

A speech recognition system for Swedish running on Android

Simon Lindholm

May 31, 2010

Abstract

A speech recognition system is one that automatically transcribes speech into text. It can be divided into two separate stages: training the system with a corpus of pre transcribed speech and using the trained system to recognize spoken words and sentences.

In this master thesis is presented a project to implement a working small-vocabulary speech recognition system for Swedish. The acoustic units and statistical models used are hidden Markov models and mel-frequency cepstral coefficients. The underlying mathematical theory is explored along with practical considerations and test results.

The training and recognition stages are implemented in C++ and Perl for use on an ordinary desktop computer. As a proof of concept, the recognition part is ported to Java for use on Android mobile phones.

Contents

1	Introduction	5
1.1	Implementation	6
1.2	The Waxholm Corpus	9
2	Signal Processing	10
2.1	Windowing	12
2.2	Mel-Frequency Cepstral Coefficients	15
2.3	Feature Vector Deltas	17
3	Definitions	18
3.1	Hidden Markov Models	18
3.2	Observation Probability Density Functions	20
3.2.1	Gaussian Probability Density Function	21
3.2.2	Multi-variate Gaussians	21
3.2.3	Multi-variate Gaussian mixtures	22
3.3	Generating Observations	23
3.4	Forward Probability	24
3.5	Backward Probability	27
3.6	Viterbi Algorithm	30
3.7	Kullback-Leibler Distance	32
4	Training	33
4.1	Phoneme Modelling	33
4.2	Clustering	34
4.2.1	K-Means Clustering Algorithm	35
4.2.2	Bees Clustering Algorithm	37
4.2.3	HMM Parameter Estimation	39
4.3	Baum-Welch Re-estimation	42
4.3.1	γ and ξ	42
4.3.2	Estimating Transition Probabilities	44
4.3.3	Estimating Observation Probability Functions	45
4.3.4	Scaling α and β	47
5	Word Recognition	52
5.1	Building The Word Graph	52
5.2	Viterbi Beam-Search	58
6	Results And Conclusions	60
6.1	Word Recognition	60
6.2	Android Port	62
6.3	Summary	62
6.4	Evaluation	63
6.5	Suggestions For Improvements	63

A	Proofs	65
A.1	Expressing γ in terms of ξ	65
A.2	Proof that $(\bar{A} \bar{\omega})$ is row stochastic	65
A.3	Proof that $\hat{\gamma} = \gamma$	67
A.4	Proof that $\hat{\xi} = \xi$	68
B	Sample configuration file	69
C	Dictionary file	71

1 Introduction

There are a number of different approaches that may be taken towards implementing a working speech recognition system. A rough outline of the approach chosen for this project is the following: Given a sizable amount of training data, construct statistical models for the individual phonemes. Connect the models of the phonemes into larger structures, representing whole words and then connect these into an even larger structure representing the whole dictionary of words the system should be able to recognize.

A data set of spoken sentences with annotations such as “*in recording x between 250 ms and 378 ms, the phoneme ‘A’ is being pronounced, between 378 ms and 407 ms, the phoneme ‘B’ is being pronounced, etc.*” is in the context of speech recognition called a *corpus*¹. The corpus in this project comes from the Waxholm project at KTH (Bertenstam et al., 1995) which contain about three and a half hours of recorded data in approximately 3900 separate audio files. About 60% of the files, or two hours and fifteen minutes of the total recordings, are annotated with phoneme boundaries. Except for a small part in English, all sentences in the Waxholm corpus are in Swedish.

For modern speech recognition systems, the most common choice of statistical model is currently *hidden Markov models*. Hidden Markov models will be explained at some detail later on, but suffice for now to say that they are used in this project to model individual phonemes. Since hidden Markov models are a type of directed graphs, the models of the phonemes conveniently lend themselves to being connected into the larger word and dictionary structures.

Sound travels through the air as vibrations, or more technically, waves of air pressure. The input from a microphone is the amplitude of the air pressure sampled at precise moments in time, as illustrated in Figure 1. The raw sound waves contain lots of information that is superfluous for recognizing speech. For instance, depending on age and various other factors, the human ear can perceive sound frequencies between roughly 200 and 20000 Hz, whilst human vocal tract rarely produces frequency components over 4000 Hz that carry significant information for decoding speech. Thus, we want to transform the raw audio signal in such a manner as to the maximum extent possible preserve relevant and disregard irrelevant information.

Many such transformation methods is based upon using a Fourier transform to convert the sound waves into a power spectrum. The raw input signal can be seen as a two-dimensional signal with the time at the x -axis and the amplitude at the y -axis. A power spectrum is essentially the same kind of signal, but with the signal broken up into the individual frequency components that makes up the full signal. If the raw input signal is a two-dimensional graph with time at the x -axis and amplitude at the y -axis, the

¹The word *corpus* is Latin, meaning *body*.

power spectrum will be a 3-dimensional graph, also with time at the x -axis, but amplitude per frequency at the z - and y -axes.

One popular signal processing method based on Fourier transforms, and the one that will be used in this project, is mel-frequency cepstral coefficients (MFCC). MFCC coefficients are derived from the power spectrum of the input signal, sampled at specific frequencies.

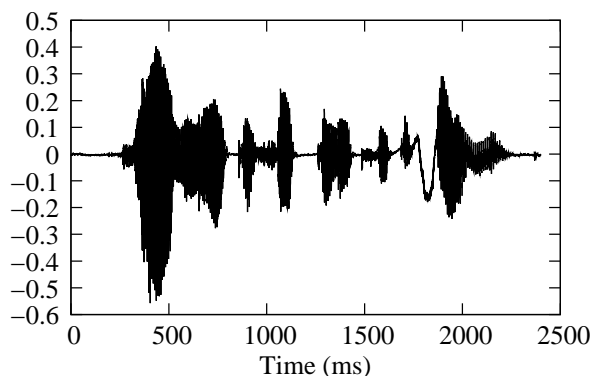


Figure 1: X/Y plot of the recorded sentence “*Jag vill åka 17.45*”

1.1 Implementation

The name chosen for the system implemented in this master thesis is *ERIS*, a backronym for Eclectic Recognizer Of Speech². It consists of several semi-independent programs which do one particular thing each, and are implemented in C++ and Perl. All programs have the prefix “*eris_*” and the subset of the programs which are implemented in Perl have the suffix “.pl”. In total, the system weighs in around about 21000 lines C++ and 4500 lines Perl code. The target platform is Linux, with some parts ported to Android³. Although portability has not been an explicit goal, the system does not contain much much platform dependent code. Porting it other platforms, should one wish to do so, shouldn’t be too much trouble.

Figure 2 provides an overview of the different programs and their relations to each other. The rectangular shaped node in the figure represent files or sets of files. The ellipses represent programs. The programs take files as input and produce other files as output. All of the programs also share a central configuration file, which is omitted from the figure for sake of brevity. The configuration file specifies most parameters which aren’t

²The name of the matron deity of confusion and discord seems more appropriate for a prototype system than Eros. It even provides an automatic memetic patch for the slightly distorted acronym.

³Which, strictly speaking, is a Linux variant as well.

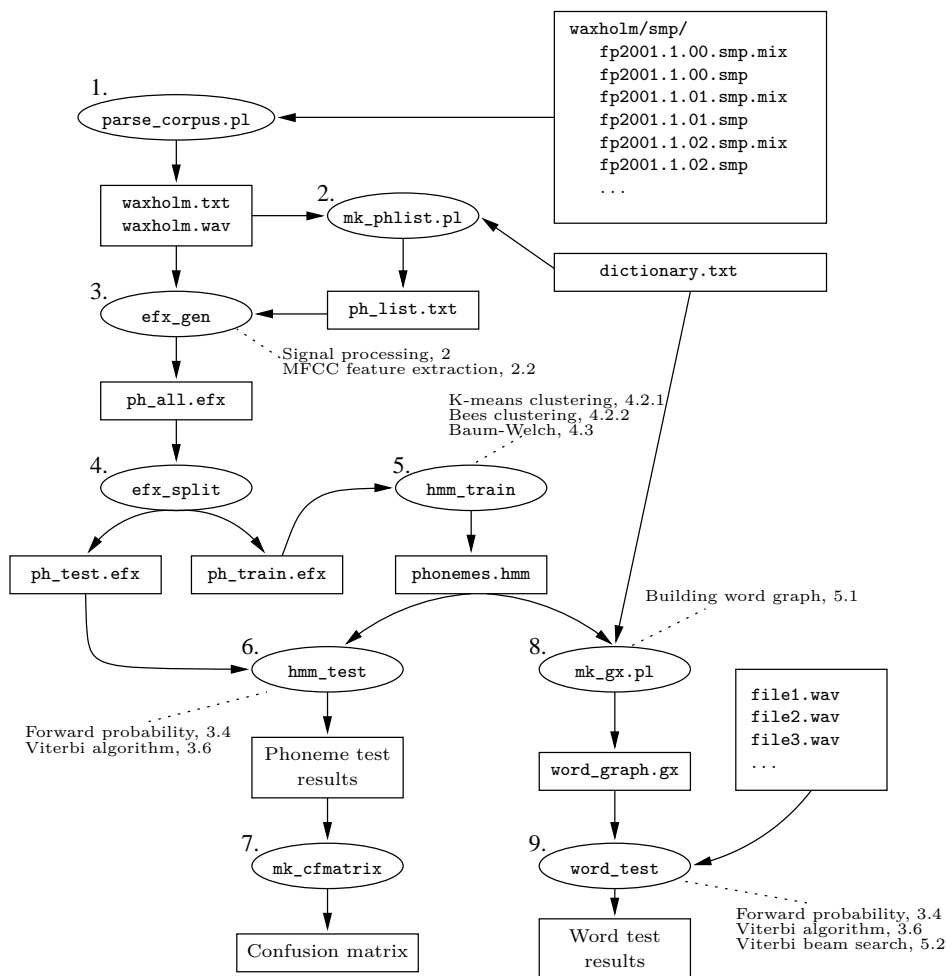


Figure 2: Implementation overview of the ERIS speech recognition system.

system paths for input or output files. Appendix B lists a sample configuration file. The idea of having a centralized configuration file is to ease scripting and running separate tests with different configuration parameters.

The chain of operations is as follows:

1. The Waxholm corpus is provided in a rather cumbersome format. It is parsed by the program `eris_parse_corpus.pl`, which writes a wave file containing all of the audio recordings (about 3 hours and 36 minutes long, and roughly 400 MB big) and an index file with the annotations of which time intervals correspond to which phonemes, words and sentences.

2. Running the whole chain of operations from start to finish is a fairly time consuming process. Exactly how long it will take varies a lot depending on which configuration settings are used, but in the order of about an hour or so on a 2.5 GHz dual-core desktop computer is typical. In order to save time, the program `eris_mk_phlist.pl` parses the dictionary file and constructs a list of which phonemes are used, so that no unnecessary phonemes are trained. The phonemes used have left and right contexts, so called triphones. With 46 phonemes and the left and right context belonging to one of nine different classes, there are a total of $9 \times 46 \times 9 = 3726$ triphones, but most don't occur in practice. In the Waxholm corpus there are about 1500 triphones, of which only about 20% will typically be used for a dictionary of about 100 words. The two most time consuming operations are training the HMM triphone models and constructing the phoneme confusion matrix. Both depend heavily on the number of triphones, so skipping unused ones is a big time saver.
3. `eris_efx_gen` reads the phoneme list, and the wave and text files produced by `eris_parse_corpus.pl` in step 1. For every phoneme, it extracts all recordings from the wave file and does signal processing in order to convert them into sequences of *feature vectors*, which are appropriate for use with the HMMs. The signal processing is detailed in Section 2. The files containing feature vectors have the suffix `.efx`.
4. `eris_efx_split` splits the `efx` file created in step 3 into a training file and a testing file. 70% of the feature vectors are written to the training file and the remaining 30% are written to the testing file.
5. Using the `efx` file containing the training sequences, `eris_hmm_train` trains a three-state hidden Markov model for each phoneme. It does so by creating crude initial approximations using the clustering algorithms described in Section 4.2, and then refining these using the Baum-Welch re-estimation algorithm, described in Section 4.3.
6. For every sequence of feature vectors in the `efx` file containing the testing sequences, `eris_hmm_test` goes through every phoneme HMM created in step 5 and calculates score of how well it matched⁴. The scores are written to a binary file.
7. The binary file with the test scores from step 6 is converted into a confusion matrix. That is, a matrix wherein each row corresponds to a set of testing sequences for a phoneme and each column specifies how many times each phoneme was the one with the highest score. The higher the values on the diagonal, relative to the non-diagonal elements

⁴Using the *forward algorithm*, described in Section 3.4.

on the same row, the better the phoneme models matched the correct testing sequences. For a hypothetical perfect match, the confusion matrix would be positive on the diagonal and zero everywhere else. An example confusion matrix is illustrated in Figure 3.

		Phoneme HMMs					
		A:	A	E:	E	M	N
Test sequences	A:	5	0	2	2	0	1
	A	1	3	3	0	2	1
	E:	1	0	7	0	1	1
	E	1	2	3	2	0	2
	M	1	3	0	2	4	1
N	0	0	1	0	0	9	

Figure 3: Example of confusion matrix with 5 phonemes.

- The `eris_mk_gx.pl` script combines the phoneme HMMs into a larger graph, consisting of the words in the dictionary. This process is detailed in Section 5.1.

A hidden Markov model, such as one representing a phoneme, is a type of graph and the *word graph*, in turn, is in every sense a hidden Markov model itself. The HMMs representing phonemes are dense and have few states, while the word graph is sparse and have many states, so the distinction in terminology arises from the difference in implementation. The phoneme HMMs are implemented using matrices, while the word graph is implemented using explicit node and edge objects.

- The program `eris_word_test` reads the file containing the word graph and an arbitrary number of wave files (specified as a command line parameter.) For each wave file, it runs it through the word graph and prints the a list in descending order of how well each word matches it. See Sections 3.4, 3.6 and 5.2.

1.2 The Waxholm Corpus

The Waxholm corpus, from the *Waxholm dialog project* at KTH (Bertenstam et al., 1995), is a collection of audio recordings of spoken sentences. Most sentences are in Swedish, but there are a few in English as well. Each recording comes with an associated file containing annotations which specifies which time intervals correspond to which sentences, words and phonemes.

The audio recordings are a collection of `.smp` files, which are essentially just raw PCM-data with a 1024 byte header. The header specifies parameters such as sample rate, number of channels, etc. All of the recordings in the corpus have only one channel and are sampled at a rate of 16 kHz. The

samples are encoded as 16 bit signed integers with the most significant byte first.

An annotation file has the same filename as its corresponding audio file, but with the suffix `.smp.mix` instead of `.smp`. The format of the `.smp.mix` files is quite irregular, likely due to being mostly manually edited. In this project, the annotation files from the Waxholm corpus are parsed by the Perl script `eris_parse_corpus.pl` in order to produce a new annotation file in a format which is more easily parsed by the other programs.

The `.smp.mix` files contains the textual representation of the spoken sentence, as well as listings of the words and phonemes in the sentence, and at which time intervals they occur. Only about half of the annotation files contain annotations down to the phoneme level.

As an example, consider Figure 1. It shows a plot of the audio clip `fp2038.11.04.smp` from the Waxholm corpus. The `smp` file has the accompanying annotation file `fp2038.11.04.smp.mix`, which is summarized in Table 1.2. The annotation files also contain various decorations of the phonemes, which indicate things such as which phonemes are stressed in the pronunciation of the word, as well as pseudo-words; such as breaths, clicks, smackings and so on. Such information has not been utilized in this project. All in all, the corpus contains about 3900 `.smp` files, totaling about three and a half hours of recorded material.

The phonemes used are *triphones*, that is, phonemes with a left and right context. The phoneme M preceded by the phoneme A and followed by O is thus a separate triphone from the M preceded by A and followed by E. Since some phonemes affect the preceding or following ones in a similar ways, the left and right contexts aren't individual phonemes, but rather classes of phonemes with similar effects. The classes used are the *type of phoneme – ToP* – from Ursin (2002).

The phonemes present in the Waxholm corpus, organized under the ToP classes they belong to, are summarized in Table 1.2. A vowel followed by a colon signifies a *long* vowel, whilst a vowel without a following colon signifies a *short* vowel. For instance, compare the pronunciations of the two Swedish words `ja` and `skicka`, which are pronounced as `JA:` and `SJIKKA`, respectively.

2 Signal Processing

The audio data found in wave files, the Waxholm `.smp` files and read from microphones, is typically in *pulse-code modulated*, PCM, format. That is, it is simply the amplitude of the sound wave sampled at a fixed frequency with uniform intervals. A uniform sample interval means that the distance between two possible sample values is an integer multiple of a constant value. That is, $|x(t_1) - x(t_2)| = N \times C$, where N is an integer and C is a constant. This is illustrated in Figure 4.

Milliseconds	Phoneme	Word
229	J	Jag
333	A:	
434	G	
434	V	Vill
472	I	
534	L	
597	Å	Åka
745	K	
838	A	
907	T	Tidigt
1044	I:	
1186	D	
1264	I	
1353	T	
1401	P	På
1504	Å:	Morgonen
1548	M	
1620	Å	
1736	R	
1794	Å	
1858	N	
1897	E0	
2033	N	

Table 1: Extracted information from `fp2038.11.04.smp.mix`.

All of the audio recordings in the Waxholm corpus are in PCM format and encoded as 16-bit signed big endian integers, which, except for the endianness, is the same format and encoding found in most wave files. Furthermore, the audio files in the Waxholm corpus are sampled at 16 kHz and have only one channel (mono.) Since integers are inconvenient to work with, the samples that are read from files or the microphone in the ERIS project are converted into double precision floating point values (`double`) in the range $[-1, 1]$. This is the format raw audio data will be assumed to be in from here on, unless explicitly stated otherwise.

When training a model to fit a particular set of training data, it is done under the assumption that said data is somehow representative of the data the model should recognize. As the raw PCM data from a microphone will almost certainly contain some noise that does not carry meaningful information for recognizing speech, it needs to be transformed in a fashion which filters out superfluous information while retaining as much relevant information as possible. The transformed signal should also be in a format that's practical to use with the model that is to be trained. For hidden Markov models, a convenient representation of the audio signal is as a sequence of real valued vectors, so called *feature vectors*. This section will describe how to transform PCM data into a sequence of such vectors.

Back vowel	A:	A	O:	O	U:	U	Ä	Ä
Frontal vowel	E:	E	E0	I:	I	Y:	Y	Ä:
	Ä	Ä3	Ä4	Ö:	Ö	Ö3	Ö4	
Stop	B	D	2D	G	K	T	2T	P
Fricative	F	H	S	2S	SJ	TJ		
Semi-vowel	J	V						
Nasal	M	N	2N	NG				
Lateral	L	2L						
Tremulant	R							
Silence	sil							

Table 2: Phonemes present in the Waxholm corpus, by ToP classes.

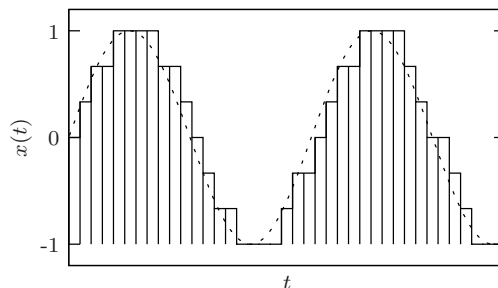


Figure 4: PCM format - fixed frequency and uniform sample intervals.

2.1 Windowing

The first step to extract feature vectors from an audio signal is to slice it up into fixed length buffers, called *windows*, each which will be used to produce one vector. The window length may vary, depending on the type of feature vectors to be extracted. In this project the windows are 25 ms long with a shift of 10 ms. Thus, a window will overlap the previous one by 15 ms. If the sample rate of the signal is 16 kHz, a window will be 400 samples long⁵. See Figure 6 for an illustration of how the windowing is performed.

Since there is an overlap of data in two consecutive windows, it's unnecessary work to read the whole window in each step. If one window of data has been used, the part of the window that overlaps with the next may just be copied from the end to the beginning, and only the non-overlapping segment needs to be read from the audio source. This is illustrated in Figure 7.

If l is the length of a window in number of samples, s is the shift length and there are a total of $n \geq l$ number of samples in a particular audio file, $1 + \lfloor (n-l)/s \rfloor$ number of windows can be completely filled. If the recording is very short, which it very well may be when generating feature vectors for

⁵ $16000 \frac{1}{s} \times 0.025s = 400$

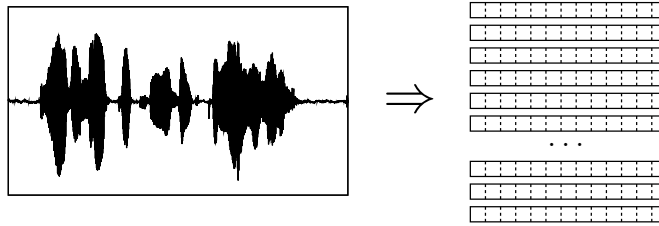


Figure 5: PCM data is transformed to a sequence of vectors.

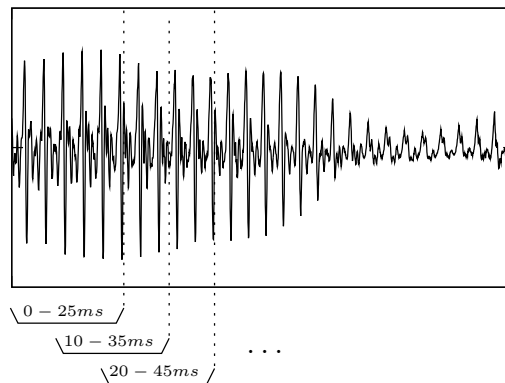


Figure 6: Window slicing.

individual phonemes, it may be desirable to allow for the last window to be only partially filled and padded with zeroes. The threshold for this project, chosen rather arbitrarily and not further explored, is that a window will be kept if it is filled to at least 80%.

When a window has been filled with PCM data, a *window function* is typically applied to it. For finite list of audio samples, a window function is essentially nothing more than a vector of weights which the samples are element-wise multiplied with. This is done in order to emphasize the effect of the samples in the middle of the window and de-emphasize the samples on the edges of the buffer. The rectangular window function is a no-operation as all weights are 1. Huang et al. (2001) defines the *generalized Hamming window* as

$$w(n) = \begin{cases} (1 - \alpha) - \alpha \cos\left(\frac{2\pi n}{N}\right) & 0 \leq n < N \\ 0 & \text{Otherwise} \end{cases} \quad (1)$$

When α is set to 0.46 it is typically referred to as *the Hamming window* and with α set to 0.5 it is the *Hann window function*⁶. The different window

⁶The Hann window is often referred to as the *Hanning window*. Though it's named

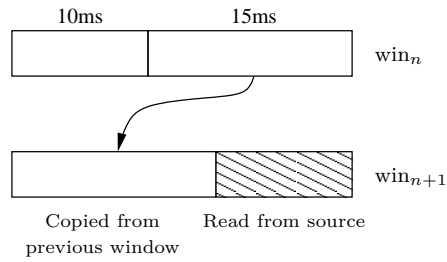


Figure 7: Window shifting.

functions implemented in this project are listed below and illustrated in Figure 8.

- **Rectangular** $w(n) = 1$
- **Hamming** $w(n) = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N}\right)$
- **Hann** $w(n) = 0.5 \left(1 - \cos\left(\frac{2\pi n}{N}\right)\right)$
- **Cosine** $w(n) = \cos\left(\frac{\pi n}{N} - \frac{\pi}{2}\right)$

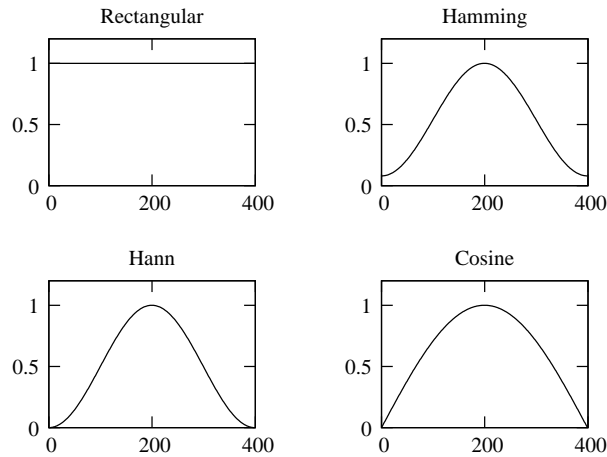


Figure 8: Window functions. Sample number $0 \leq n < N$ on the x -axis and sample weight on the y -axis.

after Julius Von Hann, the former is arguably more correct.

2.2 Mel-Frequency Cepstral Coefficients

A cepstrum⁷ is the result of taking the Fourier transform of a log-energy power spectrum of a signal. Huang et al. (2001) defines *mel-frequency cepstral coefficients* as

The *Mel-Frequency Cepstrum Coefficients* (MFCC) is a representation defined as the real cepstrum of a windowed short-time signal derived from the FFT of that signal. The difference from a real cepstrum is that a nonlinear frequency scale is used, which approximates the behavior of the auditory system.

Furthermore, they define the *discrete Fourier transform* (DFT) as

$$X_k = \sum_{n=0}^{N-1} x_n e^{-j2\pi nk/N} \quad 0 \leq k < N \quad (2)$$

The audio signal $x_n, 0 \leq n < N$ is a vector of real valued samples in the range $[-1, 1]$. The result of taking the DFT of a real valued vector is a vector of complex numbers $X_k, 0 \leq k < N$, where the imaginary and real components correspond to the amplitude coefficients of the sinus and cosinus frequency components of the audio signal expressed as a Fourier-series. Taking the logarithm of the square of the absolute of $X(k)$ yields the log-power spectrum of the signal.

The human ear does not perceive frequencies in a linear fashion. Doubling a frequency, for example, does not double the perceived pitch. The mel scale maps frequencies into these non-linear perceptions of it. The function $\text{mel}(x)$ and its inverse $\text{mel}^{-1}(y)$ are shown in Equations 3 and 4 and illustrated in Figures 9 and 10.

$$\text{mel}(x) = 1125 \log(1 + x/700) \quad (3)$$

$$\text{mel}^{-1}(y) = 700 [\exp(y/1125) - 1] \quad (4)$$

The power spectrum is averaged over a set of triangular filters, called a filter bank, which are equally spaced in the mel scale. If there are M filters in the filter bank, f_l and f_h are the lowest and highest frequencies and F_s is the sampling frequency, the frequencies for the filters are given by

$$f_m = \left(\frac{N}{F_s} \right) \text{mel}^{-1} \left(\text{mel}(f_l) + m \frac{\text{mel}(f_h) - \text{mel}(f_l)}{M + 1} \right) \quad (5)$$

$$0 \leq m \leq M + 1$$

The triangular filters $H_m(k)$ are then given by Equation 6, and are illustrated in Figure 11.

⁷The word *cepstrum* is derived from reversing the first four letters in the word *spectrum*.

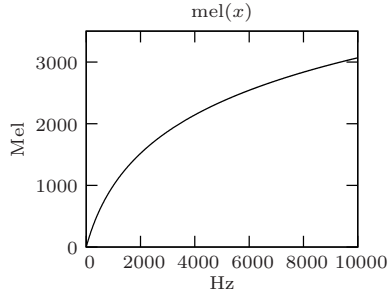


Figure 9: Mel scale.

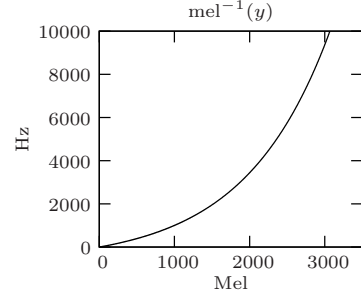


Figure 10: Inverse of mel scale.

$$H_m(k) = \begin{cases} 0 & k < f_{m-1} \\ \frac{2(k - f_{m-1})}{(f_{m+1} - f_{m-1})(f_m - f_{m-1})} & f_{m-1} \leq k \leq f_m \\ \frac{2(f_{m+1} - k)}{(f_{m+1} - f_{m-1})(f_{m+1} - f_m)} & f_m < k \leq f_{m+1} \\ 0 & k > f_{m+1} \end{cases} \quad (6)$$

$$1 \leq m \leq M$$

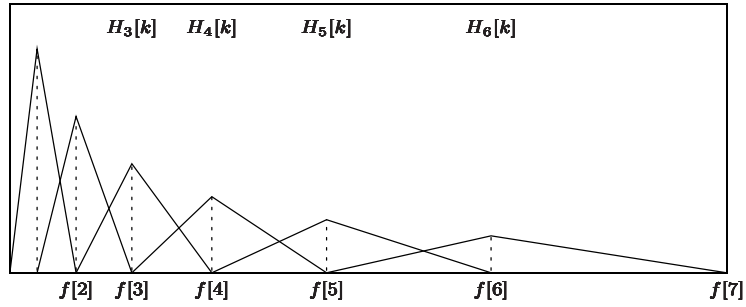


Figure 11: Filter bank.

The logarithms of the outputs of the filter bank is thus given by

$$S_m = \log \left[\sum_{k=0}^{N-1} |X_k|^2 H_m(k) \right] \quad 1 \leq m \leq M \quad (7)$$

The MFCC coefficients is then the discrete cosine transform (DCT) of the M filter outputs

$$c_n = \sum_{m=0}^{M-1} S_m \cos(\pi n(m - 1/2)/M) \quad 0 \leq n \leq M \quad (8)$$

The ERIS system uses the *Fastest Fourier Transform in the West* library (Frigo and Johnson, 2005), FFTW, for doing the Fourier transforms and discrete cosine transforms.

2.3 Feature Vector Deltas

A feature vector describes the information relevant for speech at a particular point in time. In order to increase the representativeness, information of how the signal *changes* may also be incorporated into the feature vector by use of *deltas*. Informally, a delta is simply the difference between the MFCC coefficients at some time in the future and the coefficients of some time in the past.

If the MFCC coefficients generated from a particular window is denoted by c_t , the first order delta is $\Delta c_t = c_{t+m} - c_{t-n}$. For first order deltas, Huang et al. (2001) recommends $m = 2$ and $n = 2$, that is, $\Delta c_t = c_{t+2} - c_{t-2}$ as well as the second order delta being calculated as $\Delta\Delta c_t = \Delta c_{t+1} - \Delta c_{t-1}$. With a 10 ms window shift, this means that the first order delta captures the *rate of change* of the coefficients over a 40 ms interval and the second order delta captures the *rate of the change of the change* over a 60 ms interval.

Figure 12 illustrates how the the first order delta at time t depends on the MFCC coefficients at time $t - 2$ and $t + 2$, as well as how the second order delta at time t depends on the first order deltas at time $t - 1$ and $t + 1$ (and hence, on the MFCC coefficients at time $t - 3$ and $t + 3$.)

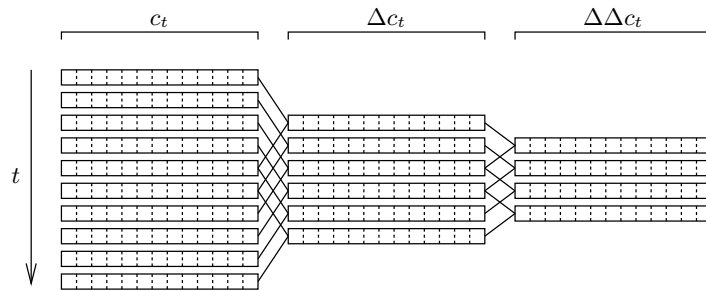


Figure 12: Calculation of first and second order deltas: $\Delta c_t = c_{t+2} - c_{t-2}$, $\Delta\Delta c_t = \Delta c_{t+1} - \Delta c_{t-1}$.

In summary:

- c_t MFCC coefficients
- $\Delta c_t = c_{t+2} - c_{t-2}$ First order delta
- $\Delta\Delta c_t = \Delta c_{t+1} - \Delta c_{t-1}$ Second order delta

If second order deltas is used with 13 MFCC coefficients, the feature vectors will be 39 elements long

$$x_t = \begin{pmatrix} c_t \\ \Delta c_t \\ \Delta\Delta c_t \end{pmatrix} \quad (9)$$

In the ERIS speech recognition system, the usage of first and second order MFCC deltas is controlled from the configuration file by specifying the respective time intervals for the keys `efx.delta1` and `efx.delta2`. For instance, setting `efx.delta1 = (2,-2)` in the configuration file specifies that first order deltas should be used, and should be calculated from the MFCC coefficients at time $t+2$ and $t-2$. Specifying `(0,0)` or `none` disables the delta. First order deltas must be enabled if second order deltas are to be used.

```
efx.delta1 = (2,-2) # '(x,y)' or 'none'
efx.delta2 = (1,-1)
```

When training phoneme models, it is done from relatively short clips of audio data. This raises the question of how to deal with edge cases. For the first few windows, the deltas will depend on audio data which would correspond to times *prior* to that of the start of the sequence, and likewise, the for the last few windows the deltas would depend on audio data corresponding to time steps *after* the end of the sequence.

The strategy chosen for the ERIS system is to calculate MFCC coefficients for audio data which precedes and follows that of the actual phoneme. In the case that the phoneme is at the very start or end of a sentence, so that there isn't enough preceding or following data, it is simply padded by zeroes. This is illustrated in Figure 13. Alternative strategies, such as simply discarding the first and last few windows, are not explored in this project.

3 Definitions

3.1 Hidden Markov Models

A *Markov chain* is a statistical model which consists of a number of states. In each time step, the model transitions from one state to another, according to a probability which only depends on the state the model is currently

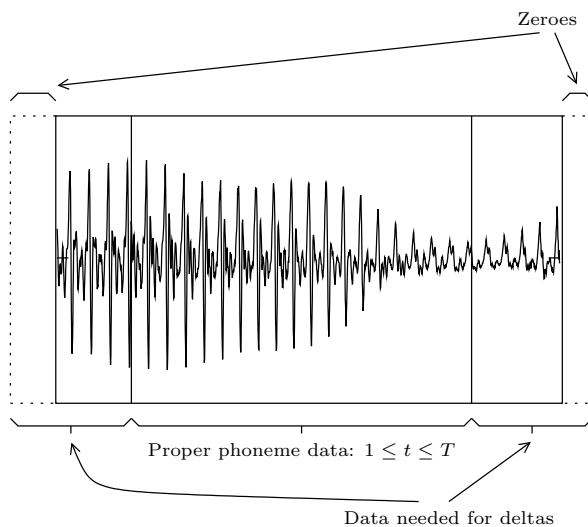


Figure 13: Padding for calculating MFCC deltas.

in. A *hidden Markov model*, HMM, is a Markov chain in which the states aren't directly visible, but instead have an observation probability function associated each state. At every time step t , the model transitions from one state to another and the new state randomly emits an observation according to observation probability function⁸. See Figure 14 for an illustration of a HMM.

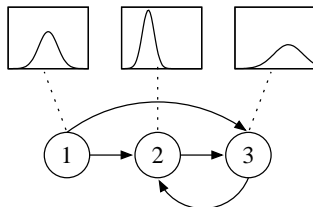


Figure 14: A hidden Markov model with three states.

There are not many limitations on the nature of the observations; they can be symbols from a discrete alphabet, real values, vectors of real values or pretty much anything else. In this project, the observations used for the HMMs are vectors of real values. More specifically, the feature vectors extracted from audio signals as described in Section 2.

The observation probability functions used are multi-variate Gaussian mixtures, which will be described in more detail later on in Section 3.2.2.

⁸Strictly speaking, an observation is something which is observed. So, the expression “emitting observations” is a bit of a misnomer.

However, since the probability for any outcome in a continuous distribution of possibilities is infinitesimal, the “probabilities” for the observations are not really probabilities in the strictest sense. As an approximation of the probability of an observation, the value of the probability *density* function for that observation is used. One consequence of this is that the “probability” of an observation can have arbitrarily high values, instead of being confined to the interval [0:1].

Formally, the components of a *hidden Markov model* are

- A set of states $S = \{1, 2, \dots, N\}$
- An initial state distribution vector $\pi = \{\pi_i\}$, $1 \leq i \leq N$
- A transition probability matrix $A = \{a_{ij}\}$, $1 \leq i, j \leq N$
- A termination probability vector $\omega = \{\omega_i\}$, $1 \leq i \leq N$
- A set of output probability functions $B = \{b_j(x)\}$

The symbol λ is used to refer to the whole set of parameters in a model:

$$\lambda = \{S, \pi, A, \omega, B\} \quad (10)$$

The initial probability π_i specifies the probability of starting in a state i . That is, the probability of being in state i at time 1. The termination probability ω_i , on the other hand, gives the probability of terminating in the state i . At any time t , the model may either transition to another state or terminate. Terminating in a state is essentially the same as transitioning to a dedicated termination state which does not consume an observation, and since at any time the the model *either* transitions to a new state or terminates, every row in the transition matrix plus the termination probability for the state which corresponds to that row should sum up to 1.

$$\left(\sum_{j=1}^N a_{ij} \right) + \omega_i = 1 \quad 1 \leq i \leq N \quad (11)$$

3.2 Observation Probability Density Functions

As previously mentioned, the *output probability functions* used are only approximations and are really *output probability density functions*, OPDFs. This means that the “probabilities” for the observations can have arbitrarily high values and are not real probabilities in the strictest sense, but will be used as if they were.

