In [1]:

```
%%html
<!--This is just for personal preference.-->
<style>
h1 {padding: 9px; color: Green; border-bottom: 3px solid Green; text-align: center;}
h2 {padding: 9px; color: MediumBlue; border-bottom: 3px solid MediumBlue; text-align: center;}
h3 {padding: 9px; color: firebrick; border-bottom: 3px solid firebrick; text-align: center;}
h4 {padding: 9px; color: olive; border-bottom: 3px solid olive; text-align: center;}
h5 {padding: 9px; color: aquamarine; border-bottom: 3px solid aquamarine; text-align: center;}
summary {color: DarkOrange;}
td {text-align: center;}
</style>
```

King County Development

An Academic Multiple Regression Analysis Project

by Devin Sarnataro (www.linkedin.com/in/devin-sarnataro-0b639b148)

September 15th, 2022

Table of Contents

Project Overview

For this project, Flatiron School (https://flatironschool.com/) provided me with a dataset of residential property sales in King County, Washington. We were instructed to create a hypothetical stakeholder, or client, and business problem that could be addressed with a multiple regression analysis, in contrast to the first project in which we were provided with both. For students who were struggling to define a stakeholder, they recommended a real estate agency that assists homeowners in selling their homes, with the specific business problem of providing the agency with the analysis necessary to tell their clients whether renovations really do increase the value of a house and if so, by how much.

Along with the dataset, Flatiron also provided me with a `.md` file with the column names and a brief description of each column (https://github.com/sarnadpy32/king_county_development/blob/master/data/column_names.md). I've also included the columns and their descriptions below if you click on the collapsible section.

— Click Here to see the Column Names and Descriptions. —

Two of the columns, `condition` and `grade`, had categories that required me to check the King County Assessor's glossary of terms (https://info.kingcounty.gov/assessor/esales/Glossary.aspx?type=r) to see what the entities within the columns specifically meant. I also used the opportunity to explore the site a little, and by doing so I was able to obtain a map of the county from the county government's iMap (https://gismaps.kingcounty.gov/iMap/) feature.

— Click Here to see a Map of King County. —

Stakeholder & Business Problem

Based on the project description, and prior to any exploration of the dataset, I already had real estate agency in mind as a client, but I didn't want to limit myself before I began my analysis. My initial thoughts were that areas with rapidly increasing home prices would be useful to a real estate agency, while areas with homes prices that were stagnant, or were perhaps even decreasing, would be useful to a state government agency, or maybe even some sort of charitable organization. Those areas may be in need of economic development or support. Even if I did end up going with a real estate agency, I was hoping to provide my hypothetical client with insights concerning other possible revelations that could be attained through a thorough analysis of the data. So, I began my analysis without identifying a client or business problem, with the purpose of seeing what was possible first.

Understanding & Preparing the Data

Importing the Necessary Modules and Functions

I started by importing the modules and functions that I knew I would most likely need. If I needed to import any others at any point, I would come back here and add them to this cell to keep everything organized. I also created some output formatters that I commonly use so that I wouldn't have to rewrite them each time.

In [2]:
```
# Importing modules and functions

import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

import os

import numpy as np
import pandas as pd
import scipy.stats as stats
import seaborn as sns
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import matplotlib.lines as mlines
from matplotlib.colors import LinearSegmentedColormap
%matplotlib inline

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

from sklearn.impute import MissingIndicator
from sklearn.preprocessing import OrdinalEncoder, OneHotEncoder, StandardScaler

from sklearn.feature_selection import RFECV
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import ShuffleSplit
```

In [3]:
```
# Output formatters

bold_red = '\033[31m\033[1m'
every_off = '\033[0m'

print(bold_red +'Making sure '+ every_off +'they worked '+ bold_red +'as intended.'+ every_off)
```

```
Making sure they worked as intended.
```

Exploring the Data

I began my analysis by importing the dataset provided to me by Flatiron School (https://flatironschool.com/). I decided to sort the dataframe by `id`, as it would make identifying any potential duplicates easier. During my analysis, I also found out that there were 70 different ZIP codes, so I downloaded the free ZIP code database available at this page (https://www.unitedstateszipcodes.org/zip-code-database/), to replace the ZIP codes with the appropriate city names. This reduced the number of categories in the column to 24 and served as a more useful tool for analysis.

In [4]:
```
# Importing and investigating the data

kc_house_data = pd.read_csv('data/kc_house_data.csv')
kc_house_data = kc_house_data.sort_values('id')

display(kc_house_data.head(3))
kc_house_data.info()
```

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sqft_above	sqft_baser
2495	1000102	4/22/2015	300000.0	6	3.00	2400	9373	2.0	NO	NONE	...	7 Average	2400	
2494	1000102	9/16/2014	280000.0	6	3.00	2400	9373	2.0	NaN	NONE	...	7 Average	2400	
6729	1200019	5/8/2014	647500.0	4	1.75	2060	26036	1.0	NaN	NONE	...	8 Good	1160	9

3 rows × 21 columns

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21597 entries, 2495 to 15937
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   id             21597 non-null  int64
 1   date           21597 non-null  object
 2   price          21597 non-null  float64
 3   bedrooms       21597 non-null  int64
 4   bathrooms      21597 non-null  float64
 5   sqft_living    21597 non-null  int64
 6   sqft_lot       21597 non-null  int64
 7   floors         21597 non-null  float64
 8   waterfront     19221 non-null  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(6), int64(9), object(6)
memory usage: 3.6+ MB
```

In [5]:
```
# Checking the number of different ZIP codes

kc_house_data.zipcode.nunique()
```

Out[5]: 70

In [6]:
```
# Importing an outside dataset to replace the ZIP codes

zip_code_data = pd.read_csv('data/zip_code_database.csv')

zip_code_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 42724 entries, 0 to 42723
Data columns (total 15 columns):
 #   Column                     Non-Null Count  Dtype  
---  ------                     --------------  -----  
 0   zip                        42724 non-null  int64  
 1   type                       42724 non-null  object 
 2   decommissioned             42724 non-null  int64  
 3   primary_city               42724 non-null  object 
 4   acceptable_cities          9287 non-null   object 
 5   unacceptable_cities        11666 non-null  object 
 6   state                      42724 non-null  object 
 7   county                     41799 non-null  object 
 8   timezone                   41926 non-null  object 
 9   area_codes                 39698 non-null  object 
 10  world_region               333 non-null    object 
 11  country                    42650 non-null  object 
 12  latitude                   42724 non-null  float64
 13  longitude                  42724 non-null  float64
 14  irs_estimated_population   42724 non-null  int64  
dtypes: float64(2), int64(3), object(10)
memory usage: 4.9+ MB
```

In [7]:
```
# Setting up the outside dataset to easily use it to replace the ZIP codes with the their corresponding cities

zip_cities = zip_code_data[['zip', 'primary_city']].set_index('zip')
```

In [8]:
```
# Replacing the ZIP codes with the their corresponding cities

kc_house_data.zipcode = kc_house_data.zipcode.map(lambda x: zip_cities.loc[x]['primary_city'])
```

In [9]:
```
# Renaming the column

kc_house_data.rename(columns={'zipcode': 'city'}, inplace=True)
```

In [10]:
```
# Checking how effective replacing ZIP code with City was in reducing the number of categories

kc_house_data.city.nunique()
```

Out[10]: 24

Duplicated ID's

I found out there were sales with duplicate `id` s. Before I could begin my analysis, I had to determine if these were resales or errors? I first changed the `date` column to make comparisons easier, and it also allowed me to easily see the period of time covered by the dataset, which was only a single year's worth of sales, specifically from May 2014 to May 2015. I then isolated all the duplicated `id` s into a single dataframe to investigate my concerns. After checking, they do seem do be resales and there were therefore no duplicates that needed to be dropped.

In [11]:
```
# There were obviously duplicates based on the number of unique ID's

kc_house_data.id.nunique()
```

Out[11]: 21420

In [12]:
```
# Changing the date column to datetime format and aanlyzing the time period

kc_house_data.date = pd.to_datetime(kc_house_data.date)

kc_house_data.date.describe(datetime_is_numeric=True)
```

Out[12]:
```
count                            21597
mean     2014-10-29 04:20:38.171968512
min                2014-05-02 00:00:00
25%                2014-07-22 00:00:00
50%                2014-10-16 00:00:00
75%                2015-02-17 00:00:00
max                2015-05-27 00:00:00
Name: date, dtype: object
```

In [13]:
```
# Isolating the potential duplicates in their own df

dup_df = kc_house_data.loc[kc_house_data.id.duplicated(keep=False)].copy()
```

In [14]:
```
# Investigating the maximum number of duplicated IDs

dup_df.id.value_counts().head(3)
```

Out[14]:
```
795000620     3
1000102       2
5430300171    2
Name: id, dtype: int64
```

In [15]:

```
# Iterating through the unique IDs in the dup_df

for id_x in dup_df.id.unique():
    uniq_id_df = dup_df.loc[dup_df.id==id_x].sort_values('date')

    p_0, p_1 = uniq_id_df.price.iloc[0], uniq_id_df.price.iloc[1]

    d_0, d_1 = uniq_id_df.date.iloc[0], uniq_id_df.date.iloc[1]

    # Any duplicated `id`s that had a resale value of less than the orginal value were my primary
    # concern as these could be errors instead of resales and I didn't feel the need to check
    # if the resale value was greater, which would make sense if people were quickly flipping
    # houses / properties
    #-----------------------------------------------------------------------------------------
    if p_1 < p_0: display(uniq_id_df)

    # And of course, any properties that had both the same date and ID would almost certainly be errors
    #-------------------------------------------------------------------------------------------------
    if d_0 == d_1:
        print('\n'+ bold_red +'X---Obvious Error---X'+ every_off +'\n')
        display(uniq_id_df)

    # As there was only one duplicated ID with three duplicates, it was of obvious concern as well
    #----------------------------------------------------------------------------------------------
    if len(uniq_id_df)==3: display(uniq_id_df)
```

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sqft_above	sqft_ba
17588	795000620	2014-09-24	115000.0	3	1.0	1080	6250	1.0	NO	NONE	...	5 Fair	1080	
17589	795000620	2014-12-15	124000.0	3	1.0	1080	6250	1.0	NO	NONE	...	5 Fair	1080	
17590	795000620	2015-03-11	157000.0	3	1.0	1080	6250	1.0	NaN	NONE	...	5 Fair	1080	

3 rows × 21 columns

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	...	grade	sqft_above	sqft_l
15263	2619920170	2014-10-01	772500.0	4	2.5	3230	4290	2.0	NO	NONE	...	9 Better	3230	

Stakeholder and Business Problem Update

As I mentioned in the section above, I discovered that the period of time covered by the dataset was only a single year. My original idea of analyzing the prices of houses in certain areas over time was therefore not going to work, nor any other analysis of the changing effects of the other features over time. While my options for a client were still open, I was going to have to perform an analysis of the effect of the features themselves on the prices of houses in the year concerned.

Understanding & Preparing the Data (cont.)

Features with Missing Data

I started by identifying the columns with missing data. Since there were only 63 entries missing a value in the `view` column, I decided to just drop those entries. I then used the `MissingIndicator()` function from `sklearn` to create an encoded column to be added to the DataFrame for each of the columns that held missing data. I could then fill in the `NaN` 's of those columns with something appropriate so that they wouldn't cause problems during my analysis. For the `waterfront` column, I then used the `OrdinalEncoder()` from `sklearn` to code and replace the column in a binary format.

In [16]:
```
# Determining which columns have missing data

kc_cols_missing = kc_house_data.isna().sum()
kc_cols_missing.loc[kc_cols_missing != 0]
```

Out[16]:
```
waterfront      2376
view              63
yr_renovated    3842
dtype: int64
```

In [17]:
```
# Just got rid of the 63 entries with a missing 'view' value

kc_house_data.dropna(subset=['view'], inplace=True)

# Looping through the two columsns that still had missing data
# Encodding the missing values in separate columns
# Filling the missing values with an appropriate value
# An since the 'waterfront' column is binary, encoded it as such
#-----------------------------------------------------------------

for col in ['waterfront', 'yr_renovated']:

    col_ser = kc_house_data[[col]].copy()
    missing_indicator = MissingIndicator()
    missing_indicator.fit(col_ser)
    missing_ser = missing_indicator.transform(col_ser).astype(int)

    if col == 'waterfront':
        col_ser[col].fillna('NO', inplace=True)
        ord_encoder = OrdinalEncoder(dtype='int64')
        ord_encoder.fit(col_ser)
        enc_ord_col_ser = ord_encoder.transform(col_ser).flatten()
        kc_house_data[col] = enc_ord_col_ser

    if col == 'yr_renovated': kc_house_data[col].fillna(0, inplace=True)

    kc_house_data[col +'_Missing'] = missing_ser
```

I found out later that the `sqft_basement` column also contained a value that was basically equivalent to `NaN`. I came back here to deal with it in the same section.

In [18]:
```
# Repeating the process above but I had to replace the '?'s first

kc_house_data.sqft_basement = \
kc_house_data.sqft_basement.map(lambda x: np.nan if x=='?' else x)

sqft_base_col = kc_house_data[['sqft_basement']]

missing_indicator = MissingIndicator()
missing_indicator.fit(sqft_base_col)
missing_ser = missing_indicator.transform(sqft_base_col).astype(int)

kc_house_data['sqft_basement_Missing'] = missing_ser

kc_house_data.sqft_basement = kc_house_data.sqft_basement.astype('float64')
```

I then dropped the `id`, `date`, `lat`, and `long` columns as they were no longer necessary for my analysis. The `date`, `lat`, and `long` columns were all inappropriate to use as parameters in the model at this point, and the `id` column is inappropriate for linear models in general.

In [19]:
```
# Dropping the unnecessary columns

kc_house_data.drop(['id', 'date', 'lat', 'long'], axis=1, inplace=True)
```

Then I split the data into the target (dependent variable) series and two separate dataframes for the independent variables, one for the numerical independent variables, and one for the categorical independent variables.

In [20]:
```
# Creating the lists of columns for each group

numericals = ['sqft_living', 'sqft_lot', 'sqft_above', 'sqft_living15', 'sqft_lot15', 'yr_built', 'yr_renovated
              'floors', 'sqft_basement', 'bedrooms', 'bathrooms']

categoricals = [col for col in kc_house_data.columns if col not in numericals and col!='price']
```

In [21]:
```
# Splitting the data into appropriate groups

kc_target = kc_house_data['price'].copy()
kc_nums = kc_house_data[numericals].copy()
kc_cats = kc_house_data[categoricals].copy()
```

Feature Distribution Visualizations Exploration

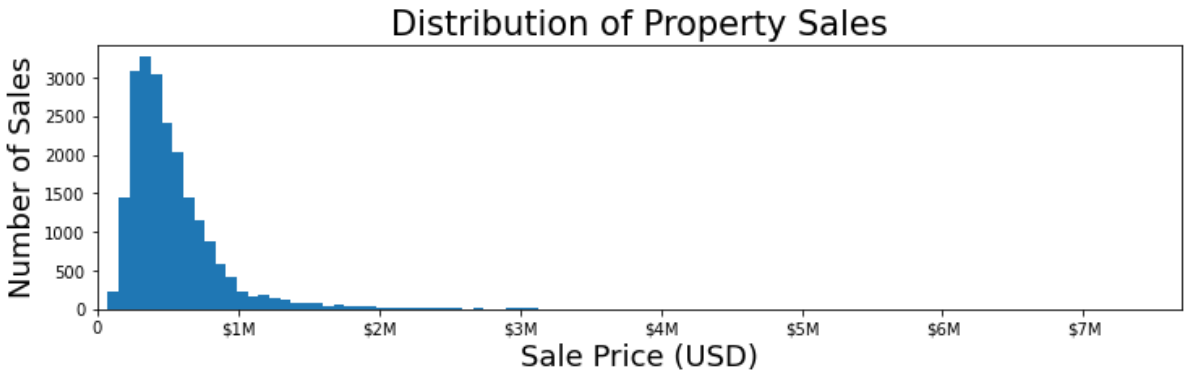
Before I created any visualizations, I wrote functions to properly format the ticks of any visualizations and any `pandas` outputs that contained currency information, as well as a function to get a lighter version of a simple, pre-named color to use in visualizations.

In [22]:
```
# Setting the rcParams that I wanted to use
# Creating the functions I would use to style colors, ticks and dataframes

plt.rcParams.update({'mathtext.fontset':'dejavuserif', 'mathtext.bf':'serif:bold'})

def get_lighter_color(color_name, light_frac):
    color_map = LinearSegmentedColormap.from_list('', [color_name,'w'], N=9)
    return color_map(light_frac)

def viz_percentage_formatter(y, pos):
    if y==0: c = '$\mathbf{0\%}$'
    else: c = '$\mathbf{'+ '{:.2f}'.format(y * 100) +'\%}$'

    return c

def viz_currency_formatter(x, pos):
    if x==0: c = '0'
    if 0<np.abs(x)<1e3: c = '${}'.format(np.abs(x))
    if 1e3<=np.abs(x)<1e6: c = '${}K'.format(round(np.abs(x)*1e-3), 2)
    if 1e6<=np.abs(x)<1e9: c = '${}M'.format(round(np.abs(x)*1e-6), 2)
    if np.abs(x)>=1e9: c = '${}B'.format(round(np.abs(x)*1e-9), 2)

    if x < 0: c = '-'+ c

    return c

def pd_currency_formatter(x):
    if x==0: c = '0'
    if 0<np.abs(x)<1e3: c = '${:f}'.format(np.abs(x))
    if 1e3<=np.abs(x)<1e6: c = '${:f}K'.format(round(np.abs(x)*1e-3), 2)
    if 1e6<=np.abs(x)<1e9: c = '${:f}M'.format(round(np.abs(x)*1e-6), 2)
    if np.abs(x)>=1e9: c = '${:f}B'.format(round(np.abs(x)*1e-9), 2)

    if x < 0: c = '-'+ c

    return c
```

Price Distribution Visualizations

I first checked the distribution of `price` (the target variable).

In [23]:
```
# Price Distribution Visualization

p_max = kc_target.max()

fig, ax = plt.subplots(figsize=(12, 3))

ax.hist(kc_target, bins=100)

ax.set_xbound(0, p_max)
ax.xaxis.set_major_formatter(viz_currency_formatter)

ax.set_xlabel("Sale Price (USD)", size=18)
ax.set_ylabel("Number of Sales", size=18)
ax.set_title("Distribution of Property Sales", size=21)

fig.set_facecolor('w')

fig.savefig('visuals/price_distribution.png' , bbox_inches='tight')

plt.show()
```



In [24]:
```
# Checking the distrubition thru .describe() to confirm the heavy skew

kc_target.describe().apply(pd_currency_formatter)
```

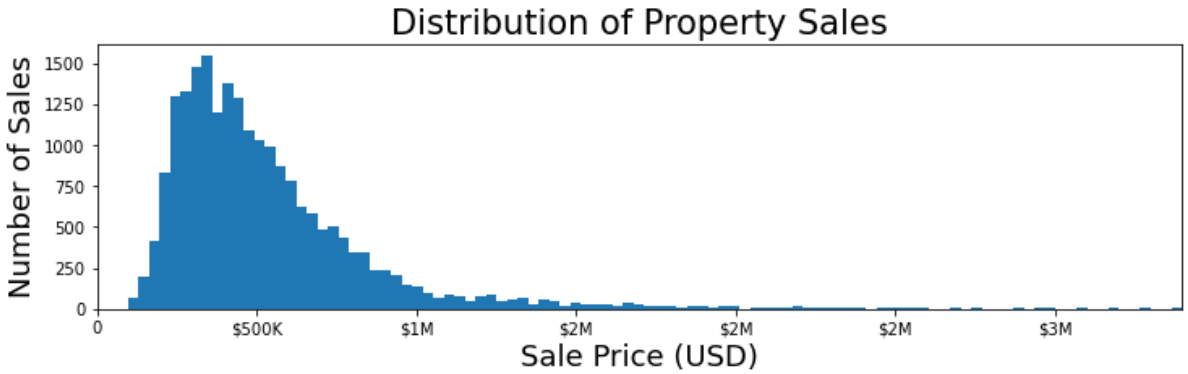
Out[24]:
```
count     $22.000000K
mean     $540.000000K
std      $366.000000K
min       $78.000000K
25%      $322.000000K
50%      $450.000000K
75%      $645.000000K
max        $8.000000M
Name: price, dtype: object
```

There seemed to be a heavy skew which was confirmed by the large difference between the 75th quantile and the max value for the series. I eliminated the outliers so that the performance of the model would be improved.

In [25]:
```
# Eliminating the outliers

kc_target = kc_target.loc[(kc_target >= kc_target.quantile(.001)) & (kc_target <= kc_target.quantile(.999))]
```

In [26]:

```
# Re-creating the distribution visualization again with the outliers removed

p_max = kc_target.max()

fig, ax = plt.subplots(figsize=(12, 3))

ax.hist(kc_target, bins=100)

ax.set_xbound(0, p_max)
ax.xaxis.set_major_formatter(viz_currency_formatter)

ax.set_xlabel("Sale Price (USD)", size=18)
ax.set_ylabel("Number of Sales", size=18)
ax.set_title("Distribution of Property Sales", size=21)

fig.set_facecolor('w')

fig.savefig('visuals/price_distribution_2.png' , bbox_inches='tight')

plt.show()
```

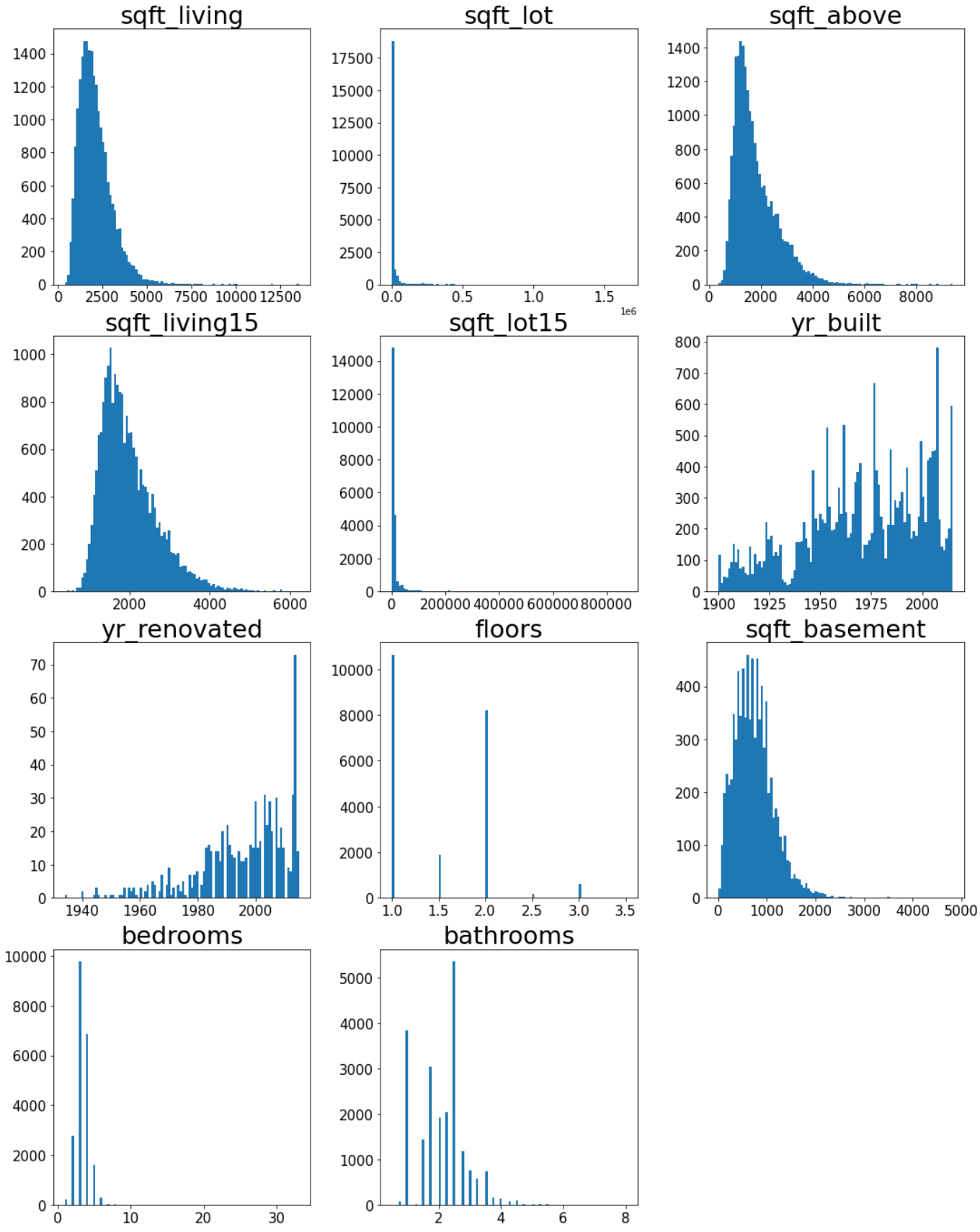


Numerical Features Distribution Visualizations

I then created visualizations to explore the distributions of the numerical features.

In [27]:

```
# Checking the distributions of the categorical features, which I changed accordingly after seeing the results

nums = len(kc_nums.columns)
n_rows = int(nums / 2)#

fig, axs = plt.subplots(nrows=n_rows, ncols=3, figsize=(18, n_rows * 6),
                        gridspec_kw={'wspace': .27})
axs = axs.flatten()

for c_i, col in enumerate(kc_nums.columns):

#     axs[c_i].hist(kc_nums[col], bins=100) # INITIAL EXPLORATION

    # Comment the following IF...ELSE statement and uncomment the line above to see the original
    # distributions
    #-----------------------------------------------------------------------------------------------
    if col not in ['sqft_lot', 'sqft_basement', 'sqft_lot15', 'yr_renovated']:
        axs[c_i].hist(kc_nums[col], bins=100)

    else: axs[c_i].hist(kc_nums[col].loc[kc_nums[col] != 0], bins=100)
    #-----------------------------------------------------------------------------------------------
    axs[c_i].set_title(col, size=27)
    axs[c_i].tick_params('both', labelsize=15)

fig.set_facecolor('w')
[ax.set_visible(False) for ax in axs if not ax.has_data()]

fig.savefig('visuals/numeric_features_distributions.png' , bbox_inches='tight')

plt.show()
```



There seemed to be spikes at `0` for the columns `sqft_lot`, `sqft_basement`, `sqft_lot15`, and `yr_renovated`. By eliminating the `0`'s in the distribution visualizations for those columns, I determined that the `sqft_basement` and `yr_renvoated` columns were heavily affected by removing the `0`'s, while the other two were not as clearly affected. It was therefore useful to turn the `sqft_basement` and `yr_renovated` into categorical features by either turning them into binary encoded features, or by splitting them into more complex categorical features.

I initially explored turning them into more complex categorical features, but in the end, I decided to just turn them into binary features.

In [28]:
```
# Turning the 'sqft_basement' and 'yr_renovated' columns into binary features

kc_nums.sqft_basement = kc_nums.sqft_basement.map(lambda x: 1 if x!=0 else x).astype(int)
kc_nums.yr_renovated = kc_nums.yr_renovated.map(lambda x: 1 if x!=0 else x).astype(int)
```

I decided to turn the `yr_built` column into a categorical feature by splitting the values into groups of every 20 years. I also knew that I would be sorting some of the `xticklabels()` of the Categorical Feature Distribution Visualizations using dictionaries, as well as making sure the dummy variables were in the correct order when I encoded the categorical features, so I just created one for the `yr_built` column here. I will address this change again later in my Insights and Conclusions section.

In [29]:
```
# Finding the time range covered by when the properties were built

built_min = kc_nums.yr_built.loc[kc_nums.yr_built!=0].min()
built_max = kc_nums.yr_built.max()

print(bold_red +'min'+ every_off +' :', built_min, '\t' + bold_red +'max'+ every_off +' :', built_max)
```

```
min : 1900      max : 2015
```

In [30]:
```
# Splitting the 'yr_built' column into groups of 20 years to see what the distribution might look like
# Would also use to sort ticks in any visualizations

yr_range = np.arange(1900, 2001, 20)

built_dict = {}
for y_i, yr in enumerate(yr_range):
    yr_20 = yr + 20
    yr_str = str(yr) +'_to_'+ str(yr_20) +'_s'
    built_dict[yr_str] = \
    kc_nums.yr_built.map(lambda x: True if yr <= x < yr_20 else np.nan).count()

built_dict
```

Out[30]:
```
{'1900_to_1920_s': 1446,
 '1920_to_1940_s': 1717,
 '1940_to_1960_s': 4198,
 '1960_to_1980_s': 4932,
 '1980_to_2000_s': 4498,
 '2000_to_2020_s': 4743}
```

In [31]:
```
# Actually making the changes to the column

yr_range = np.arange(1900, 2001, 20)

built_dict = {}
for y_i, yr in enumerate(yr_range):
    yr_20 = yr + 20
    yr_str = str(yr) +'_to_'+ str(yr_20) +'_s'
    built_dict[yr_str] = y_i + 1
    kc_nums.yr_built = \
    kc_nums.yr_built.map(lambda x: yr_str if type(x)!=str and yr <= x < yr_20 else x)
```

I then switched the three columns I had restructured from the numerical features dataframe to the categorical features dataframe.

In [32]:
```
# I needed to see where to insert the columns

kc_cats.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21534 entries, 2495 to 15937
Data columns (total 8 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   waterfront             21534 non-null  int64
 1   view                   21534 non-null  object
 2   condition              21534 non-null  object
 3   grade                  21534 non-null  object
 4   city                   21534 non-null  object
 5   waterfront_Missing     21534 non-null  int32
 6   yr_renovated_Missing   21534 non-null  int32
 7   sqft_basement_Missing  21534 non-null  int32
dtypes: int32(3), int64(1), object(4)
memory usage: 1.2+ MB
```

In [33]:
```
# Switching the columns from the numerical features to the categorical features

kc_cats.insert(5, 'basement', kc_nums.sqft_basement)
kc_cats.insert(6, 'yr_built', kc_nums.yr_built)
kc_cats.insert(7, 'renovated', kc_nums.yr_renovated)

kc_nums.drop(['sqft_basement', 'yr_built', 'yr_renovated'], axis=1, inplace=True)
```

I again eliminated any outliers to improve the performance of the model.

In [34]:
```
# Seeing the distrubution of the the remaining numerical features, showing they all basically had outliers

kc_nums.describe()
```

Out[34]:

	sqft_living	sqft_lot	sqft_above	sqft_living15	sqft_lot15	floors	bedrooms	bathrooms
count	21534.000000	2.153400e+04	21534.000000	21534.000000	21534.000000	21534.000000	21534.000000	21534.000000
mean	2079.827854	1.509060e+04	1788.557537	1986.299944	12751.079502	1.494126	3.373038	2.115712
std	917.446520	4.138021e+04	827.745641	685.121001	27255.483308	0.539806	0.926410	0.768602
min	370.000000	5.200000e+02	370.000000	399.000000	651.000000	1.000000	1.000000	0.500000
25%	1430.000000	5.040000e+03	1190.000000	1490.000000	5100.000000	1.000000	3.000000	1.750000
50%	1910.000000	7.617000e+03	1560.000000	1840.000000	7620.000000	1.500000	3.000000	2.250000
75%	2550.000000	1.068775e+04	2210.000000	2360.000000	10083.000000	2.000000	4.000000	2.500000
max	13540.000000	1.651359e+06	9410.000000	6210.000000	871200.000000	3.500000	33.000000	8.000000

In [35]:
```
# Removing the outliers for all the numerical features

for n_col in kc_nums.columns:
    kc_nums = \
    kc_nums.loc[(kc_nums[n_col] >= kc_nums[n_col].quantile(.001)) & (kc_nums[n_col] <= kc_nums[n_col].quantile(
```

In [36]:
```
# Checking the results, which made things at least somewhat better

kc_nums.describe()
```

Out[36]:

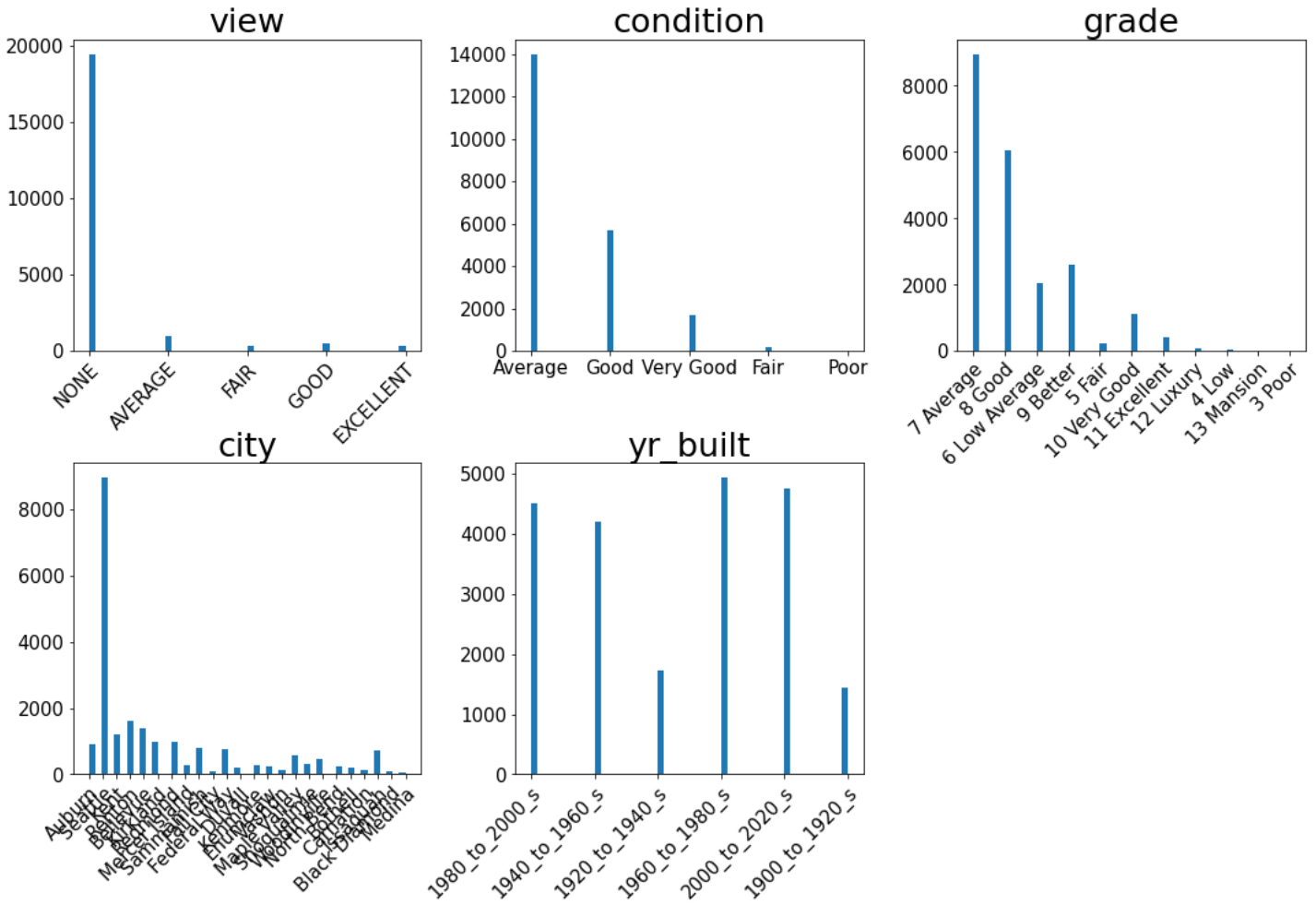
	sqft_living	sqft_lot	sqft_above	sqft_living15	sqft_lot15	floors	bedrooms	bathrooms
count	21279.000000	21279.000000	21279.000000	21279.000000	21279.000000	21279.000000	21279.000000	21279.000000
mean	2067.101744	13951.586400	1777.229334	1982.012595	12090.615113	1.490883	3.370929	2.106831
std	869.249044	30476.339079	789.855994	668.130370	22379.965811	0.537670	0.884038	0.744401
min	570.000000	713.000000	570.000000	750.000000	914.000000	1.000000	1.000000	0.750000
25%	1430.000000	5041.000000	1200.000000	1490.000000	5100.000000	1.000000	3.000000	1.750000
50%	1910.000000	7603.000000	1560.000000	1840.000000	7620.000000	1.500000	3.000000	2.250000
75%	2540.000000	10584.000000	2200.000000	2360.000000	10035.000000	2.000000	4.000000	2.500000
max	7100.000000	493534.000000	5530.000000	4920.000000	283140.000000	3.000000	8.000000	5.000000

Categorical Features Distribution Visualizations

After the numerical features were taken care of, I checked the distribution of the categorical variables. These would have to be encoded in an appropriate manner to be used in my model.

In [37]:

```
# Creating both distribution visualizations for the non-binary features,
# and a dataframe of the binary feature distrubitions

cats = len(kc_cats.columns)
n_rows = int(cats / 2)

fig, axs = plt.subplots(nrows=n_rows, ncols=3, figsize=(18, n_rows * 6),
                        gridspec_kw={'wspace': .27, 'hspace': .36})
axs = axs.flatten()

only_2_vals_df = pd.DataFrame()
empty_i = 0
for c_i, cat in enumerate(kc_cats.columns):

    ax_i = c_i if empty_i==0 else c_i - empty_i

    if kc_cats[cat].nunique() > 2:
        axs[ax_i].hist(kc_cats[cat], bins=50)
        axs[ax_i].set_title(cat, size=27)
        axs[ax_i].tick_params('both', labelsize=15)

        if cat in ['grade', 'view', 'city', 'yr_renovated', 'yr_built', 'sqft_basement']:
            plt.setp(axs[ax_i].get_xticklabels(), rotation=45, rotation_mode='anchor', ha='right', va='top')
    else:
        empty_i += 1
        only_2_vals_df = pd.concat([only_2_vals_df, kc_cats[cat].value_counts()], axis=1)

fig.set_facecolor('w')
[ax.set_visible(False) for ax in axs if not ax.has_data()]

fig.savefig('visuals/categorical_features_distributions.png' , bbox_inches='tight')

plt.show()

display(only_2_vals_df)
```



waterfront	basement	renovated	waterfront_Missing	yr_renovated_Missing	sqft_basement_Missing

	waterfront	basement	renovated	waterfront_Missing	yr_renovated_Missing	sqft_basement_Missing
0	21389	12798	20791	19164	17704	21082
1	145	8736	743	2370	3830	452

I found out later that empty spaces in the values of a column caused problems during my analysis. Also, if you look at the visualizations above, you can see that the `xticklabels` in the visualizations for the distributions of `grade`, `condition`, `view`, and `yr_built` seem to be out of order.

I therefore figured this would be an appropriate place to eliminate the spaces in the values of the `grade`, `condition`, and `city` columns, as well as creating the sorting dictionaries I previously mentioned for the `grade`, `condition`, and `view` columns so that I could sort their `xticklabels`, as well as their dummy columns.

In [38]:
```
# Making the changes necessary for the `sklearn` functions to work properly

kc_cats.grade = kc_cats.grade.map(lambda x: x.replace(' ', '_'))
kc_cats.condition = kc_cats.condition.map(lambda x: x.replace(' ', '_'))
kc_cats.city = kc_cats.city.map(lambda x: x.replace(' ', '_'))
```

In [39]:
```
# Creating dictionaries to use later for sorting and other purposeses

grade_dict = {'3_Poor':1, '4_Low':2, '5_Fair':3, '6_Low_Average':4, '7_Average':5, '8_Good':6,
              '9_Better':7, '10_Very_Good':8, '11_Excellent':9, '12_Luxury':10, '13_Mansion':11}

cond_dict = {'Poor':1, 'Fair':2, 'Average':3, 'Good':4, 'Very_Good':5}

view_dict = {'NONE':1, 'FAIR':2, 'AVERAGE':3, 'GOOD':4, 'EXCELLENT':5}

all_dicts = {'grade': grade_dict, 'condition': cond_dict, 'view': view_dict, 'yr_built': built_dict}
```

By analyzing the `value_counts()` for each categorical column, I found that there were some categories with less than 10 entries associated with them. I thought of these as the outliers of the categorical features, and I dropped them accordingly.

In [40]:
```
# Gerneating details on the non-binary columns and their categories,
# while also creating a dictionary of low frequency categories

low_freq_cats = {}

for c_i, cat in enumerate(kc_cats.columns):
    n_cats = kc_cats[cat].nunique()
    cat_type = kc_cats[cat].dtype

    if cat_type!=object and n_cats>2:
        num_cats = kc_cats[cat].value_counts().sort_index()
        low_cats = num_cats.loc[num_cats < 10].index
        for l_cat in low_cats:
            if cat not in low_freq_cats.keys(): low_freq_cats[cat] = [l_cat]
            else: low_freq_cats[cat].append(l_cat)

        print(bold_red + str(c_i) +' - '+ cat + every_off +':\tnum_vals = '+ str(n_cats), cat_type)
        print(list(num_cats.index), '\n')

    if cat_type==object:
        if cat!='city':
            cat_order = list(all_dicts[cat].keys())
            obj_cats = \
            kc_cats[cat].value_counts().sort_index(key=lambda cat_col: cat_col.map(lambda x: cat_order.index(x)
        else: obj_cats = kc_cats[cat].value_counts()

        low_cats = obj_cats.loc[obj_cats < 10].index
        if not low_cats.empty:
            for l_cat in low_cats:
                if cat not in low_freq_cats.keys(): low_freq_cats[cat] = [l_cat]
                else: low_freq_cats[cat].append(l_cat)

        print(bold_red + str(c_i) +' - '+ cat + every_off +':\tnum_vals = '+ str(n_cats) +'\tdtype =', cat_type
        print(list(obj_cats.index))
        print()
```

```
1 - view:       num_vals = 5    dtype = object
['NONE', 'FAIR', 'AVERAGE', 'GOOD', 'EXCELLENT']

2 - condition:  num_vals = 5    dtype = object
['Poor', 'Fair', 'Average', 'Good', 'Very_Good']

3 - grade:      num_vals = 11   dtype = object
['3_Poor', '4_Low', '5_Fair', '6_Low_Average', '7_Average', '8_Good', '9_Better', '10_Very_Good', '11_Excellent',
 '12_Luxury', '13_Mansion']

4 - city:       num_vals = 24   dtype = object
['Seattle', 'Renton', 'Bellevue', 'Kent', 'Kirkland', 'Redmond', 'Auburn', 'Sammamish', 'Federal_Way', 'Issaquah',
'Maple_Valley', 'Woodinville', 'Snoqualmie', 'Kenmore', 'Mercer_Island', 'Enumclaw', 'North_Bend', 'Bothell', 'Duv
all', 'Carnation', 'Vashon', 'Black_Diamond', 'Fall_City', 'Medina']

6 - yr_built:   num_vals = 6    dtype = object
['1900_to_1920_s', '1920_to_1940_s', '1940_to_1960_s', '1960_to_1980_s', '1980_to_2000_s', '2000_to_2020_s']
```

In [41]:
```
# Checking which categories were marked as low-frequency

low_freq_cats
```

Out[41]:
```
{'grade': ['3_Poor']}
```

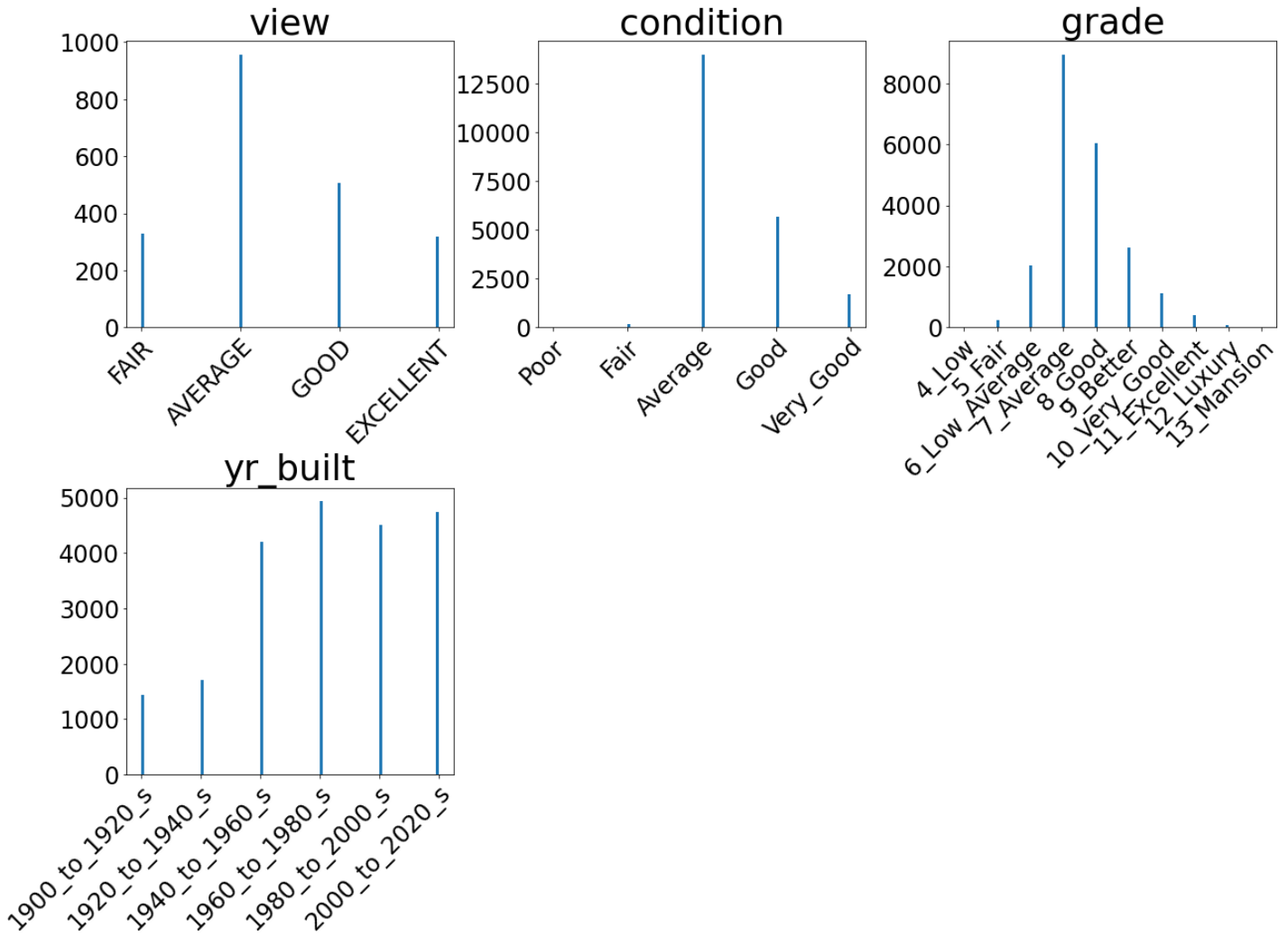
In [42]:
```
# Eliminating the low-frequency categories

for cat, low_cats in low_freq_cats.items():
    for l_cat in low_cats:
        low_index = list(kc_cats.loc[kc_cats[cat] == l_cat].index)
        if low_index: kc_cats.drop(low_index, inplace=True)
```

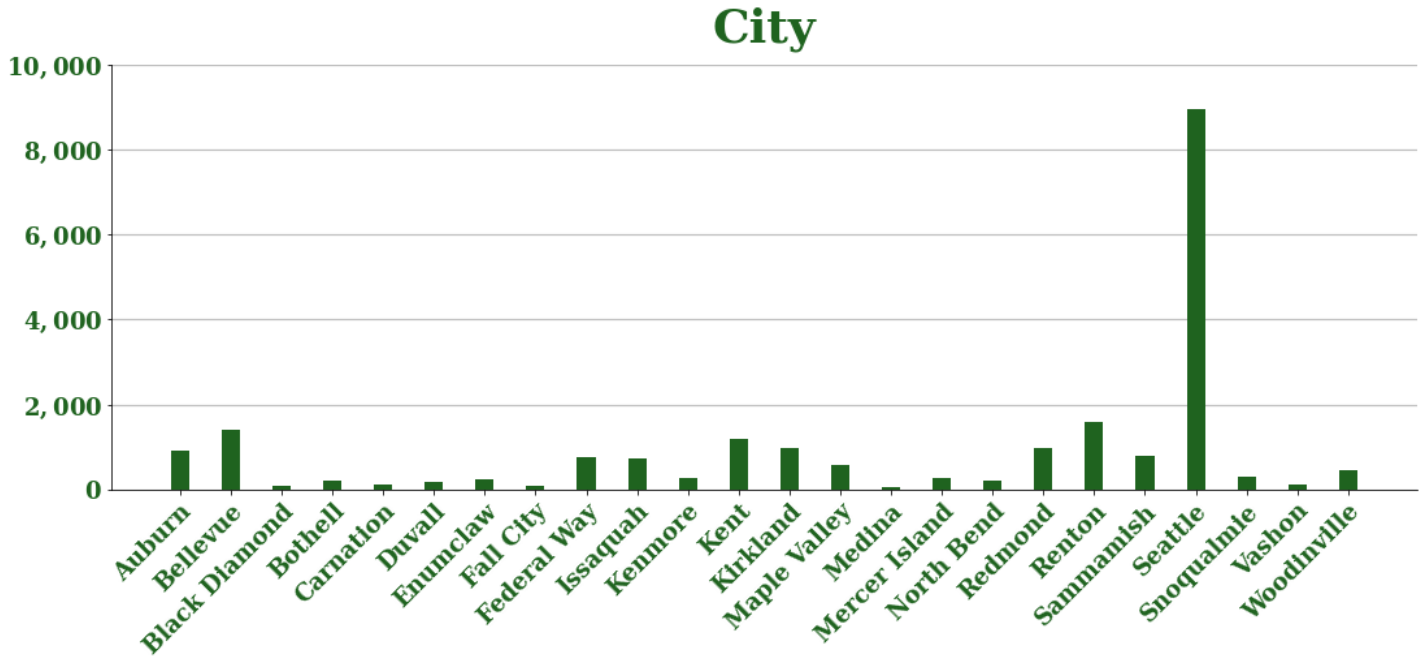
As I mentioned earlier, I found out later that empty spaces in the values of a column would cause problems in my analysis. This was also the case with `.`'s. Once they were in `str` format, I could then use the `.replace()` function to replace the `.`'s with `_`'s.

In [43]:

