Department of Electrical Engineering

Faculty Member: Munadi Sial **Date:** 26th Mar, 2025

Semester: 8th **Group:** 1

CS435 Parallel and Distributed Processing

Lab 07: Parallel Implementation of Logistic 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
Muhammad Ahmed	337619				
Usman Ayub	368149				

Introduction

This laboratory exercise will focus on using multi-threaded programming to optimize a logistic regression algorithm. Logistic regression is a supervised learning technique that incorporates sigmoid function activation with the linear regression algorithm to implement classification. Unlike regression, classification involves discrete labels such 0/1, true/false, cat/not a cat, benign/malignant etc. The sigmoid function causes the hypothesis values to take place between 0 and 1. Similar to regression, weight parameters are trained on a dataset so as to fit a model that can make accurate predictions from that dataset. To prevent overfitting, regularization can be used in logistic regression.

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 logistic 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

Logistic Regression is another basic supervised learning technique besides Linear Regression. In logistic regression, the linear regression algorithm is modified by applying a sigmoid function to the predicted value. This causes the prediction to fall between 0 to 1 values. Thus, logistic regression is actually a classification technique built from the linear regression. The sigmoid function is a type of an activation function. Aside from loss and accuracy, logistic regression also at times, requires calculation of precision and recall. This is needed when the dataset is skewed; the two class labels are not equally distributed in 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 – Sigmoid Activation Function

Download a dataset containing at least 3 features and at least 1 label with discrete binary values. For logistic regression, you will implement the following hypothesis:

$$h(x) = g(b + w_1x_1 + w_2x_2 + w_3x_3 + \dots)$$

$$g(z) = 1 / (1 + e^{-z})$$

The w represents the weights and the x represents the features. h(x) is the prediction to be calculated for each training example and its difference with the label y of that training example will represent the loss. The g(z) function represents the *sigmoid activation function*. In this task, you will write a function that takes in a value z as argument and outputs the result of the sigmoid activation g(z). Provide the code and all relevant screenshots for this task.

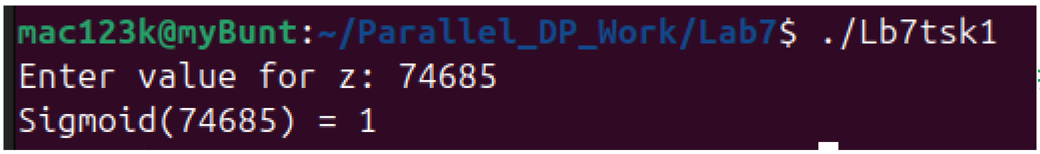
TASK 1 CODE STARTS HERE

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <cmath>
#include <string>

// Sigmoid function
double sigmoid(double z) {
   return 1.0 / (1.0 + std::exp(-z));
}

int main() {
   double z;
   std::cout << "Enter value for z: ";
   std::cin >> z;
```

```
    double result = sigmoid(z);
    std::cout << "Sigmoid(" << z << ") = " << result << std::endl;

    return 0;
}
```

TASK 1 CODE ENDS HERE

TASK 1 SCREENSHOT STARTS HERE



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

$$Cost: J(w) = \frac{1}{m}\sum_{i=0}^{m-1}\left(-y^{(i)}log\left(h\left(x^{(i)}\right)\right) - (1 - y^{(i)})log\left(1 - h\left(x^{(i)}\right)\right)\right)$$

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 <cmath>
#include <thread>
#include <mutex>

struct TrainingExample {
  std::vector<double> features;
  int label;
};

struct ThreadArgs {
  int start_idx;
  int end_idx;
  const std::vector<TrainingExample>* dataset;
  const std::vector<double>* weights;
  double sub_cost = 0.0;
};

std::mutex cost_mutex;

// Sigmoid function
double sigmoid(double z) {
  return 1.0 / (1.0 + std::exp(-z));
}

// Hypothesis function h(x) = g(b + w1x1 + w2x2 + ...)
double hypothesis(const std::vector<double>& weights, const std::vector<double>& features) {
  double z = weights[0]; // bias
  for (size_t i = 0; i < features.size(); ++i) {
    z += weights[i + 1] * features[i];
  }
  return sigmoid(z);
}
```

