Department of Electrical Engineering

Faculty Member: <Dr Attique Dawood> **Date:** <3/16/2025>

Semester: <8th Semester> **Group:** <>

CS435 Parallel and Distributed Processing

Lab 06: Parallel Implementation of Linear Regression

		PLO4	PLO4	PLO5	PLO8
		CLO3	CLO3	CLO4	CLO5
Student Name	**Reg. No**	**Viva / Quiz / Demo** 5 Marks	**Analysis of Data in Report** 5 Marks	**Modern Tool Usage** 5 Marks	**Ethics** 5 Marks
Usman Ayub	368149				
Muhammad Ahmad Naseem	337619				

Introduction /

This laboratory exercise will focus on using multi-threaded programming to optimize a linear regression algorithm which is a basic supervised learning technique for prediction of continuous labels

Objectives

The following are the main objectives of this lab:

- Create multiple threads for parallel programming
- Load a structured dataset into a C++ program
- Calculate the cost of the linear regression model
- Employ gradient descent to update the weights in the model

Lab Conduct

- Respect faculty and peers through speech and actions
- The lab faculty will be available to assist the students. In case some aspect of the lab experiment is not understood, the students are advised to seek help from the faculty.
- In the tasks, there are commented lines such as #YOUR CODE STARTS HERE# where you have to provide the code. You must put the code between the #START and #END parts of these commented lines. Do NOT remove the commented lines.
- Use the tab key to provide the indentation in python.
- When you provide the code in the report, keep the font size at 12

Theory

Linear Regression is a very basic supervised learning technique. To calculate the loss in each training example, the difference between a hypothesis (y^) and the label (y) is calculated. The hypothesis is a linear equation of the features (x) in the dataset with the coefficients (b, w1, w2,…) acting as the weight parameters. These weight parameters are initialized to random values at the start but are then trained over time to learn the model.

The cost function is used to calculated the error between the predicted y^ and the actual y. This cost is used to determine how the weights are to be adjusted in what is called the gradient descent algorithm. The gradient descent uses a step size (alpha) as a hyperparameter which can be tuned. This hyperparameter is varied to determine the model that best fits the dataset.

The terminal commands for Linux are given as:

cd <directory>	change directory
cd..	go back to previous directory
pwd	print the current directory
ls	list the contents of the current directory
touch <file.extension>	create a file
sudo apt install build-essential	install g++ (if not on system)
make	show compiled files in the makefile
./<.out>	execute .out file
chmod +x <file>	make file executable
su -	get sudo access in Virtual Box Ubuntu
usermod –a –G sudo vboxuser	get sudo access (second command

For each of the given tasks, write the code in script files and execute the scripts in Linux. Screenshots of the tasks must be taken showing all relevant output of the code.

Lab Task 1 – Load Dataset into 2D Array

Write a C++ program that loads the CSV dataset into a 2D array. Ensure that the array is of float/double type. You will need to load the features into a separate array (X_train) and the labels into another array (Y_train). After loading the dataset, display the headings and the first few rows of the dataset. Provide the code and all relevant screenshots.

TASK 1 CODE STARTS HERE

Code:

```
#include <iostream>
#include <fstream>
#include <sstream>

using namespace std;

#define MAX_ROWS 200  // Maximum rows to read
#define NUM_FEATURES 6 // Number of input features

int main() {
   ifstream file("lab6_dataset_original.csv");
   if (!file.is_open()) {
      cerr << "Error opening file!" << endl;
      return -1;
   }

   double* X[MAX_ROWS]; // Pointer array for input features
```

```
    double* Y = new double[MAX_ROWS]; // Pointer for target variable (CO2
Emissions)

    for (int i = 0; i < MAX_ROWS; i++)
        X[i] = new double[NUM_FEATURES];

    string line;
    int row = 0;

    while (getline(file, line) && row < MAX_ROWS) {
        stringstream ss(line);
        string cell;
        int col = 0, feature_idx = 0;

        if (row > 0) {  // Skip header row
            while (getline(ss, cell, ',')) {
                if (col == 3) X[row - 1][feature_idx++] = stod(cell); // Engine Size
                if (col == 4) X[row - 1][feature_idx++] = stod(cell); // Cylinders
                if (col == 7) X[row - 1][feature_idx++] = stod(cell); // City Fuel
                if (col == 8) X[row - 1][feature_idx++] = stod(cell); // Hwy Fuel
                if (col == 9) X[row - 1][feature_idx++] = stod(cell); // Comb Fuel
                if (col == 10) X[row - 1][feature_idx++] = stod(cell); // Comb MPG
                if (col == 11) Y[row - 1] = stod(cell); // CO2 Emissions
                col++;
            }
        }
        row++;
    }
    file.close();

    // Print first 5 rows for verification
    cout << "First 5 rows of data:\n";
```

```
    for (int i = 0; i < min(row - 1, 5); i++) {
        for (int j = 0; j < NUM_FEATURES; j++)
            cout << X[i][j] << " ";
        cout << "| CO2: " << Y[i] << endl;
    }

    // Free allocated memory
    for (int i = 0; i < MAX_ROWS; i++)
        delete[] X[i];
    delete[] Y;

    return 0;
}
```

TASK 1 CODE ENDS HERE

TASK 1 SCREENSHOT STARTS HERE

```
(base) usman@UsmanAyub:/mnt/e/8th semester/Pdp/Labs/lab6$ ls
'CS435 Lab 06 Manual - Parallel Implementation of Linear Regression.docx'   lab6_dataset_original.csv   task1.cpp
'PDP Lab 06 - Linear Regression.pdf'                                        makefile
(base) usman@UsmanAyub:/mnt/e/8th semester/Pdp/Labs/lab6$ make
g++ -Wall -Wextra -o task1 task1.cpp
(base) usman@UsmanAyub:/mnt/e/8th semester/Pdp/Labs/lab6$ ./task1
First 5 rows of data:
2 4 9.9 6.7 8.5 33 | CO2: 196
2.4 4 11.2 7.7 9.6 29 | CO2: 221
1.5 4 6 5.8 5.9 48 | CO2: 136
3.5 6 12.7 9.1 11.1 25 | CO2: 255
3.5 6 12.1 8.7 10.6 27 | CO2: 244
(base) usman@UsmanAyub:/mnt/e/8th semester/Pdp/Labs/lab6$
```

TASK 1 SCREENSHOT ENDS HERE

Lab Task 2 – The Training Cost *(Input X_train to Cost J)*

Write a C++ program that loads the CSV dataset, initializes the weights b, w1, w2 and w3 and also initializes a specific number of threads (with proper passing of struct arguments). The threads are to be assigned a portion of the dataset so that each thread accesses a specific range of the training examples.

After these initializations, your program must calculate the training cost given as follows:

Prediction: $h(x) = b + w_1x_1 + w_2x_2 + w_3x_3$

$$\textbf{Cost}: J(w, b) = \frac{1}{2m}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})^2$$

In the equations, the X and y are the features and labels of the training dataset respectively, m is the total number of training examples. For parallel programming, each thread computes a "sub-cost" all of which are to be added in the end to get the final cost. Execute your code using different number of threads (at least 3) and show that they give the same answer. Provide the code and all relevant screenshots.

TASK 2 CODE STARTS HERE

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <thread>
#include <mutex>

using namespace std;

#define MAX_ROWS 200  // Max dataset rows to read
#define NUM_FEATURES 3 // Number of input features

// Struct for dataset
struct Dataset {
   double** X;
   double* Y;
```

```
    int num_examples; // Number of training examples
};

// Struct for thread data
struct ThreadData {
    int start;
    int end;
    Dataset dataset;
    double b;
    double w1;
    double w2;
    double w3;
};

// Global variables
double total_cost = 0.0;
mutex cost_mutex;

// Function to compute cost for a portion of the dataset
void compute_cost(ThreadData data) {
    double sub_cost = 0.0;
    for (int i = data.start; i < data.end; i++) {
        double h_x = data.b + data.w1 * data.dataset.X[i][0] + data.w2 *
data.dataset.X[i][1] + data.w3 * data.dataset.X[i][2];
        sub_cost += (h_x - data.dataset.Y[i]) * (h_x - data.dataset.Y[i]);
    }
    lock_guard<mutex> lock(cost_mutex); // Acquire lock for thread-safe update
    total_cost += sub_cost;
}

int main() {
    ifstream file("vehicles.csv");
```

```
    if (!file.is_open()) {
        cerr << "Error opening file!" << endl;
        return -1;
    }

    // Allocate memory for dataset
    Dataset dataset;
    dataset.X = new double*[MAX_ROWS];
    dataset.Y = new double[MAX_ROWS];

    for (int i = 0; i < MAX_ROWS; i++) {
        dataset.X[i] = new double[NUM_FEATURES];
    }

    string line;
    int row = 0;

    // Read CSV data
    while (getline(file, line) && row < MAX_ROWS) {
        stringstream ss(line);
        string cell;
        int col = 0, feature_idx = 0;

        if (row > 0) { // Skip header row
            while (getline(ss, cell, ',')) {
                if (col == 3) dataset.X[row - 1][feature_idx++] = stod(cell); // Engine
Size
                if (col == 4) dataset.X[row - 1][feature_idx++] = stod(cell); // Cylinders
                if (col == 9) dataset.X[row - 1][feature_idx++] = stod(cell); // Comb
Fuel
                if (col == 11) dataset.Y[row - 1] = stod(cell); // CO2 Emissions
                col++;
```

```
            }
        }
        row++;
    }
    file.close();

    dataset.num_examples = row - 1; // Actual number of training examples

    int num_threads; // Number of threads to use (changeable)
    cout << "Enter the number of threads: ";
    cin >> num_threads;

    // Initialize weights randomly
    double b = 1.0, w1 = 0.5, w2 = -0.3, w3 = 0.8;

    vector<thread> threads;

    // Calculate batch sizes and create thread data
    int base_batch_size = dataset.num_examples / num_threads;
    int remainder = dataset.num_examples % num_threads; // Handle uneven
division

    // Create threads
    int start = 0;
    for (int i = 0; i < num_threads; i++) {
        int batch_size = base_batch_size + (i < remainder ? 1 : 0); // Add 1 to batch
size for the first 'remainder' threads
        int end = start + batch_size;

        ThreadData data = {start, end, dataset, b, w1, w2, w3};
        threads.emplace_back(compute_cost, data);
```

```
        start = end;
    }

    // Wait for threads to finish
    for (auto& t : threads)
        t.join();

    // Compute final cost
    total_cost = total_cost / (2 * dataset.num_examples);
    cout << "Total cost with " << num_threads << " threads: " << total_cost <<
endl;

    // Free allocated memory
    for (int i = 0; i < MAX_ROWS; i++)
        delete[] dataset.X[i];
    delete[] dataset.X;
    delete[] dataset.Y;

    return 0;
}
```

TASK 2 CODE ENDS HERE

TASK 2 SCREENSHOT STARTS HERE

```
(base) usman@UsmanAyub:/mnt/e/8th semester/Pdp/Labs/lab6$ ./task2
Enter the number of threads: 2
Total cost with 2 threads: 35000.1
(base) usman@UsmanAyub:/mnt/e/8th semester/Pdp/Labs/lab6$ ./task2
Enter the number of threads: 3
Total cost with 3 threads: 35000.1
(base) usman@UsmanAyub:/mnt/e/8th semester/Pdp/Labs/lab6$ ./task2
Enter the number of threads: 4
Total cost with 4 threads: 35000.1
(base) usman@UsmanAyub:/mnt/e/8th semester/Pdp/Labs/lab6$
```

TASK 2 SCREENSHOT ENDS HERE

Lab Task 3 – Gradient Descent Algorithm *(Cost J to dW, db)*

In this task, you will extend your previously written code to complete the linear regression training algorithm. To update the weights, the derivatives need to be calculated. This is the part that will use different number of threads for computing the summation. The derivatives can be calculated as follows:

$$dw_j = \frac{\partial \mathrm{J}}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}(\mathrm{h}(\mathrm{x}^{(i)}) - \mathrm{y}^{(i)})x_j{}^{(i)}$$

$$db = \frac{\partial \mathrm{J}}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(\mathrm{h}(\mathrm{x}^{(i)}) - \mathrm{y}^{(i)})$$

The gradient descent algorithm is given as follows (the *alpha* is the learning rate which is a tuning hyperparameter set to a fixed value):

$$w_j := w_j - \alpha \frac{\partial \mathrm{J}}{\partial w_j}$$

$$b := b - \alpha \frac{\partial \mathrm{J}}{\partial w_j}$$

For the submission, you will need to run the gradient descent algorithm at least 3 times to update the weights. For each time, alter the number of threads. You will need to print the weights and cost both before and after the weight update. Provide the code and all relevant screenshots of the final output.

```
### TASK 3 CODE STARTS HERE ###
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <thread>
#include <mutex>

using namespace std;

#define MAX_ROWS 200
#define NUM_FEATURES 6

struct Dataset { double** X; double* Y; int num_examples; };
struct ThreadData { int start, end; Dataset dataset; double b,
w[NUM_FEATURES], *dw[NUM_FEATURES], *db; };

mutex dw_mutexes[NUM_FEATURES];
mutex db_mutex;

void compute_derivatives(ThreadData data) {
   double local_dw[NUM_FEATURES] = {0.0}, local_db = 0.0;
   for (int i = data.start; i < data.end; i++) {
      double h_x = data.b;
      for (int j = 0; j < NUM_FEATURES; j++) h_x += data.w[j] * data.dataset.X[i][j];
      double error = h_x - data.dataset.Y[i];
      for (int j = 0; j < NUM_FEATURES; j++) local_dw[j] += error *
data.dataset.X[i][j];
      local_db += error;
   }
```

```
    for (int j = 0; j < NUM_FEATURES; j++) { lock_guard<mutex>
lock(dw_mutexes[j]); *data.dw[j] += local_dw[j]; }
    { lock_guard<mutex> lock(db_mutex); *data.db += local_db; }
}

double compute_cost(Dataset dataset, double b, double w[]) {
    double total_cost = 0.0;
    for (int i = 0; i < dataset.num_examples; i++) {
        double h_x = b;
        for (int j = 0; j < NUM_FEATURES; j++) h_x += w[j] * dataset.X[i][j];
        total_cost += (h_x - dataset.Y[i]) * (h_x - dataset.Y[i]);
    }
    return total_cost / (2 * dataset.num_examples);
}

int main() {
    ifstream file("lab6_dataset_original.csv");
    if (!file.is_open()) { cerr << "Error opening file!" << endl; return -1; }

    Dataset dataset;
    dataset.X = new double*[MAX_ROWS];
    dataset.Y = new double[MAX_ROWS];
    for (int i = 0; i < MAX_ROWS; i++) dataset.X[i] = new
double[NUM_FEATURES];

    string line;
    int row = 0;
    while (getline(file, line) && row < MAX_ROWS) {
        stringstream ss(line);
        string cell;
        int col = 0;
        if (row > 0) {
```

```
        while (getline(ss, cell, ',')) {
            if (col == 3) { try { dataset.X[row - 1][0] = stod(cell); } catch (...) { cerr
<< "err"; return -1; } }
            else if (col == 4) { try { dataset.X[row - 1][1] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
            else if (col == 7) { try { dataset.X[row - 1][2] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
            else if (col == 8) { try { dataset.X[row - 1][3] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
            else if (col == 9) { try { dataset.X[row - 1][4] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
            else if (col == 10) { try { dataset.X[row - 1][5] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
            else if (col == 11) { try { dataset.Y[row - 1] = stod(cell); } catch (...) { cerr
<< "err"; return -1; } break; }
            col++;
        }
      }
      row++;
    }
    file.close();
    dataset.num_examples = row - 1;

    double b = 1.0, w[NUM_FEATURES] = {0.5};
    double alpha = 0.001;

    for (int iteration = 0; iteration < 3; iteration++) {
      int num_threads;
      cout << "Enter threads: "; cin >> num_threads;

      double dw[NUM_FEATURES] = {0.0}, db = 0.0;
```

```
    cout << "Before: b=" << b << ", w="; for (int i = 0; i < NUM_FEATURES; i++)
cout << w[i] << " "; cout << ", Cost=" << compute_cost(dataset, b, w) << endl;

    vector<thread> threads;
    int base_batch_size = dataset.num_examples / num_threads, remainder =
dataset.num_examples % num_threads;

    for (int i = 0, start = 0; i < num_threads; i++) {
      int batch_size = base_batch_size + (i < remainder ? 1 : 0);
      int end = start + batch_size;

