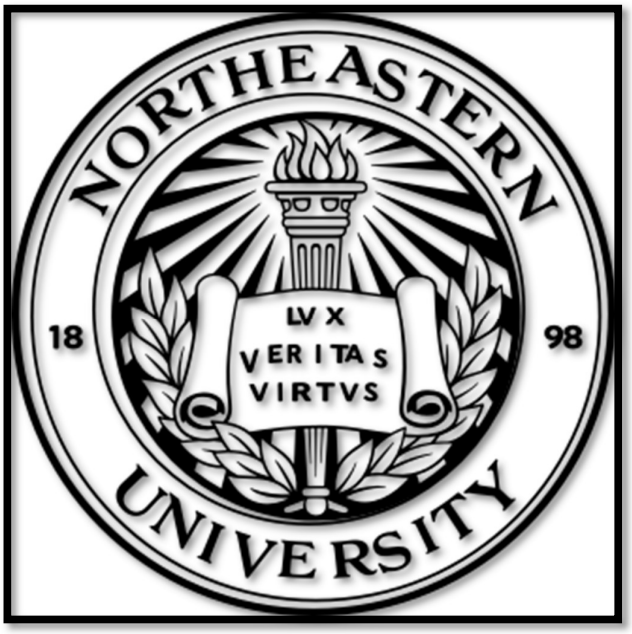

Module 2: Linear Regression

Submitted to Dr. Mary Donhoffner

By Ananya Sharma
Neu Id: 002954987

Introduction

The purpose of the data set is to understand the miles per gallon given by the different cars. Everyone wants to cover more distances with less fuel, and we are trying to understand how the various car features help us in seeking good mileage.
The data set has 398 rows and 8 columns. The data has a space complexity of 25.0 + KB. The info () function categorizes the data types majorly into float, int, and object data types. The describe function helps with the statistical analysis and forms the basis of the study. We can see that the Miles per Gallon increases with the number of cylinders. Likewise, the acceleration also increases. A glimpse into the summary of the dataset helps us to perform EDA on these aspects and check the variables for the concept of multicollinearity.

Horsepower, which is an object variable type is further subjected to closer study and using the unique function we figure out that it has some symbols that need to be removed using replace function.
Missing values are checked for and we find that we don't have any missing values.

Exploratory Data Analysis (Part 1)

The purpose of exploratory data analysis is to discover some details about the data set and to prepare our models better. Furthermore, it helps to clean the data set and bring forth a clean structure. We will use the seaborn library to create some graphs.

A pair plot (Figure 1 in the appendix) helps to see the distribution of a single variable and the relationship between the two variables. (Koehrsen, 2018). Model Year and US-made variables don't seem to be giving us so much information and can be removed from the analysis.

The Violin plot (Figure 4 in the Appendix) shows that the acceleration has a normal distribution. Also, the cars seem to be averaging 15mps acceleration. A lot of cars have 4-5 cylinders.

The scatterplot shows the relation between Miles per gallon and acceleration. We can see from figure 2 that as the number of the cylinder increases the miles per gallon reduces.
The next plot (Figure 3 in the appendix) shows a further study into the relationship between miles per gallon and the number of cylinders. Cars with five cylinders give the most optimum mpg. The next scatter plot shows that the weight of the cylinder affects the weight of the car and henceforth the mileage. Henceforth we may not like to pick a vehicle with more cylinders.

A histplot (Figure 5) is drawn to show how the various variables are distributed. The data shows that the pace of the vehicle is evenly spread, with the bulk of the cars advancing at a pace of 15 MPs. There are 52% more automobiles with four cylinders than there is overall. Our outcome variable (mpg) has a modest rightward skew. Additionally, we may observe that the scales of our parameters differ.

Next, we plot a heat map to understand the correlation between Miles per gallon and the other variables. Checking Figure 6(Appendix) tells us that Miles per gallon has a positive

relationship with the other parameters. For instance, Cylinder and Weight have a +ve correlation of 0.9. This aspect also satisfies the first assumption for our Linear Regression Model. The association between the displacement, horsepower, weight, and cylinders is substantially inverse. This suggests that the mpg drops as any one of those variables climb. A strong significant association between displacement, horsepower, weight, and cylinders contravenes the linear regression's non-multicollinearity premise. We use the Variance Inflation Factor to understand and henceforth remove the variables Displacement, Cylinders, and Weight (Variance Inflation Factor (VIF), 2022)

We need to use feature selection to limit some of these possibilities. Figure 8 shows how well we have been able to reduce the variance due to multicollinearity and bring it down substantially.MPG has reduced from 5.696 to 4.02 post applying the VIF function.

The add_constant () function is helpful in fitting the model for Ordinary Linear Regression.

Training the Model (Part 2)

In this part, we train the linear regression models and try to see which one is better. The data set is divided into two parts namely test and train. Here, the sklearn. model_selection function is used to perform the data splitting. Preprocessing. scale () brings all the variables to the same magnitude. Scaling is essential here since the linear regression model is known to penalize the coefficients. After this, we fit the linear regression model and get a test score of 0.736 and a train score of 0.754. The overall accuracy of the model is 0.736

We aim to get a more accurate model by fitting the ridge and the lasso regression model for optimization.

