## horizontal line



Speech Separation

Murali Mohan

Varun Bhatia

Reenal Boddul

Mohammed Irshad

Priyansh

# Abstract

Our business case is that an input audio file that contains multiple speakers has to be separated as an individual audio file for each speaker that is non target based audio sepration. For this task, we used [**Speaker Diarization**](https://arxiv.org/pdf/1710.10468.pdf)with **Spectral Cluster.** The speaker diarization first removes the noises in the audio file that is passed into the **LSTM network**. The output of the LSTM network is represented as the vector embeddings which are passed into the spectral cluster to find the number of speakers and group each speaker. Then the output is given in the form of tuples that contain labels, start time, and stop time for each speaker. The final task is to slice the audio and concatenate according to each speaker’s timeline.

# Introduction

Speaker Diarization is the process of partitioning an input audio into homogeneous segments according to the speaker’s identity in a multi-speaker environment. It has many applications such as understanding medical conversation, multimedia information retrieval, speaker turn analysis, audio processing, video captioning and many more. There are two significant changes applied in this speaker diarization compared to other speaker diarization models. The first one is neural network-based audio embeddings (d-vectors) which are extracted using DNN (Deep Neural Network). The second one is the use of spectral clustering to identify the number of speakers.

# Speaker Diarization System

We have referred to the research paper called ‘“Speaker Diarization with LSTM”. According to the paper, a speaker diarization system consists of the following process.

1. Speech segmentation
2. Embedding extraction
3. Speech detection
4. Clustering

The flowchart of the system is shown below.

Speech Segmentation

In this system, audio signals are transformed into frames of the width of 25ms and a step of 10ms. The size of the frames is called window and the step is known as overlap. For example, as the window starts at 0 and ends at 25ms and the next window starts at 10ms and ends at 35ms with a step size of 10ms and so on until the full length of audio is covered.

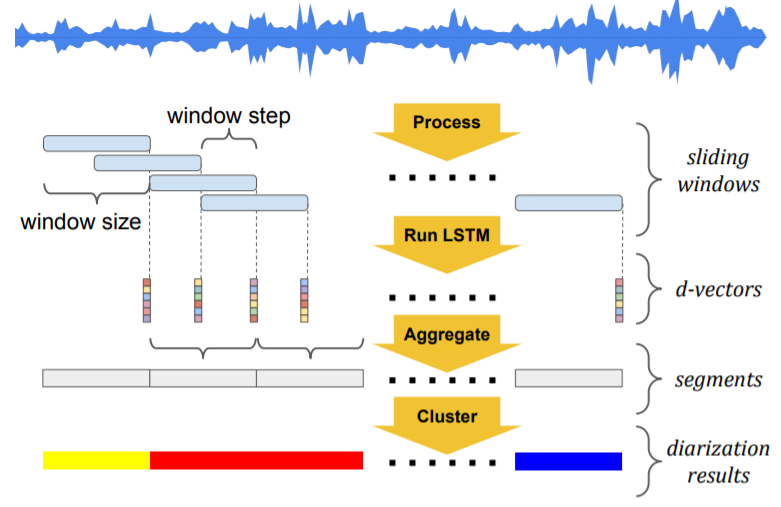


Fig.1 A flowchart of d-vector based speaker diarization system

Embedding Extraction

The vector embeddings are extracted from the audio segments in this process. In order to do that, MFCC (Mel Frequency Cepstral Coefficient) of these audio segments has to be calculated. These are basically feature-coefficients that capture variations in the speech like pitch, quality, intone, etc of the voice. They are obtained by doing a specialized Fourier Transform of the speech signal. Then the LSTM network is run on each window. The last frame output of the LSTM is used as the d-vector representation.

Speech Detection

In this system, a Voice Activity Detector (VAD) is used to determine speech segments from the audio, which are further divided into smaller non-overlapping segments using maximal segment-length (400ms used in the paper). This determines the temporal resolution of the diarization results. For each segment, the corresponding d-vectors are first L2 normalized, then averaged to form an embedding of the segment.

Clustering

For detecting the number of speakers, the Spectral cluster is used. After creating the embeddings of the segments, the clustering algorithm groups these embeddings. Then the embedding of the segments belonging to the same speakers is part of one cluster and assigned the label of the speaker. The final output is given in the form of tuples containing the label, start time and stop time for each speaker.

# Code Implementation

In our implementation, we have used the open-source repository called [Resemblyzer](https://github.com/resemble-ai/Resemblyzer). Using this repository, we have done three of our major tasks namely speech detection, speech segmentation and embedding extraction. The first step is to clone the Resemblyzer and make it our working directory.

**!git clone https://github.com/resemble-ai/Resemblyzer.git**

**!pip install webrtcvad-wheels**

**cd Resemblyzer**

The next step is that we need to upload an audio file to diarize. The 'wav' file format is only accepted. The following steps explain the code below.

1. Provide the pathname of the audio file.
2. The file path is then passed to the **preprocess\_wav** function which used VAD to trim out the silences in the audio file and also normalizes the decibel level of audio.
3. Then we create an instance of the **VoiceEncoder** class name encoder and pass 'cpu' as the default device or ‘cuda’ if GPU is available. The **embed\_utterance** takes in the processed wav file, segments it out into windows, makes MFCCs of these segments and finally creates d-vectors of these audio segments.
4. The **cont\_embeds** is a N x D matrix, where N is the number of segments created (which is equal to the number of d-vectors) and D is the dimension of each d-vector, which by default is 256.
5. **wav\_splits** is a list with the start and end time of each window for a d-vector has been created

**from resemblyzer import preprocess\_wav, VoiceEncoder**

**from pathlib import Path**

**#give the file path to your audio file**

**audio\_file\_path = '/content/drive/My Drive/Audio/commercial\_mono.wav'**

**wav\_fpath = Path(audio\_file\_path)**

**wav = preprocess\_wav(wav\_fpath)**

**encoder = VoiceEncoder("cpu")**

**\_, cont\_embeds, wav\_splits = encoder.embed\_utterance(wav,return\_partials=True, rate=16)**

The output of the preprocess\_wav() function is a NumPy array which has to be converted back into an audio file for slicing and concatenating as per the timelines of each speaker. The below code converts the array into an audio file and saves it

**from scipy.io.wavfile import write**

**#Output denoised wave after removing the pauses**

**write('/content/drive/My Drive/Audio/commercial\_mono1.wav', 16000, wav)**

We get the audio embeddings (d-vectors) from VoiceEncoder class. The output is saved in the variable called **cont\_embeds** which is passed into the spectral cluster to find the labels or number of speakers. Since this is a non-target based audio separation technique, hence we do not have any information on the number of speakers beforehand.

The spectral clustering algorithm we have used could not be able to find the optimum speakers. So we used the measurement called WCSS (Within Cluster Sum of Squares) which measures the average distance of all the points within a cluster to the cluster centroid.

We used a library called [kneed](https://github.com/arvkevi/kneed/blob/master/notebooks/decreasing_function_walkthrough.ipynb) that gives the optimum number of clusters by locating the elbow in the graph below

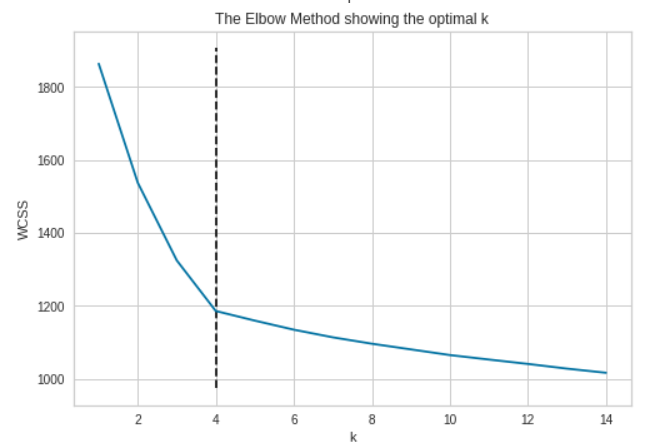


Fig.2 Elbow method to find optimum clusters

Let’s now see the code implementation of the elbow locator. To locate our elbow, WCSS measurement has to be found for a various number of clusters. Here we found the WCSS from one to fifteen clusters and saved in a list called **wcss**. The list of WCSS and the range of clusters is passed into a class called **KneeLocator** and instantiated as **kn**. To get the optimum number of clusters, we use the class variable **kn.knee.**

**#Finding optimum clusters c**

**from kneed import KneeLocator**

**from sklearn.cluster import KMeans**

**wcss = []**

**K = range(1, 15)**

**for k in K:**

**km = KMeans(n\_clusters=k)**

**km = km.fit(cont\_embeds)**

**wcss.append(km.inertia\_)**

**kn = KneeLocator(list(K), wcss, S=1.0, curve='convex', direction='decreasing')**

**c=kn.knee**

The optimum number of clusters are then passed into the spectral cluster class to find the labels.

**#Passing the clusters to get the labels**

**clusterer = SpectralClusterer(**

**min\_clusters=c,**

**max\_clusters=100,**

**p\_percentile=0.90,**

**gaussian\_blur\_sigma=1)**

**labels = clusterer.predict(cont\_embeds)**

The final part of the speaker diarization is to get the timelines of each speaker. So we have the function called **create\_labelling()** which takes **labels** from **clusterer** class and **wav\_splits** from **VoiceEncoder()** class. The final output is called **labeling** which gives the label, start time and stop time for each speaker.

**#Function to create the timelines of speakers**

**def create\_labeling(labels,wav\_splits):**

**sampling\_rate=16000**

**times = [((s.start + s.stop) / 2) / sampling\_rate for s in wav\_splits]**

**labeling = []**

**start\_time = 0**

**for i,time in enumerate(times):**

**if i>0 and labels[i]!=labels[i-1]:**

**temp = [str(labels[i-1]),start\_time,time]**

**labelling.append(tuple(temp))**

**start\_time = time**

**if i==len(times)-1:**

**temp = [str(labels[i]),start\_time,time]**

**labeling.append(tuple(temp))**

**return labeling**

**labeling = create\_labelling(labels,wav\_splits)**

The output of **labeling** is shown below.

**[('0', 0, 0.92),**

**('1', 0.92, 5.3),**

**('0', 5.3, 8.54),**

**('1', 8.54, 10.04),**

**('0', 10.04, 15.14),**

**('1', 15.14, 17.24),**

**('0', 17.24, 20.96),**

**('1', 20.96, 21.38)]**

We have the timelines for each speaker, but the timelines are not grouped, according to the speakers. So the code below groups the timelines as per the labels.

**# Grouping the timelines, according to the speakers.**

**output = {}**

**for x, y, z in labelling:**

**if x in output:**

**output[x].append((y, z))**

**else:**

**output[x] = [(y, z)]**

**values = []**

**items = output.items()**

**for item in items:**

**values.append(item[1])**

We have used a library named [pydub](https://github.com/jiaaro/pydub). Using this library, the denoised audio file can be sliced and concatenated.

First, the audio file is imported using **AudioSegment.from\_wav(FILE\_PATH).**

**from pydub import AudioSegment**

**audio = AudioSegment.from\_wav(FILE\_PATH)**

Then **labeling**, **audio** and **file\_path** for saving the audio files are passed into the function **output\_audio()** and the output is received as a list. Finally, we take the output list and save the audio files for each speaker.

**def output\_audio(timelines, audio, file\_path):**

**# Grouping the timelines**

**output = {}**

**for x, y, z in timelines:**

**if x in output:**

**output[x].append((y, z))**

**else:**

**output[x] = [(y, z)]**

**# Removing the labels and making the list of timelines**

**values = []**

**items = output.items()**

**for item in items:**

**values.append(item[1])**

**# Slicing the audio and joining them for each speaker**

**voices = []**

**n = 0**

**for i in values:**

**for j in i:**

**start\_time, stop\_time = j**

**n += audio[start\_time:stop\_time]**

**voices.append(n)**

**# Saving the final audio files**

**for i, j in enumerate(voices):**

**j.export(file\_path + f'/voice\_{i}.wav')**

# Conclusion

The compilation of codes in this document is available at this [link](https://colab.research.google.com/drive/1LAqKE1n63I_hlwcr90BWgEZDNyr1Pu5R?usp=sharing).

# References

<https://github.com/resemble-ai/Resemblyzer>

<https://arxiv.org/pdf/1910.06379.pdf>

<https://www.kaggle.com/kevinarvai/knee-elbow-point-detection>