POLITECHNIKA WROCŁAWSKA

WYDZIAŁ PODSTAWOWYCH PROBLEMÓW TECHNIKI

KIERUNEK: Informatyka

SPECJALNOŚĆ: Computer Security

PRACA DYPLOMOWA MAGISTERSKA

Voice authentication library

AUTOR:
Mariusz Pasek

PROMOTOR:
dr Filip Zagórski

OCENA PRACY:

WROCŁAW 2016

CONTENTS

ACRONYMS AND ABBREVIATIONS

N_{frames}	Number of frames in processed sample
N_{frame}	number of samples in processed frame
T	Sampling period
V_n	Vector of extracted speaker features
ω_0	Base frequency
ε_f	Numerical epsilon for float numbers
f_h	Higher bound of frequency used in mel filter bank generation in freq. domain
f_l	Lower bound of frequency used in mel filter bank generation in freq. domain
m_h	Higher bound of frequency used in mel filter bank generation in mel domain
m_l	Lower bound of frequency used in mel filter bank generation in mel domain
n_c	Number of MFCC used
n_l	Number of liftered MFCC
`ASV`	Automatic Speaker Verification
`WAV`	Waveform File Format
CDF	Cumulative Distribution Function
CMS	Cepstral Mean Subtraction
DAC	Digital to analog converter
DCT	Discrete Cosine Transform
DFT	Discrete Fourier Transform
DSP	Digital Signal Processing
DTFT	Discrete Time Fourier Transform
FFT	Fast Fourier Transform
FIR	Finished impulse response
GMM	Gaussian Mixture Model
IDTFT	Inverse Discrete Time Fourier Transform
MFCC	Mel frequency cepstral coefficient
PCA	Principal Components Analysis
SVM	Suppor Vector Machine

1. INTRODUCTION

The main goal of this thesis is to explain and implement techniques used during speaker verification process. The main applications of such algorithms are access control and transaction authentication [14]. Moreover text independent speaker verification can be used to construct protocols that would contain also some sort of a challenge (like request to say specific combination of words) and therefore would not be vulnerable to simple replaying voice of the legitimate speaker. Automatic speaker verification does not require specialized hardware (as in the case of fingerprint verification), while preserving the advantages of biometric systems (like lack of password/PIN that could leak or be forgotten). `ASV` is also often chosen as part of multilevel access control and it was deeply researched during last years [9].

This thesis consists not only of theoretical background and explanation of the whole process, but also a source code of a library and simulation results that are proving effectiveness of presented methods. The library was written in `Python` and could be successfully used as a reference model during implementing such system in lower level languages.

The part of the library that is responsible for classification of the speaker is also written in a flexible way that not only lets to include it in some other project, but also easily extend it by more specialized classification methods.

2. DIGITAL SIGNAL PROCESSING

2.1. GENERAL CONCEPT OF DSP

DSP (Digital Signal Processing) deals of analyzing some streams of data. This data could be some physical measurements (like sound, electromagnetic wave readings, temperature, voltage, etc.), but it could be also incoming stream of encrypted or compressed data. In most of the cases, the data is coming from some kind of sensor (voltmeter, microphone). In such case it comes firstly in analog form and should be initially digitalized. Digitalization is a process that consists of two steps:

— Data should be sampled in time (time quantization),
— Values of each sample needs to be quantized as well (amplitude quantizetion).

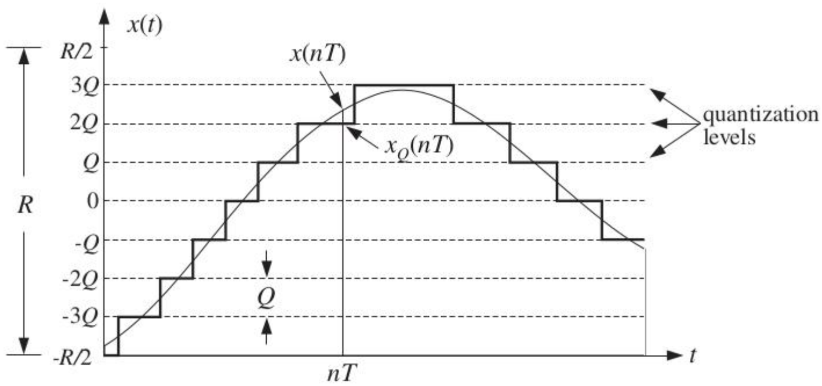


Fig. 2.1: Example quantization of analog signal [13]

Figure 2.1 shows some example process of quantizing analog signal. Quantization in time is about choosing some specific period (usually in DSP denoted as T) called sampling period. Each sample of incoming data is collected with this time interval.

In order to store each sample in digital form it is also necessary to choose a specific range and number of bits that would define values of measured signal. This approach lets to store processed samples in electronic memory for later processing. It is of course clear that choosing specific minimal step between two samples implies necessity of rounding signal - i.e., loosing part of information. Another undesired phenomena is saturation. The signal is saturated, if its initial value exceeds defined range, i.e. cannot be expressed precisely in chosen notation.

After the digitalized signal is processed it is also applied to many algorithms that are widely used during such processing (like filtering, windowing, convolving, etc.). Each of them has its own properties and some will be described later on in this chapter.

The final stage of digital signal processing is giving an output of whole process. The output could be another stream of data (like decrypted data, decompressed image/audio stream) or single message (like identity of the speaker). All of these steps that are building DSP chain are summarized in figure 2.2.

Sampling → Quantization → Processing → Formulating output

Fig. 2.2: Typical DSP chain

2.2. FOURIER TRANSFORM

2.2.1. Theoretical background

Let us define harmonic (subcarrier) of the signal. Harmonic is some periodic function. If we defined the base frequency ω_0, then we could say that frequency of any harmonic of order k would be given by following equation:

$$\omega_k = k \cdot \omega_0 \tag{2.1}$$

The subcarrier could be any periodical function, but the most widely used one is a sine wave [13]. Lets assume that we obtain following periodical signal:

$$\forall_t \; x(t) = x(t+T) \tag{2.2}$$

Then we would like to express signal x as a sum of its harmonics:

$$x(t) = \sum_{k=-\infty}^{\infty} c_k e^{jk\omega_0 t} = \sum_{k=-\infty}^{\infty} c_k e^{jk2\pi t/T} \tag{2.3}$$

Where:

— $\omega_0 = \frac{2\pi}{T}$ – base frequency,
— c_k – coefficients of harmonics,
— T – period of the signal.

With simple conversion of equation 2.3 we would obtain the following:

$$\begin{aligned}\int_T x(t)e^{-jn\omega_0 t}dt = \int_T \left(\sum_{k=-\infty}^{\infty} c_k e^{jk\omega_0 t}\right) e^{-jn\omega_0 t}dt = \\ = \sum_{k=-\infty}^{\infty} c_k \left(\int_T e^{j(k-n)\omega_0 t}dt\right)\end{aligned} \tag{2.4}$$

It is worth to notice that:

$$\int_T e^{j(k-n)\omega_0 t}dt = \begin{cases} T, & k = n \\ 0, & k \neq n \end{cases} \tag{2.5}$$

This leads to conclusion:

$$\int_T x(t)e^{-jk\omega_0 t}dt = c_k T \tag{2.6}$$

$$c_k = \frac{1}{T}\int_T x(t)e^{-jk\omega_0 t}dt \tag{2.7}$$

Equation 2.7 shows how to calculate coefficients of harmonics of each order for a given periodical signal. It is easy to see that frequency of each subcarrier is given by equation $k \cdot \omega_0$. Thanks to trigonometrical expression of complex numbers:

$$e^{j\alpha} = cos(\alpha) + jsin(\alpha) \tag{2.8}$$

it becomes clear that periodical signal x is expressed by sum of sines and cosines (just as initially written in equation 2.3). Fourier transform is linear operation that transforms periodical signal into its harmonic coefficients. Except for its continuous interpretation previously presented it has also its discrete form. It is called DFT - Discrete Fourier Transform. It is given by following equations:

$$x_n = \sum_{k=0}^{N} c_k e^{jk\omega_0 n} \tag{2.9}$$

$$c_k = \frac{1}{N}\sum_{k=0}^{N} x_n e^{-jk\omega_0 n} \tag{2.10}$$

Where equation 2.10 describes analysis of given signal, while 2.9 describes synthesis. The most commonly used variation of this algorithm is FFT (Fast Fourier Transform), which is an optimization of DFT, but works basically the same [13].

2.2.2. DFT basic properties

DFT of simple functions

It can be easily noticed from figures 2.3 - 2.5 that signal in frequency domain is antisymmetric on the imaginary side and symmetric on the real side. This property is satisfied only for real signals. It basically means that if one is processing real signals (like sound) it is not necessary to store the whole output of DFT. Instead of it one half could be skipped. This approach can let to optimize further computations and reduce memory consumption.

Another relevant property is that odd functions in frequency domain have only imaginary component, while even functions have only real component. This is simple result of the fact that sine (imaginary part) is an odd function and cosine (real part) is even.

DFT of time shifted signal

Figure 2.6 shows the result in frequency domain of shifting signal in time domain. The shifted sine function is not fully imaginary anymore. Instead of it part of its imaginary part has moved into the real part (absolute value has left the same). This is the result of rotation of samples of DFT. This is very important property in DSP. It sounds as follows: every cyclic shift in time domain results with rotation in frequency domain and vice versa.

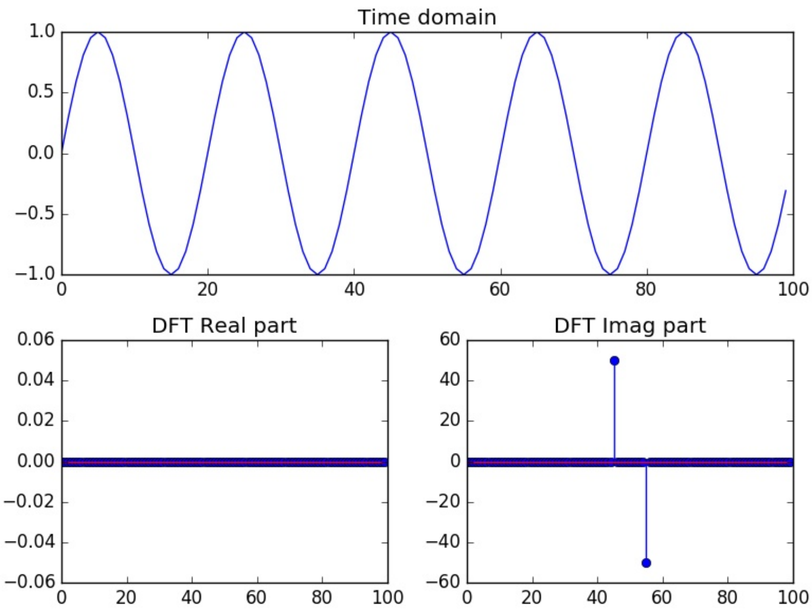


Fig. 2.3: Sine function expressed in time and frequency domain

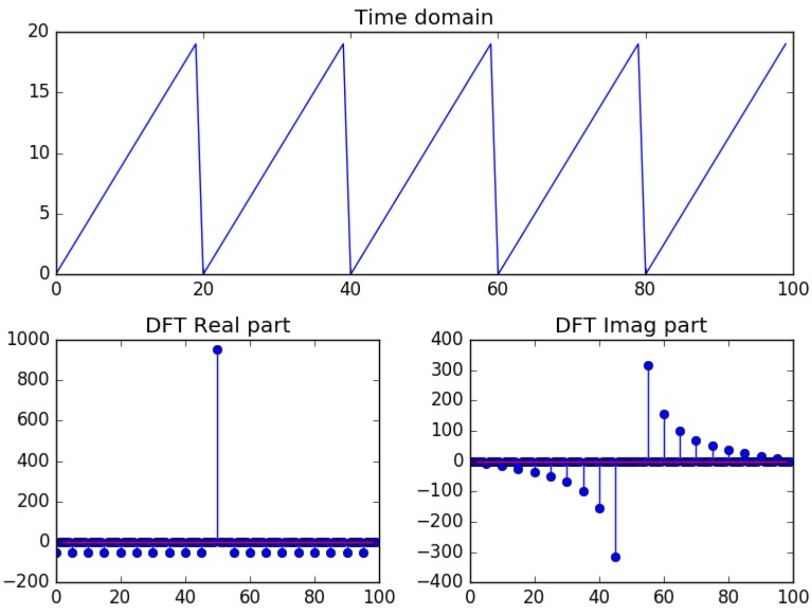


Fig. 2.4: Sawtooth function expressed in time and frequency domain

DFT of two multiplied signals

Figure 2.7 shows the result of simple multiplication of sine wave and rectangular window. The frequency domain seems to be very distorted and hard to recognize as spectrum of sine function. This is the effect of another important DFT property. Multiplication of signals in one domain results in convolution in another. Convolution of discrete, finite signals is given by equation:

$$(f * g)[n] = \sum_{m=0}^{N-1} f[m]g[n-m] \tag{2.11}$$

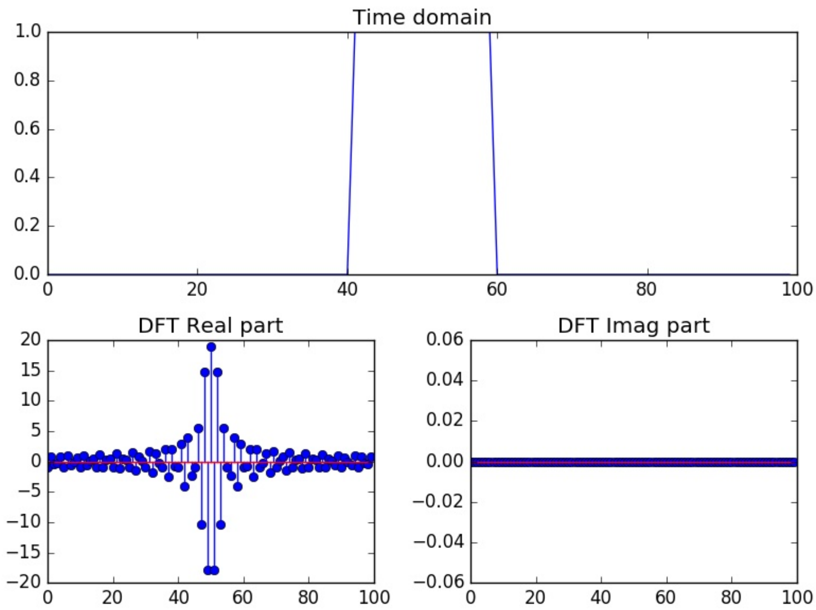


Fig. 2.5: Rectangular window expressed in time and frequency domain

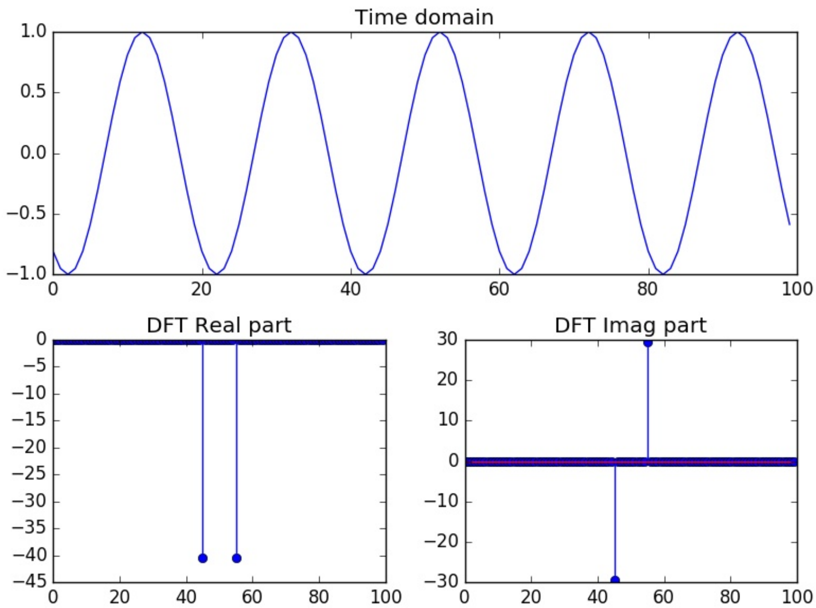


Fig. 2.6: Shifted sine function expressed in time and frequency domain

Knowing this property lets to easily explain the output of DFT of multiplying sine wave with rectangular window. This distortion can lead to serious mistakes during later processing of the signal and that is the reasoning for using different windows than rectangular.

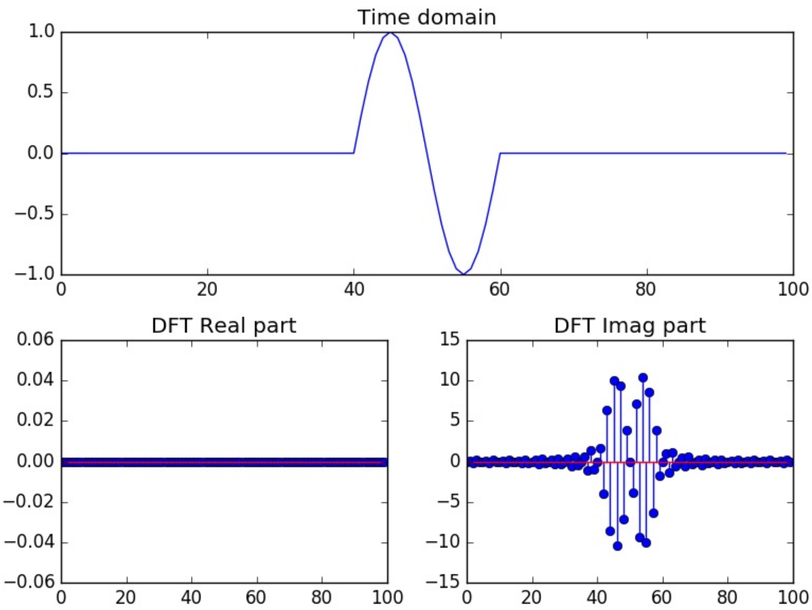


Fig. 2.7: Sine function multiplied by rectangular window expressed in time and frequency domain

2.3. SAMPLING THEOREM

As mentioned in section 2.1, the first step of digital signal processing is sampling it. In order to sample analog signal it is necessary to choose sampling period. It is a trade-off between resources consumption and precision. Too short sampling period could lead to unnecessary increase of memory consumption and could even make impossible achieving assumed performance (like real time processing). On the other choice of too long sampling period could lead to loss of relevant information.

One of the assumptions about signals in DSP is that their spectrum remains constant in some short period (like single radio symbol during wireless transmissions or single tone during speech processing). This means that collected samples are sorted in groups of the same size (lets define them as frames). In order to choose T properly one should pick it in such a way that it would be possible to calculate the amplitude of the subcarrier of the highest possible frequency. The equation that defines minimal frequency of sampling - f_s that lets to calculate amplitude of some maximal harmonic (of frequency ω_m) is called Nyquist criterion. It is presented in the equation 2.12:

$$f_s = 2 \cdot \omega_m \tag{2.12}$$

If one would try to calculate DFT of the signal without following Nyquist criterion it would be impossible to distinguish signals of some frequencies just as presented in figure 2.8.

Phenomena of aliasing causes the sensor with insufficient sampling frequency to register signals of lower frequencies that were never actually received. Instead these are just decimated signals of higher frequencies that were impossible to be registered because of not following Nyquist criterion.

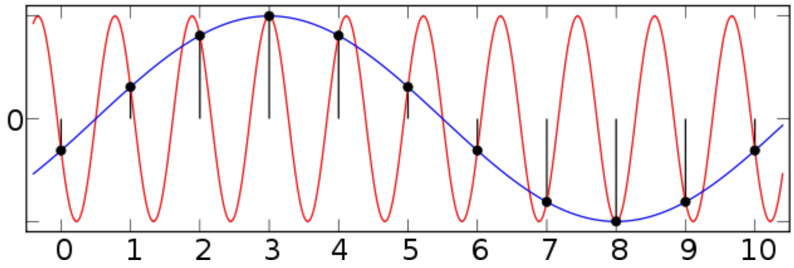


Fig. 2.8: Phenomena of aliasing [1]

In order to prevent this phenomena to happen it is necessary to apply low pass analog filters that can exclude signals of high frequencies before sampling. Therefore any DAC (digital to analog converter) uses such filter as a first step of conversion. Since the subject of this thesis is just to process digital signals and whole conversion is provided by microphone and computer sound card it is not necessary to prevent aliasing occurrence.

2.4. WINDOWING

Since during DSP we are processing just somehow preprepared frames, we need to handle some kind of windowing. It basically means that one should try to minimize spectrum distortion caused by dealing with only some part of the signal instead of its full period. Such situation causes phenomena called ”spectrum leakage”. It occurs if length of processed signal is not an integer multiplication of its period and leads to significant distortion of the original signal. The phenomena of spectrum leakage is presented in figure 2.9.

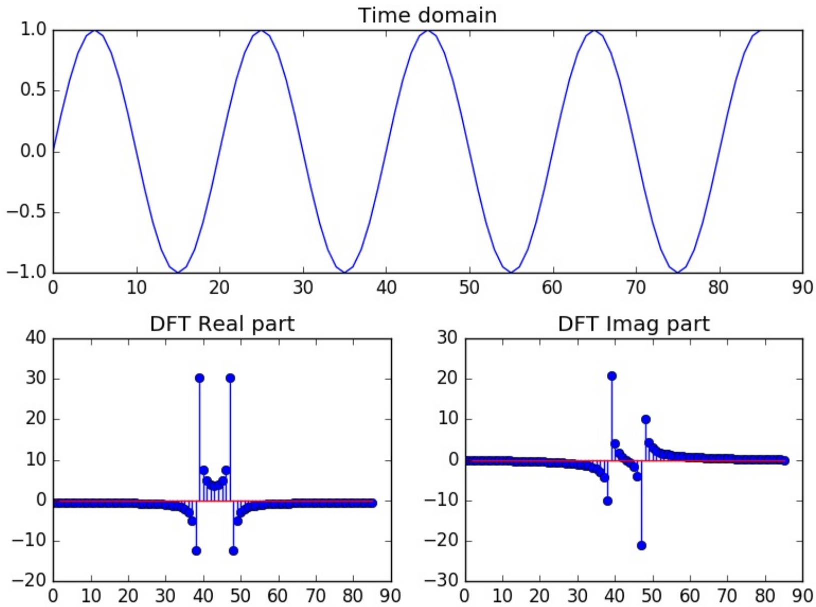


Fig. 2.9: Result of calculating dft of not integer number of periods – spectrum leakage

This is more or less the result of rectangular windowing. The way of minimizing this phenomena is to apply windows with more smooth edges. There are several windows widely used in DSP and one of the most popular is Hamming window. It is given by the equation:

$$w(n) = \alpha - \beta\, cos\left(\frac{2\pi n}{N-1}\right) \tag{2.13}$$

where:

— $\alpha = 0.54$
— $\beta = 1 - \alpha = 0.46$
— N is a number of samples in frame

The plot presenting Hamming window generated by given equation can be seen in figure 2.10. And the result of multiplying signal from figure 2.9 by presented Hamming window is shown in figure 2.11.

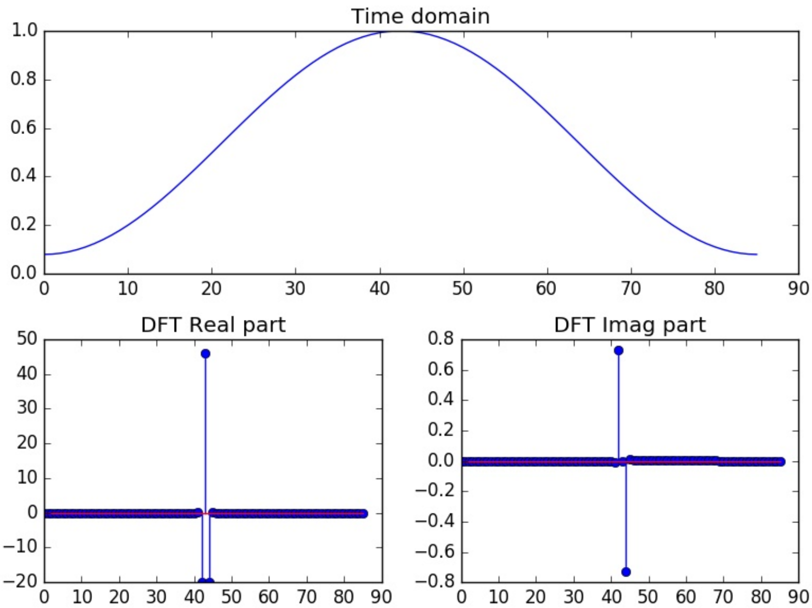


Fig. 2.10: Hamming window

It can be noticed that side peaks where not eliminated completely, but thanks to this solution it was possible to minimize effect of spectrum leakage. This phenomena was a motivation of using Hamming window as the first step of signal processing in cepstral analysis described in chapter 3.2.

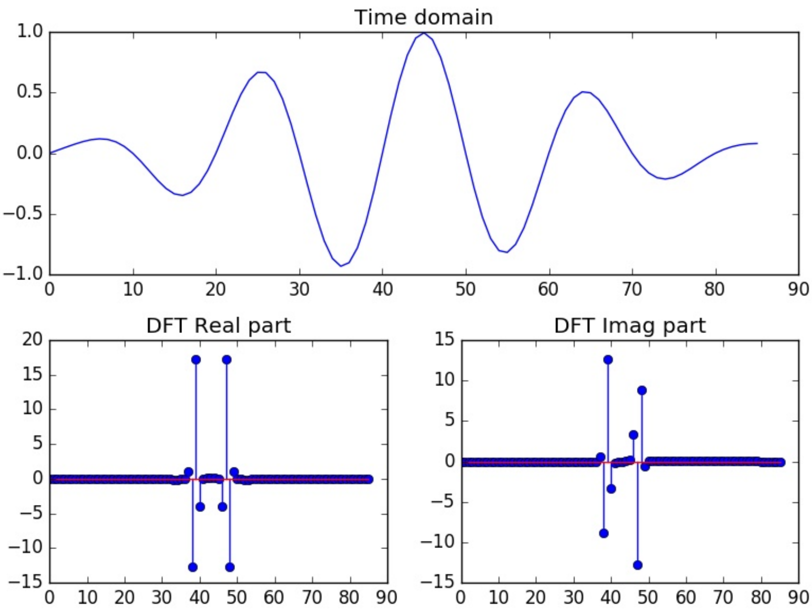


Fig. 2.11: Not integer number of sine periods multiplied by hamming window

2.5. DIGITAL FILTERS

Filtering is one of the most commonly used functionality provided by DSP. The basic task of filter is to shape the spectrum of processed signal in a desired way by multiplying its by filter's spectrum. There are several basic types of filters:

— lowpass filter,
— highpass filter,
— bandpass filter,
— bandstop filter.

The spectral presentation of these filters is shown in figure 2.12. It can be easily seen that changes in these filters are rectangular – are perfectly sharp. In real case scenarios it is not possible, to obtain such efficiency of filters in feasible computation time. That is why these filters are considered to be ideal, and commonly used filters are approximations of these. Since samples are firstly usually processed in time domain it would be convenient to map desired filter into some array of coefficients that would correspond to its spectral form. These coefficients must be convolved with incoming samples (because of basic DFT properties). One can write down following equation:

$$y(n) = \sum_{i=0}^{M-1} x(n-i) \cdot c(i) \tag{2.14}$$

Where:

— c – filter coefficients,
— M – order of the filter.

It is not the only possible form of filtering, but one of most widely used. Order of the filter defines complexity of filtering and number of samples that need to be stored. Filters of this kind are called FIR (finished impulse response) filters. This name is corresponding to very important

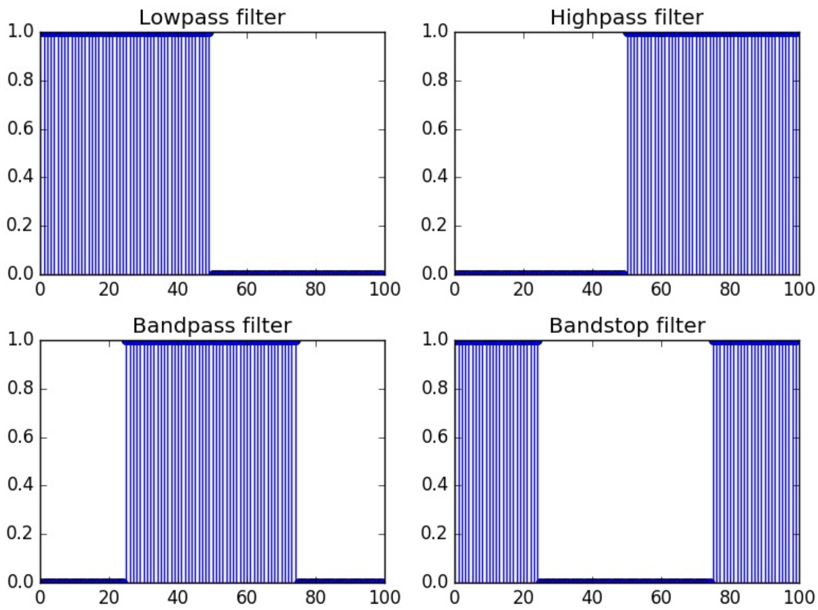


Fig. 2.12: Spectral presentation of basic filter types

property of such filter. The length of output on any input signal is never longer than M samples. It is worth to notice that if one would put Dirac's delta as an $x(n)$ signal, the output would be $c(n)$. It is very important property that lets to check coefficients of some unknown FIR filter. It is not the only kind of filter used in DSP, but only such is used in this thesis.

Ideal filter impulse response

As it was mentioned previously, the generally used filters are just approximation of ideal ones. It is necessary then to find the coefficients of ideal filter. One could start with lowpass filter. Its ideal frequency response $H(\omega)$ is bounded to its impulse response $h(k)$ by DTFT and IDTFT relationships [1]:

$$H(\omega) = \sum_{n=-\infty}^{\infty} h(n) e^{-j\omega n} \tag{2.15}$$

$$h(n) = \frac{1}{2\pi} \int_{-\pi}^{\pi} H(\omega) e^{j\omega n} d\omega \tag{2.16}$$

Since time coefficients of filters are considered to be real numbers, their spectral forms shown in figure 2.12 can be treated as symmetrical over vertical axis. Therefore one could define $H(\omega)$ of the lowpass filter in the following way:

$$H(\omega) = \begin{cases} 1, -\omega_c \leq \omega \leq \omega_c \\ 0, (-\pi \leq \omega < -\omega_c) \ \vee \ (\omega_c < \omega \leq \pi) \end{cases} \tag{2.17}$$

It lets to calculate impulse response of ideal lowpass filter:

[1] It may seem confusing that the DTFT relation is used in this consideration, but it is necessary. DFT assumes that signal is sampled and quantized, while DTFT assumes only sampling, and allows the signal change continuously and it results with spectrum being continuous function, i.e. such case is more general. Derivation of DTFT equations was not covered in this thesis, but is analogous to DFT.

