AFARR: A FRAMEWORK FOR AUTOMATIC SPEECH RECOGNITION RESEARCH

By

RICHARD L LYNCH

B.S. University of New Hampshire, 2001

THESIS

Submitted to the University of New Hampshire

in Partial Fulfillment of

the Requirements for the Degree of

Master of Science

in

Electrical Engineering

September, 2003

This thesis has been examined and approved.

Thesis Director, Dr. Andrew L. Kun
Assistant Professor of Electrical Engineering

Dr. W. Thomas Miller, III
Professor of Electrical Engineer

Dr. William Lenharth
Associate Research Professor of Electrical Engineering

Date

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Andrew L. Kun, for his time, advice, guidance, and suggestions throughout my research.

I would also like to thank Dr. William Lenharth and Dr. W. Thomas Miller, III for serving on my thesis committee and taking the time and effort to review my thesis and provide feedback.

I would especially like to thank Michael Bressack, Dr. Andrew L. Kun, Michael Martin, John Mock, and Jeremy Saunders for their time and considerable patience in recording speech samples.

I would like to thank Kelly for her constant support and reassurances that this thesis would some day be completed. I would also like to thank Mike C. for providing an amusing diversion during my thesis.

Finally, I would like to thank my family and friends for their constant support throughout my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

LIST OF ACRONYMS

AFARR ..... A Framework for Automatic Speech Recognition Research

API ..... Application Programming Interface

ASR ..... Automatic Speech Recognition

CMU ..... Carnegie Mellon University

CSLU ..... Center for Spoken Language Understanding

CSV ..... Comma Separated Values

DDX ..... Dynamic Data eXchange

DFT ..... Discrete Fourier Transform

DSP ..... Digital Signal Processing

DTW ..... Dynamic Time Warping

EIH ..... Ensemble Interval Histogram

FFT ..... Fast Fourier Transform

FIR ..... Finite Impulse Response

LGPL ..... Lesser General Public License

GUI ..... Graphical User Interface

HMM ..... Hidden Markov Model

HTK ..... Hidden Markov Model ToolKit

HTML ..... HyperText Markup Language

IDFT ..... Inverse Discrete Fourier Transform

IFCs ..... ISIP Foundation Classes

IIR ..... Infinite Impulse Response

I/O ..... Input/Output

MAP ..... Maximum A Posteriori

MFC ..... Microsoft Foundation Classes

MFCC ..... Mel Frequency Cepstral Coefficients

MLE ..... Maximum Likelihood Estimation

NN ..... Neural Network

OOP ..... Object Oriented Programming

PCM ..... Pulse Coded Modulation

PDF ..... Probability Density Function

PDF ..... Portable Document Format

PRNG ..... Pseudo Random Number Generator

RASTA ..... RelAtive SpecTrAl Processing

RAM ..... Random Access Memory

RES ..... Recognition Experimentation System

RGB ..... Red Green Blue

VQ ..... Vector Quantization

VTLN ..... Vocal Tract Length Normalization

WER ..... Word Error Rate

ABSTRACT

AFARR: A FRAMEWORK FOR AUTOMATIC SPEECH RECOGNITION RESEARCH

by

Richard L. Lynch

University of New Hampshire, September, 2003

Speech recognition is a large multidisciplinary problem. When implementing a speech recognition engine, one runs into many interrelated tasks – input/output, speech corpus management, linear algebra, neural networks, Hidden Markov Models, feature extraction, distance measures, and endpoint detection, just to name a few. All of these individual tasks must be implemented to perform speech recognition, but often times a researcher is only interested in working on a single task, for instance endpoint detection. If the researcher does not want to spend the time to implement all of the other tasks, he or she typically turns to an existing speech recognition research platform.

An ideal speech recognition research platform would be modular, flexible, easy to learn and use, well documented, and run at a reasonable speed. Prior to this thesis, such a system did not exist. Many packages were available, but all were deficient in some area. The most common failing was in the documentation area – many research platforms had poor or non-existent source code documentation.

A Framework for Automatic Speech Recognition Research (AFARR) attempts to fill in this gap in the speech recognition field. AFARR was designed from the ground up to be powerful, yet easy to understand. AFARR was written in C++ and made heavy use of object oriented programming to package related functions and variables together. The source code was heavily documented. The source code has also been processed by Doxygen to produce Portable Document Format (PDF), Hypertext Markup Language (HTML), and CHM (Windows Help File) format documentation.

Three reference speech recognition engines have been implemented in AFARR – one Hidden Markov Model, one neural network, and one dynamic time warping engine.

Speaker dependent accuracy was reasonable, reaching 100% for the dynamic time warping speech recognition engine, 99.0% for the Hidden Markov Model engine, and 95.0% for the neural network engine.

It is hoped that AFARR can serve as a starting point from which future research can be performed.

CHAPTER 1

INTRODUCTION

1.1. Problem Description

Automatic speech recognition (ASR) research has been a difficult field to begin working in. Before researchers could accomplish any ASR research, they had to go through the arduous task of obtaining a complete ASR system. A researcher could accomplish this by either building an ASR system from scratch, or by acquiring an existing ASR system.

Implementing a complete ASR system from scratch would allow excellent control over design parameters, but would also be a very time consuming approach. There are many tasks that must be performed by a speech recognition system, including handling different types of audio files, presenting a user interface, acoustic preprocessing, acoustic modeling, endpoint detection, logging facilities, and linear algebra functions. Implementing all of these functions into a cohesive system would be a challenging and time consuming process.

Alternately, an existing ASR system and its source code could be acquired. For the source code to be useful, it would need to be well documented. Ideally, general documentation of the system architecture and inline documentation in the source code would be present. It would also be helpful if the source code were well structured. The choice of programming language would be important since some languages scale to large

tasks better than others. Modular code would be ideal. Finally, speed can become a concern in speech recognition. Although today's computers have breathtaking performance, speech recognition can easily tax even the fastest personal computers.

As of late 2002, many ASR projects existed, but none completely satisfied the requirements listed above. For instance, the Matlab Auditory Toolbox[1] could be combined with the Matlab Neural Network Toolbox[2] to perform speech recognition, but the resulting system was dreadfully slow. The Hidden Markov Model Toolkit[3,4] (HTK) was also a commonly used speech recognition tool, but, as the name implies, it is focused exclusively on Hidden Markov Models. The HTK cannot support other approaches to speech recognition. The HTK also has a very challenging learning curve.

1.2. Objective

The objective of A Framework for Automatic Speech Recognition Research (AFARR) was to fill in the gap outlined in Section 1.1 – the lack of a well documented, easy to understand, and reasonably efficient speech recognition research tool. Such a tool would save new speech recognition researchers the trouble of either implementing a complete speech recognition system on their own (time costly), or trying to decipher one of the powerful, but poorly documented speech recognition systems available as of late 2002 (extremely difficult).

1.3. Solution Approach

Emphasis will be placed on making AFARR easy to understand, both from a programming and end user standpoint. It will also be a complete speech recognition

system, containing everything from corpus management, to acoustic front end, to acoustic models.

The C++ language will be used since it is an object oriented language. Object oriented languages allow related functions and variables to be grouped into classes, thereby greatly improving readability and ease of programming. Additionally, operator overloading will permit mathematical classes to be operated on in a more natural fashion.

Emphasis will also be placed on modularity – ensuring classes are as self-contained as possible, and as interchangeable as possible. For instance, replacing the Mel Frequency Cepstral Coefficients (MFCC) acoustic front end with an Ensemble Interval Histogram (EIH) model should not require modifications to the acoustic modeler.

Good coding style and documentation techniques will be emphasized to minimize the learning curve for future researchers attempting to extend AFARR.

CHAPTER 2

BACKGROUND

The fundamental concepts of AFARR are not new. Foundation classes are widely used in modern programming. In fact, AFARR makes heavy use of the Microsoft Foundation Classes (MFC). Additionally, the use of object oriented programming in digital signal processing (DSP) was proposed as early as 1980 by Kopec[5], and subsequently used to implement many DSP frameworks and environments[6,7,8]. In the field of speech recognition, there exist projects based upon similar concepts or with similar objectives as AFARR. Those projects are presented in the following sections.

2.1. Similar Projects

2.1.1. ISIP Foundation Classes (IFCs)

IFCs[9,10] are a set of speech foundation classes from the Institute for Signal and Information Processing (ISIP) at Mississippi State University. They are platform independent – running on a variety of platforms, including Linux, Solaris, and Windows (using Cygwin). They do not provide any sort of graphical user interface – they are solely focused on speech recognition tasks.

They were released on January 7, 2003, after the bulk of the AFARR work was accomplished. As such, it was not possible to include the IFCs in AFARR, or build upon

them. Although the general goal of AFARR and IFCs is similar, different tradeoffs and design decisions along the way have led to very different software.

For instance, IFCs are platform independent, whereas AFARR only runs under Microsoft Windows. However, AFARR provides a graphical interface, including plotting functions. IFCs do not provide any means of visualizing data. AFARR has also focused on providing functions to compare the accuracy and effects of different speech recognition techniques – corpus management, speech engine evaluation functions, etc. On the other hand, IFCs are pure speech foundation classes – they can be adapted to a wide variety of tasks, but do not perform any tasks out of the box.

Finally, AFARR has made heavier use of operator overloading, so matrix and vector operations appear natural. If a, b, and c are matrices, "a = b * c;" means multiply b and c, and store the result in a. The equivalent expression in IFCs would be "a.mult(b, c);". Although this is fine for simple expressions, it can quickly become cumbersome for complex expressions. Translating the AFARR expression, "W = W + mu * E * Transpose(X) * alpha;", to IFCs would require a temporary variable to hold X^T, then another temporary variable to hold E * X^T, and so forth.

In the future, AFARR classes may be modified to be compatible with IFCs classes, so that the best of both projects can be utilized.

2.1.2. CSLU Toolkit

The Center for Spoken Language Understanding (CSLU) Toolkit[11,12] combines speech recognition, speech synthesis, and facial animations with a graphical scripting language. Its Rapid Application Developer tool can generate programs capable of

following dialog scripts. The language itself is similar to Labview – operations are dragged and dropped onto the programming canvas.

The downside of the CSLU Toolkit is its lack of source code documentation, and the limitations of its graphical language. As of February 2003, the developer section of the CSLU Toolkit web site is empty, aside from a terse apology. The graphical language can perform some impressive feats, but it is unclear how to achieve functionality outside the scope of the language.

2.1.3. Hidden Markov Model Toolkit

The Hidden Markov Model Toolkit (HTK) is a toolkit designed to manipulate Hidden Markov Models (HMM). It is intended for speech recognition, but can also be used for other HMM tasks.

As with many powerful technical packages, HTK has a very steep learning curve. HTK is also entirely command line based – there is no graphical user interface. The source code documentation is sparse and there is no documentation describing how HTK can be extended.

2.1.4. Matlab Auditory Toolbox

The Matlab Auditory Toolbox is a collection of Matlab scripts that implement several popular auditory models. It can be combined with the Matlab Neural Network Toolbox, or some simple Hidden Markov Model scripts to perform speech recognition.

Matlab possesses the advantage of being widely used and understood. Additionally, there is a large code base of Matlab scripts available.

The primary disadvantage of Matlab is speed. Matlab matrix operation performance is excellent, but Matlab loop performance is poor. It is sometimes possible to improve performance through a process known as "vectorization", converting loops into vector operations[13]. However, the tradeoff of vectorization is readability – separate variables must be combined into a single matrix, operated on as a whole, and then split back into individual variables. The merging and splitting process is confusing, error prone, difficult to maintain, and difficult for other developers to understand.

2.1.5. Speech Recognition: Theory and C++ Implementation by Becchetti and Ricotti

Becchetti and Ricotti's book[14,15] describes Recognition Experimentation System (RES), a speech recognition engine they have written. It is a very well documented and structured, with pedantic attention to following a proper and safe programming style. Its speech recognition engine uses Hidden Markov Models and introduces a new technique of initializing HMM parameters. It is a reasonably complete system, including speech database functionality, mathematical classes, feature extraction classes, Hidden Markov Model implementation, a language model, and much more.

The downside of the RES system is its complexity. The complexity of RES made the book very difficult to follow and the book frequently read like an application programming interface (API) reference manual. By focusing on small implementation details and rationalizations, the big picture was difficult to see.

It is hoped that this thesis does not fall into that trap and reads more like a user's manual than a reference manual.

2.2. Speech Recognition Engine Basic Architecture

The speech recognition process starts with a sampled speech signal. The signal is typically sampled tens of thousands of times per second, however, the articulators that produce the speech – the glottis, tongue, lips, etc. – change position at a much slower rate. As a result, the speech samples contain a considerable amount of redundancy, and a process known as feature extraction is performed to eliminate the redundancy and reduce the information rate.

Typically, each feature is a multi-dimensional vector, and features are extracted every 10-40ms. These features are also sometimes called acoustic observations since they represent the observations the acoustic modeler actually examines.

Acoustic modelers typically take the feature vectors and compute the probability of each phoneme of the language having produced that feature vector. This is often calculated by assuming each phoneme produces a random feature with a Gaussian mixture probability density function (PDF). Once the probability of each phoneme is calculated, the most likely phonemes can be calculated through the use of the Viterbi search algorithm[16]. A language model (discussed below) can then be used to find the probability of potential word sequences, and these probabilities can be combined with the phoneme probabilities to produce the most likely word sequence.

Speech recognition systems employ lexicons – listings of all possible words and their pronunciations. Pronunciations are made up of phonemes – the basic sound units of a language. There are approximately 40 in American English. Lexicons can range in size from just 2 entries, for a yes/no speech recognition engine, to tens of thousands of words, for a dictation program.

Dictation programs, and other continuous speech recognizers often employ language models to improve accuracy. Language models calculate the likelihood of word sequences occurring. For instance, the phrase, "the train is late," is more likely than the phrase, "the train is Kate." If the engine were uncertain about the last word in the example, it could safely conclude the user said, "the train is late."

If the lexicon consists of L words, and N word long phrases are considered, then there are L^N possible N-word phrases. For large vocabulary speech recognition engines, L^N can become quite large. For a medium vocabulary size of 10,000 words, there are 10 quadrillion (10^{15}) 4-word combinations. Such systems are not feasible to train or use. As a result, N is typically limited to 3 (trigrams, 3-word sequences).

Although this is the typical recognition process, many variations exist. For instance, the multidimensional Gaussian probability density functions (PDFs) can be replaced with neural networks. The neural networks would then calculate the likelihood that each feature was produced by each state, and supply this information to the acoustic modeler. In addition, instead of calculating trigrams for individual words in the lexicon, it is possible to assign each word to a group (e.g. noun, verb, etc.) and then calculate the likelihood of a sequence of group assignments occurring.

2.3. Speech Recognition Engine Benchmarking

Speech recognition benchmarks typically measure the word error rate (WER) of the speech recognition engine. Although it is tempting to assume that the word error rate completely characterizes how "good" a speech recognition engine is, it is important to remember the WER is influenced by many factors.

- Vocabulary size – If the speech recognition engine knows the only valid words are yes and no, it will be much more accurate, than if it has to deal with a 10,000-word vocabulary.
- Training – Speaker dependent engines can achieve lower WERs than speaker independent engines since they are customized to a single user. The tradeoff is speaker dependent engines perform poorly when a different user tries to use the system without training it.
- Background Noise – Background noise can degrade accuracy considerably. Additionally, speaker dependent engines may have difficulty if they are trained in one environment and used in another, even if the testing environment is quieter than the training environment.
- Speech Spontaneity – Isolated words are easier to recognize than the same words in a sentence, due to a phenomenon known as coarticulation. Coarticulation is the influence of one word or phoneme on the following and proceeding word or phoneme. In extreme cases, the coarticulated phrase bears little semblance to the original phrase. For instance, "what did you" is sometimes pronounced closer to "what dija", thereby deleting a phoneme and transforming others.

CHAPTER 3

SYSTEM OVERVIEW

3.1. Overview of Classes/Functions

Although it is not necessary to understand the inner workings of the classes to work with AFARR, it is beneficial to understand what classes are present and their relationships. Table 3.1 and Table 3.2 list the various classes of AFARR, grouped by category. The following subsections briefly discuss each of the classes. Each of the following subsections is prefaced by a list of the corresponding class(es). Indented class names indicate inheritance (e.g. in Section 3.1.5, CSpeechRecognizerVQNN is derived from CSpeechRecognizer). Classes can be found in the file whose filename corresponds to the class name. For instance, CLexicon is defined in Lexicon.h and implemented in Lexicon.cpp.

C a t e g o r y			
Audio File (3.1.1)	**Feature Vector (3.1.2)**	**Acoustic Modeling (3.1.3)**	**Neural Network (3.1.4)**
CPCM	CFeature CFeatureSet CFeatureType	CHMM CHMMLeftRight	CPerceptron CSequenceRNN CLVQ1

Table 3.1 – Categorized Classes of AFARR (1)

C a t e g o r y		
Speech Recognition (3.1.5)	**Support**	
CSpeechRecognizer	CMatrix (3.1.6)	CSerialization (3.1.11)
CSpeechRecognizerHMM	CGaussian (3.1.7)	CArrayEx (3.1.12)
CSpeechRecognizerVQNN	CGaussianMixture (3.1.7)	CArrayExSrl (3.1.12)
CSpeechRecognizerDTW	CLexicon (3.1.8)	CHashTable (3.1.12)
	CLanguageModel (3.1.9)	CStringSrl (3.1.13)
	CSpeechDB (3.1.10)	CGlobalSettings (3.1.14)

Table 3.2 – Categorized Classes of AFARR (2)

3.1.1. PCM Class (Chapter 4)

- CPCM

The CPCM class represents a series of PCM samples (typically an audio file). An audio file class is present to handle the myriad of different audio file formats in use today. Once an audio file is loaded into the CPCM class, other classes can access the PCM data without having to worry about audio file headers, how samples are represented, or any of the other details of audio file formats.

3.1.2. Feature Vector Classes (Chapter 5)

- CFeature
- CFeatureSet
- CFeatureType

Feature vectors represent the important characteristics of a short period of speech (typically 10-40 ms).

Feature vectors are presented to the acoustic modeler instead of raw Pulse Code Modulation (PCM) samples to reduce computational load, and since many acoustic modelers simply cannot handle raw PCM samples.

Feature vectors are implemented using two classes – CFeature and CFeatureSet. CFeature represents a single feature vector. CFeatureSet represents a series of feature vectors (CFeature classes). A third class, CFeatureType, is used by CFeature and CFeatureSet to describe the parameters of the feature extraction process.

The feature classes presently only support Mel Frequency Cepstral Coefficients[15] (MFCC).

3.1.3. Acoustic Modeling Classes (Chapter 6)

- CHMM
  - CHMMLeftRight

The acoustic modeling classes calculate the probability that a series of feature vectors were produced by a certain word or phrase.

Acoustic modeling algorithms are necessary since a speaker never utters a word the same way twice. Variations in the utterances occur in the form of either temporal or spectral variations. Temporal variations occur when parts of an utterance are spoken faster or slower than normal. Spectral variations can occur due to background noise, coarticulation, prosody, the emotional state of the speaker, and various other causes.

CHMM implements the Baum Welch[17] and Viterbi[16] algorithms for Hidden Markov Models[14,16,17]. CHMMLeftRight is derived from CHMM and implements a left-right Hidden Markov Model.

3.1.4. Neural Network Classes (Chapter 7)

- CPerceptron
  - CSequenceRNN
- CLVQ1

AFARR includes three neural network classes. CLVQ1 implements Kohonen's LVQ1 algorithm[18]. CPerceptron implements single and multilayer Perceptron networks[18]. CSequenceRNN implements a recurrent neural network[18].

3.1.5. Speech Recognition Classes (Chapter 8)

- CSpeechRecognizer (abstract base class)
  - CSpeechRecognizerHMM
  - CSpeechRecognizerVQNN
  - CSpeechRecognizerDTW

Although all of the classes in AFARR are in some way involved in the speech recognition process, for the duration of this document, "speech recognition classes" will refer to classes derived from the CSpeechRecognizer class.

Speech recognition classes provide easy access to speech recognition functions. To ensure a consistent interface, all speech recognition classes are derived from CSpeechRecognizer, an abstract base class.

CSpeechRecognizerHMM performs speech recognition using Hidden Markov Models. CSpeechRecognizerVQNN performs speech recognition using Kohonen's first

version of learning vector quantization, LVQ1, and a Perceptron neural network. CSpeechRecognizerDTW performs speech recognition using dynamic time warping[19].

3.1.6. Matrix Class (Section 10.1)

- CMatrix

The CMatrix class encapsulates many of the linear algebra functions used by AFARR. CMatrix implements scalar/matrix addition, subtraction, multiplication, and division; matrix inversion; matrix determinant; matrix transpose; matrix augmentation; and much more. See Section 10.1 for a complete list of the CMatrix functionality. The files Matrix_Support.cpp and Matrix_Support.h define additional functions for manipulating CMatrix matrices, including the absolute value/sine/cosine/hyperbolic tangent/hyperbolic cosine of a CMatrix, creation of all zero/one/random matrices, and the creation of Discrete Fourier Transform (DFT) windows.

3.1.7. Gaussian Mixture Class (Section 10.2)

- CGaussian
- CGaussianMixture

The CGaussianMixture class represents a collection of multidimensional Gaussian PDFs added together. Gaussian mixtures are often used to represent the PDF of acoustic observations for a particular phoneme or sub-word unit. CGaussianMixture is implemented as a collection of CGaussian classes. Each CGaussian represents a single Gaussian PDF. Diagonal-only and full covariance matrix Gaussians are supported.

3.1.8. Lexicon Class (Section 10.3)

- CLexicon

The lexicon class, CLexicon, maps words to pronunciations, and vice versa. A lexicon class is essential for large vocabulary speech recognition, but also useful for small vocabulary speech recognition. CLexicon presently uses the 129,463-word Carnegie Mellon University (CMU) Pronouncing Dictionary[20].

3.1.9. Language Model Class (Section 10.4)

- CLanguageModel

The language model class, CLanguageModel, assists in computing the most likely word sequence. The language model class determines the likelihood of a word sequence. It accomplishes this by studying a large block of text, representative of the speech to be recognized, then computing single word (unigram), word pair (bigram), and three word combination (trigram) frequencies. The frequencies can be used to compute the probability of a word given its context. The probabilities can then be linearly combined to yield a word sequence probability. Smoothing techniques, such as optimal linear smoothing[17], can then ensure that word sequences have non-zero probability.