Standardizing the variables in ridge regression is done by subtracting their means and dividing them by their standard deviations. In Figure 11(in the appendix) we can see when we compare it with figure 10(Linear Regression) that the train, test, and overall score look pretty much the same. So, this model has not optimized the scales phenomenally.

Furthermore, we use the grid-search function to get the best variables that can help us to get good miles per gallon. We explore the Lasso Model since the previous Ridge model was not so conclusive.

The Lasso model also has similar test train and overall scores. We move on to the next ensemble method. (Owusu, 2021). We use a Decision Tree, Random Forest, and gradient boosting approach to trim the linear regression model.

Upon running the Decision Tree, we get an improved train score of 0.876.

The test score and the overall has increased marginally to 0.76. This is better than the previous three models.

Considering the Random Forest Model (Figure 19), we see that the model accuracy has increased to 94.4% and the overall accuracy comes to 86% approximately.

Finally, Gradient Boosting (Figure 21) gives us an overall model accuracy of 84.48% approximately which is slightly less compared to the Random Forest Model.

Optimized Model (Part 3)

In the last part of our analysis, I would say after studying some classes in predictive as well as data mining, I feel the best solution is to keep on digging till we get some answers to our questions. There is never a definite path to finding out the best solution. In this case, we find out how the actual model plots a scatter plot versus what a prediction and training from the various models could look like after fitting Ridge, Lasso, Decision Tree, and Gradient Boosting Models. Figure 22 (Refer to Appendix) shows that the gradient search function helps to get a better model using acceleration, and horsepower primarily. We use the gradient boost method to check our variables since it seems to provide neither an underfit nor an overfit model of assessment.

Conclusion

1. The model is quite clean as such, without missing data. The '?' symbol has to be replaced with the object 'Horsepower'
2. From Correlation and Scatterplots, we notice that certain variables like weight, and displacement don't contribute much towards our analysis of finding the variables that can help to determine a good mileage. We drop those variables.
3. Feature Scaling is the last step of data pre-processing and is done so that all the variables that are small or large can be synthesized to one scale. We use Lasso, Ridge, Decision Tree, Gradient Boosting, and Random Forest Approach to get the most accurate variable that gives a good Miles per hour for the car.
4. Finally, the gradient boost approach gives a test and train accuracy of 93%and 84% approximately. It seems to be an apt model and neither under-fitted nor overfitted. The overall accuracy is 85%

References

Holtz, Y. (n.d.). *Control color in seaborn heatmaps*. The Python Graph Gallery. Retrieved October 5, 2022, from https://www.python-graph-gallery.com/92-control-color-in-seaborn-heatmaps/

Owusu, P. (2021, December 12). *Multiple Regression in python using Scikit-Learn: Predicting the Miles Per Gallon (mpg) of cars.* Medium. Retrieved October 5, 2022, from https://medium.com/@powusu381/multiple-regression-in-python-using-scikit-learn-predicting-the-miles-per-gallon-mpg-of-cars

Variance Inflation Factor (VIF). (2022, July 26). Investopedia. Retrieved October 6, 2022, from https://www.investopedia.com/terms/v/variance-inflation-factor.asp

Kunwar, A. (2021, December 26). *Feature Scaling and its importance in Data Preprocessing: Normalization vs Standardization*. Medium. Retrieved October 5, 2022, from https://ai.plainenglish.io/feature-scaling-and-its-importance-in-data-preprocessing-normalization-vs-standardization-750525682766

CarMpgEDA/CarMpgEDA.ipynb at master · ashishkssingh/CarMpgEDA. (n.d.). GitHub. Retrieved October 5, 2022, from https://github.com/ashishkssingh/CarMpgEDA/blob/master/CarMpgEDA.ipynb

A, P. (n.d.). *car_mpg_predict/Car MPG prediction.ipynb at master · prince381/car_mpg_predict*. GitHub. Retrieved October 5, 2022, from https://github.com/prince381/car_mpg_predict/blob/master/Car%20MPG%20prediction.ipynb

RaviTeja Mureboina. (2020, October 15). *Linear Regression on AUTO-MPG||Feature label selection||Generate graph||Printing Real and Predicted* [Video]. YouTube. Retrieved October 6, 2022, from https://www.youtube.com/watch?

