47)

Speech Recognition by Clustering Wavelet and PLP Coefficients

by

Chiangkai Er

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997

Author .....
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by .....
Kenneth Yip
Visiting Assistant Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Speech Recognition by Clustering Wavelet and PLP Coefficients

by

Chiangkai Er

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 1997, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis explores the use of K-means clustering, wavelets and perceptual linear predictive (PLP) analysis for speech recognition. First, we would like to compare the performance of this method to those of previous speech recognition techniques. Next, we want to test if both wavelet and PLP coefficients are important to the analysis. Finally, we want to try out various means of improving the clustering method.

The task of phonetic classification is used as the basis of comparison. Three different sets of phonemes are chosen from the TIMIT database: 16 vowels, 24 consonants and 39 phonetic classes. Coefficients of the Haar wavelet transform and the 5th order PLP analysis are combined to form a 42-dimensional vector for each phoneme. Clusters of phoneme vectors obtained by K-means clustering are then used to classify test vectors. Classification experiments using the NIST train and test sets show that independent clustering of phonemes with proportional phoneme emphasis is the best clustering strategy. It yields an accuracy of 55.4% for vowel classification, 54.6% for consonant classification and 50.9% for phoneme classification. Tests with wavelet-only vectors and PLP-only vectors show that both the wavelet transform and PLP analysis are significantly important to the phonetic classifier. Results also show that wavelet coefficients are useful for detecting sound transitions, which are abundant in consonants.

Thesis Supervisor: Kenneth Yip
Title: Visiting Assistant Professor

Acknowledgments

First and foremost, I would like to thank my thesis supervisor, Professor Kenneth Yip, for his guidance and support over the past year. His constant encouragement and stimulating advice were most crucial to the successful completion of this thesis.

I am grateful to Dr. James Glass and Andrew Halberstadt of the Spoken Language Systems Group for the help and invaluable advice they have given me.

Special thanks to Rosanna Alegado, Weiyang Cheong, Tracey Ho, Alarice Huang, Steven Lee, Geoff Lee Seyon, Ben Leong, Cheewe Ng, Xuanhui Ng, Kokkeong Puah, Ricci Rivera, Michael Sy, Choonsiang Tan, Alice Wang, Evelyn Wang, Tammy Yap and Tseh-Hwan Yong for making my experience at MIT so enjoyable and memorable.

Finally, I would like to thank my parents and brother for their endless care and unfailing support throughout my years of study at MIT. They have given me the ambition and strength to succeed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement and Motivation

There are many forms of communication among humans, like the use of body gestures, drawing, writing and voice. Among all these, voice communication is certainly the one that is most commonly used. That is why extensive research in speech communication by machine has been done since the 1950s. One important aspect of speech communication is speech recognition, which is the task of extracting the message information from a voice signal so as to respond to the spoken commands. Applications of speech recognition are numerous. It can be used to automate the task of telephone operators in directory services, car rental companies and ticketing booths. Interactive toys, "hands-free" computers and automatic language translators can also be designed based on speech recognition technology.

In spite of the numerous theories that were developed for speech recognition, we are still a long way from science fiction wonders such as the HAL computer in Stanley Kubrick's *2001 - A Space Odyssey* and the talking robot R2D2 in George Lucas's *Star Wars*. It is not clear what is intrinsically wrong with our theories, but researchers have been constantly trying out new ideas in the hope of hitting upon an idea that works well. Research in artificial intelligence (AI) has shown that many hard problems for humans (e.g. mathematical manipulation, problem optimization, etc.) can be easy for machines. However, perceptual tasks like speech and vision, which seem so easy

for humans are extremely difficult for machines. These perceptual tasks appear to have many things in common. For example, they both require pattern matching and signal processing. Thus, it is very likely that huge advances in speech recognition may lead to similar improvements in machine vision, which will be a great success for artificial intelligence research.

Signal processing is an important aspect of speech recognition because a good signal representation is essential for efficient and accurate recognition. As a signal processing tool, wavelets have been used in quantum physics, electrical engineering and seismic geology for many decades. It is only quite recently that wavelet theory has been applied to new applications like radar, machine vision and image compression. However, very little research has been done in applying wavelets to speech recognition. Unlike Fourier analysis, which has constant time and frequency resolutions, wavelet analysis has higher time resolution and lower frequency resolution at high frequencies. In this sense, wavelet analysis models our auditory perception more closely because we are less sensitive to frequency fluctuations at high frequencies. That is why music is naturally distributed into octaves. To the human auditory system, frequency resolution is inversely proportional to the frequency.

Another signal processing tool that models our auditory perception closely is the perceptual linear predictive (PLP) analysis. As the predecessor of PLP analysis, the linear predictive coding (LPC) technique has been used by the speech recognition community for a long time. LPC models the vocal tract as an all-pole model and it approximates the speech waveform equally well at all frequencies of the analysis band. This property is, however, inconsistent with human hearing, which is more sensitive in certain frequency ranges. PLP analysis was developed as an attempt to correct this inconsistency. It models auditory perception by making use of the critical-band curve, equal-loudness curve and intensity-loudness power law.

Since both wavelet and PLP analyses are well suited for speech recognition, it will be interesting to investigate how well they can be put together to perform speech recognition. One way to do this is to concatenate the wavelet and PLP coefficients as a vector and treat it as a vector quantization of the speech signal. We identify

clusters of these vectors in the hope that vectors in the same cluster would sound the same to the human ear. With vectors of high dimensions, the vector space might be sparse enough for clustering to effectively separate vectors representing different sounds. Clustering can be performed using the K-means clustering algorithm, which is the algorithm to be employed in this thesis.

The objective of this thesis is to investigate how well we can perform speech recognition by applying K-means clustering on wavelet and PLP coefficients. In order for comparisons to be easily made, the method is applied to the task of phoneme classification using a standard speech database.

1.2 Methodology and Results

Phonetic classification is the speech recognition task chosen for this thesis. Three different sets of phonemes are chosen from the TIMIT speech database. For vowel classification, 16 standard American English vowels are picked. Consonant classification deals with 24 consonants and general phoneme classification is done on the CMU/MIT standard 39 phonetic classes. Coefficients of the Haar wavelet transform and the 5th order PLP analysis are combined to form a 42-dimensional vector for each phoneme. Various algorithms based on K-means clustering are then performed on the phoneme vectors. Finally, we apply the k-nearest neighbor method on the cluster centroids when classifying test vectors.

Classification experiments using the NIST train and NIST core test sets show that independent clustering of phonemes with proportional phoneme emphasis is the best clustering strategy. It yields an accuracy of 55.4% for vowel classification, 54.6% for consonant classification and 50.9% for phoneme classification. Tests with wavelet-only vectors and PLP-only vectors show that both the wavelet transform and PLP analysis are significantly important to the phonetic classifier. The results also show that wavelet coefficients are useful for detecting sound transitions, which are abundant in consonants.

The classification accuracies are lower than those obtained by previous experi-

ments, but they do not indicate that wavelets are not suitable for speech recognition. There are several factors which might have contributed to the poor performance. First, clustering might not be a suitable method for phonetic classification. Another reason is that some wavelet information is lost through signal processing. The wavelet phase information, which might be important for speech recognition, is ignored. Also, information is lost when we perform averages on the coefficients.

1.3 Thesis Overview

In this thesis, we investigate how well we can perform speech recognition by applying K-means clustering on wavelet and PLP coefficients. We would like to answer several questions at the end of the experiments. The first and most important question is how well does the "Clustering-Wavelets-PLP" combination perform in comparison with previous speech recognition techniques. Next, we would like to know if there is any significant contribution of wavelets to the analysis. Similarly, we would also like to know if the PLP coefficients are important. Finally, we want to try out various means of improving the performance of K-means clustering.

In order to be able to compare with previous work, the task of phonetic classification is picked as the basis of comparison. Chapter 2 describes phonetic classification in detail. Vowel, consonant and general phoneme classifications are done so that multiple comparisons can be made. The speech corpus used is the TIMIT speech database, which is a standard database used by the speech recognition community.

Next, in chapter 3, the details of the speech processing are described. This includes the wavelet transform and PLP analysis. The coefficients in each phoneme are averaged and concatenated to form a vector, which is then normalized to achieve best results during clustering. The distance metric is also defined here.

With the vector space defined, we are ready to perform clustering in chapter 4. Various strategies of clustering are described in this chapter. They differ from the original K-means algorithm in how the initial seeds are selected and whether all the training vectors are clustered at the same time.

After clustering, the classifier is trained by collecting phoneme statistics for each cluster. Chapter 5 describes the training phase in detail, followed by how the statistics are used to classify test vectors. The k-nearest neighbor rule is introduced here to enhance performance.

Following this, chapter 6 presents the classification results obtained. Experiments are done to test the various clustering algorithms introduced in chapter 4. In order to test the significance of the wavelet and PLP coefficients in the analysis, classification experiments are performed with only one set of coefficients at a time.

The final chapter presents a summary of the thesis as well as conclusions that can be inferred from the experimental results. Possible future extensions of this thesis are also discussed.

Chapter 2

Phonetic Classification

Much work has been done in phonetic classification using various methods such as multi-layer perceptrons (MLP) [13] and hidden Markov models (HMM) [12], with signal representations such as linear predictive coding (LPC) [12] and mel-frequency cepstral coefficients (MFCC) [14]. In view of this, we pick the task of phonetic classification for our experiments. Vowel, consonant and general phoneme classifications are done so that multiple comparisons can be made. To ensure that valid comparisons can be made, we use the TIMIT speech corpus, which is a standard database used by the speech recognition community.

2.1 Task Description

In phonetic recognition, we are given speech utterances in the form of waveforms and the task is to state the identities of the phonemes together with their start and end times. Phonetic classification, however, does not require that the locations of the phonemes be found. Instead, phonetic classification systems are given the boundaries of phonemes in the utterance. The task is simply to state the identity of each phoneme.

Experiments done in this thesis are all context-independent. By this, we mean that each phoneme is presented to the phonetic classifier independently, without any information of what sounds are before and after it. It is important to state such a

Table 2.1: The 16 vowels used for vowel classification

Vowel	TIMIT label	Example	Vowel	TIMIT label	Example
i	iy	beat	ɔ	ao	bought
I	ih	bit	u	uw	boot
e	ey	bait	ʊ	uh	book
ɛ	eh	bet	ü	ux	Tuesday
æ	ae	bat	ɝ	er	bird
ɑ	aa	cot	$ɑ^y$	ay	bite
o	ow	boat	$ɔ^y$	oy	boy
ʌ	ah	butt	$ɑ^w$	aw	about

criterion because context-dependent classifiers usually perform better than context-independent ones. Knowing what sounds preceed and follow a speech segment to be identified allows us to rule out certain choices of phonemes and increases the probabilities of other phonemes.

Table 2.1 shows the 16 vowels that are used for the vowel classification experiments. These are the vowels in American English and they match with those used in previous experiments. The 24 consonants picked for consonant classification are presented in table 2.2. Finally, general phoneme classification is performed on the 39 Kai-Fu Lee classes [12] listed in table 2.3. There are a total of 64 possible phonetic labels in the TIMIT database, but many of them are folded into the same class so that allophones and phonemes within the same group (e.g. /ʃ/ and /ʒ/) do not cause confusion for the classifier. This folding conforms to CMU/MIT standards.

2.2 Previous Work

Studies of vowel classification by humans was done by Cole [3]. Sixteen subjects participated in 10 one-hour sessions over a two-week period. They were told to classify 16 different vowels, namely /i, I, e, ɛ, æ, ɝ, ʌ, ə, ɑ, ɔ, ʊ, u, o, $ɑ^w$, $ɑ^y$, $ɔ^y$/. There were 168 tokens of each vowel extracted from the TIMIT database, making a total of 2688 tokens. The subjects were able to identify the vowels with an accuracy of 54.8% when they were presented with context-independent vowel sounds. However, when presented with one segment of context, they achieved an accuracy of 65.9%. The

Table 2.2: The 24 consonants used for consonant classification

Consonant	TIMIT label	Example	Consonant	TIMIT label	Example
p	p	pop	**ʃ**	sh	shoe
b	b	bob	**ʒ**	zh	measure
t	t	tot	**h**	hh, hv	hay
d	d	dad	**tʃ**	ch	church
k	k	kick	**dʒ**	jh	judge
g	g	gag	**m**	m	mom
f	f	five	**n**	n	non
v	v	very	**ŋ**	ng	sing
θ	th	thief	**w**	w	wet
ð	dh	they	**l**	l	led
s	s	six	**r**	r	red
z	z	zoo	**j**	y	yet

Table 2.3: The 39 Kai-Fu Lee classes

Class	TIMIT Label	Example	Class	TIMIT Label	Example
1	iy	beat	21	ng, eng	sing
2	ih, ix	bit	22	ch	church
3	eh	bet	23	jh	judge
4	ae	bat	24	dh	they
5	ah, ax	butt	25	b	bob
6	uw, ux	boot	26	d	dad
7	uh	book	27	dx	butter
8	aa, ao	cot	28	g	gag
9	ey	bait	29	p	pop
10	ay	bite	30	t	tot
11	oy	boy	31	k	kick
12	aw	about	32	z	zoo
13	ow	boat	33	sh, zh	shoe
14	l, el	led	34	v	very
15	r	red	35	f	fief
16	y	yet	36	th	thief
17	w	wet	37	s	sis
18	er, axr	bird	38	hh, hv	hay
19	m, em	mom	39	cl, epi, sil, pau	(silence or closure)
20	n, nx, en	non			

results should not be treated as upper-bounds for machine vowel classification because phonetic classification (instead of word recognition) is regarded as an unnatural task for humans.

Numerous experiments on vowel classification by machine have been done using the set of 16 vowels in table 2.1. Using auditory models, Leung [13] achieved a vowel classification accuracy of 64%. Meng [19] reported 65.6% using a similar approach. By using analysis-by-synthesis and incorporating speaker normalization information from the TIMIT *sa* sentences, Carlson and Glass [1] achieved 68.7% accuracy. Goldenthal [7] reported 68.9% with gender-specific models while Chun [2] obtained 67.6% using speaker-independent models. All of them made use of the TIMIT speech corpus.

Using the phonetic classes listed in table 2.3, Lee and Hon [12] performed phonetic recognition with hidden Markov models (HMM) and achieved an accuracy of 64.07% in context-independent experiments. With the right context-dependent phone models, the recognition rate improved to 73.80%. Pepper and Clements [20] performed recognition on the same phonetic classes and reported an accuracy of 53.3% in a speaker-independent experiment. They used a large hidden Markov model in their experiments. For phonetic classification using the same set of phoneme classes, Zahorian [28] reported 77.0% on the NIST CORE TEST set by using a binary-pair partitioned (BPP) neural network system. Leung [14] obtained 78.0% by performing classification with an MLP on a different test set. Using heterogeneous acoustic measurements, Halberstadt [8] achieved an accuracy of 79.0%. Table 2.4 summarizes the classification accuracies achieved.

2.3 Corpus Description

The TIMIT (TI-MIT) speech corpus [29] was developed to train and test speaker-independent speech recognition systems. It was recorded using a Sennheiser close-talking microphone at 16 kHz with a signal-to-noise ratio (SNR) of 42 dB. There are 630 native American English speakers from 8 dialectal regions of the United States, each speaking 10 sentences of the following types:

Table 2.4: Classification accuracies (C: classification, R: recognition)

Phonemes	Accuracy	Techniques used	Reference
16 vowels (C)	68.9%	gender-specific models	[7]
16 vowels (C)	68.7%	analysis-by-synthesis	[1]
16 vowels (C)	67.6%	speaker-independent models	[2]
16 vowels (C)	65.6%	auditory model	[19]
16 vowels (C)	64%	auditory model	[13]
39 classes (C)	79.0%	heterogeneous acoustic measurements	[8]
39 classes (C)	78.0%	MLP	[14]
39 classes (C)	77.0%	BPP neural network	[28]
39 classes (R)	64.07%	HMM	[12]
39 classes (R)	53.3%	large HMM	[20]

Table 2.5: The NIST train, development and test sets

Set	No. of Speakers	No. of Utterances	No. of Tokens
NIST TRAIN	462	3696 (*sx*, *si*)	142910
NIST DEV	50	400 (*sx*, *si*)	15334
NIST CORE TEST	24	192 (*sx*, *si*)	7333

- 2 "*sa*" calibration sentences used for dialectical studies of American English which are the same across all speakers;
- 5 "*sx*" sentences from a list of 450 phonetically-compact sentences hand-designed at MIT with emphasis on thorough coverage of phonetic pairs;
- 3 "*si*" sentences which were randomly selected from the Brown corpus by TI.

The train, development and test sets used in this thesis are the NIST TRAIN, NIST DEV and NIST CORE TEST sets respectively. Table 2.5 shows some statistics of the sets. The list of speakers in each of the sets are presented in appendix D.

The database has been divided into suggested train and test sets using the following criteria:

- About 70% to 80% of the corpus is used for training, leaving the rest for testing.
- No speaker should appear in both the training and testing data.
- All the dialect regions should be represented by both male and female speakers in both the training and testing data.

- There should be minimal overlap of text material in the training and testing data.
- Both the train and test sets should cover all phonemes, with each phoneme occurring multiple times in different contexts.

The NIST CORE TEST set was constructed using the above criteria, with 2 male speakers and 1 female speaker from each dialect, providing a set of 24 speakers. Each speaker read a different set of 5 *sx* sentences. Since each *si* sentence was read by only one speaker, these texts did not impose constraints in selecting the texts or speakers. The selected texts were also checked to ensure that each phoneme appeared at least once.

2.4 Chapter Summary

In this chapter, the task of phonetic classification was described together with the phonemes that are to be used in this thesis. Results from previous work on phonetic classification were presented too. These will be used for comparisons. The NIST TRAIN, NIST DEV and NIST CORE TEST sets derived from the TIMIT database are used in all experiments in this thesis.

Chapter 3

Vector Representation

In this chapter, we will look at the details of the speech processing done to obtain the vector representation for each phoneme. First, the waveform is power normalized so that the average power of each utterance is constant. Having done so, the classification task would not need to deal with differences in utterance loudness. Next, we perform wavelet transform and PLP analysis on the normalized waveform. The coefficients in each phoneme are then averaged and concatenated to form a vector, which is then normalized to achieve best results during clustering. Finally, to complete the definition of the vector space, we also define the distance metric.

3.1 Waveform Normalization

The phonetic classifier may be able to give better results if it can concentrate on the spectral differences and not have to deal with differences in utterance loudness. Thus, we should perform power normalization on the waveforms before they are further processed.

The objective of power normalization is to equalize the average power across utterances. For each speech waveform $s(t)$, the average power P is defined as:

$$P = \frac{1}{T} \int_0^T s(t)^2 dt \tag{3.1}$$

where T is the length of the waveform. If we divide $s(t)$ by $\sqrt{P}$, the resulting waveform will have an average power of 1. To show this, we let the new waveform be $\hat{s}(t)$:

$$\hat{s}(t) = \frac{s(t)}{\sqrt{P}} \tag{3.2}$$

Therefore, the power of $\hat{s}(t)$ is given by:

$$\begin{aligned} \hat{P} &= \frac{1}{T}\int_0^T \hat{s}(t)^2 dt \\ &= \frac{1}{T}\int_0^T \left[\frac{s(t)}{\sqrt{P}}\right]^2 dt \\ &= \frac{\frac{1}{T}\int_0^T s(t)^2 dt}{P} \\ &= 1 \end{aligned} \tag{3.3}$$

3.2 Wavelet Transform

The short-time Fourier transform (STFT) is a useful tool for speech recognition because of its ability to analyze a speech waveform into its frequency constituents. For a given signal $x(t)$ and a suitable window function $g(t)$, we have:

$$STFT(\tau, f) = \int x(t)g(t-\tau)e^{-2j\pi ft}dt \tag{3.4}$$

Here, $g(t)$ effectively restricts $x(t)$ to an interval around τ, so that the frequency decomposition is computed for the portion of $x(t)$ around τ. If we plot the square modulus of the STFT in a time-frequency plane, we get the spectrogram, which is a very common tool in signal analysis. Such a representation of the signal is analogous to a musical score, which tells precisely which pitch (frequency) is to be played at any instant of time [5]. However, the spectrogram does not have infinite resolution in both time and frequency.

We define the frequency resolution ($\frac{1}{\Delta f}$) by:

$$(\Delta f)^2 = \frac{\int f^2|G(f)|^2 df}{\int |G(f)|^2 df} \tag{3.5}$$

where $G(f)$ is the Fourier transform of $g(t)$. We can tell two sinusoids apart only if they have frequencies which differ by at least Δf [21]. Similarly, we define the time resolution ($\frac{1}{\Delta t}$) as:

$$(\Delta t)^2 = \frac{\int t^2 |g(t)|^2 dt}{\int |g(t)|^2 dt} \tag{3.6}$$

Two pulses can be distinguished from each other if they are more than Δt apart in time. It can be shown that

$$\Delta t \Delta f \geqslant \frac{1}{4\pi} \tag{3.7}$$

which is also known as the uncertainty principle or the Heisenberg inequality. This means that there is a trade-off between time and frequency resolutions. In the STFT, both Δt and Δf are fixed. Thus, we have to select the desired Δt and Δf before performing the transformation.

The human ear, however, does not perceive auditory signals with constant frequency resolutions. That is why music is naturally distributed into octaves. To the human auditory system, frequency resolution is inversely proportional to the frequency, i.e.

$$\frac{\Delta f}{f} = c \tag{3.8}$$

where c is a constant. A signal decomposition satisfying equation (3.8) has constant relative frequency resolution, and is called a "constant-Q" analysis. Since Δf is large at high frequencies, Δt can be made very small while still satisfying the uncertainty principle (3.7). Thus, using such an analysis, we have high time resolution at high frequencies. The continuous wavelet transform (CWT) has exactly the property we need. To obtain the CWT, we compute the inner product of the signal $x(t)$ with scaled versions of the same prototype function $h(t)$:

$$CWT(\tau, a) = \frac{1}{\sqrt{|a|}} \int x(t) h\left(\frac{t-\tau}{a}\right) dt \tag{3.9}$$

where a is called the scale. If $h(t)$ is chosen to be $g(t)e^{-2j\pi ft}$, then equation (3.9) will be very similar to the STFT. It can be shown that by identifying frequency as $f = \frac{f_0}{a}$ for some initial frequency f_0, the CWT does indeed satisfy equation (3.8) [4].

Like the spectrogram, the scalogram is defined as the squared modulus of the CWT. It presents the energy distribution of the signal in a time-scale plane instead of a time-frequency plane. Figure 3-1 compares the spectrogram and the scalogram. The spectrogram presents the signal's energy distribution in a regular rectangular grid, whereas the scalogram offers a variety of time and scale resolutions. Figure 3-2 shows the scalogram and spectrogram representations of a Dirac impulse at time t. Due to a fixed time resolution, it is not possible to identify the exact time at which the impulse occurs in the spectrogram. On the other hand, we are able to identify t from the scalogram by looking at small scales. This fact may be useful in locating transitions or landmarks in speech waveforms.

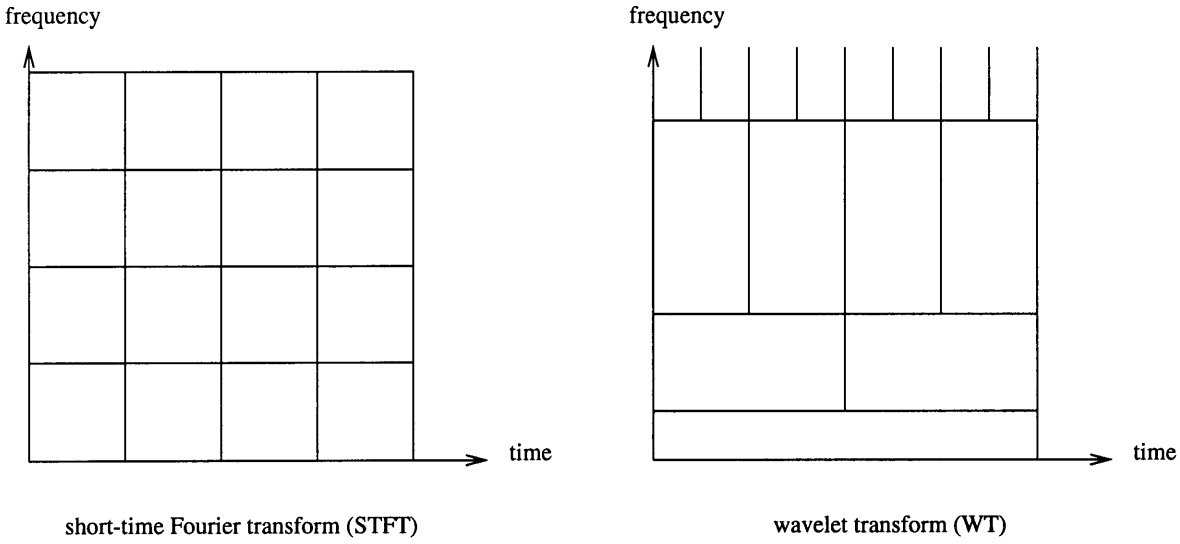

Figure 3-1: The short-time Fourier transform (STFT) and the wavelet transform (WT)

Wavelet applications in speech processing include compression of acoustic signals [27], pitch detection of speech signals [11], and music analysis and synthesis [17]. In speech recognition, work has been done in applying wavelet analysis to speech segmentation and classification [25]. Other speech recognition efforts include wavelet parameterization of speech signals [6] and the analysis of sound patterns with wavelet

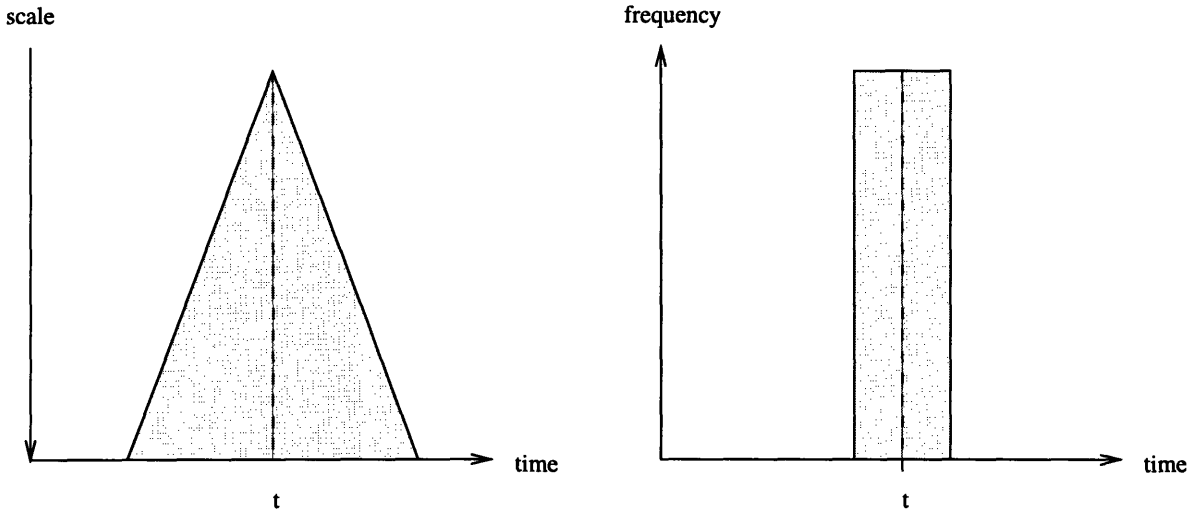

Figure 3-2: Scalogram and spectrogram of a Dirac impulse at t

transforms [18].

There are many basis functions which can be used for the wavelet transform. Some of the more common wavelet bases are the Haar, Daubechies, Morlet and Meyer wavelet families. Since it is not clear which wavelet basis is best applied to speech recognition, we select the simplest basis, namely the Haar basis, for use in this thesis.

Computing the Haar wavelet transform is very simple. Suppose we have the following signal values: [4 -2 0 6]

To get the first wavelet coefficients, we average the data points pairwise to obtain a higher-scale (lower resolution) version of the original signal: [1 3]

At the same time, we record the details that are lost due to the averaging. The first two signal values (4 and -2) differ from their average by $4 - 1 = 3$, while the last two signal values differ from their average by $0 - 3 = -3$. Hence, we have the following detail coefficients: [3 -3]

Continuing this process, we obtain the following table of averages and detail coefficients:

Scale	Averages	Detail Coefficients
1	[4 -2 0 6]	-
2	[1 3]	[3 -3]
4	[2]	[-1]

The final average and the detail coefficients are the coefficients of our wavelet transform: [2 -1 3 -3]. One way to interpret the wavelet coefficients is to express the original signal as a linear combination of the wavelet coefficients and Haar wavelets of the form shown in figure 3-3. Figure 3-4 illustrates how the original signal can be constructed from the wavelet coefficients. Note that the Haar wavelet is scaled to different sizes.

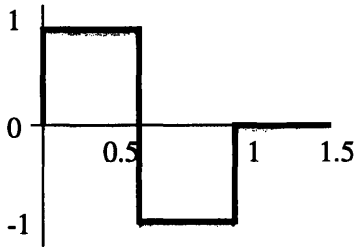

Figure 3-3: The Haar wavelet

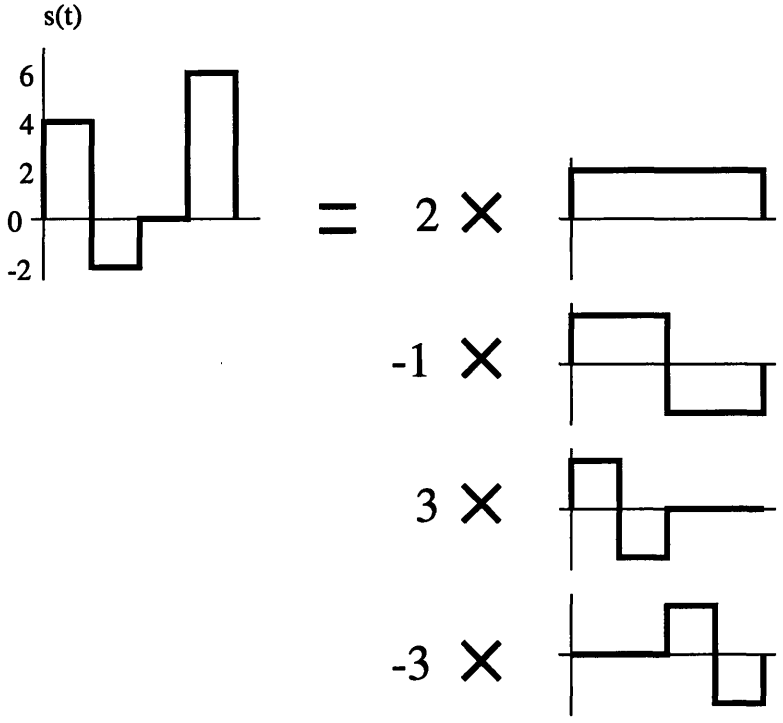

Figure 3-4: The signal expressed in terms of wavelet coefficients

Figure 3-5 shows the result of performing the Haar wavelet transform on a longer signal. In this diagram, only the detail coefficients are shown. We can obtain the wavelet coefficients at each unit of time by taking the values in the vertical strip

Table 3.1: Scales and their corresponding frequencies

Scale	Frequency	Scale	Frequency
2	4000 Hz	32	250 Hz
4	2000 Hz	64	125 Hz
8	1000 Hz	128	62.5 Hz
16	500 Hz	256	31.25 Hz

representing that instant. For example, the coefficients at $t = 5$ are given by: [1 1 -0.5]

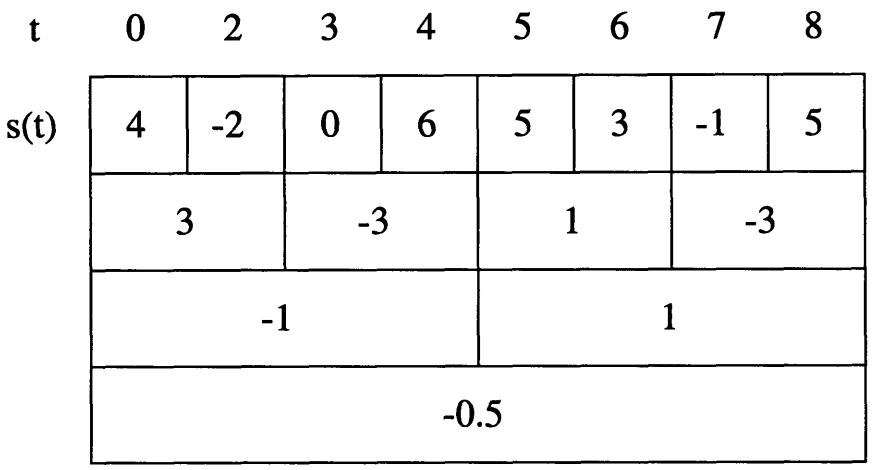

Figure 3-5: Performing the Haar wavelet transform on $s(t)$

After the Haar wavelet transform is performed on the signal, the coefficients are sampled at a rate of one set every millisecond. We then compute the decibel equivalent of the coefficients by taking the logarithm of the squared values. This is followed by a smoothing operation where the coefficients in each 40-ms window are averaged. Finally, only the coefficients of the lowest 8 scales of each millisecond are used. Coefficients of scales 512 and above correspond to frequencies that are too low to be audible. Table 3.1 shows the scales of the transform and their corresponding frequencies.

3.3 Perceptual Linear Predictive (PLP) Analysis

The linear predictive coding (LPC) [16] technique has been used by the speech recognition community for a long time. LPC models the vocal tract as an all-pole model and it approximates the speech waveform equally well at all frequencies of the anal-

ysis band. This property is, however, inconsistent with human hearing, which has a sensitivity that varies with frequency. PLP analysis [10] was developed as an attempt to correct this inconsistency. It models auditory perception by making use of the critical-band curve, equal-loudness curve and intensity-loudness power law.

To compute the PLP coefficients, several well-known engineering models of auditory perception are used. Figure 3-6 shows the various stages in PLP analysis. These stages are described in detail in the following sections.

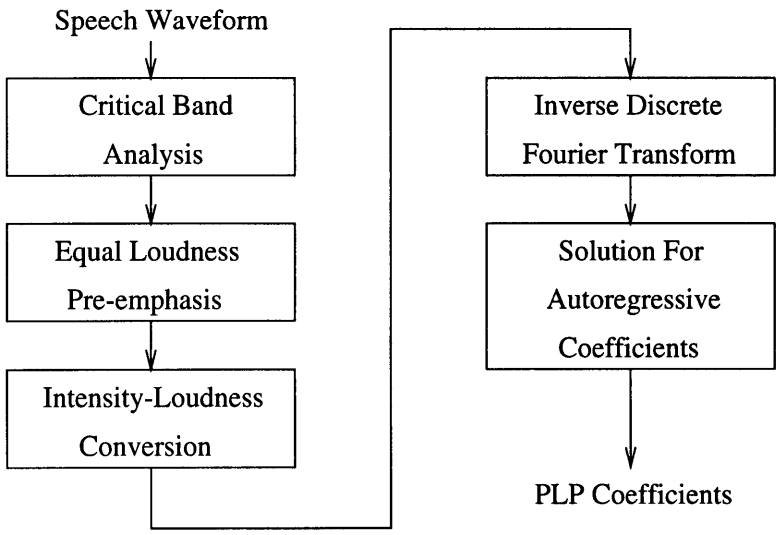

Figure 3-6: Block diagram of PLP analysis

Spectral Analysis

The discrete Fourier transform (DFT) is applied on the speech waveform to transform it into the frequency domain. The Hamming window is used, which is given by the following equation:

$$W(n) = 0.54 + 0.46\ cos\left(\frac{2\pi n}{N-1}\right) \tag{3.10}$$

where N is the length of the window, typically 20 ms.

To obtain the power spectrum $P(\omega)$, we sum the squares of the real and imaginary

parts of the Fourier transform $S(\omega)$:

$$P(\omega) = Re[S(\omega)]^2 + Im[S(\omega)]^2 \tag{3.11}$$

Critical-Band Spectral Resolution

We use the Bark-Hertz transformation due to Schroeder [23] to warp the spectrum $P(\omega)$ along its frequency axis ω into the Bark frequency Ω:

$$\Omega(\omega) = 6\ ln\left[\frac{\omega}{1200\pi} + \sqrt{\left(\frac{\omega}{1200\pi}\right)^2 + 1}\right] \tag{3.12}$$

The warped power spectrum is then convolved with the power spectrum of the simulated critical-band masking curve $\Psi(\Omega)$. In PLP analysis, the critical-band curve is given by:

$$\Psi(\Omega) = \begin{cases} 0 & \text{for } \Omega < -1.3 \\ 10^{2.5(\Omega+0.5)} & \text{for } -1.3 \leqslant \Omega \leqslant -0.5 \\ 1 & \text{for } -0.5 < \Omega < 0.5 \\ 10^{-1.0(\Omega-0.5)} & \text{for } 0.5 \leqslant \Omega \leqslant 2.5 \\ 0 & \text{for } \Omega > 2.5 \end{cases} \tag{3.13}$$

The piece-wise curve is an approximation to the asymmetric masking curve of Schroeder [23]. We get samples of the critical-band power spectrum by performing a discrete convolution of $\Psi(\Omega)$ with $P(\omega)$:

$$\Theta(\Omega_i) = \sum_{\Omega=-1.3}^{2.5} P(\Omega - \Omega_i)\Psi(\Omega) \tag{3.14}$$

The convolution reduces the spectral resolution of $\Theta(\Omega)$ in comparison with $P(\omega)$, thus allowing for down-sampling. $\Theta(\Omega)$ is sampled in approximately 1-Bark intervals in this analysis.

Equal-Loudness Preemphasis

We preemphasize the sampled $\Theta[\Omega(\omega)]$ by the following simulated equal-loudness curve:

$$\Xi[\Omega(\omega)] = E(\omega)\Theta[\Omega(\omega)] \tag{3.15}$$

where $E(\omega)$ approximates the varying sensitivity of human hearing at different frequencies [22]. It is a simulation of auditory sensitivity at about 40 dB. The approximation to be used in PLP analysis is adopted from Makhoul and Cosell [15] and is given by:

$$E(\omega) = \frac{(\omega^2 + 56.8 \times 10^6)\omega^4}{(\omega^2 + 6.3 \times 10^6)^2(\omega^2 + 0.38 \times 10^9)} \tag{3.16}$$

This is a transfer function of a filter with asymptotes of 12 dB/oct between 0 and 400 Hz, 0 dB/oct between 400 and 1200 Hz, 6 dB/oct between 1200 and 3100 Hz and 0 dB/oct between 3100 Hz and the Nyquist frequency. For applications with a Nyquist frequency above 5000 Hz, it is useful to introduce an additional term representing a steep decrease of the sensitivity of hearing for frequencies higher than 5000 Hz:

$$E(\omega) = \frac{(\omega^2 + 56.8 \times 10^6)\omega^4}{(\omega^2 + 6.3 \times 10^6)^2(\omega^2 + 0.38 \times 10^9)(\omega^6 + 9.58 \times 10^{26})} \tag{3.17}$$

The first and last samples of $\Xi[\Omega(\omega)]$, being not well-defined, are made to be equal to their nearest neighbors.

Intensity-Loudness Power Law

As the final operation before autoregressive modeling, we perform a cube-root amplitude compression as an approximation to the power law of hearing [24]:

$$\Phi(\Omega) = \sqrt[3]{\Xi(\Omega)} \tag{3.18}$$

The operation simulates the nonlinear relation between the intensity of sound and

its perceived loudness.

Autoregressive Modeling

Here, we use the autocorrelation method to approximate $\Phi(\Omega)$ as the spectrum of an all-pole model. The inverse DFT is applied to $\Phi(\Omega)$ to yield the autocorrelation function. Using the first $M + 1$ autocorrelation values, we solve the Yule-Walker equations for the autoregressive coefficients of the Mth-order all-pole model. These autoregressive coefficients can then be transformed into cepstral coefficients.

PLP Coefficients For Phoneme Vector

According to Hermansky [10], the 5th order is the optimal order of PLP analysis in suppressing speaker-dependent information from speech. Hence, we use 5th order PLP analysis in this thesis. Using the analysis described above, we obtain 6 PLP coefficients every 20 ms. These coefficients are duplicated 20 times (one for each millisecond) so that we have 6 coefficients every millisecond.

3.4 Phoneme Vector

Combining the 8 coefficients from wavelet transform and the 6 coefficients from PLP analysis, we obtain 14 coefficients every millisecond. These coefficients are then averaged for the initial, middle and final third of each phoneme, giving 14 coefficients for each third of the phoneme. Finally, we concatenate the 3 sets of 14 coefficients to form a 42-dimensional vector representation for each phoneme.

3.5 Distance Metric

We need a distance metric for nearest neighbor computations in the clustering algorithm. The most natural distance measure is the Euclidean distance. However, we do not need to compute the square root because only an ordering of distances is needed

in nearest neighbor searches. The distance metric that we are using in this thesis is defined as follows:

$$d(x,y) = \sum_{i=1}^{42}(x_i - y_i)^2 \tag{3.19}$$

This distance metric is fast to compute and it weighs each vector coefficient equally.

3.6 Vector Normalization

As noted in the previous section, the distance metric defined by equation 3.19 weighs each vector coefficient equally. Consequently, coefficients which vary a lot will affect vector distances more than those with small variances. Vector normalization is an attempt to rectify this problem.

First, we subtract the mean from each vector coefficient to obtain zero-mean coefficients. Since the distance metric squares each coefficient, it makes sense to divide the coefficients by their root-mean-square values so that the average distance from the origin is 1 for each coefficient. Equations 3.20 and 3.21 summarize the vector normalization procedure for each coefficient c. In these equations, c_i refers to coefficient c in the ith vector. N is the total number of training vectors.

$$\bar{c}_i = c_i - \frac{1}{N}\sum_{i}^{N} c_i \tag{3.20}$$

$$\hat{c}_i = \frac{\bar{c}_i}{\sqrt{\frac{1}{N}\sum_i^N \bar{c}_i^2}} \tag{3.21}$$

Hence, we eliminate the effect of coefficient variances in the calculation of distances. However, some coefficients may be more important than others for phonetic classification. That is, we need to find a set of optimal weights for the coefficients. We shall see in chapter 6, how the root-mean-square values serve as initial values in

the genetic algorithm for optimizing the weights of coefficients.

3.7 Chapter Summary

The signal processing stages that are performed to obtain the vector representation for each phoneme are summarized in figure 3-7. The first step involves normalizing the waveform so that the average power is constant across utterances. The Haar wavelet transform and PLP analysis are both performed on the normalized waveform. The result of the wavelet transform is converted to decibels, sampled every millisecond and smoothed over 40-ms windows. Only the coefficients of the lowest 8 scales are used. At the same time, a 5th order PLP analysis is done on the normalized waveform to give 6 PLP coefficients every 20 ms. These 6 coefficients are duplicated for each millisecond. Combining the wavelet and PLP coefficients, we have 14 coefficients every millisecond. The coefficients are then averaged for the initial, middle and final third of each phoneme. We combine the 14 coefficients from each third to give a 42-dimensional vector for the phoneme. Finally, vector normalization is performed on the phoneme vector.

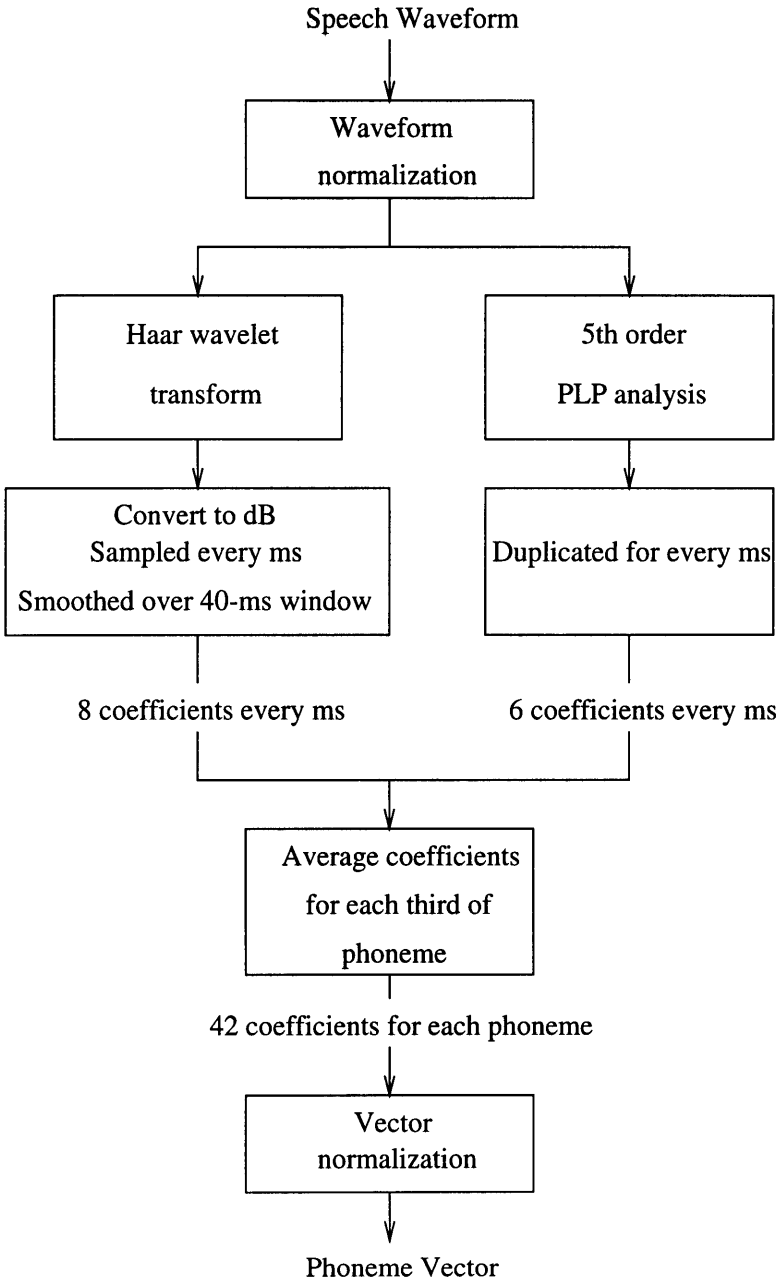

Figure 3-7: Summary of the signal processing performed to obtain phoneme vector

Chapter 4

Clustering Algorithms

With the vector space defined, we are ready to perform clustering. Consider a multi-dimensional space scattered with vectors. The objective of clustering is to identify clusters of vectors in the space, so that the vector space can be partitioned into different classes. If the vectors represent speech signals, such an operation would allow us to classify speech signals into classes representing distinct characteristics. A two-dimensional example of clustering is given in figure 4-1. Here, the crosses in bold are the centroids of the partitions and they are representative of the classes.

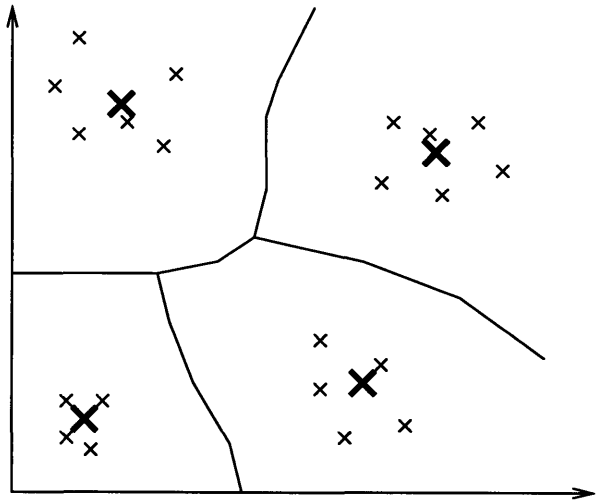

Figure 4-1: Clustering a vector space

One motivation of clustering is the hope that there will be exactly 40 partitions– each representing one phoneme. The success of clustering is, of course, dependent on

the signal representation used. When a bad signal representation is used, vectors may be close to one another when they sound different and far from one another when they are similar-sounding. Clustering will not be effective in such cases.

There are many different algorithms in the clustering literature [9]. Examples are the leader algorithm, which clusters data by identifying leading cases of partitions; the K-means algorithm which operates by switching objects from one cluster to another; the joining algorithm which joins small initial clusters to form larger clusters; and the tree-leader algorithm which adds objects to an initial clustering structure. In this thesis, we use the K-means algorithm because it is most widely-used among the various clustering algorithms in speech recognition and it is able to generate good clusters in a relatively short time.

The choice of the initial seeds (representing partitions) influences the performance of the K-means algorithm significantly. Algorithms developed in this chapter present different strategies of obtaining the initial seeds. They also differ in whether all the training vectors are clustered at the same time.

Before we develop clustering strategies, let us look at the objectives of clustering. First, a good clustering should produce clusters which contain vectors from only one phoneme each. If a cluster contains vectors from many phonemes, then it is not clear which phoneme it should represent, and thus, classification performance will suffer. Next, we would also like the clusters to be evenly spread out, so that we do not have clusters which are either too big or too small. Clusters which are too small may represent undesirable noise, while big clusters do not distinguish vectors well enough. Finally, there should not be too many clusters at the edge of the data set, as we are usually more interested in distinguishing vectors in the main data region than at the fringe. Figure 4-2 illustrates some bad clustering scenarios.

4.1 K-Means Clustering

A frequently-used clustering algorithm is called the K-means algorithm or the generalized Lloyd algorithm. Suppose we want to cluster a set of N training vectors into

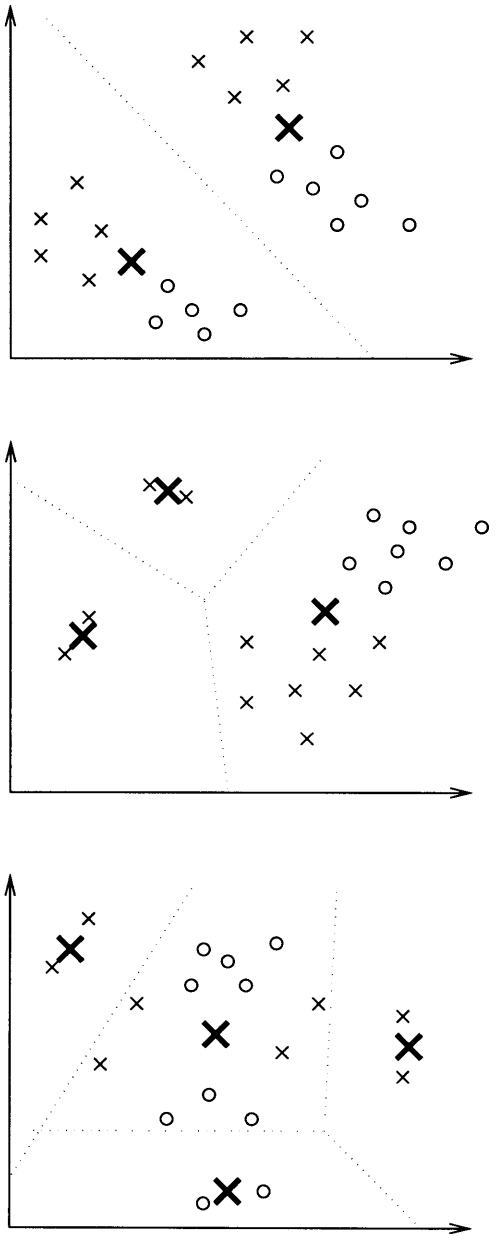

Figure 4-2: Examples of bad clustering scenarios

K clusters:

1. **Initialization:** Generate K random vectors as the initial set of cluster centroids. They are usually picked from the training set of N vectors.

2. **Nearest-Neighbor Search:** For each training vector, find the centroid that is closest (in terms of a predefined distance metric), and assign that vector to the corresponding partition.

3. **Update Centroids:** Update the centroids in each partition by computing the centroid (mean) of the training vectors assigned to that partition.

4. **Iteration:** Repeat from step 2 until further updates in step 3 do not produce significant changes.

Figure 4-3 illustrates how the centroids are updated when K-means clustering is applied to a two-dimensional space of vectors. In this diagram, the centroids are indicated by crosses in bold.

It should be noted that K-means clustering produces different results when different sets of initial centroids are chosen. In some cases, the performance can be significantly affected. Therefore, the choice of initial centroids is a very important operation.

4.2 Binary Split

One way to solve the problem of determining initial seeds for K-means clustering is to obtain the seeds in stages. The first stage is to treat the whole data set as one big cluster. We split the centroid of this cluster into two distinct vectors to be used as seeds for the next stage, which is to obtain two clusters for the data set. This process continues with the number of clusters doubled each time until the required number of clusters is reached. The procedure is called the binary split algorithm and can be formally described as follows:

1. **Initial Cluster:** The whole set of training vectors is treated as one big cluster.

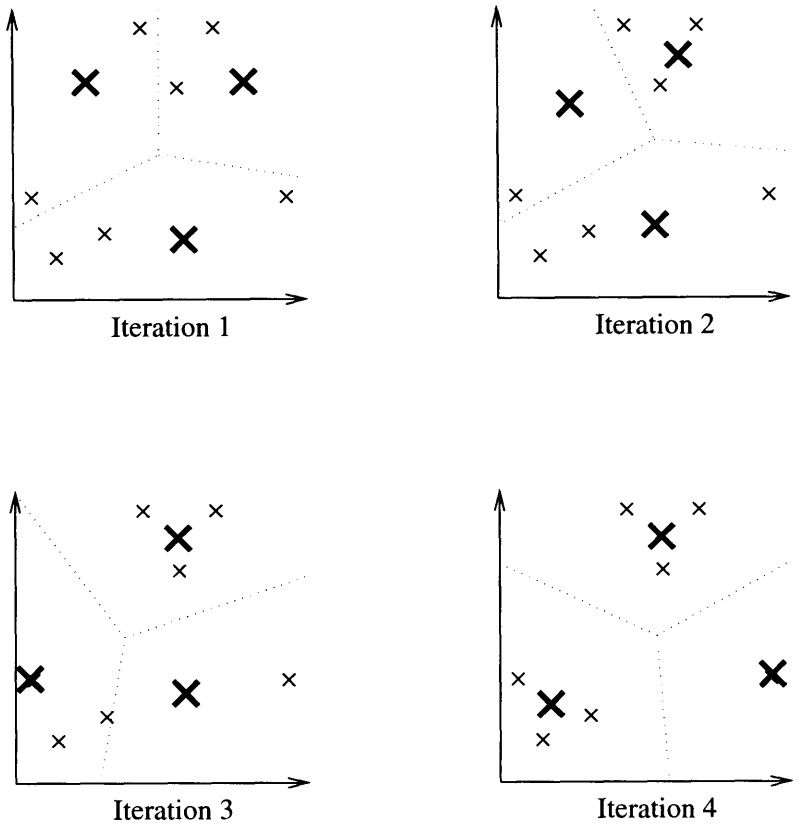

Figure 4-3: Applying K-means clustering

2. **Split Centroids:** The centroid $\vec{c}_i$ of each current cluster is split into two vectors according to the following rule:

$$\begin{aligned} \vec{c}_{1,i} &= \vec{c}_i(1+\epsilon) \\ \vec{c}_{2,i} &= \vec{c}_i(1-\epsilon) \end{aligned} \tag{4.1}$$

where ϵ is the splitting parameter, typically chosen so that $0.01 \leqslant \epsilon \leqslant 0.05$.

3. **K-means Algorithm:** Apply K-means algorithm to obtain twice the number of clusters, using $\vec{c}_{1,i}$ and $\vec{c}_{2,i}$ as the initial seeds.

4. **Iteration:** Repeat from step 2 until the required number of clusters is obtained.

By splitting each cluster centroid into two, we hope that the new clusters will be evenly spread out and that they will be of approximately the same size. These are part of the objectives of clustering.

Equal emphasis is placed on all the training vectors during the computation of centroids in the K-means algorithm. However, this results in unequal emphasis placed on phonemes because some phonemes are represented by fewer vectors than others. This may lead to bad clustering for under-represented phonemes. Our attempt to correct this situation is to assign weights to vectors. With each weight being inversely proportional to the number of vectors representing the phoneme, equal emphasis will be placed on each phoneme.

4.3 Phoneme Split

A problem with the binary split algorithm described in section 4.2 is that the splitting of centroids may not be done in the correct direction and we may end up having undesirable clusters. Figure 4-4 shows a case of bad clustering by binary split.

Instead of always splitting in the directions towards and away from the origin (as in binary split), we can split the centroids so that the new seeds are in strategic locations. Before splitting a centroid, we identify the training vectors which belong

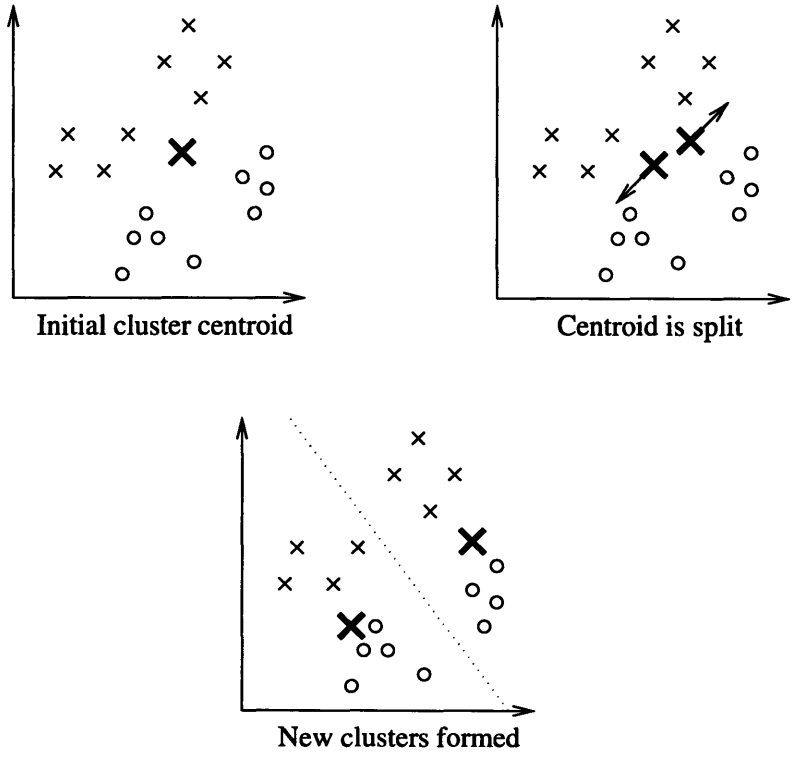

Figure 4-4: Example of bad clustering by binary split

to the cluster. If only one phoneme is represented among these vectors, we perform normal binary split (using equation 4.1). However, if more than one phoneme is represented, we identify the two phonemes which are most-represented in the cluster. Then we find the centroids of the vector sets representing the two phonemes. These are the new seeds to be used in the next clustering operation. Figure 4-5 illustrates how phoneme split corrects the binary split problem in figure 4-4.

More formally, the phoneme split algorithm can be described as follows:

1. **Initial Cluster:** The whole set of training vectors is treated as one big cluster.

2. **Split Centroids:** For each centroid $\vec{c}_i$, identify the set S_i of training vectors which belong to the cluster. If only one phoneme is represented in S_i, use equation 4.1 to split $\vec{c}_i$; else, find the two phonemes that are most-represented in S_i. Let P_1 and P_2 be the sets of training vectors in S_i that represent the two

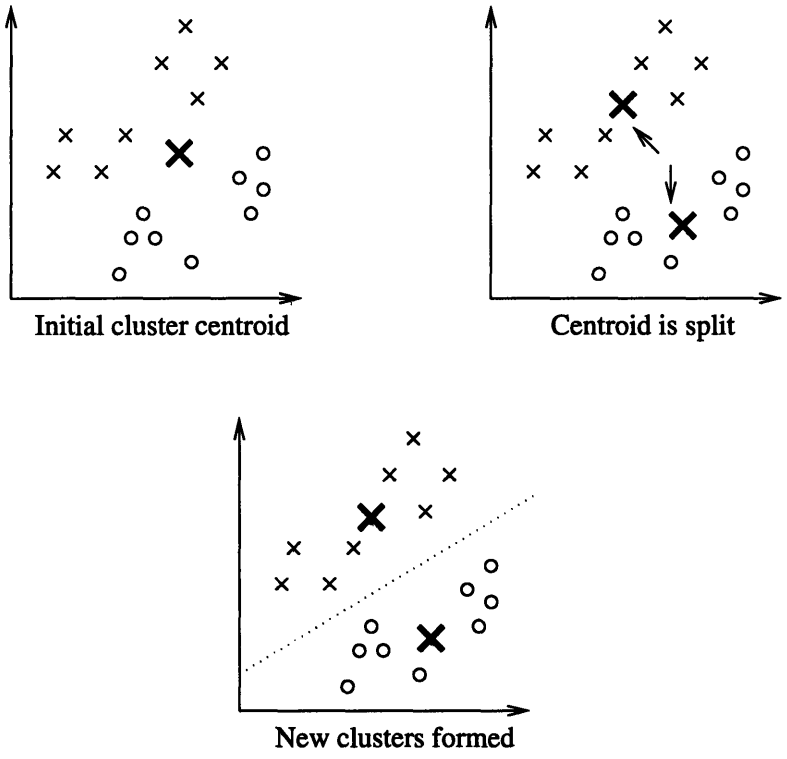

Figure 4-5: Clustering by using phoneme split

phonemes ($P_1 \subset S_i$ and $P_2 \subset S_i$). Split $\vec{c}_i$ according to the following rule:

$$\begin{aligned} \vec{c}_{1,i} &= \text{centroid of } P_1 \\ \vec{c}_{2,i} &= \text{centroid of } P_2 \end{aligned} \tag{4.2}$$

3. **K-means Algorithm:** Apply K-means algorithm to obtain twice the number of clusters, using $\vec{c}_{1,i}$ and $\vec{c}_{2,i}$ as the initial seeds.

4. **Iteration:** Repeat from step 2 until the required number of clusters is obtained.

4.4 Independent Clustering of Phonemes

Since part of the objective of clustering is to form clusters that represent only one phoneme each, it may make sense to perform clustering on each phoneme separately. That is, for each phoneme, we run the binary split algorithm on its training vectors.

At the end of the runs, we combine all the centroids. Suppose we want to obtain L clusters and there are M phonemes. Let f_i be a fraction assigned to each phoneme where $\sum_{i=1}^{M} f_i = 1$. The independent clustering technique can be summarized as follows:

1. For each phoneme i, perform K-means clustering on the set of training vectors representing it, with $K = f_i L$.

2. Take the union of all the centroid sets found.

We may assign each f_i as $\frac{1}{M}$, i.e. each phoneme gets the same number of clusters. But such an assignment may cause under-represented phonemes to have big spheres of influence. A better assignment will be to take into account how well each phoneme is represented. Let the number of training vectors representing the ith phoneme be N_i. We then define f_i as follows:

$$f_i = \frac{N_i}{N} \tag{4.3}$$

where N is the total number of training vectors.

4.5 Clustering After Independent Clustering

The independent clustering technique described in section 4.4 may produce cluster centroids which are not well-distributed since the centroids are derived independently for each phoneme. Thus, a possible improvement may be to treat the centroids as seeds to be used for a final clustering of the whole training set. Here is a formal description of the algorithm:

1. Perform independent clustering (algorithm is described in section 4.4).

2. Perform K-means clustering on all the training vectors using the centroids obtained in step 1 as initial seeds.

4.6 Chapter Summary

In this chapter, we developed several clustering algorithms based on the objectives of clustering stated at the beginning of the chapter. The binary split algorithm solves the problem of assigning initial seeds by obtaining cluster centroids in stages. It is able to obtain clusters which are evenly spread throughout the training set. A variation of this algorithm is to assign a weight to each training vector based on how well the corresponding phoneme is represented. The splitting method employed in binary split may cause undesirable clusters to be formed. The phoneme split algorithm attempts to rectify this problem by placing new seeds at strategic positions. Instead of clustering all the training vectors at the same time, it may be better to perform clustering on each phoneme separately, in the hope of producing more one-phoneme clusters. This is the independent clustering algorithm. Finally, we would like to treat the centroids obtained from independent clustering as seeds for K-means clustering on the entire training set.

It is not clear which algorithm would perform the best when used for phonetic classification. We present and compare the results of each algorithm in chapter 6.

Chapter 5

Classifying the Clusters

After the training vectors are clustered, the classifier is trained by collecting phoneme statistics for each cluster. This chapter describes how training is done and how we classify any given test vector. We also see how the k-nearest neighbor rule can be applied to enhance the performance of the classifier.

5.1 Training and Using the Classifier

The result of clustering is a set of clusters which correspond to different phonemes. In order to determine which phoneme each cluster represents, we need to collect phoneme statistics. This can be done using the same set of training vectors that are used for clustering.

For each cluster, we determine the vectors from the training set that belong to it. Then we determine the proportion f_i of each phoneme among these vectors. $F = \{f_1, f_2, \ldots, f_M\}$ is the set of statistics we keep for each cluster (M is the number of phonemes).

When the classifier is presented with a test vector, it determines which cluster the vector belongs to by finding the cluster centroid which is nearest to it. From the statistics set F of that cluster, the highest proportion f_{i^*} is picked. The phoneme indexed by i^* is then presented as the output.

5.2 Applying the k-Nearest Neighbor Rule

The classification rule described in section 5.1 is effectively the nearest neighbor method since it picks the cluster centroid that is nearest to the test vector, and uses the cluster's statistics to determine the phoneme. A more sophisticated method than the nearest neighbor method is the k-nearest neighbor method[26]. Instead of identifying only one nearest neighbor, the algorithm identifies k nearest neighbors, and the classification is made by a majority rule. That is, the classification that is most represented among the k nearest neighbors is the one assigned to the test vector.

To see how the k-nearest neighbor method works, we consider a general situation of classifying a point x among classes $A_1, A_2, \ldots, A_M$. Let n_i be the number of training points in A_i and k_i be the number of training points of class A_i among the k nearest neighbors of x. Let V be the volume of space containing just the k nearest neighbors. Then we may use the following as an estimate of the density function $p(x|A_i)$:

$$\hat{p}(x|A_i) = \frac{k_i}{n_i V} \tag{5.1}$$

Similarly, the *a priori* probability of obtaining a vector of class A_i can be estimated as follows:

$$\hat{P}(A_i) = \frac{n_i}{n} \tag{5.2}$$

where n is the total number of training points. Combining the two probability equations, we obtain:

$$\hat{p}(x, A_i) = \frac{k_i}{nV} \tag{5.3}$$

We also estimate the *a priori* probability of x:

$$\hat{p}(x) = \frac{k}{nV} \tag{5.4}$$

Therefore, we have

$$\begin{aligned} \hat{P}(A_i|x) &= \frac{\hat{p}(x, A_i)}{\hat{p}(x)} \\ &= \frac{k_i}{k} \end{aligned} \tag{5.5}$$

Since we would like to maximize the probability of classifying x correctly, we want to pick A_i such that

$$P(A_i|x) > P(A_j|x) \qquad \forall j \neq i \tag{5.6}$$

Thus, equation 5.5 describes the k-nearest neighbor rule, because $P(A_i|x) > P(A_j|x)$ if and only if $k_i > k_j$. That is, we pick the class with the highest k_i in order to maximize our probability of getting it right.

In the phonetic classifier, the statistics of the k nearest neighbors are combined by adding up the corresponding proportions to form the set $\{\sum f_1, \sum f_2, \ldots, \sum f_M\}$. The phoneme corresponding to the highest sum is assigned to the test vector.

5.3 Chapter Summary

In this chapter, we saw how the classifier is trained by recording the proportion of each phoneme in each cluster. Given a test vector, we identify the cluster it belongs to and assign it the phoneme with the highest proportion. The k-nearest neighbor method was introduced as an enhancement for the classifier.

Chapter 6

Classification Results

Results of using the phonetic classifier are presented in this chapter. The first part of the chapter involves the adjustment of parameters to optimize the performance of the classifier. This includes the determination of optimal values for K in K-means clustering, k in the k-nearest neighbor method and the weights of coefficients in vector normalization. We also look at the performance of the classifier using several different choices of phoneme vectors. In the second part of the chapter, the optimized parameters are used to test the various clustering algorithms. Finally, in order to test the significance of the wavelet and PLP coefficients in the analysis, we perform classification experiments with only one set of coefficients at a time.

6.1 Determining Optimal Parameters

In this section, we attempt to determine the optimal value for K in K-means clustering, k in the k-nearest neighbor method and the weights of coefficients in vector normalization. Training and testing data that are used in these optimizations are the NIST TRAIN and NIST DEV sets respectively. We refrain from using the NIST TEST set for optimization because that is reserved for final testing of the classifier.

To optimize the K in K-means clustering, we perform phonetic classification experiments with different values of K. Training is done using the NIST TRAIN set and testing is done on the NIST DEV set. The task is to classify vowels shown in

table 2.1 and clustering is performed with the binary split algorithm (section 4.2). From figure 6-1, we see that the performance peaks at $K = 1024$.

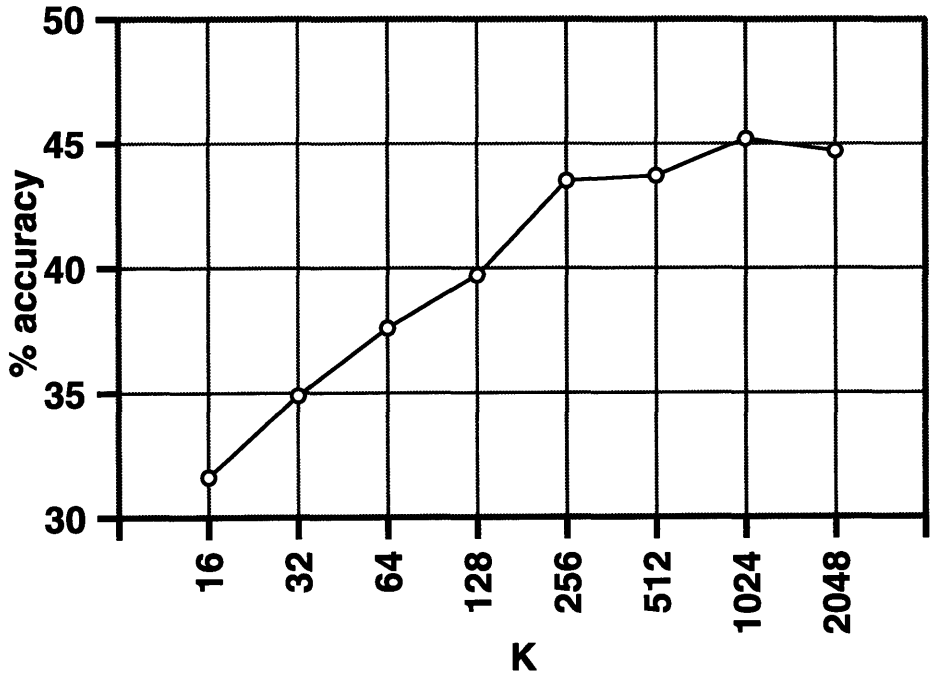

Figure 6-1: Determining optimal value of K in K-means clustering

Next, we optimize the value of k in the k-nearest neighbor method. As in the case for optimizing K in K-means clustering, we use the NIST TRAIN and NIST DEV sets for training and testing. We use the binary split algorithm with $K = 1024$ to classify vowels. Figure 6-2 presents the results of experiments with various values of k. There does not seem to be a definite optimal value of k, but k in the range $4 \leqslant k \leqslant 12$ generally gives good results. We choose $k = 8$ for use in subsequent experiments.

The genetic algorithm is used to optimize the weights of coefficients in vector normalization. The genome consists of the weights of each of the 42 coefficients. In each generation, we start off with two parents, giving rise to two offsprings. Only the fittest two individuals (out of four) are selected for the next generation. Both crossovers and mutations are implemented. The fitness function is the fraction of phonemes which are classified correctly. In order to speed up the computation of

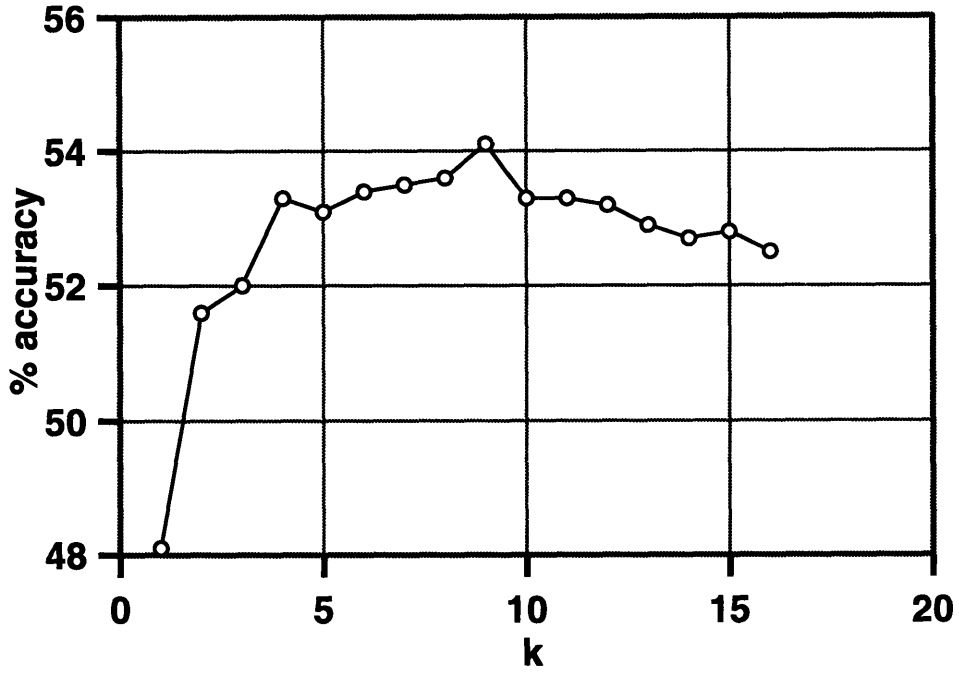

Figure 6-2: Determining optimal value of k in the k-nearest neighbor method

the fitness function, only part of the NIST TRAIN set is used for both training and testing. The task is vowel classification, using the list of vowels in table 2.1. The independent clustering algorithm described in section 4.4 is used for clustering since it is fast and it gives good classification results. Due to time limitations, the genetic algorithm is only run for 10000 generations. Results are shown in figure 6-3. Table 6.1 lists the optimized weights of coefficients.

6.2 Choice of Phoneme Vectors

In section 3.4, the phoneme vector was defined as the concatenation of coefficients from the initial, middle and final third of each phoneme, forming a vector with 42 coefficients. There are many possible ways of forming the phoneme vector. We can split the phoneme into halves, quarters or fifths. In order to determine which is the best representation, experiments are conducted to test each of the above combinations. Results are shown in figure 6-4. In these experiments, the NIST TRAIN and

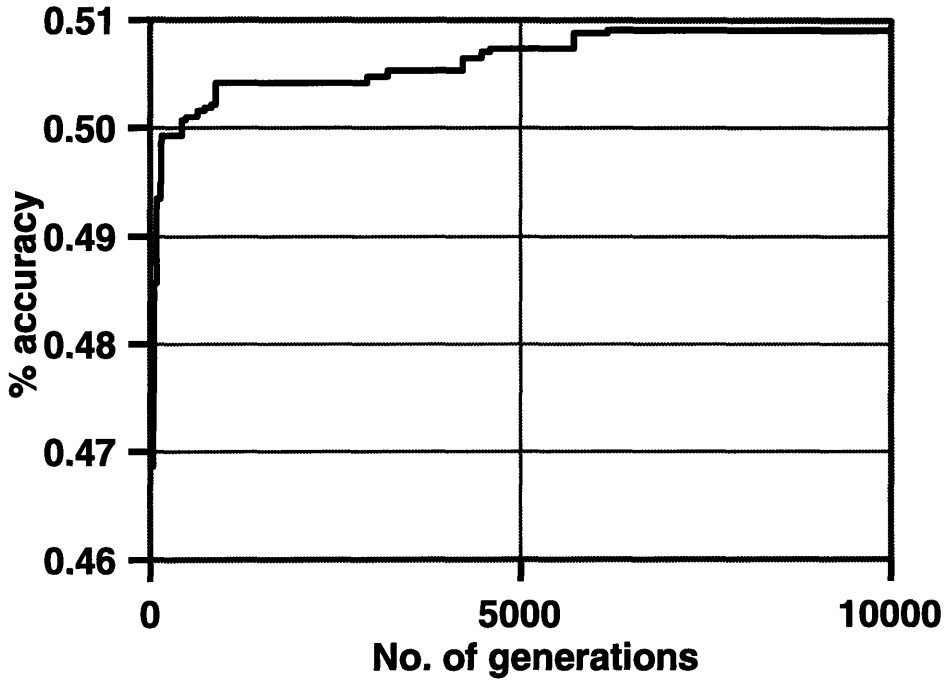

Figure 6-3: Optimizing weights of coefficients for vector normalization

Table 6.1: Optimized weights of coefficients

Coefficient	Weight	Coefficient	Weight	Coefficient	Weight
1	25.172358	15	23.546637	29	27.508614
2	19.218266	16	17.280016	30	13.651873
3	21.742237	17	11.299080	31	10.648049
4	24.898365	18	11.924601	32	15.503012
5	18.966608	19	17.581976	33	11.959195
6	25.621672	20	24.429148	34	27.137836
7	33.172913	21	32.420090	35	34.634777
8	37.476593	22	36.827389	36	38.795593
9	1.096734	23	1.184915	37	0.951037
10	0.230770	24	0.251185	38	0.217823
11	0.337857	25	0.257460	39	0.321228
12	0.225993	26	0.269493	40	0.218384
13	0.357310	27	0.185793	41	0.186709
14	0.067926	28	0.071770	42	0.071181

NIST DEV sets are used for training and testing respectively. Clustering is done by independent clustering with $K = 1024$. The k in the k-nearest neighbor method is chosen to be 8, and the task is vowel classification. The results show that it is best to use the 42-dimensional phoneme vector as defined in section 3.4.

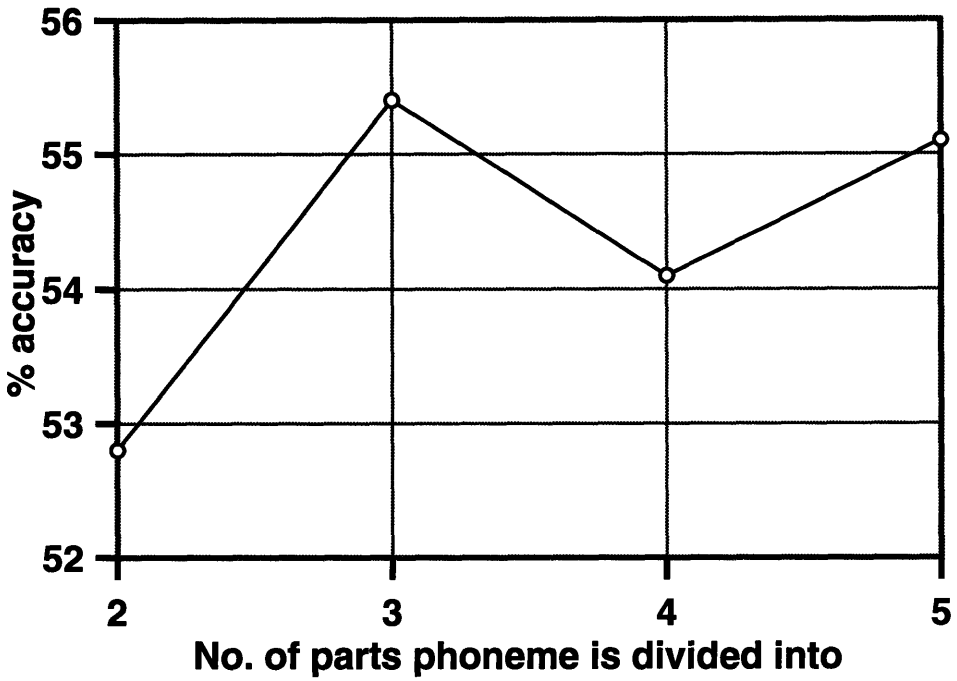

Figure 6-4: Optimizing the phoneme vector

6.3 Various Clustering Algorithms

Having optimized all the parameters for clustering and classification, we are ready to test the various clustering algorithms described in chapter 4. All the experiments in this section use the NIST TRAIN set as the training data and the NIST CORE TEST set as the testing data. Parameters take their optimized values as found in section 6.1.

Figures 6-5, 6-6 and 6-7 present the results for vowel, consonant and phoneme classifications respectively. The abbreviations for the various clustering algorithms are listed in table 6.2.

Table 6.2: Abbreviations for the various clustering algorithms

Abbreviation	Clustering Algorithm
BS	Binary split
BSW	Binary split with weights
PS	Phoneme split
PSW	Phoneme split with weights
IC	Independent clustering (equal weightage)
ICP	Independent clustering (proportional)
CAIC	Clustering after independent clustering (equal weightage)
CAICP	Clustering after independent clustering (proportional)

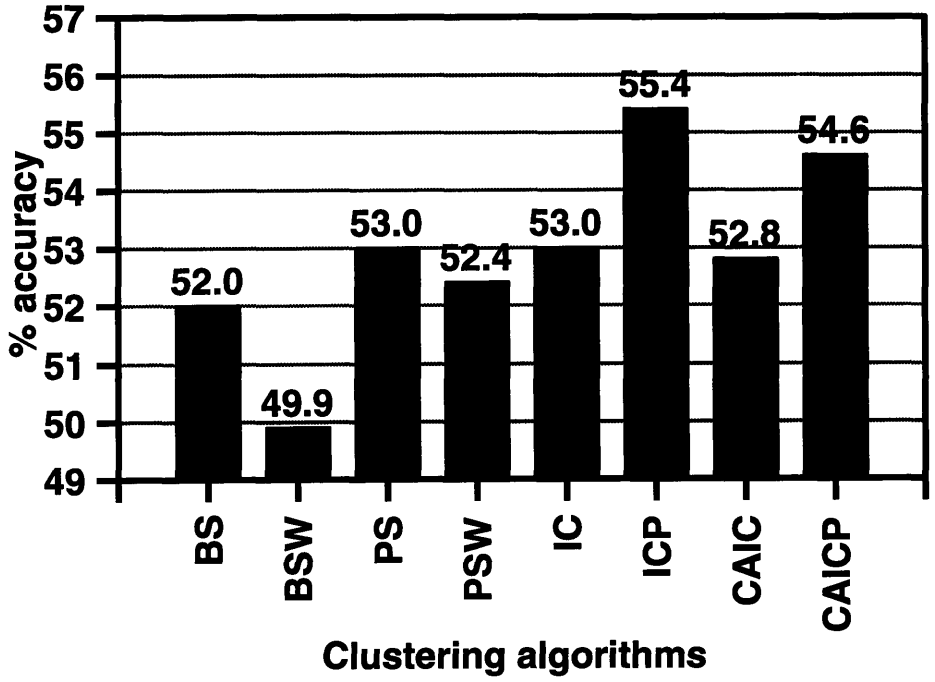

Figure 6-5: Results of vowel classification

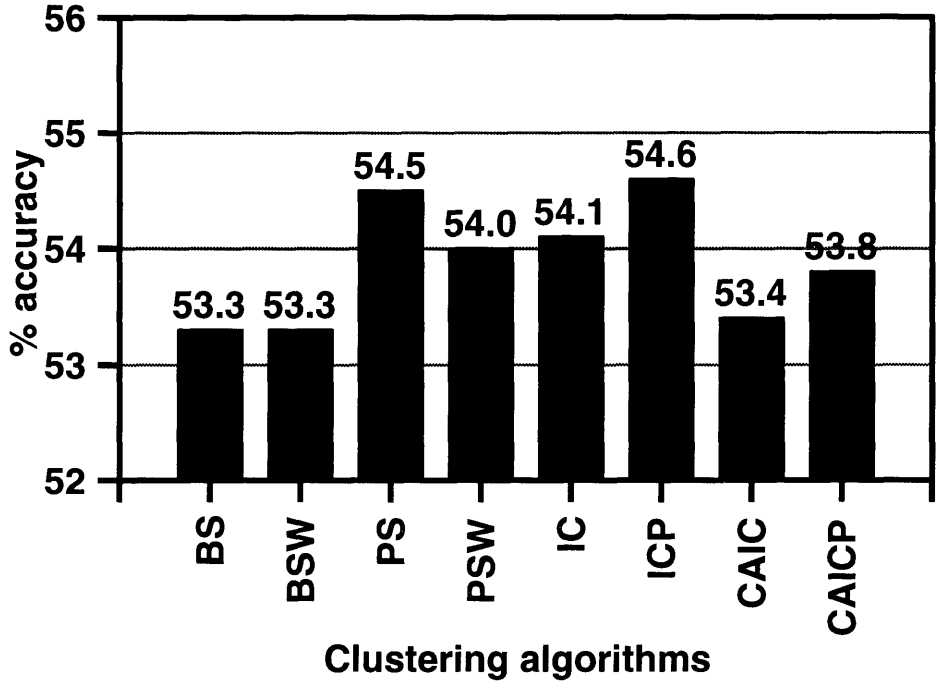

Figure 6-6: Results of consonant classification

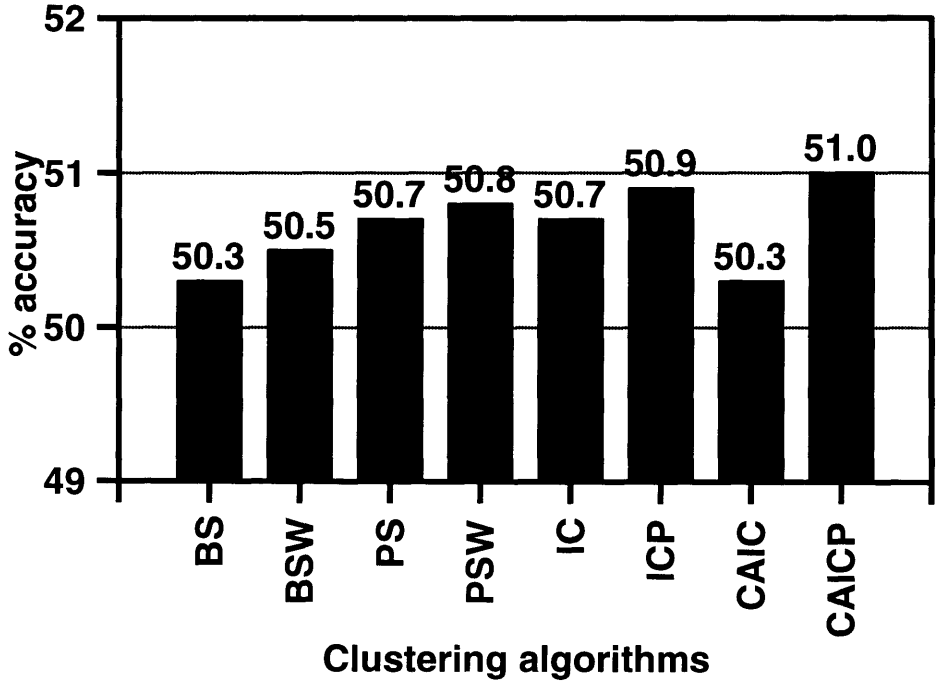

Figure 6-7: Results of phoneme classification

The results indicate that the independent clustering algorithm with proportional phoneme emphasis is the best performer. It classifies vowels with an accuracy of 55.4%, which is less than those attained by previous experiments described in section 2.2. For consonant classification, independent clustering yields an accuracy of 54.6%, which is comparable to that of vowel classification. This shows that the method is sensitive to sound transitions present in consonants too. Finally, an accuracy of 50.9% is obtained from the classification of the Kai-Fu Lee classes. The most likely reason for the drop in performance is that the number of clusters (1024) is not enough to distinguish between the phonemes in the larger phoneme set. However, due to time limitation, it is not feasible to perform clustering with more than 1024 clusters. The small number of clusters also affects the relative performances of the various clustering algorithms because improved clustering strategies do not produce much effect when there is an insufficient number of clusters to distinguish between phonemes.

From the confusion matrices in appendix C, we see that the algorithms with weighted phonemes (i.e. BSW and PSW) are able to classify phonemes which are under-represented (e.g. **u** and **ɔʸ**) more accurately. However, they do not perform as well for well-represented phonemes and consequently, the average accuracies are lower than that of their unweighted counterparts.

One major weakness of the phonetic classifier is its inability in to distinguish voiced and unvoiced consonants. It tends to classify voiced consonants (e.g. **b** and **g**) as their unvoiced counterparts (e.g. **p** and **k**). The classifier is also unable to distinguish between similar-sounding pairs of phonemes like /**n**, **ŋ**/, /**ʃ**, **ʒ**/ and /**I**, **ʊ**/.

Figure 6-8 shows the effect of the size of training data on the performance of the phonetic classifier. The independent clustering algorithm is used here.

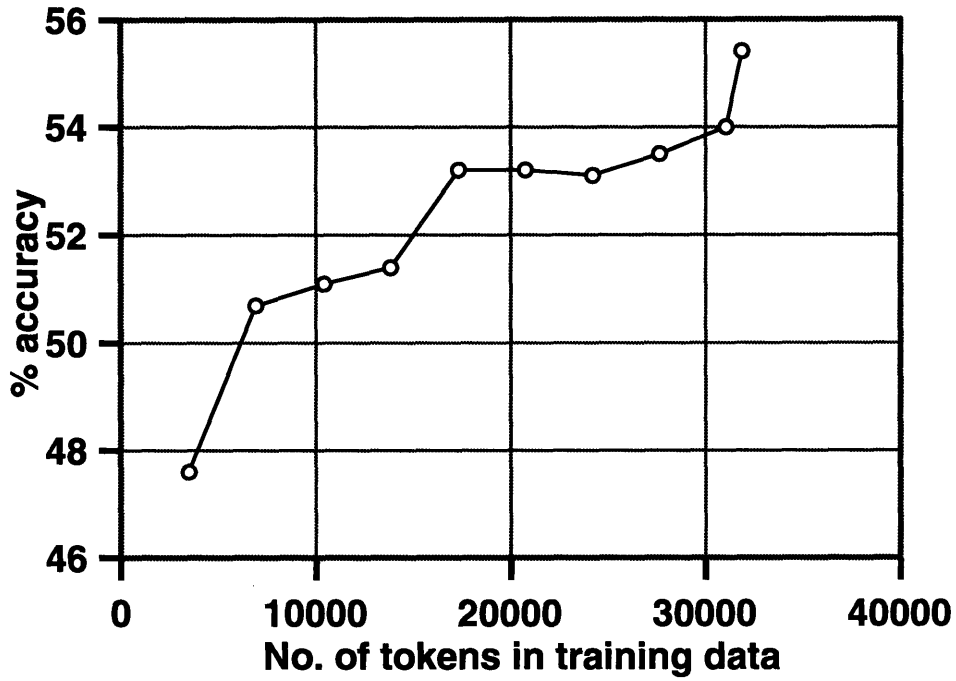

Figure 6-8: Effect of the size of training data on classification accuracy

Table 6.3: Classification results for wavelet-only phoneme vector

Phoneme Set	Wavelet-only	Wavelet & PLP	No. of Tokens
Vowel	28.5%	55.4%	1637
Consonant	42.0%	54.6%	3195
Kai-Fu Lee Classes	39.2%	50.9%	6817

6.4 Using Only Wavelet Transform

It is not clear from observing the classification results whether both the wavelet transform and PLP analysis are important for the phonetic classifier to perform well. We would like to test the significance of the PLP analysis by classifying phoneme vectors that are constructed from only wavelet coefficients. That is, each phoneme vector will have $3 \times 8 = 24$ wavelet coefficients. The weights of coefficients are picked from those corresponding to wavelet coefficients in table 6.1. The NIST TRAIN and NIST CORE TEST sets are used and clustering is performed by the independent clustering algorithm. Table 6.3 compares the classification results with those obtained using the whole set of 42 coefficients.

Table 6.4: Classification results for PLP-only phoneme vector

Phoneme Set	PLP-only	Wavelet & PLP	No. of Tokens
Vowel	53.1%	55.4%	1637
Consonant	52.3%	54.6%	3195
Kai-Fu Lee Classes	49.9%	50.9%	6817

Treating the vowel classification accuracy as a binomial distribution with mean $\mu = p = 0.554$ and standard deviation $\sigma = \sqrt{\frac{p(1-p)}{n}} = \sqrt{\frac{0.554(1-0.554)}{1637}} \approx 0.01229$, we can perform a test on the hypothesis that the drop in performance is not just due to statistical randomness. Testing at a 5% level of significance, we have $0.285 < 0.534 = 0.554 - 1.645\sigma$, which shows that the change in performance is indeed significant. Tests on consonant and phoneme classifications give the same result.

The results show that wavelet coefficients perform well in consonant and general phoneme classifications, compared to vowel classification. This is due to the ability of the wavelet transform to detect sound transitions, which are abundant in consonants.

6.5 Using Only PLP Analysis

Similarly, we would like to test the significance of the wavelet transform in the phonetic classifier. This time, we classify phoneme vectors that are constructed from only PLP coefficients, making a total of $3 \times 6 = 18$ PLP coefficients in each vector. The weights of coefficients are picked from those corresponding to PLP coefficients in table 6.1. The NIST TRAIN and NIST CORE TEST sets are used and clustering is performed by the independent clustering algorithm. Table 6.4 compares the classification results with those obtained using the whole set of 42 coefficients.

We perform tests of significance similar to what was done in section 6.4, by treating the classification accuracies as binomial distributions. Tests on vowel, consonant and phoneme classifications all show that the change in performance is significant.

6.6 Chapter Summary

In the first part of this chapter, we optimized the various parameters that may affect the phonetic classifier's performance. The K in K-means clustering and k in the k-nearest neighbor method were optimized by trying out different values and identifying the values that give the best results. To optimize the weights of coefficients in vector normalization, we used the genetic algorithm with classification performance as the fitness function. A comparison between various choices of phoneme vectors showed that the 42-dimensional phoneme vector is optimal.

