MiniProject: Introduction To Machine Learning

Cindrella SK
ee23s061

Rounit Agrawal
ns24z147

Venkatesh H
ns24z029

Varmija CD
ns24z041

Sunday 13th October, 2024

1 Introduction

This report outlines the use of two machine-learning models applied to a song dataset. The goal is to leverage various techniques to analyze song features and make predictions related to song energy and danceability. The project is divided into the following parts:

- **Part 1:** Linear Regression for predicting song energy based on audio features.
- **Part 2:** Logistic Regression to classify songs as danceable or not based on song attributes.
- **Part 3:** Implementation of the Perceptron algorithm and Multi-Layer Perceptron (MLP) for both linearly separable and non-linearly separable datasets.

2 Part 1: Linear Regression

2.1 Problem Statement

The objective of this part is to predict the energy level of songs based on features like `danceability`, `loudness`, and `tempo`. Linear regression is employed to estimate the energy based on these continuous variables.

2.2 Theory

Linear regression models the relationship between a dependent variable (song energy) and one or more independent variables (audio features). The goal is to find the best-fit line by minimizing the sum of squared residuals, represented as:

$$\hat{y} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n$$

where $\hat{y}$ is the predicted energy, and $X_1, X_2, \ldots X_n$ are the features.

2.3 Dataset Overview

The dataset consists of the following columns:

- **song_name**: Name of the song.
- **song_popularity**: Popularity score of the song.
- **song_duration_ms**: Duration of the song in milliseconds.
- **acousticness**: Confidence measure from 0 to 1 of whether the track is acoustic.
- **danceability**: Measure of how suitable a track is for dancing.
- **energy**: Perceptual measure of intensity and activity.
- **instrumentalness**: Predicts whether a track contains no vocals.
- **key**: The key the track is in.
- **liveness**: Detects the presence of an audience in the recording.
- **loudness**: Overall loudness of a track in decibels.
- **audio_mode**: Modality of the track (major or minor).
- **speechiness**: Presence of spoken words in a track.
- **tempo**: Speed or pace of the track.
- **time_signature**: Overall time signature of the track.
- **audio_valence**: Measure from 0 to 1 describing the musical positiveness of a track.

2.4 Data Cleaning and preprocessing

Before training the model, it is essential to clean and preprocess the data. The following steps were performed:

1. **Missing Values Check**: Checked for missing values using `data.isnull().sum()`. No missing values were found in the dataset, so no imputation was required.
2. **Outliers Detection**: Visualized data distributions using histograms and box plots to identify potential outliers, particularly in numerical columns like `song_duration_ms` and `loudness`. Outliers were noted but not removed initially to maintain the integrity of the dataset unless they significantly impacted model performance.

3. **Feature Standardization**: Used `StandardScaler` to standardize features, ensuring that each variable contributes equally to the model training.

2.5 Data Preprocessing

2.5.1 Correlation Analysis

A correlation matrix was computed and visualized using a heatmap to identify the relationships between features:

```
correlation_matrix = data.corr()
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm',
    linewidths=0.5)
plt.title('Correlation Heatmap of Features')
plt.xlabel('Features')
plt.ylabel('Features')
plt.show()
```

This heatmap helps identify which features have strong relationships with `energy`, aiding in feature selection.

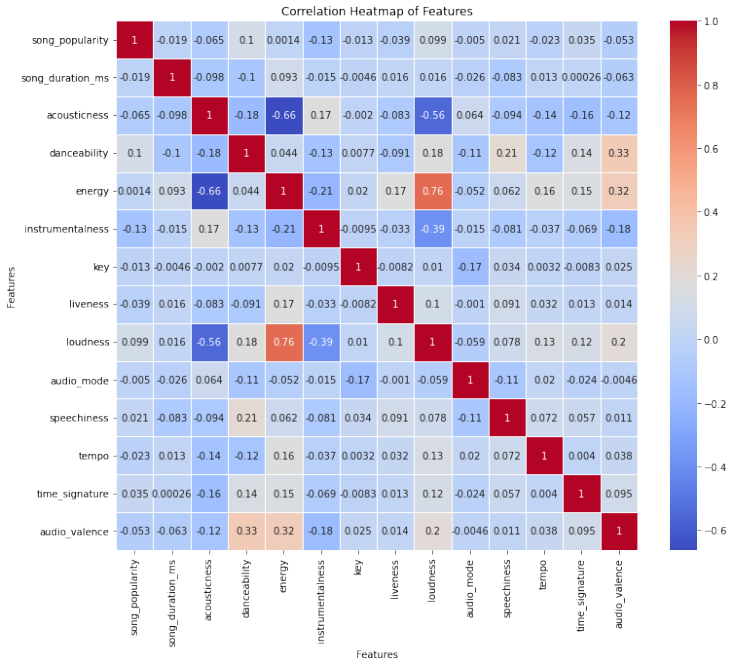

Figure 1: Correlation Heatmap of Features

2.5.2 Top Correlated Features

Selected the top 5 features that had the highest positive or negative correlation with `energy`:

- `loudness` (strong positive correlation)
- `danceability` (positive correlation)
- `acousticness` (negative correlation)
- `tempo` (moderate positive correlation)
- `instrumentalness` (negative correlation)

2.6 Model Building and Evaluation

The `LinearRegression()` model from `sklearn` was used to train on the song dataset. The following code snippet illustrates how the model was trained:

```
model = LinearRegression()
model.fit(X_train, y_train)
```

The model was evaluated using Mean Squared Error (MSE) and R-squared (R^2) metrics, which provide insights into how well the model fits the data. The predicted vs. actual plot shows the relationship between the model's predictions and the actual energy values.

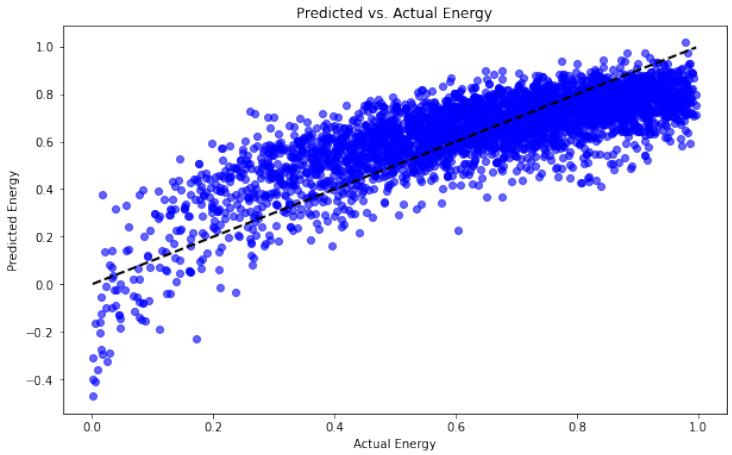

Figure 2: Predicted vs. Actual Energy Plot

3 Part 2: Logistic Regression

3.1 Problem Statement

In this part, we aim to classify songs as either danceable or not based on various features such as `loudness`, `acousticness`, and `tempo`. We define a song as danceable if its `danceability` score is greater than or equal to 0.5.

To achieve this, we employ logistic regression, a commonly used classification technique for binary outcomes. We will also experiment with different feature scaling methods (No Scaling, Standard Scaling, and Min-Max Scaling) and evaluate the model performance using various metrics such as accuracy, confusion matrix, ROC curve, and precision-recall curve.

3.2 Theory

Logistic regression is a linear model for binary classification. It estimates the probability that a sample belongs to a specific class (in this case, danceable or not). The logistic function (or sigmoid function) is used to map predicted values to probabilities, expressed as:

$$P(y = 1|X) = \frac{1}{1 + e^{-(\beta_0 + X\beta)}}$$

where:

- $P(y = 1|X)$ is the probability that a song is classified as danceable.
- β_0 is the intercept, and $X\beta$ represents the weighted sum of the input features.

3.3 Data Preprocessing

Before applying logistic regression, the dataset was preprocessed to handle missing values, drop unnecessary columns (e.g., `song_name`), and generate the binary target variable `danceable`. The following transformations were made:

- The `danceable` column was created based on the `danceability` feature (1 if `danceability` 0.5, otherwise 0).
- Non-numeric and irrelevant columns were dropped.
- Data was split into training (80%) and testing (20%) sets for model evaluation.

```
# Create the target variable 'danceable'
data['danceable'] = data['danceability'].apply(lambda x: 1
    if x >= 0.5 else 0)
X = data.drop(columns=['song_name', 'danceability', '
    danceable'])
y = data['danceable']
```

The distribution of numeric variables and their correlations were explored using a heatmap, helping us understand the relationships between the features.

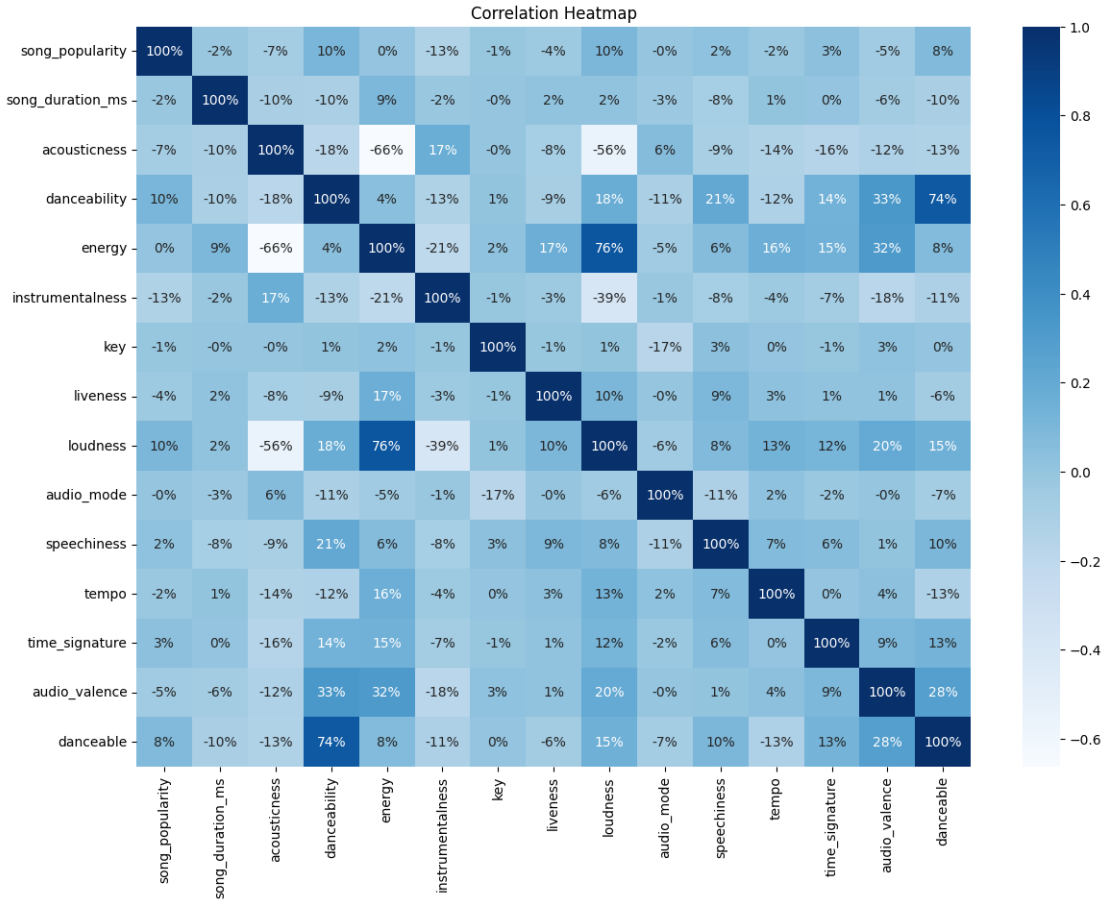

Figure 3: Correlation Heatmap of Features

3.4 Scaling Techniques

Three different scaling methods were used to preprocess the feature data before training the logistic regression model:

- **No Scaling:** The model is trained without any scaling of the input features.
- **Standard Scaling:** This technique standardizes the data by removing the mean and scaling it to unit variance.

- **Min-Max Scaling:** This method scales each feature to a given range, typically between 0 and 1.

The following code snippets show how the data was scaled:

```
# Standard Scaling
std_scaler = StandardScaler()
X_std_scaled = std_scaler.fit_transform(X)

# Min-Max Scaling
min_max_scaler = MinMaxScaler()
X_mm_scaled = min_max_scaler.fit_transform(X)
```

3.5 Model Training and Evaluation

Baseline (No Scaling): The logistic regression model was trained and evaluated using the unscaled features. The accuracy, confusion matrix, and ROC curve for the baseline model are provided below:

```
log_reg_model.fit(X_train, y_train)
accuracy = accuracy_score(y_test, y_pred)
```

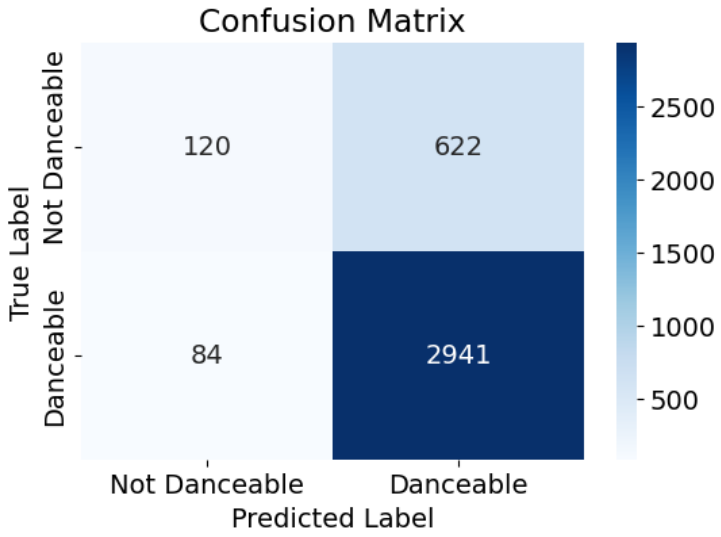

Figure 4: Confusion Matrix for Baseline Logistic Regression

Standard Scaling: After applying standard scaling, we retrained the model and observed improvements in both accuracy and the area under the ROC curve (AUC).

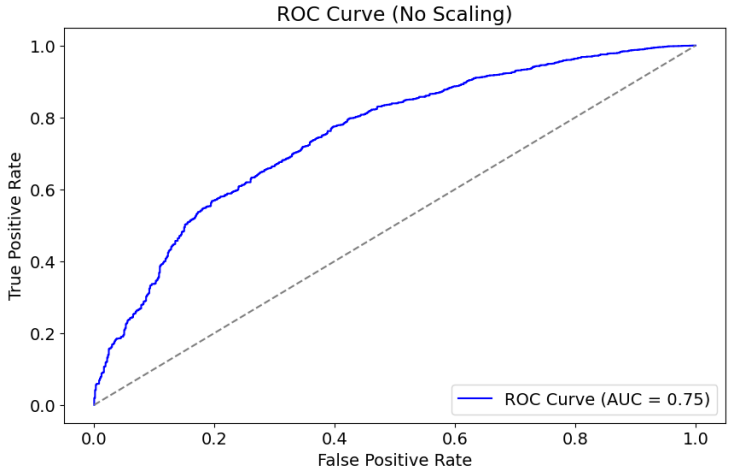

Figure 5: ROC Curve for Baseline Logistic Regression

Min-Max Scaling: The model was also trained using Min-Max scaling. The accuracy and AUC from this approach were comparable to the standard scaling method, as shown in the ROC curve in Figure 7.

3.6 Precision-Recall Curve

In addition to the ROC curve, we also plotted the precision-recall curve to evaluate the trade-off between precision and recall across different thresholds. The precision-recall curve for the Standard Scaled model in Figure 8.

3.7 Model Comparison

Finally, we compared the accuracies of the models trained using different scaling techniques and the overall accuracy and other metrics are compared to find the best scaling approach. Accuracy Comparison: No Scaling: 81.26%, Min-Max Scaling: 82.11% and Standard Scaling: 82.21%. summarizes the performance of the baseline, standard scaling, and Min-Max scaling models.

4 Part 3: Perceptron Algorithm

4.1 Overview of the Perceptron Algorithm

The Perceptron algorithm is used for binary classification when the data is linearly separable. It iteratively adjusts a weight vector w to correctly classify training samples. The goal is to find a weight vector w such that for all training points (x_i, y_i), the inequality $y_i \langle w, x_i \rangle > 0$ holds, where:

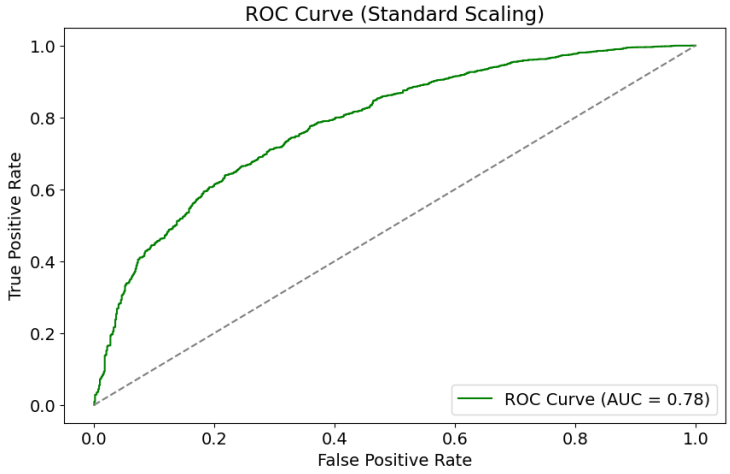

Figure 6: ROC Curve for Logistic Regression with Standard Scaling

- y_i is the label of the i-th example, which is either 1 or -1.
- x_i is the feature vector for the i-th example.
- $\langle w, x_i \rangle$ is the dot product between w and x_i.

4.1.1 Working of the Perceptron Algorithm

- **Initialization:** Start with an initial weight vector $w^{(1)} = (0, \ldots, 0)$.
- **Iteration:** At each iteration t, the algorithm checks if there exists an example (x_i, y_i) such that $y_i \langle w^{(t)}, x_i \rangle \leq 0$. If such a misclassified example is found, the weight vector is updated as:

$$w^{(t+1)} = w^{(t)} + y_i x_i$$

  This update increases the value of $y_i \langle w^{(t+1)}, x_i \rangle$, effectively moving the decision boundary to classify the i-th example correctly.
- **Stopping Condition:** The process repeats until there are no misclassified examples, meaning the algorithm has found a weight vector that correctly classifies all training points.

4.1.2 Theorem 9.1 Explanation

Theorem 9.1 states that if the dataset $(x_1, y_1), \ldots, (x_m, y_m)$ is linearly separable, the Perceptron algorithm will stop after at most $\left(\frac{R}{B}\right)^2$ iterations, where:

- $R = \max_i \|x_i\|$ is the maximum norm of the feature vectors.
- B is the minimum norm of the weight vector that can separate the data, such that for all i, $y_i \langle w, x_i \rangle \geq 1$.

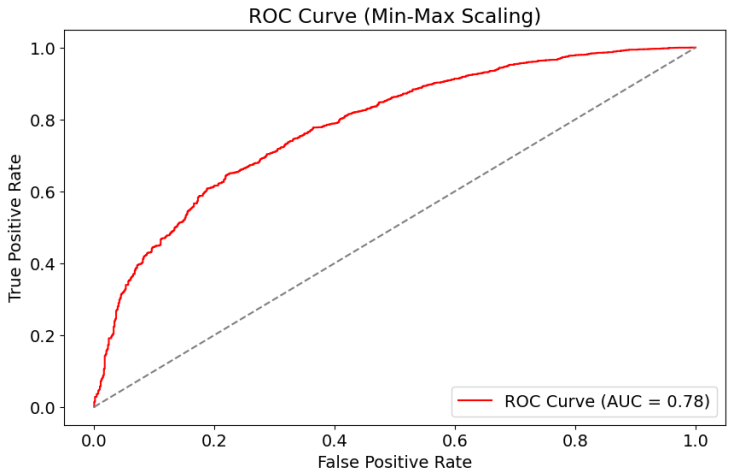

Figure 7: ROC Curve for Logistic Regression with Min-Max Scaling

4.1.3 Proof Intuition

The proof for the upper bound on iterations uses the following ideas:

- **Progress Towards the Optimal Solution:** Let w^* be the optimal weight vector. At each step, the updated weight vector $w^{(t+1)}$ becomes closer to w^*, as each update adds a correction $y_i x_i$ that moves towards the correct classification.
- **Dot Product Increase:** After each update, the dot product between $w^{(t+1)}$ and w^* increases, making the solution more correct.
- **Bounding the Norm of $w^{(t)}$:** The norm of $w^{(t)}$ increases with each update, but the number of iterations before convergence is bounded by $\left(\frac{R}{B}\right)^2$.

4.1.4 Intuition Behind the Update Rule

Each update moves the decision boundary to classify a previously misclassified point correctly. The squared norm $\|x_i\|^2$ quantifies how much each misclassification impacts the update, and the upper bound is derived from summing these effects.

4.2 Code Explanation

The following code iterates over the dataset to update the weights and bias of the Perceptron model. The logic behind the update rule and time complexity is detailed below.

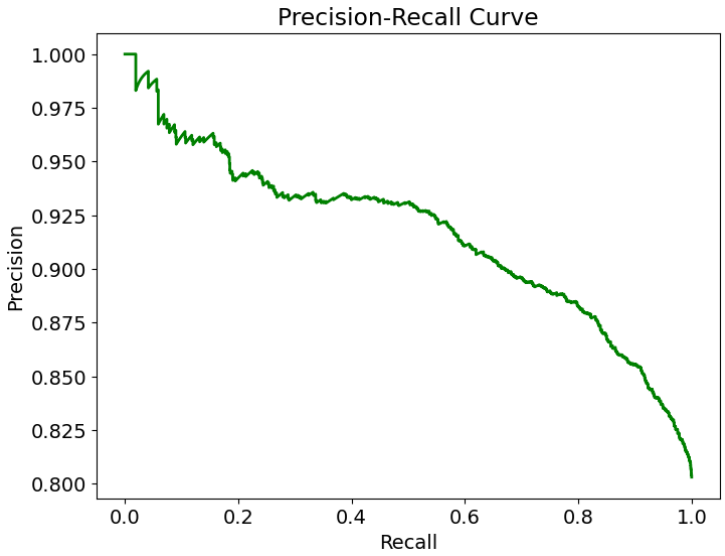

Figure 8: Precision-Recall Curve for Logistic Regression with Standard Scaling

```
for xi, yi in zip(X, y):
    if yi * (np.dot(weights, xi) + bias) <= 0:
        weights += learning_rate * yi * xi
        bias += learning_rate * yi
        error_count += 1
```

4.2.1 Explanation of the Code

- **Looping through data:** `zip(X, y)` combines the feature matrix X and the label vector y. Each iteration corresponds to one training example (x_i, y_i).
- **Condition:** The conditional checks if a sample is misclassified by the current weight vector w and bias b. If $y_i \left(\langle w, x_i \rangle + b\right) \leq 0$, the point is misclassified.
- **Weight update:** The update rule adjusts the weight vector by adding $\eta y_i x_i$, where η is the learning rate.
- **Bias update:** Similarly, the bias is adjusted by adding ηy_i, where y_i is the target label.
- **Error count:** If any misclassification is detected, the `error_count` is incremented, indicating that the algorithm needs to continue iterating.

We proof the the results from the theory and practical is similar as seen from the figure below

```
Did not converge within 12 iterations
Number of iterations: 11
Theoretical upper bound: 12.80716941666753
```

Figure 9: Theoretical vs practical results

4.3 Time Complexity Analysis

4.3.1 Iterating Over Examples

The outer loop iterates through all examples (x_i, y_i) in the training set. Let X be the feature matrix containing m examples, each with n features. Thus, iterating over X takes:

$$O(m)$$

where m is the number of training samples.

4.3.2 Dot Product Calculation

Inside the loop, the dot product `np.dot(weights, xi)` is computed between the weight vector and the feature vector x_i. Since x_i has n features, this operation takes:

$$O(n)$$

This calculation is repeated for each of the m samples, so the total time complexity for dot product calculations is:

$$O(m \cdot n)$$

4.3.3 Conditional Check

The condition $y_i(\langle w, x_i \rangle + b) \leq 0$ takes $O(1)$ time for each example. Therefore, checking for all m samples results in:

$$O(m)$$

4.3.4 Weight and Bias Update

When a misclassified example is found, the weight update $w+ = \eta y_i x_i$ takes $O(n)$ time. Similarly, updating the bias takes $O(1)$. Thus, for all m examples, the complexity is:

$$O(m \cdot n)$$

4.3.5 Overall Time Complexity for a Single Iteration

Combining all the steps, a single iteration over all examples has a time complexity of:

$$O(m \cdot n)$$

4.3.6 Total Time Complexity of the Perceptron Algorithm

The Perceptron algorithm iterates until all examples are correctly classified. In the worst-case scenario, the number of iterations T is bounded by $\left(\frac{R}{B}\right)^2$, where R is the maximum norm of the input vectors and B is the minimum norm of the separating hyperplane. Therefore, the total time complexity is:

$$O\left(\left(\frac{R}{B}\right)^2 \cdot m \cdot n\right)$$

4.3.7 Explanation of Parameters

- m: Number of training samples.
- n: Number of features in each sample.
- R: Maximum norm of the input vectors x_i.
- B: Minimum norm of the optimal weight vector that separates the data.

4.3.8 Practical Implications

The $\left(\frac{R}{B}\right)^2$ bound implies that the Perceptron is more efficient when R (the maximum input vector norm) and B (the norm of the optimal solution) are small. In practice, if the data is not well-separated, the Perceptron may require many iterations, leading to higher computational cost, especially for high-dimensional data.

5 3×1 MUX for Boolean Functions Using Perceptron and MLP

5.1 Problem Overview

The task is to implement a 3×1 Multiplexer (MUX) for a Boolean function using three variables. This problem involves two steps:

1. Implementing basic logic gates (AND, OR, NOT) using a single-layer Perceptron and combining them to create an XOR gate. The XOR function for three variables A, B, and C is defined as:

$$\text{XOR}(A, B, C) = A'B + AB' + B'C + BC'$$

2. Using a Multi-Layer Perceptron (MLP) to model the XOR function and plotting the decision boundary.

5.2 Logic Gate Implementation Using Perceptron

In this step, we implement basic logic gates such as AND, OR, and NOT using single-layer Perceptrons. These gates are then combined to form the XOR gate.

```
import torch
import torch.nn as nn

# Define the Perceptron class for logic gates
class Perceptron(nn.Module):
    def __init__(self):
        super(Perceptron, self).__init__()
        self.linear = nn.Linear(2, 1)

    def forward(self, x):
        return torch.sigmoid(self.linear(x))

# AND gate
and_gate = Perceptron()
and_gate.linear.weight = nn.Parameter(torch.tensor([[1.0,
    1.0]]))
and_gate.linear.bias = nn.Parameter(torch.tensor([-1.5]))

# OR gate
or_gate = Perceptron()
or_gate.linear.weight = nn.Parameter(torch.tensor([[1.0,
    1.0]]))
or_gate.linear.bias = nn.Parameter(torch.tensor([-0.5]))

# NOT gate function
def not_gate(x):
    return 1 - x
```

Here, the weights and biases are manually set for the AND and OR gates based on their truth tables. The NOT gate is implemented as a function.

5.2.1 XOR Gate Using AND, OR, and NOT

The XOR gate is created by combining the outputs of the AND, OR, and NOT gates. The logic for XOR is based on the Boolean equation:

$$\text{XOR}(A, B, C) = A'B + AB' + B'C + BC'$$

```
def xor_gate(A, B, C):
    A_not = not_gate(A)
    B_not = not_gate(B)
```

```
    C_not = not_gate(C)

    term1 = and_gate(torch.tensor([A_not, B]))  # A'B
    term2 = and_gate(torch.tensor([A, B_not])) # AB'
    term3 = and_gate(torch.tensor([B_not, C])) # B'C
    term4 = and_gate(torch.tensor([B, C_not])) # BC'

    return or_gate(torch.tensor([term1.item(), term2.item(),
        term3.item(), term4.item()]))
```

The XOR gate for three inputs is computed by combining four terms from the AND gate outputs, representing different combinations of A, B, and C.

5.3 MLP for 3-Input XOR Function

To handle the XOR function using MLP, we create a neural network model with a hidden layer. The MLP can model more complex patterns and learn the non-linearity of XOR.

```
# Define an MLP class for XOR function
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.hidden = nn.Linear(3, 4)  # Hidden layer with 4
            neurons
        self.output = nn.Linear(4, 1)  # Output layer

    def forward(self, x):
        x = torch.sigmoid(self.hidden(x))
        x = torch.sigmoid(self.output(x))
        return x
```

5.3.1 Training the MLP

We train the MLP on the XOR truth table with three variables. The loss is computed using binary cross-entropy loss.

```
# XOR truth table for 3 inputs
X_train = torch.tensor([
    [0, 0, 0],
    [0, 0, 1],
    [0, 1, 0],
    [0, 1, 1],
    [1, 0, 0],
    [1, 0, 1],
    [1, 1, 0],
    [1, 1, 1]
], dtype=torch.float32)
```

```
y_train = torch.tensor([0, 1, 1, 0, 1, 0, 0, 1], dtype=torch
    .float32).view(-1, 1)

# Model, loss function, optimizer
model = MLP()
criterion = nn.BCELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

# Training loop
num_epochs = 10000
for epoch in range(num_epochs):
    optimizer.zero_grad()
    output = model(X_train)
    loss = criterion(output, y_train)
    loss.backward()
    optimizer.step()
```

Here the loss results after training perceptron model.

```
Epoch 0, Loss: 0.7098304629325867
Epoch 1000, Loss: 0.6931518316268921
Epoch 2000, Loss: 0.693148136138916
Epoch 3000, Loss: 0.6931453943252563
Epoch 4000, Loss: 0.6931434869766235
Epoch 5000, Loss: 0.6931416392326355
Epoch 6000, Loss: 0.6931400895118713
Epoch 7000, Loss: 0.6931384801864624
Epoch 8000, Loss: 0.6931368112564087
Epoch 9000, Loss: 0.6931349635124207
```

Figure 10: Theoretical vs practical results

5.3.2 Plotting the Decision Boundary

After training, we plot the decision boundary for the 3-input XOR gate using the trained MLP model.

```
# Plot decision boundary
import numpy as np
import matplotlib.pyplot as plt

x1 = np.linspace(0, 1, 100)
x2 = np.linspace(0, 1, 100)
x3 = np.linspace(0, 1, 100)
xx, yy, zz = np.meshgrid(x1, x2, x3)
```

```
grid = torch.tensor(np.c_[xx.ravel(), yy.ravel(), zz.ravel()
    ], dtype=torch.float32)

# Predictions for the grid points
with torch.no_grad():
    predictions = model(grid).numpy().reshape(xx.shape)

# Visualize decision boundary
plt.contourf(xx[:, :, 0], yy[:, :, 0], predictions[:, :, 0],
     levels=[0, 0.5, 1], cmap="coolwarm", alpha=0.6)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train.squeeze
    (), cmap='coolwarm', edgecolors='k')
plt.xlabel('X1')
plt.ylabel('X2')
plt.title('Decision Boundary for 3-Input XOR')
plt.show()
```

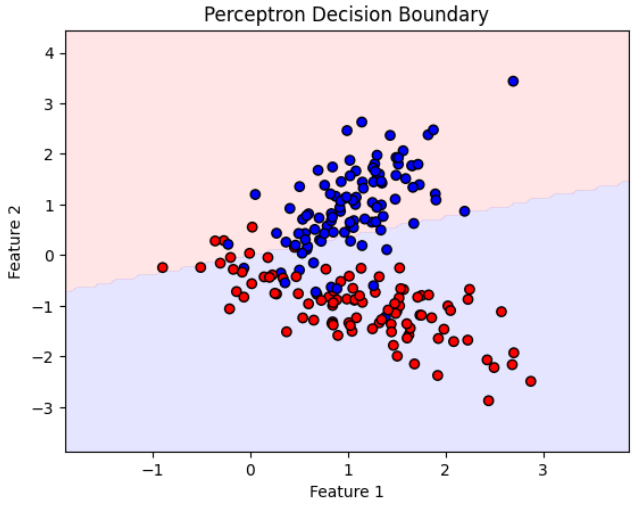

Figure 11: Decision Boundary for 3-Input XOR Using MLP

5.4 Complexity Analysis

- **Perceptron-based XOR:** The XOR function is manually constructed using AND, OR, and NOT gates. Each logic gate requires a single perceptron operation ($O(1)$), making the approach computationally efficient but not scalable to other functions.
- **MLP-based XOR:** The MLP approach generalizes well to complex nonlinear functions like XOR. The time complexity depends on the number of layers, neurons, and epochs. Each forward and backward pass is $O(m \cdot$

$n \cdot E$), where m is the number of samples, n is the number of neurons, and E is the number of epochs.