```
# Creating updated and more organized distribution visualizations for the categorical features,
# as well as a separate visualziation for the `City` feature,
# as I intended it to be used in my presentation and feature prominently on GitHub

cats = len(kc_cats.columns)
n_rows = int(cats / 2)

fig, axs = plt.subplots(nrows=n_rows, ncols=3, figsize=(18, n_rows * 6), )
fig2, city_ax = plt.subplots(figsize=(18, 6))
axs = axs.flatten()

only_2_vals_df = pd.DataFrame()
empty_i = 0
city_i = 0
for c_i, cat in enumerate(kc_cats.columns):

    ax_i = c_i if empty_i==0 else c_i - empty_i if city_i==0 else c_i - empty_i - city_i

    if kc_cats[cat].nunique() > 2:
        plot_col = kc_cats[cat].copy()

        if cat != 'city':
            cat_dict = all_dicts[cat].copy()

            if cat == 'view':
                plot_col = plot_col.where(plot_col != 'NONE').dropna()
                del cat_dict['NONE']

            col_order = sorted(cat_dict.keys(), key=lambda k: cat_dict[k])
            plot_col = \
            plot_col.sort_values(key=lambda c_col: c_col.map(lambda x: col_order.index(x)))
            axs[ax_i].hist(plot_col, bins=100)

            axs[ax_i].tick_params('both', labelsize=15)
            plt.setp(axs[ax_i].get_xticklabels(), rotation=45, rotation_mode='anchor', ha='right', va='top')

            axs[ax_i].set_title(cat, size=36)

        else:

            # For this visualization the details were obviously more important

            city_i += 1
            c_step = 2000

            plot_col = plot_col.sort_values().map(lambda x: ' '.join(x.split('_')) if '_' in x else x)

            plot_col = plot_col.value_counts().sort_index()

            city_ax.bar(x=plot_col.index, height=plot_col.values, width=.36, color=get_lighter_color((.12, .39,
                        zorder=9)

            city_ax.grid(True, axis='y', lw=1.2, alpha=.81)
            [city_ax.spines[side].set_visible(False) for side in ['top', 'right']]
            city_ax.tick_params('both', labelsize=18, labelcolor=(.12, .39, .12))
            c_ticks = np.arange(0, plot_col.max() + c_step, c_step)
            def c_y_ticks(y, pos):
                return '$\mathbf{' + '{:,}'.format(y) + '}$'
            city_ax.set_yticks(c_ticks)
            city_ax.yaxis.set_major_formatter(c_y_ticks)
            plt.setp(city_ax.get_yticklabels(), weight='bold')
            plt.setp(city_ax.get_xticklabels(), rotation=45, rotation_mode='anchor', ha='right', va='top', fami
                     weight='bold')

            city_ax.set_title(cat.title(), size=36, pad=18, weight='bold', family='serif', color=(.12, .39, .12

        if cat=='yr_built': axs[ax_i].tick_params('both', labelsize=21)

    else:
        empty_i += 1
        only_2_vals_df = pd.concat([only_2_vals_df, kc_cats[cat].value_counts()], axis=1)

fig.tight_layout(h_pad=1.5, w_pad=3)
fig.set_facecolor('w')
fig2.set_facecolor('w')
```

```
76
77 [ax.set_visible(False) for ax in axs if not ax.has_data()]
78 [ax.tick_params('both', labelsize=24) for ax in [fig.axes[0]] + fig.axes[1:5] + fig.axes[6:7]]
79
80 fig.savefig('visuals/categorical_features_distributions_2.png' , bbox_inches='tight')
81 fig2.savefig('visuals/city_distribution.png', bbox_inches='tight')
82
83 plt.show()
84
85 display(only_2_vals_df)
```





	waterfront	basement	renovated	waterfront_Missing	yr_renovated_Missing	sqft_basement_Missing
0	21388	12797	20790	19163	17703	21081
1	145	8736	743	2370	3830	452

Initial Correlation Examination

The basic point of building a regression model is to find what impact an increase in one predictor, or ***independent*** variable, has on the target, or ***dependent*** variable, holding everything else constant. Eliminating predictors that are already highly correlated with each other attempts to achieve that goal as closely as possible. The predictors that are eliminated in this section could not be considered independent and would lead to unreliable statistical measurements.

In [44]:
```
# Creating a test df to explore correlations

kc_corr_test = pd.concat([kc_target, kc_nums, kc_cats], join='inner', axis=1)

kc_corr_test.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 21252 entries, 2495 to 15937
Data columns (total 20 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   price                  21252 non-null  float64
 1   sqft_living            21252 non-null  int64  
 2   sqft_lot               21252 non-null  int64  
 3   sqft_above             21252 non-null  int64  
 4   sqft_living15          21252 non-null  int64  
 5   sqft_lot15             21252 non-null  int64  
 6   floors                 21252 non-null  float64
 7   bedrooms               21252 non-null  int64  
 8   bathrooms              21252 non-null  float64
 9   waterfront             21252 non-null  int64  
 10  view                   21252 non-null  object 
 11  condition              21252 non-null  object 
 12  grade                  21252 non-null  object 
 13  city                   21252 non-null  object 
 14  basement               21252 non-null  int32  
 15  yr_built               21252 non-null  object 
 16  renovated              21252 non-null  int32  
 17  waterfront_Missing     21252 non-null  int32  
 18  yr_renovated_Missing   21252 non-null  int32  
 19  sqft_basement_Missing  21252 non-null  int32  
dtypes: float64(3), int32(5), int64(7), object(5)
memory usage: 3.0+ MB
```

In [45]:
```
# Replacing all categorical values with numeric values, as if I had encoded them

restruc_cols = list(kc_corr_test.columns[10:14]) + list(kc_corr_test.columns[15:16])

for col in restruc_cols:
    if col!='city':
        col_dict = all_dicts[col]
        kc_corr_test[col] = kc_corr_test[col].map(lambda x: col_dict[x])
    else:
        col_dict = {}
        for c_i, city in enumerate(kc_corr_test[col].unique()): col_dict[city] = c_i
        kc_corr_test[col] = kc_corr_test[col].map(lambda x: col_dict[x])
```

In [46]:
```
# I created a simple output to find the feature most correlated with the target,
# as well as a dataframe with any features that were highly correlated with each other

corr_df = kc_corr_test.corr()

# Varaibles' Correlations with the Target
#########################################################################################################
price_corr = corr_df[['price']].copy()

price_corr.sort_values('price', ascending=False, inplace=True)

most_correlated_feature = price_corr.index[1]
most_correlated_value = price_corr.iloc[1][0]

print('\033[31m\033[4m\033[1mFeature Most Correlated with Price\033[0m:\t'+ most_correlated_feature)
print("\033[31m\033[4m\033[1mMost Correlated Feature's Value\033[0m:\t\t", round(most_correlated_value, 4))

# Varaibles' Correlation with Each Other
#########################################################################################################
corr_df = corr_df.abs().stack().reset_index().sort_values(0, ascending=False)
corr_df['pairs'] = list(zip(corr_df.level_0, corr_df.level_1))

corr_df.set_index(['pairs'], inplace=True)
corr_df.drop(['level_0', 'level_1'], axis=1, inplace=True)
corr_df.columns = ['correlation']

corr_df = corr_df[(corr_df.correlation > .8) & (corr_df.correlation < 1)]

display(corr_df)
```

Feature Most Correlated with Price: sqft_living
Most Correlated Feature's Value: 0.681

pairs	correlation
(sqft_living, sqft_above)	0.865301
(sqft_above, sqft_living)	0.865301

Removing Highly Correlated Predictors

Based on my initial correlation examination, I decided to drop the `sqft_above` column from the `kc_nums` dataframe.

In [47]:
```
# Dropping `sqft_above` column from the numerical dataframe as it was too correlated with the more important `s

kc_nums.drop('sqft_above', axis=1, inplace=True)
```

Dummy Variable Creation

I used the `OneHotEncoder()` on the appropriate columns to create a properly encoded version of `kc_cats`. I also created a dictionary with the original categorical columns as the keys and the dummy columns that were created from each of them as the values for each key. Finally, I rearranged the columns of `kc_cats` in a way that I found more logical.

In [48]:
```
# Getting the order of columns to regroup them into appropriate lists

orig_cat_cols = list(kc_cats.columns)
[print(c_i, col) for c_i, col in enumerate(orig_cat_cols)];
```

```
0 waterfront
1 view
2 condition
3 grade
4 city
5 basement
6 yr_built
7 renovated
8 waterfront_Missing
9 yr_renovated_Missing
10 sqft_basement_Missing
```

In [49]:
```
# Grouping the columns

missing_cols = orig_cat_cols[-3:]
orig_cat_cols = orig_cat_cols[:-3]

missing_cols
```

Out[49]:
```
['waterfront_Missing', 'yr_renovated_Missing', 'sqft_basement_Missing']
```

In [50]:
```
# Checking the order of the columns in the already prepared lists

num_cols = list(kc_nums.columns)

print(num_cols, '\n')
print(orig_cat_cols)
```

```
['sqft_living', 'sqft_lot', 'sqft_living15', 'sqft_lot15', 'floors', 'bedrooms', 'bathrooms']

['waterfront', 'view', 'condition', 'grade', 'city', 'basement', 'yr_built', 'renovated']
```

In [51]:
```
# Reorgnazing the columns into a way I found more logical and making sure I didn't miss any

new_cat_order = ['basement', 'grade', 'view', 'waterfront', 'condition', 'renovated', 'yr_built', 'city']

assert len(new_cat_order)==len(orig_cat_cols)
```

In [52]:
```
# Encdoding each of the categorical features that still required encoding and creating dictionaries with the
# original columns as the keys and their encoded columns as the values

kc_coded_cats = pd.DataFrame()
cat_dummy_dict = {}
for n_cat in num_cols:
    cat_dummy_dict[n_cat] = n_cat

for c_i, cat in enumerate(new_cat_order):
    if kc_cats[cat].nunique() <= 2:
        cat_df = kc_cats[cat]
        cat_dummy_dict[cat] = cat
    else:
        cat_col = kc_cats[[cat]].applymap(lambda x: cat +'_'+ x)

        if cat!='city':
            cat_order = [cat +'_'+ sub_cat for sub_cat in list(all_dicts[cat].keys())]

            ohe_encoder = OneHotEncoder(categories=[cat_order], drop='first', sparse=False)
            ohe_encoder.fit(cat_col)
            coded_cat = ohe_encoder.transform(cat_col)

            cat_df = pd.DataFrame(coded_cat, columns=list(ohe_encoder.categories_[0])[1:], index=kc_cats.index)

            cat_dummy_dict[cat] = list(ohe_encoder.categories_[0])[1:]

        if cat=='city':
            ohe_encoder = OneHotEncoder(categories='auto', drop='first', sparse=False)
            ohe_encoder.fit(cat_col)
            coded_cat = ohe_encoder.transform(cat_col)

        cat_df = pd.DataFrame(coded_cat, columns=list(ohe_encoder.categories_[0])[1:], index=kc_cats.index)

        cat_dummy_dict[cat] = list(ohe_encoder.categories_[0])[1:]

    kc_coded_cats = pd.concat([kc_coded_cats, cat_df], axis=1)
```

In [53]:
```
# Finalizing my `cat_dummy_dict` with the encoded missing cols

cat_dummy_dict['_Missing'] = missing_cols
```

In [54]:
```
# Uncomment the line below to see the `cat_dummy_dict`

# [print(cat, '\n', dum_cols, '\n') for cat, dum_cols in cat_dummy_dict.items()];
```

Preprocessed DataFrames

The last visualization that I created shows that there were significantly more entries within the city of Seattle compared to the rest of the cities. To explore the difference between the sales of residential properties inside Seattle vs. outside Seattle, which predictors were important for each, as well as the difference between the coefficients of the important predictors shared by both, I created a separate preprocessed dataframe for each. Of course, I also created a preprocessed dataframe for all of King County. Beyond the intrinsic value of building a model for the entire county, it also had the added benefit of serving as a useful comparison to the results of the separated dataframes.

In [55]:
```
# Splitting the data into properties inside of Seattle and properties Outside of Seattle
# and checking how many were in each

seattle_entries = kc_coded_cats.loc[kc_coded_cats.city_Seattle == 1].shape[0]
out_seattle_entries = kc_coded_cats.loc[kc_coded_cats.city_Seattle == 0].shape[0]

print(bold_red +'Number of Entries Inside Seattle'+ every_off +':\t'+ str(seattle_entries))
print(bold_red +'Number of Entries Outside Seattle'+ every_off +':\t'+ str(out_seattle_entries))
```

```
Number of Entries Inside Seattle:       8946
Number of Entries Outside Seattle:      12587
```

All King County Preprocessed DataFrame

In the following cells, I combined the target variable, the numerical features, and the categorical features into the preprocessed dataframe for all of King County. I then removed the dummy columns created from the `city` column. I did this because I created entirely separate models based on the `city` feature and keeping them would only add additional noise at this point. I then split the preprocessed dataframes into their `x` and `y` components. I performed one final check to make sure there were no categorical variables present with low frequencies, and that the `X` and `y` components of the dataframe had the same `x-component` in their `shape` 's, i.e. they were they same length.

In [56]:
```
# Combining the target, the numerical df, and the categorical df

kc_preproc = pd.concat([kc_target, kc_nums, kc_coded_cats], join='inner', axis=1)
```

In [57]:
```
# Removing the `city_` columns

kc_preproc = kc_preproc.loc[:, list(kc_preproc.columns.map(lambda x: 'city' not in x))]
```

In [58]:
```
# Making sure they were removed

not any(kc_preproc.columns.map(lambda x: 'city' in x))
```

Out[58]: True

In [59]:
```
# Splitting the df into its `X` and `y` components

kc_X = kc_preproc.drop('price', axis=1)
kc_y = kc_preproc['price']
```

In [60]:
```
# Creating dataframes of the sums of the binary encoded columns

kc_X_sums = kc_X[[col for col in kc_preproc.columns if col in kc_coded_cats.columns]].sum()
kc_X_sums = pd.DataFrame(kc_X_sums.values, index=kc_X_sums.index)
```

In [61]:
```
# Using that dataframe to filter out low frequency columns and then dropping those columns from the model's fea

kc_less_10_entries = kc_X_sums.loc[kc_X_sums[0] < 10].sort_values(0)

kc_X.drop(list(kc_less_10_entries.index), axis=1, inplace=True)
```

In [62]:
```
# Making sure the X and y components were the same shape

assert kc_X.shape[0]==kc_y.shape[0]
```

Seattle Preprocessed DataFrame

For the Seattle Preprocessed DataFrame, I performed the same steps as I described in the All King County Preprocessed DataFrame, but first I separated out only the Seattle properties from `kc_coded_cats` .

In [63]:
```
# Creating the inside Seattle version of the categorical dataframe

kc_coded_cats_seattle = kc_coded_cats.loc[kc_coded_cats.city_Seattle == 1].copy()
```

In [64]:
```
# Removing the `city_` columns from the inside Seattle version of the categorical dataframe

kc_coded_cats_seattle = \
kc_coded_cats_seattle.loc[:, list(kc_coded_cats_seattle.columns.map(lambda x: 'city' not in x))]
```

In [65]:
```
# Making sure they were removed

not any(kc_coded_cats_seattle.columns.map(lambda x: 'city' in x))
```

Out[65]: True

In [66]:
```
# Combining the target, the numerical df, and the Seattle categorical df

kc_preproc_seattle = pd.concat([kc_target, kc_nums, kc_coded_cats_seattle], join='inner', axis=1)
```

In [67]:
```
# Splitting the df into its `X` and `y` components

kc_X_seattle = kc_preproc_seattle.drop('price', axis=1)
kc_y_seattle = kc_preproc_seattle['price']
```

In [68]:
```
# Creating dataframes of the sums of the binary encoded columns

seattle_X_sums = kc_X_seattle[kc_coded_cats_seattle.columns].sum()
seattle_X_sums = pd.DataFrame(seattle_X_sums.values, index=seattle_X_sums.index)
```

In [69]:
```
# Using that dataframe to filter out low frequency columns and then dropping those columns from the model's fea

seattle_less_10_entries = seattle_X_sums.loc[seattle_X_sums[0] < 10].sort_values(0)

kc_X_seattle.drop(list(seattle_less_10_entries.index), axis=1, inplace=True)
```

In [70]:
```
# Making sure the X and y components were the same shape

assert kc_X_seattle.shape[0]==kc_y_seattle.shape[0]
```

Outside Seattle Preprocessed DataFrame

I separated the properties that were outside of Seattle from `kc_coded_cats`, and then performed the exact same steps as in the previous two sections to create the Outside Seattle Preprocessed DataFrame.

In [71]:
```
# Creating the outside Seattle version of the categorical dataframe

kc_coded_cats_out_seattle = kc_coded_cats.loc[kc_coded_cats.city_Seattle == 0].copy()
```

In [72]:
```
# Removing the `city_` columns from the outside Seattle version of the categorical dataframe

kc_coded_cats_out_seattle = \
kc_coded_cats_out_seattle.loc[:, list(kc_coded_cats_out_seattle.columns.map(lambda x: 'city' not in x))]
```

In [73]:
```
# Making sure they were removed

not any(kc_coded_cats_seattle.columns.map(lambda x: 'city' in x))
```

Out[73]: True

In [74]:
```
# Combining the target, the numerical df, and the outside Seattle categorical df

kc_preproc_out_seattle = \
pd.concat([kc_target, kc_nums, kc_coded_cats_out_seattle], join='inner', axis=1)
```

In [75]:
```
# Splitting the df into its `X` and `y` components

kc_X_out_seattle = kc_preproc_out_seattle.drop('price', axis=1)
kc_y_out_seattle = kc_preproc_out_seattle['price']
```

In [76]:
```
# Creating dataframes of the sums of the binary encoded columns

out_seattle_X_sums = kc_X_out_seattle[kc_coded_cats_out_seattle.columns].sum()
out_seattle_X_sums = pd.DataFrame(out_seattle_X_sums.values, index=out_seattle_X_sums.index)
```

In [77]:
```
# Using that dataframe to filter out low frequency columns and then dropping those columns from the model's fea

out_seattle_less_10_entries = out_seattle_X_sums.loc[out_seattle_X_sums[0] < 10].sort_values(0)

kc_X_out_seattle.drop(list(out_seattle_less_10_entries.index), axis=1, inplace=True)
```

In [78]:
```
# Making sure the X and y components were the same shape

assert kc_X_out_seattle.shape[0]==kc_y_out_seattle.shape[0]
```

Model Iterations

Base Models

With the preprocessed dataframes ready, I could begin building models. The base models for each preprocessed dataframe are just the relationship between the most correlated feature, `sqft_living`, and the target variable, `price`, which will be used to judge the performance of future models.

In [79]:
```
# All King County Base Model

kc_base_const = sm.add_constant(kc_X[most_correlated_feature])
kc_base_model = sm.OLS(endog=kc_y, exog=kc_base_const).fit()
```

In [80]:
```
# Seattle Base Model

base_const_seattle = sm.add_constant(kc_X_seattle[most_correlated_feature])
seattle_base_model = sm.OLS(endog=kc_y_seattle, exog=base_const_seattle).fit()
```

In [81]:
```
# Outside Seattle Base Model

base_const_out_seattle = sm.add_constant(kc_X_out_seattle[most_correlated_feature])
out_seattle_base_model = sm.OLS(endog=kc_y_out_seattle, exog=base_const_out_seattle).fit()
```

Full Models

After getting a baseline for each preprocessed dataframe, I created models for each with all the predictors available. I then began eliminating them to create a final model with only the best predictors. First, I removed predictors based on their `Variance Inflation Factor (VIF)` scores to eliminate predictors with high levels of multicollinearity missed by my initial correlation examination. I then used the `RFECV` feature selection method to eliminate all but the best predictors. Finally, I eliminated all predictors with `pvalues` less than the standard confidence level of `0.05`, to only include the predictors of the highest statistical significance.

Once the predictors for each model were finalized, I could investigate the `Linearity, Normality, and Homoscedasticity` of the predicted values generated by each model. For each of the models, a log transformation of the target variable was necessary for the model to meet the assumptions required when building multiple linear regression models. I then had a final equation that I could analyze to produce my Insights and Conclusions. I transformed the coefficients in the equation to make understanding them easier.

I also created simple dataframes with the coefficients, the r^2, and the adjusted r^2 scores from each model to make any comparisons between the models easier.

After I had created the comparison dataframes for each of the models, I could analyze the performance of the models.

All King County Full Model

In [82]:
```
# Created an appropriate copy of the X component to make sure I could still access the original without changes
# if necessary

kc_X_full = kc_X.copy()
```

In [83]:
```
# All King County First Full Model

kc_full_const = sm.add_constant(kc_X_full)
kc_full_model = sm.OLS(endog=kc_y, exog=kc_full_const).fit()
```

kc_VIF

In [84]:

```
# VIF Elimination
# Removing one dummy column from each group created from the original columns (if it was a dummy column)
# Removed the columns with a high VIF first
# Removed the columns with a mid VIF second
# Printed the details on which columns were dropped at each iteration

vif_compl = 0
while not vif_compl:
    high_cols_dict = {}
    mid_cols_dict = {}

    full_const = sm.add_constant(kc_X_full)
    kc_full_VIF_model = sm.OLS(endog=kc_y, exog=full_const).fit()

    vif = [variance_inflation_factor(full_const.values, i) for i in range(full_const.shape[1])]
    vif_df = pd.DataFrame(vif, index=full_const.columns, columns=["Variance Inflation Factor"])

    high_vif_cols = list(vif_df.loc[vif_df['Variance Inflation Factor'] > 10].index)
    high_vif_vals = list(vif_df.loc[vif_df['Variance Inflation Factor'] > 10].values)
    if 'const' in high_vif_cols:
        h_i = high_vif_cols.index('const')
        high_vif_cols.remove('const')
        high_vif_vals.pop(h_i)

    mid_vif_cols = \
    list(vif_df.loc[(vif_df['Variance Inflation Factor'] > 5) & (vif_df['Variance Inflation Factor'] < 10)].ind

    mid_vif_vals = \
    list(vif_df.loc[(vif_df['Variance Inflation Factor'] > 5) & (vif_df['Variance Inflation Factor'] < 10)].val

    if 'const' in mid_vif_cols:
        m_i = mid_vif_cols.index('const')
        mid_vif_cols.remove('const')
        mid_vif_vals.pop(m_i)

    for g_i, col_grp in enumerate([high_vif_cols, mid_vif_cols]):
        if g_i==0: grp_dict, grp_vals = high_cols_dict, high_vif_vals
        if g_i==1: grp_dict, grp_vals = mid_cols_dict, mid_vif_vals
        if col_grp:
            for c_i, col in enumerate(col_grp):
                if col in grp_dict.keys(): grp_dict[col] += grp_vals[c_i]
                else: grp_dict[col] = grp_vals[c_i]

    high_cols_df = pd.DataFrame(high_cols_dict.values(), high_cols_dict.keys(), columns=['VIF'])
    mid_VIF_cols_df = pd.DataFrame(mid_cols_dict.values(), mid_cols_dict.keys(), columns=['VIF'])

    kc_VIF = high_cols_df.sort_values('VIF', ascending=False)
    mid_VIF_cols_df = mid_VIF_cols_df.sort_values('VIF', ascending=False)