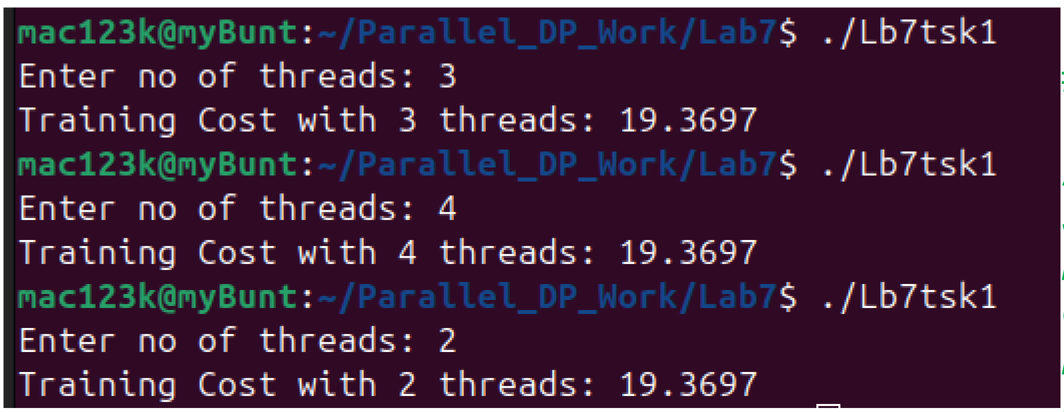
```
// Thread function to compute partial cost
void compute_cost(ThreadArgs* args) {
    double partial_cost = 0.0;

    for (int i = args->start_idx; i < args->end_idx; ++i) {
        const TrainingExample& ex = args->dataset->at(i);
        double pred = hypothesis(*args->weights, ex.features);
        int y = ex.label;

        // Avoid log(0)
        pred = std::min(std::max(pred, 1e-15), 1 - 1e-15);

        partial_cost += -y * std::log(pred) - (1 - y) * std::log(1 - pred);
    }

    std::lock_guard<std::mutex> lock(cost_mutex);
    args->sub_cost = partial_cost;
}

// CSV loader
std::vector<TrainingExample> loadCSV(const std::string& filename) {
    std::ifstream file(filename);
    std::string line;
    std::vector<TrainingExample> data;

    if (!file.is_open()) {
        std::cerr << "Failed to open file\n";
        exit(1);
    }

    std::getline(file, line); // skip header

    while (std::getline(file, line)) {
        std::stringstream ss(line);
        std::string token;
        std::vector<double> features;
        int label;
        int col_index = 0;
        int target_columns[] = {3, 4, 5, 6, 7}; // columns for features
        int label_column = 8;
```

```
        std::vector<std::string> tokens;
        while (std::getline(ss, token, ',')) {
            tokens.push_back(token);
        }

        for (int col : target_columns) {
            features.push_back(std::stod(tokens[col]));
        }

        label = std::stoi(tokens[label_column]);
        data.push_back({features, label});
    }

    return data;
}

int main() {
    std::string filename = "startup_data.csv";
    auto dataset = loadCSV(filename);
    int m = dataset.size();

    std::vector<double> weights = {0.1, 0.2, -0.1, 0.05, 0.3, -0.2}; // b, w1...w5

    int num_threads = 1;
    std::cout<<"Enter no of threads: ";
    std::cin>>num_threads;
    std::vector<std::thread> threads(num_threads);
    std::vector<ThreadArgs> args(num_threads);

    int chunk_size = m / num_threads;
    double total_cost = 0.0;

    for (int i = 0; i < num_threads; ++i) {
        args[i].start_idx = i * chunk_size;
        args[i].end_idx = (i == num_threads - 1) ? m : (i + 1) * chunk_size;
        args[i].dataset = &dataset;
        args[i].weights = &weights;

        threads[i] = std::thread(compute_cost, &args[i]);
    }
```

```
    for (int i = 0; i < num_threads; ++i) {
        threads[i].join();
        total_cost += args[i].sub_cost;
    }

    double cost = total_cost / m;
    std::cout << "Training Cost with " << num_threads << " threads: " << cost << std::endl;

    return 0;
}
```

TASK 2 CODE ENDS HERE

TASK 2 SCREENSHOT STARTS HERE



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 logistic 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:

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

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

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

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

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

The gradient descent for logistic regression may seem identical to that in linear regression, however, it should be noted that they are not the same formulas as the cost function for the logistic regression is different from that of linear regression.

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 <cmath>
#include <thread>
#include <mutex>
```

```
struct TrainingExample {
    std::vector<double> features;
    int label;
};

struct CostThreadArgs {
    int start_idx;
    int end_idx;
    const std::vector<TrainingExample>* dataset;
    const std::vector<double>* weights;
    double sub_cost = 0.0;
};

struct GradThreadArgs {
    int start_idx;
    int end_idx;
    const std::vector<TrainingExample>* dataset;
    const std::vector<double>* weights;
    std::vector<double> sub_grad_w;
    double sub_grad_b = 0.0;

    GradThreadArgs(int size) : sub_grad_w(size, 0.0) {}
};

std::mutex grad_mutex;
std::mutex cost_mutex;

double sigmoid(double z) {
    return 1.0 / (1.0 + std::exp(-z));
}

double hypothesis(const std::vector<double>& weights, const std::vector<double>& features) {
    double z = weights[0]; // bias
    for (size_t i = 0; i < features.size(); ++i) {
        z += weights[i + 1] * features[i];
    }
    return sigmoid(z);
}

void compute_cost(CostThreadArgs* args) {
```

```
    double partial_cost = 0.0;
    for (int i = args->start_idx; i < args->end_idx; ++i) {
        const TrainingExample& ex = args->dataset->at(i);
        double pred = hypothesis(*args->weights, ex.features);
        int y = ex.label;
        pred = std::min(std::max(pred, 1e-15), 1 - 1e-15);
        partial_cost += -y * std::log(pred) - (1 - y) * std::log(1 - pred);
    }

    std::lock_guard<std::mutex> lock(cost_mutex);
    args->sub_cost = partial_cost;
}

void compute_gradient(GradThreadArgs* args) {
    for (int i = args->start_idx; i < args->end_idx; ++i) {
        const TrainingExample& ex = args->dataset->at(i);
        double pred = hypothesis(*args->weights, ex.features);
        double error = pred - ex.label;
        args->sub_grad_b += error;

        for (size_t j = 0; j < ex.features.size(); ++j) {
            args->sub_grad_w[j] += error * ex.features[j];
        }
    }
}

std::vector<TrainingExample> loadCSV(const std::string& filename) {
    std::ifstream file(filename);
    std::string line;
    std::vector<TrainingExample> data;

    if (!file.is_open()) {
        std::cerr << "Failed to open file\n";
        exit(1);
    }

    std::getline(file, line); // skip header

    while (std::getline(file, line)) {
        std::stringstream ss(line);
        std::string token;
```

```
        std::vector<double> features;
        int label;
        int target_columns[] = {3, 4, 5, 6, 7}; // features
        int label_column = 8;

        std::vector<std::string> tokens;
        while (std::getline(ss, token, ',')) {
            tokens.push_back(token);
        }

        for (int col : target_columns) {
            features.push_back(std::stod(tokens[col]));
        }

        label = std::stoi(tokens[label_column]);
        data.push_back({features, label});
    }

    return data;
}

double calculate_cost(const std::vector<TrainingExample>& dataset, const std::vector<double>&
weights, int num_threads) {
    int m = dataset.size();
    int chunk_size = m / num_threads;

    std::vector<std::thread> threads(num_threads);
    std::vector<CostThreadArgs> args(num_threads);

    double total_cost = 0.0;
    for (int i = 0; i < num_threads; ++i) {
        args[i].start_idx = i * chunk_size;
        args[i].end_idx = (i == num_threads - 1) ? m : (i + 1) * chunk_size;
        args[i].dataset = &dataset;
        args[i].weights = &weights;
        threads[i] = std::thread(compute_cost, &args[i]);
    }

    for (int i = 0; i < num_threads; ++i) {
        threads[i].join();
        total_cost += args[i].sub_cost;
```

```
    }

    return total_cost / m;
}

void gradient_descent(std::vector<double>& weights, const std::vector<TrainingExample>& dataset,
int num_threads, double alpha) {
    int m = dataset.size();
    int n = weights.size() - 1;
    int chunk_size = m / num_threads;

    std::vector<std::thread> threads(num_threads);
    std::vector<GradThreadArgs> args;

    for (int i = 0; i < num_threads; ++i) {
        args.emplace_back(n);
        args[i].start_idx = i * chunk_size;
        args[i].end_idx = (i == num_threads - 1) ? m : (i + 1) * chunk_size;
        args[i].dataset = &dataset;
        args[i].weights = &weights;
    }

    for (int i = 0; i < num_threads; ++i) {
        threads[i] = std::thread(compute_gradient, &args[i]);
    }

    for (int i = 0; i < num_threads; ++i) {
        threads[i].join();
    }

    std::vector<double> grad_w(n, 0.0);
    double grad_b = 0.0;

    for (int i = 0; i < num_threads; ++i) {
        for (int j = 0; j < n; ++j) {
            grad_w[j] += args[i].sub_grad_w[j];
        }
        grad_b += args[i].sub_grad_b;
    }

    weights[0] -= alpha * grad_b / m;
```

```
    for (int j = 0; j < n; ++j) {
      weights[j + 1] -= alpha * grad_w[j] / m;
    }
}

void print_weights(const std::vector<double>& weights) {
    std::cout << "Weights: ";
    for (double w : weights) std::cout << w << " ";
    std::cout << std::endl;
}

int main() {
    std::string filename = "startup_data.csv";
    auto dataset = loadCSV(filename);
    int m = dataset.size();

    std::vector<double> weights = {0.1, 0.2, -0.1, 0.05, 0.3, -0.2};
    std::vector<int> thread_counts = {1, 2, 4};
    double alpha = 0.1;

    for (int t : thread_counts) {
      std::cout << "\n========== Gradient Descent with " << t << " threads ==========\n";
      std::cout << "Before Update:\n";
      print_weights(weights);
      double cost_before = calculate_cost(dataset, weights, t);
      std::cout << "Cost: " << cost_before << "\n";

      gradient_descent(weights, dataset, t, alpha);

      std::cout << "After Update:\n";
      print_weights(weights);
      double cost_after = calculate_cost(dataset, weights, t);
      std::cout << "Cost: " << cost_after << "\n";
    }

    return 0;
}
```

TASK 3 CODE ENDS HERE

TASK 3 SCREENSHOT STARTS HERE

```
mac123k@myBunt:~/Parallel_DP_Work/Lab7$ ./Lb7tsk1

========== Gradient Descent with 1 threads ==========
Before Update:
Weights: 0.1 0.2 -0.1 0.05 0.3 -0.2
Cost: 19.3697
After Update:
Weights: 0.0483995 -7.35947 -61.9436 -2.46673 -141.472 -0.443523
Cost: 14.9208

========== Gradient Descent with 2 threads ==========
Before Update:
Weights: 0.0483995 -7.35947 -61.9436 -2.46673 -141.472 -0.443523
Cost: 14.9208
After Update:
Weights: 0.0915995 -0.744136 1.79632 -0.28945 -31.8033 -0.207413
Cost: 14.8517

========== Gradient Descent with 4 threads ==========
Before Update:
Weights: 0.0915995 -0.744136 1.79632 -0.28945 -31.8033 -0.207413
Cost: 14.8517
After Update:
Weights: 0.1338 5.68433 63.7165 1.84556 77.8255 0.024437
Cost: 19.6185
```

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 <thread>
#include <mutex>
#include <fstream>

struct TrainingExample {
   std::vector<double> features;
   int label;
};

struct ThreadArgs {
   int start_idx;
   int end_idx;
   const std::vector<TrainingExample>* dataset;
   const std::vector<double>* weights;
   double sub_cost = 0.0;
   std::vector<double> gradient_sum;
   double bias_gradient = 0.0;
};

std::mutex mutex;

// Sigmoid function
```

```
double sigmoid(double z) {
  return 1.0 / (1.0 + std::exp(-z));
}

// Hypothesis function h(x)
double hypothesis(const std::vector<double>& weights, const std::vector<double>& features) {
  double z = weights[0]; // bias
  for (size_t i = 0; i < features.size(); ++i)
    z += weights[i + 1] * features[i];
  return sigmoid(z);
}

// Thread function to compute cost + gradients
void compute_gradients(ThreadArgs* args) {
  double partial_cost = 0.0;
  std::vector<double> local_grad(args->weights->size() - 1, 0.0);
  double bias_grad = 0.0;

  for (int i = args->start_idx; i < args->end_idx; ++i) {
    const TrainingExample& ex = args->dataset->at(i);
    double pred = hypothesis(*args->weights, ex.features);
    int y = ex.label;
    pred = std::min(std::max(pred, 1e-15), 1 - 1e-15);
    partial_cost += -y * std::log(pred) - (1 - y) * std::log(1 - pred);
    double diff = pred - y;
    bias_grad += diff;

    for (size_t j = 0; j < ex.features.size(); ++j)
      local_grad[j] += diff * ex.features[j];
  }

  std::lock_guard<std::mutex> lock(mutex);
  args->sub_cost = partial_cost;
  args->gradient_sum = local_grad;
  args->bias_gradient = bias_grad;
}

// CSV loader
std::vector<TrainingExample> loadCSV(const std::string& filename) {
  std::ifstream file(filename);
  std::string line;
```

```
    std::vector<TrainingExample> data;
    std::getline(file, line); // Skip header

    while (std::getline(file, line)) {
        std::stringstream ss(line);
        std::string token;
        std::vector<std::string> tokens;
        while (std::getline(ss, token, ','))
            tokens.push_back(token);

        std::vector<double> features;
        for (int col : {3, 4, 5, 6, 7})
            features.push_back(std::stod(tokens[col]));

        int label = std::stoi(tokens[8]);
        data.push_back({features, label});
    }
    return data;
}

int main() {
    std::string filename = "startup_data.csv";
    auto dataset = loadCSV(filename);
    int m = dataset.size();
    int feature_count = dataset[0].features.size();
    int num_threads = 1;
    int epochs = 10;
    double alpha = 0.01;

    std::vector<double> weights(feature_count + 1, 0.1); // includes bias at index 0
    std::vector<double> training_loss;

    for (int epoch = 0; epoch < epochs; ++epoch) {
        std::vector<std::thread> threads(num_threads);
        std::vector<ThreadArgs> args(num_threads);
        int chunk_size = m / num_threads;
        double total_cost = 0.0;
        std::vector<double> total_grad(feature_count, 0.0);
        double total_bias_grad = 0.0;

        for (int i = 0; i < num_threads; ++i) {
```

```
        args[i].start_idx = i * chunk_size;
        args[i].end_idx = (i == num_threads - 1) ? m : (i + 1) * chunk_size;
        args[i].dataset = &dataset;
        args[i].weights = &weights;
        args[i].gradient_sum.resize(feature_count, 0.0);
        threads[i] = std::thread(compute_gradients, &args[i]);
      }

      for (int i = 0; i < num_threads; ++i) {
        threads[i].join();
        total_cost += args[i].sub_cost;
        for (size_t j = 0; j < feature_count; ++j)
          total_grad[j] += args[i].gradient_sum[j];
        total_bias_grad += args[i].bias_gradient;
      }

      total_cost /= m;
      training_loss.push_back(total_cost);

      weights[0] -= alpha * (total_bias_grad / m); // update bias
      for (size_t j = 0; j < feature_count; ++j)
        weights[j + 1] -= alpha * (total_grad[j] / m);

      std::cout << "Epoch " << epoch + 1 << ": Cost = " << total_cost << "\n";
    }

    std::ofstream out("loss_data.txt");
    for (size_t i = 0; i < training_loss.size(); ++i)
      out << i + 1 << " " << training_loss[i] << "\n";
    out.close();

    std::cout << "\nFinal Weights:\n";
    for (size_t i = 0; i < weights.size(); ++i)
      std::cout << "w" << i << ": " << weights[i] << "\n";

    return 0;
}
```

plot code python:

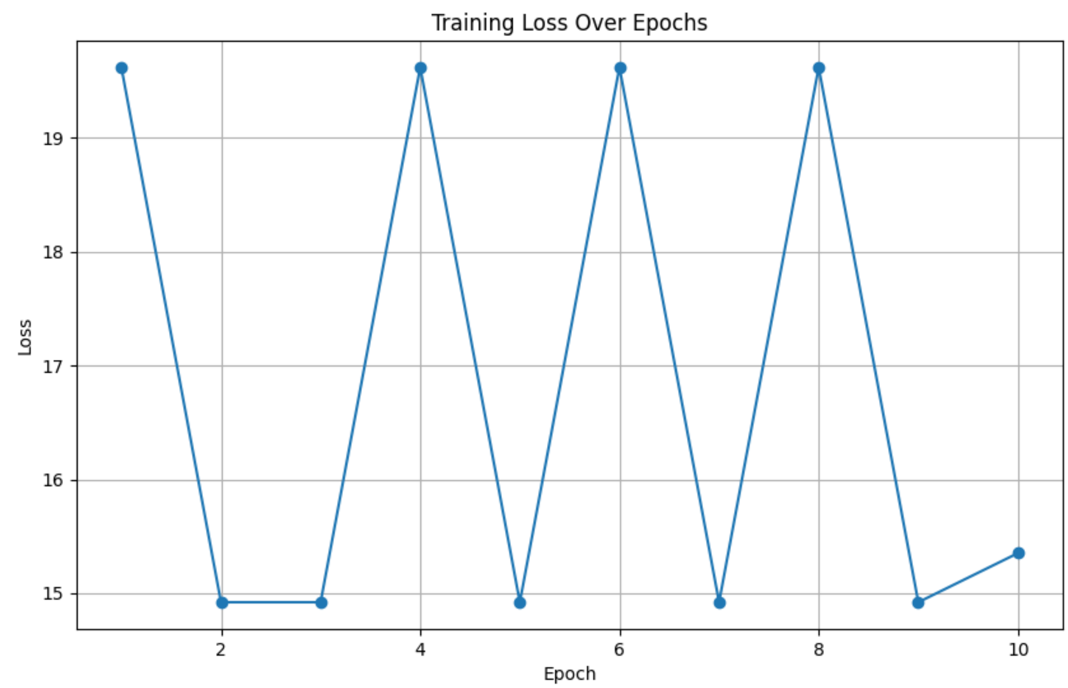
```
import matplotlib.pyplot as plt
```

```
epochs, losses = [], []
with open("loss_data.txt", "r") as f:
    for line in f:
        e, l = line.strip().split()
        epochs.append(int(e))
        losses.append(float(l))

plt.figure(figsize=(10, 6))
plt.plot(epochs, losses, marker='o')
plt.title("Training Loss Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.grid(True)
plt.savefig("training_loss_plot.png")
print("Plot saved as 'training_loss_plot.png'.")
```

TASK 4 CODE ENDS HERE

TASK 4 SCREENSHOT STARTS HERE

```
mac123k@myBunt:~/Parallel_DP_Work/Lab7$ ./Lb7tsk1
Epoch 1: Cost = 19.6185
Epoch 2: Cost = 14.9208
Epoch 3: Cost = 14.9208
Epoch 4: Cost = 19.6185
Epoch 5: Cost = 14.9208
Epoch 6: Cost = 19.6185
Epoch 7: Cost = 14.9208
Epoch 8: Cost = 19.6185
Epoch 9: Cost = 14.9208
Epoch 10: Cost = 15.3519

Final Weights:
w0: 0.10046
w1: 0.054193
w2: 3.1663
w3: 0.168379
w4: 5.19968
w5: 0.117641
```



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

Code:

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

```
#include <vector>
#include <cmath>
#include <thread>
#include <mutex>
#include <chrono>

struct TrainingExample {
   std::vector<double> features;
   int label;
};

struct ThreadArgs {
   int start_idx;
   int end_idx;
   const std::vector<TrainingExample>* dataset;
   const std::vector<double>* weights;
   double sub_cost = 0.0;
   std::vector<double> gradient_sum;
   double bias_gradient = 0.0;
};

std::mutex mutex;

double sigmoid(double z) {
   return 1.0 / (1.0 + std::exp(-z));
}

double hypothesis(const std::vector<double>& weights, const std::vector<double>& features) {
   double z = weights[0]; // bias
   for (size_t i = 0; i < features.size(); ++i)
      z += weights[i + 1] * features[i];
   return sigmoid(z);
}

void compute_gradients(ThreadArgs* args) {
   double partial_cost = 0.0;
   std::vector<double> local_grad(args->weights->size() - 1, 0.0);
   double bias_grad = 0.0;

   for (int i = args->start_idx; i < args->end_idx; ++i) {
      const TrainingExample& ex = args->dataset->at(i);
```

```
        double pred = hypothesis(*args->weights, ex.features);
        int y = ex.label;
        pred = std::min(std::max(pred, 1e-15), 1 - 1e-15);
        partial_cost += -y * std::log(pred) - (1 - y) * std::log(1 - pred);
        double diff = pred - y;
        bias_grad += diff;
        for (size_t j = 0; j < ex.features.size(); ++j)
            local_grad[j] += diff * ex.features[j];
    }

    std::lock_guard<std::mutex> lock(mutex);
    args->sub_cost = partial_cost;
    args->gradient_sum = local_grad;
    args->bias_gradient = bias_grad;
}

std::vector<TrainingExample> loadCSV(const std::string& filename) {
    std::ifstream file(filename);
    std::string line;
    std::vector<TrainingExample> data;
    std::getline(file, line); // Skip header

    while (std::getline(file, line)) {
        std::stringstream ss(line);
        std::string token;
        std::vector<std::string> tokens;
        while (std::getline(ss, token, ','))
            tokens.push_back(token);

        std::vector<double> features;
        for (int col : {3, 4, 5, 6, 7})
            features.push_back(std::stod(tokens[col]));

        int label = std::stoi(tokens[8]);
        data.push_back({features, label});
    }
    return data;
}

int main() {
    std::string filename = "startup_data.csv";
```

```
auto dataset = loadCSV(filename);
int m = dataset.size();
int feature_count = dataset[0].features.size();
std::vector<int> thread_counts = {1, 2, 3, 4, 5, 6, 8, 12};  // 8 variants
int epochs = 10;
double alpha = 0.01;

std::ofstream time_out("thread_vs_time.txt");
time_out << "Threads Time(s)\n";

for (int num_threads : thread_counts) {
  std::cout << "\n🔥 Training with " << num_threads << " thread(s)...\n";
  std::vector<double> weights(feature_count + 1, 0.1); // Reset for each test
  std::vector<double> training_loss;

  auto start_time = std::chrono::high_resolution_clock::now();

  for (int epoch = 0; epoch < epochs; ++epoch) {
    std::vector<std::thread> threads(num_threads);
    std::vector<ThreadArgs> args(num_threads);
    int chunk_size = m / num_threads;
    double total_cost = 0.0;
    std::vector<double> total_grad(feature_count, 0.0);
    double total_bias_grad = 0.0;

    for (int i = 0; i < num_threads; ++i) {
      args[i].start_idx = i * chunk_size;
      args[i].end_idx = (i == num_threads - 1) ? m : (i + 1) * chunk_size;
      args[i].dataset = &dataset;
      args[i].weights = &weights;
      args[i].gradient_sum.resize(feature_count, 0.0);
      threads[i] = std::thread(compute_gradients, &args[i]);
    }

    for (int i = 0; i < num_threads; ++i) {
      threads[i].join();
      total_cost += args[i].sub_cost;
      for (size_t j = 0; j < feature_count; ++j)
        total_grad[j] += args[i].gradient_sum[j];
      total_bias_grad += args[i].bias_gradient;
```

```
        }

        total_cost /= m;
        training_loss.push_back(total_cost);

        weights[0] -= alpha * (total_bias_grad / m); // bias update
        for (size_t j = 0; j < feature_count; ++j)
            weights[j + 1] -= alpha * (total_grad[j] / m);

        std::cout << "Epoch " << epoch + 1 << ": Cost = " << total_cost << "\n";
    }

    auto end_time = std::chrono::high_resolution_clock::now();
    double duration = std::chrono::duration<double>(end_time - start_time).count();
    time_out << num_threads << " " << duration << "\n";

    std::cout << "\nFinal Weights for " << num_threads << " threads:\n";
    for (size_t i = 0; i < weights.size(); ++i)
        std::cout << "w" << i << ": " << weights[i] << "\n";

    // Save training loss per thread count
    std::string loss_filename = "loss_threads_" + std::to_string(num_threads) + ".txt";
    std::ofstream loss_out(loss_filename);
    for (size_t i = 0; i < training_loss.size(); ++i)
        loss_out << i + 1 << " " << training_loss[i] << "\n";
    loss_out.close();
  }

  time_out.close();
  return 0;
}
```

Output:

🔥 Training with 1 thread(s)...
Epoch 1: Cost = 19.6185
Epoch 2: Cost = 14.9208
Epoch 3: Cost = 14.9208
Epoch 4: Cost = 19.6185
Epoch 5: Cost = 14.9208
Epoch 6: Cost = 19.6185

Epoch 7: Cost = 14.9208
Epoch 8: Cost = 19.6185
Epoch 9: Cost = 14.9208
Epoch 10: Cost = 15.3519

Final Weights for 1 threads:
w0: 0.10046
w1: 0.054193
w2: 3.1663
w3: 0.168379
w4: 5.19968
w5: 0.117641

💧 Training with 2 thread(s)...
Epoch 1: Cost = 19.6185
Epoch 2: Cost = 14.9208
Epoch 3: Cost = 14.9208
Epoch 4: Cost = 19.6185
Epoch 5: Cost = 14.9208
Epoch 6: Cost = 19.6185
Epoch 7: Cost = 14.9208
Epoch 8: Cost = 19.6185
Epoch 9: Cost = 14.9208
Epoch 10: Cost = 15.3519

Final Weights for 2 threads:
w0: 0.10046
w1: 0.054193
w2: 3.1663
w3: 0.168379
w4: 5.19968
w5: 0.117641

💧 Training with 3 thread(s)...
Epoch 1: Cost = 19.6185
Epoch 2: Cost = 14.9208
Epoch 3: Cost = 14.9208
Epoch 4: Cost = 19.6185
Epoch 5: Cost = 14.9208

Epoch 6: Cost = 19.6185
Epoch 7: Cost = 14.9208
Epoch 8: Cost = 19.6185
Epoch 9: Cost = 14.9208
Epoch 10: Cost = 15.3519

Final Weights for 3 threads:
w0: 0.10046
w1: 0.054193
w2: 3.1663
w3: 0.168379
w4: 5.19968
w5: 0.117641

🔥 Training with 4 thread(s)...
Epoch 1: Cost = 19.6185
Epoch 2: Cost = 14.9208
Epoch 3: Cost = 14.9208
Epoch 4: Cost = 19.6185
Epoch 5: Cost = 14.9208
Epoch 6: Cost = 19.6185
Epoch 7: Cost = 14.9208
Epoch 8: Cost = 19.6185
Epoch 9: Cost = 14.9208
Epoch 10: Cost = 15.3519

Final Weights for 4 threads:
w0: 0.10046
w1: 0.054193
w2: 3.1663
w3: 0.168379
w4: 5.19968
w5: 0.117641

🔥 Training with 5 thread(s)...
Epoch 1: Cost = 19.6185
Epoch 2: Cost = 14.9208
Epoch 3: Cost = 14.9208
Epoch 4: Cost = 19.6185

Epoch 5: Cost = 14.9208
Epoch 6: Cost = 19.6185
Epoch 7: Cost = 14.9208
Epoch 8: Cost = 19.6185
Epoch 9: Cost = 14.9208
Epoch 10: Cost = 15.3519

Final Weights for 5 threads:
w0: 0.10046
w1: 0.054193
w2: 3.1663
w3: 0.168379
w4: 5.19968
w5: 0.117641

💧 Training with 6 thread(s)...
Epoch 1: Cost = 19.6185
Epoch 2: Cost = 14.9208
Epoch 3: Cost = 14.9208
Epoch 4: Cost = 19.6185
Epoch 5: Cost = 14.9208
Epoch 6: Cost = 19.6185
Epoch 7: Cost = 14.9208
Epoch 8: Cost = 19.6185
Epoch 9: Cost = 14.9208
Epoch 10: Cost = 15.3519

Final Weights for 6 threads:
w0: 0.10046
w1: 0.054193
w2: 3.1663
w3: 0.168379
w4: 5.19968
w5: 0.117641

💧 Training with 8 thread(s)...
Epoch 1: Cost = 19.6185
Epoch 2: Cost = 14.9208
Epoch 3: Cost = 14.9208

Epoch 4: Cost = 19.6185
Epoch 5: Cost = 14.9208
Epoch 6: Cost = 19.6185
Epoch 7: Cost = 14.9208
Epoch 8: Cost = 19.6185
Epoch 9: Cost = 14.9208
Epoch 10: Cost = 15.3519

Final Weights for 8 threads:
w0: 0.10046
w1: 0.054193
w2: 3.1663
w3: 0.168379
w4: 5.19968
w5: 0.117641

🌢 Training with 12 thread(s)...
Epoch 1: Cost = 19.6185
Epoch 2: Cost = 14.9208
Epoch 3: Cost = 14.9208
Epoch 4: Cost = 19.6185
Epoch 5: Cost = 14.9208
Epoch 6: Cost = 19.6185
Epoch 7: Cost = 14.9208
Epoch 8: Cost = 19.6185
Epoch 9: Cost = 14.9208
Epoch 10: Cost = 15.3519

Final Weights for 12 threads:
w0: 0.10046
w1: 0.054193
w2: 3.1663
w3: 0.168379
w4: 5.19968
w5: 0.117641

Python Codes:

```
import matplotlib.pyplot as plt
```

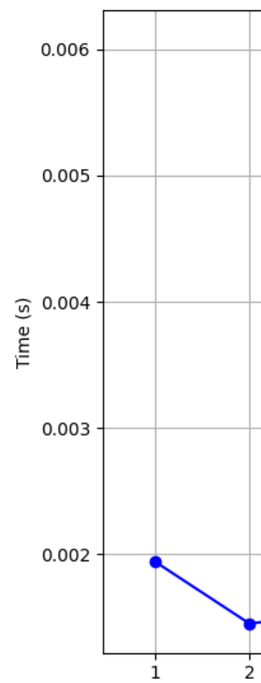
```
# Read the thread vs time data
threads = []
times = []

with open("thread_vs_time.txt", "r") as f:
    next(f)  # Skip the header
    for line in f:
        thread_count, time_taken = line.split()
        threads.append(int(thread_count))
        times.append(float(time_taken))

# Plot the data
plt.figure(figsize=(10, 6))
plt.plot(threads, times, marker='o', linestyle='-', color='b')
plt.xlabel("Number of Threads")
plt.ylabel("Time (s)")
plt.title("Time vs. Number of Threads for Training")
plt.grid(True)
plt.xticks(threads)
plt.tight_layout()
plt.savefig("time_vs_threads.png")
plt.show()



import matplotlib.pyplot as plt
import os

# List all loss files
loss_files = [f"loss_threads_{i}.txt" for i in range(1, 9)]

# Plot the loss for each thread count
plt.figure(figsize=(10, 6))

for loss_file in loss_files:
    if os.path.exists(loss_file):
        epochs = []
        losses = []
        with open(loss_file, "r") as f:
            for line in f:
```

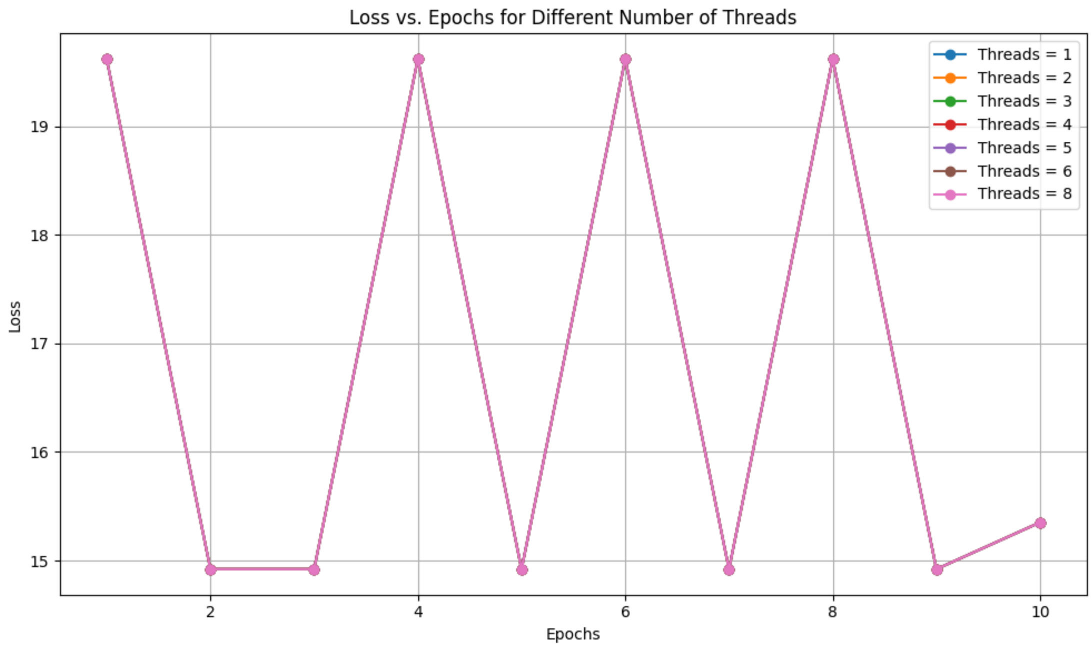
```
            epoch, loss = line.split()
            epochs.append(int(epoch))
            losses.append(float(loss))

        # Plot each thread's loss
        plt.plot(epochs, losses, label=f"Threads = {loss_file[-5]}", marker='o')

plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss vs. Epochs for Different Number of Threads")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.savefig("loss_vs_epochs.png")
plt.show()
```

TASK 5 CODE ENDS HERE

TASK 5 SCREENSHOT STARTS HERE

```
mac123k@myBunt:~/Parallel_DP_Work/Lab7$ ./Lb7tsk1


🔥 Training with 1 thread(s)...
Epoch 1: Cost = 19.6185
Epoch 2: Cost = 14.9208
Epoch 3: Cost = 14.9208
Epoch 4: Cost = 19.6185
Epoch 5: Cost = 14.9208
Epoch 6: Cost = 19.6185
Epoch 7: Cost = 14.9208
Epoch 8: Cost = 19.6185
Epoch 9: Cost = 14.9208
Epoch 10: Cost = 15.3519

Final Weights for 1 threads:
w0: 0.10046
w1: 0.054193
w2: 3.1663
w3: 0.168379
w4: 5.19968
w5: 0.117641
```





TASK 5 SCREENSHOT ENDS HERE