LnT Infotech Project Thesis
Sentiment Analysis

Chetan Vashisht

11^{th}May - 11^{th}July 2015

Contents

List of Figures

Chapter 1

Objective and Declaration

The purpose of this project is to automate sentiment analysis of conversations and phone calls. The project must detect the sentiment of the conversation based on the audio file input.
The project was done in collaboration with L&T Infotech. The project is supervised by Avadhut Sawant (Senior Architect) and performed in May 2015 to July 2015.

Chapter 2

Tools/Software and Licenses

All of the software was built in Python using some packages which are all open source and available for free download. The packages required are:
Numpy: A library for handling arrays and numerical methods.
Scipy: A scientific computing library for input, output, feature extraction, manipulation and optimization.
Pandas, csv: Working with csv files.
Sklearn: A machine learning library, for feature extraction and classification algorithms.
Matplotlib: Plotting library for python.
All of the above packages are available under the winPython package.
Another tool used was audiacity to edit the sound files. It is a free and open source software.
All codes and software are under GNU's open source license agreement.
The dataset is under the license of The University of Milano Bicocca agreement.

Chapter 3

Architecture

The project is divided into two parts, the first being identification of the sentiment from the audio signal input. Features are extracted from the audio input and a conclusion is drawn about its sentiment. The second part of the project involves splitting a conversation into the parts spoken by the two speakers and running the trained machine learning algorithm on split conversation.
Briefly the project is divided into two parts. First the machine learning algorithms are trained by the Italian speech database. Then the separated speaker's inputs are fed to the algorithm and it classifies the emotions of the separated tracks.

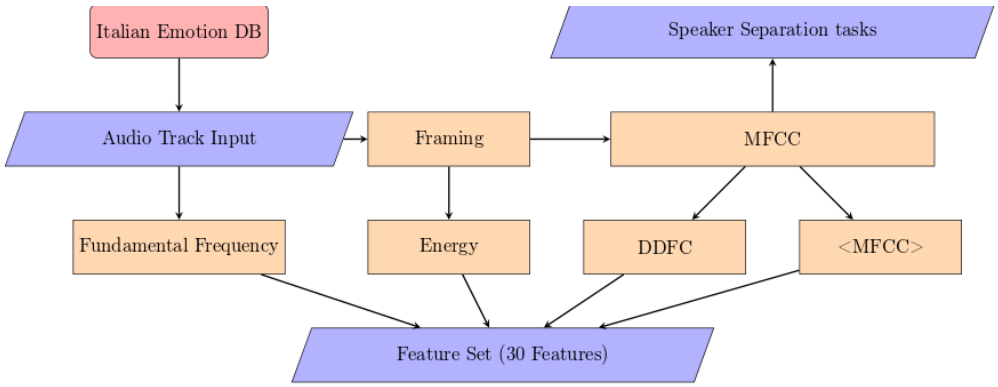

Figure 3.1: Flow chart of the project

Chapter 4

Phase I

Phase I was about collecting the dataset, feature extraction of audio files and classification to find the emotions.

4.1 Dataset collection

First a dataset is obtained from extensive research. Below are a set of useful links to datasets on the web. The dataset was obtained from University of Milano Bicocca website. (`www.mind.disco.`

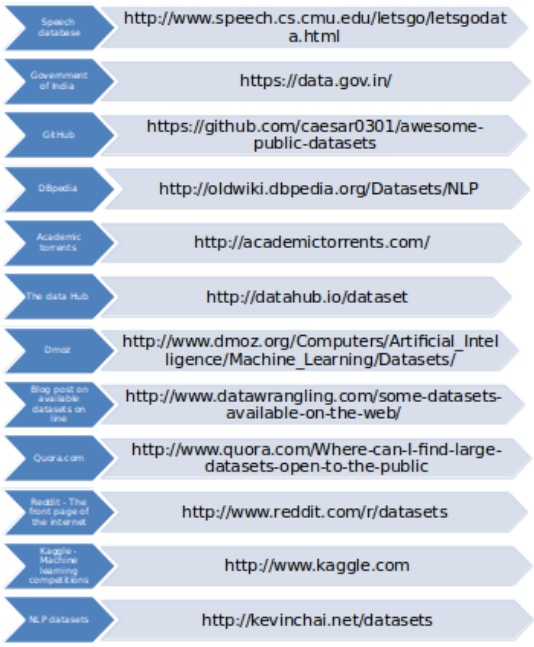

Figure 4.1: Datsets Researched

`unimib.it/Home_Page.asp?lang=en)` The dataset contains 400 recorded audio tracks in wav format recorded at 44.1 kHz. It contains 80 audio tracks of each of the five emotions Sadness, Anger, Happiness, Fear, Neutral. The free dataset obtained from the web, it required citing if a paper is being published.

