EE-679 Speech Processing

Digit Speech Recognition

Agrim Gupta | 140020003 | EE-DD-CSP, Fourth Year

1. End Pointer to segment the speech

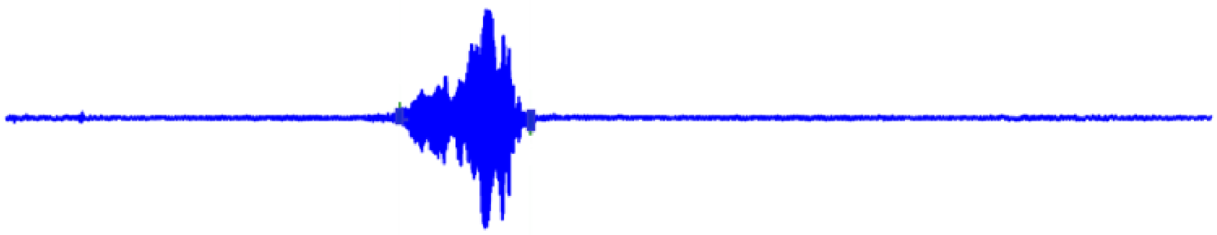
I tried out the following two approaches to segment the speech into segments:

- Normalising the s-t energy of the spectrum, and setting threshold for speech detection (Code in Appendix 1.a)
- Using ZCT and 2 tier thresholding (Based on paper by Rabiner and Sambur, code in Appendix 1.b)

In both of the above methods, tuning up the threshold was required heavily, and affected the output generated from the method. Although it seemed that the second method would perform better, but it was only marginally better, and required even further fine tuning for various speakers.

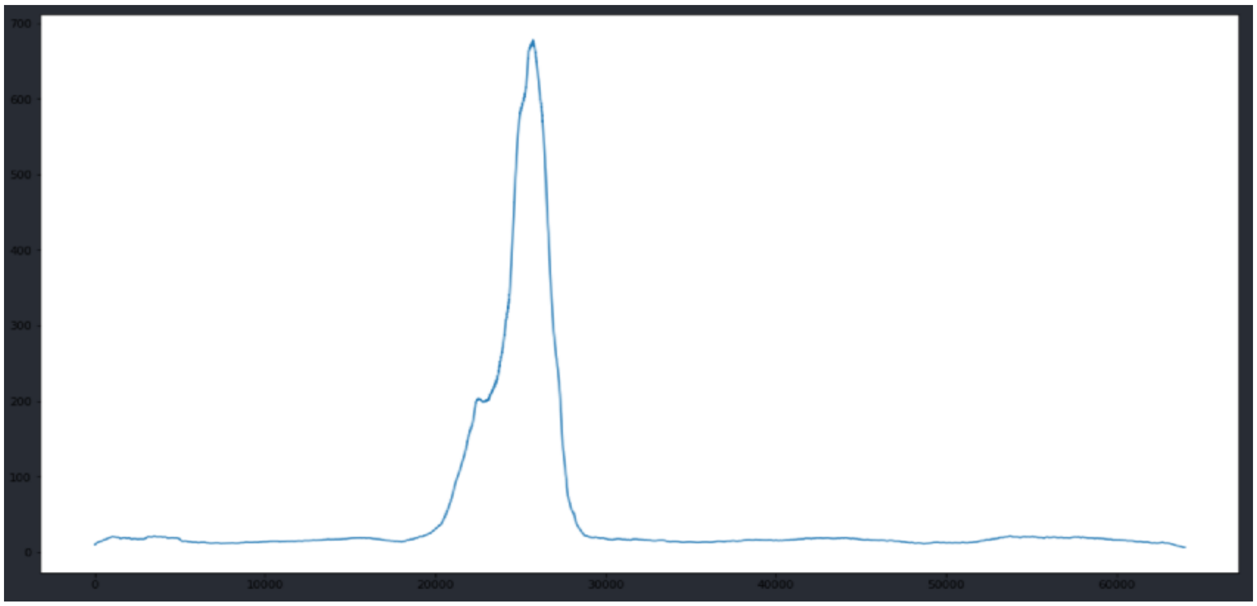
The Basic idea in both of the methods is to use the s-t energy plotl, and do thresholding to obtain the signal. Working of the first method is shown in the attached images.

Original Signal Plot. Amplitude vs Time (samples)

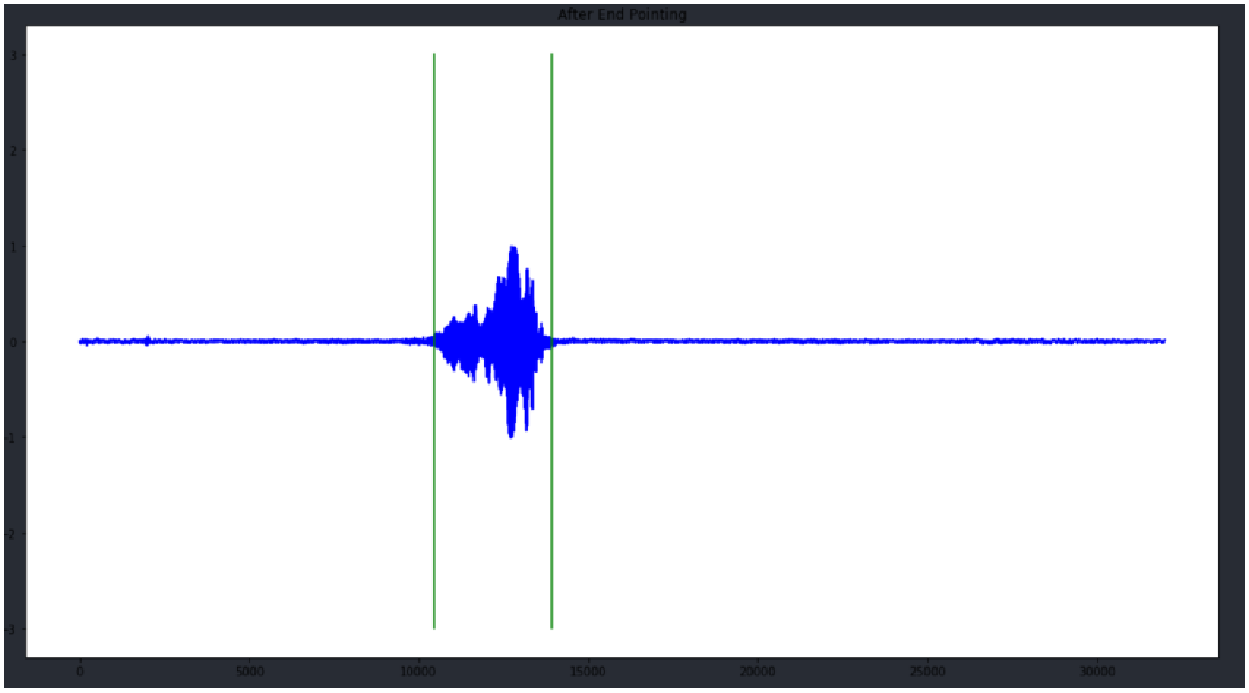


We want to get the high energy zone in middle. Firstly, we compute short time energy plot of the input waveform above, to get a plot like this :

Short Time Energy Plot: s-t Energy vs time (samples)

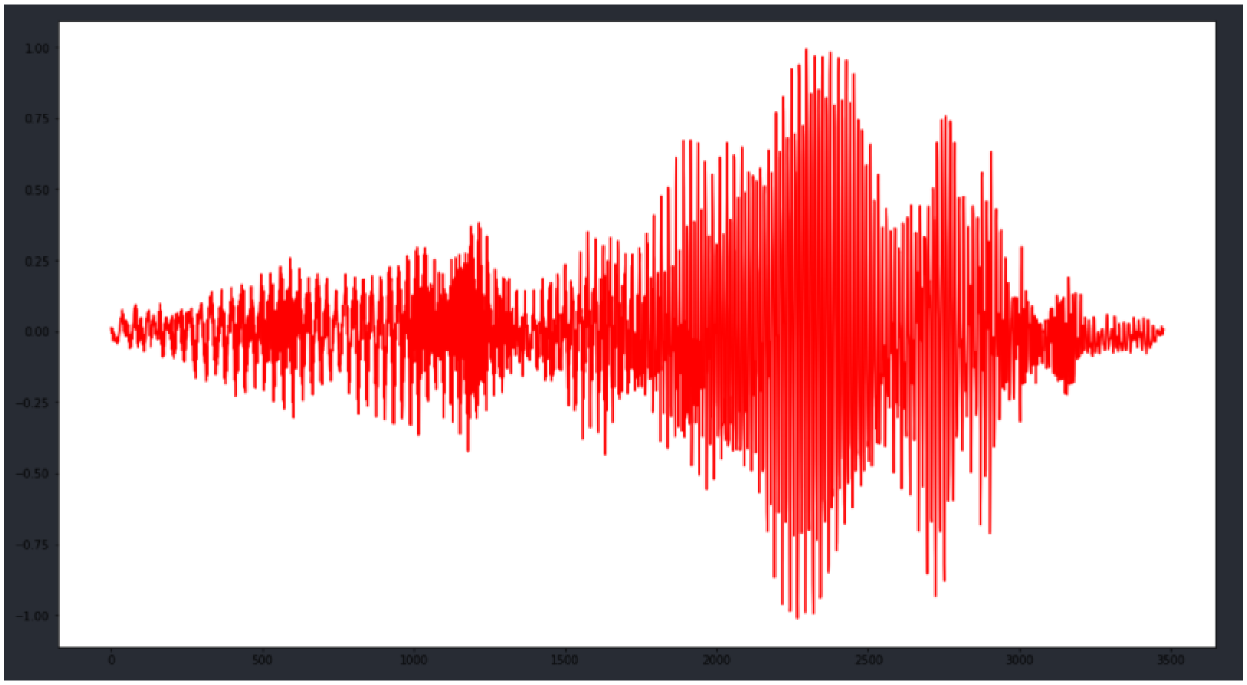


This is comparatively smoother and our hope is to use thresholding to detect the end points of the signal. By appropriate thresholding, I got a signal like the below, indicating the successful of end pointer algorithm. The Green lines represent the detected end points.



And finally, after appropriately slicing the input signal at the end points, we get the required processed signal without silence.

Processed signal vs time (samples) plot

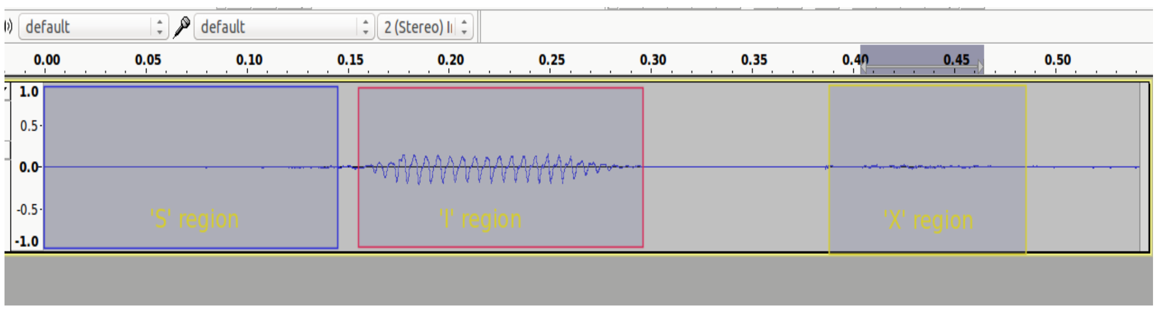


Another way of verifying the end pointer working is to listen to the wav file generated.

To help integrate with the other modules, I stored the wav files generated by end pointing separately (in ./Processed_Data/digit/samp_no.wav), i.e. for each speaker, I obtained 40 wav files, corresponding to 4 utterances of each digit. Also, the speakers were given ids, i.e. Speaker 1 has the files 1.wav,2.wav,3.wav and 4.wav. Similarly, speaker 2 has files 5.wav, 6.wav, 7.wav and 8.wav. Refer to the files in the directory (“**Processed Data”**) for further clarity on the scheme.

Takeaways and Key Points observed:

- **Choice of Higher threshold:** Choosing a higher threshold helps detection. This is because, if we have a conservative estimate of threshold, there is a good amount of silence in between. What I observed was that in these kinds of end pointing, there is a lot of confusion between the digits and 6. Looking up the plot of 6, also supports my claim here.



  As you can see in the above plot for ‘6’, there is a lot of silence (due to ‘s’ and ‘x’), and hence, having lower thresholds cause many confusions with 6. Hence, it is indeed better to choose a higher threshold and pay the penalty of missing the initial samples, then getting confused with 6.

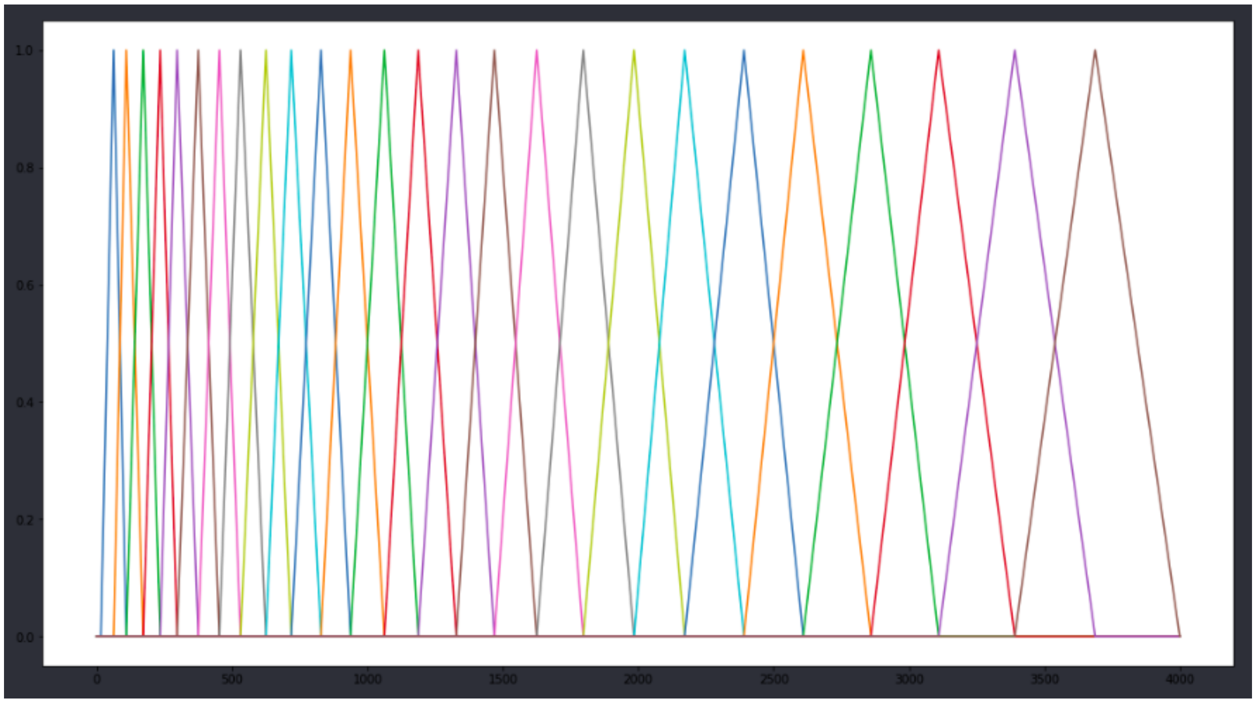
  (Plot for ‘6’ obtained using **‘Audacity’** software for Ubuntu)

- **Manual tuning to obtain better endpointing:** As indicated above, we can actually detect if the endpointing has been done properly or not by observing per speaker accuracy in “Bag of Frames” method. Taking feedback from that method, I used **Audacity** to further do manual endpointing for the speakers I was getting bad accuracy for. This also improved the accuracy

2. Developing a MFCC Feature Extractor

I designed MFCC feature extractor on lines of what was taught in class and used the tutorial uploaded on moodle to generate mel filter banks, shown in the figure on the next page. I generated 26 (standard) filter banks, as specified, but used the first 13 only, since the higher order filter banks are not required for ASR purposes (since speech formants etc are concentrated in lower frequencies). The higher order filter banks are more useful when analysing music, which has components of high frequencies in majority. You can find the MFCC code in Appendix 2.

Plot of Mel Filters generated vs Frequency(Hz): 26 filters generated between 0-4 kHz



Using the MFCC function by library actually gives much higher accuracy, since they use DCT instead of IFFT, which further utilises sparsity in signal to give better accuracy.

Again, to integrate it with the detection schemes smoothly, I stored the numpy arrays of features in "./**Extracted_Feats**/_digit_" directory. Each folder has 64 .npy files, which are the MFCCs of the 64 utterances of that digit. Thus, for detection code becomes easier and we can read directly from the files, saving time as well.

I got reasonably good accuracy without using spectral dynamic features which are captured by delta and delta-delta coefficients. However, when tested with my voice in relatively noisy environment, the need for using it was felt (Discussed in detail in Section 5)

3a. Bag of Frames method

For the bag of frames method, I tweaked the distance calculation slightly, and instead of the vanilla sum of squares, or the mean of sum of squares metric, I did sum of minimum distances from the **testing frame** to the reference frames. That is, say, WLOG that we are interested in estimating the distances between the reference digits and digit 0 spoken. Ideally digit 0 should have closest distance with digit 0 in the reference digits.

Say, the given utterance of digit 0 has $\mathbf{n_0}$ frames. We are testing it against '**dth bag**' in our reference data. Say, we have **k** samples for the **d** digit in our bag. Say, each **ith** sample, where i goes from **1** to **k,** has $\mathbf{f_i}$ number of frames.

Now, what I'm doing is, R_{0d} being the distance measure between digit 0 and digit d

$$R_{0d} = \sum_{i=1}^{k} \sum_{j=1}^{n_0} min_{f_i} \; (square\ distance\ between\ frame\ f_i\ and\ frame\ j)$$

See the below code snippet and comments in it for more clarifications.

```
for d in range(10):
            dist=0
            count=1
            for i in exc_samp:
                filename="./Extracted_Feats/"+str(d)+"/"+str(i)+".npy"
                tarr=np.load(filename)
                # Find all closest distances to the frames. Zr: test frames
                for j in range(zr.shape[0]):
                    min_dist_to_a_frame=0
                    # Find Closest Matching Frame from one zr frame to tarr frames
                    for k in range(tarr.shape[0]):
                        obt_dist=np.sum(np.abs(zr[j]-tarr[k]))
                        if(k==0):
                            min_dist_to_a_frame=obt_dist
                        else:
                            if(obt_dist<min_dist_to_a_frame):
                                min_dist_to_a_frame=obt_dist
                    # Add the minimum distance you found
                    dist=dist+min_dist_to_a_frame
                    count=count+1
                # Dist now contains
            #Avg min dist, used as a metric for prediction
            pred[d]=(dist/count)
```

Using the custom distance measure instead of naive sum of squares/mean of sum of squares improved the accuracy **drastically**. It was around 60 % for the naive measures, and increased to 82.5%. This maybe due to the schemes closeness with DTW. (The scheme can be looked upon as an approximation to DTW distance discussed ahead, to explain the observations)

Results and confusion observed:

- I got **82.5 %** accuracy with this method. Individual accuracies of the digits were:

Digit	0	1	2	3	4	5	6	7	8	9
Accuracy (%)	98.4	80	98.4	**50**	96.8	86	96.8	81.25	46.9	90.62

Digits 3 and 8 were most confused with. This is probably due to the occurrence of “double vowels” (thr**ee** and **ei**ght), followed by fricative like sounds. Thus, majority of energy lies with double vowels, which cause confusions with other digits having the same vowels, as seen from the confusion matrix **(obtained after 16 fold cross validation)** below

Predicted Digit→ Spoken Digit ↓	0	1	2	3	4	5	6	7	8	9
0	**63**	0	0	0	0	0	0	0	0	1
1	2	**51**	0	0	0	1	1	0	0	**9**
2	0	0	**63**	0	0	0	1	0	0	0
3	11	0	0	**32**	0	0	5	0	0	**16**
4	0	0	2	0	**62**	0	0	0	0	0
5	1	0	0	0	0	**55**	1	0	0	6
6	0	0	0	1	0	0	**62**	0	0	1
7	0	0	0	0	0	0	1	**52**	0	11
8	0	2	0	**7**	0	4	**18**	0	**30**	3
9	0	1	0	0	0	5	0	0	0	**58**

Thr**ee** gets most confused with z**e**ro, and nin**e.** It may appear that three should also get confused with seven, but when you look at the IPA spelling, of seven, i.e. /ˈsev(ə)n/, you realise that the main energy in seven resides in ə and not e. Also, it has reasonably good co-articulation with the nasal **n**, which makes it unique and thus escapes from confusion.

Eight get most confused with s**i**x, due to **i**. Eight also gets confused with three, due to effect of ‘**e**’ .

Looking up the IPA spelling of three, which is /ϴri/, where ϴ denotes the glottal stop, which also occurs at end of nine everytime, further increases the confusion between 3 and 9.

Another notable confusion is confusion between o**ne** and ni**ne.** This was discussed in class as well and is primarily due to common ‘**ne**’ in the two utterances. However, a strange fact is that one get notably confused with nine, but the same is not true for confusion of nine with one.

The detailed code is in Appendix 3a, and log of the output I received from the execution is in “**./Output_Logs/bag_of_frames.txt”**. The log text file contains all the predictions the algorithm made for various speakers.

3b. Vector Quantization (k-means) method

For VQ method, I obtained the clusters for each digit separately, with 4 and 8 feature vectors as indicated in Q3. Illustrating further on my method, what I did was :

" WLOG assume that we are testing it for speaker 1. So, to obtain the centroids for testing of speaker 1, I concatenated all MFCCs for each frame and each sample for all speakers albeit 1, and then used K means over the concatenated array of MFCCs to obtain 4 (and 8) centroids for each digit"

WLOG assume we are calculating centroids for speaker 1, and for digit one (i.e. centroid$_{11}$). Since there are 4 samples of sound for each speaker, we concatenate the 4 MFCCs (in "**./Extracted_Feats"** for each speaker from 2 to 12.

I.e.

$$centroid_{11} = k - means((4,8), [(.., mfcc(s_{j1}(1)) \ ++ \ mfcc(s_{j1}(2)) \ ++ \ mfcc(s_{j1}(3)) ++ \ mfcc(s_{j1}(4), .. \forall j \ \in \ (2,16))])$$

Where s_{kd}(n) denotes speaker k's, d digit's nth utterance, and '++' denoted the append operator. The distance measure between two MFCCs is the same as in section 2.

Results and confusion observed:

Confusion matrix for VQ (4 clusters): Net Accuracy: **83.9 %**

Predicted Digit→ Spoken Digit ↓	0	1	2	3	4	5	6	7	8	9
0	**62**	0	2	0	0	0	0	0	0	0
1	1	**51**	**10**	0	0	1	0	0	0	1
2	1	0	**60**	0	2	0	0	0	1	0
3	**6**	0	0	**48**	0	1	0	0	**6**	3
4	1	2	2	0	**58**	0	3	1	0	0
5	2	2	0	1	0	**48**	3	0	1	7
6	0	0	0	1	0	3	**59**	0	1	0
7	0	1	0	0	0	0	1	**62**	0	1
8	1	0	2	7	0	8	5	0	**40**	1
9	1	2	0	0	0	4	0	7	1	**49**

Confusion matrix for VQ (8 clusters): Net Accuracy: **88%**

Predicted Digit→ Spoken Digit ↓	0	1	2	3	4	5	6	7	8	9
0	**64**	0	0	0	0	0	0	0	0	0
1	0	**59**	2	0	0	0	0	1	0	2
2	**6**	2	**53**	0	2	0	0	0	1	0
3	2	0	0	**56**	0	0	0	1	**5**	0
4	0	2	3	0	**59**	0	0	0	0	0
5	2	3	0	0	0	**51**	1	0	0	**8**
6	0	0	2	1	1	1	**53**	0	6	0
7	0	0	0	0	0	0	0	**60**	0	4
8	0	2	1	**5**	0	**4**	**4**	0	**46**	2
9	0	1	0	0	0	0	0	1	0	**62**

Further comments:

- Since there are more representative vectors in 8 clusters, we expect it to be more accurate, which we observe as well here.
- The 3 and 8 anomaly discussed before can be observed here as well, albeit, to a lesser degree. Three gets confused with 8 and vice versa. However, it is worse for 8 here, as it gets confused with f**i**ve and s**i**x as well, due to the double vowel (Light yellow shaded cells). However, accuracy of three is reasonably higher than eight.
- We observe another interesting confusion here, which is between **5** and **9** (blue shaded cell). Observing the word structure, f**ive** and n**ine**, we get the reason why are these digits getting confused with. Since, most of the energy in a word is held by vowels, the confusion makes sense, and was discussed in class as well.
- Another confusion is between **0** and **2** (Green shaded cell, in k=8 confusion matrix), which maybe due to tw**o** and zer**o**. The effect is not symmetric, as z**e**ro has an 'e' as well which protects it from getting confused with two.
- Also,there are two more anomalies in confusion matrix of k=4 matrix. See the cyan cell, indicating confusion between **7** and **9**. Also, see the magenta cell indicating confusion between **1** and **2.** These again maybe due to common vowels between the words, but the magnitude of these are sort of unexpected, and maybe due to some random error.

You can divide the diagonal entries by 64 to get per digit accuracy.

The codes for getting the centroids, and as well as getting the VQ predictions are in Appendix 3b. The log output is in "**./Output_Logs/VQ4_8.txt**"

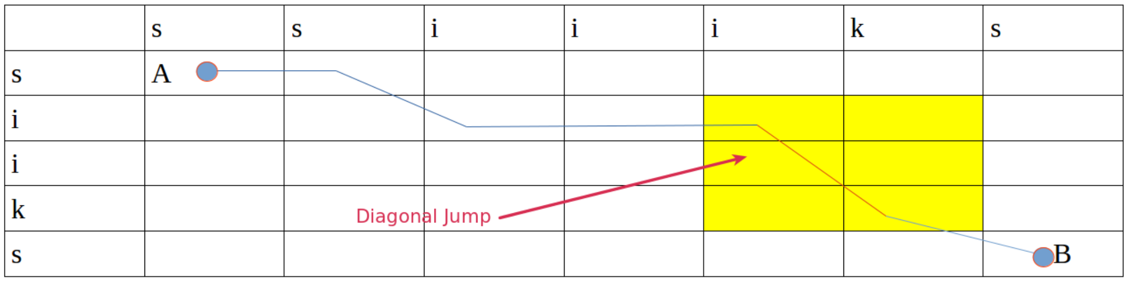
4. DTW method

In DTW method, we replace the distance measure discussed earlier in section 2 is replaced by an even smarter DTW distance scheme. DTW minimises the sum of distance between the two MFCCs, whereas the distance metric we used in sections 2 and 3 minimise the distance between 2 frames only. Hence, DTW is expected to perform even better, since it brings along the temporal order into the scenario as well.

So, in a nutshell DTW algorithm finds a warping function m=w(n), which matches the time axis of ref. and test, by going through the template & solving the foll. optimisation problem:

$$DTW \text{ distance between } T(n) \text{ and } R(n) = \sum_{n=1}^{T} d(T(n), R(w(n)))$$

Example: Utterance \siiks\ is to be matched with \ssiiiks\ (each alphabet kindof denotes a frame). We find the shortest path between A and B, with horiz. & diag. Movements & jumps



Confusion Matrix for DTW: Accuracy = **91.72%**

Predicted Digit→ Spoken Digit ↓	0	1	2	3	4	5	6	7	8	9
0	**63**	0	1	0	0	0	0	0	0	0
1	0	**60**	0	0	0	1	0	0	0	3
2	0	0	**63**	0	0	0	1	0	0	0
3	0	0	0	**62**	0	0	1	0	0	1
4	0	0	4	0	**60**	0	0	0	0	0
5	0	2	0	0	0	**55**	0	0	0	7
6	0	0	0	1	0	0	**62**	0	1	0
7	0	0	2	0	0	0	0	**62**	0	0
8	0	2	0	7	1	0	16	0	**38**	0
9	0	0	0	0	0	2	0	0	0	**62**

Confusions and comments for DTW:

- The 3 and 8 confusion discussion follows even here, but the effect is even more pronounced, since now the accuracy of 3 is much much higher than eight. This maybe due to that **eight** has both **'i'** and '**e'** leading to more confusion, and also, eight would have a good amount of co-articulation between i and e. Hence this anomaly can only be solved by taking the **deltas** and **delta deltas** in consideration, since these will give due respect to the co-articulation effects in eight and prevent the confusions of 8 with 6 and 3. Doing DTW solves three's problem however, since the double vowel is 'ee' and hence less/no co-articulation, and hence the dynamic features are not required
- The five and nine confusion follows here as well, but again more profound in five, since in **n**i**n**e, the consonant **n** is nasal and it thus is also sonorous, and reduces the energy concentration in vowels, reducing the confusion with f**ive** which has very low energy consonants.
- As expected, we get the best accuracy out of all methods tried from DTW (91.8%).

The detailed code for DTW N fold validation is there in Appendix 4. I used DTW from a python library out of shelves, due to shortage of time. The log output is in "**./Output_Logs/DTW.txt",** which contains all the predictions as well.

5. Interactive testing with real time endpointing and detection

Using Jupyter notebooks, I tried out an interactive testing of the codes written.

For predicting the digit I am using the 8 centroids generated after clustering all the feature vectors generated by **all** speakers (since anyways I'm testing on a new sample) by k means.

Here, I realised the hardness of the problem when it comes to real data, since the accuracy went downhill to around 40-50 %, maybe due to

- Noisy Recording
- Not considering **deltas** and **delta deltas**.
- Increase the number of clusters (i.e. centroids) in the algorithm
- We are also not regularising the matching condition, which is possible by use of neural networks and a regularized loss function

You can see one demonstration of working of code, in the video of the implementation : Google Drive Link

Also, you can download the ipython notebook("**Live_Test.ipynb"**) and other codes from the github repository for the project.

Appendix 1a: s-t Energy Normalisation code

```python
def stenergy_endpointer(inp,Fs):
    Ts=1/Fs
    #------------------------------------------------------------------------------
    #Compute Avg magnitude (Short Time), with window size N/20 (0-9 spoken twice)
    W=1000
    pad_zeros=np.zeros(W)
    zp_inp_t=np.append(pad_zeros,inp)
    zp_inp=np.append(zp_inp_t,pad_zeros)
    st_avg=np.array([np.sum(abs(zp_inp[i-W:i+W])) for i in range(W,zp_inp.size-W)])
    print(st_avg.size)
    print(inp.size)
    #plt.plot(st_avg,'red')

    #Compute IZCT assuming 0.2s and 0.2 silence in end
    num_samp_in_sil=int(0.2*Fs)
    sil_inp_st=st_avg[0:num_samp_in_sil]
    sil_inp_end=st_avg[st_avg.size-num_samp_in_sil:st_avg.size]
    sil_inp=np.append(sil_inp_st,sil_inp_end)

    #Compute Peak Energy Imax and Silence Energy maximum
    Imx=np.amax(st_avg)
    Imn=np.amax(sil_inp)
    #Compute ITL and ITU
    I1=0.3*(Imx-Imn)+Imn
    I2=50*Imn
    print("ITL is "+str(ITL))
    plt.plot(st_avg)
    ITU=1.2*ITL

    #Search Fwd
    start_index=num_samp_in_sil+1
    end_index=inp.size-1
    zero_st,zero_fi=search(st_avg,start_index,end_index,ITU,ITL)

    return inp[zero_st:zero_fi],zero_st,zero_fi

def search(st_avg,start_index,end_index,ITU,ITL):
    N1=0
    N2=0
    for i in range(start_index,end_index):
        flag=0
        if(st_avg[i]>ITL):
            print("Here")
            N1=i
            break
    for i in range(end_index,start_index,-1):
        flag=0
        if(st_avg[i]>ITL):
            N2=i
            break
    return int(N1/2),int(N2/2)
```

Appendix 1b: Rabiner-Sambur method (with some slight modifications)

```
# Based on paper by Rabiner and Sambur
# Returns ndarray corresponding to voiced part in filename wav file
def endpointer(filename,Fs,glob_count=1):
    Ts=1/Fs
    read_wav = read(filename+'.wav')
    inp=np.array(read_wav[1],dtype='float64')
    plt.rcParams["figure.figsize"] = (18,10)
    #plt.title("Raw Input")
    #plt.plot(inp,'blue')

    #-----------------------------------------------------------------------------
    #Compute Avg magnitude (Short Time), with window size N/20 (0-9 spoken twice)

    #Window Size Changes
    W=40

    pad_zeros=np.zeros(W)
    zp_inp_t=np.append(pad_zeros,inp)
    zp_inp=np.append(zp_inp_t,pad_zeros)
    st_energy=[np.sum(abs(zp_inp[i-W:i+W])) for i in range(W,zp_inp.size-W)]
    #plt.plot(st_avg,'red')

    #Assuming first 100ms to be noise
    num_samp_in_sil=int(0.01*Fs)
    sil_inp=st_energy[0:num_samp_in_sil]

    #Compute Peak Energy Imax and Silence Energy Imin
    Imx=np.amax(st_energy)
    Imn=np.mean(sil_inp)
    #print(Imn)
    #Compute ITL and ITU
    I1=0.0008*(Imx-Imn)+Imn
    I2=Imn
    print(I1)
    print(I2)
    #ITL also needs to be fine tuned
    ITL=min(I1,I2)
    #ITL=12250
    print(ITL)
    ITU=10*ITL

    #Search Fwd
    i=0
    while(True):
        #print(i)
        if(i==0):
            start_index=num_samp_in_sil+1
        zero_st,zero_fi=search(st_energy,start_index,ITU,ITL)
        #print (zero_fi)
        if(zero_st==-1):
            start_index=zero_fi
            continue
        filename="/home/agrim/Processed_Data/"+str(int(i/2))+"/"+str(glob_count+(i%2))+".wav"
        print (filename+" Indices "+str(zero_st)+":"+str((zero_fi)))
        end_point_sig=inp[zero_st:zero_fi]

        write(filename, 8000, np.array(end_point_sig).astype(np.dtype('i2')))
        start_index=zero_fi
        i=i+1
        if(i==20):
            break
        #plt.plot(inp[zero_st:zero_fi])

endpointer('./Raw_Data/Digits_male_8Khz/zero_to_nine_Hitesh2',8000,15)

def search(st_avg,start_index,ITU,ITL):
    N1=0
    N2=0
    m=start_index
    #Fwd Search
    while(True):
        flag=0
        #print(st_avg[m]-ITL)
        if(st_avg[m] >= ITL):
            i=m
            contender=m
            while(True):
                if(st_avg[i]<ITL):
                    flag=1
                    break
                if(st_avg[i]>=ITU):
                    contender=i
                    flag=2
                    break
                else:
                    #print("Stuck here")
                    i=i+1

            if(flag==2):
                N1=contender
                break
        if(flag==1):
            m=i+1
        if(flag==0):
            m=m+1


    #Bkwd Search
    m=N1+1
    while(True):
        flag=0
        if(st_avg[m] <= ITU):
            i=m
            contender=m
            while(True):
                if(st_avg[i]>ITU):
                    flag=1
                    break
                if(st_avg[i]<=ITL):
                    contender=i
                    flag=2
                    break
                else:
                    i=i+1

            if(flag==2):
                N2=contender
                break

        if(flag==1):
            m=i+1
        if(flag==0):
            m=m+1

    # Do not consider noise (Utterance of less than 1600 samples => 0.2 sec)
    if((N2-N1)<1600):
        return -1,N2+1
    return N1,N2
```

Appendix 2: MFCC Feature Vector generation

```
#Get ‘num_filts’ # of mel filters between freq_down and freq_up
def get_mel_filters(N,num_filts,Fs,freq_down=0,freq_up=4000):
    up_melcoef=1125*np.log(1+freq_up/700)
    down_melcoef=1125*np.log(1+freq_down/700)
    filtbank_mel=np.linspace(down_melcoef,up_melcoef,num_filts+2)
    filtbank_hz=700*(np.exp(filtbank_mel/1125)-1)
    #print (filtbank_hz)
    filtbank_bins=np.floor((2*N-2)*filtbank_hz/Fs)+1
    #print(filtbank_hz)
    print (filtbank_bins)
    #Computing filters now
    filters=np.zeros(shape=(num_filts,N))
    freq=np.arange(N)*(Fs/(2*N-2))
    for i in range(1,num_filts+1):
        fp=int(filtbank_bins[i-1])
        sp=int(filtbank_bins[i])
        tp=int(filtbank_bins[i+1])

filters[i-1]=np.hstack((np.zeros(fp),np.linspace(0,1,sp-fp+1),np.linspace(1,0,tp-sp+1)[1:],np.zeros(N-tp-1)))
.ravel()
        #plt.plot(freq,filters[i-1])
    return filters

#Extract and save features for digit in a .npy file. Num_wav indicates the file number (wav file in section
generated in section 1) to be read
def feat_ext(digit,num_wav,Fs,use_inbuilt=0):

    # Parameters
    N=512
    num_filts=26
    mel_filters=get_mel_filters(N//2+1,num_filts,Fs)
    t_analysis=0.01
    hop_samp=int(Fs*t_analysis)
    num_samp=hop_samp
    plt.show()
    for samp_under_analysis in range(1,num_wav+1):

        filename="./Processed_Data/"+str(digit)+"/"+str(samp_under_analysis)+".wav"
        print(filename+ " Read")
        read_wav = read(filename)
        inp_wo_emph=np.array(read_wav[1],dtype='float64')
        #inp=np.array(read_wav[1],dtype='float64')
        if(use_inbuilt==1):

mfcc_feat=mfcc(inp_wo_emph,samplerate=8000,winlen=0.01,winstep=0.01,numcep=13,nfilt=26,nfft=512,lowfreq=0,hig
hfreq=None,preemph=0.97,ceplifter=22,appendEnergy=True)
```

```
else:
    a=0.97
    # Pre Emphasis
    # Doing Pre Emphasis at this stage since I'd forgotten to do it in end pointing stage
    inp = np.append(inp_wo_emph[0], inp_wo_emph[1:]-a*inp_wo_emph[:-1])

    #plt.rcParams["figure.figsize"] = (18,10)
    #plt.plot(inp,'blue')

    num_feat=int((inp.size-num_samp)/hop_samp)
    num_coef=13
    mfcc_feat=np.zeros(shape=(num_feat,num_coef))
    st_ind=0
    #print (inp.size)
    for i in range(num_feat):

        end_ind=st_ind+num_samp
        #print ("Frame number: "+str(i)+" st index "+ str(st_ind)+" end index "+ str(end_ind))
        frame=inp[st_ind:end_ind]
        #---------------------------------------------------------
        #Windowed DFT
        zero_arr=np.zeros(N-num_samp)
        zero_pd=np.append(frame,zero_arr)
        hamm_w=np.append(hamming(num_samp),np.zeros((N-num_samp)))
        s_n=zero_pd*hamm_w ## x[n]=s[n]w[n]
        S_k=fft(s_n)
        #P_k=np.square(np.abs(S_k)[0:257])/num_samp
        P_k=np.abs(S_k[0:257])

        #---------------------------------------------------------
        #Computing MFCC
        epsilon=1e-8
        energy_fbank=[20*np.log10(np.sum(P_k*mel_filters[i])+epsilon) for i in range(num_filts)]
        ifft_samp=ifft(energy_fbank)
        #---------------------------------------------------------
        # Delta Delta Computation
        orig_mfcc=ifft_samp[0:13]
        #if(i>=2 and i<(num_feat-2)):
        #    delta_mfcc[i]=(mfcc[i+1]-mfcc[i-1]+2*mfcc[i+2]-2*mfcc[i-2])/10
        #else:
        #    delta_mfcc=np.zeros(13)

        #if(i>=4 and i<(num_feat-4)):
        #    dealt_delta_mfcc=(mfcc[i+1]-mfcc[i-1]+2*mfcc[i+2]-2*mfcc[i-2])/10

        #mfcc[i]=np.append(orig_mfcc,delta_mfcc[0:13])
        mfcc_feat[i]=orig_mfcc
        #print (mfcc[i])
        st_ind=st_ind+hop_samp
        #print(mfcc_feat[num_feat-1])
        filename="./Extracted_Feats/"+str(digit)+"/"+str(samp_under_analysis)
        print(filename+" Saved")
        #np.save(filename,mfcc_feat)

feat_ext(0,64,8000)
feat_ext(1,64,8000)
feat_ext(2,64,8000)
feat_ext(3,64,8000)
feat_ext(4,64,8000)
feat_ext(5,64,8000)
feat_ext(6,64,8000)
feat_ext(7,64,8000)
feat_ext(8,64,8000)
feat_ext(9,64,8000)
```

Appendix 3a: Bag of Frames N fold validation

```
accuracy=0
accuracy_perd=np.zeros(10)
for speak in range(1,65):
    print("Speaker "+ str(int((speak-1)/4)+1)+" Uterrence: "+ str(int(speak-1)%4+1))
    for digit in range(10):
        filename="./Extracted_Feats/"+str(digit)+"/"+str(speak)+".npy"
        zr=np.load(filename)
        forbidden=[int(speak/4)*4+1,int(speak/4)*4+2,int(speak/4)*4+3,int(speak/4)*4+4]
        exc_samp=[i for i in range(1,65) if i not in forbidden]
        #print(exc_samp)
        pred=np.zeros(10)
        for d in range(10):
            dist=0
            count=1
            for i in exc_samp:
                filename="./Extracted_Feats/"+str(d)+"/"+str(i)+".npy"
                tarr=np.load(filename)
                # Find all closest distances to the frames
                for j in range(zr.shape[0]):
                    min_dist_to_a_frame=0
                    # Find Closest Matching Frame
                    for k in range(tarr.shape[0]):
                        obt_dist=np.sum(np.abs(zr[j]-tarr[k]))
                        if(k==0):
                            min_dist_to_a_frame=obt_dist
                        else:
                            if(obt_dist<min_dist_to_a_frame):
                                min_dist_to_a_frame=obt_dist
                    dist=dist+min_dist_to_a_frame
                    count=count+1
                # Dist now contains
            #Avg min dist
            pred[d]=(dist/count)
            #print(str(d)+":"+str(pred[d]))
        predic=np.argmin(pred)
        if(predic==digit):
            accuracy=accuracy+1
            accuracy_perd[digit]=accuracy_perd[digit]+1
        print("Predicted: "+ str(predic)+" for expected digit: "+str(digit))
print("Net Accuracy")
print(accuracy/640)
print("Accuracy Per Digit")
print(accuracy_perd/64)
```

Appendix 3b) i: VQ codebook generation

```
from sklearn.cluster import KMeans
for speaker in range(1,17):
    print("VQ Codebooks for Speaker "+ str(speaker))
    for digit in range(10):
        forbidden=[(speaker-1)*4+1,(speaker-1)*4+2,(speaker-1)*4+3,(speaker-1)*4+4]
        print(forbidden)
        exc_samp=[i for i in range(1,65) if i not in forbidden]
        list_of_frames=[]
        # Concat all feature vectors for the particular digit
        for i in exc_samp:
            filename="./Extracted_Feats/"+str(digit)+"/"+str(i)+".npy"
            tarr=np.load(filename)
            for k in range(tarr.shape[0]):
                list_of_frames.append(tarr[k])
        arr_of_frames=np.array(list_of_frames)
        kmeans4 = KMeans(n_clusters=4).fit(arr_of_frames)
        kmeans8 = KMeans(n_clusters=8).fit(arr_of_frames)
        print("Kmeans 4")
        print(kmeans4.cluster_centers_)
        print("Kmeans 8")
        print(kmeans8.cluster_centers_)
        filename="./VQ_codebooks/Speaker "+str(speaker)+"/"+str(digit)+"/k4"
        np.save(filename,kmeans4.cluster_centers_)
        filename="./VQ_codebooks/Speaker "+str(speaker)+"/"+str(digit)+"/k8"
        np.save(filename,kmeans8.cluster_centers_)
```

Appendix 3b) ii: VQ N fold validation

```
accuracy4=0
accuracy8=0
accuracy_pred4=np.zeros(10)
accuracy_pred8=np.zeros(10)

for speak in range(1,65):
    print("Speaker "+ str(int((speak-1)/4)+1)+" Uterrence: "+ str(int(speak-1)%4+1))
    for digit in range(10):
        filename="./Extracted_Feats/"+str(digit)+"/"+str(speak)+".npy"
        zr=np.load(filename)
        #print(exc_samp)
        pred4=np.zeros(10)
        pred8=np.zeros(10)
        for d in range(10):
            dist4=0
            dist8=0
            count=1
            filename="./VQ_codebooks/Speaker "+str(int((speak-1)/4)+1)+"/"+str(d)+"/k4.npy"
            k4=np.load(filename)
            filename="./VQ_codebooks/Speaker "+str(int((speak-1)/4)+1)+"/"+str(d)+"/k8.npy"
            k8=np.load(filename)
            for j in range(zr.shape[0]):
                    # Find Closest Matching Frame
                    # Instead of adding distancesto all centroids, add distance to one centroid min
                    begin=True
                    min_dist_to_a_centroid4=0
                    for centroid in k4:
                        obt_dist=np.sum(np.abs(zr[j]-centroid))
                        if(begin):
                            min_dist_to_a_centroid4=obt_dist
                            begin=False
                        else:
                            if(obt_dist<min_dist_to_a_centroid4):
                                min_dist_to_a_centroid4=obt_dist
                    begin=True
                    min_dist_to_a_centroid8=0
                    for centroid in k8:
                        obt_dist=np.sum(np.abs(zr[j]-centroid))
                        if(begin):
                            min_dist_to_a_centroid8=obt_dist
                            begin=False
                        else:
                            if(obt_dist<min_dist_to_a_centroid8):
                                min_dist_to_a_centroid8=obt_dist
                    dist4=dist4+min_dist_to_a_centroid4
                    dist8=dist8+min_dist_to_a_centroid8
                    count=count+1
                # Dist now contains
            #Avg min dist
            pred4[d]=(dist4/count)
            pred8[d]=(dist8/count)
            #print(str(d)+":"+str(pred[d]))
        predic4=np.argmin(pred4)
        if(predic4==digit):
            accuracy4=accuracy4+1
            accuracy_pred4[digit]=accuracy_pred4[digit]+1
        predic8=np.argmin(pred8)
        if(predic8==digit):
            accuracy8=accuracy8+1
            accuracy_pred8[digit]=accuracy_pred8[digit]+1

        print("Predicted: "+ str(predic4)+ " for expected digit: "+str(digit) + " k4")
        print("Predicted: "+str(predic8)+" for expected digit: "+str(digit) + " k8")

print("Net Accuracy for k=4")
print(accuracy4/640)
print("Accuracy Per Digit for k=4")
print(accuracy_pred4/64)
print("Net Accuracy for k=8")
print(accuracy8/640)
print("Accuracy Per Digit for k=8")
print(accuracy_pred8/64)
```

Appendix 4 DTW N fold validation

```
from dtw import dtw
accuracy=0
accuracy_perd=np.zeros(10)
for speak in range(1,65):
    print("Speaker "+ str(int((speak-1)/4)+1)+" Uterrence: "+ str(int(speak-1)%4+1))
    for digit in range(10):
        filename="./Extracted_Feats/"+str(digit)+"/"+str(speak)+".npy"
        zr=np.load(filename)
        forbidden=[int(speak/4)*4+1,int(speak/4)*4+2,int(speak/4)*4+3,int(speak/4)*4+4]
        exc_samp=[i for i in range(1,65) if i not in forbidden]
        #print(exc_samp)
        pred=np.zeros(10)
        for d in range(10):
            dist=0
            count=1
            for i in exc_samp:
                filename="./Extracted_Feats/"+str(d)+"/"+str(i)+".npy"
                tarr=np.load(filename)
                dtw_dist = dtw(zr, tarr, dist=lambda x, y: norm(x - y, ord=1))
                dist=dist+dtw_dist[0]
                count=count+1

            #Avg min dist
            pred[d]=(dist/count)
            #print(str(d)+":"+str(pred[d]))
        predic=np.argmin(pred)
        if(predic==digit):
            accuracy=accuracy+1
            accuracy_perd[digit]=accuracy_perd[digit]+1
        print("Predicted: "+ str(predic)+" for expected digit: "+str(digit))
print("Net Accuracy")
print(accuracy/640)
print("Accuracy Per Digit")
print(accuracy_perd/64)
```