    VIF_drop_cols = []
    for o_col in list(cat_dummy_dict.keys())[:-1]:
        if any(kc_VIF.index.map(lambda x: o_col in x)):
            first_pred_col = kc_VIF.loc[kc_VIF.index.map(lambda x: o_col in x)].index[0]
            VIF_drop_cols.append(first_pred_col)

    if VIF_drop_cols:
        kc_X_full.drop(VIF_drop_cols, axis=1, inplace=True)
        print(bold_red +'High VIF Dropped Columns'+ every_off, '\n', VIF_drop_cols, '\n')

    if not VIF_drop_cols and not mid_VIF_cols_df.empty:
        mid_VIF_drop_cols = []
        for o_col in list(cat_dummy_dict.keys())[:-1]:
            if any(mid_VIF_cols_df.index.map(lambda x: o_col in x)):
                first_pred_col = mid_VIF_cols_df.loc[mid_VIF_cols_df.index.map(lambda x: o_col in x)].index[0]
                mid_VIF_drop_cols.append(first_pred_col)

        if mid_VIF_drop_cols:
            kc_X_full.drop(mid_VIF_drop_cols, axis=1, inplace=True)
            print(bold_red +'Mid VIF Dropped Columns'+ every_off, '\n', mid_VIF_drop_cols, '\n')

    if not VIF_drop_cols and mid_VIF_cols_df.empty:  vif_compl = 1

print(bold_red +'VIF Elimination Finished'+ every_off)
```

```
High VIF Dropped Columns
 ['Condition']
```

```
['grade_7_Average', 'condition_Average']

Mid VIF Dropped Columns
['sqft_living']

VIF Elimination Finished
```

kc_RFECV

In [85]:

```python
# RFECV Elimination
# This used machine learning to basically let the computer choose the best features, dropping the worst columns

best_model_found = 0
RFECV_round = 0
while not best_model_found:

    full_const = sm.add_constant(kc_X_full)
    kc_full_RFECV_model = sm.OLS(endog=kc_y, exog=full_const).fit()

    splitter = ShuffleSplit(n_splits=6, test_size=0.2, random_state=36)
    kc_X_for_RFECV = StandardScaler().fit_transform(kc_X_full)
    model_for_RFECV = LinearRegression()
    selector = RFECV(model_for_RFECV, cv=splitter)
    selector.fit(kc_X_for_RFECV, kc_y)

    RFECV_bad_cols = [col for c_i, col in enumerate(kc_X_full.columns) if not selector.support_[c_i]]

    if RFECV_bad_cols:
        REFCV_drop_cols = []
        for o_col in list(cat_dummy_dict.keys()):
            o_df = pd.DataFrame()
            for r_col in RFECV_bad_cols:
                if r_col.startswith(o_col) or r_col.endswith(o_col):
                    r_prob, r_coef = kc_full_RFECV_model.pvalues[r_col], kc_full_RFECV_model.params[r_col]
                    r_df = pd.DataFrame([r_prob, r_coef], columns=[r_col], index=['p_value', 'coeff']).transpos
                    o_df = pd.concat([o_df, r_df])
            if not o_df.empty:
                o_max_col = o_df.loc[o_df.p_value == o_df.p_value.max()].index[0]
                REFCV_drop_cols.append(o_max_col)
        RFECV_round += 1

        kc_X_full.drop(REFCV_drop_cols, axis=1, inplace=True)
        print(bold_red +'RFECV Dropped Columns'+ every_off, '\n', REFCV_drop_cols, '\n')

    else: best_model_found = 1

print(bold_red +'RFECV Finished'+ every_off)
```

```
RFECV Dropped Columns
['bedrooms', 'condition_Fair']

RFECV Finished
```

kc_pvals

In [86]:
```
# High `P-values` Elimination
# Removing the column with the highest p-value at each iteration and printing the results

high_pvals = 1

while high_pvals:

    kc_pval_const = sm.add_constant(kc_X_full)
    kc_pval_model = sm.OLS(endog=kc_y, exog=kc_pval_const).fit()

    high_pval_list = list(kc_pval_model.pvalues.loc[kc_pval_model.pvalues >= .05].sort_values(ascending=False).

    if high_pval_list:
        high_pval_drop_col = high_pval_list[0]
        kc_X_full.drop(high_pval_drop_col, axis=1, inplace=True)
        print(str(high_pvals) +' - '+ bold_red +'High p-value Column Dropped'+ every_off +':\n', high_pval_drop
        high_pvals += 1

    else: high_pvals = 0

print(bold_red +'High p_value Elimination Finsihed'+ every_off)
```

High p_value Elimination Finsihed

kc_fin_model

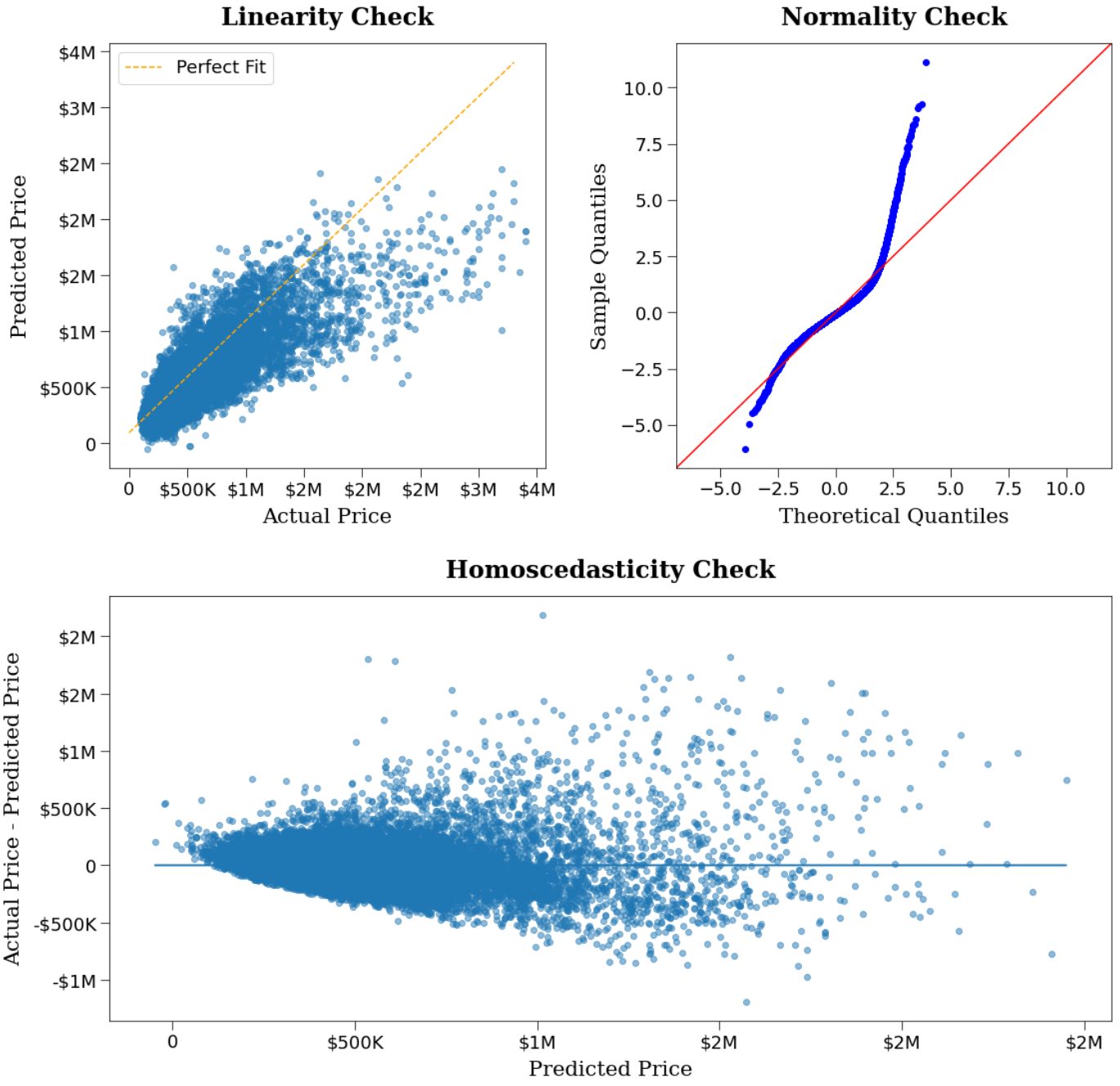
In [87]:
```
# Simply renaming the last version of the model that was created as the final model

kc_fin_const = kc_pval_const
kc_fin_model = kc_pval_model
```

Investigating the Linearity, Normality, and Homoscedasticity of ***kc_fin_model***

In [88]:
```
# Changing the `rcParams` to make my visualizaitons look better

plt.rc('figure', figsize=(18, 18), facecolor='w')
plt.rc('axes', titlesize=24, titlepad=18, titleweight='bold', labelsize=21, labelpad=9)
plt.rc('xtick', labelsize=18)
plt.rc('xtick.major', size=9)
plt.rc('ytick', labelsize=18)
plt.rc('ytick.major', size=9)
plt.rc('legend', fontsize=18)
```

In [89]:

```
# Creating the visualizations necessary to explore the Linearity, Normality, and Homoscedasticity of the model

fig = plt.figure()
gs = fig.add_gridspec(2, 2, wspace=.3, hspace=.3)

lin_ax = fig.add_subplot(gs[0, 0])
norm_ax = fig.add_subplot(gs[0, 1])
homo_ax = fig.add_subplot(gs[1, :])

kc_preds = kc_fin_model.predict(kc_fin_const)
perfect_line = np.arange(kc_y.min(), kc_y.max())
kc_resids = (kc_y - kc_preds)

lin_ax.plot(perfect_line, linestyle="--", color="orange", label="Perfect Fit")
lin_ax.scatter(kc_y, kc_preds, alpha=0.5)
lin_ax.set_xlabel("Actual Price", family='serif')
lin_ax.set_ylabel("Predicted Price", family='serif')
lin_ax.set_title('Linearity Check', family='serif')
lin_ax.legend()

sm.graphics.qqplot(kc_resids, dist=stats.norm, line='45', fit=True, ax=norm_ax)
norm_ax.set_title('Normality Check', family='serif')

homo_ax.scatter(kc_preds, kc_resids, alpha=0.5)
homo_ax.plot(kc_preds, [0 for i in range(len(kc_X_full))])
homo_ax.set_xlabel("Predicted Price", family='serif')
homo_ax.set_ylabel("Actual Price - Predicted Price", family='serif')
homo_ax.set_title('Homoscedasticity Check', family='serif')

for a_i, ax in enumerate([lin_ax, norm_ax, homo_ax]):
    if a_i!=1:
        ax.xaxis.set_major_formatter(viz_currency_formatter)
        ax.yaxis.set_major_formatter(viz_currency_formatter)
    else:
        ax.set_xlabel(ax.get_xlabel(), family='serif')
        ax.set_ylabel(ax.get_ylabel(), family='serif')

fig.savefig('visuals/kc_model_check.png' , bbox_inches='tight')

plt.show()
```



kc_log_model

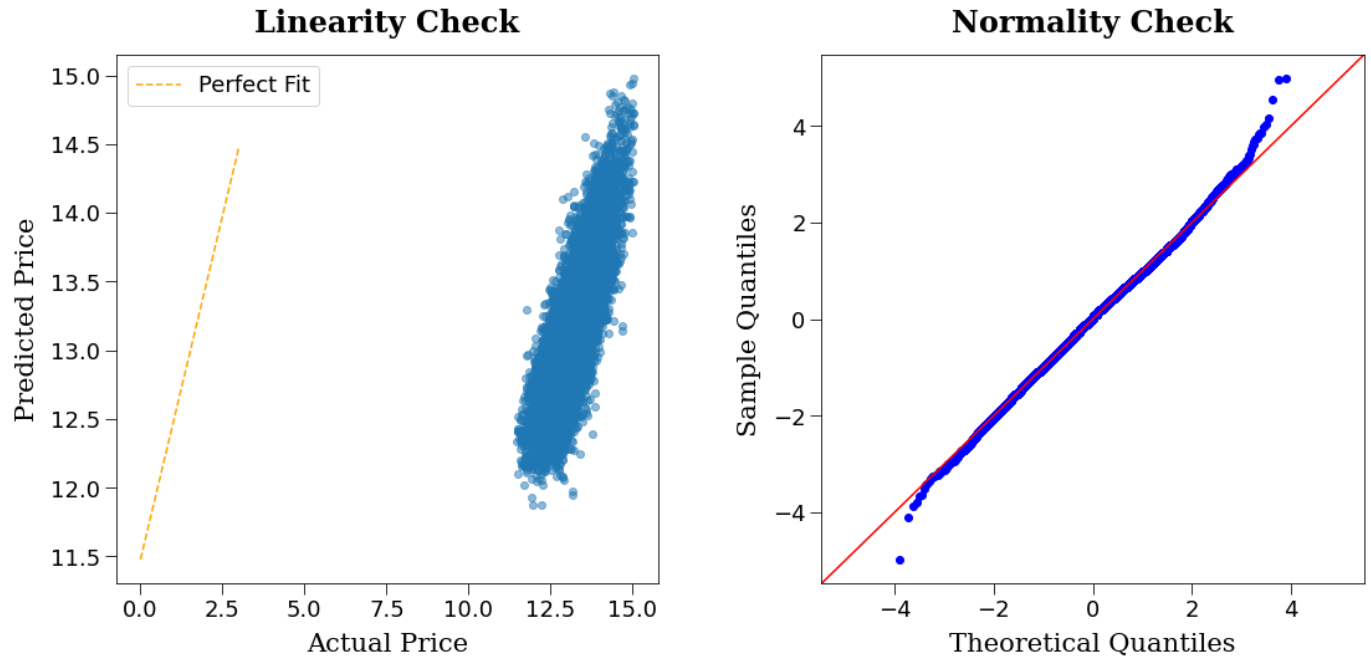
In [90]:
```
# Only log transforming the target (y component)

kc_log_y = np.log(kc_y)
```

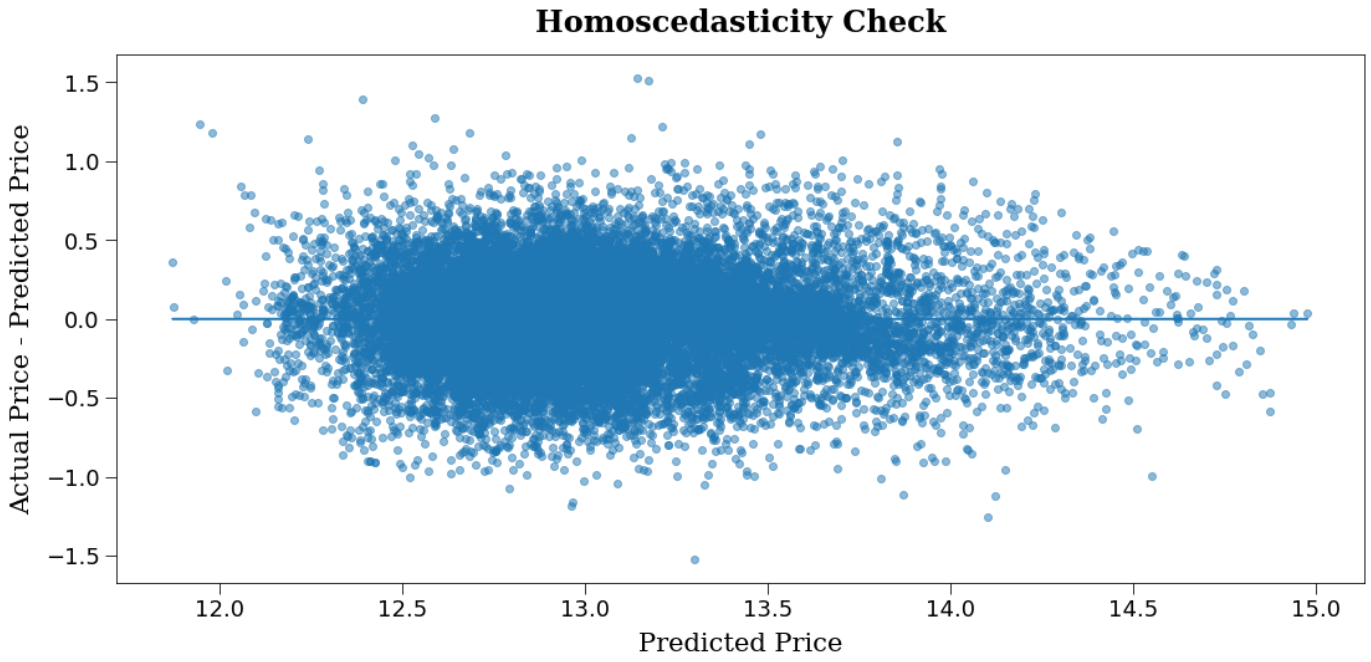
In [91]:
```
# Creating the log model

kc_log_const = sm.add_constant(kc_X_full)
kc_log_model = sm.OLS(endog=kc_log_y, exog=kc_log_const).fit()
```

Investigating the Linearity, Normality and Homoscedasticity of ***kc_log_model***

In [92]:

```
# Creating the visualizations necessary to explore the Linearity, Normality, and Homoscedasticity of the model

fig = plt.figure()
gs = fig.add_gridspec(2, 2, wspace=.3, hspace=.3)

lin_ax = fig.add_subplot(gs[0, 0])
norm_ax = fig.add_subplot(gs[0, 1])
homo_ax = fig.add_subplot(gs[1, :])

kc_preds = kc_log_model.predict(kc_log_const)
perfect_line = np.arange(kc_log_y.min(), kc_log_y.max())
kc_resids = (kc_log_y - kc_preds)

lin_ax.plot(perfect_line, linestyle="--", color="orange", label="Perfect Fit")
lin_ax.scatter(kc_log_y, kc_preds, alpha=0.5)
lin_ax.set_xlabel("Actual Price", family='serif')
lin_ax.set_ylabel("Predicted Price", family='serif')
lin_ax.set_title('Linearity Check', family='serif')
lin_ax.legend()

sm.graphics.qqplot(kc_resids, dist=stats.norm, line='45', fit=True, ax=norm_ax)
norm_ax.set_title('Normality Check', family='serif')
norm_ax.set_xlabel(norm_ax.get_xlabel(), family='serif')
norm_ax.set_ylabel(norm_ax.get_ylabel(), family='serif')

homo_ax.scatter(kc_preds, kc_resids, alpha=0.5)
homo_ax.plot(kc_preds, [0 for i in range(len(kc_X_full))])
homo_ax.set_xlabel("Predicted Price", family='serif')
homo_ax.set_ylabel("Actual Price - Predicted Price", family='serif')
homo_ax.set_title('Homoscedasticity Check', family='serif')

fig.savefig('visuals/kc_model_check_log.png' , bbox_inches='tight')

plt.show()
```





Comparing kc Models

In [93]:

```
# Creating lists of the models and their names in str format to zip() thru to create the comparison dataframe

kc_model_list = [kc_base_model, kc_full_model, kc_full_VIF_model, kc_full_RFECV_model, kc_fin_model, kc_log_mod
kc_model_names = ['base', 'full', 'full_VIF', 'full_RFECV', 'fin', 'log']
```

In [94]:

```python
# Creating the comparison dataframe

kc_r_vals = {}
kc_r_adj_vals = {}
kc_compare_params_df = pd.DataFrame()
for ols_i, (ols_model, model_name) in enumerate(zip(kc_model_list, kc_model_names)):
    model_params = ols_model.params
    model_params.name = model_name
    kc_r_vals[model_name] = ols_model.rsquared
    kc_r_adj_vals[model_name] = ols_model.rsquared_adj

    kc_compare_params_df = pd.concat([kc_compare_params_df, model_params], axis=1)

for r_i, r_dict in enumerate([kc_r_vals, kc_r_adj_vals]):
    r_name = 'r_score' if r_i==0 else 'r_adj_score'
    r_ser = pd.DataFrame(r_dict.values(), index=r_dict.keys(), columns=[r_name]).transpose()

    kc_compare_params_df = pd.concat([kc_compare_params_df, r_ser])
```

In [95]:

```
# Rounding columns for readability and displaying the comparision dataframe

for col in kc_compare_params_df.columns[:-1]:
    kc_compare_params_df[col] = kc_compare_params_df[col].round(2)

display(kc_compare_params_df)
```

	base	full	full_VIF	full_RFECV	fin	log
const	-7280.35	1851260.75	181853.63	184086.86	184086.86	1.247509e+01
sqft_living	261.48	101.41	NaN	NaN	NaN	NaN
sqft_lot	NaN	0.00	0.16	0.16	0.16	5.308354e-07
sqft_living15	NaN	43.95	84.43	85.00	85.00	1.677819e-04
sqft_lot15	NaN	-0.41	-0.34	-0.34	-0.34	-3.228181e-07
floors	NaN	32242.39	38391.34	38596.57	38596.57	7.979150e-02
bedrooms	NaN	-20563.99	1876.15	NaN	NaN	NaN
bathrooms	NaN	43649.55	77424.92	78637.52	78637.52	1.102396e-01
basement	NaN	28271.09	46838.03	46921.40	46921.40	1.023296e-01
grade_4_Low	NaN	-1845862.71	-202612.19	-205199.69	-205199.69	-7.329981e-01
grade_5_Fair	NaN	-1806822.56	-156429.53	-158160.98	-158160.98	-4.922216e-01
grade_6_Low_Average	NaN	-1751618.09	-91864.21	-92576.01	-92576.01	-2.792341e-01
grade_7_Average	NaN	-1666481.28	NaN	NaN	NaN	NaN
grade_8_Good	NaN	-1581692.34	96157.10	96053.56	96053.56	2.231718e-01
grade_9_Better	NaN	-1440300.37	267735.68	267709.22	267709.22	4.668456e-01
grade_10_Very_Good	NaN	-1248361.03	491630.80	491261.71	491261.71	6.667515e-01
grade_11_Excellent	NaN	-979479.94	796781.64	796239.55	796239.55	8.406650e-01
grade_12_Luxury	NaN	-732027.42	1073845.32	1072828.69	1072828.69	9.651193e-01
view_FAIR	NaN	108804.10	112440.75	112164.38	112164.38	1.442479e-01
view_AVERAGE	NaN	52721.43	57588.41	57406.37	57406.37	7.810800e-02
view_GOOD	NaN	109097.91	112817.17	112393.29	112393.29	9.707906e-02
view_EXCELLENT	NaN	277423.43	283667.93	283371.66	283371.66	2.578018e-01
waterfront	NaN	391638.36	403652.08	402814.20	402814.20	3.096941e-01
condition_Fair	NaN	18184.77	-26034.46	NaN	NaN	NaN
condition_Average	NaN	41651.75	NaN	NaN	NaN	NaN
condition_Good	NaN	70256.35	31452.89	32015.22	32015.22	5.174069e-02
condition_Very_Good	NaN	109975.58	71782.15	72384.31	72384.31	1.135529e-01
renovated	NaN	43209.95	50488.77	50507.01	50507.01	4.283010e-02
yr_built_1920_to_1940_s	NaN	-21387.52	-22770.59	-22905.56	-22905.56	-5.179860e-02
yr_built_1940_to_1960_s	NaN	-90311.26	-91173.55	-91062.34	-91062.34	-1.928097e-01
yr_built_1960_to_1980_s	NaN	-189001.30	-197283.50	-196955.07	-196955.07	-3.973160e-01
yr_built_1980_to_2000_s	NaN	-267220.17	-281701.49	-282060.51	-282060.51	-4.949009e-01
yr_built_2000_to_2020_s	NaN	-244943.85	-256734.33	-257185.94	-257185.94	-4.560393e-01
r_score	0.46	0.67	0.65	0.65	0.65	6.452154e-01
r_adj_score	0.46	0.67	0.65	0.65	0.65	6.447640e-01

Final kc Equation

In [96]:
```
# Taking the pieces of the final equation and making them more readable

kc_log_int = int(np.exp(kc_compare_params_df.log.dropna()[0]))
kc_log_percs = kc_compare_params_df.log.dropna()[1:-2] * 100
```

In [97]:
```
# Continuing the process of making the pieces of the equation more readable

kc_fin_percs = list(kc_log_percs.values.astype(int))
kc_fin_preds = list(kc_log_percs.index)
```

In [98]:
```
# Creating strings of the pieces of the equation with their column names

kc_perc_eq = ['{}% * {}'.format(perc, pred) for perc, pred in zip(kc_fin_percs, kc_fin_preds)]
```

In [99]:
```
# Putting the pieces together in a readable format

model_eq = 'price = '
model_int_string = str(kc_log_int) +' +\n'
model_var_string = ' +\n'.join(kc_perc_eq)
```

In [100]:
```
# Putting the final equation together and printing it

kc_log_eq = model_eq + model_int_string + model_var_string

print(kc_log_eq)
```

```
price = 261735 +
0% * sqft_lot +
0% * sqft_living15 +
0% * sqft_lot15 +
7% * floors +
11% * bathrooms +
10% * basement +
-73% * grade_4_Low +
-49% * grade_5_Fair +
-27% * grade_6_Low_Average +
22% * grade_8_Good +
46% * grade_9_Better +
66% * grade_10_Very_Good +
84% * grade_11_Excellent +
96% * grade_12_Luxury +
14% * view_FAIR +
7% * view_AVERAGE +
9% * view_GOOD +
25% * view_EXCELLENT +
30% * waterfront +
5% * condition_Good +
11% * condition_Very_Good +
4% * renovated +
-5% * yr_built_1920_to_1940_s +
-19% * yr_built_1940_to_1960_s +
-39% * yr_built_1960_to_1980_s +
-49% * yr_built_1980_to_2000_s +
-45% * yr_built_2000_to_2020_s
```

Seattle Full Model

In [101]:
```
# Inside Seattle First Full Model

seattle_full_const = sm.add_constant(kc_X_seattle)
seattle_full_model = sm.OLS(endog=kc_y_seattle, exog=seattle_full_const).fit()
```

seattle_VIF

In [102]:

```
# VIF Elimination
# Removing one dummy column from each group created from the original columns (if it was a dummy column)
# Removed the columns with a high VIF first
# Removed the columns with a mid VIF second
# Printed the details on which columns were dropped at each iteration

vif_compl = 0
while not vif_compl:
    high_cols_dict = {}
    mid_cols_dict = {}

    full_const = sm.add_constant(kc_X_seattle)
    seattle_full_VIF_model = sm.OLS(endog=kc_y_seattle, exog=full_const).fit()

    vif = [variance_inflation_factor(full_const.values, i) for i in range(full_const.shape[1])]
    vif_df = pd.DataFrame(vif, index=full_const.columns, columns=["Variance Inflation Factor"])

    high_vif_cols = list(vif_df.loc[vif_df['Variance Inflation Factor'] > 10].index)
    high_vif_vals = list(vif_df.loc[vif_df['Variance Inflation Factor'] > 10].values)
    if 'const' in high_vif_cols:
        h_i = high_vif_cols.index('const')
        high_vif_cols.remove('const')
        high_vif_vals.pop(h_i)

    mid_vif_cols = \
    list(vif_df.loc[(vif_df['Variance Inflation Factor'] > 5) & (vif_df['Variance Inflation Factor'] < 10)].ind

    mid_vif_vals = \
    list(vif_df.loc[(vif_df['Variance Inflation Factor'] > 5) & (vif_df['Variance Inflation Factor'] < 10)].val

    if 'const' in mid_vif_cols:
        m_i = mid_vif_cols.index('const')
        mid_vif_cols.remove('const')
        mid_vif_vals.pop(m_i)

    for g_i, col_grp in enumerate([high_vif_cols, mid_vif_cols]):
        if g_i==0: grp_dict, grp_vals = high_cols_dict, high_vif_vals
        if g_i==1: grp_dict, grp_vals = mid_cols_dict, mid_vif_vals
        if col_grp:
            for c_i, col in enumerate(col_grp):
                if col in grp_dict.keys(): grp_dict[col] += grp_vals[c_i]
                else: grp_dict[col] = grp_vals[c_i]

    high_cols_df = pd.DataFrame(high_cols_dict.values(), high_cols_dict.keys(), columns=['VIF'])
    mid_VIF_cols_df = pd.DataFrame(mid_cols_dict.values(), mid_cols_dict.keys(), columns=['VIF'])

    seattle_VIF = high_cols_df.sort_values('VIF', ascending=False)
    mid_VIF_cols_df = mid_VIF_cols_df.sort_values('VIF', ascending=False)

    VIF_drop_cols = []
    for o_col in list(cat_dummy_dict.keys())[:-1]:
        if any(seattle_VIF.index.map(lambda x: o_col in x)):
            first_pred_col = seattle_VIF.loc[seattle_VIF.index.map(lambda x: o_col in x)].index[0]
            VIF_drop_cols.append(first_pred_col)

    if VIF_drop_cols:
        kc_X_seattle.drop(VIF_drop_cols, axis=1, inplace=True)
        print(bold_red +'High VIF Dropped Columns'+ every_off, '\n', VIF_drop_cols, '\n')

    if not VIF_drop_cols and not mid_VIF_cols_df.empty:
        mid_VIF_drop_cols = []
        for o_col in list(cat_dummy_dict.keys())[:-1]:
            if any(mid_VIF_cols_df.index.map(lambda x: o_col in x)):
                first_pred_col = mid_VIF_cols_df.loc[mid_VIF_cols_df.index.map(lambda x: o_col in x)].index[0]
                mid_VIF_drop_cols.append(first_pred_col)

        if mid_VIF_drop_cols:
            kc_X_seattle.drop(mid_VIF_drop_cols, axis=1, inplace=True)
            print(bold_red +'Mid VIF Dropped Columns'+ every_off, '\n', mid_VIF_drop_cols, '\n')

    if not VIF_drop_cols and mid_VIF_cols_df.empty:  vif_compl = 1

print(bold_red +'VIF Elimination Finished'+ every_off)
```

```
High VIF Dropped Columns
```

```
['grade_7_Average', 'condition_Average']
```

VIF Elimination Finished

seattle_RFECV

In [103]:

```
# RFECV Elimination
# This used machine learning to basically let the computer choose the best features, dropping the worst columns

best_model_found = 0
RFECV_round = 0
while not best_model_found:

    full_const = sm.add_constant(kc_X_seattle)
    seattle_full_RFECV_model = sm.OLS(endog=kc_y_seattle, exog=full_const).fit()

    splitter = ShuffleSplit(n_splits=6, test_size=0.2, random_state=36)
    seattle_X_for_RFECV = StandardScaler().fit_transform(kc_X_seattle)
    model_for_RFECV = LinearRegression()
    selector = RFECV(model_for_RFECV, cv=splitter)
    selector.fit(seattle_X_for_RFECV, kc_y_seattle)

    RFECV_bad_cols = [col for c_i, col in enumerate(kc_X_seattle.columns) if not selector.support_[c_i]]

    if RFECV_bad_cols:
        REFCV_drop_cols = []
        for o_col in list(cat_dummy_dict.keys()):
            o_df = pd.DataFrame()
            for r_col in RFECV_bad_cols:
                if r_col.startswith(o_col) or r_col.endswith(o_col):
                    r_prob, r_coef = seattle_full_RFECV_model.pvalues[r_col], seattle_full_RFECV_model.params[r
                    r_df = pd.DataFrame([r_prob, r_coef], columns=[r_col], index=['p_value', 'coeff']).transpos
                    o_df = pd.concat([o_df, r_df])
            if not o_df.empty:
                o_max_col = o_df.loc[o_df.p_value == o_df.p_value.max()].index[0]
                REFCV_drop_cols.append(o_max_col)
        RFECV_round += 1

        kc_X_seattle.drop(REFCV_drop_cols, axis=1, inplace=True)
        print(bold_red +'RFECV Dropped Columns'+ every_off, '\n', REFCV_drop_cols, '\n')


    else: best_model_found = 1

print(bold_red +'RFECV Finished'+ every_off)
```

RFECV Dropped Columns

```
['view_AVERAGE', 'condition_Fair', 'renovated', 'yr_built_1920_to_1940_s']
```

RFECV Finished

seattle_pvals

In [104]:

```
# High `P-values` Elimination
# Removing the column with the highest p-value at each iteration and printing the results

high_pvals = 1

while high_pvals:
    seattle_pval_const = sm.add_constant(kc_X_seattle)
    seattle_pval_model = sm.OLS(endog=kc_y_seattle, exog=seattle_pval_const).fit()

    high_pval_list = \
    list(seattle_pval_model.pvalues.loc[seattle_pval_model.pvalues >= .05].sort_values(ascending=False).index)

    if high_pval_list:
        high_pval_drop_col = high_pval_list[0]
        kc_X_seattle.drop(high_pval_drop_col, axis=1, inplace=True)
        print(str(high_pvals) +' - '+ bold_red +'High p-value Column Dropped'+ every_off +':\n', high_pval_drop
        high_pvals += 1

    else: high_pvals = 0

print(bold_red +'High p_value Elimination Finsihed'+ every_off)
```

```
1 - High p-value Column Dropped:
 view_FAIR

2 - High p-value Column Dropped:
 basement

High p_value Elimination Finsihed
```

seattle_fin_model

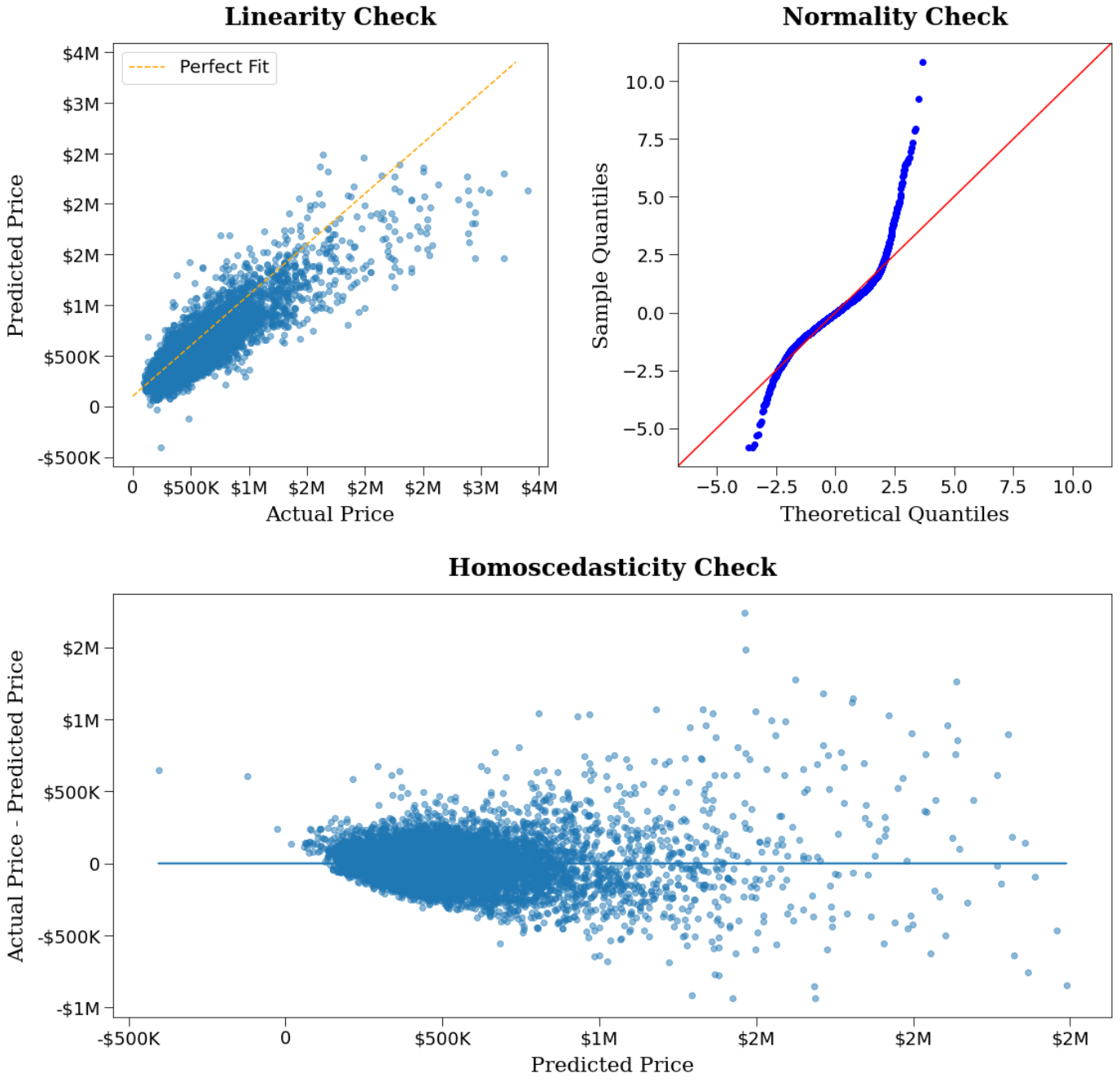
In [105]:

```
# Simply renaming the last version of the model that was created as the final model

seattle_fin_const = seattle_pval_const
seattle_fin_model = seattle_pval_model
```

Investigating the Linearity, Normality and Homoscedasticity of ***seattle_fin_model***

In [106]:

```
# Creating the visualizations necessary to explore the Linearity, Normality, and Homoscedasticity of the model

fig = plt.figure()
gs = fig.add_gridspec(2, 2, wspace=.3, hspace=.3)

lin_ax = fig.add_subplot(gs[0, 0])
norm_ax = fig.add_subplot(gs[0, 1])
homo_ax = fig.add_subplot(gs[1, :])

seattle_preds = seattle_fin_model.predict(seattle_fin_const)
perfect_line = np.arange(kc_y_seattle.min(), kc_y_seattle.max())
seattle_resids = (kc_y_seattle - seattle_preds)

lin_ax.plot(perfect_line, linestyle="--", color="orange", label="Perfect Fit")
lin_ax.scatter(kc_y_seattle, seattle_preds, alpha=0.5)
lin_ax.set_xlabel("Actual Price", family='serif')
lin_ax.set_ylabel("Predicted Price", family='serif')
lin_ax.set_title('Linearity Check', family='serif')
lin_ax.legend()

sm.graphics.qqplot(seattle_resids, dist=stats.norm, line='45', fit=True, ax=norm_ax)
norm_ax.set_title('Normality Check', family='serif')

homo_ax.scatter(seattle_preds, seattle_resids, alpha=0.5)
homo_ax.plot(seattle_preds, [0 for i in range(len(kc_X_seattle))])
homo_ax.set_xlabel("Predicted Price", family='serif')
homo_ax.set_ylabel("Actual Price - Predicted Price", family='serif')
homo_ax.set_title('Homoscedasticity Check', family='serif')

for a_i, ax in enumerate([lin_ax, norm_ax, homo_ax]):
    if a_i!=1:
        ax.xaxis.set_major_formatter(viz_currency_formatter)
        ax.yaxis.set_major_formatter(viz_currency_formatter)
    else:
        ax.set_xlabel(ax.get_xlabel(), family='serif')
        ax.set_ylabel(ax.get_ylabel(), family='serif')

fig.savefig('visuals/seattle_model_check.png' , bbox_inches='tight')

plt.show()
```



seattle_log_model

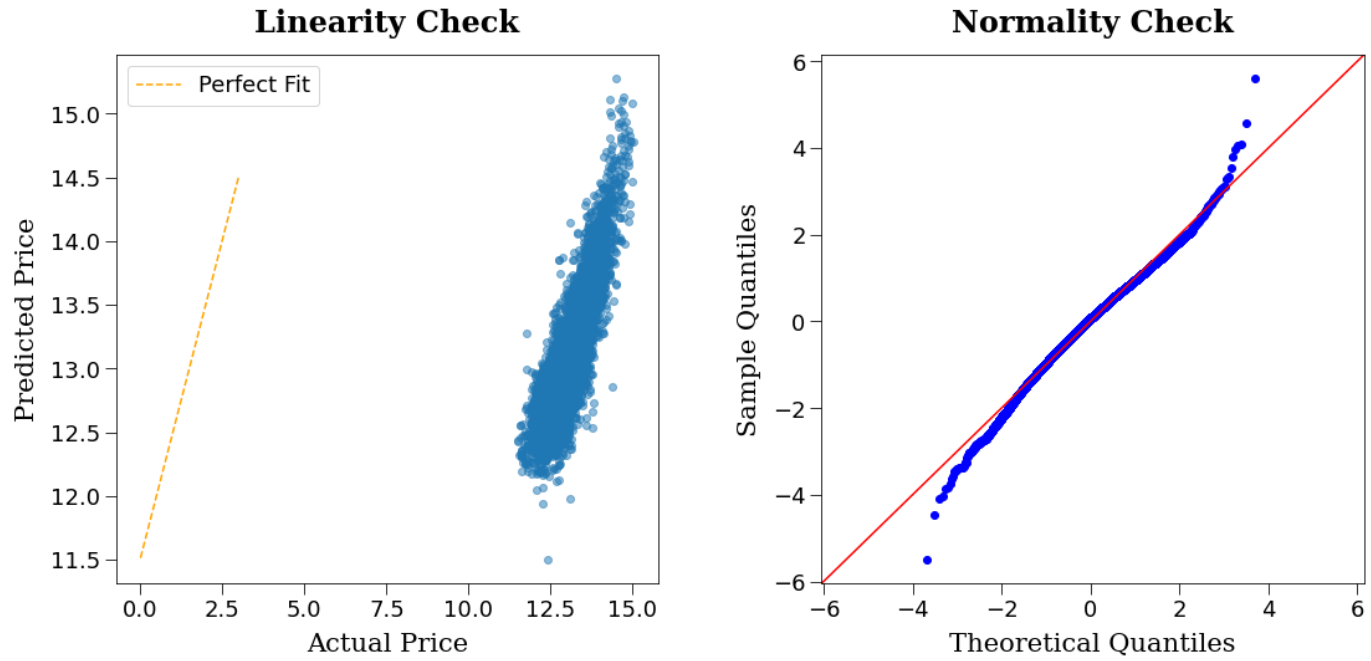
In [107]:
```
# Only log transforming the target (y component)

seattle_log_y = np.log(kc_y_seattle)
```

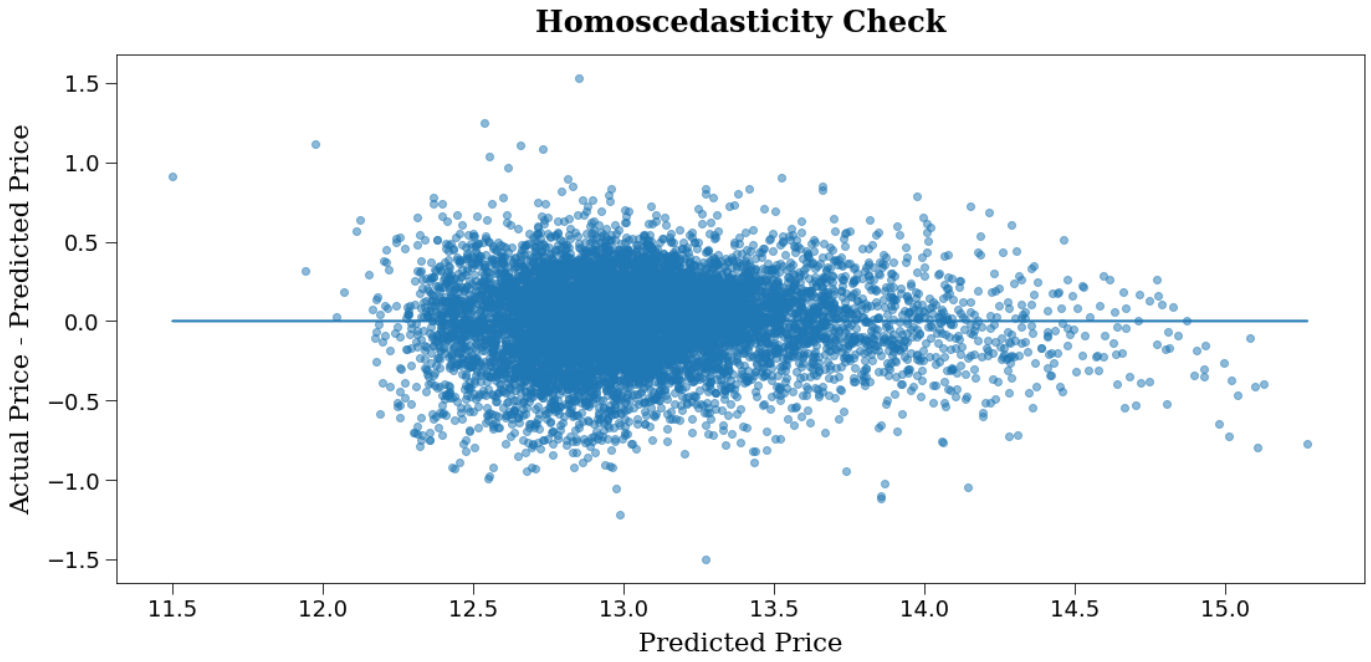
In [108]:
```
# Creating the log model

seattle_log_const = sm.add_constant(kc_X_seattle)
seattle_log_model = sm.OLS(endog=seattle_log_y, exog=seattle_log_const).fit()
```

Investigating the Linearity, Normality and Homoscedasticity of ***seattle_log_model***

In [109]:

```
# Creating the visualizations necessary to explore the Linearity, Normality, and Homoscedasticity of the model

fig = plt.figure()
gs = fig.add_gridspec(2, 2, wspace=.3, hspace=.3)

lin_ax = fig.add_subplot(gs[0, 0])
norm_ax = fig.add_subplot(gs[0, 1])
homo_ax = fig.add_subplot(gs[1, :])

seattle_preds = seattle_log_model.predict(seattle_log_const)
perfect_line = np.arange(seattle_log_y.min(), seattle_log_y.max())
seattle_resids = (seattle_log_y - seattle_preds)

lin_ax.plot(perfect_line, linestyle="--", color="orange", label="Perfect Fit")
lin_ax.scatter(seattle_log_y, seattle_preds, alpha=0.5)
lin_ax.set_xlabel("Actual Price", family='serif')
lin_ax.set_ylabel("Predicted Price", family='serif')
lin_ax.set_title('Linearity Check', family='serif')
lin_ax.legend()

sm.graphics.qqplot(seattle_resids, dist=stats.norm, line='45', fit=True, ax=norm_ax)
norm_ax.set_title('Normality Check', family='serif')
norm_ax.set_xlabel(norm_ax.get_xlabel(), family='serif')
norm_ax.set_ylabel(norm_ax.get_ylabel(), family='serif')

homo_ax.scatter(seattle_preds, seattle_resids, alpha=0.5)
homo_ax.plot(seattle_preds, [0 for i in range(len(kc_X_seattle))])
homo_ax.set_xlabel("Predicted Price", family='serif')
homo_ax.set_ylabel("Actual Price - Predicted Price", family='serif')
homo_ax.set_title('Homoscedasticity Check', family='serif')

fig.savefig('visuals/seattle_model_check_log.png' , bbox_inches='tight')

plt.show()
```





Comparing `seattle` Models

In [110]:

```
# Creating lists of the models and their names in str format to zip() thru to create the comparison dataframe

seattle_model_list = [seattle_base_model, seattle_full_model, seattle_full_VIF_model, seattle_full_RFECV_model,
                      seattle_fin_model, seattle_log_model]

seattle_model_names = ['base', 'full', 'full_VIF', 'full_RFECV', 'fin', 'log']
```

In [111]:

