Faculty of Engineering and Technology

Electrical and Computer Engineering Department

MACHINE LEARNING AND DATA SCIENCE

ENCS5341

Assignment No.1

Student's Name: Nour Rabee'

ID Number: 1191035

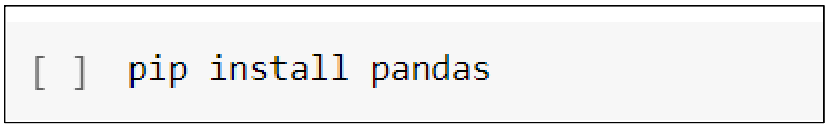
Instructor's Name: Dr. Yazan Abu Farha

Section: 2

November 30, 2023

1. Read the dataset and examine how many features and examples does it have? (Hint: you can use Pandas to load the dataset into a dataframe).

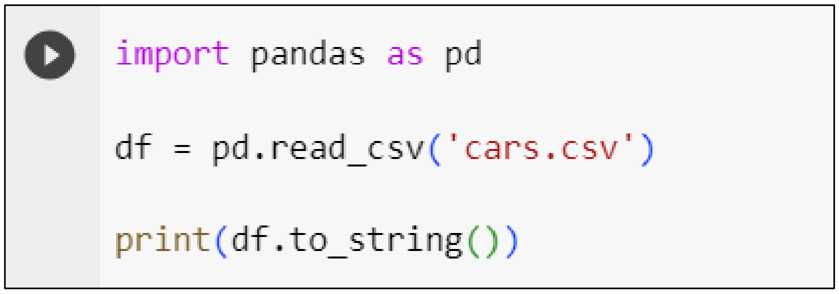
Firstly, the Pandas Library has been installed, enabling the invocation of any method within it, as illustrated in Figure 1.



```
[ ]  pip install pandas
```

Figure 1(Pandas Library Installation)

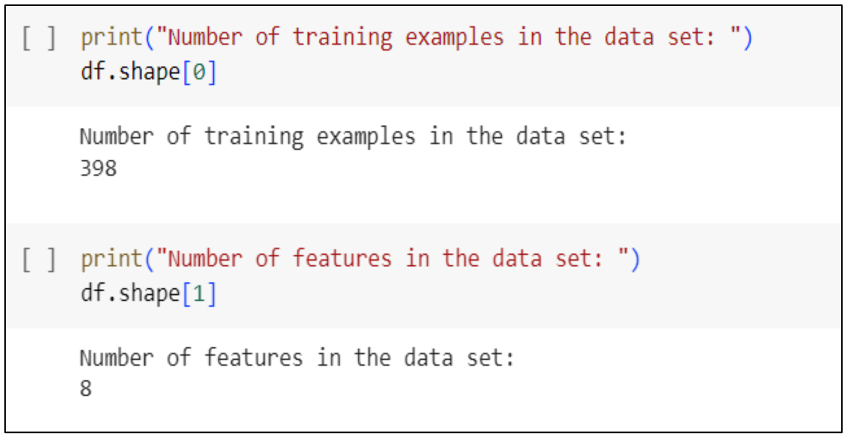
As shown in figure 2, to begin using pandas, it is imported at the beginning and given the alias 'pd' for convenience. Subsequently, the dataset is read using a method called 'read_csv,' which is designed to read CSV (comma-separated values) files.



```
import pandas as pd

df = pd.read_csv('cars.csv')

print(df.to_string())
```

Figure 2(importing Pandas Library and reading the csv file)

Using the `shape` function allows me to determine the number of features and training examples in the dataset, as demonstrated below. `shape[0]` computes the number of training examples, while `shape[1]` computes the number of features.

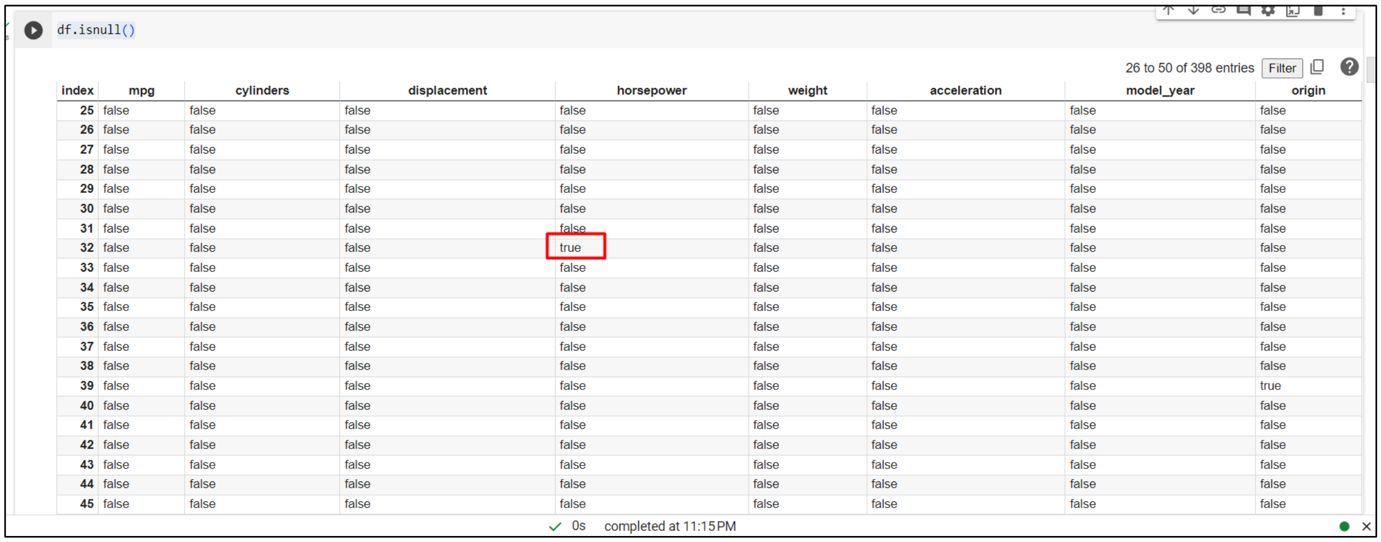


```
[ ]  print("Number of training examples in the data set: ")
     df.shape[0]

     Number of training examples in the data set:
     398

[ ]  print("Number of features in the data set: ")
     df.shape[1]

     Number of features in the data set:
     8
```

Figure 3(Number of features and training examples in the data set)

2. Are there features with missing values? How many missing values are there in each one? *(Hint: you can use* `isnull()` *from* `Pandas`*)*

The 'isnull()' function is employed to determine the presence of missing values by generating a DataFrame with boolean values of the same shape as the original 'df' (True for missing values and False for non-missing values), as depicted in Figure 4 below. In Example 32, it is observed that the 'horsepower' feature does not have a value.



```
df.isnull()
```

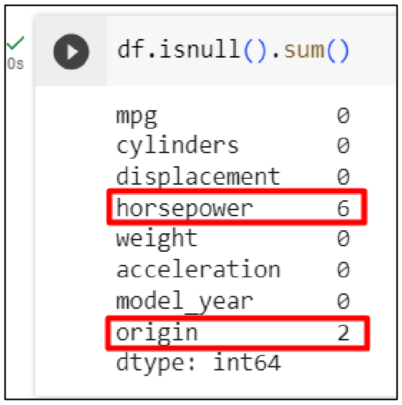
26 to 50 of 398 entries Filter

index	mpg	cylinders	displacement	horsepower	weight	acceleration	model_year	origin
25	false	false	false	false	false	false	false	false
26	false	false	false	false	false	false	false	false
27	false	false	false	false	false	false	false	false
28	false	false	false	false	false	false	false	false
29	false	false	false	false	false	false	false	false
30	false	false	false	false	false	false	false	false
31	false	false	false	false	false	false	false	false
32	false	false	false	true	false	false	false	false
33	false	false	false	false	false	false	false	false
34	false	false	false	false	false	false	false	false
35	false	false	false	false	false	false	false	false
36	false	false	false	false	false	false	false	false
37	false	false	false	false	false	false	false	false
38	false	false	false	false	false	false	false	false
39	false	false	false	false	false	false	false	true
40	false	false	false	false	false	false	false	false
41	false	false	false	false	false	false	false	false
42	false	false	false	false	false	false	false	false
43	false	false	false	false	false	false	false	false
44	false	false	false	false	false	false	false	false
45	false	false	false	false	false	false	false	false

✓ 0s completed at 11:15PM

Figure 4(Illustration of the 'isnull()' function, indicating the presence of missing values in the DataFrame)

The 'isnull().sum()' function has been utilized to count the number of True values (missing values) along each column, as demonstrated in Figure 5. It is evident that the feature 'horsepower' is characterized by 6 missing values, while 'origin' has 2.



```
df.isnull().sum()

mpg             0
cylinders       0
displacement    0
horsepower      6
weight          0
acceleration    0
model_year      0
origin          2
dtype: int64
```

Figure 5(Count of missing values per column using 'isnull().sum()'. 'Horsepower' has 6 missing values, and 'origin' has 2.)

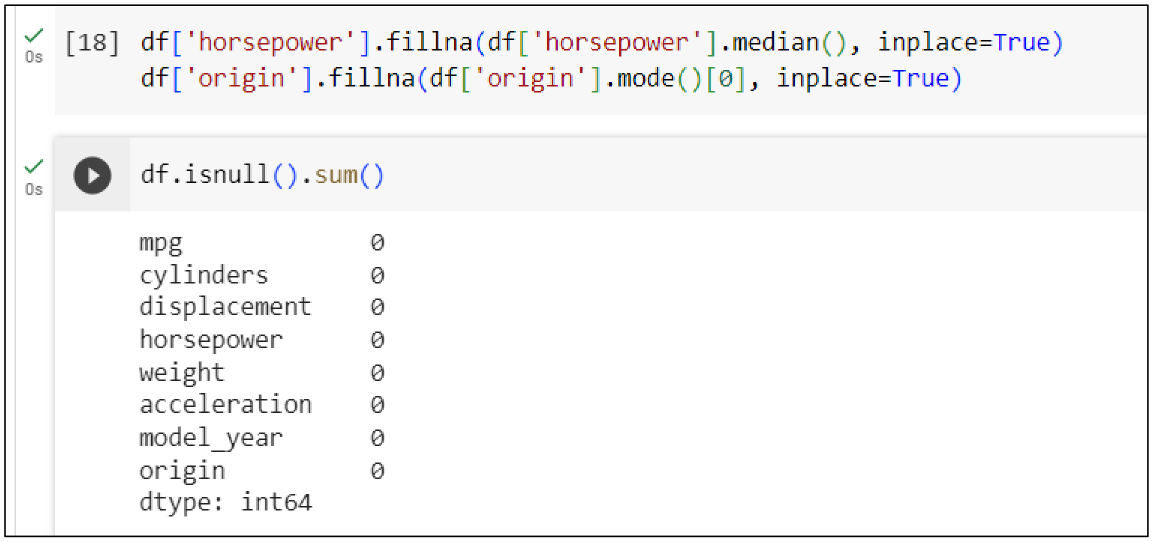
3. Fill the missing values in each feature using a proper imputation method (for example: fill with mean, median, or mode)

Missing values in the 'horsepower' and 'origin' columns of DataFrame df were filled by using the mode for 'origin' and the median for 'horsepower.'

The selection of the median for 'horsepower' is based on its positive skewness, as demonstrated in the histogram in part 5, highlighting its efficiency in handling outliers and skewed distributions for a reliable measure of central tendency. In contrast to the Mean, which is more sensitive to outliers and suitable for approximately normally distributed data. [1]

The mode is preferred for 'origin' as it is a categorical attribute identifying the most frequent category.

The original DataFrame is directly impacted by the modifications through the use of the `fillna` method with mode values.



```
[18] df['horsepower'].fillna(df['horsepower'].median(), inplace=True)
     df['origin'].fillna(df['origin'].mode()[0], inplace=True)
```

```
df.isnull().sum()
```

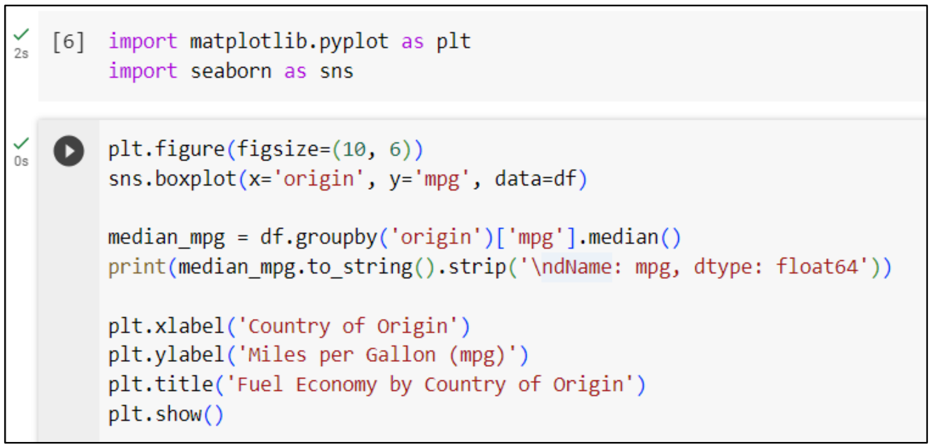
```
mpg             0
cylinders       0
displacement    0
horsepower      0
weight          0
acceleration    0
model_year      0
origin          0
dtype: int64
```

Figure 6(Filling missing values using fillna() method)

4. Which country produces cars with better fuel economy? *(Hint: use box plot that shows the mpg for each country (all countries in one plot))*

The boxplot() function was used to create box plots for each country, as seen in Figure 7. Figure 8 showcases a box plot illustrating the relationship between 'Country of Origin' and 'mpg'. To figure out which country makes cars with better fuel economy, we look at which country, on average, has the highest mpg. The median, shown by the line in each box, gives us this average.

Checking the median values for the USA, Asia, and Europe and looking at the vertical axis, we see that the median mpg for Asia is about 31.55, for Europe it's around 25.50, and for the USA, it's about 18.55. This tells us that, on average, cars from Asia have better fuel economy.

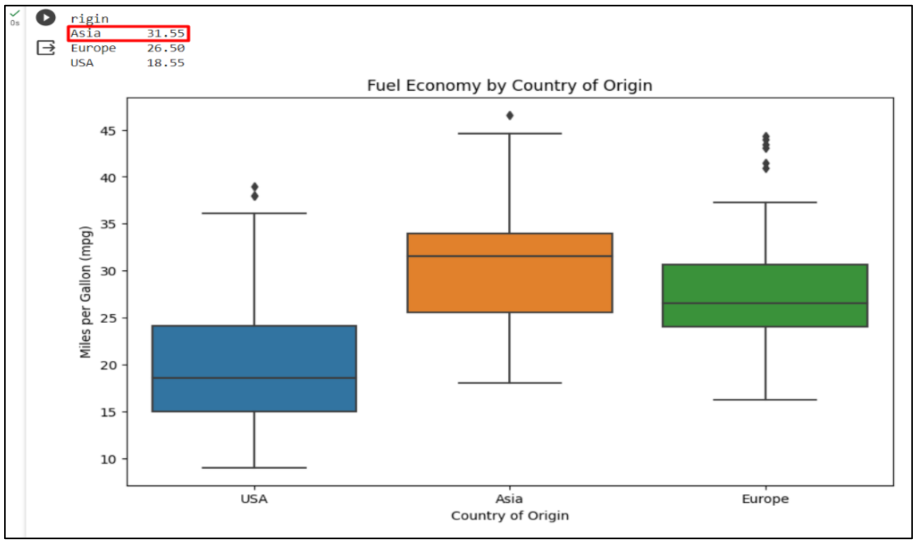


```
[6] import matplotlib.pyplot as plt
    import seaborn as sns
```

```
plt.figure(figsize=(10, 6))
sns.boxplot(x='origin', y='mpg', data=df)

median_mpg = df.groupby('origin')['mpg'].median()
print(median_mpg.to_string().strip('\ndName: mpg, dtype: float64'))

plt.xlabel('Country of Origin')
plt.ylabel('Miles per Gallon (mpg)')
plt.title('Fuel Economy by Country of Origin')
plt.show()
```

Figure 7(Utilizing 'boxplot()' to visualize 'mpg' distribution for each country of origin)

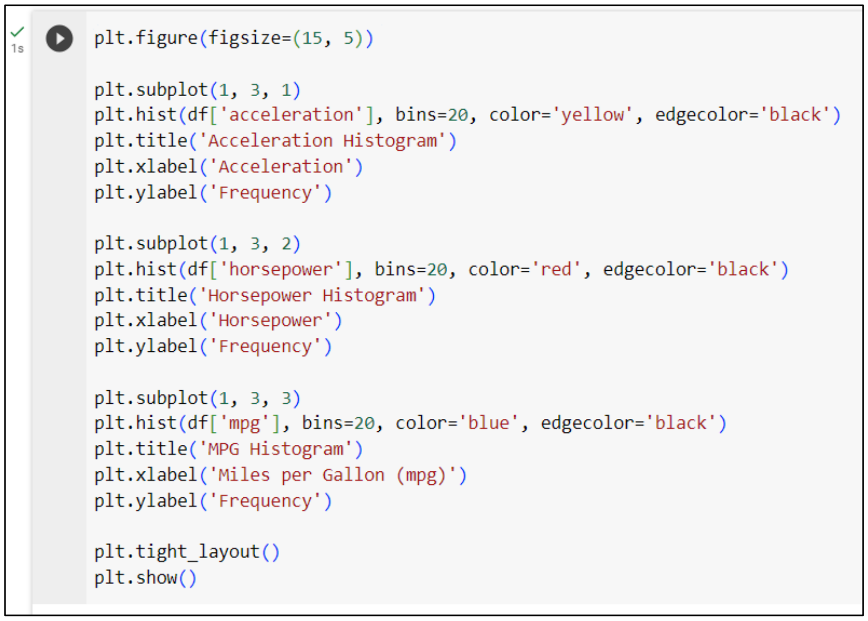
The box plot in Figure 8 gives a clear picture of how mpg varies for each country, making it easy to compare their fuel economy.



```
rigin
Asia      31.55
Europe    26.50
USA       18.55
```

Figure 8(Box plot comparing fuel efficiency (mpg) among different countries)

5. Which of the following features has a distribution that is most similar to a Gaussian: ‘acceleration’, ‘horsepower’, or ‘mpg’? Answer this part by showing the histogram of each feature.



```
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.hist(df['acceleration'], bins=20, color='yellow', edgecolor='black')
plt.title('Acceleration Histogram')
plt.xlabel('Acceleration')
plt.ylabel('Frequency')

plt.subplot(1, 3, 2)
plt.hist(df['horsepower'], bins=20, color='red', edgecolor='black')
plt.title('Horsepower Histogram')
plt.xlabel('Horsepower')
plt.ylabel('Frequency')

plt.subplot(1, 3, 3)
plt.hist(df['mpg'], bins=20, color='blue', edgecolor='black')
plt.title('MPG Histogram')
plt.xlabel('Miles per Gallon (mpg)')
plt.ylabel('Frequency')

plt.tight_layout()
plt.show()
```

Figure 9(Utilizing 'hist()' to visualize histogram distributions for 'acceleration', 'horsepower', and 'mpg')

The ‘hist()’ function has been utilized to draw the histogram plots for ‘acceleration’, ‘horsepower’, and ‘mpg’ as shown in both Figure 9 and Figure 10. It is obvious that the ‘acceleration’ is the most similar to Gaussian, while both the ‘horsepower’ and the ‘mpg’ are positively skewed.

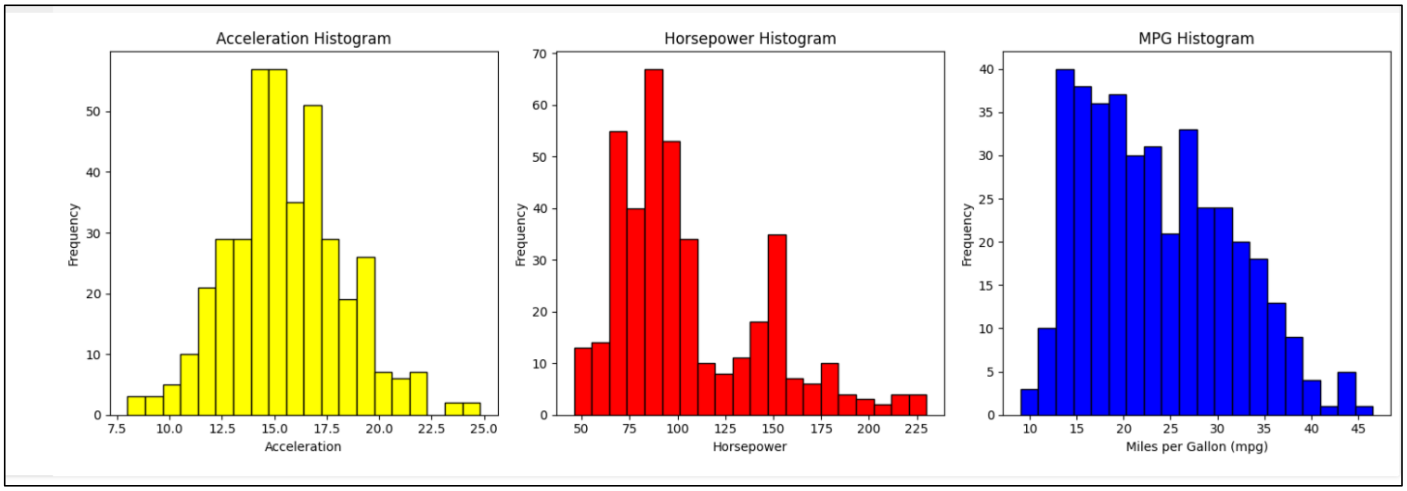
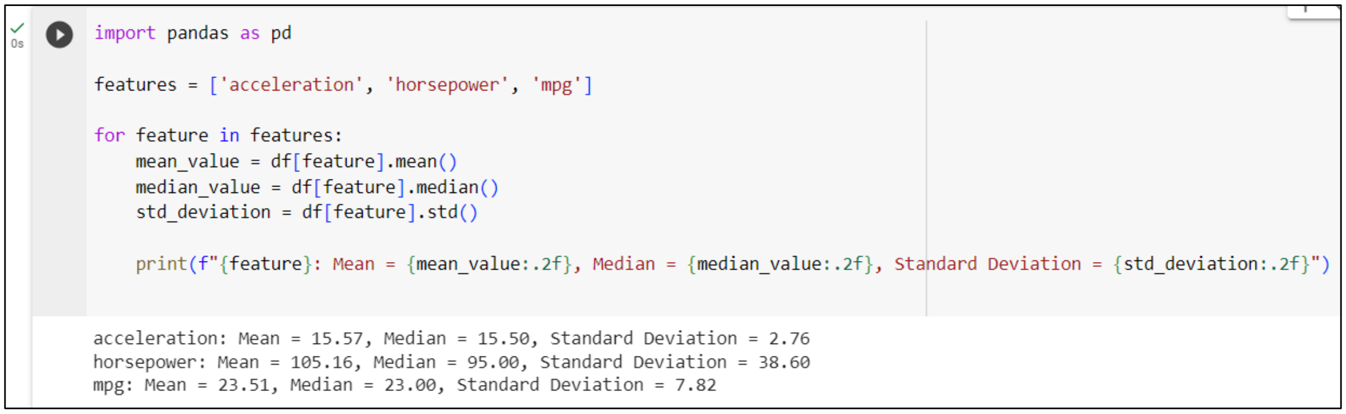


Figure 10(Comparing histograms for 'acceleration', 'horsepower', and 'mpg')

6. Support your answer for part 5 by using a quantitative measure.

The 'mean(),' 'median,' and 'std()' have been computed for each feature as shown in Figure 11. It is observed that the values of the mean and median for 'acceleration' are very close, suggesting that the data distribution may exhibit Gaussian-like properties. Additionally, it is noted that the mean value is greater than the median for both 'mpg' and 'horsepower,' indicating that the data is positively skewed, as discussed in part 5.

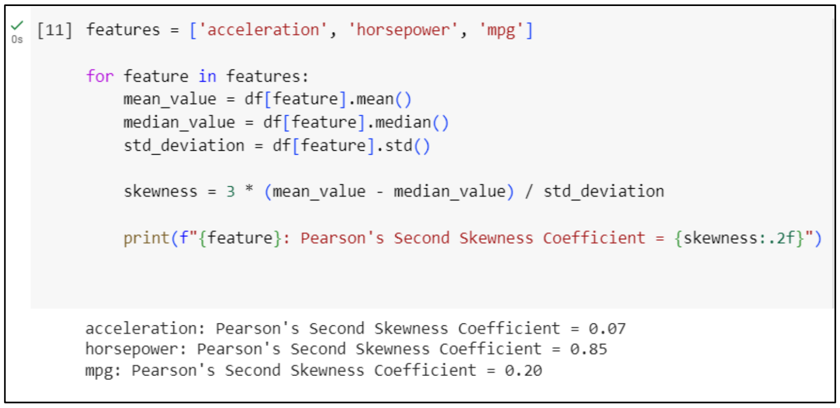


```
import pandas as pd

features = ['acceleration', 'horsepower', 'mpg']

for feature in features:
    mean_value = df[feature].mean()
    median_value = df[feature].median()
    std_deviation = df[feature].std()

    print(f"{feature}: Mean = {mean_value:.2f}, Median = {median_value:.2f}, Standard Deviation = {std_deviation:.2f}")
```

```
acceleration: Mean = 15.57, Median = 15.50, Standard Deviation = 2.76
horsepower: Mean = 105.16, Median = 95.00, Standard Deviation = 38.60
mpg: Mean = 23.51, Median = 23.00, Standard Deviation = 7.82
```

Figure 11(Comparing mean, median, and standard deviation values for 'acceleration,' 'horsepower,' and 'mpg)

The measure utilized to distinguish which of the features is mostly similar to a Gaussian distribution is Skewness, known as Pearson's second skewness coefficient. The equation S = $3(\frac{\text{mean} - \text{median}}{\text{standart deviation}})$ has been applied to all three features, and the results are shown in Figure 12. It is observed that 'horsepower' has a skewness value of S ≈ 0.8 (where S > 0), indicating a positively skewed distribution (skewed to the right). Similarly, 'mpg' exhibits the same positively skewed pattern (S= 0.2 > 0). On the other hand, 'acceleration' has S = 0.07, which is very closed to zero, suggesting a symmetric distribution (not skewed).



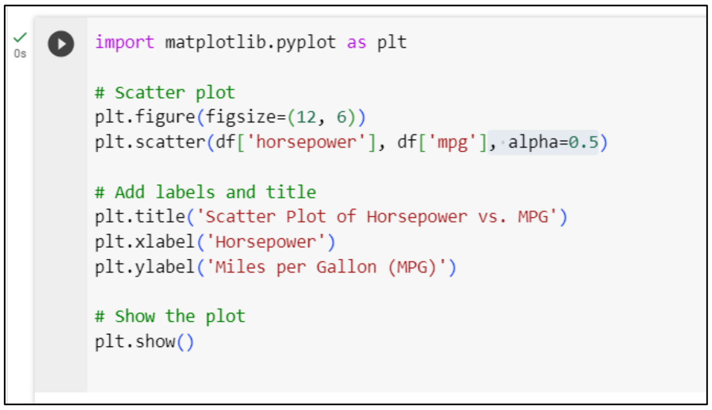
```
[11] features = ['acceleration', 'horsepower', 'mpg']

     for feature in features:
         mean_value = df[feature].mean()
         median_value = df[feature].median()
         std_deviation = df[feature].std()

         skewness = 3 * (mean_value - median_value) / std_deviation

         print(f"{feature}: Pearson's Second Skewness Coefficient = {skewness:.2f}")
```

```
acceleration: Pearson's Second Skewness Coefficient = 0.07
horsepower: Pearson's Second Skewness Coefficient = 0.85
mpg: Pearson's Second Skewness Coefficient = 0.20
```

Figure 12(Quantifying skewness using Pearson's second skewness coefficient (S) for 'acceleration,' 'horsepower,' and 'mpg)

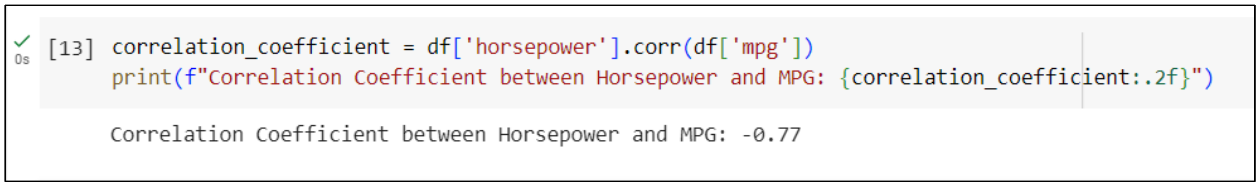
7. Plot a scatter plot that shows the ‘horsepower’ on the x-axis and ‘mpg’ on the y-axis. Is there a correlation between them? Positive or negative?

The scatter plot representing the relationship between two variables utilizes the 'scatter()' function as shown in Figure 13. The correlation coefficient has been calculated (-0.78, close to -1), signifying a strong negative correlation as shown in Figure 14. The trend of points from the top-left to the bottom-right in Figure 15 further indicates a negative correlation.



```
import matplotlib.pyplot as plt

# Scatter plot
plt.figure(figsize=(12, 6))
plt.scatter(df['horsepower'], df['mpg'], alpha=0.5)

# Add labels and title
plt.title('Scatter Plot of Horsepower vs. MPG')
plt.xlabel('Horsepower')
plt.ylabel('Miles per Gallon (MPG)')

# Show the plot
plt.show()
```

Figure 13(Code snippet utilizing 'scatter()' to create a scatter plot of 'horsepower' against 'mpg)



```
[13] correlation_coefficient = df['horsepower'].corr(df['mpg'])
     print(f"Correlation Coefficient between Horsepower and MPG: {correlation_coefficient:.2f}")

     Correlation Coefficient between Horsepower and MPG: -0.77
```

Figure 14(Calculated correlation coefficient (-0.78) indicates a strong negative correlation)

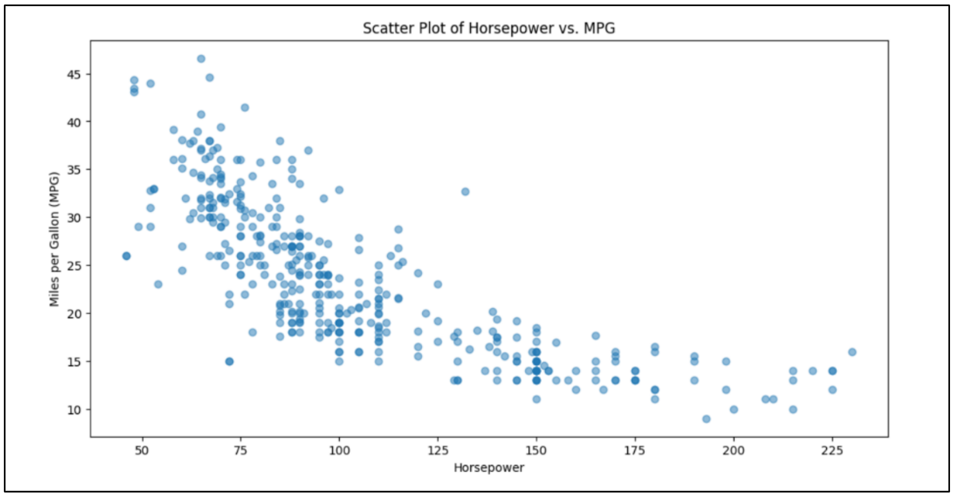
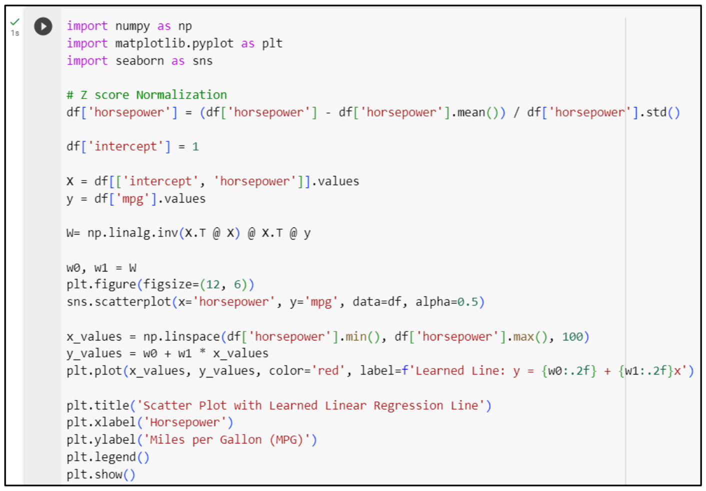


Figure 15(Scatter plot displaying the relationship between 'horsepower' and 'mpg)

8. Implement the closed form solution of linear regression and use it to learn a linear model to predict the ‘mpg’ from the ‘horsepower’. Plot the learned line on the same scatter plot you got in part 7. *(Hint: This is a simple linear regression problem (one feature). Do not forget to add x0=1 for the intercept. For inverting a matrix use np.linalg.inv* from NumPy)

In the code (Figure 16), the closed-form solution for linear regression is applied to build a predictive model for 'mpg' based on 'horsepower'. The DataFrame 'df' is extended with an intercept column, and features (X) and target variable (y) are extracted from the data frame. Using the formula $\mathbf{W} = (\boldsymbol{X^T} * \mathbf{X})^{-1} \boldsymbol{X^T} \mathbf{Y}$, the coefficients for the intercept (w0) and 'horsepower' (w1) are calculated. Figure 17 illustrates a scatter plot of 'horsepower' against 'mpg', with the learned regression line overlaid in red. This line represents the linear relationship determined by the closed-form solution, highlighting the predictive association between 'horsepower' and 'mpg'.



```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Z score Normalization
df['horsepower'] = (df['horsepower'] - df['horsepower'].mean()) / df['horsepower'].std()

df['intercept'] = 1

X = df[['intercept', 'horsepower']].values
y = df['mpg'].values

W= np.linalg.inv(X.T @ X) @ X.T @ y

w0, w1 = W
plt.figure(figsize=(12, 6))
sns.scatterplot(x='horsepower', y='mpg', data=df, alpha=0.5)

x_values = np.linspace(df['horsepower'].min(), df['horsepower'].max(), 100)
y_values = w0 + w1 * x_values
plt.plot(x_values, y_values, color='red', label=f'Learned Line: y = {w0:.2f} + {w1:.2f}x')

plt.title('Scatter Plot with Learned Linear Regression Line')
plt.xlabel('Horsepower')
plt.ylabel('Miles per Gallon (MPG)')
plt.legend()
plt.show()
```

The code shown in figure 16 also illustrates that we have done z-score normalization to ‘horsepower’ before applying it to both Closed Form Solution and Gradient Descent Algorithm (part 10). Normalizing the data can lead to faster training and better performance of the model.

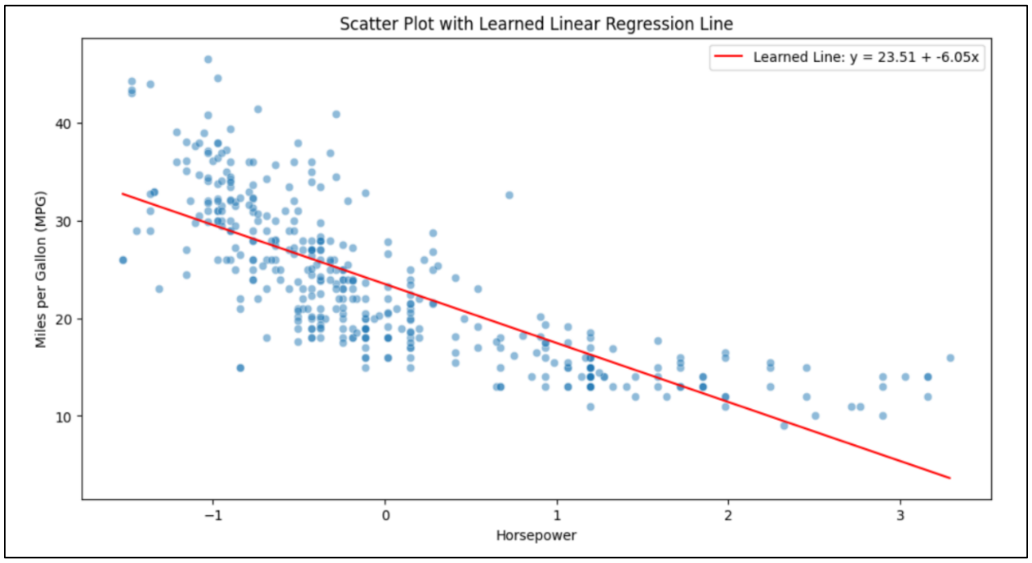


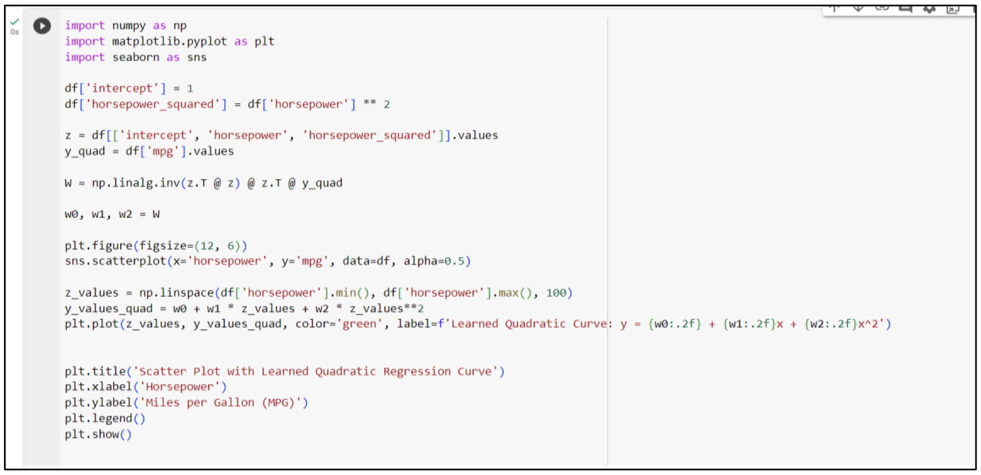
Figure 17(Scatter plot of 'horsepower' against 'mpg' with the learned linear regression line superimposed in red)

9. Repeat part 8 but now learn a quadratic function of the form

$$f = w_0 + w_1 x + w_2\, x^2$$

In this code snippet (Figure 18), a quadratic function $f = w_0 + w_1 x + w_2\, x^2$ is learned to predict 'mpg' from 'horsepower'. The DataFrame 'df' is enhanced with an intercept column and a 'horsepower_squared' column, representing the squared term. Using the closed-form solution, the coefficients (w0), (w1), and (w2) are calculated.

Nonlinear regression, as illustrated by the green curve in Figure 19, is essentially a linear weighted sum of nonlinear basis functions. This quadratic curve captures the nonlinear relationship between the features, providing a more flexible model compared to linear regression and it better fits the data.



```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

df['intercept'] = 1
df['horsepower_squared'] = df['horsepower'] ** 2

z = df[['intercept', 'horsepower', 'horsepower_squared']].values
y_quad = df['mpg'].values

W = np.linalg.inv(z.T @ z) @ z.T @ y_quad

w0, w1, w2 = W

plt.figure(figsize=(12, 6))
sns.scatterplot(x='horsepower', y='mpg', data=df, alpha=0.5)

z_values = np.linspace(df['horsepower'].min(), df['horsepower'].max(), 100)
y_values_quad = w0 + w1 * z_values + w2 * z_values**2
plt.plot(z_values, y_values_quad, color='green', label=f'Learned Quadratic Curve: y = {w0:.2f} + {w1:.2f}x + {w2:.2f}x^2')


plt.title('Scatter Plot with Learned Quadratic Regression Curve')
plt.xlabel('Horsepower')
plt.ylabel('Miles per Gallon (MPG)')
plt.legend()
plt.show()
```

Figure 18(Code implementing the closed-form solution for quadratic regression to predict 'mpg' from 'horsepower' and 'horsepower_squared')

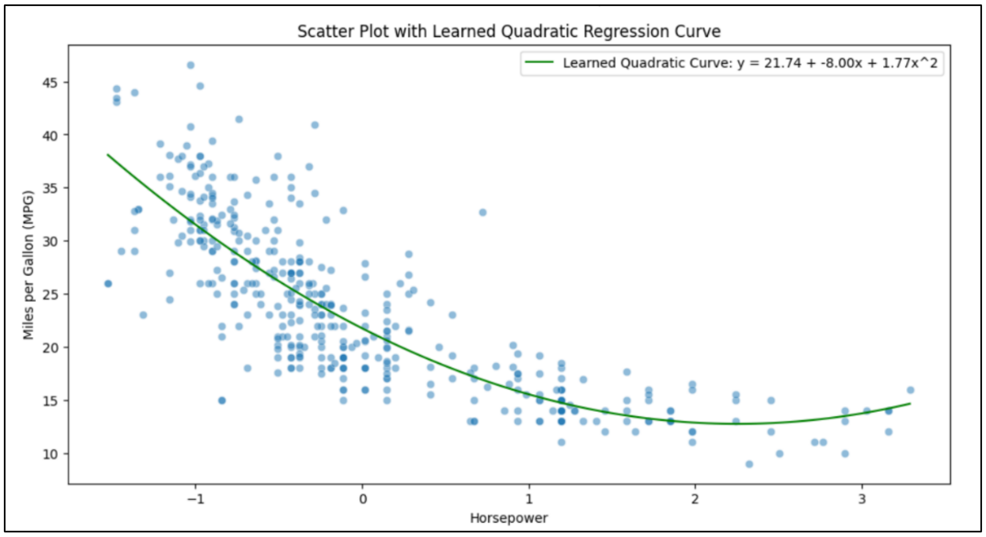
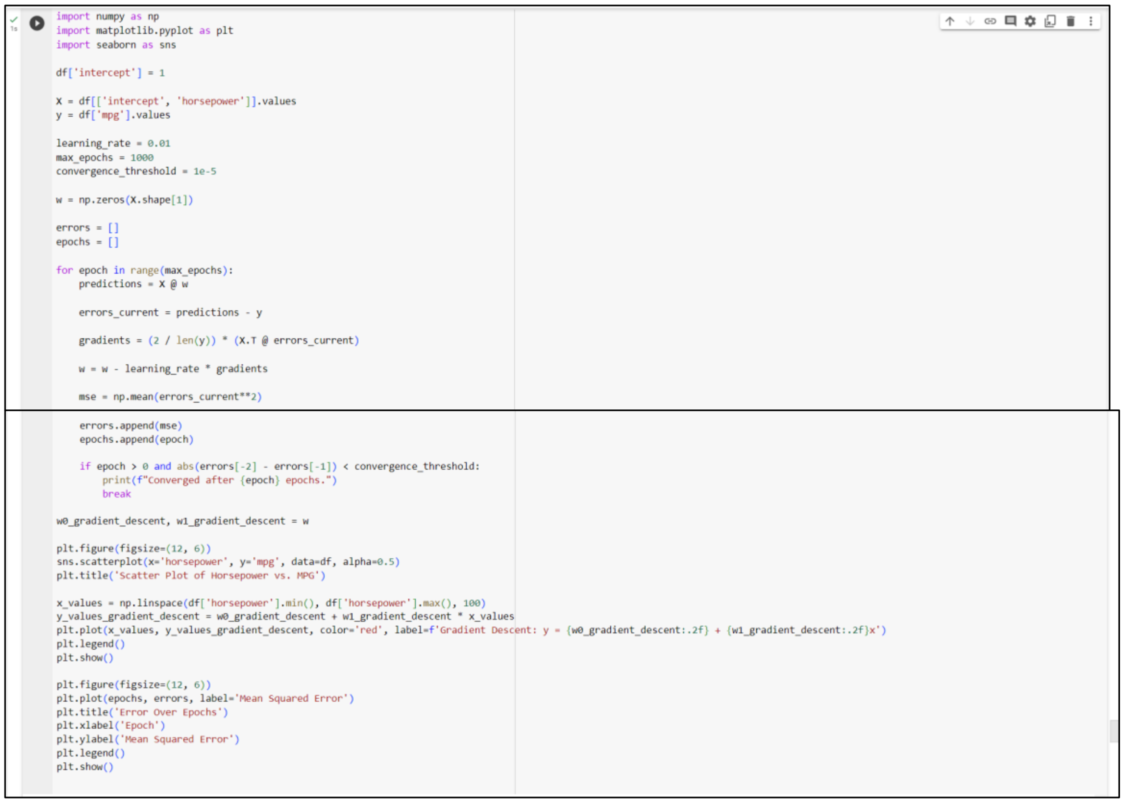


Figure 19(Scatter plot displaying the relationship between 'horsepower' and 'mpg,' featuring a green quadratic regression curve that adeptly captures nonlinear patterns)

10. Repeat part 8 (simple linear regression case) but now by implementing the gradient descent algorithm instead of the closed form solution.



```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

df['intercept'] = 1

X = df[['intercept', 'horsepower']].values
y = df['mpg'].values

learning_rate = 0.01
max_epochs = 1000
convergence_threshold = 1e-5

w = np.zeros(X.shape[1])

errors = []
epochs = []

for epoch in range(max_epochs):
    predictions = X @ w

    errors_current = predictions - y

    gradients = (2 / len(y)) * (X.T @ errors_current)

    w = w - learning_rate * gradients

    mse = np.mean(errors_current**2)
    errors.append(mse)
    epochs.append(epoch)

    if epoch > 0 and abs(errors[-2] - errors[-1]) < convergence_threshold:
        print(f"Converged after {epoch} epochs.")
        break

w0_gradient_descent, w1_gradient_descent = w

plt.figure(figsize=(12, 6))
sns.scatterplot(x='horsepower', y='mpg', data=df, alpha=0.5)
plt.title('Scatter Plot of Horsepower vs. MPG')

x_values = np.linspace(df['horsepower'].min(), df['horsepower'].max(), 100)
y_values_gradient_descent = w0_gradient_descent + w1_gradient_descent * x_values
plt.plot(x_values, y_values_gradient_descent, color='red', label=f'Gradient Descent: y = {w0_gradient_descent:.2f} + {w1_gradient_descent:.2f}x')
plt.legend()
plt.show()

plt.figure(figsize=(12, 6))
plt.plot(epochs, errors, label='Mean Squared Error')
plt.title('Error Over Epochs')
plt.xlabel('Epoch')
plt.ylabel('Mean Squared Error')
plt.legend()
plt.show()
```

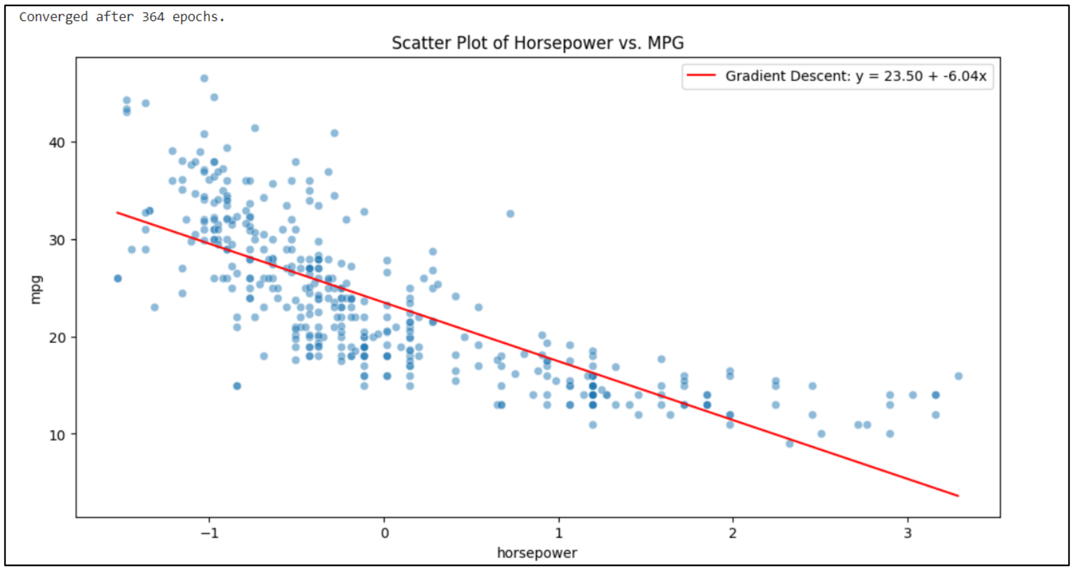
Figure 20(Implementing gradient descent for learning a linear regression model.)

Same as part 8, but Gradient Descent algorithm was applied as shown in figure 20 which differs from the closed form solution that it is an iterative method. It iteratively updates the weights of a model to minimize the error or loss function as follows:

- Choosing a starting point (Initialization)
- Calculate gradient at this point.
- Make a scaled step in the opposite direction to the gradient.
- Repeat point and until step size is smaller than the tolerance

Don't forget that we have done z-score normalization to 'horsepower' before applying algorithm in part 8. Z-score normalization is a common practice that aids convergence during gradient descent. By scaling features to have a similar range, it reduces the variance between features and helps the algorithm converge faster.

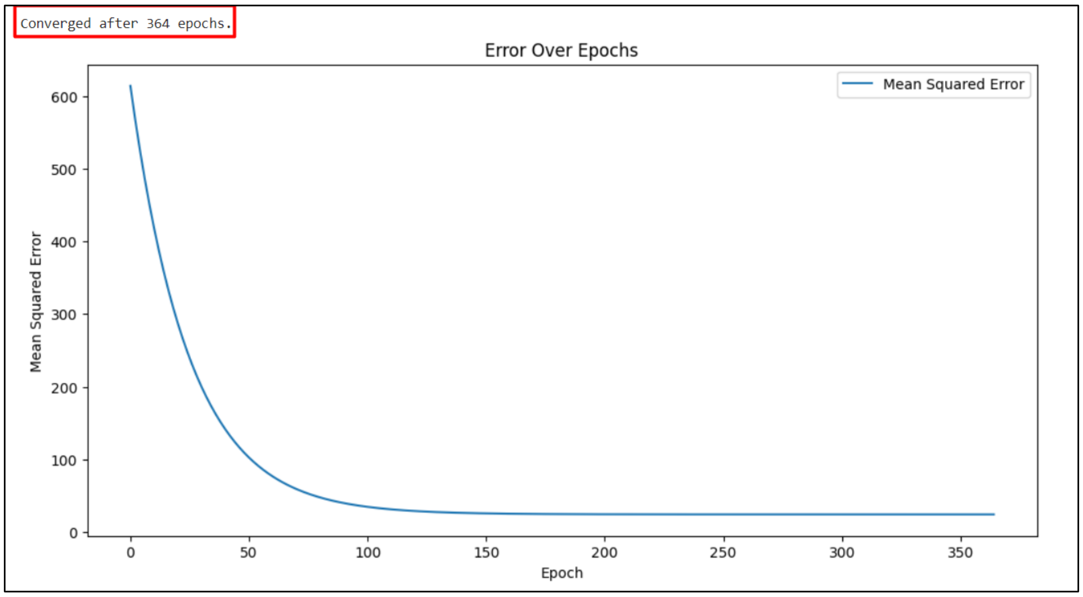
The scatter plot shown in Figure 21 illustrates the relationship between 'horsepower' and 'mpg,' with the red line representing the learned regression line obtained through gradient descent. This line closely aligns with the one derived from the closed-form solution in Part 8. Both approaches converge to the same result due to the convex nature of the objective function, ensuring a unique solution.



```
Converged after 364 epochs.
```

Figure 21(Scatter plot of 'horsepower' vs. 'mpg' with the learned regression line from gradient descent)

The Gradient Descent algorithm stops either after a set number of iterations or when it converges. As illustrated in Figure 22, the Mean Squared Error (MSE) over epochs indicates that convergence occurs around 364 iterations, where the error approaches nearly zero, indicating we've reached the solution.



```
Converged after 364 epochs.
```

Figure 22(The convergence process with Mean Squared Error (MSE) over epochs, indicating convergence around 364 iterations)