DSL251(DAV) HOMEWORK 2

Rohit Raghuwanshi
12341820
rohitrg@iitbhilai.ac.in

Latex file link-
https://www.overleaf.com/project/67a25cb170fa22672e9e32fb

Colab link-https://colab.research.google.com/drive/1xwl8sm2zuvU63ljsK72Vrj3dXQsN9shH?authuser=1#scrollTo=0QBtjnqfki0-

(This pdf consists all the questions given in homework 2)

Question 1-

CODE PART-

[1]
```
!pip install openpyxl
!pip install missingno
!pip install seaborn

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load dataset
dataset_url = "https://docs.google.com/spreadsheets/d/1-vUIn3tkLMALZ5dZVldliMO--88tQ0zQqqz5c5a8pNQ/export?format=csv"
df = pd.read_csv(dataset_url)

# --- Convert specific columns to numeric ---
numeric_columns = ['Time to Reach (hr)', 'Distance (km)']

for col in numeric_columns:
    if col in df.columns:
        df[col] = pd.to_numeric(df[col], errors='coerce')
    else:
        print(f"Warning: Column '{col}' not found in DataFrame. Skipping conversion.")

# Preprocess Data: Fill missing values with mean
numeric_columns = ['Time to Reach (hr)', 'Distance (km)']
numeric_df = df[numeric_columns]
df[numeric_columns] = numeric_df.fillna(numeric_df.mean())

# Extract features
X = df['Time to Reach (hr)'].values.reshape(-1, 1)
y = df['Distance (km)'].values.reshape(-1, 1)

# Normalize Data
X = (X - np.mean(X)) / np.std(X)
y = (y - np.mean(y)) / np.std(y)

# Initialize parameters
W = np.array([0.35])  # Initial weight
b = 0  # Initial bias
learning_rates = [0.01, 0.05, 0.1]  # Different learning rates
epochs = 100
batch_size = 8
```

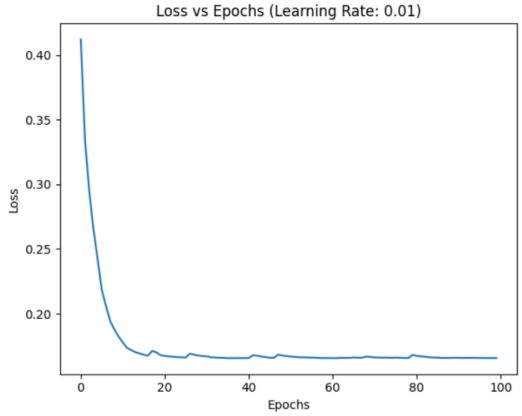
```
# Function to calculate gradient and loss
def calculate_gradient_loss(X_batch, y_batch, W, b):
    y_pred = W * X_batch + b
    dW = -2 * np.mean((y_batch - y_pred) * X_batch)
    db = -2 * np.mean(y_batch - y_pred)
    loss = np.mean((y_batch - y_pred) ** 2)
    return dW, db, loss

# Function to perform Gradient Descent
def gradient_descent(X, y, W, b, learning_rate, epochs, batch_size):
    m = len(X)
    losses = []
    gradient_norms = []

    for epoch in range(epochs):
        indices = np.random.permutation(m)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        for i in range(0, m, batch_size):
            X_batch = X_shuffled[i:i + batch_size]
            y_batch = y_shuffled[i:i + batch_size]

            dW, db, loss = calculate_gradient_loss(X_batch, y_batch, W, b)

            W -= learning_rate * dW
            b -= learning_rate * db

        loss = np.mean((y - (W * X + b)) ** 2)
        losses.append(loss)
        gradient_norms.append(np.sqrt(dW**2 + db**2))  # Gradient norm

        if epoch % 100 == 0:
            print(f"Epoch {epoch}: Loss = {loss:.4f}, W = {W[0]:.4f}, b = {b:.4f}")

    return W, b, losses, gradient_norms
```

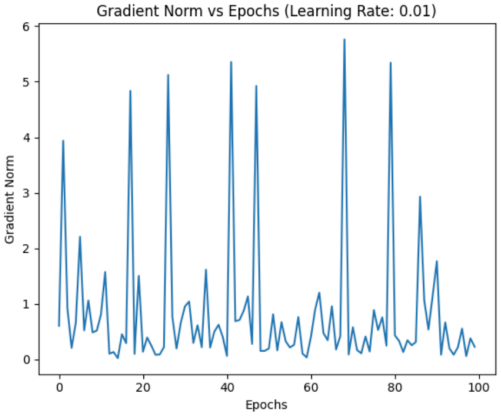
```python
# Run Gradient Descent for different learning rates
for learning_rate in learning_rates:
    print(f"\nRunning Gradient Descent with learning rate: {learning_rate}")
    W_final, b_final, losses, gradient_norms = gradient_descent(X, y, W, b, learning_rate, epochs, batch_size)

    # Plot Loss and Gradient Norm
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(range(epochs), losses)
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title(f'Loss vs Epochs (Learning Rate: {learning_rate})')

    plt.subplot(1, 2, 2)
    plt.plot(range(epochs), gradient_norms)
    plt.xlabel('Epochs')
    plt.ylabel('Gradient Norm')
    plt.title(f'Gradient Norm vs Epochs (Learning Rate: {learning_rate})')

    plt.tight_layout()
    plt.show()

# Scatter plot with regression line
plt.figure(figsize=(8, 6))
sns.regplot(x='Time to Reach (hr)', y='Distance (km)', data=df, scatter_kws={'alpha': 0.6}, line_kws={'color': 'red'})
plt.title('Scatter Plot with Regression Line')
plt.xlabel('Time to Reach (hr)')
plt.ylabel('Distance (km)')
plt.show()

# Distribution of Time to Reach (hr)
plt.figure(figsize=(8, 6))
sns.histplot(df['Time to Reach (hr)'], kde=True)
plt.title('Distribution of Time to Reach (hr)')
plt.xlabel('Time to Reach (hr)')
plt.ylabel('Frequency')
plt.show()
```

```python
# Distribution of Distance (km)
plt.figure(figsize=(8, 6))
sns.histplot(df['Distance (km)'], kde=True)
plt.title('Distribution of Distance (km)')
plt.xlabel('Distance (km)')
plt.ylabel('Frequency')
plt.show()

# Box plot of Time to Reach (hr) by Location Name (if applicable)
if 'Location Name' in df.columns:
    plt.figure(figsize=(12, 6))
    sns.boxplot(x='Location Name', y='Time to Reach (hr)', data=df)
    plt.title('Box Plot of Time to Reach (hr) by Location Name')
    plt.xticks(rotation=45, ha='right')
    plt.show()

# Box plot of Distance (km) by Location Name (if applicable)
if 'Location Name' in df.columns:
    plt.figure(figsize=(12, 6))
    sns.boxplot(x='Location Name', y='Distance (km)', data=df)
    plt.title('Box Plot of Distance (km) by Location Name')
    plt.xticks(rotation=45, ha='right')
    plt.show()
```

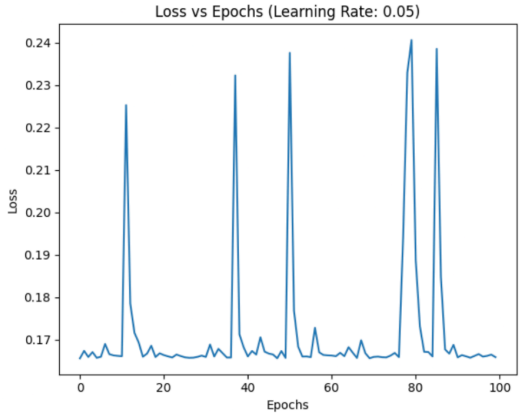
OUTPUT-

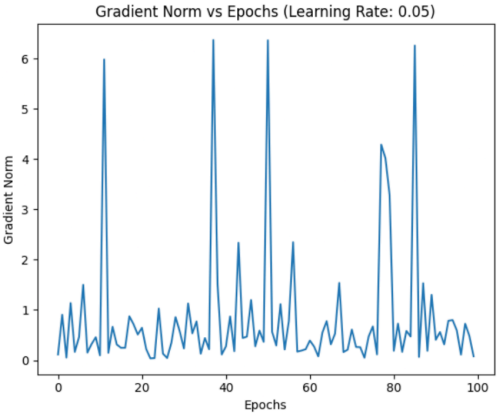
```
Running Gradient Descent with learning rate: 0.01
Epoch 0: Loss = 0.4119, W = 0.4172, b = 0.0045
```



