Assignment-1: Multivariate Linear Regression using Gradient Descent Algorithm

Sajan Kumer Sarker
Id: 2111131642
Email: sajan.sarker@northsouth.edu

Rafsan Jani Chowdhury
Id: 2011424642
Email: rafsan.chowdhury@northsouth.edu

Rosely Mohammad
Id: 2014219642
Email: rosely.mohammad@northsouth.edu

December 7, 2024

1. Introduction

In this assignment report, we will discuss about the implementation process of multivariate linear regression using the gradient descent algorithm. Air Quality dataset [1] was used to text the algorithm. It also involves data prepossessing, splitting it into training and testing datasets, feature selection, feature scaling, and finally, training the model using the implemented gradient descent algorithm. Then the model was tested on the testing dataset to evaluate its performance. The main objective is to predict air quality datasets parameters based on various other parameters. We train and test the model three times, each time focusing on predicting a different variable: first Temperature, then Relative Humidity (RH), and finally Absolute Humidity (AH).

2. Methods

Necessary Imports:

```
import numpy as np
import pandas as pd
import math, copy
import matplotlib.pyplot as plt
```

We have imported all necessary libraries and modules to be used in each notebook.

Dataset Loading:

```
from ucimlrepo import fetch_ucirepo

# fetch dataset
air_quality = fetch_ucirepo(id=360)

# data (as pandas dataframes)
air_quality_data = air_quality.data.features

print(air_quality_data)
```

Import the dataset and print a few of the first and last samples from it.

2.1 Dataset Pre-processing Functions:

Calculate Missing Values:

```
# Missing values are tagged with -200 value.
def find_missing_values(data):
  for column in data.columns:
    count = 0
    count = (data[column] == -200).sum()
    print(f"{column:13} has {count:5} occurrences.")
```

Defines a function that checks each column in a dataset for missing values tagged as -200 and prints the count of occurrences for each column.

Replace the Missing Value:

```
# replacing -200 with mean of each features.
def replace_with_mean(data):
  for column in data.columns:
    if pd.api.types.is_numeric_dtype(data[column]):
      mean_value = data[data[column] != -200][column].mean()
      data[column] = data[column].replace(-200, mean_value)
      # print(f"Mean of {column:13} is: {mean_value:8.4f}")
  return pd.DataFrame(data)
```

Defines a function that replaces missing values tagged as -200 in numeric columns of a dataset with the mean of that column, excluding the -200 values.

Remove Features:

```
# Feature remove function
def remove_feature(X, features_names):
  return X.drop(columns=features_names)
```

Defines a function that removes specified features (columns) from the dataset X.

Feature Scaling (Training Data):

```
# Mean Normalization functions for training and test dataset
tr_mean = []  # global mean
tr_std = []   # global standard deviation

def zscore_normalize_train(X):    # normalize train data only
  m, n = X.shape
  norm_X = []

  for col in range(n):
    col_values = X.iloc[:, col]
    mean = sum(col_values) / m
    tr_mean.append(mean)

    sum_sq_diff = sum((val - mean)**2 for val in col_values)
    std = (sum_sq_diff / m)**0.5
    tr_std.append(std)

    norm_col = [(val - mean) / std for val in col_values]
    norm_X.append(norm_col)

  norm_X = [list(row) for row in zip(*norm_X)]
  norm_X = pd.DataFrame(norm_X, columns=X.columns)
  return norm_X
```

We defines a function that performs Z-score normalization on the training data by standardizing each feature (column) based on its mean and standard deviation. It returns the normalized dataset.

Feature Scaling (Test Data):

```
def zscore_normalize_test(X):    # normalize test data only
  m, n = X.shape
  norm_X = []

  for col in range(n):
    col_values = X.iloc[:, col]

    norm_col = [(val - tr_mean[col]) / tr_std[col] for val in col_values]
    norm_X.append(norm_col)

  norm_X = [list(row) for row in zip(*norm_X)]
  norm_X = pd.DataFrame(norm_X, columns=X.columns)
  return norm_X
```

This will performs Z-score normalization on the test data using the mean and standard deviation calculated from the training data, and returns the normalized test dataset.

2.2 Data Pre-processing:

Pre-process the Data for Model:

```
# getting more info about the data
print(air_quality_data.info())

# Checking for missing Values
print(find_missing_values(air_quality_data))

# Delete the date and time columns, and those columns with more than 1,000 missing values
feature_name = ['Date', 'Time', 'CO(GT)', 'NMHC(GT)', 'NOx(GT)', 'NO2(GT)']
air_quality_data = remove_feature(air_quality_data, feature_name)

# replace the missing values with mean of remaining columns
air_quality_data = replace_with_mean(air_quality_data)
print(air_quality_data.head())
```

Here we first prints the general information about the dataset and checks for missing values. It then removes the Date, Time, and columns with more than 1,000 missing values, followed by replacing the missing values in the remaining columns with the mean of each column. Finally, it displays the first few rows of the cleaned dataset.

Train Test Split:

```
# find index number of 75% data
train_data_index = int(len(air_quality_data) * 0.75)

train = air_quality_data[:train_data_index] # train data
test = air_quality_data[train_data_index:]  # test data

train_data = zscore_normalize_train(train)  # normalize train data
test_data = zscore_normalize_test(test)     # normalize test data


train_data.to_csv('./AirQualityTrainingData.csv', index=False)
test_data.to_csv('./AirQualityTestData.csv', index=False)

```

```
print(f"Shape of Whole Dataset: {air_quality_data.shape}")
print(f"Shape of Training Dataset: {train_data.shape}")
print(f"Shape of Test Dataset: {test_data.shape}")

train_data = pd.read_csv('./AirQualityTrainingData.csv').to_numpy()
test_data = pd.read_csv('./AirQualityTestData.csv').to_numpy()
```

We Splits the dataset into training and testing sets, with 75% of the data used for training. It then normalizes the training and test data using Z-score normalization. The normalized datasets are saved as CSV files, and their shapes are printed. Finally, the saved CSV files are read back into numpy arrays for further use.

Split Feature and Target and Plot the Data:

```
# Split features and target for train and test dataset
X_train = train_data[:, :8]
y_train = train_data[:, 8]
X_test = test_data[:, :8]
y_test = test_data[:, 8]

# visualize the train data
X_features = ['PT08.S1(CO)','C6H6(GT)','PT08.S2(NMHC)','PT08.S3(NOx)','PT08.S4(NO2)','PT08.S5(

fig, ax = plt.subplots(1, 8, figsize=(30, 10), sharey=True)
fig.suptitle("Training Data Visualization", fontsize=16)

for i in range(0, 8):
  ax[i].scatter(X_train[:, i], y_train, marker='.', c='b')
  ax[i].set_xlabel(X_features[i])

ax[0].set_ylabel('AH')
plt.tight_layout()
plt.show()
```

Splits the training and test data into features and targets. It then visualizes the relationship between each feature and the target using scatter plots for each feature in the training dataset. The target variable AH labeled on the y-axis.

2.3 Linear Regression Algorithm Functions:

Cost Function:

```
# cost function
def compute_cost(X, y, w, b):
  m = X.shape[0]
  cost = 0.0

  for i in range(m):
    f_wb_i = np.dot(X[i], w) + b
    cost = cost + (f_wb_i - y[i])**2

  cost = cost / (2*m)
  return cost
```

This function will calculates the cost (or loss) for a linear regression model. It computes the mean squared error between the predicted values yhat and the actual target values y. The result is averaged over all data points and returned as the cost.

Gradient Calculation:

```
# compute the gradient for linear regression
def compute_gradient(X, y, w, b):
```

```
  m, n = X.shape
  dj_dw = np.zeros((n,))
  dj_db = 0

  for i in range(m):
    err = (np.dot(X[i], w) + b) - y[i]

    for j in range(n):
      dj_dw[j] = dj_dw[j] + err * X[i, j]
    dj_db = dj_db + err

  dj_dw = dj_dw / m
  dj_db = dj_db / m

  return dj_db, dj_dw
```

It calculates the gradients of the cost function with respect to the weights and bias for linear regression. It computes the partial derivatives by iterating over the dataset and accumulating the errors. The gradients are then averaged over all data points.

Gradient Descent Algorithm:

```
# gradient descent algorithm
def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function, alpha, num_iters):
  J_history = []  # store cost J and w's at each iteration for ploting graph.
  w = copy.deepcopy(w_in) # avoid modifying global w within function.
  b = b_in

  for i in range(num_iters):
    dj_db, dj_dw = gradient_function(X, y, w, b)

    w = w - alpha * dj_dw
    b = b - alpha * dj_db

    if i < 100000:
      J_history.append(cost_function(X, y, w, b))   # save cost J at each iterations

    if i %math.ceil(num_iters / 10) == 0:
      print(f'Iteration␣{i:4d}:␣cost{J_history[-1]:8.2f}')

  return w, b, J_history # return final w, b and J history for ploting
```

It performs gradient descent to optimize the weights and bias for linear regression. It iteratively updates the parameters by subtracting the gradient of the cost function scaled by the learning rate. The cost is recorded at each iteration, and progress is printed. The function returns the final optimized parameters and the history of the cost function for plotting.

Calculate MSE:

```
# mse calculation
def mse(y_actual, y_pred):
    error = 0.0

    for actual, predicted in zip(y_actual, y_pred):
        error += (actual - predicted) ** 2
    mse = error / len(y_actual)
    return mse
```

This function will calculate the MSE.

2.4 Running Linear Regression Gradient Descent Algorithm:

Finding Weight and Bias:

```
initial_w = np.zeros(8)
initial_b = 0
iterations = 1000
alpha = 3.0e-2

# run gradient descent
w_final, b_final, J_hist = gradient_descent(X_train, y_train, initial_w, initial_b, compute_co
print(f"b, w found by gradient descent: {b_final:0.2f}, {w_final}")
```

We initializes the weights as zeros, sets the initial bias to 0, and specifies the number of iterations (1000) and the learning rate (alpha). It then runs the gradient descent function on the training data to find the optimal weights and bias. Finally, it prints the learned bias and weights found by gradient descent.

Cost VS Iteration Plot:

```
plt.figure(figsize=(12, 4))
plt.plot(J_hist[:iterations])
plt.title("Cost vs Iteration")
plt.ylabel("Cost")
plt.xlabel("Iteration Steps")
plt.show()
```

We plot the cost vs iteration figure to give a visual representation of the successful training.

Training Set MSE Calculation:

```
# train set predictions
y_train_predict = np.dot(X_train, w_final) + b_final

print(f"MSE for Training Set: {mse(y_train, y_train_predict)}")
```

Calculate the MSE for the Trainng Set.

Plotting the Training vs Predicted Data:

```
# ploting actual training data vs predicted training data
X_features = ['PT08.S1(CO)','C6H6(GT)','PT08.S2(NMHC)','PT08.S3(NOx)','PT08.S4(NO2)','PT08.S5(
fig, ax = plt.subplots(1, 8, figsize=(30, 10), sharey=True)
fig.suptitle("Actual Data vs Predicted Data (Training Dataset)")

for i in range(8):
  ax[i].scatter(X_train[:,i], y_train, label='Target')
  ax[i].scatter(X_train[:,i], y_train_predict, color='#FF9300', label="Prediction")
  ax[i].set_xlabel(X_features[i])
  ax[i].legend()

ax[0].set_ylabel('AH')
plt.show()
```

This code generates a set of subplots where each subplot compares the original target values with the predicted values for a specific feature in the training dataset.

Test Set MSE Calculation:

```
# test set predictions
```

```
y_test_predict = np.dot(X_test, w_final) + b_final

print(f"MSE for Training Set: {mse(y_test, y_test_predict)}")
```

Calculate the MSE for the Test Set.

Plotting the Test vs Predicted Data:

```
# ploting actual test data vs predicted test data
X_features = ['PT08.S1(CO)','C6H6(GT)','PT08.S2(NMHC)','PT08.S3(NOx)','PT08.S4(NO2)','PT08.S5(

fig, ax = plt.subplots(1, 8, figsize=(30, 10), sharey=True)
fig.suptitle("Actual Data vs Predicted Data (Test Dataset)")

for i in range(8):
  ax[i].scatter(X_test[:,i], y_test, label='Target')
  ax[i].scatter(X_test[:,i], y_test_predict, color='#FF9300', label="Prediction")
  ax[i].set_xlabel(X_features[i])
  ax[i].legend()

ax[0].set_ylabel('AH')
plt.show()
```

This code generates a set of subplots where each subplot compares the original target values with the predicted values for a specific feature in the training dataset.

3. Experiment results

3.1 Learning Rate:

Cost vs Iteration:

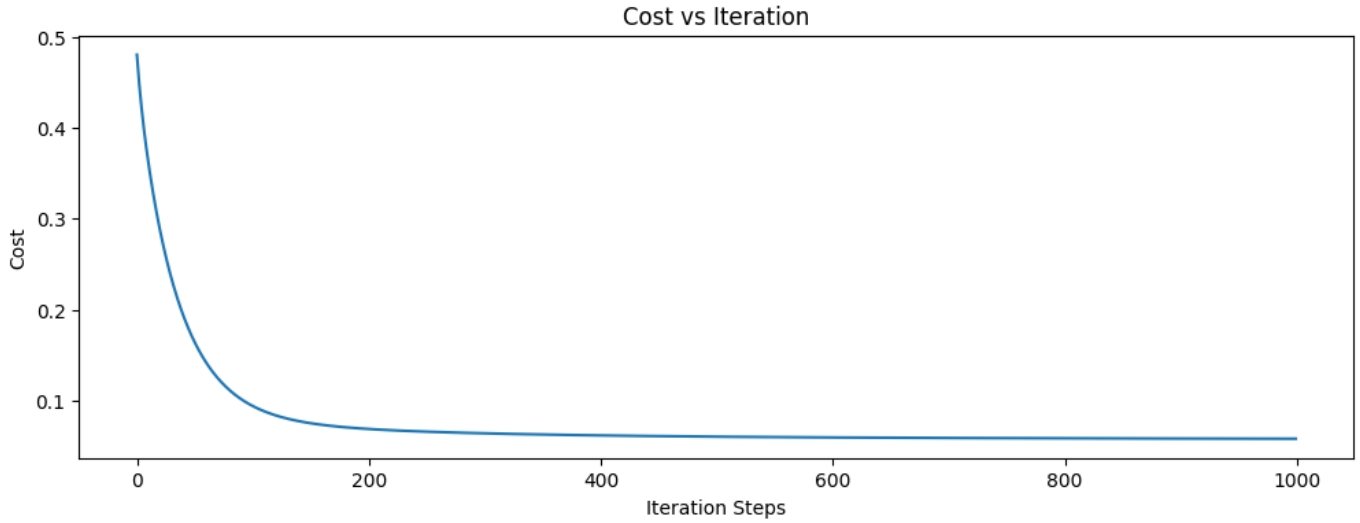

Fig. 1: Learning Rate for Alpha 3.0e-2

The cost function was computed at each iteration during the training process with learning rate 3.0e-2. The cost function represents the difference between the predicted values and the actual targets for the features. Here we see that the cost function decreases smoothly over the no. of iterations, gradually approaching the optimal solution and it levels out and does not decrease further. This indicates it that the optimization process is converging towards a minimum, suggesting that the algorithm has found a local or global minimum. Therefore this is our optimal learning rate

3.2 Analysis of Three Different Target Labels:

Training Data:

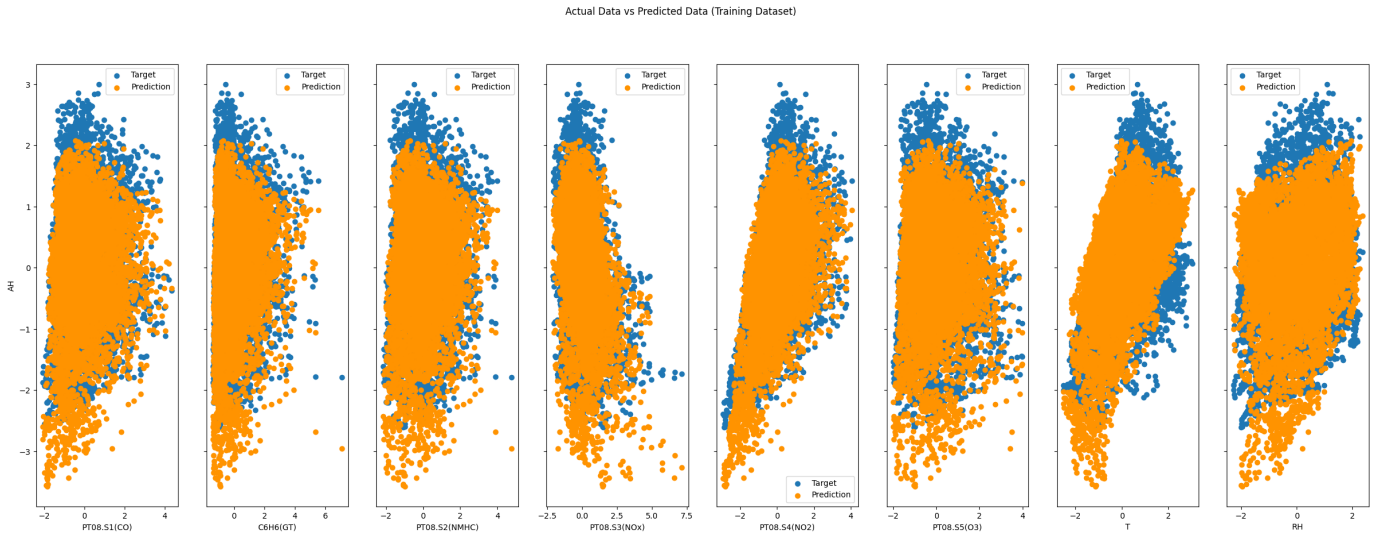

Fig. 2: Actual vs Predicted Data of Training Dataset for AH

Fig. 3: Actual vs Predicted Data of Training Dataset for RH

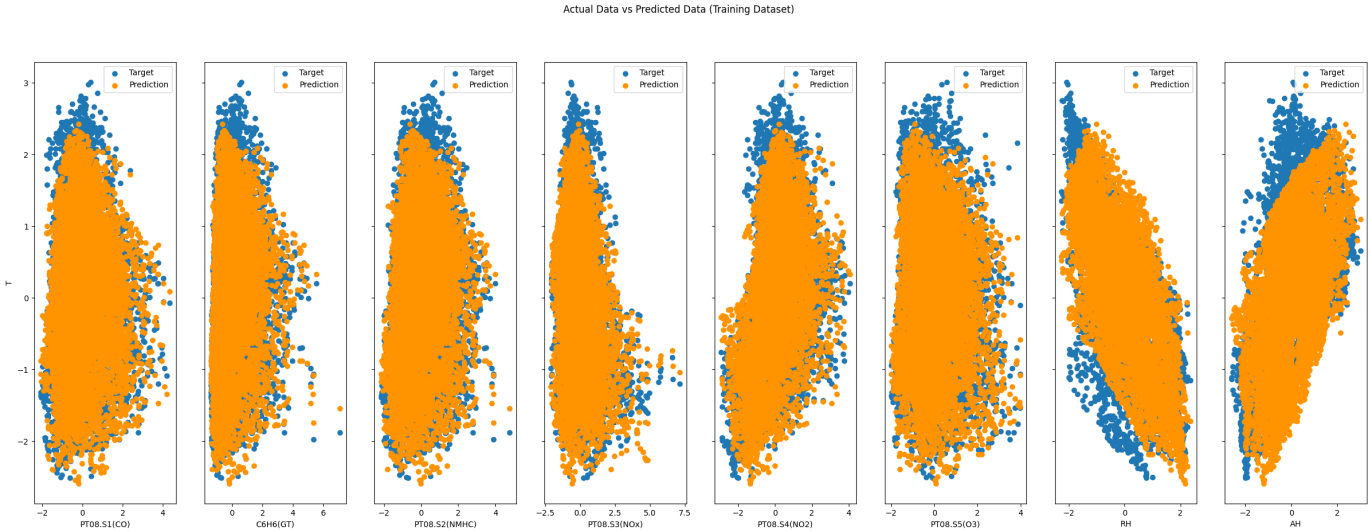

Fig. 4: Actual vs Predicted Data of Training Dataset for T

Mean Squareed Error of Training Data:

MSE for Training Dataset		
AH	**RH**	**T**
0.115	0.096	0.070

The table shows the Mean Squared Error (MSE) for the training dataset across three features: T, RH, and AH. The MSE values indicate that T has the lowest error (0.070), suggesting it is the most accurately predicted feature, while AH has the highest error (0.115), indicating it is the least accurately predicted feature among the three.

Test Data:

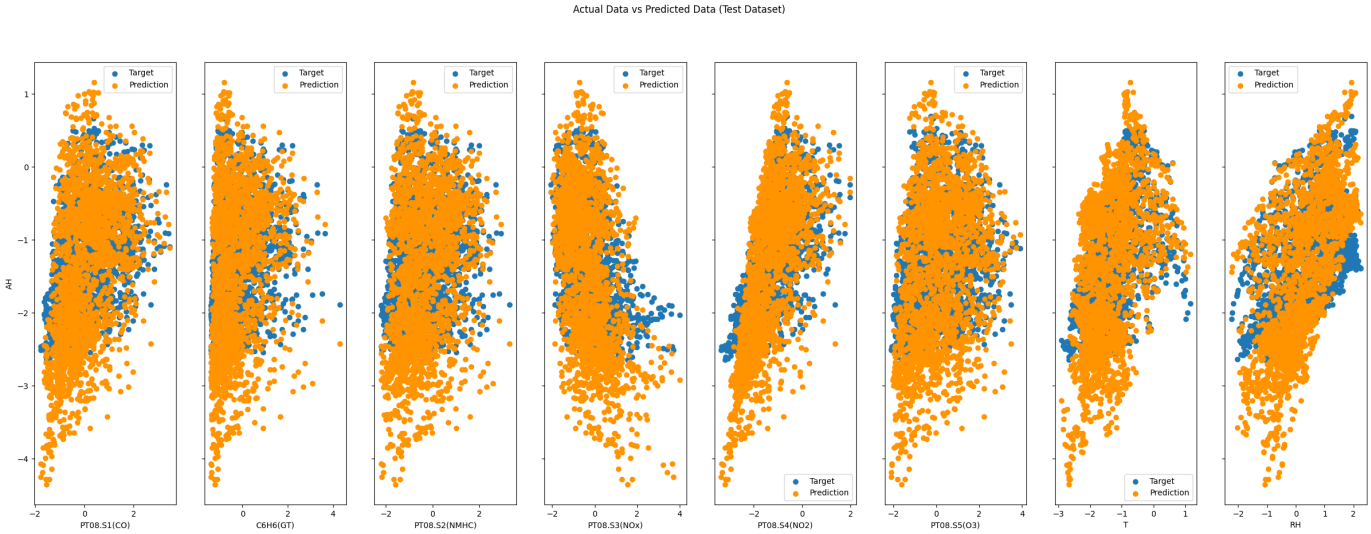

Fig. 5: Actual vs Predicted Data of Test Dataset for AH

Fig. 6: Actual vs Predicted Data of Test Dataset for RH

Fig. 7: Actual vs Predicted Data of Test Dataset for T

Mean Squareed Error of Test Data:

MSE for Test Dataset		
AH	**RH**	**T**
0.190	0.299	0.201

The table shows the Mean Squared Error (MSE) for the training dataset across three features: T, RH, and AH. The MSE values indicate that AH has the lowest error (0.190), suggesting it is the most accurately predicted feature, while RH has the highest error (0.299), indicating it is the least accurately predicted feature among the three.

4. Discussion

In this assignment we implemented the Multivariate Linear Regression using the Gradient Descent Algorithm and test it on the Air Quality Dataset [1]. We also learned to process dataset, apply feature engineering on dataset. We try to get the best performenace for our model by experimenting with different labels (AH,RH,T) and optimize the training by choosing different learning rateand no of iterations in order to get the most accurate hypothesis equation. We choose the learning rate 3.0e-2 first and it become the best learning rate so we don't need to text with other values. After training the model we check the accuracy by calculathing MSE for both training and testing dataset. We found out that we got the best lowest MSE during testing process which is 0.096 while we take RH as the target.

References

[1] U. M. L. Repository. Air quality dataset. https://archive.ics.uci.edu/dataset/360/air+quality, 2008.