Digital Signal Processing
Dialect Classification

Abstract:

Dialects are nothing but variation of pronouncing a particular language spoken by a group of people within a community. Dialects commonly represent the changing speaking patterns observed in the group of native speakers belonging to some particular region. These dialects are responsible for failure of Automatic Sound recognition systems. So Dialect Classification is an interesting Signal-Processing-Pattern-Recognition problem. The problem statement of the project is to classify the dialects of British English. There are many dialects in British English language and for the project purpose we consider nine regions (dialects).

Introduction:

There is a major misconception about dialects and accents being interchangeable words. Dialects are variations in the wordings, the grammar and the pronunciations in the same language whereas accents are just variations in pronunciations. Different dialects and accents arise due to a lot of factors like age, gender, culture, and locality.

Dataset:

The dataset contains 9 dialects from various regions of UK. Recordings of male and female speakers were made in the following dialects:

- London
- Cambridge
- Cardiff
- Liverpool
- Bradford
- Leeds
- Newcastle
- Belfast in Northern Ireland
- Dublin in the Republic of Ireland.

Three of our speaker groups are from ethnic minorities: we have recorded bilingual Punjabi-English speakers, bilingual Welsh-English speakers and speakers of Caribbean descent.

The dataset has 9 folders with 67 audio samples each. The audio samples are narrations of a Cinderella passage which can be found here.

Motivation:

The variations in wordings for different dialects has caused many ASR (Automatic Speech Recognition) applications to often fail in recognizing user's speech. This has encouraged people to find ways to distinguish dialects. Different grammar and word selection in dialects force us to understand how each dialect use phonemes. Phonemes are what differentiate words from one other in a particular dialect. By finding patterns in different phonemes which make up a word in different dialects, we can possibly try classifying them.

Approach:

Frame level breakdown:

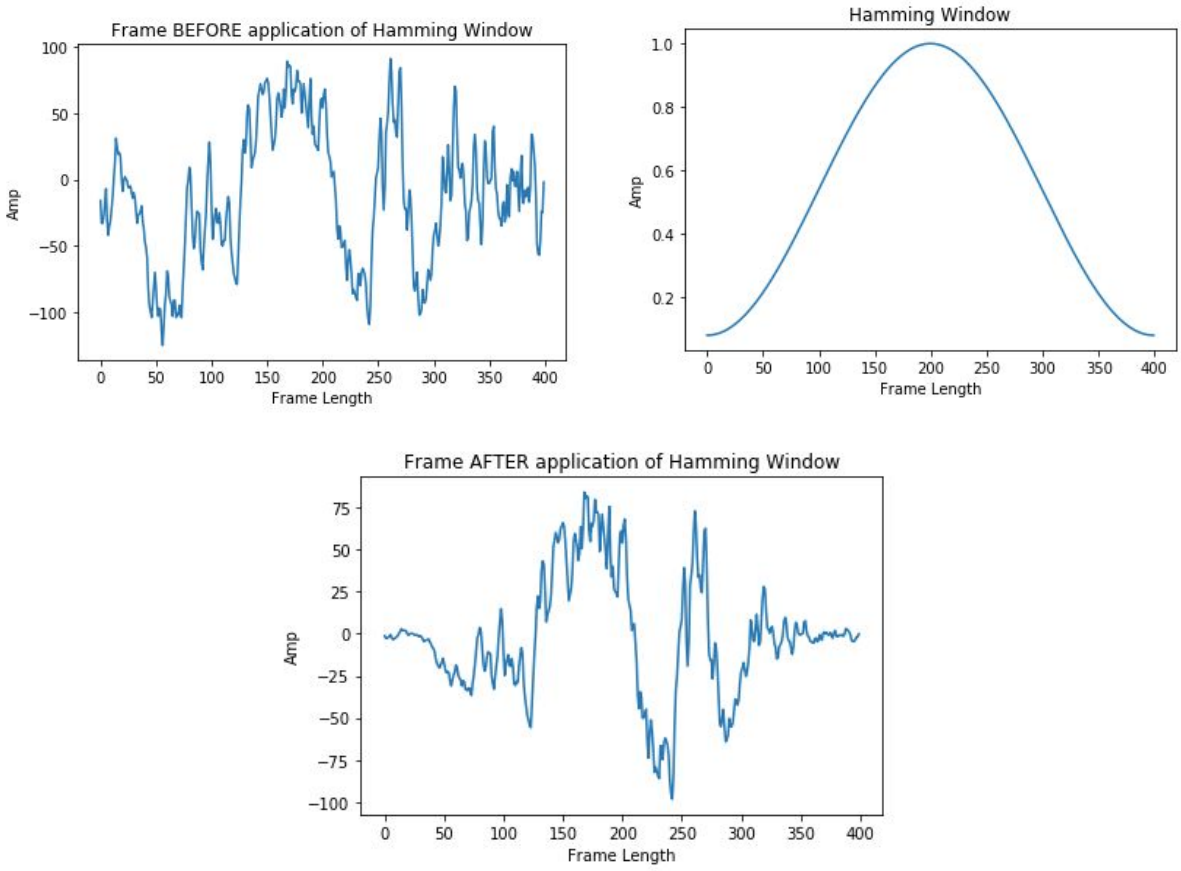
Phonemes are part of words hence we need to break down our audio samples into small parts for analysis. For this, we break down the audio signal into small parts which approximately contain enough spectral content on how different phonemes make up each word. Each small part would be one frame. Statistically, the length of one frame is usually around 25ms. Anything more or less than this makes the spectral content of the frame ambiguous. While breaking down our audio sample into frames, we make sure there is some overlap present between them to find a correlation between adjacent phonemes. This overlap is called stride (just like the stride in convolutional neural networks). Again, statistically, this stride is taken to be around 15ms. Anything less won't be useful and anything more would increase overfitting. This is done because, in dialects, the temporal positioning of phonemes in speech also plays a huge role. A phoneme may change to some other phoneme when placed adjacent to some other phoneme.

```
framelength, framestride, nfft, num_fbanks, n_cep_coeff = 0.025, 0.015, 512, 40, 12
frmlen, frmstrd, signallen = int(round(rate*framelength)),
int(round(rate*framestride)), len(data)
paddinglen = frmstrd - ((signallen - frmlen)%frmstrd)
paddedsig = np.concatenate((data, np.zeros(paddinglen)), axis = 0)
paddedsiglen = len(paddedsig)
nframes = int(np.floor((paddedsiglen - frmlen)/frmstrd) + 1)
indices = np.tile(np.arange(0, frmlen), (nframes, 1)) + np.tile((np.arange(0,
nframes*frmstrd, frmstrd)), (frmlen, 1)).T
frames = paddedsig[indices]
```

