Speech Understanding Programming Assignment 02

AUTHOR : K. K. N. SHYAM SATHVIK

ROLL.NO : B22EE036

LINK TO THE GITHUB REPOSITORY : https://github.com/boku13/SUPA_02/tree/main

Abstract:

This report contains the observations, conclusions and discussions made in the process of performing the tasks done as a part of Speech Understanding Course Programming Assignment 02.

Table of Contents

Question 1:

→ Techniques Used and Evaluation Metrics:

1. Fine-tuning with LoRA and ArcFace:

- ***Low-Rank Adaptation*** (LoRA) is a parameter-efficient fine-tuning (PEFT) method that reduces computational cost by introducing low-rank matrices to model weight adjustments. Instead of fine-tuning the entire weight matrix $W \in \mathbb{R}^{d\times k}$, LoRA constrains the update as $\Delta W = BA \quad \text{where } B \in \mathbb{R}^{d\times r},\ A \in \mathbb{R}^{r\times k}$. The final updated weight is expressed as $W' = W + \Delta W = W + BA$. The rank r is significantly smaller than $\min(d, k)$, reducing memory consumption and computational demands. The **PEFT** library, used in this assignment for the fine-tuning process, implements **LoRA** by introducing configurations like rank, scaling factor (α), target modules, and dropout. During training, only the low-rank matrices B and A are optimized, while the original weights remain frozen, preserving the base model's knowledge while efficiently adapting it to new tasks.
- ***ArcFace*** adds angular margin to improve discrimination between speakers. It's a widely popular loss function used in face identification tasks. ArcFace enhances intra-class compactness and inter-class separation by adding an angular margin, making embeddings more discriminative. The ArcFace loss is defined as follows:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\log\frac{e^{s\cos(\theta_{y_i}+m)}}{e^{s\cos(\theta_{y_i}+m)} + \sum_{j=1,j\neq y_i}^{n} e^{s\cos\theta_j}}$$

Where:

- θ_{y_i} is the angle between feature vector and weight vector
- m is the additive angular margin (typically 0.5)
- s is the feature scale (typically 30)
- N is the batch size
- n is the number of classes (speakers)

2. Evaluation Metrics for Speaker Verification:

- The *Equal Error Rate (EER)* is the point where the False Acceptance Rate (FAR) equals the False Rejection Rate (FRR). It is commonly used as a performance metric in speaker verification systems. A lower EER indicates better performance, as it reflects a reduced number of incorrect acceptances and rejections. The EER is calculated by finding the threshold where FAR equals FRR.
- *The True Acceptance Rate at 1% False Acceptance Rate (TAR@1%FAR)* measures the percentage of correct acceptances when the FAR is fixed at 1%. This metric evaluates system robustness in scenarios with strict false acceptance constraints. A higher TAR@1%FAR indicates better performance. It is mathematically represented as:

$$\text{TAR@1\%FAR} = \frac{TP}{TP+FN} \text{ at threshold where } \text{FAR} = \frac{FP}{FP+TN} = 0.01$$

- *Speaker Identification Accuracy* measures the percentage of correctly identified speakers in a given dataset. Input audio is matched with a closed set of N identities. Accuracy is calculated using the formula:

$$\text{Accuracy} = \frac{\text{Number of correct identifications}}{\text{Total number of trials}} \times 100\%$$

3. Evaluation Metrics for Speaker Verification:

- *Signal to Interference Ratio (SIR)* measures the ratio of the desired signal to the interfering signal. A higher SIR indicates more effective suppression of interference, representing better separation between speakers.
- *Signal to Artefacts Ratio (SAR)* evaluates the presence of artefacts in the reconstructed speech. It quantifies the level of distortions introduced during the enhancement process. Higher SAR values imply cleaner speech output.
- *Signal to Distortion Ratio (SDR)* measures the overall quality of the enhanced speech signal by comparing the desired speech to the residual distortions. A higher SDR denotes a better reconstruction quality.
- *Perceptual Evaluation of Speech Quality (PESQ)* PESQ is a perceptual metric that assesses speech quality as experienced by human listeners. It compares the enhanced speech to the original clean speech and scores it on a scale from -0.5 to 4.5, with higher values representing better perceptual quality.

→ Task A: Speaker Verification

1. Dataset Used:

- VoxCeleb1, VoxCeleb2: There are two versions of this dataset, VoxCeleb1 and VoxCeleb2. VoxCeleb1 consists of more than 150,000 utterances from 1251 celebrities, and VoxCeleb2 consists of more than 1,000,000 utterances from 6112 celebrities.
- Both of the above datasets were used in different ways throughout the assignment which is detailed below.

2. Methodology:

My submission involves the following files each peforming one step of the task flow.

- `evaluate_models.py` : Handles the evaluation of both pretrained and finetuned models. It calculates metrics like EER, TAR@1%FAR, and Accuracy. It also includes diagnostic functions to compare embeddings and verify weight loading.

> 💡 The diagnosis functions were necessary because both the finetuned and pretrained versions of Wavelm base plus gave the exact same accuracy for the speaker verification task, even after making sure that the LORA A and B matrices were non-zero trained values. The speaker identification accuracy went up however, during the process of finetuning.

- `pretrained_eval.py` : Contains functions to load pretrained models, extract embeddings, and compute similarity scores. It also calculates evaluation metrics for pretrained models.
- `finetune.py` : Implements the fine-tuning process using LoRA and ArcFace loss. It defines the `SpeakerVerificationModel` class and includes functions for training and evaluating the model.
- `test_audio_loading.py` : A test script to verify audio loading from the VoxCeleb2 dataset, ensuring that audio files are correctly processed before fine-tuning.
- `prepare_voxceleb2.py` : Prepares the VoxCeleb2 dataset by creating metadata files and train/test splits. It organizes the dataset for use in training and evaluation.

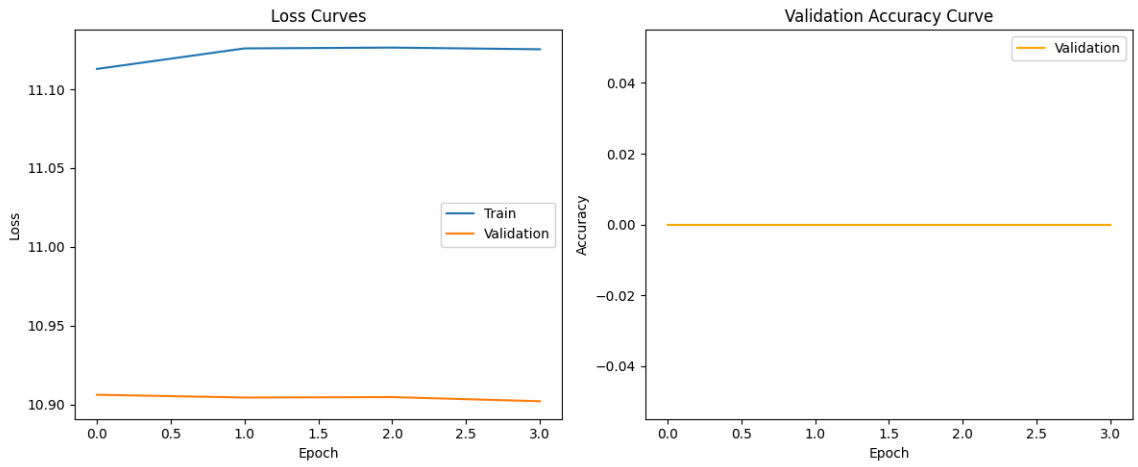
→ The primary task is to perform speaker verification using the pairs in `veri_test2.txt` in voxceleb1. This has been done using `pretrained_eval.py` . The methodology involved computing embeddings from `wavelm base plus` which is the pre-trained model I've used throughout this assignment. The computed embeddings for both the source and the target are then matched and recognised as a match if the similarity is above a threshold. This threshold is the EER threshold which is dynamically calculated. The evaluation of this task uses the VoxCeleb1 dataset, specifically the cleaned trial pairs listed in the `veri_test2.txt file` and the corresponding audio files located in the `data/vox1/wav` directory. For the fine-tuning phase, the larger VoxCeleb2 dataset is employed, with audio files expected in `data/vox2/aac` and associated metadata managed through v `ox2_train.csv` and `vox2_test.csv.`

→ For the second finetuning task, certain liberties were taken in using only a specific number of samples (100) per identity. Approximately 1/3rd of the data was used to finetune the wavelm base plus model owing to hardware memory limitations. The `finetune.py` script handles the fine-tuning process. This involves adapting the pre-trained model using LoRA (Low-Rank Adaptation) and an *ArcFace* loss function. The fine-tuning is specifically conducted on a subset of the VoxCeleb2 dataset, comprising the first 100 unique speaker identities (when sorted ascendingly), as specified in the assignment prompt.

→ After fine-tuning, the `evaluate_models.py` script compares the performance of the original pre-trained model against the newly fine-tuned version. Both models are evaluated on the same VoxCeleb1 trial list of `4000` samples which is a subset of the original trial list. The comparison uses standard speaker verification metrics: Equal Error Rate (EER) expressed as a percentage, True Accept Rate at 1% False Accept Rate (TAR@1%FAR), and overall Speaker Identification Accuracy shown in the following section.

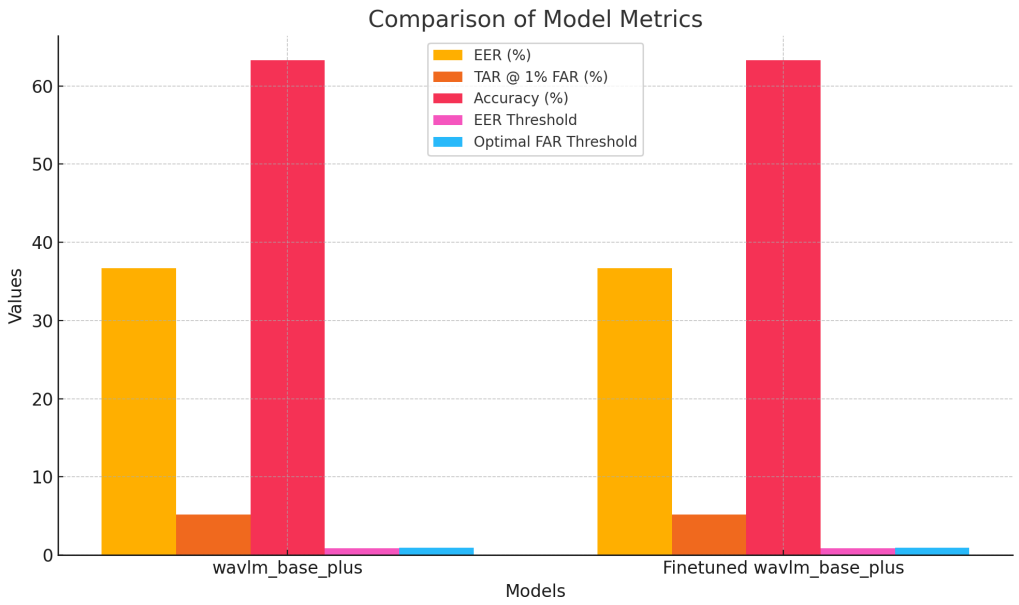
→ During evaluation, the code uses `extract_embeddings_with_lora` (in `finetune.py`) to extract embeddings by passing the input through the backbone of the `SpeakerVerificationModel.` Because the finetuned_model object has had LoRA applied and the adapters are active during evaluation (the code explicitly calls `model.enable_adapters()` or ensures `inference_mode=False during setup`), the forward pass through `model.backbone` *includes the modifications from the trained LoRA adapters.* The embeddings are then taken from the `last_hidden_state` of the backbone's output, averaged over the time dimension. Therefore, the embeddings used for evaluation **reflect the finetuning** performed by the LoRA adapters.

3. Results:



→ In the multiple runs that were performed the loss decreased from about `11.x` to `10.x` with a speaker identification accuracy increase from the base `0.7258` to `0.7346` in the 5 epoch finetuning process with about 1/3rd of the voxceleb2 dataset using

only a specific number of samples (100) per identity owing to memory limitations and compute time.



Model Name	EER (%)	TAR @ 1% FAR (%)	Accuracy (%)	EER Threshold	Num Trials	Trial Subset Path
wavlm_base_plus	36.72	5.20	63.28	0.8688	4000	data/vox1\trial_subset_4000.txt
Finetuned wavlm_base_plus	36.72	5.20	63.28	0.8688	4000	data/vox1\trial_subset_4000.txt

→ Despite the finetuning process, both the finetuned and pretrained versions of the model gave identical results despite being trained for different set of epochs, and multiple runs.

→ Additionally, to guarantee we are comparing the original pretrained model against its genuinely finetuned counterpart, several checks are implemented in the diagnostic script. The script loads separate model instances for pretrained and finetuned evaluations. The differences are then verified by comparing embeddings generated from the same audio input and checking for diverging similarity scores on trial pairs.

> 💡 In my opinion, the problem could be that during fine-tuning only the layers after the base model get fine-tuned, and not the base model itself. The improvement in the performance in the form of loss decrease and speaker identification accuracy increase is coming from the arc-face layer and not necessarily the LORA adapters. This could be because the training data (only 1/3rd) and the number of epochs trained for (5) are not enough to change the output of the base model. There is also a similar sentiment expressed by others who've used the `peft` library for fine-tuning here.

→ Task B: Speech Enhancement

1. Methodology:

My submission involves the following files each peforming one step of the task.

- `src/speech_enhancement/create_mixtures.py` : Generates overlapping multi-speaker audio datasets by mixing utterances from selected speakers.
- **s** `rc/speech_enhancement/evaluation.py` : Calculates objective speech separation quality metrics (SDR, SIR, SAR, PESQ) by comparing estimated and reference sources.
- `src/speech_enhancement/run_separation.py` : Applies a separation model to test mixtures, evaluating separation metrics, and performing speaker identification on the results.
- `src/speech_enhancement/sepformer.py` : Provides a wrapper class to load and use the pretrained SepFormer model for separating single audio files.
- `src/speech_enhancement/sepformer_direct.py` : Applies the SepFormer model in batch mode to separate mixtures listed in a directory or metadata file.

2. Dataset Creation

This part of the assignment focuses on separating overlapping speech from different speakers and then identifying those speakers in the separated audio streams. The process begins by creating specialized multi-speaker datasets derived from the VoxCeleb2. The `create_mixtures.py` script is responsible for this. It takes speaker utterances from VoxCeleb2, specifically using the first 50 unique speakers (sorted ascendingly) to generate a training mixture dataset and the next 50 unique speakers to generate a testing mixture dataset. It overlaps pairs of utterances from different speakers within these sets at various Signal-to-Noise Ratios (SNRs, typically ranging from -5dB to +5dB) to simulate realistic multi-speaker scenarios. The output is a dataset containing mixed audio files along with their corresponding original clean source files and metadata (`metadata.csv`).

3. Speech Separation with SepFormer

The core speech separation task is performed using a pre-trained SepFormer model. I used the wrapper function defined in (`sepformer.py)` , to handle loading the model (specifically speechbrain/sepformer-wham or speechbrain/sepformer-whamr) and applying it to audio files. The `run_separation.py` script orchestrates this process. It takes the *testing mixture dataset* created in the previous step as input and feeds each mixed audio file into the loaded SepFormer model. The model's objective is to output separate audio streams, ideally isolating the speech of each individual speaker present in the mixture. These separated streams are saved as individual WAV files, organized into subdirectories (inside vox2 folder) corresponding to the original mixture ID.

4. Evaluation of Separation Quality

Once the separation is performed, quality is assessed. The `evaluation.py` script handles this. It compares the separated audio streams generated by SepFormer against the original, clean source audio files that were used to create the mixtures. Standard objective metrics are calculated to quantify the separation performance: Signal-to-Distortion Ratio (SDR), Signal-to-Interference Ratio (SIR), Signal-to-Artifacts Ratio (SAR), and potentially Perceptual Evaluation of Speech Quality. These metrics help determine how well the model isolated the target speakers and how much distortion or interference remains.

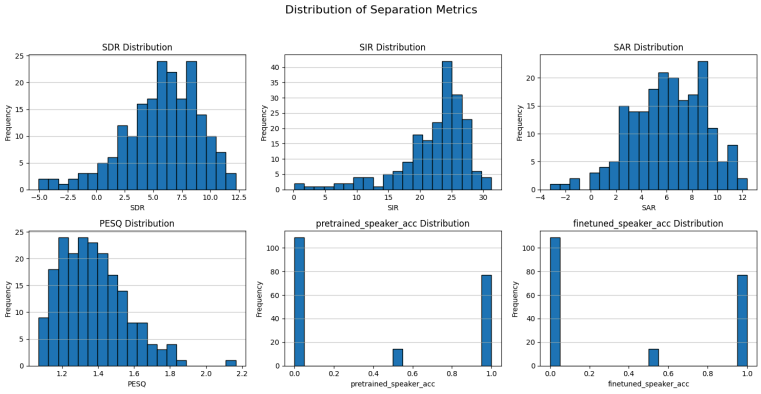
5. Speaker Identification on Separated Sources

The final step involves identifying which speaker corresponds to which separated audio stream. The `run_separation.py` script also manages this task. **It utilizes the speaker verification models developed in the first part of the assignment** (both the original pre-trained model, e.g., wavlm_base_plus, and the fine-tuned version with LoRA). Embeddings are extracted from each separated audio stream and compared (using cosine similarity) against reference embeddings derived from

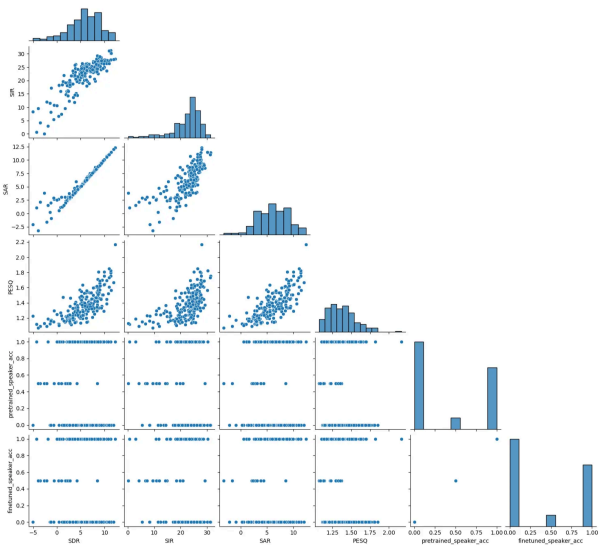
known clean samples of the speakers involved in the test mixtures. The speaker whose reference embedding has the highest similarity to the separated stream's embedding is predicted as the speaker for that stream. The *Rank-1 identification accuracy* (i.e., whether the top prediction matches the ground truth speaker) is calculated and reported for both the pre-trained and the fine-tuned speaker identification models.

6. Results:

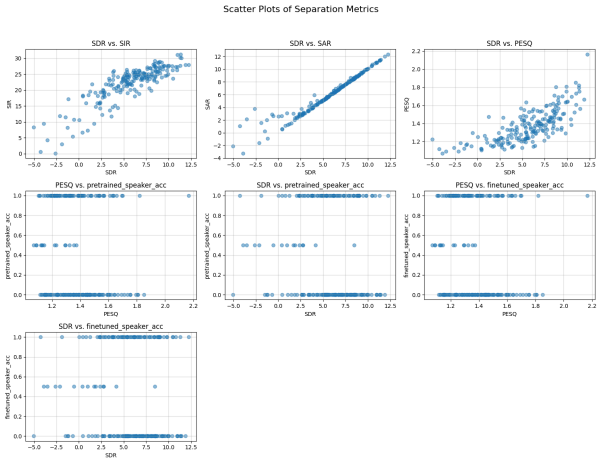
Metric	Value
SDR (Signal-to-Distortion Ratio)	5.7064
SIR (Signal-to-Interference Ratio)	22.1352
SAR (Signal-to-Artifacts Ratio)	6.0921
PESQ (Perceptual Evaluation of Speech Quality)	1.3747
Pretrained Rank-1 Speaker Accuracy	0.4200
Finetuned Rank-2 Speaker Accuracy	0.4200



Distribution of Seperation Metrics.



Pair Plots of Seperation Metrics



→ The Signal-to-Distortion Ratio (SDR) of 5.7064 suggests moderate speech separation quality, indicating some remaining distortion in the output. The Signal-to-Interference Ratio (SIR) of 22.1352 demonstrates strong interference suppression, reflecting effective separation of overlapping speakers. However, the Signal-to-Artefacts Ratio (SAR) of 6.0921 indicates noticeable artifacts in the reconstructed speech, reducing perceptual quality. The Perceptual Evaluation of Speech Quality (PESQ) score of 1.3747, on a scale from -0.5 to 4.5, suggests poor perceptual quality, likely due to artifacts and distortions. Additionally, the Rank-1 speaker identification accuracy for the WaveLM Base Plus backbone remains low at 0.42 for both the pre-trained and fine-tuned models, indicating challenges in correctly identifying speakers from the enhanced audio.

→ Task C: Combined Pipeline

1. Methodology:

My submission involves the following files each peforming one step of the task.

- `evaluate_models.py` : Handles the evaluation of both pretrained and finetuned models. It calculates metrics like EER, TAR@1%FAR, and accuracy. It also includes diagnostic functions to compare embeddings and verify weight loading.
- `pretrained_eval.py` : Contains functions to load pretrained models, extract embeddings, and compute similarity scores. It also calculates evaluation metrics for pretrained models.
- `finetune.py` : Implements the fine-tuning process using LoRA and ArcFace loss. It defines the `SpeakerVerificationModel` class and includes functions for training and evaluating the model.
- `test_audio_loading.py` : A test script to verify audio loading from the VoxCeleb2 dataset, ensuring that audio files are correctly processed before fine-tuning.
- `prepare_voxceleb2.py` : Prepares the VoxCeleb2 dataset by creating metadata files and train/test splits. It organizes the dataset for use in training and evaluation.

2. Novel Pipeline Concept and Model Architecture

The core idea in the pipeline implemented here is to create an enhanced pipeline that uses both a speaker separation model (SepFormer) and a speaker identification model (e.g., WavLM fine-tuned previously) in sequence. The novelty lies in adding a *trainable enhancement network* after the initial separation. The `EnhancedCombinedModel` class defined within `train_enhanced_pipeline.py` encapsulates this. It holds references to the speaker identification model (whose weights are frozen) and a wrapper for the *SepFormer* model. The key trainable component is enhancement_network, a *simple multi-layer 1D convolutional network* designed to take the output streams from *SepFormer* and further refine them, potentially removing artifacts or improving clarity.

3. Training the Enhancement Network

The `train_enhanced_pipeline.py` script orchestrates the training process for this combined pipeline. It **only trains the parameters of the enhancement_network**; both the SepFormer model (used operationally via its wrapper) and the embedded speaker identification model remain frozen. The script uses the *training set* of the multi-speaker mixtures created in the previous assignment part. During the forward pass, each mixture is first separated by *SepFormer*, then speaker embeddings are extracted from the separated sources using the frozen speaker model, and finally, the separated sources are passed through the `trainable enhancement_network`. The training loss used is a combination of L1 and L2, is calculated by comparing the output of this `enhancement_network` directly against the original *clean* source audio files corresponding to the input mixture. This trains the enhancement network to ideally convert the SepFormer output into something closely resembling the clean speech.

4. Evaluation Workflow

The `evaluate_enhanced_pipeline.py` script evaluates the performance of the *trained* `EnhancedCombinedModel` on the *test set* of the multi-speaker mixtures. It loads the trained pipeline (including the trained enhancement network weights and the frozen speaker/SepFormer components). For each test mixture, it performs the forward pass to get the *enhanced* separated audio streams. It then compares these enhanced streams against the ground truth *clean* source audio files to calculate the speech enhancement/separation metrics: SDR, SIR, SAR, and PESQ. A notable challenge here is ensuring the sample rates match for comparison, as SepFormer typically outputs 8kHz while the original sources are 16kHz; the script attempts basic downsampling for this comparison.

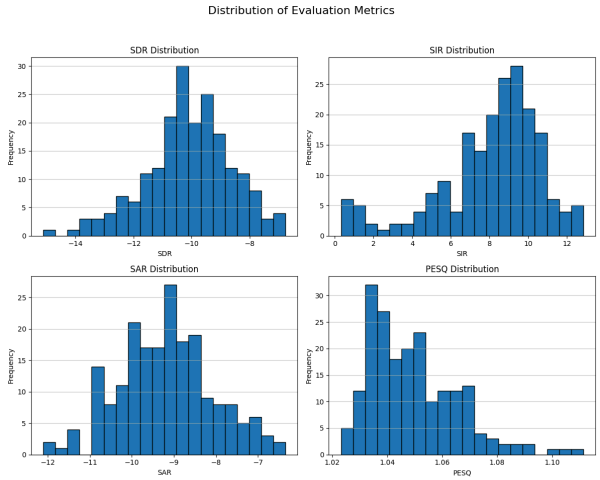
5. Speaker Identification Evaluation

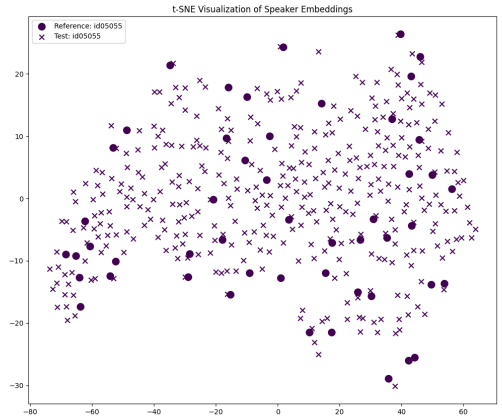
Alongside the enhancement metrics, the e`valuate_enhanced_pipeline.py` script also assesses speaker identification performance on the *enhanced* audio streams. During the forward pass of the evaluation, speaker embeddings are extracted from each enhanced output stream using the speaker identification model *embedded within the trained pipeline*. These embeddings are then compared against reference embeddings (collected from known clean samples of the speakers in the test set) to calculate the *Rank-1 identification accuracy*. This accuracy indicates how often the correct speaker is identified as the top match for each enhanced stream, using the specific speaker model integrated into the pipeline.

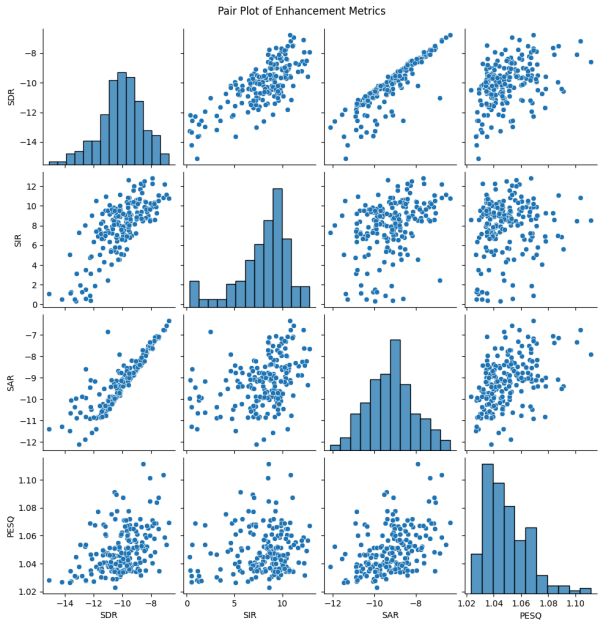
6. Results:

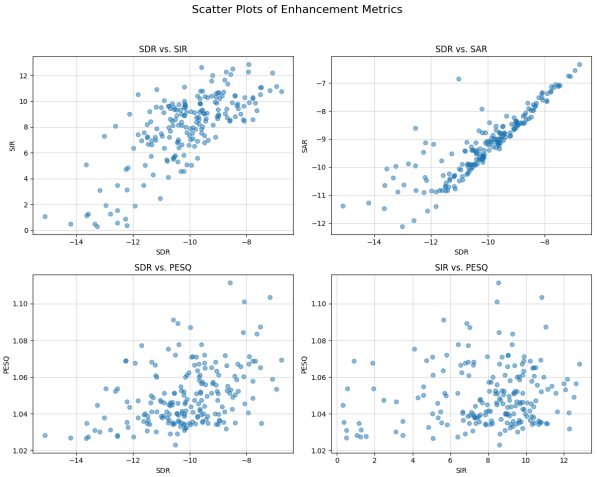
Metric	Value
SDR (Signal-to-Distortion Ratio)	-10.1024
SIR (Signal-to-Interference Ratio)	8.0073
SAR (Signal-to-Artifacts Ratio)	-9.1902
PESQ (Perceptual Evaluation of Speech Quality)	1.0495
Rank1 Accuracy	0.1550

The Signal-to-Distortion Ratio (SDR) of -10.1024 indicates severe distortion in the separated speech, suggesting a poor-quality output. The Signal-to-Interference Ratio (SIR) of 8.0073 shows limited suppression of interference, meaning the separation of overlapping speakers was ineffective. The Signal-to-Artefacts Ratio (SAR) of -9.1902 further highlights significant artifacts introduced during processing, severely degrading the speech's naturalness. The Perceptual Evaluation of Speech Quality (PESQ) score of 1.0495, on a scale from -0.5 to 4.5, suggests extremely poor perceptual quality. Additionally, the Rank-1 identification accuracy of 0.1550 indicates a substantial decline in speaker recognition performance, with the model struggling to identify speakers correctly from the enhanced audio. The low separation quality (SDR/SIR) here caused a poor Rank-1 accuracy, as it directly corrupts the input used to generate the speaker embeddings for the identification task. Improving the speech separation performance should lead to better speaker embeddings and consequently, higher Rank-1 accuracy.









Question 2:

→ Task A: MFCC Feature Extraction and Comparative Analysis of Indian Languages

This report analyzes the Mel-Frequency Cepstral Coefficients (MFCC) features extracted from speech samples of multiple Indian languages. The study focuses on three languages: Hindi, Tamil, and Telugu. The extracted MFCC spectrograms are

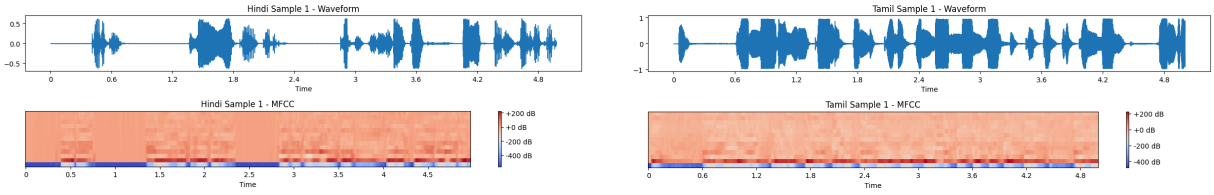
compared, and a classifier is built to predict the language of a given audio sample. The analysis also discusses phonetic and acoustic characteristics that influence the MFCC patterns.

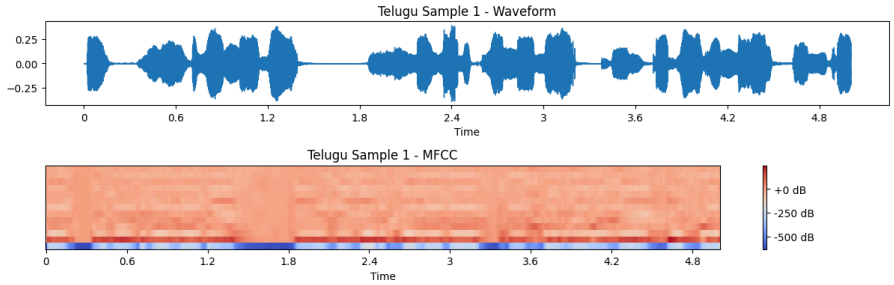
1. Dataset Handling and Loading

```
==================================================
Processing Hindi language data
==================================================
Found 25462 audio files in /root/.cache/kagglehub/datasets/hbchaitanyabharadwaj/audio-dataset-with-10-indian-languages/versions/1/Language Detection Dataset/Hindi
Limiting to 1000 random samples
Processing Hindi: 100% 1000/1000 [01:19<00:00, 22.35it/s]

Summary for Hindi:
- Processed 1000 files successfully
- Average duration: 4.99s (min: 4.98s, max: 5.00s)
- Average MFCC shape: (13, 429)
- Sample rate: 44100 Hz

==================================================
Processing Tamil language data
==================================================
Found 24196 audio files in /root/.cache/kagglehub/datasets/hbchaitanyabharadwaj/audio-dataset-with-10-indian-languages/versions/1/Language Detection Dataset/Tamil
Limiting to 1000 random samples
Processing Tamil: 100% 1000/1000 [00:53<00:00, 22.24it/s]

Summary for Tamil:
- Processed 1000 files successfully
- Average duration: 4.97s (min: 0.38s, max: 5.00s)
- Average MFCC shape: (13, 470)
- Sample rate: 48000 Hz

==================================================
Processing Telugu language data
==================================================
Found 23656 audio files in /root/.cache/kagglehub/datasets/hbchaitanyabharadwaj/audio-dataset-with-10-indian-languages/versions/1/Language Detection Dataset/Telugu
Limiting to 1000 random samples
Processing Telugu: 100% 1000/1000 [00:53<00:00, 21.29it/s]

Summary for Telugu:
- Processed 1000 files successfully
- Average duration: 4.98s (min: 1.89s, max: 5.00s)
- Average MFCC shape: (13, 429)
- Sample rate: 44100 Hz
```

- The script loads the dataset `audio-dataset-with-10-indian-language` from Kaggle using `kagglehub`, which contains audio samples in 10 Indian languages.
- The dataset contains folders for each language: Hindi, Bengali, Punjabi, Kannada, Tamil, Marathi, Telugu, Gujarati, Malayalam, Urdu. The three languages chosen for this assignment are Hindi, Telugu and Tamil.
- Each audio file is read using the library `librosa` sampled at 44.1KHz.
- The number of samples from each language was limited to `1000` owing to memory limitations of Google Colab.

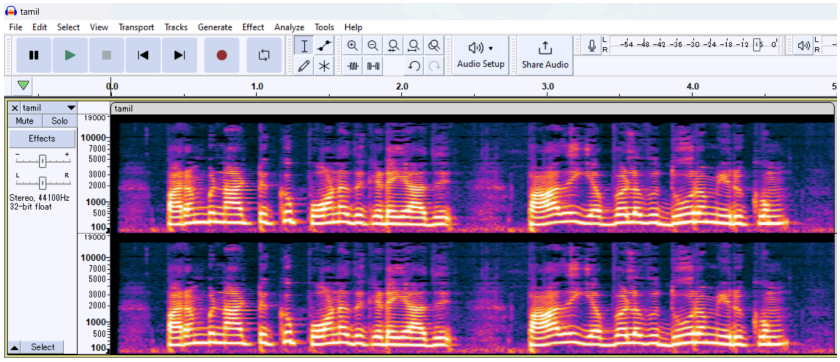
2. MFCC Extraction Process



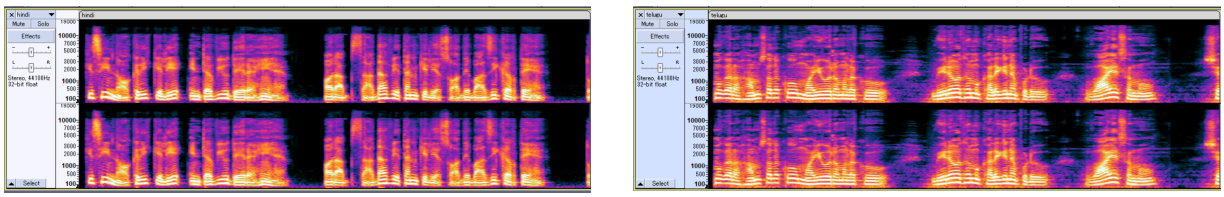


- The script extracts Mel-Frequency Cepstral Coefficients (MFCC) from each audio sample using `librosa.feature.mfcc` API.
- The `n_mfcc` (number of coefficients) is chosen as `13` to show the most relevant captured coefficients.
- The script generates MFCC spectrograms using visualization libraries such as `matplotlib.pyplot`.

3. MFCC Visualization and Spectrogram Comparison

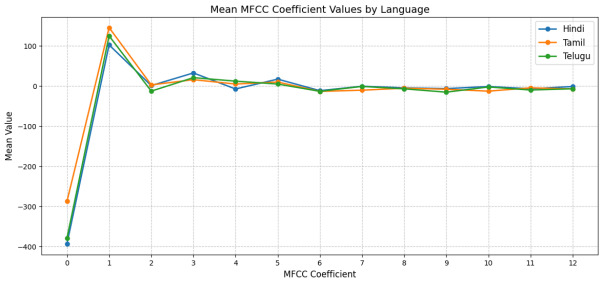


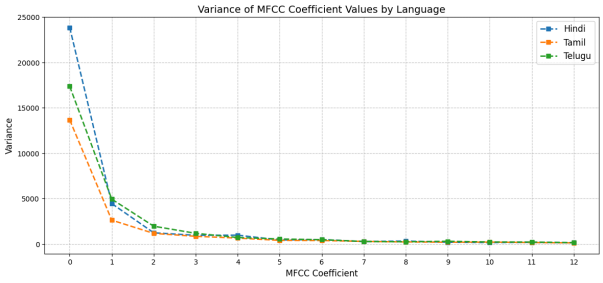
Additional Spectrograms generated using Audacity for better spectral resolution.



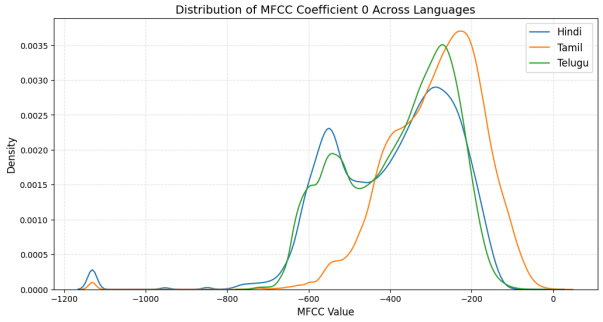
- Hindi speech exhibits rhythmic bursts with distinct pauses, Tamil has a wider amplitude range with continuous speech flow, while Telugu maintains a more even distribution with moderate pauses. Spectrograms highlight stronger energy bands in Hindi and Tamil, whereas Telugu's frequency spread is more uniform, reflecting its phonetic structure.
- Hindi's stress-timed rhythm and aspirated consonants create burst-like energy peaks. Tamil's retroflex sounds and syllable-timed nature result in sustained spectral activity, while Telugu's distinct consonant clusters and visarga sounds produce unique fading patterns in high-frequency bands.
- Features like formant stability, energy distribution along the Mel scale, and modulation rates help differentiate the languages. Hindi shows rapid phonemic shifts, Tamil has smooth transitions, and Telugu balances between the two. A classifier leveraging these features can efficiently predict the spoken language, though dialectal variations and speaker differences pose challenges.

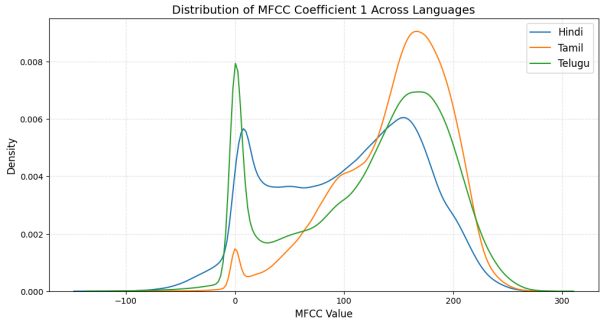
4. MFCC Statistical Comparison

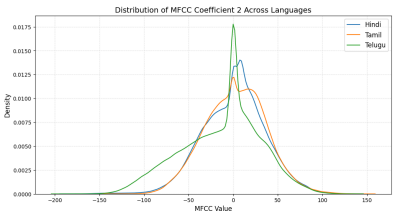


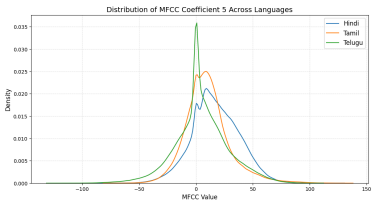


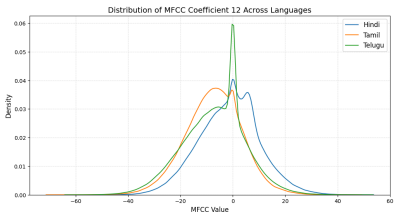
- **MFCC Feature Statistics**: Hindi has the highest variance in MFCC values, particularly in lower-order coefficients, indicating greater spectral variability. Tamil shows a more balanced variance across coefficients, reflecting its consistent phonetic structure. Telugu has the least variance in higher MFCCs, suggesting smoother spectral transitions and less articulatory fluctuation.
- **Pairwise Statistical Significance**: T-tests confirm significant differences in MFCC distributions between all three languages ($p < 0.0001$ in most cases). Hindi vs. Tamil shows strong contrasts in mid-frequency energy (MFCC 3–6), while Hindi vs. Telugu differs sharply in low-frequency components (MFCC 0–2). Tamil and Telugu share more similarities but still exhibit key variations in articulation-sensitive features (MFCC 7–12).









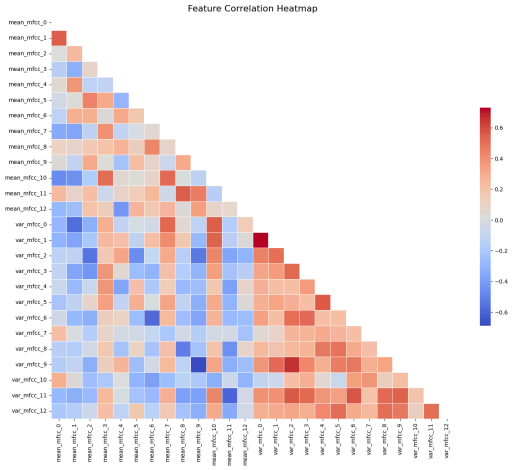


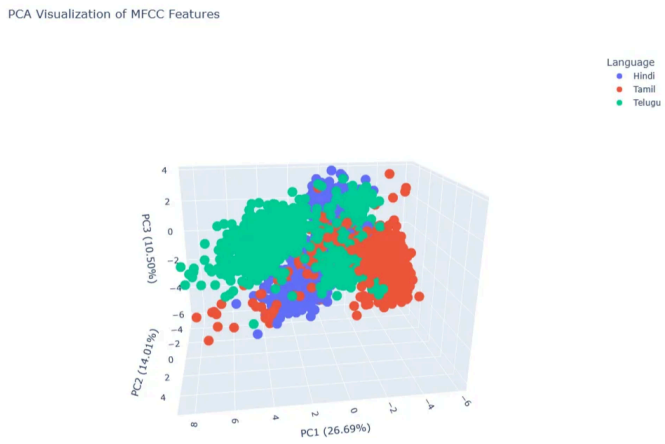
→ Task B: MFCC Feature Extraction and Comparative Analysis of Indian Languages

A classifier is built to predict the language of an audio sample using extracted MFCC features. The dataset undergoes preprocessing steps including normalization of MFCC features to ensure consistency and a train-test split for evaluation. A validation set is also extracted from the training data to fine-tune hyperparameters. The training set is used to train the model, the validation set helps adjust hyperparameters, and the test set provides final performance evaluation.

1. Dataset Preparation & Feature Extraction:

To prepare the dataset for classification, MFCC features are extracted, and statistical measures such as mean and variance are computed for each coefficient. These values are then concatenated into feature vectors, forming the dataset used for training. Labels are encoded, and features are normalized to ensure consistency across samples. A principal component analysis (PCA) visualization provides insights into feature separability.





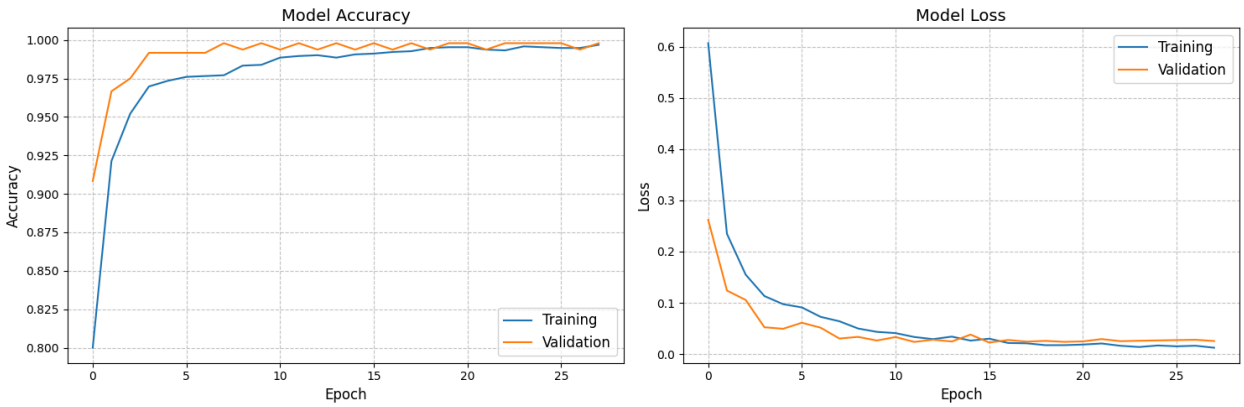
2. Neural Network Architecture:

```
Model Architecture:
LanguageClassifier(
  (model): Sequential(
    (0): Linear(in_features=26, out_features=128, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.3, inplace=False)
    (3): Linear(in_features=128, out_features=64, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.3, inplace=False)
    (6): Linear(in_features=64, out_features=3, bias=True)
  )
)
```

The neural network model used for classification consists of an input layer that takes MFCC features, two hidden layers with ReLU activation and dropout regularization, and an output layer that maps hidden representations to language classes. The model is trained using cross-entropy loss and optimized with Adam. The computed training metrics include loss, which

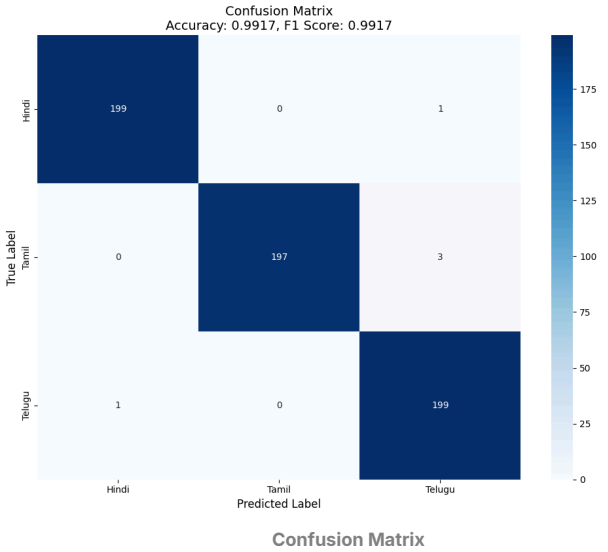
quantifies prediction error, and accuracy, which measures correct classifications relative to total predictions. Validation loss and accuracy monitor model generalization to unseen data

2. Training and Validation:

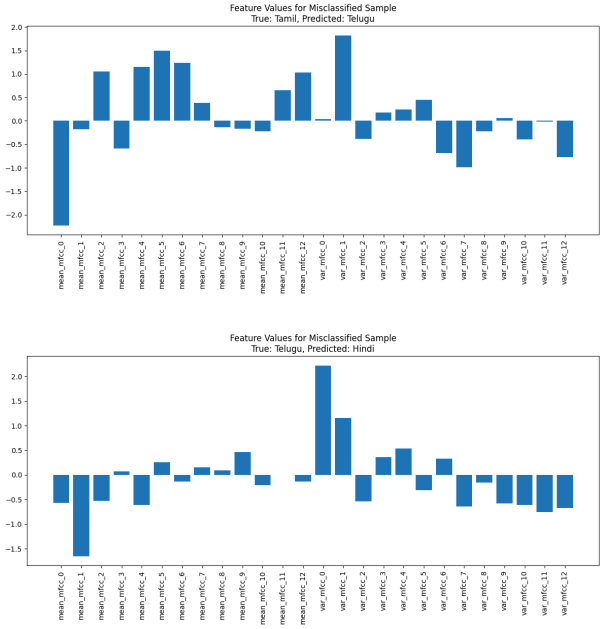


- **Model Training & Testing:**
  - The dataset is split into training (80%) and testing (20%) sets, with the target labels as the vowel class. Features are standardized using `StandardScaler`.
  - The trained models are then tested on the separate test set.

- **Performance Evaluation & Confusion Matrix:**
  - The model was evaluated using accuracy, precision, recall, and F1-score metrics. A confusion matrix was generated to analyze misclassifications, and specific cases of incorrect predictions were examined to identify patterns. Testing performance confirmed high classification accuracy, indicating that MFCC features effectively differentiate between the three languages.



Confusion Matrix



- **Distinctive Features:**
  - The table below summarizes the most distinctive MFCC features for each language. Hindi exhibits strong values in mid-frequency coefficients, while Tamil shows significant variations in both mean and variance of lower coefficients. Telugu has prominent variance in certain MFCCs, indicating a more dynamic spectral pattern. These characteristics highlight the phonetic and acoustic distinctions that contribute to language differentiation.

Language	Feature 1	Value	Feature 2	Value	Feature 3	Value	Feature 4
Hindi	*mean_mfcc_3*	0.7823	*mean_mfcc_12*	0.7451	*mean_mfcc_1*	-0.7309	*mean_mfcc_4*
Tamil	*mean_mfcc_10*	-0.8797	*var_mfcc_1*	-0.8553	*var_mfcc_0*	-0.8345	*mean_mfcc_0*
Telugu	*var_mfcc_2*	0.9324	*mean_mfcc_4*	0.6701	*var_mfcc_9*	0.6455	*mean_mfcc_9*

- **Final Report:**
  - The system works very well at the classification task.

Language	Precision	Recall	F1-Score	Support
Hindi	0.99	0.99	0.99	200
Tamil	1.00	0.98	0.99	200
Telugu	0.98	0.99	0.99	200
Overall Accuracy	**0.99**			**600**

References:

→ Question 1:

- https://www.robots.ox.ac.uk/~vgg/data/voxceleb/
- https://huggingface.co/microsoft/wavlm-base-plus/blob/main/README.md
- https://yiwenlai.medium.com/how-to-train-with-arcface-loss-to-improve-model-classification-accuracy-d4035195aeb9
- https://github.com/huggingface/peft/issues/793
- https://medium.com/@mujahidabdullahi1992/an-introduction-to-lora-unpacking-the-theory-and-practical-implementation-e665c5d78295
- https://arxiv.org/abs/2106.09685?source=post_page-----e665c5d78295----------

→ Question 2:

- https://librosa.org/doc/latest/index.html
- https://www.kaggle.com/datasets/hbchaitanyabharadwaj/audio-dataset-with-10-indian-languages