Question 1

```
import pandas as pd
data =
pd.read_csv('https://raw.githubusercontent.com/amaydixit11/Academics/
refs/heads/main/DSL251/HomeWork2/DIstanceTimeDataset%20-
%20StudentsHomeTownDistance.csv')
data.head()
```


```
{"summary":"{\n  \"name\": \"data\",\n  \"rows\": 50,\n  \"fields\":
[\n    {\n      \"column\": \"Location Name\",\n      \"properties\":
{\n        \"dtype\": \"string\",\n        \"num_unique_values\": 40,\
n        \"samples\": [\n          \"Ranchi\",\n          \"delhi\",\n
\"Pipariya(M.P)\"\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Time to Reach (hr)\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 7.690017839743818,\n        \"min\":
0.5,\n        \"max\": 48.0,\n        \"num_unique_values\": 23,\n
\"samples\": [\n          10.0,\n          35.0,\n          18.0\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"Distance (km)\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
368.37237449416835,\n        \"min\": 9.0,\n        \"max\": 1800.0,\n
\"num_unique_values\": 41,\n        \"samples\": [\n          949.0,\n
1743.0,\n          861.0\n        ],\n        \"semantic_type\":
\"\",\n        \"description\": \"\"\n      }\n    },\n    {\n
\"column\": \"Train Only\",\n      \"properties\": {\n
\"dtype\": \"number\",\n        \"std\": 0.504374939460682,\n
\"min\": 0.0,\n        \"max\": 1.0,\n        \"num_unique_values\":
2,\n        \"samples\": [\n          0.0,\n          1.0\n        ],\
n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    },\n    {\n      \"column\": \"Road Only\",\n
\"properties\": {\n        \"dtype\": \"number\",\n        \"std\":
0.2820566728469695,\n        \"min\": 0.0,\n        \"max\": 1.0,\n
\"num_unique_values\": 2,\n        \"samples\": [\n          1.0,\n
0.0\n        ],\n        \"semantic_type\": \"\",\n
\"description\": \"\"\n      }\n    },\n    {\n      \"column\":
\"Train+Road\",\n      \"properties\": {\n        \"dtype\":
\"number\",\n        \"std\": 0.5025375018797696,\n        \"min\":
0.0,\n        \"max\": 1.0,\n        \"num_unique_values\": 2,\n
\"samples\": [\n          1.0,\n          0.0\n        ],\n
\"semantic_type\": \"\",\n        \"description\": \"\"\n      }\
n    }\n  ]\n}","type":"dataframe","variable_name":"data"}
```


```
# null values
data.isnull().sum()
```

```
Location Name          2
Time to Reach (hr)     3
Distance (km)          3
Train Only             3
Road Only              3
Train+Road             3
dtype: int64
```

```
data.dropna(inplace=True)
data.isnull().sum()
```

```
Location Name          0
Time to Reach (hr)     0
Distance (km)          0
Train Only             0
Road Only              0
Train+Road             0
dtype: int64
```

```
data.describe()
```

