Modular Audio Recognition Framework and Text-Independent Speaker Identification v.0.2.0

The MARF Development Group

Represented by:

Ian Clément
Serguei A. Mokhov
Dimitrios Nicolacopoulos
Stephen Sinclair

Montréal, Québec, Canada

Mon Feb 10 05:06:31 EST 2003

Contents

0.1 Intro . . . 6
0.1.1 What is MARF? . . . 6
0.1.2 Authors . . . 6
0.1.3 Purpose . . . 6
0.1.4 Project Source and Location . . . 6
0.1.5 Why Java? . . . 7
0.2 MARF Architecture . . . 8
0.2.1 Application Point of View . . . 8
0.2.2 Packages and Physical Layout . . . 8
0.2.3 Current Limitations . . . 13
0.3 Methodology . . . 14
0.3.1 Storage . . . 14
0.3.1.1 Speaker Database . . . 14
0.3.1.2 Storing Features, Training, and Classification Data . . . 14
0.3.1.3 File Location . . . 16
0.3.1.4 Sample and Feature Sizes . . . 16
0.3.1.5 Parameter Passing . . . 16
0.3.1.6 Result . . . 16
0.3.1.7 Sample Format . . . 17
0.3.1.8 Sample Loading Process . . . 17
0.3.2 Preprocessing . . . 20
0.3.2.1 Normalization . . . 20
0.3.2.2 FFT Filter . . . 20
0.3.2.3 Low-Pass Filter . . . 23
0.3.2.4 High-Pass Filter . . . 23

0.3.2.5 Band-Pass Filter . . . 23
0.3.2.6 High Frequency Boost . . . 23
0.3.2.7 Noise Removal . . . 25
0.3.3 Feature Extraction . . . 26
0.3.3.1 The Hamming Window . . . 26
0.3.3.2 Fast Fourier Transform (FFT) . . . 26
0.3.3.3 Linear Predictive Coding (LPC) . . . 28
0.3.3.4 Random Feature Extraction . . . 30
0.3.4 Classification . . . 31
0.3.4.1 Chebyshev Distance . . . 31
0.3.4.2 Euclidean Distance . . . 31
0.3.4.3 Minkowski Distance . . . 33
0.3.4.4 Mahalanobis Distance . . . 33
0.3.4.5 Artificial Neural Network . . . 33
0.3.4.6 Random Classification . . . 35
0.4 GUI . . . 36
0.4.1 Spectrogram . . . 36
0.4.2 Wave Grapher . . . 36
0.5 Sample Data and Experimentation . . . 37
0.5.1 Sample Data . . . 37
0.5.2 Comparison Setup . . . 37
0.5.3 What Else Could/Should/Will Be Done . . . 39
0.5.3.1 Combination of Feature Extraction Methods . . . 40
0.5.3.2 Entire Recognition Path . . . 40
0.5.3.3 More Methods . . . 40
0.6 Experementation Results . . . 41
0.6.1 Notes . . . 41
0.6.2 Configuration Explained . . . 42
0.6.3 Consolidated Results . . . 43
0.7 Conclusions . . . 50
0.8 APPENDIX . . . 52
0.8.1 Spectrogram Examples . . . 52
0.8.2 MARF Source Code . . . 52

0.8.3 SpeakerIdentApp and SpeakersIdentDb Source Code 53
0.8.3.1 SpeakerIdentApp.java 53
0.8.3.2 SpeakersIdentDb.java 60
0.8.4 TODO 68

List of Figures

1 Overall Architecture 9
2 The Core Pipeline 10
3 MARF Java Packages 11
4 Storage 15
5 Preprocessing 21
6 Normalization of aihua5.wav from the testing set. 22
7 FFT of normalized aihua5.wav from the testing set. 22
8 Low-pass filter applied to aihua5.wav. 23
9 High-pass filter applied to aihua5.wav. 24
10 Band-pass filter applied to aihua5.wav. 24
11 High frequency boost filter applied to aihua5.wav. 25
12 Feature Extraction 27
13 Classification 32
14 GUI Package 36
15 LPC spectrogram obtained for ian15.wav 52
16 LPC spectrogram obtained for graham13.wav 52

List of Tables

1 Speakers contributed their voice samples. 37
2 Consolidated results, Part 1. 44
3 Consolidated results, Part 2. 45
4 Consolidated results, Part 3. 46
5 Consolidated results, Part 4. 47
6 Consolidated results, Part 5. 48
7 Consolidated results, Part 6. 49

0.1 Intro

0.1.1 What is MARF?

MARF stands for **M**odular **A**udio **R**ecognition **F**ramework.

Four students of Concordia University have started this as a course project in September 2002. Now it's a project in its own being maintained and developed as we have time for it. If you have some suggestions, contributions to make, or for bug reports, don't hesitate to contact us :-)

Please report bugs to `marf-bugs@lists.sf.net`.

0.1.2 Authors

In alphabetical order:

- Ian Clément, `i_clemen@cs.concordia.ca`
- Serguei Mokhov, `mokhov@cs.concordia.ca`, a.k.a Serge
- Dimitrios Nicolacopoulos, `d_nicola@cs.concordia.ca`, a.k.a Jimmy
- Stephen Sinclair, `step_sin@cs.concordia.ca`, a.k.a. Steve, radarsat1

For MARF-related issues please contact us at `marf-devel@lists.sf.net`.

0.1.3 Purpose

Our main goal is to build a general framework to allow developers in the audio-recognition industry (be it speech, voice, sound, etc.) to choose and apply various methods, contrast and compare them, and use them in their applications. As a proof of concept, a user frontend application for Text-Independent (TI) Speaker Identification has been created on top of the framework (the `SpeakerIdentApp` program).

0.1.4 Project Source and Location

Our project from the its inception has always been an open-source project. All releases including the most current one should most of the time be accessible via `<http://marf.sourceforge.net>`. We have complete API documentation as well as this manual and all the sources avilable to download through this web page.

0.1.5 Why Java?

We have chosen to implement our project using the Java programming language. This choice is justified by the binary portability of the Java applications as well as facilitating memory management tasks and other issues, so we can concentrate more on the algorithms instead. Java also provides us with built-in types and data-structures to manage collections (build, sort, store/retrieve) efficiently.

0.2 MARF Architecture

Before we begin, you should understand the basic MARF system architecture. Understanding how the parts of MARF interact will make the next sections somewhat clearer. This document presents architecture of the MARF system, including the layout of the physical directory structure, and Java packages.

Let's take a look at the general MARF structure in Figure 1. The MARF class is the central "server" and configuration placeholder which major method - the core pipeline - a typical pattern recognition process. The figure presents basic abstract modules of the architecture. When a developer needs to add or use a module, they derive from the generic ones,

The core pipeline sequence diagram from an application up until the very end result is presented on Figure 2. It includes all major participants as well as basic operations. The participants are the modules responsible for a typical general pattern recognition pipeline.

Consequently, the framework has the mentioned basic modules, as well as some additional entities to manage storage and serialization of the input/output data.

0.2.1 Application Point of View

An application, using the framework, has to choose the concrete configuration and submodules for preprocessing, feature extraction, and classification stages. There is an API the application may use defined by each module or it can use them through the MARF.

There are two phases in MARF's usage by an application:

- Training, i.e. `train()`
- Recognition, i.e. `recognize()`

Training is performed on a virgin MARF installation to get some training data in. Recognition is an actual identification process of a sample against previously stored patterns during training.

0.2.2 Packages and Physical Layout

The Java package structure is in Figure 3. The following is the basic structure of MARF:

```
marf.*

MARF.java - The MARF Server
            Supports Training and Recognition mode
            and keeps all the configuration settings.
```

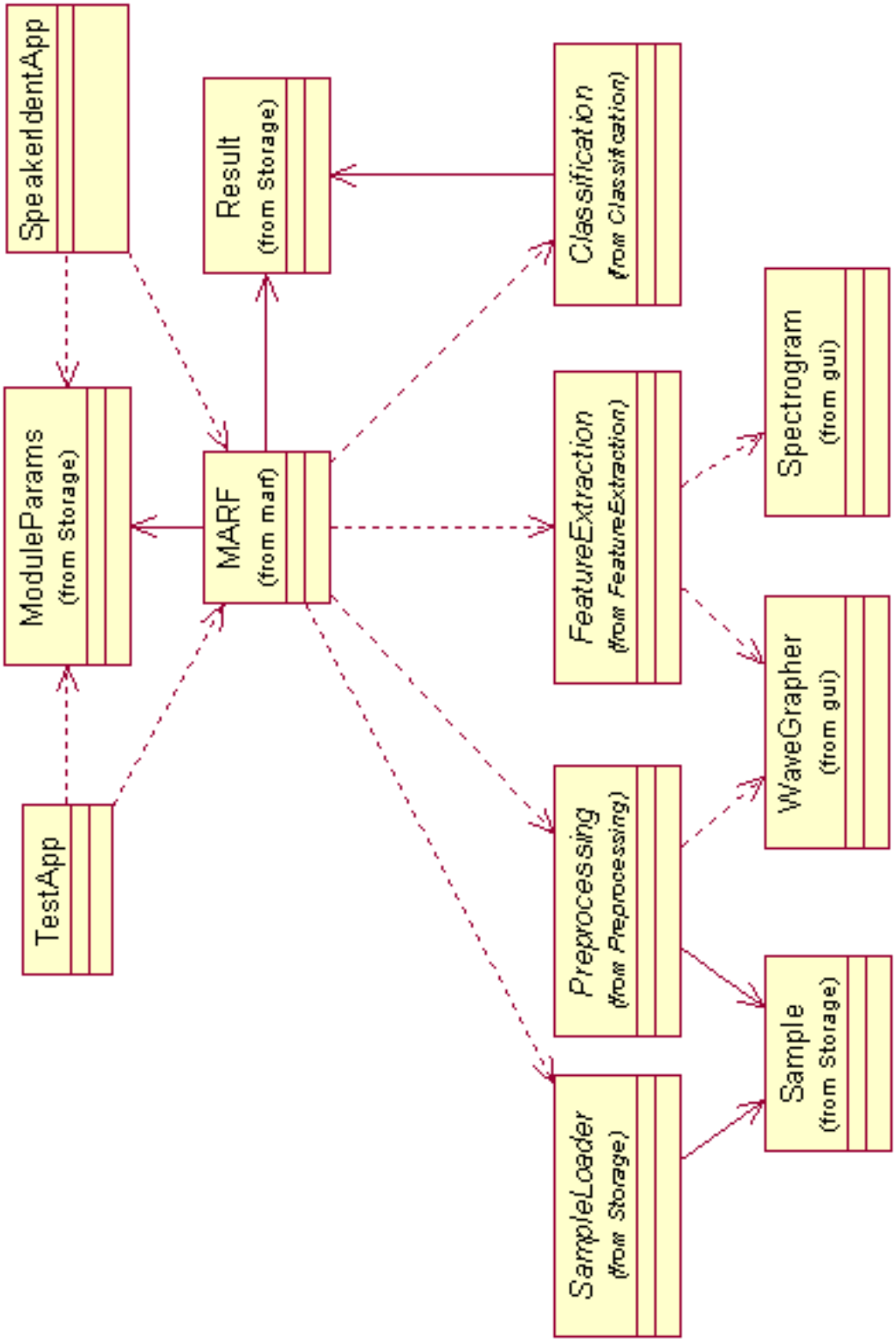

Figure 1: Overall Architecture

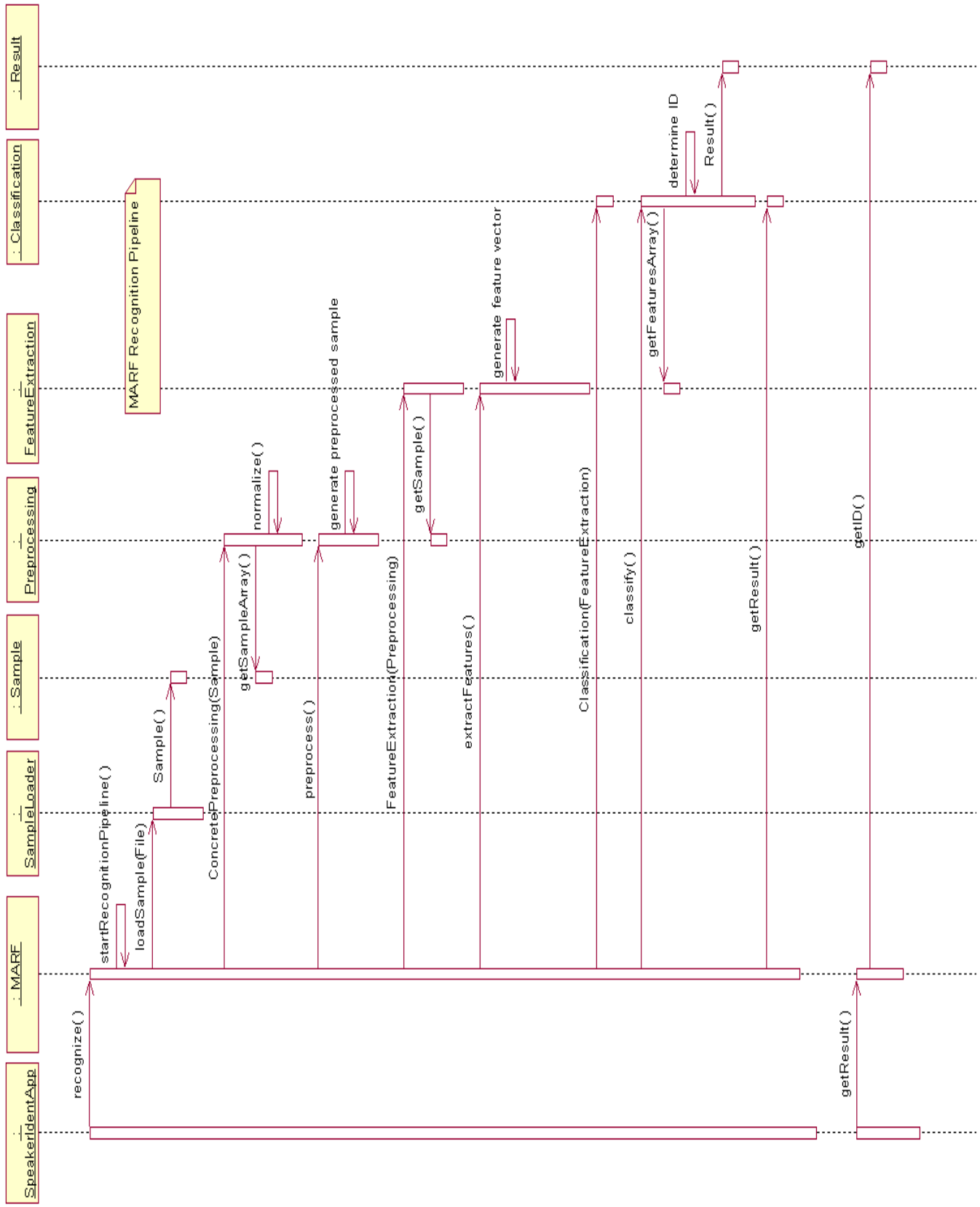

Figure 2: The Core Pipeline

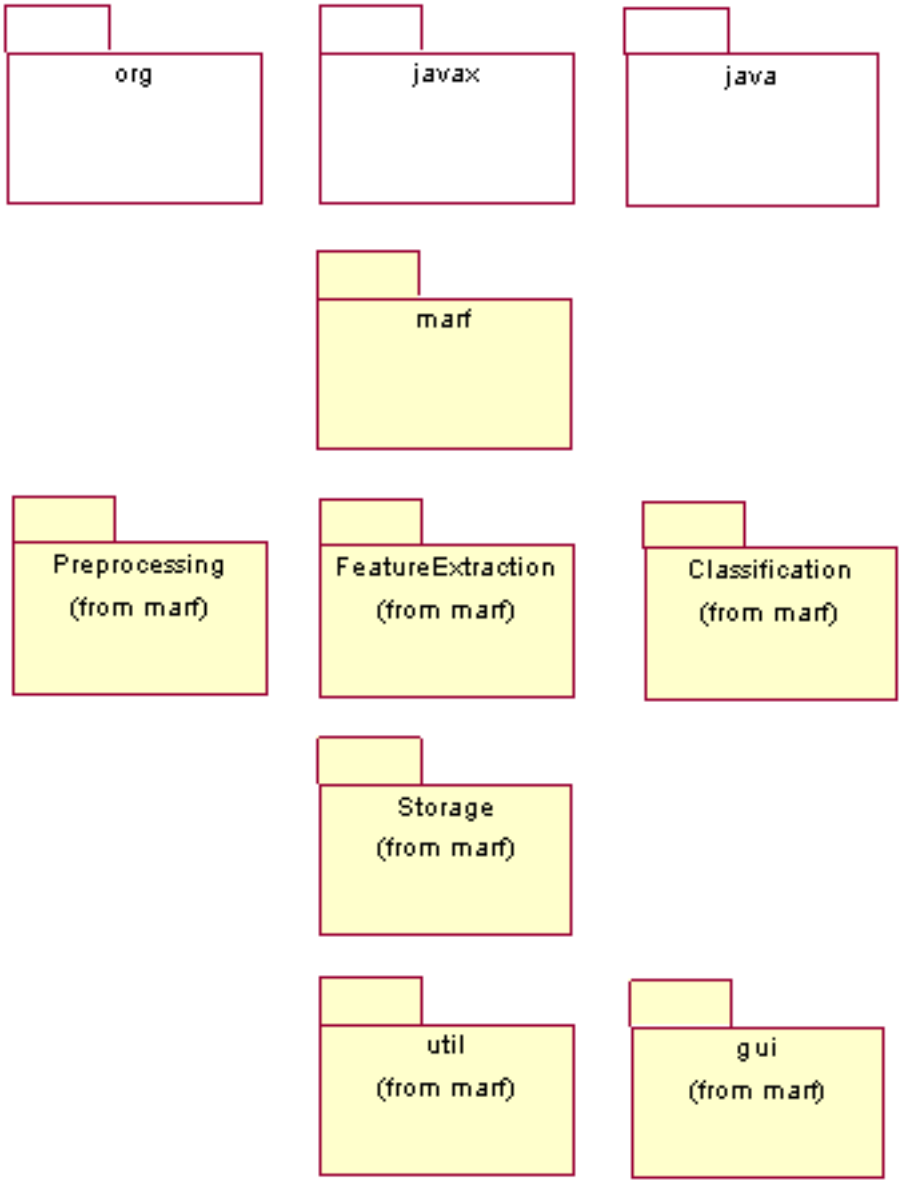

Figure 3: MARF Java Packages

```
marf.Preprocessing.* - The Preprocessing Package

/marf/Preprocessing/

   Preprocessing.java - Abstract Preprocessing Module, has to be subclassed

   /Endpoint/*.java   - Endpoint Filter as implementation of Preprocessing
   /Dummy/*.java
   /FFTFilter/
       FFTFilter.java
       LowPassFilter.java
       HighPassFilter.java
       BandpassFilter.java - Bandpass Filter as implementation of Preprocessing
       HighFrequencyBoost.java

marf.FeatureExtraction.* - The Feature Extraction Package

/marf/FeatureExtraction/
    FeatureExtraction.java
    /FFT/FFT.java        - FFT implementation of Preprocessing
    /LPC/LPC.java        - LPC implementation of Preprocessing
    /Cepstral/*.java
    /Segmentation/*.java
    /F0/*.java

marf.Classification.* - The Classification Package

/marf/Classification/
    Classification.java
    /NeuralNetwork/NeuralNetwork.java
    /Stochastic/*.java
    /Markov/*.java
    /Distance/
       Distance.java
       EuclideanDistance.java
       ChebyshevDistance.java
       MinkowskiDistance.java
       MahalonobisDistance.java

marf.Storage.* - The Physical Storage Management Interface
```

```
/marf/Storage/
    Sample.java
    ModuleParams.java
    TrainingSet.java
    Result.java
    StorageManager.java - Interface to be implemented by the above modules
    SampleLoader.java   - Should know how to load different sample format
    /Loaders/*.*        - WAV, MP3, ULAW, etc.

marf.Stats.* - The Statistics Package meant to collect various types of stats.

/marf/Stats/
    StatsCollector.java - Time took, noise removed, patterns stored, modules available, etc.

marf.gui.* - GUI to the graphs and configuration

/marf/gui/
    Spectrogram.java
    WaveGrapher.java
```

0.2.3 Current Limitations

Our current pipeline is maybe somewhat too rigid. That is, there's no way to specify more than one preprocessing or feature extraction module to process the same sample in one pass. (In the case of preprocessing one filter may be used along with normalization together, or just normalization by itself).

Also, it assumes that the whole sample is loaded before doing anything with it, instead of sending parts of the sample a bit at a time. Perhaps this simplifies things, but it won't allow us to deal with large samples at the moment. However, it's not a problem for our framework and the application since memory is cheap and samples are not too big. Additionally, we have streaming support already in the `WAVLoader` and some modules support it, but the final conversion to streaming did not happen in this release.

MARF provides only limited support for inter-module dependency. It is possible to pass module-specific arguments, but problems like number of parameters mismatch between feature extraction and classification, and so on are not tracked.

0.3 Methodology

This section presents what methods and algorithms were implemented and used in this project. We overview storage issues first, then preprocessing methods followed by feature extraction, and ended by classification.

0.3.1 Storage

Figure 4 presents basic `Storage` modules and their API.

0.3.1.1 Speaker Database

We store specific speakers in a comma-separated (CSV) file, `speakes.txt` within the application.

It has the following format:

```
<id:int>,<name:string>,<training-samples:list>,<testing-samples:list>
```

Sample lists are defined as follows:

```
<*-sample-list> := filename1.wav|filename2.wav|...
```

0.3.1.2 Storing Features, Training, and Classification Data

We defined a standard `StorageManager` interface for the modules to use. That's part of the `StorageManager` interface which each module will override because each a module has to know how to serialize itself, but the application, using MARF, should not care. Thus, this `StorageManager` is a base class with abstract methods `dump()` and `restore()`. And these functions would generalize the model's storing, in the sense that they are somehow "read" and "written".

We have to store data we used for training for later use in the classification process. For this we pass FFT (Section 0.3.3.2) and LPC (Section 0.3.3.3) feature vectors through the `TrainingSet`/`TrainingSample` class pair, which, as a result, store mean vectors (clusters) for our training models.

In the Neural Network we use XML. The only reason XML and text files have been suggested is to allow us to easily modify values in a text editor and verify the data visually.

In the Neural Network classification, we are using one net for all the speakers. We had thought that one net for each speaker would be ideal, but then we'll lose too much discrimination power. But doing this means that the net will need complete re-training for each new training utterance (or group thereof).

We have a training/testing script that lists the location of all the wave files to be trained along with the identification of the speaker - `testing.sh`.

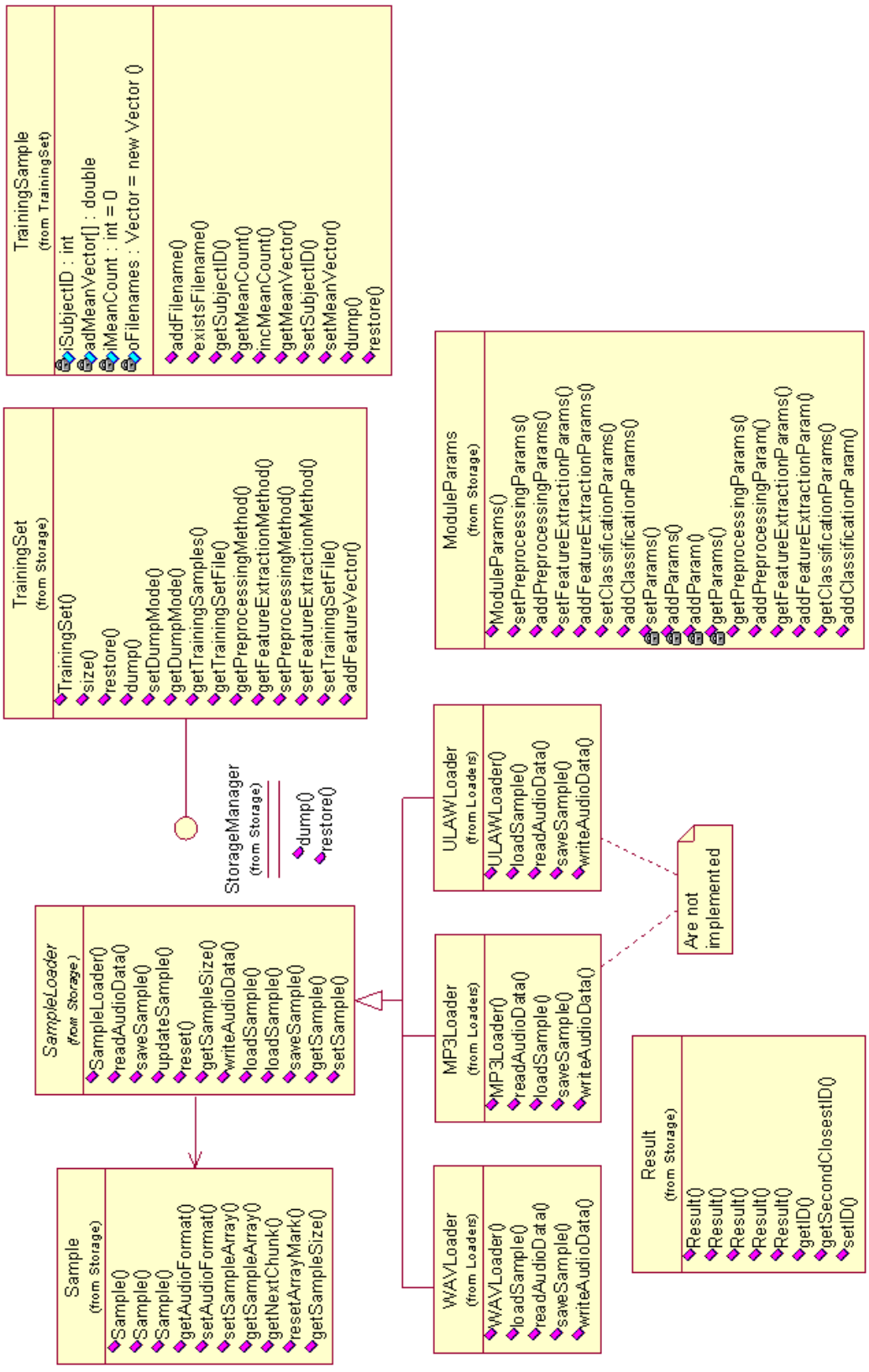

Figure 4: Storage

0.3.1.3 File Location

We decided to keep all the data and intermediate files in the same directory or subdirectories of that of the application.

- `marf.Storage.TrainingSet.*` - represent training sets (global clusters) used in training with different preprocessing and feature extraction methods; they can either be gzipped binary (.bin) or CSV text (.csv).
- `speakers.txt.stats` - binary statistics file.
- `marf.Classification.NeuralNetwork.*.xml` - XML file representing a trained Neural Net for all the speakers in the database.
- `training-samples/` - directory with WAV files for training.
- `testing-samples/` - directory with WAV files for testing.

0.3.1.4 Sample and Feature Sizes

Wave files are read and outputted as an array of data points that represents the waveform of the signal.

Different methods will have different feature vector sizes. It depends on what kind of precision one desires. In the case of FFT, a 1024 FFT will result in 512 features, being an array of "doubles" corresponding to the frequency range.

[1] said about using 3 ms for phoneme analysis and something like one second for general voice analysis. At 8 kHz, 1024 samples represents 128 ms, this might be a good compromise.

0.3.1.5 Parameter Passing

A generic `ModuleParams` container class has been created to for an application to be able to pass module-specific parameters when specifying model files, training data, amount of LPC coefficients, FFT window size, logging/stats files, etc.

0.3.1.6 Result

When classification is over, its result should be stored somehow for further retrieval by the application. We have defined the `Result` object to carry out this task. It contains ID of the subject identified as well as some additional statistics (such as second closest speaker and distances from other speakers, etc.)

0.3.1.7 Sample Format

The sample format used for our samples was the following:

- Audio Format: PCM signed (WAV)
- Sample Rate: 8000 Hz
- Audio Sample Size: 16 bit
- Channels: 1 (mono)
- Duration: from about 7 to 20 seconds

All training and testing samples were recorded through an external sound recording program (MS Sound Recorder) using a standard microphone. Each sample was saved as a WAV file with the above properties and stored in the appropriate folders where they would be loaded from within the main application. The PCM audio format (which stands for Pulse Code Modulation) refers to the digital encoding of the audio sample contained in the file and is the format used for WAV files. In a PCM encoding, an analog signal is represented as a sequence of amplitude values. The range of the amplitude value is given by the audio sample size which represents the number of bits that a PCM value consists of. In our case, the audio sample size is 16-bit which means that that a PCM value can range from 0 to 65536. Since we are using PCM-signed format, this gives an amplitude range between -32768 and 32768. That is, the amplitude values of each recorded sample can vary within this range. Also, this sample size gives a greater range and thus provides better accuracy in representing an audio signal then using a sample size of 8-bit which limited to a range of $(-128, 128)$. Therefore, a 16-bit audio sample size was used for our experiments in order to provide the best possible results. The sampling rate refers to the number of amplitude values taken per second during audio digitization. According to the Nyquist theorem, this rate must be at least twice the maximum rate (frequency) of the analog signal that we wish to digitize ([3]). Otherwise, the signal cannot be properly regenerated in digitized form. Since we are using an 8 kHz sampling rate, this means that actual analog frequency of each sample is limited to 4 kHz. However, this limitation does not pose a hindrance since the difference in sound quality is negligible ([1]). The number of channels refers to the output of the sound (1 for mono and 2 for stereo – left and right speakers). For our experiment, a single channel format was used to avoid complexity during the sample loading process.

0.3.1.8 Sample Loading Process

To read audio information from a saved voice sample, a special sample-loading component had to be implemented in order to load a sample into an internal data structure for further processing. For this, certain sound libraries (`javax.sound.sampled`) were provided from the Java programming language which enabled us to stream the audio data from the sample file. However once the data was captured, it had to be converted into readable amplitude values since the library routines only provide PCM values of the sample. This required the need to implement special routines to convert raw PCM values to actual amplitude values (see `SampleLoader` class in the `Storage` package).

The following pseudo-code represents the algorithm used to convert the PCM values into real amplitude values ([7]):

```
function readAmplitudeValues(Double Array : audioData)
{
    Integer: MSB, LSB,
             index = 0;

    Byte Array: AudioBuffer[audioData.lenth * 2];

    read audio data from Audio stream into AudioBuffer;

    while(not reached the end of stream OR index not equal to audioData.length)
    {
        if(Audio data representation is BigEndian)
        {
            // First byte is MSB (high order)
            MSB = audioBuffer[2 * index];
            // Second byte is LSB (low order)
            LSB = audioBuffer[2 * index + 1];
        }
        else
        {
            // Vice-versa...
            LSB = audioBuffer[2 * index];
            MSB = audioBuffer[2 * index + 1];
        }

        // Merge high-order and low-order byte to form a 16-bit double value.
        // Values are divided by maximum range
        audioData[index] = (merge of MSB and LSB) / 32768;
    }
}
```

This function reads PCM values from a sample stream into a byte array that has twice the length of `audioData`; the array which will hold the converted amplitude values (since sample size is 16-bit). Once the PCM values are read into `audioBuffer`, the high and low order bytes that make up the amplitude value are extracted according to the type of representation defined in the sample's audio format. If the data representation is *big endian*, the high order byte of each PCM value is located at every even-numbered position in `audioBuffer`. That is, the high order byte of the first PCM value is found at position 0, 2 for

the second value, 4 for the third and so forth. Similarly, the low order byte of each PCM value is located at every odd-numbered position (1, 3, 5, etc.). In other words, if the data representation is *big endian*, the bytes of each PCM code are read from left to right in the `audioBuffer`. If the data representation is not *big endian*, then high and low order bytes are inversed. That is, the high order byte for the first PCM value in the array will be at position 1 and the low order byte will be at position 0 (read right to left). Once the high and low order bytes are properly extracted, the two bytes can be merged to form a 16-bit double value. This value is then scaled down (divide by 32768) to represent an amplitude within a unit range $(-1, 1)$. The resulting value is stored into the audioData array, which will be passed to the calling routine once all the available audio data is entered into the array. An additional routine was also required to write audio data from an array into wave file. This routine involved the inverse of reading audio data from a sample file stream. More specifically, the amplitude values inside an array are converted back to PCM codes and are stored inside an array of bytes (used to create new audio stream). The following illustrates how this works:

```
public void writePCMValues(Double Array: audioData)
{
    Integer: word  = 0,
             index = 0;

    Byte Array: audioBytes[(number of ampl. values in audioData) * 2];

    while(index not equal to (number of ampl. values in audioData * 2))
    {
        word = (audioData[index] * 32768);
        extract high order byte and place it in appropriate position in audioBytes;
        extract low order byte and place it in appropriate position in audioBytes;
    }

    create new audio stream from audioBytes;
}
```

0.3.2 Preprocessing

This section outlines the preprocessing mechanisms considered and implemented in MARF. We present you with the API and structure in Figure 5, along with the description of the methods.

0.3.2.1 Normalization

Since not all voices will be recorded at exactly the same level, it is important to normalize the amplitude of each sample in order to ensure that features will be comparable. Audio normalization is analogous to image normalization. Since all samples are to be loaded as floating point values in the range $[-1.0, 1.0]$, it should be ensured that every sample actually does cover this entire range.

The procedure is relatively simple: find the maximum amplitude in the sample, and then scale the sample by dividing each point by this maximum. Figure 6 illustrates normalized input wave signal.

0.3.2.2 FFT Filter

The FFT filter is used to modify the frequency domain of the input sample in order to better measure the distinct frequencies we are interested in. Two filters are useful to speech analysis: high frequency boost, and low-pass filter (yet we provided more of them, to toy around).

Speech tends to fall off at a rate of 6 dB per octave, and therefore the high frequencies can be boosted to introduce more precision in their analysis. Speech, after all, is still characteristic of the speaker at high frequencies, even though they have a lower amplitude. Ideally this boost should be performed via compression, which automatically boosts the quieter sounds while maintaining the amplitude of the louder sounds. However, we have simply done this using a positive value for the filter's frequency response. The low-pass filter (Section 0.3.2.3) is used as a simplified noise reducer, simply cutting off all frequencies above a certain point. The human voice does not generate sounds all the way up to 4000 Hz, which is the maximum frequency of our test samples, and therefore since this range will only be filled with noise, it may be better just to cut it out.

Essentially the FFT filter is an implementation of the Overlap-Add method of FIR filter design. The process is a simple way to perform fast convolution, by converting the input to the frequency domain, manipulating the frequencies according to the desired frequency response, and then using an Inverse-FFT to convert back to the time domain. Figure 7 demonstrates the normalized incoming wave form translated into the frequency domain.

The code applies the square root of the hamming window to the input windows (which are overlapped by half-windows), applies the FFT, multiplies the results by the desired frequency response, applies the Inverse-FFT, and applies the square root of the hamming window again, to produce an undistorted output.

Another similar filter could be used for noise reduction, subtracting the noise characteristics from the frequency response instead of multiplying, thereby remove the room noise from the input sample.

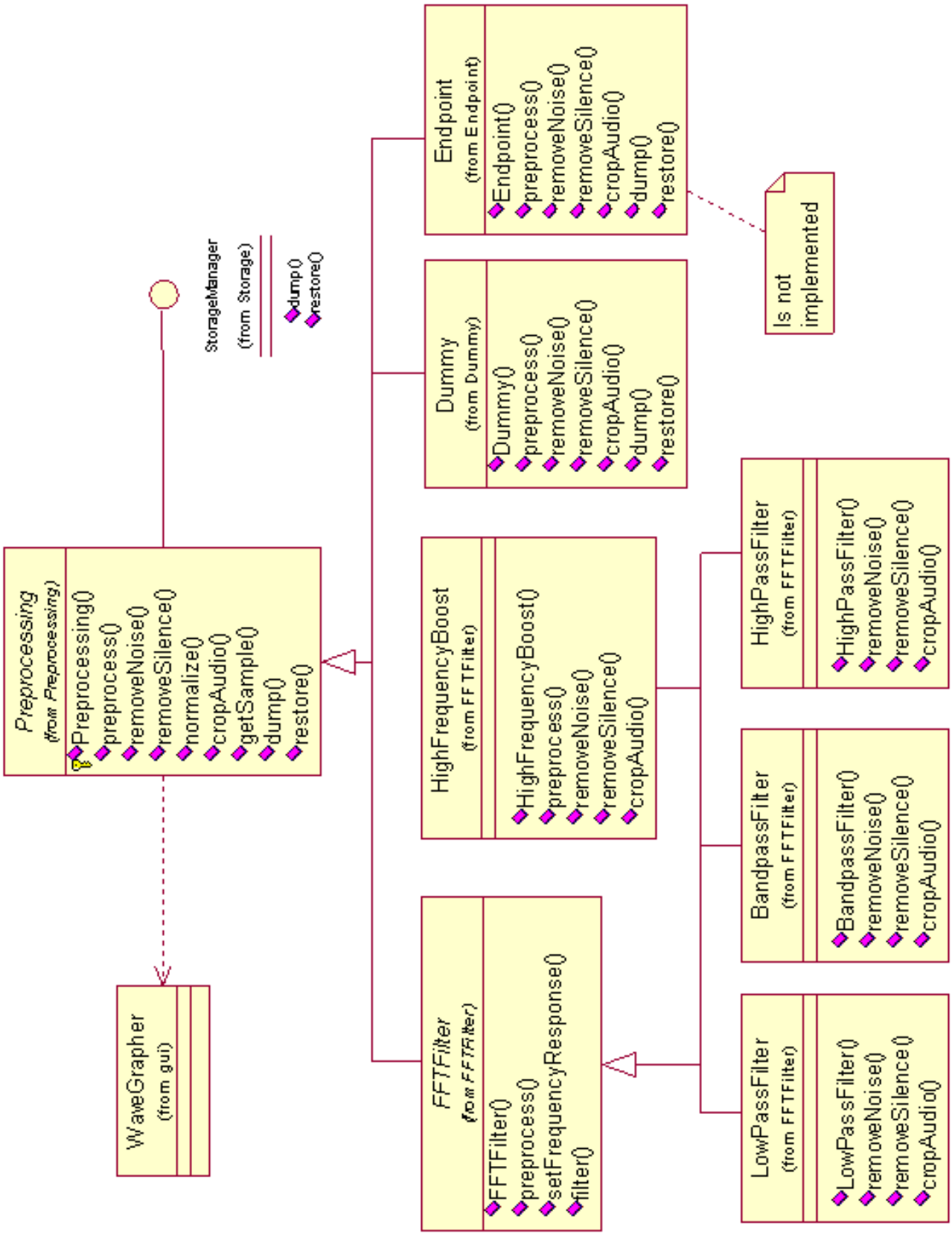

Figure 5: Preprocessing

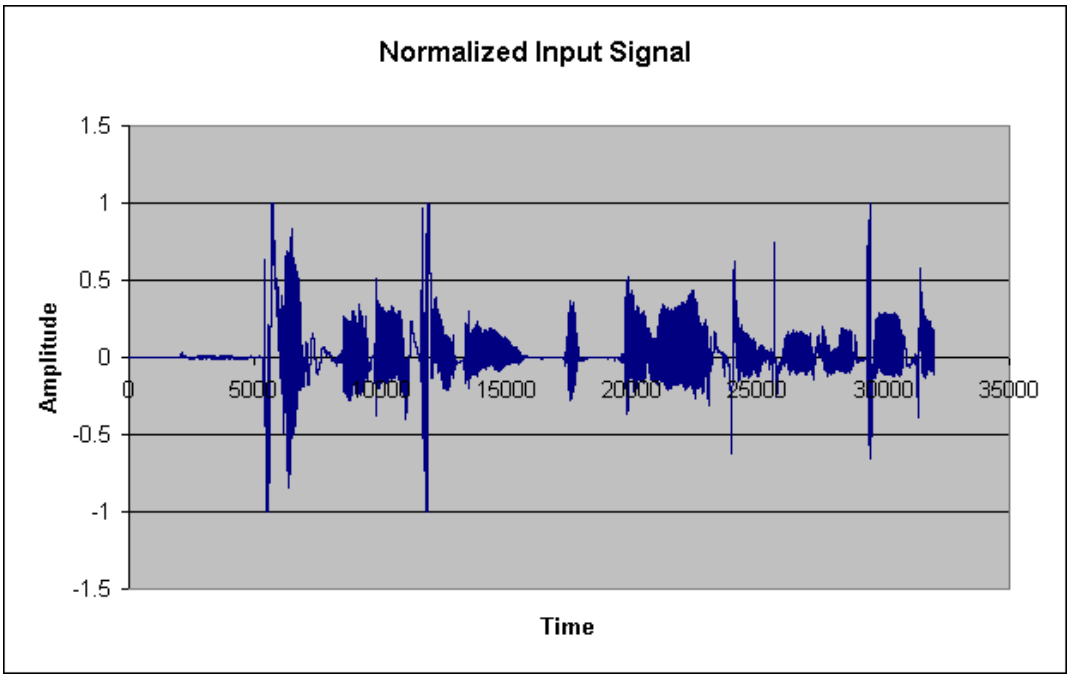

Figure 6: Normalization of aihua5.wav from the testing set.

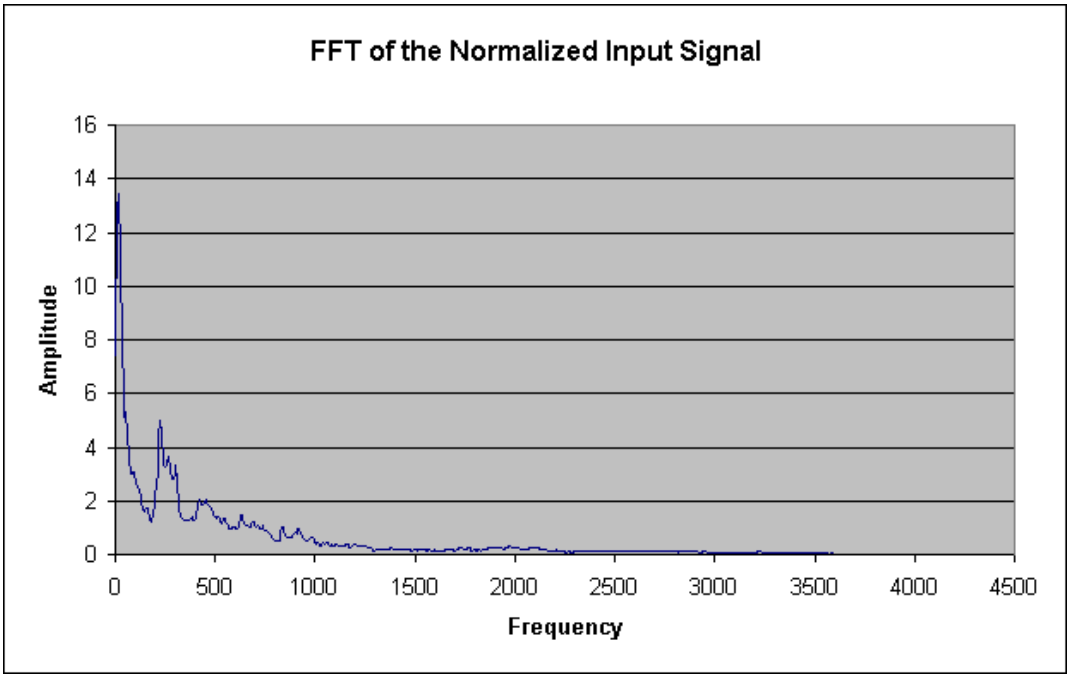

Figure 7: FFT of normalized aihua5.wav from the testing set.

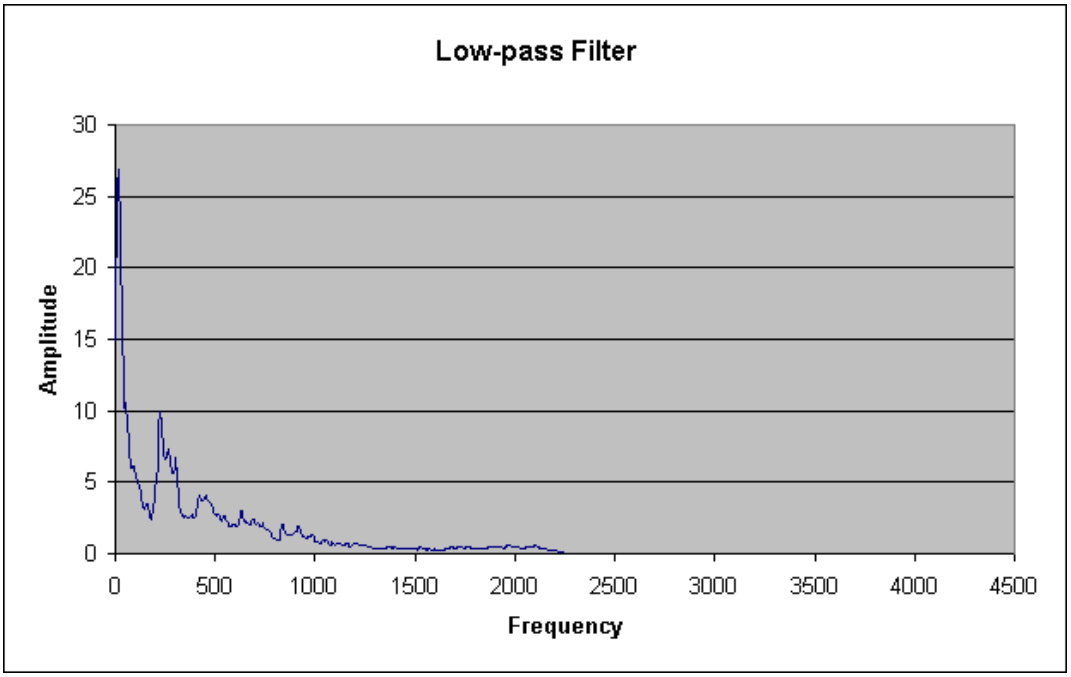

Figure 8: Low-pass filter applied to aihua5.wav.

0.3.2.3 Low-Pass Filter

The low-pass filter has been realized on top of the FFT Filter, by setting up frequency response to zero for frequencies past certain threshold chosen heuristically based on the window size where to cut off. We filtered out all the frequencies past 2853 Hz.

Figure 8 presents FFT of a low-pass filtered graph.

0.3.2.4 High-Pass Filter

As the low-pass filter, the high-pass filter (e.g. is in Figure 9) has been realized on top of the FFT Filter, in fact, it is the opposite to low-pass filter, and filters out frequencies before 2853 Hz. What would be very useful to do is to test it along with high-frequency boost, but we've never managed to do so for 0.2.0.

0.3.2.5 Band-Pass Filter

Band-pass filter in MARF is yet another instance of an FFT Filter (Section 0.3.2.2), with the default settings of the band of frequencies of $[1000, 2853]$ Hz. See Figure 10.

0.3.2.6 High Frequency Boost

This filter was also implemented on top of the FFT filter to boost the high-end frequencies. The frequencies boosted after approx. 1000 Hz by a factor of 5π, heuristically determined, and then re-normalized. See Figure 11.

