**Explanation of Python Code**

* Initially mounted my Google Drive to the Colab environment, allowing to access files stored in my Google Drive account. Then audio\_data variable as the path to audio dataset on Google Drive.
* Importing necessary libraries for audio processing, machine learning and evaluation.

**Extract\_feature function:**

* Extract\_feature function with mfcc, chroma and mel Boolean flags indicating whether to extract any one of them respectively, using soundfile library we can access our audio file.
* Reading audio data from the file and obtaining sample rate of the audio. Then finding Short-Time Fourier Transform (STFT) of the audio signal if ‘chroma’ flag is set to True.
* If mfcc flag is set to True then computing Mel-Frequency Cepstral Coefficient(MFCC) from the audio signal using librosa.feature.mfcc and storing it in result array.
* If chroma flag is set to True then computing STFT using librosa.feature.chroma\_stft and storing the result in array.
* If mel flag is set to True then computing Mel-scaled spectrogram from the audio signal using librosa.feature.melspectrogram and storing the result in array.
* Finally return the result.
* Creating a dictionary called emotions with codes from 01 to 08 as RAVDESS dataset contains emotions like neutral, calm, happy, sad, angry, fearful, disgust and surprised each assigned a values from 01 to 08 and we want observe only happy and sad so create created a list called observed\_emotions and store happy and sad emotions.

**Load\_data function:**

* This function takes argument test\_size=0.2 which represents proportion of dataset to include in the test split. Then glob.glob function returns a list of pathnames that matches specified pattern which is path to all ‘.wav’ files inside ‘Actor\_\*’ directories.
* os.path.basename() contains ‘.wav’ file excluding the directory path and splitting file\_name using dash(‘-‘) as separator to get emotion code and matching the label corresponding to emotions dictionary.
* Then appending audio feature to list ‘x’ and corresponding emotion label to the list ‘y’( features and labels).
* Finally return the splitted data into test and train according to the test\_size.
* Unpacking the values returned by the load\_data() function into x\_train, x\_test, y\_train and y\_test making data ready for machine learning models.
* Print the number of sample values in training and test sets and actual number of features in the training data.
* Using MLP(Multi-Layer Perceptron) Classification from scikit-learn and parameters like alpha represents regularization term on weights preventing overfitting, batch\_size indicates size of minibatches for stochastic optimizers, epsilon for numerical stability, hidden\_layer\_size indicates the number of neurons in each hidden layer, choosing learning\_rate as ‘adaptive’ and maximum\_iteration = 500
* Fitting the model with corresponding x\_train and y\_train data to predict new datasets and calling predict method of trained MLP model passing the x\_test feature and result is stored in y\_pred variable.
* Calculating the accuracy by comparing the number of correct predictions to the total number of predictions and printing the accuracy in percentage.
* Then calculating F1 score for each class in the test set’s true labels(‘y\_test’) and predicted labels(‘y\_pred’) by metrics module from skicit-learn and Print the Actual\_Emotion and Predicted\_Emotion in table format using pandas dataframe.
* Visualizing the result using matlplotlib.pyplot and mapping predicted emotion labels to numerical values based on ‘1’ for happy and ‘-1’ for sad, creating line chart for values and plots emotion\_values on the y-axis representing emotions over time with colour green and line width is set to 2. Labelling x-axis, y-axis as Time and Emotion and set the title of the plot as ‘Predicted Emotion over Time’ and finally displaying the plot.
* User can upload their own files by importing files module from google.colab import files.upload() allows to upload files interactively in Google Colab Environment and path of audio file is stored in uploaded variable and extracting first key from uploaded and assign it to path variable.
* Assigning number of speakers as 2(depends on audio), Language as English and model\_size as medium. Checking for condition like if language is equal to ‘English’ and model\_size not equal to ‘large’, then appending ‘.en’ to model\_name.
* Installing openai whisper and pyannote-audio from github and importing torch library for deep learning tasks, pyannote.audio for speaker diarization and embedding tasks, ‘wav’ provides functionalities for working with ‘wav’ audio files and AgglomerativeClustering a clustering algorithm from scikit-learn.
* Checking the audio is in the format of ‘.wav’ if not, then converting into ‘.wav’ format by calling subprocess function and using ‘ffmpeg’ tool to convert the audio file to ‘.wav’ format. Then whisper.load\_model() function loads a pretrained model to be loaded with argument model\_size. Storing the result in model variable for further predictions and evaluations.
* The transcribe method of the loaded model transcribe the audio file specified by the path variable and storing the result in result variable which includes information such as transcribed text, timestamps and other metadata. Accessing transcribing segments from the result (dictionary) and storing it in segments variable.
* Opening the specified audio file path in read mode using wave.open() method and retrieving total number of frames and frame rate in the audio file using getnframes(), getframerate() method. We are calculating the duration of the audio file in seconds by dividing the total number of frames/frame rate.

**Segment\_embedding function():**

* Creating instance of the Audio class provided by pyannote.audio library.
* Extracting start and end timestamps of the input segment and Segment object represents segment using start and end timestamps.
* Using crop() method to extract the audio data for the specified segment and returns the audio waveform.
* Return the extracted audio waveform using the loaded ‘embedding\_model’.
* Initializing numpy array called embeddings with a shape of length of segment and dimensionality of 192. Looping through segments list and assigning the computed embedding to the corresponding row in the embeddings array. If embeddings array contains any NAN values it can be replaced with zero and infinity using numpy’s nan\_to\_num() method.
* Performing agglomerative clustering on the embeddings calculated for each segment and assigns speaker labels to the segments based on the clustering results. After fitting the model, looping through segment and assigning a speaker label in the format ‘SPEAKER’ + labels[i].
* Generating a transcript from the segmented audio data and writing it to file named “transcript.txt”. Looping through segments list, for each segment it checks for the first segment or current segment which is different from previous segment. Finally writing speaker label and start time of the segment into the file named “transcript.txt”.
* Open the “transcript.txt” in read mode and print the contents of the file into display window and importing pandas dataframe to display the result in table format.
* Creating a list of dictionary called speaker\_dialogue and each dictionary in the list represents a segment of dialogue spoken by a speaker and “Speaker”, ”Time”, “Text” represents the speaker label, timestamp and text and converting into pandas dataframe.
* Creating a line chart using Matplotlib to visualize speaker diarization data. Then converting the extracted timestamps into seconds for smooth plotting. Plot method plots ‘seconds’ on the axis and speaker indices on y-axis. The line is marked with markers(‘o’) and line color is in blue. Setting labels for x-axis and y-axis namely Time(seconds) and Predicted Speaker Diarization and finally displaying the gridlines on the plot.