Module 15 - Regression Models in Python

Author: Khal Makhoul, W.P.G.Peterson, Mona Khalil, Greg Gordeau

Reviewer: Jessica Cervi

Expected time = 2.5 hours

Total points = 55 points

Assignment Overview

Ordinary Least Squares (OLS) regression is a closed-form and easily interpretable model used for predicting outcomes in continuous data. It gives you the ability to clearly select the most powerful predictors of an outcome and make clear and actionable decisions based on those predictions.

This assignment will test your ability to implement an OLS (ordinary least squares) regression in Python. We'll briefly review some of the lecture content, followed by an overarching research question that will be guiding this assignment. Throughout the assignment, you will be asked to use the popular `scikit-learn` and `statsmodels` libraries to implement your OLS regression. You will also create several functions throughout the assignment in order to resolve problems and roadblocks to your analysis.

This assignment is designed to build your familiarity and comfort coding in Python while also helping you review key topics from each module. As you progress through the assignment, answers will get increasingly complex. It is important that you adopt a data scientist's mindset when completing this assignment. **Remember to run your code from each cell before submitting your assignment.** Running your code beforehand will notify you of errors and give you a chance to fix your errors before submitting. You should view your Vocareum submission as if you are delivering a final project to your manager or client.

Vocareum Tips

- Do not add arguments or options to functions unless you are specifically asked to. This will cause an error in Vocareum.
- Do not use a library unless you are expicitly asked to in the question.
- You can download the Grading Report after submitting the assignment. This will include feedback and hints on incorrect questions.

Learning Objectives

- Test your proficiency in using Python and `pandas`
- Examine the mathematical foundations behind least squares regression
- Perform OLS regression on a dataset
- Use a range of Python libraries and functions available for OLS regression

Index:

Module 15- Regression Models in Python

- Question 1
- Question 2
- Question 3
- Question 4
- Question 5
- Question 6
- Question 7
- Question 8
- Question 9
- Question 10
- Question 11

Module 15 - Regression Models in Python

Ordinary least squares and linear regression

Ordinary least squares (OLS) regression is the most commonly used method for conducting a linear regression. This model attempts to find a line that minimizes the squared distance between data points and the line ("least squares"). OLS regression offers a clear method for choosing one or more *predictors* (or *features*) that predict a single outcome in a linear manner.

Linear regressions in general are widely used across industries including healthcare, economics, and social sciences. You can expect to frequently encounter data science questions in your career for which a linear regression is the best possible solution.

Regression Equation

A regression analysis yields an equation similar to the slope of a line, which is a mathematical representation of the shape of how your input variables predict your output variables. It's often presented as follows:

$$Y = \alpha + \beta_1 X_1 + \beta_2 X_2 + e$$

Y, representing a value for your outcome variable, is predicted by the slope of the line α (alpha) plus a coefficient, β (beta), multiplied by each X value. The resulting equation explains an approximation of a line, similar to the one you see below.

slope

This is similar to $Y = mx + b$, the equation for the slope of a line that you probably learned in grade school math.

Statistical Assumptions

There are several key steps to take in order to produce accurate regression analysis results. Namely, your data needs to meet key *statistical assumptions*:

- **Linearity:** your data is linearly related, or can be transformed to create a linear relationship (i.e., take the square root of a predictor).
- **Multivariate normality:** the residuals produced by your output are normally distributed.
- **Little or no multicollinearity:** your predictors are independent and not highly correlated with each other.
- **Homoscedasticity:** equal variance of errors.

We will cover all of these in detail as we analyze the data set.

For this assignment, we will use the `housing_prices.csv` dataset. We explore our data using the `head()` and `info()` methods.

In [ ]:
```python
# Load packages
import numpy as np
import pandas as pd

# Load the data
housing = pd.read_csv('./data/housing_prices.csv')

# Explore the data
housing.head()
```

Back to top

Question 1:

5 points

How many rows and columns are in the `housing` data set? Assign your answer as integers values to the tuple `ans1` below.

In [ ]:
```python
### GRADED

### YOUR SOLUTION HERE
ans1= (housing.shape[0],  housing.shape[1])
### END SOLUTION
```


Selecting Features

Regression models predict the slope of a line based on a single outcome variable predicted by any number of features. In order to generate accurate predictions and actionable conclusions, it's important to select features that meet the following criteria:

- **Strong**, linear relationship with the outcome variable.
- **Actionable**: if we can conclude that x predicts y, we need to be able to demonstrate actions to take that will impact the value of y. In the case of our housing data, we should be able to say that altering some feature of the house will have a specific impact on the price.
- **Minimal**: your model includes only features that strongly predict X, and include as few features as is necessary to draw strong conclusions.
- **Independent**: not strongly correlated with other predictors in the model. This is referred to as *collinearity* , and will be discussed further in the next steps.

So how do you start narrowing down the features?

First, let's take a look at the `housing.info()` results. There are several columns with *a lot* of missing values (i.e., more than 20%). Data with a large number of missing values limit our ability to draw conclusions. Further, it's likely that columns with significant missing values may be *missing not at random* (i.e., houses in rural areas may be more likely to be missing data on one feature than those in urban areas). Patterns in missing data can heavily skew the results of conclusions you are trying to draw.

Back to top

Question 2:

5 points

Drop all columns in the dataframe `housing` with more than 20% missing data. Re-assign the new dataframe to `housing`.

In [ ]:
```python
### GRADED

### YOUR SOLUTION HERE
housing.dropna(axis=1, thresh=int(housing_ans.shape[0] * 0.2), inplace=True)
### END SOLUTION
```

Next, let's take a look at the columns with categorical data. Since this is a *linear* model, input data needs to be continuous data or a limited number of categories that can be dummy coded (more on this later). Take a look at the unique values for the columns categorized as objects.

You can use `.select_dtypes()` to select only non-numeric columns. In order to explore the number of unique values, try using the `agg()` method with the *count* and *nunique* values.

In [ ]:
```python
housing_cts = housing.select_dtypes(include = 'object').agg(['count', 'nunique']).T
housing_cts.sort_values(by = ['nunique'], ascending = True).head(10)
```

There are quite a few categorical columns with a large number of unique values. We will cover some ways in which we can use categorical predictors in linear models, but to use all of these would be beyond the purview of this assignment and invite potential violations to our statistical assumptions.

Back to top

Question 3:

5 points

Drop all the columns with more than 3 unique values.

In [ ]:
```python
### GRADED

### YOUR SOLUTION HERE
housing.drop(columns=housing_cts[housing_cts['nunique'] > 3].index,axis=1,inplace=True)
### END SOLUTION
```

We now have 43 features to explore as potential predictors of the house's sale price. We're getting closer to a manageable number of predictors for a clear, actionable regression model. Let's continue narrowing down the scope of features by focusing on our numerical, *continuous* data. We can do this by examining *Pearson's correlations* between potential features and the outcome variable (housing price).

Pearson's correlations are single value numerical summaries that represent the *strength* and *direction* of a linear relationship. Correlation values range from -1 to 1. Values further away from 0 represent *stronger* relationships, and the sign of the correlation (positive or negative) represents the *direction* of the relationship. For example, there is likely a positive correlation between the number of bedrooms in a house and its sale price, and a negative correlation between the number of neighborhood robberies and sale prices. The graphs below depict a visual representation of Pearson correlations.

pearson-1-small.png

Python allows us to easily generate *correlation matrices* based on a pandas DataFrame using the `corr()` method.

In [ ]:
```python
housing_corr = housing.select_dtypes(include = ['int64', 'float64']).corr()
np.round(housing_corr, 2)
```

The correlation matrix generated by the `.corr()` method generates a very large table representing correlations between every single combination of features (it's a lot larger than can be easily represented in Jupyter Notebooks, so it will appear on multiple lines). You can find the correlation between any two features from your original data set by matching the variable name of one feature in the rows, and the other in the columns. You'll notice the values of `1.00` present along the diagonal running down across the matrix -- this is the value representing each feature's correlation with itself.

We're primarily interested in how all house features are correlated with the sale price of the house, which is stored in the final column of the correlation matrix. Let's isolate and print it in the code cell below, sorted by the Pearson's correlation value.

Back to top

Question 4:

5 points

From `housing_corr`, extract the column identifying all correlations with the `SalePrice` column (*including* `SalePrice`), and sort the values in descending order. Assign the resulting pandas series to `housing_sales_corr`.

In [ ]:
```python
### GRADED

### YOUR SOLUTION HERE
housing_sales_corr = housing_corr['SalePrice'].sort_values(ascending=False)
### END SOLUTION
```

Interpreting Correlation Values

Exactly how close to 1 or negative 1 do correlation values need to be in order to be included in a regression analysis?

In general, researchers consider the following values to be approximate cutoffs representing the size of a correlation:

- +/- 0.3: weak correlation; is likely statistically significant with a sufficient sample size
- +/- 0.5: moderate correlation
- +/- 0.7: strong correlation

Below is a visual representation of different strengths of correlational relationships:

correlations

Depending on the research question, number of available features, and strength of other features, researchers may either set a hard cutoff of correlation values (i.e. +/- 0.25) or evaluate them individually. Since we have quite a few features to choose from, let's filter out all features that are correlated with `SalePrice` at less than +/- 0.4. We can easily generate a list of features to include by further subsetting our `housing_sales_corr` series.

Back to top

Question 5:

5 points

Filter the `housing_sales_corr` pandas series to exclude all features with a correlation value of less than the *absolute value* of 0.6. This means that you are excluding all features between -0.6 and +0.6.

You can do this a number of ways. For example, you can use the pandas function `where()` that takes, as argument, a lambda function specifing your tolerance; and then drop the undesired values with `dropna()`. Assign the result to the series object `housing_features`.

In [ ]:
```python
### GRADED

### YOUR SOLUTION HERE
housing_features  = housing_sales_corr.where(lambda x: abs(x) > 0.6 ).dropna()
### END SOLUTION
```

We're now down to 13 numerical features to include in our regression analysis. We will likely refine this list further as we explore how the features relate to each other. For now, let's drop all other numerical columns from the data set.

Back to top

Question 6:

5 points

Drop all of the numerical features that have a weak correlation value (in this case, between -0.6 and 0.6) with the outcome variable `SalePrice`.

You will have to modify the logical operators from the previous question (i.e., >, <, |, &) in order to keep the correct columns!

In [ ]:
```python
### GRADED

### YOUR SOLUTION HERE
housing.drop(housing_sales_corr.where(lambda x: abs(x) < 0.6 ).dropna().index, axis=1, inpla
ce=True)
### END SOLUTION
```

We're down to 6 potential continuous features for our regression analysis. Now that we've narrowed down our features to a manageable number, let's start examining them in detail to determine which features are useful, actionable, and do not significantly correlate with each other. This often occurs when your data set measures several, related features (i.e., a feature such as "total bedrooms" will likely correlate with "total rooms overall" and "total square feet". It's important to choose which features are the best fit for our model.

In [ ]:
```python
housing.corr()
```

Take a look at the following pairs of columns:

- `GarageCars` and `GarageArea`
- `TotalBsmtSF` and `1stFlrSF`

Each of these pairs has a correlation of more than 0.8, which is much stronger than either column's correlation with `SalePrice`.

In order to ensure we meet our model's assumption of no collinearity, we should drop one column from each of these pairs.

Back to top

Question 7:

5 points

Drop the two columns `['1stFlrSF', 'GarageArea']`.

In [ ]:
```python
### GRADED

### YOUR SOLUTION HERE
housing.drop('1stFlrSF',1)
housing.drop('GarageArea',1)
### END SOLUTION
```

When preparing to train a regression model, it's important to keep in mind that the cutoff of correlations between features is subjective. There is no specific, objective cutoff that determines which features you should retain or drop before training your model. Instead, it's important for you to evaluate the importance of each predictor in answering your question and how important it is in relation to other available features.

Encoding Categorical Features

With additional processing, categorical features can be included as predictors in a regression model. This requires a process called *dummy coding* , where multiple categories in one column are transformed into multiple, binary columns. Take a look at the table below.

id	GarageFinish
1	RFn
2	RFn
3	Unf
4	RFn
5	Unf
6	RFn
7	RFn
8	Unf
9	Unf
10	Fin

There are 3 unique categorical values -- **RFn** (rough finish), **Unf** (unfinished), and **Fin** (finished). Using pandas, the table can be transformed to create a new separate column for each categorical option. A value of `1` indicates a positive result for that categorical option (i.e., house 3 has an unfinished garage). Below is an example of the result of a dummy coded transformation of a categorical variable.

id	GarageFinish	G_RFn	G_Unf	G_Fin
1	RFn	1	0	0
2	RFn	1	0	0
3	Unf	0	1	0
4	RFn	1	0	0
5	Unf	0	1	0
6	RFn	1	0	0
7	RFn	1	0	0
8	Unf	0	1	0
9	Unf	0	1	0
10	Fin	0	0	1

Let's select all of the categorical columns in the data set once more to determine which ones we should keep and transform using dummy coding.

In [ ]:
```python
housing.select_dtypes(include = 'object').agg(['count', 'nunique']).transpose()
```

For all features with 2 unique values, we can simply change them to binary values (i.e., 1, and 0). Let's do this for the `Central Air` column.

In [ ]:
```python
print('Original Values:',housing['CentralAir'].unique())
housing['CentralAir'] = [1 if x == 'Y' else 0 for x in housing['CentralAir']]
print('New Values:',housing['CentralAir'].unique())
```

Columns with more than 2 values require some additional processing. Unless your data is *ordinal* in nature (i.e., a ranking), you will need to *dummy code* these variables.

Let's transform the `GarageFinish` column as we just demonstrated above using `pd.get_dummies()`. The new columns are automatically appended to the end of the original `housing` dataframe, and the original `GarageFinish` column is dropped.

In [ ]:
```python
housing = pd.get_dummies(housing, columns = ['GarageFinish'], prefix = 'G')
housing.head()
```

Back to top

Question 8:

5 points

Create dummy variables for the `PavedDrive` column with the prefix `PV`. Append it to the original `housing` data set and drop `PavedDrive` once you are done.

In [ ]:
```python
### GRADED

### YOUR SOLUTION HERE
housing= pd.get_dummies(housing, columns = ['PavedDrive'], prefix = 'PV')
### END SOLUTION
```

We have one final step before analyzing our data. There are 3 more categorical columns -- `Street`, `Utilities`, and `LandSlope` that we will exclude from our model. We're looking to create a model that provides important, *actionable* information for homeowners looking to increase the sale price of their home. It's unlikely that you can alter the slope of the land, pavement of the street, or utility company. Thus, we will drop them in this stage.

In [ ]:
```python
housing = housing.drop(['Street', 'Utilities', 'LandSlope'], axis = 'columns')

housing.head()
```


Splitting our Data Frame

We've done a lot of preprocessing for our OLS regression, and we're almost ready to run the analysis! The next key step is to split our dataframe into separate X and y frames. In order to properly run our analysis, we will need our input variables (X) to be a separate object from our outcome variable (y).

In [ ]:
```python
X = housing.drop(['SalePrice'], axis = 'columns')
y = housing['SalePrice']
```

Next, let's split our data frame into a training and test set. This is a technique you will use across the majority of models you create in your data science journey as a means of validating the results of your analysis. A model is trained on your *training* set, and then validated on your *test* set to ensure the model was not **overfitted** to your training set. A good model fits your training set well, but isn't only fitted to the variation in one data set -- it also strongly predicts information given new data!

Scikit-learn has a class called `train_test_split` that will randomly split the data set's records according to the proportions we specify. Data Scientists will commonly use 20-30% of their data to test their model, and 70-80% for training. We'll use 30% of our `housing` data set for the test set, and specify a `random_state` of `1111` so that we can return to our analysis and replicate the random split later on.

In [ ]:
```python
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.30, random_state=1111)
```


Training our Model

It's time to train our model! We'll first do this using `scikit-learn`, and later on compare the results to `statsmodels`. OLS Regression can be conducted using `scikit-learn`'s `LinearRegression` class. Let's import that now and create an object of the `LinearRegression` class.

In [ ]:
```python
from sklearn.linear_model import LinearRegression

lr = LinearRegression()
```

We can then *train* our linear regression by fitting the `X_train` and `y_train` data sets.

In [ ]:
```python
lr.fit(X_train, y_train)
```

The model fitted to the training and test set now has a set of coefficients we can retrieve using the `coef_` method. These are the beta values for the regression equation representing the line of best fit generated using the least squares method. These coefficients are used to *predict* future housing sales prices given the features we included in our model.

In [ ]:
```python
lr.coef_
```

We can then use our `X_test` set to generate a list of predicted values to compare to the actual values in `y_test`. This is how we determine the accuracy of our model and its effectiveness at predicting new data.

In [ ]:
```python
y_pred = lr.predict(X_test)
```


Evaluating our Model

There are several commonly used methods and metrics for evaluating a model. Let's first evaluate the model's R^2, which is an indication of the *percent of variability in the outcome variable explained by the model's features*. We can generate and compare this value for the training and test sets.

In [ ]:
```python
training_score = lr.score(X_train, y_train)
print(training_score)
```

The R^2 value of 0.76 indicates that our features explain approximately 76% of the variability in sale prices in the training set.

Back to top

Question 9:

5 points

Evaluate the test set. Assign the result to `test_score`. Assign the result to `test_score` and print it to the console.

In [ ]:
```python
### GRADED

### YOUR SOLUTION HERE
test_score = lr.score(X_test, y_test)
print(test_score)
### END SOLUTION
```

Our model's test set R^2 score is also fairly high! It seems we've developed a robust model for predicting the sale price of a house. Let's keep evaluating the quality of our model by examining the **mean squared error** and R^2 comparing the `y_test` and `y_pred` values.

Mean squared error is the *average of the squared errors*. The larger the value, the larger the error. Let's print the R^2 and mean squared error comparing the `y_test` values to the `y_pred` predicted values. `sklearn.metrics` contains many metrics for evaluating a model, including `mean_squared_error` and `r2_score`.

In [ ]:
```python
from sklearn.metrics import mean_squared_error, r2_score

print('R-square:',r2_score(y_pred, y_test))
print('MSE:',mean_squared_error(y_pred, y_test))
```

The R-square comparing the `y_pred` and `y_test` scores is high, indicating that the predictions are fairly accurate. However, the mean squared error is also very high, suggesting there is a lot of *error* in our model. This is often due to the presence of features that are not strong predictors of the outcome variable. Let's see what features we can remove from our model, retrain it, and see if the MSE decreases.

We can easily evaluate the contribution of individual predictors using the `statsmodels` package, which offers an alternative method for fitting a regression model. Let's import the `statsmodels.api` package as its common alias, `sm`.

In [ ]:
```python
import statsmodels.api as sm
```

Next, let's train a model using the `OLS` class in statsmodels. We can use the same `X_train` and `y_train` data sets that we've used so far to fit the model. Afterward, use the `summary()` method to print the model statistics.

In [ ]:
```python
model = sm.OLS(y_train, X_train).fit()
predictions = model.predict(X_test)

model.summary()
```

As you can see, `statsmodels` offers a lot more detailed information in its summary in order to evaluate different components of the model. If you'll observe the second table, you'll see one row for each predictor in the model, and several metrics evaluating the predictor's contribution to the regression model's predictive power.

The fourth column, `P>|t|`, provides a `p` value similar to the t-test we covered in a previous week. Some features have p-values above the common threshold of .05, indicating they are likely not significant predictors of a house's sale price, or they overlap with other predictors in the model too significantly. When refining the model, you can go back to drop non-significant predictors and then retrain your model.

Back to top

Question 10:

5 points

How many predictors are *non-significant* , and should likely be dropped when fine-tuning your model? Assign your answer to `ans10` below.

In [ ]:
```python
### GRADED

### YOUR SOLUTION HERE
ans10 = len(model.pvalues.where(lambda x: x > 0.05).dropna().index)
### END SOLUTION
```


Plotting Predictions

In our last step, let's plot the relationship between our `y_pred` and `y_test` values. This is an important exercise in understanding where the predictions are accurate and where your model may be falling short.

In [ ]:
```python
import matplotlib.pyplot as plt
%matplotlib inline

plt.scatter(y_pred, y_test);
```

Back to top

Question 11:

5 points

Which of the following describes the relationship between the y predictions and test values? Assign the letter corresponding to your choice as a string to `ans11` below.

- a) There is a strong, linear relationship.
- b) There is a strong, curvilinear relationship with overestimated `y_pred` values.
- c) There is a strong, curvilinear relationship with underestimated `y_pred` values.
- d) There is a weak overall relationship.

In [ ]:
```python
### GRADED

### YOUR SOLUTION HERE
ans11 = 'c'
### END SOLUTION
```

In [ ]: