SPEAKER RECOGNITION

Speaker Recognition is the problem of identifying a speaker from a recording of their speech. It is an important topic in Speech Signal Processing and has a variety of applications, especially in security systems. Voice controlled devices also rely heavily on speaker recognition.

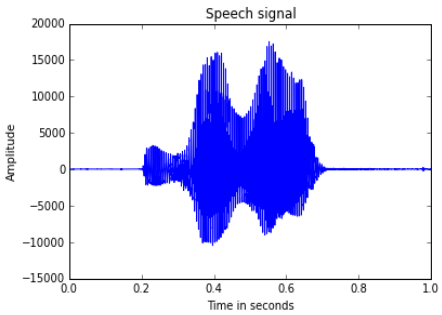
This is already a well-researched problem; my aim was not to come up with a new algorithm for speaker recognition, but to implement some already famous existing methods using Python. My motivation behind doing this independent project was to make a shift from MATLAB to Python for scientific computing. For this I primarily used the NumPy, SciPy and Matplotlib packages that have a huge repository for matrix manipulation, signal processing and plotting.

The main principle behind speaker recognition is extraction of features from speech which are characteristic to a speaker, followed by training on a data set and testing. I have relied heavily on the algorithm suggested in [1], where they extract the Mel-Frequency Cepstral Coefficients from each speaker and train them with Vector Quantization (using the LBG algorithm). I have also tried Vector Quantization by extracting the Linear Prediction Coefficients (LPCs) for training. The data set I have trained and tested on was downloaded from [1], and consists of 8 different female speakers uttering the word 'zero'. This data set is not extensive enough to give conclusive results (with only 8 training and test sets), and the results are far from satisfactory. However, my intention was to learn about and implement algorithms in Python, not carry out accurate tests. I wish to gather more data to extend this work.

1. Speech Signal

Digital speech is a one dimensional time-varying discrete signal as shown in **Figure 1a**. Various mathematical models of speech are available such as the Autoregressive Model and Sinusoidal + Residual model. A popular model of speech production says speech consists of a train of impulses of period equal to its pitch, added with random noise, controlled by a voiced/ unvoiced switch and modulated by the vocal tract which is a time-varying filter.

Speech is quasi-stationary in nature. For short intervals, the signal is stationary but over longer periods, the signal frequency varies. Hence a Short-Time Fourier Transform is needed to visualize its frequency content, as given in **Figure 1b** (also computed in Python with FFT size = Hanning window size = 256 samples and overlap of 50%). The 3D plot for STFT with time and frequency along x and y axis, and log amplitude indicated by intensity of the colour is called a Spectrogram.



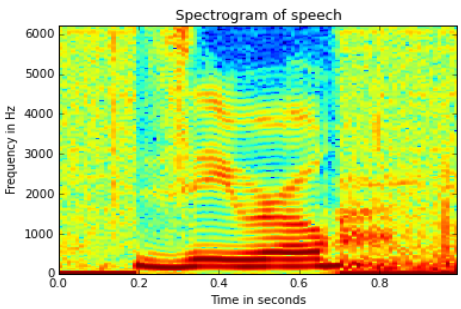


Figure 1

2. Feature Extraction

Choosing which features to extract from speech is the most significant part of speaker recognition. Some popular features are: MFCCs, LPCs, Zero-Crossing Rates etc. In this work, I have concentrated on MFCCs and LPCs. Here is a brief overview of these features.

2.1 Mel-Frequency Cepstral Coefficients

Human hearing is not linear but logarithmic in nature. This implies that our ear acts as a filter. MFCC's are based on the known variation of the human ear's critical bandwidths with frequency. Filters spaced linearly at low frequencies and logarithmically at high frequencies have been used to capture the phonetically important characteristics of speech. This is expressed in the *mel-frequency* scale. The relationship between frequency in Hz and frequency in Mel scale is given by:

$$m = 1125 \ln(1 + \frac{f}{700})$$
$$f = 700\ (e^{\frac{m}{1125}} - 1)$$

To calculate MFCCs, the steps are as follows. A very good tutorial is available in [2]. A schematic of this process is given in **Figure 2**.

1. The speech signal is divided into frames of 25ms with an overlap of 10ms. Each frame is multiplied with a Hamming window.
2. The periodogram of each frame of speech is calculated by first doing an FFT of 512 samples on individual frames, then taking the power spectrum as:

$$P(k) = \frac{1}{N} |S(k)|^2$$

   Where P(k) refers to power spectral estimate and S(k) refers to Fourier coefficients for the kth frame of speech and N is the length of the analysis window. The last 257 samples of the periodogram are preserved since it is an even function.
3. The entire frequency range is divided into *'n'* Mel filter banks, which is also the number of coefficients we want. 'For '*n*' = 12, the filter bank is shown in **Figure 3 -** a number of overlapping triangular filters with increasing bandwidth as the frequency increases.
4. To calculate filter bank energies we multiply each filter bank with the power spectrum, and add up the coefficients. Once this is performed we are left with *'n'* numbers that give us an indication of how much energy was in each filter bank.
5. We take the logarithm of these '*n*' energies and compute its Discrete Cosine Transform to get the final MFCCs.

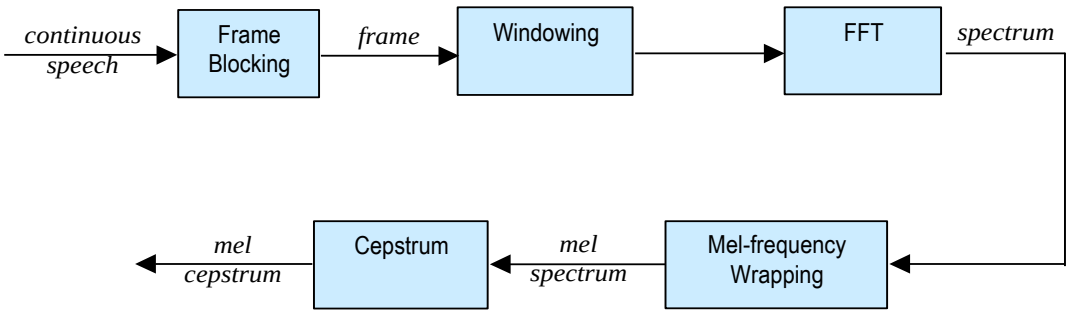


Figure 2 – MFCC Calculation Schematic

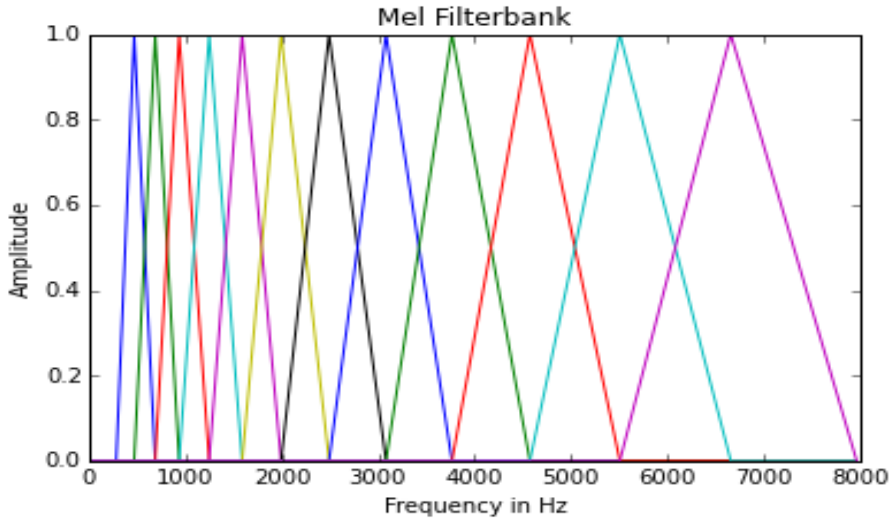


Figure 3 – 12 Mel Filter banks

The Python code for calculating MFCCs from a given speech file (.wav format) is shown in **Listing 1**.

```
from __future__ import division
from scipy.signal import hamming
from scipy.fftpack import fft, fftshift, dct
import numpy as np
import matplotlib.pyplot as plt

def hertz_to_mel(freq):
    return 1125*np.log(1 + freq/700)

def mel_to_hertz(m):
    return 700*(np.exp(m/1125) - 1)

#calculate mel frequency filter bank
def mel_filterbank(nfft, nfiltbank, fs):

    #set limits of mel scale from 300Hz to 8000Hz
    lower_mel = hertz_to_mel(300)
    upper_mel = hertz_to_mel(8000)
    mel = np.linspace(lower_mel, upper_mel, nfiltbank+2)
    hertz = [mel_to_hertz(m) for m in mel]
    fbins = [int(hz * (nfft/2+1)/fs) for hz in hertz]
    fbank = np.empty((nfft/2+1,nfiltbank))
    for i in range(1,nfiltbank+1):
        for k in range(int(nfft/2 + 1)):
            if k < fbins[i-1]:
                fbank[k, i-1] = 0
            elif k >= fbins[i-1] and k < fbins[i]:
                fbank[k,i-1] = (k - fbins[i-1])/(fbins[i] - fbins[i-1])
            elif k >= fbins[i] and k <= fbins[i+1]:
                fbank[k,i-1] = (fbins[i+1] - k)/(fbins[i+1] - fbins[i])
            else:
                fbank[k,i-1] = 0
    return fbank

def mfcc(s,fs, nfiltbank):

    #divide into segments of 25 ms with overlap of 10ms
    nSamples = np.int32(0.025*fs)
    overlap = np.int32(0.01*fs)
    nFrames = np.int32(np.ceil(len(s)/(nSamples-overlap)))

```

```
    #zero padding to make signal length long enough to have nFrames
    padding = ((nSamples-overlap)*nFrames) - len(s)
    if padding > 0:
        signal = np.append(s, np.zeros(padding))
    else:
        signal = s
    segment = np.empty((nSamples, nFrames))
    start = 0
    for i in range(nFrames):
        segment[:,i] = signal[start:start+nSamples]
        start = (nSamples-overlap)*i

    #compute periodogram
    nfft = 512
    periodogram = np.empty((nFrames,nfft/2 + 1))
    for i in range(nFrames):
        x = segment[:,i] * hamming(nSamples)
        spectrum = fftshift(fft(x,nfft))
        periodogram[i,:] = abs(spectrum[nfft/2-1:])/nSamples

    #calculating mfccs
    fbank = mel_filterbank(nfft, nfiltbank, fs)
    #nfiltbank MFCCs for each frame
    mel_coeff = np.empty((nfiltbank,nFrames))
    for i in range(nfiltbank):
        for k in range(nFrames):
            mel_coeff[i,k] = np.sum(periodogram[k,:]*fbank[:,i])

    mel_coeff = np.log10(mel_coeff)
    mel_coeff = dct(mel_coeff)
    #exclude 0th order coefficient (much larger than others)
    mel_coeff[0,:]= np.zeros(nFrames)
    return mel_coeff

```

Listing 1 – *mel_coefficients.py*

2.2 Linear Prediction Coefficients

LPCs are another popular feature for speaker recognition. To understand LPCs, we must first understand the Autoregressive model of speech. Speech can be modelled as a p^{th} order AR process, where each sample is given by:

$$x(n) = -\sum_{k=1}^{p} \alpha_k x(n-k) + u(n)$$

Each sample at the n^{th} instant depends on 'p' previous samples, added with a Gaussian noise u(n). This model comes from the assumption that a speech signal is produced by a buzzer at the end of a tube (voiced sounds), with occasional added hissing and popping sounds.

LPC coefficients are given by $\boldsymbol{\alpha}$. To estimate the coefficients, we use the Yule-Walker equations which are explained in [3]. It uses the autocorrelation function R_x. Autocorrelation at lag l is given by:

$$R(l) = \sum_{n=1}^{N} x(n)x(n-l)$$

While calculating ACF in Python, the Box-Jenkins method is used which scales the correlation at each lag by the sample variance so that the autocorrelation at lag 0 is unity.

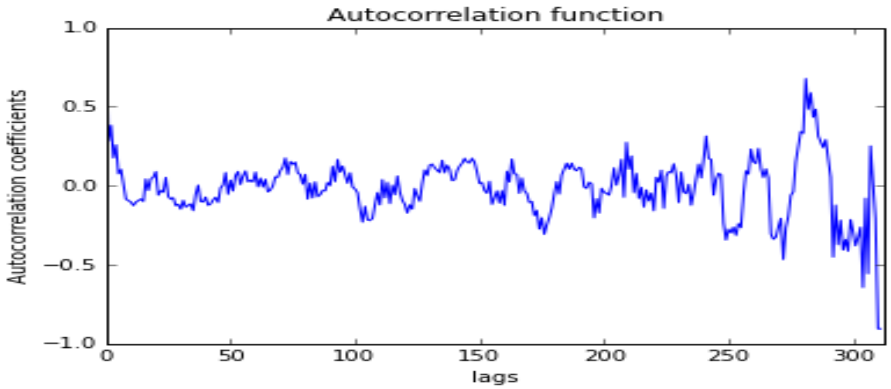


Figure 4 – ACF for one speech frame

The final form of the Yule-Walker equations is:

$$\sum_{k=1}^{p} \alpha_k R(l-k) = -R(l)$$

$$\begin{bmatrix} R(0) & R(1) & \dots & R(p-1) \\ R(1) & R(0) & \vdots & R(p-2) \\ \vdots & \vdots & \ddots & \vdots \\ R(p-1) & R(p-2) & \dots & R(0) \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_p \end{bmatrix} = - \begin{bmatrix} R(1) \\ R(2) \\ \vdots \\ R(p) \end{bmatrix}$$

The solution for $\boldsymbol{\alpha}$ is given by:

$$\boldsymbol{\alpha} = -\boldsymbol{R}^{-1}\boldsymbol{r}$$

In this case, I have normalised the LPC coefficients estimated so that they lie between [-1,1]. This was seen to give more accurate results. The Python code for calculating LPCs is given in **Listing 2**. We first divide speech into frames of 25ms with 10ms overlap, then calculate *'p'* LPCs for each frame.

```
1.  #calculate LPC coefficients from sound file
2.  from __future__ import division
3.  import numpy as np
4.
5.  def autocorr(x):
6.      n = len(x)
7.      variance = np.var(x)
8.      x = x - np.mean(x)
9.      #n numbers from last index
10.     r = np.correlate(x, x, mode = 'full')[-n:]
11.     result = r/(variance*(np.arange(n, 0, -1)))
12.     return result
13.
14. def createSymmetricMatrix(acf,p):
15.     R = np.empty((p,p))
16.     for i in range(p):
17.         for j in range(p):
18.             R[i,j] = acf[np.abs(i-j)]
```

```
    return R

def lpc(s,fs,p):

    #divide into segments of 25 ms with overlap of 10ms
    nSamples = np.int32(0.025*fs)
    overlap = np.int32(0.01*fs)
    nFrames = np.int32(np.ceil(len(s)/(nSamples-overlap)))

    #zero padding to make signal length long enough to have nFrames
    padding = ((nSamples-overlap)*nFrames) - len(s)
    if padding > 0:
        signal = np.append(s, np.zeros(padding))
    else:
        signal = s
    segment = np.empty((nSamples, nFrames))
    start = 0
    for i in range(nFrames):
        segment[:,i] = signal[start:start+nSamples]
        start = (nSamples-overlap)*i

    #calculate LPC with Yule-Walker
    lpc_coeffs = np.empty((p, nFrames))
    for i in range(nFrames):
        acf = autocorr(segment[:,i])
        r = -acf[1:p+1].T
        R = createSymmetricMatrix(acf,p)
        lpc_coeffs[:,i] = np.dot(np.linalg.inv(R),r)
        lpc_coeffs[:,i] = lpc_coeffs[:,i]/np.max(np.abs(lpc_coeffs[:,i]))

    return lpc_coeffs
```

Listing 2 – *LPC.py*

3. Feature Matching

The most popular feature matching algorithms for speaker recognition are Dynamic Time Warping (DTW), Hidden Markov Model (HMM) and Vector Quantization (VQ). Here, I have used Vector Quantization as suggested in [1].

.

VQ is a process of mapping vectors from a large vector space to a finite number of regions in that space. Each region is called a *cluster* and can be represented by its center called a *codeword*. The collection of all codewords is called a *codebook*.

Figure 5a shows a conceptual diagram to illustrate this recognition process. Only 2 feature dimensions are shown on a 2D plane for two different speakers. We can identify clusters of vectors in the plane. During training, a codeword is chosen for each cluster by minimizing the distortion between each vector in the cluster and the codeword. The collection of codewords forms a speaker specific codebook unique to each speaker. The codebook for each speaker is determined by the **LBG** algorithm, which is described later. To identify a speaker, the distance (or distortion) of the speaker's features from the all the trained codebooks is calculated. The codebook that has minimum distortion with the speaker's features is identified.

Figure 5b shows the actual 2D diagram of 5^{th} and 6^{th} features of two speakers, and their respective codebooks.

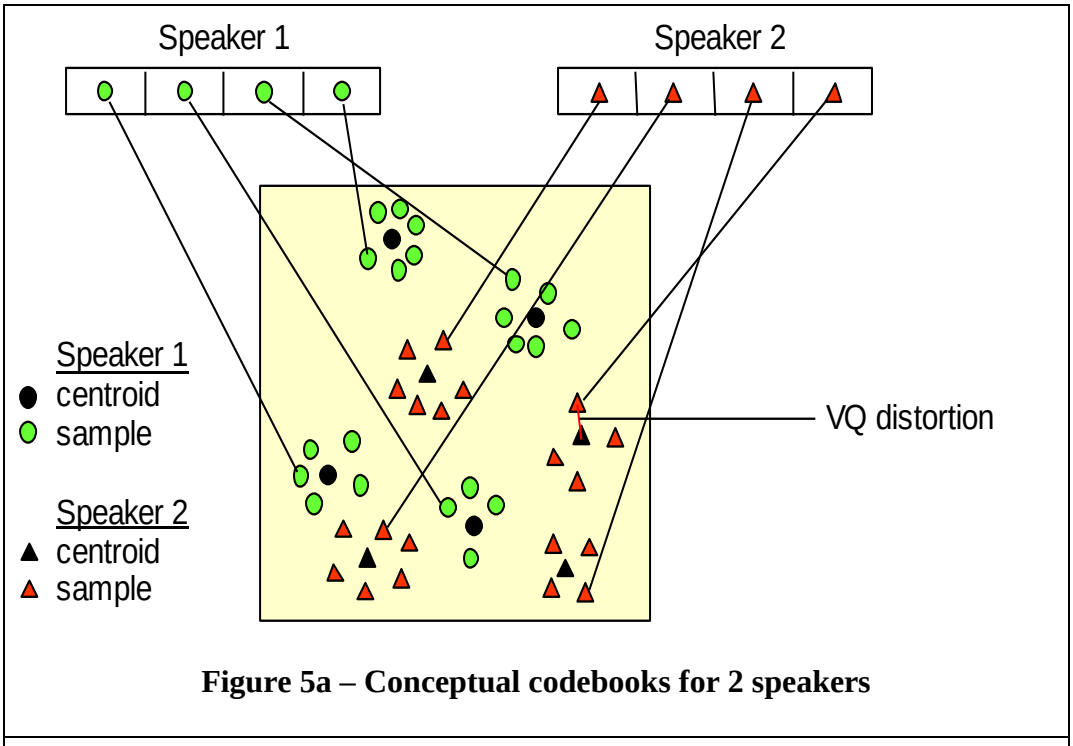


Figure 5a – Conceptual codebooks for 2 speakers

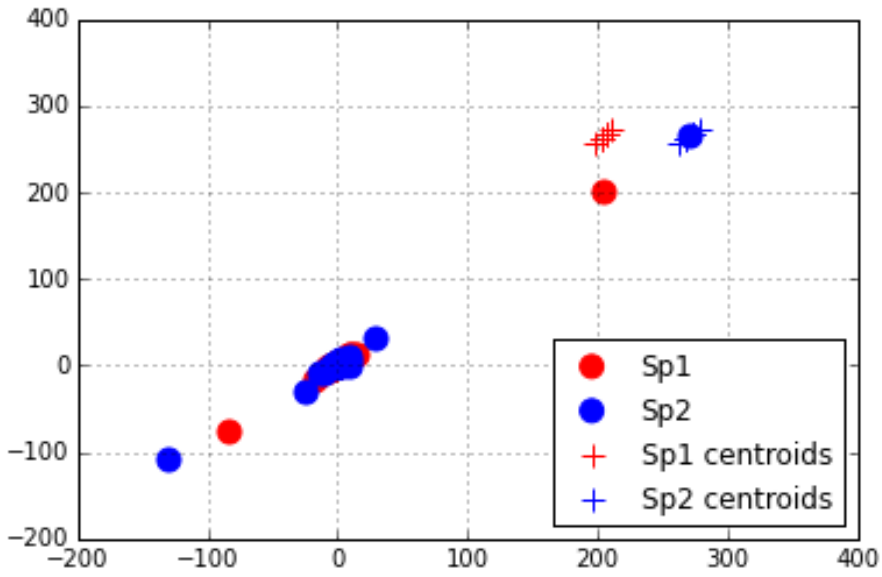


Figure 5b – Actual codebooks for 2 speakers

3.1 LBG Algorithm

The LBG algorithm [Linde, Buzo and Gray], is used for clustering a set of *L* training vectors into a set of *M* codebook vectors. The algorithm is formally implemented by the following recursive procedure:

1. Design a 1-vector codebook; this is the centroid of the entire set of training vectors (hence, no iteration is required here).
2. Double the size of the codebook by splitting each current codebook $\mathbf{y}_n$ according to the rule

$$\mathbf{y}_n^+ = \mathbf{y}_n(1+\varepsilon)$$
$$\mathbf{y}_n^- = \mathbf{y}_n(1-\varepsilon)$$

where n varies from 1 to the current size of the codebook, and ε is a splitting parameter (we choose ε =0.01).

3. Nearest-Neighbor Search: for each training vector, find the codeword in the current codebook that is closest (in terms of similarity measurement), and assign that vector to the corresponding cell (associated with the closest codeword).

4. Centroid Update: update the codeword in each cell using the centroid of the training vectors assigned to that cell.

5. Iteration 1: repeat steps 3 and 4 until vector distortion for current iteration falls below a fraction of the pervious iteration's distortion. This is to ensure that the process has converged.

6. Iteration 2: repeat steps 2, 3 and 4 until a codebook size of *M* is designed.

Intuitively, the LBG algorithm designs an *M*-vector codebook in stages. It starts first by designing a 1-vector codebook, then uses a splitting technique on the codewords to initialize the search for a 2-vector codebook, and continues the splitting process until the desired *M*-vector codebook is obtained.

Listing 3 gives the details of implementing the LBG algorithm in Python. The print statements should be used for debugging, especially to see if distortion is converging.

```
1.  #speaker specific Vector Quantization codebook using LBG algorithm
2.  from __future__ import division
3.  import numpy as np
4.
5.  #calculate Euclidean distance between two matrices
6.  def EUDistance(d,c):
7.
8.      # np.shape(d)[0] = np.shape(c)[0]
9.      n = np.shape(d)[1]
10.     p = np.shape(c)[1]
11.     distance = np.empty((n,p))
12.
13.     if n<p:
14.         for i in range(n):
15.             copies = np.transpose(np.tile(d[:,i], (p,1)))
16.             distance[i,:] = np.sum((copies - c)**2,0)
17.     else:
18.         for i in range(p):
19.             copies = np.transpose(np.tile(c[:,i],(n,1)))
20.             distance[:,i] = np.transpose(np.sum((d - copies)**2,0))
21.
22.     distance = np.sqrt(distance)
23.     return distance
24.
25. def lbg(features, M):
26.     eps = 0.01
27.     codebook = np.mean(features, 1)
28.     distortion = 1
29.     nCentroid = 1
30.     while nCentroid < M:
31.
32.         #double the size of codebook
33.         new_codebook = np.empty((len(codebook), nCentroid*2))
34.         if nCentroid == 1:
```

```
35.             new_codebook[:,0] = codebook*(1+eps)
36.             new_codebook[:,1] = codebook*(1-eps)
37.         else:
38.             for i in range(nCentroid):
39.                 new_codebook[:,2*i] = codebook[:,i] * (1+eps)
40.                 new_codebook[:,2*i+1] = codebook[:,i] * (1-eps)
41.
42.         codebook = new_codebook
43.         nCentroid = np.shape(codebook)[1]
44.         D = EUDistance(features, codebook)
45.
46.         while np.abs(distortion) > eps:
47.             #nearest neighbour search
48.             prev_distance = np.mean(D)
49.             nearest_codebook = np.argmin(D,axis = 1)
50.
51.             #cluster vectors and find new centroid
52.             for i in range(nCentroid):
53.                 #add along 3rd dimension
54.                 codebook[:,i] = np.mean(features[:,np.where(nearest_codebook == i)], 2).T
55.
56.             #replace all NaN values with 0
57.             codebook = np.nan_to_num(codebook)
58.             D = EUDistance(features, codebook)
59.             distortion = (prev_distance - np.mean(D))/prev_distance
60.             #print 'distortion' , distortion
61.
62.     #print 'final codebook', codebook, np.shape(codebook)
63.     return codebook
64.
65.
```

Listing 3 – *LBG.py*

4. Feature Training

The main algorithms needed for speaker recognition have been implemented. Now, everything needs to be brought together to train our dataset and derive codebooks for each speaker using VQ. The Python code is given in Listing 4. I have hard-coded the name of the directory where the speech files are stored and the .wav filenames, but that can be easily changed by giving them as parameters to the **training( )** function. The number of speakers is **nSpeaker** = 8. As mentioned before, speech recordings of 8 female speakers uttering the word 'zero' has been taken for training and testing. Each codebook should have 16 codewords, hence **nCentroid** = 16 (it is highly recommended to keep this number a power of 2).

Codebooks for both MFCC features and LPC features are plotted for all 8 speakers. One of them is shown in Figure 6. Lines 44 to 61 may be commented out as they are only used to plot the 5th and 6th dimension MFCC features for the first two speakers on a 2D plane.

```
1.  from __future__ import division
2.  import numpy as np
3.  from scipy.io.wavfile import read
4.  from LBG import lbg
5.  from mel_coefficients import mfcc
6.  from LPC import lpc
7.  import matplotlib.pyplot as plt
8.
9.  def training(nfiltbank, orderLPC):
10.     nSpeaker = 8
11.     nCentroid = 16
```

```
    codebooks_mfcc = np.empty((nSpeaker,nfiltbank,nCentroid))
    codebooks_lpc = np.empty((nSpeaker, orderLPC, nCentroid))
    directory = 'C:/Users/ORCHISAMA/Documents/Audio Signal Processing/Speaker recognition/test'
    fname = str()

    for i in range(nSpeaker):
        fname = '/s' + str(i+1) + '.wav'
        print 'Now speaker ', str(i+1), 'features are being trained'
        (fs,s) = read(directory + fname)
        mel_coeff = mfcc(s, fs, nfiltbank)
        lpc_coeff = lpc(s, fs, orderLPC)
        codebooks_mfcc[i,:,:] = lbg(mel_coeff, nCentroid)
        codebooks_lpc[i,:,:] = lbg(lpc_coeff, nCentroid)

        plt.figure(i)
        plt.title('Codebook for speaker ' + str(i+1) + ' with ' + str(nCentroid) +  ' centroids')

        for j in range(nCentroid):
            plt.subplot(211)
            plt.stem(codebooks_mfcc[i,:,j])
            plt.ylabel('MFCC')
            plt.subplot(212)
            markerline, stemlines, baseline = plt.stem(codebooks_lpc[i,:,j])
            plt.setp(markerline,'markerfacecolor','r')
            plt.setp(baseline,'color', 'k')
            plt.ylabel('LPC')
            plt.axis(ymin = -1, ymax = 1)
            plt.xlabel('Number of features')

    plt.show()
    print 'Training complete'

    #plotting 5th and 6th dimension MFCC features on a 2D plane
    codebooks = np.empty((2, nfiltbank, nCentroid))
    mel_coeff = np.empty((2, nfiltbank, 68))

    for i in range(2):
        fname = '/s' + str(i+1) + '.wav'
        (fs,s) = read(directory + fname)
        mel_coeff[i,:,:] = mfcc(s, fs, nfiltbank)[:,0:68]
        codebooks[i,:,:] = lbg(mel_coeff[i,:,:], nCentroid)

    plt.figure(nSpeaker + 1)
    s1 = plt.scatter(mel_coeff[0,4,:], mel_coeff[0,5,:], s = 100, color = 'r', marker = 'o')
    c1 = plt.scatter(codebooks[0,4,:], codebooks[1,5,:], s = 100, color = 'r', marker = '+')
    s2 = plt.scatter(mel_coeff[1,4,:], mel_coeff[1,5,:], s = 100, color = 'b', marker = 'o')
    c2 = plt.scatter(codebooks[1,4,:], codebooks[1,5,:], s = 100, color = 'b', marker = '+')
    plt.grid()
    plt.legend((s1, s2, c1, c2), ('Sp1','Sp2','Sp1 centroids', 'Sp2 centroids'),scatterpoints=1,
    loc = 'lower right')
    plt.show()

    return (codebooks_mfcc, codebooks_lpc)
```

Listing 4 – *train.py*

5. Testing

It is finally time to test our speaker recognition algorithm. Listing 5 contains the code for identifying the speaker by comparing their feature vector to the codebooks of all trained speakers and computing the minimum distance between them. Heads up, the results are not as accurate as I thought they'd be,

yielding an accuracy of **50% with MFCC and 37.5% with LPC**. Reasons for this low accuracy can be the fact that there wasn't enough data to train on. Other complex classification algorithms such as ANNs and SVMs should yield better results. I observed that training with**12 MFCC features** and **LPC coefficients of order 15** gives the best results. There are other parameters that can be varied, such as number of codewords in a codebook and FFT size while computing MFCCs. It is possible that a different combination of these will give higher accuracy.

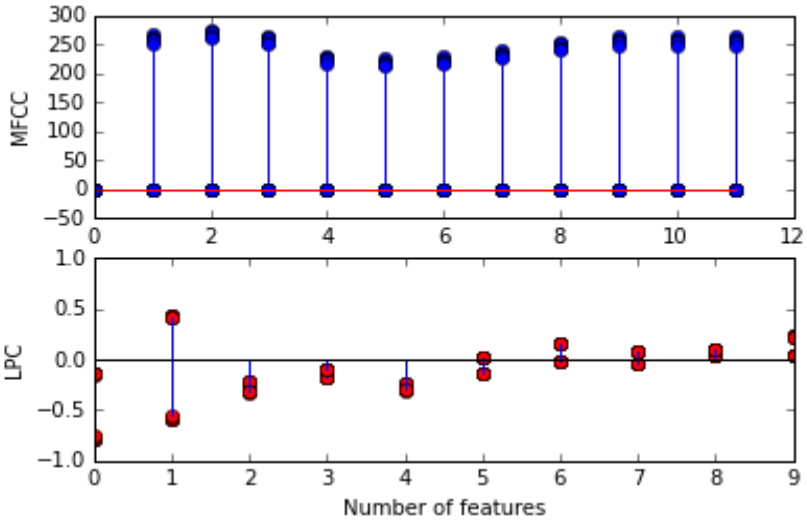


Figure 6 – Codebooks for a) MFCC features and b) LPC features

```
from __future__ import division
import numpy as np
from scipy.io.wavfile import read
from LBG import EUDistance
from mel_coefficients import mfcc
from LPC import lpc
from train import training

nSpeaker = 8
nfiltbank = 12
orderLPC = 15
(codebooks_mfcc, codebooks_lpc) = training(nfiltbank, orderLPC)
directory = 'C:/Users/ORCHISAMA/Documents/Audio Signal Processing/Speaker recognition/train'
fname = str()
nCorrect_MFCC = 0
nCorrect_LPC = 0

def minDistance(features, codebooks):
    speaker = 0
    distmin = np.inf
    for k in range(np.shape(codebooks)[0]):
        D = EUDistance(features, codebooks[k,:,:])
        dist = np.sum(np.min(D, axis = 1))/(np.shape(D)[0])
        if dist < distmin:
            distmin = dist
            speaker = k
    return speaker

for i in range(nSpeaker):
    fname = '/s' + str(i+1) + '.wav'
```

```
    print 'Now speaker ', str(i+1), 'features are being tested'
    (fs,s) = read(directory + fname)
    mel_coefs = mfcc(s,fs,nfiltbank)
    lpc_coefs = lpc(s, fs, orderLPC)
    sp_mfcc = minDistance(mel_coefs, codebooks_mfcc)
    sp_lpc = minDistance(lpc_coefs, codebooks_lpc)

    print 'Speaker', (i+1), ' in test matches with speaker ', (sp_mfcc+1), 'in train for training
    with MFCC'
    print 'Speaker', (i+1), ' in test matches with speaker ', (sp_lpc+1), 'in train for training
    with LPC'

    if i == sp_mfcc:
        nCorrect_MFCC += 1
    if i == sp_lpc:
        nCorrect_LPC += 1


percentageCorrect_MFCC = (nCorrect_MFCC/nSpeaker)*100
print 'Accuracy of result for training with MFCC is ', percentageCorrect_MFCC, '%'
percentageCorrect_LPC = (nCorrect_LPC/nSpeaker)*100
print 'Accuracy of result for training with LPC is ', percentageCorrect_LPC, '%'
```

Listing 5 – *test.py*

6. Results

The following table gives the identification results for each of the 8 speakers with MFCC and LPC coefficients and Vector Quantization with LBG algorithm for classification.

Serial Number	True speaker	Identified as (MFCC)	Identified as(LPC)
1.	S1	S1	S3
2.	S2	S6	S2
3.	S3	S6	S3
4.	S4	S8	S7
5.	S5	S3	S1
6.	S6	S6	S7
7.	S7	S7	S7
8.	S8	S8	S7
		Accuracy = 50%	**Accuracy = 37.5%**

7. Wrapping up

I had a lot of fun doing this small project, understanding algorithms and implementing them in Python. It was a good learning experience, and the resources available online helped a lot. I would like to extend it by experimenting with more features and more classification algorithms. This was not an original research work (I don't want to be accused of plagiarism ☺), but rather an exploration and implementation of existing methods (the code is mine). It might be particularly helpful for those who want to get started with Python for signal processing, or those who want to explore the vast topic of Speech Processing.

Supplement

I would like to include the Python program for plotting the spectrogram of a signal by doing an STFT. A spectrogram carries fundamental information about speech signals, and it is a basic tool that is used in all audio analysis. The default window type has been taken as a Hanning window, and the window length is equal to the number of FFT points.

```
#calculate short time fourier transform and plot spectrogram
from __future__ import division
import matplotlib.pyplot as plt
import numpy as np
from scipy.fftpack import fft, fftshift
from scipy.signal import hann

def nearestPow2(inp):
    power = np.ceil(np.log2(inp))
    return 2**power

def stft(signal, fs, nfft, overlap):
    #plotting time domain signal
    plt.figure(1)
    t = np.arange(0,len(signal)/fs, 1/fs)
    plt.plot(t,signal)
    plt.axis(xmax = 1)
    plt.xlabel('Time in seconds')
    plt.ylabel('Amplitude')
    plt.title('Speech signal')

    if not np.log2(nfft).is_integer():
        nfft = nearestPow2(nfft)
    slength = len(signal)
    hop_size = np.int32(overlap * nfft)
    nFrames = int(np.round(len(signal)/(nfft-hop_size)))
    #zero padding to make signal length long enough to have nFrames
    signal = np.append(signal, np.zeros(nfft))
    STFT = np.empty((nfft, nFrames))
    segment = np.zeros(nfft)
    start = 0
    for n in range(nFrames):
        segment = signal[start:start+nfft] * hann(nfft)
        padded_seg = np.append(segment,np.zeros(nfft))
        spec = fftshift(fft(padded_seg))
        spec = spec[len(spec)/2:]
        spec = abs(spec)/max(abs(spec))
        powerspec = 20*np.log10(spec)
        STFT[:,n] = powerspec
        start = start + nfft - hop_size

    #plot spectrogram
    plt.figure(2)
    freq = (fs/(2*nfft)) * np.arange(0,nfft,1)
    time = np.arange(0,nFrames)*(slength/(fs*nFrames))
    plt.imshow(STFT, extent = [0,max(time),0,max(freq)], origin='lower',
    cmap='jet', interpolation='nearest', aspect='auto')
    plt.ylabel('Frequency in Hz')
    plt.xlabel('Time in seconds')
    plt.axis([0,max(time),0,np.max(freq)])
    plt.title('Spectrogram of speech')
    plt.show()
    return (STFT, time, freq)
```

Listing 6 – spectrogram.py