Decision Tree Model

After developing the two iterations of the linear regression model I noticed that while the MSE decreased the MAE had a significant specifically increase tripling in value. Which leads to me thinking that the outliers may have had a greater effect on the dataset than originally predicted. For the decision tree model i'll follow a similar pattern as to the linear regression model but this model ill focus on feature importance and hyperparameter tuning.

In [ ]:
```
import pandas as pd #pandas and numpy for data manipulation
import numpy as np
from sklearn.tree import DecisionTreeRegressor # Decision Tree the algorithm i'll be using
from sklearn.feature_selection import RFE
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score
from sklearn.metrics import r2_score, mean_absolute_error, mean_squared_error #each helps us gauge the effectiv
import seaborn as sns #seaborn and matplotlib used to visualize data
import matplotlib.pyplot as plt
```

In [ ]:
```
File_Path = pd.read_excel("C:\\Users\\vturn\\OneDrive\\Cars_MPG.xlsx")
car_data = File_Path
```

Creating a dictionary to host the models performance indicators

In the first model I didn't use this and it was a bit harder to keep track of the performance of the different models towards the end this should help.

In [ ]:
```
model_performance = {
  'Model 1': {'MAE': None, 'MSE': None, 'R2': None},
  'Model 2' : {'MAE': None, 'MSE': None, 'R2': None},
  'Model 3' : {'MAE': None, 'MSE': None, 'R2': None},
  'Model 4' : {'MAE': None, 'MSE': None, 'R2': None},
  'Total Averages' : {'MAE': None, 'MSE': None, 'R2': None},
}

model_performance = pd.DataFrame(model_performance)
```

EDA(exploratory data analysis/Cleaning)

a bit redundant since we've already done this process for the linear regression model but good opportunity to be innovative and try to improve.

In [ ]:
```
def perform_eda(df):
  return (df.dtypes)

def dataframe_size(df):
  return (df.shape)
car_data_shape = dataframe_size(car_data)
print('the size of the data set is {}'.format(car_data_shape))

car_data_column_types = perform_eda(car_data)
print(car_data_column_types)
```

```
the size of the data set is (398, 9)
mpg              float64
cylinders          int64
displacement     float64
horsepower        object
weight             int64
acceleration     float64
model year         int64
origin             int64
car name          object
dtype: object
```

In [ ]:
```
def find_replace_drop_missing_values(dataframe, column):
  for value in dataframe[column]:
    if type(value) != int and type(value) != float:
      dataframe[column].replace(value, np.nan, inplace=True)
      dataframe.dropna(subset = column, inplace= True)
      print(dataframe.isna().sum())
    else:
      continue
find_replace_drop_missing_values(car_data, 'horsepower')
```

```
mpg             0
cylinders       0
displacement    0
horsepower      0
weight          0
acceleration    0
model year      0
origin          0
car name        0
dtype: int64
```

Splitting data into training and testing sets, and fitting the model.

previously I had normalized the data but for decision trees normalization isn't weighed as heavily.

In [ ]:
```
x1 = car_data[['cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'model year', 'origin']]
y1 = car_data['mpg']
x1_train, x1_test, y1_train, y1_test = train_test_split(x1, y1, test_size= 0.2, random_state= 42)

model = DecisionTreeRegressor(random_state = 42)
model.fit(x1_train, y1_train)
y1_prediction = model.predict(x1_test)
```

Feeding the model new data to get an idea of the models performance.

In [ ]:
```
pickup = {
  'cylinders': [8],
  'displacement': [307],
  'horsepower': [130],
  'weight': [3504],
  'acceleration': [12],
  'model year': [70],
  'origin': [1]
}
pickup = pd.DataFrame(pickup)
pickup_prediction = model.predict(pickup)
print(pickup_prediction)
```

```
[17.]
```

Using the MAE, MSE and r2 for model performance indicators.

i'll create functions to make the process easier for future models and eliminate the redundancy.

In [ ]:
```
def get_r2 (y0_test, y0_predictions):
    r2 = r2_score(y0_test, y0_predictions)
    r2_in_percentage = (r2 * 100) #i mentioned i view it as a percentage converting it to have the appearance o
    return('%{:.2f}'.format(r2_in_percentage))
model1_r2 = get_r2(y1_test, y1_prediction)

def get_mae(y0_test, y0_prediction):
  return(mean_absolute_error(y0_test, y0_prediction))
model1_mae = get_mae(y1_test, y1_prediction)

def get_mse(y0_test, y0_prediction):
  return(mean_squared_error(y0_test, y0_prediction))
model1_mse = get_mse(y1_test, y1_prediction)

#adding the new values to the data frame we created earlier
model_performance['Model 1']['MAE'] = model1_mae
model_performance['Model 1']['MSE'] = model1_mse
model_performance['Model 1']['R2'] = model1_r2
print(model_performance)
```

```
       Model 1 Model 2 Model 3 Model 4 Total Averages
MAE   2.259494    None    None    None           None
MSE  11.428481    None    None    None           None
R2      %77.61    None    None    None           None
```

Model performed about as expected, first step should be get the feature importance. Unlike in Linear regression we don't have coefficients in DecisionTrees so our features wont have positive and negative probabilities. Instead we can think of the features as pieces of the pie that ultimately makeup the total target in our case mpg. The larger the piece the more impact it has and vice versa.

Because coefficients aren't a valid indicator in decision tree models the process to get feature importance is slightly different.

```
In [ ]: feature_importance = model.feature_importances_  #the decision tree equivalent of (coefficient = model.coef_)
        features = x1_train.columns #this remains the same as it would in linear regression
```

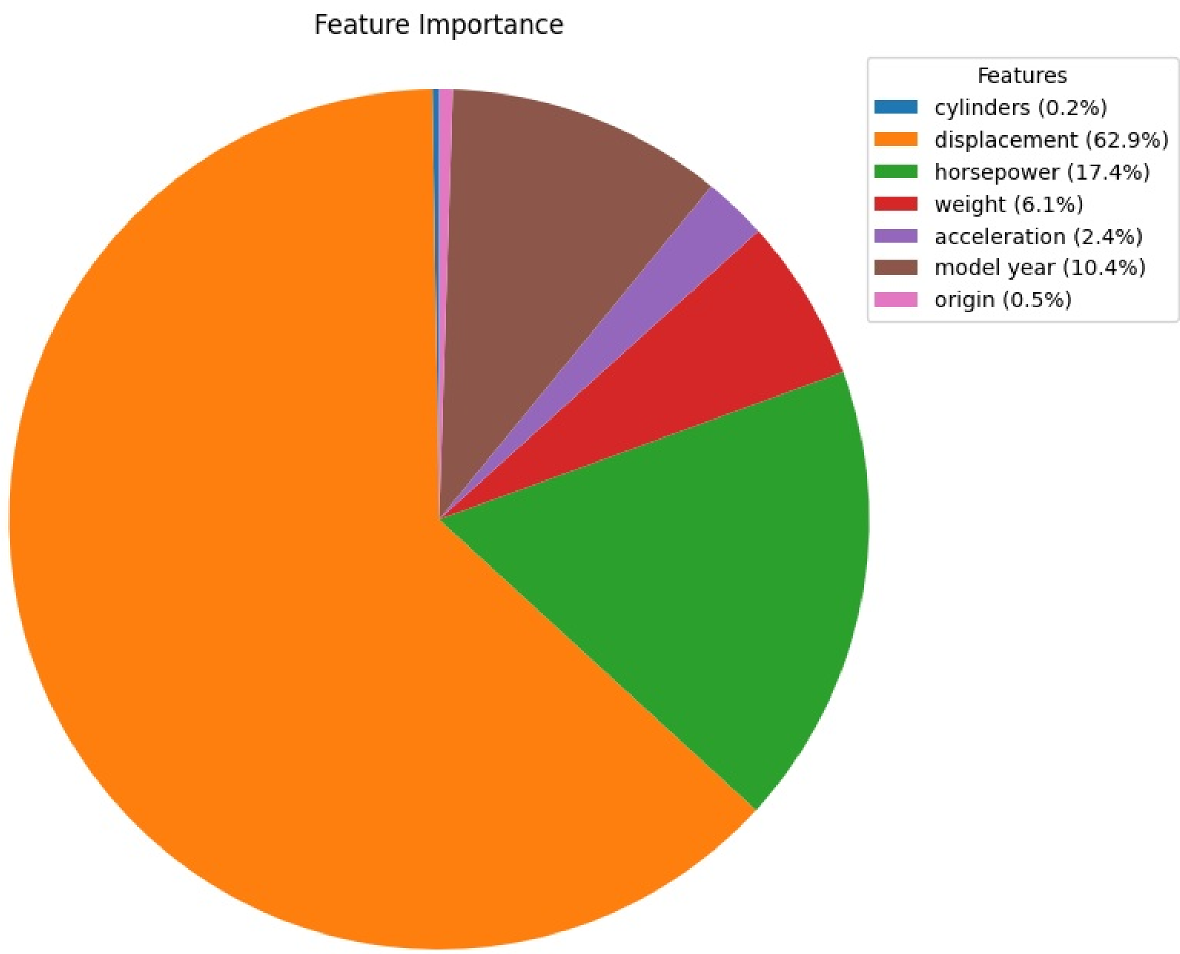
```
In [ ]: magnitude = model.feature_importances_
        magnitude_of_features = dict(zip(features,magnitude))
        magnitude_of_features = pd.DataFrame({'features':features, 'magnitude':magnitude})
        print(magnitude_of_features)
```

```
       features  magnitude
0     cylinders   0.002426
1  displacement   0.629492
2    horsepower   0.173754
3        weight   0.061156
4  acceleration   0.024077
5    model year   0.103953
6        origin   0.005143
```

Off first glance displacement plays the most significant role in determining mpg for this model followed by horsepower, thats interesting because in our linear regression model weight was the most important. This suggests that weight had a stronger linear relationship but displacement has a more significant relationship.

Visual aid for the feature importance

```
In [ ]: plt.figure(figsize=(10, 8))
        plt.pie(magnitude_of_features['magnitude'], startangle=90)
        plt.axis('equal')
        labels_with_percentages = [f"{feature} ({magnitude:.1f}%)" for feature, magnitude in zip(magnitude_of_features[
        plt.legend(title='Features', labels=labels_with_percentages, bbox_to_anchor=(0.85, 1), loc='upper left')
        plt.title('Feature Importance')
        plt.show()
```



Implementing hyper parameters for model version 2

Splitting the data per usual

```
In [ ]: x2 = car_data[['cylinders', 'displacement', 'horsepower', 'weight', 'acceleration', 'model year', 'origin']]
        y2 = car_data['mpg']
        x2_train, x2_test, y2_train, y2_test = train_test_split(x2, y2, test_size= 0.2, random_state= 42)
```

Im a terrible teacher partially because I understand how to code it and not the ins and outs of hyper parameters exactly but we can start from step 1. Creating a dictionary that contains the various parameters we'd like to implement the model will run through the dictionary trying various combinations of the parameters listed until it lands on the best results similar to a for loop. for value in grid: etc etc

In [ ]:
```
parameter_grid = {
    'criterion': ['squared_error', 'absolute_error'],
    'splitter': ['best', 'random'],
    'max_depth': [None, 10, 20, 30, 40],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
```

It's a two step process, we need to set up the environment for the hyper parameters to run through the initial combination configuration which we're doing below.

In [ ]:
```
base_model = DecisionTreeRegressor(random_state=42)
grid_search = GridSearchCV(estimator=base_model, param_grid=parameter_grid, cv=5)
grid_search.fit(x2_train, y2_train)
best_parameters = grid_search.best_params_
print(best_parameters) #returns a list of the best combination of parameters given the options provided in dict.
```

```
{'criterion': 'absolute_error', 'max_depth': 10, 'min_samples_leaf': 2, 'min_samples_split': 5, 'splitter': 'ran
dom'}
```

This returned a list of the best combination of parameters to use which we stored in the best_parameters variable in the following block we'll be using the best_parameters in model 2 and gauging its performance.

In [ ]:
```
best_model = DecisionTreeRegressor(**best_parameters)
best_model.fit(x2_train, y2_train)
y2_prediction_tuned = best_model.predict(x2_test)
```

Now that the models been successfully fitted and trained we need to get an understanding of its performance using the functions earlier and store those results in our data frame.

In [ ]:
```
model2_mae = get_mae(y1_test, y2_prediction_tuned)
model2_mse = get_mse(y1_test, y2_prediction_tuned)
model2_r2 = get_r2(y1_test, y2_prediction_tuned)


model_performance['Model 2']['MAE'] = model2_mae
model_performance['Model 2']['MSE'] = model2_mse
model_performance['Model 2']['R2'] = model2_r2

print(model_performance)
```

```
       Model 1   Model 2 Model 3 Model 4 Total Averages
MAE   2.259494  2.094937    None    None           None
MSE  11.428481   8.76019    None    None           None
R2      %77.61    %82.84    None    None           None
```

Understanding Model 2 results

The model improved all across the board improving the MSE, MAE, and R2 scores.

Going back to the emphasis placed on feature importance i'd like to remove weight, acceleration and origin because they all finished closer to last place in regards to feature importance. I'd like to observe any changes made to the model with them no longer being apart of the training and testing data samples. I'll also be changing the test size up to this point it's been 0.2 leaving .8 for training but at risk of over fitting ill be changing the test size to 0.1

Fitting and training model 3 without weight, acceleration and origin in the features.

In [ ]:
```
x3 = car_data[['cylinders', 'displacement', 'horsepower', 'model year']]
y3 = car_data['mpg']
x3_train, x3_test, y3_train, y3_test = train_test_split(x3, y3, test_size=0.1, random_state=42)

model3 = DecisionTreeRegressor(random_state=42)
model3.fit(x3_train, y3_train)

y3_prediction = model3.predict(x3_test)

model3_mae = get_mae(y3_test, y3_prediction)
model3_mse = get_mse(y3_test, y3_prediction)
model3_r2 = get_r2(y3_test, y3_prediction)
```

```
model_performance['Model 3']['MAE'] = model3_mae
model_performance['Model 3']['MSE'] = model3_mse
model_performance['Model 3']['R2'] = model3_r2

print(model_performance)
```

```
       Model 1   Model 2   Model 3 Model 4 Total Averages
MAE   2.259494  2.094937   2.39875    None           None
MSE  11.428481   8.76019  8.117062    None           None
R2      %77.61    %82.84    %86.36    None           None
```

The r2 score rose dramatically but this may be a surface level improvement because we changed the test size. This may have caused over fitting(when the model learns too much including learning the mistakes and assuming their standard) we can reduce this by pruning.

Model 4, model 3 may have suffered from over fitting re-tuning and running model again.

prior to pruning we need to split the data into training and testing sets again.

In [ ]:
```
x4 = car_data[['cylinders', 'displacement', 'horsepower', 'model year']]
y4 = car_data['mpg']
x4_train, x4_test, y4_train, y4_test = train_test_split(x4, y4, test_size=0.1, random_state=42)
```

select the model we'll be using(decision tree) and the depth. The easiest way i've found to think of depth is like traveling to and from a location. each depth represents a split and each split represents a method. In our analogy it would be different methods of transformation we decide based on a plethora of conditions for example Car, Boat, or Plane. If were going to work would we take a plane probably not we'd use a car. If we were going to an island would we take a car, no we'd use a boat similar with decision trees theres no one size fits all so by increasing the depth we allow for more options to reach our destination(target/mpg) however, since we were worried about over fitting originally i'll limit the maximum depth of this model to 4.

In [ ]:
```
model_pruned = DecisionTreeRegressor(max_depth=4, random_state=42)
```

I'll be leaving the test size to 0.1 as to keep similar conditions from model 3.

In [ ]:
```
model_pruned.fit(x4_train, y4_train)
y4_prediction = model_pruned.predict(x4_test)

model4_mae = get_mae(y4_test, y4_prediction)
model4_mse = get_mse(y4_test, y4_prediction)
model4_r2 = get_r2(y4_test, y4_prediction)

model_performance['Model 4']['MAE'] = model4_mae
model_performance['Model 4']['MSE'] = model4_mse
model_performance['Model 4']['R2'] = model4_r2

print(model_performance)
```

```
       Model 1   Model 2   Model 3   Model 4 Total Averages
MAE   2.259494  2.094937   2.39875  2.344922           None
MSE  11.428481   8.76019  8.117062  9.285618           None
R2      %77.61    %82.84    %86.36    %84.39           None
```

Now that the model has somewhat evened out i'll pull the different averages per column to compare.

In [ ]:
```
mae_list = [2.259494, 2.215823,2.39875,2.344922]
mae_average = (np.mean(mae_list))

mse_list = [11.428481,9.705222, 8.117062,9.285618 ]
mse_average = (np.mean(mse_list))

r2_score_list = [77.61,80.99,86.36,84.39]
r2_average = (np.mean(r2_score_list))
r2_average = ('%{:.2f}'.format(r2_average))

model_performance['Total Averages']['MAE'] = mae_average
model_performance['Total Averages']['MSE'] = mse_average
model_performance['Total Averages']['R2'] = r2_average
print(model_performance)
```

```
       Model 1   Model 2   Model 3   Model 4 Total Averages
MAE   2.259494  2.094937   2.39875  2.344922       2.304747
MSE  11.428481   8.76019  8.117062  9.285618       9.634096
R2      %77.61    %82.84    %86.36    %84.39         %82.34
```

In [ ]:
```
labels = ['MAE', 'MSE', 'R2']
model_1 = [2.259494, 11.428481, 77.61]
model_2 = [2.176582, 10.972373, 78.50]
```

```
model_3 = [2.398750, 8.117062, 86.36]
model_4 = [2.629617, 10.803358, 81.84]
total_averages = [2.3661, 10.0813, 81.07]
x = np.arange(len(labels))

width = 0.1
fig, ax = plt.subplots()
ax.bar(x - 3*width/2, model_1, width, label='Model 1', color='red')
ax.bar(x - width/2, model_2, width, label='Model 2', color='blue')
ax.bar(x + width/2, model_3, width, label='Model 3', color='green')
ax.bar(x + 3*width/2, model_4, width, label='Model 4', color='purple')
ax.bar(x + 5*width/2, total_averages, width, label='Total Avg', color='orange')
ax.set_ylabel('Scores')
ax.set_title('Model performance comparison')
ax.set_xticks(x)
ax.set_xticklabels(labels)
ax.legend()
plt.show()
```