4.2 Feature Extraction

The features extracted for the process of sentiment analysis are, MFCCs, DDACs, fundamental frequency and energy values.

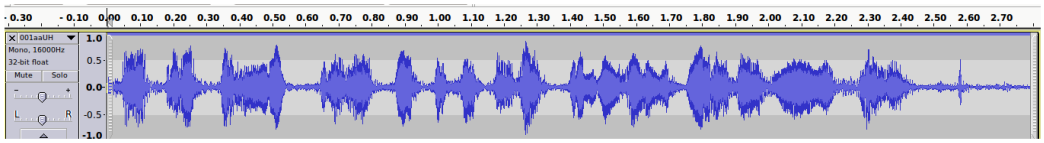

Figure 4.2: Typical waveform shown in Audiacity

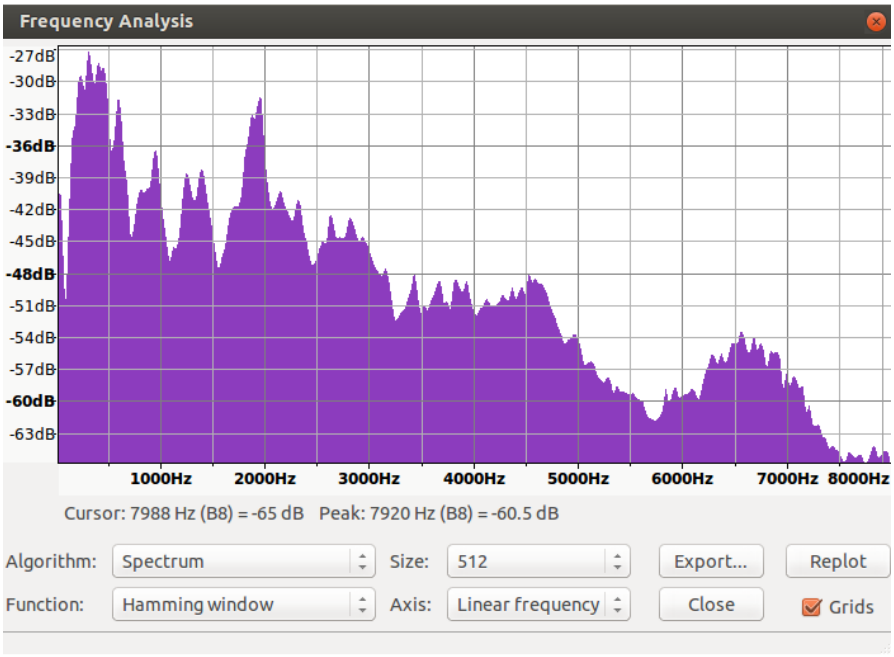

Figure 4.3: Frequency Spectrum of the waveform

4.2.1 MFCCs

MFCC's help understand the amount of energy concentrated in each frequency band. MFCC extraction follows three major steps.

The Mel Scale

The human ears are more sensitive to lower frequencies than to higher frequencies. The sensitivity is almost linear up to 1 kHz and increases logarithmically after that. It almost requires eight times more energy to listen a sound which has doubled in loudness.

$$M(f) = 1125ln(1 + f/700)$$

Framing

Since the signals are continuously changing, a short time scale is chosen so the signal seems stationary. Also we must have enough samples to get a reliable spectral estimate. If the sampling rate is 16 kHz, and the chosen time stamp is 25ms, then the sampling length is $0.025 \times 16000 = 400$. In order to ensure frame overlapping, at least 50% overlap must be present. The chosen value here is 160. The frame overlap for different window functions is different. A simple Hamming window ($h(t)$) is chosen. Now the window is multiplied to the signal to smoothen it. (It helps in getting a better Fourier transform) ($s(t) \times h(t)$) Next a Discrete Fourier Transform of the windowed function using scipy's built in fft function is taken. As the fft returns all frequencies from $-f$ to $+f$, only half the coefficients from 0 to $+f$ are taken. i.e. we take the 512 point fft and return only 256 coefficients. The corresponding ffted windows are stored in array one by one. They are used to energy calculations also. Net average

of all the fft windows is taken to provide an approximate representation during the patch.(Source file: MFCC.py)

$$S_i = \sum_{n=1}^{N} s_i(n)h(n)e^{-j2\pi kn/N}$$

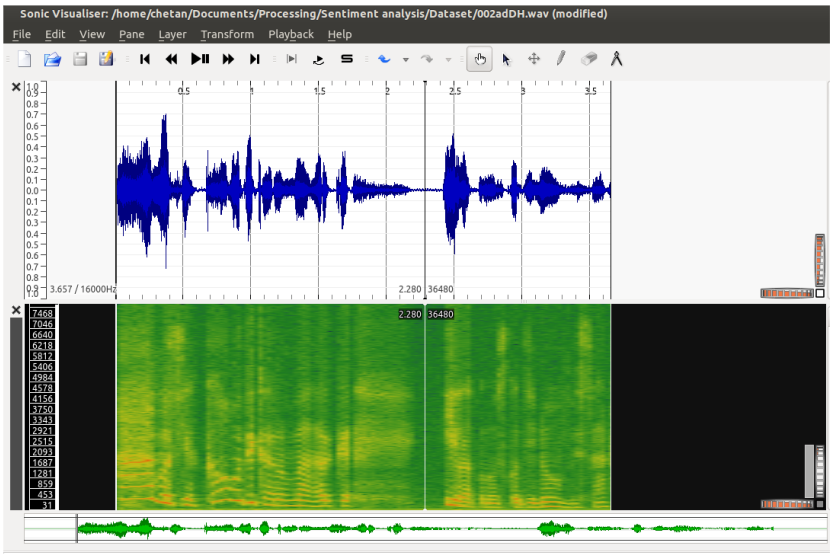

Figure 4.4: Framing or Short time Fourier transform on Sonic Visualiser

```
def frame_fft(signal, K = 512, N = 400, size1 = 256):
    h = []
    s1 = []
    for n in range(N):
        h = 0.54 - 0.46*np.cos(2*np.pi*n/(N-1))
        signal[n] = signal[n]*h

    # Taking the fft of the framed signal
    S = fft(signal, K)

    # fft give transform from -f to +f. So we return only [0,f].
    return np.array(S[:size1])
```

Bandpass Filter Generation

Since the human ear is more sensitive to lower frequencies than higher frequencies, a logarithm of the scale is taken to get the Mel scale. The frequencies are uniformly spaced in the Mel scale. Hence 25 uniformly spaced points (on the Mel scale) are found and mapped their values on the normal frequency scale.
25 unequally spaced triangular bandapss filters are obtained and map their frequency values to those on a 256 point scale. Hence a sparse 2D matrix of filter coefficients is obtained. This is then stored in a csv file and can be read conveniently later.(This is to prevent recalculation)
(Source file: bandpass_filters.py)

```
def generate_bandpass_filters(f_min = 50, f_max = 16000, N = 26, size = 256):
    f_min1 = freq_to_mel(f_min)
    f_max1 = freq_to_mel(f_max)
    f = []
    f = np.linspace(f_min1, f_max1, N)
```

```
    f = mel_to_freq(f)
    # f now contains the converted mel frequencies [300, 390,... 16000]
    # f contains the bins of importance [9, 16, ... 255]
    f = f_bin(f)

    g = np.zeros((N - 2, size))
    for i in range(N-2):
        g[i] = triangle_filter(f[i], f[i+1], f[i+2], size)
    g = np.array(g)

    # Store the numpy array into a csv file for quick access.
    # There is no need to compute the bandpass filters for each song.
    np.savetxt("Bandpass_filters.csv", g, delimiter = ',')
```

