**RESEARCH TEASER PROJECT – IIIT HYDERABAD – SPEECH ANALYTICS**

**TEAM-44**

Sanath Macha(machan76758@gmail.com)

Gnana Rohith Yanduru(rohith.yanduru@gmail.com)

Chilupuri Nalin Prabhath(nalinprabhath.chilupuri@gmail.com)

URL: https://drive.google.com/drive/folders/1MRVCiM\_ypypMGwbA9jKjYTtKuOsUnF21?usp=sharing

**Problem Statement**

Speaker Recognition

Developing a scalable and efficient AI model for speaker recognition that can accurately identify speakers from the given dataset.

**Dataset details**

[Speaker Recognition Dataset (kaggle.com)](https://www.kaggle.com/datasets/kongaevans/speaker-recognition-dataset)

This dataset contains speeches of five prominent leaders namely; Benjamin Netanyahu, Jens Stoltenberg, Julia Gillard, Margaret  
Tacher and Nelson Mandela which also represents the folder names. Each audio in the folder is a one-second 16000 sample rate PCM encoded.

Originally, the speech for each speaker was a one lengthy audio, I chunked them into one-second each for easier workability. If you combine the chunked audios from 0.wav to 1500.wav, it forms a complete speech of the respective speaker.

A folder called **background noise** contains audios that are not speeches but can be found inside and around the speaker environment e.g audience laughing or clapping. It can be mixed with the speech while training.

**Method or experimental setup (details about model architecture, training configuration and any relevant information)**

**Preprocessing**

This preprocessing algorithm uses Mel-frequency cepstral coefficients (MFCC) features to prepare audio data for speaker recognition. This is how the code is explained theoretically:

Libraries Needed: TensorFlow, librosa for audio processing, and scikit-learn for data preprocessing are imported at the start of the code.

Defining Speakers and Directories: This includes defining the speaker name list and the directory path where audio files are kept.

Feature Deletion Function: The directory location and the list of speaker names are input parameters for the ext() function. Within the operation:

a. To store extracted features and matching speaker labels, it initialises empty lists of features and labels.

b. It repeatedly goes through every audio file and speaker folder in the directory.

c. The load() function of librosa is used to load the audio data for each audio file (filename). It sets duration=1 to extract the first second of audio and sr=None to maintain the original sampling rate.

d. The feature.mfcc() function in librosa is used to extract MFCC features from the audio. The MFCCs are calculated with this function using the audio waveform. It yields a matrix with a different MFCC coefficient for each row and a separate audio frame for each column.

e. To guarantee that every feature has a zero mean and unit variance throughout the dataset, the extracted MFCCs are standardised using scikit-learn's StandardScaler().

f. The associated speaker label (encoded as an integer) is appended to the labels list, and the standardised MFCCs are appended to the features list.

Return Features and Labels: As NumPy arrays, the method returns the extracted features (X) and matching labels (y).

Feature extraction involves using the ext() method with the directory path and speaker names as parameters. Then, variables are used to store the extracted features (X) and labels (y).

Printing the Initial Feature The function shows the preprocessing results for a single audio file by printing the first set of extracted features (X[:1]).

Print Statement: To signify that the code has finished running, a print statement is provided at the end.

By extracting and standardising MFCC features, this preprocessing method transforms raw audio data into a format that can be used to train a speaker identification model. The machine learning model then uses the retrieved features as input, and the associated labels symbolise the speakers' identities.

**Splitting Data into Training, Validation and test splits**

**Label Encoding:**

Label Encoder from scikit-learn is used to convert the categorical speaker labels (y) into numerical labels.

fit\_transform() method is called on the label encoder object (l) to fit the encoder to the labels (y) and transform them into encoded labels.

The original speaker names (speakers) are assigned to the classes\_ attribute of the label encoder to maintain correspondence between numerical labels and speaker names.

Train-Test Split:

train\_test\_split function from scikit-learn is used to split the data into training, validation, and test sets.

X\_train and y\_train contain the features and labels for the training set, respectively.

X\_temp and y\_te contain the features and labels for the combined validation and test set.

test\_size=0.3 specifies that 30% of the data will be used for testing/validation.

random\_state=42 ensures reproducibility by fixing the random seed.

Further Splitting for Validation and Test Sets:

Another call to train\_test\_split is made to split X\_temp and y\_te into separate validation and test sets.

test\_size=0.5 indicates that the validation and test sets will each contain 50% of the data.

random\_state=42 is used for consistency in random splitting.

**TRAINING Configuration**

We trained the Speaker Recognition model using Logistic Regression, KNN, MLP and observed not so accurate results. Therefore we tried implementing Deep Learning Models.

We trained a speaker recognition model using an LSTM (Long Short-Term Memory) neural network architecture implemented with Keras, a high-level neural networks API running on top of TensorFlow.

Model Definition:

A sequential model is defined using tf.keras.Sequential(). This allows you to build the neural network layer by layer.

The model consists of:

An LSTM layer with 128 units as the input layer. The input\_shape is specified as the shape of the input data (X\_train.shape[1] represents the number of time steps, and X\_train.shape[2] represents the number of features).

A densely connected layer with 64 units and ReLU activation function.

A final output layer with the number of units equal to the number of speakers (len(speakers)) and a softmax activation function. Softmax activation is used for multi-class classification, where each output neuron represents the probability of a speaker.

Model Compilation:

The model is compiled using the compile() method.

The optimizer is set to 'adam', a popular optimization algorithm known for its efficiency and effectiveness.

The loss function is set to 'sparse\_categorical\_crossentropy' since the labels are integers.

'accuracy' is chosen as the metric to monitor during training.

Callbacks:

Two callbacks are defined:

EarlyStopping: Monitors the validation loss ('val\_loss') and stops training if the loss doesn't decrease for a specified number of epochs (patience=3). It restores the model weights from the epoch with the best validation loss (restore\_best\_weights=True).

ModelCheckpoint (not used in this code snippet): Saves the model weights during training if the validation loss improves.

Model Training:

The fit() method is called to train the model.

Training data (X\_train, y\_train) and validation data (X\_val, y\_val) are provided.

The number of epochs is set to 20, and the batch size is set to 32.

The callbacks argument is used to pass the early stopping callback (ea).

Early Stopping Check:

After training, the code checks if early stopping was triggered by examining the stopped\_epoch attribute of the early stopping callback (ea).

If early stopping was triggered (ea.stopped\_epoch > 0), it prints the epoch at which it occurred.

Otherwise, it prints that training completed without early stopping.

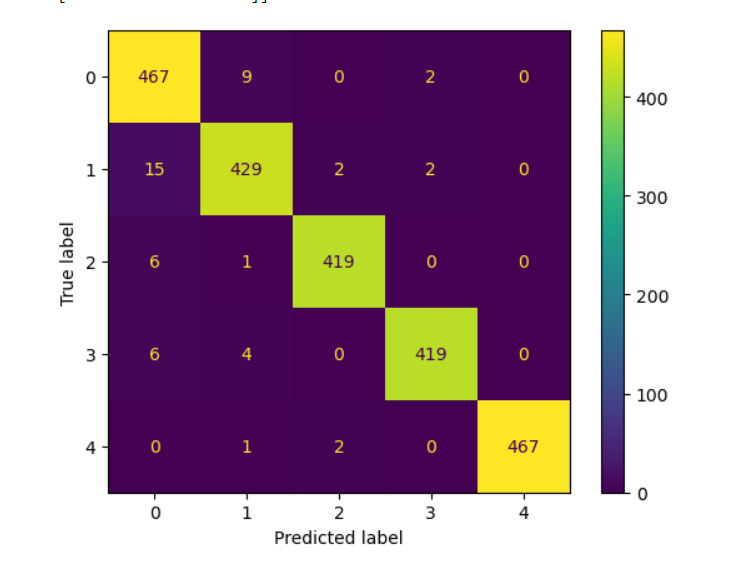
Plotting Training History:

The code plots the training loss and validation loss over epochs using matplotlib.

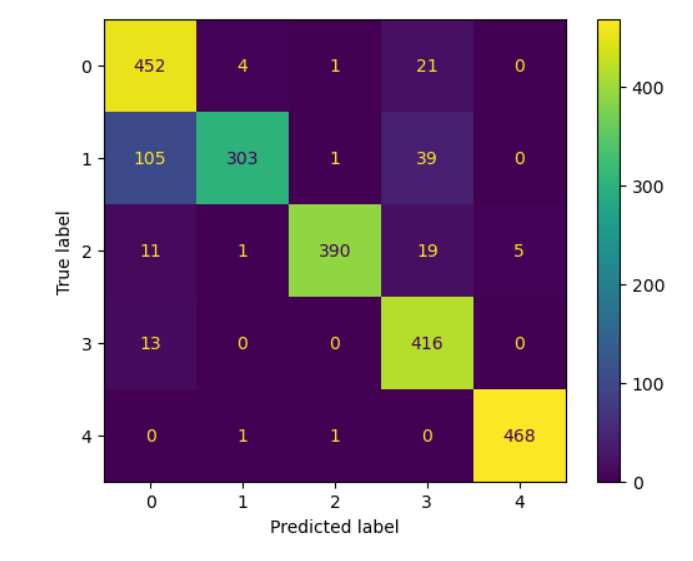
The training loss (mo.history['loss']) and validation loss (mo.history['val\_loss']) are obtained from the training history returned by the fit() method.

The plot helps visualize the training progress and the model's generalization performance.

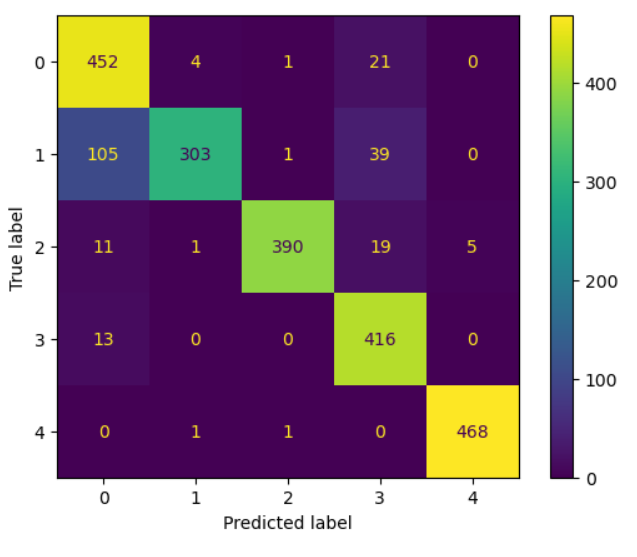
Overall, this algorithm trains a speaker recognition model using an LSTM neural network architecture, with early stopping to prevent overfitting and monitoring of training progress through loss metrics.



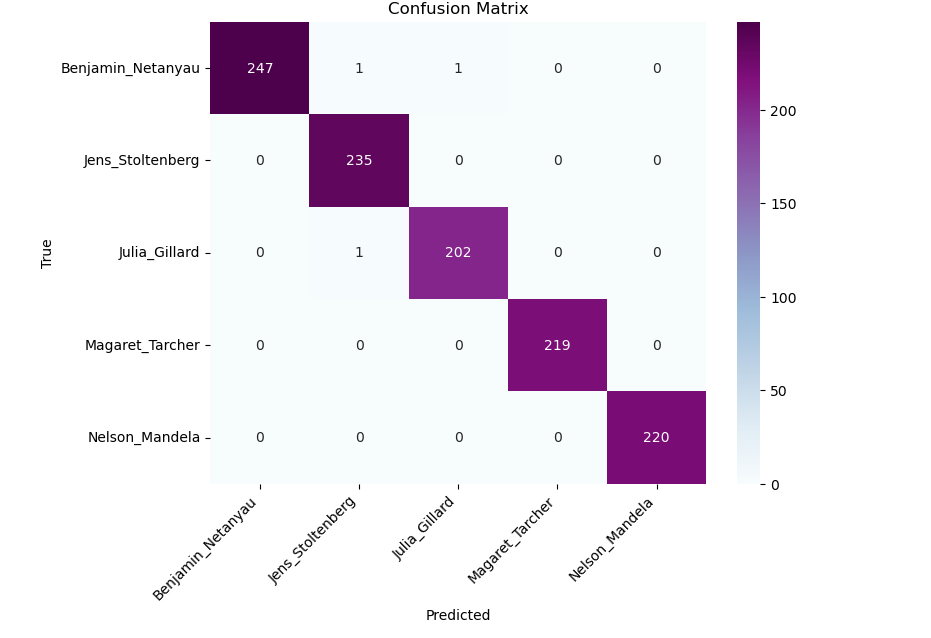
Logistic



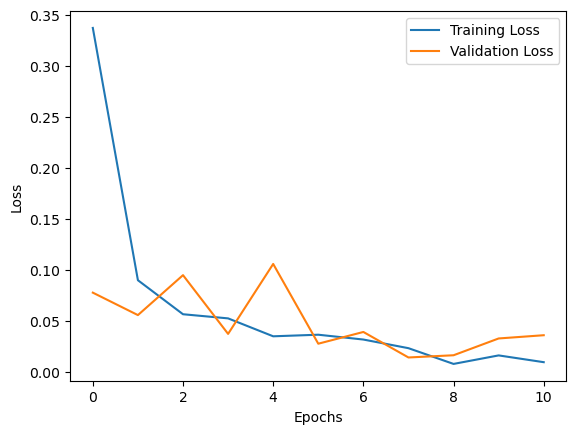
KNN



MLP



LSTM



**RESULT:**

+----------------------------------------+------------+------------+

| Model | Accuracy | F1-Score |

+========================================+============+============+

| Logistic regression | 0.977788 | 0.97804 |

+----------------------------------------+------------+------------+

| KNN CLASSIFIER | 0.901377 | 0.899911 |

+----------------------------------------+------------+------------+

| MLP CLASSIFIER | 0.984896 | 0.985054 |

+----------------------------------------+------------+------------+

| LSTM-Softmax Speaker Recognition Model | 0.99556 | 0.995561 |

+----------------------------------------+------------+------------+

**LSTM-Softmax Speaker Recognition Model** performed the best with an accuracy of 0.99556 and F1-score of 0.995561.

**Observations and Conclusion :**

* **LSTM Model Outperforms Others:** The LSTM model achieved significantly higher accuracy and F1-score compared to Logistic Regression, KNN, and MLP classifiers.
* **MLP is Competitive:** The MLP classifier performed well, with accuracy and F1-score close to the LSTM model.
* **Logistic Regression and KNN Fall Short:** While still reasonable, Logistic Regression and KNN had lower accuracy and F1-score compared to the more complex models.

**Possible Reasons for Performance:**

* **LSTM Captures Temporal Dependencies:** LSTMs are well-suited for capturing temporal dependencies in sequences, which are inherent in speech data. This makes them effective for speaker recognition based on voice characteristics that evolve over time.
* **MLP Learns Complex Relationships:** MLPs can learn complex non-linear relationships in data, potentially capturing subtle speaker-specific features.
* **Simpler Models Might Overfit:** Logistic Regression and KNN are less complex and might be prone to overfitting with limited data. They might not capture the full range of speaker variations as well as the LSTM or MLP.

**Hypotheses for Further Exploration:**

* **Data Augmentation:** Experimenting with data augmentation techniques (e.g., adding noise, speed variations) could potentially improve the performance of all models by enriching the training data.
* **Hyperparameter Tuning:** Optimizing hyperparameters specifically for each model architecture might further enhance their performance.
* **Feature Engineering:** Exploring additional features beyond MFCCs (e.g., pitch, formants) could provide more discriminative information for speaker recognition.

**In summary, the LSTM-Softmax model stands out for its impressive speaker recognition accuracy. However, depending on your specific needs and constraints, other models like MLP or simpler options like Logistic Regression might be viable alternatives.**