3.1.10. Speech Database Class (Section 10.5)

- CSpeechDB

Speech recognition research rarely involves only a handful of speech samples. Instead, a large number of speech samples are used, representing a cross section of the populace. CSpeechDB keeps track of all of the audio files in the speech corpus.

3.1.11. Serialization Class (Section 10.6)

- CSerialization

The CSerialization class provides classes with the ability to load/save themselves from/to a disk (serialization). Any class that requires serialization support should be derived from CSerializable, and implement a SerializeObject function. The SerializeObject function should load or save all of the member variables of the class to the specified file. The SerializeObject function may use CSerializable::SerializeItem to make serialization easier. The serialization process is discussed in detail in Section 10.6.

3.1.12. Array Classes (Section 10.7)

- CArrayEx
- CArrayExSrl
- CHashTable

The CArrayEx class extends the default functionality of the Microsoft Foundation Classes (MFC) CArray class to include a copy constructor, and an assignment operator. CArrayEx was necessary since CArray does not include a copy constructor, or assignment operator, yet these are important functions for any class to have. The MFC base class handles most of the array functionality.

CArrayExSrl represents an array of serializable objects. A CArrayExSrl object is also serializable. Often times classes will contain arrays of objects that will need to be serialized, and by making a serializable CArrayEx, serialization can be performed using a single SerializeItem function call (see Section 10.6 for more details).

The hash table class, CHashTable, implements a serializable hash table. Hash tables map "keys" to array locations by means of a hash function (in this case, a simple checksum modulo the table size). Hash tables are used in AFARR to quickly map dictionary words (the key) to their pronunciation.

3.1.13. String Class (Section 10.8)

- CStringSrl

The CStringSrl class extends the MFC CString class to include a printf style constructor, and AFARR style serialization.

3.1.14. Global Settings Class (Section 10.9)

- CGlobalSettings

The CGlobalSettings class provides a central repository for all application settings. Settings defined in both the initialization (INI) file, and the graphical user interface (GUI) settings window can be accessed through CGlobalSettings. Additionally, new settings for the GUI window can be defined through CGlobalSettings member functions.

3.1.15. Logging Functions (Section 10.10)

The logging functions provide a uniform way of presenting output to the user. Entries sent to the log are typically displayed both on the screen, and saved to a file with a timestamp. Comma separated value (CSV) entries, such as the accuracy of a speech recognition engine for a specific training iteration, are automatically displayed in tabular form on the screen.

3.1.16. Plotting Functions (Section 10.11)

The plotting functions provide a means of representing data in chart form. Charts can either be shown in separate windows, or embedded in the main AFARR window.

3.2. High Level Overview of the Recognition Process

The recognition process is shown in Figure 3.1. Typically, prior to speech recognition, the training and testing corpus will be loaded into CSpeechDB objects (first block of the figure), and the speech recognition engine will be initialized and trained on the training corpus. Once the engine is properly initialized and trained, the audio file containing the speech to be recognized is loaded from a file into a CPCM object (second block). The CFeatureSet class then extracts the meaningful features from the CPCM class (third block). The speech recognition class (e.g. CSpeechRecognizerHMM) then presents the features to a neural network (e.g. CPerceptron), Hidden Markov Model (CHMM), or some other acoustic modeling class, to determine the most likely state or word sequence (fourth block). The speech recognition class obscures all the messy details of how recognition was performed (fifth block), and simply returns the most likely phrase to the user (sixth block).

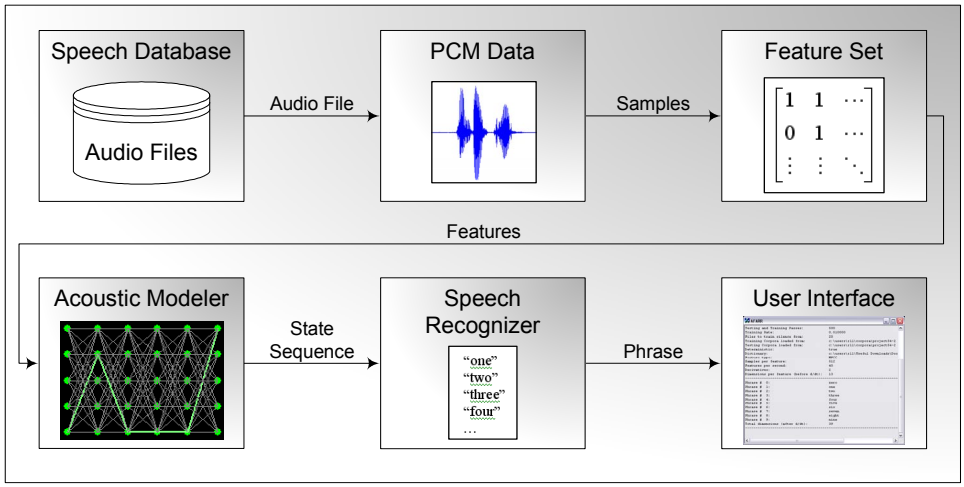


Figure 3.1 – Recognition Process

3.3. AFARR Program Flow

There are two threads in AFARR, a graphical user interface (GUI) thread, and a worker thread. The GUI thread is responsible for drawing the windows, responding to mouse clicks, and performing other user interface tasks. The GUI thread also spawns the worker thread. The worker thread is responsible for performing the speech recognition and generating results.

A diagram of the GUI is shown in Figure 3.2. There are three sections of the window – the text results, the plotting control, and the results table. The text results section displays status messages from the worker thread. The plotting control represents various types of graphical data. It shows an amplitude vs. time plot of an audio file in the figure. The results table displays results in tabular form. Presently, it represents the accuracy of the engine after each training iteration. The GUI is discussed in further detail in Section 9.2.

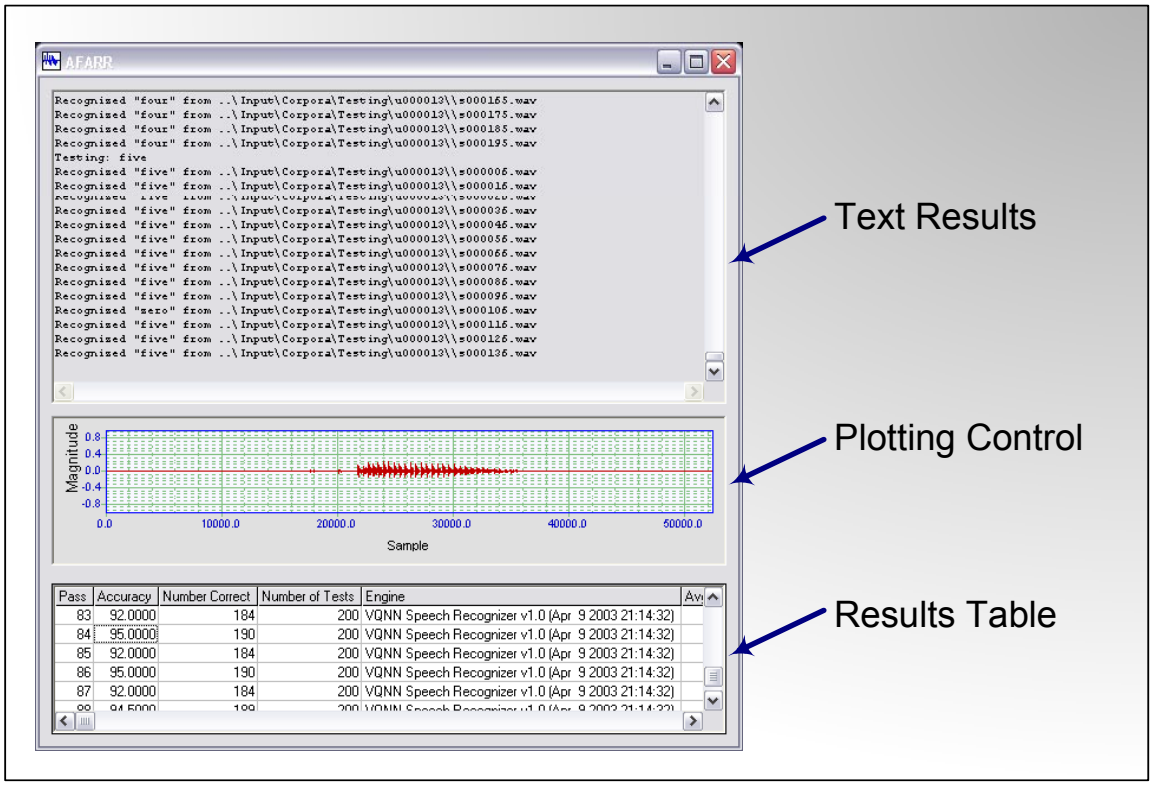


Figure 3.2 – AFARR Graphical User Interface

The GUI thread message loop (Figure 3.3) looks for three messages – the message to create the window (WM_CREATE), the timer message (WM_TIMER), and the message to close the window (IDCANCEL). Upon receipt of the WM_CREATE message, the GUI thread displays the configuration dialog, waits for the user to finish configuring settings, then creates the worker thread. Upon receipt of a WM_TIMER message with timer identifier 1000, the GUI thread processes any commands from the worker thread. If an IDCANCEL message is received, the worker thread is terminated, and AFARR exits.

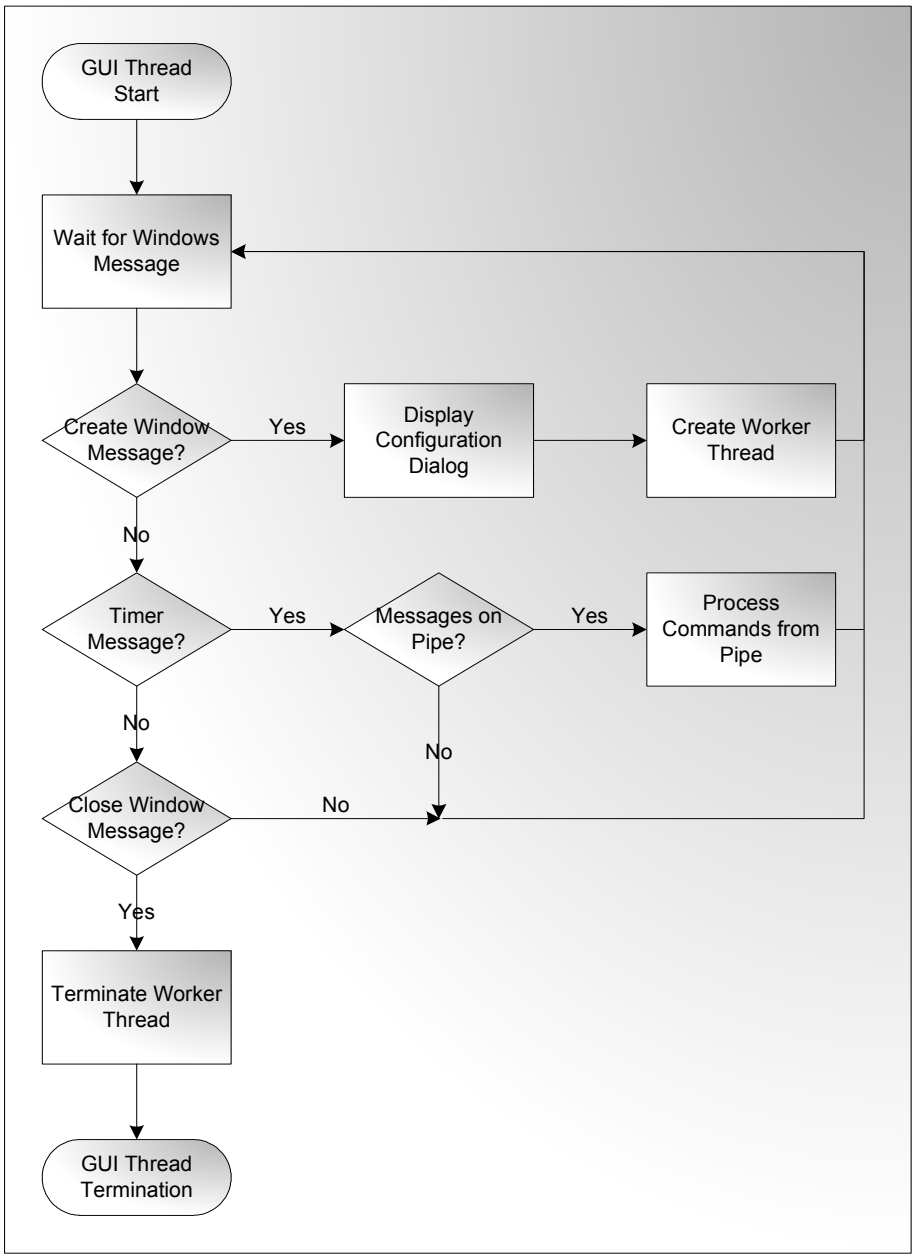


Figure 3.3 – AFARR GUI Thread Flowchart

The worker thread main function, WorkerThread(), is defined in main.cpp. The worker thread may communicate with the GUI thread by means of two anonymous pipes (a section of shared memory used for inter-thread or inter-process communication), one for communication in each direction.

Presently, two types of commands may be sent along the pipes. The first transmits log entries, and the second creates a plot. Log entries are appended to the text results control or the results table, shown in Figure 3.2. Plot messages plot data. Data may be plotted in the plotting control shown in the middle of Figure 3.2, or in separate plot windows. The destination is determined by the plotting options. Plots cannot be directly created by the worker thread since the worker thread does not have a Windows message handler.

To eliminate multithreading challenges, such as race conditions, the worker thread automatically waits for the GUI thread to completely process its message before continuing execution.

Most developers will not need to worry about the pipes since the logging and plotting functions automatically send the appropriate messages along the pipes, and the GUI automatically processes those messages.

3.4. Worker Thread

The worker thread can be thought of as the main() function of a C program. Most of the AFARR code provides support for the activities performed in the worker thread. Studying the worker thread source code is a good way to learn how AFARR works from a high level. It is also a good starting point for developers hoping to extend AFARR. The worker thread itself is defined in the main.cpp file by the WorkerThread() function.

The worker thread can be configured to perform many tasks, but presently, it follows the flowchart shown in Figure 3.4. The worker thread starts out by creating a CGlobalSettings object (see Section 10.9), and having it read in configuration settings

from the INI file specified in the configuration dialog (typically AFARR.ini, discussed in Section 9.1.2). Next, it may optionally (dependent upon configuration settings) test all of the classes. This testing is discussed in detail in Section 11.2. If any tests fail, the worker thread terminates. If all of the tests pass, all of the configuration settings are logged to the log file (see Section 10.10). The training and testing speech corpus are then loaded into CSpeechDB objects (see Section 10.5). A dictionary file (CLexicon, see Section 10.3) and/or language model (CLanguageModel, see Section 10.4) may optionally be loaded.

Once initialization is complete, one or more speech recognition engines (classes derived from CSpeechRecognizer, see Chapter 8) are evaluated. Each engine is initialized, tested on the testing corpus, and trained on the training corpus. The testing and training process is repeated as many times as specified by the initialization (INI) file. Testing is performed prior to training since the initialization routine of some speech recognition classes (e.g. CSpeechRecognizerHMM) provides some initial training. Results are plotted, logged to the log files, and displayed in the results table of the main AFARR window (see Figure 3.2).

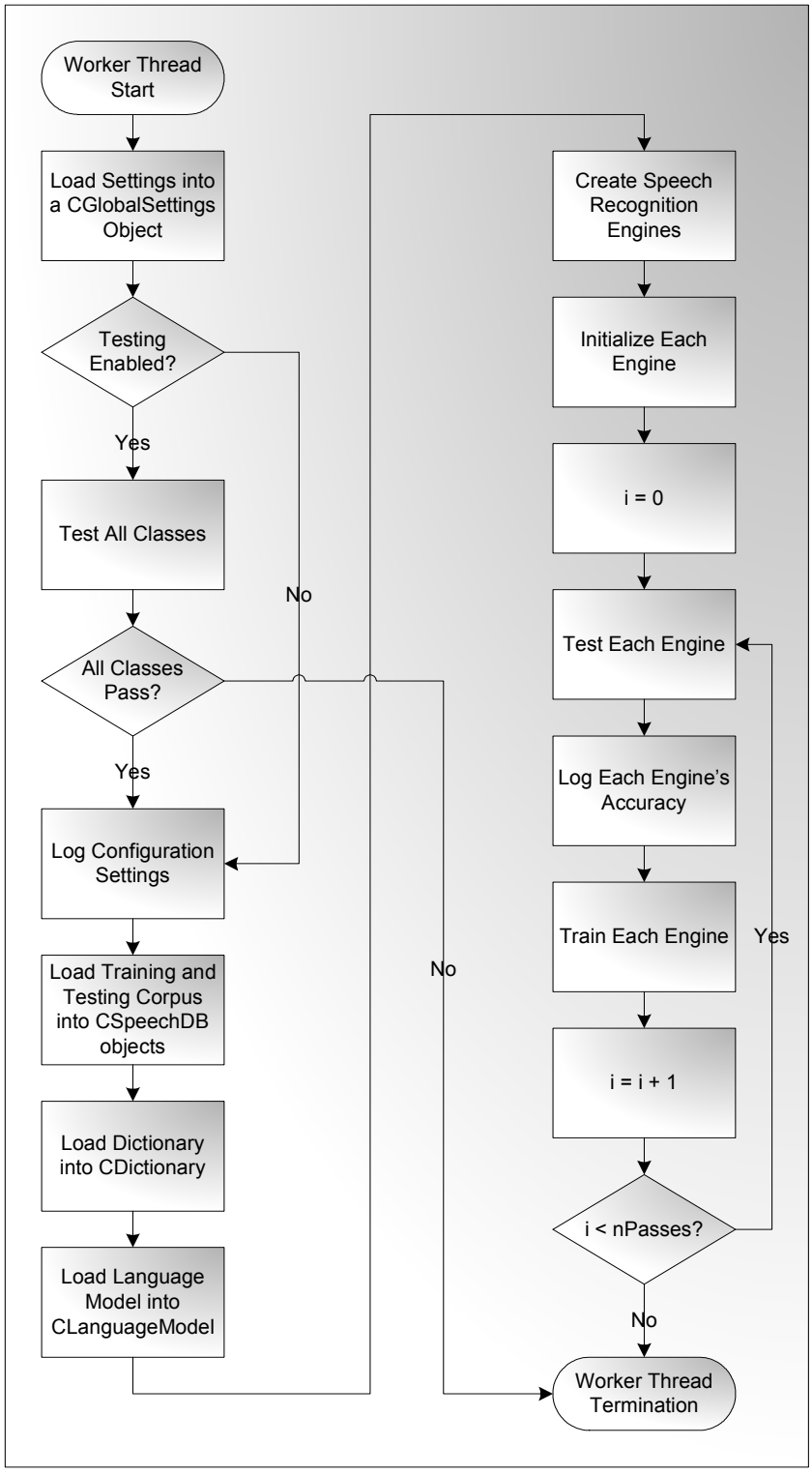


Figure 3.4 – AFARR Worker Thread Flowchart

3.5. CATlab Corpus

CATlab is a collaborative research and development effort between the University of New Hampshire and the New Hampshire Department of Safety. It is supported by the U.S. Department of Justice. It is focused on introducing and integrating advanced technologies into the operations of the New Hampshire State Police and other law enforcement agencies. All of the advanced technologies in police cruisers are controlled by an embedded PC with a graphical and speech interface, known as Project54. It is better for officers to control the Project54 system using the speech interface rather than taking their eyes off the road to use the GUI. However, police cruisers are hostile environments for ASR systems, and many commercial speech recognition engines perform poorly when the police sirens are blaring or the officer is blasting his favorite tunes. The CATlab corpus was gathered to aid in the evaluation of commercial engines, as well as provide training and testing data for AFARR.

The Microsoft Audio Collection tool was used to collect the corpus. A four element, noise canceling, Andrea DA-310 microphone was used for all of the measurements. The analog signal from the microphone was digitized with 16-bit resolution at 44.1kHz by a Sound Blaster Live 1024 sound card. The corpus was gathered in Kingsbury 252, a quiet office.

Six male New England users contributed 3,550 phrases to the corpus. Most of the speakers were aged 21 to 27. Only one speaker had a noticeable accent (a Massachusetts accent). A listing of the phrases in the corpus is included in Appendix B. The corpus is included on the thesis CD-ROM.

3.6. Upcoming Chapters

Chapter 4 will discuss the PCM audio file class (CPCM) and the libsndfile audio file library. Chapter 5 will discuss the feature representation classes and feature extraction in general. Features are extracted from CPCM objects. Chapter 6 will talk about the two Hidden Markov Model classes used by the HMM speech recognition class to model the dynamic nature of speech features. Chapter 7 will discuss the neural network classes used by the neural network speech recognition class. Chapter 8 will discuss the speech recognition (CSpeechRecognizer derived) classes in detail. Chapter 9 will go into detail about the user interface. Chapter 10 will discuss support classes, such as the matrix class and the language model class. Support classes, and especially the CMatrix class, are used throughout AFARR. Section 11.2 will discuss how AFARR classes were tested. Results will be presented and discussed in the remaining sections of Chapter 11. Finally, the conclusion is in Chapter 12.

CHAPTER 4

PCM CLASS

A CPCM object represents an audio file. A wide variety of file formats are supported through Erik de Castro Lopo's open source libsndfile library[21]. The libsndfile library is capable of reading and writing 14 standard audio file formats. It is called by the CPCM reading and writing functions. In addition to reading and writing audio files, the CPCM class can perform digital filtering (FIR/IIR filtering), RASTA[22] processing, spectral subtraction, silence removal/extraction, and resampling.

4.1. Usage

The CPCM class should be initialized by loading a file into it. The Load function will accept a filename, automatically determine its type, and load the audio file into memory. Later, CPCM objects can be saved using the Save function, and specifying an audio format.

Once loaded, individual samples can be accessed using the parentheses operator. The only argument should be the sample number. Alternately, GetPCMSamples will return a pointer to a double precision floating point array (double *). Samples range in value from -1 to +1. The length of the audio (specified in samples) can be obtained from the GetNumberOfSamples function. The sampling rate can be obtained using GetSamplingRate. If desired, the file can be resampled using the Resample function.