Annexure

```
#Import the libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
sns.set()
import os
from sklearn import preprocessing
import statsmodels as sm
from statsmodels.stats.outliers_influence import variance_inflation_factor
from sklearn.model_selection import train_test_split,GridSearchCV,RandomizedSearchCV
from sklearn.linear_model import LinearRegression,Ridge,Lasso
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor,GradientBoostingRegressor
from sklearn.metrics import r2_score,mean_squared_error
from sklearn import preprocessing
#Read the file
path_file=os.getcwd()
car_mpg=pd.read_csv("car.csv")
#Printing the first ten columns of the dataset
car_mpg.head(n=10)
car_mpg.shape
#Checking the info
car_mpg.info ()
```

#Checking for any missing values

```
car_mpg.isna().sum()
```

Car Statistical Analysis

```
car_mpg.describe()
```

#Checking for unique values in the object data type

```
car_mpg.Horsepower.unique()
```

#Checking for duplicates

```
car_mpg.duplicated()
```

Creating Pair Plots to understand if there is any outlier

```
sns.pairplot(car_mpg, x_vars=car_mpg.drop(['MPG'],axis=1, inplace=False).columns,
        y_vars= ['MPG'])
```

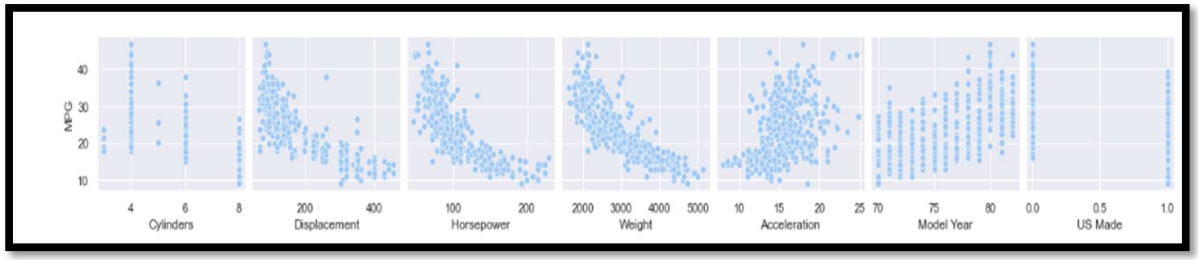

Figure 1: Pair Plot

#Create a scatter plot to check any relation between the Acceleration, Number of Cylinders and Miles Per Gallon

```
Pl. Figure(figsize=(10,8))
```

```
sns. scatterplot(x='MPG',y='Acceleration',hue='Cylinders',data=car_mpg)
```