The OPDFs used in the ERIS system are multi-variate Gaussian mixtures.

3.2.1 Gaussian Probability Density Function

The probability density function $\mathcal{N}(x)$ for a Gaussian distribution for a single real value, with the mean μ and variance σ^2 is

$$\mathcal{N}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x - \mu)^2}{2\sigma^2}\right] \quad (12)$$

Figure 15 illustrates three Gaussians with different means and variances. The higher the variance, the lower the peak of the bell shaped curve will be.

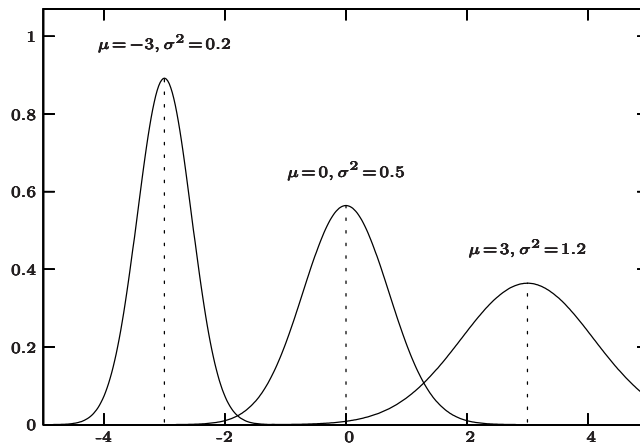


Figure 15: Gaussian probability density functions, with varying mean μ and variance σ^2 .

3.2.2 Multi-variate Gaussians

The Gaussian probability density function for real valued vectors is

$$\mathcal{N}(x) = \frac{1}{\sqrt{(2\pi)^k |\Sigma|}} \exp\left[-\frac{1}{2}(x - \mu)' \Sigma^{-1} (x - \mu)\right] \quad (13)$$

where μ is the mean vector and Σ is the covariance matrix. $|\Sigma|$ is the determinant of the covariance vector and Σ^{-1} is the inverse of the covariance matrix. If the feature vectors are composed of 13 MFCC coefficients with two levels of delta, they will be 39 elements long, which means that the mean vector also will be 39 elements long and the covariance matrix will be a 39×39 matrix. Figure 16 illustrates a multi-variate Gaussian of two dimensions.

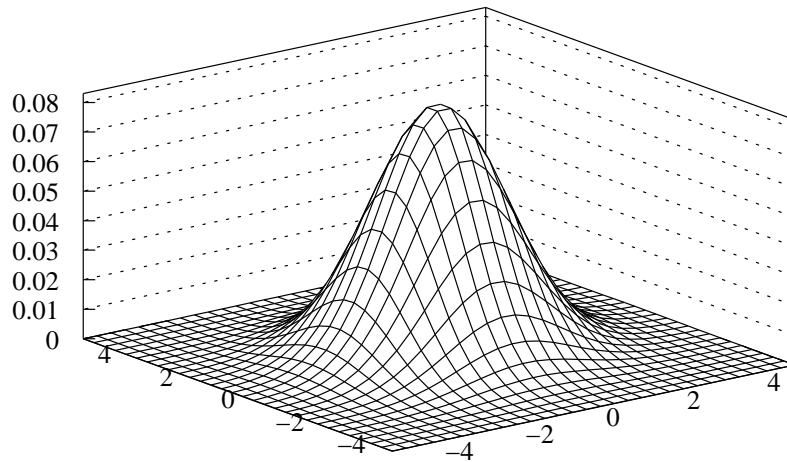


Figure 16: Multi-variate Gaussian probability density function, with $\mu = [0, 0]$ and $\Sigma = [2, 0.05; 0.05, 1.8]$.

3.2.3 Multi-variate Gaussian mixtures

Gaussian mixtures are combinations of several individual Gaussian distributions of different weights. If a Gaussian mixture consists of M number of mixture components, its probability density function is given by

$$\mathcal{N}(x) = \sum_{m=1}^M c_m \mathcal{N}_m(x) \quad (14)$$

Since there are now M different Gaussians, the parameters which are needed in order to fully specify the Gaussian mixture are:

- A vector of weights for each mixture $c_m \quad 1 \leq m \leq M$
- M number of mean vectors μ_m
- M number of covariance matrices Σ_m

Figure 17 illustrates a two dimensional Gaussian mixture with three components.

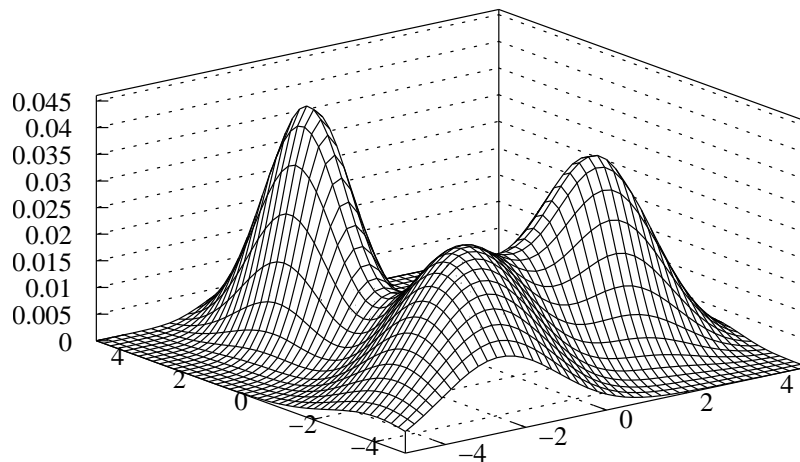


Figure 17: Multi-variate Gaussian mixture with 3 components.

3.3 Generating Observations

Algorithm 1 HMM observation generation.

function ChooseRandom(v) $\mapsto i$

If v is a non-negative row vector of length n , i is an index $1 \leq i \leq n$,
picked randomly with the probability $P(i | v) = v_i / \sum v$.

function GenerateObservation(b) $\mapsto x$

$x \leftarrow$ Vector randomly generated from the OPDF b

$i \leftarrow$ ChooseRandom(π)

$x_1 \leftarrow$ GenerateObservation(b_i)

while Random() $\geq \omega_i$ **do**

$i \leftarrow$ ChooseRandom(A_i)

$x_t \leftarrow$ GenerateObservation(b_i)

end while

$X = \langle x_1, x_2, \dots, x_T \rangle$

Given a fully specified HMM it is sometimes desirable to generate observations from that model. In a speech recognition system there may not be any direct application for this, but it can prove useful for testing whether an implementation or trained model behaves as expected. As will be seen later on, generating observations can also be used for determining the *distance* between two models.

At time $t = 1$, the state s_1 is chosen randomly according to the initial state distribution π and an observation x_1 is generated from the probability density function b_{s_1} that belongs to s_1 .

At time $t = 2$, the state s_2 is chosen randomly according to row in the transition matrix A that corresponds to previous state s_1 . The next observation x_2 is chosen by the new state s_2 's probability density function b_{s_2} . This is then repeated for $t = 3, t = 4, \dots, t = T$, producing a total of T observations. See Figure 18 and Algorithm 1.

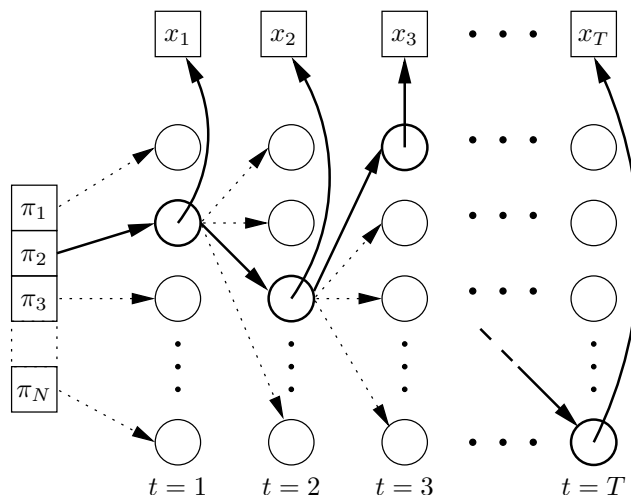


Figure 18: HMM observation generation.

3.4 Forward Probability

If given an HMM model $\lambda = \{S, \pi, A, \omega, B\}$ and sequence of observations $X = \langle x_1, x_2, \dots, x_T \rangle$, it is often of interest to find out the probability that the model would have generated that particular sequence. An obvious application of this is as a primitive way of doing word recognition. Given a set of HMM models, each of which corresponds to a word, the probability of an HMM having generated a particular sequence of observations is essentially a measurement of how well that model match the sequence. A recorded wave file, for example, can be converted into a sequence of feature vectors as described in Section 2. Then, the probability of the sequence is calculated

for each HMM and the one with the highest probability corresponds to the word which is the most likely one to have been spoken.

The most straight-forward way of calculating the probability of the sequence X is by iterating over all possible paths through the model, and calculating the probability of emitting the sequence. The probability of having generated the sequence is then the sum of the probabilities for all paths.

If N is the number of states in the HMM and $Z = \langle z_1, z_2, \dots, z_T \rangle$ is a particular path through the HMM (the sequence of states taken while traversing it.), the probability of the generating the sequence X while taking the path Z is

$$P(X | \lambda) = \sum_{\text{all } Z} P(Z | \lambda) P(X | Z, \lambda) \quad (15)$$

The probability $P(Z|\lambda)$ of taking the path Z through the HMM is simply the product of the probability of starting in state z_1 , taking the transition from z_1 to z_2 , from z_2 to z_3 , and so on until terminating in state z_T . If s_t is the state the model is in at time t – and care is taken to note the difference from z_t , which is the t :th component of the particular path Z – the probability is given by:

$$\begin{aligned} P(Z | \lambda) &= P(s_1 = z_1 | \lambda) \prod_{t=2}^T P(s_t = z_t | s_{t-1} = z_{t-1}, \lambda) \\ &= \pi_{z_1} a_{z_1 z_2} a_{z_2 z_3} \dots a_{z_{T-1} z_T} \omega_{z_T} \end{aligned} \quad (16)$$

If traversing the HMM through the specific path Z , the probability of generating the sequence of observations $X = \langle x_1, x_2, \dots, x_T \rangle$ is given by

$$\begin{aligned} P(X | Z, \lambda) &= b_{z_1}(x_1) b_{z_2}(x_2) \dots b_{z_T}(x_T) \\ &= \prod_{t=1}^T b_{z_t}(x_t) \end{aligned} \quad (17)$$

Combining Equations 16 and 17 allows for the reformulation of Equation 15 as

$$\begin{aligned} P(X|\lambda) &= \sum_{\text{all } Z} P(Z | \lambda) P(X|Z, \lambda) \\ &= \sum_{\text{all } Z} \pi_{z_1} b_{z_1}(x_1) a_{z_1 z_2} b_{z_2}(x_2) \dots a_{z_{T-1} z_T} b_{z_T}(x_T) \omega_{z_T} \end{aligned} \quad (18)$$

However, this strategy of evaluating the sequence X requires iterating over all $O(N^T)$ possible paths through the HMM, making it unusable in

practice for anything but very short sequences. It is possible to improve the time complexity, however, by noting that the optimal path through the model cannot include a suboptimal sub path.

Definition 3.1. The forward probability $\alpha_t(i)$ is defined as the probability of being in state i at time t having generated the partial observation sequence $X_1^t = \langle x_1, x_2, \dots, x_t \rangle$

$$\alpha_t(i) = P(X_1^t, s_t = i \mid \lambda) \quad (19)$$

The probability of being in state i at time $t = 1$ and seeing the first observation x_1 is simply the probability of starting in state i multiplied by the probability of state i emitting the observation x_1 .

$$\alpha_1(i) = \pi_i b_i(x_1) \quad (20)$$

The probability of being in state j at any time t , where $2 \leq t \leq T$, is then the probability of taking the transition from state i to j , multiplied by the probability of the new state j emitting the observation x_t . The probability of transitioning from i to j at time t is the probability of being in state i at time $t - 1$ multiplied by the transition probability a_{ij} . That is, $P(s_t = i, s_{t-1} = j \mid \lambda) = \alpha_{t-1}(i) a_{ij}$. The forward probability for $t > 1$ may thus be expressed recursively as the following

$$\alpha_t(j) = \left[\sum_{i=1}^N \alpha_{t-1}(i) a_{ij} \right] b_j(x_t) \quad (21)$$

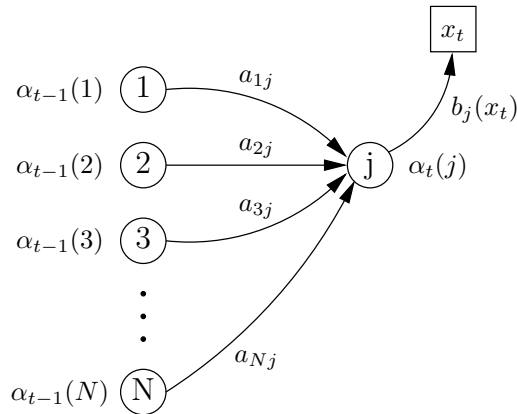


Figure 19: Calculating $\alpha_t(j)$.

- $\alpha_{t-1}(i)$ The probability of being in state i at the previous time step
- a_{ij} If being in state i , the probability of going to state j
- $b_j(x_t)$ If being in state j , the probability of observing x_t

The recursion in Equation 21 is illustrated in Figure 19.

Since $\alpha_T(i)$ is the probability of being in state i after seeing the last observation, the probability for the whole sequence $P(X|\lambda)$ is the sum of α_T for all states multiplied by their termination probabilities.

$$P(X|\lambda) = \sum_{i=1}^N \alpha_T(i) \omega_i \quad (22)$$

Combining Equations 20, 21 and 22 yields Algorithm 2. Iterating over all possible paths would have required $O(N^T)$ operations, as there are potentially N^T possible paths of length T through an HMM with N states. The main loop of the forward algorithm, which is executed $T - 1$ times, iterates through all N states and sums the probabilities of coming from any of the N states. Thus the time complexity of the forward algorithm is $O(TN^2)$ – remarkably less than $O(N^T)$.

Algorithm 2 The forward algorithm.

Step 1: Initialization

for $i \leftarrow 1$ to N **do**
 $\alpha_1(i) \leftarrow \pi_i b_i(x_1)$
end for

Step 2: Recursion

for $t \leftarrow 2$ to T **do**
for $j \leftarrow 1$ to N **do**
 $\alpha_t(j) \leftarrow \left[\sum_{i=1}^N \alpha_{t-1}(i) a_{ij} \right] b_j(x_t)$
end for
end for

Step 3: Termination

$$P(X | \lambda) \leftarrow \sum_{i=1}^N \alpha_T(i) \omega_i$$

3.5 Backward Probability

The forward probability $\alpha_t(i)$ is the probability of being in the state i after having seen the first t observations. Closely related, the *backward* probability $\beta_t(i)$ is defined as the probability of observing the rest of sequence X_{t+1}^T if being in state i at time t .

$$\beta_t(i) = P\left(X_{t+1}^T \mid s_t = i, \lambda\right) \quad (23)$$

At this point, it's not immediately obvious why the backward probability might be of interest. It is introduced here because of its close similarity with the forward probability and because they will both be needed later on in Section 4.3, when estimating HMM parameters with the Baum-Welch algorithm.

For $t < T$, the definition of the backward probability $\beta_t(i)$ is fairly straight-forward. If we are in state i at time t we can observe the rest of the sequence X_{t+1}^T by taking any transition, observing the next observation x_t and then the rest of the sequence in the same manner recursively, as per Equation 24.

$$\beta_t(i) = \sum_{j=1}^N a_{ij} b_j(x_{t+1}) \beta_{t+1}(j) \quad 1 \leq t < T \quad (24)$$

Since $\beta_t(i)$, in the equation above, is a function of x_{t+1} it cannot be used for calculating $\beta_T(i)$, as there is no symbol x_{T+1} . Instead $\beta_T(i)$ is defined as the termination probability for state i .

$$\beta_T(i) = \omega_i \quad (25)$$

To justify this definition, it should be noted that the inclusion of the termination probability vector ω is conceptually equivalent to having a dedicated non-consuming termination state to which every other state s_i has an edge. The weight of the edge – that is, its transition probability – is the termination probability ω_i . This is illustrated in Figure 20.

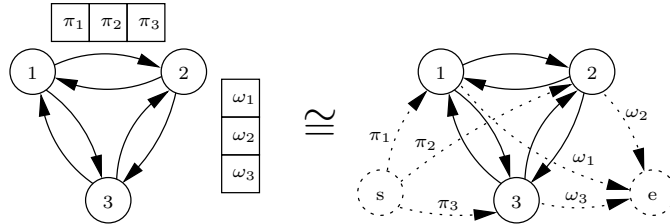


Figure 20: Having explicit π and ω vectors are equivalent to having dedicated non-consuming start and termination nodes.

Instead of the dedicated start and termination states being non-consuming, consider adding the two special symbols x_s and x_e , as illustrated in Figure 21. x_s will match the start state with the probability 1, and any other state with the probability 0. The termination state will likewise only match the termination special symbol x_e .

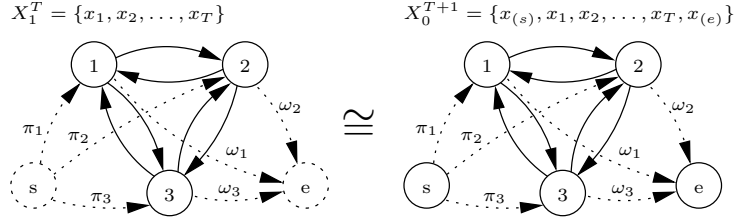


Figure 21: Augmenting the observation sequence X for consuming start and termination states.

In this view of the model, the symbol x_{T+1} is defined as $x_{(e)}$ and the same reasoning as was used for Equation 24 may be used for $\beta_T(i)$ as well. In Equation 24 all possible transitions are summed over, but in the case of the termination symbol $x_{(e)}$ all terms in the summation will be zero due to that none of the observation probability functions b_j will match it. On the other hand, a new edge from state i to the termination state with the transition probability ω_i has been introduced. Hence, the summation is replaced by taking the only transition which may yield a non-zero probability.

Per definition, $b_e(x_{(e)}) = 1$. Also, the probability of observing the remainder of the sequence, after the whole sequence has already been observed, is of course also 1. That is, $\beta_{T+1}(j) = 1$. Thus, the definition of $\beta_T(i)$ in Equation 25 is justifiable.

$$\begin{aligned} \beta_T(i) &= \omega_i b_e(x_{(e)}) \beta_{T+1}(j) \\ &= \omega_i \end{aligned}$$

Combining Equations 24 and 25 yields Algorithm 3. Figure 22 illustrates the calculation.

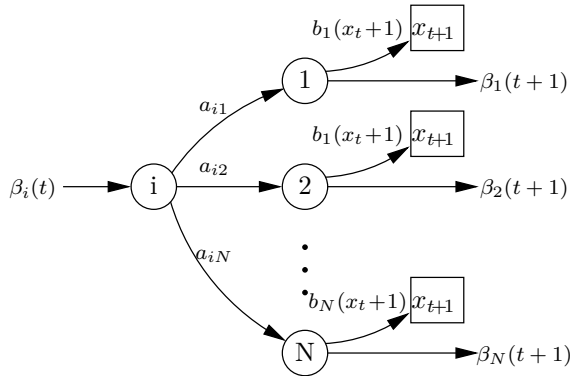


Figure 22: Calculating $\beta_t(i)$.

Algorithm 3 The backward algorithm.

Step 1: Initialization**for** $i \leftarrow 1$ to N **do**

$$\beta_T(i) \leftarrow \omega_i$$

end for**Step 2:** Recursion**for** $t \leftarrow (T - 1)$ to 1 **do****for** $i \leftarrow 1$ to N **do**

$$\beta_t(i) \leftarrow \left[\sum_{j=1}^N a_{ij} b_j(x_{t+1}) \beta_{t+1}(j) \right]$$

end for**end for****Step 3:** Termination

$$P(X | \lambda) \leftarrow \sum_{i=1}^N \alpha_T(i) \omega_i$$

3.6 Viterbi Algorithm

Closely related to the problem of finding the probability of a sequence X , is finding out what the most probable path through an HMM is, had the model indeed generated that sequence. Arguably the most common algorithm for doing this is the *Viterbi* algorithm. It is nearly identical to the forward algorithm in Section 3.4, except that instead of summing the probabilities of all possible paths up to a state, it only keeps the path with the highest probability along with pointers of which states it consists of. After the whole sequence has been iterated through, the back pointers are unwinded, revealing the most probable path.

In Algorithm 4, $V_t(i)$ is the probability of the most probable sub path that ends in state i at time t and B is the matrix of back pointers. A sub path $S = \{s_1, s_2, \dots, s_{t-1}, s_t\}$ has the probability $V_t(i)$, where $i = s_t$. $B_t(i)$ is the previous state in the sub path, so that $s_{t-1} = B_t(i)$. As not to be confused of the boundary cases in the algorithm; while V is defined for $1 \leq t \leq T$, B is only defined for $2 \leq t \leq T$.

Note that the probability of the most probable path $P(S | X, \lambda)$, which is what the Viterbi algorithm produces, will typically be different from the probability of the model having generated the sequence $P(X | \lambda)$, which the forward algorithm produces. The former is the probability of one particular path while the latter is a summation of the probabilities of all possible paths.

Algorithm 4 The Viterbi algorithm.

Step 1: Initialization

for $i = 1$ to N **do**

$$V_1(i) \leftarrow \pi_i b_i(x_1)$$

end for

Step 2: Recursion

for $t = 2$ to T **do**

for $j = 1$ to N **do**

$$V_t(j) \leftarrow \left[\max_{1 \leq i \leq N} V_{t-1}(i) a_{ij} \right] b_j(x_t)$$

$$B_t(j) \leftarrow \left[\operatorname{argmax}_{1 \leq i \leq N} V_{t-1}(i) a_{ij} \right]$$

end for

end for

Step 3: Termination

$$s_T \leftarrow \left[\operatorname{argmax}_{1 \leq i \leq N} V_T(i) \omega_i \right]$$

Step 4: Backtracking

for $t = (T - 1)$ to 1 **do**

$$s_t \leftarrow B_{t+1}(s_{t+1})$$

end for

$S = (s_1, s_2, \dots, s_T)$ Most probable sequence

$P(S | X, \lambda) = \left[\max_{1 \leq i \leq N} V_T(i) \omega_i \right]$ Probability of S

3.7 Kullback-Leibler Distance

The actual values for the parameters of two HMMs say very little of whether they will give similar probabilities for the same sequence of observations. Models with superficially similar parameters may differ a lot in how they score sequences, and models with superficially different parameters may give surprisingly similar results.

As a trivial example of this, it should be obvious that a parameter per parameter comparison between two models fails as a useful measurement if, for instance, two models are identical except for a permuted probability matrix (including the initial state distribution and termination probability vectors). It should be obvious that they are functionally equivalent, since the only difference is that the states are numbered differently, but a parameter per parameter comparison will typically yield a sizable difference.

Instead, a more sensible method of measuring the distance is the *Kullback-Leibler divergence*, or – since it used here as a distance measurement – the *Kullback-Leibler distance*. To measure the distance between two models A and B , sequences are randomly generated from either A or B . The Kullback-Leibler distance between A and B , $KL(A, B)$, is the mean of the logarithmic differences between the probabilities of the sequences for the two models. This is formalized in Equation 26, where the N number of observation sequences X_1, X_2, \dots, X_N are generated from model A .

$$KL(A, B) = \frac{1}{N} \sum_{i=1}^N \log \frac{P(X_i | A)}{P(X_i | B)} \quad (26)$$

The more sequences that are generated, the more accurate will the distance measurement be, at the expense of computational resources. Also, the more parameters there are in the models (number of states and number of Gaussian mixture components), the more sequences are needed for an accurate measurement.

One drawback of the Kullback-Leibler distance, as defined in Equation 26, is that it is asymmetric. With the same set of observation sequences X_1, X_2, \dots, X_N , $KL(A, B)$ will typically differ from $KL(B, A)$. If the distance from both directions are averaged over, however, it becomes symmetric. This is expressed in Equation 27.

$$KL_{\text{sym}}(A, B) = \frac{KL(A, B) + KL(B, A)}{2} \quad (27)$$

Since the training algorithms covered later on are non-trivial to implement and have many sources of potential errors – some quite subtle – the Kullback-Leibler distance is useful as a debugging tool during development. To ensure that an implementation of a training algorithm is at least not completely off-track, the following methodology has proved itself valuable:

1. Create two HMMs A and B with arbitrary parameters.
2. Calculate the Kullback-Leibler distance between A and B .
3. Generate sequences from A , using Algorithm 1.
4. Train B with the sequences from A .
5. The KL-distance between A and B should now be less than in step 2.

4 Training

4.1 Phoneme Modelling

The phonemes are modelled as HMMs with three states. Since the processes they model, the pronunciations of phonemes, are linear in the sense that they are unlikely to have loops of repeated features, they are modelled as left-right HMMs. This means that a transition from state i to state j is only allowed if $i \leq j$. Also, the HMM is only allowed to start in state 1 and end in the last state N . That is, for a left-right HMM, the following constraints apply:

$$\begin{aligned}
 \pi_i &= 0 & i &\neq 1 \\
 a_{ij} &= 0 & i &> j \\
 \omega_j &= 0 & j &\neq N
 \end{aligned}
 \tag{28}$$

Figure 23 illustrates an HMM model with three states and free transitions. Figure 24 illustrates a corresponding left-right model.

Additionally, one may want to ensure that in order to traverse the HMM, every single state has to be traversed. This may be done by adding the constraint that any state may only transition back to itself or to the next state. More generally, an n -degree left right model, where $n \geq 1$, is one where a state may transition only back to itself or to the n number of following states. That is, for a n -degree left-right HMM, the following constraints apply in addition to those expressed in Equation 28:

$$a_{ij} = 0 \quad j - i > n
 \tag{29}$$

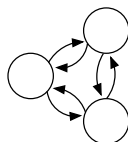


Figure 23: Free model.

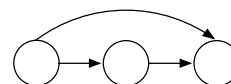


Figure 24: Left-right model.

In the ERIS configuration file, which type of HMM model is to be used is specified by setting the `hmm.statemodel` variable (see Appendix B for a sample configuration file):

- `free` Free transitions model
- `left-right` Left right model
- `left-right-n` n -degree left right model

In this project, the free and n -degree left-right models have only been used in development and debugging. All phonemes are trained as left-right models.

4.2 Clustering

Clustering is the process of grouping together similar elements. The elements that will be grouped together here are the HMM observations, that is, the feature vectors extracted in Section 2. If the feature vectors are D elements long, they can be regarded as points in D -dimensional space. For instance, a typical feature vector with 13 MFCC coefficients and two levels of deltas is a vector of 39 real values, and can thus be regarded as a point in a 39-dimensional Cartesian space. Since that many dimensions are hard to visualize, the examples and figures will be in two dimensional space, although the presented algorithms themselves have no particular limitation on the number of dimensions.

For a Cartesian space of any number of dimensions $D \geq 1$, a point P in that space can be described as by vector of D real numbers⁹: $P = (x_1, x_2, \dots, x_D)$. When constrained to three dimensions or less, x_1 , x_2 and x_3 are often referred to as the x , y and z coordinates of the point. The distance between two points $P = (x_1, x_2, \dots, x_D)$ and $Q = (y_1, y_2, \dots, y_D)$ is given by

$$\|P - Q\| = \sqrt{\sum_{i=1}^D (x_i - y_i)^2} \quad (30)$$

For two dimensions, Equation 30 is simply the Pythagorean theorem with the coordinate system translated so that either P or Q is on the origin: $c = \sqrt{a^2 + b^2}$, where c is the length of the hypotenuse and a and b are the lengths of the sides of a triangle.

If there are N number of elements, i.e. N number of feature vectors, that are to be clustered into K clusters, where $K \leq N$, it will be done in order to minimize the *within-cluster sum of squares*, WCSS. The center of a cluster $C_k, 1 \leq k \leq K$, is simply the mean of the elements in that cluster and is

⁹A *vector* in this context refers to an array of numbers, not geometrical vectors – which can also be described by *vector*, i.e. array, of numbers.

denoted by $E(C_k)$. If the size of the cluster, that is, how many elements it contains, is given by $|C_k|$

$$E(C_k) = \frac{1}{|C_k|} \sum_{x \in C_k} \quad (31)$$

Then the WCSS is the sum of the square distance for every element to the center of the cluster to which it belongs

$$\text{WCSS} = \sum_{k=1}^K \sum_{x \in C_k} \|x - E(C_k)\|^2 \quad (32)$$

The number of possible ways to cluster N elements into K clusters without any cluster being empty¹⁰ is the number of ways to pick K elements out of N possible (so that no cluster is empty) multiplied by the number of ways to distribute the remaining $N - K$ elements into K clusters.

$$\text{Number of possible clusterings} = \binom{N}{K} \times (N - K)^K$$

This number is prohibitively high for an exhaustive search for the global minimum WCSS for anything but very small numbers of N and K . Typically, heuristic algorithms for finding good-enough local minima will have to make do.

4.2.1 K-Means Clustering Algorithm

The K-means clustering algorithm is a heuristic algorithm for finding a local minimum WCSS for a set of N elements clustered into K clusters. It is conceptually simple and not particularly difficult to implement.

Of the N elements that are to be clustered, K elements are chosen at random and assigned to each cluster. The centers of the clusters $E(C_k)$ are recalculated and each of the remaining $N - K$ elements is assigned to the cluster to which it has the shortest distance. That is, if $c(x)$ is the cluster to which the element x belongs:

$$c(x) \leftarrow C_k, \quad k = \underset{1 \leq i \leq K}{\operatorname{argmin}} \|x - E(C_i)\|$$

Once all N elements have been assigned to a cluster, they are iterated through again. If an element is assigned to a cluster which *isn't* the one to which it has the shortest distance, it is reassigned. After having gone through all elements, if any there were any reassignments, the centers are recalculated again and the elements are iterated over once more. This process is then repeated until no elements can be reassigned. See Algorithm 5.

¹⁰If a cluster contains only one element, the distance for the element to the center will be 0, so it makes little sense to allow for empty clusters.

Algorithm 5 K-means clustering algorithm.

```
for  $i \leftarrow 1$  to  $K$  do
   $j \leftarrow \text{RandomInterval}(i, K)$ 
  if  $j \neq i$  then
     $\text{Swap}(x_i, x_j)$ 
  end if
   $c(x_i) \leftarrow C_i$ 
   $\text{RecalculateCenter}(C_i)$ 
end for

for  $i \leftarrow K + 1$  to  $N$  do
   $\text{RecalculateCenter}(C_i)$ 
   $\text{AddToCluster}(\text{NearestCluster}(C, x_i), x_i)$ 
end for

repeat
  for  $i \leftarrow 1$  to  $K$  do
     $\text{RecalculateCenter}(C_i)$ 
  end for
   $\text{num\_moved} \leftarrow 0$ 
  for  $i \leftarrow 1$  to  $T$  do
     $c \leftarrow \text{NearestCluster}(C, x_i)$ 
    if  $x_i \notin c$  then
       $\text{MoveToCluster}(c, x_i)$ 
       $\text{num\_moved} \leftarrow \text{num\_moved} + 1$ 
    end if
  end for
until  $\text{num\_moved} = 0$ 
```

Figure 25 illustrates 100 randomly generated two-dimensional data points clustered into three clusters using the K-means algorithm.

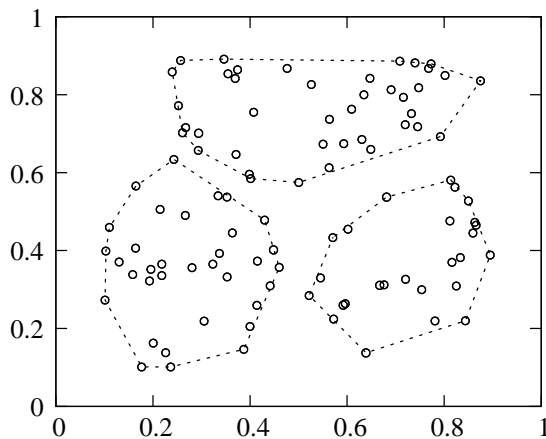


Figure 25: 100 points clustered into 3 clusters with the K-means algorithm.

Since the K-means clustering algorithm finds a set of clusters with a *local* minimum WCSS, and the particular local minimum it finds depends on the first K randomly chosen centers, it may be desirable to run the algorithm several times in order to decrease the risk of accidentally only finding an untypically bad minimum, and increase the chance of finding a really good one.

4.2.2 Bees Clustering Algorithm

The *Bees algorithm* is a heuristic search algorithm modelled after honey bees looking for flowers. The basic idea behind it is that of a number of bees going to look for flowers at random places. The bees return to the hive and share the results of their findings. Then, a number of the bees goes to search in the vicinity of the best flower patches and the rest keep looking at random places. The procedure is repeated for a fixed number of iterations or until a good enough flower patch has been found. For a heuristic optimization problem, the geographical places the bees investigate correspond to proposed solutions, and the amount of flowers at a site is the fitness of that solution.

The *Bees clustering algorithm* is, as the name implies, a clustering algorithm which uses the bees search algorithm to search for as good a way as possible to cluster a set of data points. It is described in Pham et al. (2007). Informally, it can be described as follows: Randomly generate a number of different clusterings and calculate their fitness¹¹. Then, change

¹¹Since the goal is to minimize the WCSS, a good fitness value is $WCSS^{-1}$.

the best clusterings a little bit and randomly generate new clusterings for the rest. Changing the best clusterings is the analogue of searching in the vicinity of a good flower patch. See Algorithm 6.

Of the best sites to explore further, the top e number of sites are called *elite* sites. More bees will be dispatched to search in the vicinity of the elite sites than for the rest of the top scoring sites. In the ERIS configuration file, the following parameters, listed along with their default values, may be set to control the operation of the algorithm (the prefix, `hmm.cluster.bees` is omitted. That is, the parameter `num_scouts` list below is really `hmm.cluster.bees.num_scouts`):

- `num_scouts` = 12 Total number of bees
- `num_sites` = 4 Number of best sites to explore further
- `num_elite` = 2 Number of elite sites
- `num_bees_elite` = 4 Number of bees to send to elite sites
- `num_bees_rest` = 2 Number of bees to send to non-elite sites
- `num_iterations` = 300 Number of iterations

For training HMMs, the crucial feature of the bees clustering algorithm is that one is free to implement the mutation function, the one which makes small changes to a clustering, as one see fit. If the phoneme model that is to be trained with a clustering algorithm is a left-right HMM, it must be ensured that if an element x_t belongs to cluster number k the next element x_{t+1} must belong to a cluster of number k or higher. That is,

$$c(x_t^k) \leq c(x_{t+1}^k) \quad \left\{ \begin{array}{l} 1 \leq k \leq K \\ 1 \leq t < T^k \end{array} \right. \quad (33)$$

Otherwise, there will be transitions from a higher state to a lower state. With the k-means algorithm, it is difficult to exercise such constraints. With the bees clustering algorithm, however, it is simply a matter of ensuring that the function for randomly generating new clusterings and the function which mutates one clustering into another both satisfies the constraint.

In the k-means algorithm, each data point¹² needs to have an associated pointer to which cluster it belongs. With the bees clustering algorithm, a clustering is defined by a set of *walls* which separates the observations in a sequence. Figure 26 illustrates five sequences of observations which are clustered into three clusters by being separated by two such walls.

The function which generates random clusterings simply generates random integers to serve as indexes for the walls separating the elements. The mutation function, which takes on clustering as input and randomly generates a new one by making small modifications to the original, first generates a random number of *move points*, which depends on the number of walls times the number of sequences. The more number of move points, the

¹²Feature vector / HMM observation.

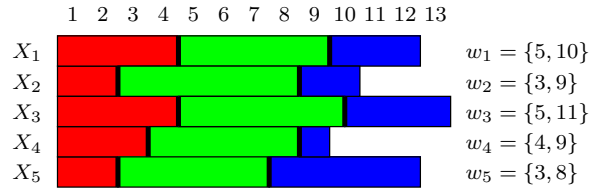


Figure 26: Five sequences of observations $X_1 \dots X_5$

larger mutation, which relates to a greater area to search in the vicinity of the flower patch. For each move point, a wall is picked at random and is randomly shifted one step to the left or right.

Algorithm 6 Bees clustering algorithm

1. Initialize the solution population.
 2. Evaluate the fitness of the population.
 - while** Stopping criterion not met **do**
 3. Form new population.
 4. Select best sites for neighborhood search.
 5. Recruit bees for selected sites (more bees for the best e sites) and evaluate fitness.
 6. Select the fittest bee from each site.
 7. Assign remaining bees to search randomly and evaluate their fitness.
 - end while**
-

4.2.3 HMM Parameter Estimation

If the parameters of an HMM model with N number of states and M number of Gaussian mixture components are to be fitted to a given set of training sequences $X = \{X_1, X_2, \dots, X_K\}$, clustering may be used to find an initial estimate of the parameters. These can then be refined using the Baum-Welch re-estimation algorithm (described in Section 4.3.)

The parameters that need to be estimated are:

- The initial state distribution vector π
- The transition probability matrix A
- The termination probability vector ω
- The Gaussian mixture weight vectors $c_j, 1 \leq j \leq N$
- The mean vector and covariance matrix for each Gaussian mixture component: μ_j^m and U_j^m

Estimating the transition probabilities (Juang and Rabiner, 1990), including the initial state distribution and termination probabilities, is a matter of counting how often one state follows another and dividing that by the total number of transitions from that state. For the initial probability π_i of an HMM state i , this means counting how many training sequences start with an element that belongs to cluster C_i , divided by the total number of training sequences K .

Similarly, the termination probabilities are given by the number of training sequences which ends with an element belonging to the cluster corresponding to a state, divided by the number of elements in the cluster. I.e. if termination is understood as a kind of special transition, the termination probability of state i is the number of “termination transitions” from i divided by the total number of transitions from i . The function $C(x, i)$ is introduced in order to express these relations formulaically.

$$C(x, i) = \begin{cases} 1 & c(x) = C_i \\ 0 & c(x) \neq C_i \end{cases} \quad (34)$$

Using Equation 34 and the above reasoning, the following equations apply for the for the initial state distribution, transition probabilities and termination probabilities:

$$\pi_i = \frac{1}{K} \sum_{k=1}^K C(x_1^k, i) \quad 1 \leq i \leq N \quad (35)$$

$$a_{ij} = \frac{\sum_{k=1}^K \sum_{t=1}^{T_k-1} C(x_t^k, i) C(x_{t+1}^k, j)}{\sum_{k=1}^K \sum_{t=1}^{T_k} C(x_t^k, i)} \quad 1 \leq i, j \leq N \quad (36)$$

$$\omega_i = \frac{\sum_{k=1}^K C(x_{T_k}^k, i)}{\sum_{k=1}^K \sum_{t=1}^{T_k} C(x_t^k, i)} \quad 1 \leq i \leq N \quad (37)$$

Note that in Equation 36 the index t in the inner sum only goes up to $T_k - 1$ in the numerator while it goes up to T_k in the denominator. The reason for this is that the fraction in Equation 36 means *the number of transitions from i to j , divided by the number of transitions from i* , and the number of transitions *from i* also includes the number of terminations in i .

Next, the parameters of the observation probability density function b_i – mixture weights, mean vectors and covariance matrices – should be estimated so that b_i fits the observations that are assigned to cluster C_i . If the

OPDF for state i only has one mixture component, $M_i = 1$, the mixture weight $c_1 = 1$ and the mean vector μ_i and covariance matrix U_i are given directly by the elements in cluster C_i :

$$\mu_i = \frac{1}{|C_i|} \sum_{x \in C_i} x \quad 1 \leq i \leq N \quad (38)$$

$$U_i = \frac{1}{|C_i|} \sum_{x \in C_i} (x - \mu_i)'(x - \mu_i) \quad 1 \leq i \leq N \quad (39)$$

If there are more than one mixture component, the elements to be fitted for the OPDF need to be clustered yet again, this time with each cluster corresponding to one Gaussian mixture component. Figure 27 illustrates the same clustering as Figure 25, but with each cluster clustered into two sub clusters – one for each mixture component of an OPDF with two components. If the elements of cluster C_i is clustered into the M_i sub clusters $C_i^1, C_i^2, \dots, C_i^{M_i}$, the mixture weight vector c_i is given by the number of elements in each sub cluster divided by the total number of elements in the cluster. The mean and covariance vectors are thus given by Equations 40 and 41, which are all but identical to Equations 38 and 39.

$$\mu_i^m = \frac{1}{|C_i^m|} \sum_{x \in C_i^m} x \quad \begin{cases} 1 \leq i \leq N \\ 1 \leq m \leq M_i \end{cases} \quad (40)$$

$$U_i^m = \frac{1}{|C_i^m|} \sum_{x \in C_i^m} (x - \mu_i^m)'(x - \mu_i^m) \quad \begin{cases} 1 \leq i \leq N \\ 1 \leq m \leq M_i \end{cases} \quad (41)$$

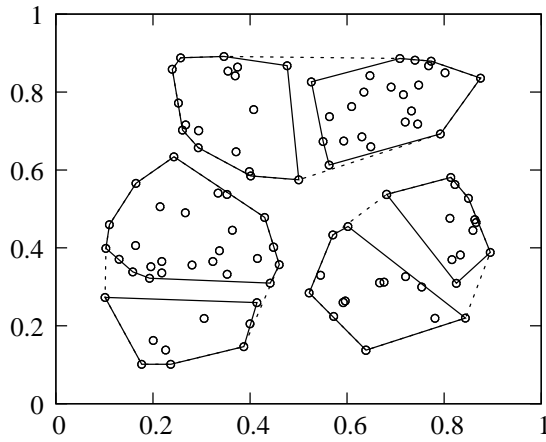


Figure 27: Clustered elements with sub clusters for Gaussian mixtures.

4.3 Baum-Welch Re-estimation

If the parameters of an HMM model has been estimated to fit a set of training sequences $X = \{X_1, X_2, \dots, X_K\}$, for instance by using one of the clustering algorithms in Section 4.2, the *Baum-Welch re-estimation algorithm* may be used to refine that estimation for an even better fit. The idea is to use the existing model to, in a sense, estimate the parameters of itself. The formulas and algorithms described in this section are taken from, and to some degree adapted from, Rabiner (1989).

4.3.1 γ and ξ

Definition 4.1. $\gamma_t(i)$ is defined as the probability of being in state i at time t , given an HMM model λ and a sequence of observations X .

$$\gamma_t(i) = P(s_t = i, X \mid \lambda) \quad (42)$$

Recall the forward and backward probabilities from Sections 3.4 and 3.5. Given a sequence of observations $X = \langle x_1, x_2, \dots, x_T \rangle$ and an HMM model λ , the forward probability $\alpha_t(i)$ is the probability of being in state i after having observed the first t observations. The backward probability $\beta_t(i)$ is the probability of observing the remainder of the sequence, X_{t+1}^T , if being in state i at time t .

$$\begin{aligned} \alpha_t(i) &= P(X_1^t, s_t = i \mid \lambda) \\ \beta_t(i) &= P(X_{t+1}^T \mid s_t = i, \lambda) \end{aligned}$$

$\gamma_t(i)$, then, can be expressed in terms $\alpha_t(i)$ and $\beta_t(i)$, if it is noted that the probability of being in state i at time t is the probability of seeing the first t observation symbols x_1, x_2, \dots, x_t , ending up in state i and then observing the remaining $T - t$ observations X_{t+1}^T , divided by the probability of being in *any* state at time t .

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^N \alpha_t(i)\beta_t(i)} \quad 1 \leq t \leq T \quad (43)$$

Definition 4.2. $\xi_t(i, j)$ is defined as the probability of taking the transition from state i to state j at time t , given an HMM model λ and a sequence of observations X . That is, it is the probability of being in state i at time t and in state j at time $t + 1$.

$$\xi_t(i, j) = P(s_t = i, s_{t+1} = j, X \mid \lambda) \quad (44)$$

ξ may also be expressed in terms of α and β . The expression is nearly identical to that of γ , except that after observing the first t observations, it includes a_{ij} , the probability of transitioning from state i to j , and $b_j(x_{t+1})$, the probability of observing the next observation x_{t+1} in state j . This is multiplied with the probability of observing the remainder of the sequence, X_{t+2}^T , divided by the probability of taking *any* transition from state i at time t .

$$\xi_t(i, j) = \frac{\alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j)}{\sum_{i=1}^N \sum_{j=1}^N \alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j)} \quad 1 \leq t < T \quad (45)$$

Take care to note that, while γ is defined for $1 \leq t \leq T$, ξ is only defined for $1 \leq t < T$. This is illustrated in Figure 28.

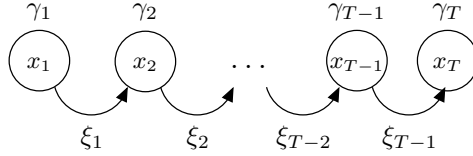


Figure 28: γ is defined for $1 \leq t \leq T$, but ξ is only defined for $1 \leq t < T$

As a convenient optimization, it turns out that it's possible to express γ in terms of ξ , as in Equation 46, by noting that the probability of being in state i at time t is the probability of taking *any* transition from state i at time t . See Appendix A.1 for a formulaic justification of this reasoning.

$$\gamma_t(i) = \sum_{j=1}^N \xi_t(i, j) \quad 1 \leq t < T \quad (46)$$

Reducing the number of terms involved in the calculations does not only make the algorithm run faster, it also reduces the risk of implementation errors. There are many different probabilities involved in Baum-Welch re-estimation, and opportunities for simplifications should be welcomed as errors are often subtle and hard to track down. However, since ξ is only defined for $t < T$, Equation 43 is still needed to calculate γ_T . However, since $\beta_T(i) = \omega_i$, it can be simplified somewhat:

$$\begin{aligned}\gamma_T(i) &= \frac{\alpha_T(i)\beta_T(i)}{\sum_{i=1}^N \alpha_T(i)\beta_T(i)} \\ &= \frac{\alpha_T(i)\omega_i}{\sum_{i=1}^N \alpha_T(i)\omega_i}\end{aligned}$$

To summarize:

$$\xi_t(i, j) = \frac{\alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j)}{\sum_{i=1}^N \sum_{j=1}^N \alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j)} \quad 1 \leq t < T \quad (45)$$

$$\gamma_t(i) = \sum_{j=1}^N \xi_t(i, j) \quad 1 \leq t < T \quad (46)$$

$$\gamma_T(i) = \frac{\alpha_T(i)\omega_i}{\sum_{i=1}^N \alpha_T(i)\omega_i} \quad (47)$$

4.3.2 Estimating Transition Probabilities

Using the estimated state and transition probabilities, γ and ξ from Section 4.3.1, the initial state distribution vector π , transition probability matrix A and termination probability vector ω of an HMM model λ may be estimated. Conceptually, the estimations are straight-forward.

$$\bar{\pi}_i = \frac{\text{Number sequences starting in state } i}{\text{Number of training sequences}} \quad (48)$$

$$\bar{a}_{ij} = \frac{\text{Number of transitions from state } i \text{ to } j}{\text{Number of times in state } i} \quad (49)$$

$$\bar{\omega}_i = \frac{\text{Number of sequences ending in state } i}{\text{Number of times in state } i} \quad (50)$$

If $\gamma_t^k(i)$ means the $\gamma_t(i)$ for the k :th training sequence, and likewise, $\xi_t(i, j)$ for the k :th training sequence by $\xi_t^k(i, j)$, the formulas for the Equations 48, 49 and 50 become:

$$\bar{\pi}_i = \frac{\sum_{k=1}^K \gamma_1^k(i)}{K} \quad (51)$$

$$\bar{a}_{ij} = \frac{\sum_{k=1}^K \sum_{t=1}^{T^{k-1}} \xi_t^k(i, j)}{\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i)} \quad (52)$$

$$\bar{\omega}_i = \frac{\sum_{k=1}^K \gamma_{T^k}^k(i)}{\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i)} \quad (53)$$

It should be noted that the summation over t in the numerator of Equation 52 only goes up to $T - 1$, as ξ is not defined for $t = T$, and a transition from a state i at time T would mean that the model terminated in i . This probability is to found in the termination probability vector ω .

Also, since the transition probability matrix A doesn't include the termination probabilities, it should not be row-stochastic in of itself, but rather the following should hold true

$$\sum_{j=1}^N (\bar{a}_{ij} + \bar{\omega}_i) = 1 \quad 1 \leq i \leq N \quad (54)$$

See Appendix A.2 for a proof that this relation does in fact hold true.

4.3.3 Estimating Observation Probability Functions

Recall that the function which determines the probability of the state i emitting the observation x , $b_i(x)$, is a multi-variate Gaussian mixture.

$$b_i(x) = \sum_{m=1}^M c_{im} \mathcal{N}(x, \mu_{im}, \Sigma_{im}) \quad \left\{ \begin{array}{l} 1 \leq i \leq N \\ \sum_{m=1}^M c_{im} = 1 \end{array} \right.$$

The parameters that need to be estimated are

- c_i Mixture weight vector $\sum_{m=1}^M c_{im} = 1$
- μ_{im} Mean vectors for the Gaussian mixtures $1 \leq m \leq M_i$
- Σ_{im} Covariance matrices for the Gaussian mixtures $1 \leq m \leq M_i$

In order to estimate these parameters, two additional terms need to be introduced: $\delta_k^t(j, m)$ and $\phi_t^k(j, m)$.

Definition 4.3. $\delta_t^k(j, m)$ is the probability of the m :th mixture component of the observation probability function for state j accounting for the observation x_t^k .

$$\delta_t^k(j, m) = \left[\frac{c_{jm} \mathcal{N}(x_t^k, \mu_{jm}, \Sigma_{jm})}{\sum_{m=1}^M c_{jm} \mathcal{N}(x_t^k, \mu_{jm}, \Sigma_{jm})} \right] \quad (55)$$

Definition 4.4. $\phi_t^k(j, m)$ is the probability of being in state j at time t , with the m :th mixture component of the observation probability function of state j accounting for the observation x_t^k .

$$\phi_t^k(j, m) = \gamma_t^k(j) \delta_t^k(j, m) \quad (56)$$

The mixture weight \bar{c}_{jm} is estimated by summing the probability of being in any state at any time with the m :th mixture component accounting for the observation, as per Equation 57. This is divided by the probability of *any* mixture component of the probability function b_j accounting for the observation.

$$\bar{c}_{jm} = \frac{\sum_{k=1}^K \sum_{t=1}^{T^k} \phi_t^k(j, m)}{\sum_{k=1}^K \sum_{t=1}^{T^k} \sum_{m=1}^M \phi_t^k(j, m)} \quad (57)$$

The mean vector $\bar{\mu}$ is the average of the observations that the m :th mixture component of the j :th state accounts for. This means summing over all observations weighted by the probability that the particular mixture component accounted for it, and then dividing this by the probability that *any* component did – as is expressed in Equation 58.

$$\bar{\mu}_{jm} = \frac{\sum_{k=1}^K \sum_{t=1}^{T^k} \phi_t^k(j, m) x_t^k}{\sum_{k=1}^K \sum_{t=1}^{T^k} \phi_t^k(j, m)} \quad (58)$$

Estimating the covariance matrix $\bar{\Sigma}$ is essentially identical to estimating the mean vector, except that it is the deviations from the mean that are summed over instead of the actual observations. Note that x_t^k and the mean vector μ_{jm} in Equation 59 are row-vectors, so the resulting covariance matrix will indeed be a square matrix.

$$\bar{\Sigma}_{jm} = \frac{\sum_{k=1}^K \sum_{t=1}^{T^k} \phi_t^k(j, m) (x_t^k - \mu_{jm})' (x_t^k - \mu_{jm})}{\sum_{k=1}^K \sum_{t=1}^{T^k} \phi_t^k(j, m)} \quad (59)$$

4.3.4 Scaling α and β

Since the “*probability*” functions really are *probability density functions*, as there is no sensible definition for probabilities of non-discrete observations, and they are built by combinations of multi-variate Gaussians for vectors of quite high dimensionality (39 for 13 MFCC coefficients and two delta levels), the probability value for an observation is typically very small. The forward and backward probabilities – α and β in Equations 21 and 24 – both involve repeated multiplications of such probabilities, as well as multiplications of the transition probabilities which are all less than or equal to 1.

$$\alpha_t(j) = \left[\sum_{i=1}^N \alpha_{t-1}(i) a_{ij} \right] b_j(x_t) \quad (21)$$

$$\beta_t(i) = \sum_{j=1}^N a_{ij} b_j(x_{t+1}) \beta_{t+1}(j) \quad (24)$$

If, for example, the smallest positive value that can be represented with a floating point data type, such as a `double`¹³, is 10^{-324} and the probabilities that are multiplied are of the order of 10^{-9} , the product will not be possible to represent after about 35 multiplications. Adding more bits to the data type used to represent the values is possible, but does not scale well for training sequences of arbitrary lengths. However, it turns out that it’s possible to work around this problem by introducing scaled versions of α and β .

First it should be note that α and β are only directly used in γ and ξ . If it is possible write those two equations in such a way that their values are still equivalent but they do not suffer from the scaling problem of repeated multiplication of small values, then that is the only change that needs to be introduced.

¹³A 64-bit IEEE floating point number, i.e. a typical `double` on most modern platforms, can represent $2^{-1074} \approx 10^{-324}$ as the smallest positive value.

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^N \alpha_t(i)\beta_t(i)} \quad (43)$$

$$\xi_t(i, j) = \frac{\alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j)}{\sum_{i=1}^N \sum_{j=1}^N \alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j)} \quad (45)$$

The scaling factors are defined as

$$c_t = \frac{1}{\sum_{i=1}^N \alpha_t(i)} \quad t = 1$$

$$c_t = \frac{1}{\sum_{j=1}^N \left[\sum_{i=1}^N \hat{\alpha}_{t-1}(i) a_{ij} \right] b_j(x_t)} \quad t > 1 \quad (60)$$

Using the scaling factor c_t , a scaled version of the forward probability $\hat{\alpha}$ may be formulated as

1. Initialization.

$$\hat{\alpha}_1(j) = c_1 \alpha_1(j) \quad (61)$$

2. Recursion

$$\hat{\alpha}_t(j) = \frac{\left[\sum_{i=1}^N \hat{\alpha}_{t-1}(i) a_{ij} \right] b_j(x_t)}{\sum_{j=1}^N \left[\sum_{i=1}^N \hat{\alpha}_{t-1}(i) a_{ij} \right] b_j(x_t)} \quad (62)$$

The initialization equation (61) may be rewritten as

$$\left. \begin{aligned} \hat{\alpha}_t(j) &= c_t \alpha_t(j) \\ &= \left(\prod_{s=1}^t c_s \right) \alpha_t(j) \end{aligned} \right\} t = 1 \quad (63)$$

The recursion step (62) may also be rewritten in a similar fashion

$$\begin{aligned}
\hat{\alpha}_t(j) &= \frac{\left[\sum_{i=1}^N \hat{\alpha}_{t-1}(i) a_{ij} \right] b_j(x_t)}{\sum_{j=1}^N \left[\sum_{i=1}^N \hat{\alpha}_{t-1}(i) a_{ij} \right] b_j(x_t)} \\
&= c_t \left[\sum_{i=1}^N \hat{\alpha}_{t-1}(i) a_{ij} \right] b_j(x_t) \\
&= c_t \left[\sum_{i=1}^N \left(\prod_{s=1}^{t-1} \alpha_{s-1}(i) a_{ij} \right) \right] b_j(x_t) \\
&= \left(\prod_{s=1}^t \alpha_s \right) \left[\sum_{i=1}^N \alpha_{t-1}(i) a_{ij} \right] b_j(x_t) \\
&= \left(\prod_{s=1}^t \alpha_s \right) \alpha_t(j) \quad t > 1
\end{aligned} \tag{64}$$

Combining Equations 63 and 64 yields an expression of $\hat{\alpha}$ defined for all t :

$$\hat{\alpha}_t(j) = \left(\prod_{s=1}^t \alpha_s \right) \alpha_t(j) \quad 1 \leq t \leq T \tag{65}$$

Two additional terms which will prove useful are C_t and D_t .

$$C_t = \prod_{s=1}^t c_s \tag{66}$$

$$D_t = \prod_{s=t}^T c_s \tag{67}$$

Combining equations (63), (64) and (66) yields

$$\hat{\alpha}_t(j) = C_t \alpha_t(j) \quad 1 \leq t \leq T \tag{68}$$

Next, a scaled version of the backward probability β is introduced.

1. Initialization

$$\hat{\beta}_T(i) = c_T \beta_T(i) \tag{69}$$

2. Recursion

$$\hat{\beta}_t(i) = c_t \left[\sum_{j=1}^N a_{ij} b_j(x_{t+1}) \hat{\beta}_{t+1}(j) \right] \tag{70}$$

Equation 69 is rewritten as

$$\hat{\beta}_T(i) = \left(\prod_{s=t}^T \right) \beta_t(i) \quad t = T \quad (71)$$

Equation 70 is also rewritten

$$\begin{aligned} \hat{\beta}_t(i) &= c_t \left[\sum_{j=1}^N a_{ij} b_j(x_{t+1}) \hat{\beta}_{t+1}(i) \right] \\ &= c_t \left[\sum_{j=1}^N a_{ij} b_j(x_{t+1}) \left(\prod_{s=t+1}^T c_s \right) \beta_{t+1}(i) \right] \\ &= \left(\prod_{s=t}^T c_s \right) \left[\sum_{j=1}^N a_{ij} b_j(x_{t+1}) \beta_{t+1}(i) \right] \\ &= \left(\prod_{s=t}^T c_s \right) \beta_t(i) \quad 1 \leq t < T \end{aligned} \quad (72)$$

Combining (71), (72) and (67) yields a similar equation for $\hat{\beta}$ as for the scaled forward probability $\hat{\alpha}$ in Equation 68.

$$\hat{\beta}_t(i) = D_t \beta_t(i) \quad 1 \leq t \leq T \quad (73)$$

Now, it is possible to rewrite γ and ξ in terms of the scaled forward and backward probabilities $\hat{\alpha}$ and $\hat{\beta}$, and it turns out they are equivalent to the unscaled versions. This means that the only thing that is needed in order to avoid overflowing the exponents in the floating point representations of the forward and backward probabilities is to change the calculations to use the scaled versions instead. All other calculations remain the same. See Appendix A.3 and A.4 for proofs of Equations 74 and 75.

$$\hat{\gamma}_t(i) = \frac{\hat{\alpha}_t(i) \hat{\beta}_t(i)}{\sum_{i=1}^N \hat{\alpha}_t(i) \hat{\beta}_t(i)} = \gamma_t(i) \quad (74)$$

$$\hat{\xi}_t(i, j) = \frac{\hat{\alpha}_t(i) a_{ij} b_j(x_{t+1}) \hat{\beta}_{t+1}(j)}{\sum_{i=1}^N \sum_{j=1}^N \hat{\alpha}_t(i) a_{ij} b_j(x_{t+1}) \hat{\beta}_{t+1}(j)} = \xi_t(i, j) \quad (75)$$

The algorithms for calculating the scaled forward and backward probabilities are presented in Algorithms 7 and 8.

Algorithm 7 The scaled forward algorithm.

Step 1: Initialization**for** $i \leftarrow 1$ to N **do**

$$\hat{\alpha}_1(i) \leftarrow \pi_i b_i(x_1)$$

end for

$$s_1 \leftarrow \text{Normalize}(\hat{\alpha}_1)$$

Step 2: Recursion**for** $t \leftarrow 2$ to T **do****for** $j \leftarrow 1$ to N **do**

$$\hat{\alpha}_t(j) \leftarrow \left[\sum_{i=1}^N \hat{\alpha}_{t-1}(i) a_{ij} \right] b_j(x_t)$$

end for

$$s_t \leftarrow \text{Normalize}(\hat{\alpha}_t)$$

end for**Step 3:** Termination

$$P(X \mid \lambda) \leftarrow \left(\prod_{t=1}^T s_t \right) \sum_{i=1}^N \alpha_T(i) \omega_i$$

Algorithm 8 The scaled backward algorithm.

{Note: The scaling factors s_t used here are those that were produced by the scaled forward algorithm – Algorithm 7.}

Step 1: Initialization**for** $i \leftarrow 1$ to N **do**

$$\hat{\beta}_T(i) \leftarrow \omega_i / s_T$$

end for**Step 2:** Recursion**for** $t \leftarrow (T - 1)$ to 1 **do****for** $i \leftarrow 1$ to N **do**

$$\hat{\beta}_t(i) \leftarrow \left[\sum_{j=1}^N a_{ij} b_j(x_{t+1}) \hat{\beta}_{t+1}(j) \right]$$

end for

$$\hat{\beta}_t(i) \leftarrow \hat{\beta}_t(i) / s_t$$

end for

5 Word Recognition

5.1 Building The Word Graph

When the HMM models representing the phonemes have been trained, a larger HMM will be built from these, using a dictionary of phonetic pronunciations for words. When doing word-recognition, the audio source – such as a wave file or a microphone – is transformed into a sequence of feature vectors (as is described in Section 2). For this sequence, the most probable path through the larger HMM, from here on referred to as the *word graph*, reveals which words were the ones most probable to have been spoken. Figure 5.1 illustrates the process of building a word graph from the set of HMM models representing phonemes and a dictionary of phonetic pronunciations.

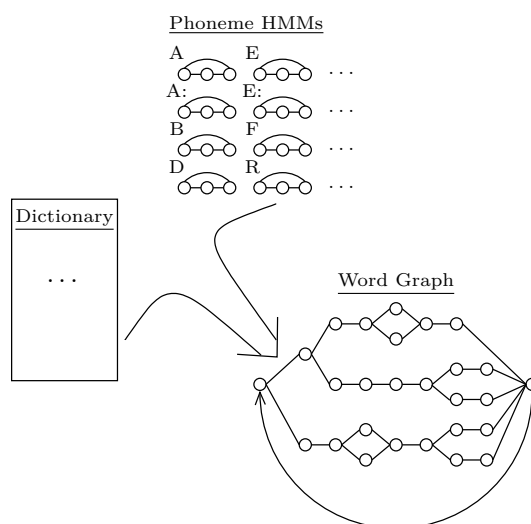


Figure 29: Building the word graph.

The dictionary is a text file which contains a list of all words that the system should be able to recognize, as well as their phonetic pronunciations. See Table 5.1 for an example of a small dictionary or Appendix C for the full dictionary used to test the ERIS system.

It is possible for a word to have more than a single pronunciation. These could be specified by having an entry for each pronunciation, but the ERIS Perl-script for constructing the word graphs, `eris_mk_gx.pl`, understands pseudo-regular-expression-like constructs – such as $(A|B|C)$ – to indicate that either the phoneme A, B or C should be pronounced. A question mark ‘?’ after a phoneme indicates that the pronunciation of it is optional. Thus $(A|B)?$ means that either A, B or neither of them should be pronounced. Consider the word `Acceptans` in the dictionary in Table 5.1. Its phonetic

pronunciation is specified as AKSEPTA(N|NG)S, which means it could be pronounced either as AKSEPTANS or AKSEPTANGS¹⁴.

Word	Phonetic pronunciation
Acceptans	A K S E P T A (N NG) S
Ja	J A:
Nej	N E J
Ring	R I NG
Stäng	S T Ä NG
Öppna	Ö P N A

Table 3: Example dictionary of phonetic pronunciations.

The dictionary is expanded from a one-to-one mapping of words to pronunciations, to a one-to-many mapping with all pronunciations explicitly stated. That is, without the pseudo-regular expressions for alternatives – as is illustrated in Figure 5.1.

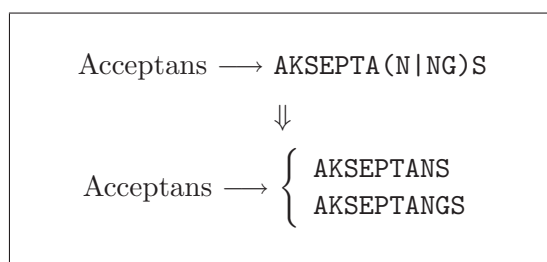


Figure 30: Dictionary expanded to one-to-many mapping.

The phonemes in the word pronunciations are then mapped into the proper triphones by inspecting the phoneme classes of the preceding and following phonemes. In Figure 5.1, the first phoneme, A, is followed by K – which is a Stop-phoneme – and hence the context-free phoneme A is replaced by one with a right-context, i.e. A_{<st>}. The next phoneme, K is preceded by A, which is a Back-vowel, and followed by S, which is a Fricative. Hence, it is replaced by the triphone <bv>K<fr>. This is repeated for the remainder of the phonemes.

Since some triphones rarely occur in practice, it's not necessarily the case that all the triphones needed by the dictionary exist in the corpus and have been trained. This is especially probable if there is threshold set, so that only phonemes with at least a minimum number of training sequences are actually trained.

¹⁴'NG' is one phoneme

Class	bv	st	fr	fv	st	st	bv	na	na	fr
Phoneme	A	K	S	E	P	T	A	N	NG	S

AKSEPTANS \longrightarrow A_{<st>} <fv>K_{<f>} <st>S_{<fv>} <fr>E_{<st>} <fv>P_{<st>} <st>T_{<bv>} <st>A_{<na>} <bv>N_{<fr>} <na>S
AKSEPTANGS \longrightarrow A_{<st>} <fv>K_{<f>} <st>S_{<fv>} <fr>E_{<st>} <fv>P_{<st>} <st>T_{<bv>} <st>A_{<na>} <bv>NG_{<fr>} <na>S

Figure 31: Word pronunciations with triphones.

In the ERIS system, if a triphone $\langle L \rangle M \langle R \rangle$ does not exist, the biphones $\langle L \rangle M$ and $M \langle R \rangle$ will be examined. If both of them exist, the biphone with the most number of training sequences will be used instead of the triphone. If neither one of them exist, the mono-phoneme M will be used as a last resort. It is assumed that all mono-phonemes will have been trained.

Given a dictionary of words with their corresponding pronunciations as lists of triphones and a set of trained phoneme HMMs, building the word graph is a fairly straight-forward task. Nodes in the graph can be either consuming or non-consuming, with the latter being called *epsilon* nodes. Epsilon nodes don't really affect the functionality of the graph, since a graph with epsilon nodes always can be converted into a graph without. Consider Figures 5.1 and 5.1. Figure 5.1 illustrates a graph of two phonemes A and B which are connected with epsilon nodes. Figure 5.1 illustrates the same graph but with the epsilon nodes removed. All paths between a pair of two consuming nodes n_1 and n_2 which only contain epsilon nodes are replaced by a direct edge between n_1 and n_2 . As this will typically increase the number of edges dramatically, it should be obvious that epsilon nodes are crucial for organizing the graph and allowing for visual inspection of it – which is valuable during development and debugging.

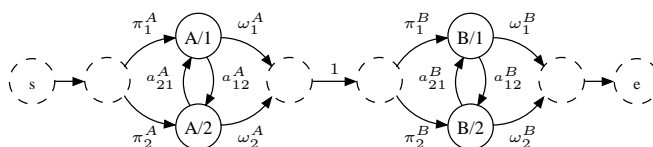


Figure 32: Graph with two phonemes A and B, with interconnecting epsilon nodes.

The word graph is essentially a three-level hierarchy, each of which contains a *start* and an *end* node. The graph itself is a *start* and an *end* node wrapping a list of words. The words, in turn, contains lists of triphones, wrapped by an initial *start* node and a terminating *end* node. Finally, each triphone also has non-consuming *start* and *end* nodes. These wrap a list of consuming nodes, each of which correspond to a state in the phoneme HMM

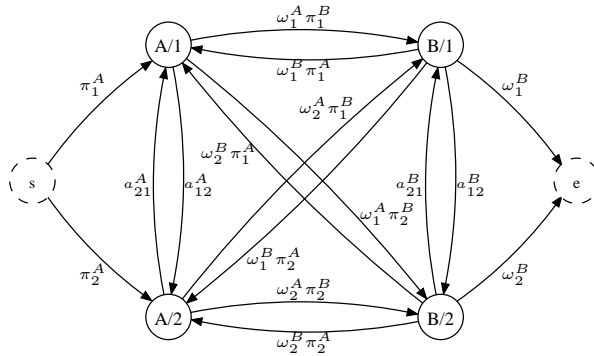


Figure 33: Graph with two phonemes A and B, without interconnecting epsilon nodes.

which was trained from the corpus. Only the sub graphs for the triphones contain consuming nodes. All others are non-consuming. See Figure 5.1 for an illustration of this hierarchy. The *start* and *end* nodes for the whole word graph are marked s and e , for the words they are marked s_w and e_w , and for the triphones they are marked s_p and e_p .

The *start* node for the whole word graph s is special in that it represents the initial state of the system. The *end* node e is also special. It is the only state in which a sequence of observations is allowed to end. That is, a consuming state i only has $\omega_i > 0$ if it has a direct edge – or a path of only epsilon nodes – to the *end* node. Figure 5.1 illustrates this, as only the node $A/3$ has $\omega > 0$. This is so that the whole of the sub graph representing a word has to be traversed in order to match a sequence of observations. In order to be able to recognize an arbitrary number of consecutive words, the *end* node has an edge back to the *start* node.

When attempting to recognize words, it is typically not interesting to know exactly which path through the word graph was the most probable for a particular sequence of observations. Rather, it is of more interest which *words* were on that path – and potentially also which time intervals they correspond to. When allowing the system to recognize several consecutive words – by the *end* node having an edge back to *start* – the *end* nodes for the words are tagged as *collecting* strings representing words. When searching through the graph for the most probable path, only nodes tagged as collecting are reported back. That is, a list of words is returned by the search algorithm, instead of a list of sub-phoneme states. The collecting states are marked by $c(\text{word})$ in Figure 5.1.

Typically, many words in the dictionary share a common prefix. For instance, the words *avbryt* and *avsluta* share the prefix *av*. When searching the graph for the most probable word sequence, it's unnecessary work to explore two different paths with the same prefix. Instead, the graph can be

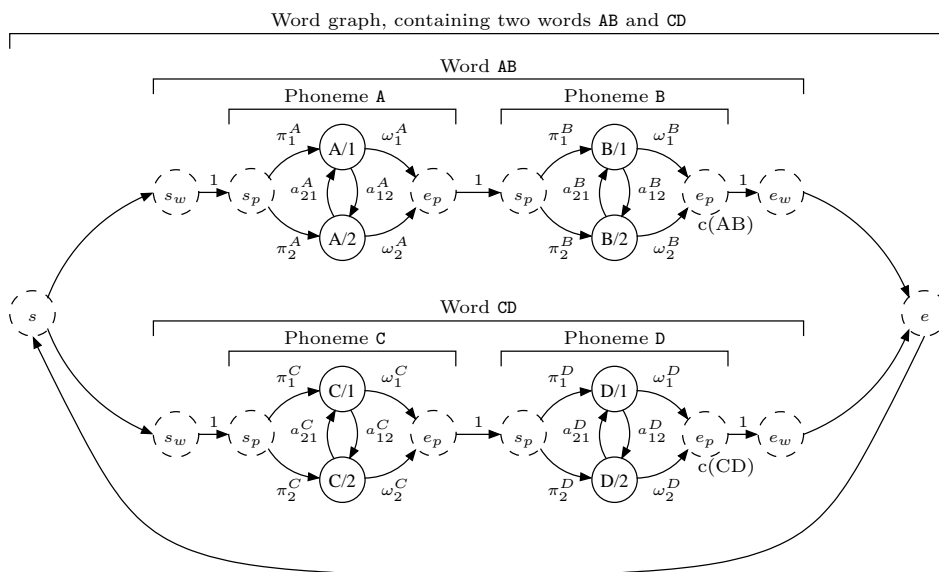


Figure 34: Word graph containing two words, consisting of two phonemes each.

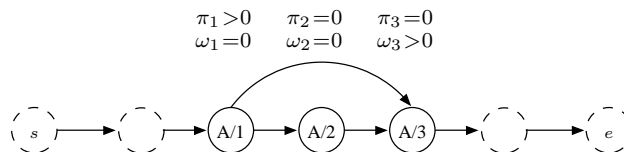


Figure 35: $A/1$ is the start node, and hence $\pi_1 > 0$. $A/3$ is the termination node, with $\omega_3 > 0$.

organized as a prefix-tree, also called a *trie* (Jelinek, 1997), as is illustrated in Figure 5.1.

Building a trie is a straight-forward procedure. For simplicity, consider the words as strings and the phonemes as characters. If given a set of sorted strings¹⁵, building a trie is simply a matter of partitioning the strings so that all strings which start with the same character belong to the same partition. For each partition, a node for the corresponding prefix-character (phoneme) is created and the procedure is repeated recursively for the partitions. See Algorithm 9.

In the ERIS system, the word graphs are stored in files with the suffix `.gx`.

¹⁵The actual order the strings doesn't matter, other than that two strings sharing a prefix need to be ordered next to each other.

Algorithm 9 Algorithm for building prefix tree.

```
function MakeNode( $x$ )  $\mapsto$  Node
    Creates a Node object containing the symbol  $x$ 

function MakeEdge( $a, b$ )
    Creates an edge from node  $a$  to node  $b$ 

function Partition( $str, p, a, b, k$ )
    while  $a \leq b$  and  $strlen(str_a) < k$  do
         $a \leftarrow a + 1$ 
    end while

    if  $a \leq b$  then
         $n \leftarrow$  MakeNode( $str_a[k]$ )
        MakeEdge( $p, n$ )

        for  $i = a + 1$  to  $b$  do
            if  $str_i[k] \neq str_{i-1}[k]$  then
                Partition( $str, n, a, i - 1, k + 1$ )
                 $n \leftarrow$  MakeNode( $str_i[k]$ )
                MakeEdge( $p, n$ )
                 $a \leftarrow i$ 
            end if
        end for

        Partition( $str, n, a, b, k + 1$ )
    end if

 $str = \langle str_1, str_2, \dots, str_N \rangle$ 
 $s \leftarrow$  MakeNode( $\langle start \rangle$ )
Partition(Sort( $str$ ),  $s, 1, N, 1$ )
```

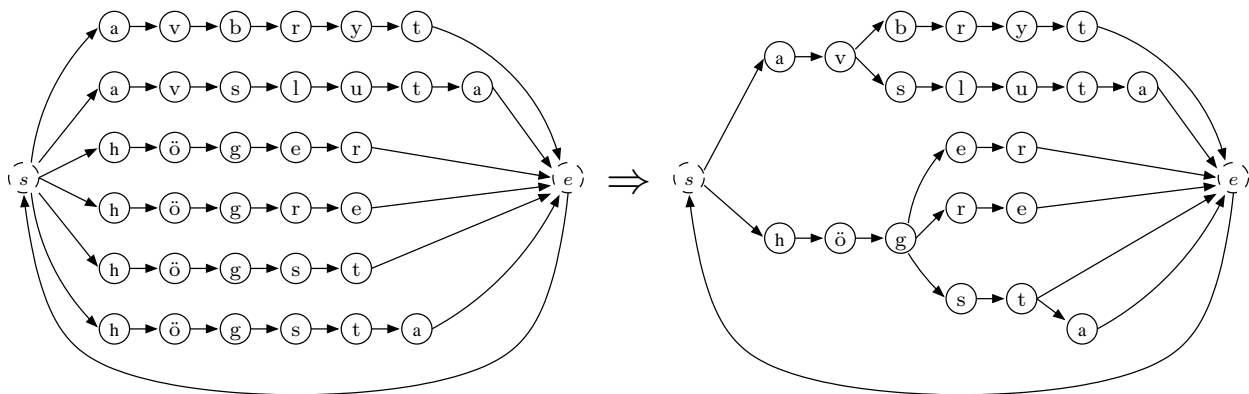


Figure 36: Words sharing the same prefix are merged to form a trie.

5.2 Viterbi Beam-Search

The Viterbi algorithm – described in Section 3.6 – has a time complexity of $O(TN^2)$. For HMM models with relatively few states, such as those representing phonemes, this is computationally cheap enough for an exhaustive search of all possible paths. As the number of states N increases, this may not any longer be the case, as the factor N^2 grows prohibitively large.

Beam search is a heuristic breadth first search that only explores the highest ranking nodes. The number of nodes it explores can either be a fixed number, or it can be a factor of how many nodes there are to explore. For instance, a beam search can ignore every node except the top 500 ones, or it can ignore every node except the top 10% ones. Consider Figure 37, which illustrates a beam search with a beam size of 4 in a tree with a branching factor of 2. In an ordinary breadth first search there would be 2^k number of nodes to explore in the k :th level. With the beam search, however, the number of nodes to explore is explicitly limited to just 4.

The Viterbi beam search algorithm is the ordinary Viterbi algorithm adapted with the beam search heuristic. The main difference, implementation-wise, is analogous to the difference between the dense HMMs for the phonemes and the sparse HMMs for the word graph. In the ordinary Viterbi algorithm, the probabilities and back pointers are stored as matrices, while in the beam search adaptation, they are stored as lists of state and backtrack objects – akin to the explicit node objects in the sparse HMMs. Figure 38 shows a class diagram¹⁶ of the implementation in the ERIS system.

¹⁶The class diagram is slightly simplified for the sake of brevity.

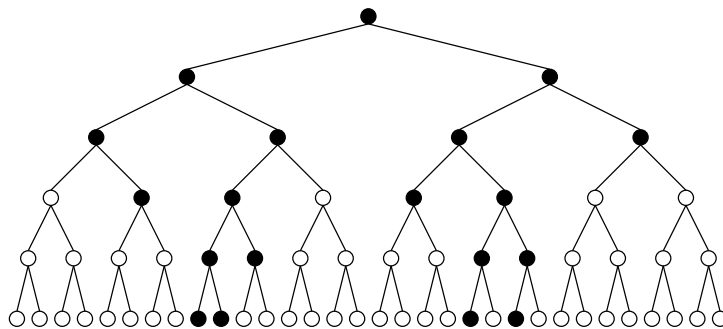


Figure 37: Beam search with beam size 4 on a tree with branching factor 2. Black nodes represent explored nodes and white nodes unexplored nodes.

A Graph object represents a word graph (not shown in the diagram, as it is essentially just a pointer to the start node) and contains Node objects which are connect by Edges. A Node object may or may not have an associated OPDF, depending on whether or not it corresponds to a consuming or non-consuming state in the word graph HMM.

The VBSearcher object represents the state of the beam search at a particular point in time. It is fed observations by calling the add-method. An object interested in words that are recognized implements the Consumer-interface and registers itself with the searcher. When the searcher successfully recognizes a word, it signals all registered consumers by calling the consume-method in the Consumer-interface.

The State object corresponds to a Node and contains the probability of the most probable path up to that node from the start of the observation sequence, as well as a backtrack pointer. Since the the backtrack matrix from the ordinary Viterbi algorithm would be very sparse as well, it also is represented by explicit node objects – BTNode.

The BTNodes make up a tree structure, where, from the point of a single node, it looks like a singly linked list which is traversed in reverse order. Please refer to Figure 39 for the layout of the backtrack nodes and their relation to the state objects. After observing an observation x_t , by having the add-method called, the searcher examines the tree of backtrack nodes. If the root only has a single branch, it means that all different paths which are to be considered start with the same word. If that is the case, the root node is removed, the node its single branch leads to is made to be the new root and its corresponding word is emitted by calling the consume-method on the registered consumers. This is repeated until the root node has more than one branch. At this point, it is no longer certain which the next emitted word should be.

6 Results And Conclusions

6.1 Word Recognition

To test the performance of the system, the 97 words listed in the dictionary in Appendix C were recorded as wave files in five sets by three different speakers. That is, all 97 words were spoken by person *A*, 97 by person *B* and 327 by person *C*, making 485 in total. The words were chosen rather arbitrarily, but with some regard as to which type of words might be expected for giving commands to a smart phone, such as one of the Android family. The words are in Swedish and the native language of all speakers is also Swedish. The full dictionary that was used is listed in Appendix C.

Since the ERIS system does not have a proper silence model, a simple utility program to automatically remove the silence in the beginning and the end of the recorded wave files was implemented. It finds the start of the non-silent audio data by approximating the power of the signal and comparing it to a hard coded threshold. Since all wave files were recorded under similar conditions, the hard coded threshold works equally well for all recordings.

Two different means of recognition were used. The Viterbi beam search algorithm, described in Section 5.2, was used to find the most likely word for a recording. It is possible to get a list of the N most probable words using Viterbi beam search, but due to time constraints, this was not implemented. As an alternative approximation of this list, the forward probability, from Section 3.4, was calculated for each word and sorted in descending order. As the Viterbi algorithm finds the most optimal path and the forward algorithm computes the probability of all possible paths, they do not produce equivalent results. They do, however, give somewhat similar results in practice. This is because the probabilities for a “match” and a “non-match” usually differ by several orders of magnitude, so the contributions of the probabilities of non-optimal paths which is included in the forward probability will typically be negligible.

The configuration file used when running the tests is listed in Appendix B. The most important parameters of which were:

- 25 ms long windows with 10 ms shift
- Cosine window function
- 13 MFCC coefficients with 29 filters
- Level-1 and level-2 MFCC deltas
- Phonemes modelled as left-right HMMs with three states
- 2 Gaussian mixture components
- 3 Baum-Welch iterations

Using the configuration parameters listed above, the forward probabilities were calculated as an approximation of an N best list. 51.8% of the 485 words were recognized correctly, with 85.0% of the words listed as being in the top 10 most probable. The results are summarized in Table 6.1. Since there are 97 *different* words, $1/97 \approx 1.03\%$ correct matches would be expected if the recognizer would simply be making random guesses. Using the Viterbi beam search algorithm, 54.2% of the words were classified correctly with a beam size of 500.

Top N	Percent correct
1	51.8
2	62.9
3	68.9
5	75.4
10	85.0
20	90.4

Table 4: Percentages of words correctly classified as being in the top N most probable.

Using the parameters in the configuration file as summarized above, different values for various parameters were tested, with only one parameter deviating from the template configuration in each test. The parameters tested were:

- Number of Gaussian mixtures
- Number of MFCC coefficients
- MFCC delta levels
- Window function

The number of Gaussian mixtures was varied from 2 to 4, and the results are plotted in Figure 40 with the percentage of correct matches on the y -axis and the beam size for the Viterbi beam search algorithm on the x -axis. Figure 41, 42 and 43 show similar plots, with different values for the number of MFCC coefficients, MFCC delta levels and window functions, respectively. Please note that the y -axis only goes from 45 to 60 percent. For the most part, the correct matches is between 50 and 55% percent. The highest percentage correct matches achieved when varying the four parameters was 57.3%, which was when using 4 Gaussian mixture components.

Not unexpectedly, the results indicate that the more Gaussian mixture components, number of MFCC coefficients and delta levels used, the better. Also, the Hamming window seems to be slightly superior to the other window functions in this particular setup.

6.2 Android Port

As a proof of concept, a simple word recognition algorithm was ported from ERIS to the Android platform. It consists of the signal processing part, as well as an implementation for calculating the forward probability for a sequence of observations and a word graph. The interaction with the Android system is minimal. The ported system reads a `gx` file containing a word graph from the Android file system, as well as a wave file which contains the audio data that should be attempted to be recognized.

The `gx` file is parsed and the word graph is reconstructed. The audio data in the wave file is processed into a sequence of feature vectors, as described in Section 2. Then, for each word in the word graph, the forward probability for the sequence is calculated. The list of words and their corresponding probabilities are sorted in descending order and printed on the screen, so that the top word is the most probable one. A screenshot of the system running in the Android emulator is shown in Figure 44.

The performance of the system has not been evaluated on an actual phone. This is mostly due to the fact there are obvious optimizations which could affect it dramatically. For instance, calculating the MFCC coefficients, as described in Section 2.2, involves a discrete Fourier transform. It so happens that implementing a fast DFT is a rather complex operation, while making a naïve implementation with no regards to performance is fairly trivial. Due to time constraints, such a trivial DFT was implemented, which has the effect of making the signal processing part a big bottle neck. Substituting the naïve FFT with a proper library, such as FFTW (Frigo and Johnson, 2005) as is used in the GNU/Linux version of ERIS, should make for a significant increase in performance.

Another significant optimization would be to use the Viterbi algorithm or the Viterbi beam search algorithm instead of calculating the forward probability for each word.

6.3 Summary

This project consisted of implementing a working speech recognition system for Swedish, using hidden Markov models with mel-frequency cepstral coefficients and multi-variate Gaussian mixtures. The target platform was GNU/Linux and Android.

The audio recordings for individual triphones was extracted from the annotated Waxholm corpus (Bertenstam et al., 1995) and trained as left-right hidden Markov models with three states. The models were trained by using the bees clustering algorithm, and then refined using the Baum-Welch re-estimation algorithm.

Using a dictionary of phonetic pronunciations of words, the smaller HMM models representing the phonemes were then combined into a larger graph

representing all the words in the dictionary, and thus, that the system should be able to recognize.

For evaluation, 97 words were recorded 5 times by 3 different speakers, giving a total of 485 recordings. The Viterbi beam search algorithm was used to find the most probable word for each recording and the forward algorithm was used to approximate an N -best list.

As a proof of concept, a simple recognition system was ported to the Android platform. The port was tested in the Android emulator.

6.4 Evaluation

The performance of the implemented system was evaluated by making 5 different recordings, by 3 different speakers, of the 97 words in the dictionary used (listed in Appendix C.) The wave files with the recorded words were automatically cropped using a small script to find the silence/non-silence boundaries by estimating the power of the audio signal.

The word graph was built from the Waxholm corpus with varying configuration parameters, and the Viterbi beam search algorithm was used to find the most probable word for each recording. The forward algorithm was used to estimate an N -best list.

Of the 485 total number of recordings, the best result obtained was about 57% were correctly classified. 85% were classified as being in the top 10 most probable words in the approximate N -best list.

6.5 Suggestions For Improvements

The ERIS system does not currently have a good silence model, which is needed for continuous speech recognition. A simple silence model could be implemented by inspecting the power of the audio signal, tag the observations as being silent frames and having special OPDFs matching only observations tagged as such.

About 57% of the 97 recorded test files were classified correctly. However, about 85% were classified as being within the top 10 most probable words. The rate of recognition could likely be improved significantly by using a language model, such as a bigram model, so that the probability of a word depends on the word preceding or following it – or both, for a more resource expensive trigram model.

Since many nodes in the word graph share the same OPDF, and thus an OPDF may be called upon to evaluate the probability of an observation more than once, it is likely that caching the result of the probability calculations in each time step in order to avoid repeated calculations might incur some increase in performance.

Similarly, it's plausible that using OPDFs of layered granularity might boost performance, by early exclusion of OPDFs with very low probability.

For instance, if when calculating the probability of an observation, it might first be done by a function which does a fast and coarse estimation using the Euclidean distance between the observation and the means of the OPDF. Only if the coarse probability is greater than some threshold is the fine grained, and slower, multi-variate Gaussian function actually called.

For the Android port, the biggest issue is that the signal processing is far too slow to run in real-time. A proper Fourier transform library would be needed to remedy this. Also, the forward algorithm is used to rank words, instead of a more proper implementation of the Viterbi beam search algorithm.

The interaction with the actual Android environment is quite minimal. Given a dictionary of words which the system should recognize, the system might want to signal recognized words by, for instance, having the ERIS system implemented as a service from which interested parties receive broadcasts.

A Proofs

A.1 Expressing γ in terms of ξ

$$(eq. (43)) \quad \gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^N \alpha_t(i)\beta_t(i)} \quad 1 \leq t \leq T$$

$$(eq. (24)) \quad \beta_t(i) = \left[\sum_{j=1}^N a_{ij} b_j(x_{t+1}) \beta_{t+1}(j) \right] \quad 1 \leq t < T$$

$$\Rightarrow \gamma_t(i) = \frac{\alpha_t(i) \left[\sum_{j=1}^N a_{ij} b_j(x_{t+1}) \beta_{t+1}(j) \right]}{\sum_{i=1}^N \left(\alpha_t(i) \left[\sum_{j=1}^N a_{ij} b_j(x_{t+1}) \beta_{t+1}(j) \right] \right)}$$

$$\Rightarrow \gamma_t(i) = \frac{\sum_{j=1}^N \alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j)}{\sum_{i=1}^N \sum_{j=1}^N \alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j)}$$

$$\Rightarrow \gamma_t(i) = \sum_{j=1}^N \left(\frac{\alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j)}{\sum_{i=1}^N \sum_{j=1}^N \alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j)} \right)$$

$$\Rightarrow \gamma_t(i) = \sum_{j=1}^N \xi_t(i, j) \quad 1 \leq t < T$$

A.2 Proof that $(\bar{A} \mid \bar{w})$ is row stochastic

It shall be proven that each row in the transition probability matrix, plus the termination probability vector, is stochastic. That is,

$$\sum_{j=1}^N (\bar{a}_{ij} + \bar{w}_i) = 1 \quad 1 \leq i \leq N$$

$$\begin{aligned}
\sum_{j=1}^N (\bar{a}_{ij} + \bar{\omega}_i) &= \left(\sum_{j=1}^N \bar{a}_{ij} \right) + \bar{\omega}_i \\
&= \left(\frac{\overbrace{\sum_{j=1}^N \sum_{k=1}^K \sum_{t=1}^{T^{k-1}} \xi_t^k(i, j)}^{\bar{a}_{ij}, \text{ eq. 52}}}{\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i)} \right) + \frac{\overbrace{\sum_{k=1}^K \gamma_{T^k}^k(i)}^{\bar{\omega}_i, \text{ eq. 53}}}{\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i)} \\
&= \frac{\overbrace{\sum_{k=1}^K \sum_{t=1}^{T^{k-1}} \sum_{j=1}^N \xi_t^k(i, j)}^{\gamma_t^k(i), \text{ eq. 46}}}{\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i)} + \frac{\sum_{k=1}^K \gamma_{T^k}^k(i)}{\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i)} \\
&= \frac{\sum_{k=1}^K \sum_{t=1}^{T^{k-1}} \gamma_t^k(i)}{\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i)} + \frac{\sum_{k=1}^K \gamma_{T^k}^k(i)}{\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i)} \\
&= \frac{\left(\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i) \right) - \left(\sum_{k=1}^K \gamma_{T^k}^k(i) \right)}{\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i)} + \frac{\sum_{k=1}^K \gamma_{T^k}^k(i)}{\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i)} \\
&= \frac{\overbrace{\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i)}^{= 1}}{\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i)} - \frac{\overbrace{\sum_{k=1}^K \gamma_{T^k}^k(i)}^{= 0}}{\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i)} + \frac{\sum_{k=1}^K \gamma_{T^k}^k(i)}{\sum_{k=1}^K \sum_{t=1}^{T^k} \gamma_t^k(i)} = 1 \quad \square
\end{aligned}$$

A.3 Proof that $\hat{\gamma} = \gamma$

$\gamma_t(i)$ is the probability of being in state i at time t . Recall Equation 43:

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^N \alpha_t(i)\beta_t(i)} \quad 1 \leq t \leq T \quad (43)$$

It shall be proven that substituting the forward and backward probabilities, α and β , with the scaled versions, $\hat{\alpha}$ and $\hat{\beta}$, yields an equivalent equation.

First, it should be noted that

$$\begin{aligned} \hat{\alpha}_t(i)\hat{\beta}_t(i) &= C_t\alpha_t(i) D_t\beta_t(i) \\ &= \left(\prod_{s=1}^t c_s\right) \alpha_t(i) \left(\prod_{s=t}^T c_s\right) \beta_t(i) \\ &= c_t \left(\prod_{s=1}^T c_s\right) \alpha_t(i) \beta_t(i) \\ &= c_t C_t \alpha_t(i) \beta_t(i) \end{aligned}$$

Next, α and β are substituted with $\hat{\alpha}$ and $\hat{\beta}$ in Equation 43

$$\begin{aligned} \hat{\gamma}_t(i) &= \frac{\hat{\alpha}_t(i)\hat{\beta}_t(i)}{\sum_{i=1}^N \hat{\alpha}_t(i)\hat{\beta}_t(i)} \\ &= \frac{c_t C_t \alpha_t(i)\beta_t(i)}{\sum_{i=1}^N c_t C_t \alpha_t(i)\beta_t(i)} \\ &= \frac{c_t C_t \alpha_t(i)\beta_t(i)}{c_t C_t \sum_{i=1}^N \alpha_t(i)\beta_t(i)} \\ &= \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^N \alpha_t(i)\beta_t(i)} = \gamma_t(i) \quad \square \end{aligned}$$

The scaling factors cancel themselves out completely, so the scaled terms can be directly used in Equation (43) without any further modifications.

A.4 Proof that $\hat{\xi} = \xi$

$\xi_t(i, j)$ is the probability of taking the transition from state i to j at time t . It shall be proven that the Equation 45 for ξ still holds if the forward and backward probabilities, α and β , are substituted by the scaled versions, $\hat{\alpha}$ and $\hat{\beta}$.

$$\begin{aligned}
\hat{\xi}_t(i, j) &= \frac{\hat{\alpha}_t(i) a_{ij} b_j(x_{t+1}) \hat{\beta}_{t+1}(j)}{\sum_{i=1}^N \sum_{j=1}^N \hat{\alpha}_t(i) a_{ij} b_j(x_{t+1}) \hat{\beta}_{t+1}(j)} \\
&= \frac{C_t \alpha_t(i) a_{ij} b_j(x_{t+1}) D_{t+1} \beta_{t+1}(j)}{\sum_{i=1}^N \sum_{j=1}^N C_t \alpha_t(i) a_{ij} b_j(x_{t+1}) D_{t+1} \beta_{t+1}(j)} \\
&= \frac{C_t D_{t+1} \alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j)}{C_t D_{t+1} \sum_{i=1}^N \sum_{j=1}^N \alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j)} \\
&= \frac{\alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j)}{\sum_{i=1}^N \sum_{j=1}^N \alpha_t(i) a_{ij} b_j(x_{t+1}) \beta_{t+1}(j)} = \xi_t(i, j) \quad \square
\end{aligned}$$

Thus, the scaling factors for ξ are cancelled out in the same fashion as for γ . $\hat{\alpha}$ and $\hat{\beta}$ can directly be used in Equation 45 without any other modifications.

B Sample configuration file

```
math {
  random_seed = 326
}

efx {
  type           = mfcc
  delta1        = (2,-2)
  delta2        = (1,-1)

  train_ratio   = 0.7

  min_seq_len   = 7
  min_num_train_seq = 20
  min_num_test_seq = 3

  window { function = cosine ; length_ms = 25 ; shift_ms = 10 ; min_filled = 0.8 }
  mfcc { num_coeff = 13 ; num_filters = 29 }
}

gx {
  silence {
    singleword {
      at_start      = false
      at_end        = false
      free_transitions = false
      optional      = true ## Add null-arc around the silence
    }

    multiword {
      at_start      = false
      at_end        = false
      free_transitions = false
      optional      = true
    }
  }

  use_triphones   = true
  min_num_seq     = 10
}

hmm {
  num_states      = 3
  num_mixtures    = 2
  use_covariance  = false
  statemodel     = left-right
  train_order     = bees, baumwelch
}
```

```
opdf {
  probf_floor = 1e-100
  gaussian {
    var_floor = 1e-4
    var_default = 1e-1
  }
}

cluster {
  uniform_weight = 0.1
  max_iterations = 120

  bees {
    num_scouts = 12
    num_sites = 4
    num_elite = 2
    num_bees_elite = 4
    num_bees_rest = 2
    num_iterations = 300
  }
}

baumwelch {
  num_iterations = 3
}
}
```

C Dictionary file

ja	-> JA:
nej	-> NEJ
avbryt	-> A:VBRY:T
avsluta	-> A:VSLU:TA
och	-> (Å: ÅK)
eller	-> EL(E EO)R
inte	-> INT(E EO)
ifall	-> IFAL
om	-> ÅM
ring	-> RING
skicka	-> SJIKA
spela	-> SPE:LA
spela upp	-> SPE:LA UP
spela in	-> SPE:LA IN
stäng	-> ST(E EO)N
öppna	-> ÖPNA
nästa	-> NESTA
förra	-> FÖ4RA
föregående	-> FÖ3REGÅ:(E EO)ND(E EO)
sätt på	-> S(E EO)T PÅ:
sätt in	-> S(E EO)T IN
sätt in i	-> S(E EO)T IN I:
stäng av	-> ST(E EO)N A:V
stopp	-> STÅP
pausa	-> PAUS
sluta	-> SLU:TA
ta bild	-> TA BILD
ta foto	-> TA FOTO
börja	-> BÖ4RJA
svara	-> SVA:RA
lägg på	-> LEG PÅ:
tyst	-> TYST
normal	-> NÅRMA:L
vanlig	-> VA:NLIG
vanliga	-> VA:NLIGA
profil	-> PROFIL
signal	-> SINGNA:L
klocka	-> KLÅKA
klockan	-> KLÅKAN
måndag	-> MÅND(A A:G)
tisdag	-> TI:SD(A A:G)
onsdag	-> ONSDA:G
torsdag	-> TO:2SD(A A:G)
fredag	-> FRE:D(A A:G)
lördag	-> LÖ32D(A A:G)
söndag	-> SÖND(A A:G)
vecka	-> VEKA
månad	-> MÅ:NAD
kalender	-> KAL(E EO)ND(E EO)R
plus	-> PLUS
minus	-> MI:NUS
gång	-> GÅNG(E EO)R

minst	-> MINST
som minst	-> SÅM MINST
mest	-> MEST
som mest	-> SÅM MEST
mindre	-> MINDR(E EO)
mera	-> ME:R(A)?
starkare	-> STARKAR(E EO)
svag	-> SVA:GAR(E EO)
högre	-> HÖ:GR(E EO)
lägre	-> LÄ:GR(E EO)
lägst	-> LÄ:GST
lägsta	-> LÄ:GST
högst	-> HÖ(G K)ST
högsta	-> HÖ(G K)STA:
vänster	-> V(E EO)NST(E EO)R
höger	-> HÖ:G(E EO)R
upp	-> UP
ner	-> NE:R
knapplås	-> KNAPPLÅ:S
knapplås på	-> KNAPPLÅ:S PÅ:
knapplås av	-> KNAPPLÅ:S A:V
lås upp	-> LÅ:S UP
tillbaka	-> TILBA:KA
bakåt	-> BA:KÅ:T
framåt	-> FRAMÅT
sök	-> SÖ:K
klicka	-> KLIK(A)?
dölj	-> DÖLJ
visa	-> VI:SA
meny	-> M(E EO)NY:
huvudmeny	-> HUVUDM(E EO)NY:
huvudmenyn	-> HUVUDM(E EO)NY:N
väckarklocka	-> VEKARKLÅKA
lägg till	-> LEG TIL
ta bort	-> TA: BÅ2T
webb	-> VEB
webbläsare	-> VEB LÄ:SAR(E EO)
ljud	-> JU:D
ljud på	-> JU:D PÅ:
ljud av	-> JU:D A:V
ljudlös	-> LÖ:S
räknare	-> RÄ:KNAR(E EO)
miniräknare	-> MI:NI RÄ:KNAR(E EO)
inställning	-> INSTELNING
inställningar	-> INSTELNINGAR

References

- Bertenstam, J., Blomberg, M., Carlson, R., Elenius, K., Granström, B., Gustafson, J., Hunnicutt, S., Högberg, J., Lindell, R., Neovius, L., de Serpa-Leitao, A., Nord, L., and Ström, N. (1995). Spoken dialogue data collection in the waxholm project.
- Frijo, M. and Johnson, S. G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231. Special issue on “Program Generation, Optimization, and Platform Adaptation”.
- Huang, X., Acero, A., and Hon, H.-W. (2001). *Spoken Language Processing. A Guide to Theory, Algorithm, and System Development*. Prentice Hall, Upper Saddle River.
- Jelinek, F. (1997). *Statistical Methods for Speech Recognition*. MIT Press, Cambridge, Massachusetts.
- Juang, B. and Rabiner, L. R. (1990). The segmental k-means algorithm for estimating parameters of hidden markov models. *IEEE Transactions On Acoustics, Speech, and Signal Processing*, 38(9):1639–1641.
- Pham, D., S. Otri, A. A., Mahmuddin, M., and Al-Jabbouli, H. (2007). Data clustering using the bees algorithm. <http://www.bees-algorithm.com/modules/2/6.pdf>.
- Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286.
- Ursin, M. (2002). Triphone clustering in finnish continuous speech recognition.

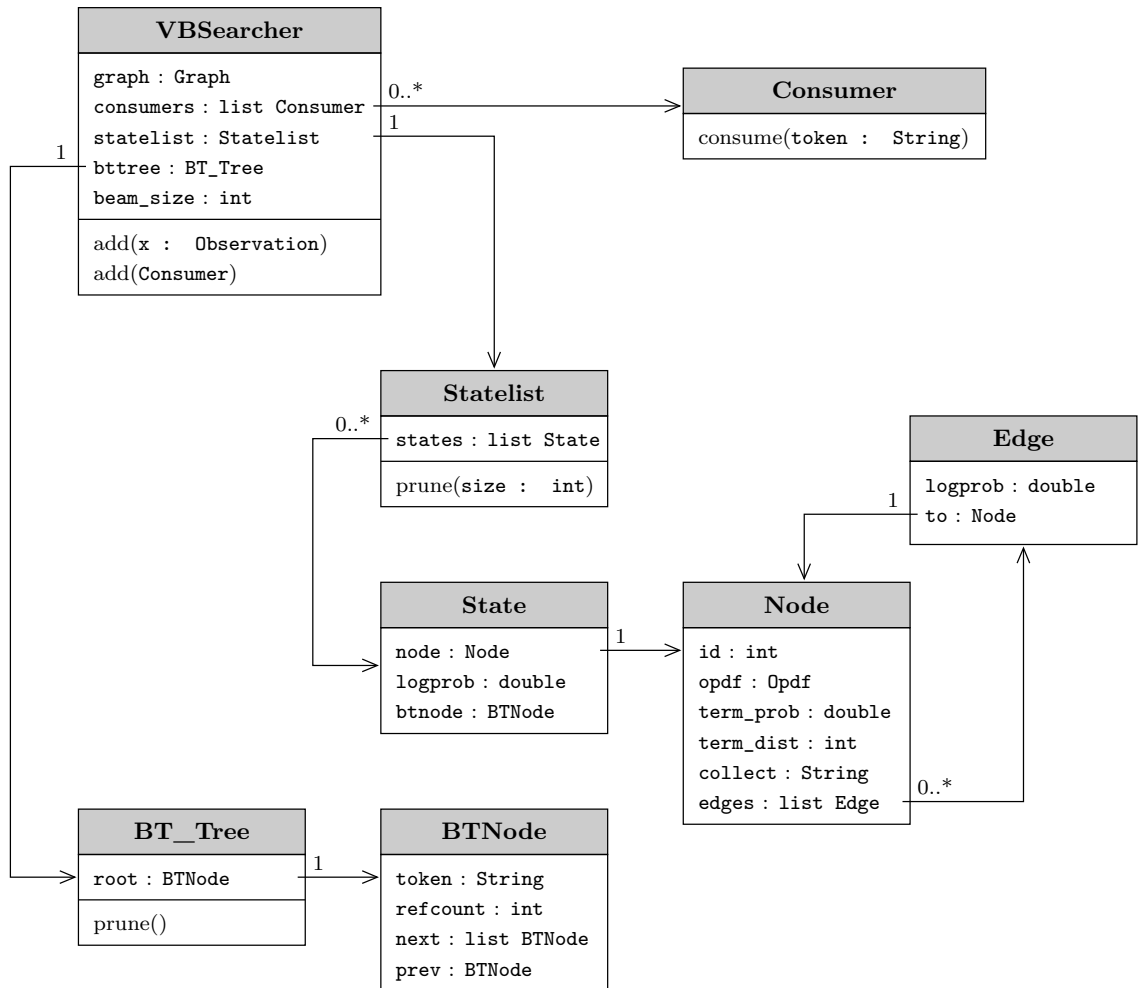


Figure 38: Class diagram of the Viterbi beam search implementation in ERIS.

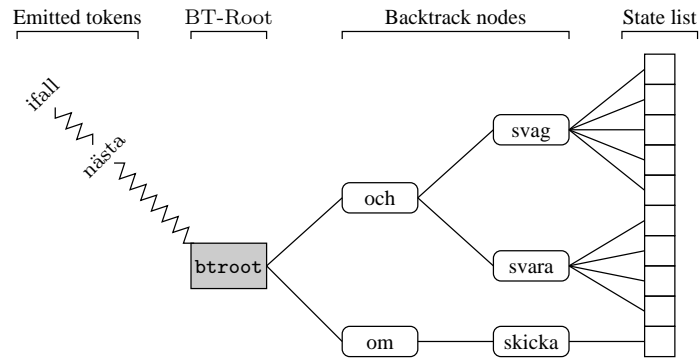


Figure 39: Organization of the backtrack nodes and their relation to the state objects.

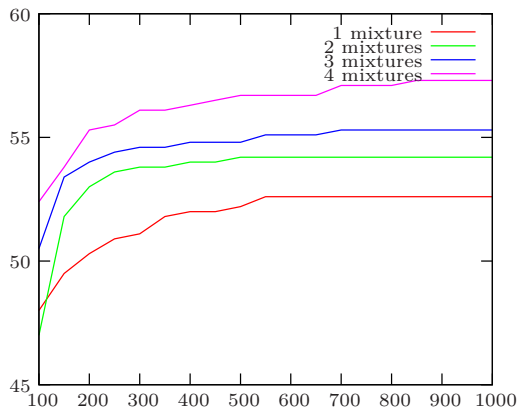


Figure 40: Percent correct matches for different number of mixtures

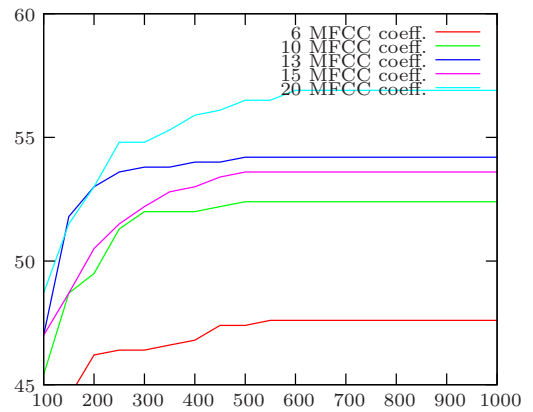


Figure 41: Percent correct matches for different number of MFCC coefficients

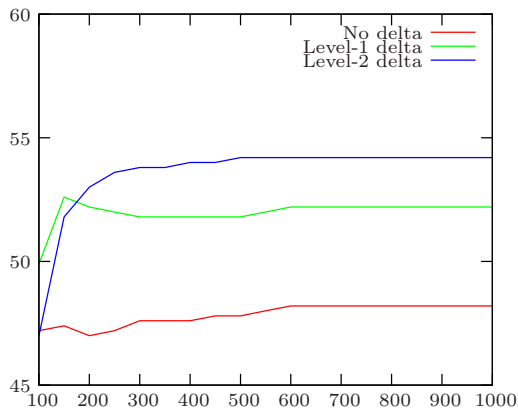


Figure 42: Percent correct matches for different delta levels

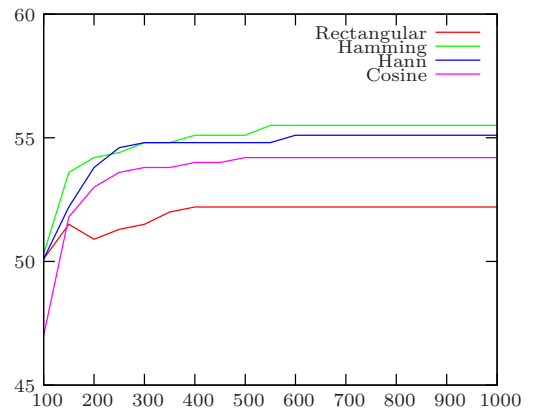


Figure 43: Percent correct matches for different window functions

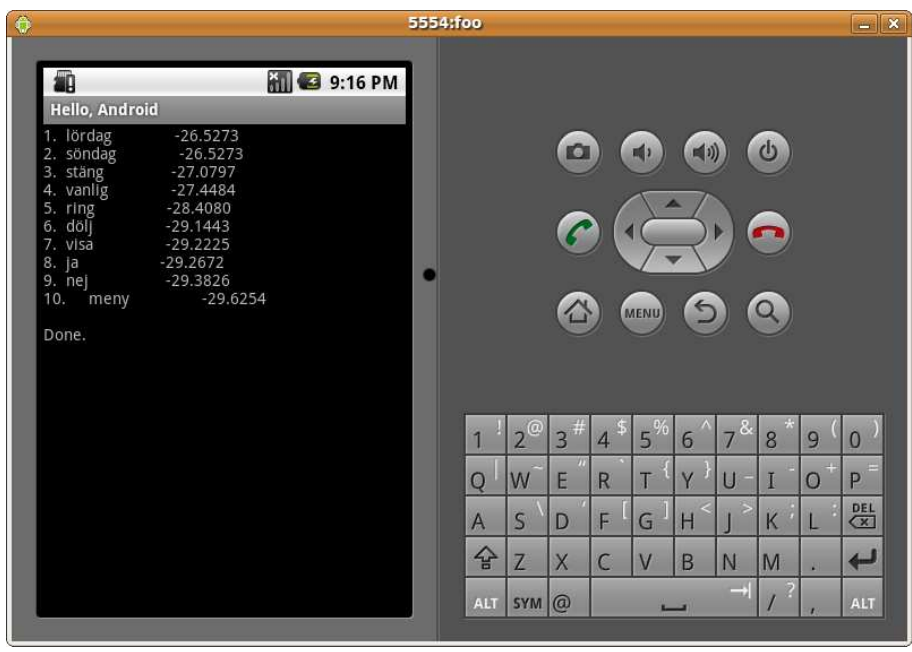


Figure 44: Screenshot of Android port running in the emulator.