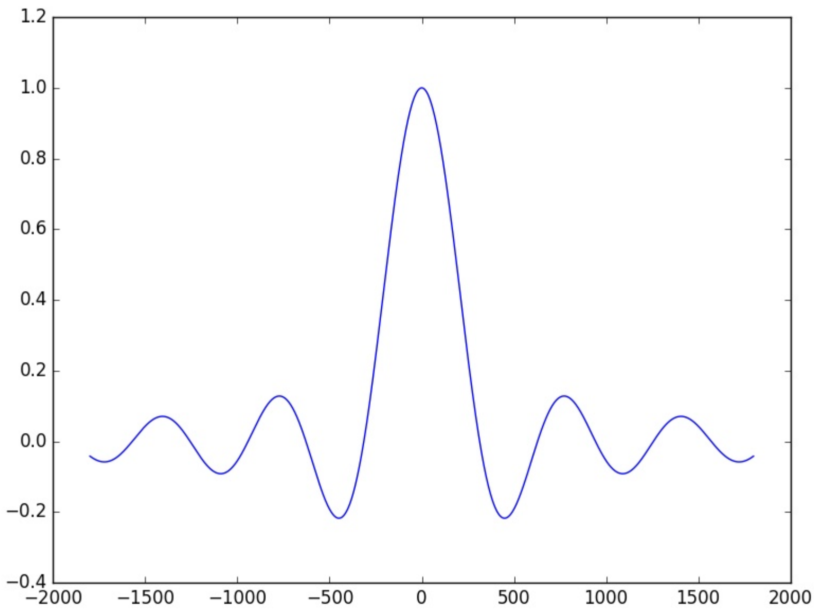


Fig. 2.13: Sinc function

$$
\begin{aligned}
h(n) = \frac{1}{2\pi}\int_{-\pi}^{\pi} H(\omega)e^{j\omega n}d\omega = \frac{1}{2\pi}\int_{-\omega_c}^{\omega_c} 1 \cdot e^{j\omega n}d\omega = \\
= \left[\frac{e^{j\omega n}}{2\pi jn}\right]_{-\omega_c}^{\omega_c} = \frac{e^{j\omega_c n} - e^{-j\omega_c n}}{2\pi jn} = \\
= \frac{cos(\omega_c n) + jsin(\omega_c n) - (cos(\omega_c n) - jsin(\omega_c n))}{2\pi jn} = \\
= \frac{sin(\omega_c n)}{\pi n}
\end{aligned} \tag{2.18}
$$

As it was shown in equation 2.18, the impulse response of lowpass filter ($h_{lowpass}(n, \omega_c)$) is given by sinc function (presented in figure 2.13). Having impulse response of single lets to obtain impulse responses for other ideal filters. Having in mind that Dirac's delta spectral form is a sequence of ones lets to write the following equations [6]:

— $h_{highpass}(n, \omega_c) = \delta(n) - h_{lowpass}(n, \omega_c)$
— $h_{bandpass}(n, \omega_a, \omega_b) = h_{lowpass}(n, \omega_b) - h_{lowpass}(n, \omega_a)$
— $h_{bandstop}(n, \omega_a, \omega_b) = \delta(n) - h_{bandpass}(n, \omega_a, \omega_b)$

Designing FIR filters

Until now domain of n was all integer numbers. In order to obtain feasible FIR filter one has to limit its order to some specific M. There are many approaches to do so. One of the most basic is rectangular window. This method composes of two steps. Truncating symmetrically impulse response and shifting it to only positive part. Truncating is necessary to reduce the order of the filter and shifting must be done for all real time applications, because it is impossible to get the value of future samples.

Rectangular window is not only one used for designing FIR filters. Hamming and Kaiser windows are used as well and each of them have unique impact on behavior of signal processing, but all of these are using the same theoretical basis already described in this chapter.

3. SPEAKER RECOGNITION AND VOICE AUTHORIZATION

3.1. GENERAL APPROACH OF SPEAKER RECOGNITION AND VOICE AUTHORIZATION

Speaker recognition process is generally divided into three steps:

Feature extraction – it is necessary in order to extract only relevant data from incoming signal that are characterizing particular speaker (or some specific sound generated by him – like single vowel). In order to do so, one must divide stream of input into some smaller groups (frames). Every single frame should be considered as a single sound – vowel in this case. It basically means that the output of feature extraction should be some coefficients that would characterize this frame.

Classification – after extracting feature from the frame one should classify it, in order to determine if it is recognized as some known vowel of previously defined speaker. This is the second step of speaker recognition and it is somewhat independent on the first.

Decision – having each frame classified lets to summarize collected information and finally decide, if stream of data was generated by known speaker or not. Decision block could also use some previous knowledge like information, about this, what the speaker actually said.

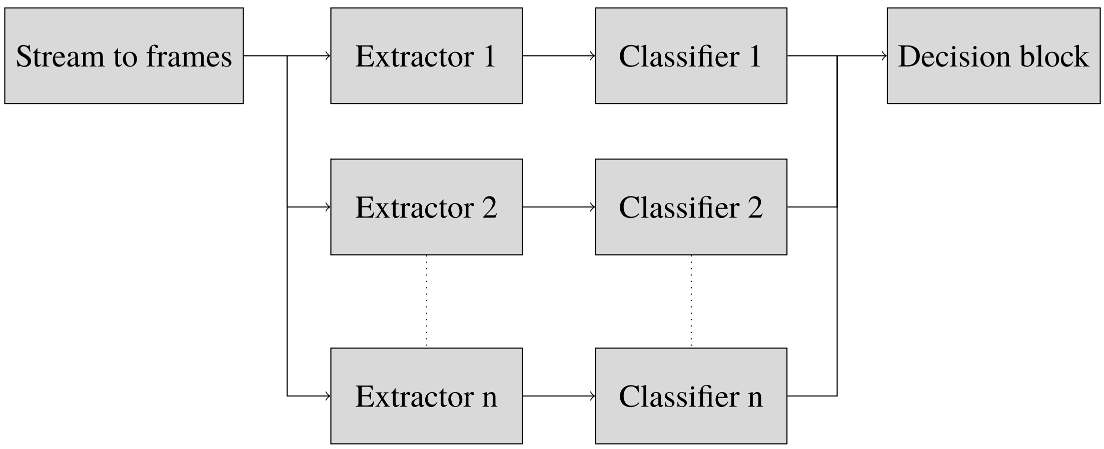


Fig. 3.1: Speaker recognition algorithm

3.1.1. Voice authorization

Voice authorization algorithm could be used as part of an authorization tool, but it is not enough. In real case scenario one should protect from replay attacks. It would be very easy to record voice of a person and then play it again, to use it as authorization sequence. That is why real voice authorization mechanism should compose of following procedures:

— challenge – verifier (Victor) queries prover (Peggie) with some sequence (some sentence for instance),
— response – prover Peggie replies with its answer,
— verification – finally Victor verifies an answer with two algorithms:
    — Speaker recognition – Victor checks, if given sentence was told by predefined person. It uses algorithm steps previously described in this chapter,
    — Speech recognition – Victor checks, if Peggie has told exactly, what was demanded in challenge. It is not the subject of this thesis, but there are many available ready solutions for this functionality (like one provided by `Google`).

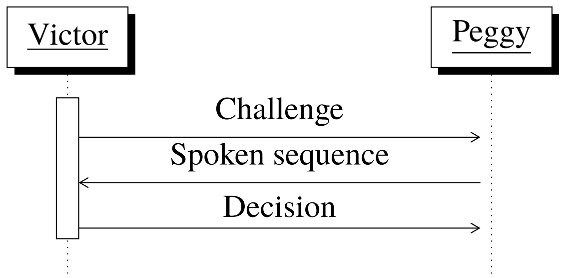


Fig. 3.2: Voice authorization message sequence

3.2. FEATURE EXTRACTION

As previously described, the first step of speaker recognition is extraction of speaker features from incoming signal. There are many approaches of this action, but one that became the most popular (as well in speech and speaker recognition systems) is MFCC extraction. It bases on similar way of processing that human ear does and it is calculated in following steps:

1. pre-emphasis,
2. decimation,
3. frame division and operations computed per frame:
    a) zero padding,
    b) applying FFT,
    c) conversion into mel scale,
    d) applying DCT,
    e) liftering,
    f) vector enhancements,
4. appending deltas,
5. cepstral mean subtraction,
6. feature warping

3.2.1. Pre-emphasis

It has been researched that speaker features are being kept by higher frequencies of incoming signal. Therefore if one does not want to focus on information that was actually said by the speaker, but wants to focus only on his identity, it is necessary, to process the signal with highpass filter. The coefficients of applied filter are as following:

— $c_0 = 1$
— $c_1 = -0.97$

These values of pre-emphasis filter are usually used in publications treating about speaker recognition [2]. An example of pre-emphasis is shown in figure 3.3. An initial signal is composed of low frequency sine wave and high frequency random noise. Filtering is canceling sine element of signal leaving only random noise.

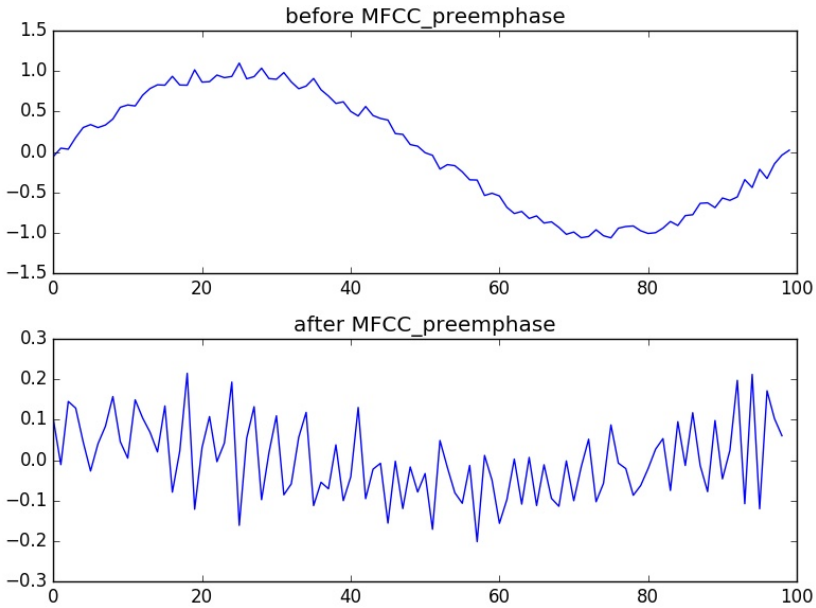


Fig. 3.3: Example of pre-emphasis high band filter application

3.2.2. Decimation

It is not a required step, but it can be essential, if it is necessary to process signals sampled with various frequencies and for optimization purposes. Decimation is an algorithm that skips desired part of samples and therefore it reduces the frequency of sampling the signal. It lets to reduce number of processed samples (with simultaneous reduction of accuracy) and can let to normalize frequency of sampling of all incoming signals (regardless of the quality of them in the first place).

3.2.3. Frame division

As it was previously presented in figure 3.1, an incoming signal must be divided into single frames. The only thing that needs consideration during framing division are two parameters: frame length and frame step.

Frame length – the length of the frame must be optimized in such a way that it would be highly probable to correspond, to the length of processed vowels (or at least part of them). It should never be to long, because in such case an algorithm would process data containing two vowels simultaneously. In speaker recognition algorithms this parameter was empirically determined as 25 ms.

Frame step – it is probable that single vowel would not exactly fit into one frame even if its length would be exactly the same, because it can be shifted. That is why it is also necessary to define frame step separately in order to increase probability of positive detection. It is also trade-off between accuracy of computation and optimization of an algorithm. The value of frame step in publications dealing with speaker recognition problem is 10 ms [4].

3.2.4. Zero padding

Since FFT is optimization of DFT algorithm that works most efficiently if applied on signals of length of power of 2, it can be necessary to extend the length of frames in such a way that their length would satisfy this condition. It is done by simple zero padding that is a neutral operation for DFT algorithm.

3.2.5. Applying Fast Fourier Transform

As mentioned previously, digital signal processing is much more comfortable in frequency domain. Therefore each zero padded frame must be transformed with FFT.

3.2.6. Conversion into mel scale

Mel filtering is the essential part of MFCC extraction. The general purpose of this step is to reduce amount of information held by samples. Depending on sampling frequency and decimation factor the frame can be of length 256, 512, 1024, etc. samples. Mel filtering lets to reduce these significant numbers into less than 30.

It was discovered that human ear is distinguishing frequencies in logarithmic manner. Mel scale is a representation of the signal that is converting any signal into the same coefficients that are calculated by human ear. This conversion consists of following parameters:

- frequency range (f_l, f_h)– this is the range of frequency that is considered to contain all features of the speaker. It is usually defined as (15 Hz, 4 kHz),
- number of coefficients (n_c)– this is a number into which length of each frame will be reduced. Value 26 is often picked for this parameter.

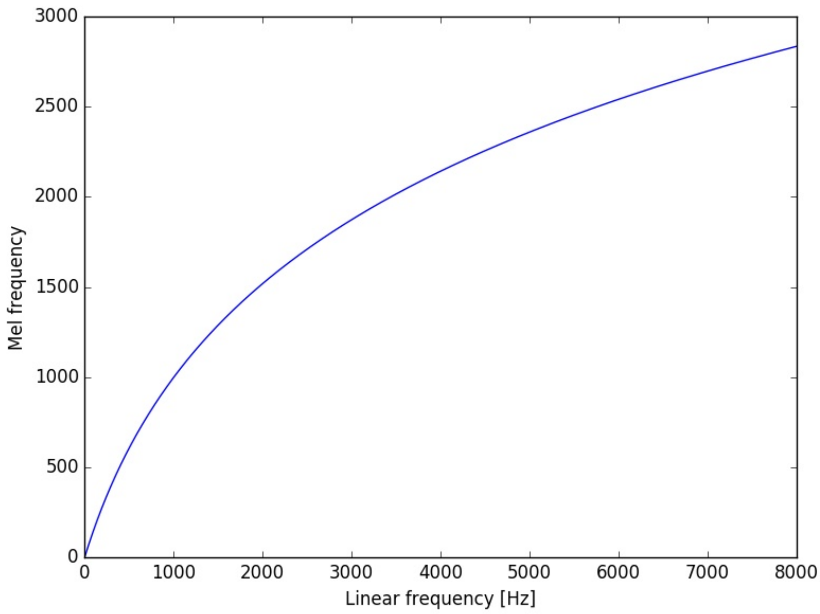


Fig. 3.4: Relation between mel and linear frequency

In order to convert a signal frame into mel scale it is necessary to prepare mel filter bank. The relation between mel and linear scale is shown in figure 3.4. Firstly, one must find mel representation of selected frequency boundaries according to following equation:

$$m(f) = 1127 \ln\left(1 + \frac{f}{700}\right) \tag{3.1}$$

In this case it would be:

— $m_l = m(f_l) = 23.89$
— $m_h = m(f_h) = 2146.08$

After that it is necessary, to calculate n_c + 2 equally spaced mel values from range (m_l, m_h). They are expressed by following equation:

$$m_n = m_l + n \cdot \frac{(m_h - m_l)}{n_c + 1} \tag{3.2}$$

This results with following array:

```
m_n =  [0023.89 0102.49 0181.09 0259.69 0338.29 0416.89 0495.49 0574.09 0652.69
0731.29 0809.89 0888.49 0967.09 1045.69 1124.28 1202.88 1281.48 1360.08 1438.68
1517.28 1595.88 1674.48 1753.08 1831.68 1910.28 1988.88 2067.48 2146.08]
```

It is now necessary to switch back into frequency domain, using equation:

$$f(m) = 700\left(e^{m/1127} - 1\right) \tag{3.3}$$

This results with obtaining logarithmic spaced frequency array, with boundaries the same as initially picked:

```
f_n =  [0015.00 0066.65 0122.02 0181.40 0245.06 0313.33 0386.52 0465.00 0549.15
0639.38 0736.12 0839.86 0951.08 1070.34 1198.22 1335.33 1482.35 1639.98 1809.00
1990.23 2184.55 2392.90 2616.31 2855.85 3112.70 3388.09 3683.38 4000.00]
```

Having this frequency array lets to produce mel filters. Each filter (F_k) is an array of N_{frame} elements, where N_{frame} is the length of the frame. Filters are indexed by range [1,n_c]. Value of each element of a filter is defined by following equation:

$$F_k(n) = \begin{cases} 0, n < f_{k-1} \\ \dfrac{n - f_{k-1}}{f_k - f_{k-1}}, f_{k-1} \leq n \leq f_k \\ \dfrac{f_{k+1} - n}{f_{k+1} - f_k}, f_k \leq n \leq f_{k+1} \\ 0, n > f_{k+1} \end{cases} \tag{3.4}$$

Example filter bank can be seen in figure 3.5. It is easy to notice that each mel filter is simply passing some part of signal spectrum.