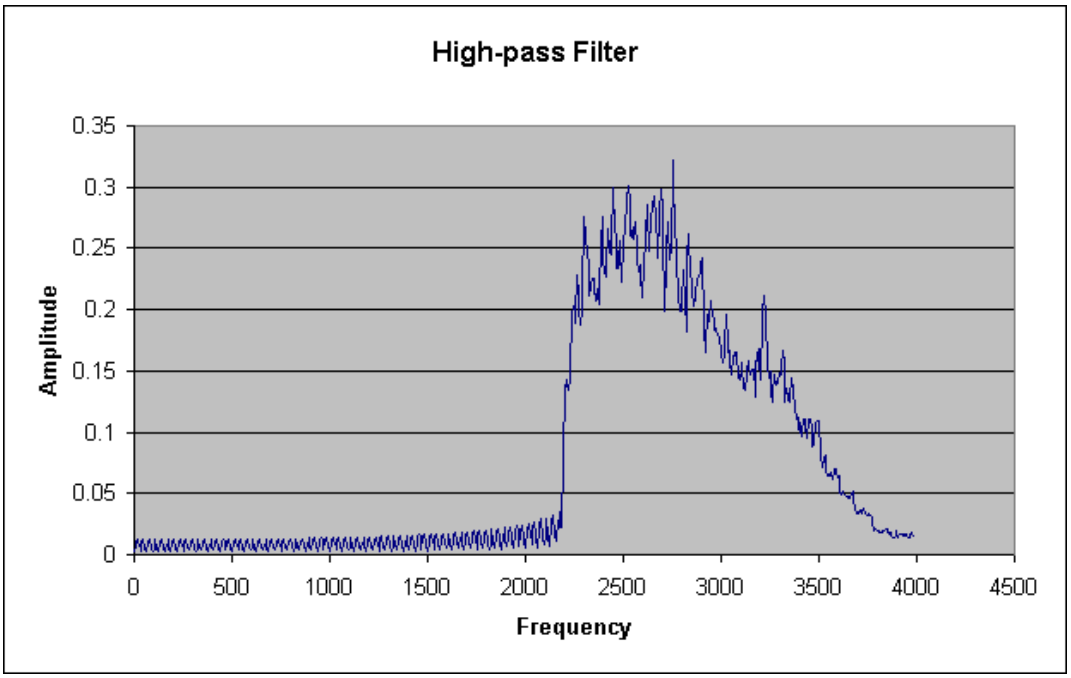

Figure 9: High-pass filter applied to aihua5.wav.

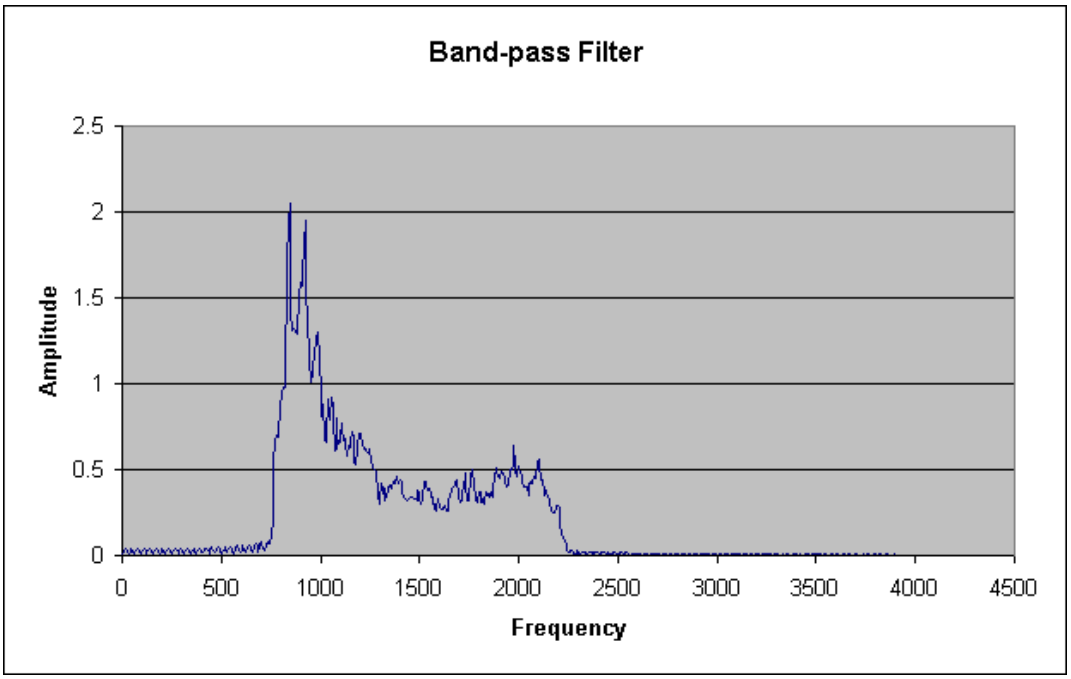

Figure 10: Band-pass filter applied to aihua5.wav.

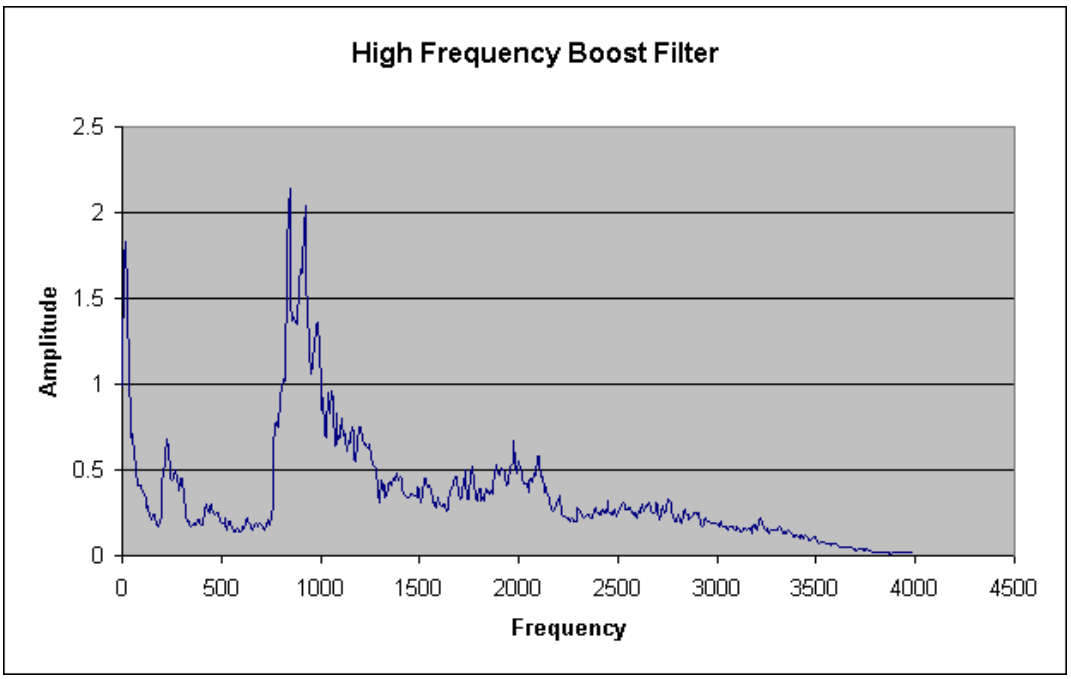

Figure 11: High frequency boost filter applied to aihua5.wav.

0.3.2.7 Noise Removal

Any vocal sample taken in a less-than-perfect (which is always the case) environment will experience a certain amount of room noise. Since background noise exhibits a certain frequency characteristic, if the noise is loud enough it may inhibit good recognition of a voice when the voice is later tested in a different environment. Therefore, it is necessary to remove as much environmental interference as possible.

To remove room noise, it is first necessary to get a sample of the room noise by itself. This sample, usually at least 30 seconds long, should provide the general frequency characteristics of the noise when subjected to FFT analysis. Using a technique similar to overlap-add FFT filtering, room noise can then be removed from the vocal sample by simply subtracting the noise's frequency characteristics from the vocal sample in question.

That is, if $S(x)$ is the sample, $N(x)$ is the noise, and $V(x)$ is the voice, all in the frequency domain, then

$$S(x) = N(x) + V(x)$$

Therefore, it should be possible to isolate the voice:

$$V(x) = S(x) - N(x)$$

Unfortunately, time has not permitted us to implement this in practice yet.

0.3.3 Feature Extraction

This section outlines feature extraction methods of our project. First we present you with the API and structure, followed by the description of the methods. The class diaram of this module set is in Figure 12.

0.3.3.1 The Hamming Window

In many DSP techniques, it is necessary to consider a smaller portion of the entire speech sample rather than attempting to process the entire sample at once. The technique of cutting a sample into smaller pieces to be considered individually is called "windowing". The simplest kind of window to use is the "rectangle", which is simply an unmodified cut from the larger sample. Unfortunately, rectangular windows can introduce errors, because near the edges of the window there will potentially be a sudden drop from a high amplitude to nothing, which can produce false "pops" and "clicks" in the analysis.

A better way to window the sample is to slowly fade out toward the edges, by multiplying the points in the window by a "window function". If we take successive windows side by side, with the edges faded out, we will distort our analysis because the sample has been modified by the window function. To avoid this, it is necessary to overlap the windows so that all points in the sample will be considered equally. Ideally, to avoid all distortion, the overlapped window functions should add up to a constant. This is exactly what the Hamming window does. It is defined as:

$$x = 0.54 - 0.46 \cdot \cos\left(\frac{2\pi n}{l-1}\right)$$

where x is the new sample amplitude, n is the index into the window, and l is the total length of the window.

0.3.3.2 Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) algorithm is used both for feature extraction and as the basis for the filter algorithm used in preprocessing. Although a complete discussion of the FFT algorithm is beyond the scope of this document, a short description of the implementation will be provided here.

Essentially the FFT is an optimized version of the Discrete Fourier Transform. It takes a window of size 2^k and returns a complex array of coefficients for the corresponding frequency curve. For feature extraction, only the magnitudes of the complex values are used, while the FFT filter operates directly on the complex results.

The implementation involves two steps: First, shuffling the input positions by a binary reversion process, and then combining the results via a "butterfly" decimation in time to produce the final frequency coefficients. The first step corresponds to breaking down the time-domain sample of size n into n frequency-domain samples of size 1. The second step re-combines the n samples of size 1 into 1 n-sized frequency-domain sample.

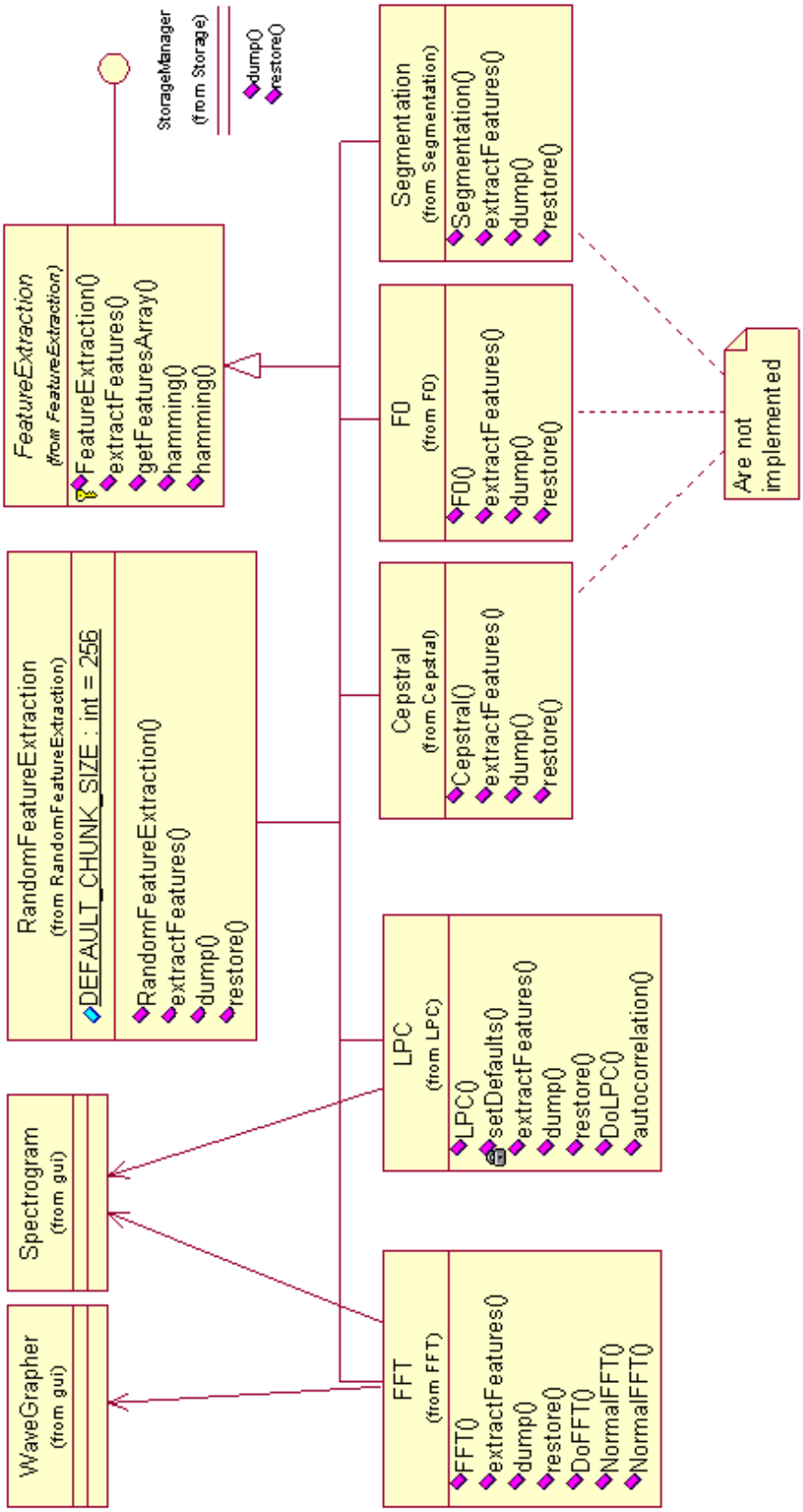

Figure 12: Feature Extraction

The code used in MARF has been translated from the C code provided in the book, "Numeric Recipes in C".

0.3.3.2.1 FFT Feature Extraction The frequency-domain view of a window of a time-domain sample gives us the frequency characteristics of that window. In feature identification, the frequency characteristics of a voice can be considered as a list of "features" for that voice. If we combine all windows of a vocal sample by taking the average between them, we can get the average frequency characteristics of the sample. Subsequently, if we average the frequency characteristics for samples from the same speaker, we are essentially finding the center of the cluster for the speaker's samples. Once all speakers have their cluster centers recorded in the training set, the speaker of an input sample should be identifiable by comparing its frequency analysis with each cluster center by some classification method.

Since we are dealing with speech, greater accuracy should be attainable by comparing corresponding phonemes with each other. That is, "th" in "the" should bear greater similarity to "th" in "this" than will "the" and "this" when compared as a whole.

The only characteristic of the FFT to worry about is the window used as input. Using a normal rectangular window can result in glitches in the frequency analysis because a sudden cutoff of a high frequency may distort the results. Therefore it is necessary to apply a Hamming window to the input sample, and to overlap the windows by half. Since the Hamming window adds up to a constant when overlapped, no distortion is introduced.

When comparing phonemes, a window size of about 2 or 3 ms is appropriate, but when comparing whole words, a window size of about 20 ms is more likely to be useful. A larger window size produces a higher resolution in the frequency analysis.

0.3.3.3 Linear Predictive Coding (LPC)

This section presents implementation of the LPC Classification module.

One method of feature extraction used in the MARF project was Linear Predictive Coding (LPC) analysis. It evaluates windowed sections of input speech waveforms and determines a set of coefficients approximating the amplitude vs. frequency function. This approximation aims to replicate the results of the Fast Fourier Transform yet only store a limited amount of information: that which is most valuable to the analysis of speech.

0.3.3.3.1 Theory The LPC method is based on the formation of a spectral shaping filter, $H(z)$, that, when applied to a input excitation source, $U(z)$, yields a speech sample similar to the initial signal. The excitation source, $U(z)$, is assumed to be a flat spectrum leaving all the useful information in $H(z)$. The model of shaping filter used in most LPC implementation is called an "all-pole" model, and is as follows:

$$H(z) = \frac{G}{\left(1 - \sum_{k=1}^{p}(a_k z^{-k})\right)}$$

Where p is the number of poles used. A pole is a root of the denominator in the Laplace transform of the input-to-output representation of the speech signal.

The coefficients a_k are the final representation if the speech waveform. To obtain these coefficients, the least-square autocorrelation method was used. This method requires the use of the autocorrelation of a signal defined as:

$$R(k) = \sum_{m=k}^{n-1} (x(n) \cdot x(n-k))$$

where $x(n)$ is the windowed input signal.

In the LPC analysis, the error in the approximation is used to derive the algorithm. The error at time n can be expressed in the following manner: $e(n) = s(n) - \sum_{k=1}^{p} (a_k \cdot s(n-k))$. Thusly, the complete squared error of the spectral shaping filter $H(z)$ is:

$$E = \sum_{n=-\infty}^{\infty} \left(x(n) - \sum_{k=1}^{p} (a_k \cdot x(n-k)) \right)$$

To minimize the error, the partial derivative $\frac{\delta E}{\delta a_k}$ is taken for each $k = 1..p$, which yields p linear equations of the form:

$$\sum_{n=-\infty}^{\infty} (x(n-i) \cdot x(n)) = \sum_{k=1}^{p} (a_k \cdot \sum_{n=-\infty}^{\infty} (x(n-i) \cdot x(n-k))$$

For $i = 1..p$. Which, using the autocorrelation function, is:

$$\sum_{k=1}^{p} (a_k \cdot R(i-k)) = R(i)$$

Solving these as a set of linear equations and observing that the matrix of autocorrelation values is a Toeplitz matrix yields the following recursive algorithm for determining the LPC coefficients:

$$k_m = \frac{\left(R(m) - \sum_{k=1}^{m-1} (a_{m-1}(k) R(m-k)) \right)}{E_{m-1}}$$

$$a_m(m) = k_m$$

$$a_m(k) = a_{m-1}(k) - k_m \cdot a_m(m-k) \text{ for } 1 \leq k \leq m-1,$$

$$E_m = (1 - k_m^2) \cdot E_{m-1}$$

.

This is the algorithm implemented in the MARF LPC module.

0.3.3.3.2 Usage for Feature Extraction The LPC coefficients were evaluated at each windowed iteration, yielding a vector of coefficient of size p. These coefficients were averaged across the whole signal to give a mean coefficient vector representing the utterance. Thus a p sized vector was used for training and testing. The value of p chosen was based on tests given speed vs. accuracy. A p value of around 20 was observed to be accurate and computationally feasible.

0.3.3.4 Random Feature Extraction

By default given a window of size 256 samples, it picks at random a number from a Gaussian distribution, and multiplies by the incoming sample frequencies. This all adds up and we have a feature vector at the end. This should be the bottom line performance of all feature extraction methods. It can also be used as a relatively fast testing module.

0.3.4 Classification

This section outlines classification methods of the MARF project. First, we present you with the API and overall structure, followed by the description of the methods. Overall structure of the modules is in Figure 13.

0.3.4.1 Chebyshev Distance

Chebyshev distance is used along with other distance classifiers for comparison. Chebyshev distance is also known as a city-block or Manhattan distance. Here's its mathematical representation:

$$d(x, y) = \sum_{k=1}^{n} (|x_k - y_k|)$$

where x and y are feature vectors of the same length n.

0.3.4.2 Euclidean Distance

The Euclidean Distance classifier uses an Euclidean distance equation to find the distance between two feature vectors.

If $A = (x_1, x_2)$ and $B = (y_1, y_2)$ are two 2-dimensional vectors, then the distance between A and B can be defined as the square root of the sum of the squares of their differences:

$$d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

This equation can be generalized to n-dimensional vectors by simply adding terms under the square root.

$$d(x, y) = \sqrt{(x_n - y_n)^2 + (x_{n-1} - y_{n-1})^2 + ... + (x_1 - y_1)^2}$$

or

$$d(x, y) = \sqrt{\sum_{k=1}^{n} (x_k - y_k)^2}$$

or

$$d(x, y) = \sqrt{(x - y)^T (x - y)}$$

A cluster is chosen based on smallest distance to the feature vector in question.

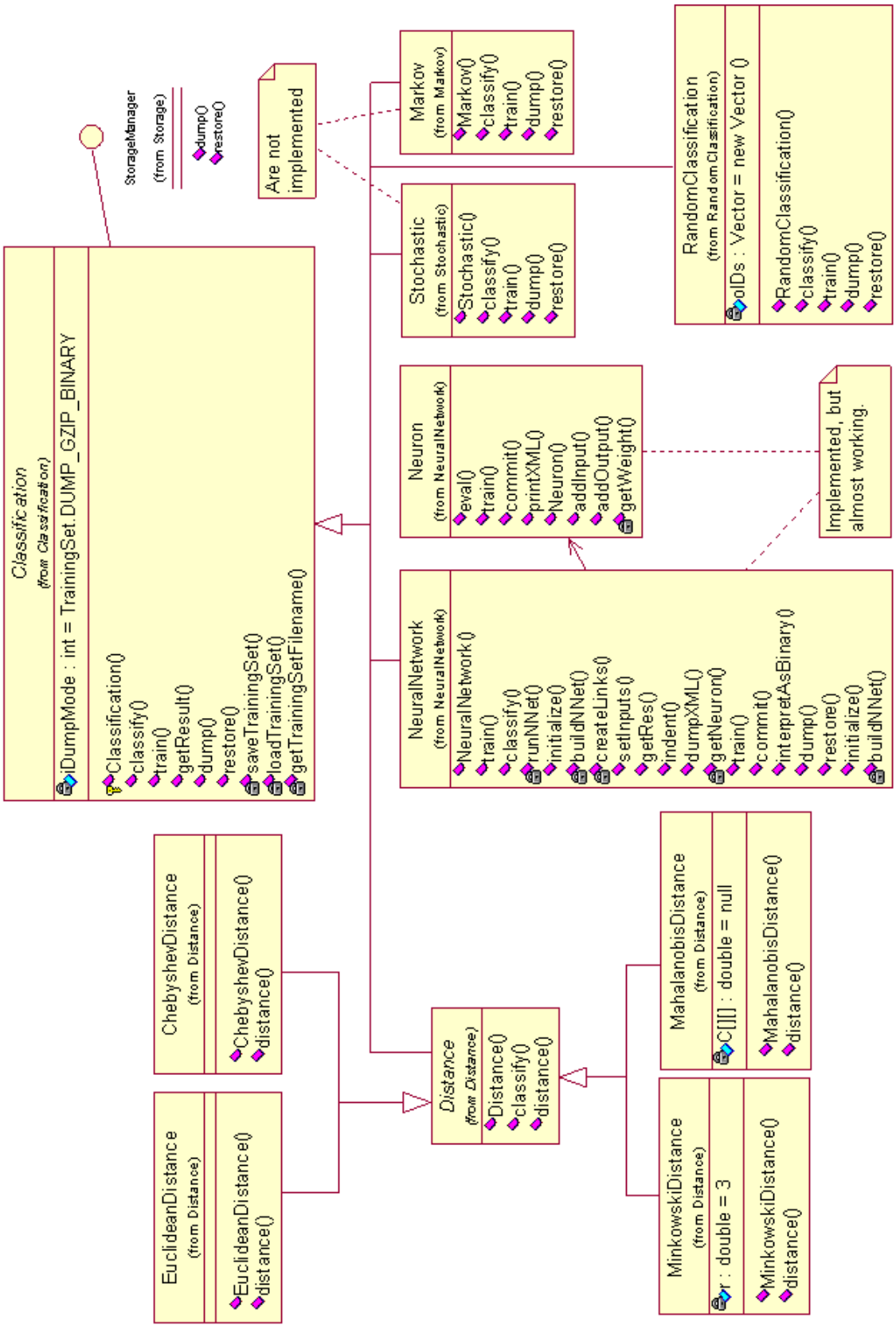

Figure 13: Classification

0.3.4.3 Minkowski Distance

Minkowski distance measurement is a generalization of both Euclidean and Chebyshev distances.

$$d(x, y) = \left(\sum_{k=1}^{n} (|x_k - y_k|)^r \right)^{\frac{1}{r}}$$

where r is a Minkowski factor. When $r = 1$, it becomes Chebyshev distance, and when $r = 2$, it is the Euclidean one. x and y are feature vectors of the same length n.

0.3.4.4 Mahalanobis Distance

This distance classification is meant to be able to detect features that tend to vary together in the same cluster if linear transformations are applied to them, so it becomes invariant from these transformations unlike all the other, previously seen distance classifiers.

$$d(x, y) = \sqrt{(x - y)C^{-1}(x - y)^T}$$

where x and y are feature vectors of the same length n, and C is a covariance matrix, learnt during training for co-related features.

In this release, namely 0.2.0, it's not implemented yet, only with the covariance matrix being an identity matrix, $C = I$, making Mahalanobis distance be the same as the Euclidean one. We plan to have it fully implemented in the next release, when we decide how we go about matrix/vector operations in MARF.

0.3.4.5 Artificial Neural Network

This section presents implementation of the Neural Network Classification module.

One method of classification used is an Artificial Neural Network. Such a network is meant to represent the neuronal organization in organisms. It's use as a classification method lies is in the training of the network to output a certain value given a particular input.

0.3.4.5.1 Theory A neuron consists of a set of inputs with associated weights, a threshold, an activation function ($f(x)$) and an output value. The output value will propagate to further neurons (as input values) in the case where the neuron is not part of the "output" layer of the network. The relation of the inputs to the activation function is as follows:

$output \longleftarrow f(in)$

where $in = \sum_{i=0}^{n} (w_i \cdot a_i) - t$, "vector" a is the input activations, "vector" w is the associated weights and t is the threshold of the network. The following activation function was used:

$sigmoid(x; c) = \frac{1}{(1+e^{-cx})}$

where c is a constant. The advantage of this function is that it is differentiable over the region $(-\infty, +\infty)$ and has derivative:

$\frac{d(sigmoid(x;c))}{dx} = c \cdot sigmoid(x;c) \cdot (1 - sigmoid(x;c))$

The structure of the network used was a Feed-Forward Neural Network. This implies that the neurons are organized in sets, representing layers, and that a neuron in layer j, has inputs from layer $j-1$ and output to layer $j+1$ only. This structure facilitates the evaluation and the training of a network. For instance, in the evaluation of a network on an input vector I, the output of neuron in the first layer is calculated, followed by the second layer, and so on.

0.3.4.5.2 Training Training in a Feed-Forward Neural Network is done through the an algorithm called Back-Propagation Learning. It is based on the error of the final result of the network. The error the propagated backward throughout the network, based on the amount the neuron contributed to the error. It is defined as follows:

$w_{i,j} \longleftarrow \beta w_{i,j} + \alpha \cdot a_j \cdot \Delta_i$

where

$\Delta_i = Err_i \cdot \frac{df}{dx(in_i)}$ for neuron i in the output layer

and

$\Delta_i = \frac{df}{dt(in_i)} \cdot \sum_{j=0}^{n}(\Delta_j)$ for neurons in other layers

The parameters α and β are used to avoid local minima in the training optimization process. They weight the combination of the old weight with the addition of the new change. Usual values for these are determined experimentally.

The Back-Propagation training method was used in conjunction with epoch training. Given a set of training input vectors Tr, the Back-Propagation training is done on each run. However, the new weight vectors for each neuron, "vector" w', are stored and not used. After all the inputs in Tr have been trained, the new weights are committed and a set of test input vectors Te, are run, and a mean error is calculated. This mean error determines whether to continue epoch training or not.

0.3.4.5.3 Usage as a Classifier As a classifier, a Neural Network is used to map feature vectors to speaker identifiers. The neurons in the input layer correspond to each feature in the feature vector. The output of the network is the binary interpretation of the output layer. Therefore the Neural Network has an input layer of size m, where m is the size of all feature vectors and the output layer has size $\lceil(\log_2(n))\rceil$, where n is the maximum speaker identifier.

A network of this structure is trained with the set of input vectors corresponding to the set of training samples for each speaker. The network is epoch trained to optimize the results. This fully trained network is then used for classification in the recognition process.

0.3.4.6 Random Classification

That might sound strange, but we have a random classifier in MARF. This is more or less testing module just to quickly test the PR pipeline. It picks an ID in the pseudo-random manner from the list of trained IDs of subjects to classification. It also serves as a bottom-line of performance (i.e. recognition rate) for all the other, slightly more sophisticated classification methods meaning performance of the aforementioned methods must be better than that of the Random; otherwise, there is a problem.

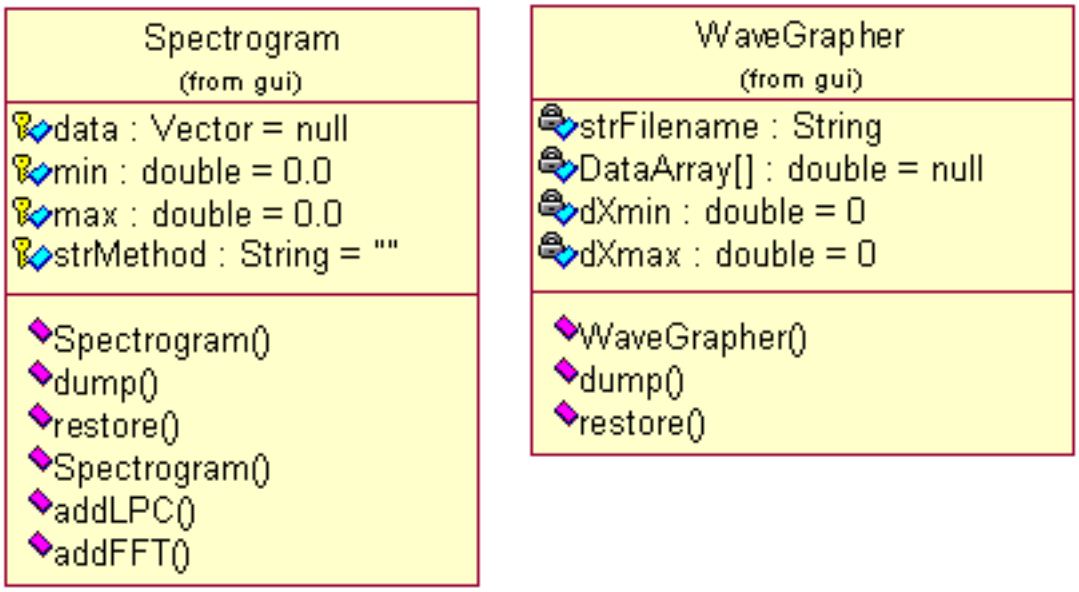

Figure 14: GUI Package

0.4 GUI

Even though this section is entitled as GUI, we don't really have any actual GUI yet (it's planned though, see TODO, 0.8.4). We do have a couple of things under the `marf.gui` package, which we do occasionaly use and evetually they will expand to be a real GUI classes. This tiny package is in Figure 14.

0.4.1 Spectrogram

Sometimes it is useful to visualize the data we are playing with. One of the typical thing when dealing with sounds, specifically voice, people are intrested in spectrograms of frequency distributions. The `Spectrogram` class was designed to handle that and produce spectrograms from both FFT and LPC algorithms and draw them. We did not manage to make it a true GUI component yet, but instead we made it to dump the spectrograms into PPM-format image files to be looked at using some graphical package. Two examples of such spectrograms are in the Appendix 0.8.1.

0.4.2 Wave Grapher

`WaveGrapher` is another class designed, as the name suggests, to draw the wave form of the incoming/preprocessed signal. Well, it doen't actually draw a thing, but dumps the sample points into a tab-delimited text file to be loaded into some plotting software, such as `gnuplot` or Excel. We also use it to produce graphs of the signal in the frequency domain istead of time domain. Examples of the graphs of data obtained via this class are in the Preprocessing Section (0.3.2).

ID	Name	Training Samples	Testing Samples
1	Serge	14	1
2	Ian	14	1
3	Steve	13	3
4	Jimmy	14	1
5	Dr. Suen	2	1
6	Margarita Mokhova	14	1
7	Alexei Mokhov	14	1
9	Graham Sinclair	12	2
10	Jihed Halimi	2	1
11	Madhumita Banerjee	3	1
13	Irina Dymova	3	1
14	Aihua Wu	14	1
15	Nick	9	1
16	Michelle Khalife	14	1
17	Shabana	7	1

Table 1: Speakers contributed their voice samples.

0.5 Sample Data and Experimentation

0.5.1 Sample Data

We have both female and male speakers, with age ranging from a college student to an university professor. The table 1 has a list of people whp have contributed their voice samples for our project (with first four being ourselves). We want to thank them once again for helping us out.

0.5.2 Comparison Setup

The main idea was to compare combinations (in MARF: *configurations*) of different methods and variations within them in terms of recognition rate performance. That means that having several preprocessing modules, several feature extraction modules, and several classification modules, we can (and did) try all their possible combinations.

That includes:

1. Preprocessing: No-filtering (just normalization), low-pass, high-pass, band-pass, and high-frequency boost filters.

2. Feature Extraction: FFT/LPC/Random algorithms comparison

3. Clasification: Distance classifiers, such as Chebyshev, Euclidean, Minkowski, and Mahalanobis distances, as well as Neural Network.

For this purpose we have written a `SpeakerIdentApp`, a command-line application for TI speaker identification. We ran it for every possible configuration with the following script, namely `testing.sh`:

```
#!/usr/bin/tcsh -f
#!/site/bin/tcsh -f

#
# Batch Processing of Training/Testing Samples
# NOTE: Make take quite some time to execute
#
# Copyright (C) 2002-2003 The MARF Development Group
#
# $Header: /cvsroot/marf/apps/SpeakerIdentApp/testing.sh,v 1.19 2003/02/05 20:27:32 mokhov Exp $
#

#
# Set environment variables, if needed
#

#setenv CLASSPATH '.:../../marf/src/marf.jar'
#setenv EXTDIRS '.:/pkg/j2sdk-1.3.1_01/jre/lib:G\:\\Programming\\Java\\jdk1.3.1_06\\jre\\lib\\ext'

#
# Set flags to use in the batch execution
#

set java = 'java'
#set debug = '-debug'
set debug = ''
set graph = ''
#set graph = '-graph'
#set spectrogram = '-spectrogram'
set spectrogram = ''

if($1 == '--reset') then
    echo "Resetting Stats..."
    $java SpeakerIdentApp --reset
    exit 0
endif

if($1 == '--retrain') then
    echo "Training..."

    # Always reset stats before retraining the whole thing
    $java SpeakerIdentApp --reset

    foreach prep (-norm -boost -low -high -band)
        foreach feat (-fft -lpc -randfe)

            # Here we specify which classification modules to use for
            # training. Since Neural Net wasn't working the default
            # distance training was performed; now we need to distinguish them
```

```
            # here. NOTE: for distance classifiers it's not important
            # which exactly it is, because the one of generic Distance is used.
            # Exception for this rule is Mahalanobis Distance, which needs
            # to learn its Covariance Matrix.

            foreach class (-cheb -mah -randcl -nn)
                echo "Config: $prep $feat $class $spectrogram $graph $debug"
                $java SpeakerIdentApp --train training-samples $prep $feat $class $spectrogram $graph $debug
            end

        end
    end

endif

echo "Testing..."

foreach file (testing-samples/*.wav)
    foreach prep (-norm -boost -low -high -band)
        foreach feat (-fft -lpc -randfe)
            foreach class (-eucl -cheb -mink -mah -randcl -nn)
                echo "=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-="
                echo "DOING FILE:"
                echo $file
                echo "Config: $prep $feat $class $spectrogram $graph $debug"
                echo "============================================="

                $java SpeakerIdentApp --ident $file $prep $feat $class $spectrogram $graph $debug

                echo "---------------------------------------------"
            end
        end
    end
end

echo "Stats:"

$java SpeakerIdentApp --stats

echo "Testing Done"

exit 0

# EOF
```

See the results section (0.6) for results analysis.

0.5.3 What Else Could/Should/Will Be Done

There is a lot more that we realistically could do, but due to lack of time, these things are not in yet. If you would like to contribute, let us know.

0.5.3.1 Combination of Feature Extraction Methods

For example, assuming we use a combination of LPC coefficients and F0 estimation, we could compare the results of different combinations of these, and discuss them later. Same with the Neural Nets (modifying number of layers and number or neurons, etc.).

We could also do a 1024 FFT analysis and compare it against a 128 FFT analysis. (That is, the size of the resulting feature vector would be 512 or 64 respectively). With LPC, one can specify the number of coefficients you want, the more you have the more precise the analysis will be.

0.5.3.2 Entire Recognition Path

`LPC` module is used to generate a mean vector of LPC coefficients for the utterance. F0 is used to find the average fundamental frequency of the utterance. The results are concatenated to form the output vector, in a particular order. The classifier would take into account the weighting of the features: Neural Network would do so implicitly if it benefits the speaker matching, and stochastic can be modified to give more weight to the F0 or vice versa, depending on what we see best (i.e.: the covariance matrix in the Mahalanobis distance (0.3.4.4)).

0.5.3.3 More Methods

Things like F0, Endpointing, Stochastic, and some other methods have not made to this release. More detailed on this aspect, please refer to the TODO list in the Appendix.

0.6 Experementation Results

0.6.1 Notes

Before we get to numbers, few notes and observations first:

1. We've got more samples since the demo. The obvious: by increasing the number of samples our results got better; with few exceptions, however. This can be explained by the diversity of the recording equipment, a lot less than uniform number of samples per speaker, and absence of noise and silence removal. All the samples were recorded in not the same environments. The results then start averaging after awhile.

2. Another observation we made from our output, is that when the speaker is guessed incorrectly, quite often the second guess is correct, so we included this in our results as if we were "guessing" right from the second attempt.

3. FUN. Interesting note, that we also tried to take some samples of music bands, and feed it to to our application along with the speakers, and application's performance didn't suffer, yet even improved because the samples were treated in the same manner. The groups were not mentioned in the table, so we name them here: Van Halen [8:1] and Red Hot Chili Peppers [10:1] (where numbers represent [training:testing] samples used).

0.6.2 Configuration Explained

Configuration parameters were exracted from the command line which SpeakerIdentApp was invoked with. They mean the following:

```
Usage:
    java SpeakerIdentApp --train <samples-dir> [options]  -- train mode
                         --ident <sample> [options]       -- identification mode
                         --stats                          -- display stats
                         --reset                          -- reset stats
                         --version                        -- display version info
                         --help                           -- display this help and exit

Options (one or more of the following):

Preprocessing:

  -norm         - use just normalization, no filtering
  -low          - use low pass filter
  -high         - use high pass filter
  -boost        - use high frequency boost filter
  -band         - use bandpass filter

Feature Extraction:

  -lpc          - use LPC
  -fft          - use FFT
  -randfe       - use random feature extraction

Classification:

  -nn           - use Neural Network
  -cheb         - use Chebyshev Distance
  -eucl         - use Euclidean Distance
  -mink         - use Minkowski Distance
  -randcl       - use random classification

Misc:

  -debug        - include verbose debug output
  -spectrogram  - dump spectrogram image after feature extraction
  -graph        - dump wave graph before preprocessing and after feature extraction
  <integer>     - expected speaker ID
```

0.6.3 Consolidated Results

Our ultimate results [1] for all configurations we can have and samples we've got are in the Table 7. Looks like our best results are with "-norm -fft -cheb", "-norm -fft -eucl", "-norm -fft -mah", "-high -fft -eucl", "-high -fft -mah" and, "-high -fft -mink" with the top result being 80%.

[1] as of Mon Feb 10 05:06:31 EST 2003

Run #	Guess	Configuration	GOOD	BAD	Recogniton Rate,%
1	1st	-band -fft -cheb	7	13	35.0
2	1st	-band -fft -eucl	9	11	45.0
3	1st	-band -fft -mah	9	11	45.0
4	1st	-band -fft -mink	6	14	30.0
5	1st	-band -fft -randcl	2	18	10.0
6	1st	-band -lpc -cheb	10	10	50.0
7	1st	-band -lpc -eucl	10	10	50.0
8	1st	-band -lpc -mah	10	10	50.0
9	1st	-band -lpc -mink	9	11	45.0
10	1st	-band -lpc -nn	0	20	0.0
11	1st	-band -lpc -randcl	3	17	15.0
12	1st	-band -randfe -cheb	2	18	10.0
13	1st	-band -randfe -eucl	2	18	10.0
14	1st	-band -randfe -mah	2	18	10.0
15	1st	-band -randfe -mink	1	19	5.0
16	1st	-band -randfe -randcl	1	19	5.0
17	1st	-boost -fft -cheb	12	8	60.0
18	1st	-boost -fft -eucl	13	7	65.0
19	1st	-boost -fft -mah	13	7	65.0
20	1st	-boost -fft -mink	12	8	60.0
21	1st	-boost -fft -randcl	1	19	5.0
22	1st	-boost -lpc -cheb	13	7	65.0
23	1st	-boost -lpc -eucl	13	7	65.0
24	1st	-boost -lpc -mah	13	7	65.0
25	1st	-boost -lpc -mink	14	6	70.0
26	1st	-boost -lpc -nn	0	20	0.0
27	1st	-boost -lpc -randcl	1	19	5.0
28	1st	-boost -randfe -cheb	5	15	25.0
29	1st	-boost -randfe -eucl	5	15	25.0

Table 2: Consolidated results, Part 1.

Run #	Guess	Configuration	GOOD	BAD	Recogniton Rate,%
30	1st	-boost -randfe -mah	5	15	25.0
31	1st	-boost -randfe -mink	4	16	20.0
32	1st	-boost -randfe -randcl	2	18	10.0
33	1st	-high -fft -cheb	15	5	75.0
34	1st	-high -fft -eucl	16	4	80.0
35	1st	-high -fft -mah	16	4	80.0
36	1st	-high -fft -mink	16	4	80.0
37	1st	-high -fft -randcl	0	20	0.0
38	1st	-high -lpc -cheb	12	8	60.0
39	1st	-high -lpc -eucl	11	9	55.00000000000001
40	1st	-high -lpc -mah	11	9	55.00000000000001
41	1st	-high -lpc -mink	9	11	45.0
42	1st	-high -lpc -nn	0	20	0.0
43	1st	-high -lpc -randcl	1	19	5.0
44	1st	-high -randfe -cheb	3	17	15.0
45	1st	-high -randfe -eucl	3	17	15.0
46	1st	-high -randfe -mah	3	17	15.0
47	1st	-high -randfe -mink	3	17	15.0
48	1st	-high -randfe -randcl	1	19	5.0
49	1st	-low -fft -cheb	15	5	75.0
50	1st	-low -fft -eucl	14	6	70.0
51	1st	-low -fft -mah	14	6	70.0
52	1st	-low -fft -mink	13	7	65.0
53	1st	-low -fft -randcl	0	20	0.0
54	1st	-low -lpc -cheb	13	7	65.0
55	1st	-low -lpc -eucl	11	9	55.00000000000001
56	1st	-low -lpc -mah	11	9	55.00000000000001
57	1st	-low -lpc -mink	11	9	55.00000000000001
58	1st	-low -lpc -nn	0	19	0.0
59	1st	-low -lpc -randcl	0	20	0.0

Table 3: Consolidated results, Part 2.

Run #	Guess	Configuration	GOOD	BAD	Recogniton Rate,%
60	1st	-low -randfe -cheb	5	15	25.0
61	1st	-low -randfe -eucl	4	16	20.0
62	1st	-low -randfe -mah	4	16	20.0
63	1st	-low -randfe -mink	5	15	25.0
64	1st	-low -randfe -randcl	2	18	10.0
65	1st	-norm -fft -cheb	16	4	80.0
66	1st	-norm -fft -eucl	16	4	80.0
67	1st	-norm -fft -mah	16	4	80.0
68	1st	-norm -fft -mink	15	5	75.0
69	1st	-norm -fft -randcl	2	18	10.0
70	1st	-norm -lpc -cheb	13	7	65.0
71	1st	-norm -lpc -eucl	13	7	65.0
72	1st	-norm -lpc -mah	13	7	65.0
73	1st	-norm -lpc -mink	14	6	70.0
74	1st	-norm -lpc -nn	1	19	5.0
75	1st	-norm -lpc -randcl	1	19	5.0
76	1st	-norm -randfe -cheb	5	15	25.0
77	1st	-norm -randfe -eucl	6	14	30.0
78	1st	-norm -randfe -mah	6	14	30.0
79	1st	-norm -randfe -mink	5	15	25.0
80	1st	-norm -randfe -randcl	1	19	5.0
81	2nd	-band -fft -cheb	13	7	65.0
82	2nd	-band -fft -eucl	13	7	65.0
83	2nd	-band -fft -mah	13	7	65.0
84	2nd	-band -fft -mink	10	10	50.0
85	2nd	-band -fft -randcl	3	17	15.0
86	2nd	-band -lpc -cheb	13	7	65.0
87	2nd	-band -lpc -eucl	12	8	60.0
88	2nd	-band -lpc -mah	12	8	60.0
89	2nd	-band -lpc -mink	12	8	60.0

Table 4: Consolidated results, Part 3.

Run #	Guess	Configuration	GOOD	BAD	Recogniton Rate,%
90	2nd	-band -lpc -nn	0	20	0.0
91	2nd	-band -lpc -randcl	3	17	15.0
92	2nd	-band -randfe -cheb	3	17	15.0
93	2nd	-band -randfe -eucl	3	17	15.0
94	2nd	-band -randfe -mah	3	17	15.0
95	2nd	-band -randfe -mink	4	16	20.0
96	2nd	-band -randfe -randcl	1	19	5.0
97	2nd	-boost -fft -cheb	15	5	75.0
98	2nd	-boost -fft -eucl	15	5	75.0
99	2nd	-boost -fft -mah	15	5	75.0
100	2nd	-boost -fft -mink	17	3	85.0
101	2nd	-boost -fft -randcl	2	18	10.0
102	2nd	-boost -lpc -cheb	15	5	75.0
103	2nd	-boost -lpc -eucl	15	5	75.0
104	2nd	-boost -lpc -mah	15	5	75.0
105	2nd	-boost -lpc -mink	16	4	80.0
106	2nd	-boost -lpc -nn	0	20	0.0
107	2nd	-boost -lpc -randcl	3	17	15.0
108	2nd	-boost -randfe -cheb	6	14	30.0
109	2nd	-boost -randfe -eucl	7	13	35.0
110	2nd	-boost -randfe -mah	7	13	35.0
111	2nd	-boost -randfe -mink	7	13	35.0
112	2nd	-boost -randfe -randcl	2	18	10.0
113	2nd	-high -fft -cheb	18	2	90.0
114	2nd	-high -fft -eucl	18	2	90.0
115	2nd	-high -fft -mah	18	2	90.0
116	2nd	-high -fft -mink	17	3	85.0
117	2nd	-high -fft -randcl	1	19	5.0
118	2nd	-high -lpc -cheb	15	5	75.0
119	2nd	-high -lpc -eucl	14	6	70.0

Table 5: Consolidated results, Part 4.

Run #	Guess	Configuration	GOOD	BAD	Recogniton Rate,%
120	2nd	-high -lpc -mah	14	6	70.0
121	2nd	-high -lpc -mink	13	7	65.0
122	2nd	-high -lpc -nn	0	20	0.0
123	2nd	-high -lpc -randcl	2	18	10.0
124	2nd	-high -randfe -cheb	4	16	20.0
125	2nd	-high -randfe -eucl	3	17	15.0
126	2nd	-high -randfe -mah	3	17	15.0
127	2nd	-high -randfe -mink	4	16	20.0
128	2nd	-high -randfe -randcl	2	18	10.0
129	2nd	-low -fft -cheb	17	3	85.0
130	2nd	-low -fft -eucl	17	3	85.0
131	2nd	-low -fft -mah	17	3	85.0
132	2nd	-low -fft -mink	17	3	85.0
133	2nd	-low -fft -randcl	1	19	5.0
134	2nd	-low -lpc -cheb	17	3	85.0
135	2nd	-low -lpc -eucl	16	4	80.0
136	2nd	-low -lpc -mah	16	4	80.0
137	2nd	-low -lpc -mink	15	5	75.0
138	2nd	-low -lpc -nn	0	19	0.0
139	2nd	-low -lpc -randcl	0	20	0.0
140	2nd	-low -randfe -cheb	7	13	35.0
141	2nd	-low -randfe -eucl	7	13	35.0
142	2nd	-low -randfe -mah	7	13	35.0
143	2nd	-low -randfe -mink	7	13	35.0
144	2nd	-low -randfe -randcl	4	16	20.0
145	2nd	-norm -fft -cheb	18	2	90.0
146	2nd	-norm -fft -eucl	18	2	90.0
147	2nd	-norm -fft -mah	18	2	90.0
148	2nd	-norm -fft -mink	17	3	85.0
149	2nd	-norm -fft -randcl	5	15	25.0

Table 6: Consolidated results, Part 5.

