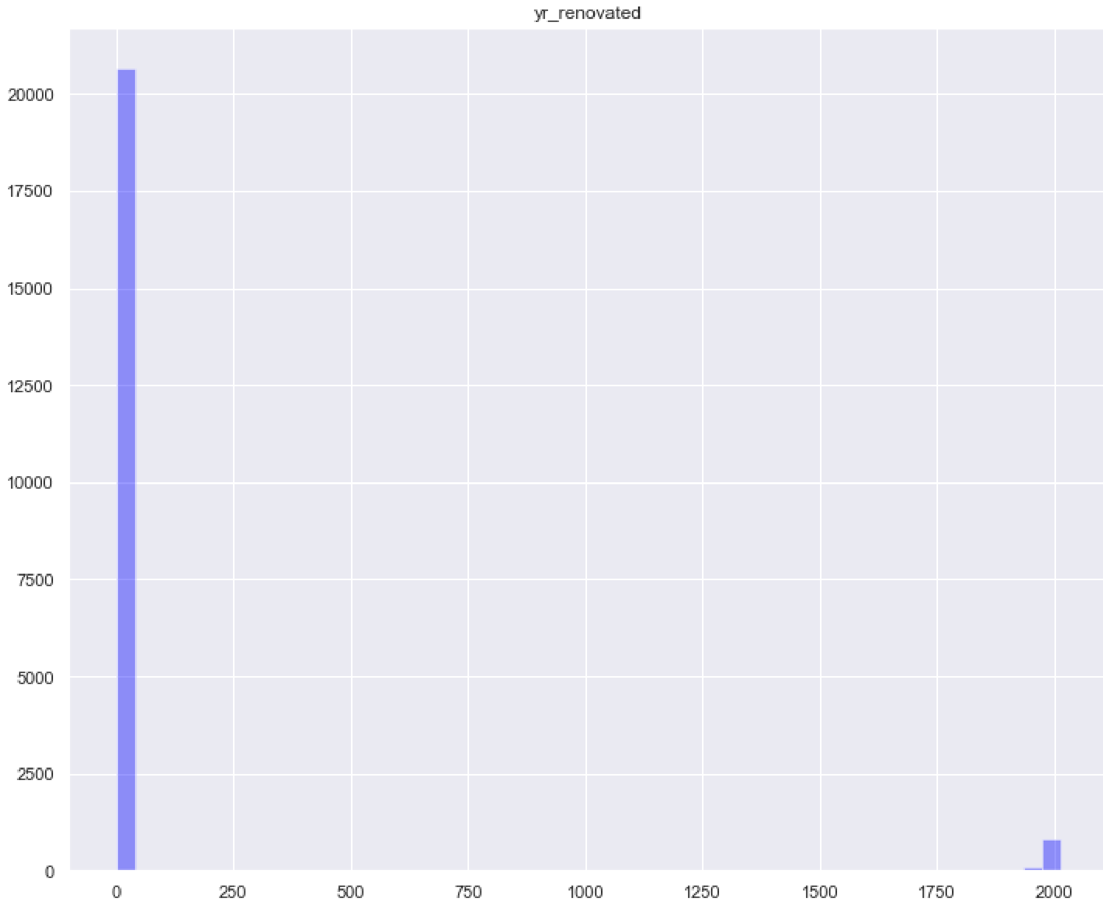
[98]:
```
Text(0.5, 1.0, 'bathrooms')
```



[99]:
```python
# From this graph we see that some houses have half a bathroom, meaning
# that some bathrooms do not have a bathtub or shower, and only have a toliet.
```

[100]:
```python
o = df['yr_renovated'].values
mean = df['yr_renovated'].mean()

sns.distplot(o, color = 'blue', kde = False).set_title('yr_renovated')
```

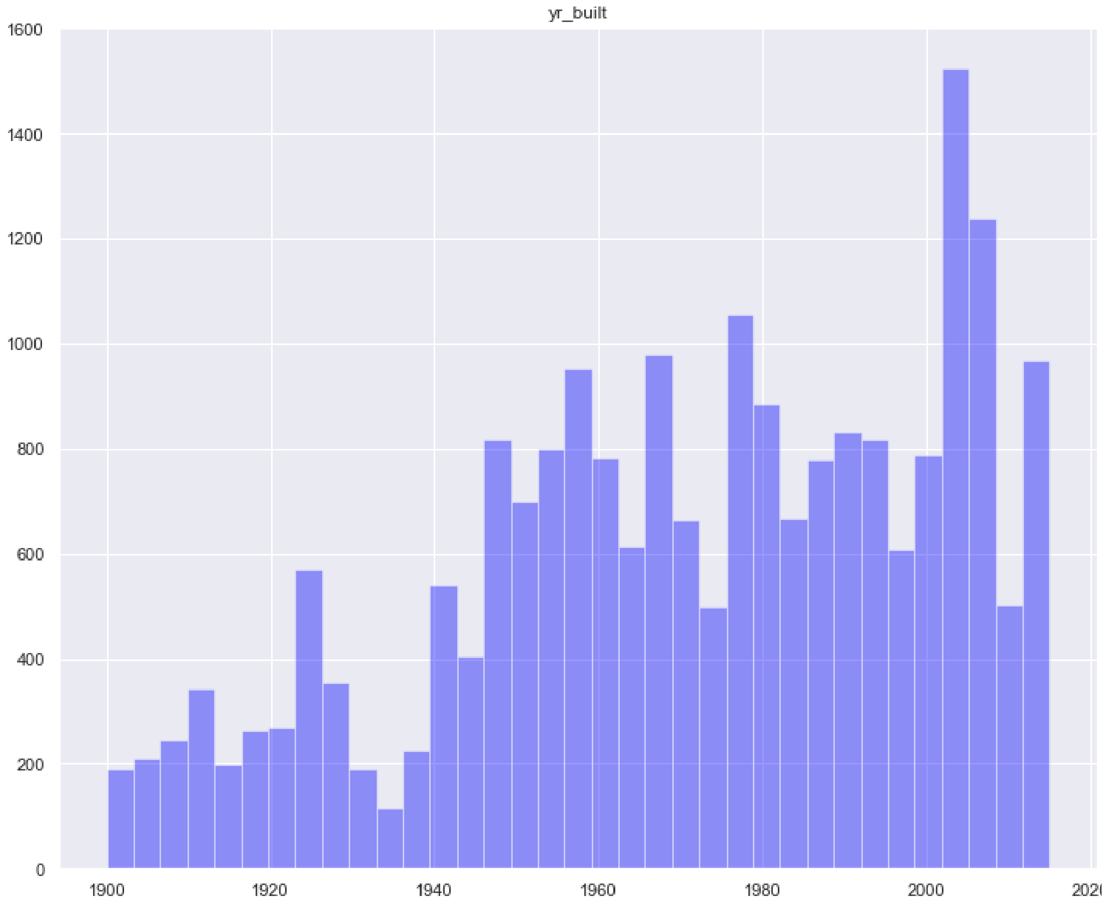
[100]:
```
Text(0.5, 1.0, 'yr_renovated')
```



[101]:
```
# From this graph we see that many of the houses have not been renovated, and␣
↪that the majority of renovated houses are from
# the 2000s
```

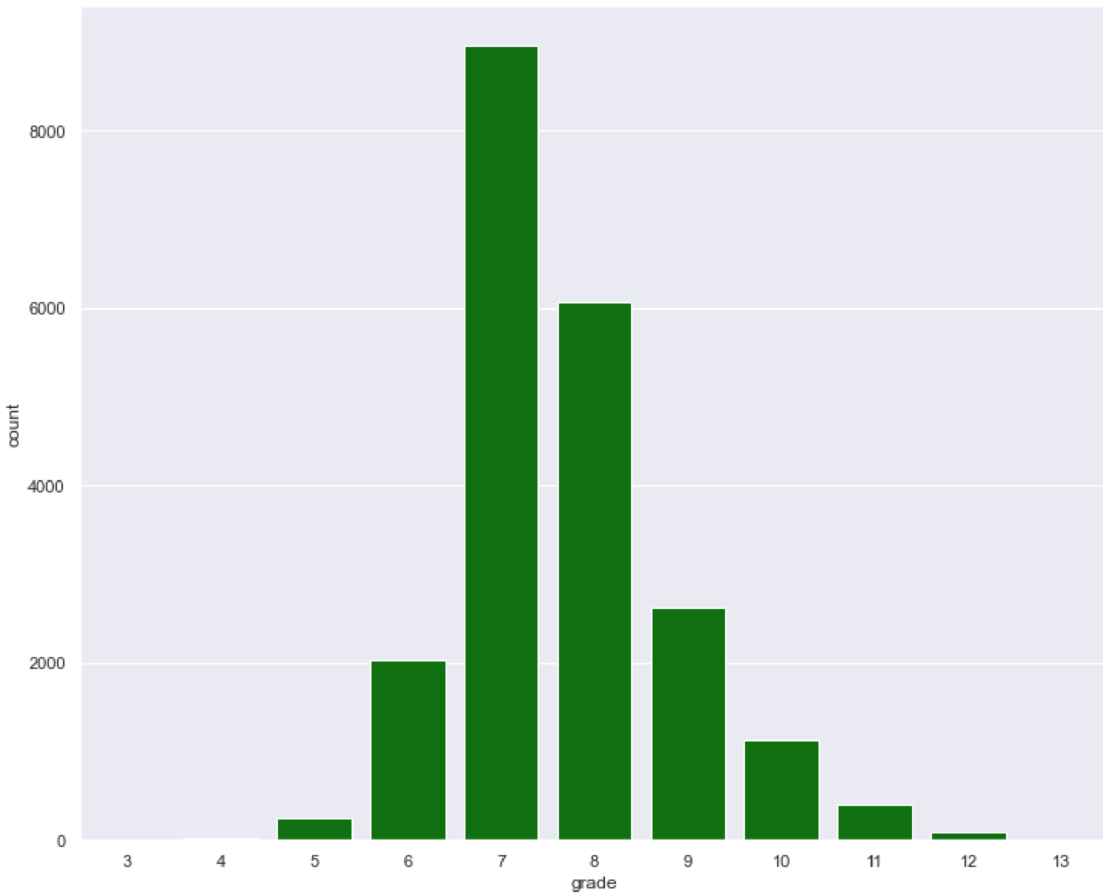
[102]:
```
o = df['yr_built'].values
mean = df['yr_built'].mean()

sns.distplot(o, color = 'blue', kde = False).set_title('yr_built')
```

[102]: Text(0.5, 1.0, 'yr_built')



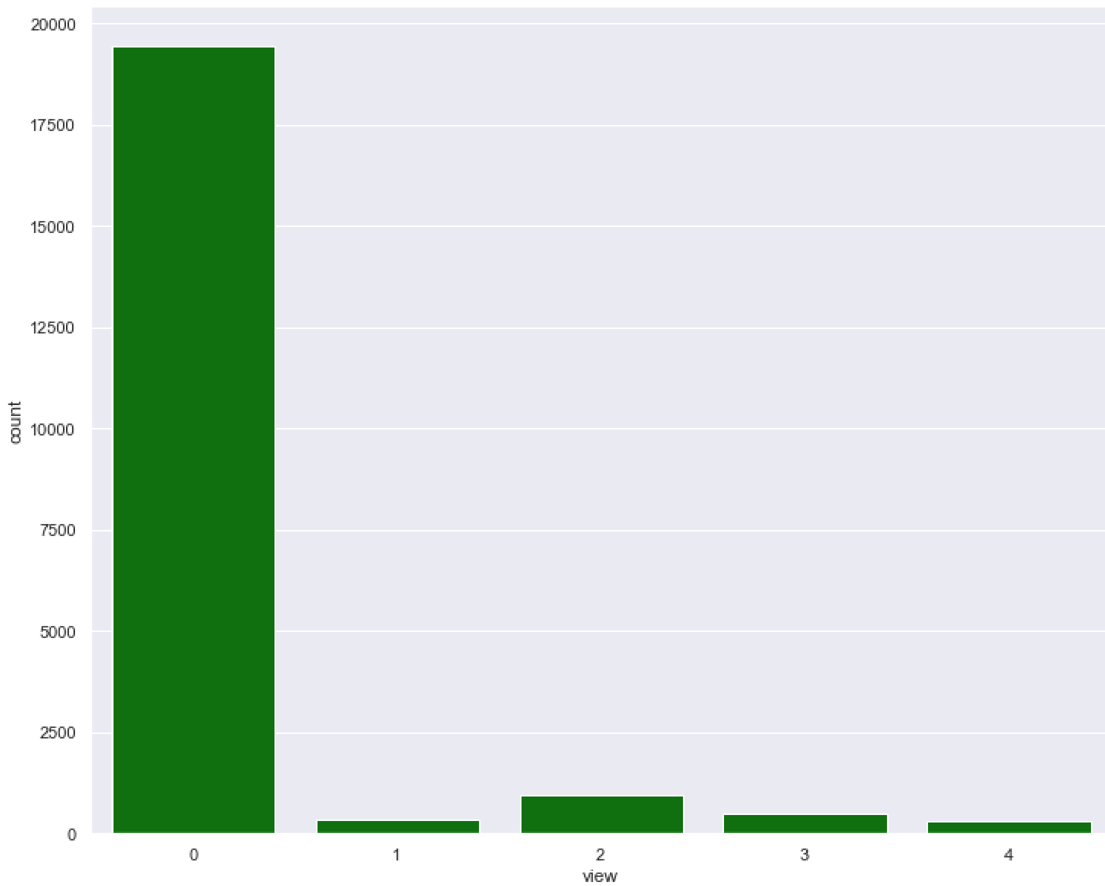
[103]:
```
# These houses are from the 1900s to present, we also see that in two years␣
↪there was a boom in building houses, never before
# seen in this county.
```

[104]:
```
ax = sns.countplot(x = 'grade', data = df, color = 'green')
```



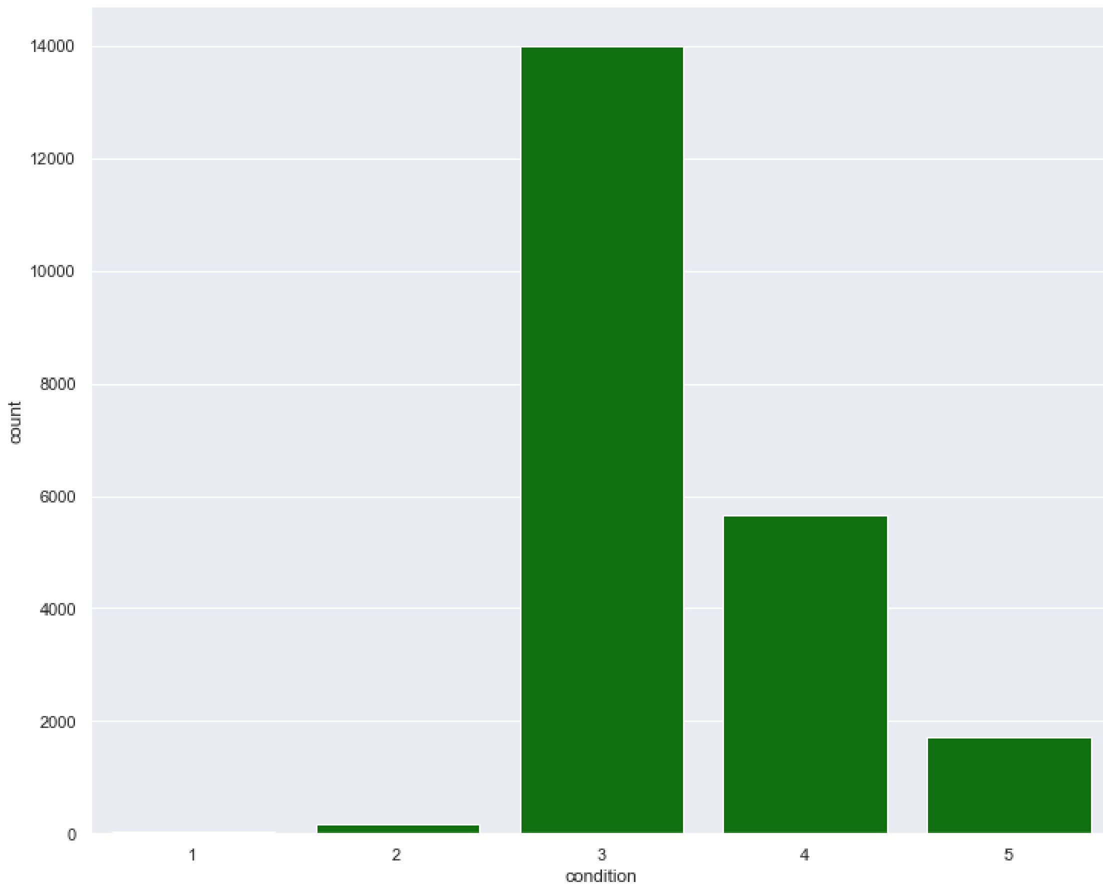
[105]:
```
# We binned this categorical data into binning. But from this graph we see that␣
↪many of these houses are graded a 7, which means
# the construction quality is average. We also see that average is the standard␣
↪when it comes to most houses, and only
# few houses are built with high quality.
```

[106]:
```
ax = sns.countplot(x = 'view', data = df, color = 'green')
```



[107]:
```
# View is whether the property has a view, and if they have a view it will be␣
↪graded by quality of that view.
# From this barchart, we see that majority have no view at all, while houses␣
↪that do have views, the majority in that category
# have a decent view.
```

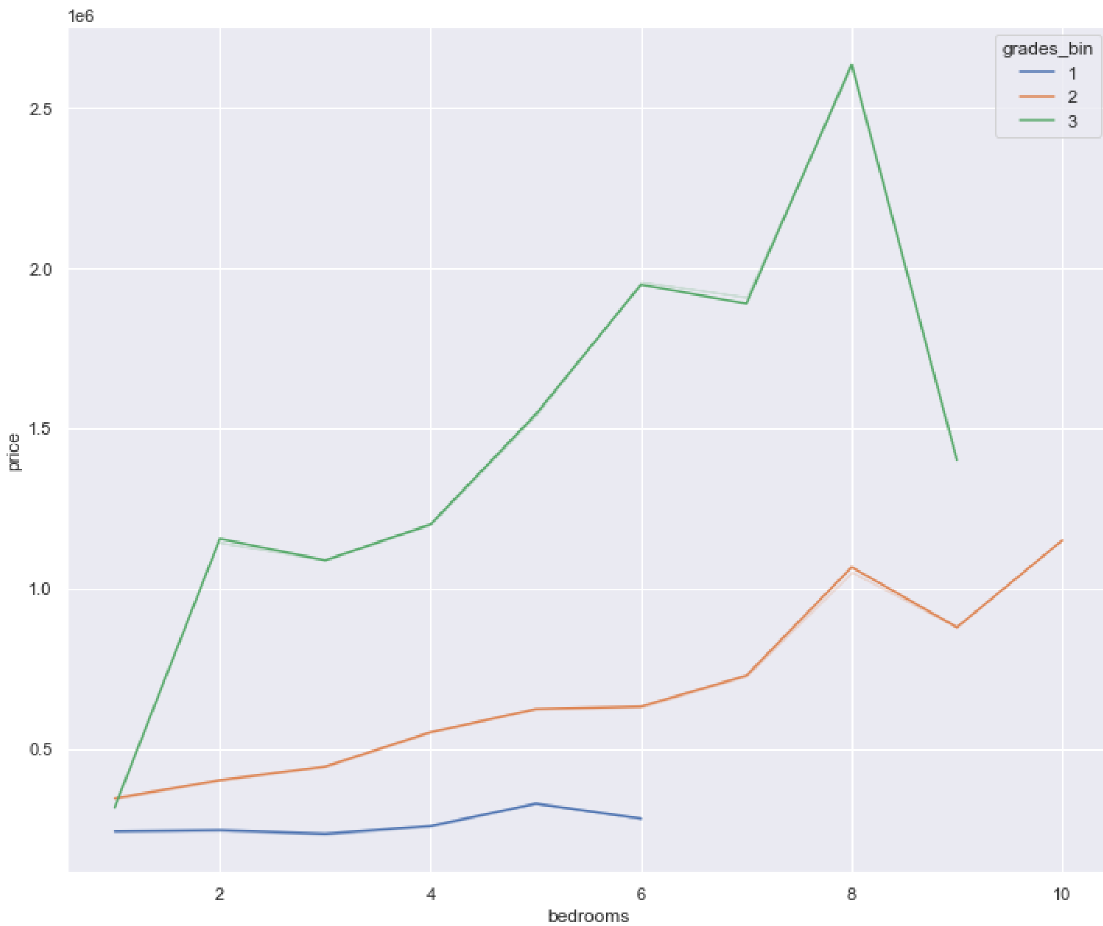
[108]:
```
ax = sns.countplot(x = 'condition', data = df, color = 'green')
```



[109]:
```
# Condition is another categorical variable that we binned, and from this most␣
↪of the houses are in decent condition.
# Few houses are in bad condition, and luckily no houses are in unacceetpable␣
↪conditions.
```

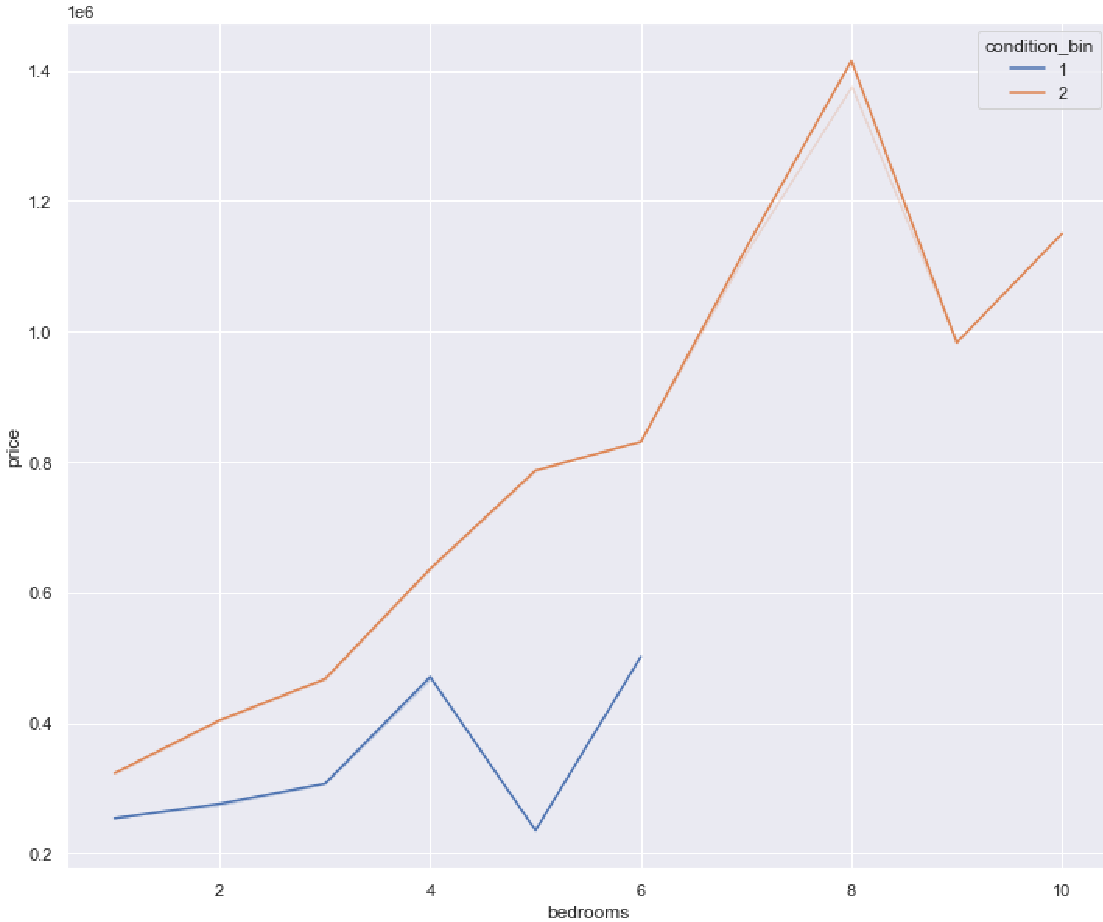
[110]:
```
#USED FOR MULTIPLE LINEAR REGRESSION
# 1-5: bad, 6 - 9: average 10 - 13 good
bins = [0, 5, 9, 13]
group_names =[1,2,3]
# 1 = bad; 2 = average; 3 = good
df['grades_bin'] = pd.cut(df['grade'], bins, labels = group_names)
subset = df[['grade', 'grades_bin']]

bins = [0, 2, 5]
group_names = [1,2]
df['condition_bin'] = pd.cut(df['condition'], bins, labels = group_names)
subset = df[['condition', 'condition_bin']]
```

```
ax = sns.lineplot(x = 'bedrooms', y = 'price', data = df, hue = 'grades_bin',␣
↪ci = False, markers = True)
```



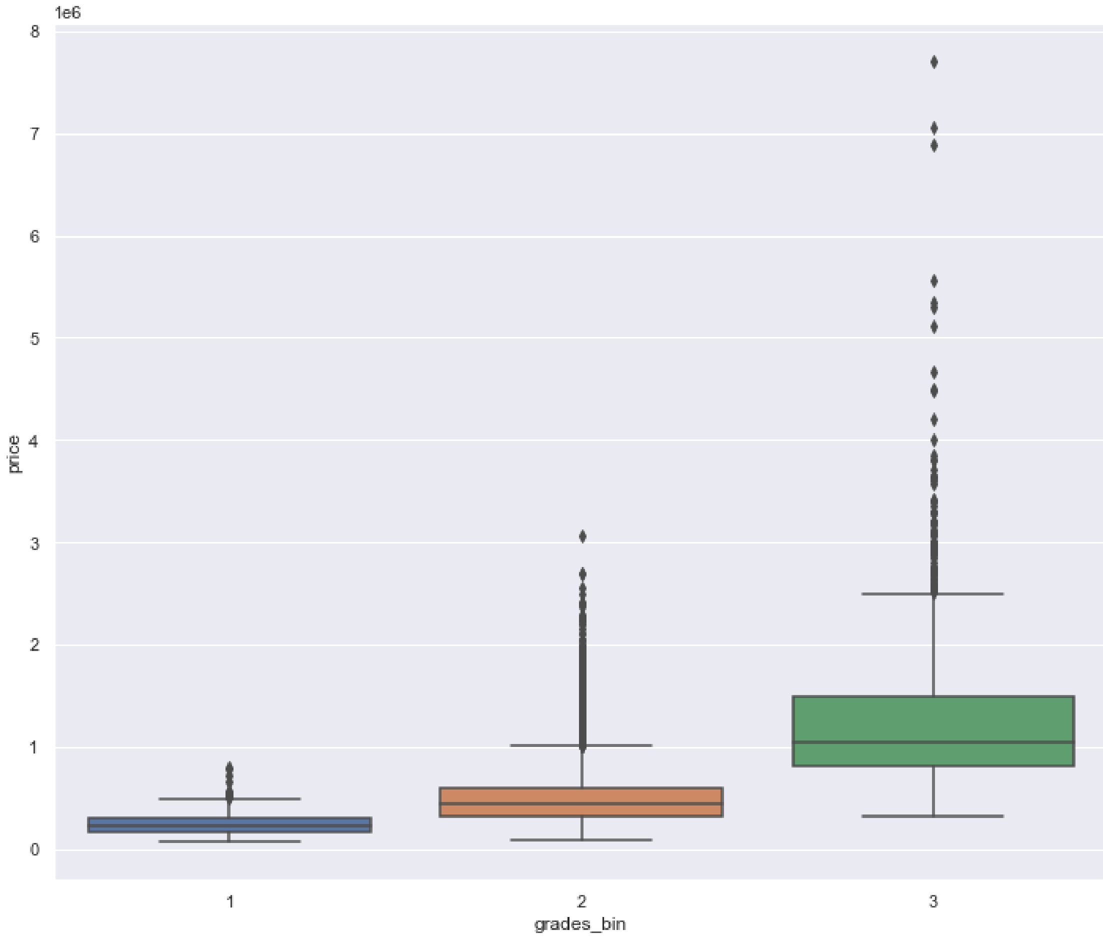
[111]:
```
# We can see that the price depends greatly on the grade of the house,␣
↪regardless of the number of bedrooms.
# The number of bedrooms increases the price, but not at a large margin as the␣
↪grade of the house shown by this graph.
```

[112]:
```
ax = sns.lineplot(x = 'bedrooms', y = 'price', data = df, hue =␣
↪'condition_bin', ci = False, markers = True)
```



[113]:
```
# We also see a similar trend when looking at the condition of the house.␣
↪Regardless of the number of bedrooms, if the condition
# is bad the price of the house will also be poor.
```

[114]:
```
ax = sns.boxplot(x = 'grades_bin', y = 'price' , data = df)
```



[115]:
```
# From this graph we can see that many of the outliers are in grade bin 3␣
↪indicating that only mansions and properties in the
# millions are built in this quality.
```

1 MULTIPLE LINEAR REGREASION

[116]:
```
#made dummies; n-1 dummies due to n dummies distorting model
MLRdf = df
```

[117]:
```
MLRdf = MLRdf[['price', 'sqft_living', 'sqft_lot', 'sqft_above',␣
↪'sqft_basement', 'bedrooms', 'bathrooms', 'grades_bin', 'view',␣
↪'condition_bin', 'yr_renovated', 'yr_built']]
MLRdf = pd.get_dummies(MLRdf, columns = ['grades_bin', 'view',␣
↪'condition_bin'], drop_first = True)
```

[118]:
```
#MLRdf = MLRdf.astype(float)
MLRdf.dtypes
```

[118]:
```
price              int64
sqft_living        int64
sqft_lot           int64
sqft_above         int64
sqft_basement      int64
bedrooms           int64
bathrooms        float64
yr_renovated       int64
yr_built           int64
grades_bin_2       uint8
grades_bin_3       uint8
view_1             uint8
view_2             uint8
view_3             uint8
view_4             uint8
condition_bin_2    uint8
dtype: object
```

[119]:
```
MLRCorr = MLRdf.corr()
MLRCorr
```

[119]:

	price	sqft_living	sqft_lot	sqft_above	sqft_basement	\
price	1.000	0.703	0.090	0.606	0.324	
sqft_living	0.703	1.000	0.174	0.877	0.435	
sqft_lot	0.090	0.174	1.000	0.184	0.016	
sqft_above	0.606	0.877	0.184	1.000	-0.052	
sqft_basement	0.324	0.435	0.016	-0.052	1.000	
bedrooms	0.320	0.596	0.035	0.496	0.309	
bathrooms	0.527	0.756	0.089	0.687	0.283	
yr_renovated	0.126	0.055	0.008	0.023	0.071	
yr_built	0.054	0.318	0.053	0.424	-0.133	
grades_bin_2	-0.487	-0.467	-0.118	-0.474	-0.084	
grades_bin_3	0.560	0.559	0.118	0.558	0.117	
view_1	0.093	0.067	-0.008	0.022	0.097	
view_2	0.149	0.135	0.038	0.078	0.135	
view_3	0.183	0.159	0.074	0.092	0.159	
view_4	0.308	0.170	0.019	0.108	0.151	
condition_bin_2	0.055	0.072	-0.037	0.064	0.030	

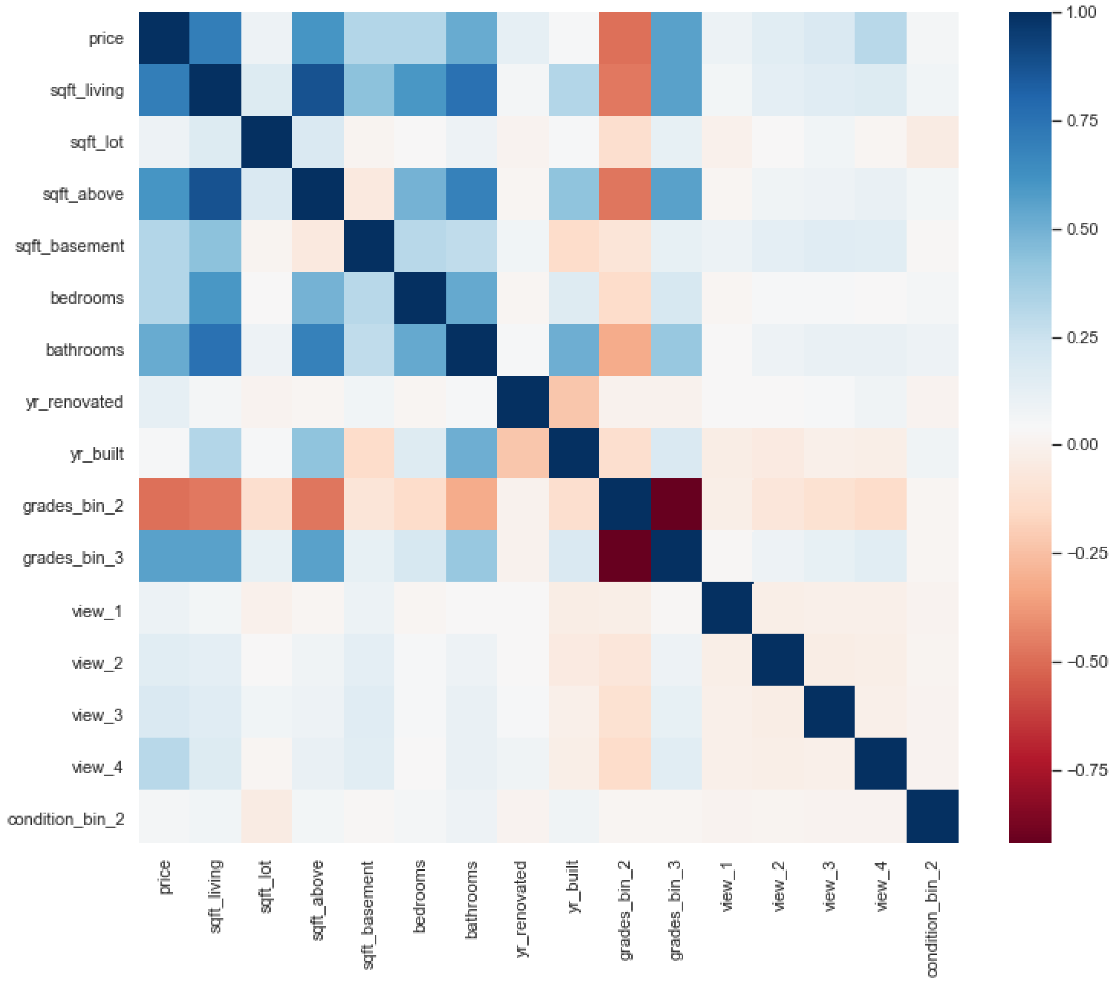
	bedrooms	bathrooms	yr_renovated	yr_built	grades_bin_2	\
price	0.320	0.527	0.126	0.054	-0.487	
sqft_living	0.596	0.756	0.055	0.318	-0.467	
sqft_lot	0.035	0.089	0.008	0.053	-0.118	
sqft_above	0.496	0.687	0.023	0.424	-0.474	
sqft_basement	0.309	0.283	0.071	-0.133	-0.084	
bedrooms	1.000	0.529	0.018	0.163	-0.135	
bathrooms	0.529	1.000	0.051	0.508	-0.318	

yr_renovated	0.018	0.051	1.000	-0.225	0.002
yr_built	0.163	0.508	-0.225	1.000	-0.122
grades_bin_2	-0.135	-0.318	0.002	-0.122	1.000
grades_bin_3	0.201	0.404	0.002	0.188	-0.920
view_1	0.023	0.038	0.034	-0.034	-0.027
view_2	0.048	0.089	0.033	-0.045	-0.079
view_3	0.053	0.113	0.051	-0.019	-0.104
view_4	0.036	0.108	0.081	-0.020	-0.136
condition_bin_2	0.060	0.087	0.008	0.081	0.023

	grades_bin_3	view_1	view_2	view_3	view_4	condition_bin_2
price	0.560	0.093	0.149	0.183	0.308	0.055
sqft_living	0.559	0.067	0.135	0.159	0.170	0.072
sqft_lot	0.118	-0.008	0.038	0.074	0.019	-0.037
sqft_above	0.558	0.022	0.078	0.092	0.108	0.064
sqft_basement	0.117	0.097	0.135	0.159	0.151	0.030
bedrooms	0.201	0.023	0.048	0.053	0.036	0.060
bathrooms	0.404	0.038	0.089	0.113	0.108	0.087
yr_renovated	0.002	0.034	0.033	0.051	0.081	0.008
yr_built	0.188	-0.034	-0.045	-0.019	-0.020	0.081
grades_bin_2	-0.920	-0.027	-0.079	-0.104	-0.136	0.023
grades_bin_3	1.000	0.030	0.089	0.118	0.146	0.024
view_1	0.030	1.000	-0.027	-0.019	-0.015	0.004
view_2	0.089	-0.027	1.000	-0.034	-0.026	0.014
view_3	0.118	-0.019	-0.034	1.000	-0.019	0.009
view_4	0.146	-0.015	-0.026	-0.019	1.000	0.004
condition_bin_2	0.024	0.004	0.014	0.009	0.004	1.000

[120]:
```
sns.heatmap(MLRCorr, xticklabels = MLRCorr.columns, yticklabels = MLRCorr.
 ↪columns, cmap = 'RdBu')
```

[120]:
```
<AxesSubplot:>
```



[121]:
```
# signs of multicollinearity
```

[122]:
```
RegressionBefore = MLRdf
RegressionAfter = MLRdf.drop(['sqft_above', 'sqft_basement'], axis = 1)


X1 = sm.tools.add_constant(RegressionBefore)
X2 = sm.tools.add_constant(RegressionAfter)

Series_before = pd.Series([variance_inflation_factor(X1.values, i) for i in␣
 ↪range(X1.shape[1])], index = X1.columns)
Series_after = pd.Series([variance_inflation_factor(X2.values, i) for i in␣
 ↪range(X2.shape[1])], index = X2.columns)

print('Data Before')
print('-' * 100)
```

```python
display(Series_before)

print('Data After')
print('-' * 100)
display(Series_after)
```

```
C:\Users\kangb\anaconda3\lib\site-
packages\statsmodels\stats\outliers_influence.py:193: RuntimeWarning: divide by
zero encountered in double_scalars
  vif = 1. / (1. - r_squared_i)
```

```
Data Before
--------------------------------------------------------------------------------
--------------------

const             7998.867
price                2.639
sqft_living            inf
sqft_lot             1.058
sqft_above             inf
sqft_basement          inf
bedrooms             1.745
bathrooms            3.130
yr_renovated         1.110
yr_built             1.849
grades_bin_2         6.849
grades_bin_3         7.930
view_1               1.025
view_2               1.056
view_3               1.073
view_4               1.146
condition_bin_2      1.023
dtype: float64
```

```
Data After
--------------------------------------------------------------------------------
--------------------

const             7347.198
price                2.632
sqft_living          4.425
sqft_lot             1.053
bedrooms             1.744
bathrooms            3.118
yr_renovated         1.107
yr_built             1.689
grades_bin_2         6.843
grades_bin_3         7.912
view_1               1.019
```

```
view_2               1.048
view_3               1.060
view_4               1.134
condition_bin_2      1.023
dtype: float64
```

[123]:
```
#VIF tells us which features to remove due to multicollinearity, for our case,␣
↪we got rid of sqft_above and sqft_basement
```

[124]:
```
X = RegressionAfter.drop('price', axis = 1)
y = RegressionAfter['price']

train_X, valid_X, train_y, valid_y = train_test_split(X, y, test_size=0.4,␣
↪random_state=1)
```

[125]:
```
regression_model = LinearRegression()

regression_model.fit(train_X, train_y)
```

[125]:
```
LinearRegression()
```

[126]:
```
print('intercept ', regression_model.intercept_)
print(pd.DataFrame({'Predictor': X.columns, 'Coefficient': regression_model.
↪coef_}))
regressionSummary(train_y, regression_model.predict(train_X))
```

```
intercept  5264320.180957988
          Predictor  Coefficient
0       sqft_living      220.696
1          sqft_lot       -0.384
2          bedrooms   -45492.328
3         bathrooms    74779.964
4       yr_renovated      13.890
5          yr_built    -2712.991
6       grades_bin_2    83088.199
7       grades_bin_3   375719.056
8             view_1   160681.736
9             view_2    89863.875
10            view_3   141819.285
11            view_4   521419.907
12   condition_bin_2    43697.109

Regression statistics

                      Mean Error (ME) : -0.0000
       Root Mean Squared Error (RMSE) : 226533.6970
            Mean Absolute Error (MAE) : 151308.3078
```

```
          Mean Percentage Error (MPE) : -11.1270
Mean Absolute Percentage Error (MAPE) : 31.3334
```

[127]:
```
pred_y = regression_model.predict(train_X)

print('adjusted r2: ', adjusted_r2_score(train_y, pred_y, regression_model))
```

```
adjusted r2:  0.6163199477937301
```

[128]:
```
house_lm_pred = regression_model.predict(valid_X)

result = pd.DataFrame({'Predicted': house_lm_pred, 'Actual': valid_y,
                       'Residual': valid_y - house_lm_pred})

print(result.head(5))
```

```
         Predicted   Actual    Residual
13224   507967.979   375000 -132967.979
20405   663935.362   530000 -133935.362
5657   1510791.019  1600000   89208.981
275     583094.073   365000 -218094.073
17614   248585.332   239950   -8635.332
```

[129]:
```
regressionSummary(valid_y, house_lm_pred)
```

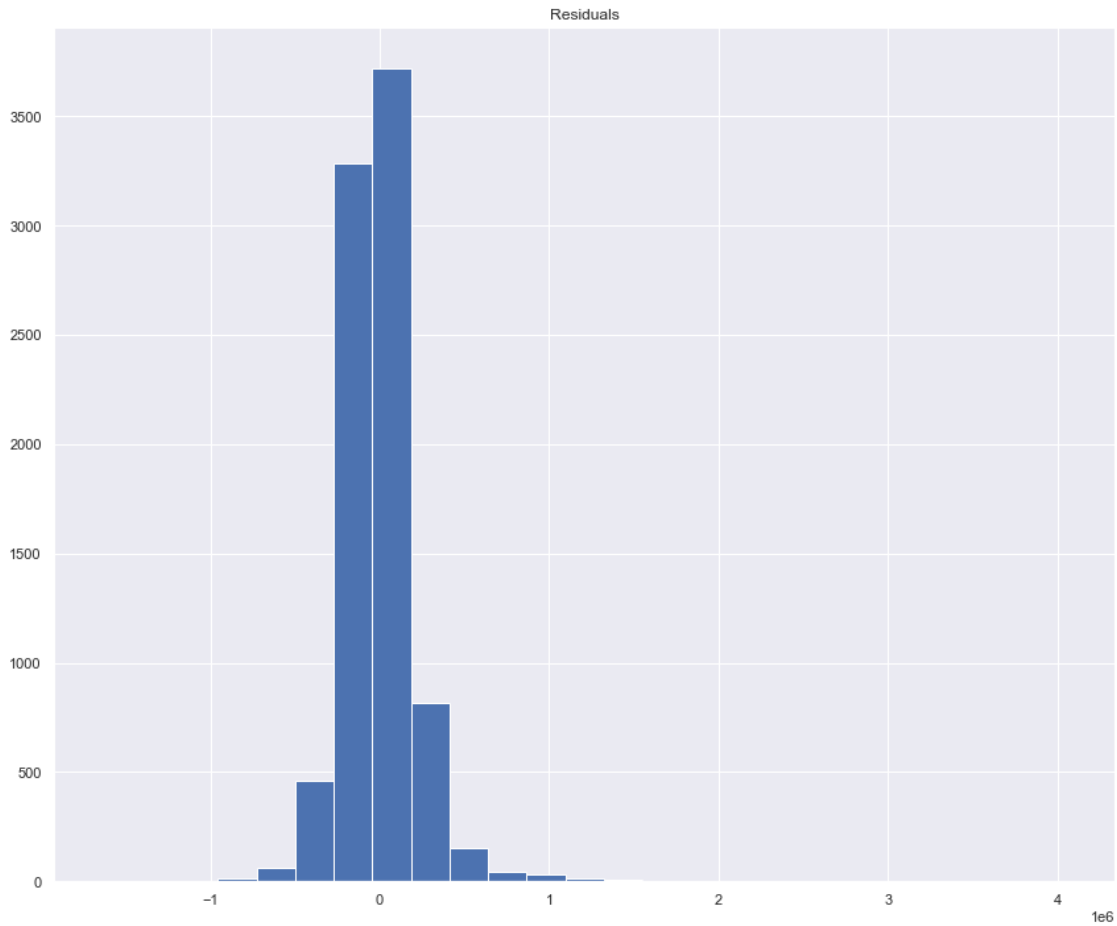
```
Regression statistics

                      Mean Error (ME) : -2299.6034
       Root Mean Squared Error (RMSE) : 226777.3056
            Mean Absolute Error (MAE) : 151003.1176
          Mean Percentage Error (MPE) : -11.6748
Mean Absolute Percentage Error (MAPE) : 31.7286
```

[130]:
```
# From these results we can see that our model is over predicting due to␣
↪negative mean percentage error.
# we also see that our errors do not have huge margins indicating that we are␣
↪not overfitting data.
# Adj r^2 came up to 61% indicating that we can explain that percentage of␣
↪variance in our model.
```

[131]:
```
all_residuals = valid_y - house_lm_pred

print(len(all_residuals[(all_residuals > -226533.6970) & (all_residuals <␣
↪226533.6970)]) / len(all_residuals))

ax = pd.DataFrame({'Residuals': all_residuals}).hist(bins=25)

plt.tight_layout()
```

```python
plt.show()
```

0.816728452270621



[132]:
```python
# 81% of our errors are between our +/- Root Mean Squared Error 226533.70
```

[133]:
```python
def train_model(variables):
    model = LinearRegression()
    model.fit(train_X[variables], train_y)
    return model

def score_model(model, variables):
    pred_y = model.predict(train_X[variables])
    # we negate as score is optimized to be as low as possible
    return -adjusted_r2_score(train_y, pred_y, model)

allVariables = train_X.columns
results = exhaustive_search(allVariables, train_model, score_model)
```

```
data = []
for result in results:
    model = result['model']
    variables = result['variables']
    AIC = AIC_score(train_y, model.predict(train_X[variables]), model)

    d = {'n': result['n'], 'r2adj': -result['score'], 'AIC': AIC}
    d.update({var: var in result['variables'] for var in allVariables})
    data.append(d)
pd.set_option('display.width', 100)
print(pd.DataFrame(data, columns=('n', 'r2adj', 'AIC') +␣
 ↪tuple(sorted(allVariables))))
pd.reset_option('display.width')
```

```
     n  r2adj         AIC  bathrooms  bedrooms  condition_bin_2  grades_bin_2
grades_bin_3  \
0    1  0.490 359772.849      False     False            False         False
False
1    2  0.528 358766.369      False     False            False         False
True
2    3  0.563 357777.695      False     False            False         False
True
3    4  0.590 356927.081      False     False            False         False
True
4    5  0.598 356691.207       True     False            False         False
True
5    6  0.605 356445.097       True      True            False         False
True
6    7  0.608 356360.251       True      True            False         False
True
7    8  0.611 356270.447       True      True            False         False
True
8    9  0.613 356187.331       True      True            False         False
True
9   10  0.615 356118.517       True      True            False         False
True
10  11  0.616 356096.653       True      True            False          True
True
11  12  0.616 356091.242       True      True            False          True
True
12  13  0.616 356089.318       True      True             True          True
True

    sqft_living  sqft_lot  view_1  view_2  view_3  view_4  yr_built
yr_renovated
0          True     False   False   False   False   False     False
False
```

```
1          True     False    False    False    False    False     False
False
2          True     False    False    False    False    False      True
False
3          True     False    False    False    False     True      True
False
4          True     False    False    False    False     True      True
False
5          True     False    False    False    False     True      True
False
6          True     False    False    False     True     True      True
False
7          True     False     True    False     True     True      True
False
8          True     False     True     True     True     True      True
False
9          True      True     True     True     True     True      True
False
10         True      True     True     True     True     True      True
False
11         True      True     True     True     True     True      True
True
12         True      True     True     True     True     True      True
True
```

[134]:

```
#Forward
def train_model(variables):
    if len(variables) == 0:
        return None
    model = LinearRegression()
    model.fit(train_X[variables], train_y)
    return model

def score_model(model, variables):
    if len(variables) == 0:
        return AIC_score(train_y, [train_y.mean()] * len(train_y), model, df=1)
    return AIC_score(train_y, model.predict(train_X[variables]), model)

best_model, best_variables = forward_selection(train_X.columns, train_model,␣
 ↪score_model, verbose=True)

print(best_variables)
```

```
Variables: sqft_living, sqft_lot, bedrooms, bathrooms, yr_renovated, yr_built,
grades_bin_2, grades_bin_3, view_1, view_2, view_3, view_4, condition_bin_2
Start: score=368479.81, constant
Step: score=359772.85, add sqft_living
Step: score=358766.37, add grades_bin_3
```

```
Step: score=357777.70, add yr_built
Step: score=356927.08, add view_4
Step: score=356691.21, add bathrooms
Step: score=356445.10, add bedrooms
Step: score=356360.25, add view_3
Step: score=356270.45, add view_1
Step: score=356187.33, add view_2
Step: score=356118.52, add sqft_lot
Step: score=356096.65, add grades_bin_2
Step: score=356091.24, add yr_renovated
Step: score=356089.32, add condition_bin_2
Step: score=356089.32, add None
['sqft_living', 'grades_bin_3', 'yr_built', 'view_4', 'bathrooms', 'bedrooms',
'view_3', 'view_1', 'view_2', 'sqft_lot', 'grades_bin_2', 'yr_renovated',
'condition_bin_2']
```

[135]:
```
#Backward
def train_model(variables):
    model = LinearRegression()
    model.fit(train_X[variables], train_y)
    return model

def score_model(model, variables):
    return AIC_score(train_y, model.predict(train_X[variables]), model)

best_model, best_variables = backward_elimination(train_X.columns, train_model,␣
↪score_model, verbose=True)

print(best_variables)
```

```
Variables: sqft_living, sqft_lot, bedrooms, bathrooms, yr_renovated, yr_built,
grades_bin_2, grades_bin_3, view_1, view_2, view_3, view_4, condition_bin_2
Start: score=356089.32
Step: score=356089.32, remove None
['sqft_living', 'sqft_lot', 'bedrooms', 'bathrooms', 'yr_renovated', 'yr_built',
'grades_bin_2', 'grades_bin_3', 'view_1', 'view_2', 'view_3', 'view_4',
'condition_bin_2']
```

[136]:
```
#stepwise
best_model, best_variables = stepwise_selection(train_X.columns, train_model,␣
↪score_model, verbose=True)

print(best_variables)
```

```
       ␣
↪---------------------------------------------------------------------------
```

```
    ValueError                                Traceback (most recent call␣
↪last)

    <ipython-input-136-f777a52f7eba> in <module>
      1 #stepwise
----> 2 best_model, best_variables = stepwise_selection(train_X.columns,␣
↪train_model, score_model, verbose=True)
      3
      4 print(best_variables)


    ~\anaconda3\lib\site-packages\dmba\featureSelection.py in␣
↪stepwise_selection(variables, train_model, score_model, direction, verbose)
    147     # we start with a model that contains no variables
    148     best_variables = [] if 'forward' in directions else␣
↪list(variables)
--> 149     best_model = train_model(best_variables)
    150     best_score = score_model(best_model, best_variables)
    151     if verbose:


    <ipython-input-135-a464e679b438> in train_model(variables)
      2 def train_model(variables):
      3     model = LinearRegression()
----> 4     model.fit(train_X[variables], train_y)
      5     return model
      6


    ~\anaconda3\lib\site-packages\sklearn\linear_model\_base.py in fit(self,␣
↪X, y, sample_weight)
    503
    504         n_jobs_ = self.n_jobs
--> 505         X, y = self._validate_data(X, y, accept_sparse=['csr',␣
↪'csc', 'coo'],
    506                                    y_numeric=True, multi_output=True)
    507


    ~\anaconda3\lib\site-packages\sklearn\base.py in _validate_data(self, X,␣
↪y, reset, validate_separately, **check_params)
    430                 y = check_array(y, **check_y_params)
    431             else:
--> 432                 X, y = check_X_y(X, y, **check_params)
    433             out = X, y
    434
```

```
    ~\anaconda3\lib\site-packages\sklearn\utils\validation.py in␣
↪inner_f(*args, **kwargs)
     70                           FutureWarning)
     71         kwargs.update({k: arg for k, arg in zip(sig.parameters,␣
↪args)})
---> 72         return f(**kwargs)
     73     return inner_f
     74


    ~\anaconda3\lib\site-packages\sklearn\utils\validation.py in␣
↪check_X_y(X, y, accept_sparse, accept_large_sparse, dtype, order, copy,␣
↪force_all_finite, ensure_2d, allow_nd, multi_output, ensure_min_samples,␣
↪ensure_min_features, y_numeric, estimator)
    793         raise ValueError("y cannot be None")
    794
--> 795     X = check_array(X, accept_sparse=accept_sparse,
    796                     accept_large_sparse=accept_large_sparse,
    797                     dtype=dtype, order=order, copy=copy,


    ~\anaconda3\lib\site-packages\sklearn\utils\validation.py in␣
↪inner_f(*args, **kwargs)
     70                           FutureWarning)
     71         kwargs.update({k: arg for k, arg in zip(sig.parameters,␣
↪args)})
---> 72         return f(**kwargs)
     73     return inner_f
     74


    ~\anaconda3\lib\site-packages\sklearn\utils\validation.py in␣
↪check_array(array, accept_sparse, accept_large_sparse, dtype, order, copy,␣
↪force_all_finite, ensure_2d, allow_nd, ensure_min_samples,␣
↪ensure_min_features, estimator)
    531
    532         if all(isinstance(dtype, np.dtype) for dtype in dtypes_orig):
--> 533             dtype_orig = np.result_type(*dtypes_orig)
    534
    535     if dtype_numeric:


    <__array_function__ internals> in result_type(*args, **kwargs)


    ValueError: at least one array or dtype is required
```

2 KNN Regression

[138]:
```
#In designing the knn regression model, the goal was to use a limited amount of␣
↪important features to predict price.
#Because KNN can require more computation with more features, we limited␣
↪features primarily to the sqft measures

#sqft_living and sqft_lot are measures of the individual record's interior␣
↪square footage and total lot square footage, respectively.
#sqft_above and sqft_basement measured the individual record's above ground␣
↪level and below ground level square footage, respectively.
#sqft_living15 and sqft_lot15 were measures of the avearages of the interior␣
↪square footage and lot square footage of the
#15 closest houses from the record.
```

[139]:
```
#Create dataframe to hold values relevant to KNN regression
knn_df = df
knn_df = knn_df[['price', 'sqft_living', 'sqft_lot', 'sqft_above',␣
↪'sqft_basement', 'sqft_living15', 'sqft_lot15']]
knn_df.head(5)
```

[139]:
```
    price  sqft_living  sqft_lot  sqft_above  sqft_basement  sqft_living15  \
0  221900         1180      5650        1180              0           1340
1  538000         2570      7242        2170            400           1690
2  180000          770     10000         770              0           2720
3  604000         1960      5000        1050            910           1360
4  510000         1680      8080        1680              0           1800

   sqft_lot15
0        5650
1        7639
2        8062
3        5000
4        7503
```

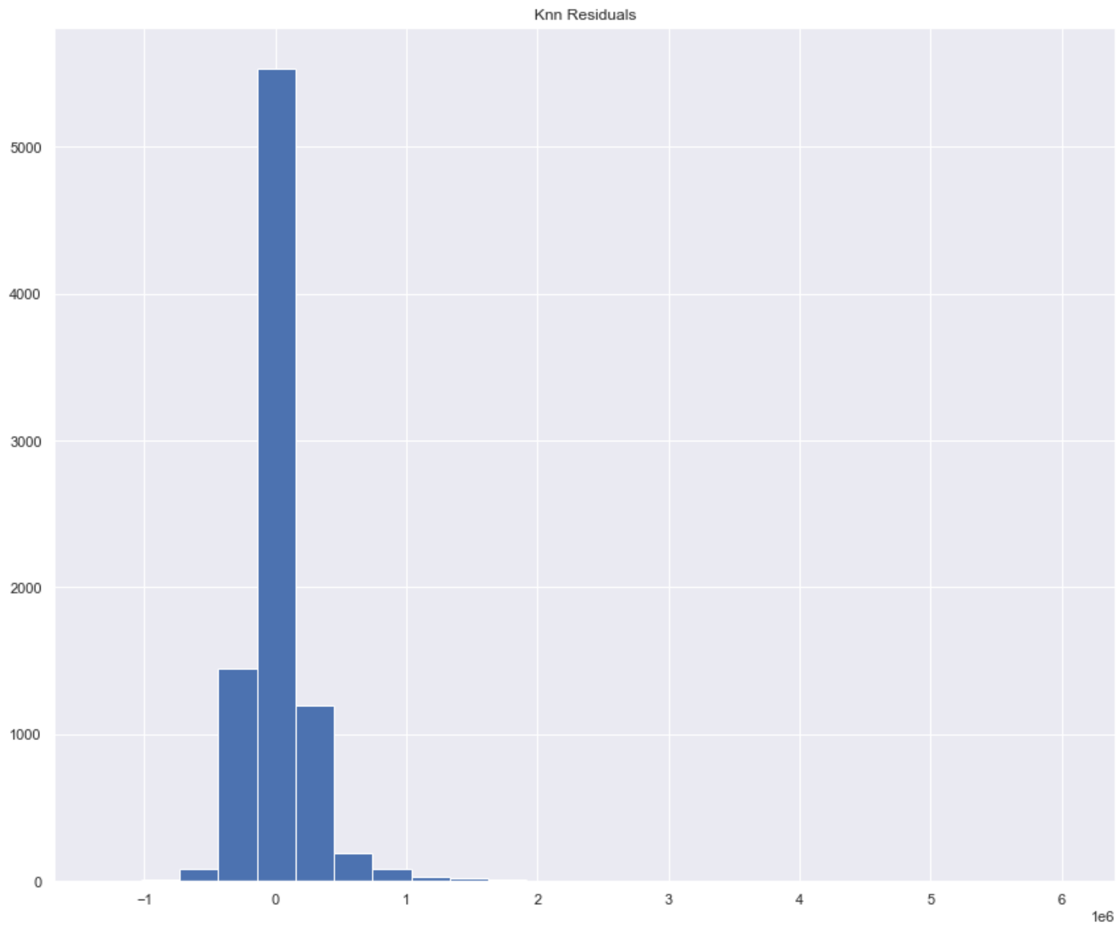
[140]:
```
#Separate dependent and independent variables, and create train and test␣
↪partitions
knn_X = knn_df[['sqft_living', 'sqft_lot', 'sqft_above', 'sqft_basement',␣
↪'sqft_living15', 'sqft_lot15']]
knn_y = knn_df['price']

knn_X_train, knn_X_test, knn_y_train, knn_y_test = train_test_split(knn_X,␣
↪knn_y, test_size=.4, random_state = 0)
```

[141]:
```
#Create knn instance and print Mean Squared Error, R2, and ME, RMSE, MAE, MPE,␣
↪and MAPE for training set

knn_reg = KNeighborsRegressor(11)
knn_reg.fit(knn_X_train, knn_y_train)
knn_y_pred=knn_reg.predict(knn_X_test)

print(mean_squared_error(knn_y_test, knn_y_pred))
print(r2_score(knn_y_test, knn_y_pred))
regressionSummary(knn_y_test, knn_y_pred)
```

```
74178373221.46252
0.4772188334963129

Regression statistics

                      Mean Error (ME) : 24225.8547
       Root Mean Squared Error (RMSE) : 272357.0693
            Mean Absolute Error (MAE) : 157793.3158
          Mean Percentage Error (MPE) : -9.9804
Mean Absolute Percentage Error (MAPE) : 30.8670
```

[142]:
```
#Calculate Knn residuals and create histogram
knn_residuals = knn_y_test - knn_y_pred

# Determine the percentage of datapoints with a residual in approx. 75%
print(len(knn_residuals[(knn_residuals > -168519.2779) & (knn_residuals <␣
↪168519.2779)]) / len(knn_residuals))

knn_residual_df = pd.DataFrame({'Knn Residuals': knn_residuals}).hist(bins=25)

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

```
0.7037766450417052
```




```
[ ]: #Our KNN Residuals histogram shows that the majority of our errors fell within␣
     ↪the $5,000+ dollar range, with the second highest
     #errors occuring within the $1,300-$1,400 dollar range. This means that, for␣
     ↪around 75% of our data set, that we would expect
     #our model's predictions to likely have an error in the range of $5,000.␣
     ↪Considering the prices of homes, this is still not
     #insignificant, and futher enhancements, such as improved model selection,␣
     ↪could improve this.
```

```
[ ]: #Linear Regression statistics

     #                      Mean Error (ME) : -2299.6034
     #       Root Mean Squared Error (RMSE) : 226777.3056
     #            Mean Absolute Error (MAE) : 151003.1176
     #          Mean Percentage Error (MPE) : -11.6748
     # Mean Absolute Percentage Error (MAPE) : 31.7286
```

[ ]:
```
#KNN Regression statistics

#                       Mean Error (ME) : 24225.8547
#        Root Mean Squared Error (RMSE) : 272357.0693
#             Mean Absolute Error (MAE) : 157793.3158
#           Mean Percentage Error (MPE) : -9.9804
# Mean Absolute Percentage Error (MAPE) : 30.8670
```

[ ]:
```
#In comparing our two regression models, we see drastic differences in the Mean Error, but RMSE, MAE, MPE and MAPE are all
#relatively similar

#A higher Mean Error in our KNN Regression tells us that the average of ther errors is around $24,225 dollars. This is much
#higher than the Linear Regression's $-2299, meaning to say that on average, our errors are much more likely to be further off.
#However, since the Mean Error is simply the average of errors, it is not always the most useful measure of predictive accuracy,
#as high negative values can be offset by similarly high positive values.

#The Root Mean Squared Error, or RMSE, is the measure of the standard deviation of the residuals. Put another way, it is a measure
#of the variance of our predictions. As the KNN RMSE of $272,357 is larger than the OSL's error at $226,777, we can say that the
#observed variance in our KNN Regression is much larger.

#The Mean Absolute Error represents the average of our absolute errors. This measure attempts to average the absolute distance
#of our predictions from the the actual observed values. Here again, our Linear Regression's MAE outperforms our KNN's error,
# with a mean error of $151,003 to KNN's $157,793.

#The Mean Absolute Percentage Error represents the percentage error in our models. Interestingly, the KNN Regression shows less
#error than the Linear Regression Error, though only within one percent. This could be due to a number of factors, including the
#feature selection.

#Ultimately, we concluded that our Linear Regression Model was better than our KNN Regression Model, though they were very similar.
#The Linear Regression model features lower ME, RMSE< and MAE, and despite having a higher MAPE, it is only within one percent.

#One thing to consider is our dataset has very high positive values, with the highest costing houses ranging in the $7,000,000 dollar
```

```
#category. Thus, these houses are likely to influence the size of errors, as␣
↪the prediction may be much lower than these
#highest outliers.
```

3 Naive Bayes

[143]:
```
#Our goal in using Naive Bayes is to categorize our records based on␣
↪categorical variables such as waterfront, view, condition
#grade and zip code

#To do this, we will create a new categorical variable based on price. We will␣
↪split our records based on three price quartiles
#derived from our data. We will refer to these as "high","middle" and "low"␣
↪income houses

#Using our independent variables, we will try to assign our records into one of␣
↪these three groups to determine if the presence
#of one of the variables, such as waterfront or view, will affect which of the␣
↪groups a record will fall into
```

[144]:
```
#Create a dataframe with varaibles we wish to use
nb_df = df[['price', 'waterfront', 'view', 'condition', 'grade', 'zipcode']]
nb_df.head(5)
```

[144]:

	price	waterfront	view	condition	grade	zipcode
0	221900	0	0	3	7	98178
1	538000	0	0	3	7	98125
2	180000	0	0	3	6	98028
3	604000	0	0	5	7	98136
4	510000	0	0	3	8	98074

[145]:
```
#use qcut to determine our low, middle and high income divisions
pd.qcut(df['price'], q=3, labels = ['low','middle','high'])
```

[145]:
```
0           low
1        middle
2           low
3          high
4        middle
          …
21575       low
21576    middle
21577    middle
21578    middle
21579       low
Name: price, Length: 21580, dtype: category
```

```
Categories (3, object): ['low' < 'middle' < 'high']
```

[146]:
```python
#Add price_cat column with label to our Naive Bayes datatframe
nb_df['price_cat'] = pd.qcut(df['price'], q=3, labels = ['low','middle','high'])
nb_df
```

```
<ipython-input-146-2225c961cf40>: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
  nb_df['price_cat'] = pd.qcut(df['price'], q=3, labels =
['low','middle','high'])
```

[146]:

	price	waterfront	view	condition	grade	zipcode	price_cat
0	221900	0	0	3	7	98178	low
1	538000	0	0	3	7	98125	middle
2	180000	0	0	3	6	98028	low
3	604000	0	0	5	7	98136	high
4	510000	0	0	3	8	98074	middle
…	…	…	…	…	…	…	…
21575	360000	0	0	3	8	98103	low
21576	400000	0	0	3	8	98146	middle
21577	402101	0	0	3	7	98144	middle
21578	400000	0	0	3	8	98027	middle
21579	325000	0	0	3	7	98144	low

[21580 rows x 7 columns]

[147]:
```python
#drop price from dataframe and convert dependent variables to categorical
nb_df.drop(columns=['price'],inplace=True)
nb_df = nb_df.astype('category')
nb_df.head(5)
```

```
C:\Users\kangb\anaconda3\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
  return super().drop(
```

[147]:

	waterfront	view	condition	grade	zipcode	price_cat
0	0	0	3	7	98178	low
1	0	0	3	7	98125	middle
2	0	0	3	6	98028	low
3	0	0	5	7	98136	high

```
4           0    0          3     8    98074     middle
```

[150]:
```python
#Separate independent variables and dependent variable,
nb_independents = ['waterfront', 'view', 'condition', 'grade', 'zipcode']
nb_dependent = 'price_cat'

nb_X = pd.get_dummies(nb_df[nb_independents])
nb_y = nb_df['price_cat']
classes = ['low','middle','high']

#Create train and test partitions
nb_X_train, nb_X_test, nb_y_train, nb_y_test = train_test_split(nb_X, nb_y,␣
↪test_size=0.40, random_state=1)
```

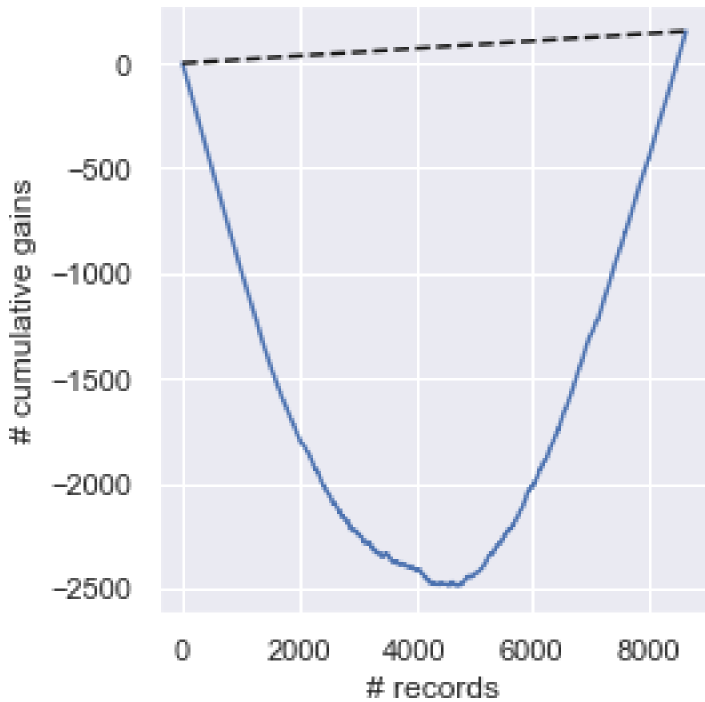
[151]:
```python
#Run Naive Bayes
price_cat_nb = MultinomialNB(alpha=0.01)
price_cat_nb.fit(nb_X_train, nb_y_train)

#Determine probabilities
predProb_train = price_cat_nb.predict_proba(nb_X_train)
predProb_valid = price_cat_nb.predict_proba(nb_X_test)

#Predict and assign class
nb_y_test_pred = price_cat_nb.predict(nb_X_test)
nb_y_train_pred = price_cat_nb.predict(nb_X_train)
```

[152]:
```python
#Print confusion matrix for both train and test sets
classificationSummary(nb_y_train, nb_y_train_pred, class_names=classes)
print()
classificationSummary(nb_y_test, nb_y_test_pred, class_names=classes)
```

```
Confusion Matrix (Accuracy 0.7395)

       Prediction
Actual     low middle   high
   low    3215     54   1119
middle      29   3506    749
  high     606    816   2854

Confusion Matrix (Accuracy 0.7433)

       Prediction
Actual     low middle   high
   low    2043     30    699
middle      12   2412    501
  high     423    551   1961
```

[153]:
```
#As seen above, our accuracy on both the train and test sets was around 74%.␣
↪What this tells us is that, around 74% of the time,
#we can expect that a record would be accurately classified into one of our 3␣
↪price categories. While 26% represents a large
#portion of the dataset, it is safe to say that our classification is accurate␣
↪enough to perhaps give us an insight into the
#different records

#The sensitivity and specificity of our model, meaning to say the ratio of␣
↪correctly identified true positives and true negatives,
#was not as important as in some cases, such as health screening. However, the␣
↪highest noticable error in sensitivity came in the
#"high" price category, with 1119 and 699 "high" predictions in our train and␣
↪tests sets being actually "low" records, respectively.
#The inputs in our Naive Bayes, namely the categorical variables 'waterfront',␣
↪'view', 'condition', 'grade', and 'zipcode',
#are likely to influence the price of a house, but the quality and specificity␣
↪of these measures is not always useful. While
#something like waterfront can seem intuitive, condition and grade are highly␣
↪speculative, and thus are difficult to asses
#when discussing property value.
```

[154]:
```
#Draw cumulative gains chart

cmg_df = pd.DataFrame({'actual': 1 - nb_y_test.cat.codes, 'prob':␣
↪predProb_valid[:, 0]})
cmg_df = cmg_df.sort_values(by=['prob'], ascending=False).reset_index(drop=True)

fig, ax = plt.subplots()
fig.set_size_inches(4, 4)
gainsChart(cmg_df.actual, ax=ax)

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




```
[ ]: #Our cumulative gains chart attempts to measure the predictive accuracy of our␣
      ↪classification. Noticably, our chart shows a negative
     #curve, meaning to say that, the more records are added, our ability to␣
      ↪accurately classify a record decreases. This is slightly
     #confusing, but our best guess is that
```

4 Cluster analysis

```
[ ]: #We will attempt to use cluster analysis to cluster our records to create␣
      ↪similar groups. We will use a dendogram with average linkage,
     #as well as Kmeans, to find the ideal number of clusters, locate the centroids␣
      ↪of these clusters, and plot our centroids as well
     #as creating a heatmap of the clusters
```

```
[3]: #Initialize cluster analysis dataframe
     clst_norm_df = df[['price', 'sqft_living', 'sqft_lot', 'sqft_above',␣
      ↪'sqft_basement', 'sqft_living15', 'sqft_lot15']]

     #Normalize dataframe
     clst_norm_df = clst_norm_df.apply(preprocessing.scale, axis=0)
```

[4]:
```
#Calculate Euclidean pairwise distances
d = pairwise.pairwise_distances(clst_norm_df, metric='euclidean')

#Create dataframe to show comparison of distance between records
pd.DataFrame(d, columns=df.index, index=df.index).head(5)
```

[4]:
```
         0         1         2             3         4         5         6  \
0  0.000000  2.355596  2.128430  2.462342e+00  1.317067  9.710737  1.576666
1  2.355596  0.000000  3.276518  1.969368e+00  1.463022  7.826964  1.791802
2  2.128430  3.276518  0.000000  3.365109e+00  2.191331  9.253110  1.705902
3  2.462342  1.969368  3.365109  2.980232e-08  2.323216  8.576070  2.736154
4  1.317067  1.463022  2.191331  2.323216e+00  0.000000  8.677286  0.940894

         7         8         9  …     21573     21574     21575     21576  \
0  0.557366  1.897073  1.939530  …  2.363881  0.785977  2.972361  4.392447
1  2.404902  1.962271  1.693143  …  1.291901  1.794730  1.584160  2.139315
2  1.660069  2.439687  1.926104  …  2.682724  2.376608  3.094776  4.908896
3  2.477565  1.278574  2.849974  …  2.890104  2.244174  3.259748  2.931958
4  1.193994  1.974778  1.057330  …  1.182956  0.752146  1.746734  3.380880

     21577     21578     21579     21580     21581     21582
0  0.786096  0.759719  2.031788  0.744835  0.862494  0.633539
1  2.016846  1.749501  1.054045  2.598928  1.668721  2.639312
2  2.353931  2.210455  2.886420  2.602874  2.435073  2.565014
3  1.946368  2.305811  2.735186  2.418934  2.272634  2.476453
4  1.082638  0.676563  1.070012  1.612722  0.707868  1.667473

[5 rows x 21583 columns]
```

[167]:
```
#Use elbow methodology to determine best number of clusters

inertia = []
for n_clusters in range(1, 15):
    kmeans = KMeans(n_clusters=n_clusters, random_state=0).fit(clst_norm_df)

    inertia.append(kmeans.inertia_ / n_clusters)
inertias = pd.DataFrame({'n_clusters': range(1, 15), 'inertia': inertia})
ax = inertias.plot(x='n_clusters', y='inertia')
plt.xlabel('Number of clusters(k)')
plt.ylabel('Average Within-Cluster Squared Distances')
plt.ylim((0, 1.1 * inertias.inertia.max()))
ax.legend().set_visible(False)
plt.show()
```

```
␣
↪---------------------------------------------------------------------------
```

```
KeyboardInterrupt                         Traceback (most recent call␣
↪last)

<ipython-input-167-13a7e42f14ec> in <module>
      3 inertia = []
      4 for n_clusters in range(1, 15):
----> 5     kmeans = KMeans(n_clusters=n_clusters, random_state=0).
↪fit(clst_norm_df)
      6
      7     inertia.append(kmeans.inertia_ / n_clusters)


~\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py in fit(self, X,␣
↪y, sample_weight)
   1066         for seed in seeds:
   1067             # run a k-means once
-> 1068             labels, inertia, centers, n_iter_ = kmeans_single(
   1069                 X, sample_weight, self.n_clusters, max_iter=self.
↪max_iter,
   1070                 init=init, verbose=self.verbose, tol=self._tol,


~\anaconda3\lib\site-packages\sklearn\cluster\_kmeans.py in␣
↪_kmeans_single_elkan(X, sample_weight, n_clusters, max_iter, init, verbose,␣
↪x_squared_norms, random_state, tol, n_threads)
    427         # center of each center for next iterations
    428         center_half_distances = euclidean_distances(centers_new) / 2
--> 429         distance_next_center = np.partition(
    430             np.asarray(center_half_distances), kth=1, axis=0)[1]
    431


<__array_function__ internals> in partition(*args, **kwargs)


~\anaconda3\lib\site-packages\numpy\core\fromnumeric.py in partition(a,␣
↪kth, axis, kind, order)
    746     else:
    747         a = asanyarray(a).copy(order="K")
--> 748     a.partition(kth, axis=axis, kind=kind, order=order)
    749     return a
    750


KeyboardInterrupt:
```

```
[ ]: #Print centroids

     centroids = pd.DataFrame(kmeans.cluster_centers_, columns=clst_norm_df.columns)
     pd.set_option('precision', 3)
     print(centroids)
     pd.set_option('precision', 6)
```

```
[ ]: #Calculate the distances of each data point to the cluster centers
     distances = kmeans.transform(clst_norm_df)

     #Reduce to the minimum squared distance of to the centroids
     minSquaredDistances = distances.min(axis=1) ** 2

     #Combine with cluster labels into a data frame
     clst_df = pd.DataFrame({'squaredDistance': minSquaredDistances, 'cluster':␣
      ↪kmeans.labels_},
         index=clst_norm_df.index)

     #Group by cluster and print information
     for cluster, data in clst_df.groupby('cluster'):
         count = len(data)
         withinClustSS = data.squaredDistance.sum()
         print(f'Cluster {cluster} ({count} members): {withinClustSS:.2f} within␣
      ↪cluster ')
```

```
[ ]: #Print centroid clusters

     centroids['cluster'] = ['Cluster {}'.format(i) for i in centroids.index]

     fig = plt.figure(figsize=(10,6))
     fig.subplots_adjust(right=3)
     ax = parallel_coordinates(centroids, class_column='cluster', colormap='Dark2',␣
      ↪linewidth=5)
     plt.legend(loc='center left', bbox_to_anchor=(0.95, 0.5))
     plt.xlim(-0.5,7.5)
     centroids
```

```
[ ]: #Euclidean distance between centroids

     print(pd.DataFrame(pairwise.pairwise_distances(kmeans.cluster_centers_,␣
      ↪metric='euclidean')))
```

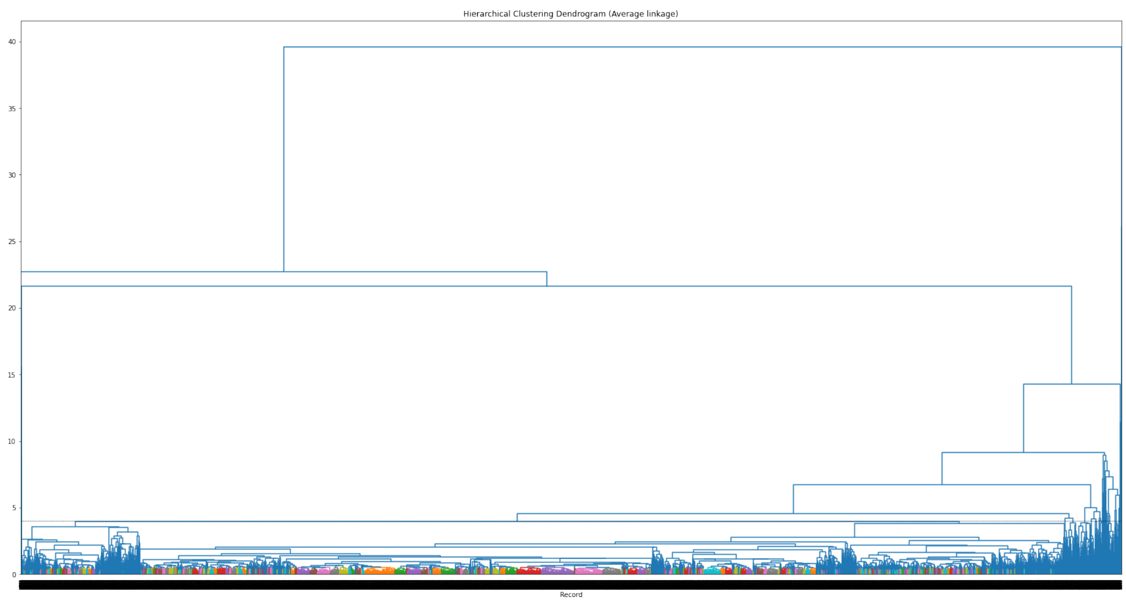
```
[ ]: #Sum of distances from a centroid to other centroids

     pd.DataFrame(pairwise.pairwise_distances(kmeans.cluster_centers_,␣
      ↪metric='euclidean')).sum(axis=0)
```

[ ]:

```
#Compute pairwise distance based on price
d_norm = pairwise.pairwise_distances(clst_norm_df[['price']],
                                     metric='euclidean')
pd.DataFrame(d_norm, columns=clst_norm_df.index, index=clst_norm_df.index).
↪head(5)
```

[5]:

```
#Create histogram

Z = linkage(clst_norm_df, method='average')

fig = plt.figure(figsize=(30, 18))
fig.subplots_adjust(bottom=0.23)
plt.title('Hierarchical Clustering Dendrogram (Average linkage)')
plt.xlabel('Record')
dendrogram(Z, labels= clst_norm_df.index, color_threshold=.5)
plt.axhline(y=4, color='black', linewidth=0.5, linestyle='dashed')
plt.show()
```



[ ]:

```
#Create heatmap of clusters

memb = fcluster(linkage(clst_norm_df, 'average'), 6, criterion='maxclust')

clst_norm_df.index = ['{}: {}'.format(cluster, state) for cluster, state in␣
↪zip(memb, clst_norm_df.index)]
sns.clustermap(clst_norm_df, method='average', col_cluster=False, ␣
↪cmap="mako_r")
plt.show()
```

5 Classification Trees

[174]:
```
Treesdf = df
```

[175]:
```
pd.set_option('display.float_format', lambda x: '%.3f' % x)
```

[176]:
```
 df['price'].describe()
```

[176]:
```
count     21580.000
mean     540381.237
std      367488.431
min       78000.000
25%      322000.000
50%      450000.000
75%      645000.000
max     7700000.000
Name: price, dtype: float64
```

[177]:
```
Treesdf['price_bin'] = pd.qcut(Treesdf['price'], q = 3, labels = ['low',␣
 ↪'middle', 'high'])
classes = ['low','middle','high']
```

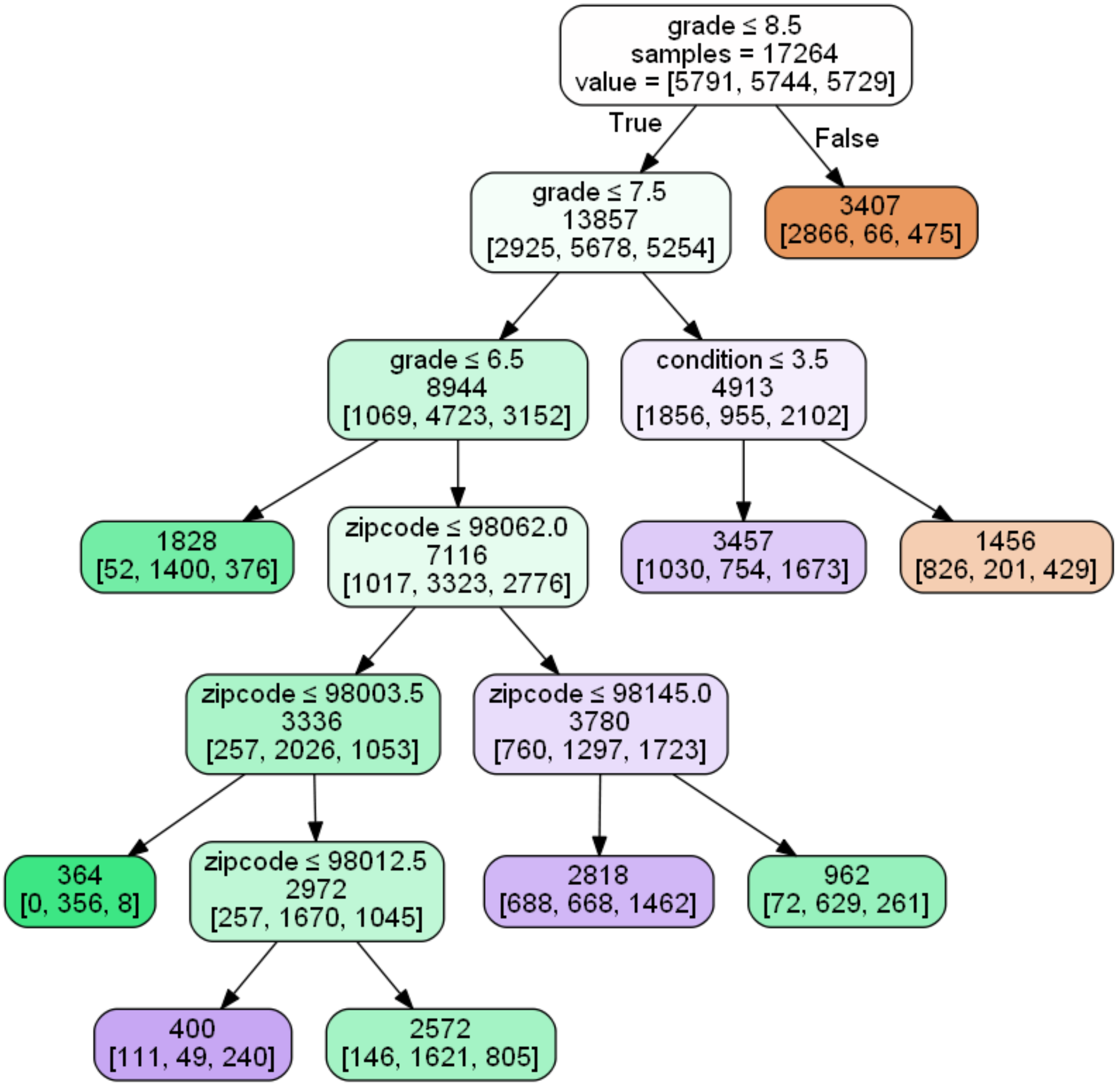
[178]:
```
y = Treesdf['price_bin']
X = Treesdf[['grade', 'zipcode', 'waterfront', 'condition']]
```

[179]:
```
train_X, valid_X, train_y, valid_y = train_test_split(X, y, test_size=0.20,␣
 ↪random_state=1)
ClassTree = DecisionTreeClassifier(min_samples_split=10,␣
 ↪min_impurity_decrease=0.005)
ClassTree.fit(train_X, train_y)
```

[179]:
```
DecisionTreeClassifier(min_impurity_decrease=0.005, min_samples_split=10)
```

[180]:
```
plotDecisionTree(ClassTree, feature_names = train_X.columns)
```

[180]:



[181]:
```
classificationSummary(train_y, ClassTree.predict(train_X), class_names =␣
 ↪classes)
classificationSummary(valid_y, ClassTree.predict(valid_X), class_names =␣
 ↪classes)
```

```
Confusion Matrix (Accuracy 0.6414)

       Prediction
Actual    low middle   high
   low   3692    270   1829
middle    267   4006   1471
  high    904   1450   3375
Confusion Matrix (Accuracy 0.6399)
```

```
        Prediction
Actual     low middle   high
   low     868     62    439
middle      63   1042    360
  high     238    392    852
```

[ ]: