Business Problem

Real Estate firm wants a model to help them accurately price houses.

- How should we price a house based on property features such as lot, footage, bedroom #, bathroom #, and renovations?

In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
import seaborn as sns
import numpy as np
import scipy.stats as stats
import statsmodels.api as sm
import warnings
warnings.filterwarnings('ignore')

from statsmodels.formula.api import ols
from sklearn.feature_selection import RFE
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score, mean_absolute_err
from sklearn.model_selection import KFold, cross_val_score
from sklearn.preprocessing import LabelEncoder,  OneHotEncoder
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
```

Load Data

In [2]:
```
df_column_names = ('data/column_names.md')
df_house = pd.read_csv('data/kc_house_data.csv')
```

In [3]:
```
df_house.head()
```

Out[3]:

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront
0	7129300520	10/13/2014	221900.0	3	1.00	1180	5650	1.0	NaN
1	6414100192	12/9/2014	538000.0	3	2.25	2570	7242	2.0	0.0
2	5631500400	2/25/2015	180000.0	2	1.00	770	10000	1.0	0.0
3	2487200875	12/9/2014	604000.0	4	3.00	1960	5000	1.0	0.0
4	1954400510	2/18/2015	510000.0	3	2.00	1680	8080	1.0	0.0

5 rows × 21 columns

- id - unique identifier for a house
- date - Date house was sold

- price - Price is prediction target
- bedrooms - Number of Bedrooms/House
- bathrooms - Number of bathrooms/bedrooms
- sqft_livingsquare - footage of the home
- sqft_lotsquare - footage of the lot
- floorsTotal - floors (levels) in house
- waterfront - House which has a view to a waterfront
- view - # of views
- condition - How good the condition is (Overall)
- grade - overall grade given to the housing unit, based on King County grading system
- sqft_above - square footage of house apart from basement
- sqft_basement - square footage of the basement
- yr_built - Built Year
- yr_renovated - Year when house was renovated
- zipcode - zip
- lat - Latitude coordinate
- long - Longitude coordinate
- sqft_living15 - The square footage of interior housing living space for the nearest 15 neighbors
- sqft_lot15 - The square footage of the land lots of the nearest 15 neighbors

Data Exploration

In [4]:
```
df_house.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   id             21597 non-null  int64  
 1   date           21597 non-null  object 
 2   price          21597 non-null  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  float64
 9   view           21534 non-null  float64
 10  condition      21597 non-null  int64  
 11  grade          21597 non-null  int64  
 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(8), int64(11), object(2)
memory usage: 3.5+ MB
```

In [5]:
```
# Fields with nulls: waterfront, view,  yr_renovated
```

In [6]:
```python
# print top 5 most frequent values in each column
for col in df_house.columns:
    print(col, '\n', df_house[col].value_counts(normalize=True).head(), '\n
```

```
condition
 3    0.649164
4    0.262861
5    0.078761
2    0.007871
1    0.001343
Name: condition, dtype: float64


grade
 7     0.415521
8     0.280826
9     0.121082
6     0.094365
10    0.052507
Name: grade, dtype: float64


sqft_above
```

In [7]:
```python
df_house['waterfront'].value_counts(normalize=True)
# Make waterfront boolean where nulls are non waterfront properties
```

Out[7]:
```
0.0    0.992404
1.0    0.007596
Name: waterfront, dtype: float64
```

In [8]:
```python
df_house['view'].value_counts(normalize=True)
# Make view boolean where nulls are properties without a view
```

Out[8]:
```
0.0    0.901923
2.0    0.044441
3.0    0.023591
1.0    0.015325
4.0    0.014721
Name: view, dtype: float64
```

In [9]:
```
df_house['yr_renovated'].value_counts(normalize=True)
# Make yr_renovated boolean where nulls are unrenovated properties
```

Out[9]:
```
0.0       0.958096
2014.0    0.004112
2003.0    0.001746
2013.0    0.001746
2007.0    0.001690
            ...
1946.0    0.000056
1959.0    0.000056
1971.0    0.000056
1951.0    0.000056
1954.0    0.000056
Name: yr_renovated, Length: 70, dtype: float64
```

In [10]:
```
# check all unique values for 'grade'
grade_values = list(df_house['grade'].unique())
grade_values.sort()
grade_values
```

Out[10]:
```
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
```

In [11]:
```
# check all unique values for 'condition'
condition_values = list(df_house['condition'].unique())
condition_values.sort()
condition_values
```

Out[11]:
```
[1, 2, 3, 4, 5]
```

In [12]:
```
# check for identical home
sum(df_house.duplicated(subset=['id']))
```

Out[12]:
```
177
```

In [13]:
```
# check for identical home / sale date rows
sum(df_house.duplicated(subset=['id','date']))
```

Out[13]:
```
0
```

In [14]:
```
# ***is it an issue that the same home was sold several times for model?***
```

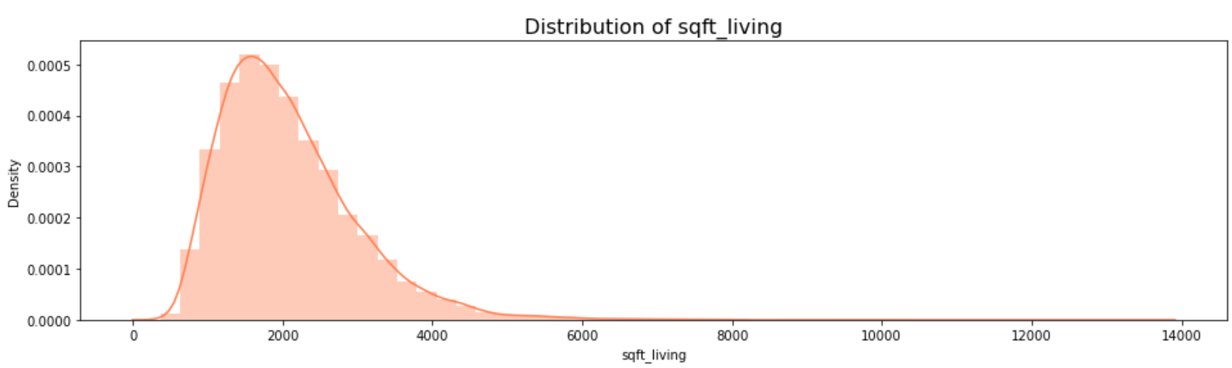
In [15]:
```
df_house['date'].value_counts(normalize=True)
```

Out[15]:
```
6/23/2014    0.006575
6/25/2014    0.006066
6/26/2014    0.006066
7/8/2014     0.005880
4/27/2015    0.005834
               ...
5/15/2015    0.000046
1/17/2015    0.000046
7/27/2014    0.000046
3/8/2015     0.000046
8/3/2014     0.000046
Name: date, Length: 372, dtype: float64
```

Only houses sold in 2014 and 2015 are in our data. This will limit how successful our model will be when used to predict pricing in the 2021 market.

In [16]:
```
# Plot of the target column price
plt.figure(figsize = (16, 4))
sns.distplot(a = df_house["price"], color = "#FF7F50")
plt.title("Distribution of Price", fontsize=16);
```

In [17]:
```
# Plot of the sqft_living column
plt.figure(figsize = (16, 4))
sns.distplot(a = df_house["sqft_living"], color = "#FF7F50")
plt.title("Distribution of sqft_living", fontsize=16);
```



Should look at the effect of log of price and log of sqft_living on the skew in data prep

Data Preparation

Remove unecessary features

Before removing features we will want to run tests to see which features are highly correlated or important to the model. However there are four columns I believe are reasonable to remove.

Columns to remove:

- id - unique identifier for a house
  - not needed for modeling
- lat - Latitude coordinate
- long - Longitude coordinate
- zipcode - zip
  - to simplify analysis we will not include location information

In [18]:
```
df_house = df_house.drop(['id', 'lat', 'long', 'zipcode'], axis=1)
```

Missing Values

In [19]:
```
# Fields with nulls: waterfront, view,  yr_renovated
df_house.waterfront.fillna(0, inplace=True)
df_house.view.fillna(0, inplace=True)
df_house.yr_renovated.fillna(0, inplace=True)
```

Data Type Conversions

In [20]:
```
# convert to datetime
# df_house['date'] = pd.to_datetime(df_house['date'])
```

In [21]:
```
# # convert zipcode to category
# df_house['zipcode']=df_house['zipcode'].astype('category')
```

Reformat Data

In [22]:
```
# create a year column instead of date
df_house['year_sold'] = df_house['date'].apply(lambda x: int(x[-4:]))
df_house['year_sold'].value_counts(normalize=True)
```

Out[22]:
```
2014    0.677038
2015    0.322962
Name: year_sold, dtype: float64
```

In [23]:
```
# Create function for season of sale to account for seasonality in pricing
def season(month):
    if month == 12 or 1 <= month <= 2:
        season = 'Winter'
    elif 3 <= month <= 5:
        season = 'Spring'
    elif 6 <= month <= 8:
        season = 'Summer'
    else:
        season = 'Fall'
    return season
```

In [24]:
```
# to use our function convert date to datetime
df_house['date'] = pd.to_datetime(df_house['date'])
# find month_sold
df_house['month_sold'] = df_house['date'].dt.month
```

In [25]:
```
df_house['season_sold'] = df_house['month_sold'].apply(season)
df_house['season_sold'].value_counts(normalize=True)
```

Out[25]:
```
Spring    0.301801
Summer    0.293004
Fall      0.234107
Winter    0.171089
Name: season_sold, dtype: float64
```

More houses sold in Spring and Summer than Fall and Winter. This could mean that houses sell at higher prices in Spring and Summer. Intuitively this makes sense based on general knowledge of housing markets.

In [26]:
```
df_house = df_house.drop(['date','month_sold'], axis=1)
```

In [27]:
```
# Convert sqft_basement col to basement col which indicates whether or not
# assume '0.0' and '?' values mean no basement
df_house['sqft_basement'] = df_house['sqft_basement'].map(lambda x: 0 if x
# convert column to float
df_house['sqft_basement'] = df_house['sqft_basement'].astype('float')
# add column called basement
df_house['basement'] = df_house['sqft_basement'].map(lambda x: 1 if x > 0 e
# remove the 'sqft_basement' column
df_house = df_house.drop(['sqft_basement'], axis=1)
```

In [28]:
```
# Change scale of grade to 0 - 10 so it's more intuitive
df_house['grade'] = df_house['grade'].map(lambda x: x-3)
grade_values = list(df_house['grade'].unique())
grade_values.sort()
grade_values
```

Out[28]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [29]:
```
# define function for year renovated only counting houses renovated 2000 or
def renovation(year):
    if year >= 2000:
        return 1
    else:
        return 0
```

In [30]:
```
# apply function to the 'yr_renovated' column
df_house['yr_renovated']  = df_house['yr_renovated'].apply(renovation)

# rename column
df_house.rename({'yr_renovated': 'Renovated_in_2000s'}, axis=1, inplace=Tru
df_house['Renovated_in_2000s'].value_counts()
```

Out[30]:
```
0    21218
1      379
Name: Renovated_in_2000s, dtype: int64
```

In [31]:
```
# ***should we remove this column completely?***
# binary classification of view
def views(count):
    if count > 0:
        return 1
    else:
        return 0
```

In [32]:
```
# apply function to 'view' column
df_house['view'] = df_house['view'].apply(views)
# rename view column
df_house.rename({'view': 'viewed'}, axis=1, inplace=True)
```

In [33]:
```
df_house['viewed'].value_counts(normalize=True)
```

Out[33]:
```
0    0.902209
1    0.097791
Name: viewed, dtype: float64
```

In [34]:
```
df_house
```

Out[34]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	viewed	condition
0	221900.0	3	1.00	1180	5650	1.0	0.0	0	3
1	538000.0	3	2.25	2570	7242	2.0	0.0	0	3
2	180000.0	2	1.00	770	10000	1.0	0.0	0	3
3	604000.0	4	3.00	1960	5000	1.0	0.0	0	5
4	510000.0	3	2.00	1680	8080	1.0	0.0	0	3
...	...	...	...	...	...	...	...	...	...
21592	360000.0	3	2.50	1530	1131	3.0	0.0	0	3
21593	400000.0	4	2.50	2310	5813	2.0	0.0	0	3
21594	402101.0	2	0.75	1020	1350	2.0	0.0	0	3
21595	400000.0	3	2.50	1600	2388	2.0	0.0	0	3
21596	325000.0	2	0.75	1020	1076	2.0	0.0	0	3

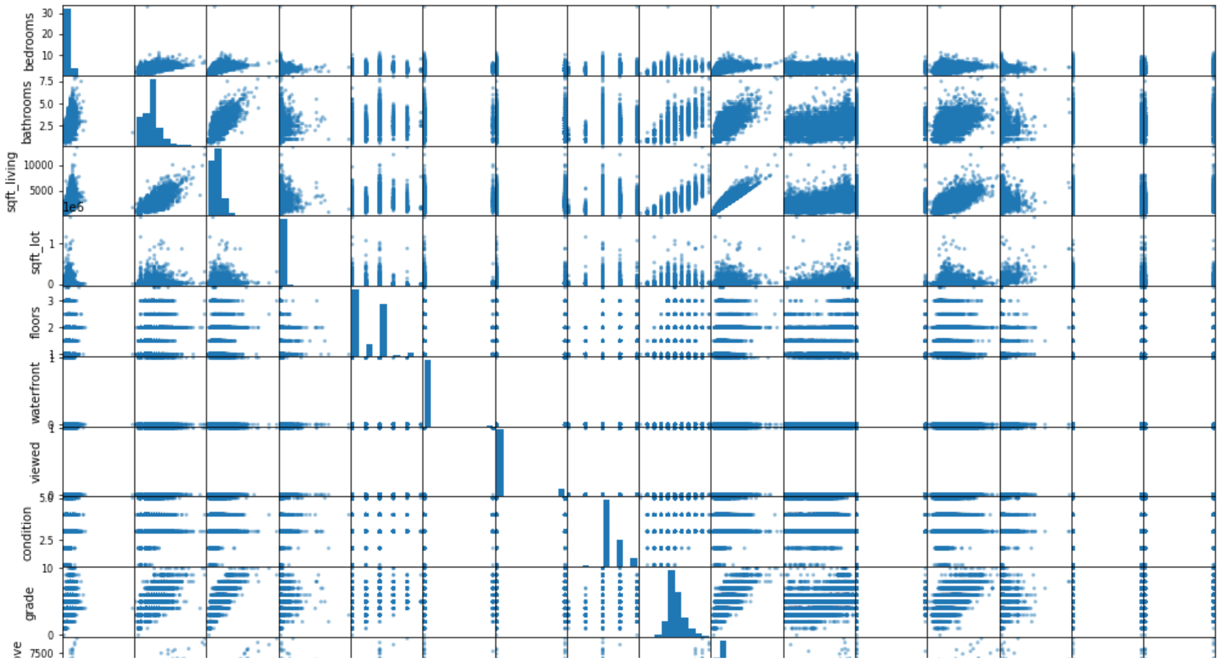
21597 rows × 18 columns

Multicollinearity

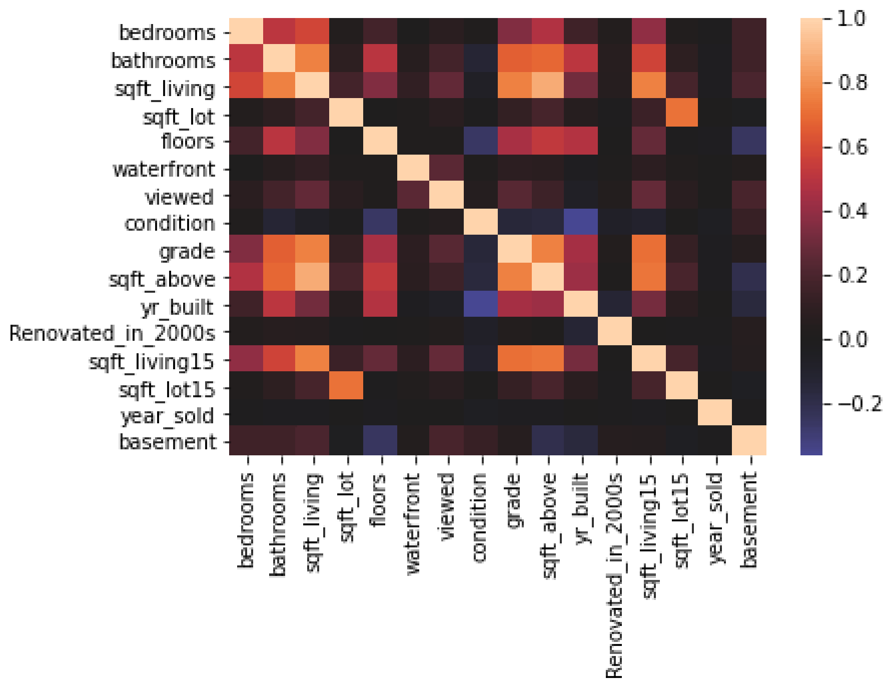
In [35]:
```
df_features = df_house.drop(['price'], axis=1)
```

In [36]:
```
fig_num = len(df_features.columns)
```

In [37]:
```
# ***Is this even useful with such a large number of features when we can u
pd.plotting.scatter_matrix(df_features,figsize = [fig_num, fig_num]);
plt.show()
```



In [38]:
```
sns.heatmap(df_features.corr(), center=0);
```



In [39]:
```
df_features.corr()
```

Out[39]:

	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	viewed
bedrooms	1.000000	0.514508	0.578212	0.032471	0.177944	-0.002127	0.079232
bathrooms	0.514508	1.000000	0.755758	0.088373	0.502582	0.063629	0.175884
sqft_living	0.578212	0.755758	1.000000	0.173453	0.353953	0.104637	0.268465
sqft_lot	0.032471	0.088373	0.173453	1.000000	-0.004814	0.021459	0.068216
floors	0.177944	0.502582	0.353953	-0.004814	1.000000	0.020797	0.016311
waterfront	-0.002127	0.063629	0.104637	0.021459	0.020797	1.000000	0.248683
viewed	0.079232	0.175884	0.268465	0.068216	0.016311	0.248683	1.000000
condition	0.026496	-0.126479	-0.059445	-0.008830	-0.264075	0.016648	0.046835
grade	0.356563	0.665838	0.762779	0.114731	0.458794	0.082818	0.235252
sqft_above	0.479386	0.686668	0.876448	0.184139	0.523989	0.071778	0.151909
yr_built	0.155670	0.507173	0.318152	0.052946	0.489193	-0.024487	-0.063826
Renovated_in_2000s	0.032953	0.063790	0.051035	-0.013414	0.004076	0.014795	0.037916
sqft_living15	0.393406	0.569884	0.756402	0.144763	0.280102	0.083823	0.271852
sqft_lot15	0.030690	0.088303	0.184342	0.718204	-0.010722	0.030658	0.064884
year_sold	-0.009949	-0.026577	-0.029014	0.005628	-0.022352	-0.005018	0.004302
basement	0.158412	0.159863	0.201198	-0.034889	-0.252465	0.039220	0.188896

In [40]:
```
# features correlated above .7 considered highly correlated
abs(df_features.corr()) > 0.7
```

Out[40]:

	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	viewed	conditio
bedrooms	True	False	False	False	False	False	False	Fals
bathrooms	False	True	True	False	False	False	False	Fals
sqft_living	False	True	True	False	False	False	False	Fals
sqft_lot	False	False	False	True	False	False	False	Fals
floors	False	False	False	False	True	False	False	Fals
waterfront	False	False	False	False	False	True	False	Fals
viewed	False	False	False	False	False	False	True	Fals
condition	False	False	False	False	False	False	False	Tru
grade	False	False	True	False	False	False	False	Fals
sqft_above	False	False	True	False	False	False	False	Fals
yr_built	False	False	False	False	False	False	False	Fals
Renovated_in_2000s	False	False	False	False	False	False	False	Fals
sqft_living15	False	False	True	False	False	False	False	Fals
sqft_lot15	False	False	False	True	False	False	False	Fals
year_sold	False	False	False	False	False	False	False	Fals
basement	False	False	False	False	False	False	False	Fals

In [41]:
```
# save absolute value of correlation matrix as a data frame
# converts all values to absolute value
# stacks the row:column pairs into a multindex
# reset the index to set the multindex to seperate columns
# sort values. 0 is the column automatically generated by the stacking

df=df_features.corr().abs().stack().reset_index().sort_values(0, ascending=F

# zip the variable name columns (Which were only named level_0 and level_1 b
df['pairs'] = list(zip(df.level_0, df.level_1))

# set index to pairs
df.set_index(['pairs'], inplace = True)

#drop level columns
df.drop(columns=['level_1', 'level_0'], inplace = True)

# rename correlation column as cc rather than 0
df.columns = ['cc']

# drop duplicates. This could be dangerous if you have variables perfectly c
df.drop_duplicates(inplace=True)
```

In [42]:
```
df[(df.cc>.7) & (df.cc <1)]
```

Out[42]:

pairs	cc
(sqft_living, sqft_above)	0.876448
(grade, sqft_living)	0.762779
(sqft_living, sqft_living15)	0.756402
(sqft_above, grade)	0.756073
(bathrooms, sqft_living)	0.755758
(sqft_living15, sqft_above)	0.731767
(sqft_lot, sqft_lot15)	0.718204
(grade, sqft_living15)	0.713867

- Remove: sqft_above,sqft_living15, sqft_lot15
- Keeping: grade, sqft_living, bathrooms
    - intuitively # of bathrooms, grade, and sqft_living should be important for house pricing
    - we can choose to remove these later if it is necessary for modeling

In [43]:
```
# Let's go ahead and drop those columns from our original dataframe
df_house = df_house.drop(['sqft_above', 'sqft_living15', 'sqft_lot15'], axi
# Let's also drop them from our features
df_features_clean = df_features.drop(['sqft_above', 'sqft_living15', 'sqft_
```

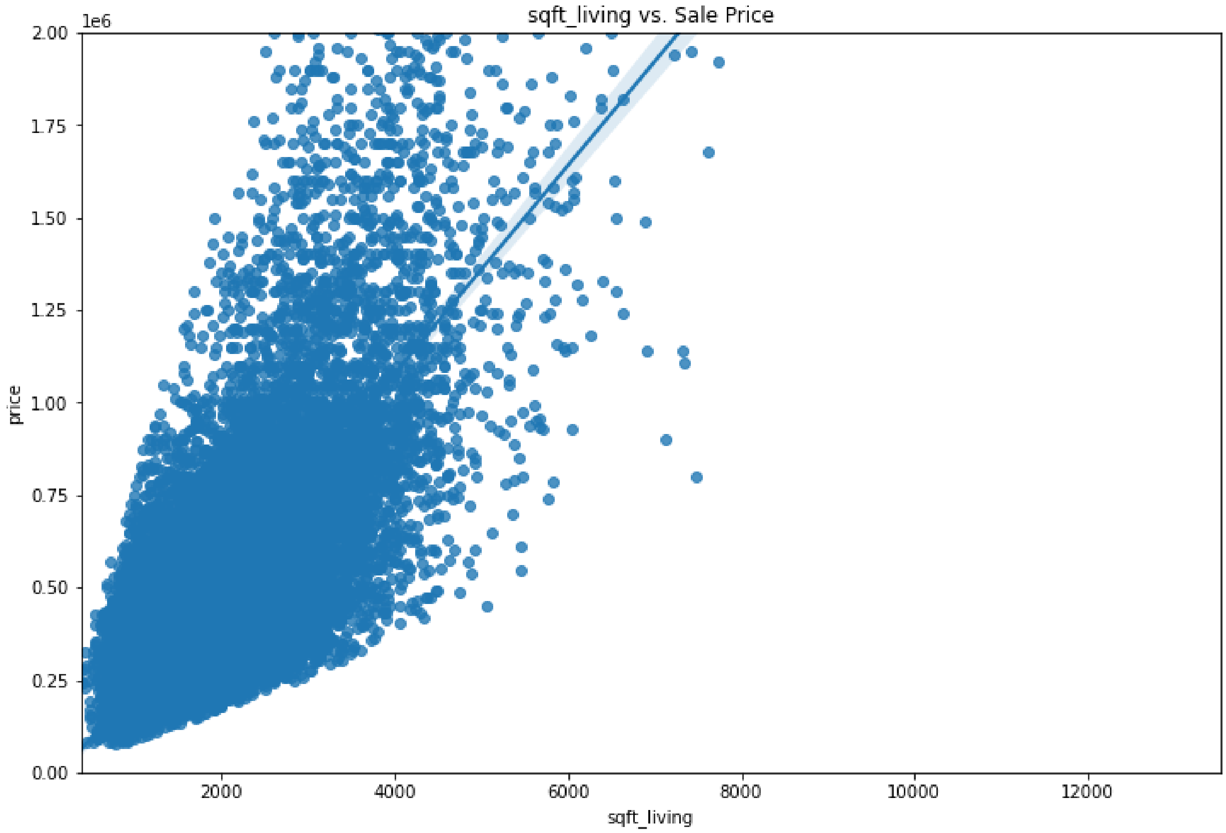
Continuous and Categorical

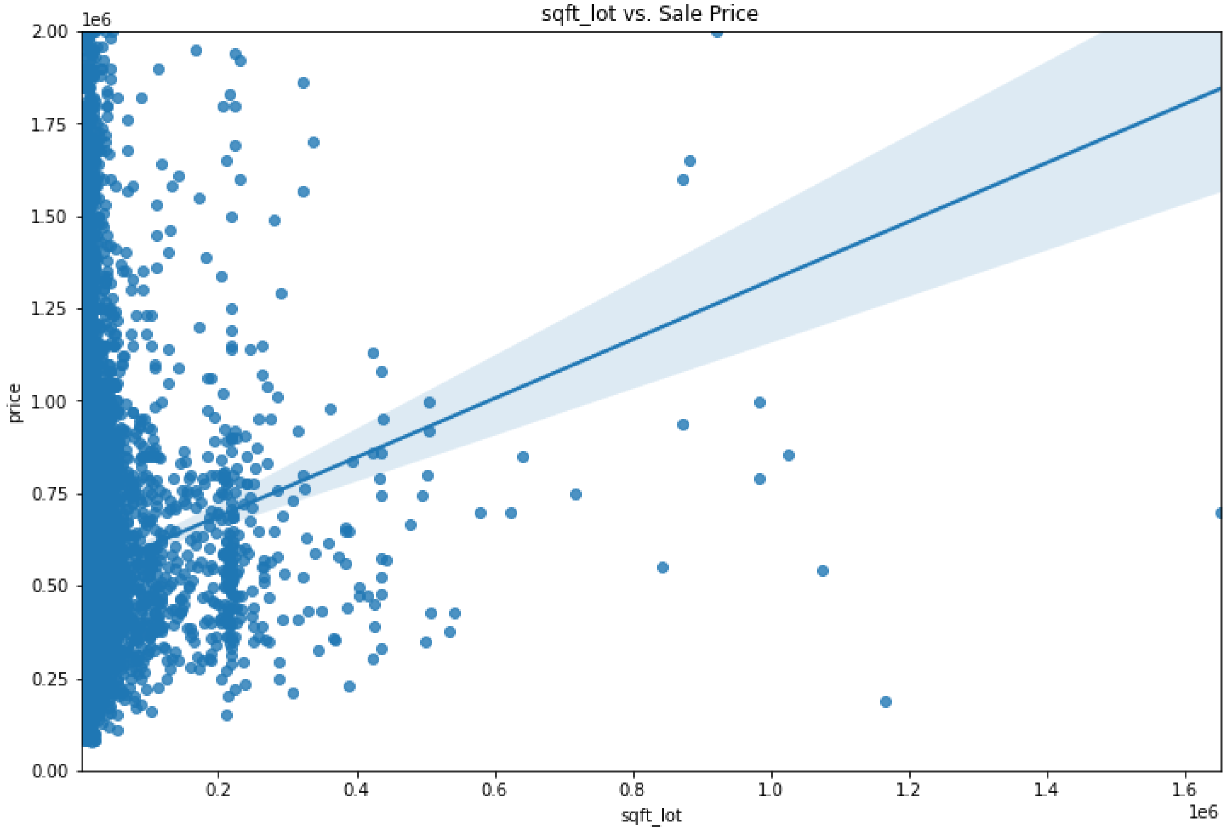
In [44]:
```
df_features_clean.columns
```

Out[44]:
```
Index(['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors',
       'waterfront', 'viewed', 'condition', 'grade', 'yr_built',
       'Renovated_in_2000s', 'year_sold', 'season_sold', 'basement'],
      dtype='object')
```

In [45]:
```
continuous = ['sqft_living', 'sqft_lot']
```

In [46]:
```
# price vs continuous variables
for variable in continuous:
    ax, figure = plt.subplots(1,1,figsize=(12,8))
    plt.ylim(0,2000000)
    sns.regplot(x=variable, y='price', data=df_house)
    plt.title("{} vs. Sale Price".format(variable))
```





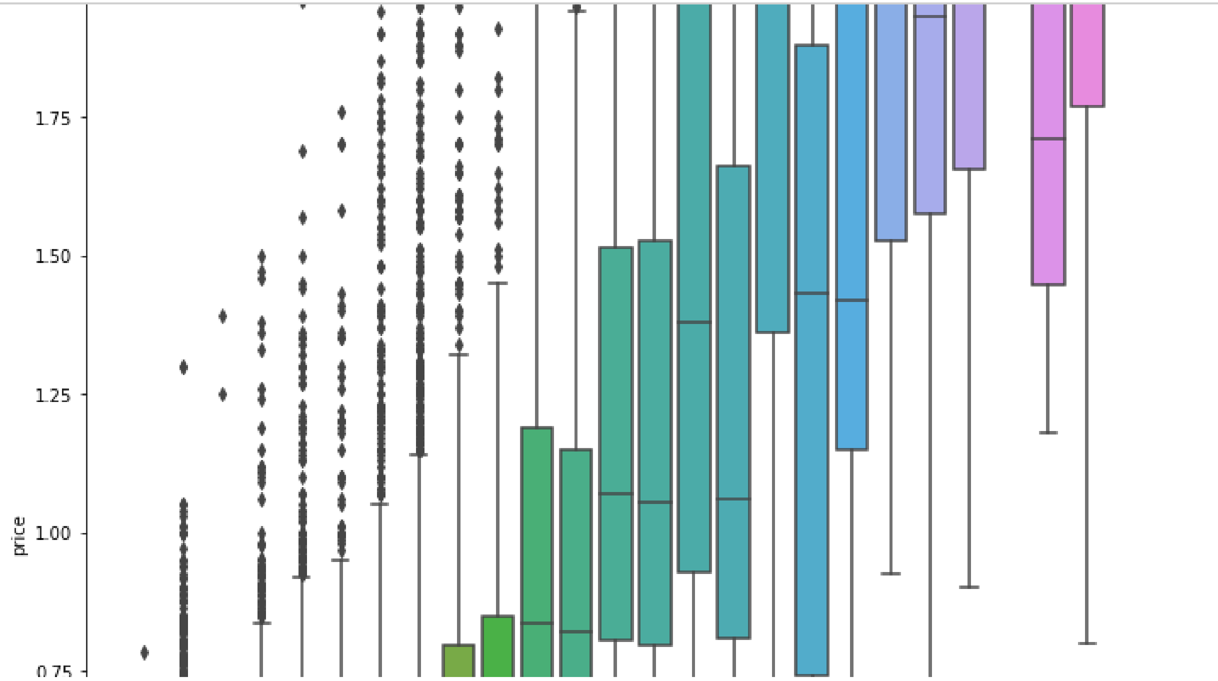
In [47]:
```
# ***should sqft lot be binned and changed to categorical?
```

In [48]:
```
# assign categorical variables to a list
categorical = ['bedrooms', 'bathrooms', 'floors', 'waterfront', 'viewed', '
                       'yr_built', 'Renovated_in_2000s', 'year_sold','sea
                       'basement']
```

In [49]:
```
#Check that we included all features:
len(categorical) + len(continuous) == len(df_features_clean.columns)
```

Out[49]:
```
True
```

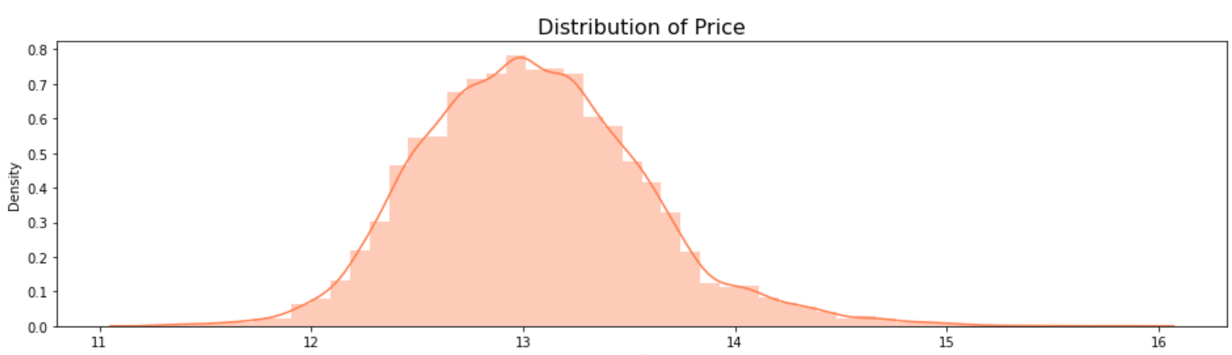
In [50]:
```
# price vs categorical variables
for variable in categorical:
    ax, figure = plt.subplots(1,1,figsize=(12,12))
    plt.ylim(0,2000000)
    sns.boxplot(x=variable, y='price', data=df_house)
    plt.title("{} vs. Sale Price".format(variable))
```



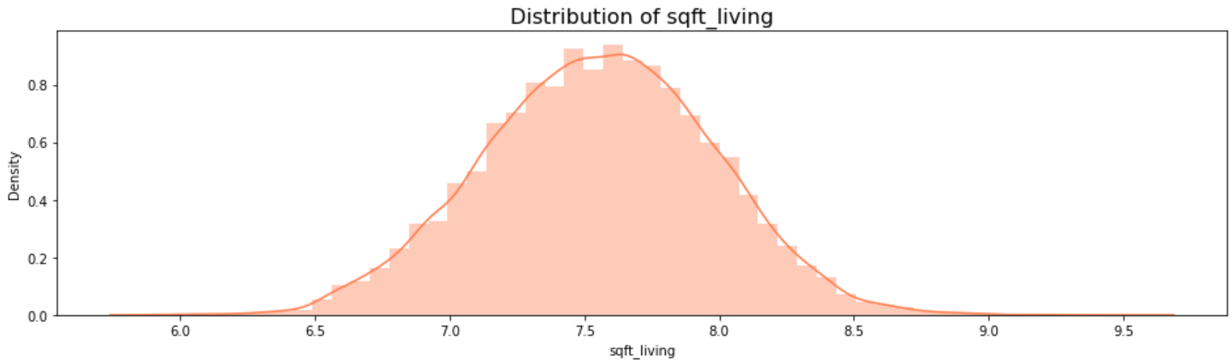
- yr_built is hard to interpret
- year_sold difference in price visually looks negligible
- season_sold Spring/Summer do have slightly higher prices than Fall/Winter

Log Transformations

In [51]:
```
non_normal = ['price']
for feat in non_normal:
    df_house[feat] = df_house[feat].map(lambda x: np.log(x))
plt.figure(figsize = (16, 4))
sns.distplot(a = df_house["price"], color = "#FF7F50")
plt.title("Distribution of Price", fontsize=16);
```



In [52]:
```
non_normal = ['sqft_living']
for feat in non_normal:
    df_house[feat] = df_house[feat].map(lambda x: np.log(x))
plt.figure(figsize = (16, 4))
sns.distplot(a = df_house["sqft_living"], color = "#FF7F50")
plt.title("Distribution of sqft_living", fontsize=16);
```



one-hot encode categorical variables

In [53]:
```
# Convert category variables data type
df_house[categorical] = df_house[categorical].astype('category')
```

In [54]:
```
# # one hot encode categoricals
df_house_ohe = pd.get_dummies(df_house[categorical], drop_first=True)
df_house_ohe.head()
```

Out[54]:

	bedrooms_2	bedrooms_3	bedrooms_4	bedrooms_5	bedrooms_6	bedrooms_7	bedrooms_8	be
0	0	1	0	0	0	0	0	
1	0	1	0	0	0	0	0	
2	1	0	0	0	0	0	0	
3	0	0	1	0	0	0	0	
4	0	1	0	0	0	0	0	

5 rows × 181 columns

In [55]:
```
# # Concatenate OHE columns with original dataframe, dropping OHE original
df_house_comb = pd.concat([df_house, df_house_ohe], axis=1)
df_house_comb.drop(['bedrooms', 'bathrooms', 'floors', 'waterfront', 'viewe
                    'yr_built', 'Renovated_in_2000s', 'year_sold','se
                    'basement'], axis=1, inplace=True)
df_house_comb.head()
```

Out[55]:

	price	sqft_living	sqft_lot	bedrooms_2	bedrooms_3	bedrooms_4	bedrooms_5	bedrooms_6
0	12.309982	7.073270	5650	0	1	0	0	0
1	13.195614	7.851661	7242	0	1	0	0	0
2	12.100712	6.646391	10000	1	0	0	0	0
3	13.311329	7.580700	5000	0	0	1	0	0
4	13.142166	7.426549	8080	0	1	0	0	0

5 rows × 184 columns

In [56]:
```
df_house.head()
```

Out[56]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	viewed	condition	grac
0	12.309982	3	1.00	7.073270	5650	1.0	0.0	0	3	
1	13.195614	3	2.25	7.851661	7242	2.0	0.0	0	3	
2	12.100712	2	1.00	6.646391	10000	1.0	0.0	0	3	
3	13.311329	4	3.00	7.580700	5000	1.0	0.0	0	5	
4	13.142166	3	2.00	7.426549	8080	1.0	0.0	0	3	

In [57]:
```
df_house.head()
```

Out[57]:

	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	viewed	condition	grac
0	12.309982	3	1.00	7.073270	5650	1.0	0.0	0	3	
1	13.195614	3	2.25	7.851661	7242	2.0	0.0	0	3	
2	12.100712	2	1.00	6.646391	10000	1.0	0.0	0	3	
3	13.311329	4	3.00	7.580700	5000	1.0	0.0	0	5	
4	13.142166	3	2.00	7.426549	8080	1.0	0.0	0	3	

In [58]:
```
df_house.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 15 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   price               21597 non-null  float64
 1   bedrooms            21597 non-null  category
 2   bathrooms           21597 non-null  category
 3   sqft_living         21597 non-null  float64
 4   sqft_lot            21597 non-null  int64
 5   floors              21597 non-null  category
 6   waterfront          21597 non-null  category
 7   viewed              21597 non-null  category
 8   condition           21597 non-null  category
 9   grade               21597 non-null  category
 10  yr_built            21597 non-null  category
 11  Renovated_in_2000s  21597 non-null  category
 12  year_sold           21597 non-null  category
 13  season_sold         21597 non-null  category
 14  basement            21597 non-null  category
dtypes: category(12), float64(2), int64(1)
memory usage: 768.6 KB
```

Normalize numeric variables?

Feature Scaling and Normalization?

split data test and train

In [59]:
```
# Divide dataset into X predictors and y target
x = df_house_comb.drop(['price'], axis=1)
y = df_house_comb[['price']]
```

In [60]:
```
# Split the data into 80% training and 20% test sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, sh
```

In [61]:
```
# Validate rows in splits look as expected
print(len(x_train), len(x_test), len(y_train), len(y_test))
```

```
17277 4320 17277 4320
```

Model 1

Linear Regression

In [62]:
```
linear_model = LinearRegression()
# Train the model on training data
linear_model.fit(x_train, y_train)
```

Out[62]:
```
LinearRegression()
```

Linear model prediction

In [63]:
```
# Predict on test data
predictions = linear_model.predict(x_test)
mse = mean_squared_error(y_test, predictions)
rmse = np.sqrt(mse)
print(rmse)
```

```
0.3055057066503532
```

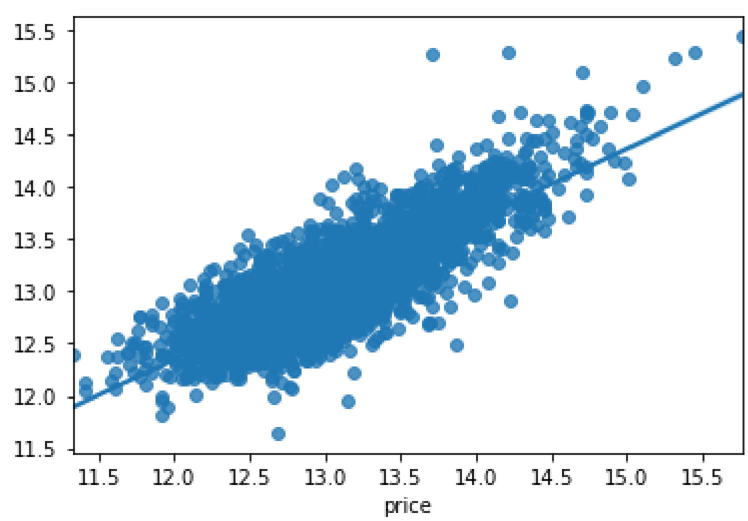
In [64]:
```
# Get how well it performed
mae_linear = mean_absolute_error(y_test, predictions)

print("Linear: {:,}".format(mae_linear))
```

```
Linear: 0.24102988548964943
```

In [65]:
```
sns.regplot(x=y_test, y=predictions)
```

Out[65]:
```
<AxesSubplot:xlabel='price'>
```



In [66]:
```
y_hat_train = linear_model.predict(x_train)
y_hat_test = linear_model.predict(x_test)
```

In [67]:
```
train_residuals = y_hat_train - y_train
test_residuals = y_hat_test - y_test
```

In [68]:
```
train_mse = mean_squared_error(y_train, y_hat_train)
test_mse = mean_squared_error(y_test, y_hat_test)
print('Train Mean Squarred Error:', train_mse)
print('Test Mean Squarred Error:', test_mse)
```

```
Train Mean Squarred Error: 0.089275685470373
Test Mean Squarred Error: 0.09333373679593167
```

In [69]:
```
print('R^2 train: %.3f, test: %.3f' % (r2_score(y_train, y_hat_train),
                                r2_score(y_test, y_hat_test)))
```

```
R^2 train: 0.679, test: 0.657
```

Model 2

Decision Tree Regressor

In [70]:
```
tree_model = DecisionTreeRegressor()

# Train the model on training data
tree_model.fit(x_train, y_train)

# Predict on test data
predictions = tree_model.predict(x_test)
```

Tree model prediction

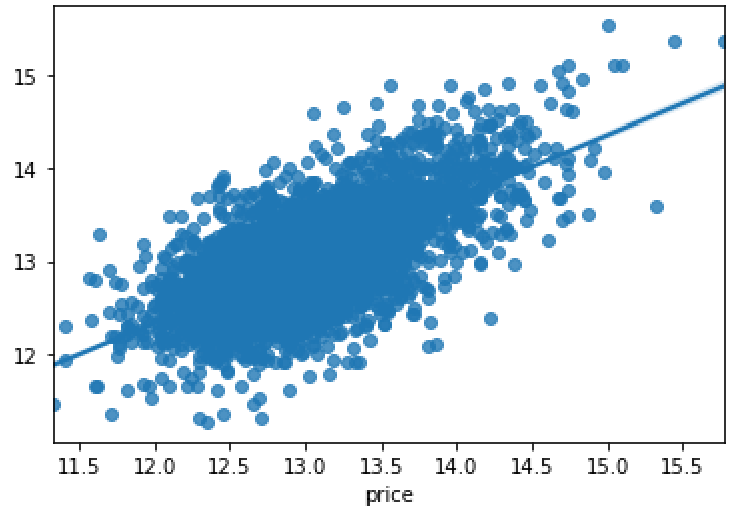
In [71]:
```
# Get how well it performed
mae_tree = mean_absolute_error(y_test, predictions)

print("Tree: {:,}".format(mae_tree))
```

```
Tree: 0.321062439428164
```

In [72]:
```
sns.regplot(x=y_test, y=predictions)
```

Out[72]: <AxesSubplot:xlabel='price'>



In [73]:
```
y_hat_train = tree_model.predict(x_train)
y_hat_test = tree_model.predict(x_test)
```

In [74]:
```
#train_residuals = y_hat_train - y_train
#test_residuals = y_hat_test - y_test
```

In [75]:
```
train_mse = mean_squared_error(y_train, y_hat_train)
test_mse = mean_squared_error(y_test, y_hat_test)
print('Train Mean Squarred Error:', train_mse)
print('Test Mean Squarred Error:', test_mse)
```

```
Train Mean Squarred Error: 2.1355561816031956e-06
Test Mean Squarred Error: 0.17756654885902834
```

In [76]:
```
print('R^2 train: %.3f, test: %.3f' % (r2_score(y_train, y_hat_train),
                r2_score(y_test, y_hat_test)))
```

```
R^2 train: 1.000, test: 0.348
```

Model 3

Random Forest Regressor

In [77]:
```
rf_model = RandomForestRegressor()

# Train the model on training data
rf_model.fit(x_train, y_train)

# Predict on test data
predictions = rf_model.predict(x_test)
```

Random Forest model prediction

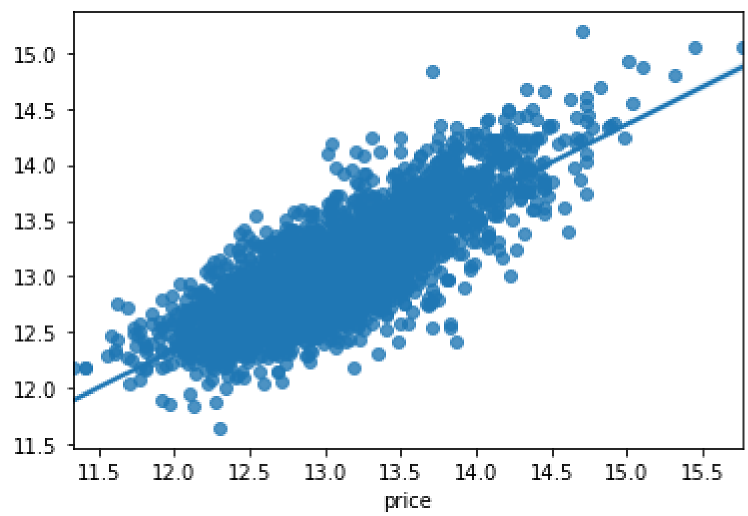
In [78]:
```
# Get how well it performed
mae_rf = mean_absolute_error(y_test, predictions)

print("Random Forest: {:,}".format(mae_rf))
```

```
Random Forest: 0.24087587149358533
```

In [79]:
```
sns.regplot(x=y_test, y=predictions)
```

Out[79]:
```
<AxesSubplot:xlabel='price'>
```



In [80]:
```
y_hat_train = rf_model.predict(x_train)
y_hat_test = rf_model.predict(x_test)
```

In [81]:
```
train_mse = mean_squared_error(y_train, y_hat_train)
test_mse = mean_squared_error(y_test, y_hat_test)
print('Train Mean Squared Error:', train_mse)
print('Test Mean Squared Error:', test_mse)
```

```
Train Mean Squared Error: 0.013291895866881082
Test Mean Squared Error: 0.09566589214810833
```

In [82]:
```
print('R^2 train: %.3f, test: %.3f' % (r2_score(y_train, y_hat_train),
                r2_score(y_test, y_hat_test)))
```

```
R^2 train: 0.952, test: 0.649
```

Model 4

OLS

In [83]:
```
x_train1 = sm.add_constant(x_train)
```

In [84]:
```
x_test1 = sm.add_constant(x_test)
```

In [85]:
```
result = sm.OLS(y_train,x_train1).fit()
```

OLS model prediction

In [86]:
```
print(result.rsquared, result.rsquared_adj)
```

```
0.6793940271574763 0.6759994860001498
```

In [87]:
```
result.summary()
```

floors_2.5	0.0356	0.028	1.259	0.208	-0.020	0.091
floors_3.0	0.1721	0.017	10.392	0.000	0.140	0.205
floors_3.5	0.1691	0.115	1.467	0.142	-0.057	0.395
waterfront_1.0	0.4584	0.029	15.735	0.000	0.401	0.516
viewed_1	0.1154	0.009	13.349	0.000	0.098	0.132
condition_2	0.0268	0.075	0.360	0.719	-0.119	0.173
condition_3	0.1718	0.070	2.445	0.014	0.034	0.310
condition_4	0.2104	0.070	2.994	0.003	0.073	0.348
condition_5	0.2689	0.071	3.808	0.000	0.131	0.407
grade_1	-0.3539	0.311	-1.137	0.256	-0.964	0.256
grade_2	-0.1957	0.306	-0.639	0.523	-0.796	0.404
grade_3	-0.0261	0.306	-0.085	0.932	-0.626	0.573
grade_4	0.2414	0.306	0.789	0.430	-0.358	0.841

Model 1 - Our best model so far is Model 1 our 'linear_model'

k-fold cross validation

In [88]:
```
cross_validation_5 = KFold(5, shuffle=True)
cross_validation_10 = KFold(5, shuffle=True)
cross_validation_20 = KFold(5, shuffle=True)
```

In [89]:
```
# set scoring argument to 'neg_mean_squared_error'. This negates mean squar
cv_5_results  = np.mean(cross_val_score(linear_model, x, y, cv=cross_valida
cv_10_results = np.mean(cross_val_score(linear_model, x, y, cv=cross_valida
cv_20_results = np.mean(cross_val_score(linear_model, x, y, cv=cross_valida
```

In [90]:
```
cv_5_results
```

Out[90]: -0.09199475508368307

In [91]:
```
cv_10_results
```

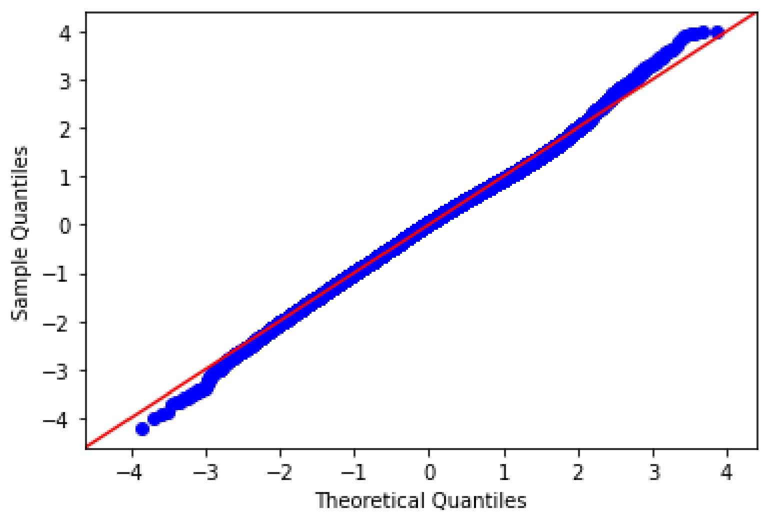
Out[91]: -0.09203311551120896

In [92]:
```
cv_20_results
```

Out[92]: -0.0919105158759422

Normality

In [93]:
```
fig = sm.graphics.qqplot(result.resid, dist=stats.norm, line='45', fit=True
```

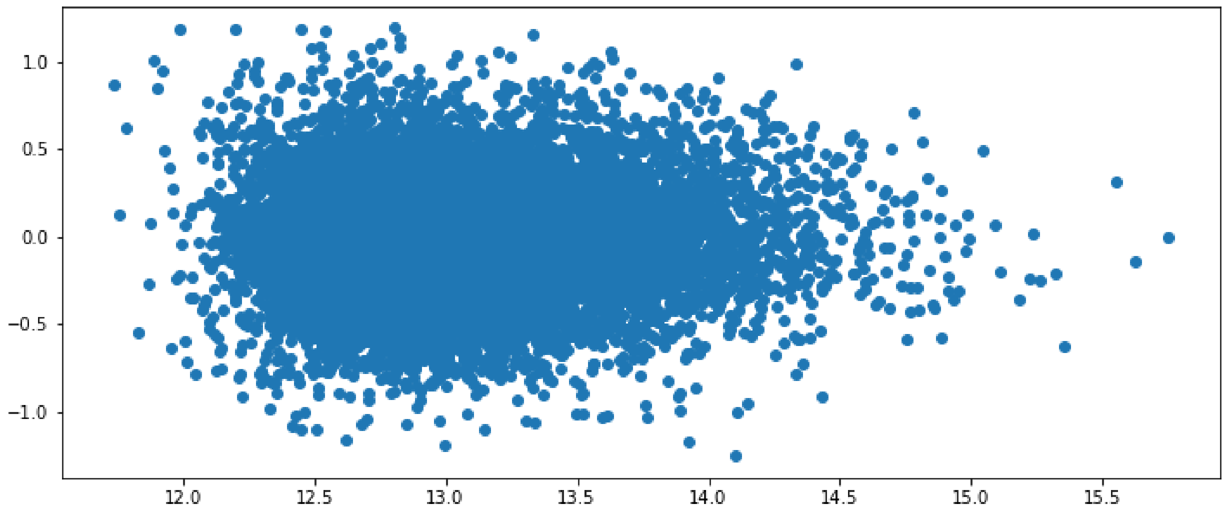


Normality looks good

Homoskedasticity

In [94]:
```
plt.figure(figsize=(12,5))
plt.scatter(result.predict(x_train1), result.resid)
```

Out[94]: <matplotlib.collections.PathCollection at 0x7fd1e772b100>



Homoskedasticity looks ok

In [95]:
```
separate the predictors and the target
= x_train
= y_train

initialize list to add r_adj values
_squared_adj_list = []

or i in range(len(X.columns)):
  print("Pick {} features".format(i+1))
  # initialize linear regression object
  linear_model = LinearRegression()
  selector = RFE(linear_model, n_features_to_select=(i+1))

  # convert y to 1d np array to prevent DataConversionWarning
  selector = selector.fit(X, y.values.ravel())

  # create list of selected columns
  selected_columns = X.columns[selector.support_ ]

  # fit linear regression model
  linear_model.fit(X[selected_columns],y)
  LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=

  # predict y_hat values
  yhat = linear_model.predict(X[selected_columns])

  # calculate performance metrics
  SS_Residual = np.sum((y-yhat)**2) # calculate SS residual
  SS_Total = np.sum((y-np.mean(y))**2) # calculate SS total
  r_squared = 1 - (float(SS_Residual))/SS_Total # calculate r_squared
  adjusted_r_squared = 1 - (1-r_squared)*(len(y)-1)/(len(y)-X[selected_colu
  print(adjusted_r_squared)
  r_squared_adj_list.append(adjusted_r_squared)
```

```
dtype: float64
Pick 26 features
price    0.574541
dtype: float64
Pick 27 features

price    0.574821
dtype: float64
Pick 28 features
price    0.574803
dtype: float64
Pick 29 features
price    0.575963
dtype: float64
Pick 30 features
price    0.576158
dtype: float64
Pick 31 features
price    0.57618
dtype: float64
```

In [96]:
```
print(len(y2))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-96-0ae178023f44> in <module>
----> 1 print(len(y2))

NameError: name 'y2' is not defined
```

In [ ]:
```
# plot r_squared adjusted against # of features
fig, ax = plt.subplots(1,1,figsize=(15,10))
x=np.array(range(1,184))
y2=r_squared_adj_list
plt.scatter(x,y2)
plt.title("Number of Features vs. R-Squared Adjusted")
plt.xlabel("Number of Features")
plt.ylabel("R-Squared Adjusted")
```

Summary

Our final model has an r-squared-adjusted value of 0.656. Our model performs about the same with the train and test data so we don't seem to have an overfitting issue. Normality and homoskedasticity looked good in our visualizations.

Our RMSE is 127k USD! This is not very confidence inspiring considering that the median home value of our analyzed dataset is 472k USD.

Expansion of data and more development is needed to make our model viable for use as a pricing model

Recommendations

- Sell houses in the Spring and Summer. More houses are purchased during this time and they tend to sell for higher prices.
- Expand data set beyond houses sold in 2014 and 2015. Having more current data would make the model more accurate.

In [ ]: