Homework 6: Predicting Housing Prices (Continued)

Due Date: 11:59 PM Tuesday, July 30

Collaboration Policy

Data science is a collaborative activity. While you may talk with others about the homework, we ask that you **write your solutions individually**. If you do discuss the assignments with others please **include their names** in the collaborators cell below.

Collaborators: *write names here*

Introduction

This assignment will continue from where we left off in in Homework 5. Recall that the linear model that you created failed to produce accurate estimates of the observed housing prices because the model was too simple. The goal of this homework is to guide you through the iterative process of specifying, fitting, and analyzing the performance of more complex linear models used to predict prices of houses in Ames, Iowa. Additionally, you will have the opportunity to choose your own features and create your own regression model!

By the end of this homework, you should feel comfortable:

1. Identifying informative variables through EDA
2. Feature engineering categorical variables
3. Using sklearn to build more complex linear models

Score Breakdown

Question	Points
Question 1a	1
Question 1b	1
Question 1c	1
Question 2a	1
Question 2b	2
Question 3a	1
Question 3b	2
Question 3c	1
Question 3d	2
Question 4	6
Question 5a	2
Question 5b	2
Total	22

In [1]:

```
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
```

In [2]:

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

The Data

As a reminder, the Ames dataset (http://jse.amstat.org/v19n3/decock.pdf) consists of 2930 records taken from the Ames, Iowa, Assessor’s Office describing houses sold in Ames from 2006 to 2010. The data set has 23 nominal, 23 ordinal, 14 discrete, and 20 continuous variables (and 2 additional observation identifiers) --- 82 features in total. An explanation of each variable can be found in the included `codebook.txt` file. The information was used in computing assessed values for individual residential properties sold in Ames, Iowa from 2006 to 2010.

The raw data are split into training and test sets with 2000 and 930 observations, respectively. To save some time, we've used a slightly modified data cleaning pipeline from last week's assignment to prepare the training data. This data is stored in `ames_train_cleaned.csv`. It consists of 1998 observations and 83 features (we added TotalBathrooms from Homework 5).

In [4]:

```
training_data = pd.read_csv("ames_train_cleaned.csv")
```

Part IV: More Feature Selection and Engineering

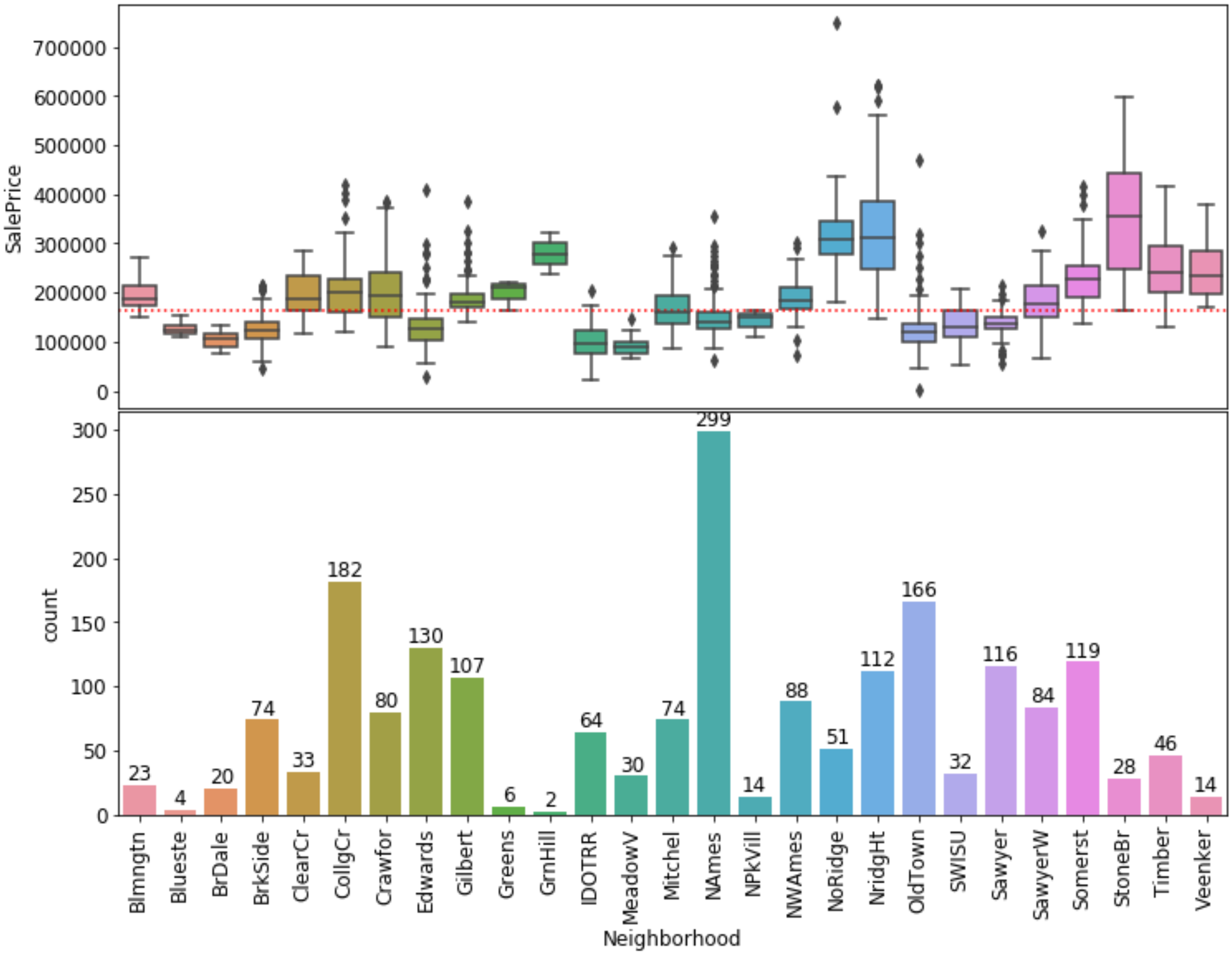
In this section, we identify two more features of the dataset that will increase our linear regression model's accuracy. Additionally, we will implement one-hot encoding so that we can include binary and categorical variables in our improved model.

Question 1: Neighborhood vs Sale Price

First, let's take a look at the relationship between neighborhood and sale prices of the houses in our data set.

In [5]:

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

sns.boxplot(
    x='Neighborhood',
    y='SalePrice',
    data=training_data.sort_values('Neighborhood'),
    ax=axs[0]
)

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

# Draw median price
axs[0].axhline(
    y=training_data['SalePrice'].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)
```



Question 1a

Based on the plot above, what can be said about the relationship between the houses' sale prices and their neighborhoods?

```
BEGIN QUESTION
name: q1a
points: 1
manual: True
```

SOLUTION: It is clear that there is quite some variation in prices across neighborhoods. Moreover, the amount of data available is not uniformly distributed among neighborhoods. North Ames, for example, comprises almost 15% of the training data while Green Hill has a scant 2 observations in this data set.

Question 1b

One way we can deal with the lack of data from some neighborhoods is to create a new feature that bins neighborhoods together. Let's categorize our neighborhoods in a crude way: we'll take the top 3 neighborhoods measured by median `SalePrice` and identify them as "rich neighborhoods"; the other neighborhoods are not marked.

Write a function that returns list of the top n most pricy neighborhoods as measured by our choice of aggregating function. For example, in the setup above, we would want to call `find_rich_neighborhoods(training_data, 3, np.median)` to find the top 3 neighborhoods measured by median `SalePrice`.

The provided tests check that you answered correctly, so that future analyses are not corrupted by a mistake.

```
BEGIN QUESTION
name: q1b
points: 1
```

In [6]:

```
def find_rich_neighborhoods(data, n=3, metric=np.median):
    """
    Input:
      data (data frame): should contain at least a string-valued Neighborhood
        and a numeric SalePrice column
      n (int): the number of top values desired
      metric (function): function used for aggregating the data in each neighbor
hood.
        for example, np.median for median prices

    Output:
      a list of the top n richest neighborhoods as measured by the metric functi
on
    """
    neighborhoods = ...
    # BEGIN SOLUTION NO PROMPT
    neighborhoods = list(
        data
        .groupby('Neighborhood')['SalePrice']
        .aggregate(metric)
        .sort_values(ascending=False)
        .head(n)
        .index.values
    )
    # END SOLUTION
    return neighborhoods

rich_neighborhoods = find_rich_neighborhoods(training_data, 3, np.median)
rich_neighborhoods
```

Out[6]:

```
['StoneBr', 'NridgHt', 'NoRidge']
```

In [7]:

```
# TEST
len(find_rich_neighborhoods(training_data, 5, np.median))
```

Out[7]:

```
5
```

In [8]:

```
# TEST
isinstance(rich_neighborhoods, list)
```

Out[8]:

```
True
```

In [9]:

```
# TEST
all([isinstance(neighborhood, str) for neighborhood in rich_neighborhoods])
```

Out[9]:

```
True
```

In [10]:

```
# TEST
set(rich_neighborhoods) == set(['StoneBr', 'NridgHt', 'NoRidge']) # Check to see
if correct neighborhoods identified
```

Out[10]:

```
True
```

In [11]:

```
# TEST
set(find_rich_neighborhoods(training_data, 2, np.min)) == set(['GrnHill', 'NoRid
ge'])
```

Out[11]:

```
True
```

Question 1c

We now have a list of neighborhoods we've deemed as richer than others. Let's use that information to make a new variable `in_rich_neighborhood`. Write a function `add_rich_neighborhood` that adds an indicator variable which takes on the value 1 if the house is part of `rich_neighborhoods` and the value 0 otherwise.

Hint: [`pd.Series.astype` (https://pandas.pydata.org/pandas-docs/version/0.23.4/generated/pandas.Series.astype.html)](https://pandas.pydata.org/pandas-docs/version/0.23.4/generated/pandas.Series.astype.html) may be useful for converting True/False values to integers.

The provided tests check that you answered correctly, so that future analyses are not corrupted by a mistake.

```
BEGIN QUESTION
name: q1c
points: 1
```

In [12]:

```
def add_in_rich_neighborhood(data, neighborhoods):
    """
    Input:
      data (data frame): a data frame containing a 'Neighborhood' column with va
lues
        found in the codebook
      neighborhoods (list of strings): strings should be the names of neighborho
ods
        pre-identified as rich
    Output:
      data frame identical to the input with the addition of a binary
      in_rich_neighborhood column
    """
    data['in_rich_neighborhood'] = ...
    data['in_rich_neighborhood'] = data['Neighborhood'].isin(neighborhoods).asty
pe('int32') # SOLUTION NO PROMPT
    return data

rich_neighborhoods = find_rich_neighborhoods(training_data, 3, np.median)
training_data = add_in_rich_neighborhood(training_data, rich_neighborhoods)
```

In [13]:

```
# TEST
sum(training_data.loc[:, 'in_rich_neighborhood'])
```

Out[13]:

191

In [14]:

```
# TEST
sum(training_data.loc[:, 'in_rich_neighborhood'].isnull())
```

Out[14]:

0

In [15]:

```
# TEST
sum(add_in_rich_neighborhood(training_data, ['NAmes']).loc[:, 'in_rich_neighborh
ood'])
```

Out[15]:

299

Question 2: Fireplace Quality

In the following question, we will take a closer look at the Fireplace_Qu feature of the dataset and examine how we can incorporate categorical features into our linear model.

Question 2a

Let's see if our data set has any missing values. Create a Series object containing the counts of missing values in each of the columns of our data set, sorted from greatest to least. The Series should be indexed by the variable names. For example, `missing_counts['Fireplace_Qu']` should return 975.

Hint: [`pandas.DataFrame.isnull` (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.isnull.html)](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.isnull.html) may help here.

The provided tests check that you answered correctly, so that future analyses are not corrupted by a mistake.

```
BEGIN QUESTION
name: q2a
points: 1
```

In [16]:

```
missing_counts = training_data.isnull().sum().sort_values(ascending=False) # SOL
UTION
missing_counts
```

Out[16]:

```
Pool_QC          1991
Misc_Feature     1922
Alley            1865
Fence            1607
Fireplace_Qu      975
Lot_Frontage      352
Garage_Cond       114
Garage_Yr_Blt     114
Garage_Finish     114
Garage_Qual       114
Garage_Type       113
Bsmt_Exposure      57
BsmtFin_Type_2     56
Bsmt_Cond          56
Bsmt_Qual          56
BsmtFin_Type_1     56
Mas_Vnr_Type       18
Mas_Vnr_Area       18
BsmtFin_SF_1        1
BsmtFin_SF_2        1
Bsmt_Unf_SF         1
Total_Bsmt_SF       1
```

```
Total_Bsmt_SF          1
Electrical             1
Land_Slope             0
Exter_Cond             0
Exterior_2nd           0
Exter_Qual             0
MS_Zoning              0
Foundation             0
Neighborhood           0
                    ...
Sale_Condition         0
Sale_Type              0
Yr_Sold                0
Mo_Sold                0
Misc_Val               0
Pool_Area              0
Screen_Porch           0
3Ssn_Porch             0
Enclosed_Porch         0
Open_Porch_SF          0
Wood_Deck_SF           0
Paved_Drive            0
Garage_Area            0
Fireplaces             0
TotalBathrooms         0
Functional             0
TotRms_AbvGrd          0
Kitchen_Qual           0
Kitchen_AbvGr          0
Bedroom_AbvGr          0
Half_Bath              0
Full_Bath              0
Bsmt_Half_Bath         0
Bsmt_Full_Bath         0
Gr_Liv_Area            0
Low_Qual_Fin_SF        0
2nd_Flr_SF             0
1st_Flr_SF             0
Central_Air            0
Order                  0
Length: 84, dtype: int64
```

In [17]:

```
# TEST
isinstance(missing_counts, pd.Series)
```

Out[17]:

```
True
```

In [18]:

```
# TEST
missing_counts.size # Should have 84 total features (82 features + TotalBathroom
s + in_rich_neighborhood)
```

Out[18]:

```
84
```

In [19]:

```
# TEST
set(missing_counts.index.values) == set(training_data.columns.values)
```

Out[19]:

```
True
```

In [20]:

```
# TEST
missing_counts.loc['Fireplace_Qu'] # Make sure you are calculating the counts co
rrectly
```

Out[20]:

```
975
```

In [21]:

```
# TEST
missing_counts.iloc[0] # Make sure you are sorting correctly
```

Out[21]:

```
1991
```

It turns out that if we look at the codebook carefully, some of these "missing values" aren't missing at all! The Assessor's Office just used `NA` to denote a special value or that the information was truly not applicable for one reason or another. One such example is the `Fireplace_Qu` variable.

```
FireplaceQu (Ordinal): Fireplace quality

       Ex   Excellent - Exceptional Masonry Fireplace
       Gd   Good - Masonry Fireplace in main level
       TA   Average - Prefabricated Fireplace in main living area or Mason
ry Fireplace inbasement
       Fa   Fair - Prefabricated Fireplace in basement
       Po   Poor - Ben Franklin Stove
       NA   No Fireplace
```

Question 2b

An `NA` here actually means that the house had no fireplace to rate. Let's fix this in our data set. Write a function that replaces the missing values in `Fireplace_Qu` with `'No Fireplace'`. In addition, it should replace each abbreviated condition with its full word. For example, `'TA'` should be changed to `'Average'`. Hint: the [DataFrame.replace (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.replace.html)](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.replace.html) method may be useful here.

The provided tests check that part of your answer is correct, but they are not fully comprehensive.

```
BEGIN QUESTION
name: q2b
points: 2
```

In [22]:

```
def fix_fireplace_qu(data):
    """
    Input:
      data (data frame): a data frame containing a Fireplace_Qu column.  Its val
ues
                         should be limited to those found in the codebook
    Output:
      data frame identical to the input except with a refactored Fireplace_Qu co
lumn
    """
    # BEGIN SOLUTION
    replacements = {
        'Fireplace_Qu': {
            'Ex': 'Excellent',
            'Gd': 'Good',
            'TA': 'Average',
            'Fa': 'Fair',
            'Po': 'Poor',
            np.nan: 'No Fireplace'
        }
    }

    data = data.replace(replacements)
    # END SOLUTION
    return data

training_data = fix_fireplace_qu(training_data)
```

In [23]:

```
# TEST
sum(training_data['Fireplace_Qu'] == 'No Fireplace') # Make sure you've replaced all the missing values with 'No Fireplace'
```

Out[23]:

```
975
```

In [24]:

```
# TEST
sum(training_data.loc[:, 'Fireplace_Qu'].isnull() == 0) # Make sure you haven't introduced anything strange
```

Out[24]:

```
1998
```

In [25]:

```
# TEST
sum(training_data.loc[:, 'Fireplace_Qu'] == 'Excellent')
```

Out[25]:

```
30
```

An Important Note on One Hot Encoding

Unfortunately, simply fixing these missing values isn't sufficient for using `Fireplace_Qu` in our model. Since `Fireplace_Qu` is a categorical variable, we will have to one-hot-encode the data. Notice in the example code below that we have to pre-specify the categories. Why? Imagine what would happen if we automatically generated the categories only from the training data. What would happen if the testing data contained a category not found in the training set? For more information on categorical data in pandas, refer to this link (https://pandas-docs.github.io/pandas-docs-travis/categorical.html). **Note that `get_dummies` removes the original column.**

In [26]:

```python
def ohe_fireplace_qu(data):
    """
    One-hot-encodes fireplace quality.  New columns are of the form fpq_QUALITY
    """
    cats = [
        'Excellent',
        'Good',
        'Average',
        'Fair',
        'Poor',
        'No Fireplace'
    ]

    cat_type = CategoricalDtype(categories=cats)

    data.loc[:, 'Fireplace_Qu'] = data.loc[:, 'Fireplace_Qu'].astype(cat_type)
    data = pd.get_dummies(data,
                          prefix='fpq',
                          columns=['Fireplace_Qu'],
                          drop_first=True)

    return data
```

In [27]:

```python
training_data = ohe_fireplace_qu(training_data)
training_data.filter(regex='fpq').head(10)
```

Out[27]:

	fpq_Good	fpq_Average	fpq_Fair	fpq_Poor	fpq_No Fireplace
0	1	0	0	0	0
1	0	0	0	0	1
2	0	0	0	0	1
3	0	1	0	0	0
4	0	1	0	0	0
5	1	0	0	0	0
6	0	0	0	0	1
7	0	1	0	0	0
8	0	0	0	0	1
9	1	0	0	0	0

Part V: Improved Linear Models

In this section, we will create linear models that produce more accurate estimates of the housing prices in Ames than the model created in Homework 5, but at the expense of increased complexity.

Question 3: Adding Covariates to our Model

It's finally time to fit our updated linear regression model using the ordinary least squares estimator! Our new model consists of the linear model from Homework 5, with the addition of the our newly created `in_rich_neighborhood` variable and our one-hot-encoded fireplace quality variables:

$$\begin{aligned}\text{SalePrice} = \theta_0 + \theta_1 \cdot \text{Gr\_Liv\_Area} + \theta_2 \cdot \text{Garage\_Area} + \theta_3 \cdot \text{TotalBathrooms} + \theta_4 \cdot \text{in\_rich\_neighb} \\ \theta_5 \cdot \text{fpq\_Good} + \theta_6 \cdot \text{fpq\_Average} + \theta_7 \cdot \text{fpq\_Fair} + \theta_8 \cdot \text{fpq\_Poor} + \theta_9 \cdot \text{fpq\_No\_Firepl}\end{aligned}$$

Question 3a

Although the firepalce quality variable that we explored in Question 2 has six categories, only five of these categories' indicator variables are inclulded in our model. Is this a mistake, or is it done intentionally? Why?

```
BEGIN QUESTION
name: q3a
points: 1
manual: True
```

SOLUTION: If we wish to compute the ordinary least square estimate of our coefficients analytically, then the the design matrix must be full rank. If each of the fireplace quality variables's six categories were represented by binary variables in our model, then the design matrix would not be full rank since the covariates of our model would not be linearly independent.

Question 3b

We still have a little bit of work to do prior to estimating our linear regression model's coefficients. Instead of having you go through the process of splitting our data into training and testing sets, selecting the pertinent convariates and creating a [`sklearn.linear_model.LinearRegression` (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html)](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html) object for our linear model again, we provide the necessary code from Homework 5.

First, we will re-import the data and split `training_data` into a training and test set.

In [28]:

```
training_data = pd.read_csv("ames_train_cleaned.csv")

# This makes the train-test split in this section reproducible across different runs
# of the notebook. You do not need this line to run train_test_split in general
np.random.seed(1337)
training_data_len = len(training_data)
shuffled_indices = np.random.permutation(training_data_len)

# Set train_indices to the first 80% of shuffled_indices and and test_indices to the rest.
train_indices = shuffled_indices[:int(training_data_len * 0.8)]
test_indices = shuffled_indices[int(training_data_len * 0.8):]

# Create train and test` by indexing into `full_data` using
# `train_indices` and `test_indices`
train = training_data.iloc[train_indices]
test = training_data.iloc[test_indices]
```

Next, we will implement a reusable pipeline that selects the required variables in our data and splits our covariates and response variable into a matrix and a vector, respectively.

In [29]:

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

def process_data_gm(data):
    """Process the data for a guided model."""
    # One-hot-encode fireplace quality feature
    data = fix_fireplace_qu(data)
    data = ohe_fireplace_qu(data)

    # Use rich_neighborhoods computed earlier to add in_rich_neighborhoods featu
re
    data = add_in_rich_neighborhood(data, rich_neighborhoods)

    # Transform Data, Select Features
    data = select_columns(data,
                          'SalePrice',
                          'Gr_Liv_Area',
                          'Garage_Area',
                          'TotalBathrooms',
                          'in_rich_neighborhood',
                          'fpq_Good',
                          'fpq_Average',
                          'fpq_Fair',
                          'fpq_Poor',
                          'fpq_No Fireplace',
                         )

    # Return predictors and response variables separately
    X = data.drop(['SalePrice'], axis = 1)
    y = data.loc[:, 'SalePrice']

    return X, y
```

We then split our dataset into training and testing sets using our data cleaning pipeline.

In [30]:

```
# Pre-process our training and test data in exactly the same way
# Our functions make this very easy!
X_train, y_train = process_data_gm(train)
X_test, y_test = process_data_gm(test)
train.head()
```

Out[30]:

	Order	PID	MS_SubClass	MS_Zoning	Lot_Frontage	Lot_Area	Street	Al
1504	2229	909475050	20	RL	NaN	20693	Pave	Na
1589	2349	527355060	60	RL	81.0	10530	Pave	Na
1617	2392	528138030	60	RL	85.0	11924	Pave	Na
835	1247	535302130	20	RL	102.0	9373	Pave	Na
1704	2520	533253050	120	RL	36.0	3640	Pave	Na

5 rows × 83 columns

Finally, we initialize a `sklearn.linear_model.LinearRegression` (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html) object as our linear model. We set the `fit_intercept = True` to ensure that the linear model has a non-zero intercept.

In [31]:

```
from sklearn import linear_model as lm

linear_model = lm.LinearRegression(fit_intercept=True)
```

After a little bit of work, it's finally time to fit our updated linear regression model. Use the cell below to estimate the model, and then use it to compute the fitted value of `SalePrice` over the training data and the predicted the values of `SalePrice` using the testing data.

The provided tests check that you answered correctly, so that future analyses are not corrupted by a mistake.

```
BEGIN QUESTION
name: q3b
points: 2
```

In [32]:

```
# Fit the model below
linear_model.fit(X_train, y_train)  # SOLUTION NO PROMPT

# Compute the fitted and predicted values of SalePrice
y_fitted = linear_model.predict(X_train) # SOLUTION
y_predicted = linear_model.predict(X_test) # SOLUTION
```

In [33]:

```
# TEST
180220 <= y_fitted.mean() <= 180225
```

Out[33]:

```
True
```

In [34]:

```
# TEST
181492 <= y_predicted.mean() <= 181512
```

Out[34]:

```
True
```

Question 3c

Let's assess the performance of our new linear regression model using the Root Mean Squared Error function that we created in Homework 5.

$$RMSE = \sqrt{\frac{\sum_{\text{houses in test set}}(\text{actual price for house} - \text{predicted price for house})^2}{\text{\# of houses}}}$$

The function is provided below.

In [35]:

```
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))
```

Now use your `rmse` function to calculate the training error and test error in the cell below.

The provided tests for this question do not confirm that you have answered correctly; only that you have assigned each variable to a non-negative number.

```
BEGIN QUESTION
name: q3c
points: 1
```

In [36]:

```
training_error = rmse(y_fitted, y_train) # SOLUTION
test_error = rmse(y_predicted, y_test) # SOLUTION
print("Training RMSE: {}\nTest RMSE: {}".format(training_error, test_error))
```

```
Training RMSE: 40491.84911146645
Test RMSE: 38754.860681844264
```

In [37]:

```
# TEST
training_error > 0
```

Out[37]:

```
True
```

In [38]:

```
# TEST
test_error > 0
```

Out[38]:

```
True
```

In [39]:

```
# HIDDEN TEST
np.isclose(training_error, 40491.849, atol=0.1)
```

Out[39]:

```
True
```

In [40]:

```
# HIDDEN TEST
np.isclose(test_error, 38754.86068, atol=0.1)
```

Out[40]:

```
True
```

Question 3d

Compare the predictive accuracy of this model to that of the model that you derived in Homework 5. Is the new model a better predictor of housing prices in Ames? If so, are the gains in accuracy significantly larger? Assume that the training and testing sets used to in Homework 5 are identical to the ones used in this homework.

```
BEGIN QUESTION
name : q3d
points: 2
manual: True
```

SOLUTION: The test RMSE of the new model's predictions is 38754.8606, and the test RMSE of Homework 5's model's predictions is 46146.643. Therefore, we conclude that our new model is more accurate. However, the relative change of RMSE between models is $\approx 16\%$. Since we know that the original model in Homework 5 did a poor job of predicting house sale prices, this signifies that there is a lot of room for improvement.

Part VI: Open-Response

The following part is purposefully left nearly open-ended. The Ames data in your possession comes from a larger data set. Your goal is to provide a linear regression model that accurately predicts the prices of the held-out homes, measured by root mean square error.

$$RMSE = \sqrt{\frac{\sum_{\text{houses in public test set}} (\text{actual price for house} - \text{predicted price for house})^2}{\text{\# of houses}}}$$

Perfect prediction of house prices would have a score of 0, so you want your score to be as low as possible!

Grading Scheme

Your grade for Question 4 will be based on your training RMSE and test RMSE. The thresholds are as follows:

Points	**3**	**2**	**1**	**0**
Training RMSE	Less than 36k	36k - 38k	38k - 40k	More than 40k

Points	**3**	**2**	**1**	**0**
Test RMSE	Less than 37k	37k - 40k	40k - 43k	More than 43k

One Hot Encoding

If you choose to include more categorical features in your model, you'll need to one-hot-encode each one. It may be helpful to read more about the arguments to `pd.get_dummies` (http://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html), which can actually handle `NaN` values and multiple categorical variables at once.

Also, remember that if a categorical variable has a unique value that is present in the training set but not in the test set, one-hot-encoding this variable will result in different outputs for the training and test sets (different numbers of one-hot columns). Watch out for this! Feel free to look back at how we one-hot-encoded `Fireplace_Qu` and specified the categories beforehand.

To generate all possible categories for a categorical variable, we suggest reading through `codebook.txt`, or finding the values programmatically across both the training and test datasets.

Question 4: Your Own Linear Model

Just as in the guided model above, you should encapsulate as much of your workflow into functions as possible. Below, we have initialized `final_model` for you. Your job is to select better features and define your own feature engineering pipeline in `process_data_fm`.

To evaluate your model, we will process training data using your `process_data_fm`, fit `final_model` with this training data, and compute the training RMSE. Then, we will process the test data with your `process_data_fm`, use `final_model` to predict sale prices for the test data, and compute the test RMSE. See below for an example of the code we will run to grade your model:

```
training_data = pd.read_csv('ames_train_cleaned.csv')
test_data = pd.read_csv('ames_test_cleaned.csv')

X_train, y_train = process_data_fm(training_data)
X_test, y_test = process_data_fm(test_data)

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

training_rmse = rmse(y_predicted_train, y_train)
test_rmse = rmse(y_predicted_test, y_test)
```

Note: It is your duty to make sure that all of your feature engineering and selection happens in `process_data_fm`, and that the function performs as expected without errors. We will **NOT** accept regrade requests that require us to go back and run code that require typo/bug fixes.

Hint: Some features may have missing values in the test set but not in the training set. Make sure `process_data_fm` handles missing values appropriately for each feature!

```
BEGIN QUESTION
name: q4
points: 6
```

In [41]:

```
final_model = lm.LinearRegression(fit_intercept=True) # No need to change this!

def process_data_fm(data):
    # BEGIN SOLUTION
    # list(test_data.select_dtypes(include=[np.number]).columns.values) # shows
all numeric features
    data = add_in_rich_neighborhood(data, rich_neighborhoods)

    # Transform Data, Select Features
    data = select_columns(data,
                          'SalePrice',
                          'Gr_Liv_Area',
                          'Garage_Area',
                          'in_rich_neighborhood',
                          'Lot_Area',
                          'Year_Built',
                          'Fireplaces',
                          'Overall_Qual'
                         )
    # END SOLUTION
    # Return predictors and response variables separately
    X = data.drop(['SalePrice'], axis = 1)
    y = data.loc[:, 'SalePrice']
    X = X.fillna(0) # SOLUTION NO PROMPT
    X = (X - np.mean(X, axis=0)) / (np.std(X, axis=0) + 0.00001) # SOLUTION NO P
ROMPT
    return X, y
```

In [42]:

```
# TEST
def rmse(predicted, actual):
    return np.sqrt(np.mean((actual - predicted)**2))

training_data = pd.read_csv('ames_train_cleaned.csv')
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(y_predicted_train, y_train)

training_rmse <= 40000
```

Out[42]:

True

In [43]:

```
# TEST
def rmse(predicted, actual):
    return np.sqrt(np.mean((actual - predicted)**2))

training_data = pd.read_csv('ames_train_cleaned.csv')
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(y_predicted_train, y_train)

training_rmse <= 38000
```

Out[43]:

```
True
```

In [44]:

```
# TEST
def rmse(predicted, actual):
    return np.sqrt(np.mean((actual - predicted)**2))

training_data = pd.read_csv('ames_train_cleaned.csv')
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(y_predicted_train, y_train)

training_rmse <= 36000
```

Out[44]:

```
True
```

In [45]:

```
# TEST
def rmse(predicted, actual):
    return np.sqrt(np.mean((actual - predicted)**2))

training_data = pd.read_csv('ames_train_cleaned.csv')
test_data = pd.read_csv('ames_test_cleaned.csv')

X_train, y_train = process_data_fm(training_data)
X_test, y_test = process_data_fm(test_data)

X_test_public, y_test_public = X_test[::2], y_test[::2]
X_test_private, y_test_private = X_test[1::2], y_test[1::2]

final_model.fit(X_train, y_train)
y_predicted_train = final_model.predict(X_train)
y_predicted_test_public = final_model.predict(X_test_public)
y_predicted_test_private = final_model.predict(X_test_private)

training_rmse = rmse(y_predicted_train, y_train)
public_test_rmse = rmse(y_predicted_test_public, y_test_public)
private_test_rmse = rmse(y_predicted_test_private, y_test_private)

public_test_rmse <= 43000
```

Out[45]:

```
True
```

In [46]:

```python
# TEST
def rmse(predicted, actual):
    return np.sqrt(np.mean((actual - predicted)**2))

training_data = pd.read_csv('ames_train_cleaned.csv')
test_data = pd.read_csv('ames_test_cleaned.csv')

X_train, y_train = process_data_fm(training_data)
X_test, y_test = process_data_fm(test_data)

X_test_public, y_test_public = X_test[::2], y_test[::2]
X_test_private, y_test_private = X_test[1::2], y_test[1::2]

final_model.fit(X_train, y_train)
y_predicted_train = final_model.predict(X_train)
y_predicted_test_public = final_model.predict(X_test_public)
y_predicted_test_private = final_model.predict(X_test_private)

training_rmse = rmse(y_predicted_train, y_train)
public_test_rmse = rmse(y_predicted_test_public, y_test_public)
private_test_rmse = rmse(y_predicted_test_private, y_test_private)

public_test_rmse <= 40000
```

Out[46]:

```
True
```

In [47]:

```
# TEST
def rmse(predicted, actual):
    return np.sqrt(np.mean((actual - predicted)**2))

training_data = pd.read_csv('ames_train_cleaned.csv')
test_data = pd.read_csv('ames_test_cleaned.csv')

X_train, y_train = process_data_fm(training_data)
X_test, y_test = process_data_fm(test_data)

X_test_public, y_test_public = X_test[::2], y_test[::2]
X_test_private, y_test_private = X_test[1::2], y_test[1::2]

final_model.fit(X_train, y_train)
y_predicted_train = final_model.predict(X_train)
y_predicted_test_public = final_model.predict(X_test_public)
y_predicted_test_private = final_model.predict(X_test_private)

training_rmse = rmse(y_predicted_train, y_train)
public_test_rmse = rmse(y_predicted_test_public, y_test_public)
private_test_rmse = rmse(y_predicted_test_private, y_test_private)

public_test_rmse <= 37000
```

Out[47]:

```
True
```

Question 5: EDA for Feature Selection

In the following question, explain a choice you made in designing your custom linear model in Question 4. First, make a plot to show something interesting about the data. Then explain your findings from the plot, and describe how these findings motivated a change to your model.

Question 5a

In the cell below, create a visualization that shows something interesting about the dataset.

```
BEGIN QUESTION
name: q5a
points: 2
manual: True
format: image
```

In [48]:

```
# Code for visualization goes here
# BEGIN SOLUTION
# END SOLUTION
```

Question 5b

Explain any conclusions you draw from the plot above, and describe how these conclusions affected the design of your model. After creating the plot, did you add/remove certain features from your model, or did you perform some other type of feature engineering? How significantly did these changes affect your rmse?

```
BEGIN QUESTION
name: q5b
points: 2
manual: True
```

SOLUTION: Explain the plot and why its relevant.

Before You Submit

Make sure that if you run Kernel > Restart & Run All, your notebook produces the expected outputs for each cell. Congratulations on finishing the assignment!