```
# Creating the comparison dataframe

seattle_r_vals = {}
seattle_r_adj_vals = {}
seattle_compare_params_df = pd.DataFrame()
for ols_i, (ols_model, model_name) in enumerate(zip(seattle_model_list, seattle_model_names)):
    model_params = ols_model.params
    model_params.name = model_name
    seattle_r_vals[model_name] = ols_model.rsquared
    seattle_r_adj_vals[model_name] = ols_model.rsquared_adj

    seattle_compare_params_df = pd.concat([seattle_compare_params_df, model_params], axis=1)

for r_i, r_dict in enumerate([seattle_r_vals, seattle_r_adj_vals]):
    r_name = 'r_score' if r_i==0 else 'r_adj_score'
    r_ser = pd.DataFrame(r_dict.values(), index=r_dict.keys(), columns=[r_name]).transpose()

    seattle_compare_params_df = pd.concat([seattle_compare_params_df, r_ser])
```

In [112]:
```
# Rounding columns for readability and displaying the comparision dataframe

for col in seattle_compare_params_df.columns[:-1]:
    seattle_compare_params_df[col] = seattle_compare_params_df[col].round(2)

display(seattle_compare_params_df)
```

	base	full	full_VIF	full_RFECV	fin	log
const	7148.41	491438.27	158847.89	155356.89	150633.28	12.569419
sqft_living	293.17	135.15	136.21	136.39	134.33	0.000201
sqft_lot	NaN	-2.51	-2.51	-2.54	-2.48	-0.000006
sqft_living15	NaN	131.77	131.11	131.46	131.75	0.000183
sqft_lot15	NaN	-10.48	-10.53	-10.54	-10.41	-0.000018
floors	NaN	26516.04	26529.99	26635.76	29577.71	0.025384
bedrooms	NaN	-20287.84	-20608.57	-20596.79	-20490.59	-0.028837
bathrooms	NaN	22132.33	22340.04	22311.97	21428.51	0.045524
basement	NaN	-6765.95	-7093.05	-7230.78	NaN	NaN
grade_5_Fair	NaN	-553126.46	-85545.22	-87652.26	-85594.76	-0.411990
grade_6_Low_Average	NaN	-536149.09	-62415.59	-62528.72	-61946.55	-0.245897
grade_7_Average	NaN	-474582.53	NaN	NaN	NaN	NaN
grade_8_Good	NaN	-393683.57	80506.23	80651.86	80722.49	0.194855
grade_9_Better	NaN	-190465.58	283354.67	283346.02	284470.63	0.438257
grade_10_Very_Good	NaN	63240.34	536847.91	536678.08	538149.36	0.565386
grade_11_Excellent	NaN	451839.39	925217.63	925367.66	927254.11	0.705951
grade_12_Luxury	NaN	786496.10	1261792.74	1262217.00	1261689.34	1.184853
view_FAIR	NaN	13241.93	14928.80	14573.30	NaN	NaN
view_AVERAGE	NaN	2284.04	2245.71	NaN	NaN	NaN
view_GOOD	NaN	40582.96	40385.02	39867.15	38758.08	0.044745
view_EXCELLENT	NaN	151865.47	150499.34	150208.06	148506.59	0.214510
waterfront	NaN	288776.95	287809.10	287742.45	287184.80	0.380997
condition_Fair	NaN	123737.20	-17472.86	NaN	NaN	NaN
condition_Average	NaN	142630.60	NaN	NaN	NaN	NaN
condition_Good	NaN	164661.54	22273.24	22633.66	22578.01	0.070457
condition_Very_Good	NaN	197201.90	54683.97	55144.57	54934.00	0.102440
renovated	NaN	-1184.31	-374.69	NaN	NaN	NaN
yr_built_1920_to_1940_s	NaN	-5859.97	-4599.04	NaN	NaN	NaN
yr_built_1940_to_1960_s	NaN	-70917.57	-69411.88	-66702.13	-66135.54	-0.148344
yr_built_1960_to_1980_s	NaN	-170904.35	-169779.18	-166989.49	-166736.97	-0.321357
yr_built_1980_to_2000_s	NaN	-197923.68	-197007.95	-194238.87	-193683.66	-0.303857
yr_built_2000_to_2020_s	NaN	-202398.48	-201521.89	-198890.26	-200151.41	-0.341686
r_score	0.49	0.75	0.75	0.75	0.75	0.697796
r_adj_score	0.49	0.75	0.75	0.75	0.75	0.697006

Final seattle Model

In [113]:
```
# Taking the pieces of the final equation and making them more readable

seattle_log_int = int(np.exp(seattle_compare_params_df.log.dropna()[0]))
seattle_log_percs = seattle_compare_params_df.log.dropna()[1:-2] * 100
```

In [114]:
```
# Continuing the process of making the pieces of the equation more readable

seattle_fin_percs = list(seattle_log_percs.values.astype(int))
seattle_fin_preds = list(seattle_log_percs.index)
```

In [115]:
```
# Creating strings of the pieces of the equation with their column names

seattle_perc_eq = ['{}% * {}'.format(perc, pred) for perc, pred in zip(seattle_fin_percs, seattle_fin_preds)]
```

In [116]:
```
# Putting the pieces together in a readable format

model_eq = 'price = '
model_int_string = str(seattle_log_int) +' +\n'
model_var_string = ' +\n'.join(seattle_perc_eq)
```

In [117]:
```
# Putting the final equation together and printing it

seattle_log_eq = model_eq + model_int_string + model_var_string

print(seattle_log_eq)
```

```
price = 287626 +
0% * sqft_living +
0% * sqft_lot +
0% * sqft_living15 +
0% * sqft_lot15 +
2% * floors +
-2% * bedrooms +
4% * bathrooms +
-41% * grade_5_Fair +
-24% * grade_6_Low_Average +
19% * grade_8_Good +
43% * grade_9_Better +
56% * grade_10_Very_Good +
70% * grade_11_Excellent +
118% * grade_12_Luxury +
4% * view_GOOD +
21% * view_EXCELLENT +
38% * waterfront +
7% * condition_Good +
10% * condition_Very_Good +
-14% * yr_built_1940_to_1960_s +
-32% * yr_built_1960_to_1980_s +
-30% * yr_built_1980_to_2000_s +
-34% * yr_built_2000_to_2020_s
```

Outside Seattle Full Model

In [118]:
```
# Outside Seattle First Full Model

out_seattle_full_const = sm.add_constant(kc_X_out_seattle)
out_seattle_full_model = sm.OLS(endog=kc_y_out_seattle, exog=out_seattle_full_const).fit()
```

out_seattle_VIF

In [119]:

```
# VIF Elimination
# Removing one dummy column from each group created from the original columns (if it was a dummy column)
# Removed the columns with a high VIF first
# Removed the columns with a mid VIF second
# Printed the details on which columns were dropped at each iteration

vif_compl = 0
while not vif_compl:
    high_cols_dict = {}
    mid_cols_dict = {}

    full_const = sm.add_constant(kc_X_out_seattle)
    out_seattle_full_VIF_model = sm.OLS(endog=kc_y_out_seattle, exog=full_const).fit()

    vif = [variance_inflation_factor(full_const.values, i) for i in range(full_const.shape[1])]
    vif_df = pd.DataFrame(vif, index=full_const.columns, columns=["Variance Inflation Factor"])

    high_vif_cols = list(vif_df.loc[vif_df['Variance Inflation Factor'] > 10].index)
    high_vif_vals = list(vif_df.loc[vif_df['Variance Inflation Factor'] > 10].values)
    if 'const' in high_vif_cols:
        h_i = high_vif_cols.index('const')
        high_vif_cols.remove('const')
        high_vif_vals.pop(h_i)

    mid_vif_cols = \
    list(vif_df.loc[(vif_df['Variance Inflation Factor'] > 5) & (vif_df['Variance Inflation Factor'] < 10)].ind

    mid_vif_vals = \
    list(vif_df.loc[(vif_df['Variance Inflation Factor'] > 5) & (vif_df['Variance Inflation Factor'] < 10)].val

    if 'const' in mid_vif_cols:
        m_i = mid_vif_cols.index('const')
        mid_vif_cols.remove('const')
        mid_vif_vals.pop(m_i)

    for g_i, col_grp in enumerate([high_vif_cols, mid_vif_cols]):
        if g_i==0: grp_dict, grp_vals = high_cols_dict, high_vif_vals
        if g_i==1: grp_dict, grp_vals = mid_cols_dict, mid_vif_vals
        if col_grp:
            for c_i, col in enumerate(col_grp):
                if col in grp_dict.keys(): grp_dict[col] += grp_vals[c_i]
                else: grp_dict[col] = grp_vals[c_i]

    high_cols_df = pd.DataFrame(high_cols_dict.values(), high_cols_dict.keys(), columns=['VIF'])
    mid_VIF_cols_df = pd.DataFrame(mid_cols_dict.values(), mid_cols_dict.keys(), columns=['VIF'])

    out_seattle_VIF = high_cols_df.sort_values('VIF', ascending=False)
    mid_VIF_cols_df = mid_VIF_cols_df.sort_values('VIF', ascending=False)

    VIF_drop_cols = []
    for o_col in list(cat_dummy_dict.keys())[:-1]:
        if any(out_seattle_VIF.index.map(lambda x: o_col in x)):
            first_pred_col = out_seattle_VIF.loc[out_seattle_VIF.index.map(lambda x: o_col in x)].index[0]
            VIF_drop_cols.append(first_pred_col)

    if VIF_drop_cols:
        kc_X_out_seattle.drop(VIF_drop_cols, axis=1, inplace=True)
        print(bold_red +'High VIF Dropped Columns'+ every_off, '\n', VIF_drop_cols, '\n')

    if not VIF_drop_cols and not mid_VIF_cols_df.empty:
        mid_VIF_drop_cols = []
        for o_col in list(cat_dummy_dict.keys())[:-1]:
            if any(mid_VIF_cols_df.index.map(lambda x: o_col in x)):
                first_pred_col = mid_VIF_cols_df.loc[mid_VIF_cols_df.index.map(lambda x: o_col in x)].index[0]
                mid_VIF_drop_cols.append(first_pred_col)

        if mid_VIF_drop_cols:
            kc_X_out_seattle.drop(mid_VIF_drop_cols, axis=1, inplace=True)
            print(bold_red +'Mid VIF Dropped Columns'+ every_off, '\n', mid_VIF_drop_cols, '\n')

    if not VIF_drop_cols and mid_VIF_cols_df.empty:  vif_compl = 1

print(bold_red +'VIF Elimination Finished'+ every_off)
```

```
High VIF Dropped Columns
 ['grade_7_Average', 'condition_Average', 'yr_built_1980_to_2000_s']

Mid VIF Dropped Columns
 ['sqft_living']

VIF Elimination Finished
```

out_seattle_RFECV

In [120]:

```python
# RFECV Elimination
# This used machine learning to basically let the computer choose the best features, dropping the worst columns

best_model_found = 0
RFECV_round = 0
while not best_model_found:

    full_const = sm.add_constant(kc_X_out_seattle)
    out_seattle_full_RFECV_model = sm.OLS(endog=kc_y_out_seattle, exog=full_const).fit()

    splitter = ShuffleSplit(n_splits=6, test_size=0.2, random_state=36)
    out_seattle_X_for_RFECV = StandardScaler().fit_transform(kc_X_out_seattle)
    model_for_RFECV = LinearRegression()
    selector = RFECV(model_for_RFECV, cv=splitter)
    selector.fit(out_seattle_X_for_RFECV, kc_y_out_seattle)

    RFECV_bad_cols = [col for c_i, col in enumerate(kc_X_out_seattle.columns) if not selector.support_[c_i]]

    if RFECV_bad_cols:
        REFCV_drop_cols = []
        for o_col in list(cat_dummy_dict.keys()):
            o_df = pd.DataFrame()
            for r_col in RFECV_bad_cols:
                if r_col.startswith(o_col) or r_col.endswith(o_col):
                    r_prob, r_coef = \
                    out_seattle_full_RFECV_model.pvalues[r_col], out_seattle_full_RFECV_model.params[r_col]
                    r_df = pd.DataFrame([r_prob, r_coef], columns=[r_col], index=['p_value', 'coeff']).transpos
                    o_df = pd.concat([o_df, r_df])
            if not o_df.empty:
                o_max_col = o_df.loc[o_df.p_value == o_df.p_value.max()].index[0]
                REFCV_drop_cols.append(o_max_col)
        RFECV_round += 1

        kc_X_out_seattle.drop(REFCV_drop_cols, axis=1, inplace=True)
        print(bold_red +'RFECV Dropped Columns'+ every_off, '\n', REFCV_drop_cols, '\n')


    else: best_model_found = 1

print(bold_red +'RFECV Finished'+ every_off)
```

```
RFECV Finished
```

out_seattle_pvals

In [121]:

```
# High `P-values` Elimination
# Removing the column with the highest p-value at each iteration and printing the results

high_pvals = 1

while high_pvals:
    out_seattle_pval_const = sm.add_constant(kc_X_out_seattle)
    out_seattle_pval_model = sm.OLS(endog=kc_y_out_seattle, exog=out_seattle_pval_const).fit()

    high_pval_list = \
    list(out_seattle_pval_model.pvalues.loc[out_seattle_pval_model.pvalues >= .05].sort_values(ascending=False)

    if high_pval_list:
        high_pval_drop_col = high_pval_list[0]
        kc_X_out_seattle.drop(high_pval_drop_col, axis=1, inplace=True)
        print(str(high_pvals) +' - '+ bold_red +'High p-value Column Dropped'+ every_off +':\n', high_pval_drop
        high_pvals += 1

    else: high_pvals = 0

print(bold_red +'High p_value Elimination Finsihed'+ every_off)
```

```
1 - High p-value Column Dropped:
 grade_4_Low

2 - High p-value Column Dropped:
 condition_Fair

High p_value Elimination Finsihed
```

out_seattle_fin_model

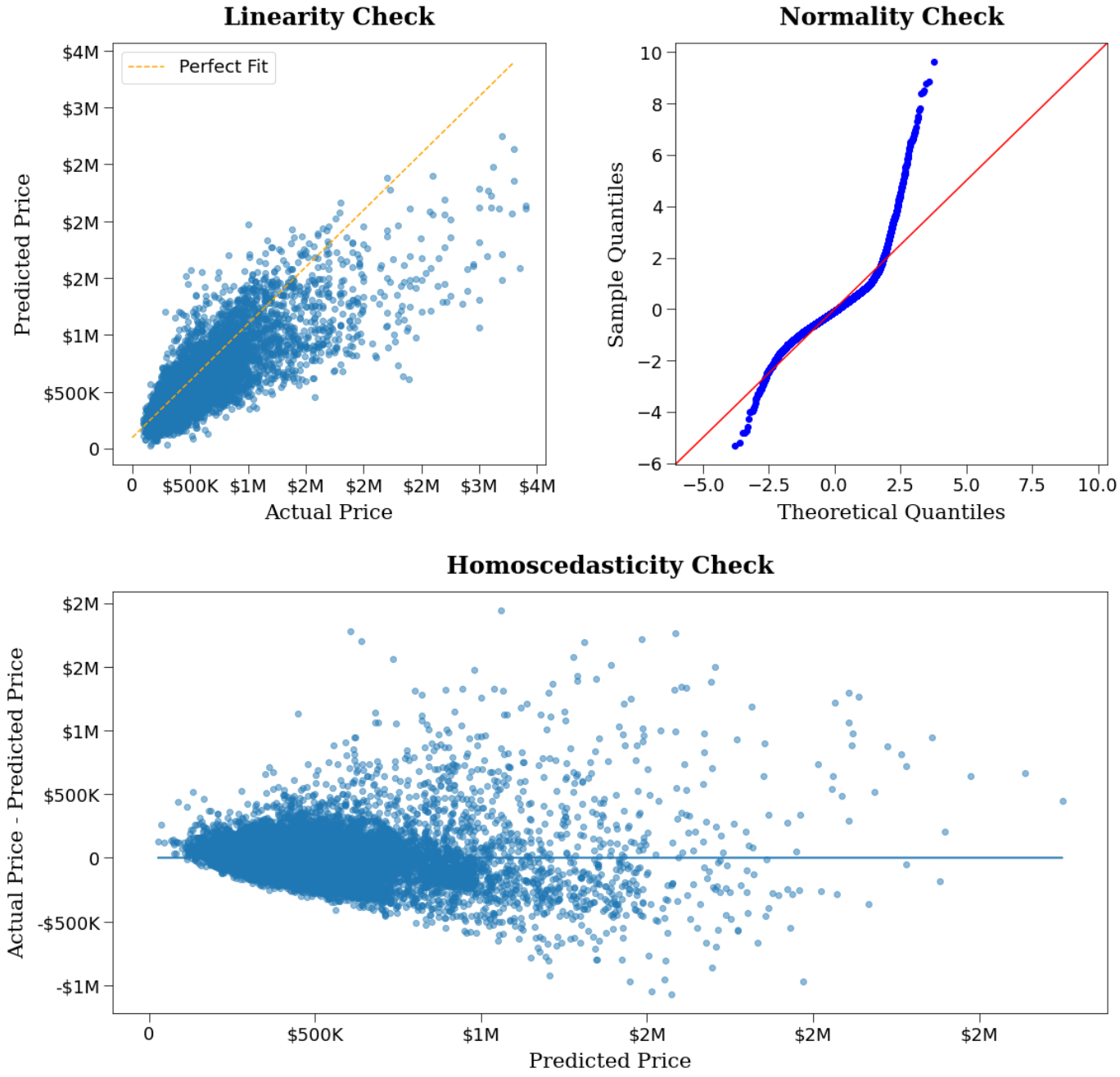
In [122]:

```
# Simply renaming the last version of the model that was created as the final model

out_seattle_fin_const = out_seattle_pval_const
out_seattle_fin_model = out_seattle_pval_model
```

Investigating the Linearity, Normality and Homoscedasticity of ***out_seattle_fin_model***

In [123]:

```
# Creating the visualizations necessary to explore the Linearity, Normality, and Homoscedasticity of the model

fig = plt.figure()
gs = fig.add_gridspec(2, 2, wspace=.3, hspace=.3)

lin_ax = fig.add_subplot(gs[0, 0])
norm_ax = fig.add_subplot(gs[0, 1])
homo_ax = fig.add_subplot(gs[1, :])

out_seattle_preds = out_seattle_fin_model.predict(out_seattle_fin_const)
perfect_line = np.arange(kc_y_out_seattle.min(), kc_y_out_seattle.max())
out_seattle_resids = (kc_y_out_seattle - out_seattle_preds)

lin_ax.plot(perfect_line, linestyle="--", color="orange", label="Perfect Fit")
lin_ax.scatter(kc_y_out_seattle, out_seattle_preds, alpha=0.5)
lin_ax.set_xlabel("Actual Price", family='serif')
lin_ax.set_ylabel("Predicted Price", family='serif')
lin_ax.set_title('Linearity Check', family='serif')
lin_ax.legend()

sm.graphics.qqplot(out_seattle_resids, dist=stats.norm, line='45', fit=True, ax=norm_ax)
norm_ax.set_title('Normality Check', family='serif')

homo_ax.scatter(out_seattle_preds, out_seattle_resids, alpha=0.5)
homo_ax.plot(out_seattle_preds, [0 for i in range(len(kc_X_out_seattle))])
homo_ax.set_xlabel("Predicted Price", family='serif')
homo_ax.set_ylabel("Actual Price - Predicted Price", family='serif')
homo_ax.set_title('Homoscedasticity Check', family='serif')

for a_i, ax in enumerate([lin_ax, norm_ax, homo_ax]):
    if a_i!=1:
        ax.xaxis.set_major_formatter(viz_currency_formatter)
        ax.yaxis.set_major_formatter(viz_currency_formatter)
    else:
        ax.set_xlabel(ax.get_xlabel(), family='serif')
        ax.set_ylabel(ax.get_ylabel(), family='serif')

fig.savefig('visuals/out_seattle_model_check.png' , bbox_inches='tight')

plt.show()
```



out_seattle_log_model

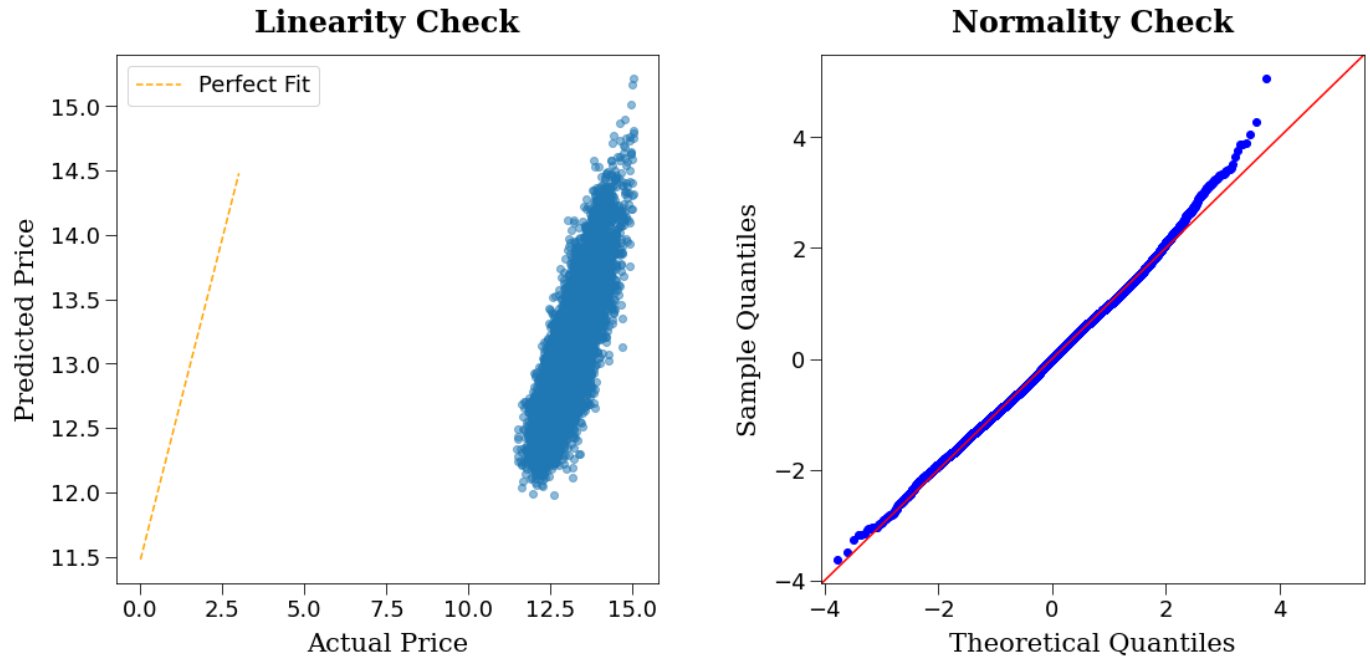
In [124]:
```
# Only log transforming the target (y component)

out_seattle_log_y = np.log(kc_y_out_seattle)
```

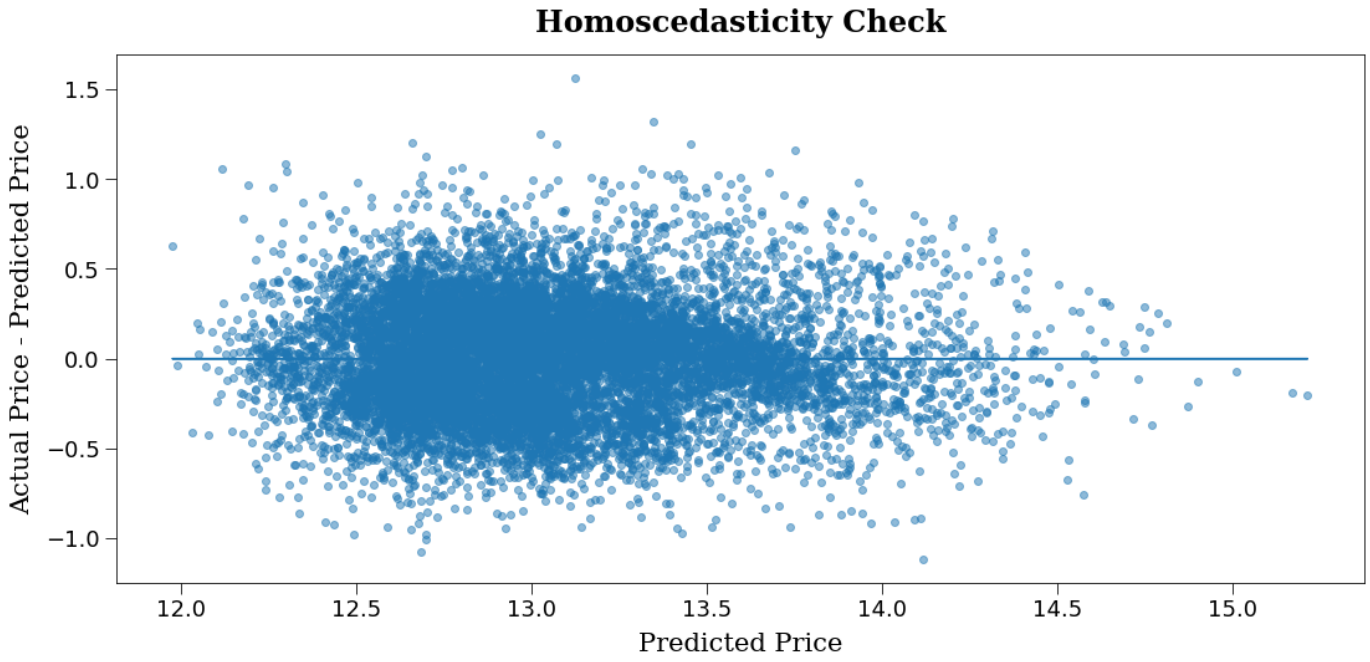
In [125]:
```
# Creating the log model

out_seattle_log_const = sm.add_constant(kc_X_out_seattle)
out_seattle_log_model = sm.OLS(endog=out_seattle_log_y, exog=out_seattle_log_const).fit()
```

Investigating the Linearity, Normality and Homoscedasticity of ***out_seattle_log_model***

In [126]:

