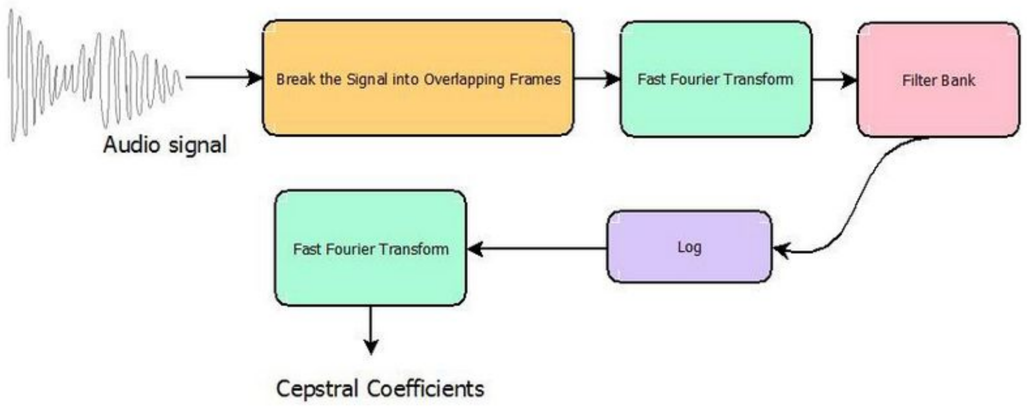
# Documentation On Learning Journey And Key-concepts

## ONE SHOT LEARNING:

Classification under the restriction that we may observe a single example of each possible class before making a prediction about a test instance is called *one-shot learning*.

## MFCC:

The 1st step in any automatic speech recognition system is to extract features i.e. identify the components of the audio signal that are good for identifying the linguistic content and discarding all the other stuff which carries information like background noise, emotion etc. Mel Frequency Cepstral Coefficients (MFCCs) are a feature widely used in automatic speech recognition.
Any sound generated by humans is determined by the shape of their vocal tract. If this shape can be determined correctly, any sound produced can be accurately represented. The envelope of the time power spectrum of the speech signal is representative of the vocal tract and MFCC accurately represents this envelope.



Here, Filter bank refers to the mel filters (converting to mel scale) and Cepstral Coefficients are nothing but MFCCs.

References:
1. http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/
2. https://medium.com/prathena/the-dummys-guide-to-mfcc-aceab2450fd
3. https://towardsdatascience.com/understanding-audio-data-fourier-transform-fft-spectrogram-and-speech-recognition-a4072d228520

## Libraries in python:

1. **python_speech_features:**
   This library provides common peech features for ASR including MFCCs and filterbank energies.

   Supported features:

   - python_speech_features.mfcc() - Mel Frequency Cepstral Coefficients
   - python_speech_features.fbank() - Filterbank Energies
   - python_speech_features.logfbank() - Log Filterbank Energies
   - python_speech_features.ssc() - Spectral Subband Centroids

   https://python-speech-features.readthedocs.io/en/latest/

2. **LibROSA**

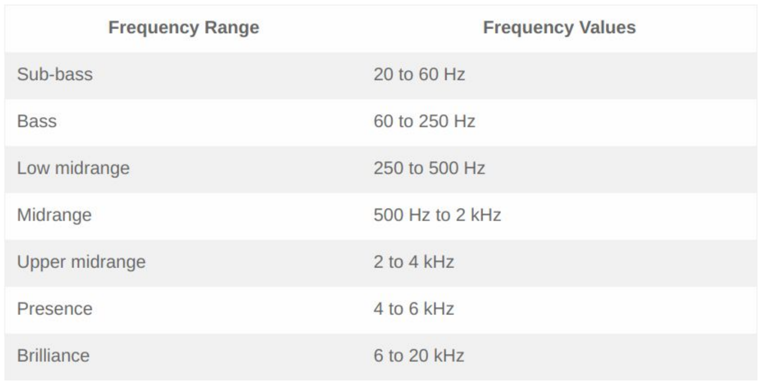
   It is a python package for music and audio analysis. It provides the building blocks necessary to create music information retrieval systems.

   https://librosa.org/librosa/v0.4.0/index.html

## AUDIO SPECTRUM:

The audio spectrum is the audible frequency range at which humans can hear and spans from 20 Hz to 20,000 Hz.

The audio spectrum range spans from 20 Hz to 20,000 Hz and can be effectively broken down into seven different frequency bands, with each band having a different impact on the total sound.



| Frequency Range | Frequency Values |
|---|---|
| Sub-bass | 20 to 60 Hz |
| Bass | 60 to 250 Hz |
| Low midrange | 250 to 500 Hz |
| Midrange | 500 Hz to 2 kHz |
| Upper midrange | 2 to 4 kHz |
| Presence | 4 to 6 kHz |
| Brilliance | 6 to 20 kHz |

A **sound spectrum** is a representation of a sound – usually a short sample of a sound – in terms of the amount of vibration at each individual frequency. It is usually presented as a graph of either power or pressure as a function of frequency. The power or pressure is usually measured in decibels and the frequency is measured in vibrations per second (or hertz, abbreviation Hz) or thousands of vibrations per second (kilohertz, abbreviation kHz).

**References:**

1. https://www.teachmeaudio.com/mixing/techniques/audio-spectrum
2. https://newt.phys.unsw.edu.au/jw/sound.spectrum.html

## COMMON SPEAKER RECOGNITION DATASETS:

1. **Common Voice**
   This dataset contains hundreds of thousands of voice samples for voice recognition. It includes over 500 hours of speech recordings alongside speaker demographics. To build the corpus, the content came from user submitted blog posts, old movies, books, and other public speech.

2. **Google Audioset**
   It is an expanding ontology of 635 audio event classes and a collection of 2,084,320 human-labeled 10-second sound clips drawn from YouTube videos.

3. **VoxCeleb**
   It is a large-scale speaker identification dataset. It contains around 100,000 phrases by 1,251 celebrities, extracted from YouTube videos, spanning a diverse range of accents, professions and age.

4. **LibriSpeech ASR Corpus**

It consists of approximately 1,000 hours of 16kHz read English speech. The data is derived from read audiobooks from the LibriVox project (a volunteer effort responsible for the creation of approximately 8,000 public domain audio books, the majority of which are in English).

5. **2000 HUB5 English Evaluation Transcripts**
   It was developed by the Linguistic Data Consortium (LDC) and consists of transcripts of 40 English telephone conversations.

6. **CALLHOME American English Speech**
   It was developed by the Linguistic Data Consortium (LDC) and consists of 120 unscripted 30-minute telephone conversations between native speakers of English.

7. **The CHiME-5 Dataset**
   This dataset deals with the problem of conversational speech recognition in everyday home environments. Speech material was elicited using a dinner party scenario.

8. **TED_LIUM Corpus**
   The TED-LIUM corpus was made from audio talks and their transcriptions available on the TED website. It consists of 2,351 audio talks, 452 hours of audio and 2,351 aligned automatic transcripts in STM format.

9. **Free Spoken Digit Dataset**
   A simple audio/speech dataset consisting of recordings of spoken digits. The recordings are trimmed so that they are silent at the beginnings and ends.

Reference:
https://lionbridge.ai/datasets/best-speech-recognition-datasets-for-machine-learning/
https://www.cmswire.com/digital-asset-management/9-voice-datasets-you-should-know-about/
http://www.openslr.org/12 (LibriSpeech ASR Corpus)

## You-Only-Speak-Once:

**https://github.com/sampathkethineedi/You-Only-Speak-Once**

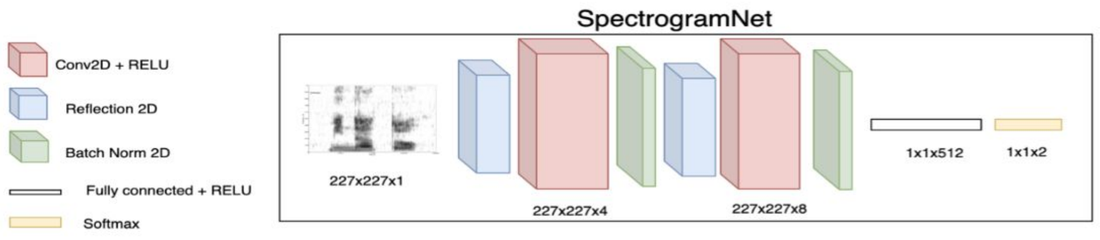
You-Only-Speak-Once aims to develop a one shot learning based voice authentication system using siamese network. A neural network embedding system that maps utterances to a hyperspace where speaker similarity is measured by cosine similarity.

**Dataset used**: LibriSpeech Corpus

Two different network designs were tested for voice authentication system: *SpectrogramNet* and *FBankNet.*

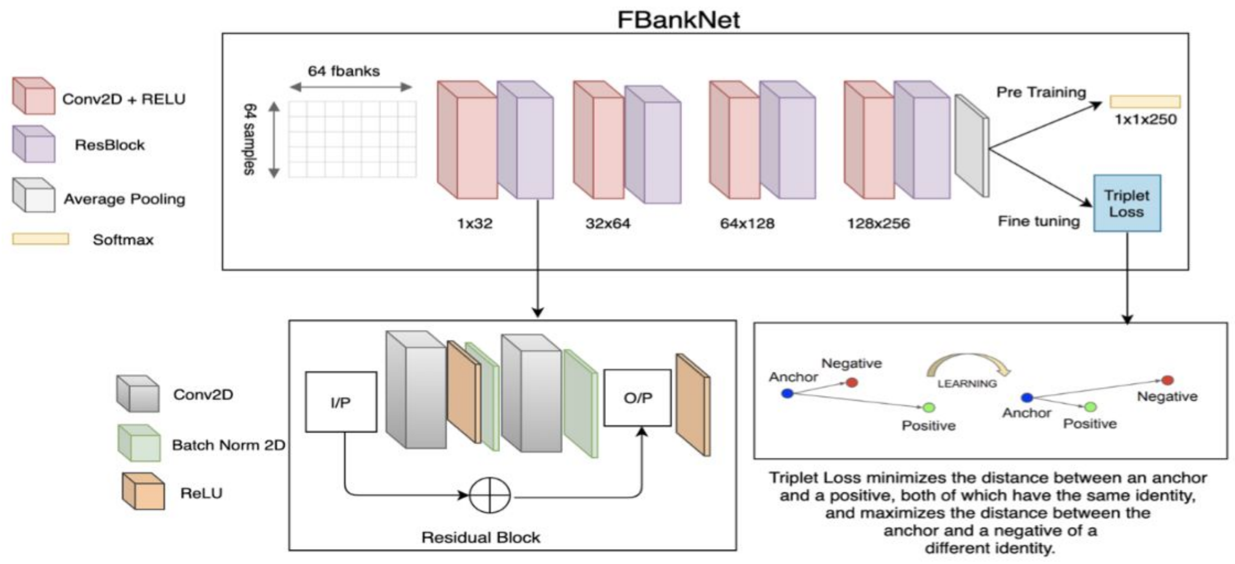
## SpectrogramNet:

The audio samples are split into frames of 5 seconds using a stride of 4 seconds. For each of these frames, we extracted spectrograms of dimensions 227 x 227 x 1, which served as inputs into our Neural Network. Next we split the dataset into a train-set comprising of 200 speakers and a test-set with 50 speakers, with each speaker being represented by ~250 spectrograms.



## FBankNet:

Each audio file was split into frames of 25ms with a stride of 10ms. The first 64 FBank coefficients were calculated from each sample and grouped 64 frames together to generate a training sample of size 64 x 64 x 1 as inputs to Conv2D. Total number of samples obtained was more than half a million which was split into 95% and 5% for train and test respectively.



**The FBank model creation can be seen in the file bae_model.py**

## **test_api.py** file

➔ PyAudio provides python bindings for PortAudio, the cross-platform audio I/O library. With PyAudio, Python can be used to play and record audio.
➔ To record or play audio, open a stream on the desired device with audio parameters using ***pyaudio.Pyaudio.open().***
➔ **Pickle** is used for serializing and de-serializing Python object structures, also called marshalling or flattening. Serialization refers to the process of converting an object in memory to a byte stream that can be stored on disk or sent over a network. Later on, this character stream can then be retrieved and de-serialized back to a Python object.
➔ **Librosa**'s load function will read in the path to an audio file, and return a tuple with two items. The first item is an 'audio time series'(type: array) corresponding to audio track. The second item in the tuple is the sampling rate that was used to process the audio.
➔ The **wave module** in Python's standard library is an easy interface to the audio WAV format. The functions in this module can write audio data in raw format to a file-like object and read the attributes of a WAV file.

**Functions in test_api.py file**

- **load_spk()**

Embeddings of the speakers are loaded. Unpickling of files are done with the help of pickle library (the process of loading a pickled file back into a python program)

- **save_spk()**
  This is used to write in a file. It first undergoes pickling and then saved in a file.
- **delete()**
  To delete/remove any speaker
- **register()**
  To add new speaker:
  Preprocessing of audio file is done.
  - extract_fbank(): Firstly, using librosa the audio file's time series is obtained and normalised. For this the gte_fbanks() function is used. Then slicing is done to get sample sets of 64 frames each.

  Obtaining Embeddings of the preprocessed fbanks.
  - get_embeddings(): FBankCrossEntropyNet function is used to obtain the embeddings.

  Mean of the embeddings are calculated and saved in the data directory.
- **recognise()**
  The audio file's time series is obtained and preprocessing is done. Corresponding embeddings are obtained. The cosine distance between the embeddings of the audio file to be recognised and the stored embeddings are calculated. Mean distance of the embeddings is calculated and stored in a list named positive_mean_list. The maximum among this list is calculated and if it is greater than the threshold, then the audio is matched with one of the speakers.
- **listen()**
  Listening to the audio and dividing it into frames. The audio data is written in raw format to a file using the wave module.

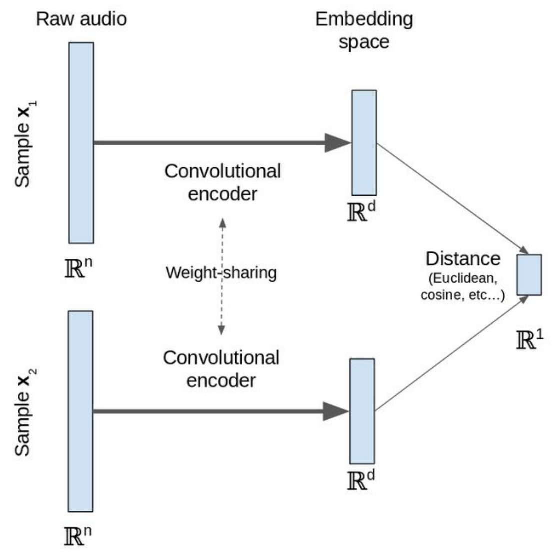
## BUILDING A SPEAKER IDENTIFICATION SYSTEM FROM SCRATCH WITH DEEP LEARNING

### Siamese Network

The idea of running two identical convolutional neural networks on two different inputs and then comparing them is called Siamese network architecture.

Goal of learning:
Parameters of NN define an encoding $f(x^{(i)})$
Learn parameters so that,
If $x^{(i)}$, $x^{(j)}$ are the same person $||f(x^{(i)}) - f(x^{(j)})||^2$ is small.
If $x^{(i)}$, $x^{(j)}$ are the same person $||f(x^{(i)}) - f(x^{(j)})||^2$ is large.

Unlike most common neural network architectures these networks take two separate samples as inputs instead of just one. Each of these two samples is mapped from a high-dimensional input space into a low-dimensional space by an encoder network. The "siamese" nomenclature comes from the fact that the two encoder networks are "twins" as they share the same weights and learn the same function. These two networks are then joined at the top by a layer that calculates a measure of distance (e.g. euclidean distance) between the two samples in the embedding space. The network is trained to make this distance small for similar samples and large for dissimilar samples.



Schematic of a siamese network. The samples are mapped from a high-dimensional space to a low dimensional space i.e. n >> d and then a distance measure is calculated between them.

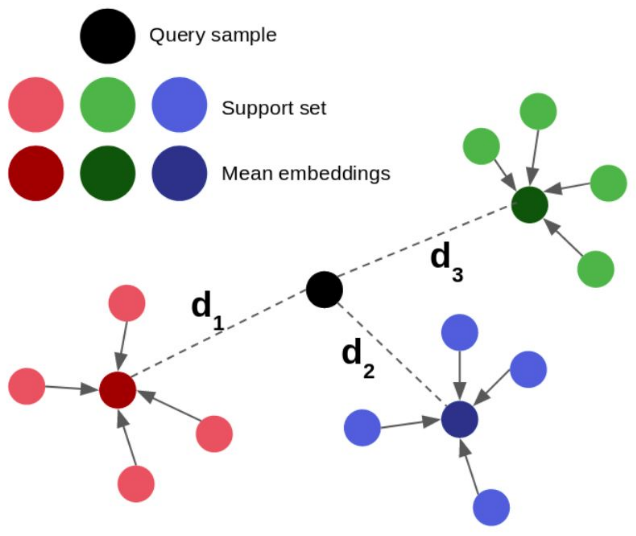
**n-shot Learning**

The ability to quickly generalise to previously unseen classes is often framed as the ability to perform n-shot learning i.e. the ability to recognize a previously unseen class after having only seen *n* (where n is small) examples. A models performance at few-shot learning is measured with n-shot, k-way classification tasks which are run as follows:

- A model is given a query sample belonging to a new, unseen class

- It is also given a support set consisting of n examples each from k different unseen classes
- The model then has to identify which sample(s) in the support belong to the class of the query sample

**Siamese networks for n-shot learning**
Convert the query samples and all of the support set samples into their representation in the embedding space and then perform a nearest-neighbour classification. first calculate the mean position of the embeddings belonging to each class, then take the mean embedding that is closest to the query sample embedding to be the best class.



n-shot classification with n=4. The prediction of the model is the class with the minimum distance (**d**_1, **d**_2, **d**_3) from its mean embedding to the query sample.

## RESEMBLYZER

Resemblyzer helps in deriving a **high-level representation of a voice** through a deep learning model called the voice encoder. Given an audio file of speech, it creates a summary vector of 256 values (an embedding, often shortened to "embed") that summarizes the characteristics of the voice spoken. Resemblyzer has many uses:

- **Voice similarity metric**: compare different voices and get a value on how similar they sound. This leads to other applications:
    - **Speaker verification**: create a voice profile for a person from a few seconds of speech (5s - 30s) and compare it to that of new audio. Reject similarity scores below a threshold.

  - **Speaker diarization**: figure out who is talking when by comparing voice profiles with the continuous embedding of a speech segment.
  - **Fake speech detection**: verify if some speech is legitimate or fake by comparing the similarity of possible fake speech to real speech.
- **High-level feature extraction**: you can use the embeddings generated as feature vectors for your machine learning models! This also leads to other applications:
  - **Voice cloning**
  - **Component analysis**: figure out accents, tones, prosody, gender, … through component analysis of the embeddings.
  - **Virtual voices**: create entirely new voice embeddings by sampling from a prior distribution.
- **Loss function**: you can backpropagate through the voice encoder model and use it as a perceptual loss for your deep learning model! The voice encoder is written in PyTorch.

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

## **audio.py** file:

Functions defined:

- **preprocess_wav(fpath_or_wav: Union[str, Path, np.ndarray], source_sr: Optional[int]=None)**

  Applies preprocessing operations to a waveform either on disk or in memory such that The waveform will be resampled to match the data hyperparameters.

  Parameters:

  *fpath_or_wav*: either a either a filepath to an audio file (many extensions are supported, not just .wav), either the waveform as a numpy array of floats.
  *source_sr*: if passing an audio waveform, the sampling rate of the waveform before preprocessing. After preprocessing, the waveform'speaker sampling rate will match the data hyperparameters. If passing a filepath, the sampling rate will be automatically detected and this argument will be ignored.

- **wav_to_mel_spectogram(wav)**

Derives a mel spectrogram ready to be used by the encoder from a preprocessed audio waveform. This is not a log-mel spectrogram.

- **trim_long_silences(wav):**

  Ensures that segments without voice in the waveform remain no longer than a threshold determined by the VAD parameters in params.py.

  Parameter:

  wav: the raw waveform as a numpy array of floats

  return: the same waveform with silences trimmed away (length <= original wav length)

- **normalize_volume(wav, target_dBFS, increase_only=False, decrease_only=False)**

  Used by the pre_process function to normalize the volume of the waves

## **hparams.py** file:

All the parameters are defined here and initialised with certain values, like the Mel-filterbank, Audio sampling rate, No. of spectrogram frames in partial utterance, voice activation detection window size, audio volume normalization, model parameters.

## **voice_encoder.py** file:

Class VoiceEncoder is created:

- **_init_(self, device: Union[str, torch.device]=None, verbose=True**)

  param- device: either a torch device or the name of a torch device (e.g. "cpu", "cuda"). If None, defaults to cuda if it is available on your machine, otherwise the model will run on cpu. Outputs are always returned on the cpu, as numpy arrays.

  The network is defined here (lstm,linear,relu), the target device is obtained and the pretrained model speaker weights is loaded.

- **forward(self, mels: torch.FloatTensor)**

Computes the embeddings of a batch of utterance spectrograms.

:param mels: a batch of mel spectrograms of same duration as a float32 tensor of shape (batch_size, n_frames, n_channels)

:return: the embeddings as a float 32 tensor of shape (batch_size, embedding_size).

Embeddings are positive and L2-normed, thus they lay in the range [0, 1].

- **compute_partial_slices(n_samples: int, rate, min_coverage):**

  Computes where to split an utterance waveform and its corresponding mel spectrogram to obtain partial utterances of <partials_n_frames> each. Both the waveform and the mel spectrogram slices are returned, so as to make each partial utterance waveform correspond to its spectrogram. The returned ranges may be indexing further than the length of the waveform. It is recommended that you pad the waveform with zeros up to wav_slices[-1].stop.
  Parameters:

  *n_samples*: the number of samples in the waveform

  :param rate: how many partial utterances should occur per second. Partial utterances must cover the span of the entire utterance, thus the rate should not be lower than the inverse of the duration of a partial utterance. By default, partial utterances are 1.6s long and the minimum rate is thus 0.625.

  *min_coverage*: when reaching the last partial utterance, it may or may not have enough frames. If at least <min_pad_coverage> of <partials_n_frames> are present, then the last partial utterance will be considered by zero-padding the audio. Otherwise, it will be discarded. If there aren't enough frames for one partial utterance, this parameter is ignored so that the function always returns at least one slice.

  return: the waveform slices and mel spectrogram slices as lists of array slices. Index respectively the waveform and the mel spectrogram with these slices to obtain the partial utterances.

- **embed_utterance(self, wav: np.ndarray, return_partials=False, rate=1.3, min_coverage=0.75)**

Computes an embedding for a single utterance. The utterance is divided in partial utterances and an embedding is computed for each. The complete utterance embedding is the L2-normed average embedding of the partial utterances.

*Parameters*

*wav:* a preprocessed utterance waveform as a numpy array of float32

*return_partials:* if True, the partial embeddings will also be returned along with the wav slices corresponding to each partial utterance.

*rate:* how many partial utterances should occur per second. Partial utterances must cover the span of the entire utterance, thus the rate should not be lower than the inverse of the duration of a partial utterance. By default, partial utterances are 1.6s long and the minimum rate is thus 0.625.

*min_coverage:* when reaching the last partial utterance, it may or may not have enough frames. If at least <min_pad_coverage> of <partials_n_frames> are present, then the last partial utterance will be considered by zero-padding the audio. Otherwise, it will be discarded. If there aren't enough frames for one partial utterance, this parameter is ignored so that the function always returns at least one slice.

:return: the embedding as a numpy array of float32 of shape (model_embedding_size,). If <return_partials> is True, the partial utterances as a numpy array of float32 of shape (n_partials, model_embedding_size) and the wav partials as a list of slices will also be returned.

- **embed_speaker(self, wavs: List[np.ndarray], **kwargs):**

  Compute the embedding of a collection of wavs (presumably from the same speaker) by averaging their embedding and L2-normalizing it.

  Parameters

  wavs: list of wavs a numpy arrays of float32.

  kwargs: extra arguments to embed_utterance()

:return: the embedding as a numpy array of float32 of
shape(model_embedding_size,).

## **demo01_similarity.py** file:

DEMO 01: shows how to compare speech segments (=utterances) between them to get a metric on how similar their voices sound. Utterances from the same speaker to have a high similarity, and those from distinct speakers to have a lower one.

## **demo02_diarization.py** file:

how this similarity measure can be used to perform speaker diarization (telling who is speaking when in a recording).

## **demo03_projection.py** file:

DEMO 03: shows one way to visualize these utterance embeddings. Since they are 256-dimensional, it is much simpler to get an overview of their manifold if it's dimensionality is reduced first. By doing so, we can observe clusters that form for utterances of identical characteristics. What we'll see is that clusters form for distinct speakers, and they are very tight and even linearly separable.

## **demo04_clustering.py** file:

DEMO 04: building from the previous demonstration, we'll show how natural properties of the voice can emerge through analysis of the embeddings. The dimensionality reduction algorithm UMAP will create clusters from embeddings with similar features. When provided with samples from many distinct speakers, it tends to create two clusters for each sex. This is what we'll how here, by using the speaker metadata file provided in the LibriSpeech dataset to retrieve the sex of each speaker. The distinction was learned entirely in an unsupervised manner.

## **demo05_fake_speech_detection.py** file:

DEMO 05: shows how to achieve a modest form of fake speech detection with Resemblyzer. First the audio is loaded and preprocess the audio. The embeddings are computed and compared against the ground truth embeddings along with average similarities. The scores are then plotted.