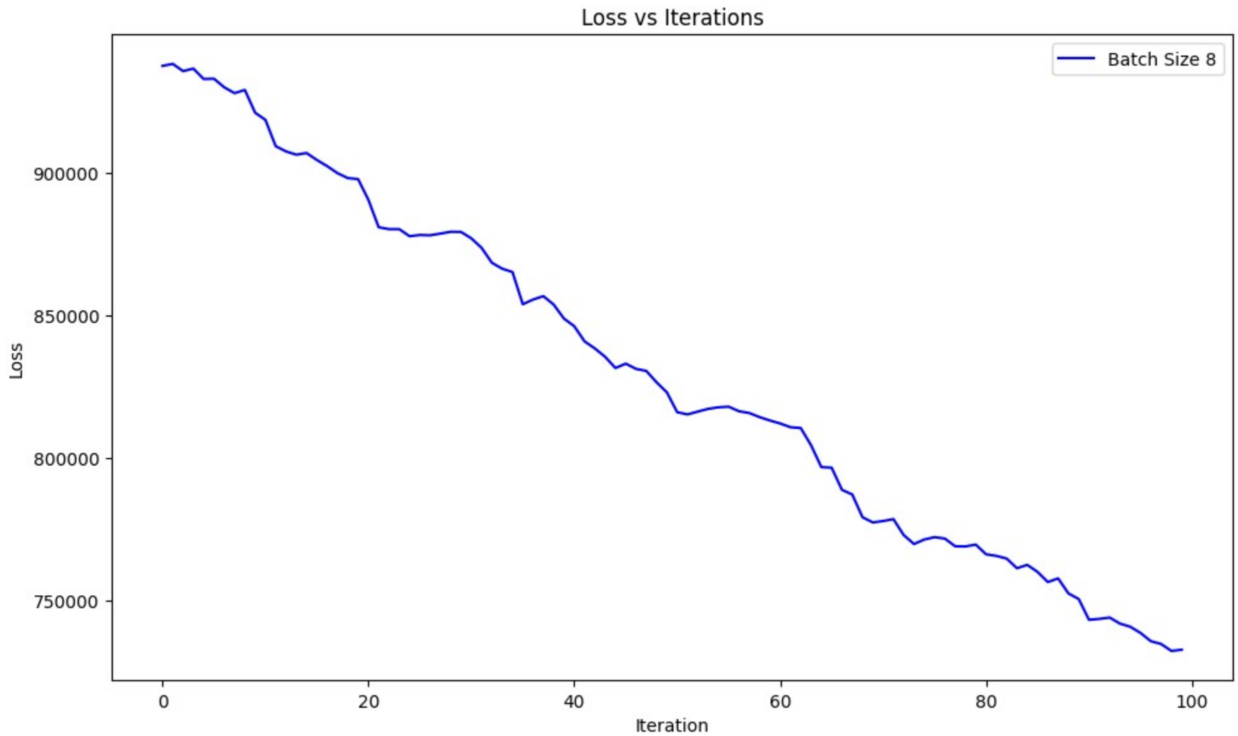

{"summary":"{\n \"name\": \"data\",\n \"rows\": 8,\n \"fields\": [\n   {\n   \"column\": \"Time to Reach (hr)\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 17.22022636218284,\n   \"min\": 0.5,\n   \"max\": 48.0,\n   \"num_unique_values\": 8,\n   \"samples\": [\n   17.7068085106383,\n   18.0,\n   47.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Distance (km)\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 592.391500893145,\n   \"min\": 9.0,\n   \"max\": 1800.0,\n   \"num_unique_values\": 8,\n   \"samples\": [\n   898.5744680851063,\n   900.0,\n   47.0\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Train Only\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 16.472130803126635,\n   \"min\": 0.0,\n   \"max\": 47.0,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.46808510638297873,\n   1.0,\n   0.504374939460682\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Road Only\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 16.55149536319266,\n   \"min\": 0.0,\n   \"max\": 47.0,\n   \"num_unique_values\": 5,\n   \"samples\": [\n   0.0851063829787234,\n   1.0,\n   0.2820566728469695\n   ],\n   \"semantic_type\": \"\",\n   \"description\": \"\"\n   }\n   },\n   {\n   \"column\": \"Train+Road\",\n   \"properties\": {\n   \"dtype\": \"number\",\n   \"std\": 16.473290225180502,\n   \"min\": 0.0,\n   \"max\": 47.0,\n   \"num_unique_values\": 5,\n   \"samples\": [\n

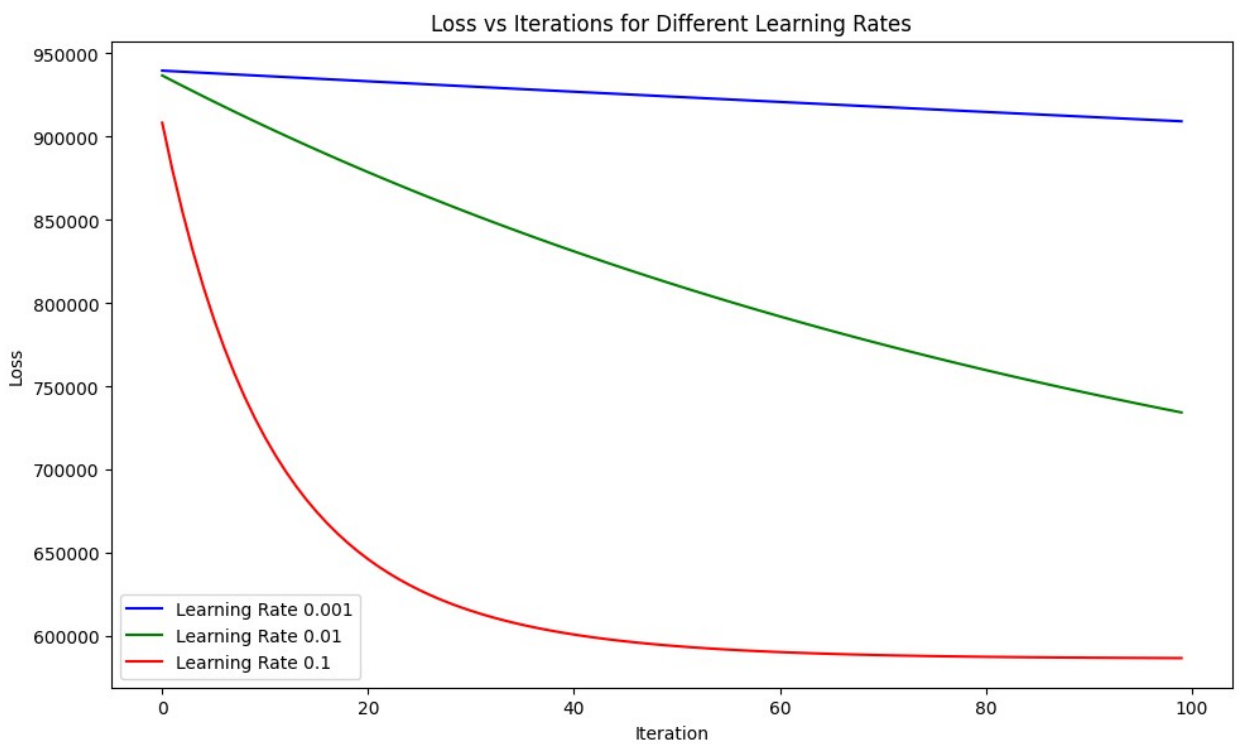
```
0.44680851063829785,\n          1.0,\n          0.5025375018797696\n
],\n        \"semantic_type\": \"\",\n        \"description\": \"\"\n
}\n    }\n  ]\n}","type":"dataframe"}
```


```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler

class GradientDescent:
    def __init__(self, learning_rate=0.01, n_iterations=100):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.loss_history = []
        self.gradient_norm_history = []

    def initialize_weights(self, n_features):
        # Initialize W as 0.35
        self.weights = np.full(n_features, 0.35)

    def compute_gradient(self, X, y, batch_indices=None):
        if batch_indices is None:
            batch_indices = np.arange(len(y))

        X_batch = X[batch_indices]
        y_batch = y[batch_indices]

        # Predictions (non-negative)
        y_pred = np.maximum(0, np.dot(X_batch, self.weights))

        # Compute gradients
        gradient = np.dot(X_batch.T, (y_pred - y_batch)) /
len(batch_indices)
        return gradient

    def compute_loss(self, X, y):
        y_pred = np.maximum(0, np.dot(X, self.weights))
        return np.mean((y_pred - y) ** 2)

    def train(self, X, y, batch_size=None):
        self.initialize_weights(X.shape[1])
        n_samples = len(y)

        for i in range(self.n_iterations):
            if batch_size:
                # Batch gradient descent
                batch_indices = np.random.choice(n_samples,
batch_size, replace=False)
                gradient = self.compute_gradient(X, y, batch_indices)
```

```
            else:
                # Full gradient descent
                gradient = self.compute_gradient(X, y)

            # Update weights
            self.weights -= self.learning_rate * gradient

            # Record loss and gradient norm
            self.loss_history.append(self.compute_loss(X, y))

self.gradient_norm_history.append(np.linalg.norm(gradient))

def plot_training_progress(X, y, batch_sizes=[8]):
    fig, axes = plt.subplots(2, 1, figsize=(10, 12))
    colors = ['b', 'g', 'r']

    # Train with specified batch size
    gd = GradientDescent(learning_rate=0.01, n_iterations=100)
    gd.train(X, y, batch_size=batch_sizes[0])

    # Plot loss
    axes[0].plot(gd.loss_history, color=colors[0], label=f'Batch Size
{batch_sizes[0]}')
    axes[0].set_title('Loss vs Iterations')
    axes[0].set_xlabel('Iteration')
    axes[0].set_ylabel('Loss')
    axes[0].legend()

    # Compare different learning rates
    learning_rates = [0.001, 0.01, 0.1]
    for lr, color in zip(learning_rates, colors):
        gd = GradientDescent(learning_rate=lr, n_iterations=100)
        gd.train(X, y)
        axes[1].plot(gd.loss_history, color=color, label=f'Learning
Rate {lr}')

    axes[1].set_title('Loss vs Iterations for Different Learning
Rates')
    axes[1].set_xlabel('Iteration')
    axes[1].set_ylabel('Loss')
    axes[1].legend()

    plt.tight_layout()
    return fig, gd

def main(data):
    # Prepare data
    df = data.copy()
    df['Time to Reach (hr)'] = pd.to_numeric(df['Time to Reach (hr)'],
errors='coerce')
```

```
    df['Distance (km)'] = pd.to_numeric(df['Distance (km)'],
errors='coerce')
    df = df.dropna()

    # Extract features and target
    X = df[['Time to Reach (hr)']].values
    y = df['Distance (km)'].values

    # Scale features
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    # Train and visualize
    fig, model = plot_training_progress(X_scaled, y)
    plt.show()

    # Print final weights and loss
    print(f"Final weights: {model.weights}")
    print(f"Final loss: {model.loss_history[-1]:.2f}")

    return model

# Example usage:
if __name__ == "__main__":
    model = main(data)
```






```
Final weights: [553.06107155]
Final loss: 586584.18
```

Question 3

```
import pandas as pd
import numpy as np

# Install openpyxl if not already installed
!pip install openpyxl

def xlsx_to_dataframe(file_path):
  try:
    df = pd.read_excel(file_path, engine='openpyxl')  # Use openpyxl
engine
    return df
  except FileNotFoundError:
    print(f"Error: File not found at '{file_path}'")
    return None
  except Exception as e:
    print(f"An error occurred: {e}")
    return None

# Example usage
# file_path =
'https://raw.githubusercontent.com/amaydixit11/Academics/refs/heads/
main/DSL251/HomeWork2/HW2Q3.csv'
file_path = '/content/DSAICourseInterestRelevanceSurvey.xlsx'
df = xlsx_to_dataframe(file_path)

if df is not None:

    try:
      cropped_df = df.iloc[0:24, 0:11] # Crop the DataFrame
      cropped_matrix = cropped_df.values # Convert to NumPy matrix
      print(cropped_matrix)

    except IndexError:
      print("Error: Cropping indices out of bounds. Check your
DataFrame dimensions.")
    except Exception as e:
      print(f"An error occurred during cropping or matrix conversion:
{e}")
```

```
Requirement already satisfied: openpyxl in
/usr/local/lib/python3.11/dist-packages (3.1.5)
Requirement already satisfied: et-xmlfile in
/usr/local/lib/python3.11/dist-packages (from openpyxl) (2.0.0)
[['Student 1' 4.0 3.0 4.0 1.0 1.0 1.0 4.0 5.0 5.0 5.0]
 ['Student 2' 3.0 3.0 3.0 1.0 1.0 1.0 4.0 5.0 5.0 5.0]
 ['Student 3' 4.0 4.0 3.0 3.0 4.0 2.0 4.0 5.0 4.0 nan]
 ['Student 4' 3.0 4.0 4.0 1.0 1.0 1.0 5.0 5.0 5.0 2.0]
 ['Student 5' 3.0 3.0 4.0 3.0 3.0 2.0 4.0 5.0 5.0 4.0]
```

```
 ['Student 6' 3.0 3.0 5.0 1.0 1.0 1.0 4.0 5.0 5.0 5.0]
 ['Student 7' 3.0 4.0 4.0 1.0 1.0 1.0 4.0 4.0 4.0 5.0]
 ['Student 8' 3.0 4.0 2.0 2.0 3.0 2.0 4.0 4.0 4.0 5.0]
 ['Student 9' 3.0 4.0 4.0 2.0 3.0 1.0 4.0 4.0 4.0 4.0]
 ['Student 10' 4.0 4.0 3.0 3.0 2.0 2.0 4.0 4.0 5.0 4.0]
 ['Student 11' 4.0 5.0 4.0 2.0 3.0 3.0 5.0 5.0 5.0 4.0]
 ['Student 12' 3.0 3.0 3.0 1.0 1.0 2.0 5.0 4.0 3.0 5.0]
 ['Student 13' 2.0 3.0 3.0 1.0 1.0 2.0 4.0 4.0 3.0 3.0]
 ['Student 14' 3.0 3.0 4.0 2.0 2.0 2.0 4.0 5.0 4.0 4.0]
 ['Student 15' 4.0 4.0 5.0 3.0 3.0 1.0 4.0 5.0 4.0 5.0]
 ['Student 16' 3.0 3.0 4.0 1.0 1.0 1.0 4.0 4.0 4.0 4.0]
 ['Student 17' 4.0 4.0 3.0 1.0 1.0 1.0 5.0 5.0 5.0 5.0]
 ['Student 18' 3.0 4.0 3.0 2.0 1.0 3.0 4.0 4.0 4.0 5.0]
 ['Student 19' 3.0 3.0 4.0 1.0 2.0 1.0 4.0 3.0 5.0 5.0]
 ['Student 20' 1.0 5.0 3.0 1.0 1.0 2.0 1.0 5.0 4.0 3.0]
 ['Student 21' 3.0 4.0 3.0 2.0 3.0 1.0 4.0 4.0 4.0 5.0]
 ['Student 22' 4.0 5.0 5.0 1.0 1.0 2.0 4.0 1.0 5.0 3.0]
 ['Student 23' 2.0 2.0 2.0 1.0 1.0 4.0 4.0 2.0 2.0 4.0]
 ['Student 24' 3.0 5.0 5.0 2.0 1.0 1.0 4.0 4.0 4.0 5.0]]
```

```python
#randomly suffle the student


if df is not None:
    try:
        # Crop the DataFrame (adjust indices as needed)
        cropped_df = df.iloc[0:24, 0:11]

        # Shuffle the DataFrame randomly
        shuffled_df = cropped_df.sample(frac=1).reset_index(drop=True)
        shuffled_df = shuffled_df.drop(columns=['Unnamed: 0'])
        # Display the shuffled DataFrame in tabular format
        print(shuffled_df)

    except IndexError:
        print("Error: Cropping indices out of bounds. Check your 
DataFrame dimensions.")
    except Exception as e:
        print(f"An error occurred during shuffling or display: {e}")
```

```
    MAL100  MAL101  MAL403  EEL101  ECL101  BML101  CSL100  CSL201  
CSL202  \
0      4.0     4.0     3.0     3.0     4.0     2.0     4.0     5.0   
4.0   
1      3.0     3.0     4.0     1.0     1.0     1.0     4.0     4.0   
4.0   
2      2.0     3.0     3.0     1.0     1.0     2.0     4.0     4.0   
3.0   
3      3.0     4.0     4.0     1.0     1.0     1.0     5.0     5.0   
5.0   
```

4	4.0	3.0	4.0	1.0	1.0	1.0	4.0	5.0	5.0
5	3.0	3.0	3.0	1.0	1.0	1.0	4.0	5.0	5.0
6	4.0	5.0	4.0	2.0	3.0	3.0	5.0	5.0	5.0
7	3.0	3.0	4.0	3.0	3.0	2.0	4.0	5.0	5.0
8	3.0	4.0	4.0	1.0	1.0	1.0	4.0	4.0	4.0
9	3.0	4.0	3.0	2.0	3.0	1.0	4.0	4.0	4.0
10	3.0	3.0	4.0	2.0	2.0	2.0	4.0	5.0	4.0
11	3.0	3.0	3.0	1.0	1.0	2.0	5.0	4.0	3.0
12	3.0	4.0	4.0	2.0	3.0	1.0	4.0	4.0	4.0
13	2.0	2.0	2.0	1.0	1.0	4.0	4.0	2.0	2.0
14	4.0	4.0	3.0	3.0	2.0	2.0	4.0	4.0	5.0
15	3.0	4.0	2.0	2.0	3.0	2.0	4.0	4.0	4.0
16	4.0	5.0	5.0	1.0	1.0	2.0	4.0	1.0	5.0
17	3.0	5.0	5.0	2.0	1.0	1.0	4.0	4.0	4.0
18	3.0	3.0	5.0	1.0	1.0	1.0	4.0	5.0	5.0
19	3.0	4.0	3.0	2.0	1.0	3.0	4.0	4.0	4.0
20	1.0	5.0	3.0	1.0	1.0	2.0	1.0	5.0	4.0
21	4.0	4.0	3.0	1.0	1.0	1.0	5.0	5.0	5.0
22	3.0	3.0	4.0	1.0	2.0	1.0	4.0	3.0	5.0
23	4.0	4.0	5.0	3.0	3.0	1.0	4.0	5.0	4.0

	DSL201
0	NaN
1	4.0
2	3.0
3	2.0
4	5.0
5	5.0
6	4.0

```
7       4.0
8       5.0
9       5.0
10      4.0
11      5.0
12      4.0
13      4.0
14      4.0
15      5.0
16      3.0
17      5.0
18      5.0
19      5.0
20      3.0
21      5.0
22      5.0
23      5.0

def replace_data(df, percentage):
    try:
        # Exclude first row and first column
        df_to_modify = df.iloc[:, 1:]

        total_cells = df_to_modify.size
        num_cells_to_replace = int(total_cells * (percentage / 100))

        # Generate random row and column indices for the modified
DataFrame
        row_indices = np.random.choice(df_to_modify.index,
size=num_cells_to_replace, replace=True)
        col_indices = np.random.choice(df_to_modify.columns,
size=num_cells_to_replace, replace=True)

        # Replace values in the original DataFrame using the modified
indices
        for row, col in zip(row_indices, col_indices):
              df.loc[row, col] = np.nan
        return df
    except (ValueError, TypeError):
        print("Error: Invalid input. Percentage must be a number
between 0 and 100.")
        return df  # Return original DataFrame on error
    except Exception as e:
        print(f"An error occurred: {e}")
        return df
```

Problem Statement: Missing Data Prediction and Analysis

You are provided with a dataset (DSAICourseInterestRelevanceSurvey.xlsx) containing survey responses about student interest and courses. Your task is to simulate the scenario of missing data and build a model to predict those missing values.

Tasks:

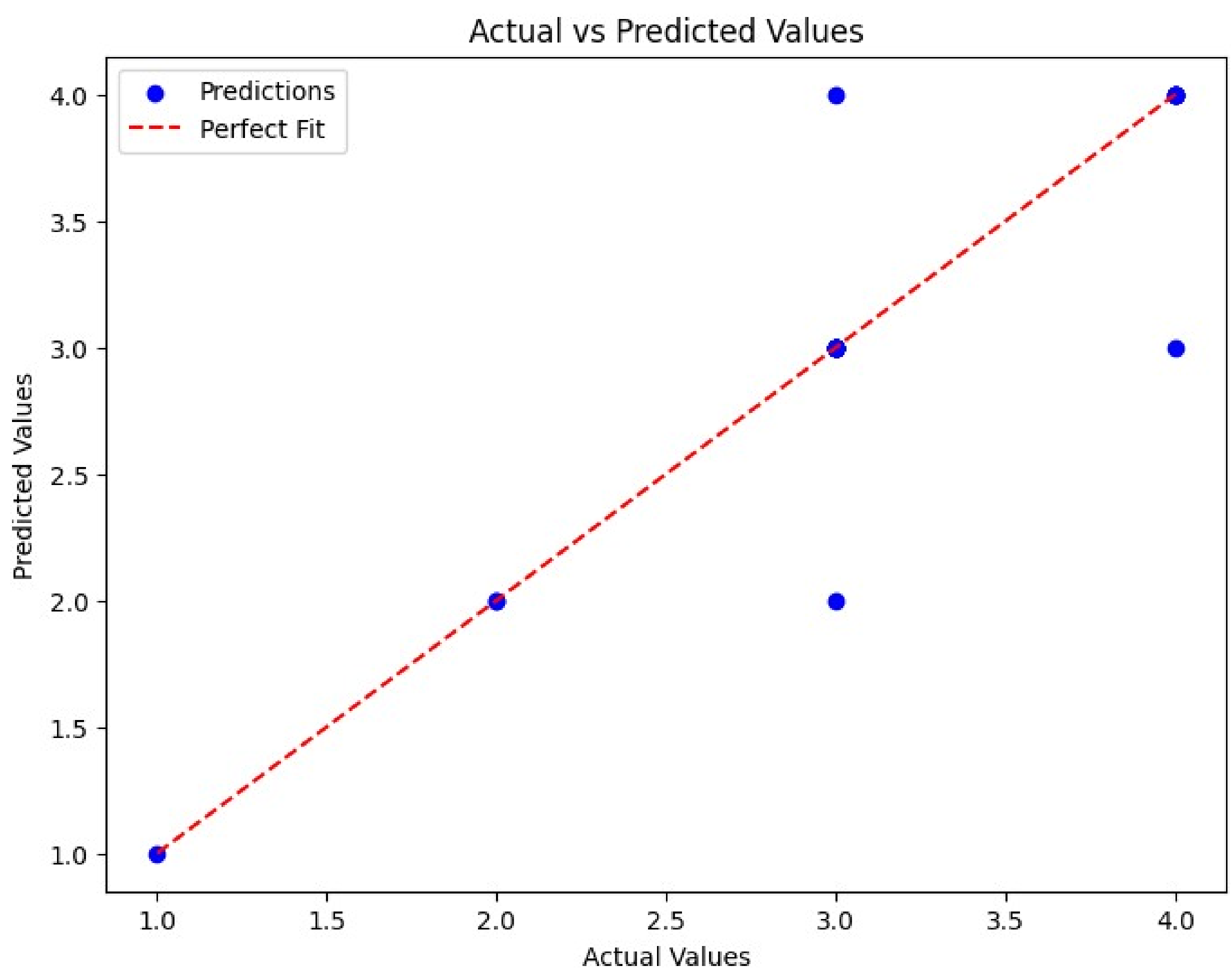
Data Preprocessing: Experiment with missing data percentages ranging from 20% to 80%.

Model Building: Write a algorithms such that it predicts the missing values(e.g., Linear Regression).

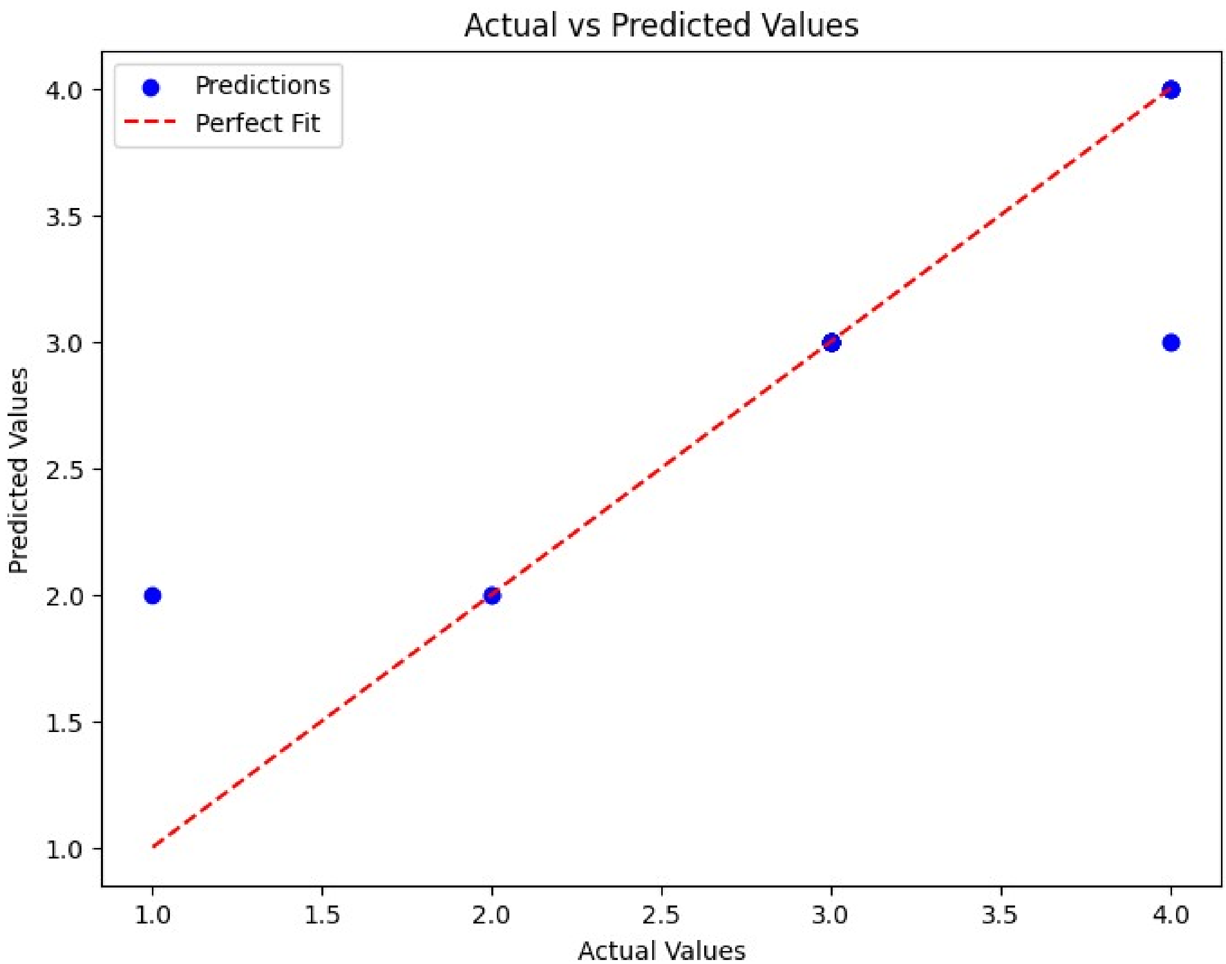
Evaluation and Visualization: Calculate the Mean Squared Error (MSE) to evaluate the accuracy of your predictions. The MSE measures the average squared difference between the actual and predicted values. Create a plot to visualize the relationship between the actual and predicted values.

```
import pandas as pd
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
from sklearn.impute import SimpleImputer
from sklearn.impute import KNNImputer



df = shuffled_df.copy()
imputer = KNNImputer(n_neighbors=5)
df = imputer.fit_transform(df).round()
# print(df)
for percentage in [20, 40, 60, 80]:
    modified_df = replace_data(shuffled_df, percentage)

    # print(modified_df)
    # Step 3: imputation
    # imputer = SimpleImputer(strategy='mean')
    # modified_df = imputer.fit_transform(modified_df)
    imputer = KNNImputer(n_neighbors=5)
    modified_df = imputer.fit_transform(shuffled_df)

    # Step 4: Split data into features (X) and target (y) for
prediction
    X = modified_df[:, 1:]
    y = df[:, 0]

    # Step 5: Train the Linear Regression Model
    model = LinearRegression()
    model.fit(X, y)
```

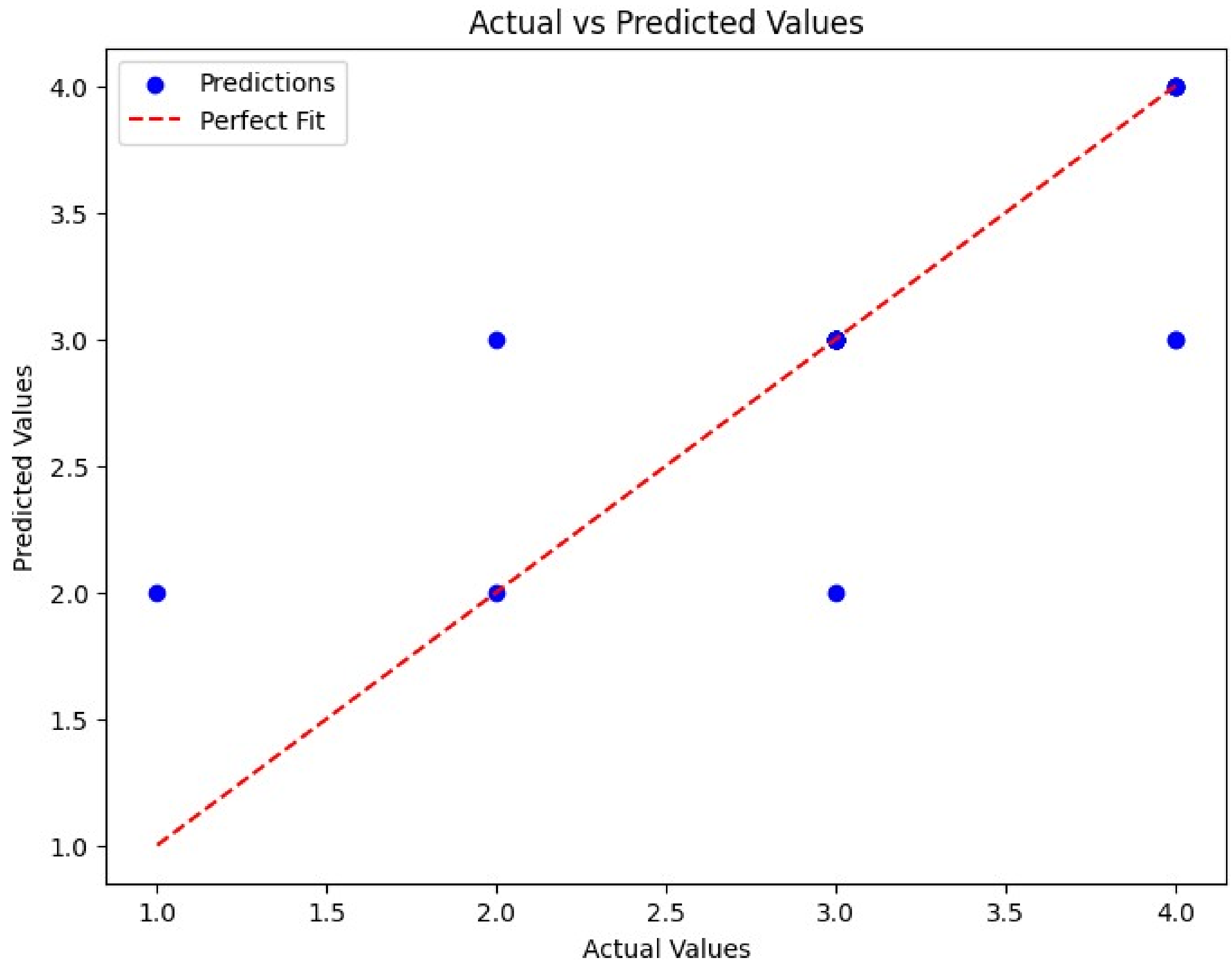
```
    predictions = model.predict(X).round()

    # Step 7: Calculate Mean Squared Error (MSE)
    mse = mean_squared_error(y, predictions)
    print(f'Mean Squared Error: {mse}')

    # Step 8: Plot the actual vs predicted values
    plt.figure(figsize=(8, 6))
    plt.scatter(y, predictions, color='blue', label='Predictions')
    plt.plot([y.min(), y.max()], [y.min(), y.max()], color='red',
linestyle='--', label='Perfect Fit')
    plt.xlabel('Actual Values')
    plt.ylabel('Predicted Values')
    plt.title('Actual vs Predicted Values')
    plt.legend()
    plt.show()

Mean Squared Error: 0.125
```



