Table of Contents

1.0 Introduction

1.1 Objective

The goal of this project is to detect speaker changes in the IEMOCAP dataset using fundamental frequency (F0) and MFCC features. This involves feature extraction, data preprocessing, model development, and performance evaluation to classify speaker changes accurately.

2.0 Understanding the Dataset

The IEMOCAP dataset consists of dialog sessions with two speakers. The dataset includes audio recordings, speaker labels, and transcript information, providing a rich resource for speaker change detection. Each dialog session contains multiple audio files with different speakers. The speaker labels stored as .rttm files in the reference folder specify the identity of the speaker at different timestamps. We will be mainly using features such as fundamental frequency (F0) and Mel-Frequency Cepstral Coefficients (MFCCs) for speaker change detection.

For this assignment, the ideas are inspired by "Speaker Change Detection in Broadcast TV Using Bidirectional Long Short-Term Memory Networks"[1] by Ruiqing Yin et al. This study treats speaker change detection as a sequence labeling problem using Bidirectional Long Short-Term Memory (Bi-LSTM) networks. Their approach demonstrates the advantages of deep learning over conventional methods like Gaussian divergence and Bayesian Information Criterion (BIC), which rely on adjacent sliding windows for segmentation.

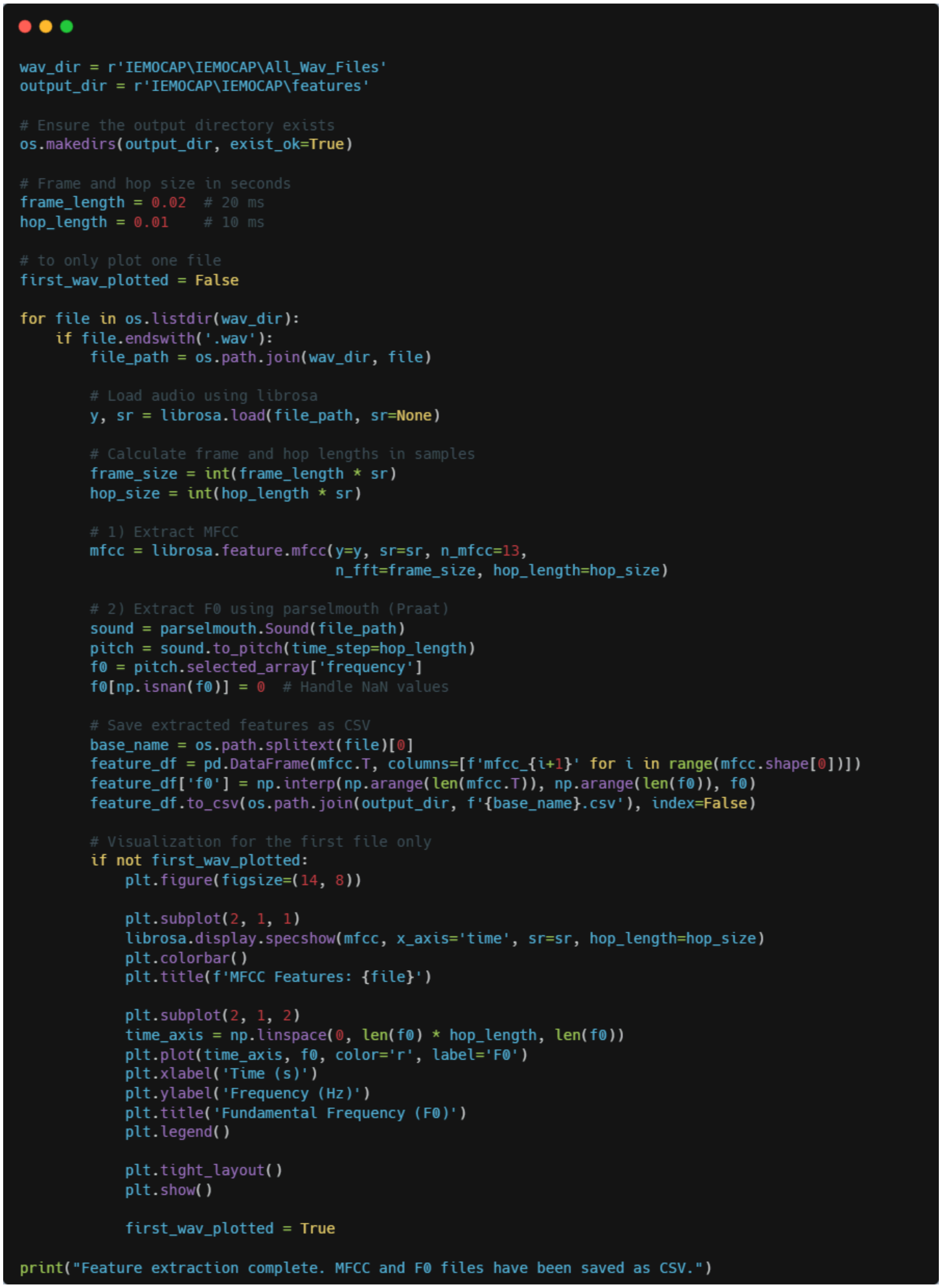
The Bi-LSTM model proposed in the paper is trained to classify each frame in an audio sequence as either a speaker change (1) or no change (0). Instead of using fixed-length adjacent windows like traditional methods, Bi-LSTM networks process the entire sequence in both forward and backward directions, enabling a more contextual understanding of speaker transitions.

3.0 Feature Extraction

To correctly detect speaker change, the extracted features directly impact the effectiveness of classification models. In this study, we utilize Mel-Frequency Cepstral Coefficients (MFCCs) and Fundamental Frequency (F0) as the primary features for detecting speaker transitions. These features are widely used in speech processing due to their ability to capture essential characteristics of human speech.

Fundamental Frequency (F0) represents the pitch of a speaker's voice and is a key indicator of speaker identity. Since F0 primarily captures the voiced components of speech, silent segments do not register any meaningful F0 values. To maintain consistency in feature representation, silent regions are assigned an F0 value of zero. This ensures that non-voiced segments do not interfere with the speaker change detection process. The extraction of F0 is carried out using parselmouth, a wrapper for the Praat speech processing tool, which provides precise pitch tracking over time.

MFCCs are used to capture the spectral characteristics of speech. These coefficients are derived from the short-time power spectrum of the audio signal, making them effective in representing the phonetic content of speech. In this study, we extract 13 MFCC coefficients per frame using librosa, a Python library for audio analysis. The MFCCs provide essential information on speaker-specific variations in speech signals, helping distinguish between different speakers.



```python
wav_dir = r'IEMOCAP\IEMOCAP\All_Wav_Files'
output_dir = r'IEMOCAP\IEMOCAP\features'

# Ensure the output directory exists
os.makedirs(output_dir, exist_ok=True)

# Frame and hop size in seconds
frame_length = 0.02  # 20 ms
hop_length = 0.01    # 10 ms

# to only plot one file
first_wav_plotted = False

for file in os.listdir(wav_dir):
    if file.endswith('.wav'):
        file_path = os.path.join(wav_dir, file)

        # Load audio using librosa
        y, sr = librosa.load(file_path, sr=None)

        # Calculate frame and hop lengths in samples
        frame_size = int(frame_length * sr)
        hop_size = int(hop_length * sr)

        # 1) Extract MFCC
        mfcc = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13,
                                    n_fft=frame_size, hop_length=hop_size)

        # 2) Extract F0 using parselmouth (Praat)
        sound = parselmouth.Sound(file_path)
        pitch = sound.to_pitch(time_step=hop_length)
        f0 = pitch.selected_array['frequency']
        f0[np.isnan(f0)] = 0  # Handle NaN values

        # Save extracted features as CSV
        base_name = os.path.splitext(file)[0]
        feature_df = pd.DataFrame(mfcc.T, columns=[f'mfcc_{i+1}' for i in range(mfcc.shape[0])])
        feature_df['f0'] = np.interp(np.arange(len(mfcc.T)), np.arange(len(f0)), f0)
        feature_df.to_csv(os.path.join(output_dir, f'{base_name}.csv'), index=False)

        # Visualization for the first file only
        if not first_wav_plotted:
            plt.figure(figsize=(14, 8))

            plt.subplot(2, 1, 1)
            librosa.display.specshow(mfcc, x_axis='time', sr=sr, hop_length=hop_size)
            plt.colorbar()
            plt.title(f'MFCC Features: {file}')

            plt.subplot(2, 1, 2)
            time_axis = np.linspace(0, len(f0) * hop_length, len(f0))
            plt.plot(time_axis, f0, color='r', label='F0')
            plt.xlabel('Time (s)')
            plt.ylabel('Frequency (Hz)')
            plt.title('Fundamental Frequency (F0)')
            plt.legend()

            plt.tight_layout()
            plt.show()

            first_wav_plotted = True

print("Feature extraction complete. MFCC and F0 files have been saved as CSV.")
```

Figure 3.0.1 Code for Feature Extraction

The code above shows the feature extraction process is performed using a sliding window approach to ensure temporal continuity in the extracted features. A frame length of 20 milliseconds with a 10-millisecond hop size is used, allowing for smooth transitions between consecutive frames. The MFCC and F0 features are computed for each frame and stored in structured CSV files for further processing. However note that different frame lengths and hop sizes are experimented to yield the best predictions later on.

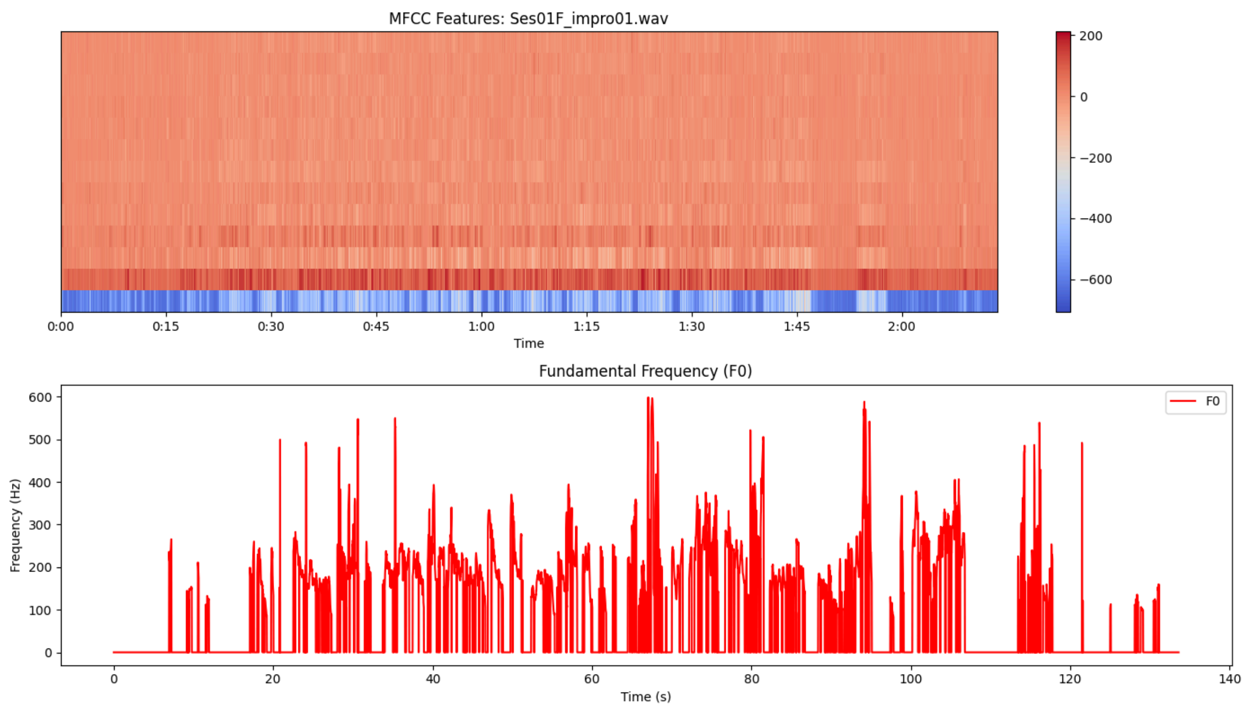
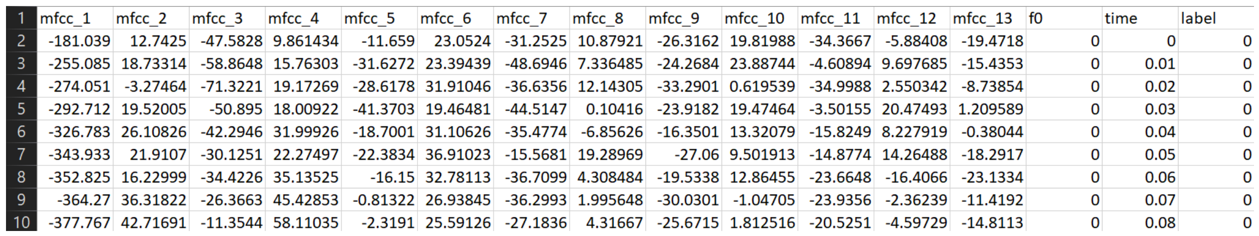


Figure 3.0.2 MFCC and F0 Over Time

To verify the accuracy of feature extraction, we visualize the extracted MFCC and F0 values for a sample audio file. The MFCCs are represented as a spectrogram, providing a time-frequency representation of speech, while the F0 contour is plotted as a time-series graph, illustrating pitch variations over time. These visualizations confirm the successful extraction of features, with clear distinctions in spectral and pitch characteristics between different speakers.



1	mfcc_1	mfcc_2	mfcc_3	mfcc_4	mfcc_5	mfcc_6	mfcc_7	mfcc_8	mfcc_9	mfcc_10	mfcc_11	mfcc_12	mfcc_13	f0	time	label
2	-181.039	12.7425	-47.5828	9.861434	-11.659	23.0524	-31.2525	10.87921	-26.3162	19.81988	-34.3667	-5.88408	-19.4718	0	0	0
3	-255.085	18.73314	-58.8648	15.76303	-31.6272	23.39439	-48.6946	7.336485	-24.2684	23.88744	-4.60894	9.697685	-15.4353	0	0.01	0
4	-274.051	-3.27464	-71.3221	19.17269	-28.6178	31.91046	-36.6356	12.14305	-33.2901	0.619539	-34.9988	2.550342	-8.73854	0	0.02	0
5	-292.712	19.52005	-50.895	18.00922	-41.3703	19.46481	-44.5147	0.10416	-23.9182	19.47464	-3.50155	20.47493	1.209589	0	0.03	0
6	-326.783	26.10826	-42.2946	31.99926	-18.7001	31.10626	-35.4774	-6.85626	-16.3501	13.32079	-15.8249	8.227919	-0.38044	0	0.04	0
7	-343.933	21.9107	-30.1251	22.27497	-22.3834	36.91023	-15.5681	19.28969	-27.06	9.501913	-14.8774	14.26488	-18.2917	0	0.05	0
8	-352.825	16.22999	-34.4226	35.13525	-16.15	32.78113	-36.7099	4.308484	-19.5338	12.86455	-23.6648	-16.4066	-23.1334	0	0.06	0
9	-364.27	36.31822	-26.3663	45.42853	-0.81322	26.93845	-36.2993	1.995648	-30.0301	-1.04705	-23.9356	-2.36239	-11.4192	0	0.07	0
10	-377.767	42.71691	-11.3544	58.11035	-2.3191	25.59126	-27.1836	4.31667	-25.6715	1.812516	-20.5251	-4.59729	-14.8113	0	0.08	0

Figure 3.0.3 First 10 Lines of the Dataset with Extracted Features

The first 10 lines of the dataset with extracted 13 MFCC coefficients and F0 are shown.

3.1 Labelling

3.1.1 Problem with Given Labels

Based on the .rttm files in the reference folder, which record the start time and duration of each speaker, speaker changes can be identified using the start times. However, a thorough manual inspection of the audio reveals that the start times in the .rttm files are not entirely accurate. For instance, in Ses01F_impro01, the label indicates that the speaker begins speaking at 6.28 seconds, whereas manual verification shows the actual speech onset occurs around 6.9 seconds. This discrepancy highlights labeling inaccuracies, which significantly contribute to the model's reduced performance. Since manual labeling is impractical for the entire dataset, the provided .rttm files are still being used for speaker change labeling despite these inaccuracies.

3.1.2 Class Imbalances

Initial analysis of the labels indicates a highly imbalanced class distribution, with class 1, representing speaker changes, accounting for only 0.3% of the entire dataset. The workaround is to add more neighbouring classes as proposed in the research paper mentioned above.

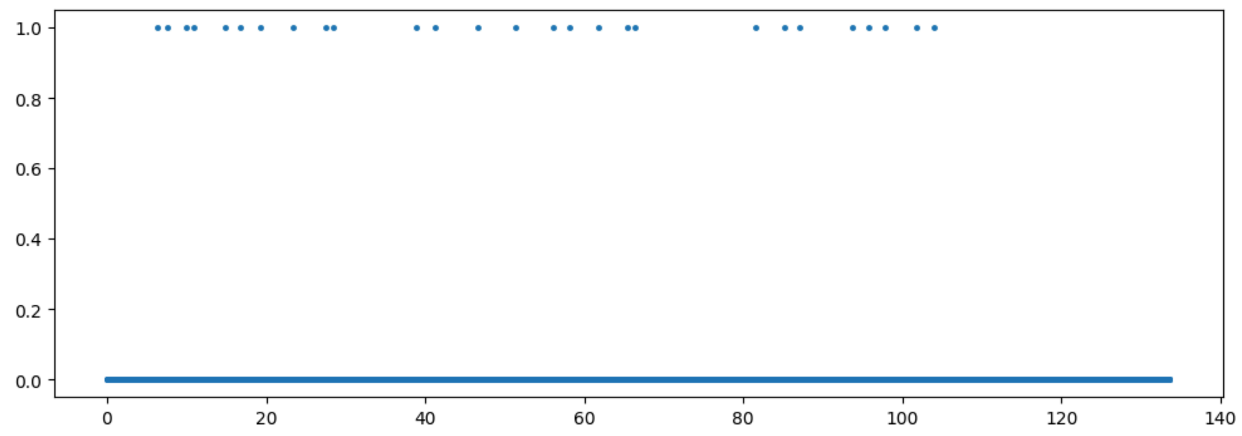


Figure 3.1.1 Class Labelling Over Time in Ses01F_impro01 Before Adding Class Neighbour

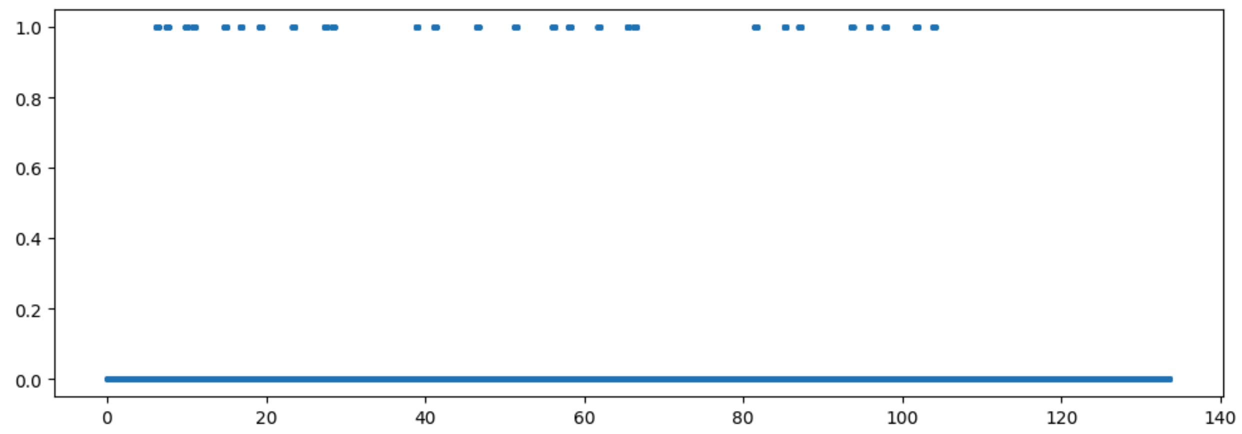
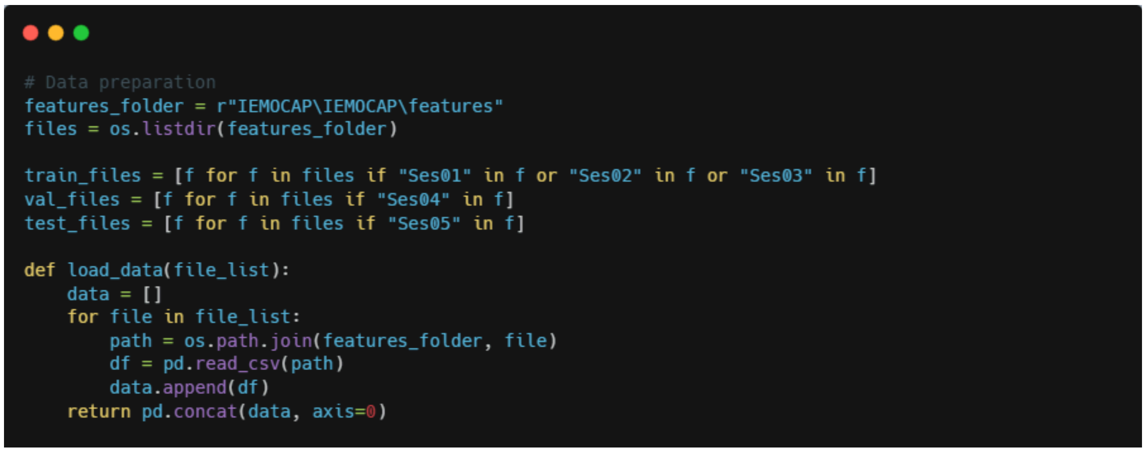


Figure 3.1.2 Class Labelling Over Time in Ses01F_impro01 After Adding Class Neighbour

The figure above shows the comparison of the class label over time in Ses01F_impro01 with and without adding neighbouring classes. Adding the class neighbours up to 150 ms bumped up the percentage of minority class which is class 1 to 6.37%. Which is still highly imbalanced but this step helps to reduce the imbalance.

4.0 Data Preprocessing

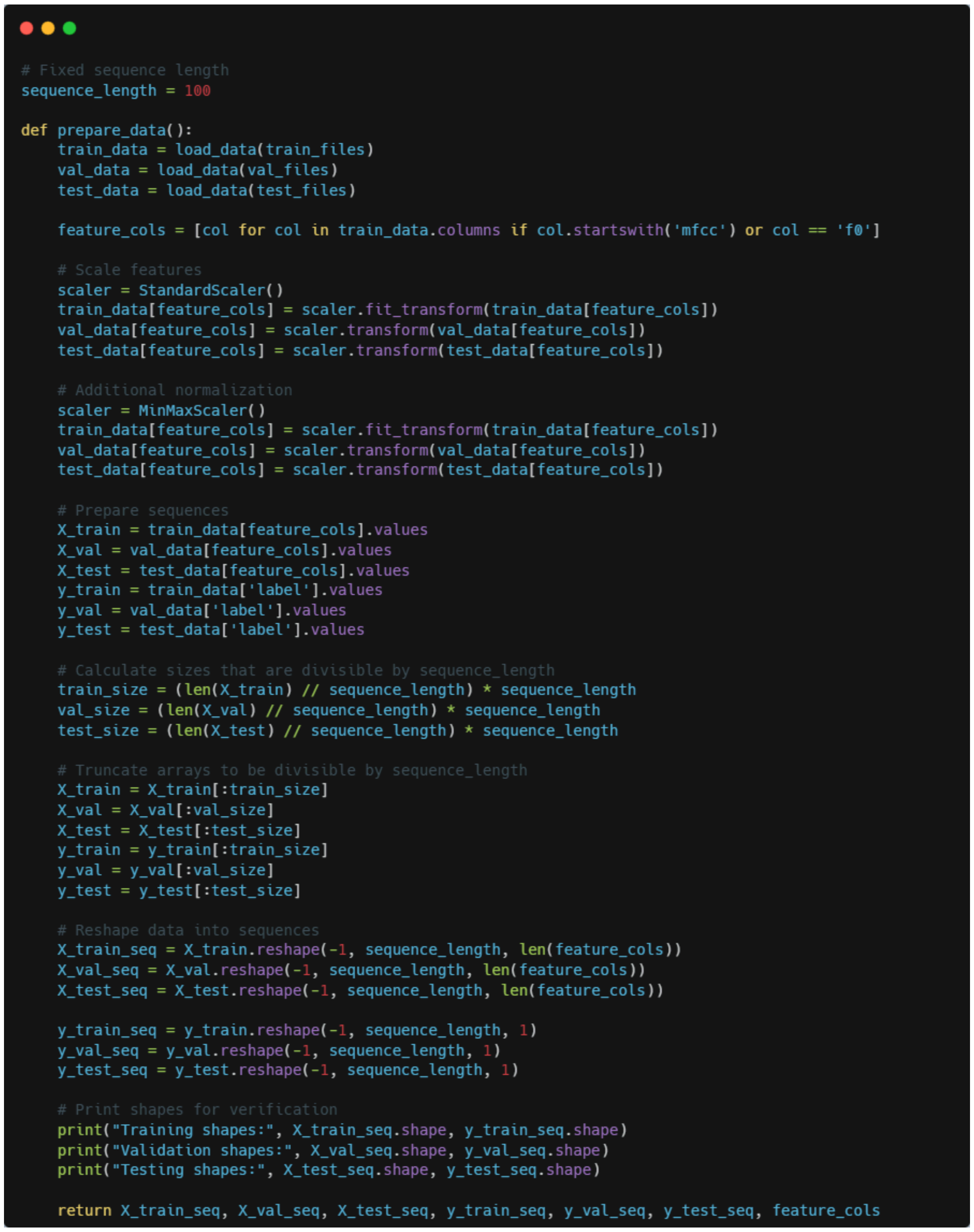
The quality and structure of the input data significantly influence the performance of the speaker change detection model. The preprocessing phase involves loading, normalizing, and structuring the dataset into sequences suitable for deep learning models.



```python
# Data preparation
features_folder = r"IEMOCAP\IEMOCAP\features"
files = os.listdir(features_folder)

train_files = [f for f in files if "Ses01" in f or "Ses02" in f or "Ses03" in f]
val_files = [f for f in files if "Ses04" in f]
test_files = [f for f in files if "Ses05" in f]

def load_data(file_list):
    data = []
    for file in file_list:
        path = os.path.join(features_folder, file)
        df = pd.read_csv(path)
        data.append(df)
    return pd.concat(data, axis=0)
```

Figure 4.0.1 Loading the Data

The processed dataset consists of multiple CSV files, each containing extracted F0 features along with labels indicating speaker change points. These files are first loaded using the load_data function, which concatenates them into a single dataset for each of the training, validation, and test sets. Since the dataset is derived from different dialog sessions, merging these files ensures that the model is trained on diverse speaker transitions, improving generalization.



```python
# Fixed sequence length
sequence_length = 100

def prepare_data():
    train_data = load_data(train_files)
    val_data = load_data(val_files)
    test_data = load_data(test_files)

    feature_cols = [col for col in train_data.columns if col.startswith('mfcc') or col == 'f0']

    # Scale features
    scaler = StandardScaler()
    train_data[feature_cols] = scaler.fit_transform(train_data[feature_cols])
    val_data[feature_cols] = scaler.transform(val_data[feature_cols])
    test_data[feature_cols] = scaler.transform(test_data[feature_cols])

    # Additional normalization
    scaler = MinMaxScaler()
    train_data[feature_cols] = scaler.fit_transform(train_data[feature_cols])
    val_data[feature_cols] = scaler.transform(val_data[feature_cols])
    test_data[feature_cols] = scaler.transform(test_data[feature_cols])

    # Prepare sequences
    X_train = train_data[feature_cols].values
    X_val = val_data[feature_cols].values
    X_test = test_data[feature_cols].values
    y_train = train_data['label'].values
    y_val = val_data['label'].values
    y_test = test_data['label'].values

    # Calculate sizes that are divisible by sequence_length
    train_size = (len(X_train) // sequence_length) * sequence_length
    val_size = (len(X_val) // sequence_length) * sequence_length
    test_size = (len(X_test) // sequence_length) * sequence_length

    # Truncate arrays to be divisible by sequence_length
    X_train = X_train[:train_size]
    X_val = X_val[:val_size]
    X_test = X_test[:test_size]
    y_train = y_train[:train_size]
    y_val = y_val[:val_size]
    y_test = y_test[:test_size]

    # Reshape data into sequences
    X_train_seq = X_train.reshape(-1, sequence_length, len(feature_cols))
    X_val_seq = X_val.reshape(-1, sequence_length, len(feature_cols))
    X_test_seq = X_test.reshape(-1, sequence_length, len(feature_cols))

    y_train_seq = y_train.reshape(-1, sequence_length, 1)
    y_val_seq = y_val.reshape(-1, sequence_length, 1)
    y_test_seq = y_test.reshape(-1, sequence_length, 1)

    # Print shapes for verification
    print("Training shapes:", X_train_seq.shape, y_train_seq.shape)
    print("Validation shapes:", X_val_seq.shape, y_val_seq.shape)
    print("Testing shapes:", X_test_seq.shape, y_test_seq.shape)

    return X_train_seq, X_val_seq, X_test_seq, y_train_seq, y_val_seq, y_test_seq, feature_cols
```

Figure 4.0.2 Processing the Loaded Data

Once the dataset is loaded, feature scaling is performed to ensure uniformity across different feature values. First, StandardScaler is applied to normalize the F0 feature by removing the mean and scaling it to unit variance. This transformation standardizes the feature distribution, reducing the impact of extreme values. Following this, MinMaxScaler is applied to further normalize the data into a range between 0 and 1, ensuring numerical stability and preventing biases in model learning.

The dataset is segmented into smaller chunks. The sequence_length parameter, set to 100, determines the size of each sequence. The dataset is then truncated to ensure that its size is a multiple of the chosen sequence length. This prevents uneven sequences that could disrupt model training. The data is then reshaped into a three-dimensional structure, where each entry represents a segment of 100 consecutive frames, with each frame containing a single feature value for F0. Note that the sequence_length parameter is to be experimented with different values to yield the most accurate model.

The labels are also reshaped to match the sequence format. Instead of treating individual frames as separate samples, grouping them into sequences allows the model to learn temporal dependencies. Speaker transitions often occur gradually, and structuring the data in sequences provides contextual information, improving classification accuracy.

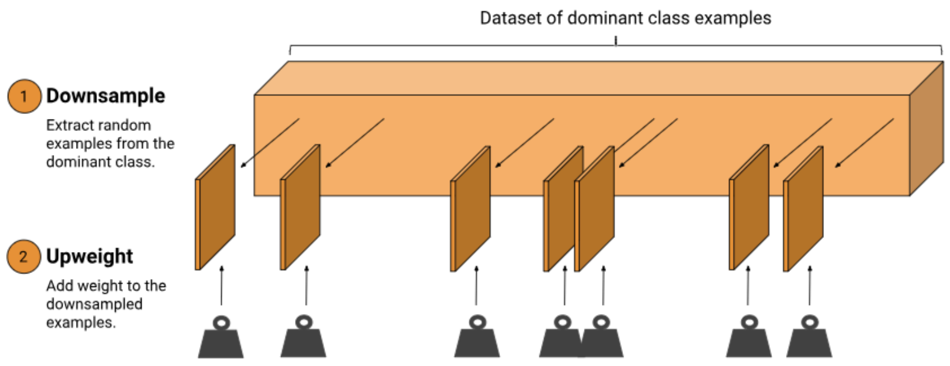
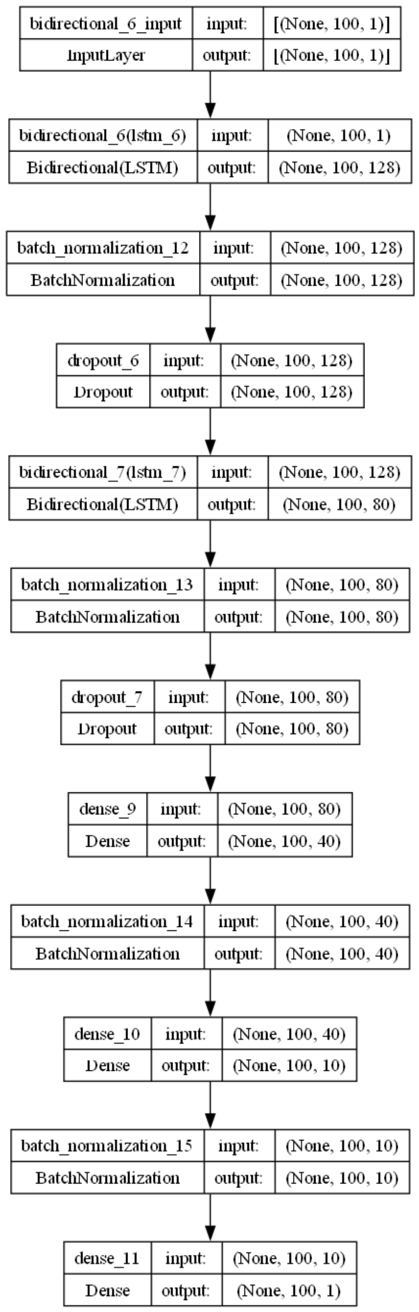

Dataset of dominant class examples
1 Downsample
Extract random examples from the dominant class.
2 Upweight
Add weight to the downsampled examples.

Figure 4.0.3 Downsampling and Upweighting [2]

As discussed in resources such as Google's Machine Learning Crash Course. One approach to mitigate the effect of imbalanced data is to undersample the majority class, which entails reducing the number of non-speaker change instances to achieve a more balanced class distribution. This method helps prevent the model from becoming biased towards the majority class. However, a potential drawback of undersampling is the inadvertent removal of informative samples from the majority class, which could lead to a loss of valuable information. In addition to undersampling, the technique of upweighting the majority class was applied. This approach assigns higher weights to the majority class during model training, thereby increasing its influence on the learning process to prevent bias to the minority class by the model. By upweighting the majority class, the model is encouraged to pay more attention to these instances, which can help mitigate the effects of class imbalance.[2]

5.0 Model Development

5.1 Implementation Details



bidirectional_6_input | input: | [(None, 100, 1)]
InputLayer | output: | [(None, 100, 1)]
bidirectional_6(lstm_6) | input: | (None, 100, 1)
Bidirectional(LSTM) | output: | (None, 100, 128)
batch_normalization_12 | input: | (None, 100, 128)
BatchNormalization | output: | (None, 100, 128)
dropout_6 | input: | (None, 100, 128)
Dropout | output: | (None, 100, 128)
bidirectional_7(lstm_7) | input: | (None, 100, 128)
Bidirectional(LSTM) | output: | (None, 100, 80)
batch_normalization_13 | input: | (None, 100, 80)
BatchNormalization | output: | (None, 100, 80)
dropout_7 | input: | (None, 100, 80)
Dropout | output: | (None, 100, 80)
dense_9 | input: | (None, 100, 80)
Dense | output: | (None, 100, 40)
batch_normalization_14 | input: | (None, 100, 40)
BatchNormalization | output: | (None, 100, 40)
dense_10 | input: | (None, 100, 40)
Dense | output: | (None, 100, 10)
batch_normalization_15 | input: | (None, 100, 10)
BatchNormalization | output: | (None, 100, 10)
dense_11 | input: | (None, 100, 10)
Dense | output: | (None, 100, 1)

Figure 4.0.2 Architecture of LSTM

The speaker change detection model is developed using a Bidirectional Long Short-Term Memory (Bi-LSTM) network. Bi-LSTMs are well-suited for sequence-based tasks, as they can capture both past and future contextual dependencies, making them particularly effective for detecting transitions in speech. [3]

The model architecture consists of two stacked Bidirectional LSTM layers, with 64 and 40 units respectively, allowing it to process temporal dependencies from both forward and backward directions. Batch normalization layers are incorporated after each LSTM layer to stabilize learning and improve convergence. Dropout layers with a dropout rate of 0.5 are applied to prevent overfitting and enhance generalization.

Batch normalization is a crucial component of this architecture. It normalizes activations within each batch during training, reducing internal covariate shift. This improves model stability by ensuring that activations remain within a reasonable range, allowing the network to learn more efficiently. By adjusting and scaling activations dynamically, batch normalization facilitates faster convergence and improves generalization, especially in deep networks like Bi-LSTMs. The inclusion of batch normalization after each LSTM and dense layer helps mitigate the issue of vanishing or exploding gradients, leading to more stable training dynamics.

Following the LSTM layers, the model includes two fully connected dense layers, with 40 and 10 neurons respectively, each followed by batch normalization to ensure stability in weight updates. The final output layer consists of a single neuron with a sigmoid activation function, responsible for binary classification of speaker changes. The model is compiled using binary cross-entropy loss, which is appropriate for handling binary classification tasks.

To improve generalization and model accuracy, 5-Fold Cross-Validation (KFolds=5) is implemented. This technique ensures that the model is trained and validated on different subsets of the dataset, reducing variance and improving performance evaluation. Each fold serves as a validation set once, while the remaining data is used for training. This prevents the model from relying too heavily on specific training samples and helps assess its performance more comprehensively.

During model training, early stopping is applied, monitoring the validation loss to halt training if performance degrades, thereby preventing overfitting. Additionally, ReduceLROnPlateau is used to dynamically adjust the learning rate, reducing it when the validation loss plateaus to ensure stable convergence.

Initially, SMORMS3 optimization was tested, but it did not yield significant improvements. SMORMS3 (Sqrt Mean Over Root Mean Square) is an adaptive learning rate optimization algorithm designed to efficiently update network weights while avoiding issues of vanishing gradients. Unlike traditional stochastic gradient descent, it dynamically adjusts learning rates for individual parameters based on their historical gradients, preventing oscillations and promoting faster convergence. However, despite its theoretical advantages in adaptive optimization, the implementation in this model did not result in superior performance compared to more established optimizers like Adam. Therefore, Adam optimizer was selected due to its adaptive moment estimation, which adjusts learning rates effectively and is known for its efficient performance in deep learning applications. [4]

The model was trained using a batch size of 700 for 100 epochs, with performance being evaluated using 5-fold cross-validation and validation data. After training, the model was saved to facilitate further analysis and deployment.

5.2 Model training and validation

The performance of the Bi-LSTM model for speaker change detection is assessed through training and validation loss as well as accuracy trends over multiple epochs. The provided plots for different batch sizes illustrate the model's learning process and generalization capability.

5.2.1 Batch Size = 700

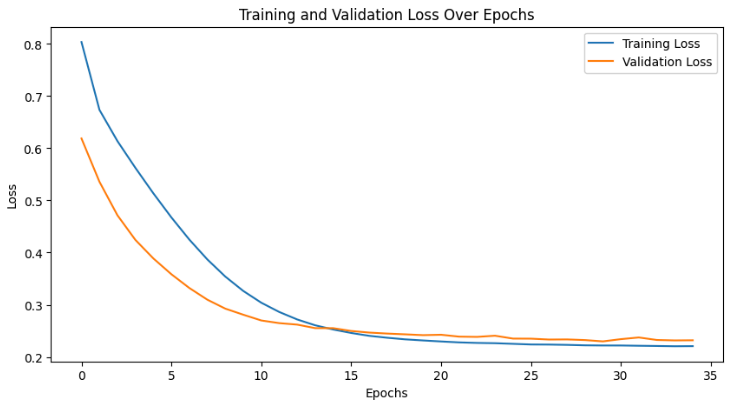


Figure 5.2.1.1 Training and Validation Loss for batch size of 700

The first graph presents the training and validation loss over 35 epochs. Initially, the training loss starts at a high value of approximately 0.8, while the validation loss begins at around 0.6. Both losses exhibit a downward trend as training progresses, indicating that the model is learning from the data.

Around epoch 15, the validation loss begins to stabilize, closely following the training loss. This suggests that the model generalizes well to unseen data without significant overfitting. If the validation loss had diverged while training loss continued decreasing, it would indicate overfitting. However, in this case, both losses plateau at a similar level, confirming that the model is well-regularized and learns effectively.

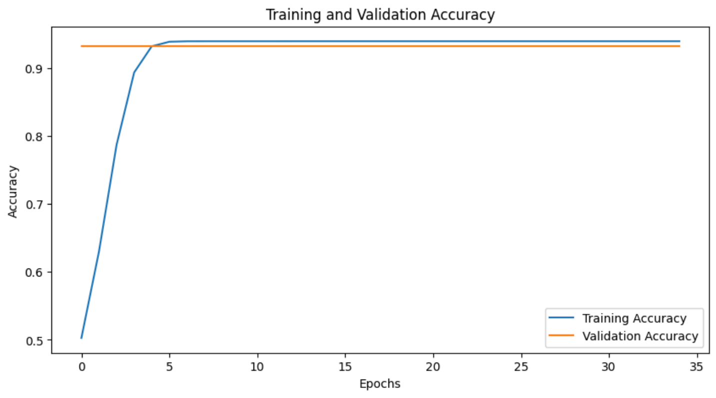


Figure 5.2.1.2 Training and Validation Accuracy for batch size of 700

The displays training and validation accuracy over epochs. Initially, the training accuracy starts at 50%, rapidly increasing to over 90% within the first 10 epochs. However, the validation accuracy remains almost constant after a few epochs, not improving significantly. If it is consistently lower than the training accuracy and does not improve, it indicates overfitting.

After epoch 5, the training and validation accuracy converge, reaching a stable level of approximately 95%. The absence of a large gap between training and validation accuracy suggests that the model does not suffer from overfitting and maintains strong generalization.

5.2.2 Batch Size = 32

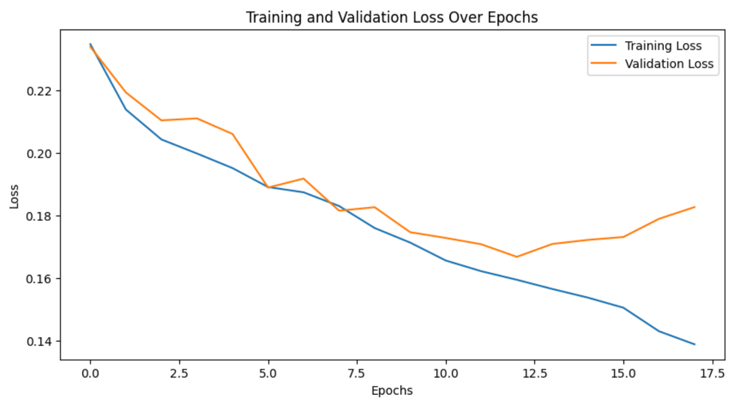


Figure 5.2.2.1 Training and Validation Loss for batch size of 32

The first graph plots training loss and validation loss against the number of epochs. The training loss exhibits a steady downward trend, indicating that the model is effectively learning from the training data. This aligns with the expectation that as the model adjusts its weights, it reduces the error on the training set. The validation loss initially follows a decreasing trend, which suggests that the model is generalizing well. However, after epoch 6-7, the validation loss begins to fluctuate and slightly increase, indicating potential overfitting and is halted by early stopping. The best weights are also restored upon the halt of overfitting.

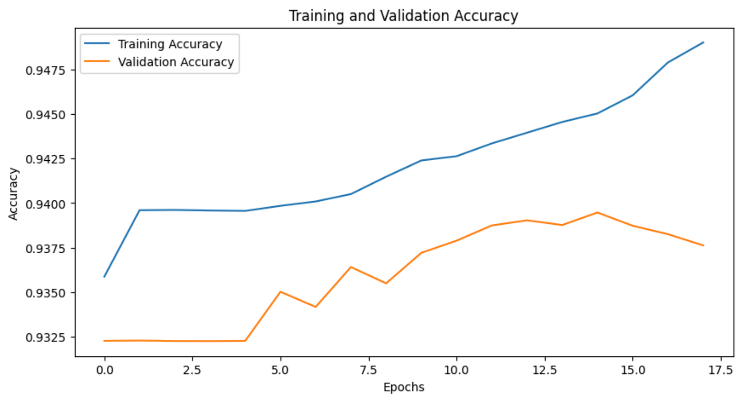


Figure 5.2.2.1 Training and Validation Accuracy for batch size of 32

The second graph shows the accuracy trends for both training and validation data across epochs. The model achieves a high training accuracy, increasing consistently and approaching 0.95. The validation accuracy also follows a similar upward trend initially but plateaus and fluctuates after epoch 10-12.

6.0 Results and Discussion

The evaluation of the speaker change detection model is conducted using various performance metrics, including test accuracy, F1-score, precision, and recall. The primary objective is to analyze how different sequence lengths affect model performance and to identify the optimal sequence length for detecting speaker transitions. The evaluation process involves training multiple models with varying sequence lengths and selecting the best-performing model based on the highest F1-score.

6.1 Performance Across Sequence Lengths

The evaluation results for different sequence lengths (50, 100, 150, 200, 250, and 300) are summarized using four key performance metrics.

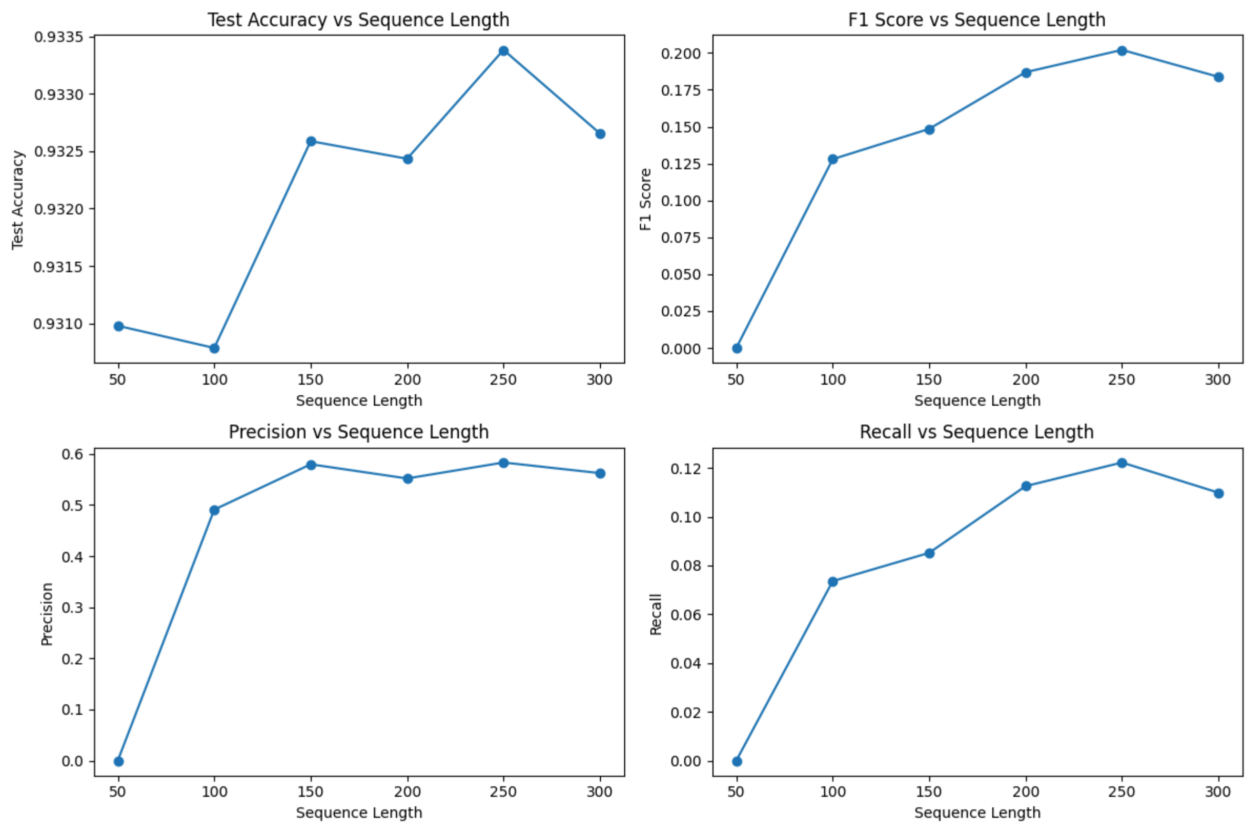


Figure 6.1.1 Metrics across different sequence length

Metrics	Analysis
Accuracy	● The test accuracy generally increases with longer sequence lengths, peaking at sequence length 250. ● A minor fluctuation is observed at sequence length 100, possibly due to the model struggling to capture contextual dependencies effectively at shorter lengths. ● The overall trend suggests that longer sequences provide richer contextual information, enhancing the model's ability to detect speaker transitions. ● However, beyond sequence length 250, the improvement is minimal, indicating diminishing returns in accuracy as sequence length increases.
F1 Score	● The F1-score improves progressively with increasing sequence length, peaking at sequence length 250, and then slightly declines at sequence length 300. ● The decline beyond 250 suggests that excessive sequence length may introduce unnecessary complexity, leading to marginal deterioration in classification performance. ● This trend highlights the importance of optimizing sequence length to maintain a balance between contextual awareness and model efficiency.
Precision	● Precision improves steadily up to sequence length 150, indicating that the model gains confidence in its predictions. ● However, beyond 150, a slight decline is observed, with a notable drop at sequence length 200 and 250. ● This suggests that while longer sequences improve contextual understanding, they may also increase false positive rates, where non-change events are misclassified as speaker changes. ● The decline in precision at higher sequence lengths indicates a trade-off between capturing transitions and avoiding unnecessary detections.
Recall	● Recall consistently improves with longer sequence lengths, reaching its highest value at sequence length 250. ● This improvement suggests that longer sequences allow the model to detect more speaker change events, thereby reducing false negatives. ● However, a slight decline is observed after sequence length 250, likely due to excessive sequence length introducing noise or redundant information that weakens the model's ability to focus on crucial transition points. ● The diminishing return in recall at longer sequences highlights the need to balance sequence length to prevent unnecessary complexity.

The sequence length of 250 with the highest F1 score is selected for further evaluation.

6.2 Threshold Tuning for F1-Score

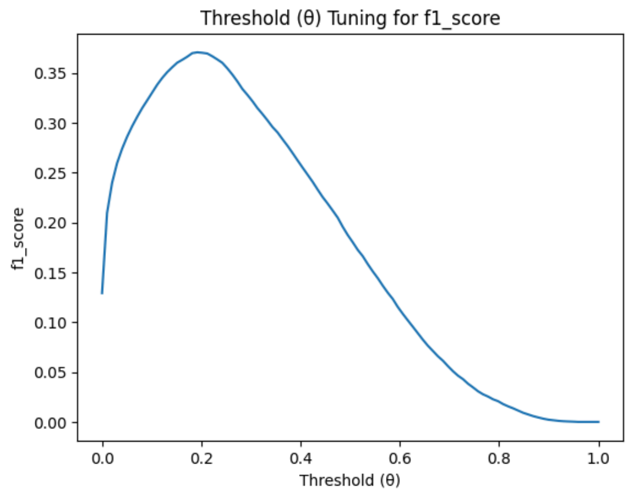
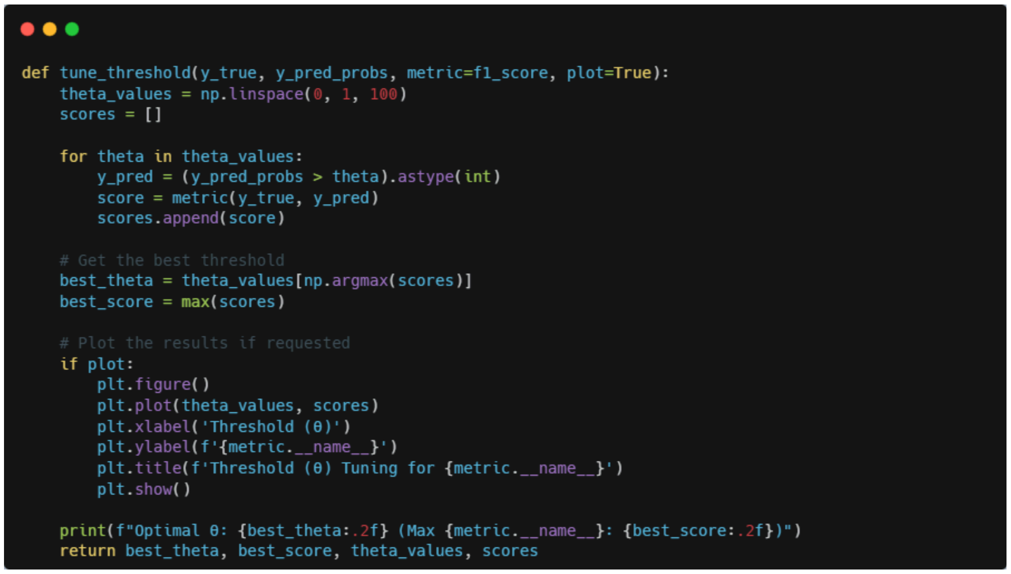


Figure 6.2.1 F1 score across theta

One of the critical aspects of binary classification is determining an optimal decision threshold (θ). The threshold tuning curve shows the variation of the F1-score as the threshold changes. Based on *figure 6.2.1*, the F1-score peaks at θ = 0.19 with F1 score of 0.37, indicating that this threshold provides the best balance between precision and recall. Beyond θ = 0.19, the F1-score decreases, indicating that stricter thresholds result in higher false negatives, while lower thresholds increase false positives. Selecting θ = 0.19 ensures that the model achieves optimal detection accuracy without excessive misclassifications.

The function of threshold tuning is as follows.



```python
def tune_threshold(y_true, y_pred_probs, metric=f1_score, plot=True):
    theta_values = np.linspace(0, 1, 100)
    scores = []

    for theta in theta_values:
        y_pred = (y_pred_probs > theta).astype(int)
        score = metric(y_true, y_pred)
        scores.append(score)

    # Get the best threshold
    best_theta = theta_values[np.argmax(scores)]
    best_score = max(scores)

    # Plot the results if requested
    if plot:
        plt.figure()
        plt.plot(theta_values, scores)
        plt.xlabel('Threshold (θ)')
        plt.ylabel(f'{metric.__name__}')
        plt.title(f'Threshold (θ) Tuning for {metric.__name__}')
        plt.show()

    print(f"Optimal θ: {best_theta:.2f} (Max {metric.__name__}: {best_score:.2f})")
    return best_theta, best_score, theta_values, scores
```

Figure 6.2.2 Function to tune and plot the threshold

6.3 Classification Report

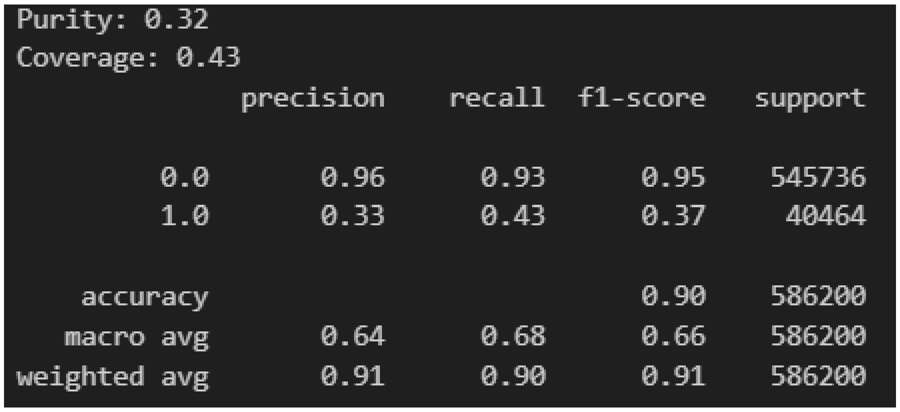

```
Purity: 0.32
Coverage: 0.43
              precision    recall  f1-score   support

         0.0       0.96      0.93      0.95    545736
         1.0       0.33      0.43      0.37     40464

    accuracy                           0.90    586200
   macro avg       0.64      0.68      0.66    586200
weighted avg       0.91      0.90      0.91    586200
```

Figure 6.3.1 Results of the model with best parameters

Based on *figure 6.3.1*, the model with the best parameters achieves an accuracy of 90%, indicating strong generalization. The precision for speaker change detection (Class 1) is significantly lower (0.33), suggesting the model is lacking in predicting Class 1 correctly, precision of 0.33 also shows that when the model predicts Class 1, it is 33% correct all the time. The recall for Class 1 (0.43) is higher than precision, meaning the model captures speaker changes but also includes a high number of false positives.

Purity and coverage are two additional measures used to evaluate the alignment between predicted speaker changes and ground truth which is more suitable for our case.

Measures	Value	Analysis
Purity	0.32	● 32% of the detected speaker change points align correctly with actual speaker changes. ● The relatively low purity means that many predicted speaker changes do not align with actual changes, leading to false positives.
Coverage	0.43	● 43% of actual speaker changes are captured by the model. ● The model correctly identifies some speaker changes but still misses a significant portion.

6.4 ROC and Precision-Recall Curve Analysis

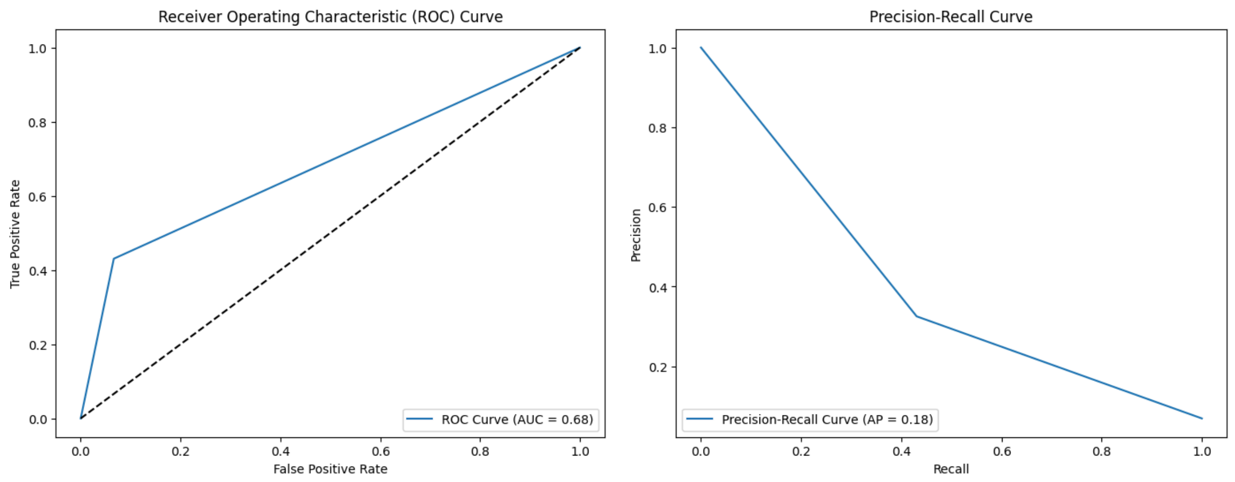


Figure 6.4.1 ROC and Precision-Recall Curve

For the ROC curve that measures the model's ability to distinguish between speaker changes and non-changes. The AUC (Area Under the Curve) = 0.68, indicating moderate classification performance. The model is better than random guessing (AUC = 0.50) but has room for improvement.

Precision-Recall Curve on the other hand shows the Average Precision (AP) of 0.18, meaning that precision drops significantly as recall increases. This justifies that with the low precision observed in the classification report, suggesting that the model struggles to maintain precision while improving recall.

6.5 Speaker Change Detection Visualisation

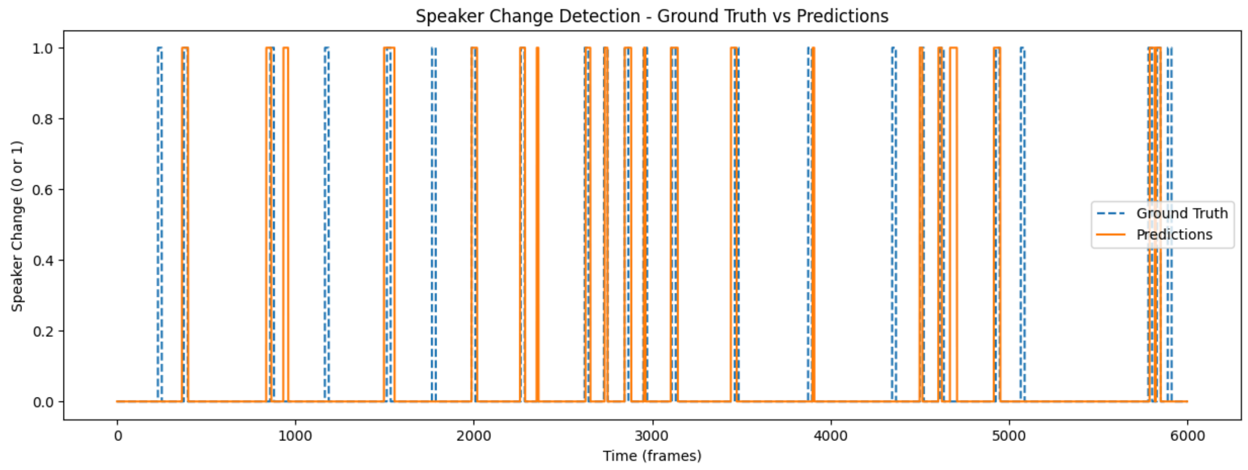


Figure 6.5.1 Ground truth and model prediction for speaker change

The first 6000 frames of final comparison of ground truth with model predictions reveals that the model successfully detects many speaker change events but often introduces additional false positives. Some speaker changes are missed entirely, indicating limitations in recall.

7.0 Conclusion

The speaker change detection model demonstrates promising performance, particularly in recall, but faces challenges in precision and false positive reduction. The analysis of threshold tuning, ROC, Precision-Recall curves, and purity-coverage metrics provides insights into potential areas of improvement. Future work should focus on refining decision thresholds, exploring advanced architectures, and incorporating post-processing techniques to enhance detection accuracy.

8.0 References

[1]R. Yin, H. Bredin, and C. Barras, “Speaker Change Detection in Broadcast TV Using Bidirectional Long Short-Term Memory Networks,” Interspeech 2022, pp. 3827–3831, Aug. 2017, doi: https://doi.org/10.21437/interspeech.2017-65.

[2]“Datasets: Imbalanced datasets,” *Google for Developers*, 2024. https://developers.google.com/machine-learning/crash-course/overfitting/imbalanced-datasets

[3]A. Graves, S. Fernández, and J. Schmidhuber, “Bidirectional LSTM Networks for Improved Phoneme Classification and Recognition,” *Lecture Notes in Computer Science*, pp. 799–804, 2005, doi: https://doi.org/10.1007/11550907_126.

[4]“chainer.optimizers.SMORMS3 — Chainer 7.8.1 documentation,” *Chainer.org*, 2022. https://docs.chainer.org/en/stable/reference/generated/chainer.optimizers.SMORMS3.html (accessed Jan. 26, 2025).