      ThreadData data = {start, end, dataset, b,
{w[0],w[1],w[2],w[3],w[4],w[5]},
{&dw[0],&dw[1],&dw[2],&dw[3],&dw[4],&dw[5]}, &db}; // Initialize w and dw
inside the struct
      threads.emplace_back(compute_derivatives, data);
      start = end;
    }

    for (auto& t : threads) t.join();
    for (int i = 0; i < NUM_FEATURES; i++) w[i] -= alpha * (dw[i] /
dataset.num_examples);
    b -= alpha * (db / dataset.num_examples);

    cout << "After: b=" << b << ", w="; for (int i = 0; i < NUM_FEATURES; i++)
cout << w[i] << " "; cout << ", Cost=" << compute_cost(dataset, b, w) << endl;
  }

  for (int i = 0; i < MAX_ROWS; i++) delete[] dataset.X[i];
  delete[] dataset.X;
  delete[] dataset.Y;
```

```
    return 0;
}
```

TASK 3 CODE ENDS HERE

TASK 3 SCREENSHOT STARTS HERE

```
(base) usman@UsmanAyub:/mnt/e/8th semester/Pdp/Labs/lab6$ ./task3
Enter threads: 4
Before: b=1, w=0.5 0 0 0 0 0 , Cost=37052.6
After: b=1.26612, w=1.50089 1.7834 3.83799 2.57787 3.27133 6.42284 , Cost=2310.49
Enter threads: 4
Before: b=1.26612, w=1.50089 1.7834 3.83799 2.57787 3.27133 6.42284 , Cost=2310.49
After: b=1.23977, w=1.47899 1.71976 3.67099 2.45379 3.12371 5.40883 , Cost=1712.93
Enter threads: 4
Before: b=1.23977, w=1.47899 1.71976 3.67099 2.45379 3.12371 5.40883 , Cost=1712.93
After: b=1.24473, w=1.56214 1.84753 3.92036 2.6113 3.33185 5.21298 , Cost=1531.94
(base) usman@UsmanAyub:/mnt/e/8th semester/Pdp/Labs/lab6$
```

TASK 3 SCREENSHOT ENDS HERE

Lab Task 4 – Training Algorithm

In this task, you will use the previously written code to perform the actual training. This can be done in two stages. In the first stage, calculate the cost function on the entire training dataset using a specific number of threads. You will need to store this training loss for later plotting. In the second stage, use the gradient descent portion to update the weights and bias using a specific number of threads. This iteration over the entire dataset is called an *epoch*. You will need to perform the training over several epochs (the epoch number is a hyperparameter you must select at the start of the training). Thus, you will compute the training cost and update the weights at each epoch. At the last epoch, note down the final weight values and plot the training loss (y-axis) over the epochs (x-axis). For the task submission, use a single-threaded approach to find the number of epochs needed to complete the training. Provide the code, the initial weights, the final weights and all relevant screenshots.

```
### TASK 4 CODE STARTS HERE ###
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <cmath>
#include <limits>

using namespace std;

#define MAX_ROWS 200
#define NUM_FEATURES 6

struct Dataset { double** X; double* Y; int num_examples; };
struct ThreadData { int start, end; Dataset dataset; double b,
w[NUM_FEATURES], *dw[NUM_FEATURES], *db; };

double compute_cost(Dataset dataset, double b, double w[]) {
   double total_cost = 0.0;
   for (int i = 0; i < dataset.num_examples; i++) {
      double h_x = b;
      for (int j = 0; j < NUM_FEATURES; j++) h_x += w[j] * dataset.X[i][j];
      total_cost += (h_x - dataset.Y[i]) * (h_x - dataset.Y[i]);
   }
   return total_cost / (2 * dataset.num_examples);
}

void compute_derivatives(ThreadData data, double dw[], double& db) {
   double local_dw[NUM_FEATURES] = {0.0};
   double local_db = 0.0;
```

```
    for (int i = 0; i < data.dataset.num_examples; i++) {
        double h_x = data.b;
        for (int j = 0; j < NUM_FEATURES; j++) h_x += data.w[j] * data.dataset.X[i][j];
        double error = h_x - data.dataset.Y[i];
        for (int j = 0; j < NUM_FEATURES; j++) local_dw[j] += error *
data.dataset.X[i][j];
        local_db += error;
    }
    for(int i = 0; i < NUM_FEATURES; ++i) dw[i] = local_dw[i];
    db = local_db;
}

int main() {
    ifstream file("lab6_dataset_original.csv");
    if (!file.is_open()) { cerr << "Error!" << endl; return -1; }

    Dataset dataset;
    dataset.X = new double*[MAX_ROWS];
    dataset.Y = new double[MAX_ROWS];
    for (int i = 0; i < MAX_ROWS; i++) dataset.X[i] = new
double[NUM_FEATURES];

    string line;
    int row = 0;
    while (getline(file, line) && row < MAX_ROWS) {
        stringstream ss(line);
        string cell;
        int col = 0;
        if (row > 0) {
          while (getline(ss, cell, ',')) {
              if (col == 3) { try { dataset.X[row - 1][0] = stod(cell); } catch (...) { cerr
<< "err"; return -1; } }
```

```
            else if (col == 4) { try { dataset.X[row - 1][1] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
            else if (col == 7) { try { dataset.X[row - 1][2] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
            else if (col == 8) { try { dataset.X[row - 1][3] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
            else if (col == 9) { try { dataset.X[row - 1][4] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
            else if (col == 10) { try { dataset.X[row - 1][5] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
            else if (col == 11) { try { dataset.Y[row - 1] = stod(cell); } catch (...) { cerr
<< "err"; return -1; } break; }
            col++;
          }
        }
        row++;
    }
    file.close();
    dataset.num_examples = row - 1;

    double b = 1.0, w[NUM_FEATURES] = {0.5};
    double alpha = 0.001;
    double convergence_threshold = 1e-3;
    int max_epochs = 10000, num_epochs = 0;
    double previous_cost = numeric_limits<double>::max();
    bool converged = false;

    cout << "Initial: b=" << b << ", w="; for (int i = 0; i < NUM_FEATURES; i++)
cout << w[i] << " "; cout << endl;

    vector<double> training_losses;
    double dw[NUM_FEATURES] = {0.0}, db = 0.0;
```

```
    for (int epoch = 0; epoch < max_epochs; epoch++) {
        double cost = compute_cost(dataset, b, w);
        training_losses.push_back(cost);

        // Show results *before* updating to show the values at each stage
        cout << "Epoch " << epoch + 1 << ": b=" << b << ", w=";
        for (int i = 0; i < NUM_FEATURES; i++) cout << w[i] << " ";
        cout << ", Cost=" << cost << endl;

        if (abs(cost - previous_cost) < convergence_threshold) {
            cout << "Converged after " << epoch + 1 << " epochs." << endl;
            converged = true;
            num_epochs = epoch+1;
            break;
        }

       ThreadData data = {0, dataset.num_examples, dataset, b,
{w[0],w[1],w[2],w[3],w[4],w[5]}, {0}, &db};
        compute_derivatives(data, dw, db);

        for (int i = 0; i < NUM_FEATURES; i++) w[i] -= alpha * (dw[i] /
dataset.num_examples);
        b -= alpha * (db / dataset.num_examples);

        previous_cost = cost;
         num_epochs = epoch+1;
    }

     if(!converged){
            cout << "Did not converge within " << max_epochs << " epochs. " << endl;
        }
```

```
    cout << "Final (Epochs: " << num_epochs << "): b=" << b << ", w="; for (int i =
0; i < NUM_FEATURES; i++) cout << w[i] << " "; cout << endl;

    ofstream loss_file("training_loss.csv");
    if (loss_file.is_open()) {
        for (size_t i = 0; i < training_losses.size(); i++) {
            loss_file << i + 1 << " " << training_losses[i] << endl;
        }
        loss_file.close();
        cout << "Training loss data written to training_loss.txt" << endl;
    } else {
        cerr << "Unable to open training_loss.txt for writing." << endl;
    }

    for (int i = 0; i < MAX_ROWS; i++) delete[] dataset.X[i];
    delete[] dataset.X;
    delete[] dataset.Y;

    return 0;
}
```

Python Code:

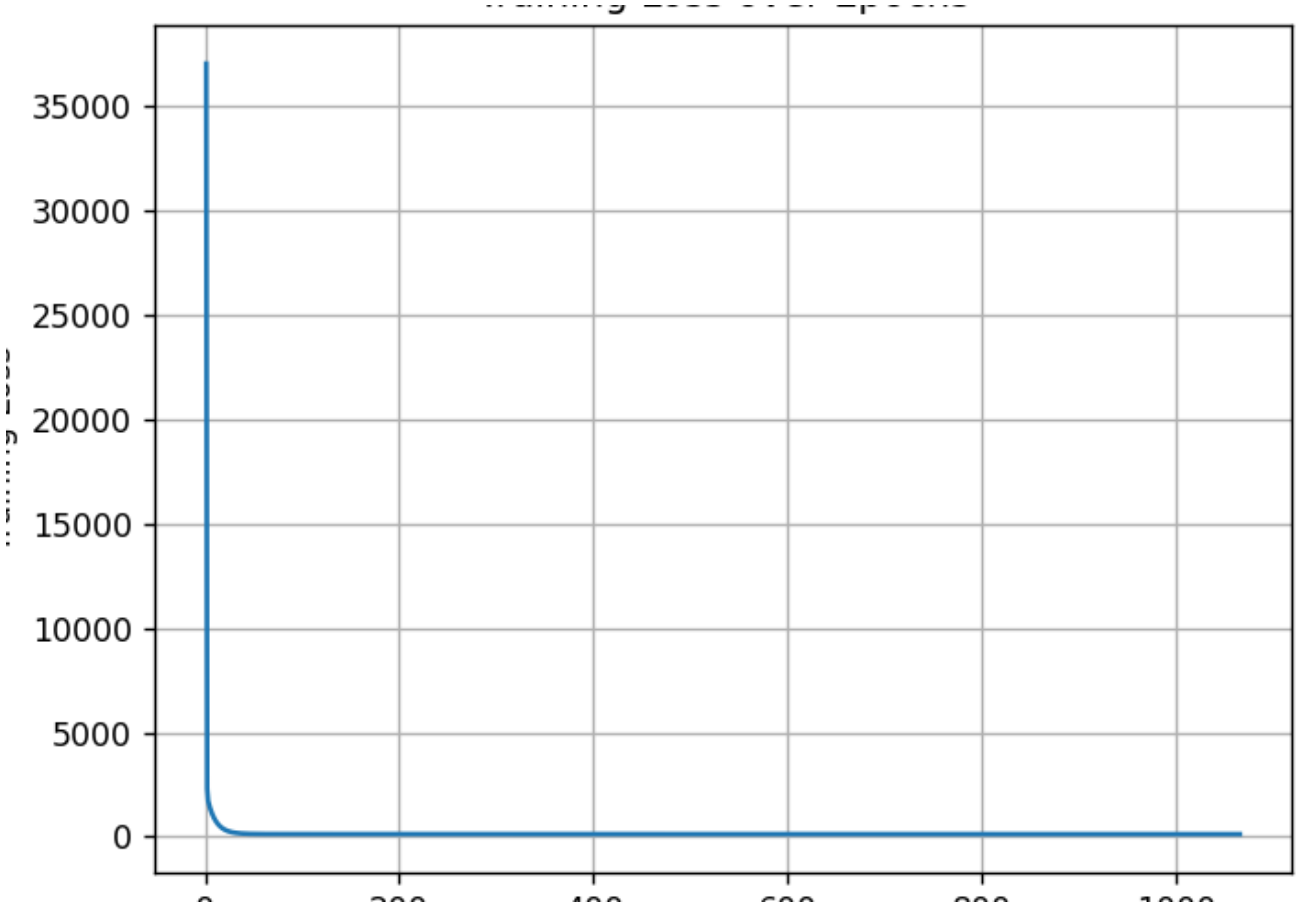
```
import matplotlib.pyplot as plt

def plot_loss(filename="training_loss.txt"):
    epochs, losses = [], []
    with open(filename, 'r') as f:
        for line in f:
            try:
                epoch, loss = map(float, line.split())
                epochs.append(int(epoch)) #Cast epoch as int
                losses.append(loss)
            except ValueError:
```

```
            continue #Skip rows with errors


    plt.plot(epochs, losses)
    plt.xlabel('Epoch')
    plt.ylabel('Training Loss')
    plt.title('Training Loss over Epochs')
    plt.grid(True)
    plt.show()

if __name__ == "__main__":
    plot_loss()
```

TASK 4 CODE ENDS HERE

TASK 4 SCREENSHOT STARTS HERE

```
Epoch 1049: b=1.44415, w=2.78108 5.58064 6.97971 4.24153 5.77096 0.762109 , Cost=126.711
Epoch 1050: b=1.44432, w=2.78077 5.58155 6.97957 4.24137 5.77083 0.762113 , Cost=126.71
Epoch 1051: b=1.44449, w=2.78046 5.58247 6.97943 4.24121 5.7707 0.762116 , Cost=126.709
Epoch 1052: b=1.44466, w=2.78014 5.58339 6.97928 4.24106 5.77057 0.762119 , Cost=126.708
Epoch 1053: b=1.44483, w=2.77983 5.5843 6.97914 4.2409 5.77044 0.762123 , Cost=126.707
Epoch 1054: b=1.445, w=2.77952 5.58522 6.979 4.24074 5.77031 0.762126 , Cost=126.706
Epoch 1055: b=1.44517, w=2.77921 5.58613 6.97886 4.24058 5.77018 0.762129 , Cost=126.705
Epoch 1056: b=1.44534, w=2.77889 5.58704 6.97871 4.24043 5.77005 0.762132 , Cost=126.704
Epoch 1057: b=1.44551, w=2.77858 5.58795 6.97857 4.24027 5.76992 0.762135 , Cost=126.703
Epoch 1058: b=1.44568, w=2.77827 5.58886 6.97843 4.24012 5.76979 0.762139 , Cost=126.702
Epoch 1059: b=1.44585, w=2.77795 5.58977 6.97829 4.23996 5.76966 0.762142 , Cost=126.701
Epoch 1060: b=1.44602, w=2.77764 5.59067 6.97815 4.2398 5.76953 0.762145 , Cost=126.7
Epoch 1061: b=1.44619, w=2.77732 5.59158 6.97801 4.23965 5.7694 0.762148 , Cost=126.699
Epoch 1062: b=1.44636, w=2.77701 5.59249 6.97787 4.23949 5.76927 0.762151 , Cost=126.698
Epoch 1063: b=1.44653, w=2.77669 5.59339 6.97773 4.23934 5.76915 0.762154 , Cost=126.697
Epoch 1064: b=1.4467, w=2.77638 5.59429 6.97759 4.23919 5.76902 0.762157 , Cost=126.696
Epoch 1065: b=1.44687, w=2.77606 5.5952 6.97745 4.23903 5.76889 0.76216 , Cost=126.695
Epoch 1066: b=1.44704, w=2.77575 5.5961 6.97731 4.23888 5.76876 0.762163 , Cost=126.694
Converged after 1066 epochs.
Final (Epochs: 1066): b=1.44704, w=2.77575 5.5961 6.97731 4.23888 5.76876 0.762163
Training loss data written to training_loss.txt
(base) usman@UsmanAyub:/mnt/e/8th semester/Pdp/Labs/lab6$
```



TASK 4 SCREENSHOT ENDS HERE

Lab Task 5 – Single-threaded Vs. Multi-threaded Comparison

In this task, you will vary the number of threads and training over the previously determined epochs in order to check for any training optimization. You will need to provide the training vs. epoch for at least 8 different numbers of threads. For each number of threads, determine the time taken to complete the training. Make another plot showing the number of threads (x-axis) and the time taken (y-axis)

TASK 5 CODE STARTS HERE

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <thread>
#include <mutex>
#include <cmath>
#include <limits>
#include <chrono>

using namespace std;

#define MAX_ROWS 200
#define NUM_FEATURES 6
#define TOTAL_EPOCHS 1066 // Previously determined epochs

struct Dataset { double** X; double* Y; int num_examples; };
struct ThreadData { int start, end; Dataset dataset; double b,
w[NUM_FEATURES], *dw[NUM_FEATURES], *db; };

mutex dw_mutexes[NUM_FEATURES];
mutex db_mutex;

double compute_cost(Dataset dataset, double b, double w[]) {
   double total_cost = 0.0;
   for (int i = 0; i < dataset.num_examples; i++) {
      double h_x = b;
      for (int j = 0; j < NUM_FEATURES; j++) h_x += w[j] * dataset.X[i][j];
      total_cost += (h_x - dataset.Y[i]) * (h_x - dataset.Y[i]);
   }
```

```
    return total_cost / (2 * dataset.num_examples);
}

void compute_derivatives(ThreadData data) {
    double local_dw[NUM_FEATURES] = {0.0}, local_db = 0.0;
    for (int i = data.start; i < data.end; i++) {
        double h_x = data.b;
        for (int j = 0; j < NUM_FEATURES; j++) h_x += data.w[j] * data.dataset.X[i][j];
        double error = h_x - data.dataset.Y[i];
        for (int j = 0; j < NUM_FEATURES; j++) local_dw[j] += error *
data.dataset.X[i][j];
        local_db += error;
    }
     for (int j = 0; j < NUM_FEATURES; j++) { lock_guard<mutex>
lock(dw_mutexes[j]); *data.dw[j] += local_dw[j]; }
    { lock_guard<mutex> lock(db_mutex); *data.db += local_db; }
}

int main() {
    ifstream file("lab6_dataset_original.csv");
    if (!file.is_open()) { cerr << "Error!" << endl; return -1; }

    Dataset dataset;
    dataset.X = new double*[MAX_ROWS];
    dataset.Y = new double[MAX_ROWS];
    for (int i = 0; i < MAX_ROWS; i++) dataset.X[i] = new
double[NUM_FEATURES];

    string line;
    int row = 0;
    while (getline(file, line) && row < MAX_ROWS) {
        stringstream ss(line);
```

```
        string cell;
        int col = 0;
        if (row > 0) {
          while (getline(ss, cell, ',')) {
              if (col == 3) { try { dataset.X[row - 1][0] = stod(cell); } catch (...) { cerr
<< "err"; return -1; } }
              else if (col == 4) { try { dataset.X[row - 1][1] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
              else if (col == 7) { try { dataset.X[row - 1][2] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
              else if (col == 8) { try { dataset.X[row - 1][3] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
              else if (col == 9) { try { dataset.X[row - 1][4] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
              else if (col == 10) { try { dataset.X[row - 1][5] = stod(cell); } catch (...) {
cerr << "err"; return -1; } }
              else if (col == 11) { try { dataset.Y[row - 1] = stod(cell); } catch (...) { cerr
<< "err"; return -1; } break; }
              col++;
          }
        }
        row++;
    }
    file.close();
    dataset.num_examples = row - 1;

    double b = 1.0, w[NUM_FEATURES] = {0.5};
    double alpha = 0.001;
    const int num_epochs = TOTAL_EPOCHS;  //Fixed the number of epochs

    // Thread counts to test
    vector<int> thread_counts = {1, 2, 4, 8, 12, 16, 20, 24};
```

```
    ofstream time_file("thread_times.txt");  // Store thread counts and times

    for (int num_threads : thread_counts) {
        cout << "Training with " << num_threads << " threads..." << endl;

        // Re-initialize weights and bias for each thread count
        b = 1.0;
        for (int i = 0; i < NUM_FEATURES; i++) w[i] = 0.5;
         vector<double> training_losses; //reset for different thread counts

        auto start_time = chrono::high_resolution_clock::now(); // Start timer

        for (int epoch = 0; epoch < num_epochs; epoch++) {
            double dw[NUM_FEATURES] = {0.0}, db = 0.0;
            vector<thread> threads;
            int base_batch_size = dataset.num_examples / num_threads;
            int remainder = dataset.num_examples % num_threads;

            // Create threads
            int start = 0;
            for (int i = 0; i < num_threads; i++) {
                int batch_size = base_batch_size + (i < remainder ? 1 : 0);
                int end = start + batch_size;

                ThreadData data = {start, end, dataset, b,
    {w[0],w[1],w[2],w[3],w[4],w[5]},
    {&dw[0],&dw[1],&dw[2],&dw[3],&dw[4],&dw[5]}, &db};
                threads.emplace_back(compute_derivatives, data);
                start = end;
            }
            for (auto& t : threads) t.join();
```

```
        for (int i = 0; i < NUM_FEATURES; i++) w[i] -= alpha * (dw[i] /
dataset.num_examples);
        b -= alpha * (db / dataset.num_examples);
       double cost = compute_cost(dataset, b, w); //Calculate cost at each epoch
        training_losses.push_back(cost);
      }
      auto end_time = chrono::high_resolution_clock::now();  // End timer
      auto duration = chrono::duration_cast<chrono::milliseconds>(end_time -
start_time);
      double time_taken = duration.count();

      cout << "Training with " << num_threads << " threads took " << time_taken
<< " ms" << endl;
      time_file << num_threads << " " << time_taken << endl; //Record in
time_file for plotting

      ofstream loss_file("training_loss_" + to_string(num_threads) + ".txt");
        if (loss_file.is_open()) {
          for (size_t i = 0; i < training_losses.size(); i++) {
            loss_file << i + 1 << " " << training_losses[i] << endl;
          }
          loss_file.close();
          cout << "Training loss data for " << num_threads << " written" << endl;

        }else {

         cerr << "Unable to open training loss files" << endl;
        }

    }
    time_file.close();
```

```
    for (int i = 0; i < MAX_ROWS; i++) delete[] dataset.X[i];
    delete[] dataset.X;
    delete[] dataset.Y;

    return 0;
}
```