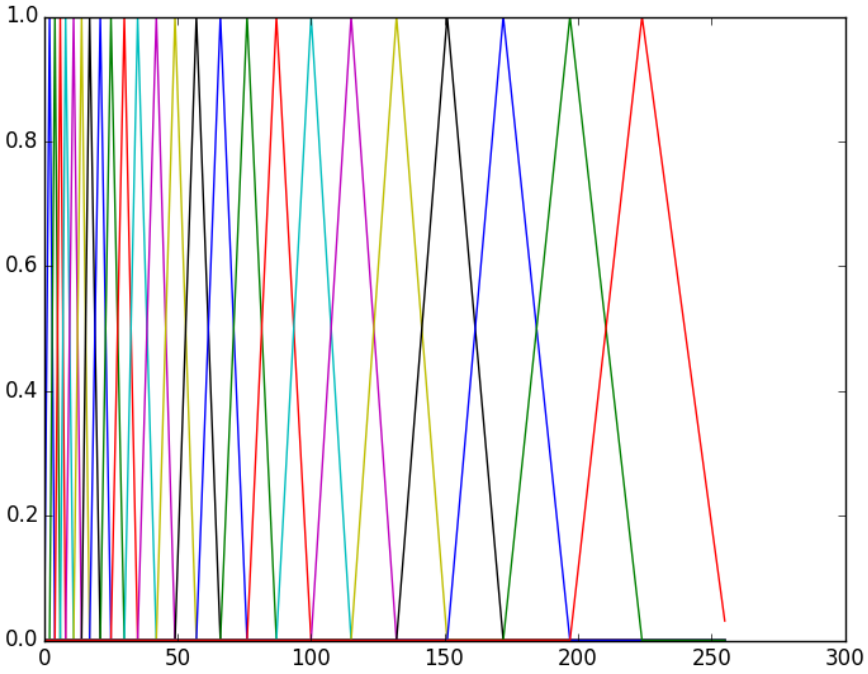

Figure 4.5: Bandpass Filters

Energy of the bands

Now the above two segments are combined to obtain the energy from each band. Multiplying the filter with the average value obtained above, a value is set for the energy of each frequency bin. Then a logarithm of the energy values is taken.
Then a (DCT) discrete cosine transforms to uncorrelate the energy bands. The DCT is obtained from the scipy package. It returns a set of real coefficients to obtain the 25 point MFCC. Since the last few are of minimal importance only the first 13 coefficients are considered. A time average of the MFCCs is used as a feature for each audio track.(Source file: MFCC.py)

```
def mfcc(signal_fft, plot = 0, no_of_coeff = 13):
    # Mel Cepstal Frequency Coefficents
    # frame -> filters -> log -> DCT -> return first 13 coeff
    filters = np.genfromtxt("Bandpass_filters.csv", delimiter = ',')
    energy = []
    signal_fft = np.multiply(signal_fft, signal_fft)
    for i in range(len(filters)):
        energy.append(sum(abs(signal_fft)*filters[i]))
```

```
    if (np.sum(energy) != 0):
        energy = np.log10(energy)
        energy = dct(energy)
    if(plot == 1):
        plt.plot(energy[:no_of_coeff])
        plt.show()
    return energy[:no_of_coeff]
```

