# Assignment 5: Perceptron Report

Introduction

This report details the implementation and analysis of the Perceptron algorithm applied to the dataset provided in `data.csv`. Two primary approaches were investigated as per the assignment requirements: the Heuristic Perceptron (Part 1) and the Gradient Descent Perceptron with a Sigmoid activation function (Part 2). The objective was to understand the algorithms' behavior, particularly the impact of the learning rate, and visualize the learning process.

## Setup and Data Loading

The implementation utilized standard Python libraries: `pandas` for data handling, `numpy` for numerical operations, and `matplotlib` for plotting. A random seed (42) was set for reproducible weight initialization.

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Load the dataset
try:
    data = pd.read_csv('data.csv', header=None, names=['x1', 'x2', 'label'])
    X = data[['x1', 'x2']].values
    y = data['label'].values
    print("Data loaded successfully.")
    print(f"Features shape: {X.shape}, Labels shape: {y.shape}\n")
except FileNotFoundError:
    print("Error: data.csv not found.")
    exit()

# Set random seed
np.random.seed(42)
```

Part 1: Heuristic Perceptron

Implementation Details

This method used a simple rule based on the assignment's "left box". It classified points using a step function (0 or 1 based on score sign). It only learned from mistakes, nudging the dividing line slightly based on the error type (+ for false negative, - for false positive). The process stopped if no mistakes were made in a full pass or if maximum iterations were reached.

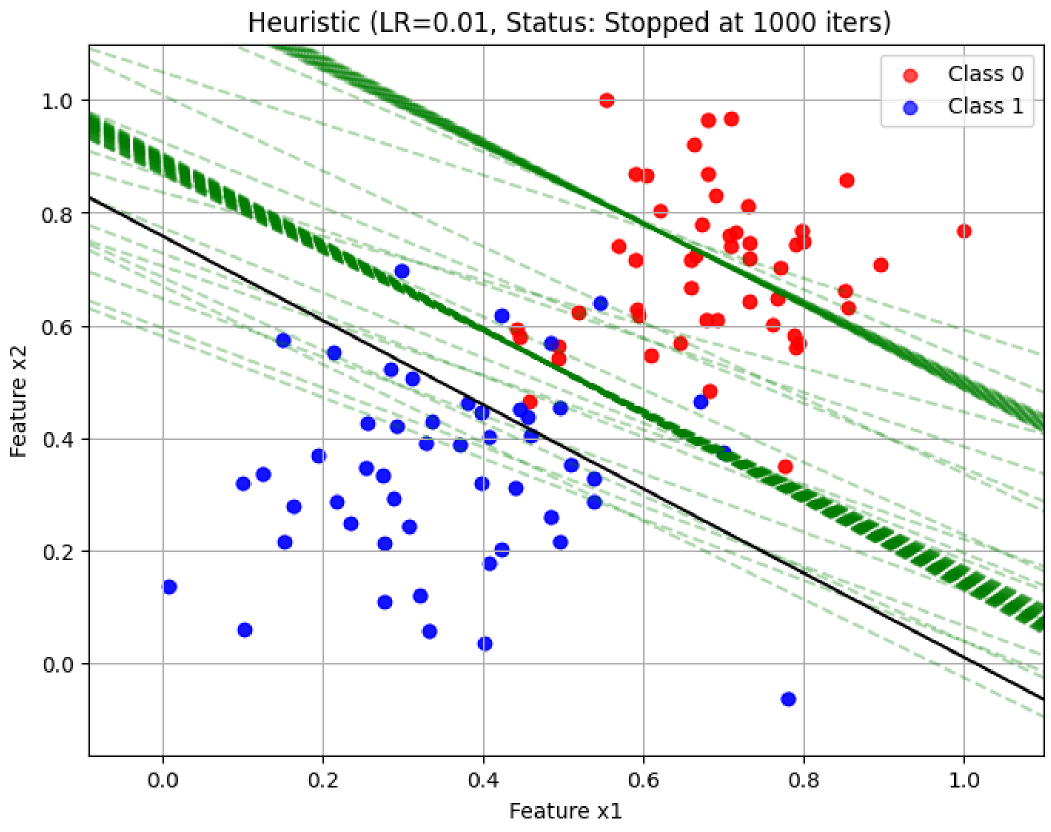
Code Snippet: Heuristic Perceptron Function

```
def run_part1_heuristic(X, y, learning_rate, max_iterations):
    """ Heuristic Perceptron"""
    n_samples, n_features = X.shape
    W = np.random.rand(n_features) * 0.02 - 0.01 # Initialize weights
    b = np.random.rand() * 0.02 - 0.01        # Initialize bias

    plt.figure(figsize=(8, 6)) # New figure for this run
    # (Plotting helper function 'plot_decision_boundary' is assumed defined elsewhere)
    plot_decision_boundary(X, y, W, b, 'Initial', learning_rate, "Part 1: Heuristic",
color='red', linestyle='-')

    converged = False
    updates_made = 0
    for iteration in range(max_iterations):
        misclassified_in_epoch = 0
        for idx, xi in enumerate(X):
            yi = y[idx]
            linear_output = np.dot(xi, W) + b
            predicted_class = 1 if linear_output >= 0 else 0 # Step function classification

            if predicted_class != yi: # Update only if misclassified
                misclassified_in_epoch += 1
                updates_made += 1
                # Apply specific update rules from PDF Box 1
                if predicted_class == 0: # Predicted 0, must be 1 -> Increase score
                    b += learning_rate
                    W += learning_rate * xi
            elif predicted_class == 1: # Predicted 1, must be 0 -> Decrease score
                b -= learning_rate
                W -= learning_rate * xi

            # Plot intermediate boundary occasionally
            if updates_made % (n_samples // 2) == 0:
                 plot_decision_boundary(X, y, W, b, f'Update {updates_made}',
learning_rate, "Part 1: Heuristic", color='green', linestyle='--')

        if misclassified_in_epoch == 0: # Check convergence
            converged = True
            break

    final_iterations_info = f"Converged in {iteration + 1} iters" if converged else
f"Stopped at {max_iterations} iters"
    print(f"Part 1 Result (LR={learning_rate}): {final_iterations_info}. Total Updates:
{updates_made}")
    plot_decision_boundary(X, y, W, b, final_iterations_info, learning_rate, "Part 1:
Heuristic", color='black', linestyle='-', show_legend=True) # Final plot
```

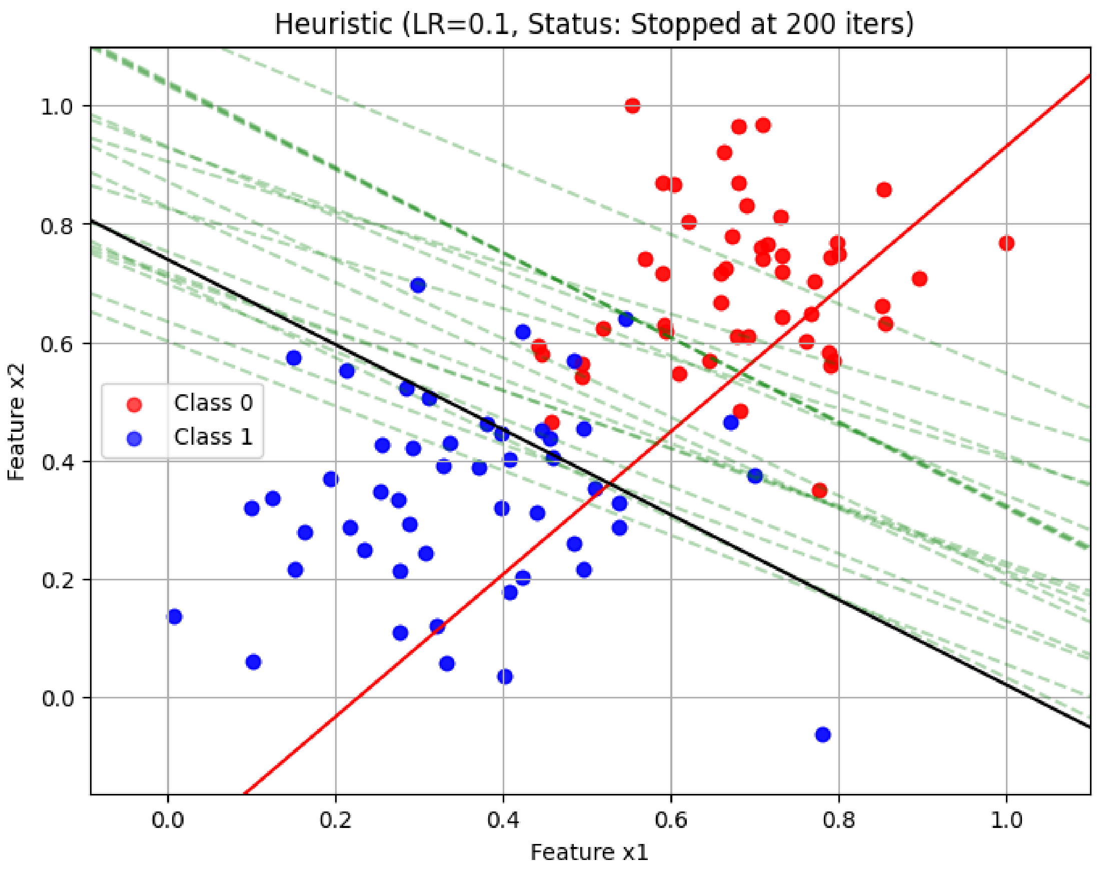
Results and Analysis

In our tests (with seed 42), this method **didn't fully separate** the data within the time limits (1000 tries for LR=0.01, 200 for others). It always got stuck on a few specific points. A tiny learning rate (0.01) was extremely slow and still didn't finish. Even faster rates (0.1, 1.0) got stuck. This simple rule can be sensitive; the starting position or points near the boundary might have made it hard for this rule to get them all right in this run. This method struggled to find a perfect dividing line quickly in this specific test.

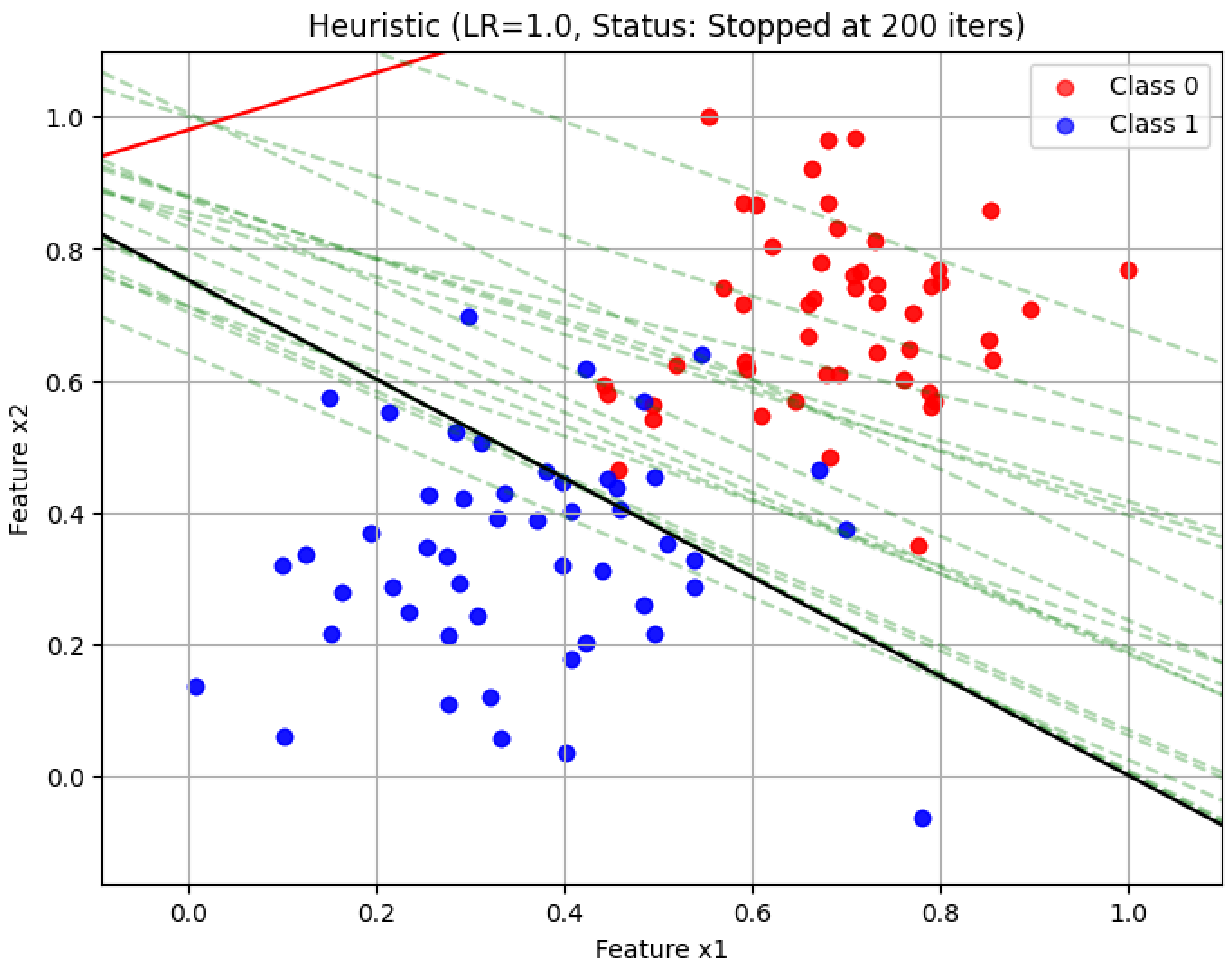
Plot : **Heuristic Perceptron (LR=0.01)**



Plot : Heuristic **Perceptron (LR=0.1)**


Heuristic (LR=0.1, Status: Stopped at 200 iters)
Class 0
Class 1
Feature x1
Feature x2
0.0
0.2
0.4
0.6
0.8
1.0

Plot : Heuristic **Perceptron (LR=1.0)**



Part 2: Gradient Descent Perceptron

Implementation Details

This method used a "smoother" approach with Gradient Descent and Sigmoid activation ("right box" rules). It calculated a score between 0 and 1 (Sigmoid) and adjusted the line after *every* point based on how far the score was from the true label (0 or 1). It ran for 100 epochs and tracked error using Log Loss every 10 cycles.

Code Snippets: Sigmoid, Log Loss, GD Perceptron Function

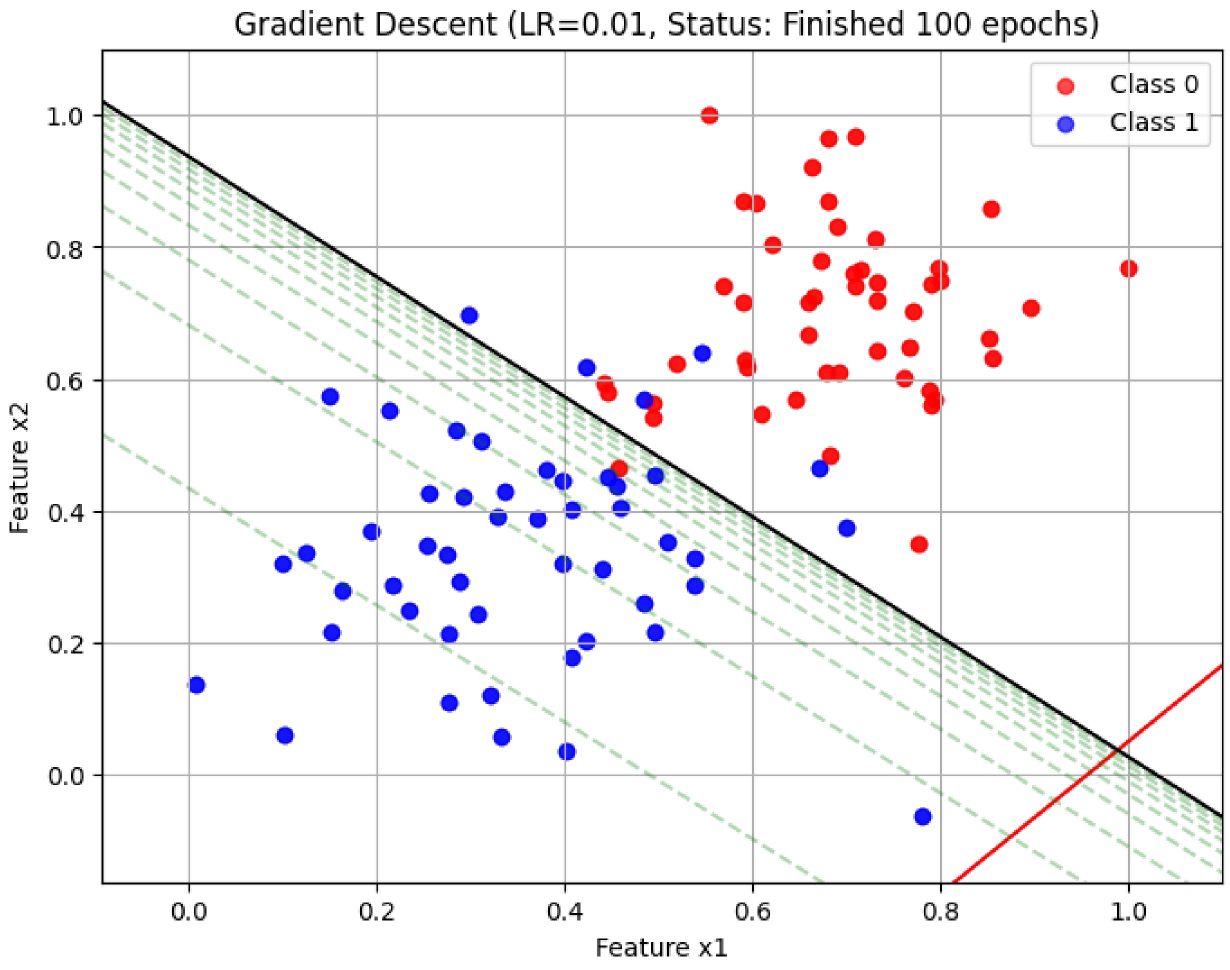
```
# Activation function for Part 2
def sigmoid(z):
    z = np.clip(z, -50, 50)
    return 1.0 / (1.0 + np.exp(-z))

# Loss function for Part 2
def calculate_log_loss(y_true, y_pred_proba):
    epsilon = 1e-15 # Avoid log(0)
    y_pred_clipped = np.clip(y_pred_proba, epsilon, 1 - epsilon)
    loss = - (y_true * np.log(y_pred_clipped) + (1 - y_true) * np.log(1 - y_pred_clipped))
    return np.mean(loss)

# (Using the run_part2_gradient_descent function defined previously)
def run_part2_gradient_descent(X, y, learning_rate, epochs):
    n_samples, n_features = X.shape
    W = np.random.rand(n_features) * 0.02 - 0.01 # Initialize weights
    b = np.random.rand() * 0.02 - 0.01         # Initialize bias

    log_loss_history = []
    epoch_numbers_for_plot = []

    plt.figure(figsize=(8, 6)) # New figure for boundary plot
    plot_decision_boundary(X, y, W, b, 'Initial', learning_rate, " Gradient Descent",
color='red', linestyle='-')

    for epoch in range(epochs):
        y_preds_epoch = np.zeros(n_samples)
        for idx, xi in enumerate(X):
            yi = y[idx]
            z = np.dot(xi, W) + b
            y_hat = sigmoid(z) # Use sigmoid activation
            y_preds_epoch[idx] = y_hat
            error = yi - y_hat # Error based on sigmoid output