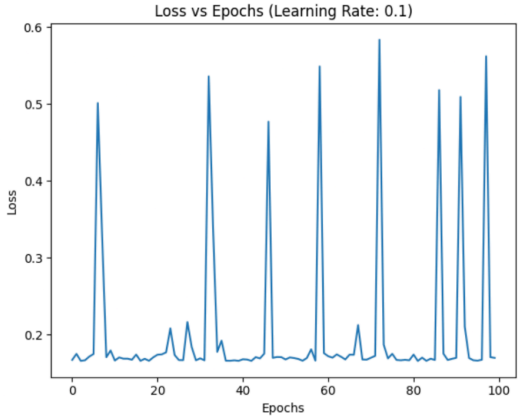


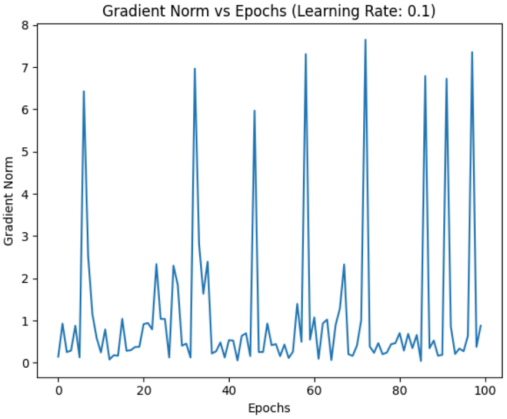

```
Running Gradient Descent with learning rate: 0.05
Epoch 0: Loss = 0.1656, W = 0.9118, b = 0.0001
```



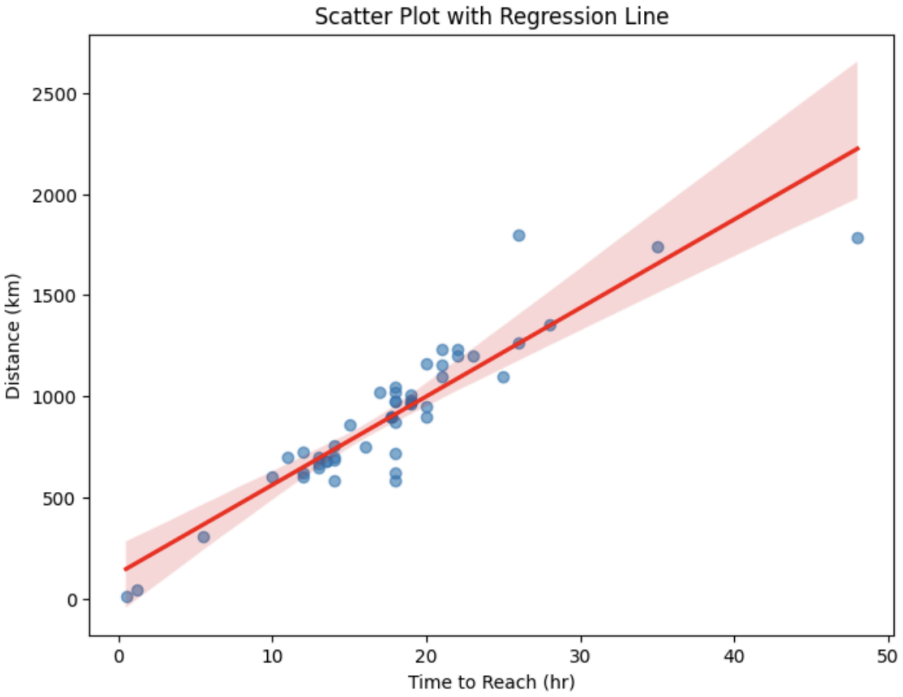


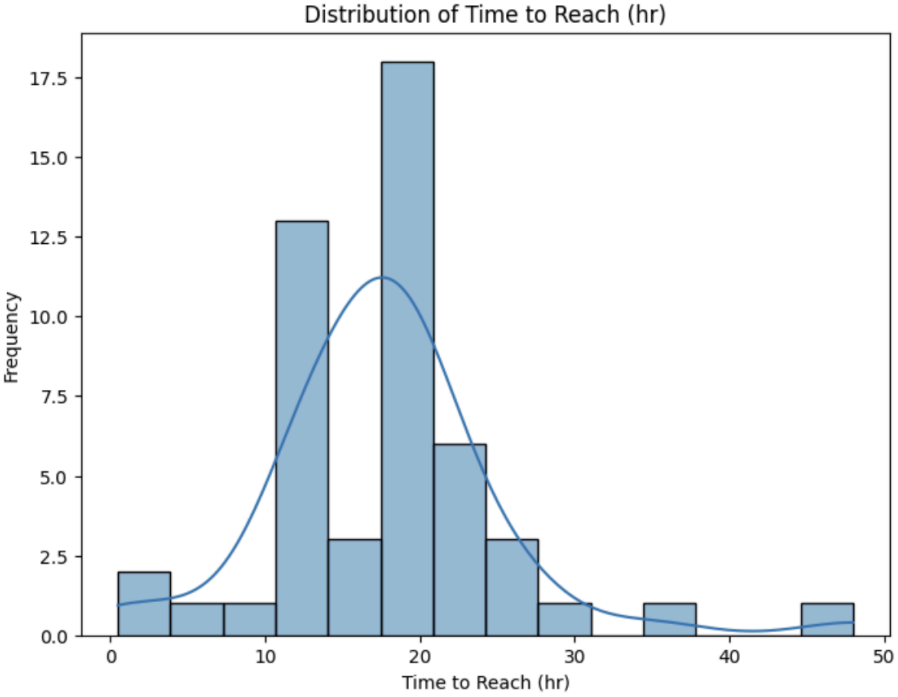

```
Running Gradient Descent with learning rate: 0.1
Epoch 0: Loss = 0.1670, W = 0.9117, b = 0.0368
```

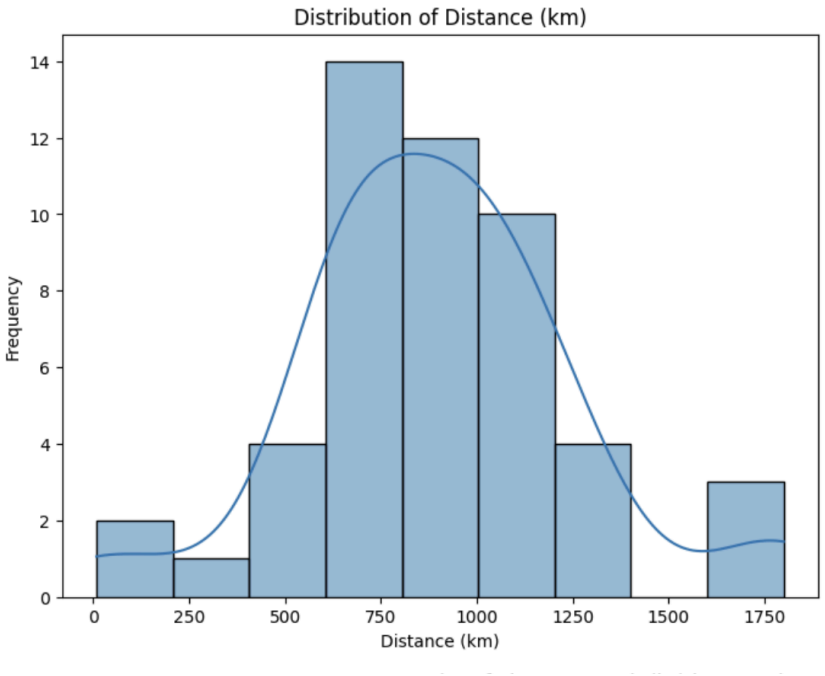


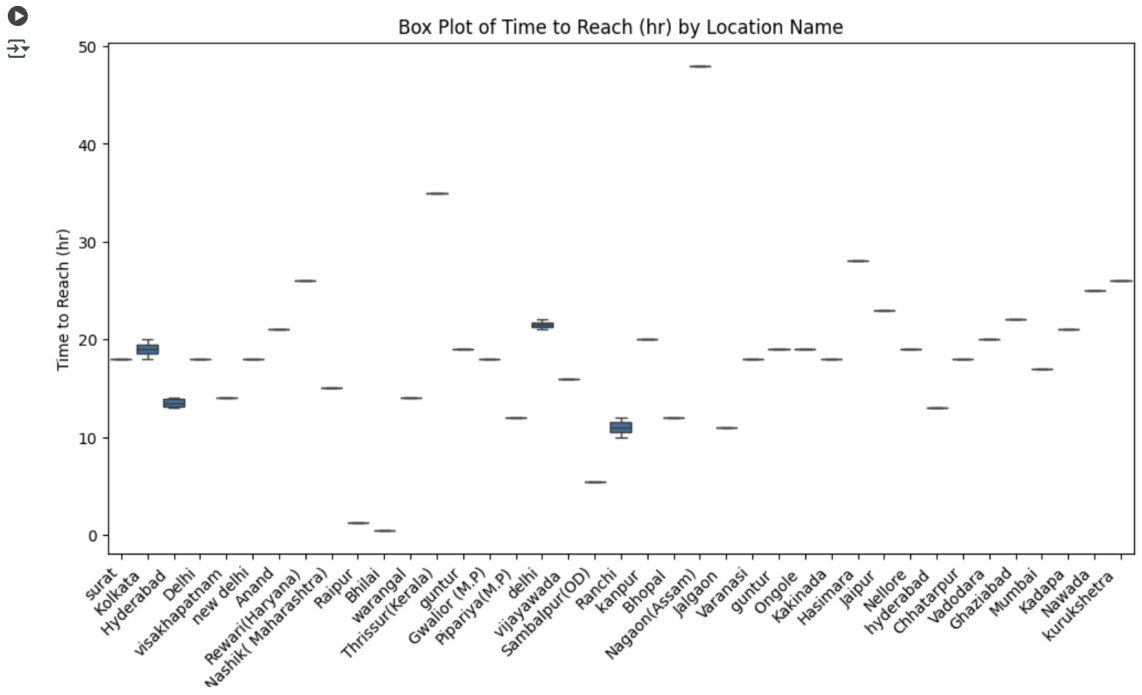


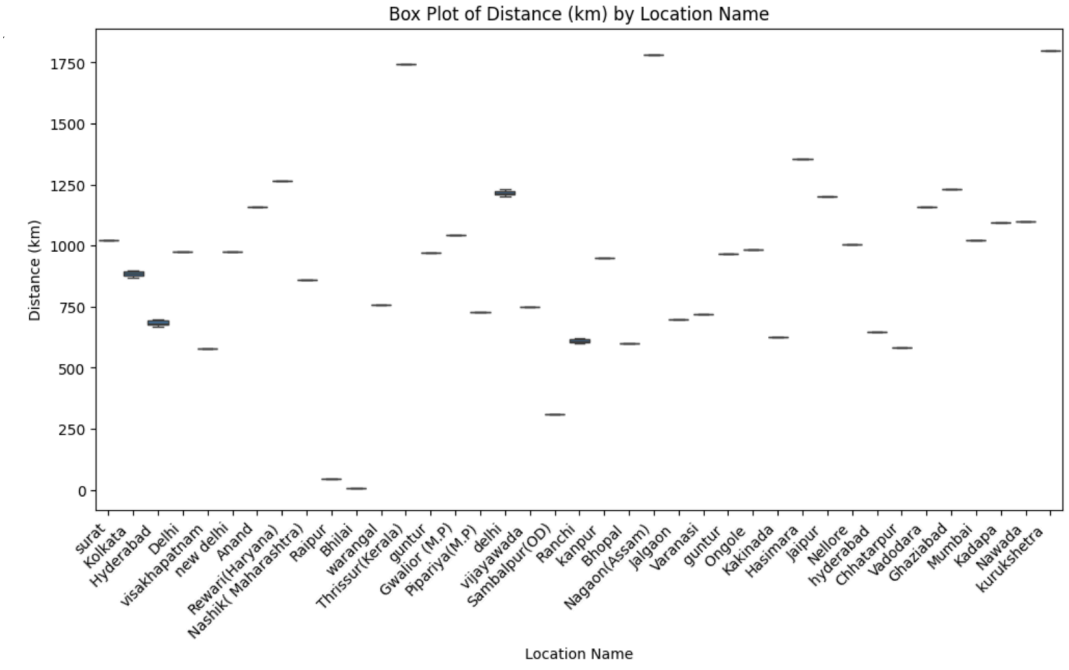
Epochs


Scatter Plot with Regression Line
Distance (km)
Time to Reach (hr)
0
500
1000
1500
2000
2500
0
10
20
30
40
50


Distribution of Time to Reach (hr)
Frequency
Time to Reach (hr)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
0
10
20
30
40
50


Distribution of Distance (km)
Frequency
Distance (km)
0
2
4
6
8
10
12
14
0
250
500
750
1000
1250
1500
1750


Box Plot of Time to Reach (hr) by Location Name
Time to Reach (hr)
0
10
20
30
40
50
surat
Kolkata
Hyderabad
Delhi
visakhapatnam
new delhi
Anand
Rewari(Haryana)
Nashik(Maharashtra)
Raipur
Bhilai
warangal
Thrissur(Kerala)
guntur
Gwalior (M.P)
Pipariya(M.P)
delhi
vijayawada
Sambalpur(OD)
Ranchi
kanpur
Bhopal
Nagaon(Assam)
Jalgaon
Varanasi
guntur
Ongole
Kakinada
Hasimara
Jaipur
Nellore
hyderabad
Chhatarpur
Vadodara
Ghaziabad
Mumbai
Kadapa
Nawada
kurukshetra


Box Plot of Distance (km) by Location Name
Distance (km)
0
250
500
750
1000
1250
1500
1750
surat
Kolkata
Hyderabad
Delhi
visakhapatnam
new delhi
Anand
Rewari(Haryana)
Nashik(Maharashtra)
Raipur
Bhilai
warangal
Thrissur(Kerala)
guntur
Gwalior (M.P)
Pipariya(M.P)
delhi
vijayawada
Sambalpur(OD)
Ranchi
kanpur
Bhopal
Nagaon(Assam)
Jalgaon
Varanasi
guntur
Ongole
Kakinada
Hasimara
Jaipur
Nellore
hyderabad
Chhatarpur
Vadodara
Ghaziabad
Mumbai
Kadapa
Nawada
kurukshetra
Location Name

LATEX CODE PART-

```
\documentclass{article}
\usepackage{minted}  % For code formatting
\usepackage[a4paper, margin=1in]{geometry} % Adjust page margins
\usepackage{graphicx} % For images if needed

\begin{document}

\title{Gradient Descent and Data Visualization}
\author{ROHIT RAGHUWANSHI}
\maketitle

\section{Python Code}

\begin{minted}[fontsize=\small, breaklines]{python}
!pip install openpyxl
!pip install missingno
!pip install seaborn

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load dataset
dataset_url = "https://docs.google.com/spreadsheets/d/1-
vUIn3tkLMALZ5dZVldliMO--88tQ0zQqqz5c5a8pNQ/export?format=csv"
df = pd.read_csv(dataset_url)

# --- Convert specific columns to numeric ---
numeric_columns = ['Time to Reach (hr)', 'Distance (km)']

for col in numeric_columns:
    if col in df.columns:
        df[col] = pd.to_numeric(df[col], errors='coerce')
    else:
        print(f"Warning: Column '{col}' not found in DataFrame. Skipping
        conversion.")

# Preprocess Data: Fill missing values with mean
numeric df = df[numeric columns]
```

```
# Preprocess Data: Fill missing values with mean
numeric_df = df[numeric_columns]
df[numeric_columns] = numeric_df.fillna(numeric_df.mean())

# Extract features
X = df['Time to Reach (hr)'].values.reshape(-1, 1)
y = df['Distance (km)'].values.reshape(-1, 1)

# Normalize Data
X = (X - np.mean(X)) / np.std(X)
y = (y - np.mean(y)) / np.std(y)

# Initialize parameters
W = np.array([0.35])  # Initial weight
b = 0  # Initial bias
learning_rates = [0.01, 0.05, 0.1]  # Different learning rates
epochs = 100
batch_size = 8

# Function to calculate gradient and loss
def calculate_gradient_loss(X_batch, y_batch, W, b):
    y_pred = W * X_batch + b
    dW = -2 * np.mean((y_batch - y_pred) * X_batch)
    db = -2 * np.mean(y_batch - y_pred)
    loss = np.mean((y_batch - y_pred) ** 2)
    return dW, db, loss

# Function to perform Gradient Descent
def gradient_descent(X, y, W, b, learning_rate, epochs, batch_size):
    m = len(X)
    losses = []
    gradient_norms = []

    for epoch in range(epochs):
        indices = np.random.permutation(m)
        X_shuffled = X[indices]
        y_shuffled = y[indices]

        for i in range(0, m, batch_size):
```

```
            X_batch = X_shuffled[i:i + batch_size]
            y_batch = y_shuffled[i:i + batch_size]

            dW, db, loss = calculate_gradient_loss(X_batch, y_batch, W, b)

            W -= learning_rate * dW
            b -= learning_rate * db

        loss = np.mean((y - (W * X + b)) ** 2)
        losses.append(loss)
        gradient_norms.append(np.sqrt(dW**2 + db**2))  # Gradient norm

        if epoch % 100 == 0:
            print(f"Epoch {epoch}: Loss = {loss:.4f}, W = {W[0]:.4f}, b =
            {b:.4f}")

    return W, b, losses, gradient_norms

# Run Gradient Descent for different learning rates
for learning_rate in learning_rates:
    print(f"\nRunning Gradient Descent with learning rate: {learning_rate}")
    W_final, b_final, losses, gradient_norms = gradient_descent(X, y, W, b,
    learning_rate, epochs, batch_size)

    # Plot Loss and Gradient Norm
    plt.figure(figsize=(12, 5))
    plt.subplot(1, 2, 1)
    plt.plot(range(epochs), losses)
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title(f'Loss vs Epochs (Learning Rate: {learning_rate})')

    plt.subplot(1, 2, 2)
    plt.plot(range(epochs), gradient_norms)
    plt.xlabel('Epochs')
    plt.ylabel('Gradient Norm')
    plt.title(f'Gradient Norm vs Epochs (Learning Rate: {learning_rate})')

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

```
# Scatter plot with regression line
plt.figure(figsize=(8, 6))
sns.regplot(x='Time to Reach (hr)', y='Distance (km)', data=df, scatter_kw
{'alpha': 0.6}, line_kws={'color': 'red'})
plt.title('Scatter Plot with Regression Line')
plt.xlabel('Time to Reach (hr)')
plt.ylabel('Distance (km)')
plt.show()

# Distribution of Time to Reach (hr)
plt.figure(figsize=(8, 6))
sns.histplot(df['Time to Reach (hr)'], kde=True)
plt.title('Distribution of Time to Reach (hr)')
plt.xlabel('Time to Reach (hr)')
plt.ylabel('Frequency')
plt.show()

# Distribution of Distance (km)
plt.figure(figsize=(8, 6))
sns.histplot(df['Distance (km)'], kde=True)
plt.title('Distribution of Distance (km)')
plt.xlabel('Distance (km)')
plt.ylabel('Frequency')
plt.show()

# Box plot of Time to Reach (hr) by Location Name (if applicable)
if 'Location Name' in df.columns:
    plt.figure(figsize=(12, 6))
    sns.boxplot(x='Location Name', y='Time to Reach (hr)', data=df)
    plt.title('Box Plot of Time to Reach (hr) by Location Name')
    plt.xticks(rotation=45, ha='right')
    plt.show()

# Box plot of Distance (km) by Location Name (if applicable)
if 'Location Name' in df.columns:
    plt.figure(figsize=(12, 6))
    sns.boxplot(x='Location Name', y='Distance (km)', data=df)
    plt.title('Box Plot of Distance (km) by Location Name')
    plt.xticks(rotation=45, ha='right')
    plt.show()
```

```
    plt.show()
\end{minted}

\end{document}
```

OUTPUT(LATEX)-

Gradient Descent and Data Visualization

ROHIT RAGHUWANSHI

February 5, 2025

1 Python Code

```
!pip install openpyxl
!pip install missingno
!pip install seaborn

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Load dataset
dataset_url =
↪  "https://docs.google.com/spreadsheets/d/1-vUIn3tkLMALZ5dZVldliMO--88tQ0zQqqz5c5a8pNQ
df = pd.read_csv(dataset_url)

# --- Convert specific columns to numeric ---
numeric_columns = ['Time to Reach (hr)', 'Distance (km)']

for col in numeric_columns:
    if col in df.columns:
        df[col] = pd.to_numeric(df[col], errors='coerce')
    else:
        print(f"Warning: Column '{col}' not found in DataFrame. Skipping conversion.")

# Preprocess Data: Fill missing values with mean
numeric_df = df[numeric_columns]
df[numeric_columns] = numeric_df.fillna(numeric_df.mean())

# Extract features
X = df['Time to Reach (hr)'].values.reshape(-1, 1)
y = df['Distance (km)'].values.reshape(-1, 1)

# Normalize Data
X = (X - np.mean(X)) / np.std(X)
y = (y - np.mean(y)) / np.std(y)
```

```python
# Initialize parameters
W = np.array([0.35])  # Initial weight
b = 0  # Initial bias
learning_rates = [0.01, 0.05, 0.1]  # Different learning rates
epochs = 100
batch_size = 8

# Function to calculate gradient and loss
def calculate_gradient_loss(X_batch, y_batch, W, b):
    y_pred = W * X_batch + b
    dW = -2 * np.mean((y_batch - y_pred) * X_batch)
    db = -2 * np.mean(y_batch - y_pred)
    loss = np.mean((y_batch - y_pred) ** 2)
    return dW, db, loss
```

```
    # Function to perform Gradient Descent
    def gradient_descent(X, y, W, b, learning_rate, epochs, batch_size):
        m = len(X)
        losses = []
        gradient_norms = []

        for epoch in range(epochs):
            indices = np.random.permutation(m)
            X_shuffled = X[indices]
            y_shuffled = y[indices]

            for i in range(0, m, batch_size):
                X_batch = X_shuffled[i:i + batch_size]
                y_batch = y_shuffled[i:i + batch_size]

                dW, db, loss = calculate_gradient_loss(X_batch, y_batch, W, b)

                W -= learning_rate * dW
                b -= learning_rate * db

            loss = np.mean((y - (W * X + b)) ** 2)
            losses.append(loss)
            gradient_norms.append(np.sqrt(dW**2 + db**2))  # Gradient norm

            if epoch % 100 == 0:
                print(f"Epoch {epoch}: Loss = {loss:.4f}, W = {W[0]:.4f}, b = {b:.4f}")

        return W, b, losses, gradient_norms

    # Run Gradient Descent for different learning rates
    for learning_rate in learning_rates:
        print(f"\nRunning Gradient Descent with learning rate: {learning_rate}")
        W_final, b_final, losses, gradient_norms = gradient_descent(X, y, W, b, learning_rate,
        ↪  epochs, batch_size)

        # Plot Loss and Gradient Norm
        plt.figure(figsize=(12, 5))
        plt.subplot(1, 2, 1)
        plt.plot(range(epochs), losses)
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.title(f'Loss vs Epochs (Learning Rate: {learning_rate})')

        plt.subplot(1, 2, 2)
        plt.plot(range(epochs), gradient_norms)
        plt.xlabel('Epochs')
        plt.ylabel('Gradient Norm')
        plt.title(f'Gradient Norm vs Epochs (Learning Rate: {learning_rate})')

        plt.tight_layout()
        plt.show()

    # Scatter plot with regression line
    plt.figure(figsize=(8, 6))
    sns.regplot(x='Time to Reach (hr)', y='Distance (km)', data=df, scatter_kws={'alpha': 0.6},
    ↪  line_kws={'color': 'red'})
    plt.title('Scatter Plot with Regression Line')
    plt.xlabel('Time to Reach (hr)')
    plt.ylabel('Distance (km)')
    plt.show()
```

```
# Distribution of Time to Reach (hr)
```



```
plt.figure(figsize=(8, 6))
sns.histplot(df['Time to Reach (hr)'], kde=True)
plt.title('Distribution of Time to Reach (hr)')
plt.xlabel('Time to Reach (hr)')
plt.ylabel('Frequency')
plt.show()

# Distribution of Distance (km)
plt.figure(figsize=(8, 6))
sns.histplot(df['Distance (km)'], kde=True)
plt.title('Distribution of Distance (km)')
plt.xlabel('Distance (km)')
plt.ylabel('Frequency')
plt.show()

# Box plot of Time to Reach (hr) by Location Name (if applicable)
if 'Location Name' in df.columns:
    plt.figure(figsize=(12, 6))
    sns.boxplot(x='Location Name', y='Time to Reach (hr)', data=df)
    plt.title('Box Plot of Time to Reach (hr) by Location Name')
    plt.xticks(rotation=45, ha='right')
    plt.show()

# Box plot of Distance (km) by Location Name (if applicable)
if 'Location Name' in df.columns:
    plt.figure(figsize=(12, 6))
    sns.boxplot(x='Location Name', y='Distance (km)', data=df)
    plt.title('Box Plot of Distance (km) by Location Name')
    plt.xticks(rotation=45, ha='right')
    plt.show()
```

Key Observations-

1. Dataset Handling:
    - Loads data from a Google Sheets CSV link.

    - Converts `Time to Reach (hr)` and `Distance (km)` to numeric, handling missing values by filling them with column means.
2. Feature Processing:
    - Extracts `X` (Time) and `y` (Distance), then normalizes them.
3. Gradient Descent Implementation:
    - Uses Batch Gradient Descent with different learning rates (`0.01, 0.05, 0.1`).
    - Updates weights (`W`) and bias (`b`) iteratively for `100` epochs using mean squared error (MSE) loss.
4. Visualization of Training:
    - Plots Loss vs. Epochs and Gradient Norm vs. Epochs to analyze convergence.
5. Data Visualizations:
    - Scatter Plot with Regression Line (Seaborn `regplot`).
    - Histograms for `Time to Reach (hr)` and `Distance (km)`.
    - Box Plots grouped by `Location Name` (if applicable).

Main Takeaways:

Uses gradient descent for regression.
Handles missing values & normalization.
Compare different learning rates.
Provides multiple data visualizations.

QUESTION 2-

ROHIT RAGHUWANSHI(12341820)

QUESTION NO 5.5

Functions

1. $f_1(x) = \sin(x_1)\cos(x_2), \quad x \in \mathbb{R}^2$
2. $f_2(x, y) = x^\top y, \quad x, y \in \mathbb{R}^n$
3. $f_3(x) = xx^\top, \quad x \in \mathbb{R}^n$

Part a: Dimensions of

$\frac{\partial f_i}{\partial x}$ 1. For $f_1(x)$: - $x \in \mathbb{R}^2$ implies x_1 and x_2 are scalars. - Dimension of $\frac{\partial f_1}{\partial x}$: $\mathbb{R}^2$ (1 output, 2 inputs).

2. For $f_2(x, y)$: - $x, y \in \mathbb{R}^n$ implies $x^\top y$ is a scalar. - Dimension of $\frac{\partial f_2}{\partial x}$: $\mathbb{R}^n$ (1 output, n inputs).

3. For $f_3(x)$: - $xx^\top$ results in an $n \times n$ matrix. - Dimension of $\frac{\partial f_3}{\partial x}$: $\mathbb{R}^{n \times n}$ (n outputs, n inputs).

Part b: Compute the Jacobians

1. For $f_1(x)$:

$$J_{f_1} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \end{bmatrix} = \begin{bmatrix} \cos(x_1)\cos(x_2) & -\sin(x_1)\sin(x_2) \end{bmatrix}$$

Dimension: 1×2.

2. For $f_2(x, y)$:

$$J_{f_2} = \begin{bmatrix} \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \frac{\partial f_2}{\partial y_1} & \cdots & \frac{\partial f_2}{\partial y_n} \end{bmatrix} = \begin{bmatrix} y^\top & x^\top \\ x^\top & 0 \end{bmatrix}$$

Dimension: $1 \times n$.

3. For $f_3(x)$:

$$J_{f_3} = \frac{\partial (xx^\top)}{\partial x} = 2x\mathrm{d}x^\top \quad \text{(using the product rule)}$$

Dimension: $n \times n$.

QUESTION NO 5.8

Part a

Compute the derivative $\frac{df}{dx}$ using the chain rule.
The function is given as:

$$f(z) = \exp\left(-\frac{1}{2}z\right), \quad z = g(y) = S^{-1}y, \quad y = h(x) = x - \mu$$

Step 1: Understanding the Components

- $x, \mu \in \mathbb{R}^D$
- $S \in \mathbb{R}^{D \times D}$
- $y = h(x)$: dimension D
- $z = g(y)$: dimension D
- $f(z)$: dimension $\mathbb{R}$

Step 2: Applying the Chain Rule

$$\frac{df}{dx} = \frac{df}{dz} \cdot \frac{dz}{dy} \cdot \frac{dy}{dx}$$

Step 3: Compute Each Partial Derivative

1. $\frac{df}{dz} = -\frac{1}{2}\exp\left(-\frac{1}{2}z\right)$ Dimension: $\mathbb{R}$
2. $\frac{dz}{dy} = S^{-1}$ Dimension: $\mathbb{R}^{D \times D}$
3. $\frac{dy}{dx} = I$ Dimension: $\mathbb{R}^{D \times D}$

Step 4: Final Derivative

$$\frac{df}{dx} = \frac{1}{2}f(z)S^{-1}$$

Part b

Compute the derivative $\frac{df}{dx}$ for

$$f(x) = \text{tr}(xx^\top + \sigma^2 I), \quad x \in \mathbb{R}^D$$

Step 1: Understanding the Components

- $xx^\top$: matrix dimension $D \times D$
- I: identity matrix, dimension $D \times D$
- $f(x)$ has dimension $\mathbb{R}$.

Step 2: Applying the Derivative

$$\frac{df}{dx} = \frac{d}{dx}\text{tr}(xx^\top) = 2x^\top$$

Dimension: $\mathbb{R}^D$

Part c

Compute the derivative $\frac{df}{dx}$ using the chain rule for

$$f = \tanh(z), \quad z = Ax + b, \quad x \in \mathbb{R}^N, \quad A \in \mathbb{R}^{M \times N}, \quad b \in \mathbb{R}^M$$

Step 1: Understanding the Components

- x: dimension N
- z: dimension M
- f: dimension M

Step 2: Applying the Chain Rule

$$\frac{df}{dx} = \frac{df}{dz} \cdot \frac{dz}{dx}$$

Step 3: Compute Each Partial Derivative

1. $\frac{df}{dz} = \text{diag}(1 - \tanh^2(z))$ Dimension: $M \times M$
2. $\frac{dz}{dx} = A$ Dimension: $M \times N$

Step 4: Final Derivative

$$\frac{df}{dx} = \text{diag}(1 - \tanh^2(z))A$$

Dimension: $M \times N$

LATEX CODE-

```latex
\documentclass{article}
\usepackage{amsmath}
\usepackage{amsfonts}
\usepackage{titlesec}

% Set font size for section titles
\titleformat{\section}{\huge\bfseries}

\begin{document}

\begin{center}

    {\Huge ROHIT RAGHUWANSHI(12341820)}\\

    {\Large QUESTION NO 5.5}\\
\end{center}

% Functions
\section*{Functions}
1. \( f_1(x) = \sin(x_1) \cos(x_2), \quad x \in \mathbb{R}^2 \)

2. \( f_2(x, y) = x^{\top} y, \quad x, y \in \mathbb{R}^n \)

3. \( f_3(x) = xx^{\top}, \quad x \in \mathbb{R}^n \)

% Part a: Dimensions of Partial Derivatives
\subsection*{Part a: Dimensions of } \( \frac{\partial f_i}{\partial x} \)
1. For \( f_1(x) \):
   - \( x \in \mathbb{R}^2 \) implies \( x_1 \) and \( x_2 \) are scalars.
   - Dimension of \( \frac{\partial f_1}{\partial x} \): \( \mathbb{R}^2 \) (1 output, 2 inputs).

2. For \( f_2(x, y) \):
   - \( x, y \in \mathbb{R}^n \) implies \( x^{\top} y \) is a scalar.
   - Dimension of \( \frac{\partial f_2}{\partial x} \): \( \mathbb{R}^{n} \) (1 output, n inputs).

3. For \( f_3(x) \):
   - \( xx^{\top} \) results in an \( n \times n \) matrix.
   - Dimension of \( \frac{\partial f_3}{\partial x} \): \( \mathbb{R}^{n \times n} \) (n outputs, n inputs).

% Part b: Jacobians
\subsection*{Part b: Compute the Jacobians}
1. For \( f_1(x) \):
   \[
   J_{f_1} = \begin{bmatrix}
   \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2}
   \end{bmatrix} = \begin{bmatrix}
   \cos(x_1) \cos(x_2) & -\sin(x_1) \sin(x_2)
   \end{bmatrix}
   \]
```

```
   Dimension: \( 1 \times 2 \).

2. For \( f_2(x, y) \):
   \[
   J_{f_2} = \begin{bmatrix}
   \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_2}{\partial x_n} \\
   \frac{\partial f_2}{\partial y_1} & \cdots & \frac{\partial f_2}{\partial y_n}
   \end{bmatrix} =
   \begin{bmatrix}
   y^{\top} & x^{\top} \\
   x^{\top} & 0
   \end{bmatrix}
   \]
   Dimension: \( 1 \times n \).

3. For \( f_3(x) \):
   \[
   J_{f_3} = \frac{\partial (xx^{\top})}{\partial x} = 2x\mathrm{d}x^{\top} \quad \text{(using tl
   product rule)}
   \]
   Dimension: \( n \times n \).

% Large Heading for Question No 5.8
\begin{center}
    {\Large QUESTION NO 5.8}\\
  \end{center}

% Part a
\section*{Part a}
Compute the derivative \( \frac{df}{dx} \) using the chain rule.

The function is given as:
\[
f(z) = \exp\left(-\frac{1}{2} z\right), \quad z = g(y) = S^{-1} y, \quad y = h(x) = x - \mu
\]

\subsection*{Step 1: Understanding the Components}
\begin{itemize}
    \item \( x, \mu \in \mathbb{R}^D \)
    \item \( S \in \mathbb{R}^{D \times D} \)
    \item \( y = h(x) \): dimension \( D \)
    \item \( z = g(y) \): dimension \( D \)
    \item \( f(z) \): dimension \( \mathbb{R} \)
\end{itemize}

\subsection*{Step 2: Applying the Chain Rule}
\[
\frac{df}{dx} = \frac{df}{dz} \cdot \frac{dz}{dy} \cdot \frac{dy}{dx}
\]

```

```latex
\subsection*{Step 3: Compute Each Partial Derivative}
1. \( \frac{df}{dz} = -\frac{1}{2} \exp\left(-\frac{1}{2} z\right) \)
   Dimension: \( \mathbb{R} \)

2. \( \frac{dz}{dy} = S^{-1} \)
   Dimension: \( \mathbb{R}^{D \times D} \)

3. \( \frac{dy}{dx} = I \)
   Dimension: \( \mathbb{R}^{D \times D} \)

\subsection*{Step 4: Final Derivative}
\[
\frac{df}{dx} = \frac{1}{2} f(z) S^{-1}
\]

% Part b
\section*{Part b}
Compute the derivative \( \frac{df}{dx} \) for
\[
f(x) = \text{tr}(xx^{\top} + \sigma^2 I), \quad x \in \mathbb{R}^D
\]

\subsection*{Step 1: Understanding the Components}
\begin{itemize}
    \item \( xx^{\top} \): matrix dimension \( D \times D \)
    \item \( I \): identity matrix, dimension \( D \times D \)
    \item \( f(x) \) has dimension \( \mathbb{R} \).
\end{itemize}

\subsection*{Step 2: Applying the Derivative}
\[
\frac{df}{dx} = \frac{d}{dx} \text{tr}(xx^{\top}) = 2x^{\top}
\]
Dimension: \( \mathbb{R}^D \)

% Part c
\section*{Part c}
Compute the derivative \( \frac{df}{dx} \) using the chain rule for
\[
f = \tanh(z), \quad z = Ax + b, \quad x \in \mathbb{R}^N, \quad A \in \mathbb{R}^{M \times N},
\quad b \in \mathbb{R}^M
\]

\subsection*{Step 1: Understanding the Components}
\begin{itemize}
    \item \( x \): dimension \( N \)
    \item \( z \): dimension \( M \)
    \item \( f \): dimension \( M \)
\end{itemize}
```

```
\subsection*{Step 2: Applying the Chain Rule}
\[
\frac{df}{dx} = \frac{df}{dz} \cdot \frac{dz}{dx}
\]

\subsection*{Step 3: Compute Each Partial Derivative}
1. \( \frac{df}{dz} = \text{diag}(1 - \tanh^2(z)) \)
   Dimension: \( M \times M \)

2. \( \frac{dz}{dx} = A \)
   Dimension: \( M \times N \)

\subsection*{Step 4: Final Derivative}
\[
\frac{df}{dx} = \text{diag}(1 - \tanh^2(z)) A
\]
Dimension: \( M \times N \)

\end{document}
```

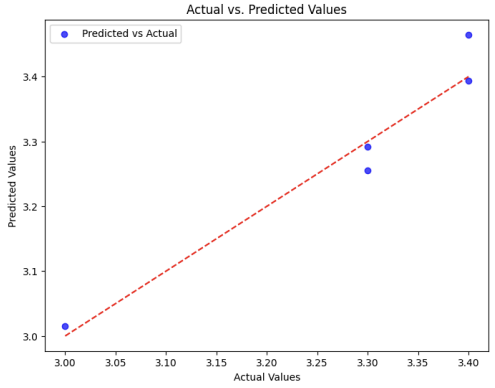
QUESTION 3-

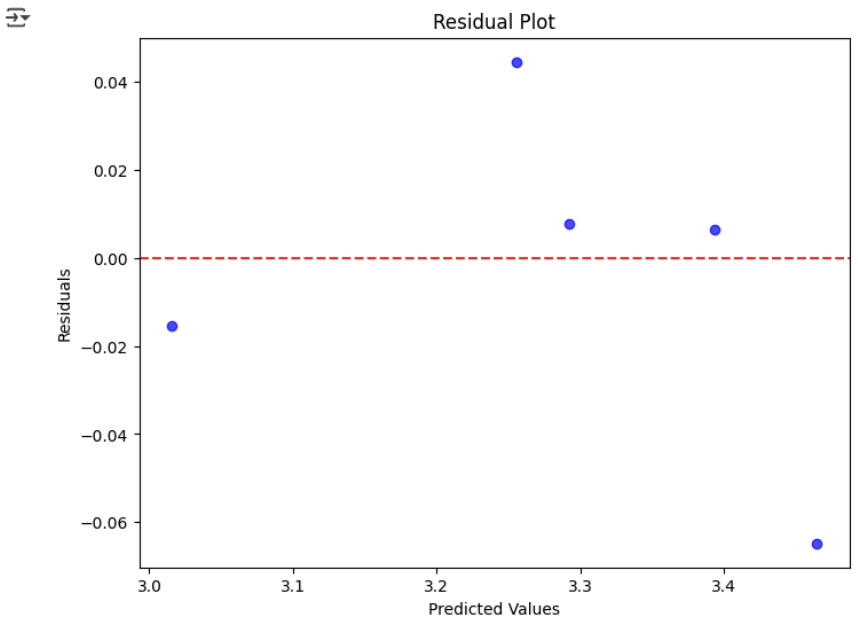
CODE PART-

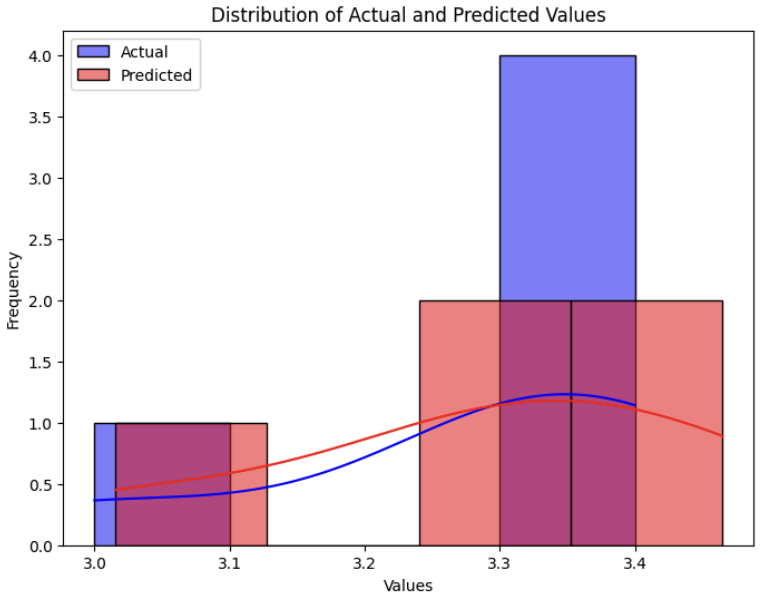
```python
# Install necessary libraries (if not installed)
# !pip install openpyxl missingno

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import missingno as msno
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer

# Load the dataset (Replace with actual file path)
df = pd.read_excel("DSAICourseInterestRelevanceSurvey.xlsx")  # Adjust the file path as needed

# Convert to numeric, handling errors
df = df.apply(pd.to_numeric, errors='coerce')

# Function to introduce missing values
def introduce_missing_values(df, missing_percentage=20):
    df_missing = df.copy()
    total_values = df_missing.size
    missing_count = int((missing_percentage / 100) * total_values)

    # Randomly select indices to set as NaN
    np.random.seed(42)
    missing_indices = np.random.choice(total_values, missing_count, replace=False)

    # Convert to 2D indices
    row_indices, col_indices = np.unravel_index(missing_indices, df_missing.shape)
    df_missing.values[row_indices, col_indices] = np.nan

    return df_missing

# Introduce 20% missing values
df_missing = introduce_missing_values(df, 20)

# Splitting data into known and unknown (missing) values
known_data = df_missing[df_missing.notna().any(axis=1)]
missing_data = df_missing[df_missing.isnull().any(axis=1)]

# Preparing features (X) and target (y) for model training
X_known = known_data.dropna(axis=1, how='all')  # Drop completely empty columns
y_known = X_known.mean(axis=1)  # Target variable: row mean (as an approximation)

# Splitting into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_known, y_known, test_size=0.2, random_state=42)

# Impute missing values in X_train and X_test using the mean
imputer = SimpleImputer(strategy='mean')  # Replace 'mean' with other strategies if needed
X_train = imputer.fit_transform(X_train)
X_test = imputer.transform(X_test)

# Train Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Predict missing values
X_missing = missing_data.dropna(axis=1, how='all')  # Drop empty columns
predicted_values = model.predict(X_missing.fillna(X_train.mean()))  # Fill missing with mean

# Replace missing values with predictions using .fillna()
df_filled = df_missing.fillna(pd.Series(predicted_values, index=df_missing.index[df_missing.isnull().any(axis=1)]))

# Evaluate using MSE
mse = mean_squared_error(y_test, model.predict(X_test))
print(f"Mean Squared Error (MSE): {mse:.4f}")

# 1. Actual vs Predicted values
plt.figure(figsize=(8, 6))
plt.scatter(y_test, model.predict(X_test), alpha=0.7, color="blue", label="Predicted vs Actual")
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color="red", linestyle="dashed")  # Reference line
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Actual vs. Predicted Values")
plt.legend()
plt.show()
```

```python
# 2. Residual plot
residuals = y_test - model.predict(X_test)
plt.figure(figsize=(8, 6))
plt.scatter(model.predict(X_test), residuals, alpha=0.7, color="blue")
plt.axhline(y=0, color="red", linestyle="dashed")  # Reference line at zero
plt.xlabel("Predicted Values")
plt.ylabel("Residuals")
plt.title("Residual Plot")
plt.show()

# 3. Distribution of actual and predicted values
plt.figure(figsize=(8, 6))
sns.histplot(y_test, label="Actual", color="blue", kde=True)
sns.histplot(model.predict(X_test), label="Predicted", color="red", kde=True)
plt.xlabel("Values")
plt.ylabel("Frequency")
plt.title("Distribution of Actual and Predicted Values")
plt.legend()
plt.show()

print(df_missing.isnull().sum().sum())  # Total number of missing values
print(df_missing.isnull().sum())  # Missing values per column

#Missing data matrix
plt.figure(figsize=(18, 8))  # Wider figure
msno.matrix(df_missing, sparkline=False, fontsize=12)
plt.title("Missing Data Matrix ", fontsize=14)
plt.show()
```

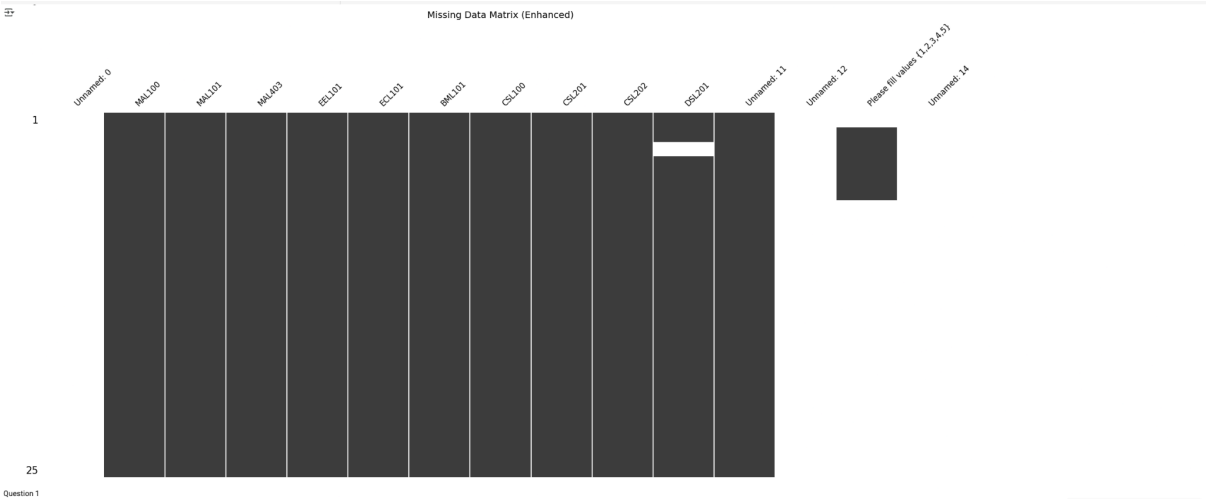
```
/usr/local/lib/python3.11/dist-packages/sklearn/utils/validation.py:2732: UserWarning: X has feature names, but LinearRegression was fitted without feature names
  warnings.warn(
Mean Squared Error (MSE): 0.0013
```




Residual Plot
0.04
0.02
0.00
−0.02
−0.04
−0.06
Residuals
3.0
3.1
3.2
3.3
3.4
Predicted Values




```
96
Unnamed: 0                          25
MAL100                               0
MAL101                               0
MAL403                               0
EEL101                               0
ECL101                               0
BML101                               0
CSL100                               0
CSL201                               0
CSL202                               0
DSL201                               1
Unnamed: 11                          0
Unnamed: 12                         25
Please fill values {1,2,3,4,5}      20
Unnamed: 14                         25
dtype: int64
<Figure size 1800x800 with 0 Axes>
```



LATEX CODE-

```latex
\documentclass{article}
\usepackage{graphicx}
\usepackage{amsmath}
\usepackage{minted}  % For Python code highlighting
\usepackage{xcolor}
\usepackage{hyperref}
\usepackage{booktabs}

\title{Data Analysis and Missing Value Imputation}
\author{ROHIT RAGHUWANSHI}
\date{\today}

\begin{document}

\maketitle

\section{Introduction}
This document presents a Python script for handling missing values, training a Linear Regression model, and evaluating predictions using visualizations.

\section{Python Code}

The following Python script performs the following tasks:
\begin{itemize}
    \item Reads the dataset and converts data to numeric format.
    \item Introduces missing values randomly (20\% of the dataset).
    \item Splits data into known and missing parts.
    \item Trains a Linear Regression model to predict missing values.
    \item Evaluates model performance using Mean Squared Error (MSE).
    \item Visualizes results with scatter plots, histograms, and a missing data matrix.
\end{itemize}

\begin{minted}[fontsize=\footnotesize, breaklines, bgcolor=lightgray]{python}
# Install necessary libraries (if not installed)
# !pip install openpyxl missingno

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import missingno as msno
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer

# Load the dataset (Replace with actual file path)
df = pd.read_excel("DSAICourseInterestRelevanceSurvey.xlsx")  # Adjust the file path as needed

# Convert to numeric, handling errors
df = df.apply(pd.to_numeric, errors='coerce')

# Function to introduce missing values
def introduce_missing_values(df, missing_percentage=20):
    df_missing = df.copy()
    total_values = df_missing.size
    missing_count = int((missing_percentage / 100) * total_values)

    np.random.seed(42)
    missing_indices = np.random.choice(total_values, missing_count, replace=False)

```

```
    row_indices, col_indices = np.unravel_index(missing_indices, df_missing.shape)
    df_missing.values[row_indices, col_indices] = np.nan

    return df_missing

# Introduce 20% missing values
df_missing = introduce_missing_values(df, 20)

# Splitting data into known and unknown (missing) values
known_data = df_missing[df_missing.notna().any(axis=1)]
missing_data = df_missing[df_missing.isnull().any(axis=1)]

# Preparing features (X) and target (y) for model training
X_known = known_data.dropna(axis=1, how='all')  # Drop completely empty columns
y_known = X_known.mean(axis=1)  # Target variable: row mean (as an approximation)

# Splitting into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_known, y_known, test_size=0.2, random_state=42)

# Impute missing values using the mean
imputer = SimpleImputer(strategy='mean')
X_train = imputer.fit_transform(X_train)
X_test = imputer.transform(X_test)

# Train Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Predict missing values
X_missing = missing_data.dropna(axis=1, how='all')
predicted_values = model.predict(X_missing.fillna(X_train.mean()))

# Replace missing values with predictions
df_filled = df_missing.fillna(pd.Series(predicted_values, index=df_missing.index[df_missing.isnull().any(axis=1)]))

# Evaluate using MSE
mse = mean_squared_error(y_test, model.predict(X_test))
print(f"Mean Squared Error (MSE): {mse:.4f}")

# 1. Actual vs Predicted values
plt.figure(figsize=(8, 6))
plt.scatter(y_test, model.predict(X_test), alpha=0.7, color="blue", label="Predicted vs Actual")
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], color="red", linestyle="dashed")
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Actual vs. Predicted Values")
plt.legend()
plt.show()

# 2. Residual plot
residuals = y_test - model.predict(X_test)
plt.figure(figsize=(8, 6))
plt.scatter(model.predict(X_test), residuals, alpha=0.7, color="blue")
plt.axhline(y=0, color="red", linestyle="dashed")
plt.xlabel("Predicted Values")
plt.ylabel("Residuals")
plt.title("Residual Plot")
plt.show()

```

```
# 3. Distribution of actual and predicted values
plt.figure(figsize=(8, 6))
sns.histplot(y_test, label="Actual", color="blue", kde=True)
sns.histplot(model.predict(X_test), label="Predicted", color="red", kde=True)
plt.xlabel("Values")
plt.ylabel("Frequency")
plt.title("Distribution of Actual and Predicted Values")
plt.legend()
plt.show()

print(df_missing.isnull().sum().sum())  # Total number of missing values
print(df_missing.isnull().sum())  # Missing values per column

# Missing data matrix
plt.figure(figsize=(18, 8))
msno.matrix(df_missing, sparkline=False, fontsize=12)
plt.title("Missing Data Matrix", fontsize=14)
plt.show()
\end{minted}

\section{Results and Discussion}
\subsection{Mean Squared Error}
The performance of the model is evaluated using the Mean Squared Error (MSE), which measures the difference between actual and predicted values.

\begin{equation}
    MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2
\end{equation}

The computed MSE in this experiment is displayed in the output.

\subsection{Visualizations}
\begin{itemize}
    \item \textbf{Actual vs. Predicted Scatter Plot:} Shows the correlation between actual and predicted values.
    \item \textbf{Residual Plot:} Highlights the errors in predictions.
    \item \textbf{Distribution Plot:} Compares the distributions of actual and predicted values.
    \item \textbf{Missing Data Matrix:} Provides an overview of missing values.
\end{itemize}

\section{Conclusion}
This script effectively handles missing data by predicting values using a trained Linear Regression model. The results demonstrate the effectiveness of using statistical imputation and predictive modeling to fill missing values.

\end{document}
```

Data Analysis and Missing Value Imputation

ROHIT RAGHUWANSHI

February 5, 2025

1 Introduction

This document presents a Python script for handling missing values, training a Linear Regression model, and evaluating predictions using visualizations.

2 Python Code

The following Python script performs the following tasks:

- Reads the dataset and converts data to numeric format.
- Introduces missing values randomly (20% of the dataset).
- Splits data into known and missing parts.
- Trains a Linear Regression model to predict missing values.
- Evaluates model performance using Mean Squared Error (MSE).
- Visualizes results with scatter plots, histograms, and a missing data matrix.

```
# Install necessary libraries (if not installed)
# !pip install openpyxl missingno

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import missingno as msno
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer

# Load the dataset (Replace with actual file path)
df = pd.read_excel("DSAICourseInterestRelevanceSurvey.xlsx")  # Adjust the file
↪  path as needed
```

```python
# Convert to numeric, handling errors
df = df.apply(pd.to_numeric, errors='coerce')

# Function to introduce missing values
def introduce_missing_values(df, missing_percentage=20):
    df_missing = df.copy()
    total_values = df_missing.size
    missing_count = int((missing_percentage / 100) * total_values)

    np.random.seed(42)
    missing_indices = np.random.choice(total_values, missing_count,
    ↪  replace=False)

    row_indices, col_indices = np.unravel_index(missing_indices,
    ↪  df_missing.shape)
    df_missing.values[row_indices, col_indices] = np.nan

    return df_missing

# Introduce 20% missing values
df_missing = introduce_missing_values(df, 20)

# Splitting data into known and unknown (missing) values
known_data = df_missing[df_missing.notna().any(axis=1)]
missing_data = df_missing[df_missing.isnull().any(axis=1)]

# Preparing features (X) and target (y) for model training
X_known = known_data.dropna(axis=1, how='all')  # Drop completely empty columns
y_known = X_known.mean(axis=1)  # Target variable: row mean (as an
↪  approximation)

# Splitting into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X_known, y_known,
↪  test_size=0.2, random_state=42)

# Impute missing values using the mean
imputer = SimpleImputer(strategy='mean')
X_train = imputer.fit_transform(X_train)
X_test = imputer.transform(X_test)

# Train Linear Regression model
model = LinearRegression()
model.fit(X_train, y_train)

# Predict missing values
X_missing = missing_data.dropna(axis=1, how='all')
predicted_values = model.predict(X_missing.fillna(X_train.mean()))

# Replace missing values with predictions
df_filled = df_missing.fillna(pd.Series(predicted_values,
↪  index=df_missing.index[df_missing.isnull().any(axis=1)]))

# Evaluate using MSE
mse = mean_squared_error(y_test, model.predict(X_test))
print(f"Mean Squared Error (MSE): {mse:.4f}")

# 1. Actual vs Predicted values
```

```
plt.figure(figsize=(8, 6))
plt.scatter(y_test, model.predict(X_test), alpha=0.7, color="blue",
↪  label="Predicted vs Actual")
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()],
↪  color="red", linestyle="dashed")
plt.xlabel("Actual Values")
plt.ylabel("Predicted Values")
plt.title("Actual vs. Predicted Values")
plt.legend()
plt.show()

# 2. Residual plot
residuals = y_test - model.predict(X_test)
plt.figure(figsize=(8, 6))
plt.scatter(model.predict(X_test), residuals, alpha=0.7, color="blue")
plt.axhline(y=0, color="red", linestyle="dashed")
plt.xlabel("Predicted Values")
plt.ylabel("Residuals")
plt.title("Residual Plot")
plt.show()

# 3. Distribution of actual and predicted values
plt.figure(figsize=(8, 6))
sns.histplot(y_test, label="Actual", color="blue", kde=True)
sns.histplot(model.predict(X_test), label="Predicted", color="red", kde=True)
plt.xlabel("Values")
plt.ylabel("Frequency")
plt.title("Distribution of Actual and Predicted Values")
plt.legend()
plt.show()

print(df_missing.isnull().sum().sum())  # Total number of missing values
print(df_missing.isnull().sum())  # Missing values per column

# Missing data matrix
plt.figure(figsize=(18, 8))
msno.matrix(df_missing, sparkline=False, fontsize=12)
plt.title("Missing Data Matrix", fontsize=14)
plt.show()
```

3 Results and Discussion

3.1 Mean Squared Error

The performance of the model is evaluated using the Mean Squared Error (MSE), which measures the difference between actual and predicted values.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{1}$$

The computed MSE in this experiment is displayed in the output.

3.2 Visualizations

- **Actual vs. Predicted Scatter Plot:** Shows the correlation between actual and predicted values.
- **Residual Plot:** Highlights the errors in predictions.
- **Distribution Plot:** Compares the distributions of actual and predicted values.
- **Missing Data Matrix:** Provides an overview of missing values.

4 Conclusion

This script effectively handles missing data by predicting values using a trained Linear Regression model. The results demonstrate the effectiveness of using statistical imputation and predictive modeling to fill missing values.

Key Observations from the Code

1. Handling Missing Data
    - The script introduces 20% missing values randomly into the dataset.
    - Uses `missingno` to visualize missing data.
    - Splits data into known (non-missing) and unknown (missing) values.
2. Data Preprocessing
    - Converts all data to numeric format, handling errors gracefully.
    - Drops completely empty columns before training.
    - Uses mean imputation to fill missing values before model training.
3. Model Training
    - Uses Linear Regression to predict missing values based on available data.
    - Splits the dataset into training (80%) and testing (20%) for validation.
    - Evaluates model performance using Mean Squared Error (MSE).
4. Predictions & Replacement
    - Predicts missing values and replaces them in the dataset.
    - Uses the mean of training data to fill missing inputs before prediction.
5. Performance Evaluation & Visualization

    - Scatter Plot compares actual vs. predicted values.
    - Residual Plot shows the distribution of errors.
    - Histogram compares actual vs. predicted value distributions.
    - Missing Data Matrix provides an overview of missing values.
6. MSE Calculation
    - Computes Mean Squared Error (MSE) to evaluate model accuracy.