Having filter bank prepared, one should simply apply each filter on previously calculated frame spectrum. This operation can be described by following equation:

$$S_k(n) = S(n) \cdot F_k(n) \tag{3.5}$$

where:

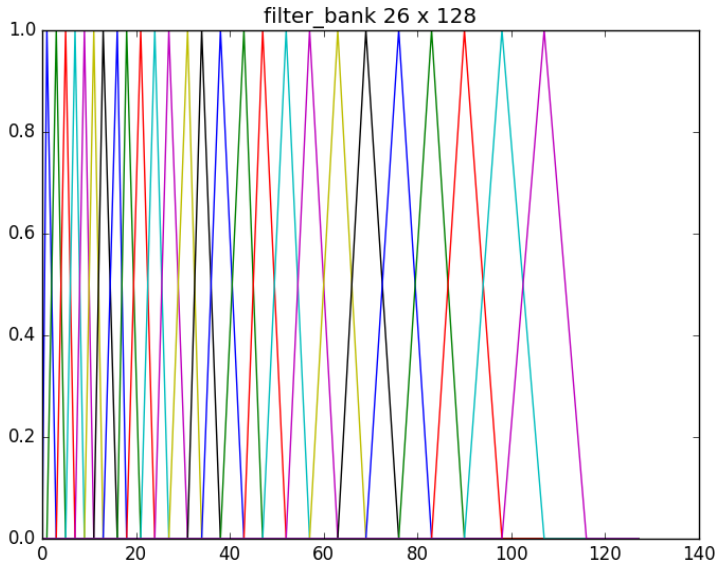


Fig. 3.5: Example filter bank

- $S_k(n)$ – current frame filtered with k-th mel filter
- $S(n)$ – current frame in frequency domain
- $F_k(n)$ – k-th mel filter

Having n_c filtering result, it is necessary to calculate logarithmic power of each result in the following way:

$$MFCC_k = \ln\left(\varepsilon_f + \sqrt{\sum_{n=0}^{N_{frame}} |S_k(n)|^2}\right) \tag{3.6}$$

Addition of ε_f is necessary in order to prevent numerical errors if sum from equation 3.6 would be equal to 0. The procedure ends with extracting n_c MFCC from each frame.

3.2.7. Applying Discrete Cosine Transform

The last step that requires doing is to switch back into time domain. In theory this could be done by inverse FFT algorithm, but this would again result with result in complex numbers. Therefore it is better to use Discrete Cosine Transform DCT. Functionally this operation is analogous to Fourier Transform, except for the fact that it is defined for real numbers.

3.2.8. Liftering

One of the properties of MFCC is that their variance and the average numerical value are decreasing with increase of their index. It basically means that only some subsequence of already calculated $MFCC_k$ array should be taken for further processing. Another MFCC property is that the first coefficient is proportional to the mean of the log spectral energy channels and indicates overall lever for the speech frame. It means that information held by first coefficient is too generous to be used for speaker recognition [12].

Because of following properties of MFCC, one should define another parameter of feature extraction process – n_l – this is number of MFCC used for further processing. This steps results with construction of V_n, which is vector of extracted speaker features in particular frame, where:

$$\forall_{n\in[1,n_l]} V_n = MFCC_{n+1} \tag{3.7}$$

3.2.9. Vector enhancements

The result of liftering is vector V_n and practically it contains all of the features of the speaker. Although the accuracy of classification algorithms described in section 3.3 can be enhanced by some further processing of feature vector. Such actions are common in data mining techniques and usually they do not depend on kind of processed data.

Appending deltas

Previously described MFCC extraction assumed that all of the cepstral coefficients are stationary, although while analysis of speaker signal it is also relevant to take into consideration speed and acceleration of pronounced vowels [12]. The popular algorithm for approximating these dynamic parameters is linear regression. The n-th dynamic coefficient of order K of current frame can be evaluated in the following way:

$$\delta_n^K = \frac{\sum_{k=-K}^{K} k \cdot (V_{n+k} - V_{n-k})}{2\sum_{k=-K}^{K} k^2} \tag{3.8}$$

Such calculated deltas of order 1 and 2 are then appended to feature vector as presented:

$$\tilde{V} = [V_1\ V_2\ ...\ V_{n_l} \quad \delta_1^1\ \delta_2^1\ ...\ \delta_{n_l}^1 \quad \delta_1^2\ \delta_2^2\ ...\ \delta_{n_l}^2]^T \tag{3.9}$$

Cepstral mean subtraction

One of difficulties during classifying incoming samples is changing channel influence on received signal. One of possibilities of compensating it is Cepstral Mean Subtraction CMS algorithm [17]. There are many approaches of calculating it and the most straightforward method looks as follows:

$$\forall_{n\in[1,3\cdot n_l]} \forall_{k\in[1,N_{frames}]} \mathrm{A}_{n,k} = \mathrm{A}_{n,k} - \frac{1}{N_{frames}} \sum_{l=1}^{N_{frames}} \mathrm{A}_{n,l} \tag{3.10}$$

where:

— A – matrix of coefficients of whole processed sample.

Feature warping

The last step of parametrization used in this thesis is feature warping. This operation lets to reduce impact of the channel properties on incoming stream of data and makes it also more independent of additive noise [8]. Feature warping is simply nonlinear conversion of distribution of every extracted coefficient independently that is supposed to make it similar to normal distribution. In order to do so one has to proceed following steps:

1. Get single row (r) of extracted coefficients,
2. Find minimum and maximum of r – it lets to calculate μ and σ values of desired normal distribution, according to following equation:

$$\mu = \frac{\min r + \max r}{2} \tag{3.11}$$

$$\sigma = \frac{\max r - \min r}{6} \tag{3.12}$$

3. Calculate Cumulative Distribution Function of chosen normal distribution (p). Example of such CDF is shown in figure 3.6,

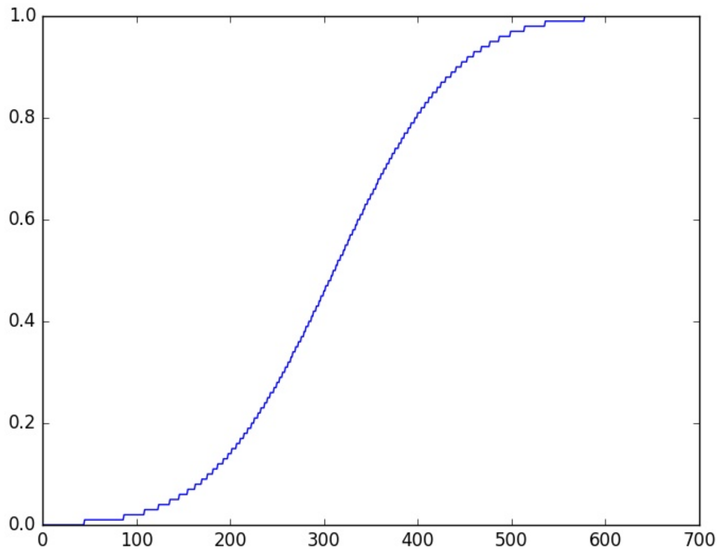


Fig. 3.6: Example of CDF of normal distribution

4. Calculate CDF of r (q)– example shown in figure 3.7,

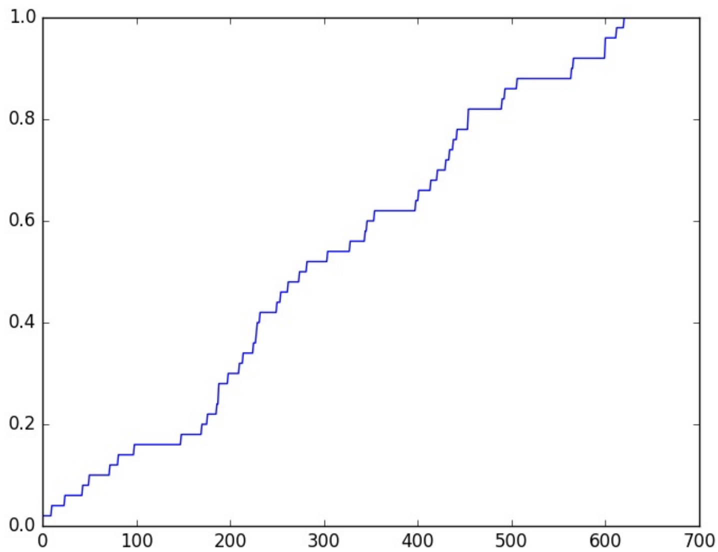


Fig. 3.7: CDF of noised signal

5. Prepare remapping table for vector r. Table would indicate that each α value should be substituted with β, where β can be calculated from following equation [10]:

$$q(\alpha) = p(\beta) \tag{3.13}$$

6. Remap r with prepared table – example showin in figure 3.8.

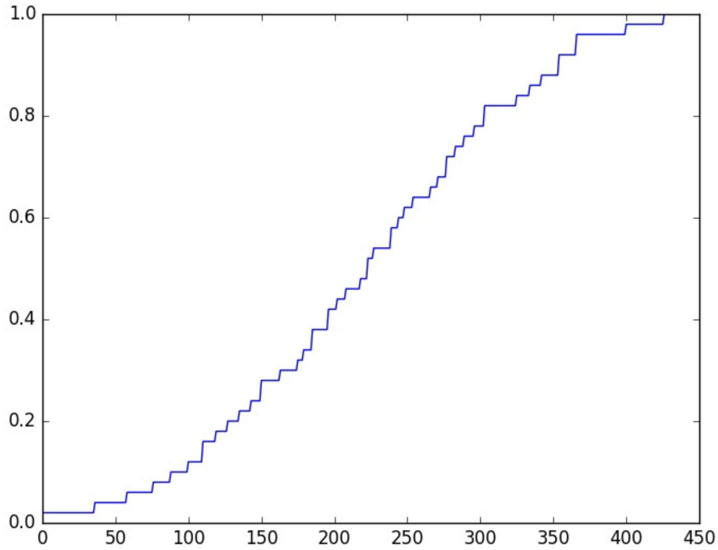


Fig. 3.8: CDF of noised signal after feature warping

As previously mentioned this method lets to remove additive noise and undesired properties of channel from extracted signal. The effectiveness of feature warping can be seen in figure 3.9, where an output of feature warping of initial signal and its distortion is very similar .

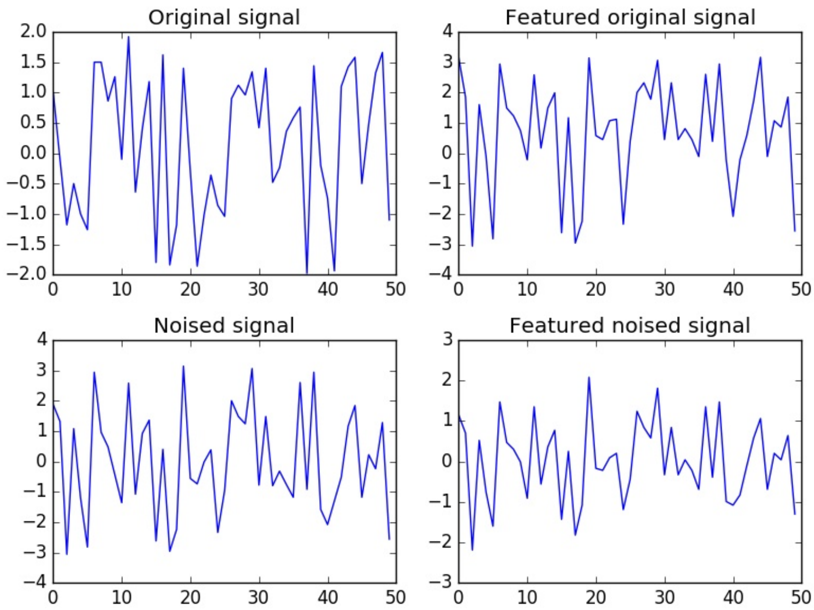


Fig. 3.9: Summary of feature warping effectiveness

3.3. CLASSIFICATION

The general goal of classification methods is to extract the only necessary information of incoming samples and to collect them as set of coefficients that would let to classify other samples properly. Problem defined in this thesis is Binary Classification [3]. One can find many techniques used to solve such problem and some of them will be described in this section.

3.3.1. Popular classification techniques

Gaussian Mixtures

The general assumption of speaker model during Gaussian Mixtures Modeling is that it composes of several states and the observed output is the result of hidden (impossible to be directly measured) state. Distribution of the output for specific i-th state is given by following equation:

$$b_i(x) = \frac{1}{(2\pi)^{D/2}|\Sigma_i|^{1/2}} \times exp\left[-\frac{1}{2}(x-\mu_i)^T(\Sigma_i)^{-1}(x-\mu_i)\right] \tag{3.14}$$

where:

— D – number of dimensions of measured features,
— μ_i – the state mean vector,
— Σ_i – the state covariance matrix.

Function b_i can be interpreted as multidimensional Gaussian probability density function. It basically means that for some fixed state of researched model observed vectors would fall into Gaussian distribution (accordingly to number of dimensions of extracted features). Such assumption leads to following GMM equation:

$$p(x|\lambda) = \sum_{i=1}^{M} p_i b_i(x) \tag{3.15}$$

where:

— $\forall_{i \in [1,M]}\ \lambda = (p_i, \mu_i, \Sigma_i)$ – certain reference model parameters,
— p_i – probability of occurring specific state for reference model.

