NATIONAL UNIVERSITY OF MODERN LANGUAGES ISLAMABAD

MACHINE LEARNING

Final Project

Submitted to
Ms. Qurat Raja

Submitted By
Syed Qasim Ali, Junaid Asif & Adeel Naeem
(BSAI-161), (BSAI-144) & (BSAI-146)

Submission Date: December 18th, 2024

Analysis of Binary Detection Techniques

Dataset: TORGO dysarthric speech dataset

This dataset contains speech samples from individuals with dysarthria, a motor speech disorder caused by neurological conditions. Dysarthria affects speech intelligibility, making its detection a challenging yet essential problem in speech recognition systems.

Objective:

The goal of this project is to analyze the performance of multiple binary detection techniques on the TORGO dysarthric speech dataset. Students will implement and compare both traditional and advanced detection algorithms to evaluate their accuracy and effectiveness.

Traditional Techniques

1. **CNN (Convolutional Neural Network):**

    - **Goal:** Classify audio features (MFCCs or Mel Spectrograms) into dysarthric or non-dysarthric.
    - **Steps:**
        - Extract **MFCCs** as input features.
        - Build a **SimpleCNN** model using nn.Conv1d layers.
        - Train and evaluate the model.
    - **Evaluation:** Report accuracy, precision, recall, F1-score, and confusion matrix.

2. **RNN (Recurrent Neural Network):**

    - **Goal:** Use sequential learning to capture temporal dependencies in audio data.
    - **Steps:**
        - Extract **MFCCs** or sequential features.
        - Use an nn.RNN model with two layers.Train and evaluate the model using Cross-Entropy Loss.
    - **Evaluation:** Similar metrics as CNN.

3. **LSTM (Long Short-Term Memory):**

    - **Goal:** Capture long-term dependencies in the audio data.
    - **Steps:**
        - Replace the RNN with an nn.LSTM.
        - Use the final hidden state for classification.
    - **Hybrid CNN-LSTM:**
      Use a CNN for feature extraction and an LSTM for sequence learning.

Advanced Techniques

1. **Contrastive Learning:**
   - **Goal:** Learn embeddings where positive pairs (similar samples) are close and negative pairs are far apart.
   - **Steps:**
     - Generate positive and negative pairs from the dataset.
     - Use a contrastive loss function (e.g., NT-Xent or Triplet Loss).
     - Train a simple encoder network to map audio features to embeddings.
   - **Tools:** Use PyTorch for loss and pair generation.

2. **Knowledge Graphs:**
   - **Goal:** Represent relationships between speakers and their conditions as a graph.
   - **Steps:**
     - Create a graph with nodes (speakers) and edges (similarity relationships).
     - Use Graph Neural Networks (e.g., GCNConv from PyTorch Geometric).
   - **Tools:** PyTorch Geometric.

3. **Transformer-Based Models (e.g., Wav2Vec2):**
   - **Goal:** Use pre-trained speech models like Wav2Vec2 for feature extraction and fine-tuning.
   - **Steps:**
     - Load a pretrained Wav2Vec2 model using HuggingFace Transformers.
     - Fine-tune it on the TORGO dataset for binary classification.
   - **Tools:** HuggingFace Transformers Library.

4. **Self-Supervised Learning (HuBERT, BYOL):**
   - **Goal:** Use HuBERT or BYOL models for unsupervised feature learning.
   - **Steps:**
     - Use a pretrained HuBERT model to extract embeddings.
     - Train a lightweight classification head (fully connected layers).
   - **Tools:** HuggingFace for HuBERT.

5. **Graph Neural Networks (e.g., TKMGNN):**
   - **Goal:** Use GNNs to model complex relationships in the data.
   - **Steps:**
     - Use PyTorch Geometric to create a GNN architecture.
     - Define a graph where nodes represent audio samples and edges represent similarity.
   - **Tools:** PyTorch Geometric.

Methodology of Detection Techniques

1. CNN (Convolutional Neural Network):

Objective:
A CNN is used to extract meaningful patterns and spatial features from the audio feature representations (e.g., MFCCs or Mel Spectrograms) to classify speech as Dysarthric or Non-Dysarthric.

Methodology:

1. **Feature Extraction:**
   - Extract MFCCs (Mel Frequency Cepstral Coefficients) or Mel Spectrograms from the raw audio files using librosa.
   - Each audio file is represented as a feature matrix of shape (num_timesteps, num_features).
   - **Example:**
     - Input shape = (batch_size, 40) where 40 is the number of MFCCs.

2. **Model Architecture:**
   - Conv1D layers process the feature matrix along the time axis to capture local patterns.
   - ReLU Activation adds non-linearity.
   - MaxPooling reduces the dimensionality to retain only important information.
   - Fully Connected Layers (FC) produce the classification logits (2 classes: Dysarthric/Control).

3. **Architecture:**

```
class SimpleCNN(nn.Module):
    def __init__(self, input_dim=40, output_dim=2):
        super(SimpleCNN, self).__init__()
        self.network = nn.Sequential(
            nn.Conv1d(1, 16, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=2),
            nn.Conv1d(16, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.MaxPool1d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.Linear(32 * (input_dim // 4), 128),
            nn.ReLU(),
            nn.Linear(128, output_dim)
        )

    def forward(self, x):
        x = x.unsqueeze(1)  # Add channel dimension
        return self.network(x)
```

4. **Training:**
    - **Loss Function:** CrossEntropyLoss for binary classification.
    - **Optimizer:** Adam with a learning rate of 0.001.
    - **Batch size:** 32.
    - Train for 50 epochs.

5. **Evaluation:**
    - Accuracy, Precision, Recall, and F1-Score are computed on the test set.
    - Confusion matrix is plotted to visualize the performance.

2. RNN (Recurrent Neural Network):

Objective:
An RNN is used to model the sequential nature of the audio feature data. It learns patterns over time to classify speech as dysarthric or non-dysarthric.

Methodology:

1. **Feature Extraction:**
    - Extract sequential features like MFCCs from the raw audio files.
    - Input shape = (batch_size, seq_len, num_features).

2. **Model Architecture:**
    - **RNN Layer**: Processes the sequential feature vectors.
    - **Hidden State**: Captures temporal dependencies over time steps.
    - **Fully Connected Layer**: Maps the final hidden state to the output classes.

3. **Architecture:**

```
class SimpleRNN(nn.Module):
    def __init__(self, input_dim=40, hidden_dim=128, output_dim=2):
        super(SimpleRNN, self).__init__()
        self.rnn = nn.RNN(input_size=input_dim, hidden_size=hidden_dim,
num_layers=2, batch_first=True)
        self.fc = nn.Sequential(
            nn.Linear(hidden_dim, 64),
            nn.ReLU(),
            nn.Linear(64, output_dim)
        )

    def forward(self, x):
        x, _ = self.rnn(x)  # Output from RNN
        x = x[:, -1, :]  # Last hidden state
        x = self.fc(x)
        return x
```

4. **Training:**
    - **Loss Function:** CrossEntropyLoss.
    - **Optimizer:** Adam.
    - **Batch size:** 32.
    - Train for 50 epochs.

5. **Evaluation:**
    - **Metrics:** Accuracy, Precision, Recall, and F1-Score.
    - Confusion matrix is plotted.

3. LSTM (Long Shot Term Memory):

Objective:
An LSTM addresses the limitations of simple RNNs by effectively learning long-term dependencies in sequential data. It works well for audio classification where temporal patterns matter.

Methodology:

1. **Feature Extraction:**
    - Extract MFCCs or Mel Spectrograms.
    - Input shape: (batch_size, seq_len, num_features).

2. **Model Architecture:**
    - Replace the RNN layer with an LSTM layer.
    - LSTM captures long-term dependencies using gates (input, forget, and output gates).
    - Final hidden state is passed to the Fully Connected layer for classification.

3. **Architecture:**

```
class SimpleLSTM(nn.Module):
    def __init__(self, input_dim=40, hidden_dim=128, output_dim=2):
        super(SimpleLSTM, self).__init__()
        self.lstm = nn.LSTM(input_size=input_dim, hidden_size=hidden_dim,
num_layers=2, batch_first=True)
        self.fc = nn.Sequential(
            nn.Linear(hidden_dim, 64),
            nn.ReLU(),
            nn.Linear(64, output_dim)
        )

    def forward(self, x):
        x, _ = self.lstm(x)  # Output from LSTM
        x = x[:, -1, :]  # Last hidden state
        x = self.fc(x)
        return x
```

4. **Training:**
    - **Loss Function:** CrossEntropyLoss.
    - **Optimizer:** Adam.
    - Train for 50 epochs.
5. **Evaluation:**
    - **Metrics:** Accuracy, Precision, Recall, F1-Score.
    - Confusion matrix.

4. Hybrid CNN-LSTM Model:

Objective:

- **Combine the strengths of CNN and LSTM:**
    - CNN extracts spatial features from the input data (e.g., MFCCs).
    - LSTM processes the extracted features to model temporal dependencies.

Methodology:

1. **Feature Extraction:**
    - Extract MFCCs or Mel Spectrograms.
    - **Input shape:** (batch_size, seq_len, num_features).

2. **Model Architecture:**
    - **CNN Layers:**
        - Extract spatial features from the input.
    - **Reshape for LSTM:**
        - Output from CNN layers is reshaped for LSTM.
    - **LSTM Layers:**
        - LSTM processes the extracted features over time.
    - **Fully Connected Layers:**
        - Perform classification.

3. **Architecture:**

```
class HybridCNNLSTM(nn.Module):
  def __init__(self, input_dim=40, cnn_output_dim=32, lstm_hidden_dim=128, output_dim=2):
      super(HybridCNNLSTM, self).__init__()

      # CNN Layers
      self.conv1 = nn.Conv1d(1, 16, kernel_size=3, padding=1)
      self.conv2 = nn.Conv1d(16, cnn_output_dim, kernel_size=3, padding=1)
      self.pool = nn.MaxPool1d(2, 2)
      self.relu = nn.ReLU()

      # LSTM Layer
      self.lstm = nn.LSTM(input_size=cnn_output_dim, hidden_size=lstm_hidden_dim, num_layers=2, batch_first=True)
```

```
        # Fully Connected Layers
        self.fc = nn.Sequential(
            nn.Linear(lstm_hidden_dim, 64),
            nn.ReLU(),
            nn.Linear(64, output_dim)
        )

    def forward(self, x):
        x = x.unsqueeze(1)  # Add channel dimension for CNN
        x = self.relu(self.conv1(x))
        x = self.pool(x)
        x = self.relu(self.conv2(x))
        x = self.pool(x)

        x = x.permute(0, 2, 1)  # Reshape for LSTM
        x, _ = self.lstm(x)
        x = x[:, -1, :]  # Last hidden state
        x = self.fc(x)
        return x
```

Summary of Differences:

- **CNN**: Good for extracting spatial features.
- **RNN**: Learns temporal dependencies but struggles with long sequences.
- **LSTM**: Improves over RNN with gates to handle long-term dependencies.
- **Hybrid CNN-LSTM**: Combines CNN and LSTM for both spatial and temporal feature extraction.

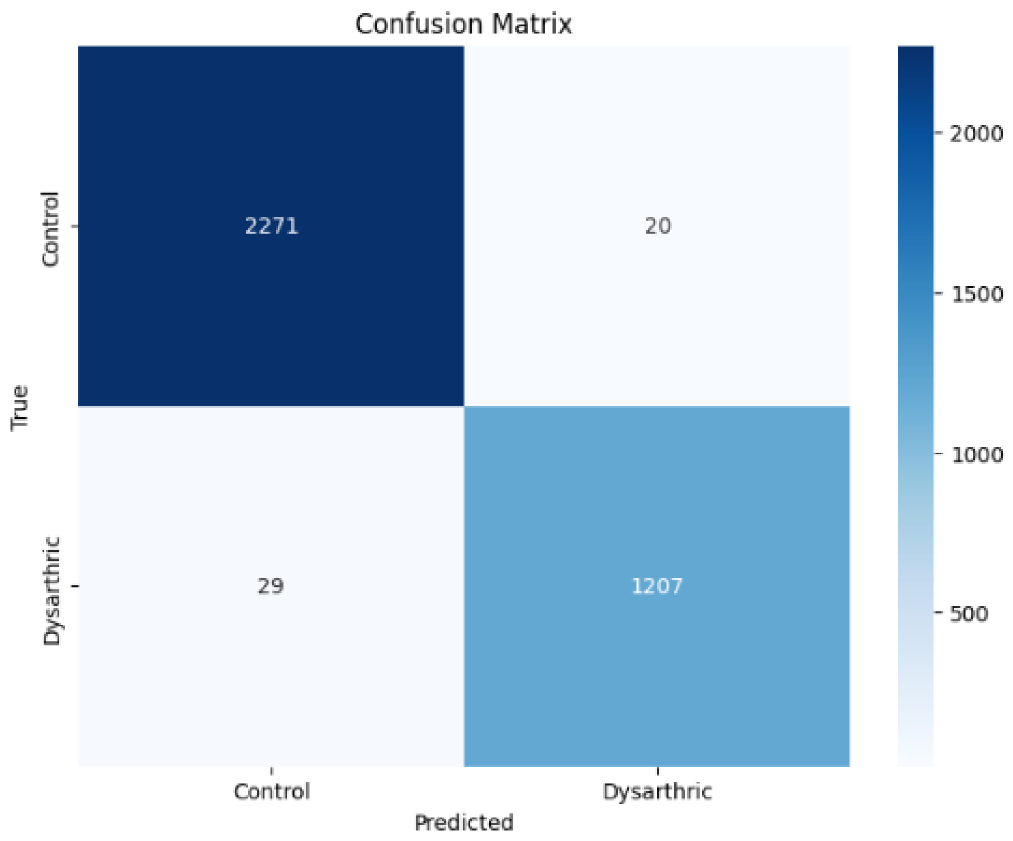
Results and Comparison

Technique	Accuracy	Precision	Recall	F1_score
CNN	0.9861	0.9855	0.9839	0.9847
RNN	0.9711	0.9667	0.9701	0.9684
LSTM	0.9790	0.9785	0.9753	0.9769
Hybrid CNN-LSTM	0.9691	0.9689	0.9630	0.9658
Wav2vec2	0.9920	0.9950	0.9950	0.9900
Contrastive Learning	0.9800	0.9843	0.9833	0.9887

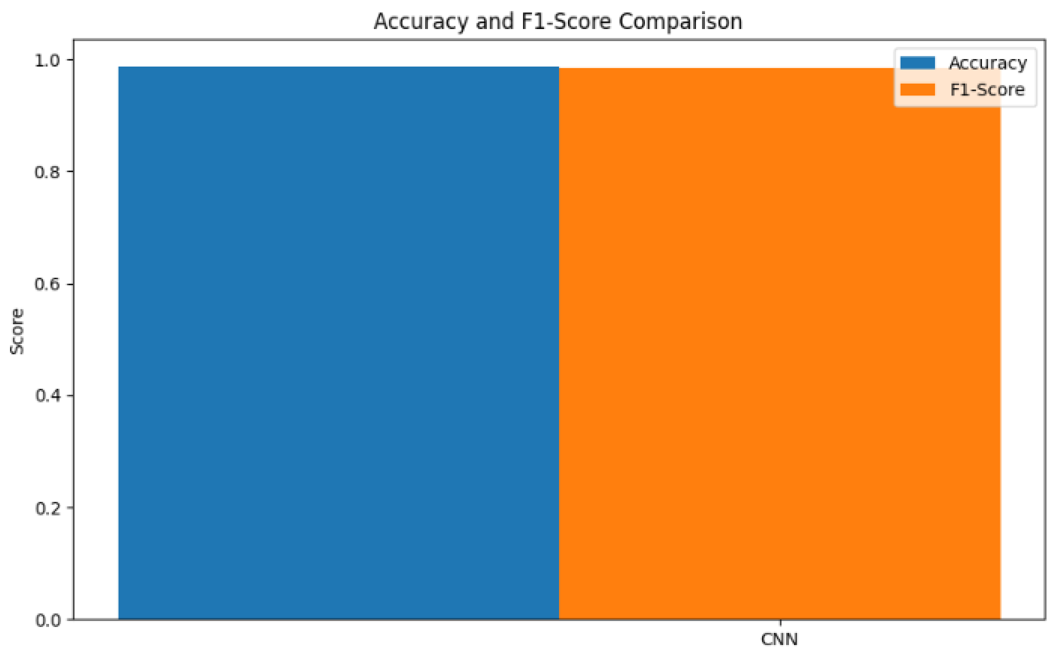
Visual Representation

1. **CNN (Convolutional Neural Network):**

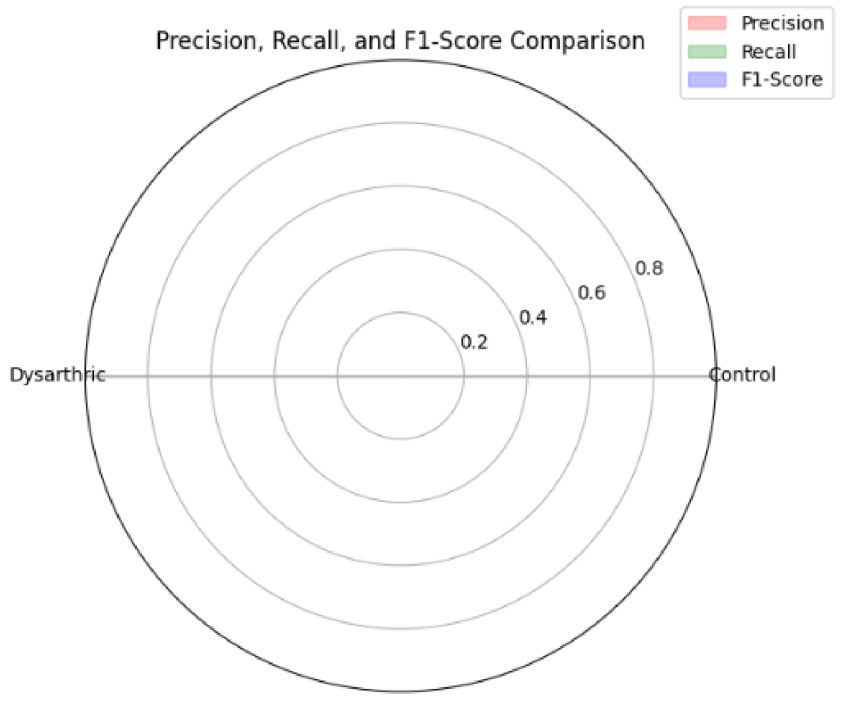
Confusion Matrix:



Bar-chart: (Accuracy & F1_score)

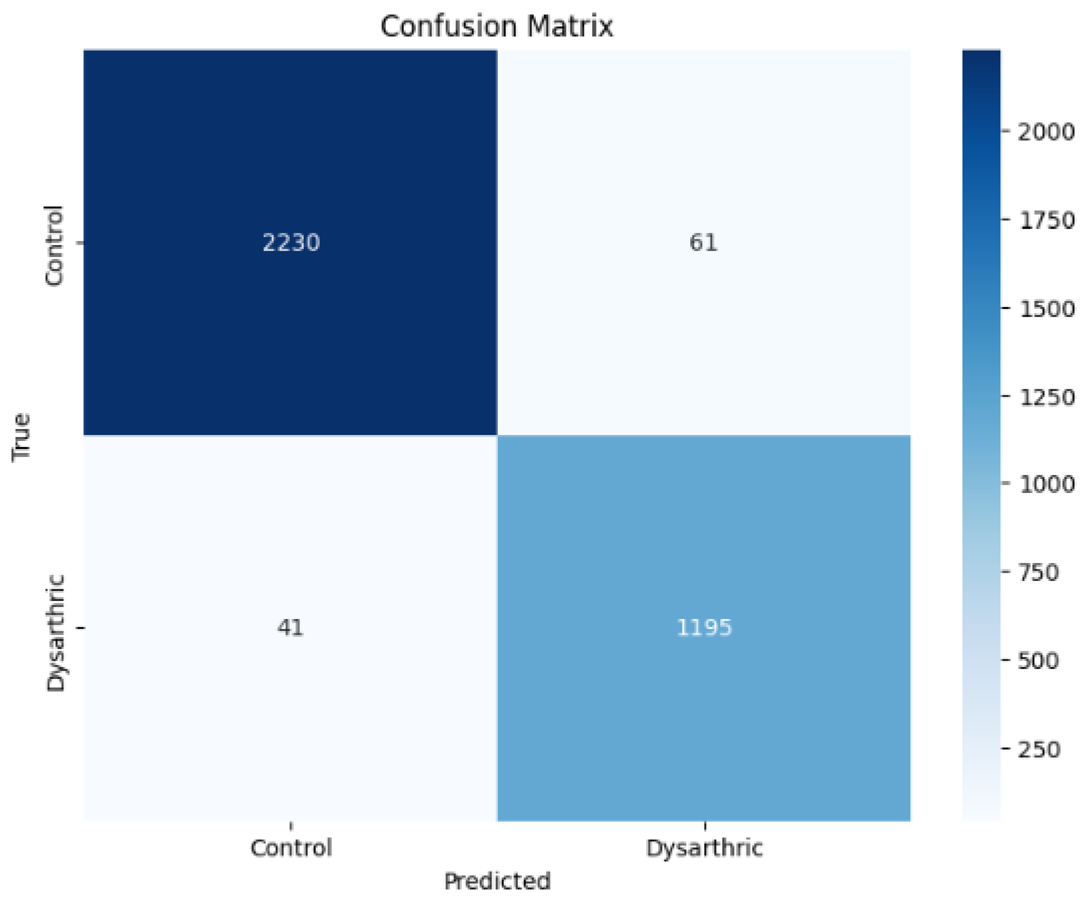


Radar-Chart: (Precision, recall & f1-score)

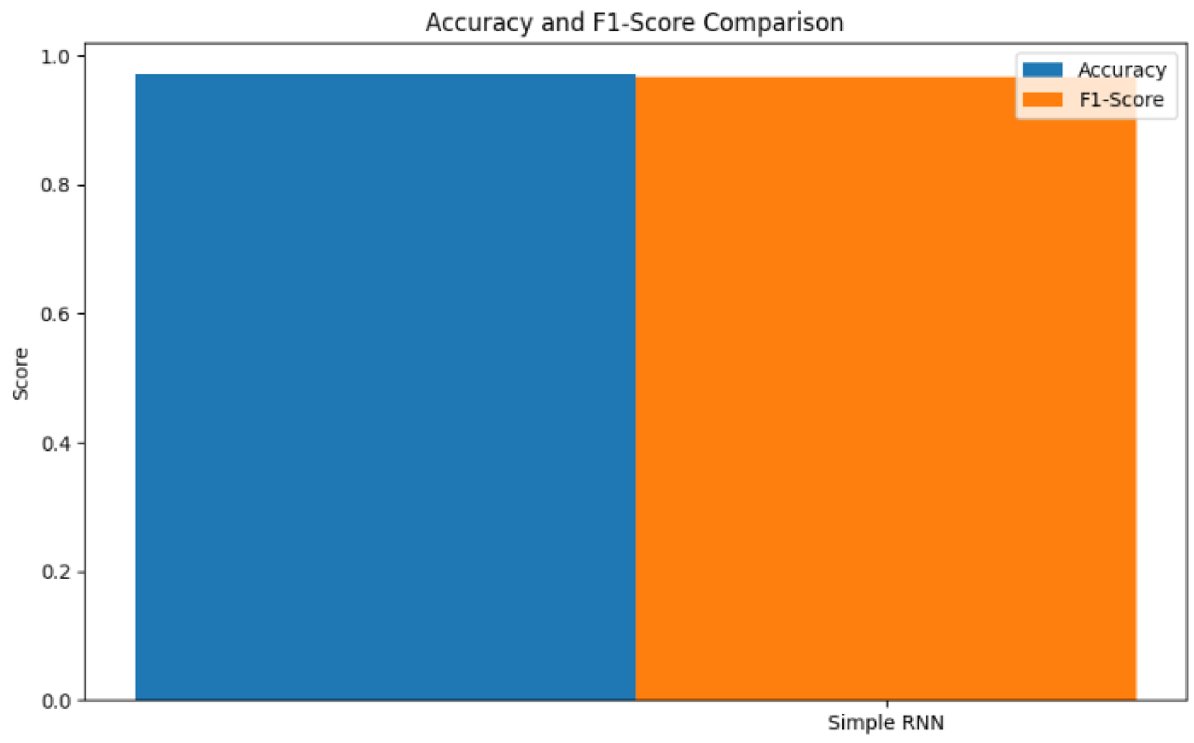


2. RNN (Recurrent Neural Network):

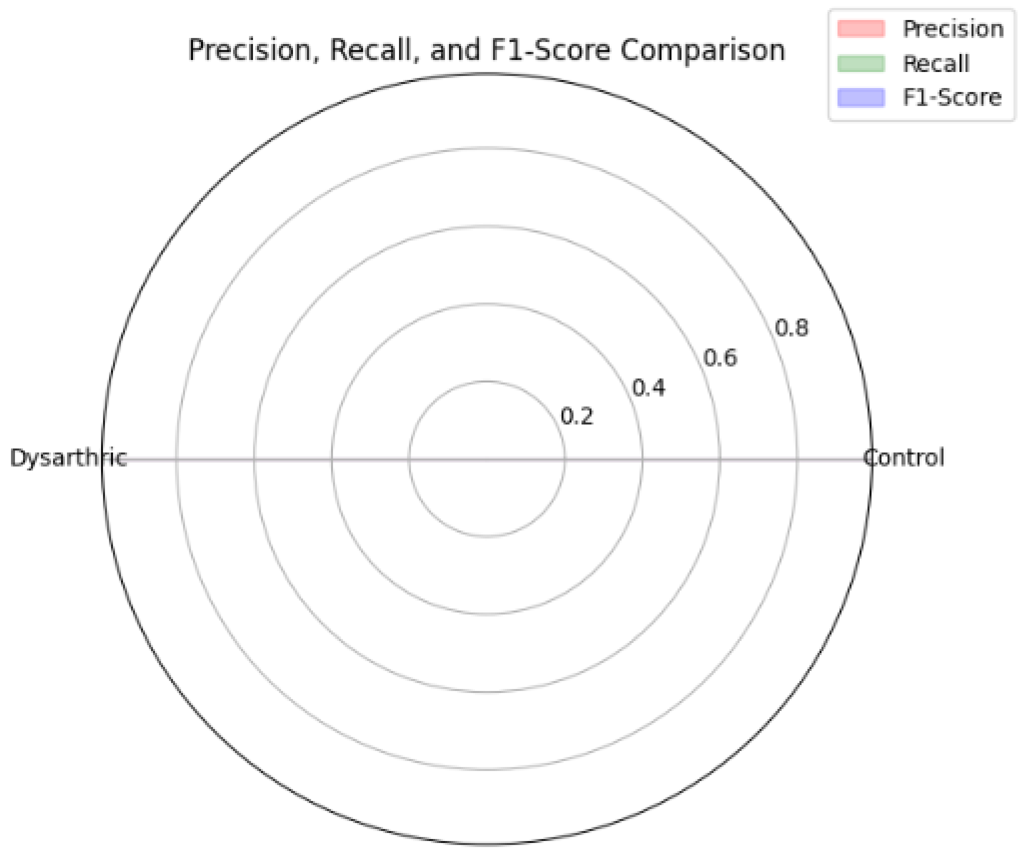
Confusion matrix:



Bar-chart:

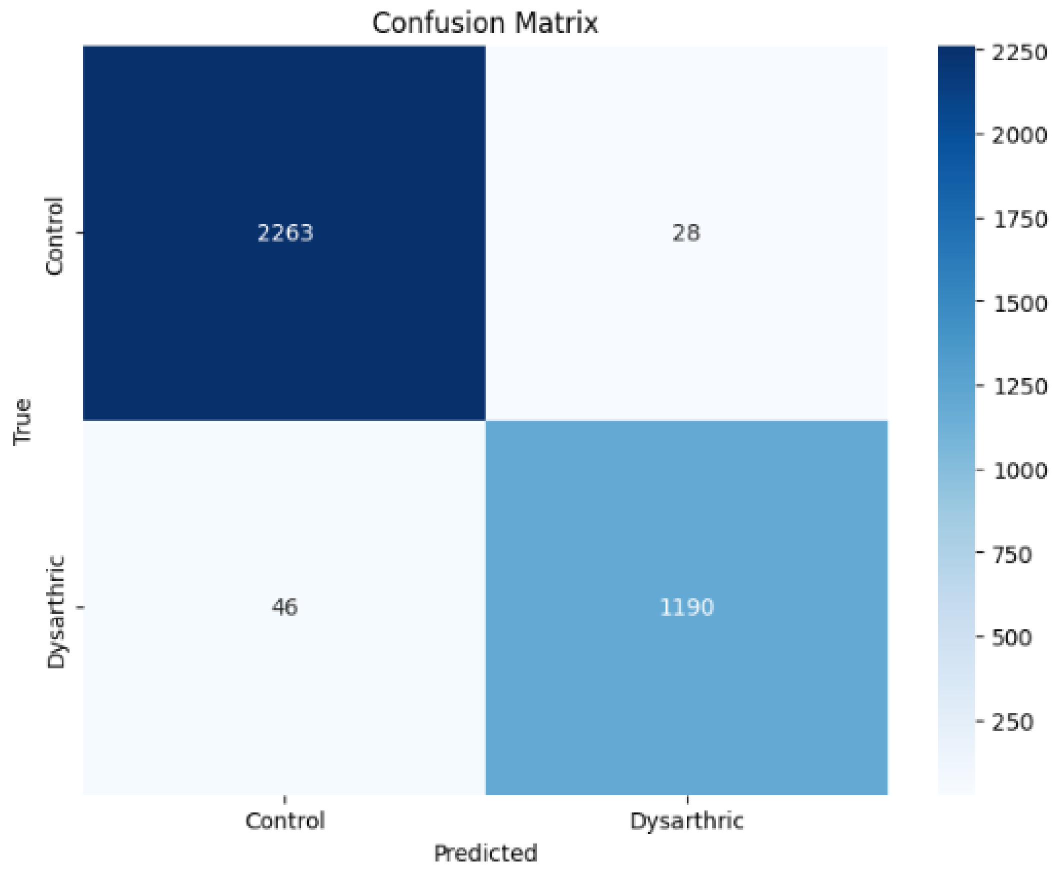


Radar-chart:

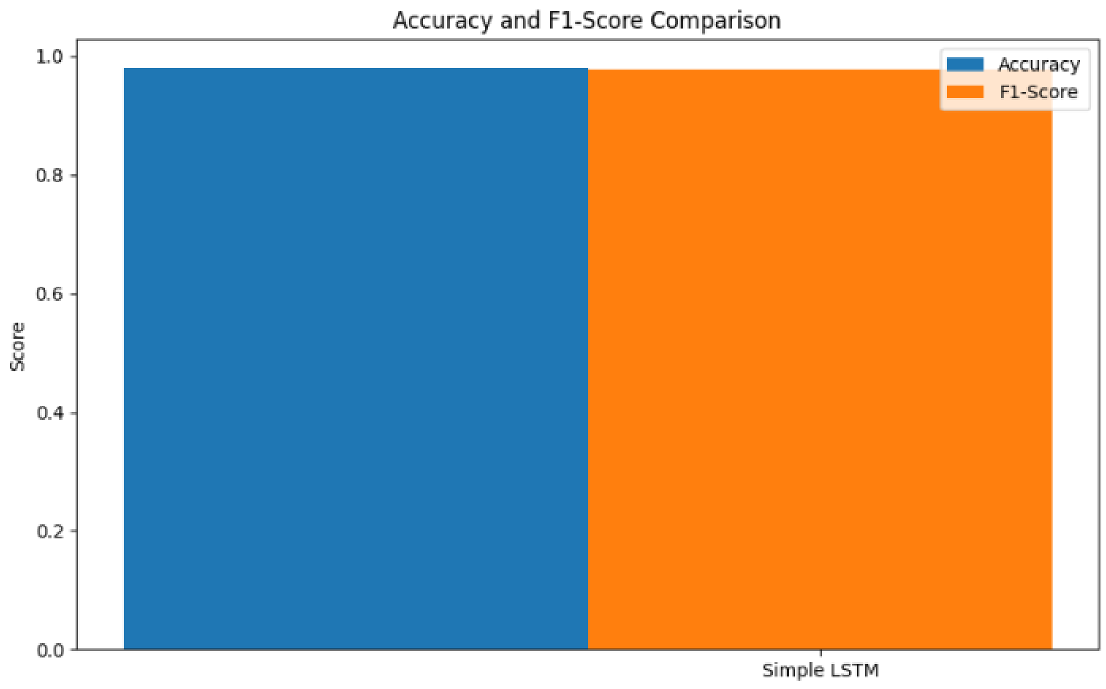


3. LSTM (Long Shot Term Memory):

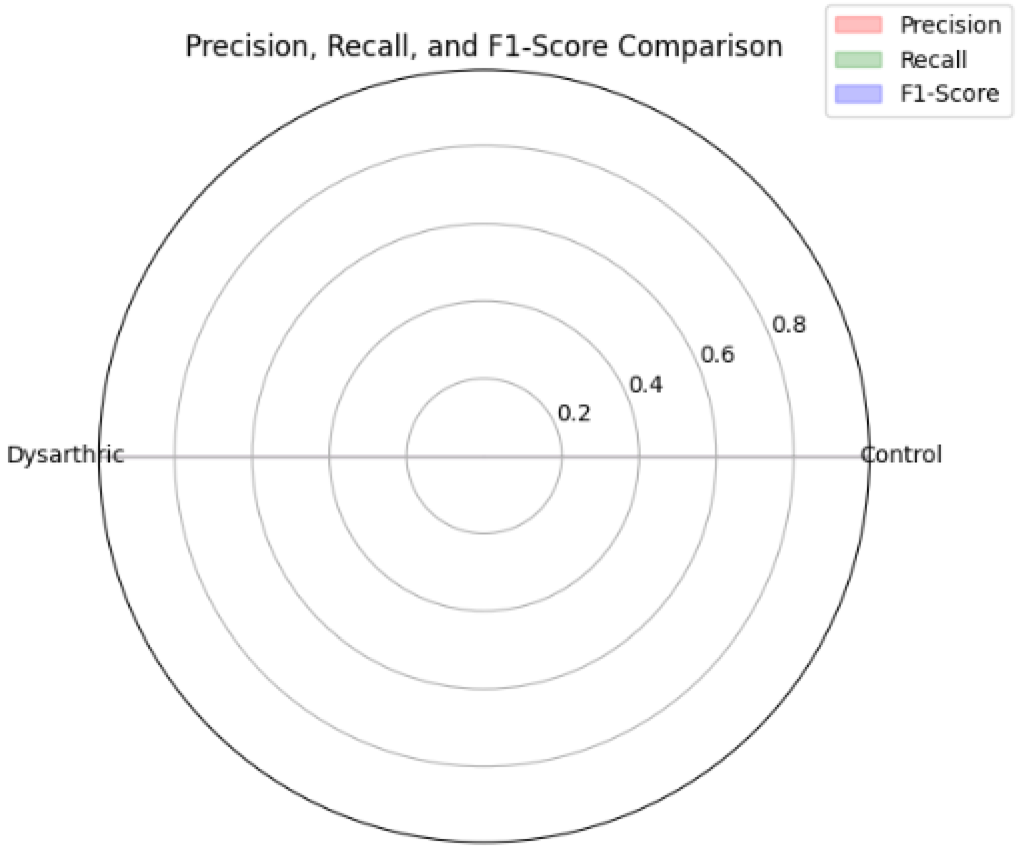
Confusion matrix:



Bar-chart:

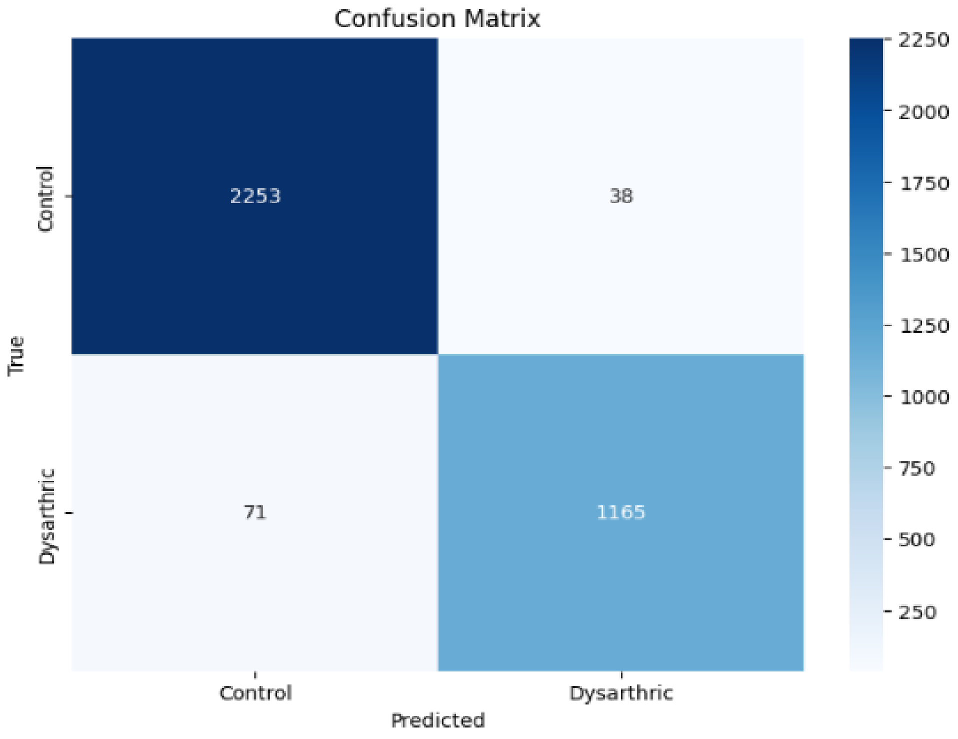


Radar-chart:

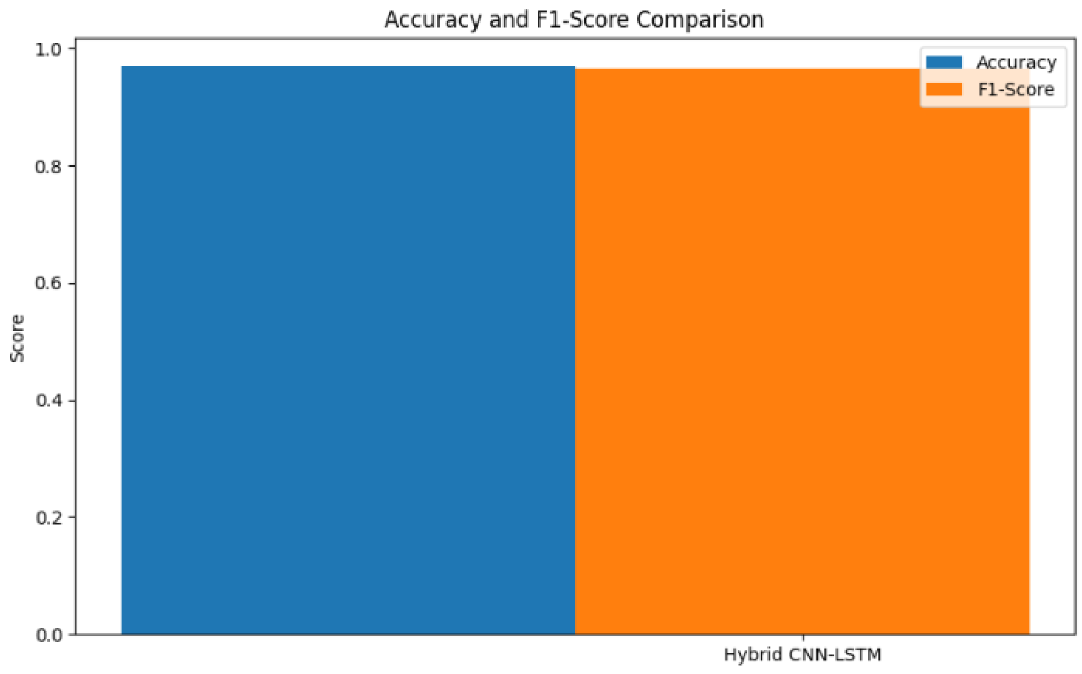


4. Hybrid CNN-LSTM Model:

Confusion matrix:



Bar-chart:



Radar-chart: