housing_appraisal

June 21, 2021

[5]:
```
import numpy as np

import pandas as pd
from pandas.api.types import CategoricalDtype

%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings("ignore")

import zipfile
import os

plt.rcParams['figure.figsize'] = (12, 9)
plt.rcParams['font.size'] = 12
```

1 Predicting Housing Prices

The goal of this project is to model and predict/estimate property values based on individual house features in the CCAO dataset. The CCAO (Cook County Assessor's Office, Illinois) dataset consists of over 500 thousand records describing houses sold in Cook County in recent years. Property values determine the amount of property tax paid by property owners through out the United States, and an accurate property value estimation is crucial to the fairness behind property tax laying. We will conduct exploratory data analysis, feature engineering, and fitting to create a model that best predict housing prices based on real world data.

[6]:
```
# Extract data from zip file
# Where train split was done by sklearn.model_selection, train_test_split
with zipfile.ZipFile('cook_county_data.zip') as item:
    item.extractall()
```

[8]:
```
training_data = pd.read_csv("cook_county_train.csv", index_col='Unnamed: 0')
test_data = pd.read_csv("cook_county_test.csv", index_col='Unnamed: 0')
```

[16]:
```
# Sale Price is provided in the training data
assert 'Sale Price' in training_data.columns.values
# Sale Price is hidden in the test data
assert 'Sale Price' not in test_data.columns.values
```

[21]:
```
# There are 62 total features included on each house, we are interested in Sale␣
↪Price
training_data.columns.values
```

[21]:
```
array(['PIN', 'Property Class', 'Neighborhood Code', 'Land Square Feet',
       'Town Code', 'Apartments', 'Wall Material', 'Roof Material',
       'Basement', 'Basement Finish', 'Central Heating', 'Other Heating',
       'Central Air', 'Fireplaces', 'Attic Type', 'Attic Finish',
       'Design Plan', 'Cathedral Ceiling', 'Construction Quality',
       'Site Desirability', 'Garage 1 Size', 'Garage 1 Material',
       'Garage 1 Attachment', 'Garage 1 Area', 'Garage 2 Size',
       'Garage 2 Material', 'Garage 2 Attachment', 'Garage 2 Area',
       'Porch', 'Other Improvements', 'Building Square Feet',
       'Repair Condition', 'Multi Code', 'Number of Commercial Units',
       'Estimate (Land)', 'Estimate (Building)', 'Deed No.', 'Sale Price',
       'Longitude', 'Latitude', 'Census Tract',
       'Multi Property Indicator', 'Modeling Group', 'Age', 'Use',
       "O'Hare Noise", 'Floodplain', 'Road Proximity', 'Sale Year',
       'Sale Quarter', 'Sale Half-Year', 'Sale Quarter of Year',
       'Sale Month of Year', 'Sale Half of Year', 'Most Recent Sale',
       'Age Decade', 'Pure Market Filter', 'Garage Indicator',
       'Neigborhood Code (mapping)', 'Town and Neighborhood',
       'Description', 'Lot Size'], dtype=object)
```

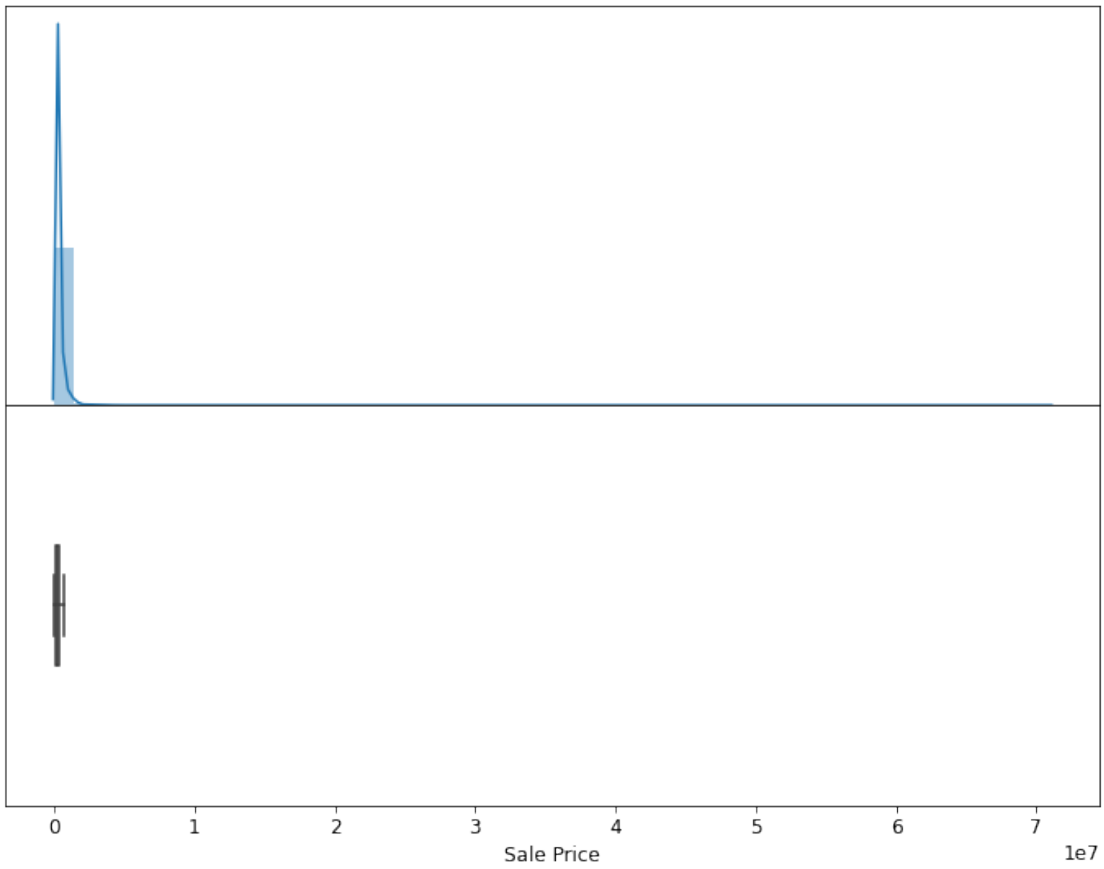
2 Exploratory Data Analysis

Since the variable we are trying to predict is the `SalePrice`, we will plot its distribution using the training data and look at the results

[33]:
```
def plot_distribution(data, label):
    fig, axs = plt.subplots(nrows=2)

    sns.distplot(
        data[label],
        ax=axs[0]
    )
    sns.boxplot(
        data[label],
        width=0.3,
        ax=axs[1],
        showfliers=False,
```

```
    )

    # Align axes
    spacer = np.max(data[label]) * 0.05
    xmin = np.min(data[label]) - spacer
    xmax = np.max(data[label]) + spacer
    axs[0].set_xlim((xmin, xmax))
    axs[1].set_xlim((xmin, xmax))

    # Remove some axis text
    axs[0].xaxis.set_visible(False)
    axs[0].yaxis.set_visible(False)
    axs[1].yaxis.set_visible(False)

    # Put the two plots together
    plt.subplots_adjust(hspace=0)

    # Adjust boxplot fill to be white
    axs[1].artists[0].set_facecolor('white')
```

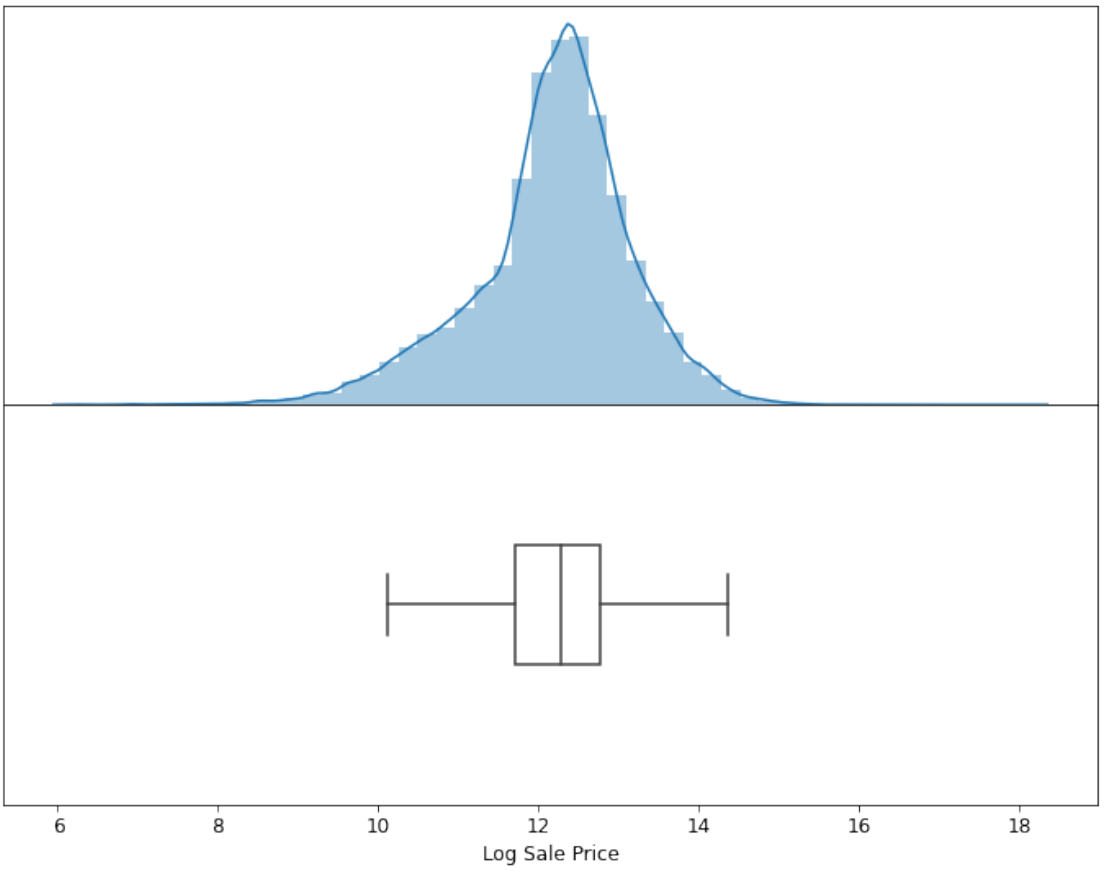
[34]:
```
plot_distribution(training_data, label='Sale Price')
```



It is clear that there exist outliers in our data, so we will perform outlier removal, as well as a log transformation of `SalePrice` to zoom in on the data and plot the data after the transformations.

[37]:
```
def remove_outliers(data, variable, lower=-np.inf, upper=np.inf):
    """
    Input:
      data (data frame): the table to be filtered
      variable (string): the column with numerical outliers
      lower (numeric): observations with values lower than this will be removed
      upper (numeric): observations with values higher than this will be removed

    Output:
      a winsorized data frame with outliers removed

    Note: This function should not change mutate the contents of data.
    """
    return data[(data[variable] >= lower) & (data[variable] <= upper)]
```

[38]:
```
training_data = training_data[training_data['Sale Price'] >= 500]
training_data['Log Sale Price'] = np.log(training_data['Sale Price'])
plot_distribution(training_data, label='Log Sale Price');
```



We can see that the distribution of the Log Sale Price in the training set is left-skewed. The median of the Log Sale Price is also greater than the mean, and at least 25% of the houses in the training set sold for more than $200,000

3 Feature Engineering

We want to take a look at number of bedrooms, Log Sale Price, and Log Building Square Feet to start off with.

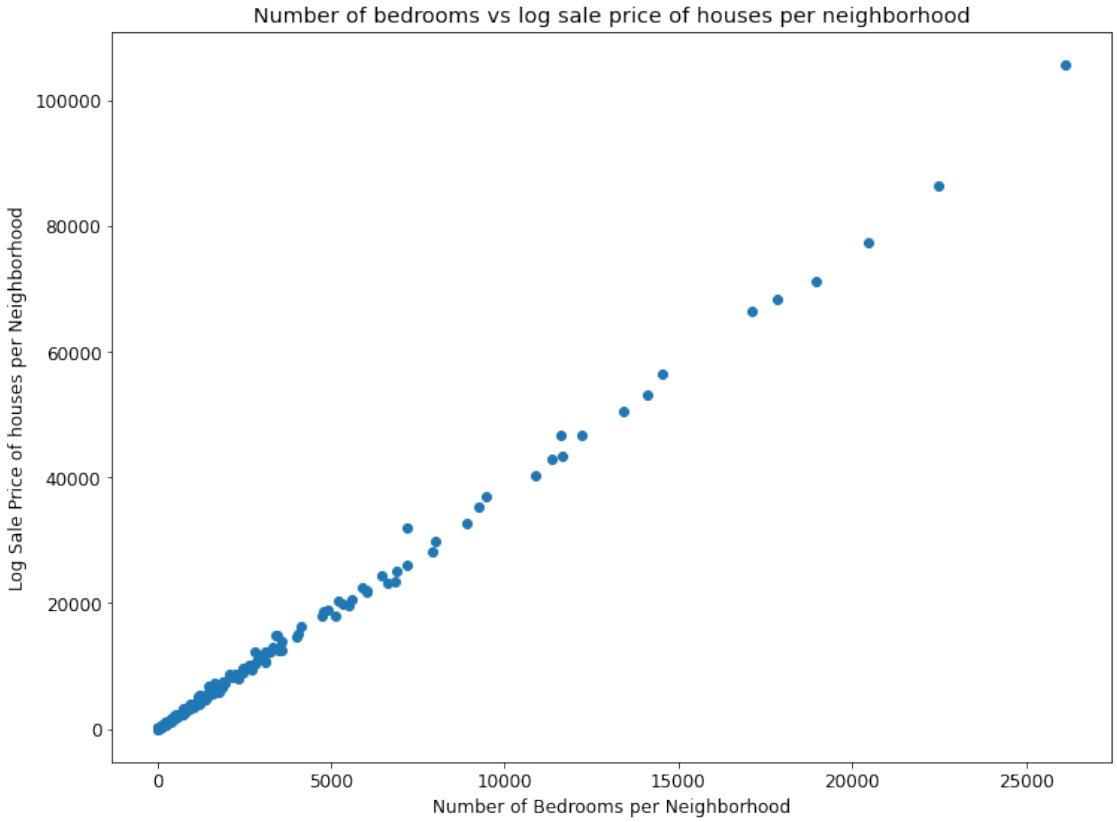
[41]:
```
# The Description column contain information about the number of bedrooms, so␣
↪we will add total bedroom that way
def add_total_bedrooms(data):
    """
    Input:
      data (data frame): a data frame containing at least the Description␣
↪column.
    """
    with_rooms = data.copy()
    with_rooms['Bedrooms'] = data['Description'].str.extract(r'(\d+) of which␣
↪are bedrooms')[0].astype('int')
```

```
    return with_rooms

training_data = add_total_bedrooms(training_data)
```

We create a visualization to see if there is any correlation between number of bedrooms and the property value

[42]:
```
data = training_data.groupby('Neighborhood Code', as_index=False).
↪agg('sum')[['Bedrooms', 'Log Sale Price']]
plt.xlabel('Number of Bedrooms per Neighborhood')
plt.ylabel('Log Sale Price of houses per Neighborhood')
plt.title('Number of bedrooms vs log sale price of houses per neighborhood')
plt.scatter(data['Bedrooms'], data['Log Sale Price'])
```

[42]:
```
<matplotlib.collections.PathCollection at 0x7efecafe16a0>
```



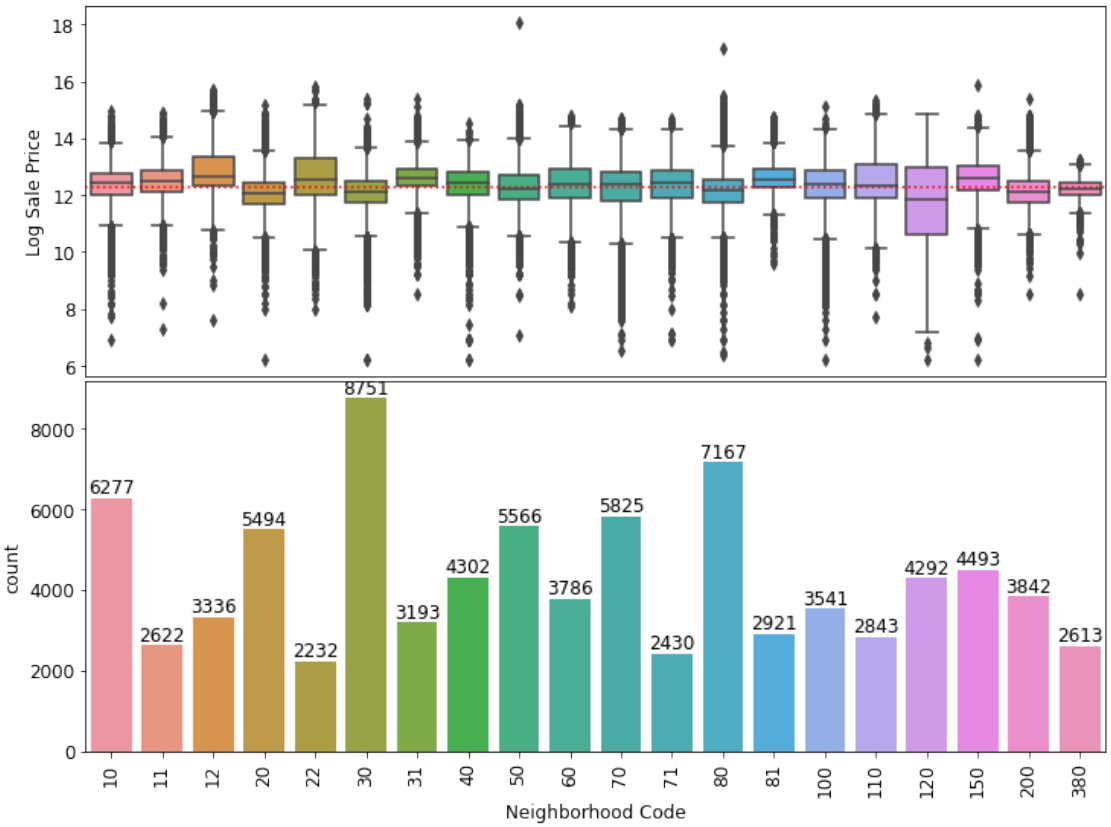
We could also look at expensive neighborhoods and take that into consideration. To avoid over-plotting, we decide to take a look at top 20 most expensive neighborhoods.

[44]:
```
def plot_categorical(neighborhoods, data, with_filter=True):
    if not with_filter:
```

```
        neighborhoods = data
    fig, axs = plt.subplots(nrows=2)

    sns.boxplot(
        x='Neighborhood Code',
        y='Log Sale Price',
        data=neighborhoods.sort_values('Neighborhood Code'),
        ax=axs[0],
    )

    sns.countplot(
        x='Neighborhood Code',
        data=neighborhoods.sort_values('Neighborhood Code'),
        ax=axs[1],
    )

    # Draw median price
    axs[0].axhline(
        y=data['Log Sale Price'].median(),
        color='red',
        linestyle='dotted'
    )

    # Label the bars with counts
    for patch in axs[1].patches:
        x = patch.get_bbox().get_points()[:, 0]
        y = patch.get_bbox().get_points()[1, 1]
        axs[1].annotate(f'{int(y)}', (x.mean(), y), ha='center', va='bottom')

    # Format x-axes
    axs[1].set_xticklabels(axs[1].xaxis.get_majorticklabels(), rotation=90)
    axs[0].xaxis.set_visible(False)

    # Narrow the gap between the plots
    plt.subplots_adjust(hspace=0.01)
```

[45]:
```
nei = training_data['Neighborhood Code'].value_counts().head(20).index
top_20_neighborhoods = training_data[training_data['Neighborhood Code'].
 ↪isin(nei)]
```

[46]:
```
plot_categorical(neighborhoods=top_20_neighborhoods, data=training_data)
```



[47]:
```
def find_expensive_neighborhoods(data, n=3, metric=np.median):
    """
    Input:
      data (data frame): should contain at least a string-valued Neighborhood
        and a numeric 'Sale Price' column
      n (int): the number of top values desired
      metric (function): function used for aggregating the data in each␣
↪neighborhood.
        for example, np.median for median prices

    Output:
      a list of the top n richest neighborhoods as measured by the metric␣
↪function
    """
    neighborhoods = data.groupby('Neighborhood Code').agg(metric).
↪sort_values(by='Log Sale Price', ascending=False).head(n).index

    # This makes sure the final list contains the generic int type used in␣
↪Python3, not specific ones used in numpy.
    return [int(code) for code in neighborhoods]
```

```
expensive_neighborhoods = find_expensive_neighborhoods(training_data, 3, np.
↪median)
expensive_neighborhoods
```

[47]: [44, 94, 93]

4 Modeling

We will utilize our `triaining_data` to perform the actual training and testing of our model, and reserve `testing_data` for the final evaluation. We perfom another train test split below.

[55]:
```
def train_test_split(data):
    data_len = data.shape[0]
    shuffled_indices = np.random.permutation(data_len)
    return data.iloc[shuffled_indices[:round(data_len * 0.8) - 1], :], data.
↪iloc[shuffled_indices[round(data_len * 0.8) - 1 :], :]
```

We will use the ordinary least squares estimator to fit a linear regression model. Starting off with something simple by using only 2 features: the **number of bedrooms** in the household and the **log-transformed total area covered by the building** (in square feet).

1st model choice

$$\text{Log Sale Price} = \theta_0 + \theta_1 \cdot (\text{Bedrooms})$$

2nd model choice

$$\text{Log Sale Price} = \theta_0 + \theta_1 \cdot (\text{Bedrooms}) + \theta_2 \cdot (\text{Log Building Square Feet})$$

[53]:
```
def process_data_gm(data, pipeline_functions, prediction_col):
    """Process the data for a guided model."""
    for function, arguments, keyword_arguments in pipeline_functions:
        if keyword_arguments and (not arguments):
            data = data.pipe(function, **keyword_arguments)
        elif (not keyword_arguments) and (arguments):
            data = data.pipe(function, *arguments)
        else:
            data = data.pipe(function)
    X = data.drop(columns=[prediction_col]).to_numpy()
    y = data.loc[:, prediction_col].to_numpy()
    return X, y

def select_columns(data, *columns):
    """Select only columns passed as arguments."""
    return data.loc[:, columns]
```

[54]:
```
# Reload and preprocess the data for 1st model
full_data = pd.read_csv("cook_county_train.csv")
np.random.seed(1337)
train, test = train_test_split(full_data)

train, test = remove_outliers(train, 'Sale Price', lower=500),␣
↪remove_outliers(test, 'Sale Price', lower=500)
train['Log Sale Price'] = np.log(train['Sale Price'])
test['Log Sale Price'] = np.log(test['Sale Price'])
train, test = add_total_bedrooms(train)[['Log Sale Price', 'Bedrooms']],␣
↪add_total_bedrooms(test)[['Log Sale Price', 'Bedrooms']]


m1_train = process_data_gm(train, [], 'Log Sale Price')
m1_test = process_data_gm(test, [], 'Log Sale Price')

X_train_m1, y_train_m1, X_test_m1, y_test_m1 = m1_train[0], m1_train[1],␣
↪m1_test[0], m1_test[1]


np.random.seed(1337)


# Reload and preprocess the data for 2nd model
full_data = pd.read_csv("cook_county_train.csv")
train, test = train_test_split(full_data)

train, test = remove_outliers(train, 'Sale Price', lower=500),␣
↪remove_outliers(test, 'Sale Price', lower=500)
train['Log Sale Price'] = np.log(train['Sale Price'])
test['Log Sale Price'] = np.log(test['Sale Price'])
train['Log Building Square Feet'] = np.log(train['Building Square Feet'])
test['Log Building Square Feet'] = np.log(test['Building Square Feet'])

train = add_total_bedrooms(train)[['Log Sale Price', 'Bedrooms', 'Log Building␣
↪Square Feet']]
test = add_total_bedrooms(test)[['Log Sale Price', 'Bedrooms', 'Log Building␣
↪Square Feet']]

m2_train = process_data_gm(train, [], 'Log Sale Price')
m2_test = process_data_gm(test, [], 'Log Sale Price')

X_train_m2, y_train_m2, X_test_m2, y_test_m2 = m2_train[0], m2_train[1],␣
↪m2_test[0], m2_test[1]
```

[56]:
```
from sklearn import linear_model as lm

linear_model_m1 = lm.LinearRegression(fit_intercept=True)
linear_model_m2 = lm.LinearRegression(fit_intercept=True)
```

[57]:
```
# Fit the 1st model
linear_model_m1.fit(X_train_m1, y_train_m1)
# Compute the fitted and predicted values of Sale Price for 1st model
y_fitted_m1 = linear_model_m1.predict(X_train_m1)
y_predicted_m1 = linear_model_m1.predict(X_test_m1)

# Fit the 2nd model
linear_model_m2.fit(X_train_m2, y_train_m2)
# Compute the fitted and predicted values of Sale Price for 1st model
y_fitted_m2 = linear_model_m2.predict(X_train_m2)
y_predicted_m2 = linear_model_m2.predict(X_test_m2)
```

5 Model Evaluation

We will use the Root Mean Squared Error function to evaluate the performance of these 2 models.

[59]:
```
def rmse(predicted, actual):
    """
    Calculates RMSE from actual and predicted values
    Input:
      predicted (1D array): vector of predicted/fitted values
      actual (1D array): vector of actual values
    Output:
      a float, the root-mean square error
    """
    return np.sqrt(np.mean((actual - predicted)**2))
```

[ ]:
```
# Training and test errors for the 1st model
training_error_m1 = rmse(y_fitted_m1, train['Log Sale Price'])
test_error_m1 = rmse(y_predicted_m1, test['Log Sale Price'])

# Training and test errors for the 1st model (in its original values before the␣
 ↪log transform)
training_error_m1_delog = rmse(np.exp(y_fitted_m1), np.exp(train['Log Sale␣
 ↪Price']))
test_error_m1_delog = rmse(np.exp(y_predicted_m1), np.exp(test['Log Sale␣
 ↪Price']))


# Training and test errors for the 2nd model
training_error_m2 = rmse(y_fitted_m2, train['Log Sale Price'])
```

```
test_error_m2 = rmse(y_predicted_m2, test['Log Sale Price'])


# Training and test errors for the 2nd model (in its original values before the␣
 ↪log transform)
training_error_m2_delog = rmse(np.exp(y_fitted_m2), np.exp(train['Log Sale␣
 ↪Price']))
test_error_m2_delog = rmse(np.exp(y_predicted_m2), np.exp(test['Log Sale␣
 ↪Price']))

print("1st Model\nTraining RMSE: {}\nTest RMSE: {}\n".format(training_error_m1,␣
 ↪test_error_m1))
print("1st Model (no log transform)\nTraining RMSE: {}\nTest RMSE: {}\n".
 ↪format(training_error_m1_delog, test_error_m1_delog))
print("2nd Model\nTraining RMSE: {}\nTest RMSE: {}\n".format(training_error_m2,␣
 ↪test_error_m2))
print("2nd Model (no log transform)\nTraining RMSE: {}\nTest RMSE: {}\n".
 ↪format(training_error_m2_delog, test_error_m2_delog))
```

[61]:
```
# Parameters from 1st model
theta0_m1 = linear_model_m1.intercept_
theta1_m1 = linear_model_m1.coef_[0]

# Parameters from 2nd model
theta0_m2 = linear_model_m2.intercept_
theta1_m2, theta2_m2 = linear_model_m2.coef_

print("1st Model\n 0: {}\n 1: {}".format(theta0_m1, theta1_m1))
print("2nd Model\n 0: {}\n 1: {}\n 2: {}".format(theta0_m2, theta1_m2,␣
 ↪theta2_m2))
```

```
1st Model
 0: 10.571725401040084
 1: 0.4969197463141442
2nd Model
 0: 1.9339633173823714
 1: -0.030647249803554506
 2: 1.4170991378689641
```

We see that there is probably omitted variable bias in the first model. θ_1 in model 1 tries to account for every feature that's not just bedrooms. In model 2, number of bedrooms and log building square feet is probably related (more bedrooms equals more number of log building square feet), so the coefficients are trying to account for this.

6 Modeling Continue

This time, we will look at `Log Sale Price`, `Log Land Square Feet`, `Log Age` as features for our model.

[63]:
```
def process_data_fm(data):
    data = remove_outliers(data, 'Sale Price', 500, 230000)
    data = data.dropna()
    data['Log Sale Price'] = np.log(data['Sale Price'])
    data['Log Land Square Feet'] = np.log(data['Land Square Feet'])
    data['Log Age'] = np.log(data['Age'])
    # Return predictors and response variables separately
    X = select_columns(data, 'Log Land Square Feet', 'Log Age', 'Lot Size')
    y = data.loc[:, 'Log Sale Price']
    return X, y
```

[64]:
```
training_data = pd.read_csv("cook_county_train.csv", index_col='Unnamed: 0')
final_model = lm.LinearRegression(fit_intercept=True)
X_train, y_train = process_data_fm(training_data)

final_model.fit(X_train, y_train)
y_predicted_train = final_model.predict(X_train)

training_rmse = rmse(np.exp(y_predicted_train), np.exp(y_train))
training_rmse
```

[64]:
```
68261.90883295538
```

[65]:
```
final_model.coef_
```

[65]:
```
array([ 2.52024591e-01, -2.88630261e-01, -6.08147009e-06])
```