Hamming window:

Breaking our audio samples into frames abruptly in the time domain will give rise to abrupt changes at edges of the spectral analysis. To avoid this, we use a Hamming window to the frames to smoothen out the frame endings.

```
frames *= np.hamming(frmlen)`
```



Power spectrum calculation:

Now that our frames are ready for spectral analysis, we find the power spectrum (Periodogram) of each frame to analyze frequency content in the phonemes.

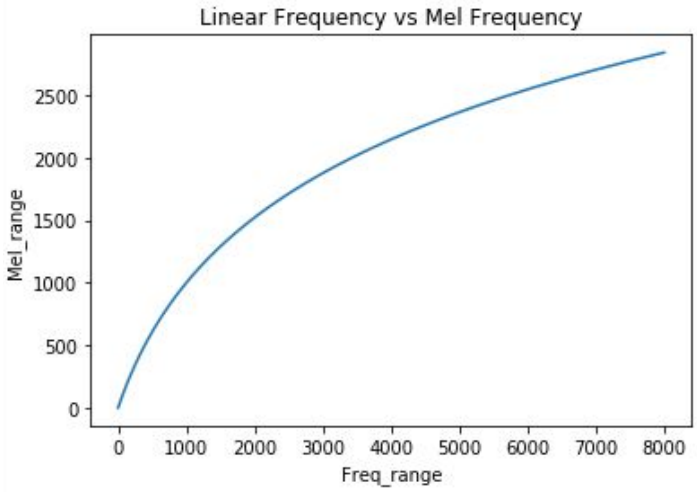
Human cochlea vibrates at different spots depending on the frequency of incoming sounds and each part of cochlea triggers different nerve fibers. Each of the nerve fibers

have the ability to inform the brain about certain different frequencies that are present in the received signal. Power spectrum tells us the frequencies present in the frame.

```
frame_fft = np.absolute(np.fft.rfft(frames, n = nfft, axis = 1))
frame_periodogram = np.square(frame_fft)/nfft
```

Instead of analyzing the bare spectrum, we scale the spectrum to match our perceptual frequency ranges. Humans can hear frequencies between 20 to 20KHz, but our hearing is more discriminative at lower frequencies than at higher frequencies. Cochlea cannot discern the difference between two closely spaced frequencies and this effect becomes more phenomenon as frequency increases. Thus, we analyze higher frequencies with a wider range and lower frequencies with a smaller range to catch phoneme based features. To convert our power spectrum range from the generic range to a more perceptual range, we use the Mel scale. Mel scale normalizes the frequency scale to match our perceptual frequency distinguishing capabilities. This can be seen in the following table.

Hz	20	160	394	670	1000	1420	1900	2450	3120	4000	5100	6600	9000	14000
mel	0	250	500	750	1000	1250	1500	1750	2000	2250	2500	2750	3000	3250



We see that an increment of around 240Hz, from 160Hz to 394Hz is equivalent to 250 Mels and the same jump of 250 Mels at higher frequencies is a jump of 5000Hz from 9000Hz to 14000Hz.

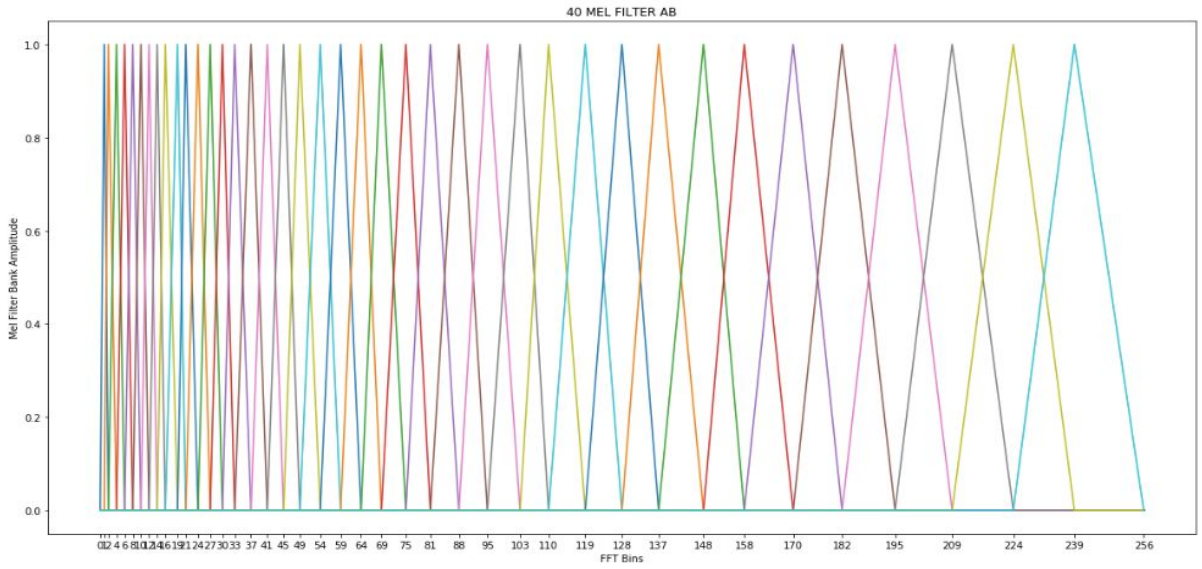
We use filter banks to create a new spectral picture of the frame.

Mel scale conversion using filterbanks:

Filter banks are triangular filters which when multiplied with the original frame spectrum, give us a new spectral picture. We use the Mel scale to appropriately select the widths of the filter banks. The filter bank triangle starts from Mel scale value m, peaks at m+1 and then comes down to zero at m+2. Next filter bank starts at m+1 instead of m+2, this is to avoid blind-spots in our new spectral analysis. You can see that in this way, at higher frequencies, the filter bank width is big, this is because our hearing and speech generation works poorly at higher frequencies, hence our filter will take a bigger range to accommodate changes in the spectrum.

```
low_mel_lim = 0
up_mel_lim = rate/2
#for 40 filter banks, we need 42 mel points
mel_top = freq_to_mel(up_mel_lim)
mel_range = np.linspace(0, mel_top, num_fbanks+2)
freq_range = mel_to_freq(mel_range)
bins = np.floor((nfft + 1) * freq_range/rate)
fbank = np.zeros((num_fbanks, int(np.floor(nfft / 2 + 1))))
for m in range(1, num_fbanks + 1):
    lower = int(bins[m - 1]) # lower
    peak = int(bins[m]) # peak
    upper = int(bins[m + 1]) # upper
    for k in range(lower, peak):
        fbank[m - 1, k] = (k - bins[m - 1]) / (bins[m] - bins[m - 1])
    for k in range(peak, upper):
        fbank[m - 1, k] = (bins[m + 1] - k) / (bins[m + 1] - bins[m])
filter_banks = np.dot(frame_periodogram, fbank.T)
filter_banks = 20*np.log10(filter_banks)
```

We then multiply our filter bank with the spectrum of each frame to find patterns in different frequency ranges. This is done with all the frames generated from the audio sample.



DCT application:

Humans don't hear loudness on linear scale, thus we convert filtered frame outputs into decibels and apply a de-correlation transform like **DCT** (discrete cosine transform) to remove correlations between frames. This is done because machine learning algorithms sometimes fail when there is a heavy correlation. The DCT coefficients are almost statistically independent. DCT removes higher frequency components and this is important because, in most speech data, information mostly resides in the lower frequency parts than the higher frequency parts. It is also the shape of the spectrum which is more important than the actual values.

The coefficients after applying DCT are called **log MFSC** (Mel frequency spectral coefficients). We take the first 12 log MFSC coefficients because the high-frequency data doesn't help us much in dialect classification as most of the human speech spectrum is at lower frequencies.

```
n_cep_coeff = 12
mfcccoeff = dct(filter_banks, type = 2, axis = 1, norm = 'ortho')[:,
0:(n_cep_coeff)]
```

Removing white noise:

On playing the audio files, we observe a significant amount of white noise which is possibly from the recording instruments. We can assume the white noise to be a linear

impulse response h(t) which is convolving with the input (frame) and remove this white noise by subtracting the mean of the filter banks from the filter banks.

Delta and Delta-Delta coefficients:

Till now we've worked on individual phoneme spectral content but dialects also have different velocities and acceleration of transition between phonemes. We noticed that the speech is faster in IDR3 compared to speech in IDR2. We can create delta coefficients (velocity) and delta-delta coefficients (acceleration) to learn these features.

```
N = 2
def ctpn(n_cep_coeff, coeff_type, t, n):
    if((t+n) > n_cep_coeff-1):
        return coeff_type[:,n_cep_coeff-1]
    elif(0 <= (t+n) <= n_cep_coeff-1):
        return coeff_type[:, t+n]

def ctmn(n_cep_coeff, coeff_type, t, n):
    if((t-n) < 0):
        return coeff_type[:,0]
    elif(0 <= (t-n) <= n_cep_coeff-1):
        return coeff_type[:, t-n]

def deltacoeff(t, coeff_type):
    dt = 0
    for n in range(1,N):
        dt+= n*(ctpn(n_cep_coeff, coeff_type, t, n) - ctmn(n_cep_coeff, coeff_type,
t, n))/2*np.square(n)
    return dt

def deltacoeff_gen(coeff_type, n_cep_coeff):
    deltacoef = np.zeros(coeff_type.shape)
    for t in range(0, n_cep_coeff):
        dt = deltacoeff(t, coeff_type)
        deltacoef[:, t] = dt
    return deltacoef

def deltadeltacoeff_gen(deltacoef, n_cep_coeff):
    deltadeltacoef = np.zeros(deltacoef.shape)
    for t in range(0, n_cep_coeff):
        ddt = deltacoeff(t)
        deltadeltacoef[:, t] = ddt
    return deltadeltacoef
```

Training:

We write down all the coefficient data into a pandas DataFrame object. This contains 67*9 rows and a lot of columns.

In order to preprocess our raw log MFSC, delta and delta delta coefficients, we take the mean of each coefficient in each dialect, which gives us 12 average coefficient values for each dialect. This is done because the spectral content of each frame isn't very important for generalization. It would be helpful to understand what kind of spectral content is present in the phonemes of each dialect. After finding the means, our DataFrame would now be of the size 67x9 rows and 36 columns. In order to get more insight from out coefficients, we also find the min-value, max value, standard deviation, skewness and median of each of our coefficients. This creates a DataFrame of 67x9 rows and 216 + 1(labels) columns.

We then split the dataset into 80% training data and 20% testing data using sklearn's train_test_split.

We trained an SVC classifier and found the hyper parameters using grid search. The accuracy was around 85-94%. There is a huge variation in accuracy when we retrain the model. This is because our dataset is relatively very small and the random shuffle and split of the dataset using sklearn's train_test_split causes large variations in our model accuracy.