Python Code:

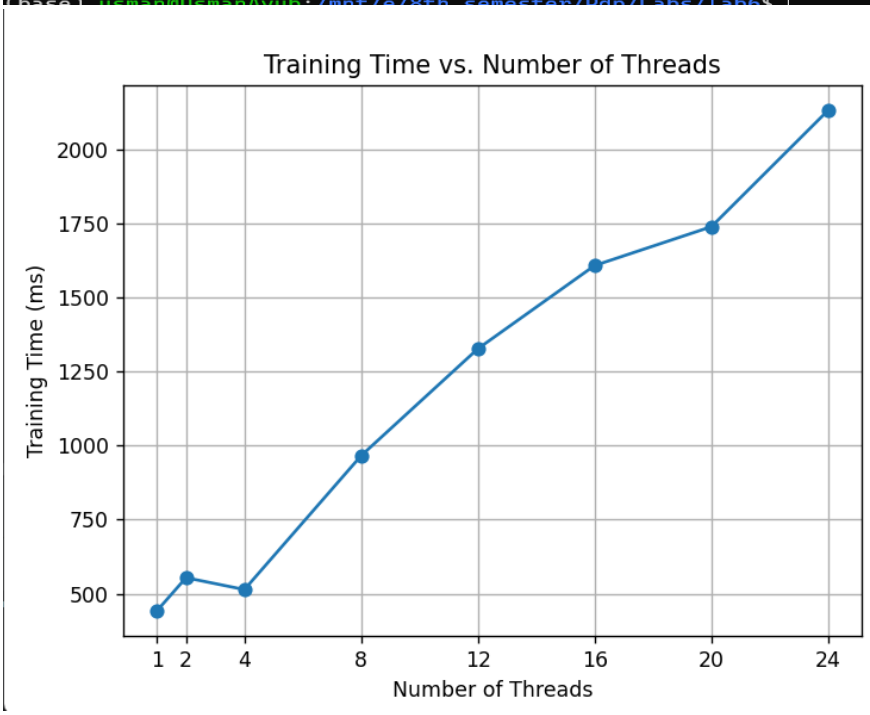
```
import matplotlib.pyplot as plt
import numpy as np

def plot_thread_times(filename="thread_times.txt"):
    threads, times = [], []
    with open(filename, 'r') as f:
        for line in f:
            try:
                thread, time = map(float, line.split())
                threads.append(int(thread))
                times.append(time)
            except ValueError:
                continue

    plt.plot(threads, times, marker='o')
    plt.xlabel('Number of Threads')
    plt.ylabel('Training Time (ms)')
    plt.title('Training Time vs. Number of Threads')
    plt.grid(True)
    plt.xticks(threads) # Ensure x-axis ticks match thread counts
    plt.show()


    def plot_training_losses(thread_counts):
```

```
    plt.figure(figsize=(10, 6)) #Adjust sizing of plot

    for threads in thread_counts:
     filename = f"training_loss_{threads}.txt"
     epochs, losses = [], []

     try:
        with open(filename, 'r') as f:
           for line in f:
              try:
                 epoch, loss = map(float, line.split())
                 epochs.append(int(epoch))
                 losses.append(loss)
              except ValueError:
                 continue

        plt.plot(epochs, losses, label=f'{threads} Threads')
     except FileNotFoundError:
        print(f"File not found: {filename}")

    plt.xlabel('Epoch')
    plt.ylabel('Training Loss')
    plt.title('Training Loss vs. Epoch for Different Thread Counts')
    plt.grid(True)
    plt.legend()
    plt.show()

if __name__ == "__main__":
    plot_thread_times()
    thread_counts_to_plot = [1, 2, 4, 8, 12, 16, 20, 24]  # Must match C++ code
    plot_training_losses(thread_counts_to_plot)
```

TASK 5 CODE ENDS HERE

TASK 5 SCREENSHOT STARTS HERE

```
(base) usman@UsmanAyub:/mnt/e/8th semester/Pdp/Labs/lab6$ ./task5
Training with 1 threads...
Training with 1 threads took 442 ms
Training loss data for 1 written
Training with 2 threads...
Training with 2 threads took 554 ms
Training loss data for 2 written
Training with 4 threads...
Training with 4 threads took 514 ms
Training loss data for 4 written
Training with 8 threads...
Training with 8 threads took 966 ms
Training loss data for 8 written
Training with 12 threads...
Training with 12 threads took 1327 ms
Training loss data for 12 written
Training with 16 threads...
Training with 16 threads took 1608 ms
Training loss data for 16 written
Training with 20 threads...
Training with 20 threads took 1739 ms
Training loss data for 20 written
Training with 24 threads...
Training with 24 threads took 2132 ms
Training loss data for 24 written
```



TASK 5 SCREENSHOT ENDS HERE