Equation 3.15 can be interpreted in the following way: *The probability that received sample x came from reference model λ is equal to sum of the probabilities that x was generated by λ in state i multiplied by probability of occurrence of state i.*

In speaker recognition analysis the state can be interpreted as specific sound generated by a speaker (like vowel). The probabilities p_i are more bounded to text dependent information, thus they are considered to be equal to each other. The task of GMM algorithm is to define number of states and to find model parameters λ. They are obtained in unsupervised manner by using the expectation-maximization algorithm (EM) [15].

Principal Component Analysis

PCA is an algorithm that lets to reduce number of dimensions of extracted feature vectors. It basically tries to transform the basis of received vectors into another that would better express given data set. It serves to remove redundancy of information and its output is used as an input for algorithms like previously described GMM. The PCA assumes:

— linearity of processed data,
— orthonormality of vectors of original basis.

PCA is trying to find matrix P that would transform original basis X into Y according to following equation:

$$\mathrm{PX} = \mathrm{Y} \tag{3.16}$$

in such a way that data transformed into new basis would better express searched properties. The measure of quality of obtained new properties is the covariance between them. The covariance equal to 0 indicates no correlation between properties, i.e., good representation of processed properties. Therefore covariance matrix $\mathrm{S_X}$ is extracted of processed data and coefficients of matrix P are picked in such a way that Y would be as close to diagonal matrix as its possible.

It is only necessary to assume specific number of properties to be extracted of given data and this should be chosen carefully according to exact classification problem [16].

Neural networks

Neural network is a concept that basis on this, how human brain works. It consists of neurons and each neuron has several inputs and only one output. Its mathematical model is shown in figure 3.10.

The equation computed by neuron that has N inputs is the following:

$$y = f(b + \sum_{i=1}^{N} x_i \cdot w_i) \tag{3.17}$$

where:

— $f : (-\infty, \infty) \mapsto [-1, 1]$ – activate function,
— b – bias of the neuron,
— w_i – weights of the neuron,

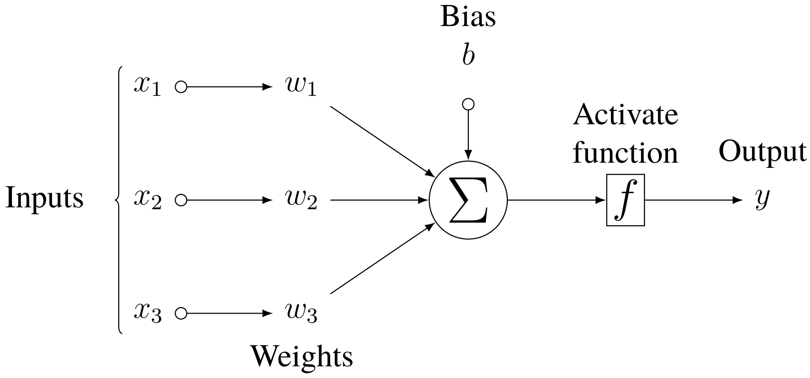
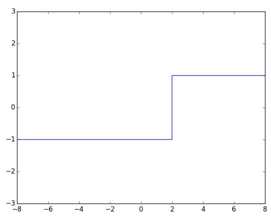
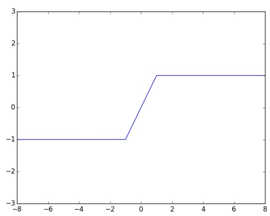


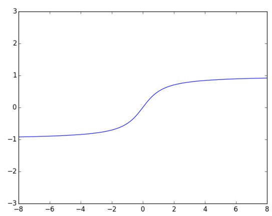
Fig. 3.10: Neuron model [11]



(a) threshold



(b) piecewise linear



(c) scaled arctangent

Fig. 3.11: Example activate functions

— x_i – incoming input.

Usually in single neural network all of the neurons share the same activate function. Typical examples of such functions are shown in figure 3.11.

Neurons are connected in a network according to chosen architecture. The choice of architecture depends on the nature of the problem. In authentication problem the number of inputs could be equal to the quantity of dimensions of feature vector and the output could be only one number (authentication successful or failed). The number and size of hidden layers could be chosen in an experimental way. Example neural network architecture is shown in figure 3.12.

The process of learning neural network is supposed to find such set of neurons coefficients that would force the network to give proper results according to reference data [7].

Support Vector Machine

In SVM the learning data is assumed to consist of pairs (x_i, y_i), where:

— x_i – data point,
— y_i – predicted result of classification (1 if point belongs to specific class and -1 when its not).

SVM is a classification method that is focused on finding such a hyperplane that would hold specific relation with learning points. In general chosen hyperplane would be defined by two vectors w and b and all of its points x would have to hold the following equation:

$$w^T x + b = 0 \tag{3.18}$$

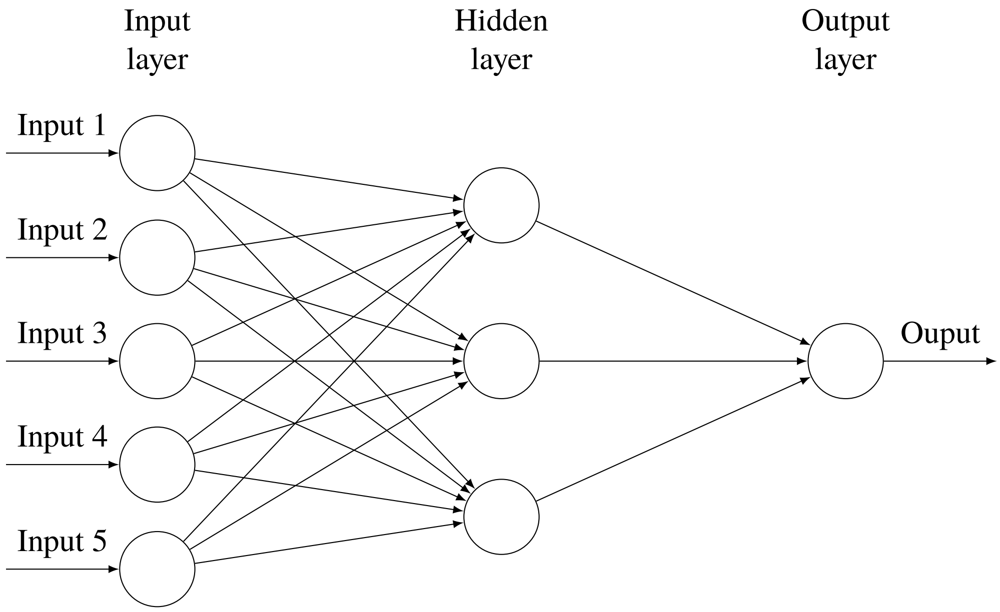


Fig. 3.12: Example neural network architecture [11]

Having these vectors lets to construct the following classifier:

$$h_{w,b}(x) = g(w^T x + b) \tag{3.19}$$

where:

$$g(x) = \begin{cases} 1, x \geq 0 \\ -1, x < 0 \end{cases} \tag{3.20}$$

During learning process (finding vectors w and b) one defines *functional margin* as:

$$\gamma_i = y_i(w^T x_i + b) \tag{3.21}$$

It is worth to notice that γ_i can be interpreted as measure of confidence for specific point, because the greater γ_i is, the more distant of chosen hyperplane point x_i is. Moreover if γ_i is positive, then $h_{w,b}(x_i) = y_i$ – which means that point x_i is classified properly.

Although scaling w and b by the same positive constant would also increase γ_i without any significant correction of chosen parameters. That is why it is comfortable to assume that $||w|| = 1$. In such case γ_i would be a reliable measure of quality of chosen vectors w and b. Since data set consists of N samples, the overall functional margin is defined as:

$$\gamma = \min_{i \in [1,N]} \gamma_i \tag{3.22}$$

It leads to conclusion that SVM algorithm during learning process is trying to determine values of vectors w and b in such a way that they would maximize overall functional margin γ [5].

3.3.2. Synergy of classification techniques

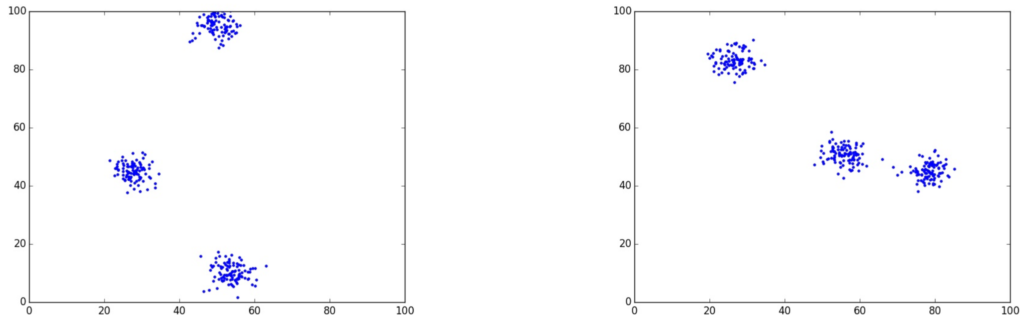
In many real systems several classification techniques are used simultaneously for interpreting the same results and their outputs (like logarithmic probability) are then combined in some way.

The simplest solution of interpreting such probability would be comparing it with some threshold chosen during learning phase. Thanks to having many samples and many techniques, this approach could be extended by using weighted/moving average of probabilities between different classifiers. Results of simple classification algorithm are also shown in section 3.3.3.

3.3.3. Example classification results

Figure 3.13 is showing some random data of 2 classes. The algorithm applied to process this data was GMM (number of states of reference model was set to 3). Two classifier were used:

— one used to fit to first set of data,
— second used to fit to second set of data.



(a) class 1 (b) class 2

Fig. 3.13: Sets of data

After fitting process both sets were applied to first trained classifier. Figure 3.14 shows logarithmic probability of belonging to set for both sets. It can be easily noticed that class 1 reaches much higher score then class 2. It lets to choose a threshold of defining sample as belonging or not belonging to specific class. The threshold chosen in this case was -15. Then it was possible to construct classifier that would distinguish three kinds of samples:

— belonging to class 1,
— belonging to class 2,
— unassigned points.

An algorithm used to distinguish samples is following:

$$\mathcal{A}(x) = \begin{cases} \text{CLASS\_1}, & \text{GMM}_1.\text{score}(x) \geq -15 \\ \text{CLASS\_2}, & \text{GMM}_2.\text{score(x)} \geq -15 \, \wedge \, \text{GMM}_1.\text{score}(x) < -15 \\ UNASSIGNED, & otherwise \end{cases} \tag{3.23}$$

Described algorithm $\mathcal{A}$ was applied to random set of data. Results of used classification are shown in figure 3.15. Where:

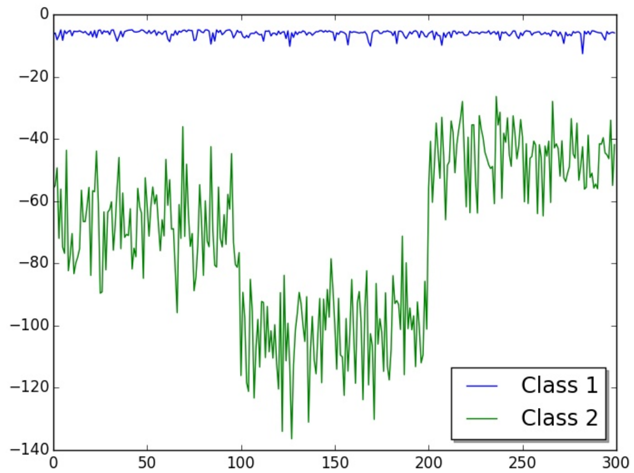


Fig. 3.14: Comparing of scores of datas belonging to two classes

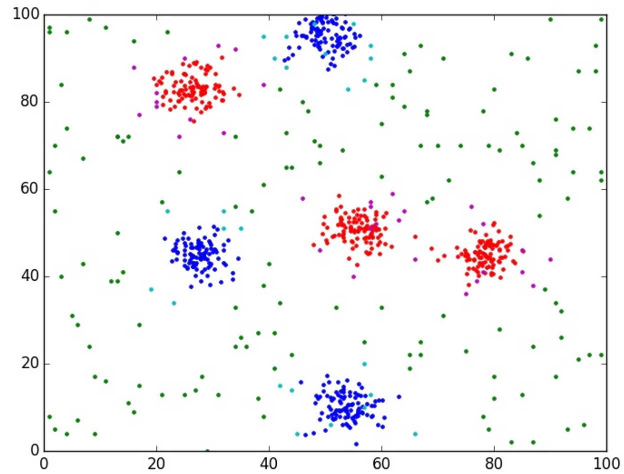


Fig. 3.15: Classified data

- blue dots – samples initially belonging to class 1,
- red dots – samples initially belonging to class 2,
- cyan dots – samples of random set assigned to class 1,
- magenta dots – samples of random set assigned to class 2,
- green dots – unassigned samples.

4. IMPLEMENTATION DESCRIPTION

Voice authorization mechanism described in chapter 3 was implemented in `Python 2.7` language. It generally consists of two logical parts:

— processing – it consists of all of the mathematical operations executed on incoming samples,
— interface – all classes defined in processing could be used separately and imported in different projects, but GUI interface is a simple wrapper that lets to use full functionality of presented library without necessity of writing any code. Since it is not focused on theoretical aspect of speaker recognition it will not be covered in this thesis.

4.1. LIBRARY CONFIGURATION

In order to make configuration of the library easy and interface independent it is mostly done by json file (called `.config` by default). The example `.config` is shown in listing 4.1.1.

Listing 4.1.1: Example `.config` file

```
{
    "detecting_properties": {
        "GMM": true,
        "PCA": true
    },
    "dump_options": {
        "debug_prints": true,
        "dumps_enabled": true,
        "selected_frame": 100
    },
    "parametrization_properties": {
        "decimation_factor": 5,
        "delta_depth": 2,
        "filter_coeff": 0.97,
        "frame_length": 25,
        "frame_step": 10,
        "input_freq": 44100,
        "mel_params": {
            "freq_bounds": [
                15,
                4000
            ],
            "num_coeff": 26
        },
        "steps": {
```

```
                "append_deltas": 1,
                "mean_subtraction": 0,
                "perform_feature_warping": 1,
                "perform_windowing": 1
            }
        },
        "record_options": {
            "duration": "3",
            "frequency": "44100"
        }
}
```

4.2. HELPER CLASSES

4.2.1. Constants

Non processing parameters of the library (like file paths and file patterns) are defined in this class. It lets to easily distinguish algorithmic configuration from environmental one.

4.2.2. Wave processor

Only one method of this class is used by other classes. It is called `extract_raw_data` and its task is to decode `WAV` file into plain array of numbers.

4.2.3. Mel filter bank

This class is supposed to help during mel filtering phase. In its constructor it prepares array of filters. Its only function used publicly after creation of an object is `filter_data` and its body is presented in listing 4.2.1.

Listing 4.2.1: Body of `filter_data` function

```python
def filter_data(self, data):
    res = []
    for mel_filter in self.filters:
    arr = zip(mel_filter, data)
    res.append([mel_coeff * data_el for (mel_coeff, data_el) in arr])
return res
```

It is easy to notice that this function simply gets input array of size N and then filters it with every filter in its bank, resulting with matrix of size $N \times M$ on output, where:

— N – number of data samples,
— M – number of used filters.

4.2.4. Data dumper

This is a class that is used for all of the operations that should load/store data of any kind (except for `WAV` files). It is also used to store samples of processing steps as plots that could help to analyze it. The interface of this class consists of following functions:

- `store_json_data` – writes given data into specified json file
- `load_json_data` – loads and returns data from the specified json file
- `store_binary_data` – writes given data into specified binary file
- `load_binary_data` – loads and returns data from the specified binary file
- `dump_plot` – stores given data as a specified kind of plot. Supported plot modes are the following:
  - `dim_2_plot` – simple plot of one dimensional array,
  - `dim_2_plot_mult` – multiple plot of arbitrary number of arrays given as an argument,
  - `dim_3_plot` – 3 dimensional plot of 2 dimensional array, where 3rd dimension is expressed by color,
  - `dim_2_rect` – bar graph.

4.3. SIGNAL PARAMETRIZER

The full implementation of this class can be found in appendix A. Its only publicly used method is `extract_mfcc` and it will be covered in details in this section.

Listing 4.3.1: Preemphase

```
1  if self.dump_options['dumps_enabled']:
2      DataDumper.dump_plot(data, 'original_signal', 'dim_2_plot')
3
4  if self.log_callback is not None:
5      self.log_callback.set('{:40} pre_emphase'.format(filename))
6
7  data = self.pre_emphase(data)
8
9  if self.dump_options['dumps_enabled']:
10     DataDumper.dump_plot(data, 'preemphased_signal', 'dim_2_plot')
```

Listing 4.3.1 shows the first step of MFCC extraction, which is preemphase. Lines 1-2,9-10 are used for dumping processing results of each stage and lines 4-5 are setting log callback that is used by GUI (in order to inform user of the program about every step of application). These lines are not bounded to extraction algorithm and can be ignored by proper configuration provided in $.config$ file.

Listing 4.3.2: Decimation and dividing into frames

```
if self.log_callback is not None:
self.log_callback.set('{:40} decimate'.format(filename))
data = self.decimate(data)

if self.log_callback is not None:
self.log_callback.set('{:40} divide_into_frames'.format(filename))
frames = self.divide_into_frames(data)
```

Listing 4.3.2 shows another two steps of MFCC extraction which is decimating incoming data and dividing it into frames. Decimation is used for optimization reasons. It lets to significantly reduce calculation complexity without loss of quality of results.

Listing 4.3.3: Frame processing

```
res = []

for ind, frame in enumerate(frames):

    if self.log_callback is not None:
        self.log_callback.set('{:40} processing {} of {} frames'
                             .format(filename, ind, len(frames)))

    frame = self.zero_pad(frame)

    if self.dump_options['dumps_enabled'] and ind == \
       self.dump_options['selected_frame']:
        DataDumper.dump_plot(frame, 'padded_signal', 'dim_2_plot')

    if self.steps['perform_windowing']:
        frame = self.apply_windowing(frame)

    if self.dump_options['dumps_enabled'] and ind == \
       self.dump_options['selected_frame']:
        DataDumper.dump_plot(frame, 'windowed_signal', 'dim_2_plot')

    frame = self.apply_fft(frame)

    if self.dump_options['dumps_enabled'] and ind == \
       self.dump_options['selected_frame']:
        DataDumper.dump_plot(frame, 'after_fft_signal', 'dim_2_plot')

    mel_filter_results = self.mel_filter_bank.filter_data(frame)
    vector = []

    if self.dump_options['dumps_enabled'] and ind == \
    self.dump_options['selected_frame']:
        DataDumper.dump_plot(mel_filter_results,
                             'mel_filtered_signal',
                             'dim_3_plot')

    for mel_filter_result in mel_filter_results:
        power = self.calculate_signal_power(mel_filter_result)
        vector.append(math.log(power))

    if self.dump_options['dumps_enabled'] and ind == \
       self.dump_options['selected_frame']:
        DataDumper.dump_plot(vector, 'mel_power_signal', 'dim_2_rect')

```

```
45      vector = self.discrete_cosine_transform(vector)
46
47      if self.dump_options['dumps_enabled'] and ind == \
48         self.dump_options['selected_frame']:
49          DataDumper.dump_plot(vector,
50                               'cosine_transformed_signal',
51                               'dim_2_rect')
52
53      vector = self.liftering(vector)
54
55      if self.dump_options['dumps_enabled'] and ind == \
56         self.dump_options['selected_frame']:
57          DataDumper.dump_plot(vector, 'liftered_signal', 'dim_2_rect')
58
59      res.append(vector)
60
61  if self.dump_options['dumps_enabled']:
62      DataDumper.dump_plot(res, 'parametrized_raw', 'dim_3_plot')
```

Listing 4.3.3 is showing steps executed on each processed frame. These are:

— zero padding (line 9),
— windowing (line 16),
— applying FFT (line 22),
— mel filtering (line 28),
— calculating power for each (lines 37-39),
— calculating DCT of obtained powers (line 45),
— liftering (line 53),

After execution this part of code, the variable res holds a matrix of MFCC vectors extracted from each data frame.

Listing 4.3.4: Vectors enhancements

```
if self.steps['append_deltas']:
    if self.log_callback is not None:
        self.log_callback.set('{:40} appending deltas'.format(filename))

    res = self.append_deltas_and_deltasdeltas(res)

    if self.dump_options['dumps_enabled']:
        DataDumper.dump_plot(res, 'parametrized_with_delta', 'dim_3_plot')

if self.steps['perform_feature_warping']:
    if self.log_callback is not None:
        self.log_callback.set('{:40} feature warping'.format(filename))

    res = self.feature_warp(res)

```

```
16     if self.dump_options['dumps_enabled']:
17         DataDumper.dump_plot(res, 'feature_warped', 'dim_3_plot')
18
19 if self.steps['mean_subtraction']:
20     if self.log_callback is not None:
21         self.log_callback.set('{:40} cepstral mean subtraction'
22             .format(filename))
23
24     res = self.cepstral_mean_subtraction(res)
25
26     if self.dump_options['dumps_enabled']:
27         DataDumper.dump_plot(res,
28                              'cepstral_mean_subtraction',
29                              'dim_3_plot')
30
31 return res
```

Listing 4.3.4 shows final processing of feature vectors which are:

— appending deltas and deltasdeltas (line 5),
— feature warping (line 14),
— cepstral mean subtraction (line 24).

The last line of this listing returns result, which are enhanced MFCC vectors grouped in array.

4.4. CLASSIFIER

The whole code of Classifier class can be found in appendix B. This class is a container of classification algorithms.

Listing 4.4.1: Classifier constructor

```
def __init__(self,
             detect_properties,
             dump_options,
             name,
             log_callback = None):

    self.dump_options = dump_options
    self.log_callback = log_callback
    self.name = name

    self.algorithms = []

    if detect_properties['PCA']:
        self.algorithms.append(PCA(log_callback,
                                   dump_options,
                                   letter))
```

```

    if detect_properties['GMM']:
        self.algorithms.append(GMM(log_callback,
                                   dump_options,
                                   letter))
```

During its initialization it detects which algorithms where selected for learning process. Currently GMM and PCA are supported.

Listing 4.4.2: Loading and storing Classifier

```
def store_to_file(self):

    for algorithm in self.algorithms:
        algorithm.log_callback = None

    DataDumper.store_binary_data(self.algorithms,
                                 Constants.MODEL_PATH + '_' + self.name)

def load_from_data(self):

    self.algorithms = \
        DataDumper.load_binary_data(Constants.MODEL_PATH + '_' + self.name)

    for algorithm in self.algorithms:
        algorithm.log_callback = self.log_callback
```

Listing 4.4.2 shows two methods that are storing/loading Classifier to/from binary data. This functionality lets to reuse once learned Classifier for further purposes.

Listing 4.4.3: Fit and decision method

```
def fit(self, data):

    for algorithm in self.algorithms:
        algorithm.fit(data)

    if self.log_callback is not None:
        self.log_callback.set('idle')

def decision(self, datas, labels):

    min_len = min([len(el) for el in datas])
    datas = [el[:min_len] for el in datas]

    arr = zip(labels, datas)

```

```
    for algorithm in self.algorithms:
        decisions = []

        for label, data in arr:
            if self.dump_options['debug_prints']:
                print 'processing {}'.format(label)
            decisions.append(algorithm.decision(data))

        DataDumper.dump_plot(decisions,
                             'score of {}'.format(algorithm.name()),
                                                 'dim_2_plot_mult',
                                                 labels)

    if self.log_callback is not None:
        self.log_callback.set('idle')
```

`Fit` function simply iterates over all of the attached algorithms. However `decision` function takes two arguments:

- `datas` – it is an array of streams of data, where single stream is considered to be one sentence of a speaker,
- `labels` – it is an array of labels that are supposed to identify each processed sentence.

This function firstly equalizes length of all of the data sets and then executes following steps for each algorithm:

- prepares empty decision array
- iterates over all of the sentences in `datas` array:
  - runs `decision` function of current algorithm on current sentence
  - puts result on the end of decisions array
- plots all of decisions in single plot

Having such plots lets to compare accuracy of each sentence to reference model. It does not return explicit answer on the question if the sentence was said by specific user, but lets to estimate probability of such event and construct custom way of interpreting results.

5. SIMULATION RESULTS

In order to verify the correctness of proposed algorithm, test data from 3 speakers was collected (speaker 1 and 3 were male, and 2nd was female). The first speaker provided long (~ 25 seconds long) reference data and all of the speakers provided 2 short pieces (~ 5 seconds long) challenges. The main goal of simulation was to prove that speaker verification could be treated as text independent.

5.1. PARAMETRIZATION

5.1.1. Operations performed on whole signal

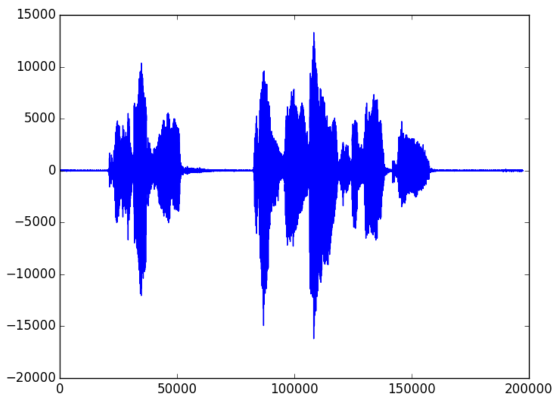


Fig. 5.1: Original signal

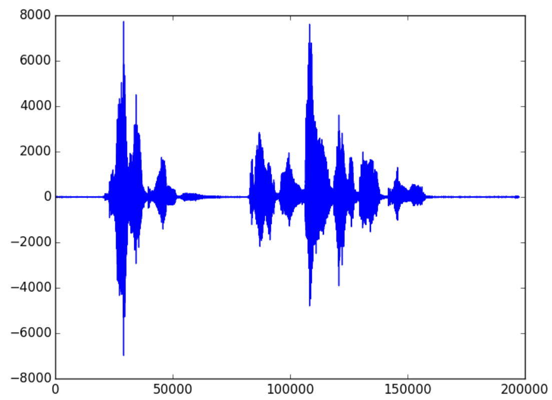


Fig. 5.2: Preemphased signal

5.1.2. Operations performed on each frame

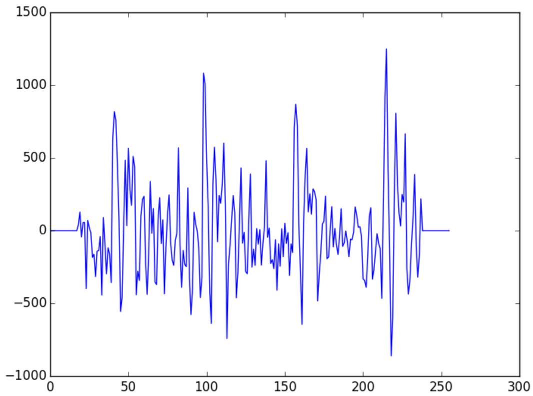


Fig. 5.3: Padded signal

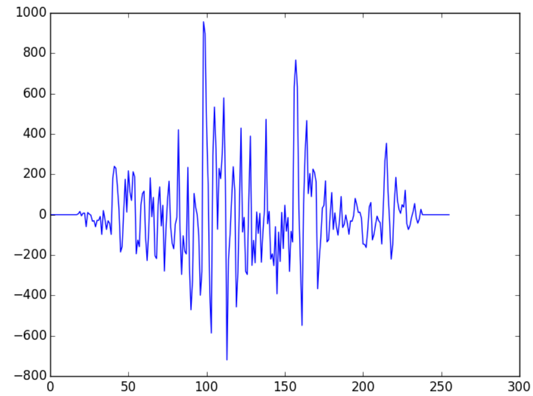


Fig. 5.4: Windowed signal

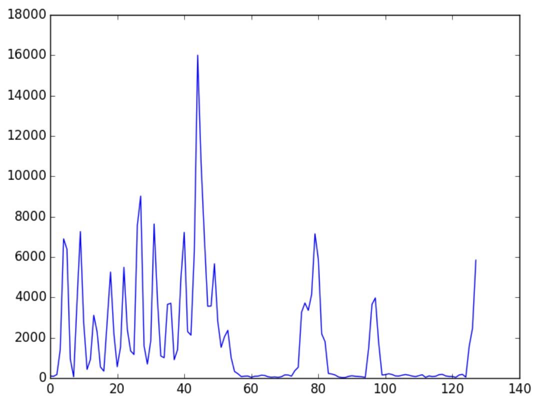


Fig. 5.5: Signal after application of FFT

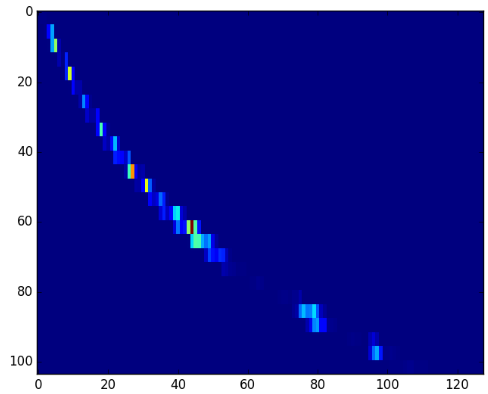


Fig. 5.6: Frame after mel filtering

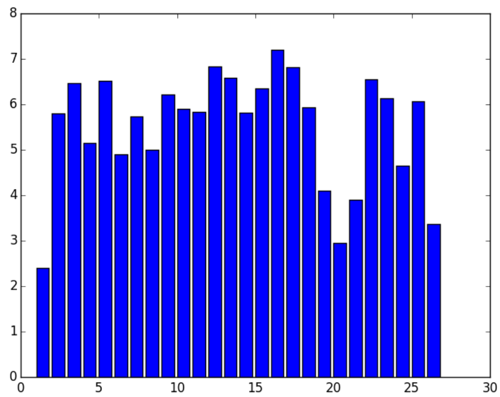


Fig. 5.7: Logarithmic power of each mel filter result

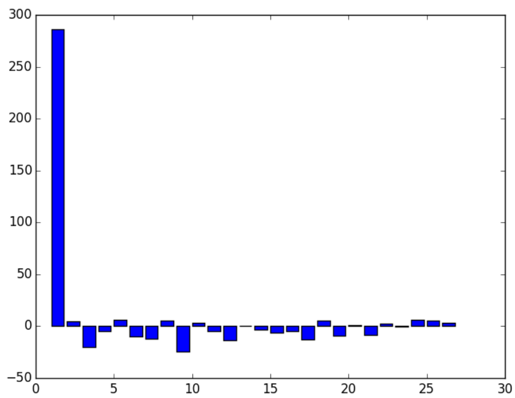


Fig. 5.8: Frame after cosine transormation

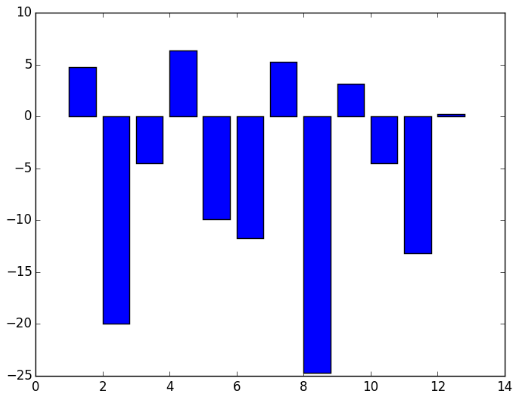


Fig. 5.9: Frame after application of liftering

5.1.3. Full matrix dumps

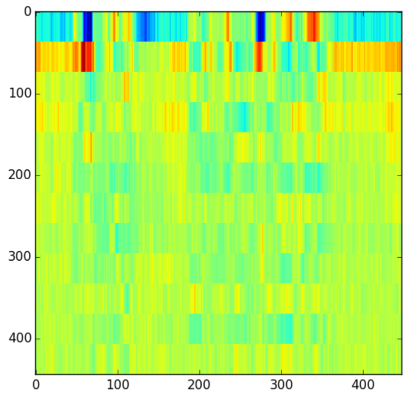


Fig. 5.10: Whole frame parametrization

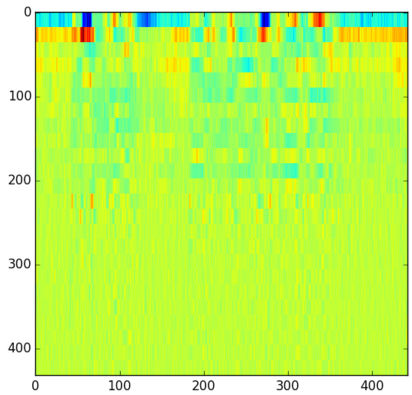


Fig. 5.11: Frame parametrization with deltas and deltasdeltas applied

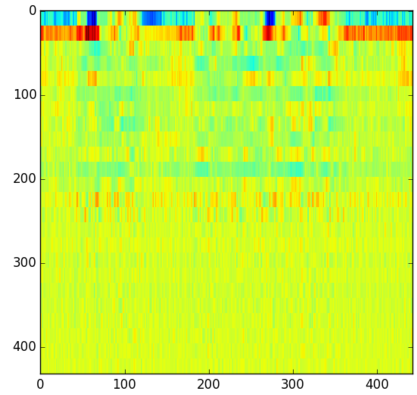


Fig. 5.12: Frame parametrization after feature warping

5.2. ADJUSTING

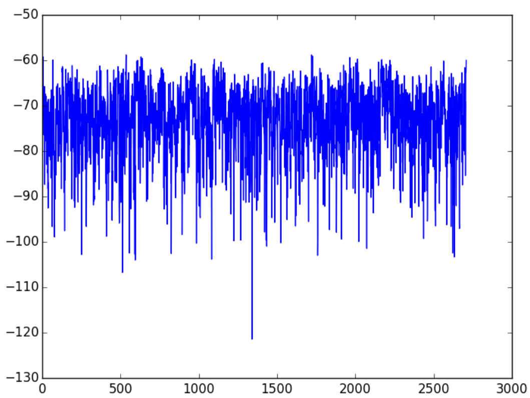


Fig. 5.13: Decision output of PCA for reference data

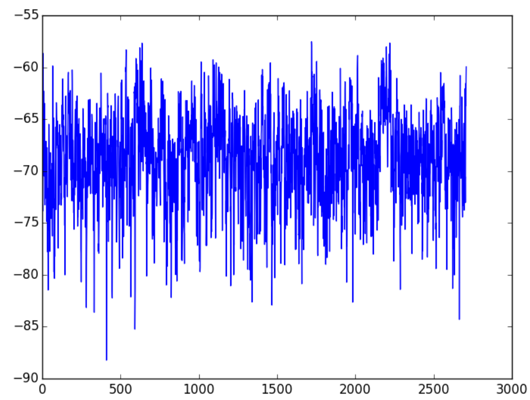


Fig. 5.14: Decision output of GMM for reference data

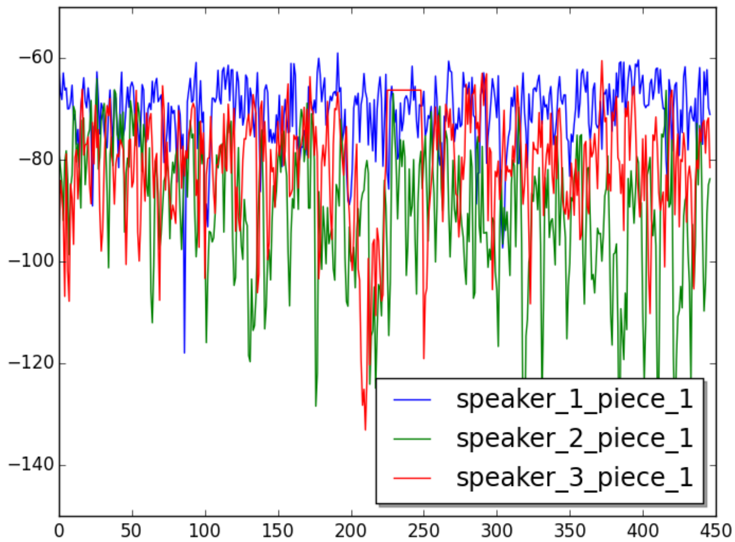


Fig. 5.15: Score comparison of PCA for piece 1

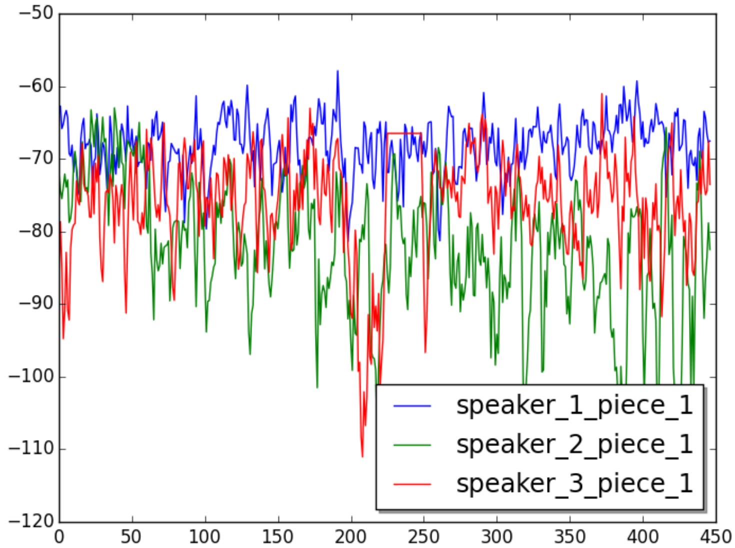


Fig. 5.16: Score comparison of GMM for piece 1

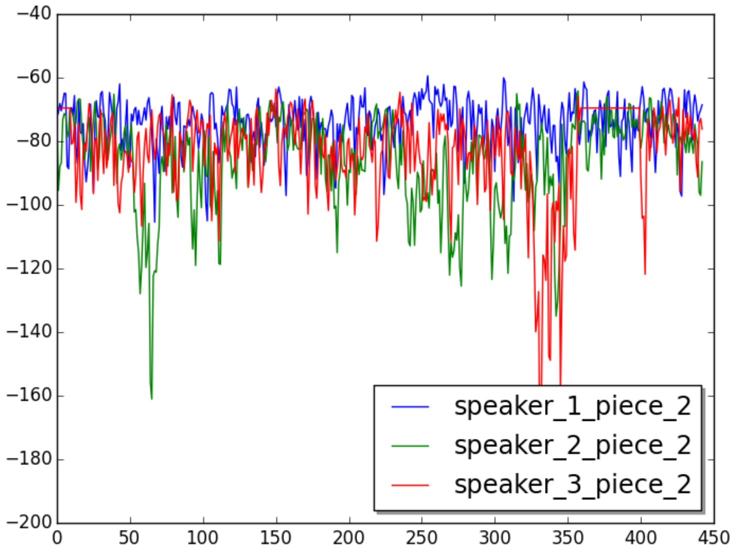


Fig. 5.17: Score comparison of PCA for piece 2

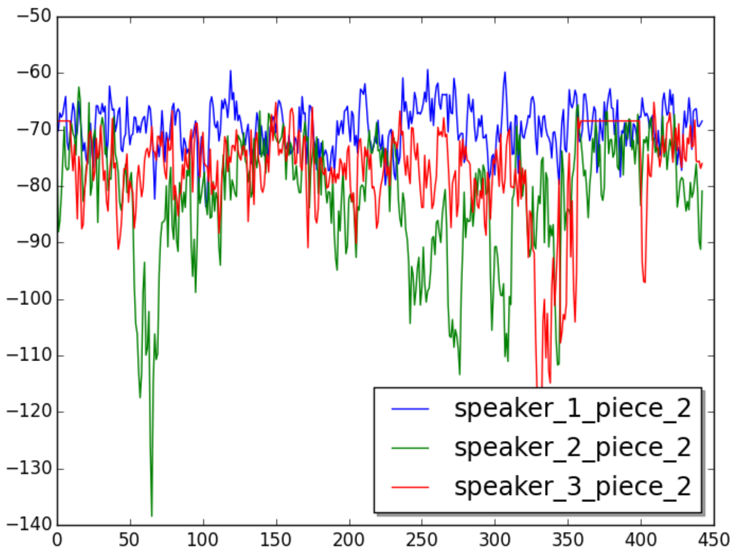


Fig. 5.18: Score comparison of GMM for piece 2

In figures 5.15-5.18 it can be easily seen that speaker 1 reaches always the highest score. Moreover speaker 3 gets higher score then speaker 2 since 1 and 3 are both male. These relations can be also seen in console output, where average score for each case is given:

```
PCA ref score is:  -73.5257023843
GMM ref score is:  -68.9503100157

# PIECE 1
# PCA
```

```
processing speaker_1_piece_1
PCA score is:  -70.7049469961
processing speaker_2_piece_1
PCA score is:  -89.9448093412
processing speaker_3_piece_1
PCA score is:  -81.1213851514

# GMM

processing speaker_1_piece_1
GMM score is:  -68.2533621405
processing speaker_2_piece_1
GMM score is:  -82.7294472701
processing speaker_3_piece_1
GMM score is:  -75.8071632791

# PIECE 2
# PCA

processing speaker_1_piece_2
PCA score is:  -74.2494058079
processing speaker_2_piece_2
PCA score is:  -85.7860600889
processing speaker_3_piece_2
PCA score is:  -82.8986110641

# GMM

processing speaker_1_piece_2
GMM score is:  -69.5582396506
processing speaker_2_piece_2
GMM score is:  -82.7054689999
processing speaker_3_piece_2
GMM score is:  -77.5987134361
```

6. CONCLUSION

As it was shown section 5.3 the presented method gives satisfying results. However much more tests should be executed in order to assure proper reliability of system constructed on this library.

It is also worth mentioning that all of applied classification methods in this thesis where basing on only one class of input samples. Methods like SVM or artificial neural networks require samples that belong and not belong to desired set. It basically means that collecting large database of audio samples lets to improve accuracy of these detections. Therefore the mostly guarded part of commercial systems used for speaker recognition are not just algorithms, but huge database collected over years of building such system.

Another step of developing this library could be collecting large test data set and try to adjust all of the parameters in order to provide optimal results. It could be also worth considering to add some new classification algorithms and invent some probabilities combining function. After having this library tested and considered as reliable one could try to implement all of these algorithms on real digital signal processor or microcontroller in order to construct some commercial device used for voice authorization. One could simply rewrite this library to some low level programming language that would compute all of these operations much faster, then this theoretical model.

Appendices

A. SOURCE CODE OF SIGNALPARAMETRIZER CLASS

```
import sys
import math
from numpy.fft import fft
from scipy.fftpack import dct
from decimal import Decimal

from src.processing.datadumper import DataDumper
from src.processing.melfilterbank import MelFilterBank


class SignalParametrizer:

    @classmethod
    def default_processing_parameters(cls):
        parameters = {}

        parameters['mel_params'] = {}
        parameters['mel_params']['freq_bounds']   = (15, 4000) # [Hz]
        parameters['mel_params']['num_coeff']     = 26
        parameters['filter_coeff']                = 0.97
        parameters['input_freq']                  = 44100      # [Hz]
        parameters['decimation_factor']           = 5
        parameters['frame_length']                = 25         # [ms]
        parameters['frame_step']                  = 10         # [ms]
        parameters['delta_depth']                 = 2

        parameters['steps'] = {
                        'perform_windowing'        : True,
                        'append_deltas'            : True,
                        'perform_feature_warping' : True,
                        'mean_subtraction'         : True
                    },

        return parameters

    @classmethod
    def dump_none_parameters(cls):
        parameteres = {
        'selected_frame'            : 0,
        'dumps_enabled'             : False
        }
```

```python

        return parameteres

    @classmethod
    def dump_all_parameters(cls):
        parameteres = {
        'selected_frame'            : 83,
        'dumps_enabled'             : True
        }

        return parameteres

    def __init__(self, parameters, dump_options, log_callback=None):

        self.filter_coeff      = parameters['filter_coeff']
        self.input_freq        = parameters['input_freq']
        self.decimation_factor = parameters['decimation_factor']
        self.frame_length      = parameters['frame_length']
        self.frame_step        = parameters['frame_step']
        self.delta_depth       = parameters['delta_depth']
        self.steps             = parameters['steps']

        self.log_callback      = log_callback
        self.dump_options      = dump_options

        self.mel_filter_bank = \
            MelFilterBank(parameters['mel_params'],
                          self.estimate_frame_length(),
                          self.input_freq / self.decimation_factor,
                          self.dump_options['dumps_enabled'])

    def extract_mfcc(self, data, filename):

        if self.dump_options['dumps_enabled']:
            DataDumper.dump_plot(data,
                                 'original_signal',
                                 'dim_2_plot')

        if self.log_callback is not None:
            self.log_callback.set('{:40} pre_emphase'
                                  .format(filename))
        data = self.pre_emphase(data)

        if self.dump_options['dumps_enabled']:
            DataDumper.dump_plot(data,
                                 'preemphased_signal',
                                 'dim_2_plot')