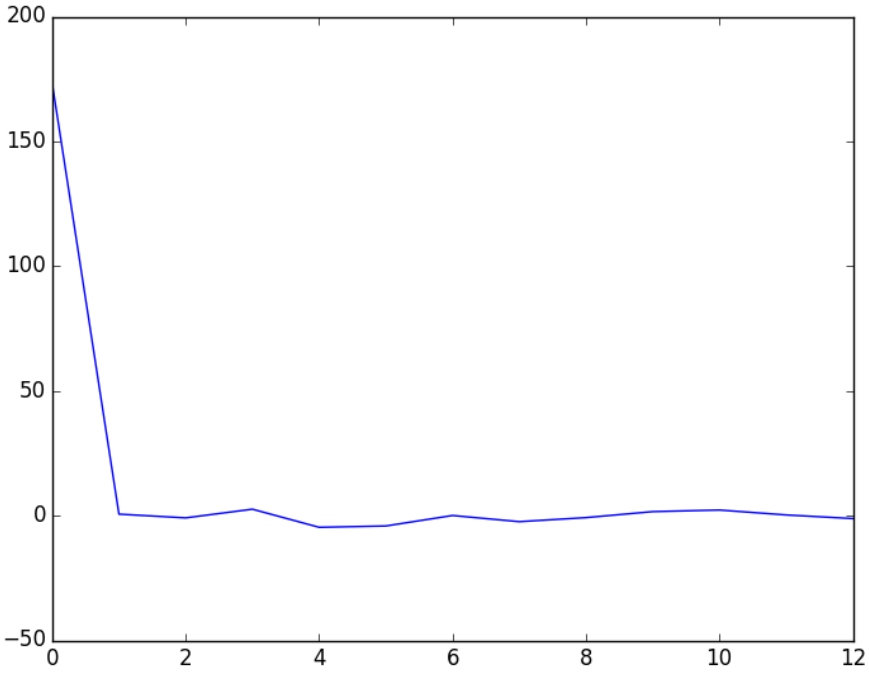

Figure 4.6: Mel Cepstal Frequency Coefficents

4.2.2 Delta Delta Acceleration Coefficents

There are used to measure the time delay features of the MFCCs. They are obtained by subtracting the MFCC values of consecutive frames. A lot of the emotion data is hidden in the rate of change of energy. This gives 12 more coefficients.(Source: MFCC.py)

$$d_t = \frac{\sum_{\theta=1}^{\theta_{max}} \theta(c_{t+\theta} - c_{t-\theta})}{2\sum_{\theta=0}^{\theta_{max}} \theta^2}$$

```
def ddfc(MFCC, no_of_coeff = 13):
    # Delta Delta Frequrncy Coefficients
    DDFC = np.zeros(no_of_coeff)
    for i, mfcc in enumerate(MFCC):
        if (i==0 or i==1 or i == len(MFCC)-2 or i == len(MFCC)-1):
            continue
        else:
            DFCC = np.vstack([DDFC,(2*(MFCC[i+2] - MFCC[i-2]) +
            (MFCC[i+1] - MFCC[i-1]))/10])

    return np.mean(DFCC, axis = 0)
```

4.2.3 Fundamental Frequency

There are a few methods to determine the pitch and fundamental frequency of a voice signal. 3 of the methods are listed below and all three are used as features.(Source: PitchDetection.py)

Zero Crossing Method

Count the total number of zero crossings and divide by the period. It?s fast and accurate for long and periodic signals. But it fails if there is a low signal which is corrupted by noise.

```
def zeroCrossing(signal, fs):
    # Find all indices right before a rising-edge zero crossing
    indices = find((signal[1:] >= 0) & (signal[:-1] < 0))
    return fs/mean(diff(indices))
```

Simple FFT

Find the fft and determine the peak. This fails if the harmonics are also very prevalent. It is improved by parabolic interpolation techniques. It is more accurate than the zero crossing method.

```
def freq_from_fft(signal, fs):
    # Compute Fourier transform of windowed signal
    windowed = signal * blackmanharris(len(signal))
    f = rfft(windowed)
    i = argmax(abs(f))

    # Convert to equivalent frequency
    return fs*i/len(windowed)
```

Autocorrelation

Find the auto correlation function and find its peak. It gives a good idea of the pitch perceived by the human ear. Implementation is difficult and costly in terms of computation.

```
def autocorrelation(sig, fs):
    # Reject the second half to remove the lag coefficents.
    corr = fftconvolve(sig, sig[::-1], mode = 'full')
    corr = corr[len(corr)/2:]

    # Find occurance of a minimum. Need the array index.
    # It is unrelable due to noise. Hence we take the nect value.
    d = diff(corr)
    start = find(d > 0)[0]
    peak = argmax(corr[start:]) + start
    return fs/peak
```

4.2.4 Energy Estimation

The energy of the time domain signal is obtained by summing the signal squared over the window. The frame energies is obtained for the whole track and compared to find, the minimum, maximum and average values. This adds three more features.(Source file: MFCC.py)

```
def frame_power(S, N):
    # Calculating the absolute power of the frame
    return 1/N * (np.sum(np.abs(S))**2)
feat = []
feat.append(max(frame_energy))
feat.append(min(frame_energy))
feat.append(sum(frame_energy)/len(frame_energy))
```

4.3 Models and Classification

Several possible machine learning algorithms are out there. There are 30 features extracted for each track. The obtained features are stored in a csv file for easy access. Each of the machine learning algorithms are tried on the feature set and Forest classifier proves to be the most successful one (providing an accuracy of around 55%).

4.3.1 Boosting

Random forests are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.

4.3.2 Neural Networks

Artificial Neural Networks (ANNs) are a family of statistical learning models inspired by biological neural networks (the central nervous systems of animals, in particular the brain) and are used to estimate or approximate functions that can depend on a large number of inputs and are generally unknown. Artificial neural networks are generally presented as systems of interconnected "neurons" which send messages to each other. The connections have numeric weights that can be tuned based on experience, making neural nets adaptive to inputs and capable of learning.

4.3.3 Support Vector Machines

Support Vector Machines (SVMs) are supervised learning models with associated learning algorithms that analyze data and recognize patterns, used for classification and regression analysis. Given a set of training examples, each marked for belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other, making it a non-probabilistic binary linear classifier. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible.

4.3.4 K Nearest Neighbours

K Nearest Neighbors algorithm (KNN) is a non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space. KNN is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until classification. The k-NN algorithm is among the simplest of all machine learning algorithms.

4.3.5 Naive Bayes classifier

Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features.

Chapter 5

Phase II

The idea here is to split the audio conversation file into the respective speakers, then apply the trained engine on these to get an output sentiment. For this a few approaches were tried with no success. The general problem is that the classifiers for this part are largely unsupervised and heavily misclassify the frames.
Changes were made in terms of pre emphasis, machane learning algorithms used and software for carrying out this simulation.

5.1 Unsupervised Learning

5.1.1 Gaussian Mixture Model

Just extract MFCCs of the conversation clip and feed it to a Gaussian Mixture Model. This approach failed as the classifier kept alternating between the possible outputs. Since the classifier was heavily misclassifying, another approach was adopted, taking into account that this much information was not enough.(Source file: something.py)

```
rate, signal1 = read("conv.wav")
signal = signal1
signal_fft = zeros(256)
i = 0
while(i + 400 < len(signal)):
    signal_fft = vstack([signal_fft, frame_fft(signal[i : i+400],
    K = 512, N = 400, size1 = 256)])
    i += 160
signal_fft = delete(signal_fft, 0, axis = 0)

MFCC = zeros(13)
for i in signal_fft:
    MFCC = vstack([MFCC, mfcc(i)])

MFCC = delete( MFCC, 0, axis = 0)
savetxt("MFCC.csv", MFCC, delimiter = ',')
# MFCC = genfromtxt("MFCC.csv", delimiter = ',')
clf = mixture.GMM(n_components = [0,1], covariance_type = 'full')
clf.fit(MFCC)
prob = clf.score_samples(MFCC)
```

5.1.2 Weigthed MFCCs

This approach was to reduce the dimentionality of the MFCCs by taking weigthed MFCCs. The MFCCs are normalised based on the maximum and minimum values. Then the variance is used as the weigth. First approach was to see the difference in consecutive frames and the second was to store absolute values. Neither of them yielded any successful results.(Source file: separation.py)

```
maximum = np.amax(MFCC, axis = 0)
minimum = np.amin(MFCC, axis = 0)

MFCC = (MFCC - minimum)/(maximum - minimum)
# print maximum, minimum
weights = np.var(MFCC, axis = 0)

Weighted_MFCC = np.multiply(MFCC, weights)
Weighted_MFCC = np.sum(Weighted_MFCC, axis = 1)
print Weighted_MFCC, Weighted_MFCC.shape
```

5.1.3 Peak Frequency

Since different speakers have different fundamental frequencies, the peak will show up in different MFCC bandpass filters. Contrary to the assumption, most of the energy for all speakers was found in the 0^{th} band indicating the presence of low frequency noise.(Source file was composed in Ipython and lost in transaction)

5.1.4 Pre-emphasis and signal conditioning

This approach is based on signal conditioning to remove any chunk of the signal bounded by the axis by two consecutive zeros and a peak whose value is 100 times lesser than the maximum possible peak. The reconstruction lead to a noisy output.(Source file: class.py)

```
class part:
    def __init__(self, start, signal):
        self.start = start
        self.maximum, self.end = self.locator(signal[start:], start)
        self.maxima = abs(signal[self.maximum])

    def sign(self, i):
        # If it is positive return True, else return False.
        if i > 0:
            return True
        return False

    def locator(self, signal, start):
        # Look for the general direction, then locate the maxima and zero value.
        maximum, positive = 0, False
        i = 0
        while(signal[i] == 0):
            i += 1

        positive = self.sign(signal[i])

        while (self.sign(signal[i]) == positive and i + start < length):
            i += 1
            # print signal[i]
            if(positive == True):
                if(signal[i] > signal[maximum]):
                    maximum = i
            else:
                if(signal[i] < signal[maximum]):
                    maximum = i

        maximum = start + maximum
        end = start + i
        return maximum, end

    def display(self):
```

```
        # Prints the elements of the class
        print "Start " + str(self.start) + "\tMaximum " +
        str(self.maximum) + "\tEnd " + str(self.end)

rate, signal = read("conv.wav")
original_list = []
i = 0
length = len(signal)
while i < len(signal)-100:
    # print i
    object = part(i, signal)
    i = object.end
    original_list.append(object)
print "end"
global_max = max( map( lambda x: x.maxima, original_list))
new_list = filter( lambda x: x.maxima >= float(global_max)/100.0, original_list)
sig1 = []
for i in original_list:
    sig1 = concatenate([sig1, signal[i.start:i.end]])
    i.display()

write("unedited.wav", rate, sig1)
```

5.1.5 K Means in MATLAB

Final approach was to try with MFCCs and DDFCs in a k-means classifier in MATLAB. This also misclassified.

```
clf = mixture.GMM(n_components = [0,1], covariance_type = 'full')
clf.fit(MFCC)
prob = clf.score_samples(MFCC)
```

5.2 Supervised Learning

A 1 second clip of both speakers was manually clipped and used to train the classifier. 5 different machine learning algorithms all gave completely wrong results.(Source file: splitting.py)

```
def reading():
    rate, signal = read("dad.wav")
    MFCC0 = inputs(signal)
    MFCC0 = labeller(MFCC0, 0)

    rate, signal = read("daughter.wav")
    MFCC1 = inputs(signal)
    MFCC1 = labeller(MFCC1, 1)

    MFCC = vstack([MFCC0, MFCC1])
    print MFCC.shape

    rate, signal = read("conv.wav")
    MFCC_conv = inputs(signal)

    # SVM
    clf = SVC()
    clf.fit(MFCC[:, :-1], MFCC[:, -1])
    results = clf.predict(MFCC_conv)
    # results = column_stack((MFCC_conv, result))
```

Chapter 6

Conclusion

```
S       H       F       A       N
Forset Classifier:

[[9 0 2 0 9]
 [0 9 0 3 1]
 [5 0 8 1 1]
 [0 5 1 9 2]
 [4 2 0 0 7]]
0.538461538462
```

Figure 6.1: Confusion Matrix

Phase I has been successfully implemented with upto 55% accuracy. The classification model used is a random forest. The accuracy levels are still low and can be increased up to 70% by extracting more features like quartile and median features.
The classifier misclassified neutral for sadness and vice versa. This is as understandable even in a normally it is difficult to decode when a person is sad.
Phase II requires more research to be successful. It also needs a much better dataset to work with. Unsupervised learning requires more than just MFCCs to be successful. A future implementation with i-vectors will hopefully yield better results.

Bibliography

[1] MFCC Generation by on practical cryptography page http://practicalcryptography.com/miscellaneous/machine-learning/guide-mel-frequency-cepstral-coefficients-mfccs/

[2] Source code for python implementation of MFCCs on https://github.com/jameslyons/python_speech_features

[3] Pitch detection ideas from DSP Stack exchange http://dsp.stackexchange.com/questions/411/tips-for-improving-pitch-detection

[4] Dataset from University of Milano - Bicocca's MIND page http://www.mind.disco.unimib.it/Home_Page.asp?lang=en

[5] Idea of implementation and inspiration from Kaggle https://www.kaggle.com/

[6] Pitch detection algorithms stright out of Github https://gist.github.com/endolith/255291#L38