Simple Emotions Detection in Speech using FastDTW

I. Introduction

A. Context of Emotion Recognition in Audio

Emotion recognition from audio signals has become an increasingly important field of study in artificial intelligence and human-computer interaction. The ability to accurately detect and interpret human emotions from speech can have far-reaching applications in various domains, including healthcare, customer service, and entertainment.

The primary challenge in emotion recognition from audio lies in the complex and variable nature of human speech. Emotions can be expressed through various acoustic features such as pitch, intensity, and rhythm, which can vary significantly across individuals and contexts.

B. Problem Statement: Efficient Emotion Detection using DTW and Neural Networks

This project focuses on developing an efficient emotion detection system using Dynamic Time Warping (DTW) and neural networks. Our approach combines the strengths of DTW, a technique well-suited for time series analysis, with the power of neural networks to create a robust emotion recognition system.

Specifically, we explore the use of FastDTW, an approximation algorithm that addresses the computational limitations of classical DTW, and compare its performance with the traditional DTW method.

C. Rationale for the Chosen Approach

1. DTW allows for flexible alignment of time series data, which is crucial when dealing with speech signals of varying lengths and speeds.
2. By incorporating neural networks, we can leverage their ability to learn complex patterns and representations from high-dimensional data, potentially improving the overall accuracy of emotion classification.

This comparison aims to identify the most efficient and accurate approach for emotion detection in audio signals.

D. Dataset Overview

For this study, we utilize the `Ryerson Audio-Visual Database of Emotional Speech and Song (RAVDESS)` dataset. This dataset consists of speech audio files representing various emotions, providing a robust foundation for training and evaluating our emotion recognition models. The RAVDESS dataset includes recordings of professional actors expressing eight emotions: `neutral, calm, happy, sad, angry, fearful, disgust, and surprised`.

For our project, we focus on a subset of these emotions:

- `Neutral`
- `Happy`

- `Sad`
- `Angry`

This selection creates a more focused and manageable scope for our analysis while still covering a range of distinct emotional states.

In the following sections, we will detail our methodology, including:

1. **Data preparation**
2. **Feature extraction**
3. **Implementation of classical DTW and FastDTW**
4. **Development of a neural network model**

We will then present our results, comparing the performance of these approaches in terms of accuracy, speed, and scalability. Finally, we will discuss the implications of our findings, the limitations of our approach, and potential avenues for future research in this exciting field of emotion recognition from audio signals.

II. Analysis

A. Data Preparation

The first step in our emotion recognition project involved preparing the RAVDESS dataset for analysis. This process was crucial to ensure that our subsequent feature extraction and model training steps would be effective. Here's an overview of our data preparation process:

1. Dataset Selection and File Organization

We focused on a subset of the RAVDESS dataset, specifically selecting audio files that met the following criteria:

- **Modality:** Audio-only (code `03`)
- **Vocal channel:** Speech (code `01`)
- **Emotions:** Neutral (`01`), Happy (`03`), Sad (`04`), and Angry (`05`)
- We included both emotional intensities (`01` for normal, `02` for strong)
- We included all statements, repetitions, and actors

To streamline our analysis, we developed a function to filter and organize the relevant audio files:

```
def filter_ravdess_files(base_dir):
    relevant_files = []
    relevant_emotions = {"01", "03", "04", "05"}  # neutral, happy, sad, angry

    for actor_folder in os.listdir(base_dir):
        actor_path = os.path.join(base_dir, actor_folder)
        if os.path.isdir(actor_path):
            for filename in os.listdir(actor_path):
                if filename.endswith(".wav"):
                    parts = filename.split("-")
                    if (parts[0] == "03" and parts[1] == "01" and
                        parts[2] in relevant_emotions):
                        full_path = os.path.join(actor_path, filename)
                        emotion = get_emotion(filename)
                        relevant_files.append({
                            "filename": filename,
                            "path": full_path,
                            "emotion": emotion,
                            "actor": actor_folder,
                        })

    return pd.DataFrame(relevant_files)
```

This function creates a data frame containing information about each relevant audio file, including its filename, path, emotion label, and actor identifier.

2. Feature Extraction Preparation

We prepared our data for feature extraction using the `prepare_data` function. This function performs the following steps:

1. Filters the relevant audio files using the `filter_ravdess_files` function.
2. Extracts features from each audio file using the `extract_features` function.
3. Organizes the extracted features into a pandas DataFrame.
4. Saves the features to a CSV file for further analysis.

Here's an overview of the `extract_features` function:

```
def extract_features(y, sr, n=13):
    # Extract MFCCs (n=13)
    mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=n)

    # Extract other features
    chroma = librosa.feature.chroma_stft(y=y, sr=sr)
    mel = librosa.feature.melspectrogram(y=y, sr=sr)
    contrast = librosa.feature.spectral_contrast(y=y, sr=sr)
    tonnetz = librosa.feature.tonnetz(y=librosa.effects.harmonic(y), sr=sr)
    spectral_centroid = librosa.feature.spectral_centroid(y=y, sr=sr)
    zero_crossing_rate = librosa.feature.zero_crossing_rate(y)
    rms = librosa.feature.rms(y=y)

    # Compute mean and standard deviation of the features
    # ... [codes that compute mean and std for each feature]

    # Combine all features into a single feature vector
    combined_features = np.hstack([
        mfccs.flatten(), mfccs_mean, mfccs_std,
        chroma.flatten(), chroma_mean, chroma_std,
        mel.flatten(), mel_mean, mel_std,
        contrast.flatten(), contrast_mean, contrast_std,
        tonnetz.flatten(), tonnetz_mean, tonnetz_std,
        spectral_centroid.flatten(), spectral_centroid_mean,
spectral_centroid_std,
        zero_crossing_rate.flatten(), zero_crossing_rate_mean,
zero_crossing_rate_std,
        rms.flatten(), rms_mean, rms_std
    ])
    # Return a dictionary of all features separately and the combined feature
vector
    return {
        "mfccs": mfccs,
        "chroma": chroma,
        "mel": mel,
        "contrast": contrast,
        "tonnetz": tonnetz,
        "spectral_centroid": spectral_centroid,
        "zero_crossing_rate": zero_crossing_rate,
        "rms": rms,
        "combined_features": combined_features,
    }
```

This function extracts a comprehensive set of audio features, including MFCCs, chroma, mel spectrograms, spectral contrast, tonnetz, spectral centroid, zero crossing rate, and RMS energy. It computes both the raw features and their statistical summaries (mean and standard deviation).

3. Data Processing Notes

- We did not apply any additional audio preprocessing techniques (e.g., noise reduction, silence removal) since the quality of audio in the dataset is very good and consistent.
- Due to time constraints, we did not implement data augmentation techniques. This could be an area for future improvement, especially if working with a larger or more diverse dataset.

The data preparation process resulted in a clean, organized dataset with extracted features ready for model training and evaluation. This careful preparation laid the foundation for the subsequent stages of our emotion recognition project.

B. Feature Selection and Extraction

In our emotion recognition project, we carefully selected a set of audio features that are known to capture various aspects of speech that can be indicative of emotional states. Here's an overview of the features we extracted and their potential relevance to emotion recognition:

1. Mel-Frequency Cepstral Coefficients (MFCCs)

- **Description:** MFCCs represent the short-term power spectrum of a sound, based on a linear cosine transform of a log power spectrum on a nonlinear mel scale of frequency.
- MFCCs are excellent at capturing the overall spectral envelope of speech, which can vary with different emotional states. For example, anger might be associated with higher energy in certain frequency bands.

2. Chroma Features

- **Description:** Chroma features represent the 12 different pitch classes and their intensities.
- While more commonly used in music analysis, chroma features can capture tonal content in speech, which may vary with emotional state. For instance, happiness might be associated with higher pitch variability.

3. Mel Spectrogram

- **Description:** A spectrogram with the frequency scale converted to the mel scale, which is more closely aligned with human auditory perception.
- Mel spectrograms can reveal patterns in speech energy distribution across frequencies and time, which may differ between emotional states.

4. Spectral Contrast

- **Description:** Measures the level difference between peaks and valleys in the spectrum.
- This can help distinguish between "clear" and "muddy" sounds, which might be indicative of different emotional states. For example, anger might have sharper spectral contrasts than sadness.

5. Tonnetz

- **Description:** Represents harmonic and tonal content of audio signals.
- While primarily used in music analysis, tonnetz features might capture subtle harmonic changes in speech associated with different emotions.

6. Spectral Centroid

- **Description:** Represents the "center of mass" of the spectrum.
- This feature is often associated with the perceived brightness of a sound. Brighter sounds (higher spectral centroid) might be associated with more active or positive emotions.

7. Zero Crossing Rate

- **Description:** The rate at which the signal changes from positive to negative or back.
- This can be an indicator of the noisiness or breathiness of speech, which might vary with emotional state. For instance, a higher zero crossing rate might be associated with more agitated emotional states.

8. Root Mean Square (RMS) Energy

- **Description:** Represents the loudness of the signal over time.
- The overall energy of speech can be a strong indicator of certain emotions. For example, anger or excitement might be associated with higher RMS energy, while sadness might have lower energy.

For each of these features, we extracted both the raw values over time and computed statistical measures (mean and standard deviation) to capture both the dynamic and overall characteristics of the speech signal.

By combining these diverse features, we aim to capture a comprehensive representation of the speech signal that can effectively differentiate between various emotional states. The multidimensional nature of this feature set allows our models to learn complex patterns that may be indicative of different emotions in speech.

C. Algorithm Selection and Comparison

In this section, we explore the implementation of Classical Dynamic Time Warping (DTW) and compare it with FastDTW for emotion recognition in audio signals.

1. Recap on DTW and its Limitations

Dynamic Time Warping (DTW) is a technique used to find an optimal alignment between two time-dependent sequences. In the context of speech emotion recognition, DTW can be used to compare audio features of different lengths, making it particularly useful for comparing speech samples that may vary in duration.

Here's the implementation of our classical DTW function:

```python
def classical_dtw(x, y, dist=euclidean):
    """Compute Dynamic Time Warping (DTW) distance between two sequences."""
    nx, ny = len(x), len(y)
    cost_matrix = np.zeros((nx + 1, ny + 1))
    cost_matrix[0, 1:] = np.inf
    cost_matrix[1:, 0] = np.inf

    for i in range(1, nx + 1):
        for j in range(1, ny + 1):
            cost = dist(x[i - 1], y[j - 1])
            cost_matrix[i, j] = cost + min(
                cost_matrix[i - 1, j], cost_matrix[i, j - 1], cost_matrix[i - 1,
j - 1]
            )

    return cost_matrix[nx, ny]
```

This implementation calculates the DTW distance between two sequences using a specified distance function (default is Euclidean distance).

Classical DTW, while effective, has some limitations:

- **Time Complexity:** The algorithm has a time complexity of O(N^2), where N is the length of the sequences being compared. This can be computationally expensive for long sequences.
- **Space Complexity:** It requires O(N^2) space to store the cost matrix.
- Lack of Scalability: Due to its quadratic time and space complexity, classical DTW struggles with large datasets or real-time applications.

2. Introduction to FastDTW

FastDTW is an approximation of DTW that attempts to address the limitations of classical DTW. It uses a multilevel approach to compute a DTW-like alignment with linear time and space complexity.

a. Advantages

- **Linear Time Complexity:** FastDTW has a time complexity of O(N), making it much faster than classical DTW for long sequences.
- **Linear Space Complexity:** It requires only O(N) space, allowing it to handle much larger datasets.

- **Scalability:** The improved time and space complexity makes FastDTW more suitable for large-scale or real-time applications.
- **Accuracy:** While an approximation, FastDTW often produces results very close to those of classical DTW.

b. Implementation Overview

For my part of the project, I used the FastDTW library implementation. And my teammate will be implemented our own version of FastDTW from scratch in her individual code. This implementation will replace the imported function in the `fast_dtw.py` file.

The basic algorithm for FastDTW can be summarized as follows:

1. **Coarsen** the time series to a low resolution
2. Find the optimal warp path at the low resolution
3. **Project** this path to a higher resolution
4. **Refine** the path at the higher resolution
5. Repeat steps 3-4 until the full resolution is reached

3. Experimental Setup for Comparison

To compare Classical DTW and FastDTW, we set up experiments using the RAVDESS dataset and the features we extracted earlier. We implemented both algorithms and tested them on the same subset of data.

For our experiments, we created a simple `MoodDetectionModel` and optimized it using Bayesian optimization. Here's an overview of our setup:

```
class MoodDetectionModel:
    """Mood detection model using DTW."""

    def __init__(self, templates, distance_func, feature_weights=None):
        self.templates = templates
        self.distance_func = distance_func
        self.feature_weights = feature_weights if feature_weights is not None
else [1.0] * 5

    def fit(self, X, y):
        return self

    def predict(self, X):
        return [
            classify_emotion(
                sample, self.templates, self.distance_func, self.feature_weights
            )
            for _, sample in X.iterrows()
        ]

    def score(self, X, y):
        y_pred = self.predict(X)
        return accuracy_score(y, y_pred)

def optimize_model(df, templates, distance_func, n_iter=50, logger=None):
    """Optimize model parameters using Bayesian optimization."""
    X = df.drop("emotion", axis=1)
    y = df["emotion"]

    model = MoodDetectionModel(templates, distance_func)

    search_space = {
        f"feature_weights_{i}": Real(0.01, 10.0, prior="log-uniform") for i in
range(5)
    }

    opt = BayesSearchCV(
        model,
        search_spaces=search_space,
        n_iter=n_iter,
        cv=5,
        n_jobs=-1,
        verbose=2,
        random_state=42,
    )

    opt.fit(X, y)

    best_params = [opt.best_params_[f"feature_weights_{i}"] for i in range(5)]
    best_score = opt.best_score_

    return best_params, best_score, opt.cv_results_
```

This setup allows us to optimize the feature weights for our mood detection model using either Classical DTW or FastDTW as the distance function.

Our comparison focused on three key aspects:

1. **Accuracy:** How well each algorithm performed in classifying emotions
2. **Speed:** The time taken to process the dataset
3. **Scalability:** How performance changed with increasing data size

4. Results and Analysis

a. Accuracy Comparison

Both Classical DTW and FastDTW showed comparable accuracy in emotion classification:

- **Classical DTW:**
    - **Best score (50 iterations):** 0.2531122166943062 (Derivative DTW)
    - **Best score (50 iterations):** 0.14286346047540077 (Classical DTW)
- **FastDTW:**
    - **Best score (50 iterations):** 0.21587617468214484 (Derivative DTW)
    - **Best score (50 iterations):** 0.14733001658374792 (Euclidean Distance)

b. Speed Comparison

FastDTW significantly outperformed Classical DTW in terms of speed:

- **Classical DTW:**
  - **1 iteration:** 7m20.6s
  - **50 iterations:** 78m32.5s
- **FastDTW:**
  - **1 iteration:** 1m5.9s
  - **50 iterations:** 11m58.4s

c. Scalability Analysis

As the dataset size increased, we observed that:

- Classical DTW's performance degraded quickly, with computation time increasing quadratically.
- FastDTW maintained relatively consistent performance, with only a linear increase in computation time.

These results demonstrate that while both algorithms offer similar accuracy, FastDTW provides significant advantages in terms of speed and scalability, making it more suitable for large-scale emotion recognition tasks.

D. Neural Network for Emotion Recognition

1. Rationale for Adding a Neural Network Approach

The DTW-based methods showed promise in emotion recognition, we decided to explore a neural network approach to potentially capture more complex patterns in the audio features. Neural networks have shown great success in various audio processing tasks, including emotion recognition, due to their ability to learn hierarchical representations of data.

2. Architecture of the Proposed Neural Network

We designed a neural network that combines LSTM (Long Short-Term Memory) layers to process temporal features with fully connected layers to integrate statistical features. Here's the architecture of our `EmotionClassifier`:

```python
class EmotionClassifier(nn.Module):
    def __init__(self, mfcc_dim, stat_dim, hidden_size, num_classes):
        super(EmotionClassifier, self).__init__()
        self.lstm = nn.LSTM(mfcc_dim, hidden_size, batch_first=True)
        self.fc1 = nn.Linear(hidden_size + stat_dim, 64)
        self.fc2 = nn.Linear(64, num_classes)
        self.relu = nn.ReLU()

    def forward(self, mfccs, stat_features):
        _, (hidden, _) = self.lstm(mfccs)
        combined = torch.cat((hidden.squeeze(0), stat_features), dim=1)
        x = self.relu(self.fc1(combined))
        x = self.fc2(x)
        return x
```

This architecture processes MFCCs through an LSTM layer and combines the output with statistical features before passing through fully connected layers for classification.

3. Feature Selection and Preprocessing

We used a combination of temporal and statistical features:

1. **Temporal Features:** We primarily used MFCCs (Mel-Frequency Cepstral Coefficients) as our temporal features. MFCCs capture the short-term power spectrum of sound and are widely used in speech recognition tasks.
2. **Statistical Features:** We included statistical measures (mean and standard deviation) of various audio features including MFCCs, chroma, mel spectrogram, spectral contrast, tonnetz, spectral centroid, zero crossing rate, and RMS energy.

Here's how we prepared our features:

```python
class EmotionDataset(Dataset):
    def __init__(self, features, labels):
        self.features = features
        self.labels = labels

    def __getitem__(self, idx):
        mfccs = np.array(json.loads(self.features[idx]["mfccs"]))
        mfccs = mfccs.T  # Transpose to (num_time_steps, 13)
        statistical_features = np.concatenate(
            [
                json.loads(self.features[idx][f"{feat}_mean"])
                + json.loads(self.features[idx][f"{feat}_std"])
                for feat in [
                    "mfccs", "chroma", "mel", "contrast", "tonnetz",
                    "spectral_centroid", "zero_crossing_rate", "rms",
                ]
            ]
        )
        return (
            torch.FloatTensor(mfccs),
            torch.FloatTensor(statistical_features),
            torch.LongTensor([self.labels[idx]]),
        )
```

4. Feature Alignment using FastDTW

Before feeding our temporal features (MFCCs) into the neural network, we used FastDTW to align these sequences. This alignment is crucial because:

1. It allows us to compare and process audio samples of different lengths.
2. It helps to match corresponding parts of the audio, regardless of speaking speed or pauses.

We used the MFCCs for alignment as they capture the overall spectral envelope of the speech signal. Here's how we performed the alignment:

```python
def align_sequences(sequences):
    """Align sequences using FastDTW."""
    max_len = max(seq.shape[1] for seq in sequences)
    reference = np.zeros((sequences[0].shape[0], max_len))
    aligned_sequences = []
    for seq in sequences:
        distance, path = fastdtw(reference.T, seq.T, dist=euclidean)
        aligned_seq = np.zeros_like(reference)
        for ref_idx, seq_idx in path:
            aligned_seq[:, ref_idx] = seq[:, seq_idx]
        aligned_sequences.append(aligned_seq)

    return np.array(aligned_sequences)
```

This alignment process ensures that all MFCC sequences have the same length, allowing for batch processing in the neural network.

5. Training Process and Challenges

We trained our model using the following process:

1. **Data splitting:** We split our dataset into training and validation sets.
2. **Feature alignment:** We aligned the MFCC sequences using FastDTW.
3. **Model initialization:** We created an instance of our `EmotionClassifier`.
4. **Training loop:** We used cross-entropy loss and Adam optimizer for training.

Here's a snippet of our training process:

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=lr)

for epoch in range(num_epochs):
    model.train()
    for mfccs, stat_features, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(mfccs, stat_features)
        loss = criterion(outputs, labels.squeeze())
        loss.backward()
        optimizer.step()

    # Validation step
    model.eval()
    with torch.no_grad():
        for mfccs, stat_features, labels in val_loader:
            outputs = model(mfccs, stat_features)
            # ... calculate validation metrics
```

6. Performance Evaluation

The performance of our neural network model, while promising, did not significantly outperform the DTW-based approaches. There are several potential reasons for this:

1. **Limited tuning time:** Due to time constraints, we were unable to extensively tune the hyperparameters of our model.
2. **Feature selection:** The combination of features we used may not be optimal for the neural network approach.

3. **Dataset limitations:** The RAVDESS dataset, while high-quality, is relatively small and may not provide enough diversity for the neural network to generalize well.

Here are the best results from our experiments:

Classical DTW

```
Epoch 50/50, Train Loss: 0.6006, Train Acc: 75.05%, Val Loss: 0.9219, Val Acc:
62.22%
Best Validation Accuracy: 62.2222 at Epoch 19.0
Corresponding Training Accuracy: 61.6387
Time Taken: 2m 22.6s
```

FastDTW

```
Epoch 50/50, Train Loss: 0.6901, Train Acc: 70.02%, Val Loss: 1.0673, Val Acc:
52.59%
Best Validation Accuracy: 62.9630 at Epoch 40.0
Corresponding Training Accuracy: 67.0391
Time Taken: 1m 50s
```

An intersting observation from our study is that FastDTW outperformed classical DTW not only in terms of speed but also in accuracy.

We hypothesize that this unexpected result might be attributed to the relatively small and homogeneous nature of our dataset. The approximation introduced by FastDTW could potentially act as a form of regularization, mitigating overfitting issues that may be more pronounced with classical DTW on such a dataset. However, this hypothesis warrants further investigation through more comprehensive studies to validate and understand this phenomenon.

III. Conclusion

A. Summary of Findings

Our project explored various approaches to emotion recognition in speech, focusing on the comparison between Dynamic Time Warping (DTW) techniques and neural network-based methods. Key findings include:

1. **Classical DTW vs. FastDTW:**
   - Both methods showed comparable accuracy in emotion classification.
   - FastDTW significantly outperformed Classical DTW in terms of speed and scalability.
   - FastDTW proved to be more suitable for large-scale emotion recognition tasks due to its linear time and space complexity.
2. **Neural Network Approach:**
   - Our LSTM-based model showed promise in capturing temporal aspects of speech emotions.
   - The performance was comparable to DTW-based methods, with the best validation accuracy reaching about 61%.
   - The neural network approach demonstrated potential for improvement with further optimization.
3. **Feature Importance:**

- MFCCs proved to be crucial features for both DTW and neural network approaches.
- The combination of temporal (MFCCs) and statistical features (derived from various audio characteristics) provided a comprehensive representation of emotional speech.

B. Limitations of the Current Approach

Despite the progress made, our approach has several limitations:

1. **Dataset Size and Diversity:** The RAVDESS dataset, while high-quality, is relatively small and may not capture the full range of emotional expressions across different demographics.
2. **Limited Hyperparameter Tuning:** Due to time constraints, we couldn't perform extensive hyperparameter optimization for our models, potentially limiting their performance.
3. **Feature Selection:** While we used a range of audio features, the optimal combination for emotion recognition might not have been achieved.
4. **Generalizability:** Our models were trained and tested on acted emotions in a controlled environment, which may not fully represent real-world emotional expressions.

C. Future Research Directions

Based on our findings and limitations, several avenues for future research emerge:

1. **Expanded Dataset:** Incorporating larger and more diverse datasets, possibly combining multiple sources or collecting real-world emotional speech data.
2. **Advanced Neural Architectures:** Exploring more sophisticated neural network architectures, such as attention mechanisms or transformer models, which have shown promise in various speech processing tasks.
3. **Multi-modal Approach:** Integrating other modalities (e.g., text transcripts, visual cues) for a more comprehensive emotion recognition system.
4. **Extensive Hyperparameter Optimization:** Employing advanced hyperparameter tuning techniques (e.g., Bayesian optimization) for both DTW-based and neural network models.
5. **Real-time Emotion Recognition:** Adapting our models for real-time processing, which could open up applications in live customer service or mental health monitoring.
6. **Cross-cultural Emotion Recognition:** Investigating how well our models generalize across different languages and cultures, and adapting them for cross-cultural emotion recognition.
7. **Multi-label Classification:** Extending our models to recognize multiple emotions simultaneously, acknowledging that human emotional expressions are often complex and may involve a blend of different emotions. This approach could provide a more nuanced understanding of emotional states in speech.

D. Personal Learning Outcomes and Project Value

This project has been an invaluable learning experience, providing insights into various aspects of audio processing, machine learning, and emotion recognition:

1. **Audio Feature Extraction:** I gained practical experience in extracting and analyzing various audio features, understanding their relevance to emotion recognition.
2. **Time Series Analysis:** Implementing and comparing DTW algorithms deepened my understanding of time series alignment and its applications.

3. **Neural Network Design:** Designing and implementing a custom neural network architecture for audio processing enhanced my skills in deep learning.
4. **Comparative Analysis:** The project improved my ability to design experiments, compare different algorithms, and critically analyze results.
5. **Interdisciplinary Application:** This work highlighted the intersection of signal processing, machine learning, and psychology, showcasing the potential of AI in understanding human emotions.

IV. References

1. Salvador, S., & Chan, P. (2007). FastDTW: Toward Accurate Dynamic Time Warping in Linear Time and Space. Intelligent Data Analysis, 11(5), 561-580.
2. Müller, M. (2007). Dynamic Time Warping. In Information Retrieval for Music and Motion (pp. 69-84). Springer Berlin Heidelberg.

V. Appendix

Appendix A: Code Repository

The complete code for this project is available in the following GitHub repository:

Simple Emotion Detection

This repository contains all the Python scripts, including data preparation, feature extraction, DTW implementations, and neural network models discussed in this report.

Appendix B: Key Functions

Here are some key functions from our implementation:

B.1 Classical DTW

```
def classical_dtw(x, y, dist=euclidean):
    nx, ny = len(x), len(y)
    cost_matrix = np.zeros((nx + 1, ny + 1))
    cost_matrix[0, 1:] = np.inf
    cost_matrix[1:, 0] = np.inf

    for i in range(1, nx + 1):
        for j in range(1, ny + 1):
            cost = dist(x[i - 1], y[j - 1])
            cost_matrix[i, j] = cost + min(
                cost_matrix[i - 1, j], cost_matrix[i, j - 1], cost_matrix[i - 1,
j - 1]
            )

    return cost_matrix[nx, ny]
```

B.2 Feature Extraction

```python
def extract_features(y, sr, n=13):
    mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=n)
    chroma = librosa.feature.chroma_stft(y=y, sr=sr)
    mel = librosa.feature.melspectrogram(y=y, sr=sr)
    contrast = librosa.feature.spectral_contrast(y=y, sr=sr)
    tonnetz = librosa.feature.tonnetz(y=librosa.effects.harmonic(y), sr=sr)
    spectral_centroid = librosa.feature.spectral_centroid(y=y, sr=sr)
    zero_crossing_rate = librosa.feature.zero_crossing_rate(y)
    rms = librosa.feature.rms(y=y)

    # ... [code to compute mean and std for each feature]

    return {
        "mfccs": mfccs,
        "chroma": chroma,
        "mel": mel,
        "contrast": contrast,
        "tonnetz": tonnetz,
        "spectral_centroid": spectral_centroid,
        "zero_crossing_rate": zero_crossing_rate,
        "rms": rms,
        "combined_features": combined_features,
    }
```

B.3 Neural Network Model

```python
class EmotionClassifier(nn.Module):
    def __init__(self, mfcc_dim, stat_dim, hidden_size, num_classes):
        super(EmotionClassifier, self).__init__()
        self.lstm = nn.LSTM(mfcc_dim, hidden_size, batch_first=True)
        self.fc1 = nn.Linear(hidden_size + stat_dim, 64)
        self.fc2 = nn.Linear(64, num_classes)
        self.relu = nn.ReLU()

    def forward(self, mfccs, stat_features):
        _, (hidden, _) = self.lstm(mfccs)
        combined = torch.cat((hidden.squeeze(0), stat_features), dim=1)
        x = self.relu(self.fc1(combined))
        x = self.fc2(x)
        return x
```