```

```
        if self.log_callback is not None:
            self.log_callback.set('{:40} decimate'
                                  .format(filename))
        data = self.decimate(data)

        if self.log_callback is not None:
            self.log_callback.set('{:40} divide_into_frames'
                                  .format(filename))
        frames = self.divide_into_frames(data)

        res = []

        for ind, frame in enumerate(frames):

            if self.log_callback is not None:
                self.log_callback.set('{:40} processing {} of {} frames'
                                      .format(filename, ind+1, len(frames)))
            frame = self.zero_pad(frame)

            if self.dump_options['dumps_enabled'] and ind == \
                    self.dump_options['selected_frame']:
                DataDumper.dump_plot(frame,
                                     'padded_signal',
                                     'dim_2_plot')

            if self.steps['perform_windowing']:
                frame = self.apply_windowing(frame)

            if self.dump_options['dumps_enabled'] and ind == \
                    self.dump_options['selected_frame']:
                DataDumper.dump_plot(frame,
                                     'windowed_signal',
                                     'dim_2_plot')

            frame = self.apply_fft(frame)

            if self.dump_options['dumps_enabled'] and ind == \
                    self.dump_options['selected_frame']:
                DataDumper.dump_plot(frame,
                                     'after_fft_signal',
                                     'dim_2_plot')

            mel_filter_results = self.mel_filter_bank.filter_data(frame)
            vector = []

            if self.dump_options['dumps_enabled'] and ind == \
                    self.dump_options['selected_frame']:
                DataDumper.dump_plot(mel_filter_results,
```

```
                                     'mel_filtered_signal',
                                     'dim_3_plot')

            for mel_filter_result in mel_filter_results:
                power = self.calculate_signal_power(mel_filter_result)
                vector.append(math.log(power))

            if self.dump_options['dumps_enabled'] and ind == \
                    self.dump_options['selected_frame']:
                DataDumper.dump_plot(vector,
                                     'mel_power_signal',
                                     'dim_2_rect')

            vector = self.discrete_cosine_transform(vector)

            if self.dump_options['dumps_enabled'] and ind == \
                    self.dump_options['selected_frame']:
                DataDumper.dump_plot(vector,
                                     'cosine_transformed_signal',
                                     'dim_2_rect')

            vector = self.liftering(vector)

            if self.dump_options['dumps_enabled'] and ind == \
                    self.dump_options['selected_frame']:
                DataDumper.dump_plot(vector,
                                     'liftered_signal',
                                     'dim_2_rect')

            res.append(vector)

        if self.dump_options['dumps_enabled']:
            DataDumper.dump_plot(res,
                                 'parametrized_raw',
                                 'dim_3_plot')

        if self.steps['append_deltas']:
            if self.log_callback is not None:
                self.log_callback.set('{:40} appending deltas'
                                      .format(filename))
            res = self.append_deltas_and_deltasdeltas(res)

            if self.dump_options['dumps_enabled']:
                DataDumper.dump_plot(res,
                                     'parametrized_with_delta',
                                     'dim_3_plot')

        if self.steps['perform_feature_warping']:
```

```
            if self.log_callback is not None:
                self.log_callback.set('{:40} feature warping'
                                      .format(filename))

            res = self.feature_warp(res, filename)
            if self.dump_options['dumps_enabled']:
                DataDumper.dump_plot(res,
                                     'feature_warped',
                                     'dim_3_plot')

        if self.steps['mean_subtraction']:
            if self.log_callback is not None:
                self.log_callback.set('{:40} cepstral mean subtraction'
                                      .format(filename))

            res = self.cepstral_mean_subtraction(res)
            if self.dump_options['dumps_enabled']:
                DataDumper.dump_plot(res, 'cepstral_mean_subtraction',
                                     'dim_3_plot')

        return res

    def estimate_frame_length(self):
        freq = self.input_freq / self.decimation_factor
        dt = 1./freq
        frame_len  = int(math.floor(self.frame_length / (dt * 1000)))
        power = 1

        while power < frame_len:
            power *= 2

        return power

    def pre_emphase(self, data):
        result = []

        for ind, el in enumerate(data[1:]):
            result.append(el - self.filter_coeff * data[ind-1])

        return result

    def decimate(self, data):
        return data[::self.decimation_factor]

    def divide_into_frames(self, data):
        freq = self.input_freq / self.decimation_factor
        dt = 1./freq
        frame_len  = int(math.floor(self.frame_length / (dt * 1000)))
```

```
        frame_step = int(math.floor(self.frame_step   / (dt * 1000)))

        ind = 0
        res = []

        while (ind + frame_len) < len(data):
            res.append(data[ind:ind+frame_len])
            ind += frame_step

        return res

    def zero_pad(self, frame):
        power = 1

        while power < len(frame):
            power *= 2

        diff = power - len(frame)
        left = int(math.floor(diff/2))
        right = diff - left

        left_pad  = [0 for i in range(left)]
        right_pad = [0 for i in range(right)]

        return left_pad + frame + right_pad

    def apply_windowing(self, frame):
        res = []
        N = len(frame)

        for n, el in enumerate(frame):
            coeff = 0.53836 - 0.46164 * math.cos(2 * math.pi * n / (N-1))
            res.append(coeff * el)

        return res

    def apply_fft(self, frame):
        res = fft(frame)
        return res[:len(res)/2]

    def calculate_signal_power(self, mel_filter_result):

        res = sys.float_info.epsilon

        for el in mel_filter_result:
            res += abs(el)**2

        res /= len(mel_filter_result)
```

```python
        return res ** 0.5

    def discrete_cosine_transform(self, vector):
        return dct(vector)

    def liftering(self, vector):
        return vector[1:13]

    def cepstral_mean_subtraction(self, arg):
        num_coeffs = len(arg[0])
        means = []

        for i in range(num_coeffs):
            res = 0
            for el in arg:
                res += el[i]
            means.append(res / len(arg))

        for vector in arg:
            for i in range(num_coeffs):
                vector[i] -= means[i]

        return arg

    def append_deltas_and_deltasdeltas(self, arg):

        divider = 2. * sum(n**2 for n in range(self.delta_depth))

        res = []

        for t in range(self.delta_depth, len(arg)-self.delta_depth):
            vector = [0 for i in range(len(arg[0]))]

            for n in range(self.delta_depth):
                for k in range(len(arg[0])):
                    val = n * (arg[t+n][k] - arg[t-n][k]) / divider
                    vector[k] += val

            res.append(list(arg[t]) + list(vector))

        return res

    def feature_warp(self, arg, filename=None):

        epsilon = Decimal(10) ** -2

        def warp_single_row(signal):

```

```
def get_histogram(arr):
    max_arr = max(arr)
    min_arr = min(arr)

    histogram = [0] * (int((max_arr - min_arr) /
                        float(epsilon))+1)

    for el in arr:
        index = (int((el - min_arr) / float(epsilon)))
        histogram[index] += 1

    return histogram

def histogram_to_cdf(histogram):
    cdf = []
    for i in range(len(histogram)):
        cdf.append(sum(histogram[:(i+1)]))

    all_samples = float(sum(histogram))
    cdf = [float(Decimal(el / all_samples)
                .quantize(epsilon))
           for el in cdf]

    return cdf

def prepare_norm_cdf(sigma, mu):

    def normcdf(x, arg_mu, arg_sigma):

        def erfcc(x):

            z = abs(x)
            t = 1. / (1. + 0.5*z)
            r = t * math.exp(-z*z-1.26551223+
                             t*(1.00002368+t*(.37409196+
                t*(.09678418+t*(-.18628806+t*(.27886807+
                t*(-1.13520398+t*(1.48851587+t*(-.82215223+
                t*.17087277)))))))))
            if (x >= 0.):
                return r
            else:
                return 2. - r

        t = x-arg_mu
        y = 0.5*erfcc(-t/(arg_sigma*math.sqrt(2.0)))
        if y>1.0:
            y = 1.0
        return float(Decimal(y).quantize(epsilon))
```

```
        return [0] + \
               [normcdf(mu - 3*sigma +
                        float(epsilon) * idx, mu, sigma)
                      for idx in range(int(6 * sigma /
                                              float(epsilon)))] + [1]

    def first_greater(arr, arg):
        for idx in range(len(arr)):
            if arg <= arr[idx]:
                return idx

        return len(arg)-1

    def translate_signal(signal, values):
        result = []

        for el in signal:
            found = False
            for (old, new) in values:
                if (old - float(epsilon)) <= \
                        el <= \
                        (old + float(epsilon)):
                    result.append(new)
                    found = True
                    break

            if not found:
                result.append(el)

        return result

    max_sig = max(signal)
    min_sig = min(signal)

    sigma = (max_sig - min_sig) / 6.
    mu    = (min_sig + max_sig) / 2.

    sig_hist = get_histogram(signal)
    sig_cdf  = histogram_to_cdf(sig_hist)
    norm_cdf = prepare_norm_cdf(sigma, mu)

    indices = [first_greater(norm_cdf, el) for el in sig_cdf]
    values = [(min_sig + idx * float(epsilon), min_sig +
               indices[idx] * float(epsilon))
              for idx in range(len(indices))]
    return translate_signal(signal, values)
```

```
        num_coeffs  = len(arg[0])
        num_vectors = len(arg)

        res = []

        for idx in range(num_vectors):
            res.append([0] * num_coeffs)

        for idx in range(num_coeffs):

            if self.log_callback is not None:
                self.log_callback.set('{:40} warping {} of {} coefficients'
                                      .format(filename, idx+1, num_coeffs))

            row = [el[idx] for el in arg]
            row = warp_single_row(row)

            for row_idx, el in enumerate(row):
                res[row_idx][idx] = el

        return res
```

B. SOURCE CODE OF CLASSIFIER CLASS

```
from sklearn import mixture
from sklearn import decomposition
import numpy

from src.processing.datadumper import DataDumper
from src.Constants import Constants


class Algorithm(object):

    def __init__(self, log_callback, dump_options, letter):
        self.log_callback = log_callback
        self.dump_options = dump_options
        self.letter = letter

    def fit(self, data):
        pass

    def decision(self, data):
        pass

    def name(self):
        return ''

    def log_line(self, text):
        if self.log_callback != None:
            self.log_callback.set(self.letter +
                                  '_' + self.name() +
                                  '_' + text)


class GMM(Algorithm):

    def __init__(self, log_callback, dump_options, letter):
        Algorithm.__init__(self, log_callback, dump_options, letter)

    def fit(self, data):

        self.gmm = mixture.GMM(n_components=16)

        self.log_line('fitting GMM')
```

```

        self.gmm.fit(data)

        gmm_res = self.gmm.score(data)

        if self.dump_options['debug_prints']:
            print 'GMM ref score is:  {}'\
                .format(sum(gmm_res) / float(len(gmm_res)))

        if self.dump_options['dumps_enabled']:
            DataDumper.dump_plot(gmm_res, self.name() +
                                 '_decision_output',
                                 'dim_2_plot')

    def decision(self, data):

        gmm_res = self.gmm.score(data)

        if self.dump_options['dumps_enabled']:
            DataDumper.dump_plot(gmm_res, self.name() +
                                 '_decision_output',
                                 'dim_2_plot')

        if self.dump_options['debug_prints']:
            print 'GMM score is:  {}'\
                .format(sum(gmm_res) / float(len(gmm_res)))

        return gmm_res

    def name(self):
        return 'GMM'


class PCA(Algorithm):

    def __init__(self, log_callback, dump_options, letter):
        Algorithm.__init__(self, log_callback, dump_options, letter)

    def fit(self, data):

        self.pca = decomposition.ProbabilisticPCA(n_components=16)

        self.log_line('fitting PCA')

        self.pca.fit(numpy.asarray(data))

        pca_res = self.pca.score(numpy.asarray(data))

```

```
        if self.dump_options['debug_prints']:
            print 'PCA ref score is:  {}'\
                .format(sum(pca_res) / float(len(pca_res)))

        if self.dump_options['dumps_enabled']:
            DataDumper.dump_plot(pca_res, self.name() +
                                 '_decision_output',
                                 'dim_2_plot')

    def decision(self, data):

        pca_res = self.pca.score(numpy.asarray(data))

        if self.dump_options['dumps_enabled']:
            DataDumper.dump_plot(pca_res, self.name() +
                                 '_decision_output',
                                 'dim_2_plot')

        if self.dump_options['debug_prints']:
            print 'PCA score is:  {}'\
                .format(sum(pca_res) / float(len(pca_res)))

        return pca_res

    def name(self):
        return 'PCA'


class Classifier:

    def __init__(self,
                 detect_properties,
                 dump_options,
                 name,
                 log_callback = None):

        self.dump_options = dump_options
        self.log_callback = log_callback
        self.name = name

        self.algorithms = []

        if detect_properties['PCA']:
            self.algorithms.append(PCA(log_callback,
                                       dump_options,
                                       name))

        if detect_properties['GMM']:
```

```python
            self.algorithms.append(GMM(log_callback,
                                       dump_options,
                                       name))

    def fit(self, data):

        for algorithm in self.algorithms:
            algorithm.fit(data)

        if self.log_callback is not None:
            self.log_callback.set('idle')

    def decision(self, datas, labels):

        min_len = min([len(el) for el in datas])
        datas = [el[:min_len] for el in datas]

        arr = zip(labels, datas)

        for algorithm in self.algorithms:
            decisions = []

            for label, data in arr:
                if self.dump_options['debug_prints']:
                    print 'processing {}'.format(label)
                decisions.append(algorithm.decision(data))
            DataDumper.dump_plot(decisions,
                                 'score of {}'.format(algorithm.name()),
                                                      'dim_2_plot_mult',
                                                      labels)

        if self.log_callback is not None:
            self.log_callback.set('idle')

    def store_to_file(self):

        for algorithm in self.algorithms:
            algorithm.log_callback = None

        DataDumper.store_binary_data(self.algorithms,
                              Constants.MODEL_PATH + '_' + self.name)

    def load_from_data(self):

        self.algorithms = \
            DataDumper.load_binary_data(Constants.MODEL_PATH + '_' + self.name)
```

```
        for algorithm in self.algorithms:
            algorithm.log_callback = self.log_callback
```

BIBLIOGRAPHY

[1] *https://en.wikipedia.org/wiki/aliasing*. 2016.

[2] Achint Oommen Thomas, J.L., *Text independent speaker identification*, CSE 666 Term Project Presentation.

[3] Alex Smola, S.V., *Introduction to machine learning* (Departments of Statistics and Computer Science Purdue University, 2010).

[4] Anand Vardhan Bhalla, S.K., *Performance improvement of speaker recognition system*, International Journal of Advanced Research in Computer Science and Software Engineering. March 2012.

[5] BURGES, C.J., *A tutorial on support vector machines for pattern recognition*, Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

[6] Group, V.P., *Introduction to digital signal processing*, Micro-Measurements. 2010.

[7] Jain, A.K., *Artificial networks: A tutorial*, Michigan State University. 1996.

[8] Jason Pelecanos, S.S., *Feature warping for robust speaker verification*, A Speaker Odyssey. 2001.

[9] Joseph P. Campbell, J., *Speaker Recognition* (Department of Defense Fort Meade, MD).

[10] Julien Epps, E.A., *Speech characterization and feature extraction for speaker recognition*, School of Electrical Engineering and Telecommunications University of New South Wales, Australia. 2011.

[11] Medina, G., *Latex stack exchange*, `http://tex.stackexchange.com/questions/132444/diagram-of-an-artificial-neural-network`. Accessed: 2016-04-09.

[12] Meseguer, N.A., *Speech Analysis for Automatic Speech Recognition* (Norwegian University of Science and Technology Department of Electronics and Telecommunications, 2009).

[13] Orfanidis, S.J., *Introduction to Signal Processing* (Rutgers University, 2010).

[14] Reynolds, D.A., *An overview of automatic speaker recognition technology*, MIT Lincoln Laboratory, Lexington, MA USA.

[15] Reynolds, D.A., *Automatic speaker recognition using gaussian mixture speaker models*, THE LINCOLN LABORATORY JOURNAL. 1995.

[16] Shlens, J., *A tutorial on principal component analysis*, Derivation, Discussion and Singular Value Decomposition. 2003.

[17] Westphal, M., *The use of cepstral means in conversational speech recognition*, Interactive Systems Laboratories - University of Karlsruhe.

[18] Yang, T., *The Algorithms of Speech Recognition, Programming and Simulating in MATLAB* (2012).