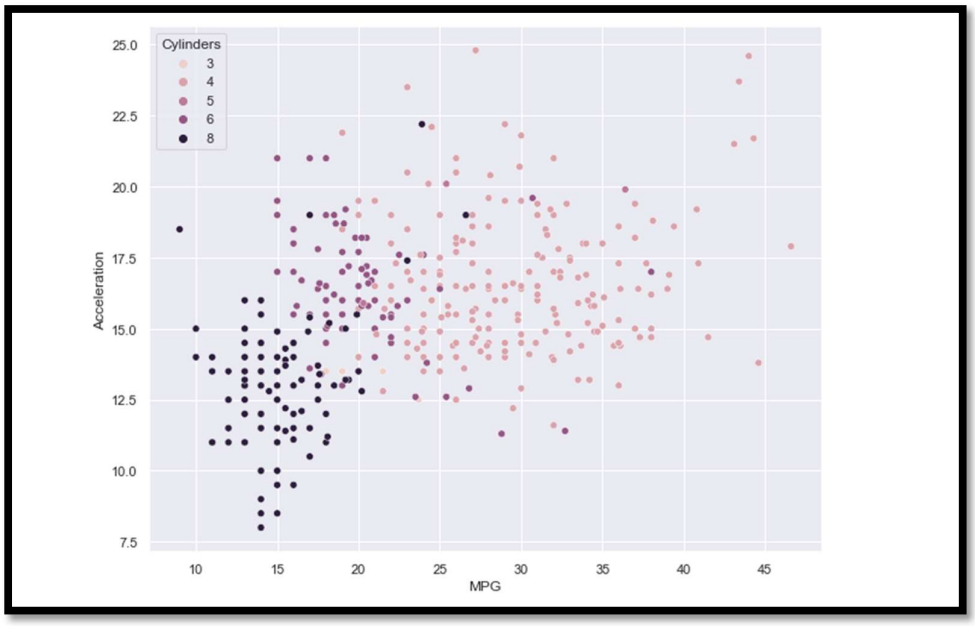

Figure 2: Scatter Plot

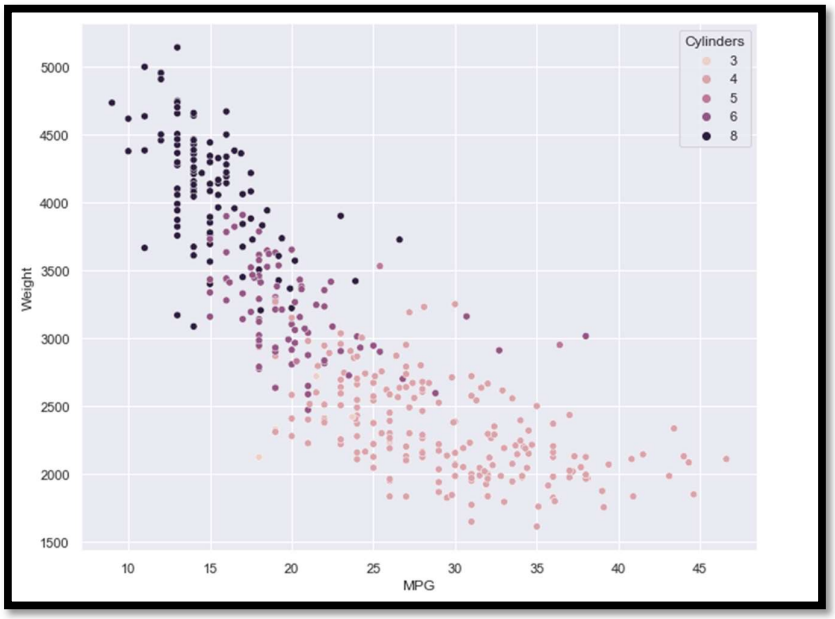

Figure 3 Scatterplot weight verus cylinder

#Violin Plot Cylinder versus Miles per Gallon

```
plt.figure(figsize=(10,8))

sns.violinplot(x='Cylinders',y='MPG',data=car_mpg)
```

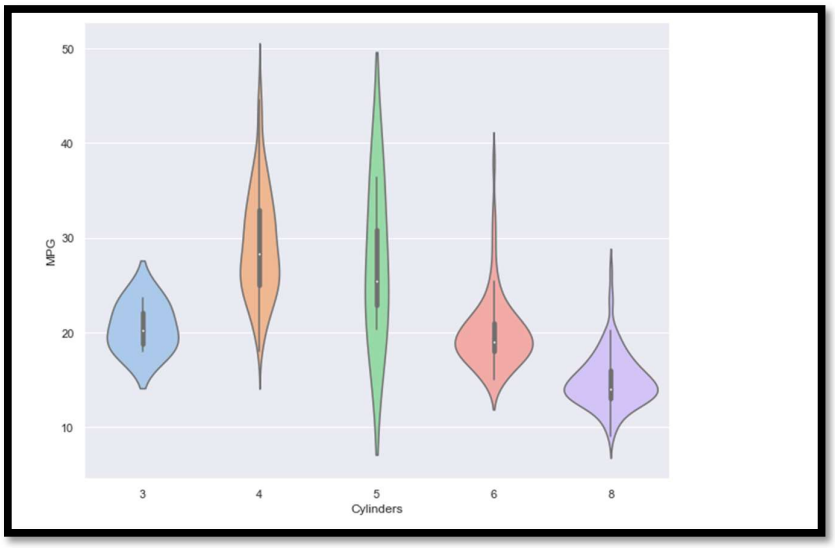

Figure 4: Violin Plots

#Removing the question Mark and filling the space with the mean horsepower

```
car_mpg.Horsepower = car_mpg.Horsepower.apply(str).str.replace('?','NaN').astype(float)

car_mpg. Horsepower.fillna(car_mpg.Horsepower.mean(),inplace=True)

car_mpg.Horsepower = car_mpg.Horsepower.astype(float)
```

Print the info of the data again to see the datatypes of columns

```
car_mpg.info ()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 8 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   MPG           398 non-null    float64
 1   Cylinders     398 non-null    int64
 2   Displacement  398 non-null    float64
 3   Horsepower    398 non-null    float64
 4   Weight        398 non-null    int64
 5   Acceleration  398 non-null    float64
 6   Model Year    398 non-null    int64
 7   US Made       398 non-null    int64
dtypes: float64(4), int64(4)
memory usage: 25.0 KB
```

Figure 5: Info

Visualizing the distribution of the features of the car

```
car_mpg.hist(figsize=(12,8),bins=20)

plt.show()
```

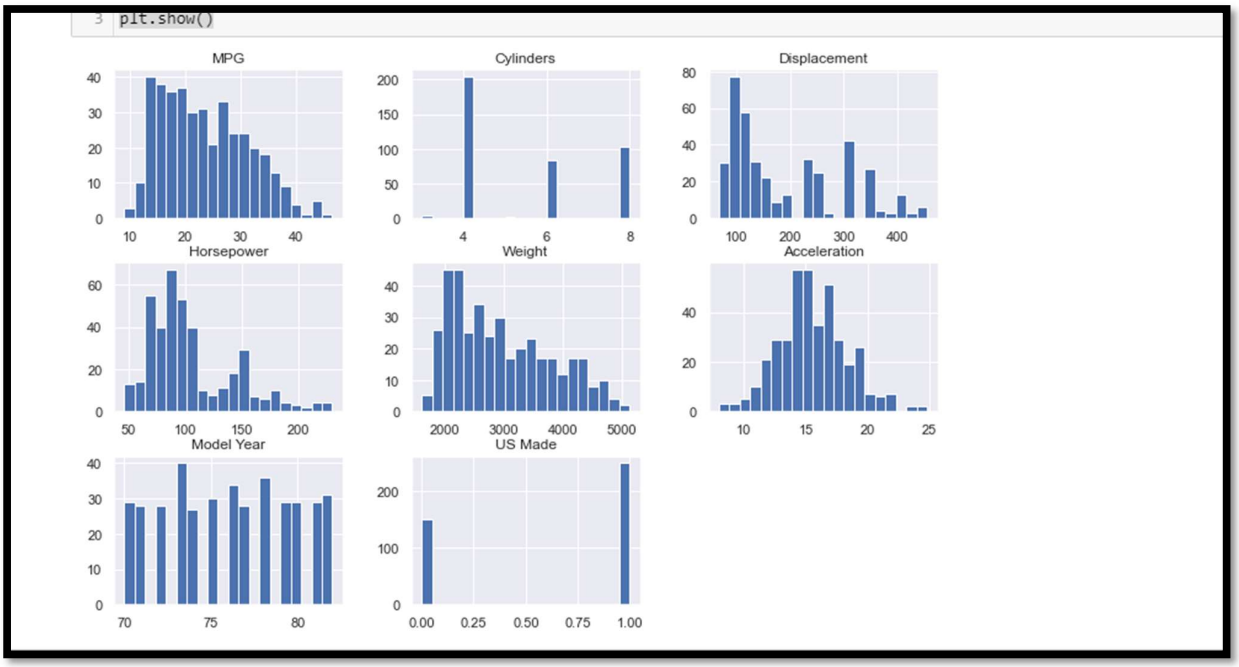

Figure 6: Distribution of features

#Correlation between miles per gallon and the other variables in the dataset

```
plt.figure(figsize=(10,6))
sns.heatmap(car_mpg.corr(),cmap='PiYG',annot=True)
plt.title('Heatmap displaying the relationship between\nthe features of the data',
fontsize=13)
plt.show()
```

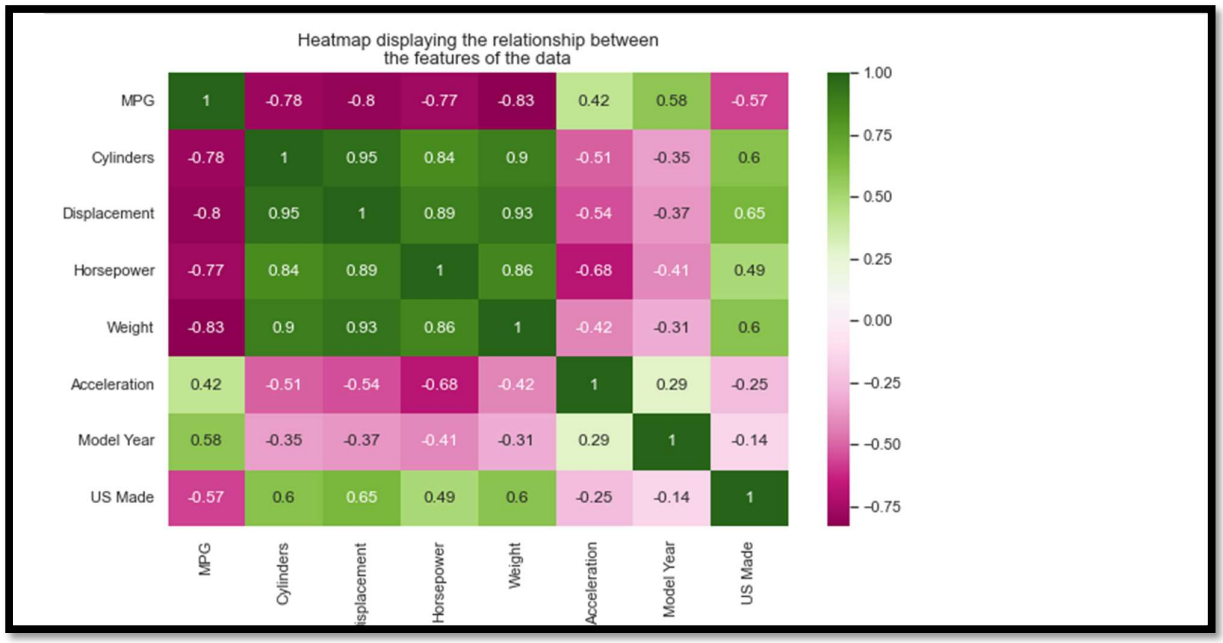

Figure 7: Correlation between the variables

#Checking the multi collinearity using VIF ()

X1 = sm.tools.add_constant(car_mpg)

X1

[4]:

	const	MPG	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model Year	US Made
0	1.0	18.0	8	307.0	130.0	3504	12.0	70	1
1	1.0	15.0	8	350.0	165.0	3693	11.5	70	1
2	1.0	18.0	8	318.0	150.0	3436	11.0	70	1
3	1.0	16.0	8	304.0	150.0	3433	12.0	70	1
4	1.0	17.0	8	302.0	140.0	3449	10.5	70	1
...	...	...	...	...	...	...	...	...	...
393	1.0	27.0	4	140.0	86.0	2790	15.6	82	1
394	1.0	44.0	4	97.0	52.0	2130	24.6	82	0
395	1.0	32.0	4	135.0	84.0	2295	11.6	82	1
396	1.0	28.0	4	120.0	79.0	2625	18.6	82	1
397	1.0	31.0	4	119.0	82.0	2720	19.4	82	1

398 rows × 9 columns

Figure 8: Checking multi-collinearity

```
series1 = pd.Series([variance_inflation_factor(X1.values,i) for i in
          range(X1.shape[1])],index=X1.columns)
```

Dropping the columns that highly correlate with each other

```
new_cars = car_mpg. drop(['Cylinders','Displacement','Weight'], axis=1)
# Using VIF method again after doing a feature selection

X2 = sm.tools.add_constant(new_cars)
series2 = pd.Series([variance_inflation_factor(X2.values,i) for i in range(X2.shape[1])],

                    index=X2.columns)
 print ('Series before feature selection: \n\n {} \n'.format(series1))

print ('Series after feature selection: \n\n {}'. format(series2))
```

```
Series before feature selection:

const           763.446231
MPG               5.696929
Cylinders        10.720660
Displacement     23.435172
Horsepower        9.043348
Weight           13.694543
Acceleration      2.512519
Model Year        2.035465
US Made           2.092441
dtype: float64

Series after feature selection:

const           691.164414
MPG               4.020904
Horsepower        4.136177
Acceleration      2.022900
Model Year        1.663170
US Made           1.614071
dtype: float64
```

Figure 9: Series before and after feature selection

Correlation b/w the mpg and the remaining variables

```
print ('Correlation between mpg and the remaining variables:\n\n {}'.
        format(new_cars.corr().MPG))
```

```
Correlation between mpg and the remaining variables:

MPG             1.000000
Horsepower     -0.771437
Acceleration    0.420289
Model Year      0.579267
US Made        -0.568192
Name: MPG, dtype: float64
```

Figure 10: Correlation between mpg and other variables

```
X = new_cars.drop('MPG',axis=1)  # create a DataFrame of feature/independent variables
```

```
y = new_cars.MPG

# Create a series of the dependent variable

# The feature/independent variables are not of the same scale, so we scale them

# Feature scaling gives faster computing.

X_scaled = preprocessing.scale(X)

X_scaled = pd.DataFrame(X_scaled,columns=X.columns)

# Split data into training and testing data

X_train,X_test,y_train,y_test = train_test_split(X_scaled,y,test_size=.3,random_state=0)

model = LinearRegression()

# Initialize the LinearRegression model

model.fit(X_train,y_train)  # we fit the model with the training data


linear_pred = model.predict(X_test)  # make prediction with the fitted model

# Score the model on the train set

print ('Train score: {} \n'. format(model.score(X_train,y_train)))

# Score the model on the test set

print ('Test score: {} \n'. format(model.score(X_test,y_test)))

# Calculate the overall accuracy of the model

print ('Overall model accuracy: {} \n'. format(r2_score(y_test,linear_pred)))

# Compute the mean squared error of the model

print ('Mean Squared Error: {}'. format(mean_squared_error(y_test,linear_pred)))
```

```
Train score: 0.7544298891613411

Test score: 0.7362936246735721

Overall model accuracy: 0.7362936246735721

Mean Squared Error: 16.851333847236184
```

Figure 11: Linear Regression Score

```
# let's fit a ridge regression model and see if we can get a higher accuracy
ridge = Ridge(alpha=.01)
ridge.fit(X_train,y_train)  # fit the model with the training data
ridge_pred = ridge.predict(X_test)  # make predictions
# Score the model to check the accuracy
print ('Train score: {} \n'. format(ridge.score(X_train,y_train)))
print ('Test score: {} \n'. format(ridge.score(X_test,y_test)))
print ('Overall model accuracy: {} \n'. format(r2_score(y_test,ridge_pred)))
print ('Mean Squared Error: {}'. format(mean_squared_error(y_test,ridge_pred)))
```

```
Train score: 0.7544298877689312

Test score: 0.7362921449368379

Overall model accuracy: 0.7362921449368379

Mean Squared Error: 16.851428405199304
```

Figure 12: Better and Optimised results after Ridge

```
#Explore Ridge Model
ridge_model = Ridge ()
param = {'alpha': [0,0.1,0.01,0.001,1]}
# Initialize the grid search
ridge_search = GridSearchCV(ridge_model,param,cv=5,n_jobs=-1)
ridge_search.fit(X_train,y_train)
# Fit the model
```

```
GridSearchCV(cv=5, estimator=Ridge(), n_jobs=-1,
             param_grid={'alpha': [0, 0.1, 0.01, 0.001, 1]})
```

Figure13: Grid Search

```
print ('Best parameter found:\n {}'. format(ridge_search.best_params_))
print ('Train score: {} \n'. format(ridge_search.score(X_train,y_train)))
print ('Test score: {}'. format(ridge_search.score(X_test,y_test)))
```

```
Best parameter found:
{'alpha': 1}
Train score: 0.7544162904974933

Test score: 0.736133629303902
```

Figure 14: Parameters Explored after Ridge

```
# Lasso Model
lasso = Lasso ()
param['max_iter'] = [1000,10000,100000,1000000]
lasso_search = GridSearchCV(lasso,param,cv=5,n_jobs=-1)
 # Initialize the grid search
lasso_search.fit(X_train,y_train)  # fit the model
# Print out the best parameters and score it
print ('Best parameter found:\n {} \n'. format(lasso_search.best_params_))
print ('Train score: {} \n'. format(lasso_search.score(X_train,y_train)))
print ('Test score: {}'. format(lasso_search.score(X_test,y_test)))
```

```
Best parameter found:
{'alpha': 0, 'max_iter': 1000}

Train score: 0.7544298891613411

Test score: 0.7362936246735721
```

Figure 15: Parameter explored after Lasso

```
# Implementing Regression
# Splitting X and y data into training and testing data
xtrain,xtest,ytrain,ytest = train_test_split(X,y,test_size=.2)
dtree = DecisionTreeRegressor() # initialize a DecisionTreeRegressor model

params = {'max_features': ['auto','sqrt','log2'],
        'min_samples_split': [2,3,4,5,6,7,8,9],
```

```
'min_samples_leaf': [1,2,3,4,5,6,7,8,9],
'max_depth': [2,3,4,5,6,7]}              # define the hyperparameters

tree_search = GridSearchCV(dtree, params,cv=5,n_jobs=-1)  # initialize the grid search

tree_search.fit(xtrain,ytrain)   # fit the model
```

```
: GridSearchCV(cv=5, estimator=DecisionTreeRegressor(), n_jobs=-1,
               param_grid={'max_depth': [2, 3, 4, 5, 6, 7],
                           'max_features': ['auto', 'sqrt', 'log2'],
                           'min_samples_leaf': [1, 2, 3, 4, 5, 6, 7, 8, 9],
                           'min_samples_split': [2, 3, 4, 5, 6, 7, 8, 9]})
```

Figure 16: Decision Tree Model

```
tree_pred = tree_search.predict(xtest)  # make predictions with the model

# Print out the best parameters found and score the model
print ('Best parameter found:\n {} \n'. format(tree_search.best_params_))
print ('Train score: {} \n'. format(tree_search.score(xtrain,ytrain)))
print ('Test score: {} \n'. format(tree_search.score(xtest,ytest)))
print ('Overall model accuracy: {} \n'. format(r2_score(ytest,tree_pred)))
print ('Mean Squared Error: {}'. format(mean_squared_error(ytest,tree_pred)))
```

```
Best parameter found:
{'max_depth': 5, 'max_features': 'log2', 'min_samples_leaf': 1, 'min_samples_split': 2}

Train score: 0.8764643206017688

Test score: 0.7611275863316822

Overall model accuracy: 0.7611275863316822

Mean Squared Error: 16.839012211030983
```

Figure 17: Decision Tree Optimization

```
# Using RandomForestRegressor modelto find the best Parameters

forest = RandomForestRegressor()
# We add the n_estimators parameter in our previous parameter dictionary
params['n_estimators'] = [100,200,300,400,500]
forest_search = RandomizedSearchCV(forest,params,cv=5,n_jobs=-1, n_iter=50)
# initialize the search
forest_search.fit(xtrain,ytrain)  # fit the model
```

```
RandomizedSearchCV(cv=5, estimator=RandomForestRegressor(), n_iter=50,
                   n_jobs=-1,
                   param_distributions={'max_depth': [2, 3, 4, 5, 6, 7],
                                        'max_features': ['auto', 'sqrt',
                                                         'log2'],
                                        'min_samples_leaf': [1, 2, 3, 4, 5, 6,
                                                             7, 8, 9],
                                        'min_samples_split': [2, 3, 4, 5, 6, 7,
                                                              8, 9],
                                        'n_estimators': [100, 200, 300, 400,
                                                         500]})
```

Figure 18: Random Forest

```
forest_pred = forest_search.predict(xtest)  # make prediction with the model
# Print out the best parameters and score the model
print ('Best parameter found:\n {} \n'. format (forest_search. best_params_))
print ('Train score: {} \n'. format (forest_search. score(xtrain,ytrain)))
print ('Test score: {} \n'. format (forest_search. score(xtest,ytest)))
print ('Overall model accuracy: {} \n'. format(r2_score(ytest,forest_pred)))
print ('Mean Squared Error: {}'. format(mean_squared_error(ytest,forest_pred)))
```

```
Best parameter found:
{'n_estimators': 300, 'min_samples_split': 3, 'min_samples_leaf': 1, 'max_features': 'log2', 'max_depth': 7}

Train score: 0.9449110285946899

Test score: 0.8533205988782191

Overall model accuracy: 0.8533205988782191

Mean Squared Error: 10.339981032828542
```

Figure 19: Model Optimisation with Random Forest Regressor

```
# Train a GradientBoostingRegressor model
gradient_model = GradientBoostingRegressor () # instantiate the model
# Append a learning_rate parameter to the parameter dictionary
params['learning_rate'] = [0.05,0.1,0.2,0.3,0.4,0.5]
gradient_search = RandomizedSearchCV(gradient_model,params,cv=5,n_jobs=-1,
                     n_iter=50)   # initialize the search
gradient_search.fit(xtrain,ytrain)   # fit the model
```

```
RandomizedSearchCV(cv=5, estimator=GradientBoostingRegressor(), n_iter=50,
                   n_jobs=-1,
                   param_distributions={'learning_rate': [0.05, 0.1, 0.2, 0.3,
                                                          0.4, 0.5],
                                        'max_depth': [2, 3, 4, 5, 6, 7],
                                        'max_features': ['auto', 'sqrt',
                                                         'log2'],
                                        'min_samples_leaf': [1, 2, 3, 4, 5, 6,
                                                             7, 8, 9],
                                        'min_samples_split': [2, 3, 4, 5, 6, 7,
                                                              8, 9],
                                        'n_estimators': [100, 200, 300, 400,
                                                         500]})
```

Figure 20: Gradient Boosting Regressor Model

```
gradient_pred = gradient_search.predict(xtest)  # make predictions with the model
# Print out the best parameters and score the model
print ('Best parameter found:\n {} \n'. format(gradient_search.best_params_))
print ('Train score: {} \n'. format(gradient_search.score(xtrain,ytrain)))
print ('Test score: {} \n'. format(gradient_search.score(xtest,ytest)))
print ('Overall model accuracy: {} \n'. format(r2_score(ytest,gradient_pred)))
print ('Mean Squared Error: {} \n'. format(mean_squared_error(ytest,gradient_pred)))
print ('This model is not too overfitted and it has low mean squared error \
so, we use this one.....')
```

```
Best parameter found:
{'n_estimators': 200, 'min_samples_split': 3, 'min_samples_leaf': 2, 'max_features': 'log2', 'max_depth': 3, 'learning_rate':
0.05}

Train score: 0.9381725797665585

Test score: 0.8447794191936404

Overall model accuracy: 0.8447794191936404

Mean Squared Error: 10.942080818218312

This model is not too overfitted and it has low mean squared error so we use this one.....
```

Figure 21: Best Parameter

```
# Model to see how close our predictions are to the actual values.
new_cars
  # Create a new DataFrame of the feature variables
newcars_new = new_cars.drop('MPG',axis=1)
# Make a DataFrame of the actual mpg and the predicted mpg
data = pd.DataFrame({'Actual mpg':new_cars.MPG.values,'Predicted
mpg':gradient_search.predict(newcars_new.values)})
#  Make a scatter plot of the actual and the predicted mpg of a car
plt.figure(figsize=(12,8))
plt.scatter(data.index,data['Actual mpg'].values,label='Actual mpg')
plt.scatter(data.index,data['Predicted mpg'].values,label='Predicted mpg')
plt.title('Comparing the Actual mpg values to the Predicted mpg values\nModel accuracy =
85%',fontsize=16)
plt.xlabel('Car index')
plt.ylabel('Mile Per Gallon (mpg)')
plt.legend(loc='upper left')
plt.show()
```

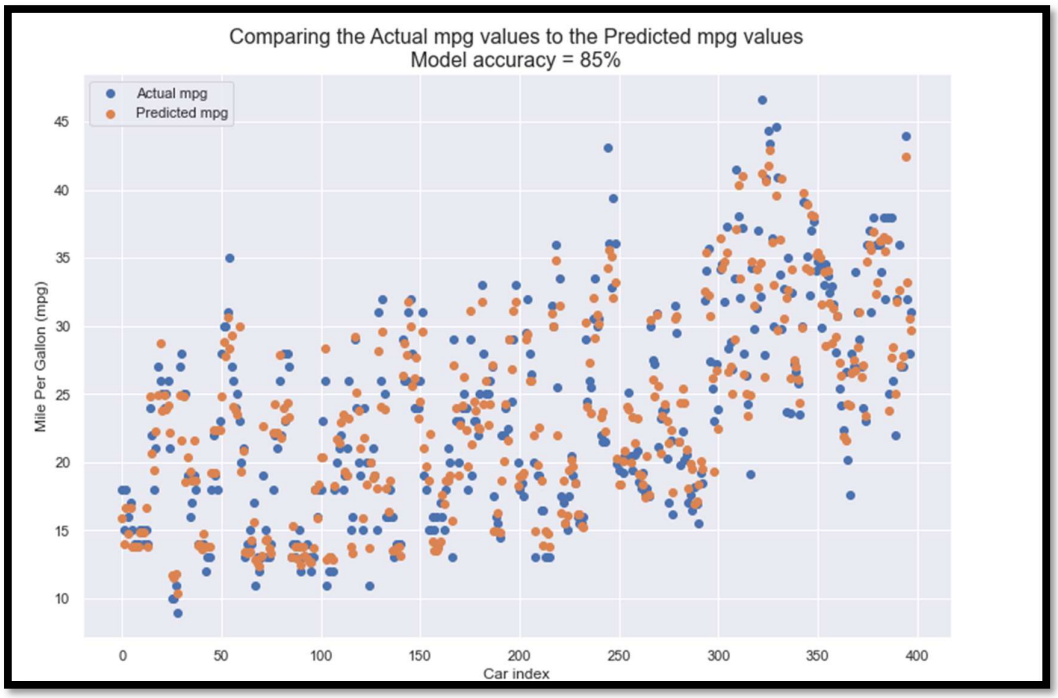

Figure 22: Actual MPG vs Predicted MPG