CSCI 538 Final Project

In this notebook, we take a look at housing data collected from King County, Washington from May 2014-2015. The data has a number of different features included in the data set that describe different houses that were sold during that time period. The problem to be solved is to see if we can train a machine learning algorithm to model the data that was collected and use that model to predict the prices of houses that could be sold in the future. While trying to solve this problem we will look at exploration of the data, cleaning the data to prepare for the training of the model, and finally we will train the model and evaluate how well it was trained.

For this project we will be using the `scikit-learn` python library that includes many simple and efficient tools for predictive data analysis. Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities.

Housing Data Exploration

In the first section the housing data will be explored by taking a look at the different features of the data and visualizing it using different plotting methods. Exploration into the correlation between the different features will show how each one affects the price of the house. Exploring the data is a vital part of the machine learning process because before you can train a model you have to know what kind of data your dealing with. Any missing data must be fixed otherwise it will affect the training and subsequently the performance of the model.

In [1]:
```
#Importing pyton libraries needed to load, explore, visualize, and manipulate the data
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sbn
import pandas as pd
```

In [2]:
```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import SGDRegressor
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.linear_model import ElasticNet
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import r2_score
from sklearn.model_selection import train_test_split
```

In [3]:
```
#Loading the data into a pandas dataframe so that we can better visualize and manipulate
data = pd.read_csv('kingcoutyhousingdata.csv')
```

Using the `head()` function lets us explore the first 5 instances of the data and outputs the values for each of the features. This allows a quick overview of the features and the data types that are present in the data

In [4]:
```
#Displaying the first 5 instances in the dataset
data.head()
```

Out[4]:

	price	bedrooms	bathrooms	livingsq	sqft_lot	floors	waterfront	view	condition	grade	sqft_above	sqft_ment	yr_built	yr_re_ated	zipcode	lat
0	221900	3	1	1180	5650	1	0	0	3	7	1180	0	1955	0	98178	47.5112
1	538000	3	2	2570	7242	2	0	0	3	7	2170	400	1951	1991	98125	47.7210
2	180000	2	1	770	10000	1	0	0	3	6	770	0	1933	0	98028	47.7379
3	604000	4	3	1960	5000	1	0	0	5	7	1050	910	1965	0	98136	47.5208
4	510000	3	2	1680	8080	1	0	0	3	8	1680	0	1987	0	98074	47.6168

Before any training on the data begins, information such as data types and the number of null entries for features is very important to review. Any features that are not numerical must be converted to numberical features because the regression algorithms will not train on those features. Also, any null entries will also affect the regression algorithm as it will actually think that the feature is 0 instead of missing. Looking at the `info()` function, we can see that all features are numerical and there are no non-null entries for each feature.

There are a total of 19 different features with a breif description of each:
Price: Sale price
Bedrooms: Number of bedrooms
Bathrooms: Number of bathrooms
Livingsq: Size of living area in square feet
Sqft_lot: Size of lot in square feet
Floors: Number of floors
Waterfront: '1' if the porperty has a waterfront, '0' if not
View: An index from 0 to 4 of how good the view on the property is
Condition: Condition of the house, ranked from 1 to 5
Grade: Classification by construction quality which refers to the types of materials used and the quality of workmanship; the higher the better.
Sqft_above: Square feet above ground
Sqft_ment: Square feet below ground
Yr_built: Year built
Yr_re_ated: Year renovated. '0' if never renovated
Zipcode: 5 digit zip code
Lat: Latitude
Long: Longitude
Sqft_ng15: Average size of interior housing living space for the closest 15 houses, in square feet
Sqft_lot15: Average size of land lost for the closest 15 houses, in square feet

In [5]:
```
#Description of the different features with their non-null count and data types
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 21613 entries, 0 to 21612
Data columns (total 19 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   price       21613 non-null  int64
 1   bedrooms    21613 non-null  int64
 2   bathrooms   21613 non-null  int64
 3   livingsq    21613 non-null  int64
 4   sqft_lot    21613 non-null  int64
 5   floors      21613 non-null  int64
 6   waterfront  21613 non-null  int64
 7   view        21613 non-null  int64
 8   condition   21613 non-null  int64
 9   grade       21613 non-null  int64
 10  sqft_above  21613 non-null  int64
 11  sqft_ment   21613 non-null  int64
 12  yr_built    21613 non-null  int64
 13  yr_re_ated  21613 non-null  int64
 14  zipcode     21613 non-null  int64
 15  lat         21613 non-null  float64
 16  long        21613 non-null  float64
 17  sqft__ng15  21613 non-null  int64
 18  sqft_lot15  21613 non-null  int64
dtypes: float64(2), int64(17)
memory usage: 3.1 MB
```

The `describe()` function gives a more statistical description of the data providing details such as the mean, standard devation (std), minimum value, maximum value, and the values that represents the 25, 50, and 75 percentiles.

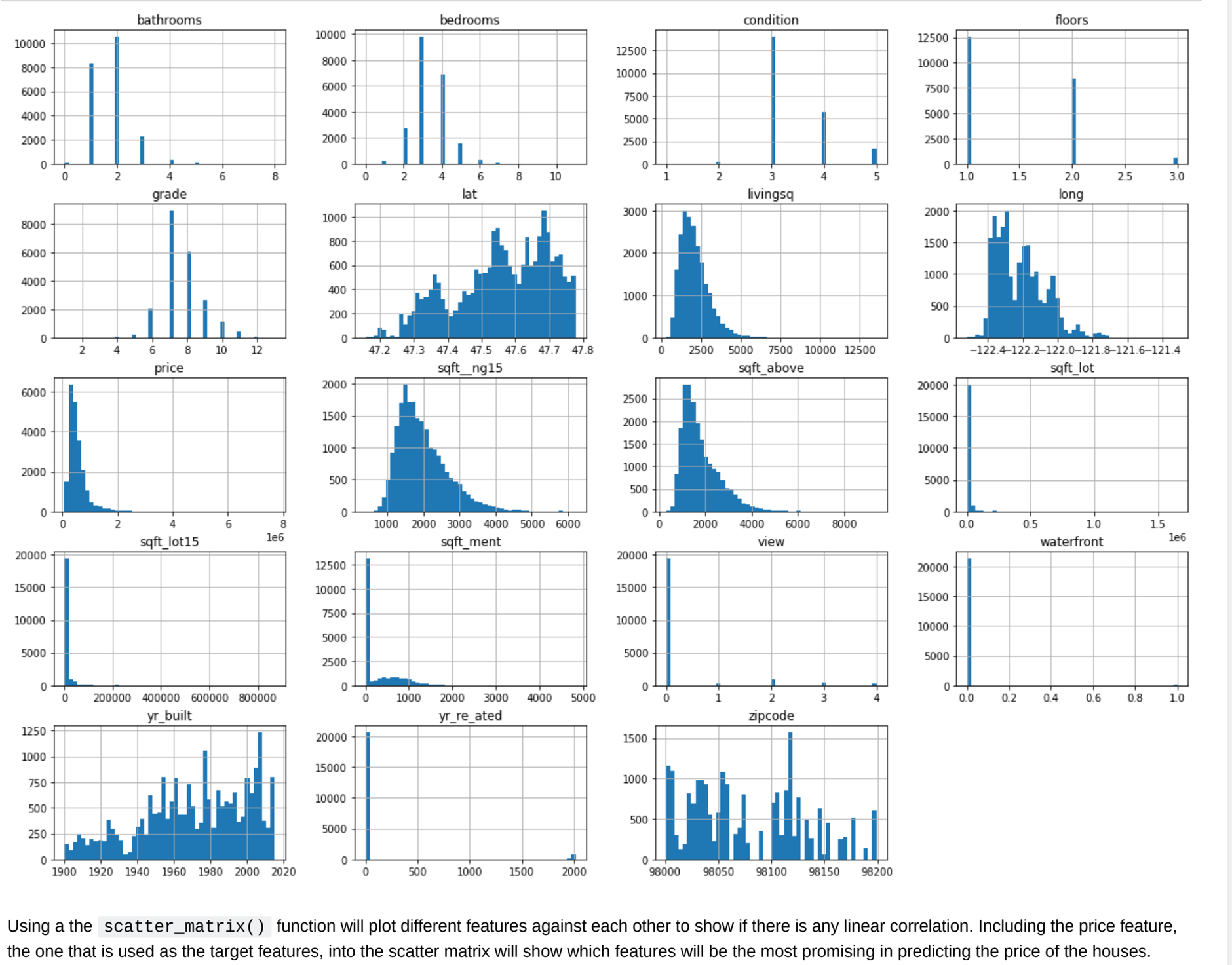
In [6]:
```
data.describe().T
```

Out[6]:

	count	mean	std	min	25%	50%	75%	max
price	21613.0	540088.141767	367127.196483	75000.0000	321950.000	450000.0000	645000.000	7.700000e+06
bedrooms	21613.0	3.369454	0.907964	0.0000	3.000	3.0000	4.000	1.100000e+01
bathrooms	21613.0	1.749734	0.734873	0.0000	1.000	2.0000	2.000	8.000000e+00
livingsq	21613.0	2079.899736	918.440897	290.0000	1427.000	1910.0000	2550.000	1.354000e+04
sqft_lot	21613.0	15106.967566	41420.511515	520.0000	5040.000	7618.0000	10688.000	1.651359e+06
floors	21613.0	1.446213	0.551894	1.0000	1.000	1.0000	2.000	3.000000e+00
waterfront	21613.0	0.007542	0.086517	0.0000	0.000	0.0000	0.000	1.000000e+00
view	21613.0	0.234303	0.766318	0.0000	0.000	0.0000	0.000	4.000000e+00
condition	21613.0	3.409430	0.650743	1.0000	3.000	3.0000	4.000	5.000000e+00
grade	21613.0	7.656873	1.175459	1.0000	7.000	7.0000	8.000	1.300000e+01
sqft_above	21613.0	1788.390691	828.090978	290.0000	1190.000	1560.0000	2210.000	9.410000e+03
sqft_ment	21613.0	291.509045	442.575043	0.0000	0.000	0.0000	560.000	4.820000e+03
yr_built	21613.0	1971.005136	29.373411	1900.0000	1951.000	1975.0000	1997.000	2.015000e+03
yr_re_ated	21613.0	84.402258	401.679240	0.0000	0.000	0.0000	0.000	2.015000e+03
zipcode	21613.0	98077.939805	53.505026	98001.0000	98033.000	98065.0000	98118.000	9.819900e+04
lat	21613.0	47.560053	0.138564	47.1559	47.471	47.5718	47.678	4.777760e+01
long	21613.0	-122.213896	0.140828	-122.5190	-122.328	-122.2300	-122.125	-1.213150e+02
sqft__ng15	21613.0	1986.552492	685.391304	399.0000	1490.000	1840.0000	2360.000	6.210000e+03
sqft_lot15	21613.0	12768.455652	27304.179631	651.0000	5100.000	7620.0000	10083.000	8.712000e+05

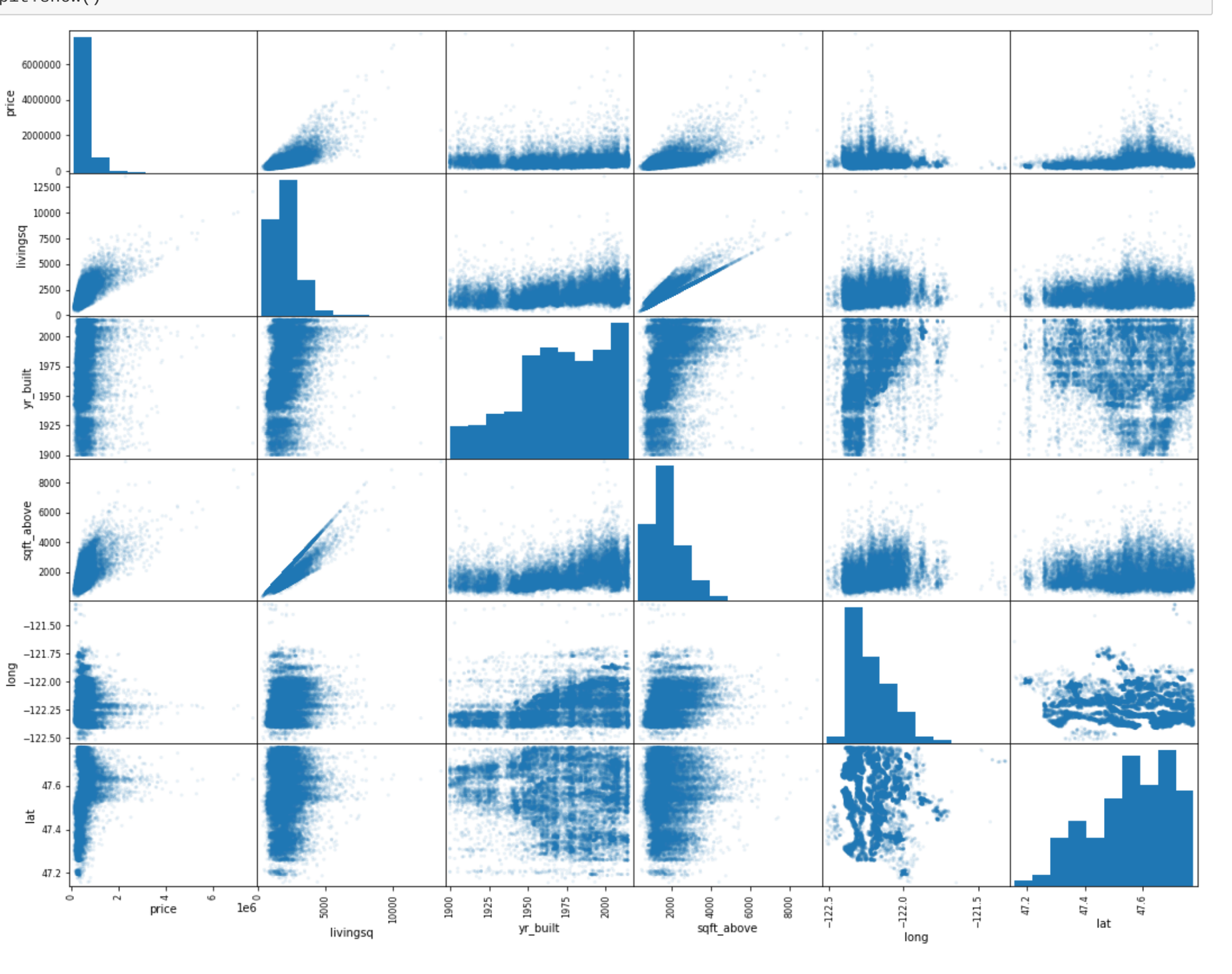
Visualizing the histograms of the data can reveal a few things which can then be used during the data cleanup phase. Looking at the data here we can see that some of the features are very tail-heavy and do not exhibit a normal distribution. Many of the features also have very different scales. Bedrooms range from 0 to 11 but livingsq ranges from 290 to 2550. Different scales like this are problems for some machine learning models, and thus we may need to scale attributes like these with widely different ranges to have a common similar range of values.

In [7]:
```
data.hist(bins=50, figsize=(20, 15))
plt.show()
```



Using a the `scatter_matrix()` function will plot different features against each other to show if there is any linear correlation. Including the price feature, the one that is used as the target features, into the scatter matrix will show which features will be the most promising in predicting the price of the houses.

In [9]:
```
#Import the scatter_matrix function
from pandas.plotting import scatter_matrix
attributes = ['price', 'livingsq', 'yr_built', 'sqft_above', 'long', 'lat']
scatter_matrix(data[attributes], alpha=0.1, figsize=(18, 14))
plt.show()
```



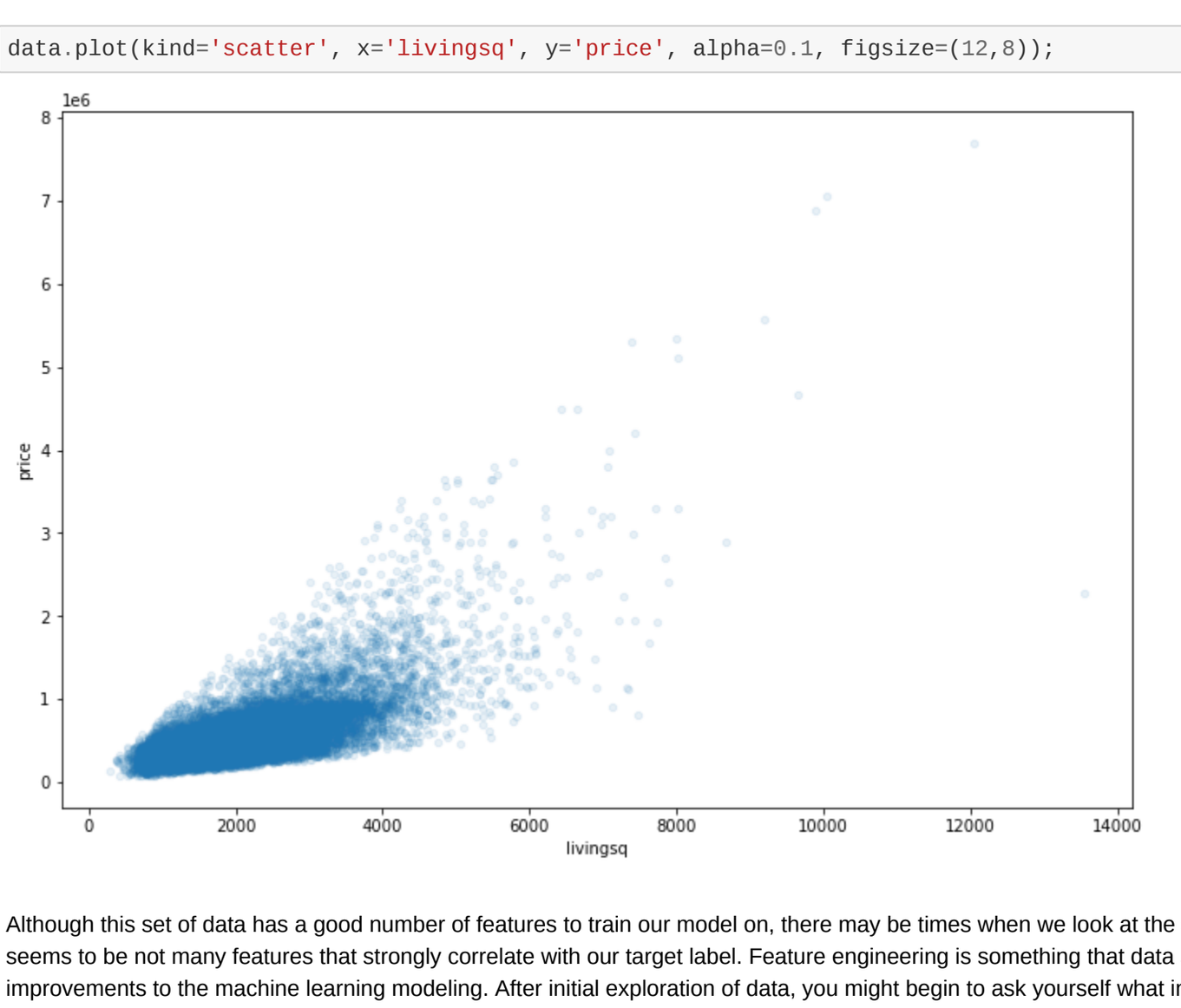
Similarly, the correlation function `corr_matrix` will give us numerical values to look at to help decide which functions provide a good linear correlation and which ones don't. Correlation matrix values range from -1 to 1. When it is close to 1 (a positive correlation) it means that there is a strong linear correlation between the features. This means that when one feature increases, the other usually increases in some standard ratio. A negative correlation, near to -1, means a correlation but in the opposite direction. When one increases, the other decreases in some standard linear ratio. A correlation of 0 means that there doesn't appear to be a relationship, when one increases, the other is as likely to increase or decrease or stay the same.

In [9]:
```
corr_matrix = data.corr()
corr_matrix['price'].sort_values(ascending=False)
```

Out[9]:
```
price         1.000000
livingsq      0.702035
grade         0.667434
sqft_above    0.605567
sqft__ng15    0.585379
bathrooms     0.510072
view          0.397293
sqft_ment     0.323816
bedrooms      0.315438
lat           0.307003
waterfront    0.266369
floors        0.237211
yr_re_ated    0.126434
sqft_lot      0.089661
sqft_lot15    0.082447
yr_built      0.054012
condition     0.036362
long          0.021626
zipcode      -0.053203
Name: price, dtype: float64
```

Just by looking at the numbers, it shows that `livingsq` is the most linearly correlated feature with the target `price` and we can visualize this by looking at the pair on a scatter plot. By looking at the scatter plot below, it is fairly obvious that there is a strong linear correlation between these two features. You can see what looks like an upward trend such that an increase in `livingsq` means there is an increase in the `price`.

In [16]:
```
data.plot(kind='scatter', x='livingsq', y='price', alpha=0.1, figsize=(12,8));
```



Although this set of data has a good number of features to train our model on, there may be times when we look at the correlation matrix data and there seems to be not many features that strongly correlate with our target label. Feature engineering is something that data scientists can do to provide significant improvements to the machine learning modeling. After initial exploration of data, you might begin to ask yourself what information might be useful for a model to learn to predict from. We can look at the existing features and create features with a clearer signal from existing data.

Below you can see that the bedrooms and bathrooms features provide a decent correlation with the target label. Just to experiment, we can create another feature called `rooms_per_sqft` by adding the bedroom and bathroom features together and dividing them by the total `livingsq`.

In [11]:
```
data['rooms_per_sqft'] = (data['bedrooms'] + data['bathrooms']) / data['livingsq']
```

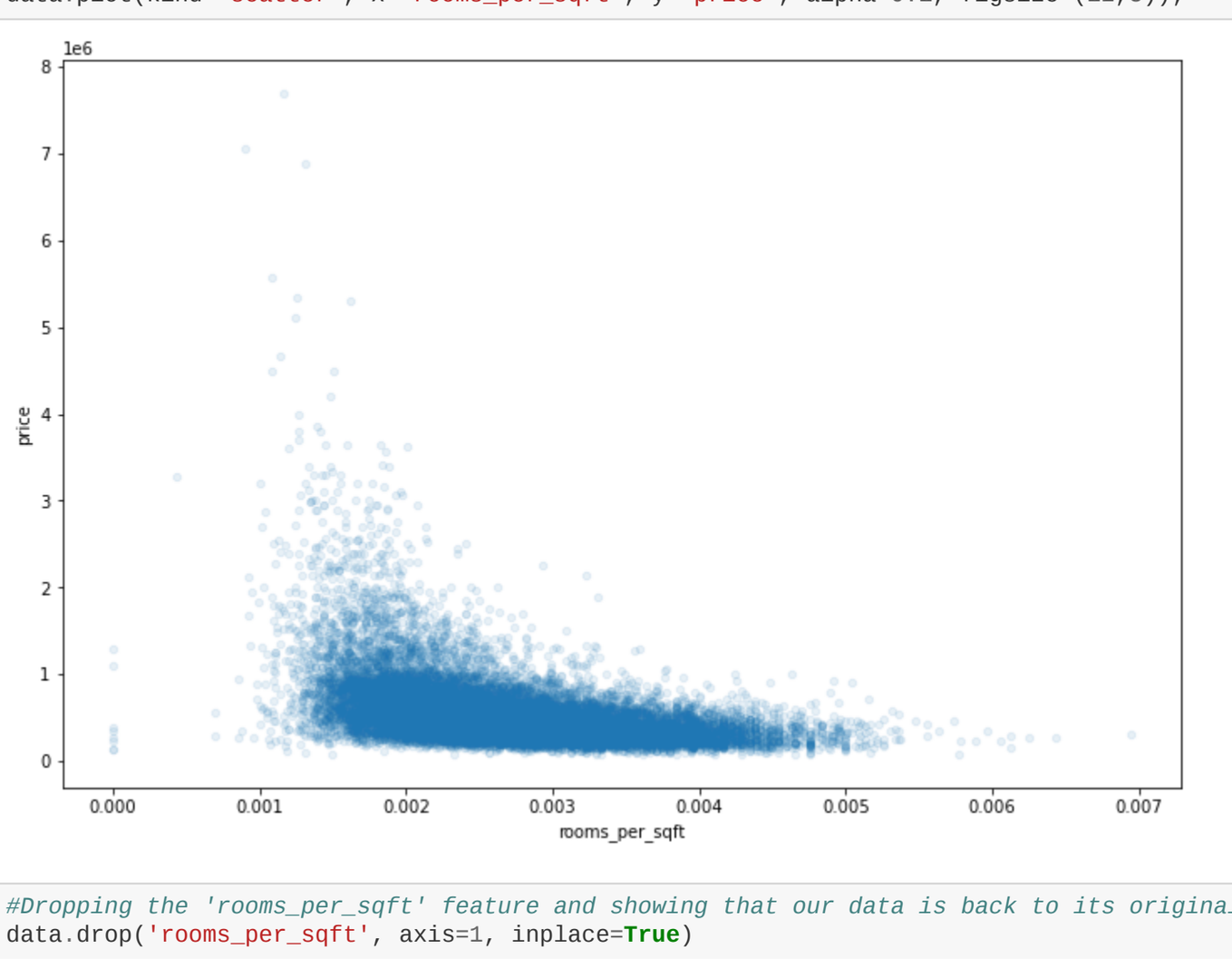
By running the `corr_matrix` function again with this new feature we can see that we've created a new feature that has a good negative correlation with the target label.

In [12]:
```
corr_matrix = data.corr()
corr_matrix['price'].sort_values(ascending=False)
```

Out[12]:
```
price             1.000000
livingsq          0.702035
grade             0.667434
sqft_above        0.605567
sqft__ng15        0.585379
bathrooms         0.510072
view              0.397293
sqft_ment         0.323816
bedrooms          0.315438
lat               0.307003
waterfront        0.266369
floors            0.237211
yr_re_ated        0.126434
sqft_lot          0.089661
sqft_lot15        0.082447
yr_built          0.054012
condition         0.036362
long              0.021626
zipcode          -0.053203
rooms_per_sqft   -0.461442
Name: price, dtype: float64
```

The downward trend is obvious here but it looks more like a polynomial correlation more than a linear one. This feature engineering was just for example purposes and we won't be actually using it in the modeling of our regression algorithm.

In [13]:
```
data.plot(kind='scatter', x='rooms_per_sqft', y='price', alpha=0.1, figsize=(12,8));
```



In [14]:
```
#Dropping the 'rooms_per_sqft' feature and showing that our data is back to its original state
data.drop('rooms_per_sqft', axis=1, inplace=True)
```

Preparing the Data for Linear Regression Algorithms

Once the data been explored, the next step is to prepare it so that we can train our machine learning models with it. The first thing that can be done is to drop features from the data set. Dropping features that don't have strong correlations with the target label can help the linear regression algorithms fit the training data without overfitting it. Overfitting happens when the algorithms tries too hard to fit the training data that it doesn't generalize well to unseen data.

Looking at the correlation matrix, there are a number of features that a very low correlation with the target label for each instance. Features such as `zipcode`, `yr_re_ated`, `sqft_lot15`, can be dropped to without very little hinderance to the accuracy and precision of our model and will also speed up the training.

In [15]:
```
#Dropping the 3 features that are not needed then printing the first 5 data instances to show those features are gon
e
data.drop(['zipcode', 'yr_re_ated', 'sqft_lot15'], axis=1, inplace=True)
data.head()
```

Out[15]:

	price	bedrooms	bathrooms	livingsq	sqft_lot	floors	waterfront	view	condition	grade	sqft_above	sqft_ment	yr_built	lat	long	sqft__ng15
0	221900	3	1	1180	5650	1	0	0	3	7	1180	0	1955	47.5112	-122.257	1340
1	538000	3	2	2570	7242	2	0	0	3	7	2170	400	1951	47.7210	-122.319	1690
2	180000	2	1	770	10000	1	0	0	3	6	770	0	1933	47.7379	-122.233	2720
3	604000	4	3	1960	5000	1	0	0	5	7	1050	910	1965	47.5208	-122.393	1360
4	510000	3	2	1680	8080	1	0	0	3	8	1680	0	1987	47.6168	-122.045	1800

The next step in preparing the data is to split the dataset up into a training set and a testing it. This is done so that we can help train our model so that it generalizes well to unseen data. If we were to train our model on our whole data set, we wouldn't be able to evaluate the performance of it because it would always predict correctly when asked to make a prediction. Training on only a portion of the data allows for the evaluation of the learned hypotesis function on that test data that was not seen during the training. Once we see how the hypothesis function performs on the unseen data, we can then start to make adjustments on the model to get it to the best it can be.

Using the `sklearn` `train_test_split` function allows the data to be split into a training set (which will be used to train the model with), and a test set which will be used to evaulate the model to gauge how successful the training was. The test size `0.2` specifies how big we want the testing data set to be. The parameter `random_state` which is set to 42 will ensure that anyone else who runs this notebook will have the same instances split into the training and testing sets and therefore have the same results in their evaulation of the model.

In [16]:
```
data_train, data_test = train_test_split(data, test_size=0.2, random_state=42)
print(data_train.shape)
print(data_test.shape)
```

```
(17290, 16)
(4323, 16)
```

The data is now split into a training set with 17290 instances and a testing set with 4323 instances and the next step is to further prepare our training dataset by dropping the price feature and putting it into a separate vector. This will let us train different models with different parameters so that the models can learn to predict the target label `price` based on the training data.

In [17]:
```
housing = data_train.drop('price', axis=1)
housing_labels = data_train['price'].copy()
housing_test = data_test.drop('price', axis=1)
housing_test_labels = data_test['price'].copy()

print(housing.shape)
print(housing_labels.shape)
print(housing_test.shape)
print(housing_test_labels.shape)
```

```
(17290, 15)
(17290,)
(4323, 15)
(4323,)
```

At this point, there dataset is pretty clean and is ready to be used to train machine learning models.

Training Linear Regression Algorithms

We now can train a number of machine learning models, evaluate, and then tune them a bit to try and get the best one that we can then use to predict future price of houses in the same area. All models from `sklearn` are full estimator - predictors. The purpose of a supervised learning machine learning algorithm is to generate a hypothesis function *h*() from the training data, and use the learned hypothesis to predict the target labels of values of the unseen data.

In [18]:
```
linear_regression = LinearRegression()
linear_regression.fit(housing, housing_labels)
```

Out[18]:
```
LinearRegression()
```

Now that the model has been trained (fitted), we can now test out and evaluate how well it makes predictions. Before we do that though it would be nice to run some formal evaluations on our model to see how well it performs. Using performance measures such as the `R-squared` score and `confidence intervals` we can see how well the model is fitted and is making perdictions on the unseen data.

The `R-squared` score is a measure of the scatter of the data points around the fitted regression line. It is also called the coefficient of determination, or the coefficient of multiple determination for multiple regression. For the same data set, higher R-squared values represent smaller differences between the observed data and the fitted values. `R-squared` scores are in the range of 0 to 1 and is normally measured as a percentage of the variance in the dependent variable that the independent variables explain collectively.

In [19]:
```
lin_reg_predict = linear_regression.predict(housing_test)
print("R^2 Score: ",r2_score(housing_test_labels, lin_reg_predict))
```

```
R^2 Score:  0.6955638499670669
```

The Linear Regression algorithm model we trained can explain about 69.5% of the total variance between the predicted values and the actual values. This number alone doesn't really tell us much about well of a number the model is predicting because of different biases in the model. For instance, some predictions could be really close to the regression line but some could be very far which would explain a R^2 score of 69.5%.

The R^2 score isn't the only metric we can use to evaluate a model. Calulating the model's Mean Absolute Error (MAE) will give us a sum of the difference between the data and our model's prediction (whether it be over predicting or under predicting) for each instance of data. Using the Linear Regression model we can see that our MAE for the entirety of our test data that it was off by over 128,000. MAE isn't always a good metric to use because the under-predicted prices will work to cancel our the over-predicted ones.

In [20]:
```
print("Mean Absolute Error: ", mean_absolute_error(housing_test_labels, lin_reg_predict))
```

```
Mean Absolute Error:  128673.0942688452
```

Yet another metric we can use is the Root Mean Squared Error (RMSE) which squares all of the residual values. This gives us a better overall picture of the model and will penalize our model more heavily for any big outliers in the predictions. The sum is made over all of the residual vs. actual values so that prices that are under-predicted will add on the the total error of the model instead of working with the over-predicted ones like in the MAE metric. The RMSE should actually be higher than what we computed with the MAE and we can see that below with a RMSE score of 214,531.

In [21]:
```
lin_reg_mse = mean_squared_error(housing_test_labels, lin_reg_predict)
lin_reg_rmse = np.sqrt(lin_reg_mse)
print("Root Mean Squared Error: ", lin_reg_rmse)
```

```
Root Mean Squared Error:  214531.24773355035
```

Another evaluation that works well in evaluating a model is it's `confidence interval`. We can gauge performance if we compute a 95% confidence interval. This 95% confidence interval is for the generalization error of the target label. Computing the 95% confidence interval on our Linear Regression model shows that the model is 95% certain the the predicted price of an instance of data will be between 193,852 - 233,384 and have an error no larger than $230,384! This is a huge error and not something that we want our model to be doing.

In [22]:
```
from scipy import stats
confidence = 0.95
squared_errors = (lin_reg_predict - housing_test_labels)**2
print("95% Confidence Level: ", np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1, loc=squared_errors.mea
n(), scale=stats.sem(squared_errors))))
```

```
95% Confidence Level:  [193852.91520895 233384.57483994]
```

We can see here that by using a standard Linear Regression algorithm our model's predictions were off by an average of about $200,000. This is quite a large amount which would never work in the standard business model. It is hard to know exactly why the RMSE was so high sometimes its that there is not enough features or enough strongly correlated features. The linear regression algorithm is not a very powerful algorithm and was likely underfitting the training data which is why we see such a big difference between the predicted priced and actual target labels. We can next try a much more powerful model to see if we can get better performance from it.

The Decision Tree Regressor is a much more powerful algorithm that we can use to see if we can get a better outcome. We will use the same evaluation techniques that we did with our Linear Regression model.

In [23]:
```
from sklearn.tree import DecisionTreeRegressor

tree_regression = DecisionTreeRegressor()
tree_regression.fit(housing, housing_labels)

tree_reg_predict = tree_regression.predict(housing_test)
print("Score: ",r2_score(housing_test_labels, tree_reg_predict))
```

```
Score:  0.7115009595915565
```

In [24]:
```
print("Mean Absolute Error: ", mean_absolute_error(housing_test_labels, tree_reg_predict))

tree_reg_mse = mean_squared_error(housing_test_labels, tree_reg_predict)
tree_reg_rmse = np.sqrt(tree_reg_mse)
print("Root Mean Squared Error: ", tree_reg_rmse)

squared_errors = (tree_reg_predict - housing_test_labels)**2
print("95% Confidence Level: ", np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1, loc=squared_errors.mea
n(), scale=stats.sem(squared_errors))))
```

```
Mean Absolute Error:  105698.45327318991
Root Mean Squared Error:  208840.45685213033
95% Confidence Level:  [171272.3383085  240612.67416067]
```

Using the Decision Tree Regressor our MAE and RMSE performance metrics are down a little bit from the Linear Regression model but it looks like there is still a lot of room for improvement. A more formal evaluation of the model using `k-fold cross-validation` can help us to see if the Decision Tree Regressor model was overfitting the data. For `k-fold cross-validatoin`, we split the training data up into roughly `k` equal pieces. The model is then fitted and evaluated `k` different times then the average and variance of the performance is computed across the training runs. Each run will take out 1 of the `k` fols, train with the data from the other `k-1` folds and evaluate performance on the held out fold. This gives a much better estimate of how well a model wil generalize on unseen data.

In [28]:
```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_regression, housing, housing_labels, scoring='neg_mean_squared_error', cv=10)
tree_rmse_scores = np.sqrt(-scores)

print("Scores: ", scores)
print("Mean: ", scores.mean())
print("Stanard Deviation: ", scores.std())
```

```
Scores:  [-2.82174726e+10 -4.59547707e+10 -3.09898252e+10 -3.37590419e+10
 -3.20648074e+10 -4.00955861e+10 -2.48642057e+10 -3.34804650e+10
 -4.27870614e+10 -2.98706042e+10]
Mean:  -34208384007.385353
Stanard Deviation:  6355606124.193297
```

Wow! You can see here how the Decision Tree Regressor has done very poorly with a cross-validation evaluation. The model is highly overfitting the training data and you can see how well it won't generalize well to the unseen data. This is one of the downfalls to the Decision Tree Regression algorithms.

Lastly, we can the Random Forest Regressor which is actually a collection of many decision trees. This is an ensemble machine learning model and normally exhibits better performance than the single decision trees.

In [29]:
```
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor(n_estimators=100, random_state = 42)
forest_reg.fit(housing, housing_labels)

forest_predict = forest_reg.predict(housing_test)
print("Score: ",r2_score(housing_test_labels, forest_predict))
```

```
Score:  0.8510543751897715
```

In [30]:
```
print("Mean Absolute Error: ", mean_absolute_error(housing_test_labels, forest_predict))

forest_reg_mse = mean_squared_error(housing_test_labels, forest_predict)
forest_reg_rmse = np.sqrt(forest_reg_mse)
print("Root Mean Squared Error: ", forest_reg_rmse)

squared_errors = (forest_predict - housing_test_labels)**2
print("95% Confidence Interval: ", np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1, loc=squared_errors.
mean(), scale=stats.sem(squared_errors))))
```

```
Mean Absolute Error:  73710.66701127964
Root Mean Squared Error:  150057.02204113491
95% Confidence Interval:  [123850.01432183 172323.51458866]
```

We can see from the above metrics that the models we are training are trending in the correct direction but we are not to the point where there are models that could be a confident predictor of housing prices.

Conclusion

After exploring the housing data, cleaning it some, and training a few models the model that were trained didn't end up being useful enough to be able to use it to predict any future housing prices. The models we looked at could have been tweaked some but with how far off our performance metrics were it was unlikely that we would have seen something that would be used.

Further investigation into the data could reveal that we didn't have enough of a linear relationship between the target label and the features we used to train the model. Improvements could be made to our model by dropping more features or engineering new ones. More complete examination of the data might determine that there a polynomal fit to our selected features which would require a different set of algorithms and features that are fit to a polynomial set of theta parameters.