0 Analysis of House Sales in a King County

Please fill out:

- Student name: **Abeselom Fanta**
- Student pace: **Flex**
- Scheduled project review date/time: **November 22, 2021 5:00 pm EST**
- Instructor name: **Abhineet Kulkarni**
- Blog post URL: https://medium.com/@afanta/bayesian-linear-regression-dea1b7e7cf48 (https://medium.com/@afanta/bayesian-linear-regression-dea1b7e7cf48)

Table of Contents

1 Overview

This project focuses on housing prices in the northwestern part of Washington state. The data collected shows the prices of with other valuable information, such as date of construction, if the houses are renovated, the areas of square footage of different parts of the house and more.

Based on the information provided, it is important to develop a model that can predict house prices not included in the dataset. The aspiration of this project is also to identify key metrics that are crucial to determine sale price.

2 Business Understanding

As potential house buyer, our clients need to make an informed decision when they decide to buy a house in King's County (https://info.kingcounty.gov). Therefore, our firm can set the price at which available houses could be sold at a marginally profitable value. For such analysis, a data was collected that shows the selling price of houses in 2014 and 2015. The data is analyzed in this notebook. Supplementary information such as grades and conditions assigned to a particular domicile can be accessed at King County Assessor Website (https://info.kingcounty.gov/assessor/esales/Glossary.aspx?type=r). This project will try to address the following question:

- What are the primary factors for house pricing?
- Do zip codes, grades and other qualitative factors in the county determine house prices?
- Are there unforeseen factors that could help predict housing price?
- What is the best linear model to predict house prices in the future?
- What are the additional information required to help make better prediction?
- What are the recommendations based on the model generated?

3 Data Understanding

Data for this project was collected from King's County official website (https://info.kingcounty.gov). The data contains rich information. However, we need to further inspect the data to understand what is available and how to make our best guess for missing data.

Let's start by importing libraries that will be used in this project.

In [1]:
```
####### Uncomment and install these packaged before
####### running an interactive fitting plot
# ! pip install dash jupyter_dash dill
```

In [2]:

```
#Import libraries to be used in this project
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec
import seaborn as sns
import numpy as np
import scipy.stats as stats
from scipy.spatial.distance import cdist
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_validate, train_test_split
from sklearn.metrics import r2_score, mean_squared_error
from sklearn.impute import SimpleImputer
import statsmodels.api as sm
from statsmodels.formula.api import ols
from statsmodels.stats.outliers_influence import variance_inflation_factor
import plotly.express as px
import plotly.graph_objects as go
from dash import dcc
from dash import html
from dash.dependencies import Input, Output
from jupyter_dash import JupyterDash
from calendar import month_abbr
from itertools import combinations
import dill as pickle
from os.path import exists
import warnings
warnings.filterwarnings("ignore")

sns.set_style('darkgrid')
```

The dataset can be loaded with `pandas` as follows:

In [3]:
```
# Let's import and explore the data
df = pd.read_csv('data/kc_house_data.csv')

# Make a copy of the master file
houses_df = df.copy()

# Display columns and first five rows
print(houses_df.columns)
houses_df.head()
```

```
Index(['id', 'date', 'price', 'bedrooms', 'bathrooms', 'sqft_living',
       'sqft_lot', 'floors', 'waterfront', 'view', 'condition', 'grade',
       'sqft_above', 'sqft_basement', 'yr_built', 'yr_renovated', 'zipcode',
       'lat', 'long', 'sqft_living15', 'sqft_lot15'],
      dtype='object')
```

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	NO
2	5631500400	2/25/2015	180000.0	2	1.00	770	10000	1.0	NO
3	2487200875	12/9/2014	604000.0	4	3.00	1960	5000	1.0	NO
4	1954400510	2/18/2015	510000.0	3	2.00	1680	8080	1.0	NO

5 rows × 21 columns

To understand the geographical location of the houses and their relative prices, an interactive map plot is generated using the code below.

In [4]:

```
# Populate 'prices_map' with dollar sign and commas for mapping label
houses_df['price_map'] = houses_df['price'].apply(lambda x: '${:,.2f}'.forma

# Call plotly.express to plot an interactive map of the houses
fig = px.scatter_mapbox(houses_df,
                        lat="lat", lon="long",
                        color = "price",
                        hover_name = "price_map",
                        hover_data = ["bedrooms", "bathrooms","zipcode"],
                        zoom = 8,
                        width = 900, height = 700,
                        color_continuous_scale=px.colors.sequential.RdBu,
                        title = 'King County House Sales')

# Set layout to 'open-street-map'. Other mapbox styles require API key
fig.update_layout(mapbox_style="open-street-map")

# Set scroll to zoom function to false. Use the UI to zoom instead
fig.show(config={'scrollZoom': False})

# Remove 'price_map' column
houses_df.drop('price_map',axis=1,inplace=True)
```

Based on the map shown above alone, we can surmise that there is a big variance how prices. Although most houses remain below $1 million, few anomalies exist. We will explore the data further more.

4 Data Preparation

Prior to analysis, it is important to know the nature of the data. `pandas.info()` can show us the data types in each column.

In [5]:
```
1 # Find the data types of each column
2 houses_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 21 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   id             21597 non-null  int64  
 1   date           21597 non-null  object 
 2   price          21597 non-null  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: 3.5+ MB
```

At a first glance, we can see that the column `date` is an object that can be converted to `date`

type. `grade` contains numerical and text information. `view` and `condition` also have object data type. `sqft_basement` should be numeric but has an `object` data type.

4.1 Remove Columns

The `id` column helps as an identifier and has no relevance to the analysis. `date` is changed to `datetime` formate.

In [6]:
```
# Drop id
houses_df.drop('id',axis=1,inplace=True)

# Change date to datetime
houses_df.date = pd.to_datetime(houses_df.date, infer_datetime_format=True)
```

4.2 Data Cleaning

In this section, the data is going to cleaned. The three important data cleaning used in this project are type casting, imputing `nan` or `null` values, and convert some categorical variables to numeric.

4.2.1 Type Casting

The column `grade` has both descriptive and numeric values. Although the descriptive information can be helpful, numeric values are more valuable. The following code splits the string in the column and takes the numeric grade value.

In [7]:
```
# Change qualitative grade to quantitative
houses_df.grade = houses_df.grade.apply(lambda x: int(x.split()[0]))

# Check if all changed values are numbers
sorted(houses_df.grade.unique())
```

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

The column `sqft_basement` has some characters. These can be replaced by creating a custom function that changed non-numeric character to `numpy.nan` .

In [8]:
```
# Convert 'sqft_basement' to float and np.nan
def to_float(feature):
    try:
        return float(feature)
    except:
        return np.nan

# Apply 'to_float' function to 'sqft_basement'
houses_df.sqft_basement = houses_df.sqft_basement.apply(to_float)

# Check the datatype of the column
houses_df.sqft_basement.dtype
```

Out[8]:
```
dtype('float64')
```

4.2.2 Impute `NaN` or `Null` values

Columns with numeric values that contain `NaN` can me imputed using `sklearn.impute` utility. But it could become challenging if categorical columns have `NaN` values. But first, let's find out which columns have `NaN` or `Null` values.

In [9]:
```
# Identify rows and columns with NaN values
houses_na = houses_df.loc[houses_df.isnull().any(axis=1),
                          houses_df.columns[houses_df.isnull().any()]].colum
houses_na
```

Out[9]:
```
Index(['waterfront', 'view', 'sqft_basement', 'yr_renovated'], dtype='object')
```

As can be seen, `sqft_basement` and `yr_renovated` can be numeric, hence can also be imputed.

For `waterfront` and `view` columns, instead of using imputer, we are going to make a best guess. The data provides latitude and longitudinal information. Therefore, we can use the coordinates to find the another house in the dataset that is closest to the houses in question. The logic is that, if the neighboring house has great views, so does that house with incomplete information. The same logic can also be applied the question if the house is at a waterfront.

In [10]:
```
# Find indices that that do not have NaN values
houses_len = np.arange(len(houses_df))

# Iterate over the columns with NaN values
for col in houses_na:

    # Check if the columns are strings
    if houses_df[col].dtype == object:

        # Isolate the lat and long of the NaN values from each column
        na_locs = houses_df.loc[houses_df[col].isnull(),['lat','long']]
        na_idx = list(na_locs.index)

        # Do the same for non-NaN values
        not_na_locs = houses_df.loc[houses_df[col].notnull(),['lat','long']]
        not_na_idx = list(not_na_locs.index)


        # Calculate the Euclidean distance of each house with NaN column
        # to those with houses without NaN
        dist_mat = cdist(not_na_locs.to_numpy(),na_locs.to_numpy(),
                         'euclidean')

        # Find the closes house to each NaN column house, i.e., min_dist
        closest_house = [not_na_idx[np.where(x==np.min(x))[0][0]]
                         for x in dist_mat.T]

        # Assign values to NaN column from a neighboring house
        houses_df.loc[na_idx,col] = houses_df.loc[closest_house,col].tolist(
```

Now that `view` and `waterfront` data have been imputed, let us see if the imputation has fully taken effect.

In [11]:
```
# Run the previous code to see if 'waterfront' and 'view' values have been i
houses_na = houses_df.loc[houses_df.isna().any(axis=1),
              houses_df.columns[houses_df.isna().any()]].columns
```

Great! Now, we can impute the numeric columns using `SimpleImputer` .

In [12]:
```
# Impute colums with NaN
# Use mode as a strategy
imp_mode = SimpleImputer(missing_values=np.nan,
                         strategy='most_frequent')

# Fit the imputer
imp_mode = imp_mode.fit(houses_df[houses_na])

# Replace values in 'houses_na' columns
houses_df[houses_na] = imp_mode.transform(houses_df[houses_na])
```

4.2.3 Convert Categorical Data to Numeric

Some of the categorical data provided have gradations (i.e. fair, poor, average, good and excellent). Obviously, these can be treated as categorical and can be encoded with one-hot-encoding. But the other alternative is to assign numerical values, which for some, like `waterfront` can be interpreted as one-hot-encoding.

In [13]:
```
# One-hot encode 'waterfront' column
# There are only two choices here. Therefore, we don't
# have to use pd.get_dummies
houses_df.loc[houses_df.waterfront == 'YES','waterfront'] = 1
houses_df.loc[houses_df.waterfront == 'NO','waterfront'] = 0
houses_df.waterfront = houses_df.waterfront.astype(int)
```

To see what kind of qualitative grading is used, the unique values in `view` and `condition` are explored.

In [14]:
```
# Find categorical columns
categoricals = list(houses_df.columns[houses_df.dtypes == object])

for cat in categoricals:
    print(cat,': ',houses_df[cat].unique())
```

```
view :  ['NONE' 'GOOD' 'EXCELLENT' 'AVERAGE' 'FAIR']
condition :  ['Average' 'Very Good' 'Good' 'Poor' 'Fair']
```

Clearly, the orders must be rearranged to reflect the qualities of view and condition. Result from label encoding can used to replace the respective columns.

In [15]:
```
# Find categorical columns
categoricals = list(houses_df.columns[houses_df.dtypes == object])

# Label encoding for view
views = ['NONE', 'FAIR', 'AVERAGE', 'GOOD', 'EXCELLENT']
for view in views:
    houses_df.loc[houses_df.view==view,'view'] = views.index(view)
houses_df.view=houses_df.view.astype(int)

# Label encoding for condition
conds = ['Poor', 'Fair', 'Average', 'Good', 'Very Good']
for cond in conds:
    houses_df.loc[houses_df.condition==cond,'condition'] = conds.index(cond)
houses_df.condition=houses_df.condition.astype(int)
```

The age of the houses can be computed from the `date` and `yr_built` data.

In [16]:
```
# Get the ages of the houses
houses_df['age'] = houses_df.date.dt.year - houses_df.yr_built

# Remove 'yr_built' column
houses_df.drop('yr_built', axis=1, inplace=True)
```

The same can be applied to duration from which the houses were renovated to the time they were sold.

In [17]:
```
# Get the ages of houses since renovation
# Assign the age by default
houses_df['age_renovated'] = 0

# Change those whose 'yr_renovated' value is not zero
idx = (houses_df.yr_renovated!=0)
houses_df.loc[idx,'age_renovated'] = houses_df.loc[idx,'date'].dt.year - hou

# Remove 'yr_renovated'
houses_df.drop('yr_renovated', axis=1, inplace=True)
```

Now, our data should be all numerical and can be passed to regression functions with no discernible problem.

In [18]:
```
houses_df.head()
```

Out[18]:

	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition
0	2014-10-13	221900.0	3	1.00	1180	5650	1.0	0	0	2
1	2014-12-09	538000.0	3	2.25	2570	7242	2.0	0	0	2
2	2015-02-25	180000.0	2	1.00	770	10000	1.0	0	0	2
3	2014-12-09	604000.0	4	3.00	1960	5000	1.0	0	0	4
4	2015-02-18	510000.0	3	2.00	1680	8080	1.0	0	0	2

In [19]:
```
1 houses_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21597 entries, 0 to 21596
Data columns (total 20 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   date           21597 non-null  datetime64[ns]
 1   price          21597 non-null  float64
 2   bedrooms       21597 non-null  int64
 3   bathrooms      21597 non-null  float64
 4   sqft_living    21597 non-null  int64
 5   sqft_lot       21597 non-null  int64
 6   floors         21597 non-null  float64
 7   waterfront     21597 non-null  int32
 8   view           21597 non-null  int32
 9   condition      21597 non-null  int32
 10  grade          21597 non-null  int64
 11  sqft_above     21597 non-null  int64
 12  sqft_basement  21597 non-null  float64
 13  zipcode        21597 non-null  int64
 14  lat            21597 non-null  float64
 15  long           21597 non-null  float64
 16  sqft_living15  21597 non-null  int64
 17  sqft_lot15     21597 non-null  int64
 18  age            21597 non-null  int64
 19  age_renovated  21597 non-null  float64
dtypes: datetime64[ns](1), float64(7), int32(3), int64(9)
memory usage: 3.0 MB
```

5 Modeling

Our data has now been cleaned and ready for modeling. In this section, the overall distribution of each column is shown. Moreover, we will check the linearity of each feature with price. Multicollinearity is also another important aspect that needs to be investigated. Pearson correlation is used in this section but later on, variance inflation factor will be used on cross-validation data if new features are added through polynomial model fitting or interactions between features.

Some continuous variation can have more statistical significance if binned by intervals. Therefore, we will apply binning to age and number of years between the last year of renovation and sale year. Multiple regression techniques will then be applied to the data. For this particular case, the regression models will eliminate features with large p-values.

The models are then evaluated on training and test datasets. Models with high coefficient of correlation will be selected for cross validation test with random splits. In the post-modeling section, normality and homoscedasticity section of the coefficient of correlation will be plotted. Multicollinearity on a the new fit will also be performed.

But first, let's investigate the distribution of each column.

In [20]:
```
# Plot histograms for each continuous feature and target
# Find the unique values in 'houses_df'
unique_vals = houses_df.nunique()

# Skip 'date' and 'price'
unique_vals.drop(index=['date','price'], inplace=True)

# Plot the distributions
fig, axes = plt.subplots(6,3,figsize=(15, 20))
for ax, col in zip(axes.flatten(),unique_vals.index):
    sns.histplot(houses_df[col],
                 color='tab:blue',ax=ax).set_title(col,fontsize=15)
    ax.set_ylabel(ax.get_ylabel(),fontsize=13)
    ax.set_xlabel('')

# Adjust vertical spacing between subplots
plt.subplots_adjust(hspace = 0.5);

# Save plot
plt.savefig('images/features_hist.png')
```

In the plots above we can see that all features that appear to measure area and number of rooms have a positively skewed normal distribution, which is unsuprising given that with adequate data being collected a skewed normal distribution is expected. Latitude and longitude indicate location. So, it is hard to say they have strong influence over price but there are some clustering effects (i.e., houses concentrating in specific locations). Given the plots though, we can safely say that the houses are concentrated in higher latitude (more closer to the poles than the equator) and lower longitudes (more on the west side than east). Zip code also show significant variability where more houses are sold on record in some zipcodes than others. Most of the properties are at the waterfront and have medium grade (around 7 or 8) with average condition.

5.1 Linearity Check

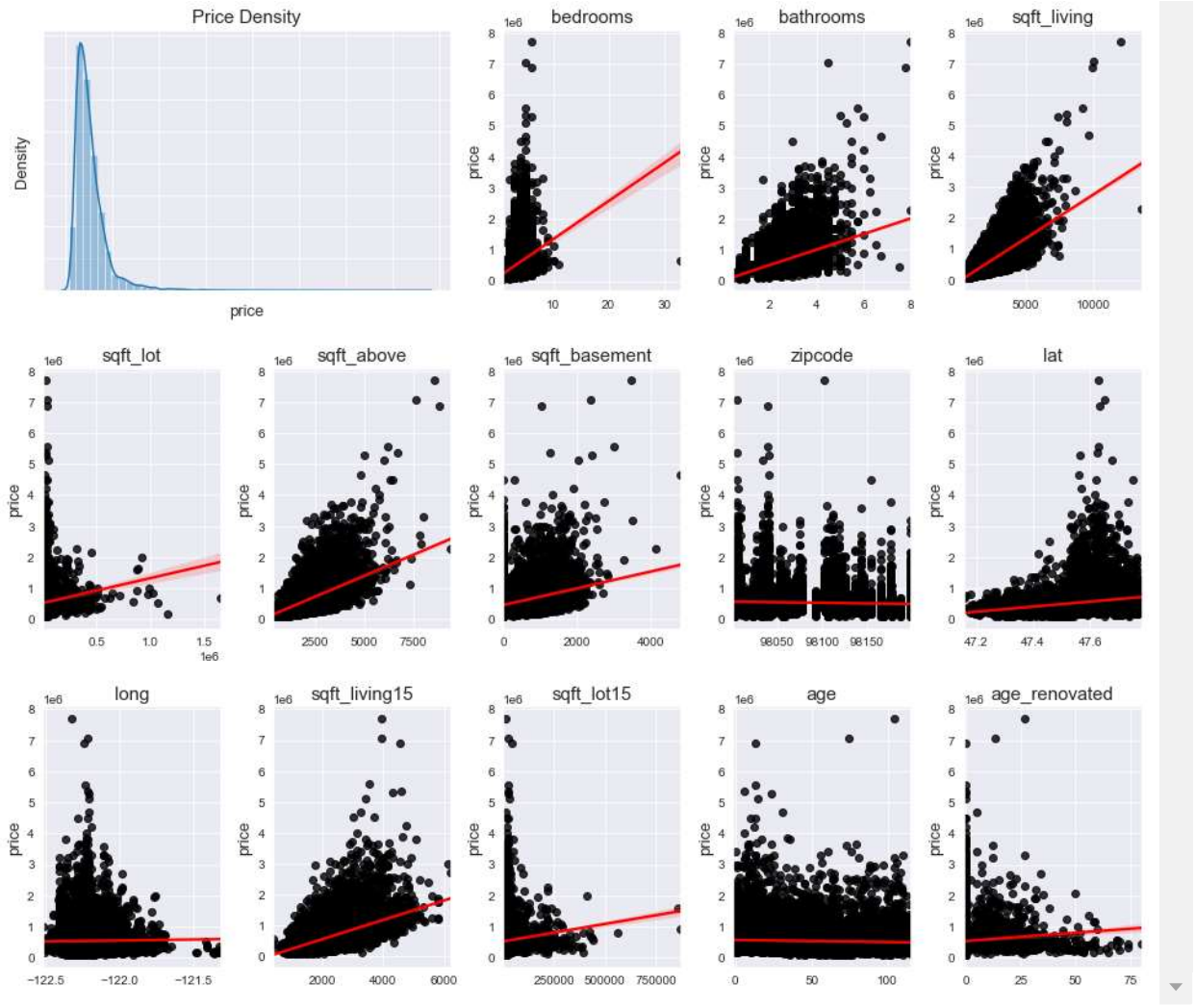
In this section, we will investigate the linearity of each feature. Most of the features have continuous values but both continuous and discrete features will be checked if they have strong relationship with price. However, this step is a cursory one to indicate what sort of methods to use in the analysis.

5.1.1 Continuous Features

A list of continuous features were identified, with unique values over 20. The approach can also inadvertently include features that are discrete but have more distinct values. The plot below shows the density distribution of prices along with its linearity with other features.

In [21]:

```
# Initialize plot
fig = plt.figure(figsize=(16, 14), constrained_layout=True)

# Create objects
gs = GridSpec(3, 5, figure=fig)


# Histogram plot of the house prices
ax_ = fig.add_subplot(gs[0,:2])
sns.distplot(houses_df['price'], color='tab:blue',
             ax=ax_).set_title('Price Density',fontsize=15)

# Set and adjust sizes of labels and ticks
ax_.set_ylabel(ax_.get_ylabel(),fontsize=13)
ax_.set_xlabel(ax_.get_xlabel(),fontsize=13)
ax_.set_xticklabels(ax_.get_xmajorticklabels(),fontsize=12);
ax_.set_yticklabels(ax_.get_ymajorticklabels(),fontsize=12);

# Identify continuous features to check for linearity
continuous = ['bedrooms','bathrooms','sqft_living','sqft_lot','sqft_above',
              'sqft_basement','zipcode','lat','long',
              'sqft_living15', 'sqft_lot15', 'age','age_renovated']

# Iterate through each feature
for i, feat in enumerate(continuous):
    ax = fig.add_subplot(gs[(i+2)//5,(i+2)%5])
    sns.regplot(feat, 'price', data=houses_df, ax=ax,
                scatter_kws={"color": "black"},
                line_kws={"color": "red"}).set_title(feat,fontsize=15);
    ax.set_xlabel('')
    ax.set_ylabel('price',fontsize=13)

# Adjust the width and height of the subplots
plt.subplots_adjust(hspace = 0.3)
plt.subplots_adjust(wspace = 0.3);

# Save plot
plt.savefig('images/linearity_check.png')
```



There is a lot to unpack from the plot above. First, `sqft_living` and `sqft_above` show strong relationship with price. This needs to be quantified with appropriate metrics, which will be performed later. From the plot, one can also see that there is a poor relation between `age` and `age_renovated` with `price`. Therefore, another strategy needs to be formulated to assess how age impacts sale price, where ages of houses are binned in groups as opposed to raw comparison. Another reason why `age_renovated` did a poor fit could be due to imputation to mode that was performed on the column earlier. The same effect can be observed with `sqft_basement` column.

One important observation is that `zipcode` does not have a good correlation with `price`. It's important to note that zip codes are not expected to have collinearity with price since some zip codes could have more expensive listing than others. Therefore, we need to remove zip code from continuous feature set and treat it as discrete instead.

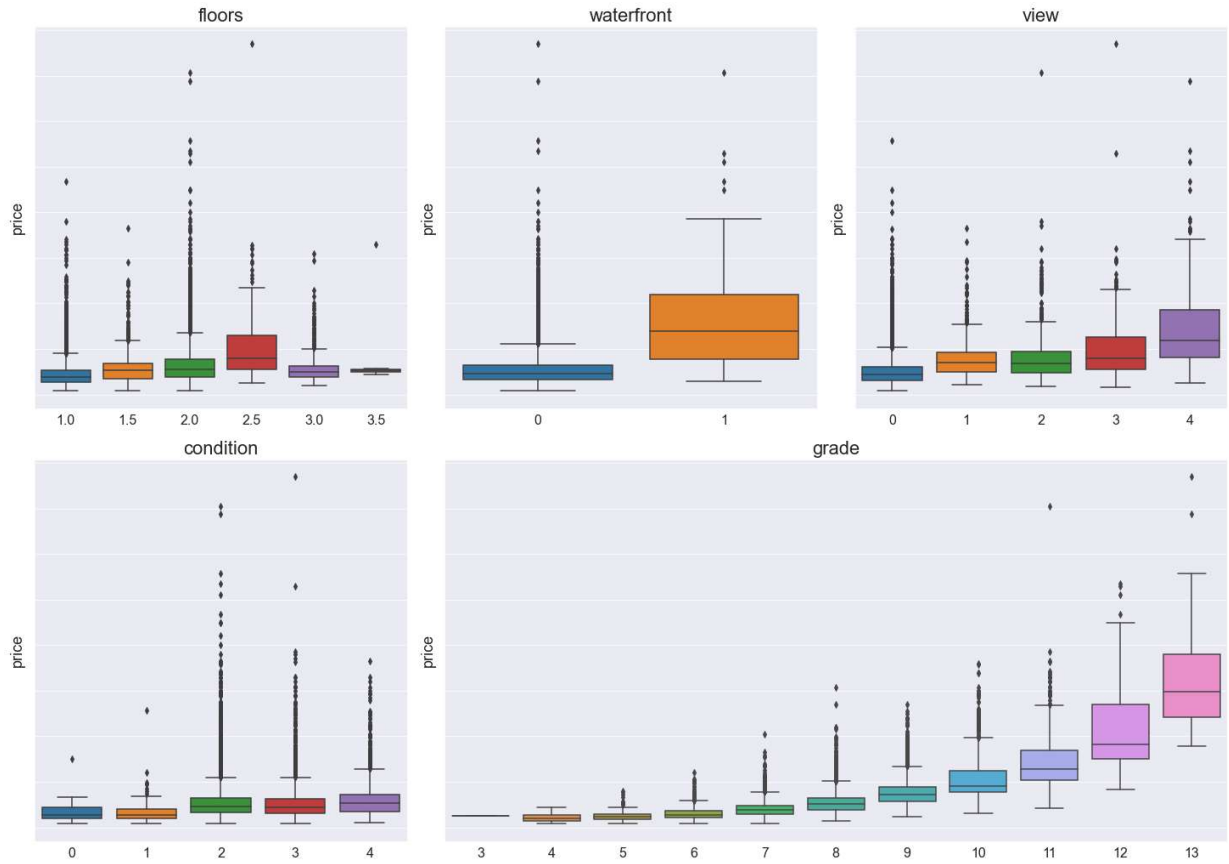
In [22]:
```
# Remove zipcode from continuous feature and treat it as discrete instead
continuous.remove('zipcode')
```

5.1.2 Discrete Features

Discrete features can help us understand how they mutually `price`. Here, `floors`, `view`,

`waterfront` , `condition` and `grade` are plotted below.

In [23]:
```
# Identify the descrete columns
discrete = ['floors','waterfront','view','condition','grade']

# Box plot of the descrete features with price
fig = plt.figure(figsize=(20, 14), constrained_layout=True)

# Create objects
gs = GridSpec(2, 3, figure=fig)

# Iterate through each feature
for i, feat in enumerate(discrete):
    if i<4:
        ax = fig.add_subplot(gs[i//3,i%3])
    else:
        ax = fig.add_subplot(gs[i//3,i%3:])
    sns.boxplot(x=houses_df[feat],y=houses_df.price,
                ax=ax).set_title(feat,fontsize=22);
    ax.set_xticklabels(ax.get_xmajorticklabels(),fontsize=16);
    ax.set_yticklabels(ax.get_ymajorticklabels(),fontsize=16);
    ax.set_xlabel('')
    ax.set_ylabel('price',fontsize=18)

# Save plot
plt.savefig('images/discrete_features.png')
```



As can be seen in the plot above, there are some correlations that are notable. Although most of the discrete functions appear to have incremental relationship with `price` , this seems

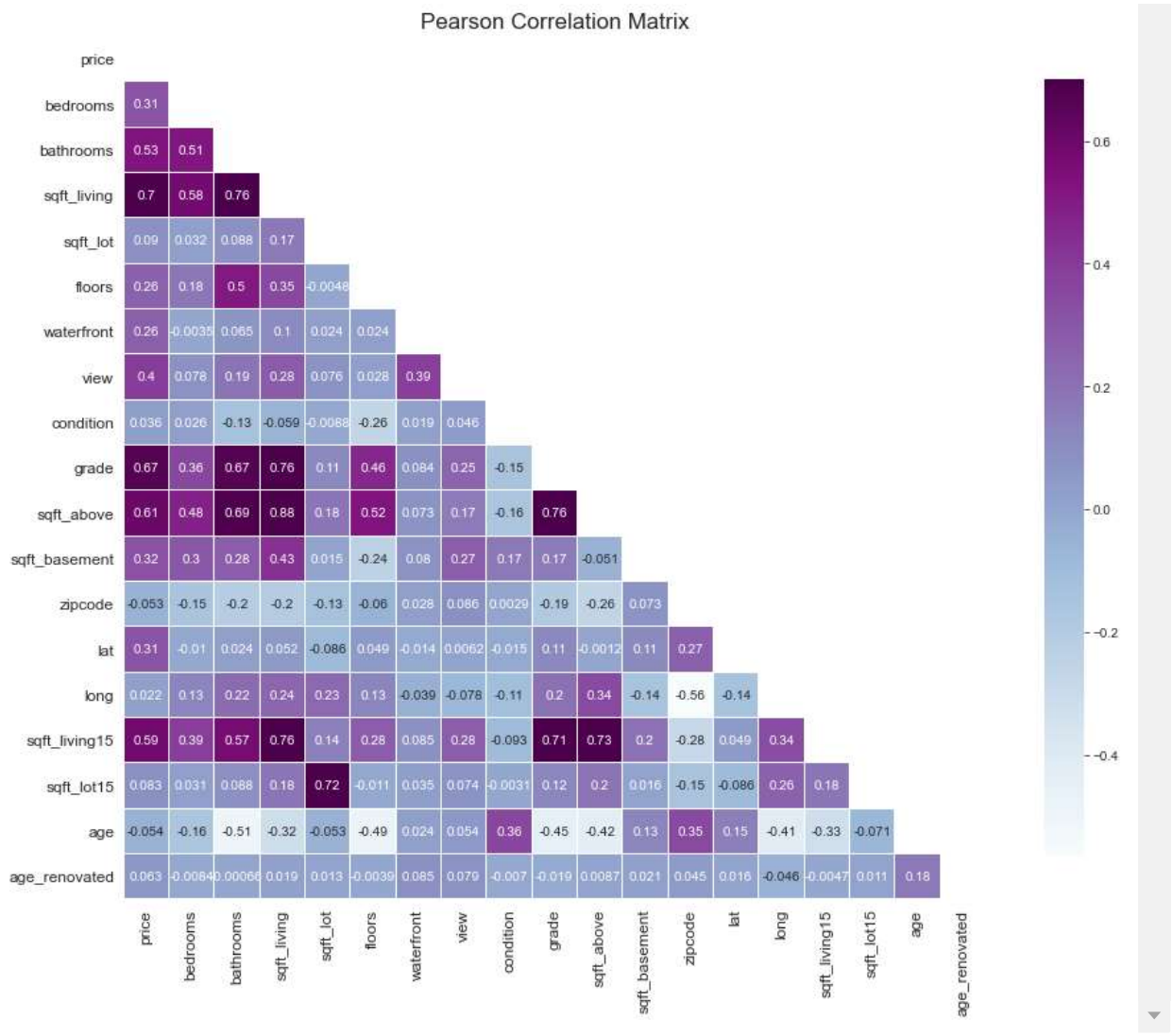
inconsequential for `floors`. The column `grade` has excellent correlation with `price`. The column `view` shows that view could dectate house price. Therefore, it should be noted that treating `view` as non-categorical feature was a good choice, just as for `waterfront`. Later on, we will create models with categorical `view` and `waterfront` columns and see if there is any appreciable effect if changes were not made.

5.2 Correlation of Features

Now that there is a qualitative understanding of feature-target relationship, it's time to quantify what those relationships are and how to best exploit them.

In [24]:

```
# Change seaborn style
sns.set_style('white')

# Select all features except date
features = houses_df.columns[1:]

# Create a triangular mask to show the lower triangular section of the heatm
mask = np.zeros_like(houses_df[features].corr(), dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

# Initialize plot
fig, ax = plt.subplots(figsize=(16, 12))
plt.title('Pearson Correlation Matrix',fontsize=17)

# Plot the Pearson Correlation Matrix for the numeric columns
sns.heatmap(houses_df[features].corr(),linewidths=0.25,vmax=0.7,square=True,
            cmap="BuPu",linecolor='w',annot=True,annot_kws={"size":10},
            mask=mask,cbar_kws={"shrink": .9},ax=ax);

# Adjust font size of ticks
ax.set_xticklabels(ax.get_xmajorticklabels(),fontsize=12);
ax.set_yticklabels(ax.get_ymajorticklabels(),fontsize=12);

# Save plot
plt.savefig('images/corr.png')
```

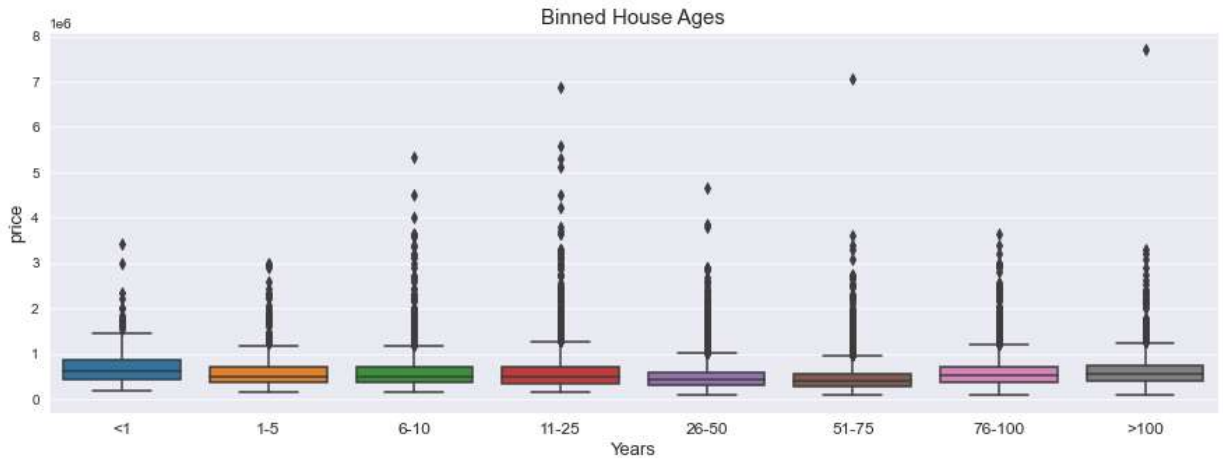


The correlation matrix paints better picture on how these features have some sort of interdependence. For example, `sqft_above` and `sqft_above` have the highest correlation with each other. `grade` also has strong correlation with most of the features that quantify the property areas. The first instinct as a data scientist would be to throw away one of the most highly correlated features and rightfully so, but the method to be employed later (stepwise selection) will automatically do that for us. So, we will keep the features as they are for now.

5.3 Binning

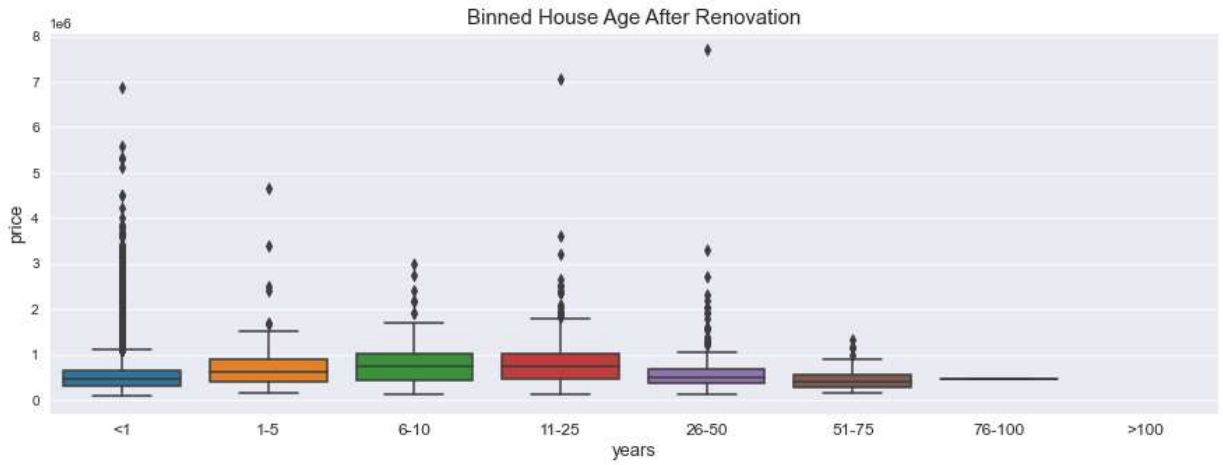
Based on the linearity plots shown earlier, it is probably fair to surmise that age related features do not do well with `price`. In this subsection, we will create bins for `age` and `age_renovated` columns and treat those columns as discrete features. In this case, we will take different ranges: < 1 year, 1 to 5, 6 to 10, 11 to 25, 26 to 50, 50 to 75, 75 to 100, and >100 years.

In [25]:
```
# Binning ages into different categories
## Define bins by interval
bins = [-1,0,5,10,25,50,75,100,200]
labels = ['<1','1-5','6-10','11-25','26-50','51-75','76-100','>100']

## Store binned age data
age_binned = pd.cut(houses_df.age, bins=bins, labels=labels)

## Plot age bins
## Reset style
sns.set_style('darkgrid')
fig, ax = plt.subplots(figsize=(15,5))
sns.boxplot(x=age_binned,y=houses_df.price, ax=ax)
ax.set_title('Binned House Ages',fontsize=15)
ax.set_xlabel('Years');

# Change label names for OLS
labels_new = ['lt_1','yr_1_to_5','yr_6_to_10','yr_11_to_25','yr_26_to_50',
              'yr_51_to_75', 'yr_76_to_100','gt_100']

# Apply the bins
age_binned = pd.cut(houses_df.age, bins=bins, labels=labels_new)

# Adjust font size of labels and ticks
ax.set_xticklabels(ax.get_xmajorticklabels(),fontsize=12);
ax.set_xlabel(ax.get_xlabel(),fontsize=13)
ax.set_ylabel(ax.get_ylabel(),fontsize=13)

# Save plot
plt.savefig('images/age_bin.png')
```



The plot shows that there are not discernible differences between different age bins. Nevertheless, the bins are stored for modeling stored. The same technique is also applied to `age_renovated` column, as shown below.

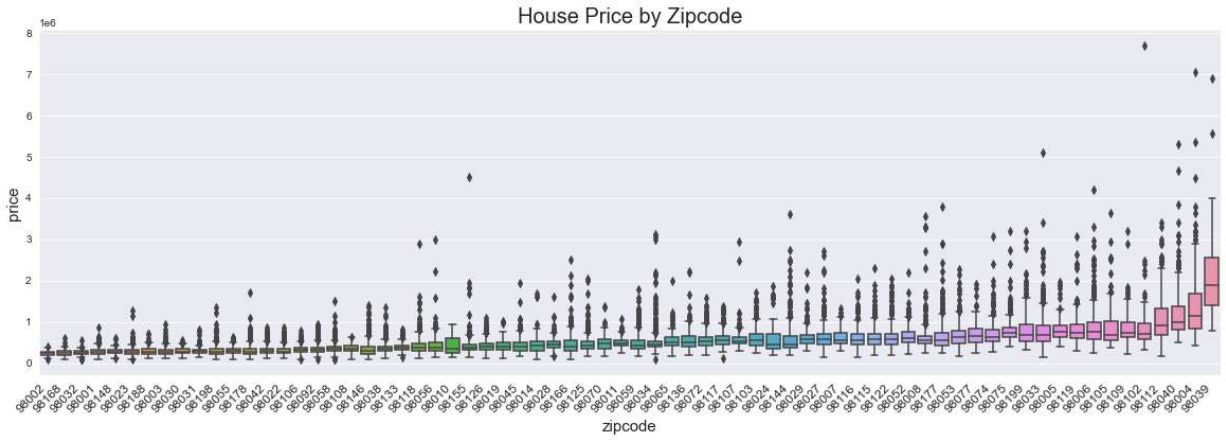
In [26]:
```
# Binning renovation ages into different categories
## Store binned renovated age data
age_renov_binned = pd.cut(houses_df.age_renovated, bins=bins,
                                          labels=labels)

## Plot renovated age bins
fig, ax = plt.subplots(figsize=(15,5))
sns.boxplot(x=age_renov_binned,y=houses_df.price, ax=ax)
ax.set_title('Binned House Age After Renovation',fontsize=15)
ax.set_xlabel('years',fontsize=13);
ax.set_ylabel('price',fontsize=13);

# Change label names for OLS
age_renov_binned = pd.cut(houses_df.age_renovated, bins=bins,
                                          labels=labels_new)

# Adjust font size of labels and ticks
ax.set_xticklabels(ax.get_xmajorticklabels(),fontsize=12);
ax.set_xlabel(ax.get_xlabel(),fontsize=13)
ax.set_ylabel(ax.get_ylabel(),fontsize=13)

# Save plot
plt.savefig('images/renovation_age_bin.png')
```



Similar to `age`, there appears to be little to no variation between different bins. The column `zipcode` was excluded from continuous features. In the plot below, we will show the argument that zip code does, in fact, should be treated a categorical instead of discrete numerical feature.

In [27]:

```
# Sort zipcodes by mean prices
# Assign zipcode and price to a new variable
price_zip_df = houses_df[['zipcode','price']].copy()

# Group by 'zipcode' and aggregate the mean
tmp = price_zip_df.groupby('zipcode')['price'].agg('mean')

# Sort by mean values and reset index
tmp = tmp.sort_values().reset_index().reset_index()

# Convert zipcodes to ranking of their mean value
price_zip_df.zipcode = price_zip_df.zipcode.apply(lambda x: int(tmp[tmp.zipc

# Plot zipcode bins
fig, ax = plt.subplots(figsize=(20,6))
sns.boxplot(x=price_zip_df.zipcode,y=price_zip_df.price, ax=ax)

# Reduce fontsize and rotate xtick for readability
ax.set_xticklabels(ax.get_xmajorticklabels(),fontsize=12,rotation=45,ha='rig

# Rename ticks to match with zipcodes
plt.xticks(ax.get_xticks(), tmp.zipcode)

# Adjust font sizes
ax.set_title('House Price by Zipcode', fontsize=20)
ax.set_xlabel('zipcode',fontsize=15)
ax.set_ylabel('price',fontsize=15);

# Save plot
plt.savefig('images/price_by_zipcode.png')
```



As shown in the plot above, zip codes do not asymptomatically increase with price. In fact, each zip code should be treated as a separate and independent feature. Therefore, it's obligatory to convert each zip code into its own categorical data. Afterwards, the `zipcode` column can be removed from the main dataframe.

In [28]:
```
# Convert zipcodes to one-hot encoding
zipcode_encoded = pd.get_dummies(houses_df.zipcode, drop_first=True)
zipcode_encoded.columns = ['zip_'+str(i) for i in zipcode_encoded.columns]

# Append zipcode_encoded to house_df and drop zipcode column
houses_df = pd.concat([houses_df,zipcode_encoded],axis=1)
houses_df.drop('zipcode',axis=1,inplace=True)
```

Lastly, we will investigate if the sale month and year have any impact on price despite the limited availability of data (only two years of data is used in this project). The months of sale can be extracted from the `date` column as follows.

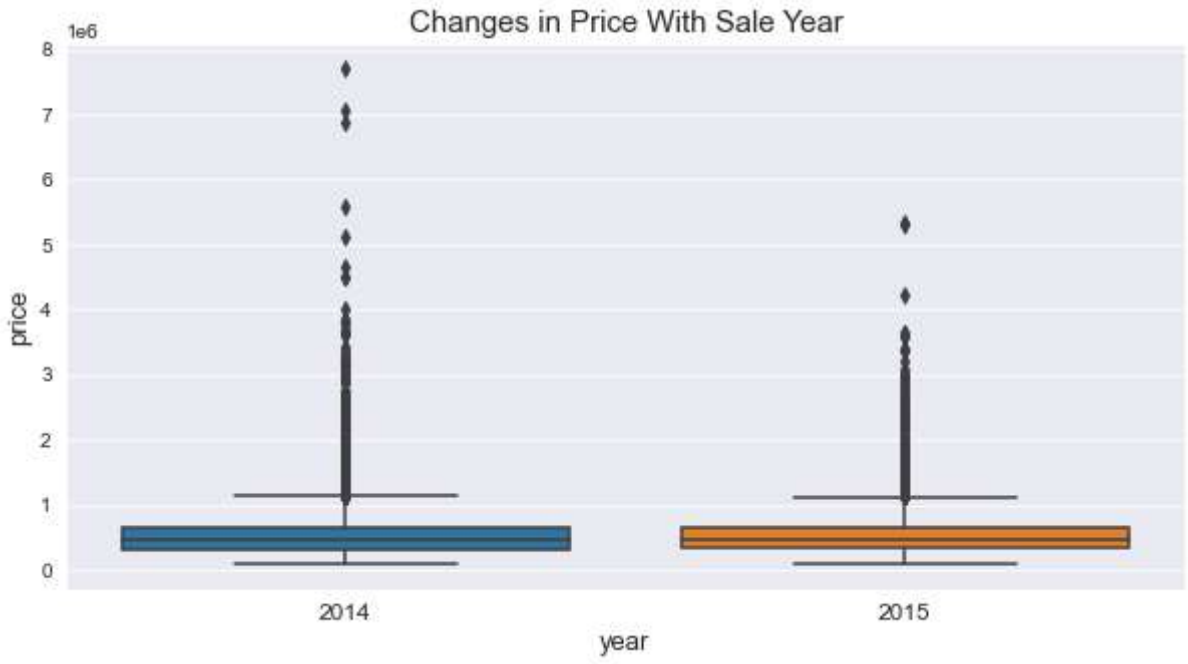
In [29]:
```
# Investigate if prices vary with months

# Create a new dataframe, assign 'date' column and extract month
months = houses_df[['date']]
months.columns = ['month']
months.month = months.month.dt.month


## Plot price changes with month
fig, ax = plt.subplots(figsize=(15,5))
sns.boxplot(x=months.month,y=houses_df.price, ax=ax)

# Convert xticks into shortened month names
month_names = [month_abbr[i] for i in sorted(months.month.unique())]
plt.xticks(ax.get_xticks(), month_names)

# Adjust font sizes of title, labels and ticks
ax.set_title('Changes in Price With Sale Month',fontsize=15)
ax.set_xticklabels(ax.get_xmajorticklabels(),fontsize=12)
ax.set_xlabel('month',fontsize=13)
ax.set_ylabel('price',fontsize=13);

# Save plot
plt.savefig('images/price_by_month.png')
```

Clearly, there is no relation between which month a sale is made to the price a house is sold. Months data will be later used as a separate categorical feature with each column representing a respective month. Now, let's look at year of sale.

In [30]:
```
# Convert month numeration to abbreviated names
months.month = months.month.apply(lambda x: month_names[x-1])

# Create an encoded feature for month
month_encoded = pd.get_dummies(months.month)
```

Now, let's look at year of sale.

In [31]:
```
# Investigate if prices vary with sale year
# Plot price changes with year
fig, ax = plt.subplots(figsize=(10,5))
sns.boxplot(x=houses_df.date.dt.year,y=houses_df.price, ax=ax)

# Adjust font sizes of title, labels and ticks
ax.set_title('Changes in Price With Sale Year',fontsize=15);
ax.set_xticklabels(ax.get_xmajorticklabels(),fontsize=12);
ax.set_xlabel('year',fontsize=13)
ax.set_ylabel('price',fontsize=13);

# Save plot
plt.savefig('images/price_by_year.png')
```



Similar to months, there is no significant difference between 2014 and 2015 sales. Now, the `date` column can now be dropped since we have extracted all relevant information from it. In addition, the years in the column are 2014 and 2015, which is a hardly useful information to perform any form of fitting.

In [32]:
```
# Drop 'date' from 'houses_df'
houses_df.drop('date', axis=1, inplace=True)
```

5.4 Multiple Linear Regressions

It's now time to make multiple linear regression analysis of the data. We first begin by splitting the data into train and test datasets.

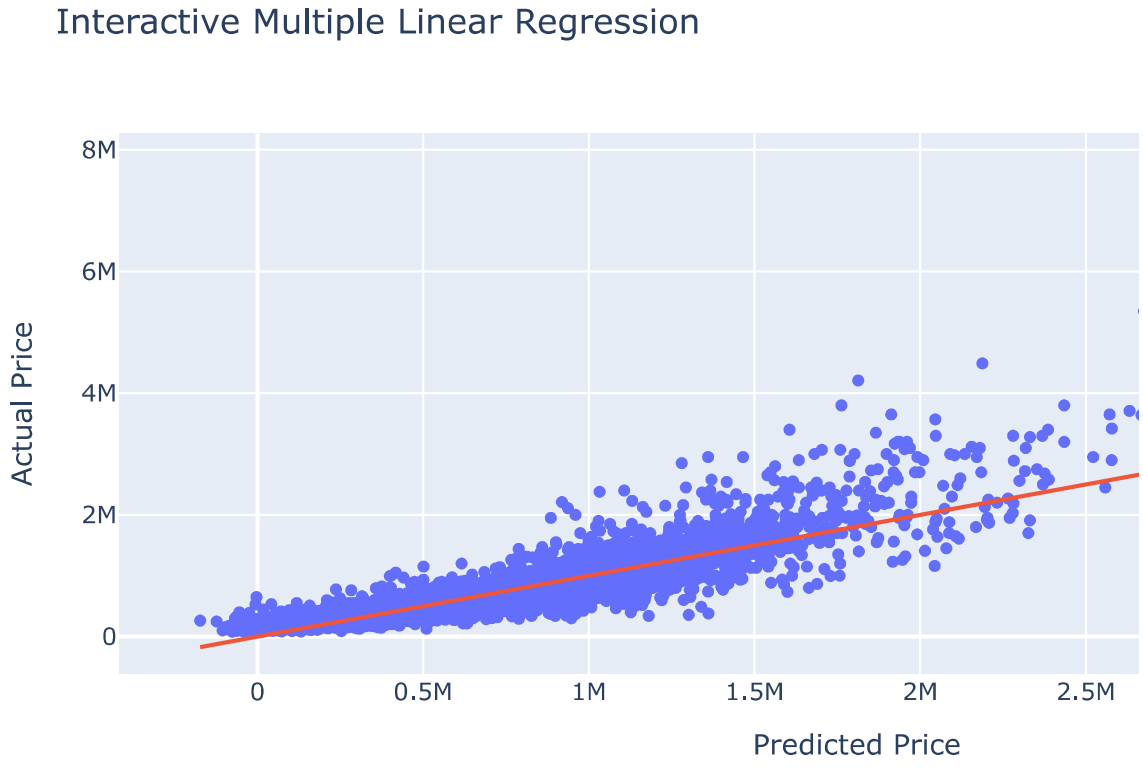
In [33]:
```
# Create train and test sets
train_data,test_data = train_test_split(houses_df, test_size = 0.2,
                                        random_state=42)
```

An interactive fit is shown with all features. Users can check and uncheck features and see how the predictions change in real-time.

In [34]:

```
# Use JupyterDash as the default plot
app = JupyterDash(__name__)

# Select all features except price (target) and date
options = []
for col in train_data.columns:
    if ('price' in col) or ('date' in col):
        continue
    else:
        options.append({'label': col, 'value': col})

# Set the layout
app.layout = html.Div([
    # Create the graph first
    dcc.Graph(id='graph'),
    # Populate checklists with feature names
    dcc.Checklist(
        id = 'features-selected',
        options = options,
        value = list(train_data.columns)[2:],
        labelStyle = {'display': 'inline-block'}
    )
])

# Callback function, triggered when there's an event
@app.callback(
    Output('graph', 'figure'),
    Input('features-selected', 'value')
)

# Update the graph when checkbox event it triggered
def update_graph(features_selected):
    # Update only if at least one feature is selected
    if len(features_selected)>0:
        # Create feature and target variables
        X = train_data[features_selected].copy()
        y = train_data.price

        # Initialize sklearn linear regression
        linreg = LinearRegression()


        # Fit and predict values
        linreg.fit(X,y)
        y_preds = linreg.predict(X)

        # Compute the coefficient of correlation
        r_squared = r2_score(y, y_preds)

        # Store predictions and actual values in a dataframe
        results = pd.DataFrame(np.concatenate((y_preds.reshape(-1,1),
                                               np.array(y).reshape(-1,1)),
                                              axis=1),
                               columns=['predicted_price','actual_price'])

        # Sort with predicted_price, useful to create a fit line
```

```
        results = results.sort_values(by='predicted_price')

        # Initialize a plotly graph object
        # Make a scatter plot (actual vs prediction)
        fig = go.Figure(layout_title_text="Interactive Multiple Linear Regre
        fig.add_traces(go.Scatter(x=results['predicted_price'],
                                  y=results['actual_price'],
                                  mode='markers',
                                  name = 'Observations'))

        # Add fit line
        fig.add_traces(go.Scatter(x=results['predicted_price'],
                                  y=results['predicted_price'],
                                  mode='lines',
                                  name = 'Model',
                                  hovertemplate='R-squared: {:.3f}'.format(r
                                 ))

        # Set axes titles
        fig.update_xaxes(title_text="Predicted Price")
        fig.update_yaxes(title_text="Actual Price")
        return fig

# Run plotly server
app.enable_dev_tools(dev_tools_hot_reload =True)
app.run_server(mode='inline', port = 8070, dev_tools_ui=True,
               dev_tools_hot_reload =True, threaded=True, debug=False)
```



☐ bedrooms ☑ bathrooms ☑ sqft_living ☑ sqft_lot ☑ floors ☑ waterfront ☑ view
☑ condition ☑ grade ☑ sqft_above ☑ sqft_basement ☑ lat ☑ long ☑ sqft_living15
☑ sqft_lot15 ☑ age ☑ age_renovated ☑ zip_98002 ☑ zip_98003 ☑ zip_98004
☑ zip_98005 ☑ zip_98006 ☑ zip_98007 ☑ zip_98008 ☑ zip_98010 ☑ zip_98011
☑ zip_98014 ☑ zip_98019 ☑ zip_98022 ☑ zip_98023 ☑ zip_98024 ☑ zip_98027
☑ zip_98028 ☑ zip_98029 ☑ zip_98030 ☑ zip_98031 ☑ zip_98032 ☑ zip_98033
☑ zip_98034 ☑ zip_98038 ☑ zip_98039 ☑ zip_98040 ☑ zip_98042 ☑ zip_98045
☑ zip_98052 ☑ zip_98053 ☑ zip_98055 ☑ zip_98056 ☑ zip_98058 ☑ zip_98059
☑ zip_98065 ☑ zip_98070 ☑ zip_98072 ☑ zip_98074 ☑ zip_98075 ☑ zip_98077
☑ zip_98092 ☑ zip_98102 ☑ zip_98103 ☑ zip_98105 ☑ zip_98106 ☑ zip_98107

5.5 Iterative Modeling

5.5.1 Stepwise Selection

In this step, features are selected or eliminated based on their fit criteria: their p-values. To make this process, stepwise selection methods with statsmodels ols function. This function does recursive feature elimination (same as `RFE` in `sklearn`) but the number of features need not be specified, which makes the function run slower as more features are included. The source of this function can be found in [this link (https://datascience.stackexchange.com/questions/24405/how-to-do-stepwise-regression-using-sklearn/24447#24447)](https://datascience.stackexchange.com/questions/24405/how-to-do-stepwise-regression-using-sklearn/24447#24447).

In [35]:

```
# Stepwise forward and backward selection function.
def stepwise_selection(X, y,
                       initial_list=[],
                       threshold_in=0.01,
                       threshold_out = 0.05,
                       verbose=True):
    """ Perform a forward-backward feature selection
    based on p-value from statsmodels.api.OLS
    Arguments:
        X - pandas.DataFrame with candidate features
        y - list-like with the target
        initial_list - list of features to start with (column names of X)
        threshold_in - include a feature if its p-value < threshold_in
        threshold_out - exclude a feature if its p-value > threshold_out
        verbose - whether to print the sequence of inclusions and exclusions
    Returns: list of selected features
    Always set threshold_in < threshold_out to avoid infinite looping.
    See https://en.wikipedia.org/wiki/Stepwise_regression for the details
    """
    included = list(initial_list)
    while True:
        changed=False
        # forward step
        excluded = list(set(X.columns)-set(included))
        new_pval = pd.Series(index=excluded)
        for new_column in excluded:
            model = sm.OLS(y, sm.add_constant(pd.DataFrame(X[included+[new_c
            new_pval[new_column] = model.pvalues[new_column]
        best_pval = new_pval.min()
        if best_pval < threshold_in:
            best_feature = new_pval.idxmin()
            included.append(best_feature)
            changed=True
            if verbose:
                print('Add  {:30} with p-value {:.6}'.format(best_feature, b

        # backward step
        model = sm.OLS(y, sm.add_constant(pd.DataFrame(X[included]))).fit()
        # use all coefs except intercept
        pvalues = model.pvalues.iloc[1:]
        worst_pval = pvalues.max() # null if pvalues is empty
        if worst_pval > threshold_out:
            changed=True
            worst_feature = pvalues.idxmax()
            included.remove(worst_feature)
            if verbose:
                print('Drop {:30} with p-value {:.6}'.format(worst_feature,
        if not changed:
            break
    return included
```

5.5.2 Proposed models

Type *Markdown* and LaTeX: α^2

In [36]:
```
# Store models, selected columns, prediction column,
# and transformer functions
models = []
columns = []
pred_column = []
transforms = []

# Function to generate formula
def model_gen(X, outcome):
    # Define predictors and formula
    predictors = [i for i in X.columns if i != outcome]

    # Run stepwise_selection for the best features
    columns_chosen = stepwise_selection(X[predictors], X[outcome])

    # Create formula based on columns chosen
    pred_sum = '+'.join(columns_chosen)
    formula = outcome + '~' + pred_sum

    # Fit the model using ols
    model = ols(formula=formula, data=X[columns_chosen+[outcome]]).fit()
    return model, columns_chosen
```

Model 1: All Features Fit

In the first model, all features are fitted without transformation.

In [37]:

```
# Fit with all features
def transform (df):
    '''
    All features fit
    '''
    return df


# Check if the 'model.pickle' exists
if not exists('model.pickle'):

    # Apply transform
    X = transform(train_data)

    # Obtain model and columns with best fit
    outcome = 'price'
    model, column = model_gen(X, outcome)

    # Append dataset and model
    models.append(model)
    columns.append(column)
    transforms.append(transform)
    pred_column.append(outcome)

else:
    # Check if 'model_runs' exist in globals
    if 'model_runs' not in globals():
        # Load models
        with open('model.pickle', 'rb') as f:
            model_runs = pickle.load(f)
    # Select model from 'model_runs' dictionary
    model = model_runs['models'][0]

# Model summary
model.summary()
```

Out[37]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.808
Model:	OLS	**Adj. R-squared:**	0.808
Method:	Least Squares	**F-statistic:**	1230.
Date:	Mon, 22 Nov 2021	**Prob (F-statistic):**	0.00
Time:	05:02:38	**Log-Likelihood:**	-2.3171e+05
No. Observations:	17277	**AIC:**	4.635e+05
Df Residuals:	17217	**BIC:**	4.640e+05
Df Model:	59		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-2.653e+07	6.62e+05	-40.048	0.000	-2.78e+07	-2.52e+07

Model 2: One-Hot Encoded `age` Bins Fit

Here, one-hot encoded `age` dataframe is augmented to the train dataset and fit to a linear regression.

In [38]:

```
# Fit with age bin categoricals
age_bin_encoded = pd.get_dummies(age_binned, drop_first=True)

# Transformer function
def transform(df):
    '''
    One-Hot encoded age bins fit
    '''
    # Drop 'age' column
    X = df.drop(columns=['age'])

    # Return dataframe concatenated with encoded age bins
    return pd.concat([X,age_bin_encoded.loc[X.index,:]],axis=1)

# Check if the 'model.pickle' exists
if not exists('model.pickle'):

    # Apply transform
    X = transform(train_data)

    # Obtain model and columns with best fit
    outcome = 'price'
    model, column = model_gen(X, outcome)

    # Append dataset and model
    models.append(model)
    columns.append(column)
    transforms.append(transform)
    pred_column.append(outcome)
else:
    # Check if 'model_runs' exist in globals
    if 'model_runs' not in globals():
        # Load models
        with open('model.pickle', 'rb') as f:
            model_runs = pickle.load(f)
    # Select model from 'model_runs' dictionary
    model = model_runs['models'][1]

# Model summary
model.summary()
```

Out[38]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.809
Model:	OLS	**Adj. R-squared:**	0.808
Method:	Least Squares	**F-statistic:**	1174.
Date:	Mon, 22 Nov 2021	**Prob (F-statistic):**	0.00
Time:	05:02:38	**Log-Likelihood:**	-2.3169e+05
No. Observations:	17277	**AIC:**	4.635e+05
Df Residuals:	17214	**BIC:**	4.640e+05
Df Model:	62		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-2.662e+07	6.78e+05	-39.257	0.000	-2.8e+07	-2.53e+07

Model 3: One-Hot Encoded `age_renovation` Bins Fit

Similar to model 2, one-hot encoded `age_renovated` is concatenated with our dataframe.

In [39]:

```
# Fit with renovated age bin categoricals
age_rnv_bin_encoded = pd.get_dummies(age_renov_binned, drop_first=True)

# Transformer function
def transform (df):
    '''
    One-Hot encoded renovation age bins fit
    '''
    # Drop 'age_renovated' column
    X = df.drop(columns=['age_renovated'])

    # Return dataframe concatenated with encoded age-since-renovation bins
    return pd.concat([X,age_rnv_bin_encoded.loc[X.index,:]],axis=1)


# Check if the 'model.pickle' exists
if not exists('model.pickle'):

    # Apply transform
    X = transform(train_data)

    # Obtain model and columns with best fit
    outcome = 'price'
    model, column = model_gen(X, outcome)

    # Append dataset and model
    models.append(model)
    columns.append(column)
    transforms.append(transform)
    pred_column.append(outcome)
else:
    # Check if 'model_runs' exist in globals
    if 'model_runs' not in globals():
        # Load models
        with open('model.pickle', 'rb') as f:
            model_runs = pickle.load(f)
    # Select model from 'model_runs' dictionary
    model = model_runs['models'][2]

# Model summary
model.summary()
```

Out[39]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.809
Model:	OLS	**Adj. R-squared:**	0.809
Method:	Least Squares	**F-statistic:**	1197.
Date:	Mon, 22 Nov 2021	**Prob (F-statistic):**	0.00
Time:	05:02:38	**Log-Likelihood:**	-2.3167e+05
No. Observations:	17277	**AIC:**	4.635e+05
Df Residuals:	17215	**BIC:**	4.639e+05
Df Model:	61		

Covariance Type: nonrobust

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	2.667e+07	9.61e+05	40.226	0.000	2.8e+07	2.54e+07

Model 4: One-Hot Encoded `view` and `condition` Fit

A special case is considered below, where `view` and `condition` are treated as categorical features instead of discrete numericals.

In [40]:

```
### Special case ###
# Convert 'view' and 'condition' back to categoricals
view_encoded = pd.get_dummies(houses_df.view,prefix='view', drop_first=True)
cond_encoded = pd.get_dummies(houses_df.condition,prefix='cond', drop_first=

# Transformer function
def transform(df):
    '''
    One-Hot encoded view and condition fit
    '''
    # Drop view and condition columns
    X = df.drop(columns=['view','condition'])

    # Return df concatenated with encoded view and condition
    return pd.concat([X,view_encoded.loc[X.index,:],
                      cond_encoded.loc[X.index,:]],axis=1)

# Check if the 'model.pickle' exists
if not exists('model.pickle'):


    # Apply transform
    X = transform(train_data)

    # Obtain model and columns with best fit
    outcome = 'price'
    model, column = model_gen(X, outcome)

    # Append dataset and model
    models.append(model)
    columns.append(column)
    transforms.append(transform)
    pred_column.append(outcome)
else:
    # Check if 'model_runs' exist in globals
    if 'model_runs' not in globals():
        # Load models
        with open('model.pickle', 'rb') as f:
            model_runs = pickle.load(f)
    # Select model from 'model_runs' dictionary
    model = model_runs['models'][3]

# Model summary
model.summary()
```

Out[40]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.811
Model:	OLS	**Adj. R-squared:**	0.810
Method:	Least Squares	**F-statistic:**	1208.
Date:	Mon, 22 Nov 2021	**Prob (F-statistic):**	0.00
Time:	05:02:38	**Log-Likelihood:**	-2.3160e+05
No. Observations:	17277	**AIC:**	4.633e+05

Df Residuals:	17215	**BIC:**	4.638e+05
Df Model:	61		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]

Model 5: Continuous Features Log and Normalized Fit

Normalization of log transformed features is carried out in the model below.

In [41]:

```
# Fit log transformed features and normalized

# Compute and store mean and std of log transformed training set
log_mean = pd.Series(np.log(train_data[continuous]).mean())
log_std = pd.Series(np.log(train_data[continuous]).std())

# Transformer function
def transform(df):
    '''
    Continuous features log and normalize fit
    '''
    # Copy the dataframe
    X = df.copy()

    # Take natural log of continuous features
    X[continuous] = np.log(X[continuous])

    # Apply normalization
    for col in continuous:
        X[col] = (X[col]-log_mean[col])/log_std[col]

    # Rename normalized columns
    X.columns = [i+'_log_norm' if i in continuous else i for i in X.columns]

    # Find columns with NaN
    # This happens if the features have values <=0
    house_na = X.columns[X.isna().any()]

    # Restore NaN columns
    for na in house_na:
        orig_col = na.split('_')
        mod_na = '_'.join(orig_col[:-2])
        X[mod_na] = df[mod_na]

    # Remove NaN columns
    return X.drop(columns=house_na)

# Check if the 'model.pickle' exists
if not exists('model.pickle'):


    # Apply transform
    X = transform(train_data)

    # Obtain model and columns with best fit
    outcome = 'price'
    model, column = model_gen(X, outcome)

    # Append dataset and model
    models.append(model)
    columns.append(column)
    transforms.append(transform)
    pred_column.append(outcome)
else:
    # Check if 'model_runs' exist in globals
    if 'model_runs' not in globals():
```

```
        # Load models
        with open('model.pickle', 'rb') as f:
            model_runs = pickle.load(f)
    # Select model from 'model_runs' dictionary
    model = model_runs['models'][4]

# Model summary
model.summary()
```

Out[41]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.783
Model:	OLS	**Adj. R-squared:**	0.782
Method:	Least Squares	**F-statistic:**	985.4
Date:	Mon, 22 Nov 2021	**Prob (F-statistic):**	0.00
Time:	05:02:38	**Log-Likelihood:**	-2.3278e+05
No. Observations:	17277	**AIC:**	4.657e+05
Df Residuals:	17213	**BIC:**	4.662e+05
Df Model:	63		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-1.417e+07	1.78e+06	-7.959	0.000	-1.77e+07	-1.07e+07

Model 6: `log(price)` fit

Looking at how the price is distributed, it is a fair assumption that price might fit better if log of price was used. The model below does just that.

In [42]:

```
# Fit log transformed target
# Check if the 'model.pickle' exists

# Transformer function
def transform(df):
    '''
    log(price) fit
    '''
    # Copy the dataframe
    X = df.copy()

    # Create a column with log of 'price'
    X['log_price'] = np.log(X.price)

    # Drop 'price' column
    return X.drop('price', axis=1)

if not exists('model.pickle'):
    # Apply transform
    X = transform(train_data)

    # Obtain model and columns with best fit
    outcome = 'log_price'
    model, column = model_gen(X, outcome)

    # Append dataset and model
    models.append(model)
    columns.append(column)
    transforms.append(transform)
    pred_column.append(outcome)
else:
    # Check if 'model_runs' exist in globals
    if 'model_runs' not in globals():
        # Load models
        with open('model.pickle', 'rb') as f:
            model_runs = pickle.load(f)
    # Select model from 'model_runs' dictionary
    model = model_runs['models'][5]

# Model summary
model.summary()
```

Out[42]:

OLS Regression Results

Dep. Variable:	log_price	**R-squared:**	0.875
Model:	OLS	**Adj. R-squared:**	0.875
Method:	Least Squares	**F-statistic:**	1680.
Date:	Mon, 22 Nov 2021	**Prob (F-statistic):**	0.00
Time:	05:02:38	**Log-Likelihood:**	4524.5
No. Observations:	17277	**AIC:**	-8903.
Df Residuals:	17204	**BIC:**	-8337.

Df Model:	72					
Covariance Type:	nonrobust					
	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**

Model 7: Second Order Polynomial Fit

Regression models generally have better fit with higher order polynomials. To avoid overfitting, only continuous features were squared.

In [43]:

```
# Fit with second degree polynomial transformation

# Transformer function
def transform(df):
    '''
    Second order polynomial fit
    '''
    # Copy the dataframe
    X = df.copy()

    # Square columns with 3 or more unique values
    for col in X.columns:
        if col!='price' and len(X[col].unique())>2:
            X[col+'_sq'] = X[col]**2
    return X


# Check if the 'model.pickle' exists
if not exists('model.pickle'):

    # Apply transform
    X = transform(train_data)

    # Obtain model and columns with best fit
    outcome = 'price'
    model, column = model_gen(X, outcome)

    # Append dataset and model
    models.append(model)
    columns.append(column)
    transforms.append(transform)
    pred_column.append(outcome)
else:
     # Check if 'model_runs' exist in globals
    if 'model_runs' not in globals():
        # Load models
        with open('model.pickle', 'rb') as f:
            model_runs = pickle.load(f)
    # Select model from 'model_runs' dictionary
    model = model_runs['models'][6]

# Model summary
model.summary()
```

Out[43]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.840
Model:	OLS	**Adj. R-squared:**	0.840
Method:	Least Squares	**F-statistic:**	1437.
Date:	Mon, 22 Nov 2021	**Prob (F-statistic):**	0.00
Time:	05:02:38	**Log-Likelihood:**	-2.3014e+05
No. Observations:	17277	**AIC:**	4.604e+05
Df Residuals:	17213	**BIC:**	4.609e+05

Df Model:	63
Covariance Type:	nonrobust

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-2.177e+04	550.768	-39.521	0.000	-2.28e+04	-2.07e+04

Model 8: Second Order Polynomial and `log(price)` Fit

The model below tries to take the best of two worlds: log of target and polynomial fit. Similar to the case above, continuous features are selected for higher order variable.

In [44]:

```
# Let's transform 'price' and apply polynomial regression
# Check if the 'model.pickle' exists

# Transformer function
def transform(df):
    '''
    Second order polynomial and log(price) fit
    '''
    # Copy the dataframe
    X = df.copy()

    # Square columns with 3 or more unique values
    for col in X.columns:
        if col!='price' and len(X[col].unique())>2:
            X[col+'_sq'] = X[col]**2

    # Create a column with log of 'price'
    X['log_price'] = np.log(X.price)

    # Remove price
    return X.drop('price',axis=1)


if not exists('model.pickle'):

    # Apply transform
    X = transform(train_data)

    # Obtain model and columns with best fit
    outcome = 'log_price'
    model, column = model_gen(X, outcome)

    # Append dataset and model
    models.append(model)
    columns.append(column)
    transforms.append(transform)
    pred_column.append(outcome)
else:
    # Check if 'model_runs' exist in globals
    if 'model_runs' not in globals():
        # Load models
        with open('model.pickle', 'rb') as f:
            model_runs = pickle.load(f)
    # Select model from 'model_runs' dictionary
    model = model_runs['models'][7]

# Model summary
model.summary()
```

Out[44]:

OLS Regression Results

Dep. Variable:	log_price	**R-squared:**	0.885
Model:	OLS	**Adj. R-squared:**	0.884
Method:	Least Squares	**F-statistic:**	1671.

Date:	Mon, 22 Nov 2021	**Prob (F-statistic):**	0.00
Time:	05:02:38	**Log-Likelihood:**	5195.7
No. Observations:	17277	**AIC:**	-1.023e+04
Df Residuals:	17197	**BIC:**	-9611.
Df Model:	79		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**

Model 9: Two Feature-Interaction Fit

It is a possibility that interaction terms can provide better fit quality instead of just using the features alone. In the code block below, the fit is made by augmenting the data with two-features interaction.

In [45]:

```
# Interactions between two features

# Transformer function
def transform(df):
    '''
    Two feature-interaction fit
    '''
    # Copy the dataframe
    X = df.copy()

    # First find the unique values
    unq = X.nunique()

    # Remove categorical features
    unq = unq[unq>2]

    # Get column combination
    col_comb = combinations(unq.index,2)

    # Iterate though each combination and multiply features
    for a,b in col_comb:
        if a!='price' and b!='price':
            X[a+'_x_'+b] = X[a]*X[b]
    return X

# Check if 'model.pickle' file exists
if not exists('model.pickle'):

    # Apply transform
    X = transform(train_data)

    # Obtain model and columns with best fit
    outcome = 'price'
    model, column = model_gen(X, outcome)

    # Append dataset and model
    models.append(model)
    columns.append(column)
    transforms.append(transform)
    pred_column.append(outcome)
else:
    # Check if 'model_runs' exist in globals
    if 'model_runs' not in globals():
        # Load models
        with open('model.pickle', 'rb') as f:
            model_runs = pickle.load(f)
    # Select model from 'model_runs' dictionary
    model = model_runs['models'][8]

# Model summary
model.summary()
```

Out[45]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.871

Model:	OLS	**Adj. R-squared:**	0.871
Method:	Least Squares	**F-statistic:**	1129.
Date:	Mon, 22 Nov 2021	**Prob (F-statistic):**	0.00
Time:	05:02:39	**Log-Likelihood:**	-2.2827e+05
No. Observations:	17277	**AIC:**	4.567e+05
Df Residuals:	17173	**BIC:**	4.575e+05
Df Model:	103		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	1.381e+08	9.16e+06	15.066	0.000	1.2e+08	1.56e+08

Model 10: One-Hot Encoded Sale Month Fit

The last models accounts for monthly variation in sales.

In [46]:

```
# Effect of month on sale price

# Transformer function
def transform (df):
    '''
    One-Hot encoded sale month fit
    '''
    # Copy the dataframe
    X = df.copy()

    # Return dataframe concatenated with encoded month
    return pd.concat([X,month_encoded.loc[X.index,:]],axis=1)


# Check if the 'model.pickle' exists
if not exists('model.pickle'):

    # Apply transform
    X = transform(train_data)

    # Obtain model and columns with best fit
    outcome = 'price'
    model, column = model_gen(X, outcome)

    # Append dataset and model
    models.append(model)
    columns.append(column)
    transforms.append(transform)
    pred_column.append(outcome)

else:
    # Check if 'model_runs' exist in globals
    if 'model_runs' not in globals():
        # Load models
        with open('model.pickle', 'rb') as f:
            model_runs = pickle.load(f)
    # Select model from 'model_runs' dictionary
    model = model_runs['models'][9]

# Model summary
model.summary()
```

Out[46]:

OLS Regression Results

Dep. Variable:	price	**R-squared:**	0.810
Model:	OLS	**Adj. R-squared:**	0.809
Method:	Least Squares	**F-statistic:**	1180.
Date:	Mon, 22 Nov 2021	**Prob (F-statistic):**	0.00
Time:	05:02:39	**Log-Likelihood:**	-2.3165e+05
No. Observations:	17277	**AIC:**	4.634e+05
Df Residuals:	17214	**BIC:**	4.639e+05

Df Model:	62
Covariance Type:	nonrobust

	coef	std err	t	P>|t|	[0.025	0.975]

The models generator code above can take a long time to run. Therefore, to save time, the models, columns with small p-values, target columns and transformer functions are stored to file. For this project, `dill` was chosen over `pickle` because `dill` supports storing functions in a dictionary to file.

In [47]:
```
# Save to dill
# Check if the 'model.pickle' exists
if not exists('model.pickle'):
    # Store models, columns, transforms and pred_columns
    # in a dictionary
    model_runs = {'models':models, 'columns':columns, 'transforms':transform
                  'pred_column':pred_column}
    with open('model.pickle', 'wb') as f:
        pickle.dump(model_runs, f)
```

6 Regression Results

6.1 Models Evaluation

In this section, models are evaluated for their fir performance.

6.1.1 Model Performance

The root mean square error (RMSE) and coefficient of correlation are computed for each model.

In [48]:

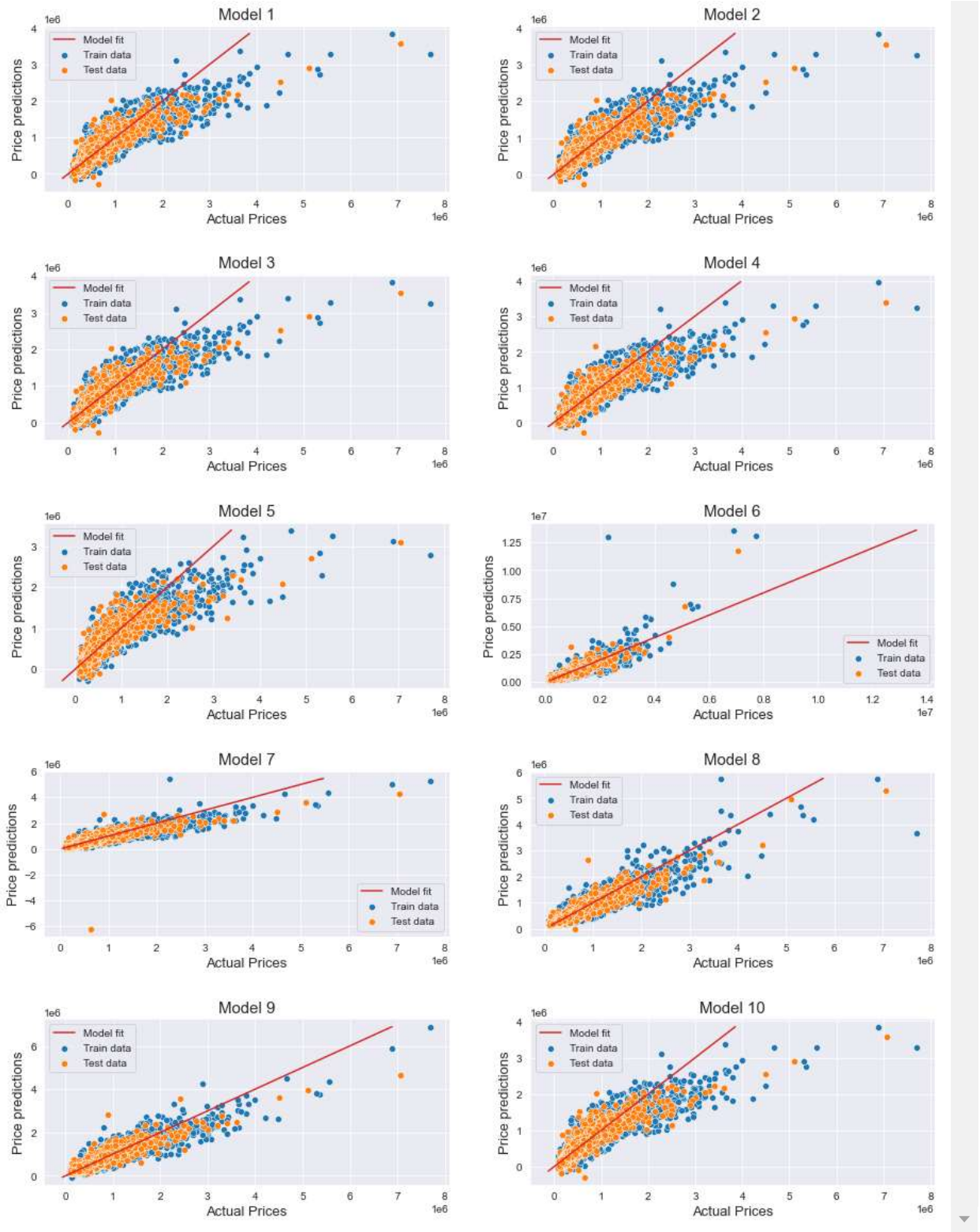
```
# Run model fit on test dataset and store results in a dataframe
rep_cols = ['Model','RMSE (train)','RMSE (test)',
            'R-squared (train)','R-squared (test)','Description']

# Create 'report' dataframe with 'rep_cols' columns
report = pd.DataFrame(columns=rep_cols)

# Change y to exp(y) if log(y) if fitted
def transform_y(df,pred_col):
    if 'log' in pred_col:
        df = np.exp(df)
    return df

# Initialize model subplots
fig, axes = plt.subplots(5,2,figsize=(15,20))

# Iterate through each subplot and model
for ax, i in zip(axes.flatten(),range(len(model_runs['models']))):
    # Model number
    vals = [i+1]

    # Transform train_data
    X_train = model_runs['transforms'][i](train_data.copy())

    # Select columns with small p-values
    target_train = X_train[model_runs['pred_column'][i]]

    # Predict train based on model
    train_pred = model_runs['models'][i].predict(X_train)

    # Get the prediction column
    pred_col = model_runs['pred_column'][i]

    # Transform target and train predictions
    target_train = transform_y(target_train,pred_col)
    train_pred = transform_y(train_pred,pred_col)

    # Compute root mean squared error of train data
    rmse_train = mean_squared_error(target_train,train_pred)

    # Add train RMSE to report
    vals.append(round(np.sqrt(rmse_train),2))

    # Transform test data
    X_test = model_runs['transforms'][i](test_data.copy())

    # Get target column in test data
    target_test = X_test[model_runs['pred_column'][i]]

    # Run predictions to the test data
    test_pred = model_runs['models'][i].predict(X_test)

    # Transform target and prediction of test data
    target_test = transform_y(target_test,pred_col)
    test_pred = transform_y(test_pred,pred_col)

```

```
    # Compute test RMSE
    rmse_test = mean_squared_error(target_test,test_pred)

    # Add test RMSE to report
    vals.append(round(np.sqrt(rmse_test),2))

    # Get train R-square from the model
    train_r_2 = model_runs['models'][i].rsquared

    # Add train R-square to report
    vals.append(round(train_r_2,3))

    # Calculate test R-square
    test_r_2 = r2_score(target_test,test_pred)

    # Add test R-square to report
    vals.append(round(test_r_2,3))

    # Add transformer docstring as description
    vals.append(model_runs['transforms'][i].__doc__.strip())

    # Append report vector to report dataframe
    report.loc[i,:] = vals

    # Plot train target and prediction
    sns.scatterplot(y=train_pred, x=target_train, ax=ax, label='Train data')

    # Plot test target and prediction
    sns.scatterplot(y=test_pred, x=target_test, ax=ax, label='Test data')

    # Add fit line to the plot
    sns.lineplot(x=train_pred, y=train_pred, ax=ax, label='Model fit',
                 color='tab:red')

    # Add labels and legend
    ax.set_xlabel('Actual Prices',fontsize=13)
    ax.set_ylabel('Price predictions',fontsize=13)
    ax.set_title('Model '+str(i+1),fontsize=15)
    ax.legend()

# Adjust height between subplots
plt.subplots_adjust(hspace = 0.5);

# Save plot
plt.savefig('images/models_fit.png')

# Set 'Model' as an index for readability
report.set_index('Model')
```

Out[48]:

Model	RMSE (train)	RMSE (test)	R-squared (train)	R-squared (test)	Description
1	161559	163545	0.808	0.795	All features fit
2	161338	163530	0.809	0.795	One-Hot encoded age bins fit

Model	RMSE (train)	RMSE (test)	R-squared (train)	R-squared (test)	Description
3	161166	163565	0.809	0.795	One-Hot encoded renovation age bins fit
4	160548	163532	0.811	0.795	One-Hot encoded view and condition fit
5	171907	178821	0.783	0.754	Continuous features log and normalize fit
6	174698	149610	0.875	0.828	log(price) fit
7	147470	182549	0.84	0.744	Second order polynomial fit
8	128776	123806	0.885	0.882	Second order polynomial and log(price) fit
9	132351	135068	0.871	0.86	Two feature-interaction fit
10	161009	163108	0.81	0.796	One-Hot encoded sale month fit



Based on the results, taking log of price was the best choice because models 6 and 8 have the highest r-square value on the train dataset. Also, a simple polynomial fit had one anomalous prediction on the test dataset. Interaction between features also yielded a good fit, although it is difficult to ascertain if that could have been caused through overfitting. Age, renovation age and sale month one hot encoding did not improve the performance. One hot encoded view and condition also did not perform well.

6.1.2 Cross Validation Results

Cross-validation on 20 fold K-fold was performed if the models have good prediction with the entire data combined.

In [49]:

```
# Cross validation report

# Initialize CV list
scores_cv = []

# Iterate through each model
for i in range(len(report)):

    # Transform the entire dataset
    X = model_runs['transforms'][i](houses_df.copy())

    # Get target column
    y = X[model_runs['pred_column'][i]]

    # Remove target column from X
    X.drop(model_runs['pred_column'][i],axis=1,inplace=True)

    # Initialize sklearn linear regression
    linreg = LinearRegression()

    # Run 20 cross validation tests
    scores = cross_validate(linreg, X, y, cv=20,
                            scoring=('r2'),
                             return_train_score=True)


    # Add scores to list
    scores_cv.append(scores)

# Create cross validation dataframe
report_cv = pd.DataFrame(scores_cv)

# Take the average of the 20 cross validations
report_cv = report_cv.applymap(lambda x: round(np.mean(abs(x)),3))

# Reverse the column order to match with 'report' dataframe
report_cv = report_cv[report_cv.columns[::-1]]

# Remove fit time and score time columns
report_cv.drop(columns=['fit_time','score_time'],inplace=True)

# Concatenate report_cv with model column
report_cv = pd.concat([report.Model,report_cv],axis=1)

# Add description
report_cv['Description'] = report.Description

# Rename columns
report_cv.columns = [i for i in rep_cols if 'RMSE' not in i]

# Set Model as index for readability
report_cv.set_index('Model')
```

Out[49]:

Model	R-squared (train)	R-squared (test)	Description

Model	R-squared (train)	R-squared (test)	Description
1	0.807	0.804	All features fit
2	0.807	0.805	One-Hot encoded age bins fit
3	0.808	0.805	One-Hot encoded renovation age bins fit
4	0.809	0.807	One-Hot encoded view and condition fit
5	0.780	0.778	Continuous features log and normalize fit
6	0.876	0.874	log(price) fit
7	0.840	0.813	Second order polynomial fit
8	0.885	0.876	Second order polynomial and log(price) fit
9	0.874	0.861	Two feature-interaction fit
10	0.808	0.806	One-Hot encoded sale month fit

Once again, model 8 scored higher than the rest of the models, which means our assessment will be based upon this particular model's predictions. In the code, it's likely to find the RMSE. However, since `sklearn` shuffles the data before indexing, it would be difficult to determine the true values counterparts to the prediction. This cause a rather small RMSE value when `log_price` is used as target.

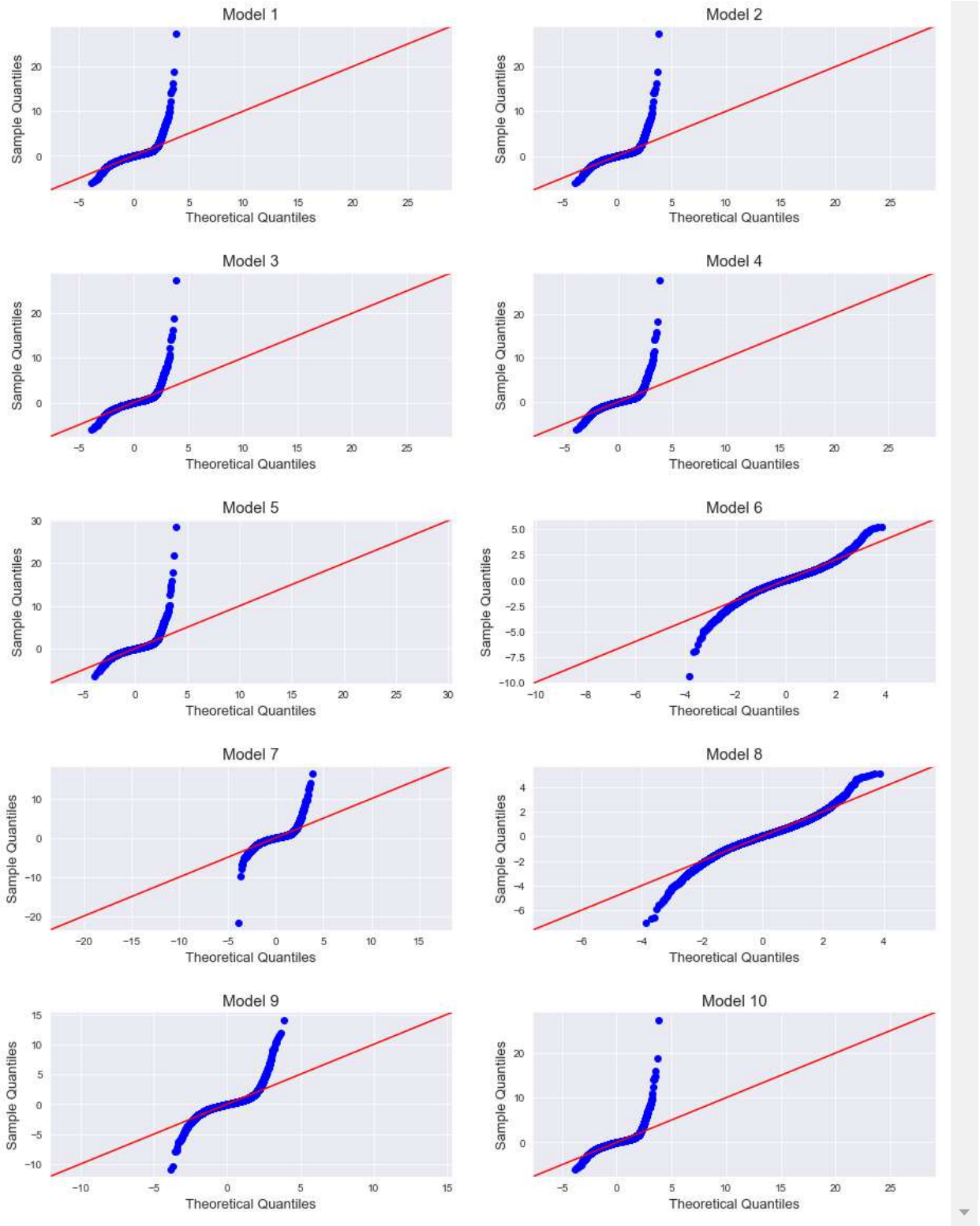
6.2 Post-modeling Assumption Check

6.2.1 Normality of Residuals

In this section, normalities of residuals is checked. For this task, `statsmodel`'s `qqplot` is used. If the residual quantile lie on the line, it would mean that our assumption of residuals being normally distributed becomes valid.

In [50]:

```
# Initialize QQ subplots
fig, axes = plt.subplots(5,2,figsize=(15,20))

# Iterate through each axis
for i, ax in enumerate(axes.flatten()):

    # Plot QQ plots for each model
    sm.graphics.qqplot(model_runs['models'][i].resid, dist=stats.norm,
                       line='45', fit=True, ax=ax)

    # Set the title of subplot, adjust fontsizes
    ax.set_title('Model '+str(i+1),fontsize=15)
    ax.set_xlabel(ax.get_xlabel(),fontsize=13);
    ax.set_ylabel(ax.get_ylabel(),fontsize=13);

# Adjust horizontal space between subplots
plt.subplots_adjust(hspace = 0.5);

# Save plot
plt.savefig('images/qqplots.png')
```



As expected, model 8 has the closest residual distribution to the theoretical normal distribution compared to others. The tail ends of the residual distribution do suffer for all prediction, which makes sense since houses with extremely low and high prices often fail to follow the trend for the average.

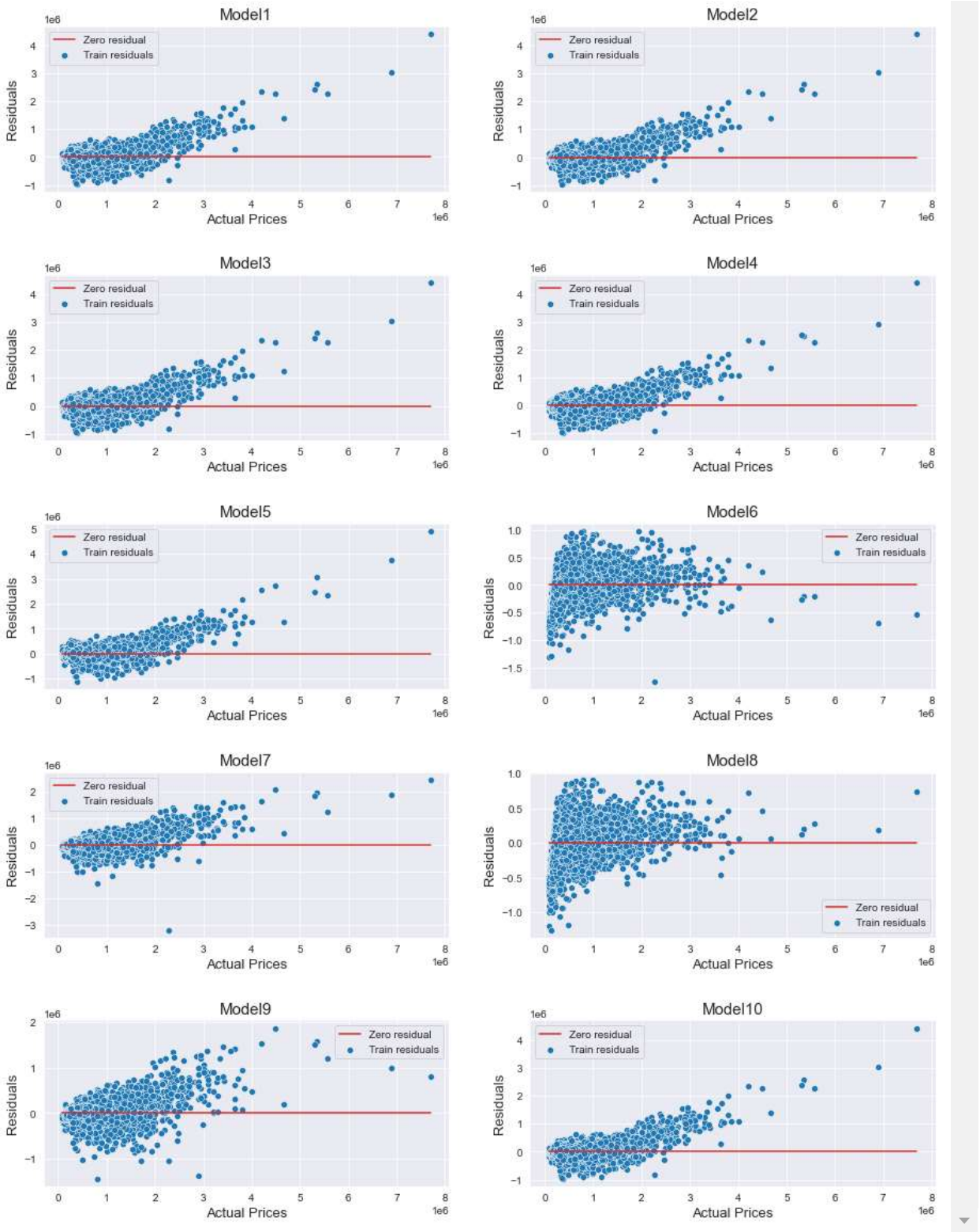
6.2.2 Homoscedasticity

Homoscedasticity is the desired outcome to generalize the model for all cases. The following plots

generate the residuals concentration with price in the train dataset.

In [51]:

```
# Initialize subplots for normality test
fig, axes = plt.subplots(5,2,figsize=(15,20))

# Iterate through each axis
for i, ax in enumerate(axes.flatten()):
    # Scatter plot with price and residuals of train data
    sns.scatterplot(x=train_data.price, y=model_runs['models'][i].resid,
                    ax=ax, label='Train residuals')

    # Plot the zero residual for a normal distribution assumption
    sns.lineplot(x=[int(train_data.price.min()),int(train_data.price.max())]
                 y=[0,0],
                 ax=ax, label='Zero residual',
                 color='tab:red')

    # Add labels and legend
    ax.set_xlabel('Actual Prices',fontsize=13)
    ax.set_ylabel('Residuals',fontsize=13)
    ax.set_title('Model'+str(i+1),fontsize=15)
    ax.legend()

# Adjust spaces between subplots
plt.subplots_adjust(hspace = 0.5);

# Save plot
plt.savefig('images/normality_plots.png')
```



Sad to say that none of the models are homoscedastic. But models 6 and 8 have almost symmetrically bounded residuals.

6.2.3 Multicollinearity

In section 5.2, we saw how the features are correlated. Another metric to measure multicollinearity is variance inflation factor (VIF). The following function (credit to Joél Collins) returns the VIF values for all features.

In [52]:
```
# Function to calculate variance inflation factor
def vif(features_df):
    # Select values from dataframe
    rows = features_df.values

    # Create a VIF report dataframe
    vif_df = pd.DataFrame()

    # Calculate the VIF of each feature with the rest
    vif_df["VIF"] = [variance_inflation_factor(rows, i) for i in range(featu

    # Add features column
    vif_df["feature"] = features_df.columns

    # Return a dataframe of features and their VIF
    return vif_df
```

The function is then applied to all transformed features in the `houses_df` .

In [53]:

```
# Initialize VIF report
report_vif = []

# Store VIF results
vif_df_store = []

# Iterate through each model
for i in range(len(report)):
    # Transform a copy of the entire dataset
    X = model_runs['transforms'][i](houses_df.copy())

    # Remove target column
    X.drop(model_runs['pred_column'][i],axis=1,inplace=True)

    # Select features with small p-values
    X = X[model_runs['columns'][i]]

    # Calculate VIF
    vif_df = vif(X)

    # Store VIF dataframe
    vif_df_store.append(vif_df)

    # Dictionary to store VIF<=5 and VIF<=20 features
    rep_dict = {'Model':i+1,'Total Number of Features':len(X.columns),
                'Features with VIF<=5':len(vif_df[vif_df.VIF<=5]),
                'Features with VIF<=20':len(vif_df[vif_df.VIF<=20])}

    # Append dictionary to list
    report_vif.append(rep_dict)

# Create VIF report dataframe
report_vif = pd.DataFrame(report_vif)

# Set model as index for readability
report_vif.set_index('Model')
```

Out[53]:

Model	Total Number of Features	Features with VIF<=5	Features with VIF<=20
1	59	48	50
2	62	52	54
3	61	50	52
4	61	52	53
5	63	53	58
6	72	62	64
7	64	55	57
8	81	66	70
9	103	63	73
10	62	51	53

A VIF less than 5 is the desired value to make sure that multicollinearity is kept at minimum among features. There is a fair amount of retention of features in most model except for model 9, where interactions are included. For model 8, the following columns scored low VIF.

In [54]:
```
tmp = vif_df_store[7]
tmp[tmp.VIF<=5].feature.to_list()
```

Out[54]:
```
['grade',
 'zip_98004',
 'zip_98022',
 'zip_98168',
 'zip_98040',
 'zip_98112',
 'zip_98178',
 'zip_98039',
 'condition',
 'waterfront',
 'zip_98033',
 'zip_98115',
 'zip_98199',
 'zip_98105',
 'zip_98117',
 'zip_98103',
 'zip_98119',
 'zip_98116',
 'zip_98107',
 'zip_98109',
 'zip_98122',
 'zip_98102',
 'zip_98136',
 'sqft_lot',
 'zip_98006',
 'zip_98005',
 'zip_98144',
 'zip_98008',
 'zip_98052',
 'zip_98029',
 'zip_98010',
 'zip_98007',
 'zip_98027',
 'zip_98126',
 'floors_sq',
 'bathrooms',
 'zip_98125',
 'zip_98198',
 'zip_98032',
 'zip_98075',
 'zip_98053',
 'zip_98074',
 'zip_98177',
 'zip_98034',
 'sqft_lot_sq',
 'zip_98118',
 'zip_98023',
 'zip_98059',
 'zip_98065',
 'zip_98166',
 'zip_98045',
 'zip_98070',
 'zip_98056',
```

```
'zip_98038',
'zip_98024',
'bedrooms_sq',
'bedrooms',
'zip_98133',
'zip_98072',
'zip_98077',
'zip_98188',
'zip_98092',
'zip_98055',
'zip_98031',
'zip_98011',
'sqft_lot15_sq']
```

In [55]:
```
tmp = vif_df_store[7]
tmp[tmp.VIF<=20].feature.to_list()
```

Out[55]:
```
['sqft_basement_sq',
 'grade',
 'age_sq',
 'view_sq',
 'zip_98004',
 'zip_98022',
 'zip_98168',
 'zip_98040',
 'zip_98112',
 'zip_98178',
 'zip_98039',
 'condition',
 'waterfront',
 'zip_98033',
 'zip_98115',
 'zip_98199',
 'zip_98105',
 'zip_98117',
 'zip_98103',
 'zip_98119',
 'zip_98116',
 'zip_98107',
 'zip_98109',
 'zip_98122',
 'zip_98102',
 'zip_98136',
 'sqft_lot',
 'zip_98006',
 'zip_98005',
 'zip_98144',
 'zip_98008',
 'zip_98052',
 'zip_98029',
 'zip_98010',
 'zip_98007',
 'zip_98027',
 'zip_98126',
 'floors_sq',
 'bathrooms',
 'zip_98125',
 'zip_98198',
 'zip_98032',
 'zip_98075',
 'zip_98053',
 'zip_98074',
 'zip_98177',
 'zip_98034',
 'sqft_lot_sq',
 'zip_98118',
 'zip_98023',
 'zip_98059',
 'zip_98065',
 'zip_98166',
```

```
 'zip_98045',
 'zip_98070',
 'zip_98056',
 'zip_98038',
 'zip_98024',
 'bedrooms_sq',
 'view',
 'bedrooms',
 'zip_98133',
 'zip_98072',
 'zip_98077',
 'zip_98188',
 'zip_98092',
 'zip_98055',
 'zip_98031',
 'zip_98011',
 'sqft_lot15_sq']
```

6.3 The Best Regression Model

Finally, model 8 is chosen as the best predictor of prices based on the given data. In order to understand the effects of each feature, we will use the exponents of the parameters in the model and see how a single increase in value can change the price.

In [56]:
```
params = model_runs['models'][7].params
params_coeff = [(i,j) for i,j in zip(params.index,params)]
params_coeff
```

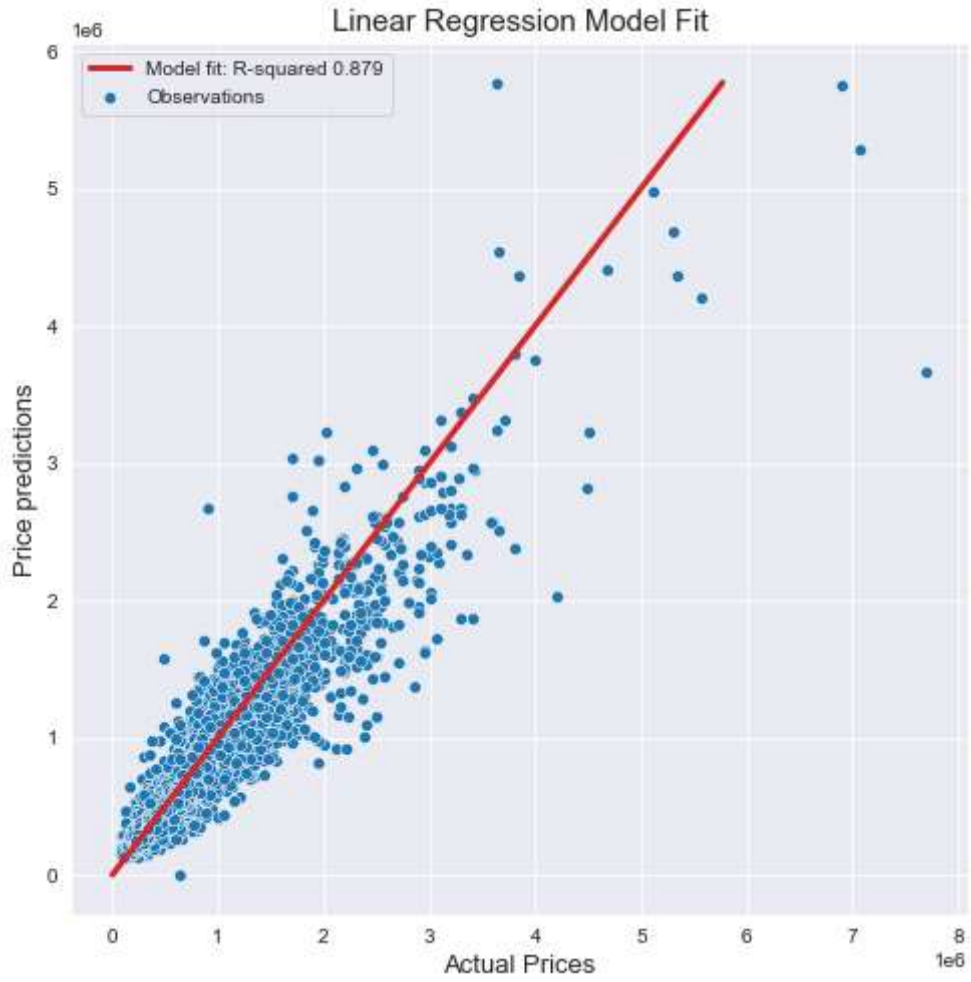
Out[56]:
```
[('Intercept', -0.37958419041761154),
 ('sqft_above_sq', -1.29403316569129e-08),
 ('lat_sq', -1.8482474751410227),
 ('sqft_basement_sq', -2.0506715434878797e-08),
 ('sqft_living15_sq', -2.3672837060862316e-08),
 ('sqft_living', 0.00021218640737016088),
 ('grade', 0.088396133198032),
 ('age_sq', 2.5375323903637093e-05),
 ('lat', 176.62218684072926),
 ('view_sq', 0.0076896770377054),
 ('zip_98004', 0.7637171856669349),
 ('zip_98022', 0.30994142488043963),
 ('zip_98168', -0.18221239881345028),
 ('zip_98040', 0.5657693078996715),
 ('zip_98112', 0.6445713684409486),
 ('zip_98178', -0.10712961512516661),
 ('zip_98039', 0.9183563319542892),
 ('condition', 0.054084490881062375),
 ('sqft_living15', 0.00018459252249657554),
 ('waterfront', 0.4475740490256618),
 ('age', -0.002118826804138522),
 ('zip_98033', 0.4155133429302097),
 ('zip_98115', 0.3953944211919352),
 ('zip_98199', 0.41123258970781557),
 ('zip_98105', 0.5262445480501226),
 ('zip_98117', 0.3544358033564645),
 ('zip_98103', 0.37886270488185503),
 ('zip_98119', 0.534305358068641),
 ('zip_98116', 0.35988017684902246),
 ('zip_98107', 0.38811702561931855),
 ('zip_98109', 0.5478907051501481),
 ('zip_98122', 0.3990368363662253),
 ('zip_98102', 0.5359657632487894),
 ('zip_98136', 0.31622695637775544),
 ('sqft_lot', 1.1147603877068804e-06),
 ('zip_98006', 0.36078319575275813),
 ('zip_98005', 0.40710808217251376),
 ('zip_98144', 0.2924760000774056),
 ('zip_98008', 0.34175036164220324),
 ('zip_98052', 0.2855518214578713),
 ('zip_98029', 0.3491237696937226),
 ('zip_98010', 0.27327021474461843),
 ('zip_98007', 0.34466781545564773),
 ('zip_98027', 0.2821226778851782),
 ('zip_98126', 0.1681462898949337),
 ('floors_sq', -0.017008523039601126),
 ('sqft_above', 0.00011661810441650894),
 ('bathrooms', 0.034381805033262),
 ('zip_98125', 0.15852165629845444),
 ('zip_98198', -0.09772615675572588),
 ('zip_98032', -0.13511569569447984),
 ('zip_98075', 0.27571571969055947),
```

```
('zip_98053', 0.2560041437316676),
('zip_98074', 0.2583796349914801),
('zip_98177', 0.1331966914587362),
('zip_98034', 0.15993803734248294),
('sqft_lot_sq', -6.019373337086129e-13),
('zip_98118', 0.13062277621261362),
('zip_98023', -0.0739014977160925),
('zip_98059', 0.11088934124707783),
('zip_98065', 0.19366812430933428),
('zip_98166', 0.05368461303615607),
('zip_98045', 0.21682830520141733),
('zip_98070', 0.04423859010033099),
('zip_98056', 0.06826046485684611),
('zip_98038', 0.12771705910247097),
('zip_98024', 0.19855337510842586),
('bedrooms_sq', -0.0067501921119212625),
('view', 0.041355781284253657),
('bedrooms', 0.04036323954941903),
('sqft_living_sq', -7.970517586488905e-09),
('zip_98133', 0.0239986067772706),
('zip_98072', 0.11653403029109222),
('long_sq', 0.28488385887445),
('long', 69.24781006647183),
('zip_98077', 0.10230781340602102),
('zip_98188', -0.09893319631729924),
('zip_98092', 0.05120915800711179),
('zip_98055', -0.05877713710347465),
('zip_98031', -0.04675867256456745),
('zip_98011', 0.05331541453024308),
('sqft_lot15_sq', -4.991723968911573e-13)]
```

Applying the selected model to the entire dataset gives the plot blow.

In [57]:

```
# Initialize plot
fig, ax = plt.subplots(figsize=(8,8))

# Choose model 8
i = 7

# Transform houses_data
houses_df_mod = model_runs['transforms'][i](houses_df)

# Select columns with small p-values
target = houses_df_mod[model_runs['pred_column'][i]]

# Predict log prices
preds = model_runs['models'][i].predict(houses_df_mod)

# Get the prediction column
pred_col = model_runs['pred_column'][i]

# Transform target and preds columns
target = transform_y(target,pred_col)
preds = transform_y(preds,pred_col)

# Calculate test R-square
r_2 = r2_score(target,preds)

# Plot target and preds
sns.scatterplot(y=preds, x=target, ax=ax, label='Observations')

# Add fit line to the plot
sns.lineplot(x=preds, y=preds, ax=ax,
                  label='Model fit: R-squared {}'.format(round(r_2,3)),
                  lw=3, color='tab:red')

# Add labels and legend
ax.set_xlabel('Actual Prices',fontsize=13)
ax.set_ylabel('Price predictions',fontsize=13)
ax.set_title('Linear Regression Model Fit',fontsize=15)
ax.legend();
plt.savefig('images/final_model.png')
```



7 Recommendations

Since the prices are transformed to log, these coefficients are hard to create a quantitative relationship between a single increase in one feature and its effect on the price. We can, however, have qualitative description.

Latitude and longitude appear to have dominating effect on the price. This could be a statistical error on the account of latitudes and longitudes have very small changes as we move from one house to the next. They can provide good tips in classification problems (like SVM or other kernel methods) by creating boundaries around neighborhoods with similar prices.

Some zip codes, namely 98168, 98178, 98198, 98032, 98023, 98188, 98055 and 98031, reduce the house price, while the remaining statistically significant zip codes appreciate it. Given the data at hand, it is hard to determine why this is but merits further investigation. Age also appears to have a negative effect on price as well.

Waterfront properties increase the price significantly. Also, houses with excellent views and condition cost more than the rest. The number of floors, square footage of interior housing living space for the nearest 15 neighbors and basement, and number of years after renovation have no significance in predicting prices.

8 Next Steps

The first step after this project would be to gather more data with that includes additional sale years. This can help to make better predictions as to how prices can vary with year. It also helps understand how the real estate business is responding to various economical fluctuations.

It is also important to investigate why some zip codes have over/undervalued prices. This step might appear inconsequential to the firm from profits perspective but rather a public service to create equitable community. It could shade some light on municipal solutions as to why some houses are too expensive to afford.

As a firm with interest in making better predictions, however, including median earning by zip code can significantly increase the prediction accuracy. This goes hand-in-hand with gathering more data.

The last important step is to use employ other predictive models, besides regression, that can perform significantly better. Regression requires data massaging before a good model is produced. Other models can, and most of the time, outperform linear regression.