Run #	Guess	Configuration	GOOD	BAD	Recogniton Rate,%
150	2nd	-norm -lpc -cheb	16	4	80.0
151	2nd	-norm -lpc -eucl	15	5	75.0
152	2nd	-norm -lpc -mah	15	5	75.0
153	2nd	-norm -lpc -mink	17	3	85.0
154	2nd	-norm -lpc -nn	1	19	5.0
155	2nd	-norm -lpc -randcl	1	19	5.0
156	2nd	-norm -randfe -cheb	8	12	40.0
157	2nd	-norm -randfe -eucl	9	11	45.0
158	2nd	-norm -randfe -mah	9	11	45.0
159	2nd	-norm -randfe -mink	8	12	40.0
160	2nd	-norm -randfe -randcl	2	18	10.0

Table 7: Consolidated results, Part 6.

0.7 Conclusions

So, our best configuration yielded 80% correctness of our work when identifying subjects. Having a total of 15 speakers (well, and two music bands) that means 13-14 subjects identified correctly out of 17 per run.

The main reasons the recognition rate could be that low is due to ununiform sample taking, lack of good preprocessing techniques, such as noise/silence removal, and lack of sophisticated classification modules (e.g. Stochastic models).

Even though for commercial and University-level research standards 80% recognition rate is considered to be very low as opposed to a required minimum of 95%-97% and above, we think it is still reasonably well for a one-semester school project. That still involved a substantial amount of research and findings considering our workload and lack of experience in the area.

We would like to thank Dr. Suen and Mr. Sadri for the course and help provided.

Bibliography

[1] O'Shaughnessy, Douglas. (2000), "Speech Communications", IEEE Press. New Jersey, US.

[2] Sprenger, S. (1999), "The DFT â pied", <http://www.dspdimension.com/html/dftapied.html>

[3] Ifeachor/Jervis (2002) "Digital Signal Processing", Prentice Hall. New Jersey, US.

[4] William H. Press, et al. (1993), "Numerical Recipes in C", 2^{nd} edition, Cambridge University Press. Cambridge, UK.

[5] Russell, S., Norvig, P. (1995), "Artificial Intelligence: A Modern Approach", Prentice Hall. New Jersey, US.

[6] Flanagan D. (1997), "Java in a Nutshell", 2^{nd} Edition, O'Reily & Associates, Inc.. CA, US, ISBN: 1-56592-262-X

[7] Sun Microsystems, Inc., "The Java Website", <http://java.sun.com>

0.8 APPENDIX

0.8.1 Spectrogram Examples

As produced by the `Spectrogram` class.

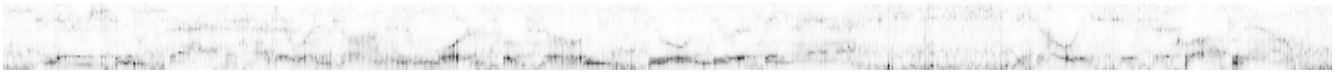

Figure 15: LPC spectrogram obtained for ian15.wav

Figure 16: LPC spectrogram obtained for graham13.wav

0.8.2 MARF Source Code

You can download the code from `<http://marf.sourceforge.net>`, specifically:

- The latest unstable version: `<http://marf.sourceforge.net/marf.tar.gz>`
- Browse code and revision history online: `<http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/marf/>`

API documentation in the HTML format can be found in the documentation distribution, or for the latest version please consult: `<http://marf.sourceforge.net/api/>`. If you want to participate in developement, there is a developers version of the API: `<http://marf.sourceforge.net/api-dev/>`, which includes all the private constructs into the docs as well.

0.8.3 SpeakerIdentApp and SpeakersIdentDb Source Code

0.8.3.1 SpeakerIdentApp.java

```
/*
 * SpeakerIdentApp: Indentfies a speaker regardless what speaker says.
 *
 * $Header: /cvsroot/marf/apps/SpeakerIdentApp/SpeakerIdentApp.java,v 1.26 2003/02/08 16:50:16 mokhov Exp $
 */

import marf.*;
import marf.util.*;
import marf.Storage.*;

import java.io.*;

/**
 * Class SpeakerIdentApp
 * <p>Indentfies a speaker independently of text, based on the MARF framework</p>
 */
public class SpeakerIdentApp
{
    /*
     * Versionning
     */
    public static final int MAJOR_VERSION = 0;
    public static final int MINOR_VERSION = 2;
    public static final int REVISION      = 0;

    public static void main(String argv[])
    {
        // Since new API was introduced in 0.2.0
        validateVersions();

        /*
         * Database of speakers
         */
        SpeakersIdentDb db = new SpeakersIdentDb("speakers.txt");

        try
        {
            db.connect();
            db.query();

            /*
             * If supplied in the command line,
             * the system when classifying will output next
             * to the guessed one
             */
            int iExpectedID = -1;

            /*
             * Default MARF setup
             */

            MARF.setPreprocessingMethod(MARF.DUMMY);
```

```
MARF.setFeatureExtractionMethod(MARF.FFT);
MARF.setClassificationMethod(MARF.EUCLIDEAN_DISTANCE);
MARF.setDumpSpectrogram(false);
MARF.setSampleFormat(MARF.WAV);

MARF.DEBUG = false;

// parse extra arguments
// XXX: maybe it's time to move it to a sep. method
for(int i = 2; i < argv.length; i++)
{
    try
    {
        // Preprocessing

        if(argv[i].compareTo("-norm") == 0)
            MARF.setPreprocessingMethod(MARF.DUMMY);

        else if(argv[i].compareTo("-boost") == 0)
            MARF.setPreprocessingMethod(MARF.HIGH_FREQUENCY_BOOST_FFT_FILTER);

        else if(argv[i].compareTo("-high") == 0)
            MARF.setPreprocessingMethod(MARF.HIGH_PASS_FFT_FILTER);

        else if(argv[i].compareTo("-low") == 0)
            MARF.setPreprocessingMethod(MARF.LOW_PASS_FFT_FILTER);

        else if(argv[i].compareTo("-band") == 0)
            MARF.setPreprocessingMethod(MARF.BANDPASS_FFT_FILTER);

        // Feature Extraction

        else if(argv[i].compareTo("-fft") == 0)
            MARF.setFeatureExtractionMethod(MARF.FFT);

        else if(argv[i].compareTo("-lpc") == 0)
            MARF.setFeatureExtractionMethod(MARF.LPC);

        else if(argv[i].compareTo("-randfe") == 0)
            MARF.setFeatureExtractionMethod(MARF.RANDOM_FEATURE_EXTRACTION);

        // Classification

        else if(argv[i].compareTo("-nn") == 0)
        {
            MARF.setClassificationMethod(MARF.NEURAL_NETWORK);

            ModuleParams m = new ModuleParams();

            // Dump/Restore Format of the TrainingSet
            m.addClassificationParam(new Integer(TrainingSet.DUMP_GZIP_BINARY));

            // Training Constant
            m.addClassificationParam(new Double(1.0));

            // Epoch number
            //m.addClassificationParam(new Integer(100));
```

```
            m.addClassificationParam(new Integer(1000));

            // Min. error
            //m.addClassificationParam(new Double(0.01));
            m.addClassificationParam(new Double(4.27));

            MARF.setModuleParams(m);
        }

        else if(argv[i].compareTo("-eucl") == 0)
            MARF.setClassificationMethod(MARF.EUCLIDEAN_DISTANCE);

        else if(argv[i].compareTo("-cheb") == 0)
            MARF.setClassificationMethod(MARF.CHEBYSHEV_DISTANCE);

        else if(argv[i].compareTo("-mink") == 0)
        {
            MARF.setClassificationMethod(MARF.MINKOWSKI_DISTANCE);

            ModuleParams m = new ModuleParams();

            // Dump/Restore Format
            m.addClassificationParam(new Integer(TrainingSet.DUMP_GZIP_BINARY));

            // Minkowski Factor
            m.addClassificationParam(new Double(6.0));

            MARF.setModuleParams(m);
        }

        else if(argv[i].compareTo("-mah") == 0)
            MARF.setClassificationMethod(MARF.MAHALANOBIS_DISTANCE);

        else if(argv[i].compareTo("-randcl") == 0)
            MARF.setClassificationMethod(MARF.RANDOM_CLASSIFICATION);

        // Misc

        else if(argv[i].compareTo("-spectrogram") == 0)
            MARF.setDumpSpectrogram(true);

        else if(argv[i].compareTo("-debug") == 0)
            MARF.DEBUG = true;

        else if(argv[i].compareTo("-graph") == 0)
            MARF.setDumpWaveGraph(true);

        else if(Integer.parseInt(argv[i]) > 0)
            iExpectedID = Integer.parseInt(argv[i]);
    }
    catch(NumberFormatException e)
    {
        // Number format exception should be ignored
        // XXX [SM]: Why?

        MARF.debug("SpeakerIdentApp.main() - NumberFormatException: " + e.getMessage());
    }
```

```
} // extra args

/*
 * Identification
 */
if(argv[0].compareTo("--ident") == 0)
{
    if(iExpectedID < 0)
        iExpectedID = db.getIDByFilename(argv[1], false);

    // Store config and error/sucesses for that config
    String strConfig = "";

    if(argv.length > 2) // Get config from the command line
        for(int i = 2; i < argv.length; i++)
            strConfig += argv[i] + " ";

    else // query MARF for it's current config
        strConfig = MARF.getConfig();

    MARF.setSampleFile(argv[1]);
    MARF.recognize();

    db.getName(MARF.queryResultID());

    System.out.println("               Config: " + strConfig);
    System.out.println("         Speaker's ID: " + MARF.queryResultID());
    System.out.println("   Speaker identified: " + db.getName(MARF.queryResultID()));

    if(iExpectedID > 0)
    {
        // Second best
        Result oResult = MARF.getResult();

        System.out.println("Expected Speaker's ID: " + iExpectedID);
        System.out.println("     Expected Speaker: " + db.getName(iExpectedID));
        System.out.println("       Second Best ID: " + oResult.getSecondClosestID());
        System.out.println("     Second Best Name: " + db.getName(oResult.getSecondClosestID()));

        db.restore();
        {
            // 1st match
            db.addStats(strConfig, (MARF.queryResultID() == iExpectedID ? true : false));

            // 2nd best: must be true if either 1st true or second true (or both :))
            boolean bSecondBest =
                MARF.queryResultID() == iExpectedID
                ||
                oResult.getSecondClosestID() == iExpectedID;

            db.addStats(strConfig, bSecondBest, true);
        }
        db.dump();
    }
}
```

```
/*
 * Training
 */
else if(argv[0].compareTo("--train") == 0)
{
    try
    {
        File[] dir = new File(argv[1]).listFiles();

        String strFileName;

        // XXX: this loop has to be in MARF
        for(int i = 0; i < dir.length; i++)
        {
            strFileName = dir[i].getPath();

            if(strFileName.endsWith(".wav"))
            {
                MARF.setSampleFile(strFileName);

                int id = db.getIDByFilename(strFileName, true);

                if(id == -1)
                    System.out.println("No speaker found for \"" + strFileName + "\".");
                else
                {
                    MARF.setCurrentSubject(id);
                    MARF.train();
                }
            }
        }
    }
    catch(NullPointerException e)
    {
        System.out.println("Folder \"" + argv[1] + "\" not found.");
        System.exit(-1);
    }

    System.out.println("Done training on folder \"" + argv[1] + "\".");
}

/*
 * Stats
 */
else if(argv[0].compareTo("--stats") == 0)
{
    db.restore();
    db.printStats();
}

/*
 * Reset Stats
 */
else if(argv[0].compareTo("--reset") == 0)
{
    db.resetStats();
    System.out.println("SpeakerIdentApp: Statistics has been reset.");
```

```
        }

        /*
         * Versionning
         */
        else if(argv[0].compareTo("--version") == 0)
        {
            System.out.println("Text-Independent Speaker Identification Application, v." + getVersion());
            System.out.println("Using MARF, v." + MARF.getVersion());

            validateVersions();
        }

        /*
         * Help
         */
        else if(argv[0].compareTo("--help") == 0)
        {
            usage();
        }

        /*
         * Invalid major option
         */
        else
            throw new Exception("Unrecognized option: " + argv[0]);
    }

    /*
     * No arguments have been specified
     */
    catch(ArrayIndexOutOfBoundsException e)
    {
        System.err.println(e.getMessage());
        e.printStackTrace(System.err);
        usage();
    }

    /*
     * MARF-specific errors
     */
    catch(MARFException e)
    {
        System.err.println(e.getMessage());
        e.printStackTrace(System.err);
    }

    /*
     * Invalid option and/or option argument
     */
    catch(Exception e)
    {
        System.out.println(e.getMessage());
        e.printStackTrace(System.err);
        usage();
    }
```

```
        /*
         * Regardless whatever happens, close the db connection.
         */
        finally
        {
            try
            {
                MARF.debug("Closing DB connection...");
                db.close();
            }
            catch(Exception e)
            {
                MARF.debug("Closing DB connection failed: " + e.getMessage());
                e.printStackTrace(System.err);
                System.exit(-1);
            }
        }
    }

    private static final void usage()
    {
        System.out.println
        (
            "Usage:\n" +
            "    java SpeakerIdentApp --train <samples-dir> [options]  -- train mode\n" +
            "                         --ident <sample> [options]       -- identification mode\n" +
            "                         --stats                          -- display stats\n" +
            "                         --reset                          -- reset stats\n" +
            "                         --version                        -- display version info\n" +
            "                         --help                           -- display this help and exit\n\n" +

            "Options (one or more of the following):\n\n" +

            "Preprocessing:\n\n" +
            "  -norm         - use just normalization, no filtering\n" +
            "  -low          - use low pass filter\n" +
            "  -high         - use high pass filter\n" +
            "  -boost        - use high frequency boost filter\n" +
            "  -band         - use bandpass filter\n" +
            "\n" +

            "Feature Extraction:\n\n" +
            "  -lpc          - use LPC\n" +
            "  -fft          - use FFT\n" +
            "  -randfe       - use random feature extraction\n" +
            "\n" +

            "Classification:\n\n" +
            "  -nn           - use Neural Network\n" +
            "  -cheb         - use Chebyshev Distance\n" +
            "  -eucl         - use Euclidean Distance\n" +
            "  -mink         - use Minkowski Distance\n" +
            "  -randcl       - use random classification\n" +
            "\n" +

            "Misc:\n\n" +
            "  -debug        - include verbose debug output\n" +
```

```
            "  -spectrogram  - dump spectrogram image after feature extraction\n" +
            "  -graph        - dump wave graph before preprocessing and after feature extraction\n" +
            "  <integer>     - expected speaker ID\n" +
            "\n"
        );

        System.exit(0);
    }

    private static final String getVersion()
    {
        return MAJOR_VERSION + "." + MINOR_VERSION + "." + REVISION;
    }

    public static final int getIntVersion()
    {
        return MAJOR_VERSION * 100 + MINOR_VERSION * 10 + REVISION;
    }

    public static final void validateVersions()
    {
        if(MARF.getIntVersion() < (0 * 100 + 2 * 10 + 0))
        {
            System.out.println
            (
                "Your MARF version (" + MARF.getVersion() +
                ") is too old. This application requires 0.2.0 or above."
            );
        }
    }
}

// EOF
```

0.8.3.2 SpeakersIdentDb.java

```
import marf.*;

import java.io.*;
import java.util.*;
import java.util.zip.*;
import java.awt.*;

/**
 * Class SpeakersIdentDb
 * Manages database of speakers on the app. level
 */
public class SpeakersIdentDb implements Serializable
{
    /**
     * DB filename
     */
    private String strDbFile;
```

```
/**
 * Hashes "config string" -> Vector(FirstMatchPoint(XSuccesses, YFailures), SecondMatchPoint(XSuccesses, YFailures))
 */
private Hashtable oStatsPerConfig = null;

/**
 * A vector of vectors of speakers info pre-loded on connect()
 */
private Hashtable oDB = null;

/**
 * Indicate whether we are connected or not
 */
boolean bConnected = false;

/**
 * "Database connection"
 */
BufferedReader br = null;

/**
 * @param pstrFileName filename of a CSV file with IDs and names of speakers
 */
public SpeakersIdentDb(String pstrFileName)
{
    this.strDbFile = pstrFileName;
    this.oDB = new Hashtable();
    this.oStatsPerConfig = new Hashtable();
}

/**
 * @param strFileName Name of a .wav file for which ID must be returned
 * @return int ID
 */
public int getIDByFilename(String pstrFileName, boolean pbTraining) throws Exception
{
    String str;

    // Extract actual file name without preceeding path (if any)
    if(pstrFileName.lastIndexOf('/') >= 0)
        str = pstrFileName.substring(pstrFileName.lastIndexOf('/') + 1, pstrFileName.length());
    else if(pstrFileName.lastIndexOf('\\') >= 0)
        str = pstrFileName.substring(pstrFileName.lastIndexOf('\\') + 1, pstrFileName.length());
    else
        str = pstrFileName;

    Enumeration oIDs = oDB.keys();

    // Traverse all the info vectors looking for sample filename
    while(oIDs.hasMoreElements())
    {
        Integer id = (Integer)oIDs.nextElement();

        //System.out.println("File: " + pstrFileName + ", id = " + id.intValue());

        Vector oSpeakerInfo = (Vector)oDB.get(id);
        Vector oFilenames;
```

```
            if(pbTraining == true)
                oFilenames = (Vector)oSpeakerInfo.elementAt(1);
            else
                oFilenames = (Vector)oSpeakerInfo.elementAt(2);

            // Start from 1 because 0 is speaker's name
            for(int i = 0; i < oFilenames.size(); i++)
            {
                String tmp = (String)oFilenames.elementAt(i);

                if(tmp.compareTo(str) == 0)
                    return id.intValue();
            }
        }

        return -1;
    }

    /**
     * @param piID ID of a person in the DB to return a name for
     * @return name string
     * @exception Exception
     */
    public final String getName(final int piID) throws Exception
    {
        //MARF.debug("getName() - ID = " + piID + ", db size: " + oDB.size());
        String strName;

        Vector oDBEntry = (Vector)oDB.get(new Integer(piID));

        if(oDBEntry == null)
            strName = "Unknown Speaker (" + piID + ")";
        else
            strName = (String)oDBEntry.elementAt(0);

        return strName;
    }

    public void connect() throws Exception
    {
        // That's where we should establish file linkage and keep it until closed
        try
        {
            this.br = new BufferedReader(new FileReader(this.strDbFile));
            this.bConnected = true;
        }
        catch(IOException e)
        {
            throw new Exception
            (
                "Error opening speaker DB: \"" + this.strDbFile + "\": " +
                e.getMessage() + "."
            );
        }
    }
```

```
public void query() throws Exception
{
    // That's where we should load db results into internal data structure

    String tmp;
    int id = -1;

    try
    {
        tmp = br.readLine();

        while(tmp != null)
        {
            StringTokenizer stk = new StringTokenizer(tmp, ",");
            Vector oSpeakerInfo = new Vector();

            // get ID
            if(stk.hasMoreTokens())
                id = Integer.parseInt(stk.nextToken());

            // speaker's name
            if(stk.hasMoreTokens())
            {
                tmp = stk.nextToken();
                oSpeakerInfo.add(tmp);
            }

            // training file names
            Vector oTrainingFilenames = new Vector();

            if(stk.hasMoreTokens())
            {
                StringTokenizer oSTK = new StringTokenizer(stk.nextToken(), "|");

                while(oSTK.hasMoreTokens())
                {
                    tmp = oSTK.nextToken();
                    oTrainingFilenames.add(tmp);
                }
            }

            oSpeakerInfo.add(oTrainingFilenames);

            // testing file names
            Vector oTestingFilenames = new Vector();

            if(stk.hasMoreTokens())
            {
                StringTokenizer oSTK = new StringTokenizer(stk.nextToken(), "|");

                while(oSTK.hasMoreTokens())
                {
                    tmp = oSTK.nextToken();
                    oTestingFilenames.add(tmp);
                }
            }
```

```
            oSpeakerInfo.add(oTestingFilenames);

            MARF.debug("Putting ID=" + id + " along with info vector of size " + oSpeakerInfo.size());

            this.oDB.put(new Integer(id), oSpeakerInfo);

            tmp = br.readLine();
        }
    }
    catch (IOException e)
    {
        throw new Exception
        (
            "Error reading from speaker DB: \"" + this.strDbFile +
            "\": " + e.getMessage() + "."
        );
    }
}

public void close() throws Exception
{
    // Close file
    if(bConnected == false)
        throw new Exception("SpeakersIdentDb.close() - not connected");

    try
    {
        this.br.close();
        this.bConnected = false;
    }
    catch(IOException e)
    {
        throw new Exception(e.getMessage());
    }
}

public void addStats(String pstrConfig, boolean pbSuccess)
{
    addStats(pstrConfig, pbSuccess, false);
}

public void addStats(String pstrConfig, boolean pbSuccess, boolean pbSecondBest)
{
    Vector oMatches = (Vector)oStatsPerConfig.get(pstrConfig);
    Point oPoint = null;

    if(oMatches == null)
    {
        oMatches = new Vector(2);
        oMatches.add(new Point());
        oMatches.add(new Point());
    }
    else
    {
        if(pbSecondBest == false)
            oPoint = (Point)oMatches.elementAt(0); // Firts match
        else
```

```
                oPoint = (Point)oMatches.elementAt(1); // Second best match
        }

        int x = 0; // # of successes
        int y = 0; // # of failures

        if(oPoint == null) // Didn't exist yet; create new
        {
            if(pbSuccess == true)
                x = 1;
            else
                y = 1;

            oPoint = new Point(x, y);

            if(oPoint == null)
            {
                System.out.println("point null!!!!");
                System.exit(-1);
            }

            if(oMatches == null)
            {
                System.out.println("matches null!!!!");
                System.exit(-1);
            }

            if(oMatches.size() == 0)
            {
                System.out.println("matches 0!!!!");
                System.exit(-1);
            }

            if(pbSecondBest == false)
                oMatches.setElementAt(oPoint, 0);
            else
                oMatches.setElementAt(oPoint, 1);

            oStatsPerConfig.put(pstrConfig, oMatches);
        }

        else // There is an entry for this config; update
        {
            if(pbSuccess == true)
                oPoint.x++;
            else
                oPoint.y++;
        }
    }

    public final void printStats() throws Exception
    {
        String[]    astrResults = new String[oStatsPerConfig.size() * 2];
        Enumeration oStatsEnum  = oStatsPerConfig.keys();

        int iResultNum = 0;
```

```
        while(oStatsEnum.hasMoreElements())
        {
            String strConfig = (String)oStatsEnum.nextElement();

            for(int i = 0; i < 2; i++)
            {
                Point oGoodBadPoint = (Point)((Vector)oStatsPerConfig.get(strConfig)).elementAt(i);
                String strGuess = (i == 0) ? "1st" : "2nd";

                astrResults[iResultNum++] =
                    strGuess + " " +
                    "CONFIG: " + strConfig +
                    ", GOOD: " + oGoodBadPoint.x +
                    ", BAD: " + oGoodBadPoint.y +
                    ", %: " + ((double)oGoodBadPoint.x / (double)(oGoodBadPoint.x + oGoodBadPoint.y)) * 100;
            }
        }

        Arrays.sort(astrResults);

        for(int i = 0; i < astrResults.length; i++)
            System.out.println(astrResults[i]);
    }

    public final void resetStats() throws IOException
    {
        oStatsPerConfig.clear();
        dump();
    }

    public void dump() throws IOException
    {
        FileOutputStream fos = new FileOutputStream(this.strDbFile + ".stats");
        GZIPOutputStream gzos = new GZIPOutputStream(fos);
        ObjectOutputStream out = new ObjectOutputStream(gzos);

        out.writeObject(this.oStatsPerConfig);
        out.flush();
        out.close();
    }

    public void restore() throws IOException
    {
        try
        {
            FileInputStream fis = new FileInputStream(this.strDbFile + ".stats");
            GZIPInputStream gzis = new GZIPInputStream(fis);
            ObjectInputStream in = new ObjectInputStream(gzis);

            this.oStatsPerConfig = (Hashtable)in.readObject();
            in.close();
        }
        catch(FileNotFoundException e)
        {
            System.out.println("NOTICE: File " + this.strDbFile + ".stats does not seem to exist. Creating a new one....");
            resetStats();
        }
```

```
        catch(ClassNotFoundException e)
        {
            throw new IOException("SpeakerIdentDd.retore() - ClassNotFoundException: " + e.getMessage());
        }
    }
}

// EOF
```

0.8.4 TODO

```
MARF TODO/Wishlist
------------------

$Header: /cvsroot/marf/marf/TODO,v 1.36 2003/02/10 09:56:35 mokhov Exp $


THE APPS

- SpeakerIdentApp
  - GUI
  - Real Time recording (does it belong here?)
  - Move dir. read from the app to MARF in training section {0.3.0}
  - Enhance batch recognition (do not re-load training set per sample) {0.3.0}
  - Add --batch-ident option {0.3.0}
  - Enhance options with arguments, e.g. -fft=1024, -lpc=40, -lpc=40,256, keeping the existing defaults
  - Add option: -data-dir=DIRNAME other than default to specify a dir where to store training sets and stuff
  - Add -mah=r
  - Add single file training option
  - make binary/optimized distro
  - Dump stats: -classic -latex -csv
  - Improve on javadoc
  - ChangeLog
  * Sort the stats
  * Add classification methods to training in testing.sh

- SpeechRecognition

- InstrumentIdentification

- LanguageIdentification

+- Fix TestNN

- Regression Tests?
  - one script calls all the apps and compares new results vs. expected

* batch plan execution


THE BUILD SYSTEM

  * global makefile in /marf
  * fix doc's global makefile
  * Global Makefile for apps
    (will descend to each dir and build all the apps.)
  - Build and package distrubution
    - MARF
    - App
  - Perhaps at some point we'd need make/project files for other Java IDEs, such as
    - Sun Studio (Forte is dead)
    - IBM Visual Age
    ? Visual J++

THE FRAMEWORK
```

```
- Preprocessing
  * Move BandPassFilter and HighFrequencyBoost under FFTFilter package with CVS comments
  * Tweak the filter values of HighPass and HighFrequencyBoost filters
  - Make dump()/restore() to serialize filtered output {0.3.0}
  - Implement
    - Enable changing values of frequency boundaries and coeffs. in filters by an app.
    - Endpoint {1.0.0}
    * Bandpass filter {0.2.0}
    - Highpass Filter with High Frequency Boost together {0.?.0}
    - "Compressor" [steve]
    - Methods:  {1.0.0}
      - removeNoise()
      - removeSilence()
      - cropAudio()

- Feature Extraction
  - Make modules to dump their
    features for future use by NNet and maybe others {0.3.0}
  - Implement {1.0.0}
    - F0
    - Cepstral
    - Segmentation
    * RandomFeatureExtraction {0.2.0}

- Classification
  - Implement
    * Minkowski's Distance {0.2.0}
    +---- Mahalanobis Distance {0.3.0}
    - Stochastic [serge] {1.0.0}
      - Gaussian Mixture Models
    - Hidden Markov Models {1.0.0}
    * RandomClassification {0.2.0}

  - Fix and document NNet {0.*.0}
    * dump()/retore() {0.2.0}
    - add % of correct/incorrect expected to train() {0.3.0}
    - ArrayList ---> Vector, because ArrayList is NOT thread-safe {0.3.0}
    +- Epoch training

  - Distance Classifiers
    - make distance() throw an exception maybe?
    * Move under Distance package

- Speech package
  - Recognition
  - Dictionaries
  - Generation

- Stats {0.3.0}
  - Move stats collection from the app and other places to StatsCollector
  - Timing
  - Batch progress report

- Algos
  - Algorithm decoupling to marf.algos or marf.algorithms or ... {0.4.0}
  - marf.algos.Search
```

```
  - marf.util.DataStructures -- Node / Graph --- to be used by the networks and state machines
  - move hamming() from FeatureExtraction

- GUI {0.5.0}
  - Make them actual GUI components to be included into App
    - Spectrogram
      * Fix filename stuff (module_dirname to module_filename)
    - WaveGrapher
  - Fix WaveGrapher
    - Sometimes dumps files of 0 length
    - Make it actually ouput PPM or smth like that (configurable?)
    - Too huge files for samp output.
    - Have LPC understand it
  - Config tool
  - Web interface?

- MARF.java
  - Concurrent use of modules of the same type (e.g. FFT and F0)
  - impl
    ? streamedRecognition()
    +- train()
       - Add single file training
    - Inter-module compatibility (i.e. specific modules can only work
      with specific modules and not others)
      - Module Compatibility Matrix
      - Module integer and String IDs
  +- enhance error reporting
  - Server Part {2.0.0}

- Exceptions {0.3.0}
  - StorageException
  ? Have all marf exceptions inherit from util.MARFException

- marf.util
  - Move NeuralNetowork.indent()
  - Move MARF.debug() --> marf.util.debug.debugln()
  - marf.util.debug.debug()
  ? marf.util.upgrade

- Storage
  - ModuleParams: have Hastables instead of Vectors
    to allow params in any order and in any number.
  - Keep all data files under marf.data dir, like training sets, XML, etc {0.3.0}
  - DUMP_BINARY (w/o compression) {0.3.0}
  - Move DUMP_* flags up to StorageManager?
  +- Revise TrainingSet stuff
    - TrainingSet
      - upgradability {?.?.?}
      - convertability: bin.gzip <-> bin <-> csv
  - FeatureSet {0.3.0}

- Clean up
  - CVS:
    - Remove --x permissions introduced from windoze in files:
      - /marf/doc/src/tex/sampleloading.tex
      - /marf/doc/src/graphics/*.png
      - /marf/doc/src/graphics/arch/*.png
```

```
      - /marf/doc/src/graphics/fft_spectrograms/*.ppm
      - /marf/doc/src/graphics/lpc_spectrograms/*.ppm
      - /marf/doc/arch.mdl
      - /marf/src/marf/Classification/Distance/EuclideanDistance.java
      - /marf/src/marf/Preprocessing/FFTFilter.java
      - /apps/SpeakerIdentApp/SpeakerIdentApp.jpx
      - /apps/SpeakerIdentApp/testing-samples/*.wav
      - /apps/SpeakerIdentApp/testing-samples/*.wav
      - /apps/TestFilters/TestFilters.*
    - Rename /marf/doc/sgml to /marf/doc/styles
    - Remove /marf/doc/styles/ref
    * Move disatance classifiers with CVS log
      to Distance
    * remove uneeded attics and corresponding dirs
      * "Ceptral"
      * Bogus samples

THE CODE

- Define coding standards
- Propagate them throughout the code

THE DOCS

- docs [s]
  - autosync history and the report
  - ChangeLog
  * report components [serge]
  - Arch Update [serge]
    +- gfx model (rr)
      - gui: add StorageManager
    * update doc
    * newer images
  * better doc format and formulas
  - index
  - Results:
    - Add modules params used, like r=6 in Minkowski, FFT input 1024, etc

* web site
  * CVS
  * autoupdate from CVS

EOF
```