Spectral subtraction and relative spectral (RASTA) processing can be performed using the SpectralSubtraction and Rastafy functions, respectively.

Finite impulse response (FIR)/infinite impulse response (IIR) filtering can be performed using the Filter function, and supplying the FIR or IIR filter coefficients.

ExtractSilence and RemoveSilence create a new CPCM object, containing only the silence and only the speech, respectively. A simple level detector is implemented in SpeechStart and SpeechEnd to aid the ExtractSilence and RemoveSilence functions.

4.2. libsndfile

The libsndfile library is an open source C library that provides access to a wide variety of audio file formats through a standard library interface. It was released under the GNU Lesser General Public License (LGPL)[23]. It was originally developed for Linux, but has been ported to Win32 systems. It was designed to support both little endian (least significant byte first) and big endian (most significant byte first) data, and can operate correctly on little endian and big endian processors. Presently, libsndfile (and therefore CPCM) support the 14 audio file formats listed in Table 4.1 and Table 4.2. An "R" indicates the corresponding format can be read, and a "W" indicates the corresponding format can be written.

Owner	Microsoft	SGI / Apple	Sun / DEC / NeXT	Header less	Paris Audio File	Commodore Amiga	Sphere Nist
Extension	WAV	AIFF / AIFC	AU / SND	RAW	PAF	IFF / SVX	WAV
Unsigned 8 bit PCM	R/W	R/W		R/W			
Signed 8 bit PCM		R/W	R/W	R/W	R/W	R/W	R/W
Signed 16 bit PCM	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Signed 24 bit PCM	R/W	R/W	R/W	R/W	R/W		R/W
Signed 32 bit PCM	R/W	R/W	R/W	R/W			R/W
32 bit float	R/W	R/W	R/W	R/W			
64 bit double	R/W	R/W	R/W	R/W			
u-law encoding	R/W	R/W	R/W	R/W			R/W
A-law encoding	R/W	R/W	R/W	R/W			R/W
IMA ADPCM	R/W						
MS ADPCM	R/W						
GSM 6.10	R/W	R/W		R/W			
G721 ADPCM 32kbps			R/W				
G723 ADPCM 24kbps			R/W				
G723 ADPCM 40kbps			R/W				
12 bit DWVW		R/W		R/W			
16 bit DWVW		R/W		R/W			
24 bit DWVW		R/W		R/W			
Ok Dialogic ADPCM				R/W			
8 bit DPCM							
16 bit DPCM							

Table 4.1 – Sound File Compatibility (1/2)

Owner	IRCAM	Creative	Sound Forge	GNU Octave 2.0	GNU Octave 2.1	Portable Voice Format	Fasttracker 2
Extension	SF	VOC	W64	MAT4	MAT5	PVF	XI
Unsigned 8 bit PCM		R/W	R/W		R/W		
Signed 8 bit PCM						R/W	
Signed 16 bit PCM	R/W	R/W	R/W	R/W	R/W	R/W	
Signed 24 bit PCM	R/W		R/W				
Signed 32 bit PCM	R/W		R/W	R/W	R/W	R/W	
32 bit float	R/W		R/W	R/W	R/W		
64 bit double			R/W	R/W	R/W		
u-law encoding	R/W	R/W	R/W				
A-law encoding	R/W	R/W	R/W				
IMA ADPCM			R/W				
MS ADPCM			R/W				
GSM 6.10			R/W				
G721 ADPCM 32kbps							
G723 ADPCM 24kbps							
G723 ADPCM 40kbps							
12 bit DWVW							
16 bit DWVW							
24 bit DWVW							
Ok Dialogic ADPCM							
8 bit DPCM							R/W
16 bit DPCM							R/W

Table 4.2 – Sound File Compatibility (2/2)

The libsndfile library was added to AFARR to improve compatibility and reduce development time. Although it would have been possible to develop a custom audio file library, such an endeavor would have been redundant and it would have cost a considerable amount of time.

CHAPTER 5

FEATURE CLASSES

5.1. Usage

Acoustic observations are represented using two classes – CFeature, and CFeatureSet. CFeature represents a single feature, or acoustic observation. The CFeature class also performs the actual feature extraction. CFeatureSet, on the other hand, manages a collection of CFeature classes, directs the extraction of features from a CPCM object, and can perform other operations on the entire set of features, such as mean subtraction.

Internally, CFeatures represent features as a CMatrix object, so regular CMatrix operations, such as addition, subtraction, division, and distance metrics can easily be performed on features.

5.2. Feature Extraction

Feature extraction is performed by calling the Extract function of CFeatureSet and supplying it with a CPCM object and a CFeatureType object. The CFeatureType object (Figure 5.1) is typically filled by CGlobalSettings, and contains a description of what sort of features are desired.

```
class CFeatureType
{
      public:
            // ...
            CStringSrl m_strName;
            int m_nSamplesPerFeature;
            int m_iFeaturesPerSecond;
            int m_nDerivatives;
            int m_nDimPreDDT;
            int m_nDimTotal;
            int m_nFFTSize;
};
```

Figure 5.1 – CFeatureType Declaration

CFeatureType presently contains seven elements. The m_strName element contains the name of the type of feature desired (e.g. "MFCC"). The m_nSamplesPerFeature element indicates the number of samples used to calculate each feature, typically the window size. The m_nFFTSize element indicates the number of samples to be supplied to the FFT algorithm. Typically, it is larger than m_nSamplesPerFeature, and results in the samples being zero padded. The m_iFeaturesPerSecond element defines how frequently features should be calculated. The m_nDerivatives variable specifies how many time derivatives of the features should be taken (typically 2). The m_nDimPreDDT variable specifies the number of dimensions the features should have, prior to differentiation. The m_nDimTotal variable specifies the number of dimensions the features should have after differentiation.

CFeatureSet::Extract takes a CPCM object containing the speech samples and a CFeatureType object, and creates the appropriate number of CFeature objects. The CFeature objects are then initialized, and have their Extract function called. CFeature::Extract takes three arguments, the CPCM object, the CFeatureType object, and the first sample to be used in the feature. The CFeature object can determine the last sample to be used in the feature from the CFeatureType object. CFeature::Extract is

responsible for the actual feature extraction, typically by calling a support function (e.g. the mfcc function, which implements Section 5.4).

5.3. Access to Features

Individual features in a CFeatureSet object can be accessed by using the GetFeature member function and supplying it with the zero based index of the feature. A NULL pointer will be returned if an invalid feature index is specified. The number of features in a CFeatureSet can be obtained using the Count member function.

Access to the actual feature data in a CFeature is accomplished using the CFeature::GetFeaturePointer function. GetFeaturePointer returns a pointer to an m_nDimTotal x 1 sized CMatrix object (a column vector).

5.4. Mel Frequency Cepstral Coefficients

Mel Frequency Cepstral Coefficients (MFCC) are widely used in acoustic front-ends. Many alternate feature types are available, but MFCC was selected for its good accuracy and low computational complexity. MFCC extraction is a multistage process, consisting of preprocessing, windowing, Fourier transformation, frequency domain filtering, log energy computation, and cepstrum computation (Inverse Discrete Fourier Transform, IDFT). A diagram of the process is included in Figure 5.2. Each block of the figure is discussed in the following five sections.

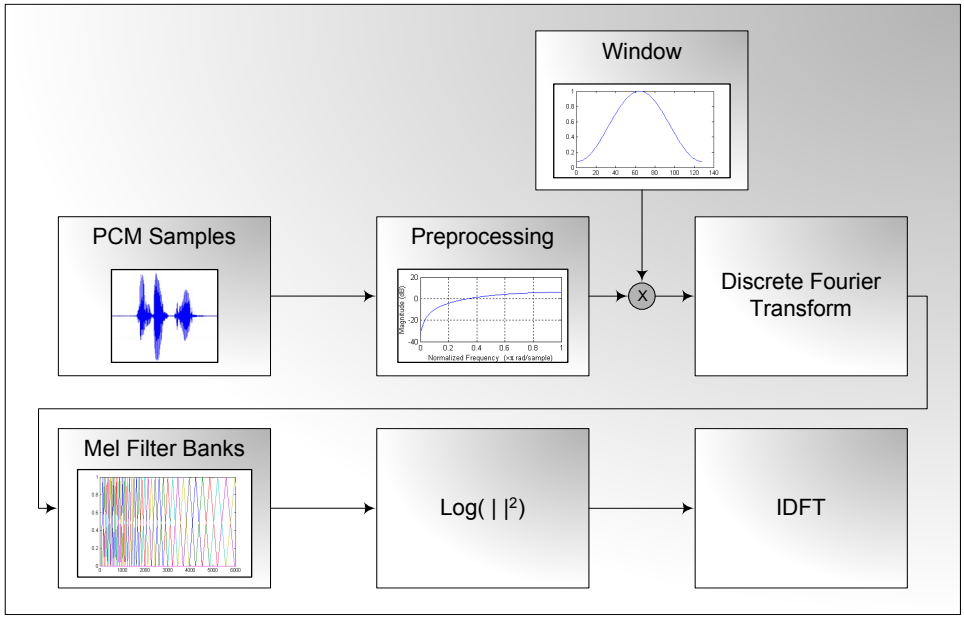


Figure 5.2 – MFCC Process

5.4.1. Preprocessing

Current speech recognition systems rely heavily on vowels for speech recognition. Vowels are characterized by well defined spectral features, with peaks in the spectrum known as formants. Higher frequency formants have smaller amplitudes than the lower frequency formants, however, they are just as important as the lower frequency formants. To compensate for this difference in amplitudes, a simple preemphasis filter is used. Typically, preemphasis filters have the transfer function shown in Equation 5.1[15].

$$H(z) = 1 - a \cdot z^{-1}$$

$$where:$$
$$0 \le a < 1$$

Equation 5.1 – Preemphasis Transfer Function

A typical value of a is 0.97, resulting in 25dB of gain in higher frequencies relative to lower frequencies (see Figure 5.3).

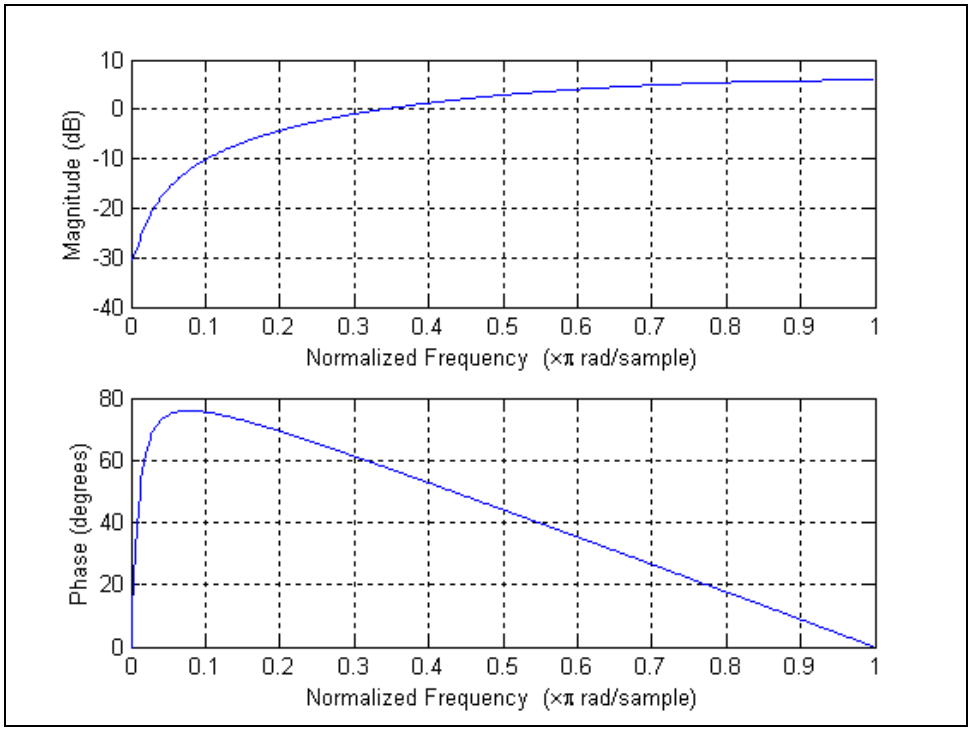


Figure 5.3 – Preemphasis Filter Characteristics

In addition to preemphasis, silence removal is sometimes performed prior to feature extraction, since certain automatic speech recognition (ASR) techniques (e.g. HMM) may experience significant accuracy and performance reduction if long runs of silence are presented to the acoustic modeler[15].

5.4.2. Windowing

The basic discrete Fourier transform[24] (DFT) algorithm exhibits a phenomenon known as leakage when the input signal is either not periodic, or includes a non-integer number of periods. To combat DFT leakage, the input signal is windowed. Windowing

consists of simply multiplying the input signal by a window function in the time domain. Common windowing functions are listed in Table 5.1.

Window	Equation
Rectangular	$w(n) = 1$
Triangle	$w(n) = 1 - \left\| \frac{n - \frac{N}{2}}{\frac{N}{2}} \right\|$
Hanning	$w(n) = 0.5 - 0.5 \cdot \cos(2\pi \frac{n}{N-1})$
Hamming	$w(n) = 0.54 - 0.46 \cdot \cos(2\pi \frac{n}{N-1})$

Table 5.1 – Common Window Functions

5.4.3. Discrete Fourier Transform

The DFT transforms the windowed input signal from the time domain into the frequency domain. Zero padding is sometimes used to improve the frequency resolution of the DFT.

The FFTW library[25,26] provides the actual Fast Fourier Transform (FFT) routines. FFT is a more efficient version of the DFT algorithm, for window sizes that are powers of 2.

5.4.4. Filtering

The center frequencies of the Mel filter banks are spaced linearly below 1kHz, and logarithmically above 1kHz[15]. A set of 40 mel banks were used in AFARR – 13 linear

filters and 27 logarithmic filters. Rather than calculating IIR filter coefficients for all 40 filters and actually performing IIR filtering, the FFT algorithm is used, and the FFT bins are added together using a triangular weighting function. Equation 5.2 lists the relevant equations.

$$y(c) = \sum_{i=0}^{N_{FFT}-1} w_c(i) \cdot |x(i)|$$

where

$x(i)$ is the output of the ith bin of the FFT algorithm and the input to this stage of processing

$y(c)$ is the output of the cth filter and the output of this stage

$$w_c(i) = \begin{cases} h(c) \cdot \dfrac{FFTf_c(i) - f_l(c)}{f_c(c) - f_l(c)} & \text{if } FFTf_c(i) > f_l(c) \text{ and } FFTf_c(i) < f_c(c) \\ h(c) \cdot \dfrac{f_u(c) - FFTf_c(i)}{f_u(c) - f_c(c)} & \text{if } FFTf_c(i) \geq f_c(c) \text{ and } FFTf_c(i) < f_u(c) \\ 0 & \text{elsewhere} \end{cases}$$

$$h(c) = \frac{2}{f_u(c) - f_l(c)}$$

$$FFTf_c(i) = f_s \cdot \frac{i}{N_{FFT}}$$

$$f(c) = \begin{cases} f_0 + c \cdot S_{Linear} & \text{for } 0 \leq c \leq N_{Linear} \\ f(c-1) \cdot S_{Log} & \text{for } N_{Linear} < c \leq N_{Linear} + N_{Log} + 1 \end{cases}$$

$f_u(c) = f(c+2)$

$f_c(c) = f(c+1)$

$f_l(c) = f(c)$

N_{FFT} is equal to the number of FFT points (2048)

N_{Linear} is equal to the number of linear filters (13)

N_{Log} is equal to the number of logarithmic filters (27)

S_{Linear} is equal to the spacing of the linear filters (66.667 Hz)

S_{Log} is equal to the spacing of the logarithmic filters (1.0712)

f_0 is the lower bound of the first filter (133.33 Hz)

c is the filter number ($0 \leq i < N_{Linear} + N_{Log}$)

i refers to the FFT bin ($0 \leq i < N_{FFT}$)

Equation 5.2 – MFCC Filter Equations

5.4.5. Log Energy and IDFT Computation

The log energy of the outputs of the filter banks are taken, and fed into the IDFT algorithm to produce MFCC coefficients (Equation 5.3).

$$mfcc_k = \sqrt{\frac{2}{N_{Linear} + N_{Log}}} \cdot \left| \sum_{c=0}^{N_{Linear}+N_{Log}-1} e(c) \cdot e^{\frac{j \cdot \pi \cdot (c-0.5) \cdot k}{N_{Linear}+N_{Log}}} \right|$$

which reduces to the DCT since e(c) is symmetric:

$$mfcc_k = \sqrt{\frac{2}{N_{Linear} + N_{Log}}} \cdot \left(\sum_{c=0}^{N_{Linear}+N_{Log}-1} e(c) \cdot \cos\left(\frac{\pi \cdot (c-0.5) \cdot k}{N_{Linear} + N_{Log}} \right) \right)$$

where:

$e(c) = \log_{10} |y(c)|$

N_{Linear} is equal to the number of linear filters (13)

N_{Log} is equal to the number of logarithmic filters (27)

c is the filter number ($0 \le i < N_{Linear} + N_{Log}$)

k refers to the MFCC coefficient index ($0 \le k < 13$)

Equation 5.3 – Log Energy and IDFT Calculations

CHAPTER 6

HIDDEN MARKOV MODEL CLASSES

6.1. CHMM

CHMM implements a generic continuous multidimensional input Hidden Markov Model class. CHMM supports Baum-Welch training and a full Viterbi search. It does not contain any functions specific to modeling speech, and cannot initialize the transition probabilities and Gaussian parameters by itself.

Notable variables are listed in Table 6.1.

Variable Name	Description
m_cP	n x m Matrix. Transition probabilities. Row indicates next state. Column indicates present state.
m_cPI	n x 1 Matrix (column vector). Initial state probabilities.
m_cFinalProb	n x 1 Matrix (column vector). Final state probabilities.
m_ppcTransitions	Array of CHMMTransition pointers. Each CHMMTransition encapsulates a CGaussianMixture. The Gaussian mixture represents the observation PDF for that

	transition. The Gaussian mixtures can be configured using the SetupTransition function.
m_cAlpha	n x m Matrix. Forward probabilities matrix, used in Baum-Welch training. Row indicates time step. Column indicates the state number.
m_cBeta	n x m Matrix. Backward probabilities matrix, used in Baum-Welch training. Row indicates time step. Column indicates the state number.
m_cKstar	n x m Matrix. Normalizing matrix, used in Baum-Welch training. Row indicates next state. Column indicates present state.
m_cPt	n x m x j Matrix. Transition probability matrix, used in Baum-Welch training. First index represents time step. Second index represents next state. Third index represents present state.
m_fUpdateThreshold	Represents the minimum number of times a transition must occur for Baum-Welch to update the Gaussian PDF parameters.
m_fMinTransitionProb	Represents the minimum value an entry in m_cP may take on.

	Invalid transitions (whose transition probability is zero) are not affected by m_fMinTransitionProb.
m_fMinVariance	The minimum value a Gaussian variance parameter may take on.

Table 6.1 – Notable CHMM Member Variables

6.2. CHMMLeftRight

CHMMLeftRight is derived from CHMM and implements a left-right Hidden Markov Model. Each phrase or word has a fixed number of HMM states assigned to it. The HMM transition probabilities are initially set up in a simple left-right fashion – each state may either transition back to itself, or to the next node in the left-right sequence. HMM parameters are initialized by uniformly dividing up the training data and calculating the mean and variance of the features of the resulting blocks of samples. For instance, in Figure 6.1, each word is divided into four states. The first state represents the first one fourth of the word, the second state represents the second one fourth of the word, and so forth. Additionally, a special silence state exists, which models the absence of speech.

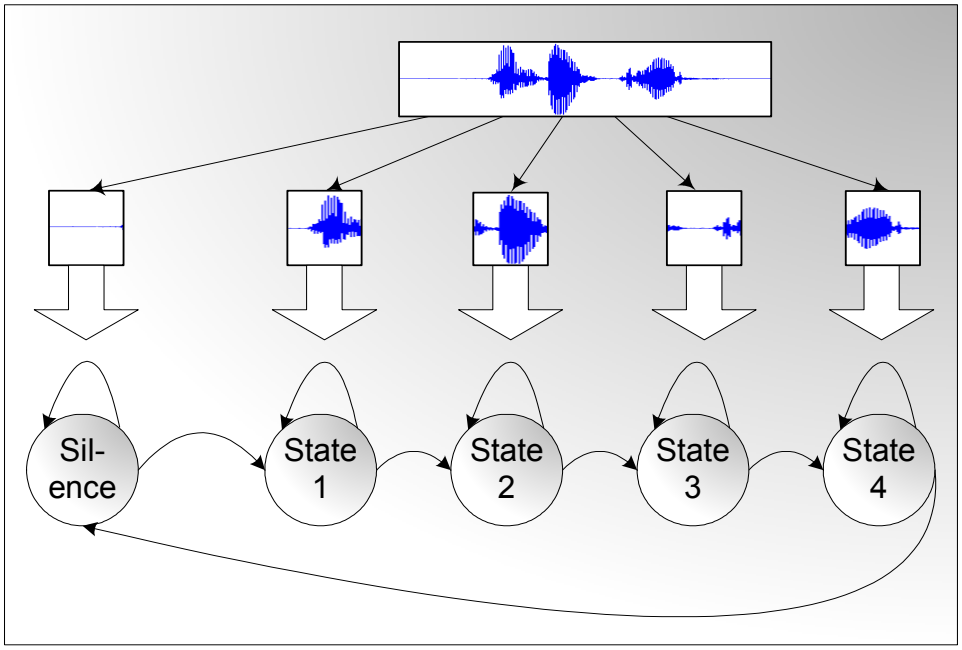


Figure 6.1 – Left-Right Hidden Markov Model Arrangement

CHMMLeftRight is primarily intended for modeling speech. As such, it imposes some restrictions on the HMM. First of all, the diagonal values of the covariance matrix (variances) are not allowed to fall below an adjustable threshold. Second, the mean and covariance of a transition are not updated during Baum-Welch training unless that transition has occurred an adjustable number of times. Forward and backward probabilities are normalized at each time step to avoid underflows. The Baum-Welch algorithm was adjusted to accept normalized probabilities. Finally, certain transition probabilities are fixed. For instance, the HMM must always begin and end in the silence state. The probability of the first state of each word is fixed at $^{1}/_{N}$, where N is number of words in the vocabulary.

CHAPTER 7

NEURAL NETWORK CLASSES

7.1. CPerceptron

The CPerceptron class implements a simple Perceptron, if a single layer is specified, or a feedforward multilayer Perceptron with the backpropagation learning algorithm, if multiple layers are specified. Sigmoid and linear are presently the only activation functions defined, but additional activation functions can easily be defined by modifying the ActivationFunction and ActivationFunctionDerviative functions.

Figure 7.1 shows a diagram of a Perceptron neuron. The output of an individual neuron is equal to the dot product of the inputs and the weights applied to the non-linear function. If there is only a single layer of neurons, the output of the individual neurons becomes the output of the entire network.

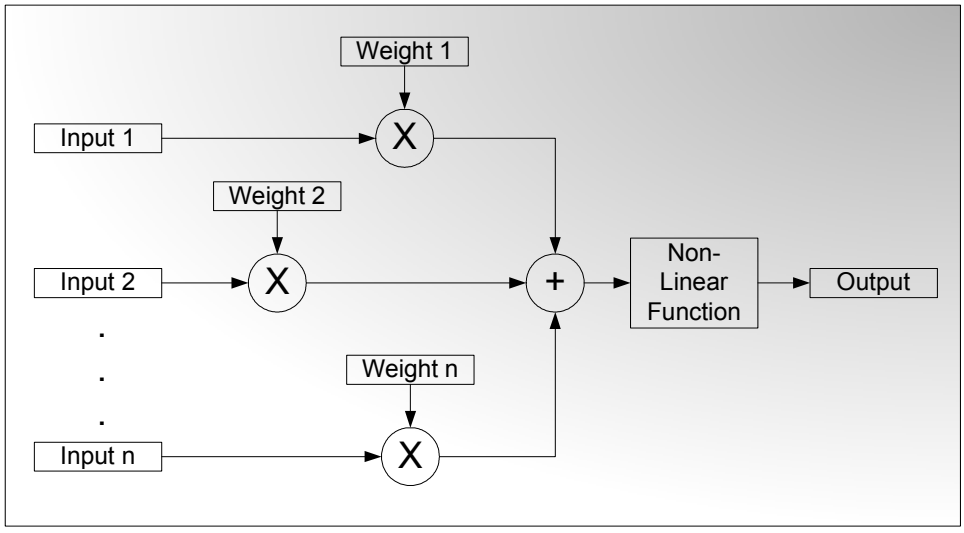


Figure 7.1 – Perceptron Neuron

In the case of a multilayer network, Figure 7.2 is used. In this case, the outputs of each layer of the neural network feeds into the inputs of the following layer. The outputs of the final layer become the outputs of the network.

The number of layers is adjustable. Three layers are shown in Figure 7.2. The number of neurons in each layer is also adjustable. Figure 7.2 shows four neurons in the input layer, three in the hidden layer (second layer), and three in the output (third) layer.

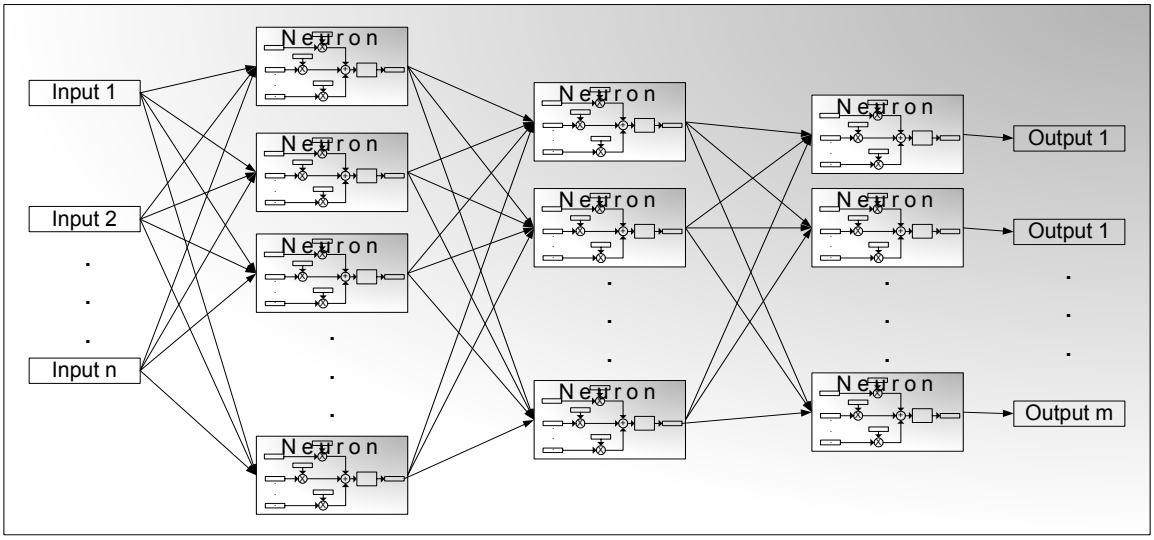


Figure 7.2 - Multilayer Perceptron

7.2. CLVQ1

The CLVQ1 class implements Kohonen's first version of learning vector quantization, LVQ1. Vector quantization is a method of compressing input vectors, which may encompass a large space, into a smaller set of prototypes that provide a good approximation to the input space. Input vectors are assigned to a prototype based on some distortion measure. Often times, the distortion measure is based on the L_2 norm of the vector (Euclidean distance). Input vectors are assigned to the prototype whose input vector distortion measure is the smallest.

In the LVQ1 algorithm, the prototypes are updated according to the following rule. Each input vector in the training set is presented to the LVQ1 algorithm and classified. Prototypes sharing the same classification as the input vector are moved closer to the input vector. Prototypes with a different classification than the input vector are moved farther away from the input vector. The process repeats until all of the input vectors are presented to the LVQ1 neural network, and then the set of input vectors is presented and used to update again and again until the prototypes reach an equilibrium point.

By applying LVQ1 to the input data, prior to presenting it to the acoustic modeling layer, the computation load on the acoustic modeling layer is significantly reduced. Instead of processing a 39 dimensional acoustic observation, now only one dimension must be processed. Additionally, due to vector quantization, the acoustic observations can only take on a finite set of values. As such, the discrete input Hidden Markov Model algorithm can be used instead of the continuous input HMM algorithm.

CHAPTER 8

RECOGNITION CLASSES

8.1. Overview

Speech recognition classes are responsible for hiding all of the details of the speech recognition process from the main function, and providing a simple and uniform interface to the worker thread.

The uniform interface allows the worker thread to initialize, train, and evaluate each engine, without concern for which speech recognition class is being used, or how the speech recognition class works.

The uniform interface is ensured through the use of an abstract base class. An abstract base class is a base class with virtual function declarations, but no function bodies. As such, abstract base classes cannot be instantiated. Classes must be derived from the abstract base class, and those derived classes must implement the virtual functions of the abstract base class. The derived classes can then be instantiated.

Three such classes have been derived from CSpeechRecognizer – CSpeechRecognizerHMM, CSpeechRecognizerDTW, and CSpeechRecognizerVQNN. CSpeechRecognizerHMM uses the traditional Hidden Markov Model acoustic modeler. CSpeechRecognizerDTW performs dynamic time warping (a means of temporally aligning the testing and reference speech samples) to perform speech recognition. CSpeechRecognizerVQNN uses the LVQ1 algorithm to partition the feature space, and a

Perceptron network to recognize words. The engines will be discussed in detail in Section 8.3.

8.2. Usage

A minimal speech recognition engine must implement five functions – initialization, training, recognition, deinitialization, and serialization. The following sections discuss each function. The corresponding CSpeechRecognizer member function name is listed in parentheses in the section names.

8.2.1. Initialization (Init)

The initialization function receives a copy of the global settings and the training corpus as arguments. The initialization routine will typically randomly initialize neural networks, or choose a reasonable starting point for algorithms like Baum-Welch.

8.2.2. Training (Train)

The training function performs a single pass of training on a set of training audio files and corresponding phrases. The speech recognition class is responsible for translating the phrases from words to phonemes, state names, or whatever representation the acoustic modeler uses.

8.2.3. Recognition (Recognize)

The recognition function should attempt to recognize the phrase in the single supplied audio file. The recognition function should return the most likely word sequence present

in the audio file, and discard all other hypotheses. In the future, it may be modified to return a list of the most likely phrases and the confidence levels. The language model could then assist in the selection of the most likely phrase.

8.2.4. Deinitialization (Clear)

Deinitialization frees all resources allocated by the speech recognition engine, and resets the recognition engine to its default state. Deinitializing the speech recognition engine should undo all training.

8.2.5. Serialization (SerializeObject)

It is often desirable to save the state of the speech recognition engine, so it does not have to be trained every time it is loaded. The serialization function (inherited from CSerializable) loads and saves the state of the speech recognition engine object to a file. It is not necessary to initialize the class prior to loading it with SerializeObject.

8.3. Implementations

8.3.1. Hidden Markov Model Speech Recognition Engine

The Hidden Markov Model Speech Recognition class, CSpeechRecognizerHMM, implements a classical HMM speech recognition engine. Features are extracted from the training/testing speech at a uniform rate, then fed into the acoustic modeling class, CHMMLeftRight. CHMMLeftRight implements a Hidden Markov Model with multidimensional, continuous inputs. A basic left-right model, with an adjustable number of nodes per phrase is used. Transitions observation probabilities are represented by a

multidimensional Gaussian PDF. The Hidden Markov Model class assumes transitions produce the observations rather than the states producing the observations.

Features fed into the CHMMLeftRight class can be used to either initialize, train, or find the best path through the HMM.

Initialization is performed by uniformly splitting each phrase into a fixed number of states. The mean and covariance matrix for each state and transitions between states are then estimated from the training data. No attempt is made to time-align the phrases during initialization – that is left to the training stage.

Training is performed using the regular Baum-Welch algorithm with a few variations, as discussed in Chapter 6. The HMM recognition process is shown in Figure 8.1. The process follows the general approach discussed in Section 3.2. First, features are extracted from the CPCM object into CFeatureSet (first and second block in the figure). Next, the Viterbi algorithm is used to find the most likely state sequence that could have produced the observed features (third block). Finally, the state sequence from the Viterbi algorithm is translated into phrases (fourth block) and returned from CSpeechRecognizerHMM::Recognize (fifth block). Presently, the Viterbi implementation considers paths from every state to every other state at every time step.

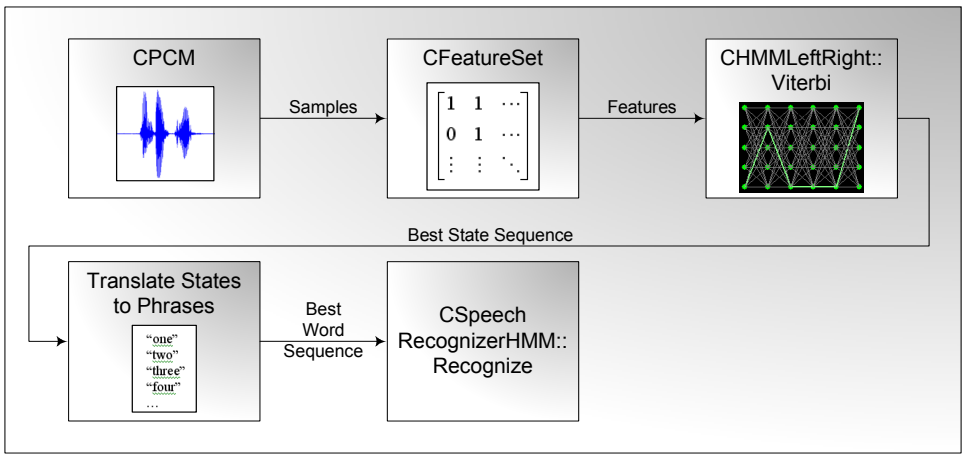


Figure 8.1 – HMM SR Recognition Process

8.3.2. Neural Network Speech Recognition Engine

In the neural network speech recognition engine, CSpeechRecognizerVQNN (Figure 8.2), features are initially extracted from a CPCM object into a CFeatureSet (second block of the figure). Features then undergo vector quantization (third block, LVQ1). At this point, features are one dimensional, and may only take on a finite set of values (one of the VQ points). These features are used to build a pattern for the Perceptron network (fourth block). For each VQ point, there are ten Perceptron inputs. Each of the ten inputs represents a different level of activity for that VQ point. Only one of the ten will be set to +1. The rest will be set to -1.

For example, suppose inputs 10-19 are associated with VQ point #1. If VQ point #1 were not very active during the observed utterance, then input 10 would be set to +1, and inputs 11-19 would be set to -1. On the other hand, if VQ point #1 were the most active VQ point, then input 19 would be set to +1, and the rest would be set to -1.

The Perceptron network then classifies the pattern of activity levels as one of the phrases (fifth block), and CSpeechRecognizerVQNN::Recognizer returns the best phrase (sixth block).

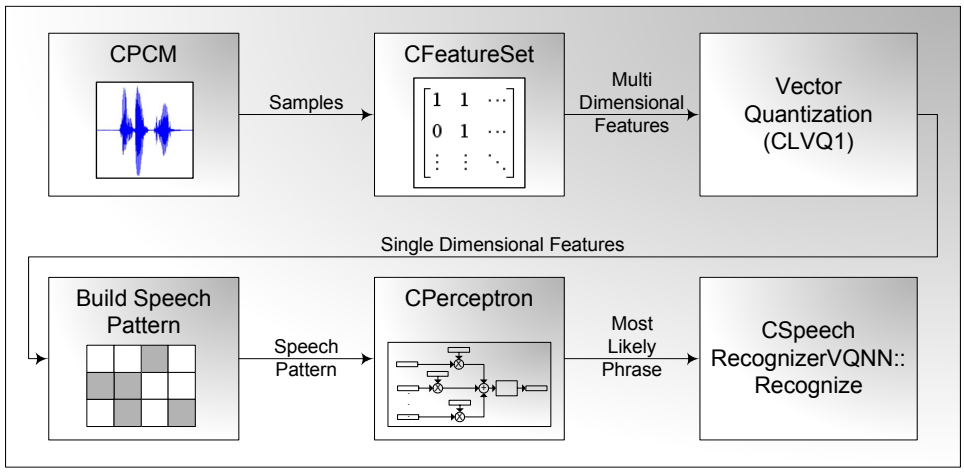


Figure 8.2 – CSpeechRecognizerVQNN Diagram

8.3.3. Dynamic Time Warping Speech Recognition Engine

Dynamic time warping refers to the process of temporally aligning the features in the test word to the reference word. This is necessary since people never speak the same word twice at the exact same rate and with the same emphasis. DTW compensates for this phenomenon.

The DTW recognition process (Figure 8.3) starts off with the regular feature extraction stage (second row of the figure). However, instead of comparing the extracted features to a model representing different words, the features are compared to actual features of training words (third row). A distance metric is then calculated (fourth row). Often times, a simple Euclidean distance is acceptable. This process is performed on every reference word. Whichever reference word has the lowest distance to the test word

is selected as the most likely word (fifth row) and returned by CSpeechRecognizerDTW::Recognize (bottom row).

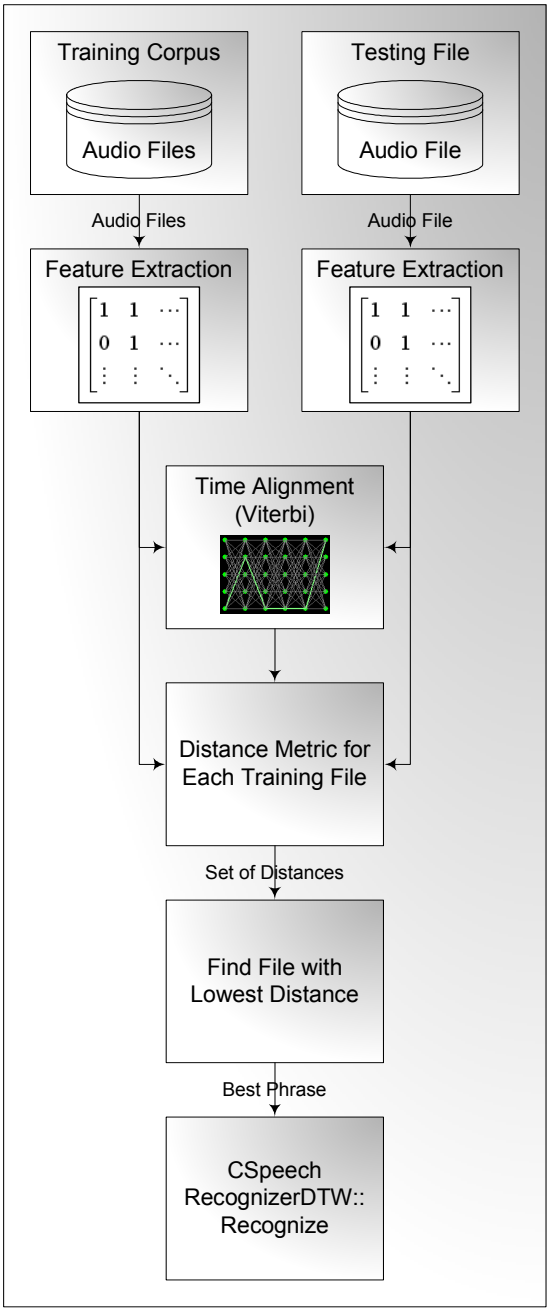


Figure 8.3 – CSpeechRecognizerDTW Diagram

The major downside of DTW is that it is exceedingly inefficient. The DTW algorithm is very slow since the test phrase must be individually compared to every

instance of every word in the lexicon. The DTW algorithm also consumes a considerable amount of memory/storage space due to the fact that it must store examples of every phrase somewhere.

However, it is a simple and reasonably accurate algorithm, and it may be useful to understand its implementation, in order to understand how the classes of AFARR fit together.

CHAPTER 9

USER INTERFACE

With few exceptions, ASR research systems available today lack an easy to understand user interface. Many employ command line interfaces, requiring the user to memorize dozens of command line options, and to sift through large volumes of output sent to the console. AFARR attempts to break from this trend.

9.1. Configuration

AFARR has two means of configuration – a configuration dialog box, and an INI file. The configuration dialog box contains settings that are largely unrelated to each other. For instance, it presents a choice of which engines to test. The INI file contains settings that are related to each other, and are therefore grouped into "profiles." For instance, the INI file can contain both a regular profile, and a high speed/low quality profile for debugging purposes. Profile selection is performed using the configuration dialog.

9.1.1. Configuration Dialog

The configuration dialog (Figure 9.1) is populated by the RegisterSettings function. RegisterSettings calls RegisterSettingString, RegisterSettingBool, RegisterSettingInt, and RegisterSettingDouble in CGlobalSettings to create string, Boolean, integer, and double precision floating point settings in the dialog box.

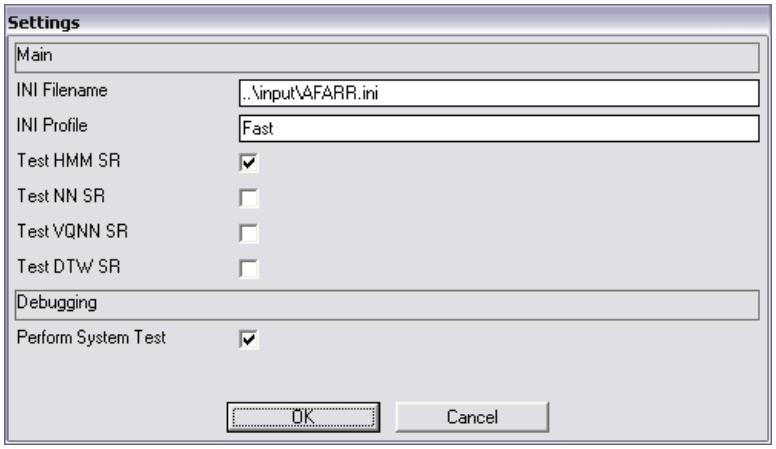


Figure 9.1 – AFARR Configuration Dialog

Settings are identified by section name (e.g. Debugging) and setting name (e.g. Perform System Test). Developers do not need to worry about resource files, message maps, dynamic data exchange (DDX), or any other GUI issues; all of this is automatically handled by the RegisterSettingsXXXX and GetSettingsXXXX functions.

The source code of the RegisterSettings function is included in Figure 9.2. Presently, it creates two string settings and four Boolean settings in the "Main" section, and a single Boolean setting in the "Debugging" section.

```
void RegisterSettings(CGlobalSettings * pcGS)
{
      pcGS->RegisterSettingString("Main", "INI Filename",
      "..\\input\\AFARR.ini");
      pcGS->RegisterSettingString("Main", "INI Profile", "Normal");
      pcGS->RegisterSettingBool("Main", "Test HMM SR", false);
      pcGS->RegisterSettingBool("Main", "Test NN SR", false);
      pcGS->RegisterSettingBool("Main", "Test VQNN SR", false);
      pcGS->RegisterSettingBool("Main", "Test DTW SR", false);
      pcGS->RegisterSettingBool("Debugging", "Perform System Test",
      false);
}
```

Figure 9.2 – RegisterSettingString and RegisterSettingBool Example

To access the string, Boolean, integer, and double precision floating point settings, GetSettingString, GetSettingBool, GetSettingInt, and GetSettingDouble are called. In the example, Figure 9.3, if the "Perform System Test" option is checked (i.e. set to true), the SystemTest function will be called, and the program will log any errors.

```
if (cGS.GetSettingBool("Debugging", "Perform System Test"))
{
      if (SystemTest())
      {
            Log(ERROR_LOG, "Errors detected - worker thread
                  terminating.");
            return 1;
      }
}
```

Figure 9.3 – GetSettingBool Example

9.1.2. INI File

The INI file is a text file containing configuration settings grouped into sections. Sections names are enclosed in brackets. Individual settings (known as keys) are specified on subsequent lines, and separated from their value by an equal sign (=).

Minimal acronyms and abbreviations were used in the names of keys to maximize readability of the INI file. An example AFARR INI file is shown in Figure 9.4. The example contains five sections – Normal, Fast, ZeroOne, Numbers, and Letters. Normal

and Fast contain different profiles. ZeroOne, Numbers, and Letters contain lexicons for the speech recognition engines.

```
[Normal]
UserName = Richard Lynch
TrainingCorpusPath = ..\Input\Corpora\Training
TestingCorpusPath = ..\Input\Corpora\Testing
Phrases = Numbers
Subunits = 15
Deterministic = true
Dictionary = ..\Input\cmudict-0.6d.txt

DefaultLog = ..\Output\asr2.log
ErrorLog = ..\Output\error.log
CSVResultsLog = ..\Output\results.csv

TrainingRate = 0.01

FeatureType = MFCC
FilesToTrainSilenceFrom = 20
FeaturesPerSecond = 40
Derivatives = 2
Dimensions = 13
SamplesPerFeature = 1024
FFTSize = 2048

Passes = 500

###############################################################################

[Fast]
UserName = Richard Lynch
TrainingCorpusPath = ..\Input\Corpora\Training
TestingCorpusPath = ..\Input\Corpora\Testing
Phrases = ZeroOne
Subunits = 5
Deterministic = true
Dictionary = ..\Input\cmudict-0.6d.txt

DefaultLog = ..\Output\asr2.log
ErrorLog = ..\Output\error.log
CSVResultsLog = ..\Output\results.csv

TrainingRate = 0.01

FeatureType = MFCC
FilesToTrainSilenceFrom = 5
FeaturesPerSecond = 20
Derivatives = 0
Dimensions = 13
SamplesPerFeature = 1024
FFTSize = 2048

Passes = 100
```

```
##############################################################################

[ZeroOne]
0 = zero
1 = one

##############################################################################

[Numbers]
0 = zero
1 = one
2 = two
3 = three
4 = four
5 = five
6 = six
7 = seven
8 = eight
9 = nine

##############################################################################

[Letters]
0 = A Adam
1 = B Boston
2 = C Charlie
3 = D David
4 = E Edward
5 = F Frank
6 = G George
7 = H Henry
8 = I Ida
9 = J John
10 = K King
11 = L Lincoln
12 = M Mary
13 = N Nora
14 = O Ocean
15 = P Paul
16 = Q Queen
17 = R Robert
18 = S Sam
19 = T Tom
20 = U Union
21 = V Victor
22 = W Walter
23 = X X Ray
24 = Y Young
25 = Z Zebra
```

Figure 9.4 – Example INI File

The keys in each profile are largely self-explanatory. "UserName" specifies the name of the speaker whose speech is to be used. "TrainingCorpusPath" and

"TestingCorpusPath" specify the root directory of the training and testing corpus, respectively. Phrases specifies the section containing the lexicon. Deterministic determines if the random number generator should be seeded with the same value with each execution of AFARR. Setting it to true is sometimes useful for debugging purposes. Dictionary specifies the filename of the dictionary.

"DefaultLog", "ErrorLog", and "CSVResultsLog" specify the filenames of the log files. The first receives all messages written to the console area of the GUI. The second logs any error messages generated by AFARR. The third log gathers results from testing in comma separated value format, suitable for importing into a spreadsheet program.

"TrainingRate" specifies the training learning rate for algorithms such as LMS. "FeatureType" specifies what sort of features should be used for training and testing. "FilesToTrainSilenceFrom" specifies how many files should be used to train the silence state or model of an engine. It is adjustable since every speech file contains some silence, and training the silence state on every file would be very time consuming.

"FeaturesPerSecond" specifies how often feature vectors should be computed. "Derivatives" specifies how many time derivatives should be taken of the feature vectors. "Dimensions" specifies the number of dimensions the feature vectors should have prior to differentiation. "SamplesPerFeature" specifies how many samples should be used in computing each feature vector (the window size). "FFTSize" specifies the sum of SamplesPerFeature and any zero padding desired. FFTSize should be equal to or greater than SamplesPerFeature. A larger FFTSize or SamplesPerFeature results in better frequency resolution. A smaller SamplesPerFeature results in better time resolution.

The last three sections of the example contain the set of words the speech recognition engine(s) are expected to recognize. They are referenced by the "Phrases" key in the profiles.

It is often times convenient to perform speech recognition using different sets of settings (profiles). Different profiles might be used for the sake of comparison, or perhaps for debugging purposes. In the example, a normal profile has been defined containing typical settings. A fast profile has also been defined to make debugging faster and easier. The fast profile sacrifices accuracy in exchange for speed by using fewer and less complex features.

9.2. Graphical User Interface

The graphical user interface (Figure 9.5) consists of three sections – the text results, the plotting control, and the results table. The text results section displays all log entries sent to DEFAULT_LOG (see Section 10.10). The plotting control typically displays the current CPCM object, and can be accessed by creating a plot with the title "Inline". The results table can be updated by sending comma delimited log entries to CSV_RESULTS_LOG. Writing comma separated entries to the CSV_HEADER log will update the header row of the results table. The results table can display any sort of text or numerical data, but typically displays accuracy results.

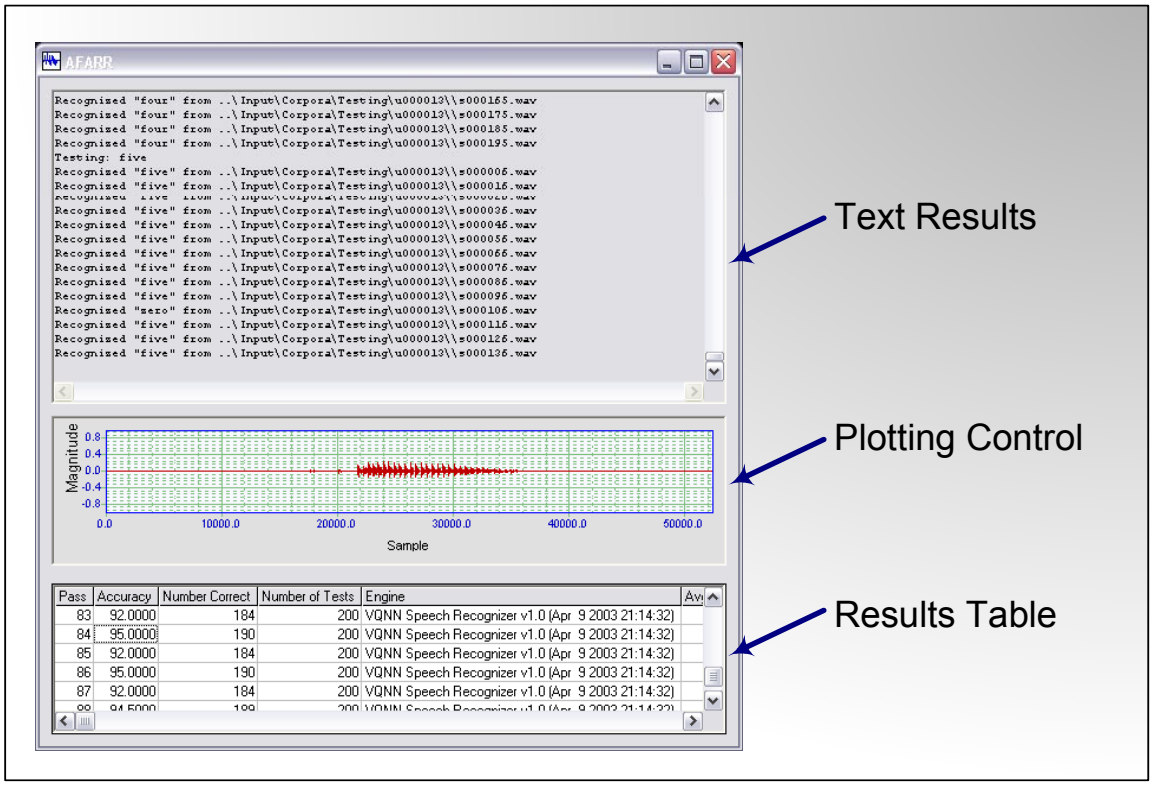


Figure 9.5 – AFARR Graphical User Interface

CHAPTER 10

SUPPORT CLASSES AND FUNCTIONS

10.1. Matrix Class (CMatrix)

CMatrix is one of the most important classes in AFARR. Nearly all classes directly or indirectly manipulate CMatrix objects. CMatrix objects represent the samples of the CPCM class, transition probabilities in CHMM, data to plot in the graphing controls, the mean and covariance in CGaussian, neuron weights in the neural network classes, and many other things.

CMatrix provides the following functionality:

- Binary operators acting on two CMatrix classes, or a CMatrix class and a scalar:
  - Addition, subtraction, multiplication, division, and comparison
- Linear algebra functions:
  - Determinant, inverse, transpose, and augmentation
- Miscellaneous functions:
  - Reshape matrix, shuffle elements in the matrix, normalize a subset of the elements to add up to 1.0, sum of all elements, sum of the square of all elements
- DSP functions:
  - FIR/IIR filtering

- Distance metric to another CMatrix:
    - Euclidean, Itakura-Saito, COSH

Addition, subtraction, multiplication, and division operators for CMatrix have been overloaded, so CMatrix algebraic expressions appear very natural and similar to their Matlab equivalent expression. For instance, to add matrices x and y, and store the result in z, the developer only needs to write "z = x + y;". Order of operations is automatically handled correctly.

The parentheses operator has also been overloaded to access an individual element in the matrix. The arguments are the index of the desired element. In two dimensional matrices, the row index comes first, and the column index comes second. For instance, to access the element at row 5 and column 0 of two dimensional matrix m, one would use the expression "m(5, 0)". It is important to note that CMatrix uses zero-based indices, so a 10 element vector will have entries 0 through 9, not 1 through 10.

Matrix_Support.cpp also defines several functions useful for working with CMatrix objects. The Augment function augments two matrices and returns the result. The ones and zeros functions are similar to their Matlab counterparts – they create a matrix containing just ones or zeros of the specified size. The diag function is also similar to its Matlab counterpart – it takes a one dimensional matrix, and places the element from that matrix onto the diagonal of a new two dimensional matrix. The rand and randn functions behave similar to their Matlab counterparts, generating matrices of random numbers over [0, 1) and normally distributed random numbers. Finally, tanh, cosh, cos, sin, and abs calculate the hyperbolic tangent, hyperbolic cosine, cosine, sine, and absolute value of a CMatrix on an element by element basis.

Since CMatrix is such an important part of AFARR, every effort has been made to make it as intuitive to use, as possible. People familiar with Matlab should have an easy time learning the CMatrix functions, since many CMatrix functions share the same name and arguments as their Matlab counterparts.

However, one important difference is that CMatrix functions involving the creation of a new matrix (e.g. ones, zeros, randn) should include the number of dimensions before the dimensions themselves. For instance, to create a two dimensional 10x4 matrix of ones, one would use the function call "ones(2, 10, 4)." This could not be avoided since C++ provides no means of determining the number of arguments present.

10.2. Gaussians Classes (CGaussian, CGaussianMixture)

Two classes are involved with representing multidimensional Gaussian PDFs – CGaussian and CGaussianMixture. CGaussian represents a single Gaussian PDF (Equation 10.1) with a mean m and covariance matrix U. CGaussianMixture represents a weighted sum of multiple Gaussian PDFs.

$$f(y) = \frac{1}{2\pi\sqrt{|U|}} e^{-\frac{1}{2}(y-m)\cdot U^{-1}\cdot(y-m)^T}$$

Equation 10.1 – Gaussian PDF

Gaussian mixtures are often used to model the likelihood of a phoneme generating a particular feature vector. A single Gaussian PDF is sometimes sufficient for speaker dependent recognition, but for speaker independent recognition, Gaussian mixtures are typically required.

Additionally, due to their squared component, single Gaussian densities decay too fast as the features deviate from the mean. Gaussian mixtures overcome this problem[17]. Figure 10.1 illustrates the problem and the effect of using a Gaussian mixture.

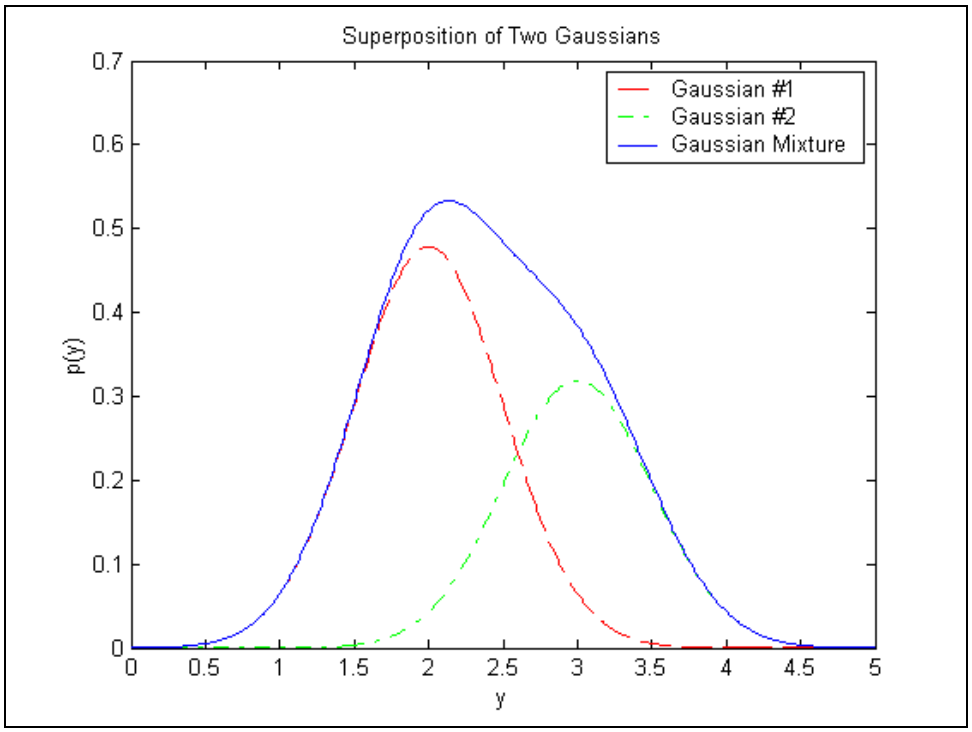


Figure 10.1 – Gaussian Mixture

10.3. Lexicon Class (CLexicon)

CLexicon translates words to and from their phonetic representations. Presently, it only supports dictionaries in the CMU Pronouncing Dictionary format. However, the CMU Pronouncing Dictionary format is quite simple, and it should be easily possible to convert other dictionaries into a compatible format. Each line in the CMU Pronouncing Dictionary consists of a word, followed by a delimiter, followed by its phonetic representation. AFARR will accept either a space or a tab as a delimiter. The phonetic

representation is just a concatenation of the phonemes in Table 10.1, with phonemes separated by a space, and any numbers indicating stress. The CMU Pronouncing Dictionary uses 39 phonemes and three stress levels. A "0" indicates no stress, a "1" indicates primary stress, and a "2" indicates secondary stress.

Phoneme	Example	Translation
AA	odd	AA D
AE	at	AE T
AH	hut	HH AH T
AO	ought	AO T
AW	cow	K AW
AY	hide	HH AY D
B	be	B IY
CH	cheese	CH IY Z
D	dee	D IY
DH	thee	DH IY
EH	Ed	EH D
ER	hurt	HH ER T
EY	ate	EY T
F	fee	F IY
G	green	G R IY N
HH	he	HH IY
IH	it	IH T
IY	eat	IY T
JH	gee	JH IY
K	key	K IY
L	lee	L IY
M	me	M IY
N	knee	N IY
NG	ping	P IH NG
OW	oat	OW T
OY	toy	T OY
P	pee	P IY
R	read	R IY D
S	sea	S IY
SH	she	SH IY
T	tea	T IY
TH	theta	TH EY T AH
UH	hood	HH UH D
UW	two	T UW
V	vee	V IY
W	we	W IY
Y	yield	Y IY L D

Z	zee	Z IY
ZH	seizure	S IY ZH ER

Table 10.1 – CMU Pronouncing Dictionary Phonemes

An example entry of the CMU Pronouncing Dictionary is included in Figure 10.2. In the example entry, the word being transcribed is "talk." It consists of three phonemes, T, AO, and K. The primary stress is on the middle phoneme.

```
TALK  T AO1 K
```

Figure 10.2 – Example CMU Pronouncing Dictionary Entry

Presently, CLexicon is unused. However, it is included for completeness and since future researchers may have a need for it.

10.4. Language Model Class (CLanguageModel)

The CLanguageModel class implements a simple language model. CLanguageModel is capable of calculating unigram, bigram, and trigram word probabilities.

CLanguageModel can be trained on a text file using the Train function. Unigram, bigram, and trigram probabilities can be obtained using the UnigramProb, BigramProb, and TrigramProb functions, respectively. Each of these probabilities is calculated using Equation 10.2[27].

$$P(w_i \mid w_{i-N+1},...,w_{i-1}) = \frac{C(w_{i-N+1},...,w_i)}{C(w_{i-N+1},...,w_{i-1})}$$

Equation 10.2 – N-gram Probability Equation

N represents the order of the language model (e.g. N=3 for trigrams). C(X) represents the number of times word sequence X occurred in the training corpus. The w_i symbol represents the word whose probability is being estimated. The w_{i-1}, w_{i-2}, …

symbols represent the context (previous words). As an example, for N = 3, the equation becomes Equation 10.3.

$$P(w_i \mid w_{i-2}, w_{i-1}) = \frac{C(w_{i-2}, w_{i-1}, w_i)}{C(w_{i-2}, w_{i-1})}$$

Equation 10.3 – Trigram Probability Equation

Equation 10.3 states the probability of word w_i occurring, given w_{i-2} and w_{i-1} were the two most recent words, is equal to the number of times word sequence w_{i-2}, w_{i-1}, w_i has occurred divided by the number of times word sequence w_{i-2}, w_{i-1} has occurred. For sparse training sets, the numerator can easily be zero, leading to an impossible word sequence. In extreme cases, both the numerator and denominator can be zero, leading to an undefined probability. When the denominator is zero, bigram, or unigram probabilities can be used instead of the trigram probability. Additionally, smoothing techniques, such as optimal linear smoothing[17], can compensate for the numerator being zero.

If the numerator of Equation 10.3 is equal to zero, the UnigramProb, BigramProb, and TrigramProb function will return 0. If the denominator of Equation 10.3 is equal to zero, the UnigramProb, BigramProb, and TrigramProb function will return -1.

10.5. Speech Database Class (CSpeechDB)

Working with individual audio files is sometimes useful, but more commonly, it is necessary to work with a large group of speech files from a diverse set of speakers. Saying a speech recognition method achieves 95% accuracy is more impressive when the

testing set consists of 1,000 samples from a wide variety of speakers, than from 20 samples of a single speaker's speech.

CSpeechDB keeps track of a collection of audio files. Presently, it only supports audio files organized by the Microsoft Audio Collection tool, but it can be easily extended to support other organization methods. Once CSpeechDB is initialized, it can be queried for the filenames of all files matching a certain criterion. A query might ask for all filenames of "one" being spoken, or all audio files containing speech from a certain user.

The Microsoft Audio Collection tool organizes speech files into a series of sequentially numbered directories. Each directory contains a collection of sequentially numbered .WAV files and a user.ini file. As with most INI files, the user.ini file contains a set of settings grouped into sections. Sections are enclosed in brackets. Individual settings (keys) consist of a name and a value, separated by an equal sign (=).

The user.ini file contains three sections – Sentences, AdminInfo, and UserInfo.

The key names for the Sentences section are sequential numbers, starting with 0. The string assigned to the key is the phrase present in the .WAV file corresponding to the key name.

The AdminInfo section contains information about the microphone, sound card, language, institution, and other information specified by the administrator of the audio collection.

The UserInfo section contains the speaker's name, age, gender, phone number, speech recognition experience, dialects, and any comments s/he has.

An example user.ini is shown in Figure 10.3.

```
[Sentences]
1=main screen
2=radar
3=lock
4=stop
5=front antenna
6=volume increase
7=volume decrease
8=records
9=plate type
10=license number
11=license state
12=gender
13=10 20 New Hampshire
14=10 20 Florida
15=10 21 Florida
16=10 22 Florida
17=G P S
18=report
19=help
20=video
21=play
22=rewind
23=wider
24=focus
25=microphone
26=lights
27=wigwags
28=wigwags off
29=pierce
30=pursuit
31=sirens off
32=male
33=female
34=radio
35=A Adam
36=B Boston
37=C Charlie
38=V Victor
39=Mississippi
40=Massachusetts

[AdminInfo]
PhrasePath=.\phrase.ini
UserDir=c:\users\rll\collect
Microphone=Andrea 4 Element DA-310
SoundCard=Sound Blaster Live 1024
Language=English
Institution=UNH
SamplesPerSec=44100
UserNum=1
StartPhrase=1
NumPhrases=40
Discrete=0
Dialect0=Standard
Dialect1=
Dialect2=
```

```
Dialect3=
Dialect4=
Dialect5=
Dialect6=
Dialect7=

[UserInfo]
Name=Michael Bressack
Gender=Male
Age=21
PhoneNumber=
Experience=Beginning User
Dialect=Standard
Comments=
```

Figure 10.3 – Example user.ini from the Microsoft Audio Collection Tool

10.6. Serialization Class (CSerialization)

Most classes in AFARR support serialization – the ability to load and save their state to a file. All classes supporting serialization are derived from CSerialization. CSerialization provides the ability to serialize some basic data types – int, double, char *, bool, and unsigned long. Each of these can be loaded/saved with a call to CSerialization::SerializeItem. Additionally, CSerialization::SerializeItem can serialize any class derived from CSerialization, by calling that class' SerializeObject function.

Classes derived from CSerialization must define how they should be serialized using their SerializeObject function. SerializeObject should either load the entire class from a file, or save the entire class to a file, depending on the arguments supplied. SerializeObject is automatically called by SerializeItem, so there is no need for developers to actually call the SerializeObject function of any class.

Figure 10.4 and Figure 10.5 demonstrate how CPerceptron implements serialization. CPerceptron is derived from CSerializable (Figure 10.4), and implementes a SerializeObject function. The SerializeObject function calls SerializeItem for each

member variable (Figure 10.5). Three double precision floating point numbers, one integer, and one CArrayExSrl object are serialized. Although CArrayExSrl is not one of the basic types SerializeItem supports, CArrayExSrl is derived from CSerializable, and therefore can be serialized through its SerializeObject function (automatically called).

```
class CPerceptron : public CSerializable
{
      // Public Functions
      public:
      ...
            // Serialization
            bool SerializeObject(FILE * hFile, bool bSave);
      ...
      // Protected Variables
      protected:
            CArrayExSrl<CMatrix,CMatrix> m_arWeights;
            double m_fActivationSlope;
            int m_nLayers;

      // Private Variables
      private:
            CStringSrl m_strActivationFunction;
            double m_fInputScaling;
            double m_fOutputScaling;
};
```

Figure 10.4 – CPerceptron Declaration

```
bool CPerceptron::SerializeObject(FILE * hFile, bool bSave)
{
      if (!bSave)
            Clear();

      if (SerializeType(hFile, bSave, "CPerceptron"))
            return true;

      SerializeItem(hFile, bSave, m_arWeights);
      SerializeItem(hFile, bSave, m_fActivationSlope);
      SerializeItem(hFile, bSave, m_nLayers);
      SerializeItem(hFile, bSave, m_strActivationFunction);
      SerializeItem(hFile, bSave, m_fInputScaling);
      SerializeItem(hFile, bSave, m_fOutputScaling);

      return false;
}
```

Figure 10.5 – CPerceptron::SerializeObject Definition

10.7. Array Classes (CArrayEx, CArrayExSrl)

10.7.1. CArrayEx

The CArrayEx class extends the Microsoft Foundation Classes (MFC) array class, CArray, to support a copy constructor and assignment operator. A copy constructor is necessary so arrays can be passed by value in addition to being passed by reference. An assignment operator is also handy so arrays can be copied as a whole, rather than having to copy the individual elements from one array to another. It is uncertain why Microsoft chose to not include this functionality in CArray in the first place.

Both CArray and CArrayEx are template classes – they can represent any sort of data. The type of data represented by the array is specified in the array declaration. For instance, CArrayEx<CString, CString> represents an array of CStrings. The first template argument specifies the type of objects stored in the array. The second template argument specifies the type of object used to access the objects in the array. Typically, both arguments are the same type.

10.7.2. CArrayExSrl

The CArrayExSrl class extends CArrayEx to support AFARR style serialization. It adds the convenience of being able to serialize an array of elements with one command.

10.8. String Class (CStringSrl)

The string class, CStringSrl is an extension of the MFC CString class. It adds AFARR style serialization, and a new printf style constructor.

The printf style constructor provides the convenience of being able to supply formatted strings as arguments to functions, without the need for temporary variables.

Normally, to display two integers separated by a comma in a message box, the following commands would be used:

```
CString strTmp;
strTmp.Format("%d,%d", integer_a, integer_b);
AfxMessageBox(strTmp);
```

Figure 10.6 – Displaying a Message Box with CString

However, with CStringSrl, this can be reduced to one line:

```
AfxMessageBox(CStringSrl("%d,%d", integer_a, integer_b));
```

Figure 10.7 – Displaying a Message Box with CStringSrl

10.9. Global Settings Class (CGlobalSettings)

The CGlobalSettings class is responsible for loading the AFARR INI file. All of the settings are stored in public member variables of CGlobalSettings for other classes to use. Table 10.2 lists the CGlobalSettings member variables intended for developer use.

Member Variable	Description
m_nUnits	Number of units (phrases, words, etc.)
m_nSubunits	Number of HMM nodes per unit.
m_nFilesToTrainSilenceFrom	Number of files used to train the silence node.
m_strUser	Selected user. Blank indicates all users are selected.
m_strTrainingCorpusPath	Root directory of the training corpus.
m_strTestingCorpusPath	Root directory of the testing corpus.

m_strDictionary	Dictionary path.
m_fTrainingRate	HMM update rate.
m_nPasses	Number of training/testing passes.
m_cFT	Feature description.
m_arPhrases	Active lexicon.
m_bDeterministic	True indicates a fixed pseudo-random number generator (PRNG) seed. False indicates a random PRNG seed.

Table 10.2 – CGlobalSettings Member Variables

10.10. Logging Function (Log)

Results can be logged to the console and a text file by using the Log function. The first argument of the Log function specifies the logging destination. Four destinations are presently valid: DEFAULT_LOG, ERROR_LOG, CSV_RESULTS_LOG, and CSV_HEADER. DEFAULT_LOG is where most informational messages are sent. Messages sent to DEFAULT_LOG are displayed in the upper section of the main dialog box.

ERROR_LOG is intended for logging serious errors. Optionally, the error message may be sent to a modal dialog box, thereby forcing user intervention.

Comma separated value (CSV) results should be sent to the CSV_RESULTS_LOG destination. A table will be generated in the results table of the AFARR GUI with the results. The header column can be updated by sending the header column names in comma separated value format to the CSV_HEADER destination.

The second through final arguments of the Log function are printf style arguments. For instance, Log(DEFAULT_LOG, "%.1f", 123.123) would log "123.1" to the default log file and the text results section of the main AFARR GUI.

The location of the log files can be updated by modifying the INI file.

10.11. Plotting Functions (Plot, PlotTrellis)

Often times, it is helpful to be able to visualize data with a plot. AFARR developers can quickly generate a plot with a simple Plot call, while users desiring more control can modify the CGraphSettings object returned by the Plot call to tweak the plot. A diagram of the plot window is shown in Figure 10.8.

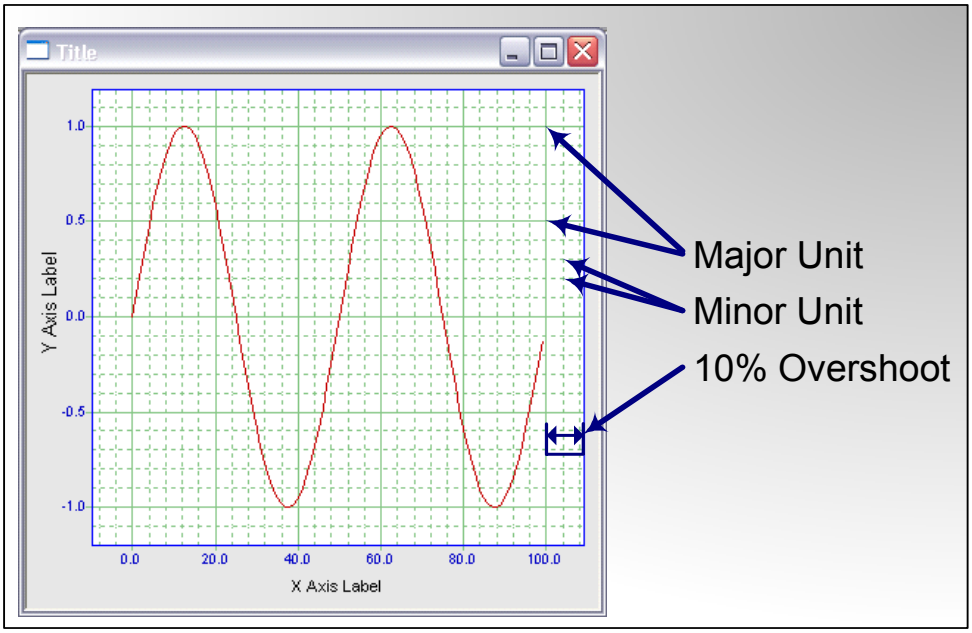


Figure 10.8 – Plot Diagram

The function prototype for the Plot function is shown in Figure 10.9. Three arguments are required, and two are optional. The three required arguments are the X and Y matrices, and the title for the plot. The X and Y matrices should be one

dimensional, but will automatically be flattened column by column into a one dimensional array if they are higher dimensional. The title of the plot will go in the title of the plot window. If a plot window already exists with the same title, the same plot window will be reused. Otherwise, a new plot window will be created. If a plot title of "Inline" is given, the plot will be displayed in the main AFARR window. The two optional arguments are the X and Y axis labels.

```
class CGraphSettings * Plot(CMatrix & cX, CMatrix & cY, const char *
      pszTitle, const char * pszXLabel = NULL, const char * pszYLabel =
      NULL);
```

Figure 10.9 – Plot Function Prototype (1)

An alternate Plot function prototype is shown in Figure 10.10. This function call is intended for graphically displaying CPCM objects.

```
class CGraphSettings * Plot(CPCM & cPCM, const char * pszTitle =
      "Inline", const char * pszXLabel = "Sample", const char *
      pszYLabel = "Magnitude");
```

Figure 10.10 – Plot Function Prototype (2)

A PlotTrellis function (Figure 10.11) is also provided for plotting trellises, such as those searched by the Viterbi algorithm. The number of states and time steps must be specified, and the title, and X and Y axis labels are optional.

```
class CGraphSettings * PlotTrellis(int nStates, int nTimes, const char
      * pszTitle = "Trellis", const char * pszXLabel = "Time", const
      char * pszYLabel = "State");
```

Figure 10.11 – PlotTrellis Function Prototype

All of the plotting functions return a pointer to a CGraphSettings object. This object can be used to manipulate the plot after the Plot call. Figure 10.12 shows the declaration of the CGraphSettings class.

```
class CGraphSettings
{
      public:
```

```
            CGraphSettings();
            ~CGraphSettings();

            void SetupTrellis(int nStates, int nTimes);

            CDataSeries * AddSeries();
            CDataSeries * GetSeries(int iSeries);
            CDataSeries * GetSeries(int iPresentState, int iNextState);
            void DeleteSeries(int iSeries);
            void Clear();
            int GetNumberOfTrellisStates();

      public:
            CString m_strXAxisLabel;
            CString m_strYAxisLabel;
            CString m_strTitle;
            CString m_strXAxisFormat;
            CString m_strYAxisFormat;

            int m_iChartType;

            double m_fYMinimum;
            double m_fYMaximum;
            double m_fXMinimum;
            double m_fXMaximum;

            bool m_bTitle;
            bool m_bXAxisLabel;
            bool m_bYAxisLabel;

            bool m_bXMajorGridLines;
            bool m_bXMinorGridLines;
            bool m_bYMajorGridLines;
            bool m_bYMinorGridLines;

            bool m_bAutoXMajorUnits;
            bool m_bAutoXMinorUnits;
            bool m_bAutoYMajorUnits;
            bool m_bAutoYMinorUnits;

            double m_fXMajorUnit;
            double m_fYMajorUnit;
            double m_fXMinorUnit;
            double m_fYMinorUnit;

            int m_nMajorGridLineStyle;
            int m_nMinorGridLineStyle;

            double m_fAutomaticLimitsOvershoot;

            int m_iControlMargin;
            int m_iTitleMargin;
            int m_iXAxisLabelMargin;
            int m_iYAxisLabelMargin;

            int m_iXTickSize;
            int m_iYTickSize;
```

```
            COLORREF m_clrBackground;
            COLORREF m_clrPlotArea;
            COLORREF m_clrMajorGridLines;
            COLORREF m_clrMinorGridLines;
            COLORREF m_clrOutlines;
            COLORREF m_clrTitle;
            COLORREF m_clrXAxisLabel;
            COLORREF m_clrYAxisLabel;
            COLORREF m_clrXAxisText;
            COLORREF m_clrYAxisText;

            LOGFONT m_lfTitleFont;
            LOGFONT m_lfXAxisFont;
            LOGFONT m_lfYAxisFont;
            LOGFONT m_lfXAxisLabelFont;
            LOGFONT m_lfYAxisLabelFont;

            bool m_bUpdate;

      private:
            CArrayExSrl<CDataSeries, CDataSeries> m_arSeries;

            int m_nTrellisStates;
};
```

Figure 10.12 – CGraphSettings Declaration

Whenever a CGraphSettings member variable is updated, the m_bUpdate flag should be set to true. Otherwise, the plot will not refresh itself.

The first three strings are automatically filled in by the Plot calls, and contain the title and axis labels. The two axis format strings contain printf style format descriptions for the axis labels. By default, they are equal to "%.1f", meaning axis labels should contain a single number with one decimal place precision.

The m_iChartType element should not be modified, and is automatically filled in by the Plot command. The m_fXMaximum, m_fXMinimum, m_fYMinimum, and m_fYMaximum elements define the axis limits for the plot. If the maximum is equal to the minimum, then the limits are automatically calculated from the supplied data.

The next three Booleans determine whether the title, X-axis label, and Y-axis label are shown. By default, they are all set to true. The following four Booleans determine

whether the major and minor grid lines are shown. The next four Booleans, m_bAutoXMajorGridLines, m_bAutoYMajorGridLines, m_bAutoXMinorGridLines, amd m_bAutoYMinorGridLines determine whether or not the major and minor units should be automatically calculated. If any of these are set to false, the corresponding double from m_fXMajorUnit, m_fXMinorUnit, m_fYMajorUnit, and m_fYMinorUnit should be set appropriately.

m_nMajorGridLineStyle and m_nMinorGridLineStyle contain a CPen style pen style. A list of valid pen styles is shown in Table 10.3. These two variables determine the style of the major and minor grid lines.

Value	Description
PS_SOLID	Creates a solid pen.
PS_DASH	Creates a dashed pen. Valid only when the pen width is 1 or less, in device units.
PS_DOT	Creates a dotted pen. Valid only when the pen width is 1 or less, in device units.
PS_DASHDOT	Creates a pen with alternating dashes and dots. Valid only when the pen width is 1 or less, in device units.
PS_DASHDOTDOT	Creates a pen with alternating dashes and double dots. Valid only when the pen width is 1 or less, in device units.

Table 10.3 – Pen Styles[28]

The m_fAutomaticLimitsOvershoot element defines how much of the axis limits should overshoot the actual data. For instance, if the x values of the data range from 0 to

100, an overshoot of 0.1 (i.e. 10%) would result in the graphing control setting the limits to -10 and 110 (see Figure 10.8). This only applies if automatic axis limits are used.

The following six values define margin sizes and the size of tick marks on the plot. They normally will not need to be modified.

The group of COLORREF variables define the colors for the plot and are fairly self explanatory. COLORREF values are generated by the RGB (red, green, blue) macro. RGB(255, 0, 0) corresponds to red, RGB(0, 255, 0) corresponds to green, RGB(0, 0, 255) corresponds to blue, etc.

The LOGFONT structures describe the fonts to be used for title, axes, and axis labels. For convenience, a ConfigureLogFontStructure function was written which automatically sets up the structure for the provided font name and font size (in points).

CHAPTER 11

RESULTS/DISCUSSION

11.1. Compile Time Verification

Good coding practices were used throughout the source code to maximize source code readability and minimize the likelihood of bugs. Some of the coding conventions used in AFARR are included in Appendix A. AFARR compiles successfully, and therefore, contains no syntax errors.

11.2. Run Time Verification – Basic Functionality

Good coding practices do not completely eliminate bugs. Many bugs are caught by the compiler (e.g. syntax errors), but many will escape its cursory checks. In large applications, such as AFARR, minor bugs in one class can cause seemingly random errors to occur in other classes. Tracking down the true source of such bugs can be a tedious and time consuming process.

To deal with this problem, many of the key classes have a testing function that ensures the class is functioning properly. AFARR can be configured to call each of these functions during start up. Each class is tested as independently of all other classes, as is possible. Testing functions are listed in Table 11.1.

Functionality	Testing Function
CMatrix Class	CMatrixTest
Matrix Support Functions	MatrixSupportTest
CFeature Class	CFeatureTest
CGaussian Class	CGaussianTest
CPerceptron Class	CPerceptronTest
CSequenceRNN Class	CSequenceRNNTest
CHashTable, CHashChain Classes	CHashTableTest
General Support Functions	SupportTest
FFT Functions	FFTTest

Table 11.1 – Testing Functions

An example of a testing function is included in Figure 11.1. The figure shows an excerpt from the CMatrix test function, CMatrixTest. In the figure, two matrices are instantiated, and initialized as two element 1-d matrices. The first matrix is assigned [3 4]. The second matrix is assigned [6 8]. The Euclidean distance between the two matrices is then evaluated. If the distance does not come out to 5, an error message is shown. The nErrors variable is also incremented, which will cause the program to halt after all of the tests are complete (not shown).

```
CMatrix cA, cB;

cA.Init(1, 2);
cB.Init(1, 2);

cA(0) = 3;
cA(1) = 4;

cB(0) = 6;
cB(1) = 8;

if (cA.Distance(cB, "Euclidean") != 5)
{
      nErrors++;
      Log(ERROR_LOG, "Euclidean distance error (%.8f)",
            a.Distance(b, "Euclidean"));
}
```

Figure 11.1 – CMatrixTest Excerpt

AFARR passes all of the testing functions, and therefore, the functionality listed in Table 11.1 operates correctly.

11.3. Run Time Verification – Speech Functionality

Speech functionality was tested by performing classic isolated digit recognition. Each engine was trained on a corpus of 405 utterances (approximately 40 of each digit) and tested on a corpus of 200 utterances (20 of each digit). One speaker (the author) was used. The speech recognition accuracy results were measured for clean speech – no noise was added to the training or testing audio files. Results were measured on a Pentium 4 2.26GHz system with 512MB of random access memory (RAM) running Windows XP. The results are presented in the following sections.

11.3.1. HMM Speech Recognition Engine Results

The HMM speech recognition engine settings are listed in Table 11.2.

Description	Value
Feature Type	MFCC
Features Per Second	50
Time Derivates	2
Number of Dimensions per Feature	39
MFCC Window Size	1024
MFCC FFT Size	2048 (1024 samples of zero stuffing)
Silence Trained From	20 Files
HMM Nodes Per Phrase	15
Training Rate	0.01
Minimum Variance Limit	0.01
Minimum Transition Probability Limit	10^{-4}
Gaussians per PDF	1

Table 11.2 – HMM SR Settings

Initial HMM speech recognition engine results were poor, achieving only 86.5% accuracy. However, initial results were obtained using a window and FFT size of 512 samples, and a feature rate of 40 per second. When the HMM speech recognition engine was retested using a 1024 sample window with 1024 samples of zero stuffing, and features were extracted 50 times per second, accuracy increased to 99.0% (see Table 11.3). Baum-Welch training was unable to improve the accuracy beyond the initial 99.0% (Figure 11.2). Presumably, 99.0% is the maximum accuracy attainable using the training set. A table of results is included in Appendix C.

Metric	Result
Highest Accuracy	99.0%
Average Training Time Per Phrase	6.4 seconds
Average Recognition Time Per Phrase	3.2 seconds

Table 11.3 – HMM SR Results

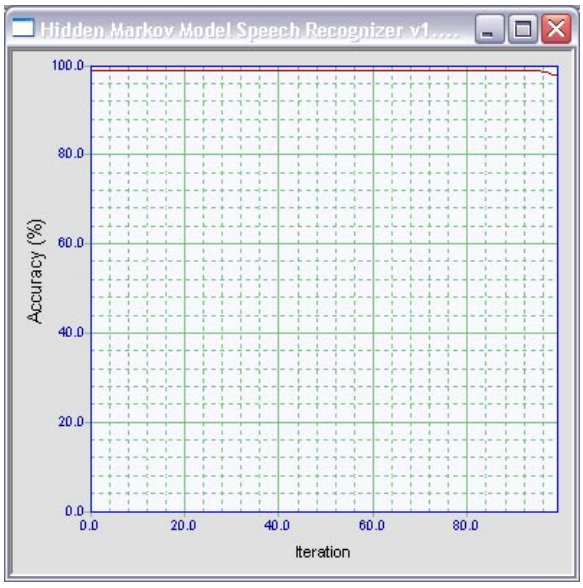


Figure 11.2 – HMM Engine Accuracy vs. Training Iterations

11.3.2. Neural Network Speech Recognition Engine Results

Neural network speech recognition engine settings are listed in Table 11.4.

Description	Value
Feature Type	MFCC
Features Per Second	50
Time Derivates	2

Number of Dimensions per Feature	39
MFCC Window Size	1024
MFCC FFT Size	2048 (1024 samples of zero stuffing)
Vector Quantization Points	256
Perceptron Learning Rate Scheduling Algorithm	Search Then Converge
Learning Rate	0.1
Time Constant	50

Table 11.4 – Neural Network SR Settings

Initially, it was felt that a suitable number of LVQ1 points could be chosen by plotting the mean Euclidean distance from speech features to an LVQ1 neuron vs. the number of neurons (Figure 11.3). A sharp falloff in the distance would indicate a sufficient number of neurons. However, since there were no sharp falloffs in the plot to indicate an appropriate number of neurons, 256 was selected as the number of LVQ1 neurons. It is a typical codebook size used in speech feature vector quantization[29].

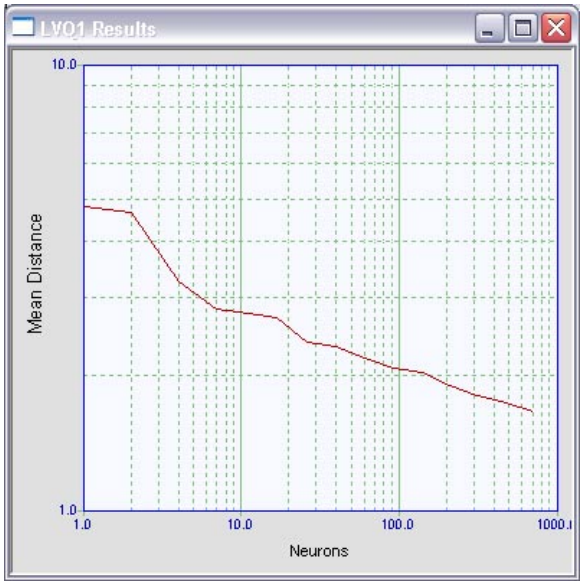


Figure 11.3 – LVQ1 Mean Error vs. Number of Neurons

A summary of the results is included in Table 11.5. A table containing the complete results is included in Appendix D. The accuracy of the engine as a function of the number of training iterations is shown in Figure 11.4. The neural networks took 59 training iterations to converge. Testing was stopped after 119 training iterations.

Metric	Result
Highest Accuracy	95.0%
Average Training Time Per Phrase	2.3 seconds
Average Recognition Time Per Phrase	2.4 seconds

Table 11.5 – Neural Network SR Results

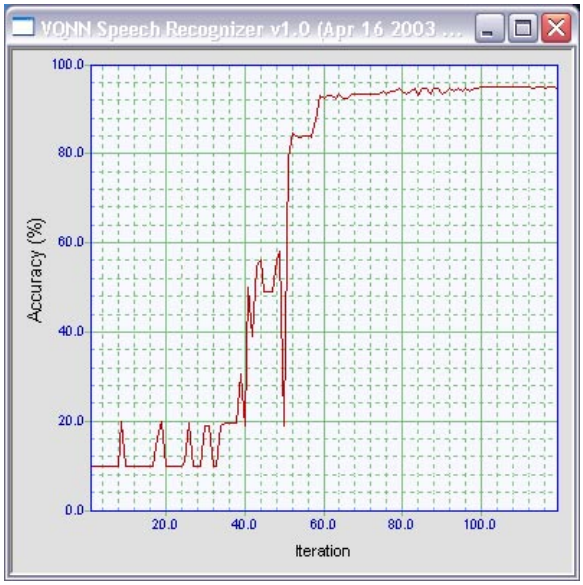


Figure 11.4 – Neural Network Engine Accuracy vs. Training Iterations

11.3.3. DTW Speech Recognition Engine Results

The DTW speech recognition engine settings are listed in Table 11.6.

Description	Value
Feature Type	MFCC
Features Per Second	40
Time Derivates	2
Number of Dimensions per Feature	39
MFCC Window Size	1024
MFCC FFT Size	2048 (1024 samples of zero stuffing)

Table 11.6 – DTW SR Settings

The DTW speech recognition engine had both the highest accuracy of the three engines, 100%, and the longest recognition time, 6.9 seconds (see Table 11.7). The long recognition time is to be expected since dynamic time warping is not intended for use in production systems, but is rather typically used as an educational tool for people who are new to speech recognition.

Metric	Result
Highest Accuracy	100%
Average Training Time Per Phrase	2.0 seconds
Average Recognition Time Per Phrase	6.9 seconds

Table 11.7 – DTW SR Results

No plots were included of the DTW accuracy vs. training iterations since additional training iterations have no effect on DTW accuracy.

11.4. Doxygen

One of the challenges of any large program is maintaining proper documentation. It is one of the least glamorous parts of software development, one that is often neglected, but also one of the most important, if others are to work on the code. To make the generation of documentation as painless as possible, Doxygen[30], an automated documentation extraction system was used. Doxygen can generate an online/offline manual for a set of source code. It can automatically extract the structure of C++ files, determine class relationships, and build diagrams from this information.

The format of the comments in the source code is very natural. Minimal effort is required to adapt existing comments to conform to the Doxygen comment format. An example Doxygen comment block is shown in Figure 11.5.

```
////////////////////////////////////////////////////////////////////////
/// \brief
/// Ensure all the classes are working correctly
///
/// Return:
///    false     - Everything is okay
///    true      - Failure somewhere
///
////////////////////////////////////////////////////////////////////////

bool SystemTest()
{
```

Figure 11.5 – Example Function Description

The comment block consists of two section – a short description, and a long description. The short description is the line following the "\brief" keyword. It is typically one line long, and is included in summaries. The longer description is separated from the short description by a blank line. It can be any length and describes the

function/class in more detail. The comment block should precede the function being commented.

It is important to note that three slashes are required for Doxygen to process the comment block. Doxygen (intentionally) ignores regular C++ comments with only two slashes.

It is also possible to comment the preceding class/function/variable by using "///<" instead of "///". In Figure 11.6, two variables in a class are documented. In the first case, m_nDim, the comment precedes the variable declaration, and in the second case, the comment follows the variable declaration.

```
class CHMM : public CSerializable
{
      // Other declarations
      // ...
      // Protected Variables
      protected:
            /// The number of dimensions
            int m_nDim;

            double m_fUpdateThreshold; ///< The minimum number of times
                  a transition must occur to allow a mean or
                  covariance matrix update
```

Figure 11.6 – Example Variable Descriptions

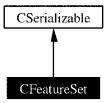
Additional functionality and comment styles are available, but the two comment styles shown above were the primary methods used with AFARR.

Examples of the Doxygen output are included in Figure 11.7 and Figure 11.8. The Doxygen documentation itself is included on the thesis CD-ROM.

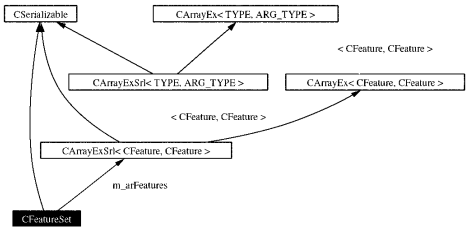
5.6 CFeatureSet Class Reference

Collection of features.

Inheritance diagram for CFeatureSet:



Collaboration diagram for CFeatureSet:



Public Member Functions

- CFeatureSet ()
  Default constructor.
- CFeatureSet (const CFeatureSet &cArg)
  Copy constructor.
- ~CFeatureSet ()
  Destructor.
- void Init ()
  Initialize the feature set.
- void Copy (CFeatureSet &cArg)
  Copy another feature set into this one.

Figure 11.7 – Example Doxygen Output (1)

5.6.3.4 CMatrix CFeatureSet::SimpleHMM (CMatrix * *pcP*, CMatrix * *pcMean*, CMatrix * *pcVar*, int *nDim*, int *nObservations*)

Build a simple HMM in this feature set for testing purposes.

The HMM generates observations based on transitions.

```
pcP           - Probability matrix.
                First index = next state.
                Second index = present state.
pcMean        - Mean matrix.
                First index = next state.
                Second index = present state.
pcVar         - Variance matrix.
                First index = next state.
                Second index = present state.
nDim          - Number of dimensions of each feature
nObservations - Number of observations
```

Currently always starts in state 0

Returns 1-d CMatrix of the actual state sequence

Definition at line 304 of file feature_set.cpp.

```
305 {
306     int state;
307     int index;
308     CMatrix cRtn;
309
310     Clear();
311
312     cRtn.Init(1, nObservations + 1);
313
314     state = 0;
315     cRtn(0) = state;
316
317     m_arFeatures.SetSize(nObservations);
318
319     for (index = 0; index < nObservations; index++)
320     {
321         double r;
322         int next_state;
323         int i;
324         CFeature cFeature;
325         double orig_r;
326
327         orig_r = r = ranf();
328
329         for (i = 0; r >= 0 && i < pcP->Bounds(0); i++)
330             r -= (*pcP)(i, state);
```


Figure 11.8 – Example Doxygen Output (2)

11.5. Discussion

Good coding practices and careful coding have resulted in source code that compiles correctly. Additionally, all of the classes discussed in Section 11.2 pass their respective test functions, indicating that those classes are not only valid from a syntax standpoint, but also from a functional standpoint.

All of the results of the isolated digit recognition testing are reasonable and summarized in Table 11.8. The DTW recognition engine was the most accurate with 100% accuracy in speaker dependent mode. However, it was also the slowest, taking 6.9 seconds to recognize each phrase. The HMM recognition engine had the second best accuracy, 99.0%, and but still a long recognition time, 3.4 seconds. The neural network engine was fastest at 2.4 seconds per phrase, but also the least accurate, with 95.0% accuracy. Endpoint information was cached, so the training and testing times do not include the time required to perform endpoint detection. Accuracy could be improved through a variety of means. However, the goal of AFARR is to provide a starting point for speech research, not a completely optimized production speech recognition system.

Engine	Highest Accuracy	Average Training Time Per Phrase	Average Recognition Time Per Phrase
Hidden Markov Model Engine	99.0%	6.4 sec	3.2 sec
Neural Network Engine	95.0%	2.3 sec	2.4 sec
Dynamic Time Warping Engine	100%	2.0 sec	6.9 sec

Table 11.8 – Results Summary

CHAPTER 12

CONCLUSION

AFARR fills in an important gap in the speech recognition field. It is the only speech recognition research package available with good documentation and coded in a fast and efficient language. Other packages exist, but are implemented in less efficient languages, thereby incurring a (sometimes severe) performance hit, or lack documentation, and are therefore difficult to extend or modify. AFARR also features some graphing functions for visualizing data, and an easy to use user interface. The learning curve for AFARR is much more mild than the learning curve of other speech recognition packages.

12.1. Status

Although the accuracy of the AFARR engines is below the accuracy of commercial engines, the goal of AFARR was not to produce a commercial grade, fully optimized engine, but rather an easy to use and extend framework for future researchers. The goal has been achieved. AFARR implements three reference speech recognition engines in C++ with a succinct graphical user interface.

Much attention was paid to ensuring the source code was structured in an intuitive and consistent manner. Object oriented programming techniques ensured the code was modular and packaged together related functions. Abstract base classes were used in certain cases to ensure similar classes shared a similar interface.

The source code is heavily documented, and technical documentation is available in HTML[31], PDF[32], and Windows Help File (CHM)[33] format.

AFARR is presently configured using an INI file and a GUI dialog box. Additional settings can easily be added – typically requiring only a line or two of extra code.

The thesis CD-ROM includes this thesis, the thesis defense, AFARR source code, Doxygen documentation, and the CATlab corpus. Maintaining the documentation is trivial – the comment format is easy to learn and documentation can be automatically recompiled by calling the "Make Documentation.cmd" script on the thesis CD-ROM.

12.2. Future Work

12.2.1. Phoneme Based Recognition

The three speech recognition engines implemented in AFARR are word/phrase based recognition engines. The engines recognize entire words rather than individual phonemes. This approach is sufficient for small vocabularies, but does not scale well to large vocabulary systems (e.g. a dictation system). For large vocabulary speech recognition systems, a phoneme based recognizer is necessary.

Dynamic time warping, and the neural network approach used in AFARR are unsuitable for phoneme based speech recognition. However, adapting the AFARR HMM speech recognition engine to phoneme based recognition should not be very difficult. Ideally, the training data would have already been segmented (i.e. the first and last sample of each phoneme would be known). The individual phonemes of each audio file would then be extracted from a CPCM object, and used to train the Hidden Markov Model, instead of entire words. Minor modifications would be necessary to the speech

database, to store the segmentation data; to the Hidden Markov Model class, to connect the phonemes to each other; and the CSpeechRecognizerHMM class, to properly translate phonemes into words.

12.2.2. Real Time Recognition

Pseudo-real-time speech recognition is easily possible by performing real time endpoint detection on the samples from the microphone in real time, and then feeding the resulting speech to the recognition engines.

To perform true real time speech recognition, endpoint detection would also have to be performed on the samples from the microphone in real time. However, once speech is detected, feature extraction would also begin, and be performed in real time. The extracted features could then be fed into the Viterbi algorithm (HMM or DTW engine) or used to build the input to the neural network (NN engine). These are moderately difficult changes since state information would have to be maintained between successive calls to the Viterbi algorithm or the neural network pattern building code. Presently, the code assumes all of the features of a phrase are available when the CSpeechRecognizer::Recognize function is called.

12.2.3. Miscellaneous Enhancements

There is much potential for future work with AFARR. Compatibility with IFCs would enable researchers to enjoy the advantages of both systems. A faster and more accurate endpoint detection would be beneficial. Additional acoustic modeling and feature extraction methods would increase AFARR's usefulness. Additional types of

neural networks could easily be implemented. A Matlab interface would be highly beneficial – melding the ease of Matlab with the speed of C++ code. AFARR can serve as a starting point for research into any of these areas.

REFERENCES

[1] Slaney, Malcolm. Auditory Toolbox. Interval Research Corporation: Technical Report #1998-010, 1998. <http://rvl4.ecn.purdue.edu/~malcolm/interval/1998-010/>

[2] The MathWorks – Neural Network Toolbox. The Mathworks. May 2003 <http://www.mathworks.com/products/neuralnet/>

[3] HTK Hidden Markov Model Toolkit – Speech Recognition Research Toolkit. Cambridge University Engineering Department. December 19, 2002 <http://htk.eng.cam.ac.uk/>

[4] Young, S. J. "The HTK Hidden Markov Model Toolkit: Design and Philosophy." CUED/F-INFENG/TR.152 (September 6, 1994).

[5] Kopec, Gary E. "The Representation of Discrete-Time Signals and Systems in Programs." Massachusetts Institute of Technology Ph.D. Thesis, Cambridge, MA (1980).

[6] Kopec, Gary E. "The Integrated Signal Processing System ISP." IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-32, No. 4 (August 1984).

[7] Karjalainen, Matti. "DSP Software Integration by Object-Oriented Programming: A Case Study of QuickSig." IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. 7, pp 21-31 (April 1990).

[8] Winograd, Joseph M., and S. Hamid Nawab. "A C++ Software Environment for the Development of Embedded Signal Processing Systems." Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, pp 2715-2718 (May 1995).

[9] Automatic Speech Recognition: Software. Institute for Signal and Information Processing/Mississippi State University. January 2003 <http://www.isip.msstate.edu/projects/speech/software/>

[10] Alphonso, I. "ISIP Foundation Classes." ISIP Spring 1999 Seminar Series (January 29, 1999).

[11] Sutton, Cole, et al. "Universal Speech Tools: the CSLU Toolkit." Proceedings of the International Conference on Spoken Language Processing (ICSLP), Sydney, Australia (Nov 1998).

[12] Toolkit: about. Center for Spoken Language Understanding. December 4, 2002 <http://cslu.cse.ogi.edu/toolkit/>

[13] "Code Vectorization Guide." Mathworks Technical Note #1109, Revision 2.0 (October 15, 2002). <http://www.mathworks.com/support/tech-notes/1100/1109.shtml>

[14] Becchetti, Claudio and Lucio Prina Ricotti. "Automatic Speech Recognition Implementation with Object Oriented Programming: The 'RES' System." COST 249 Meeting, Rome (February 17-18, 1997).

[15] Becchetti, Claudio and Lucio Prina Ricotti. Speech Recognition: Theory and C++ Implementation. West Sussex: John Wiley & Sons Ltd, 1999.

[16] Rabiner, L. R., and B. H. Juang. Fundamentals of Speech Recognition. Prentice Hall, 1993.

[17] Jelinek, Frederick. Statistical Methods for Speech Recognition. Cambridge: Massachusetts Institute of Technology, 1997.

[18] Ham, Fredric M., and Ivica Kostanic. Principles of Neurocomputing for Science & Engieering. New York: McGraw-Hill, 2001.

[19] Sakoe, H. and S. Chiba. "Dynamic Programming Algorithm Optimiziation for Spoken Word Recognition." IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol ASSP-26, 43-49 (1978).

[20] The CMU Pronouncing Dictionary. Bob Weide, Carnegie Mellon University. March 2003 <http://www.speech.cs.cmu.edu/cgi-bin/cmudict>

[21] libsndfile. Erik de Castro Lopo. February 2, 2003 <http://www.zip.com.au/~erikd/libsndfile/>

[22] Hermansky, H., and N. Morgan. "RASTA Processing of Speech." IEEE Transactions on Speech and Audio Processing, Vol. 2, No. 4: 578-589.

[23] GNU Lesser Public License. Free Software Foundation. February 1999 <http://www.gnu.org/copyleft/lesser.html>

[24] Lyons, Richard G. Understanding Digital Signal Processing. Reading: Addison-Wesley, 1997.

[25] FFTW Home Page. Matteo Frigo and Steven G. Johnson. March 2003 <http://www.fftw.org/>

[26] Frigo, Matthew, and Steven G. Johnson. "FFTW: An Adaptive Software Architecture for the FFT." International Conference on Acoustics, Speech, and Signal Processing, Vol 3 (1998): 1381-1384.

[27] Kim, Woosung, Sanjeev Khudanpur, and Jun Wu. "Smoothing Issues in the Structured Language Model." Eurospeech 2001, Vol 1, pp. 717-720, September 2001.

[28] Member Functions (CPen::CPen()). Microsoft. March 2003 <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_mfc_CPen.3a3a.CPen.asp>

[29] Digalakis, V., L. Neumeyer, and M. Perakakis. "Quantization of Cepstral Parameters for Speech Recognition Over the World Wide Web." Proceedings on International Conference on Acoustics, Speech, and Signal Processing, pp 989-992, Vol. 2, Seattle, Washington.

[30] Doxygen. Dimitri van Heesch. February 20, 2003 <http://www.doxygen.org>

[31] HTML 4.01 Specification. World Wide Web Consortium. May 2003 <http://www.w3.org/TR/html4/>

[32] What is Adobe PDF? Adobe Systems Incorporated. May 2003 <http://www.adobe.com/products/acrobat/adobepdf.html>

[33] Microsoft HTML Help 1.4 SDK. Microsoft Corporation. May 2003 <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/htmlhelp/html/vsconHH1Start.asp>

Appendix A

Coding Conventions

Most AFARR classes implement the functions defined in Table A.1 and Table A.2.

Member Function Name	Description
Init	Initialization function. Often times, initialization information will not be available when the object is constructed, so a separate initialization function is implemented. A separate initialization function also makes it possible to release and reinitialize an object without releasing it from memory.
Clear	Deinitializes an object, releasing all allocated memory and resetting all member variables to the default values.
Copy	Copies the specified object into this object.
(Assignment operator) *(Copy constructor)*	It is important to implement a copy constructor so C++ can properly pass objects by value. Assignment operators are also convenient to have. Both typically call the Copy function.

Table A.1 – Common Member Function Names

Variable Prefix	Meaning
*g_*	Global variable
*m_*	Member variable
i	Integer
n	Integer representing how many of something there are. e.g. nDimensions represents the number of dimensions
f	Floating point (double)
p	Pointer. May be repeated to indicate the number of levels of indirection. e.g., pp represents a pointer to a pointer
sz	C string (NULL terminated)
c	Class
str	String class (CString or CStringSrl)

Table A.2 – Variable Naming Convention

Appendix B

Speech Corpus Statistics

B.1. Summary

Six male New England users contributed 3,550 phrases to the corpus. Most of the speakers were aged 21 to 27. Only one speaker had a noticeable accent (a Massachusetts accent). The corpus is included on the thesis CD-ROM.

B.2. Statistics by Phrase

Phrase	Entries
one	75
two	75
three	75
four	75
five	75
six	75
seven	75
eight	75
nine	75
zero	69
lock	18
A Adam	18
B Boston	18
C Charlie	18
V Victor	18
D David	15
E Edward	15
F Frank	15
G George	15
H Henry	15
I Ida	15
J John	15
K King	15
L Lincoln	15
M Mary	15

N Nora	15
O Ocean	15
P Paul	15
Q Queen	15
R Robert	15
S Sam	15
T Tom	15
U Union	15
W Walter	15
X X Ray	15
Y Young	15
Z Zebra	15
number zero	15
main screen	12
radar	12
stop	12
front antenna	12
records	12
plate type	12
license number	12
license state	12
gender	12
G P S	12
report	12
video	12
play	12
rewind	12
wider	12
focus	12
microphone	12
pursuit	12
male	12
female	12
Mississippi	12
Massachusetts	12
volume up	12
volume down	12
lock off	12
front antenna off	12
number one	9
number two	9
number three	9
number four	9

number five	9
number six	9
number seven	9
number eight	9
number nine	9
volume increase	6
volume decrease	6
10 20 New Hampshire	6
10 20 Florida	6
10 21 Florida	6
10 22 Florida	6
help	6
lights	6
wigwags	6
wigwags off	6
pierce	6
sirens off	6
radio	6
back	6
perform check	6
check records	6
repeat results	6
repeat	6
reset data	6
clear data	6
string input	6
letter codes	6
cancel	6
paste	6
first name	6
initial	6
middle initial	6
last name	6
date of birth	6
washington	6
emergency signals	6
test	6
radio control	6
channel up	6
channel down	6
zone up	6
zone down	6
operator license number	6

rear antenna	6
rear antenna off	6
stationary mode	6
same mode	6
opposite mode	6
range up	6
range down	6
squelch	6
faster	6
slower	6
video help	6
play on	6
record on	6
record	6
rewind off	6
forward	6
food	6
forward off	6
focus on	6
focus off	6
microphone on	6
microphone off	6
auto focus	6
closer	6
close	6
done	6
strobes	6
strobes off	6
front strobes	6
front strobes off	6
rear strobes	6
rear strobes off	6
left alley	6
left alley off	6
right alley	6
right alley off	6
wig wags	6
wig wags off	6
take downs	6
take downs off	6
rear floods	6
rear floods off	6
lights off	6

pursuit off	6
lights and siren	6
lights and siren off	6
wail	6
wail off	6
yelp	6
yelp off	6
piercer	6
piercer off	6
siren off	6
P A	6
radio mode	6
manual mode	6
hands free mode	6
troop a	6
wanted person	6
talk	6
l p channel 2	6
ports	6
dover	6
durham	6
london	6
keene	6
exeter	6
tilton	6
wakefield	6
hanover	6
rochester p d	6
serial number	6
clear	6
check license	6
sex	6
v i n	6
M V records	6
plate number	6
plate state	6
Alabama	6
Alaska	6
Arizona	6
Arkansas	6
California	6
Colorado	6
Connecticut	6

D C	6
Delaware	6
Florida	6
Georgia	6
Hawaii	6
Idaho	6
Illinois	6
Indiana	6
Iowa	6
Kansas	6
Kentucky	6
Louisiana	6
Maine	6
Maryland	6
Michigan	6
Minnesota	6
Missouri	6
Montana	6
Nebraska	6
Nevada	6
New Hampshire	6
New Jersey	6
New Mexico	6
New York	6
North Carolina	6
North Dakota	6
Ohio	6
Oklahoma	6
Oregon	6
Pennsylvania	6
Rhode Island	6
South Carolina	6
South Dakota	6
Tennessee	6
Texas	6
Utah	6
Vermont	6
Virginia	6
West Virginia	6
Wisconsin	6
Wyoming	6
confirmed	6
yes	6

correct	6
O K	6
no	6
wrong	6
error	6
applications	6
speech input	6
radio controls	6
alert	6
city	6
high	6
slower off	6
radar off	6
unlock	6
distance up	6
distance down	6
squelch off	6
s q l off	6
reset	6
G P S help	6
A V L	6
home	6
radio alert	6
lane closed	6
accident	6
accident ahead	6
emergency	6
emergency vehicle	6
stop transmitting	6
test off	6
type	6
type code	6
manufacturer	6
caliber	6
article query	6
gun query	6
vehicle query	6
S Sam E Edward I Ida Zero Eight Six	5
D David W Walter J John Z Zebra L Lincoln G George	5
Five One One Nine Four Six	5
B Boston L Lincoln A Adam Three Seven Six	5
E Edward W Walter M Mary I Ida F Frank F Frank	5
I Ida U Union E Edward O Ocean P Paul Seven	5

Zero Eight Zero Eight Eight Zero One	5
Three Two One Two Zero Three	5
V Victor O Ocean Six Five Nine One	5
Two Three Three Five Five Five Eight	5
Zero Nine Five Six Zero Five	5
A Adam L Lincoln I Ida Zero Nine Three Nine	5
Four N Nora Eight F Frank J John X X Ray G George	5
E Edward E Edward Two Two Two Six	5
Four Zero Six Zero Zero Four	5
Six Three Six Six One Nine Two	5
Y Young Q Queen H Henry E Edward N Nora I Ida A Adam	5
Seven Six Five Five Three Seven Zero	5
N Nora F Frank Q Queen Seven Seven P Paul	5
Z Zebra M Mary O Ocean S Sam N Nora D David S Sam	5
Q Queen F Frank W Walter G George Z Zebra Eight Three	5
B Boston Seven Six Zero Zero One One	5
C Charlie D David L Lincoln B Boston A Adam P Paul S Sam	5
Three Nine Nine One Zero Six Two	5
Seven S Sam M Mary D David X X Ray L Lincoln	5
K King V Victor B Boston T Tom D David R Robert	5
Zero Three Three Two Zero Zero Zero	5
C Charlie J John S Sam U Union V Victor D David	5
Zero P Paul H Henry J John K King I Ida L Lincoln	5
W Walter Four Nine Seven Nine Three Three	5
Nine Two Four Eight Three A Adam A Adam	5
Two Zero Two Three One Zero	5
Four Three Five Six Three Zero V Victor	5
Q Queen X X Ray E Edward I Ida M Mary J John F Frank	5
T Tom N Nora Y Young G George U Union V Victor	5
M Mary B Boston Z Zebra L Lincoln A Adam S Sam	5
Y Young M Mary S Sam D David G George V Victor V Victor	5
W Walter H Henry E Edward U Union Five Six	5
Two Nine X X Ray Three Q Queen S Sam D David	5
Three Seven Three Four Four S Sam	5
G George D David T Tom U Union T Tom X X Ray J John	5
Nine Four Two One Eight Five	5
Three Zero Seven Four Six Four	5
Z Zebra U Union W Walter B Boston D David Y Young	5
E Edward U Union Y Young Y Young G George B Boston L Lincoln	5
L Lincoln O Ocean Q Queen S Sam O Ocean X X Ray	5
Six Zero One Six Six One	5
T Tom Z Zebra Y Young Three Five Three	5
Zero Four Three Three Three Two P Paul	5

Five Seven Five Five Four Four	5
One Six Two E Edward T Tom H Henry D David	5
Z Zebra K King J John P Paul H Henry I Ida C Charlie	5
I Ida P Paul Z Zebra D David T Tom N Nora L Lincoln	5
Three Three Nine Four A Adam One S Sam	5
P Paul N Nora Four Zero Two Four One	5
P Paul G George V Victor X X Ray P Paul X X Ray	5
One One Six Three Two Four Four	5
C Charlie A Adam Six Six Six Four Seven	5
Five Nine Z Zebra L Lincoln R Robert Q Queen	5
Seven Two I Ida J John O Ocean C Charlie	5
Zero Six Four Five Zero S Sam	5
Three Two Eight Zero Two Zero Two	5
C Charlie Nine Zero T Tom E Edward L Lincoln	5
G George Y Young F Frank O Ocean Q Queen Zero	5
Z Zebra K King L Lincoln R Robert A Adam A Adam	5
C Charlie Z Zebra U Union C Charlie Z Zebra M Mary	5
D David Seven Six Three Seven Seven	5
Nine N Nora I Ida S Sam E Edward D David	5
B Boston E Edward W Walter C Charlie E Edward B Boston	5
B Boston F Frank P Paul M Mary Z Zebra Two One	5
Two Nine Four One Eight One Eight	5
H Henry D David X X Ray X X Ray V Victor D David	5
K King P Paul Six Four Two Zero I Ida	5
Three One Five Three Eight One Eight	5
M Mary Q Queen Nine Two Nine Three Seven	5
Four Four Two O Ocean Z Zebra I Ida	5
Two Nine Five Nine Zero Six Seven	5
K King D David Six Zero Eight Five	5
Six Eight Four Zero Five One	5
Zero Four Seven Four Five One	5
Q Queen K King N Nora U Union W Walter Zero Three	5
Four Five Four O Ocean H Henry W Walter D David	5
Four Eight Five Six Three Seven	5
N Nora S Sam N Nora A Adam A Adam C Charlie U Union	5
Eight Five Five H Henry A Adam B Boston	5
Zero Four Four Y Young Q Queen G George One	5
F Frank F Frank P Paul Z Zebra P Paul Y Young	5
Two Eight Five Seven Eight Two	5
Q Queen Six C Charlie E Edward P Paul O Ocean J John	5
Five Five Eight Three Three Three Three	5
I Ida Zero Zero Four Two Four One	5
One Two N Nora D David X X Ray B Boston V Victor	5

B Boston L Lincoln Eight Nine Six Five Two	5
G George I Ida B Boston F Frank Q Queen O Ocean	5
K King K King D David C Charlie C Charlie R Robert	5
Zero Eight Three Zero A Adam C Charlie	5
J John H Henry L Lincoln K King D David N Nora	5
Q Queen S Sam T Tom D David X X Ray U Union B Boston	5
X X Ray H Henry Y Young R Robert U Union B Boston	5
Nine One Nine Zero Four R Robert	5
Nine Seven Zero Five Four Seven	5
B Boston X X Ray J John I Ida R Robert O Ocean Three	5
One Eight Zero Five Six Six Nine	5
B Boston W Walter Two Eight Three Nine	5
Z Zebra L Lincoln Zero S Sam F Frank R Robert	5
Eight Zero Four Eight Five Zero	5
Six Six Four Zero Three Eight One	5
O Ocean Z Zebra V Victor G George Z Zebra P Paul	5
Three Nine Six One I Ida X X Ray Zero	5
M Mary U Union C Charlie R Robert G George Four Nine	5
T Tom W Walter R Robert S Sam F Frank X X Ray B Boston	5
Three One Nine One Eight Seven One	5
Z Zebra H Henry C Charlie G George X X Ray F Frank	5
B Boston Q Queen X X Ray E Edward V Victor P Paul	5
X X Ray G George N Nora E Edward Seven Nine Seven	5
Nine Seven Two One O Ocean S Sam Y Young	5
W Walter U Union Six Two Nine Eight Three	5
U Union S Sam X X Ray W Walter Q Queen T Tom	5
B Boston Two Eight Nine Eight One Nine	5
D David Z Zebra S Sam Six One Nine Eight	5
Five Nine Five Nine Five Zero	5
R Robert O Ocean F Frank R Robert R Robert B Boston K King	5
I Ida Seven Zero Two Y Young B Boston A Adam	5
Four Eight Three Nine Zero Zero One	5
T Tom M Mary E Edward I Ida X X Ray F Frank	5
V Victor F Frank L Lincoln Q Queen T Tom L Lincoln	5
Nine Zero Four Two Nine L Lincoln	5
L Lincoln Two Seven Four Eight Six	5
Nine Six Eight Four Five Two	5
Two Four One One Two Two Five	5
Six One Zero Eight Five Two Two	5
E Edward L Lincoln K King H Henry J John Z Zebra P Paul	5
Nine K King Y Young N Nora B Boston O Ocean	5
F Frank Z Zebra C Charlie A Adam G George R Robert F Frank	5
H Henry F Frank Seven Four Eight Zero	5

H Henry B Boston W Walter W Walter Q Queen G George Q Queen	5
Seven Two Zero Six Nine Nine	5
Two Two Seven One Nine Nine	5
Six Zero Zero Nine Six Eight Eight	5
Zero Three Eight Seven Nine T Tom X X Ray	5
T Tom K King A Adam V Victor A Adam G George	5
I Ida Q Queen Z Zebra I Ida H Henry N Nora	5
Seven One Eight One Zero Seven Five	5
Five Five Six Eight Four Three	5
U Union Q Queen V Victor N Nora One J John G George	5
Zero Six Four Six One Two Two	5
Two One Six Five Zero Eight Five	5
Five P Paul V Victor B Boston K King C Charlie B Boston	5
Y Young M Mary U Union E Edward Z Zebra F Frank O Ocean	5
Two Four Zero One Zero N Nora D David	5
alternate direction	4

Table B.1 – Statistics by Phrase

Appendix C

HMM SR Engine Results

Results gathered on Fri May 30 17:55:44 2003.

Pass	Accuracy	Number Correct	Number of Tests	Avg. Training Time	Avg. Testing Time
0	99	198	200	00:00:00.000	00:00:03.230
1	99	198	200	00:00:06.254	00:00:03.232
2	99	198	200	00:00:06.326	00:00:03.232
3	99	198	200	00:00:06.353	00:00:03.239
4	99	198	200	00:00:06.357	00:00:03.233
5	99	198	200	00:00:06.350	00:00:03.229
6	99	198	200	00:00:06.905	00:00:03.250
7	99	198	200	00:00:06.362	00:00:03.230
8	99	198	200	00:00:06.349	00:00:03.229
9	99	198	200	00:00:06.350	00:00:03.227
10	99	198	200	00:00:06.353	00:00:03.229
11	99	198	200	00:00:06.353	00:00:03.229
12	99	198	200	00:00:06.358	00:00:03.232
13	99	198	200	00:00:06.393	00:00:03.229
14	99	198	200	00:00:06.389	00:00:03.229
15	99	198	200	00:00:06.396	00:00:03.229
16	99	198	200	00:00:06.396	00:00:03.229
17	99	198	200	00:00:06.396	00:00:03.230
18	99	198	200	00:00:06.395	00:00:03.229
19	99	198	200	00:00:06.396	00:00:03.230
20	99	198	200	00:00:06.397	00:00:03.234
21	99	198	200	00:00:06.394	00:00:03.230
22	99	198	200	00:00:06.392	00:00:03.230
23	99	198	200	00:00:06.391	00:00:03.229
24	99	198	200	00:00:06.396	00:00:03.229
25	99	198	200	00:00:06.396	00:00:03.229
26	99	198	200	00:00:06.402	00:00:03.235
27	99	198	200	00:00:06.410	00:00:03.232
28	99	198	200	00:00:06.405	00:00:03.235
29	99	198	200	00:00:06.405	00:00:03.232
30	99	198	200	00:00:06.404	00:00:03.234
31	99	198	200	00:00:06.902	00:00:03.289
32	99	198	200	00:00:06.410	00:00:03.231
33	99	198	200	00:00:06.410	00:00:03.233

34	99	198	200	00:00:06.424	00:00:03.241
35	99	198	200	00:00:06.426	00:00:03.238
36	99	198	200	00:00:06.429	00:00:03.238
37	99	198	200	00:00:06.425	00:00:03.239
38	99	198	200	00:00:06.431	00:00:03.240
39	99	198	200	00:00:06.440	00:00:03.238
40	99	198	200	00:00:06.444	00:00:03.240
41	99	198	200	00:00:06.432	00:00:03.241
42	99	198	200	00:00:06.441	00:00:03.239
43	99	198	200	00:00:06.439	00:00:03.238
44	99	198	200	00:00:06.440	00:00:03.238
45	99	198	200	00:00:06.436	00:00:03.239
46	99	198	200	00:00:06.437	00:00:03.240
47	99	198	200	00:00:06.442	00:00:03.238
48	99	198	200	00:00:06.442	00:00:03.240
49	99	198	200	00:00:06.444	00:00:03.242
50	99	198	200	00:00:06.445	00:00:03.238
51	99	198	200	00:00:06.445	00:00:03.241
52	99	198	200	00:00:06.444	00:00:03.240
53	99	198	200	00:00:06.441	00:00:03.240
54	99	198	200	00:00:06.441	00:00:03.240
55	99	198	200	00:00:06.446	00:00:03.241
56	99	198	200	00:00:06.445	00:00:03.240
57	99	198	200	00:00:06.444	00:00:03.240
58	99	198	200	00:00:06.452	00:00:03.241
59	99	198	200	00:00:06.448	00:00:03.241
60	99	198	200	00:00:06.455	00:00:03.240
61	99	198	200	00:00:06.450	00:00:03.242
62	99	198	200	00:00:06.450	00:00:03.240
63	99	198	200	00:00:06.453	00:00:03.241
64	99	198	200	00:00:06.449	00:00:03.244
65	99	198	200	00:00:06.458	00:00:03.244
66	99	198	200	00:00:06.468	00:00:03.229
67	99	198	200	00:00:06.459	00:00:03.732
68	99	198	200	00:00:06.631	00:00:03.936
69	99	198	200	00:00:07.190	00:00:04.537
70	99	198	200	00:00:06.690	00:00:03.285
71	99	198	200	00:00:06.554	00:00:03.305
72	99	198	200	00:00:07.293	00:00:03.823
73	99	198	200	00:00:07.376	00:00:03.576
74	99	198	200	00:00:07.139	00:00:03.653
75	99	198	200	00:00:06.858	00:00:03.173
76	99	198	200	00:00:06.645	00:00:03.154

77	99	198	200	00:00:06.639	00:00:03.159
78	99	198	200	00:00:06.636	00:00:03.157
79	99	198	200	00:00:06.643	00:00:03.168
80	99	198	200	00:00:07.169	00:00:03.174
81	99	198	200	00:00:06.645	00:00:03.157
82	99	198	200	00:00:06.642	00:00:03.155
83	99	198	200	00:00:06.642	00:00:03.156
84	99	198	200	00:00:06.640	00:00:03.156
85	99	198	200	00:00:06.643	00:00:03.156
86	99	198	200	00:00:06.643	00:00:03.164
87	99	198	200	00:00:06.648	00:00:03.156
88	99	198	200	00:00:06.641	00:00:03.157
89	99	198	200	00:00:06.646	00:00:03.156
90	99	198	200	00:00:06.643	00:00:03.157
91	99	198	200	00:00:06.641	00:00:03.155
92	99	198	200	00:00:06.969	00:00:03.487
93	99	198	200	00:00:07.211	00:00:03.642
94	99	198	200	00:00:10.905	00:00:05.755
95	99	198	200	00:00:12.104	00:00:03.821
96	98.5	197	200	00:00:07.192	00:00:03.595
97	98.5	197	200	00:00:07.004	00:00:03.493
98	98	196	200	00:00:06.898	00:00:03.462
99	98	196	200	00:00:06.905	00:00:03.462

Table C.1 – HMM SR Engine Results

Appendix D

Neural Network SR Engine Results Table

Results gathered on Thu May 08 15:50:00 2003.

Pass	Accuracy	Number Correct	Number of Tests	Avg. Training Time	Avg. Testing Time
0	10	20	200	00:00:00.000	00:00:02.445
1	10	20	200	00:00:02.278	00:00:02.466
2	10	20	200	00:00:00.057	00:00:02.493
3	10	20	200	00:00:00.057	00:00:02.480
4	10	20	200	00:00:00.057	00:00:02.460
5	10	20	200	00:00:00.057	00:00:02.470
6	10	20	200	00:00:00.057	00:00:02.461
7	10	20	200	00:00:00.057	00:00:02.453
8	10	20	200	00:00:00.057	00:00:02.448
9	10	20	200	00:00:00.057	00:00:02.444
10	10	20	200	00:00:00.057	00:00:02.436
11	10	20	200	00:00:00.057	00:00:02.437
12	10	20	200	00:00:00.057	00:00:02.436
13	10	20	200	00:00:00.057	00:00:02.435
14	10	20	200	00:00:00.057	00:00:02.434
15	10	20	200	00:00:00.057	00:00:02.435
16	10	20	200	00:00:00.057	00:00:02.435
17	10	20	200	00:00:00.057	00:00:02.461
18	10	20	200	00:00:00.058	00:00:02.439
19	12	24	200	00:00:00.057	00:00:02.439
20	10	20	200	00:00:00.057	00:00:02.434
21	10	20	200	00:00:00.057	00:00:02.437
22	10	20	200	00:00:00.057	00:00:02.437
23	10	20	200	00:00:00.057	00:00:02.436
24	10	20	200	00:00:00.057	00:00:02.436
25	10	20	200	00:00:00.057	00:00:02.431
26	20	40	200	00:00:00.057	00:00:02.431
27	10	20	200	00:00:00.057	00:00:02.431
28	10	20	200	00:00:00.057	00:00:02.432
29	10	20	200	00:00:00.057	00:00:02.431
30	10	20	200	00:00:00.058	00:00:02.431
31	31.5	63	200	00:00:00.057	00:00:02.431
32	19.5	39	200	00:00:00.057	00:00:02.430
33	20	40	200	00:00:00.057	00:00:02.430

34	10	20	200	00:00:00.057	00:00:02.430
35	19.5	39	200	00:00:00.057	00:00:02.429
36	10	20	200	00:00:00.057	00:00:02.430
37	10	20	200	00:00:00.057	00:00:02.426
38	10	20	200	00:00:00.057	00:00:02.427
39	24.5	49	200	00:00:00.057	00:00:02.426
40	29	58	200	00:00:00.057	00:00:02.427
41	29.5	59	200	00:00:00.057	00:00:02.428
42	29.5	59	200	00:00:00.057	00:00:02.427
43	20	40	200	00:00:00.057	00:00:02.426
44	22.5	45	200	00:00:00.057	00:00:02.427
45	28	56	200	00:00:00.057	00:00:02.428
46	19.5	39	200	00:00:00.057	00:00:02.430
47	20	40	200	00:00:00.057	00:00:02.907
48	20	40	200	00:00:00.057	00:00:02.679
49	20	40	200	00:00:00.058	00:00:02.592
50	34	68	200	00:00:00.058	00:00:02.478
51	40	80	200	00:00:00.058	00:00:02.483
52	50	100	200	00:00:00.065	00:00:02.536
53	49	98	200	00:00:00.059	00:00:02.517
54	59.5	119	200	00:00:00.057	00:00:02.432
55	49.5	99	200	00:00:00.058	00:00:02.432
56	49.5	99	200	00:00:00.057	00:00:02.432
57	51.5	103	200	00:00:00.057	00:00:02.431
58	59.5	119	200	00:00:00.057	00:00:02.471
59	59.5	119	200	00:00:00.058	00:00:02.432
60	60	120	200	00:00:00.057	00:00:02.428
61	78.5	157	200	00:00:00.057	00:00:02.430
62	73.5	147	200	00:00:00.057	00:00:02.427
63	86.5	173	200	00:00:00.057	00:00:02.428
64	80.5	161	200	00:00:00.057	00:00:02.429
65	94	188	200	00:00:00.057	00:00:02.428
66	84	168	200	00:00:00.057	00:00:02.429
67	84.5	169	200	00:00:00.058	00:00:02.428
68	94.5	189	200	00:00:00.057	00:00:02.427
69	86	172	200	00:00:00.057	00:00:02.428
70	94.5	189	200	00:00:00.057	00:00:02.428
71	84.5	169	200	00:00:00.057	00:00:02.428
72	86.5	173	200	00:00:00.057	00:00:02.428
73	94.5	189	200	00:00:00.057	00:00:02.428
74	84.5	169	200	00:00:00.057	00:00:02.428
75	86.5	173	200	00:00:00.057	00:00:02.428
76	94.5	189	200	00:00:00.057	00:00:02.428

77	85	170	200	00:00:00.057	00:00:02.429
78	88.5	177	200	00:00:00.057	00:00:02.428
79	95	190	200	00:00:00.057	00:00:02.426
80	85	170	200	00:00:00.057	00:00:02.428
81	94.5	189	200	00:00:00.057	00:00:02.426
82	94.5	189	200	00:00:00.057	00:00:02.427
83	95	190	200	00:00:00.057	00:00:02.426
84	94.5	189	200	00:00:00.057	00:00:02.427
85	95	190	200	00:00:00.057	00:00:02.424
86	92	184	200	00:00:00.057	00:00:02.427
87	95	190	200	00:00:00.057	00:00:02.428
88	94	188	200	00:00:00.057	00:00:02.427
89	95	190	200	00:00:00.057	00:00:02.428
90	93	186	200	00:00:00.057	00:00:02.426
91	95	190	200	00:00:00.057	00:00:02.428
92	94	188	200	00:00:00.057	00:00:02.427
93	95	190	200	00:00:00.057	00:00:02.428
94	93	186	200	00:00:00.057	00:00:02.428
95	95	190	200	00:00:00.057	00:00:02.427
96	93	186	200	00:00:00.057	00:00:02.426
97	95	190	200	00:00:00.057	00:00:02.427
98	93	186	200	00:00:00.057	00:00:02.426
99	95	190	200	00:00:00.057	00:00:02.426
100	92.5	185	200	00:00:00.057	00:00:02.427
101	95	190	200	00:00:00.057	00:00:02.425
102	95	190	200	00:00:00.057	00:00:02.426
103	95	190	200	00:00:00.057	00:00:02.428
104	95	190	200	00:00:00.057	00:00:02.427
105	95	190	200	00:00:00.057	00:00:02.428
106	95	190	200	00:00:00.057	00:00:02.427
107	95	190	200	00:00:00.057	00:00:02.426
108	95	190	200	00:00:00.057	00:00:02.427
109	95	190	200	00:00:00.057	00:00:02.428
110	95	190	200	00:00:00.057	00:00:02.428
111	95	190	200	00:00:00.057	00:00:02.427
112	95	190	200	00:00:00.057	00:00:02.426
113	95	190	200	00:00:00.057	00:00:02.428
114	95	190	200	00:00:00.057	00:00:02.449
115	95	190	200	00:00:00.057	00:00:02.429
116	95	190	200	00:00:00.057	00:00:02.427
117	95	190	200	00:00:00.057	00:00:02.429
118	95	190	200	00:00:00.057	00:00:02.429

119	95	190	200	00:00:00.057	00:00:02.429

Table D.1 – Neural Network SR Engine Results

It should be noted that the average testing time was only 0.06 seconds for all iterations, except for iteration 1 since training date is cached after the first training pass and used to accelerate training. No testing data was cached, except for the position of the first and last speech sample.