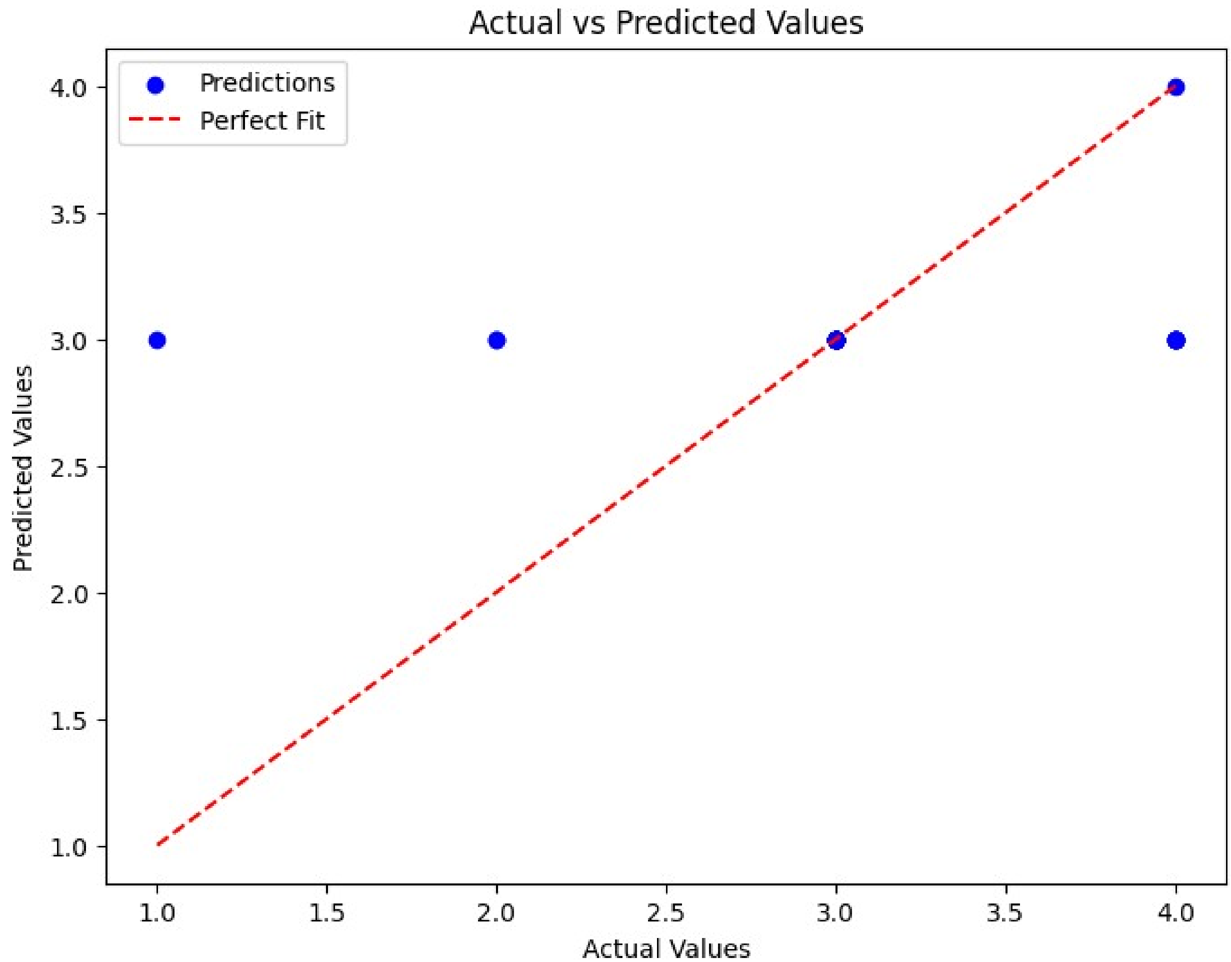
```
Mean Squared Error: 0.125
```




```
Mean Squared Error: 0.20833333333333334
```




```
Mean Squared Error: 0.5
```


Actual vs Predicted Values
Predictions
Perfect Fit
4.0
3.5
3.0
2.5
2.0
1.5
1.0
Predicted Values
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Actual Values