Assignment 4: Perceptron Report

Name: Khushi Choudhary
ID: 012624158

1. Introduction

This report details the implementation and analysis of two Perceptron learning algorithms for binary classification, as required by Assignment 4. The objective was to classify data points loaded from `data.csv` using:

- The Heuristic Perceptron algorithm (based on direct error correction).
- The Gradient Descent Perceptron algorithm (using sigmoid activation and log loss minimization).

The report compares these two approaches, analyzes the effect of the learning rate (primarily focusing on the observed rate of 0.1) and the number of iterations/epochs, examines convergence behavior and the resulting decision boundaries based on the provided notebook execution, and discusses observations made during experimentation.

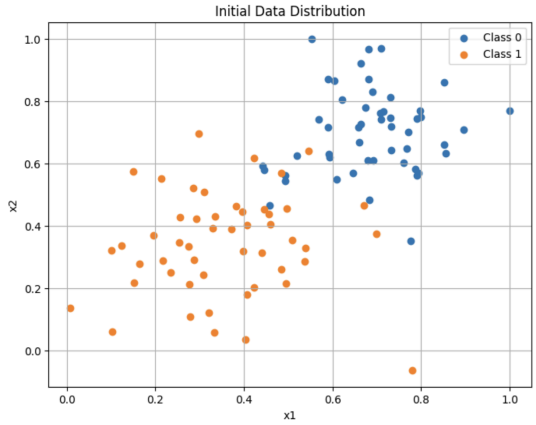
2. Methodology

The implementation was carried out using Python with the NumPy, Pandas, and Matplotlib libraries within a Jupyter Notebook environment.

- Data Handling: The `data.csv` dataset was loaded using Pandas, assuming no header row. Columns were named 'x1', 'x2', and 'label'. Features (X) and labels (y) were extracted. Labels were checked and confirmed/adjusted to be 0 and 1. The initial data distribution was visualized using a scatter plot (See `Assignment4.ipynb`, Cell [26] output).

```
CSV loaded successfully without header.
Data Head:
         x1        x2  label
0  0.78051 -0.063669      1
1  0.28774  0.291390      1
2  0.40714  0.178780      1
3  0.29230  0.421700      1
4  0.50922  0.352560      1

Plotting Initial Data...
```

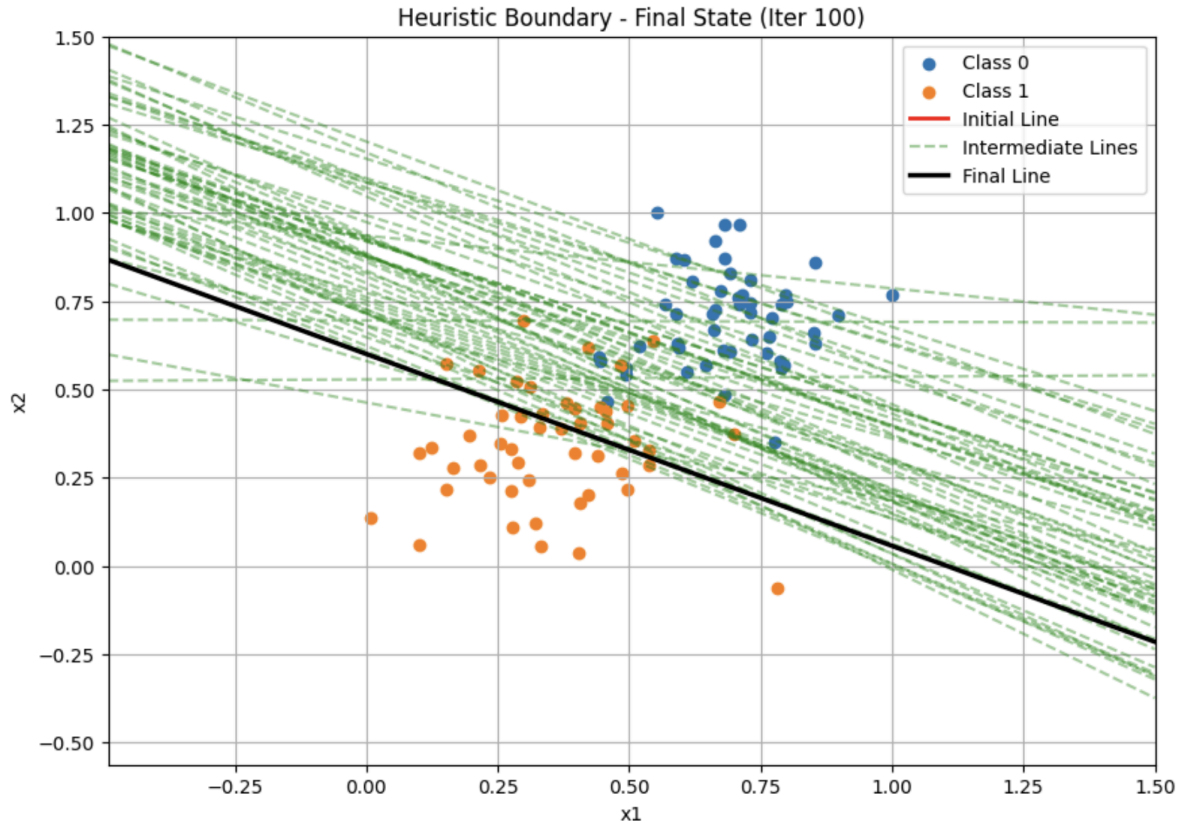


- Heuristic Perceptron: A function `perceptron_heuristic` was implemented (Cell [27]). It processes data point-by-point. A bias term was added by horizontally stacking a column of ones to the input features `X`. For misclassified points (determined using a `sign` activation function), weights (including bias) were updated directly based on the error (`target - prediction`) and the learning rate (0.1). The history of weights after each individual update was stored.

- Gradient Descent Perceptron: A function `perceptron_gradient_descent` was implemented (Cell [30]). It uses the `sigmoid` activation function. A bias term was added similarly. Batch gradient descent was employed: the gradient of the binary cross-entropy (log loss) was calculated over the entire dataset for each epoch. Weights were updated based on this gradient and the learning rate (0.1). The history of weights and the log loss were recorded every 10 epochs.

- Visualization:
  - A function `plot_decision_lines` was used to display the data points along with the initial (red), intermediate (dashed green), and final (black) decision boundaries derived fromthe stored weight history for both algorithms.

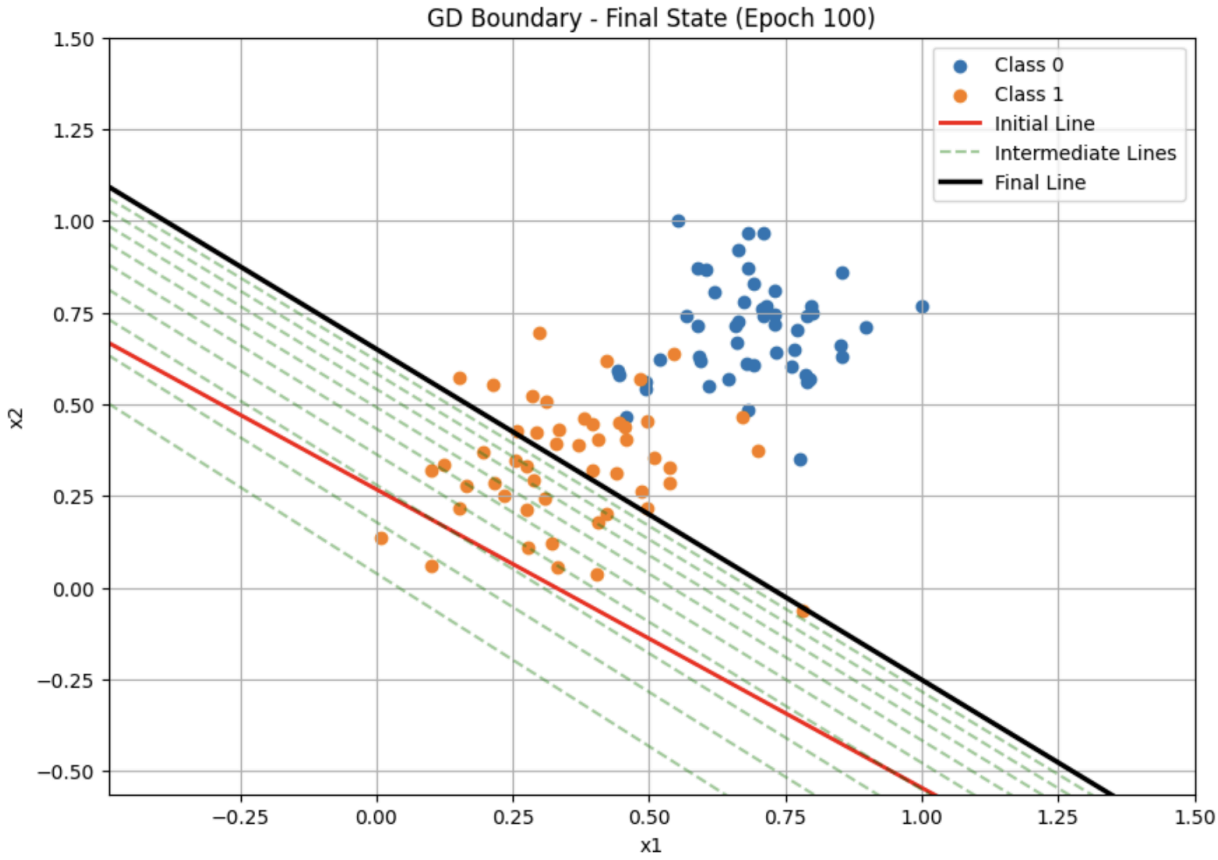
```
Heuristic: Reached max iterations (100).
```



  - A function `plot_loss_curve` was used to display the log loss against the epoch number (at 10-epoch intervals) for the Gradient Descent method.

- Experimentation: Both algorithms were run with a learning rate of 0.1 and 100 iterations/epochs, as shown in the notebook execution.

```
Gradient Descent: Finished 100 epochs. Final Loss: 0.6054
```




```
Gradient Descent training finished.
```

3. Results and Analysis

Comparison: Heuristic vs. Gradient Descent (Based on LR=0.1, Iter/Epochs=100)

- Update Mechanism: The Heuristic method updated weights frequently whenever a point was misclassified within an iteration (evident from the density of green lines in its plot). Gradient Descent calculated an average gradient across all points and updated weights only once per epoch.

- Activation & Output: Heuristic used a hard threshold (`sign` function), while Gradient Descent used the smooth `sigmoid` function, calculating probabilities before thresholding for loss/gradient purposes.

- Convergence:
  - Heuristic: The output indicates "Reached max iterations (100)". This means the algorithm did not achieve perfect separation (zero errors in an epoch) within 100 iterations at LR=0.1. The boundary shifted many times during training.
  - Gradient Descent: The loss curve (Cell [34] output) shows a consistent decrease from ~0.69 down to ~0.60 over 100 epochs. The loss is still decreasing slightly at the end, suggesting it hasn't fully converged

to a minimum but has made significant progress. The boundary evolution plot (Cell [33] output) shows smoother, less frequent adjustments (plotted every 10 epochs) compared to the heuristic plot.

- Sensitivity: While only LR=0.1 was shown, typically Gradient Descent can be more sensitive to learning rate choice – too high can cause divergence (loss increase), too low can cause very slow convergence. The Heuristic method might tolerate a wider range, though high rates can cause oscillations.

- Final Boundary: For LR=0.1, the final black decision boundaries from both methods (Cell [29] and Cell [33] outputs) appear visually similar and provide reasonable, though likely not perfect, separation of the two classes in the dataset. There might be subtle differences due to the different update rules and convergence points.

3.2. Effect of Learning Rates

Based on the single run (LR=0.1) and general knowledge:

- Low Learning Rate (e.g., 0.001, 0.01):
  - Heuristic: Would likely require significantly more than 100 iterations to approach convergence, with very small boundary adjustments per update.
  - Gradient Descent: Convergence would be much slower; the loss curve would decrease very gradually, potentially needing thousands of epochs to reach a similar loss value as achieved with LR=0.1 in 100 epochs.

- Moderate Learning Rate (e.g., 0.1):
  - Heuristic: As observed, it made progress but didn't fully converge in 100 iterations. It appears stable.
  - Gradient Descent: As observed, it showed efficient and stable convergence, with significant loss reduction over 100 epochs.

- High Learning Rate (e.g., 1.0, 5.0):
  - Heuristic: Might cause the decision boundary to oscillate significantly between updates and potentially fail to settle down, even if the data is separable.
  - Gradient Descent: Likely to cause oscillations in the loss curve (overshooting the minimum) or even divergence (loss increasing) due to large, unstable weight updates.

- Optimal Rate: For this dataset and implementation, LR=0.1 seems like a reasonable starting point, providing stable learning for both methods within 100 iterations/epochs, although neither fully converged in that time. Further experimentation would be needed to fine-tune.

3.3. Convergence Behavior and Final Decision Boundaries

- Convergence:
  - Heuristic: Did not converge within 100 iterations at LR=0.1. Convergence is only guaranteed for linearly separable data, and even then, might require many iterations depending on the data and learning

rate. Reaching `max_iter` is common if the data isn't perfectly separable or the learning rate/iteration count is insufficient.

- Gradient Descent: Showed good convergence behavior, steadily minimizing the log loss. It aims to find weights that minimize the overall loss, which often results in a reasonable boundary even for non-separable data. The 100 epochs allowed significant loss reduction, but the curve suggests further small improvements might be possible with more epochs.

- Final Boundaries: The final boundaries achieved by both methods with LR=0.1 were visually similar and represented plausible linear separators for the data. The subtle differences likely stem from the GD method optimizing a global loss function versus the heuristic method reacting to individual errors sequentially.

3.4. Difficulties and Observations

- Implementation: Ensuring the bias term was correctly added (using `np.hstack`) and handled in dot products was crucial. Debugging array shapes during weight updates or gradient calculations is a common task.

- Parameter Tuning: While only one set of parameters was run in the final notebook, finding the optimal `learning_rate` and `max_iter`/`epochs` generally requires experimentation. A poor choice can lead to non-convergence or instability.

- Visualization: The heuristic plot (Cell [29] output) with updates stored after every misclassification resulted in many overlapping green lines, making it slightly difficult to discern the exact path, although it clearly shows the boundary was actively adjusted. Plotting every Nth line helps, as done in the `plot_decision_lines` function. The GD plots (boundary every 10 epochs, loss curve) were clearer for tracking overall progress.

- Algorithm Quirks: The heuristic method's progress depends on the order of data presentation (though not explicitly shuffled here). The GD batch method provides a more stable update based on the entire dataset each epoch. The final loss for GD (~0.60) indicates the data is likely not perfectly linearly separable with a sigmoid activation, as perfect separation would result in a much lower loss.

4. Conclusion

This assignment provided practical experience in implementing and comparing the Heuristic and Gradient Descent approaches for the Perceptron algorithm using a functional programming style.

Both methods successfully learned a linear decision boundary for the dataset using a learning rate of 0.1. The Heuristic method reacted directly to errors, leading to frequent boundary adjustments but didn't fully converge in 100 iterations. The Gradient Descent method showed smooth, stable convergence by minimizing the global log loss, achieving significant loss reduction in 100 epochs, though also not reaching a perfect minimum.

The learning rate is clearly critical. The chosen rate of 0.1 provided reasonable performance for both methods in this instance, but slight variations could significantly alter convergence speed or stability.

Overall, for this dataset, both algorithms yielded similar final decision boundaries. The Gradient Descent approach, coupled with its loss curve, provides a clearer picture of the learning progress and convergence towards an optimum, making it potentially more suitable when fine-tuning or dealing with data that may not be perfectly separable.