            # Apply Gradient Descent update rules
            b += learning_rate * error
            W += learning_rate * error * xi

        # Plot intermediate boundary every 10 epochs
        if (epoch + 1) % 10 == 0 and epoch < epochs - 1:
            plot_decision_boundary(X, y, W, b, f'Epoch {epoch+1}', learning_rate, "Part 2:
```

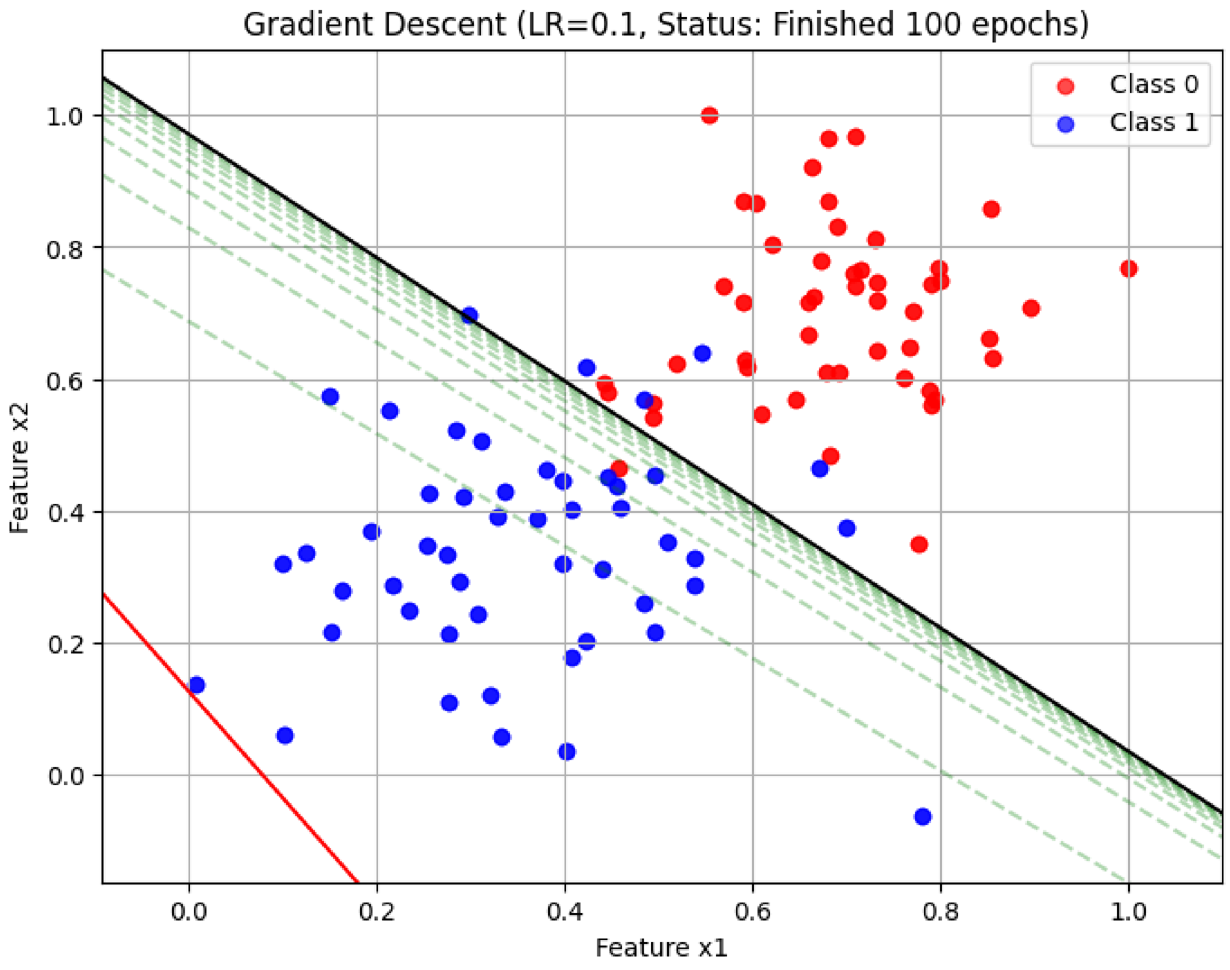
```
Gradient Descent", color='green', linestyle='--')

        # Compute and store log loss every 10 epochs
        if (epoch + 1) % 10 == 0:
            current_loss = calculate_log_loss(y, y_preds_epoch)
            log_loss_history.append(current_loss)
            epoch_numbers_for_plot.append(epoch + 1)

    final_epoch_info = f"Finished {epochs} epochs"
    print(f"Part 2 Result (LR={learning_rate}): {final_epoch_info}. Final LogLoss:
{log_loss_history[-1]:.4f}")
    plot_decision_boundary(X, y, W, b, final_epoch_info, learning_rate, "Part 2: Gradient
Descent", color='black', linestyle='-', show_legend=True) # Final boundary plot

    # Plot Log Loss Error history
    plt.figure(figsize=(8, 6))
    plt.plot(epoch_numbers_for_plot, log_loss_history, marker='o')
    plt.title(f'Log Loss vs. Epochs (LR={learning_rate})'); plt.xlabel('Number of Epochs');
plt.ylabel('Log Loss Error')
    plt.xticks(epoch_numbers_for_plot); plt.grid(True); plt.show()
```

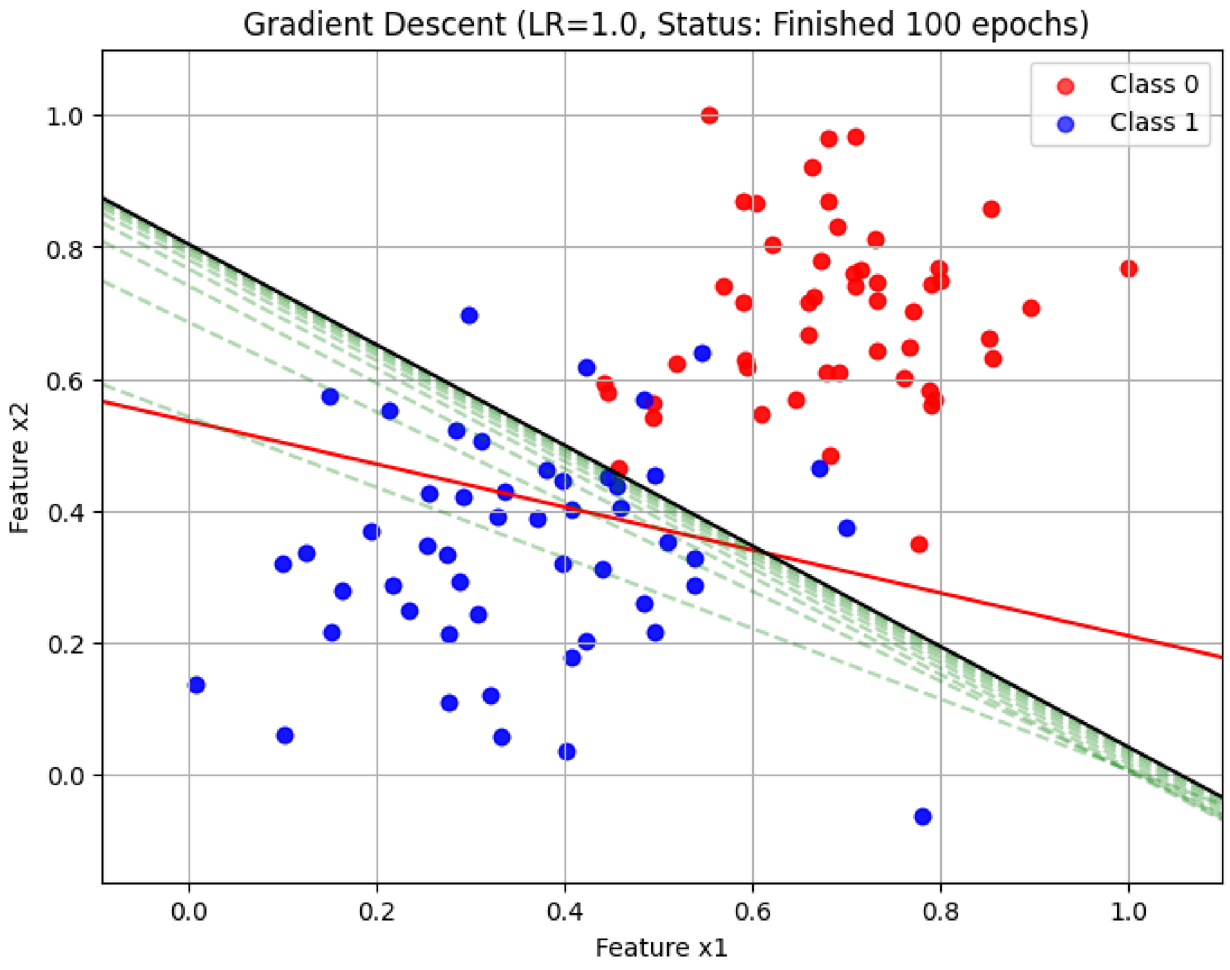
Results and Analysis

This smoother method **worked well**. The Log Loss error went down steadily over 100 epochs, showing successful learning. The learning rate choice mattered: LR=0.1 seemed best, learning quickly and stabilizing well. LR=0.01 was slow but steady. LR=1.0 learned fast initially but might have been less stable. The Log Loss graph clearly showed learning progress (error decreasing). This method seemed more reliable and stable than the simple rule in Part 1.

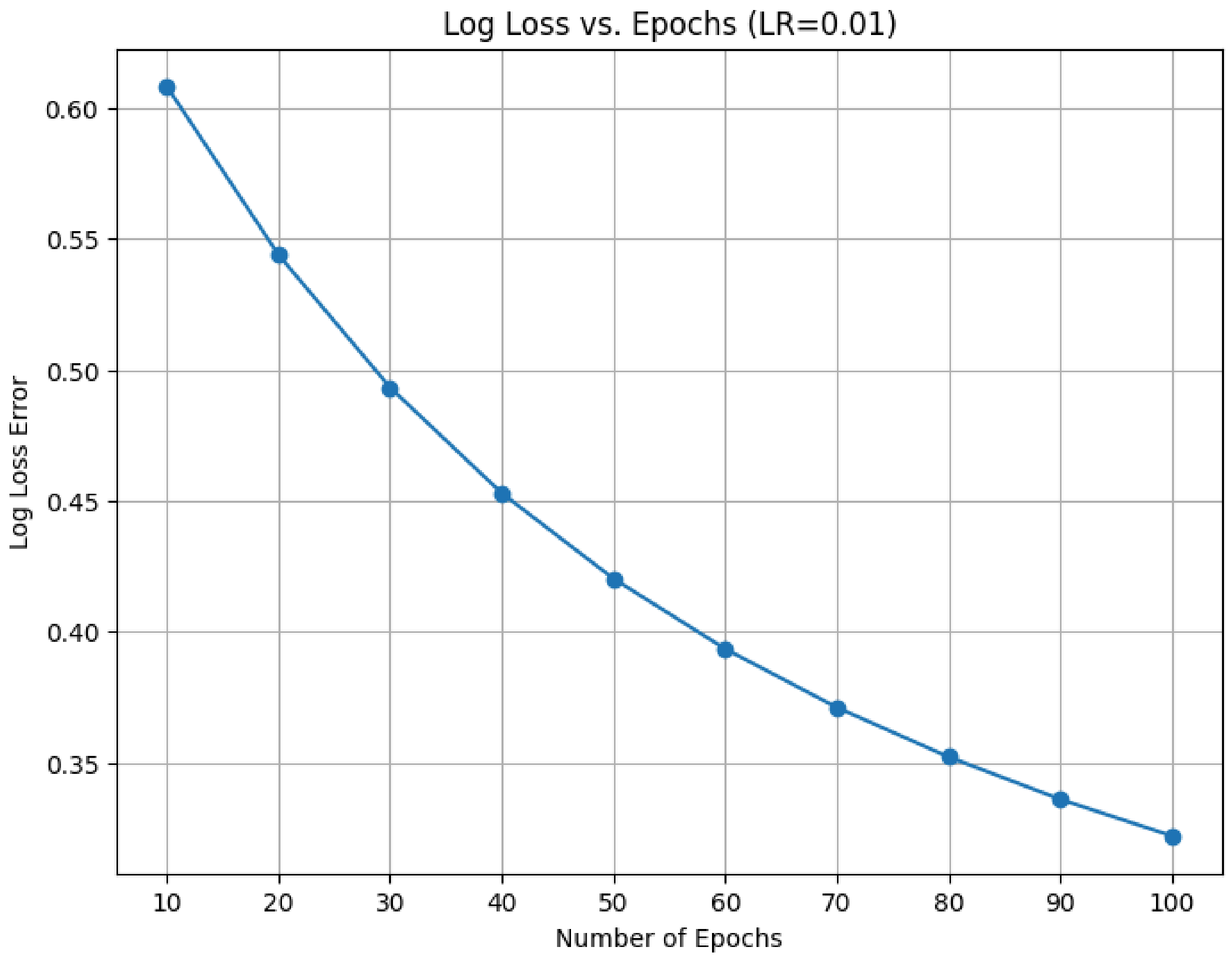
Plots : Gradient Descent Boundaries (LR=0.01, 0.1, 1.0)

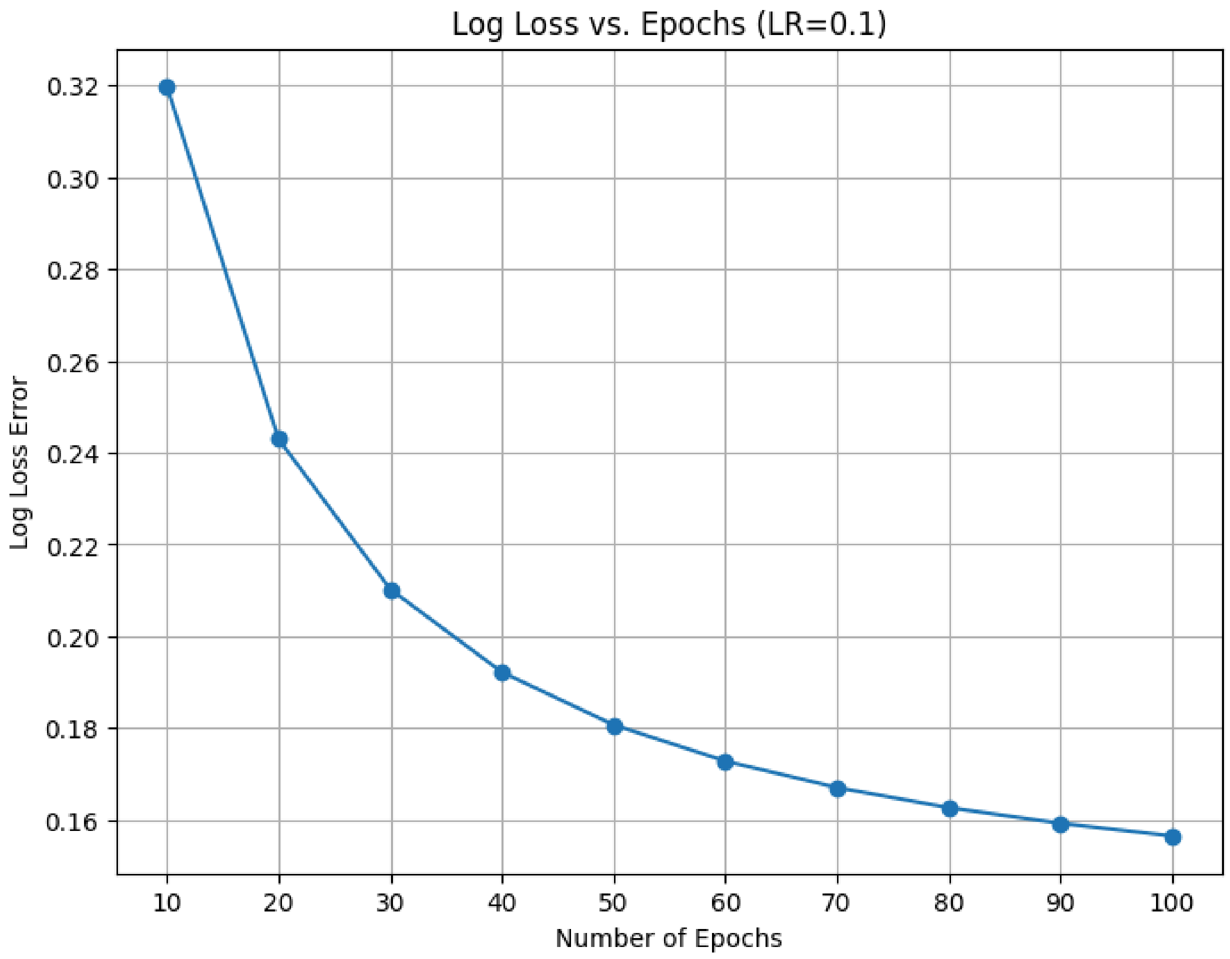

Gradient Descent (LR=0.01, Status: Finished 100 epochs)
Class 0
Class 1
Feature x1
Feature x2
0.0
0.2
0.4
0.6
0.8
1.0

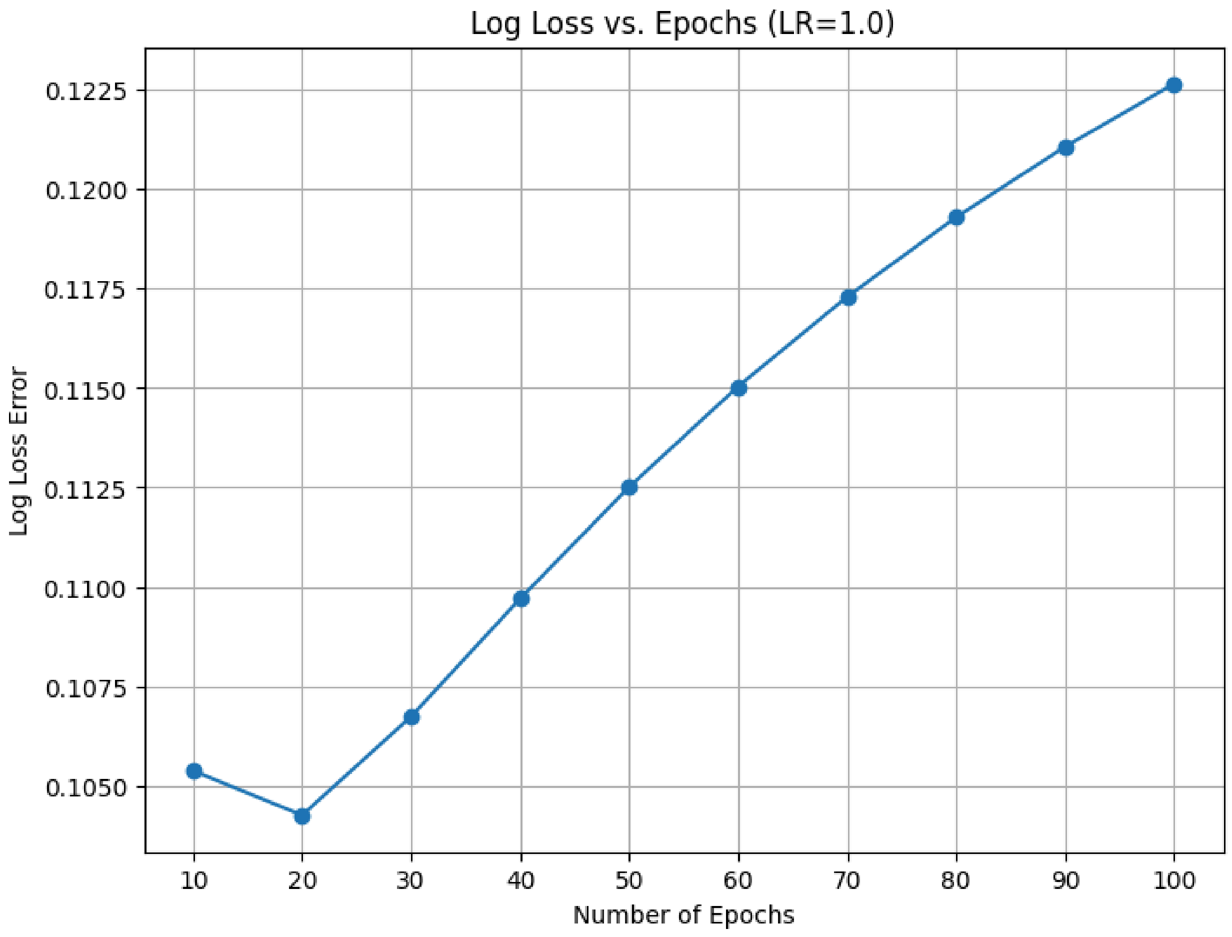

Gradient Descent (LR=0.1, Status: Finished 100 epochs)
Class 0
Class 1
Feature x1
Feature x2
0.0
0.2
0.4
0.6
0.8
1.0



Plots 8-10: Gradient Descent Log Loss (LR=0.01, 0.1, 1.0)


Log Loss vs. Epochs (LR=0.01)
Log Loss Error
Number of Epochs
0.60
0.55
0.50
0.45
0.40
0.35
10
20
30
40
50
60
70
80
90
100


Log Loss vs. Epochs (LR=0.1)
Log Loss Error
Number of Epochs
0.32
0.30
0.28
0.26
0.24
0.22
0.20
0.18
0.16
10
20
30
40
50
60
70
80
90
100



Conclusion

Both Perceptron methods were implemented and tested. The Heuristic approach (Part 1), while simple, failed to converge within the iteration limits for this specific run, highlighting its sensitivity. The Gradient Descent approach with Sigmoid activation (Part 2) successfully learned to classify the data, demonstrating more stable learning dynamics as visualized by the decreasing Log Loss. The learning rate was shown to be a critical parameter influencing the speed and effectiveness of both algorithms. The