```
# Creating the visualizations necessary to explore the Linearity, Normality, and Homoscedasticity of the model

fig = plt.figure()
gs = fig.add_gridspec(2, 2, wspace=.3, hspace=.3)

lin_ax = fig.add_subplot(gs[0, 0])
norm_ax = fig.add_subplot(gs[0, 1])
homo_ax = fig.add_subplot(gs[1, :])

out_seattle_preds = out_seattle_log_model.predict(out_seattle_log_const)
perfect_line = np.arange(out_seattle_log_y.min(), out_seattle_log_y.max())
out_seattle_resids = (out_seattle_log_y - out_seattle_preds)

lin_ax.plot(perfect_line, linestyle="--", color="orange", label="Perfect Fit")
lin_ax.scatter(out_seattle_log_y, out_seattle_preds, alpha=0.5)
lin_ax.set_xlabel("Actual Price", family='serif')
lin_ax.set_ylabel("Predicted Price", family='serif')
lin_ax.set_title('Linearity Check', family='serif')
lin_ax.legend()

sm.graphics.qqplot(out_seattle_resids, dist=stats.norm, line='45', fit=True, ax=norm_ax)
norm_ax.set_title('Normality Check', family='serif')
norm_ax.set_xlabel(norm_ax.get_xlabel(), family='serif')
norm_ax.set_ylabel(norm_ax.get_ylabel(), family='serif')

homo_ax.scatter(out_seattle_preds, out_seattle_resids, alpha=0.5)
homo_ax.plot(out_seattle_preds, [0 for i in range(len(kc_X_out_seattle))])
homo_ax.set_xlabel("Predicted Price", family='serif')
homo_ax.set_ylabel("Actual Price - Predicted Price", family='serif')
homo_ax.set_title('Homoscedasticity Check', family='serif')

fig.savefig('visuals/out_seattle_model_check_log.png' , bbox_inches='tight')

plt.show()
```





Comparing `out_seattle` Models

In [127]:

```
# Creating lists of the models and their names in str format to zip() thru to create the comparison dataframe

out_seattle_model_list = [out_seattle_base_model, out_seattle_full_model, out_seattle_full_VIF_model,
                          out_seattle_full_RFECV_model, out_seattle_fin_model, out_seattle_log_model]
out_seattle_model_names = ['base', 'full', 'full_VIF', 'full_RFECV', 'fin', 'log']
```

In [128]:

```
# Creating the comparison dataframe

out_seattle_r_vals = {}
out_seattle_r_adj_vals = {}
out_seattle_compare_params_df = pd.DataFrame()
for ols_i, (ols_model, model_name) in enumerate(zip(out_seattle_model_list, out_seattle_model_names)):
    model_params = ols_model.params
    model_params.name = model_name
    out_seattle_r_vals[model_name] = ols_model.rsquared
    out_seattle_r_adj_vals[model_name] = ols_model.rsquared_adj

    out_seattle_compare_params_df = pd.concat([out_seattle_compare_params_df, model_params], axis=1)

for r_i, r_dict in enumerate([out_seattle_r_vals, out_seattle_r_adj_vals]):
    r_name = 'r_score' if r_i==0 else 'r_adj_score'
    r_ser = pd.DataFrame(r_dict.values(), index=r_dict.keys(), columns=[r_name]).transpose()

    out_seattle_compare_params_df = pd.concat([out_seattle_compare_params_df, r_ser])
```

In [129]:

```
# Rounding columns for readability and displaying the comparision dataframe

for col in out_seattle_compare_params_df.columns[:-1]:
    out_seattle_compare_params_df[col] = out_seattle_compare_params_df[col].round(2)

display(out_seattle_compare_params_df)
```

	base	full	full_VIF	full_RFECV	fin	log
const	-88031.96	1623854.61	-71551.43	-71551.43	-72877.59	1.191244e+01
sqft_living	274.79	117.21	NaN	NaN	NaN	NaN
sqft_lot	NaN	0.09	0.28	0.28	0.27	7.592585e-07
sqft_living15	NaN	59.57	100.06	100.06	100.09	2.170949e-04
sqft_lot15	NaN	-0.29	-0.22	-0.22	-0.22	-1.465068e-07
floors	NaN	-33303.69	-17430.15	-17430.15	-17429.06	6.300600e-03
bedrooms	NaN	-17308.27	7787.86	7787.86	7859.85	1.535420e-02
bathrooms	NaN	50836.90	88325.45	88325.45	88620.09	1.254691e-01
basement	NaN	-12974.73	15322.16	15322.16	15217.88	4.353457e-02
grade_4_Low	NaN	-1546358.38	-68083.09	-68083.09	NaN	NaN
grade_5_Fair	NaN	-1542575.16	-70039.11	-70039.11	-69234.79	-2.987118e-01
grade_6_Low_Average	NaN	-1512696.03	-44490.69	-44490.69	-44259.22	-1.998210e-01
grade_7_Average	NaN	-1460580.49	NaN	NaN	NaN	NaN
grade_8_Good	NaN	-1396660.56	79979.87	79979.87	80128.48	2.084065e-01
grade_9_Better	NaN	-1290985.95	218543.60	218543.60	218659.91	4.119348e-01
grade_10_Very_Good	NaN	-1126670.65	421548.82	421548.82	421580.38	6.038023e-01
grade_11_Excellent	NaN	-895490.94	694388.83	694388.83	694329.46	7.448251e-01
grade_12_Luxury	NaN	-620163.85	1008847.43	1008847.43	1008671.92	8.751767e-01
view_FAIR	NaN	188561.25	195214.73	195214.73	194746.14	2.115879e-01
view_AVERAGE	NaN	58891.43	69604.22	69604.22	69726.34	7.951031e-02
view_GOOD	NaN	85552.92	88884.54	88884.54	88931.12	6.684284e-02
view_EXCELLENT	NaN	351261.56	367216.01	367216.01	367194.68	2.613644e-01
waterfront	NaN	584099.57	597325.25	597325.25	597656.29	4.678498e-01
condition_Fair	NaN	-119905.87	-29974.66	-29974.66	NaN	NaN
condition_Average	NaN	-92324.19	NaN	NaN	NaN	NaN
condition_Good	NaN	-62367.90	35968.72	35968.72	36503.41	4.907338e-02
condition_Very_Good	NaN	-26097.65	77089.86	77089.86	77828.57	1.324609e-01
renovated	NaN	139378.96	160733.11	160733.11	161218.90	1.626134e-01
yr_built_1920_to_1940_s	NaN	341.04	71212.78	71212.78	69159.60	1.454178e-01
yr_built_1940_to_1960_s	NaN	74516.72	158097.61	158097.61	157501.35	2.579424e-01
yr_built_1960_to_1980_s	NaN	-20379.14	60449.98	60449.98	60335.75	8.779453e-02
yr_built_1980_to_2000_s	NaN	-78898.06	NaN	NaN	NaN	NaN
yr_built_2000_to_2020_s	NaN	-67913.07	23061.71	23061.71	23224.04	2.788341e-02
r_score	0.50	0.67	0.65	0.65	0.65	6.556790e-01
r_adj_score	0.50	0.67	0.65	0.65	0.65	6.549568e-01

Final `out_seattle` Equation

In [130]:
```
# Taking the pieces of the final equation and making them more readable

out_seattle_log_int = int(np.exp(out_seattle_compare_params_df.log.dropna()[0]))
out_seattle_log_percs = out_seattle_compare_params_df.log.dropna()[1:-2] * 100
```

In [131]:
```
# Continuing the process of making the pieces of the equation more readable

out_seattle_fin_percs = list(out_seattle_log_percs.values.astype(int))
out_seattle_fin_preds = list(out_seattle_log_percs.index)
```

In [132]:
```
# Creating strings of the pieces of the equation with their column names

out_seattle_perc_eq = ['{}% * {}'.format(perc, pred) for perc, pred in zip(out_seattle_fin_percs, out_seattle_f
```

In [133]:
```
# Putting the pieces together in a readable format

model_eq = 'price = '
model_int_string = str(out_seattle_log_int) +' +\n'
model_var_string = ' +\n'.join(out_seattle_perc_eq)
```

In [134]:
```
# Putting the final equation together and printing it

out_seattle_log_eq = model_eq + model_int_string + model_var_string

print(out_seattle_log_eq)
```

```
price = 149110 +
0% * sqft_lot +
0% * sqft_living15 +
0% * sqft_lot15 +
0% * floors +
1% * bedrooms +
12% * bathrooms +
4% * basement +
-29% * grade_5_Fair +
-19% * grade_6_Low_Average +
20% * grade_8_Good +
41% * grade_9_Better +
60% * grade_10_Very_Good +
74% * grade_11_Excellent +
87% * grade_12_Luxury +
21% * view_FAIR +
7% * view_AVERAGE +
6% * view_GOOD +
26% * view_EXCELLENT +
46% * waterfront +
4% * condition_Good +
13% * condition_Very_Good +
16% * renovated +
14% * yr_built_1920_to_1940_s +
25% * yr_built_1940_to_1960_s +
8% * yr_built_1960_to_1980_s +
2% * yr_built_2000_to_2020_s
```

Analyzing Model Performance

Unfortunately, none of the models I created could truly be relied upon to predict the sales prices of a residential property. The r^2, and the adjusted r^2 scores never even broke a value of .7, meaning that the models were not reliable as predictive algorithms. I could have refined the models further to increase their predictive abilities, but I will discuss why I didn't in the Future Investigations section. While the Insights and Conclusions I gleaned from my analysis are certainly an important first step, more data will need to be gathered if King County Development is to build a regression model that serves as a reliable predictive algorithm.

Stakeholder and Business Problem Decision

Based on the results obtained through the three models I created, I chose a real estate developer as the stakeholder for this project. While I could have chosen a real estate agency, I felt a developer could make better use of the insights I gained through my analysis. Real estate agencies would be limited by the desires of their client and the physical location of the client's property. Developers have more freedom in their decision making, both in terms what changes to make to the properties they acquire, and what properties to acquire in the first place. They may ultimately rely on investors to acquire the property, but they will need an analysis like this, to convince those investors of a property's / design's value. A real estate developer could also take on clients simply wanting renovations, or even remodeling services. My analysis would just be a step in a process, of course, which I will discuss more in the Future Investigations section. I named my hypothetical client King County Development.

For the specific business problem, I chose to provide the real estate developer with key insights into which property features were the most relevant in predicting the sales price, and how much the sales price would be affected by either a one-unit increase in those predictors, or if a property were to have certain categorical features.

Insights and Conclusions

By creating the separate models, I was able to gain valuable insights. I created a dataframe and visualiztions with the coefficients of the three log-transformed models to identify the differences between the predictors that were included in each model and their values more easily. I used the visualizations in my presentation to my stakeholder.

Intercepts

- The Seattle model had the highest intercept, the entire King County model had the second highest, and the Outside Seattle model had the lowest.
- Seattle Model - $287,626.85
- Entire King County Model - $261,735.06
- Outside Seattle Model - $149,110.47

`sqft` Columns

- These columns were not important on a per unit basis, but I could still judge them by whether they were included, whether they were positive or negative, and if they were $= 0.02$ in the dataframe / visualizations, or ≈ 0.
- `sqft_living`
  - It was only even included for the prices of residential property inside Seattle, and the fact that it was 0.02 and positive, showing that living space is very important inside Seattle.
- `sqft_living15`
  - It was positive 0.02, or very important, in all three models.
- `sqft_lot`
  - It was positive ≈ 0 in the entire King County and the Outside Seattle models, and negative ≈ 0 in the Seattle model, meaning that it was beneficial to the former, and detrimental to the latter, but not as important or harmful as if it was 0.02.
- `sqft_lot15`
  - It was negative ≈ 0, or detrimental, in all three models.

`floors`

- This was interesting. It was most important in the entire King County model. In the Seattle model, it was still important, but in the Outside Seattle model, it was barely important at all.

`bedrooms`

- Another interesting predictor. It wasn't even included in the entire King County model, and it actually had a negative effect in the Seattle model. However, it did have a positive effect in the Outside Seattle model.

`bathrooms`

- It was very important in the entire King County model, and even more so in the Outside Seattle model. In the Seattle model it was important, especially compared to the previous predictors, but not nearly as much as in the other models.

`basement`

- It wasn't even included in the Seattle model. It was very important in the entire King County model. In the Outside Seattle model, it was important, and its worth noting that having a basement adds more value than additional floors outside of Seattle.

`grade`

- This seems to be the most important feature when determining the price of a property. The categories in this feature resulted in the largest changes in price, both positively and negatively. Anything below `7_Average` had a negative effect if it was included at all, and the lower the `grade`, the worse the effect, while `7_Average` itself was not included in any of the models. Anything above `7_Average` had a positive effect, and the size of the effect increases as `grade` does. A few things that are worth noting, is how much of an increase in value is gained by increasing the `grade` from `8_Good` to `9_Better`, as well as the increase from `9_Better` to `10_Very_Good`, and another is how much a `grade` of `12_Luxury` increases the price for Seattle properties.
- Specific Recommendation
  - Create a database of example pictures of properties with each `grade` value (*as many as can be gathered*)
  - These could be used to guide King County Development as they design new properties, or act as a guide, or as an incentive in any promotional material, for any clients seeking renovation or remodeling services.

`view`

- The `FAIR` and `AVERAGE` dummy columns weren't even included in the Seattle model, while even having a `FAIR` view was very important in the Outside Seattle model. It is also important in the entire King County model, although less so. Having an `AVERAGE` view added a similar increase in value in both the Outside Seattle and entire King County models, so an `AVERAGE` view could possibly be the standard / baseline in Seattle, as the `grade_7_Average` column seemed to be the standard / baseline in the `grade` column. It is also especially worth noting how much of increase having an `EXCELLENT` view has on the price.
- Specific Recommendation
  - Again, create a database of example pictures of properties with each `view` value, which could be used for the purposes I've previously described.

`waterfront`

- Having a waterfront property understandably adds significant value to the price, in all three models, and is the most significant in the Outside Seattle model.

`condition`

- The `Fair` and `Average` dummy columns weren't included in any of the models, and the `Average` column again seemed to be standard / baseline. Having a `condition` of `Good` was slightly more important inside of Seattle, while having a `condition` of `Very_Good` was slightly more important outside of Seattle.
- Specific Recommendation
  - Once again, create a database of example pictures of properties with each `condition` value, which could be used for the purposes I've previously described.

`renovated`

- An extremely important insight was gained for this predictor. It wasn't even included in the Seattle model, and it corresponded to a much smaller increase in value in the entire King County model than in the Outside Seattle model, where it was very important.

`yr_built`

- This was the most difficult predictor to analyze. In the entire King County model, the effect was negative for each dummy column and, generally, was less negative the further back in time the property was built. In the Seattle model, the `1920s to 1940s` dummy column wasn't even included, neither was the `1980s to 2000s` in the Outside Seattle model. Both the Seattle and Outside Seattle models showed that older built properties seemed to be more valuable in general, although the trend was much clearer Outside Seattle. Just to clarify that there was an overall negative effect as the `yr_built` increases in each model, I created a copy of this notebook in which I did not restructure the `yr_built` column (https://github.com/sarnadpy32/king_county_development/blob/master/Phase%202%20-%20Project%20-%20yr_built%20changed.ipynb), which confirmed my hypothesis. I included the `yr_built` results from the other notebook below.

In [135]:
```
# Taking the `log` column from each of the comparison dataframes, renaming them, and combining them into a
# dataframe to easily compare the results from the three models

kc_log = kc_compare_params_df.log.copy()
seattle_log = seattle_compare_params_df.log.copy()
out_seattle_log = out_seattle_compare_params_df.log.copy()

kc_log.name = 'kc'
seattle_log.name = 'seattle'
out_seattle_log.name = 'out_seattle'

log_compare_df = pd.concat([kc_log, seattle_log, out_seattle_log], axis=1)

log_comp_df = log_compare_df.copy()

log_compare_df = log_compare_df.fillna('-')
```

In [136]:
```
# Reformatting the intercept of the model to make it easily understandable

log_compare_df.loc['const'] = log_compare_df.loc['const'].apply(lambda x: np.exp(x)).round(2)
```

In [137]:
```
# Reformatting the feature coefficients of the model to make it easily understandable

log_compare_df.iloc[1:-2] = log_compare_df.iloc[1:-2].applymap(lambda x: '{:.2f}%'.format(x*100) if type(x)!=st
```

In [138]:
```
# Reformatting the r-squared & the adj r-squared of the model to make it easily understandable

log_compare_df.iloc[-2:] = log_compare_df.iloc[-2:].applymap(lambda x: round(x, 3))
```

In [139]:
```
# Uncomment to print the dataframe in markdown appropriate format

# print(log_compare_df.to_markdown(colalign=['center']*4))
```

You can see the markdown version of the final equation comparison dataframe for the three models from the primary analysis below.

	kc	seattle	out_seattle
const	$262K	$288K	$149K
sqft_living	-	0.02%	-
sqft_lot	0.00%	-0.00%	0.00%
sqft_living15	0.02%	0.02%	0.02%
sqft_lot15	-0.00%	-0.00%	-0.00%
floors	7.98%	2.54%	0.63%
bedrooms	-	-2.88%	1.54%
bathrooms	11.02%	4.55%	12.55%
basement	10.23%	-	4.35%
grade_4_Low	-73.30%	-	-
grade_5_Fair	-49.22%	-41.20%	-29.87%
grade_6_Low_Average	-27.92%	-24.59%	-19.98%
grade_7_Average	-	-	-
grade_8_Good	22.32%	19.49%	20.84%
grade_9_Better	46.68%	43.83%	41.19%
grade_10_Very_Good	66.68%	56.54%	60.38%
grade_11_Excellent	84.07%	70.60%	74.48%
grade_12_Luxury	96.51%	118.49%	87.52%
view_FAIR	14.42%	-	21.16%
view_AVERAGE	7.81%	-	7.95%
view_GOOD	9.71%	4.47%	6.68%
view_EXCELLENT	25.78%	21.45%	26.14%
waterfront	30.97%	38.10%	46.78%
condition_Fair	-	-	-
condition_Average	-	-	-
condition_Good	5.17%	7.05%	4.91%
condition_Very_Good	11.36%	10.24%	13.25%
renovated	4.28%	-	16.26%
yr_built_1920_to_1940_s	-5.18%	-	14.54%
yr_built_1940_to_1960_s	-19.28%	-14.83%	25.79%
yr_built_1960_to_1980_s	-39.73%	-32.14%	8.78%
yr_built_1980_to_2000_s	-49.49%	-30.39%	-
yr_built_2000_to_2020_s	-45.60%	-34.17%	2.79%
r_score	0.645	0.698	0.656
r_adj_score	0.645	0.697	0.655

You can see the markdown version of the `yr_built` results from the final equation comparison dataframe for the three models from the secondary analysis (https://github.com/sarnadpy32/king_county_development/blob/master/Phase%202%20-%20Project%20-%20yr_built%20changed.ipynb) below. In that version, in case you didn't catch my reason for creating a secondary analysis, I performed an analysis without transforming the `yr_built` feature into a categorical feature.

	kc	seattle	out_seattle
yr_built	-0.58%	-0.44%	-0.32%

Visualizations of the Coefficients of the Three Log Models

In [140]:

```
# Printing the original column lists to see how to group the columns accordingly when I created the visualizati

print(bold_red +'num_cols:'+ every_off, num_cols)
print(bold_red +'new_cat_order:'+ every_off, new_cat_order)
```

```
num_cols: ['sqft_living', 'sqft_lot', 'sqft_living15', 'sqft_lot15', 'floors', 'bedrooms', 'bathrooms']
new_cat_order: ['basement', 'grade', 'view', 'waterfront', 'condition', 'renovated', 'yr_built', 'city']
```

In [141]:

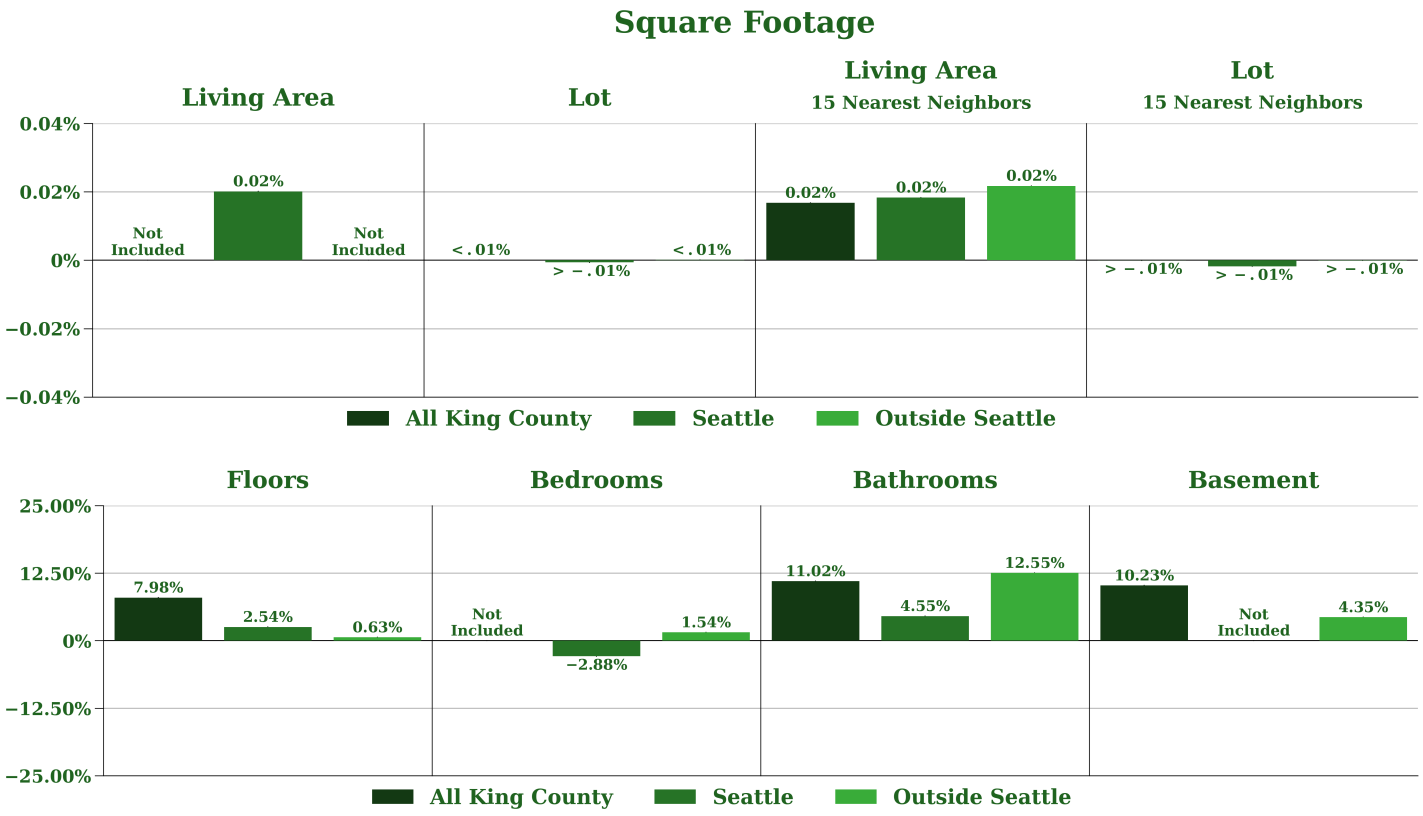
```
# Splitting the columns into appropriate lists

sqft_cols = num_cols[:4]
fl_bed_bath_base_cols = num_cols[4:] + ['basement']
perc_groups = \
[sqft_cols, fl_bed_bath_base_cols, ['grade'], ['view', 'waterfront'], ['condition', 'renovated'], ['yr_built']]
```

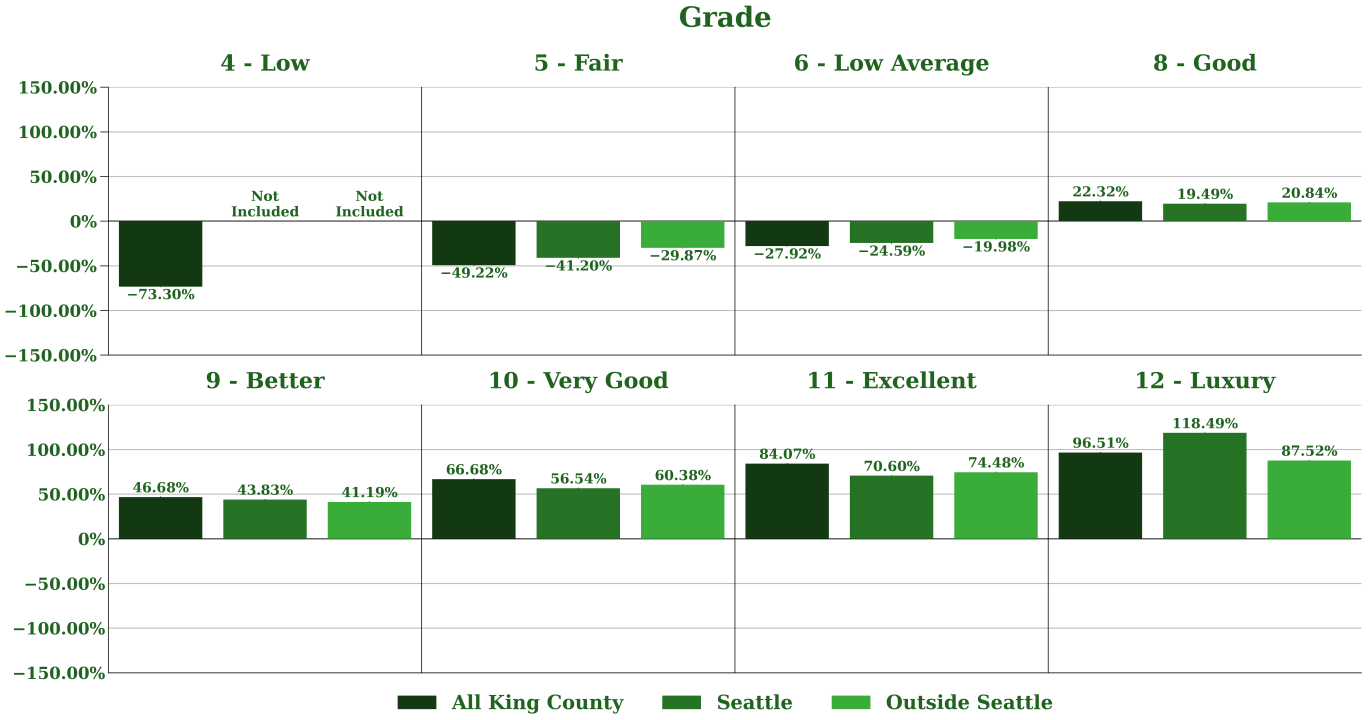
In [142]:

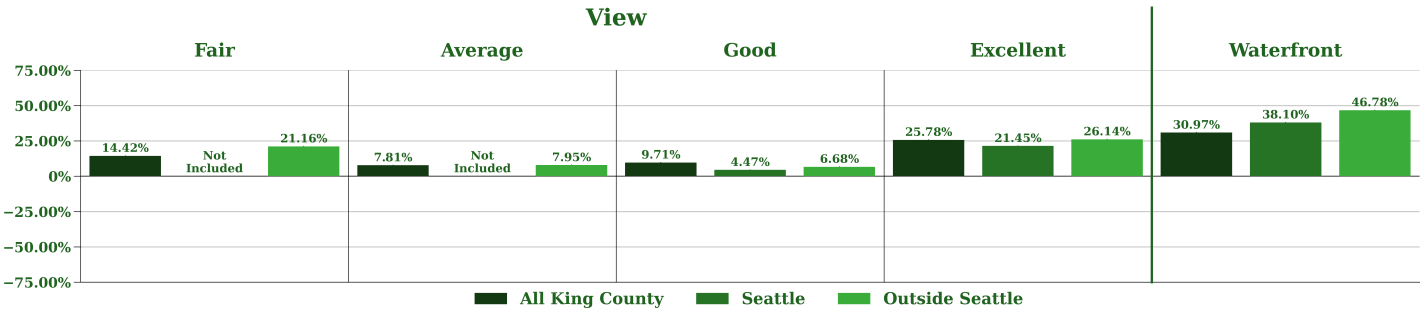
```
# I iterated thru the lists I created and created an appropriate dataframe for each situation

for p_i, perc_group in enumerate(perc_groups):

    # As I planned to show the visualizations I created to my stakeholder, I wanted to control the details prec
    # This meant changing things depending on the group, such as the title, xlim, ylim, xticks, yticks, and the
    # labels for the bars
    #-----------------------------------------------------------------------------------------------------
    if 'grade' in perc_group or 'view' in perc_group or 'condition' in perc_group or 'yr_built' in perc_group:
        r_col = 'grade' if 'grade' in perc_group else 'view' if 'view' in perc_group \
                else 'condition' if 'condition' in perc_group else 'yr_built'

        plot_df = log_comp_df.loc[log_comp_df.index.map(lambda x: r_col in x or x in perc_group)]

    else: plot_df = log_comp_df.loc[perc_group]

    plot_df = plot_df.dropna(how='all')

    p_min = plot_df.min(axis=1).max()
    p_max = plot_df.max(axis=1).max()
    p_idx = plot_df.index

    plot_df = plot_df.reset_index().rename(columns={'index': 'cat_col'})

    c_wrap = 5 if len(plot_df) % 5 == 0 else 4 if len(plot_df) % 4 == 0 else 3
    sep_x = 0

    g = sns.catplot(col='cat_col', data=plot_df, saturation=.5, kind="bar", ci=None, col_wrap=c_wrap,
                    palette=[get_lighter_color(c, .09) for c in [(0, .3, 0), (0, .6, 0), (0, .9, 0)]])

    g.fig.dpi = 300
    g.fig.subplots_adjust(wspace=0)

    for p_col in p_idx:
        p_ax = g.axes_dict[p_col]

        if p_i==0: p_lim, p_step = .0011, .0001
        else: p_lim, p_step = 1.5, .25

        # Tick and Lim customization
        #-----------------------------------------------------------------------------------------------------
        p_m = [m for m in np.arange(0, p_lim, p_step) if m >= np.abs(p_min) and m >= p_max]
        if p_m:
            p_m = p_m[0]

            if p_m==.25:
                p_step = .125
                p_ax.set_ylim(-p_m, p_m)

            else: p_ax.set_ylim(-p_m - p_step, p_m + p_step)

            if p_i!=0:
                if p_m==.25: ticks = np.arange(0, p_m + .01, p_step)
                elif p_i==2: ticks = np.arange(0, 1.51, .5)
                else: ticks = np.arange(0, p_m + p_step + .01, p_step)

                ticks = np.append(np.delete(np.flip(ticks), -1) * -1, ticks)
                p_ax.yaxis.set_ticks(ticks)

        # Bar label customization
        #-----------------------------------------------------------------------------------------------------
        p_rects = p_ax.findobj(mpatches.Rectangle)[:-1]
        for rect in p_rects:
            rect.set_zorder(9)
            r_val, r_wid = rect.get_height(), rect.get_width()
            r_x = rect.get_xy()[0] + (r_wid / 2)

            annot_kws = dict(textcoords='offset pixels', size=15, ha='center', arrowprops={'arrowstyle': '-'},
                             family='serif', weight='bold', color=(.12, .39, .12))

            if np.isnan(r_val):
                p_ax.annotate('Not\nIncluded', xy=(r_x, 0), xytext=(0, 9), va='bottom', **annot_kws)
            else:
                if r_val > 0: r_low, r_off, r_va = '<.01', 9, 'bottom'
                if r_val < 0: r_low, r_off, r_va = '>-.01', -9, 'top'
```

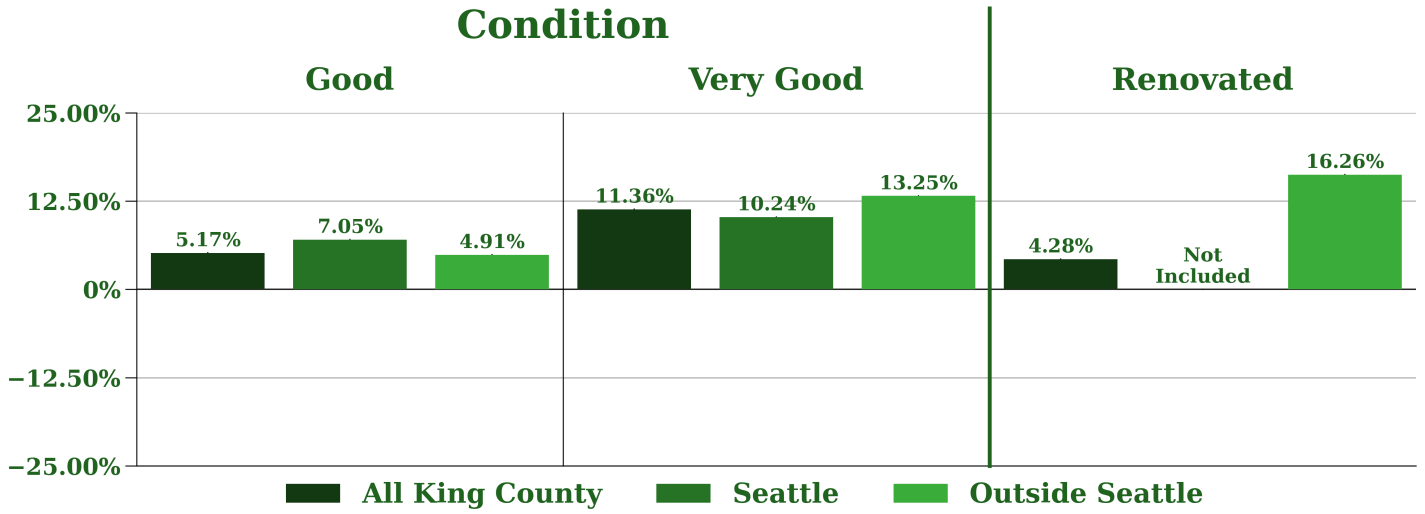
```
                if round(np.abs(r_val), 4) < .0001: r_text = '$\mathbf{'+ r_low +'\%}$'
                elif r_val==0: r_text = '$\mathbf{0\%}$'
                else: r_text = '$\mathbf{'+'{:.2f}'.format(round(r_val, 4) * 100) +'\%}$'

                p_ax.annotate(r_text, xy=(r_x, r_val), xytext=(0, r_off), va=r_va, **annot_kws)

        # Title customization
        #-------------------------------------------------------------------------------------------------
        if '_' in p_col:
            p_col_list = p_col.split('_')

            if p_i==0:
                fig_tit = 'Square Footage'
                if '15' not in p_col_list[1]:
                    if 'living' in p_col: p_ax_tit = p_col_list[1].title() +' Area'
                    else: p_ax_tit = p_col_list[1].title()
                else:
                    if 'living' in p_col: p_ax_tit = p_col_list[1][:-2].title() +' Area'
                    else: p_ax_tit = p_col_list[1][:-2].title()

            if p_i!=0 and p_i!=5:
                fig_tit = p_col_list[0].title()
                if p_i==2:
                    if len(p_col_list)==4: p_ax_tit = ' - '.join(p_col_list[1:3]) +' '+ p_col_list[3]
                    else: p_ax_tit = ' - '.join(p_col_list[1:])
                if p_i==3: p_ax_tit = p_col_list[1].title()
                if p_i==4:
                    if p_col=='Good': p_ax_tit = p_col_list[1]
                    else: p_ax_tit = ' '.join(p_col_list[1:])

            if p_i==5: fig_tit, p_ax_tit = 'Year Built', p_col_list[2] +"'s " + ' '.join(p_col_list[3:5]) +"'s'

        if '_' not in p_col: p_ax_tit = p_col.title()

        # Settings that were basically the same for all the visualizations
        #-------------------------------------------------------------------------------------------------
        p_ax.axhline(0, color='k', lw=.9)
        p_ax.yaxis.set_major_formatter(viz_percentage_formatter)
        plt.setp(p_ax.get_yticklabels(), weight='bold', color=(.12, .39, .12))
        p_ax.tick_params('x', length=0, labelbottom=False)
        p_ax.grid(True, 'major', 'y', lw=1.2, alpha=.81)
        if p_col!=p_idx[0]: p_ax.tick_params('y', length=0)

        if p_col not in ['sqft_living15', 'sqft_lot15']: p_ax.set_title(p_ax_tit, family='serif', color=(.12, .
        else:
            p_tit = p_ax.set_title(p_ax_tit, family='serif', pad=45, color=(.12, .39, .12))

            renderer = g.fig.canvas.get_renderer()
            p_txt_x_max, p_txt_x_min = p_tit.get_tightbbox(renderer).max[0], p_tit.get_tightbbox(renderer).min[
            p_txt_x = ((p_txt_x_max - p_txt_x_min) / 2) + p_txt_x_min
            p_txt_y = p_tit.get_tightbbox(renderer).min[1]
            p_txt_x, p_txt_y = g.fig.transFigure.inverted().transform((p_txt_x, p_txt_y))

            p_txt = '$\mathbf{15}$ Nearest Neighbors'
            g.fig.text(p_txt_x, p_txt_y - .03, p_txt, size=18,  family='serif', weight='bold', ha='center', va=
                       color=(.12, .39, .12))

        # I also wanted to add divsion lines to separate the visualizations that were not necessarily the same
        # they belonged in the same group, which I set up along the way, as well as a title customization
        #-------------------------------------------------------------------------------------------------
        if p_i in [3, 4]:
            tit_i, sep_i = 1 if p_i==3 else 0, len(plot_df) - 2

            if p_col==list(g.axes_dict.keys())[tit_i]:
                fig_tit_x = g.fig.transFigure.inverted().transform(p_ax.get_tightbbox(renderer).max)[0]
            if p_col==list(g.axes_dict.keys())[sep_i]:
                sep_x_1 = g.fig.transFigure.inverted().transform(p_ax.get_tightbbox(renderer).max)[0]
                sep_y = g.fig.transFigure.inverted().transform(p_ax.get_tightbbox(renderer).min)[1] - .0005
            if p_col==list(g.axes_dict.keys())[sep_i + 1]:
                sep_x_2 = g.fig.transFigure.inverted().transform(p_ax.get_tightbbox(renderer).min)[0]
                sep_x = ((sep_x_2 - sep_x_1) / 2) + sep_x_1


    # Legend Details
    #-------------------------------------------------------------------------------------------------
```

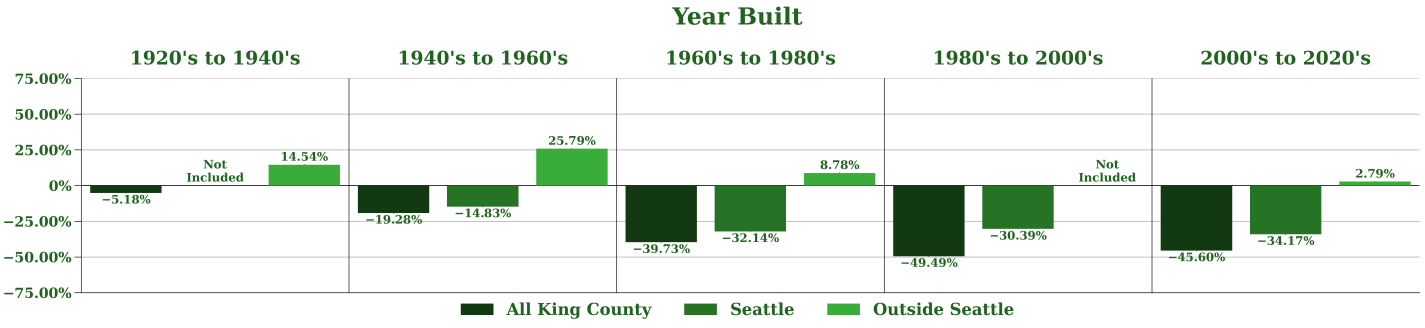
```
    fig_box = g.fig.dpi_scale_trans.transform((g.fig.get_tightbbox(renderer).width, g.fig.get_tightbbox(rendere
    fig_hlf = g.fig.transFigure.inverted().transform(fig_box)[0] / 2

    leg_txts = ["All King County", "Seattle", "Outside Seattle"]
    leg_txt_prop = dict(family='serif', weight='bold', size=21)
    leg_x = fig_hlf

    g.fig.legend(p_rects, leg_txts, loc=8, bbox_to_anchor=(leg_x, -.03), ncol=3, prop=leg_txt_prop,
                 labelcolor=(.12, .39, .12), frameon=False)

    # Title Details
    #-------------------------------------------------------------------------------------------------
    if p_i!=1:
        sup_x = fig_hlf
        if p_i==0: sup_x, sup_y = sup_x + .03, 1.11
        elif p_i==2: sup_y = 1.02
        elif p_i in [3, 4]: sup_x = fig_tit_x
        else: sup_y = 1.05

        g.fig.suptitle(r''+ fig_tit +'', x=sup_x, y=sup_y, size=30, family='serif', weight='bold', va='bottom',
                       color=(.12, .39, .12))

    # Division Lines
    #-------------------------------------------------------------------------------------------------
    fig_box = g.fig.dpi_scale_trans.transform((g.fig.get_tightbbox(renderer).width, g.fig.get_tightbbox(rendere
    fig_hght = g.fig.transFigure.inverted().transform(fig_box)[1]
    if p_i in [3, 4]:
        g.fig.add_artist(mlines.Line2D([sep_x, sep_x], [sep_y, fig_hght], lw=3, color=(.12, .39, .12)))

    # Saving the visualization
    #-------------------------------------------------------------------------------------------------
    g.fig.savefig('visuals/presentation_pic_'+ str(p_i + 1), bbox_inches='tight')
```





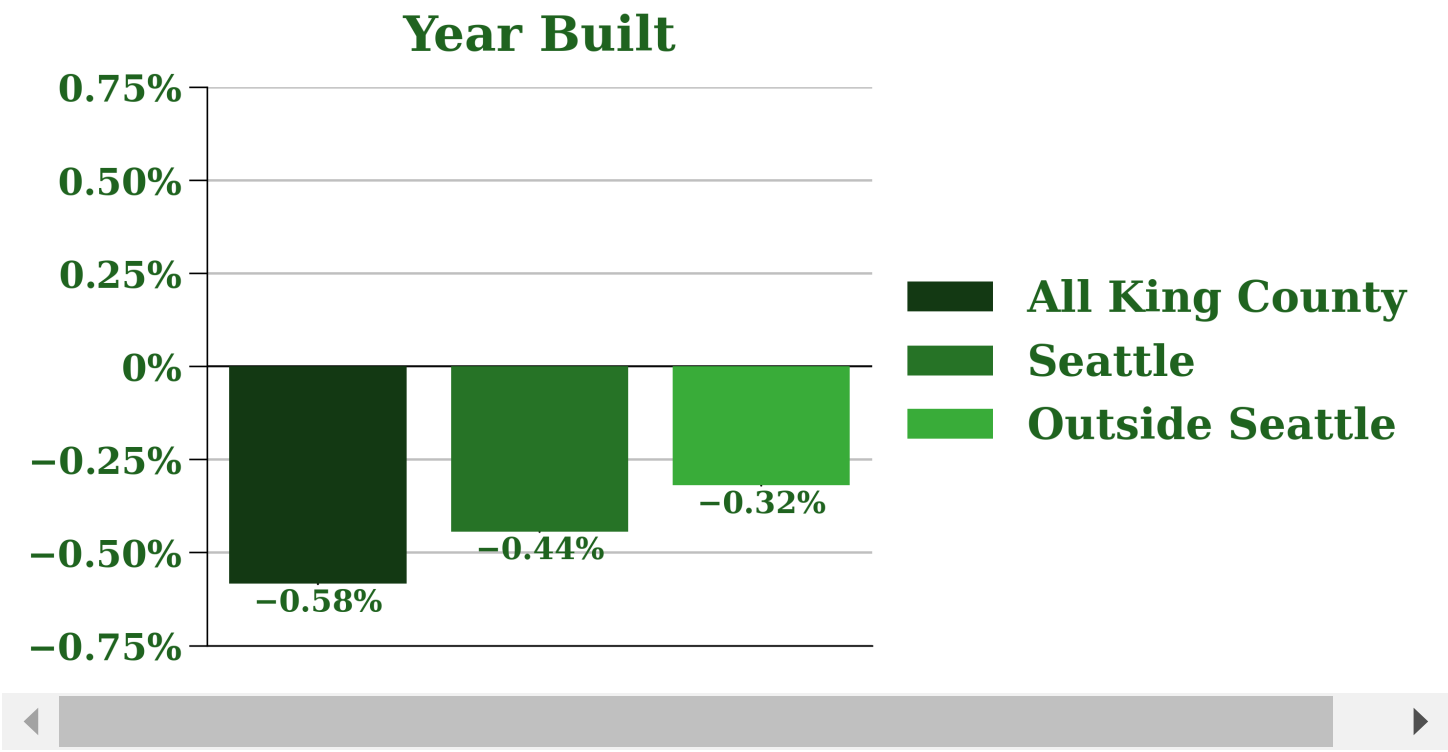






yr_built Results from the Secondary Analysis (https://github.com/sarnadpy32/king_county_development/blob/master/Phase%202%

%20Project%20-%20yr_built%20changed.ipynb)



Future Investigations

Besides the picture databases I mentioned regarding the `grade`, `view`, and `condition` features, there is a lot more work that King County Development will need to perform if they want to be profitable, efficient in their spending, and if they want to provide the highest quality services to their clients and / or investors. There is more valuable information that can no doubt be gleaned from the dataset that was provided to me for this project, including testing interactions and / or building polynomial regression models, but the dataset only covered one year's worth of sales, so any further investigation would still be limited.

A much larger dataset, possibly with even more predictors to build models with, can undoubtedly be obtained (https://kingcountyexec.govqa.us/WEBAPP/_rs/). Data on commercial property could also be gathered. At that point there should be enough data to build separate models for each city in King County, and to also analyze the various ZIP codes within each city. They should then develop maps of each city and / or ZIP code, separated with the appropriate zoning laws, to see what is even possible in every inch of King County.

Areas of high potential should be identified, and the owners of any potentially lucrative properties should be approached to gauge their interest in either their property being acquired, or if they are at least interested in any renovation or remodeling services. If not, King County Development will at least be fully prepared should any enticing properties become available, or if they are approached for their other services.
The image below is from a video (https://www.youtube.com/watch?v=ZeRd3aurWz8) from the Real Estate for Noobs (https://www.youtube.com/c/RealEstateforNoobs) YouTube channel, and the steps that it shows is still only one small piece of the puzzle when it comes to real estate development.


DUE DILIGENCE
ANALYZE ZONING REGULATIONS
STUDY NEIGHBORHOOD
FIND HIGHEST & BEST USE
PITCH TO INVESTORS