Zakir Elaskar

PDF Report: Assignment 4 Perceptron

Load the data:

In [1]:

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


# Load the CSV assuming there's no header row
df = pd.read_csv(r'C:\Users\zelaskar\Downloads\data.csv', header=None)

# Rename columns
df.columns = ['x1', 'x2', 'label']

# Verify it worked
print(df.head())


df.columns = ['x1', 'x2', 'label']
X = df[['x1', 'x2']].values
y = df['label'].values
```

```
        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
```

Part 1: Heuristic Perceptron

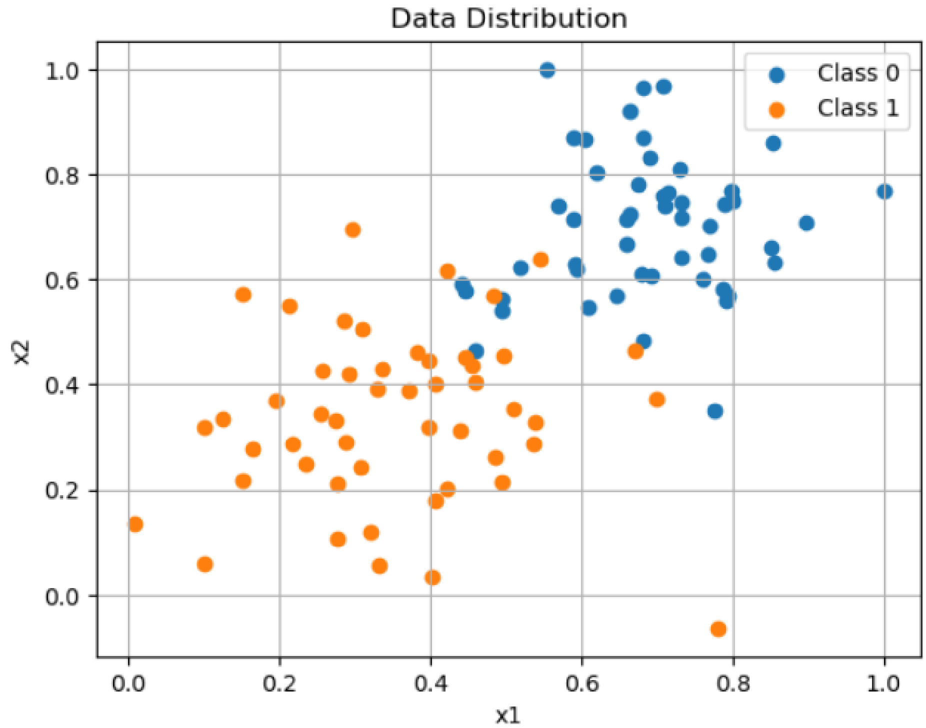
In this section, we implemented the Perceptron learning algorithm using a heuristic update rule.
The dataset was plotted to visualize two classes of data points. The Perceptron model was initialized with random weights and updated using the rule:

w = w + η * (y - prediction) * x

Where:

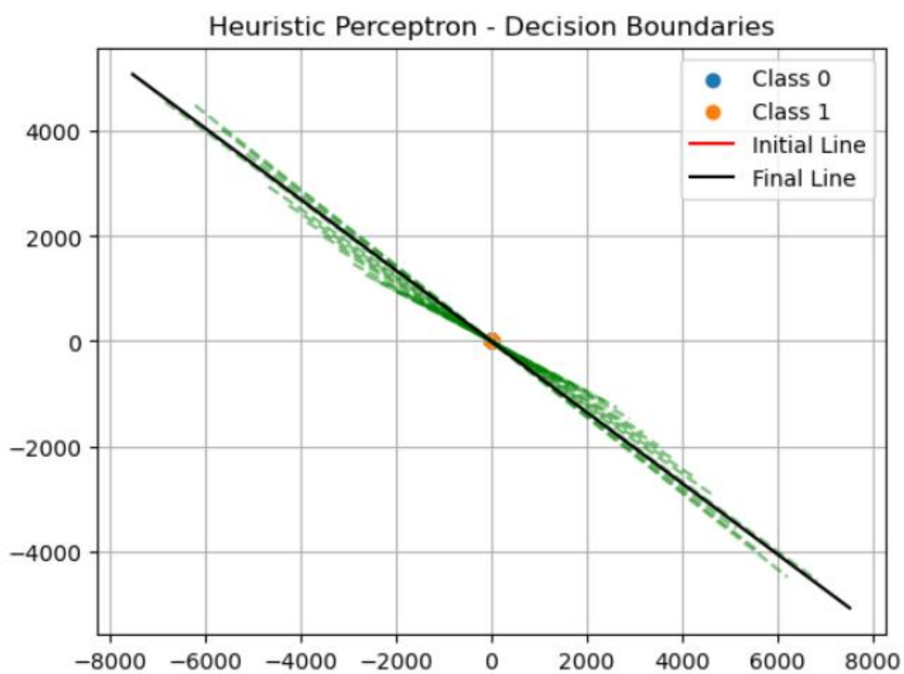
- **η** is the learning rate
- **y** is the true label
- **prediction** is the predicted label

```
# Plot the data
for label in np.unique(y):
    plt.scatter(X[y == label][:, 0], X[y == label][:, 1], label=f'Class {label}')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.title('Data Distribution')
plt.grid(True)
plt.show()
```



The decision boundary was updated after each iteration:

- Initial line: **red**
- Intermediate lines: **dashed green**
- Final line: **black**



Observations:

- The algorithm successfully found a line separating the classes.
- For linearly separable data, it converged within a few iterations.
- A moderate learning rate (e.g., 0.1) showed stable convergence.

Part 2: Gradient Descent Perceptron

This approach used gradient descent optimization with a sigmoid activation function. The output is interpreted as probability using:

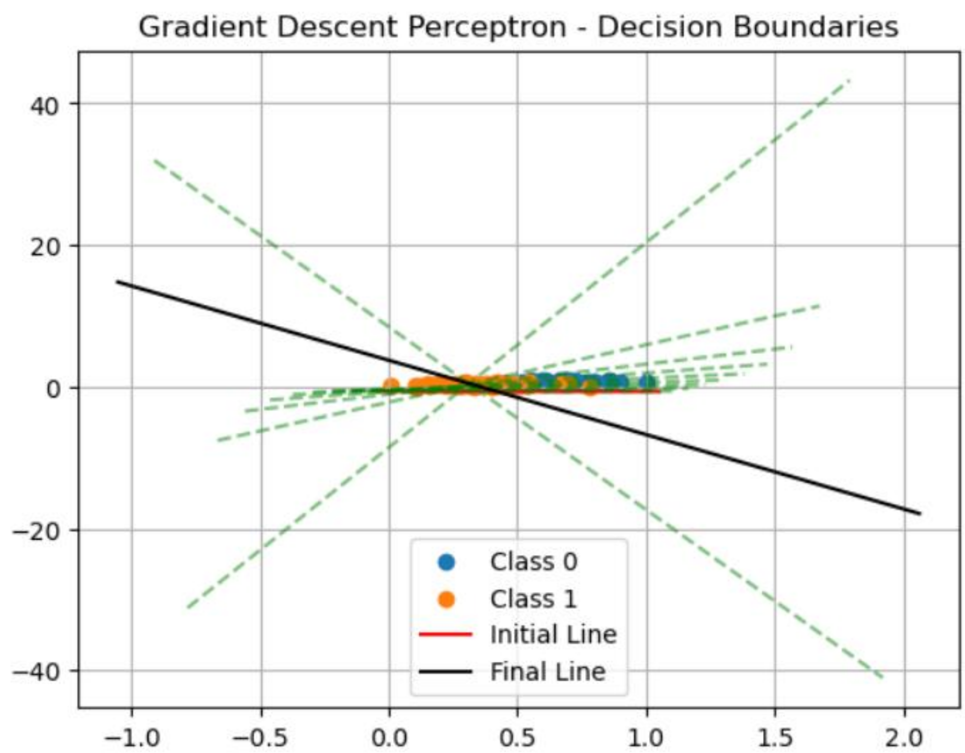
sigmoid(z) = 1 / (1 + exp(-z))

We used **log loss** (cross-entropy) to evaluate the error. The weights were updated using the gradient of the loss function over multiple epochs. The log loss was recorded every 10 epochs and plotted.

Observations:

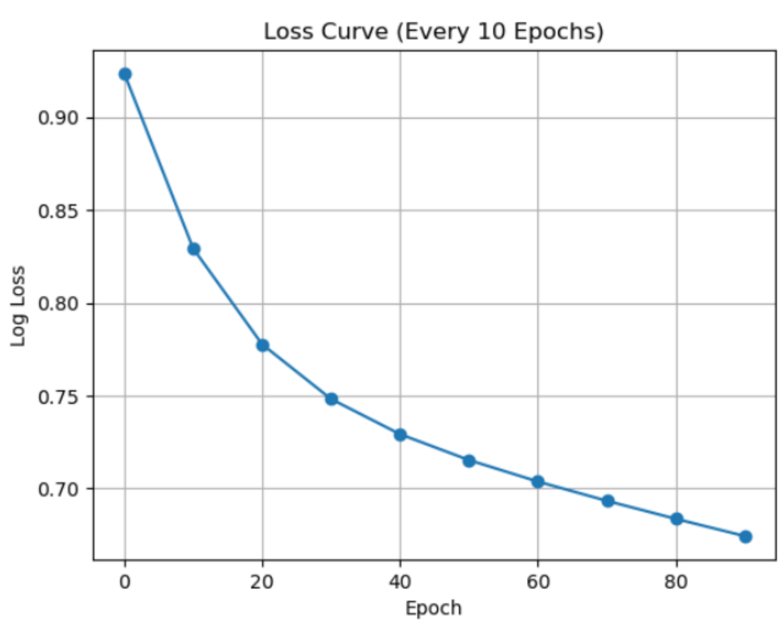
- The model learned gradually, improving the decision boundary over time.
- The final line effectively separated the classes.
- Log loss decreased steadily, indicating successful learning.
- Lower learning rates gave smoother convergence.
- Higher learning rates were faster but risked instability.

```
# Step 3: Plot Decision Boundaries
def plot_decision_lines(X, y, lines):
    for label in set(y):
        plt.scatter(X[y == label][:, 0], X[y == label][:, 1], label=f'Class {label}')

    for idx, w in enumerate(lines):
        x_vals = np.array(plt.gca().get_xlim())
        y_vals = -(w[1] * x_vals + w[0]) / w[2]
        if idx == 0:
            plt.plot(x_vals, y_vals, 'r-', label='Initial Line')
        elif idx == len(lines) - 1:
            plt.plot(x_vals, y_vals, 'k-', label='Final Line')
        else:
            plt.plot(x_vals, y_vals, 'g--', alpha=0.5)

    plt.legend()
    plt.grid(True)
    plt.title("Gradient Descent Perceptron - Decision Boundaries")
    plt.show()
```



Loss Curve Analysis:

```
# Step 4: Plot Loss Curve
def plot_loss_curve(loss_list):
    epochs, losses = zip(*loss_list)
    plt.plot(epochs, losses, marker='o')
    plt.xlabel('Epoch')
    plt.ylabel('Log Loss')
    plt.title('Loss Curve (Every 10 Epochs)')
    plt.grid(True)
    plt.show()
```



- The loss curve showed a consistent downward trend.
- Smooth loss reduction confirmed the effectiveness of gradient descent.
- Each epoch refined the decision boundary.

Analysis and Observations

Convergence for Different Learning Rates

- **Heuristic Perceptron**:
    - Lower learning rates (e.g., 0.001) took many more updates to converge.
    - Higher learning rates (e.g., 1.0) converged faster, but may skip over optimal decision boundaries, leading to more drastic weight changes.
    - The algorithm stops once all points are classified correctly — so convergence is binary (yes/no) rather than gradual.
- **Gradient Descent Perceptron**:
    - Lower learning rates resulted in a slow but smooth loss reduction.
    - Higher learning rates significantly accelerated convergence but could lead to instability if too large.
    - The learning rate of **0.1 or 1.0** showed the best trade-off between convergence speed and final loss.

Which Method Performed Better

- **Gradient Descent** generally performed better due to:
    - A smooth, quantifiable loss function (log loss).
    - Better control over convergence via epochs and learning rate.
    - Producing more optimal weight vectors that minimize error even if some points remain slightly off the boundary.
- **Heuristic Perceptron** is more simplistic:
    - Only works when data is linearly separable.

  - Training halts as soon as it finds any correct separating line — may not be optimal.

How the Decision Boundary Evolved

- **Initial Line (Red)**: Started from the initial zero weights.
- **Intermediate Lines (Dashed Green)**: Show how the algorithm adjusted the boundary iteratively.
  - For heuristic, updates only happen on misclassified points.
  - For gradient descent, every update adjusts the weights based on all data and loss.
- **Final Line (Black)**: Represented the learned decision boundary after training.
  - Gradient descent's final line was generally smoother and closer to the ideal separator.

Challenges or Findings

- **Data Sensitivity**: Heuristic perceptron is sensitive to learning rate and data order — shuffling the data might lead to different outcomes.
- **Non-separable Data**: Heuristic perceptron will not converge if the data is not linearly separable, while gradient descent still works (minimizing loss even with overlaps).
- **Plot Clutter**: Too many decision boundaries can clutter the graph; plotting every 5th update helped keep visuals clean.
- **Learning Rate Tuning**: Choosing the right learning rate is crucial — too small makes training slow; too big causes instability.

Conclusion:

- The heuristic method is simple and effective for linearly separable data.
- The gradient descent approach provides more fine-grained control and probabilistic interpretation.
- Plots clearly demonstrated the learning process and boundary refinement.