The second part of the chapter presented a series of classification results using the various clustering algorithms. The results showed that the independent clustering algorithm performs the best for phonetic classification. It classifies vowels with an accuracy of 55.4%, which is less than those attained by previous experiments. For consonant classification, independent clustering yields an accuracy of 54.6%, showing that the method is sensitive to sound transitions present in consonants. Finally, an accuracy of 50.9% is obtained from the classification of the Kai-Fu Lee classes, which is comparable to previous work. Tests with wavelet-only coefficients and PLP-only coefficients showed that both the wavelet transform and PLP analysis are significantly important for the phonetic classifier. The test with wavelet-only coefficients also showed that wavelet coefficients are useful for detecting sound transitions, which are abundant in consonants.

Chapter 7

Conclusions and Future Work

7.1 Summary and Conclusions

In this thesis, we explored the use of K-means clustering, wavelet transform and PLP analysis in phonetic classification. The task of phonetic classification was chosen because previous work was available for comparison. To ensure that proper comparisons can be made, we chose a standard speech database (TIMIT) and used standard sets of data for training, development and testing (NIST data sets).

Chapter 2 described the task of phonetic classification together with the phonemes that were used in this thesis. For the purpose of comparison, we presented some results from previous work on phonetic classification. The TIMIT speech corpus and the NIST data sets used in all the experiments in this thesis were briefly described.

The signal processing stages that are performed to obtain the vector representation for each phoneme were presented in chapter 3. The first step involves normalizing the waveform so that the average power is constant across utterances. The Haar wavelet transform and PLP analysis are both performed on the normalized waveform. The result of the wavelet transform is converted to decibels, sampled every millisecond and smoothed over 40-ms windows. Only the coefficients of the lowest 8 scales are used. At the same time, a 5th order PLP analysis is done on the normalized waveform to give 6 PLP coefficients every 20 ms. These 6 coefficients are duplicated for each millisecond. Combining the wavelet and PLP coefficients, we have 14 coefficients

every millisecond. The coefficients are then averaged for the initial, middle and final third of each phoneme. We combine the 14 coefficients from each third to give a 42-dimensional vector for the phoneme. Finally, vector normalization is performed on the phoneme vector.

With the vector space defined, we were ready to perform clustering in chapter 4. We developed several clustering algorithms based on the objectives of creating one-phoneme clusters; creating clusters which are evenly spread out; and minimizing the number of clusters at the fringe of the data set. The binary split algorithm solves the problem of assigning initial seeds by obtaining cluster centroids in stages. It is able to obtain clusters which are evenly spread throughout the training set. A variation of this algorithm is to assign a weight to each training vector based on how well the corresponding phoneme is represented. The splitting method employed in binary split may cause undesirable clusters to be formed. The phoneme split algorithm attempts to rectify this problem by placing new seeds at strategic positions. Instead of clustering all the training vectors at the same time, it may be better to perform clustering on each phoneme separately, in the hope of producing more one-phoneme clusters. This is the independent clustering algorithm. Finally, we would like to treat the centroids obtained from independent clustering as seeds for K-means clustering on the entire training set.

In chapter 5, we saw how the classifier is trained by recording the proportion of each phoneme in each cluster. Given a test vector, we identify the cluster it belongs to and assign it the phoneme with the highest proportion. The k-nearest neighbor method was introduced as an enhancement for the classifier.

Finally, in chapter 6, we optimized the various parameters that may affect the phonetic classifier's performance. These include the K in K-means clustering and k in the k-nearest neighbor method. The genetic algorithm was used to optimize the weights of coefficients in vector normalization. A comparison between various choices of phoneme vectors showed that the 42-dimensional phoneme vector is optimal. This was followed by a series of classification results using the various clustering algorithms. The results showed that the independent clustering algorithm with

proportional phoneme emphasis performs the best for phonetic classification. Tests with wavelet-only coefficients and PLP-only coefficients showed that both the wavelet transform and PLP analysis are significantly important for the phonetic classifier.

Using the best combination of parameters and the best clustering method, the phonetic classifier classifies vowels with an accuracy of 55.4%, which is less than those attained by previous experiments. For consonant classification, independent clustering yields an accuracy of 54.6%. An accuracy of 50.9% is obtained from the classification of the Kai-Fu Lee classes. One major weakness of the phonetic classifier is its inability to distinguish voiced and unvoiced consonants. It is also unable to distinguish between several similar-sounding pairs of phonemes.

From the experiments, we saw that independent clustering with proportional phoneme emphasis (ICP) is a good clustering strategy for phonetic classification. Not only does it classify vectors more accurately, it is also fast to compute because clustering is done separately for each phoneme. It is rather surprising that the algorithms with weighted phonemes (i.e. BSW and PSW) do not perform as well as their unweighted counterparts. They are better at classifying under-represented phonemes, but are less accurate at classifying well-represented phonemes.

Although the results show that wavelet coefficients are inferior to PLP coefficients in performing phonetic classification, they are not completely irrelevant. The experiment with wavelet-only coefficients show that wavelet coefficients perform quite well in consonant classification. This is due to the ability of the wavelet transform to detect sound transitions, which are abundant in consonants.

It is important to note that the results do not indicate that wavelets are not suitable for speech recognition. There are many factors which might have contributed to the poor performance. First, the results show that classification accuracies do not vary much when different clustering algorithms are used. One possible explanation is that the signal representation chosen is bad. However, PLP coefficients have been used in other experiments and they have produced good results. Hence, a better explanation would be that clustering is not a suitable method for phonetic classification. As an extension of the thesis, we can simply apply the k-nearest neighbor method to

the entire training vector set so that no information is lost through clustering. Another possible reason for the poor performance is that some wavelet information is lost through signal processing. The method of obtaining wavelet coefficients described in section 3.2 does not retain the wavelet phase information, which might be important for speech recognition. Also, information is lost when we average the coefficients in each third of the phoneme.

7.2 Future Work

Since K-means clustering works for any set of vectors, we are not limited to using only wavelet and PLP coefficients. In fact, clustering works best when the vector space is sparse, and adding more coefficients will most likely make the space more sparse. Possible additions to the vectors are short-time Fourier transform (STFT) coefficients, mel-frequency spectral and cepstral coefficients (MFSC and MFCC respectively), and time derivatives of the various speech coefficients.

We chose the Haar wavelet basis for the wavelet transform used in this thesis because it is the simplest basis available and we do not know which basis is the most suitable for speech recognition. It would be interesting to use other wavelet bases such as the Daubechies, Morlet and Meyer wavelet families. One of these wavelet families might prove to be more suitable for speech recognition than the Haar wavelet basis.

It will be interesting to perform experiments using wavelet and PLP coefficients as signal representations for a HMM-based phonetic recognition system. In this way, we can compare the "Wavelet-PLP" representation with other signal representations such as LPC and MFCC. Similar experiments can also be conducted on phonetic recognition systems using artificial neural networks (ANN).

Another possible extension of this thesis is to apply the "Clustering-Wavelets-PLP" combination to other speech recognition tasks. Instead of phonetic classification, we can extend it to phonetic recognition, where the phoneme boundaries are not given. In this case, the clustering will be done on vectors representing the speech

signal at each instant of time, rather than on vectors representing phonemes. The same method can also be applied to recognizing distinctive features for distinctive feature-based speech recognition systems. By incorporating other techniques such as time alignment, dynamic programming and HMMs, we can perform recognition of higher-level structures like words, phrases and sentences.

Appendix A

Programs

This appendix describes the programs that are written for this thesis and how they are used to perform phonetic classification. Figure A-1 shows how signal processing is done to obtain phoneme vectors. Training and testing of the phonetic classifier are shown in figure A-2. Finally, figure A-3 shows how summaries and confusion matrices are obtained from the score file.

- **au2dat <in>.au <out>.dat**
  Converts audio file <in>.au to signal data file <out>.dat.
- **wt <signal>.dat <wt>.dat [maxscales]**
  Performs Haar wavelet transform on signal <signal>.dat. [maxscales] is an optional parameter that states the maximum no. of scales needed. The wavelet coefficients are stored in <wt>.dat.
- **ds <s>.dat <wt>.dat <ds>.dat <dwt>.dat <step>**
  Down-samples the signal <s>.dat and wavelet transform <wt>.dat to <ds>.dat and <dwt>.dat respectively. The down-sampling step used is given by <step>.
- **swt <wt>.dat <swt>.dat <window>**
  Converts wavelet coefficients in <wt>.dat to dB and performs smoothing operation with a window size of <window>. The result is stored in <swt>.dat.
- **plp <signal>.dat <plp>.dat**
  Performs PLP analysis on <signal>.dat. The PLP coefficients are stored in <plp>.dat.
- **catsets <set1>.dat <set2>.dat <concatenated>.dat**
  Concatenates two data sets <set1>.dat and <set2>.dat. Concatenated data is

stored in <concatenated>.dat. The two data sets must have the same number of dimensions.

- **mergesets <set1>.dat <set2>.dat <merged>.dat**
  Merges two data sets <set1>.dat and <set2>.dat to form <merged>.dat. The two data sets must have the same length.

- **makephonedat <set>.dat <label>.lbl <out>.dat**
  Makes phoneme vectors from vector coefficients <set>.dat and phoneme labels <label>.lbl. The phoneme vectors are stored in <out>.dat.

- **catphonedat <datlist> <setdir> <concatenated>.dat**
  Concatenates phoneme vector files listed in <datlist>. The vector files are to be found in directory <setdir>. The result is stored in <concatenated>.dat.

- **createkm1 <out>.km <train>.dat <K> <phoneN>**
  Performs K-means clustering using <train>.dat as training vectors. <K> is the no. of clusters and <phoneN> is the no. of phonemes. The K-means data is stored in <out>.km. Algorithm used is binary split without phoneme weights.

- **createkm2 <out>.km <train>.dat <K> <phoneN>**
  Performs K-means clustering using <train>.dat as training vectors. <K> is the no. of clusters and <phoneN> is the no. of phonemes. The K-means data is stored in <out>.km. Algorithm used is binary split with phoneme weights.

- **createkm3 <out>.km <train>.dat <K> <phoneN>**
  Performs K-means clustering using <train>.dat as training vectors. <K> is the no. of clusters and <phoneN> is the no. of phonemes. The K-means data is stored in <out>.km. Algorithm used is phoneme split without phoneme weights.

- **createkm4 <out>.km <train>.dat <K> <phoneN>**
  Performs K-means clustering using <train>.dat as training vectors. <K> is the no. of clusters and <phoneN> is the no. of phonemes. The K-means data is stored in <out>.km. Algorithm used is phoneme split with phoneme weights.

- **createkm5 <out>.km <train>.dat <K> <phoneN>**
  Performs K-means clustering using <train>.dat as training vectors. <K> is the no. of clusters and <phoneN> is the no. of phonemes. The K-means data is stored in <out>.km. Algorithm used is independent clustering with equal phoneme emphasis.

- **createkm6 <out>.km <train>.dat <K> <phoneN>**
  Performs K-means clustering using <train>.dat as training vectors. <K> is the no. of clusters and <phoneN> is the no. of phonemes. The K-means data is stored in <out>.km. Algorithm used is independent clustering with proportional phoneme emphasis.

- **createkm7 <out>.km <train>.dat <K> <phoneN>**
  Performs K-means clustering using <train>.dat as training vectors. <K> is the no. of clusters and <phoneN> is the no. of phonemes. The K-means data is stored in <out>.km. Algorithm used is clustering after independent clustering with equal phoneme emphasis.

- **createkm8 <out>.km <train>.dat <K> <phoneN>**
  Performs K-means clustering using <train>.dat as training vectors. <K> is the no. of clusters and <phoneN> is the no. of phonemes. The K-means data is stored in <out>.km. Algorithm used is clustering after independent clustering with proportional phoneme emphasis.

- **testkm <in>.km <test>.dat <class-lbl>.dat**
  Classifies vectors given by <test>.dat into their clusters. <in>.km is the K-means data. The cluster indices and phoneme labels are stored in <class-lbl>.dat.

- **createpc <out>.pc <class-lbl>.dat <K> <phoneN>**
  Creates phoneme classifier statistics file <out>.pc. Cluster indices and phoneme labels are given as <class-lbl>.dat. <K> is the no. of clusters and <phoneN> is the no. of phonemes.

- **testpc <in>.km <in>.pc <test>.dat <out>.dat <Knn>**
  Performs phonetic classification on test vectors <test>.dat. K-means data and phoneme classifier statistics are given by <in>.km and <in>.pc respectively. <Knn> is the k parameter in the k-nearest neighbor method. Classification results and phoneme labels are stored in <out>.dat.

- **initsc <in>.pc <out>.sc**
  Initializes score file <out>.sc. Phoneme classifier statistics file <in>.pc provides the no. of phonemes.

- **updatesc <in/out>.sc <result>.dat**
  Updates score file <in/out>.sc using the classification results and phoneme labels given by <result>.dat.

- **ga <out>.ga <train>.dat <K> <phoneN>**
  Performs genetic algorithm to determine optimal weights of coefficients. Clustering is done on vectors in <train>.dat. <K> is the no. of clusters and <phoneN> is the no. of phonemes. Output is written into <out>.ga.

- **present <in>.sc**
  Presents score file <in>.sc in summarized text form.

- **cmatrix <in>.sc**
  Presents confusion matrix in text form. Score information is extracted from score file <in>.sc.

- **cmtexv <in>.sc**
  Presents vowel confusion matrix in LaTeX form. Score information is extracted from score file <in>.sc.

- **cmtexc <in>.sc**
  Presents consonant confusion matrix in LaTeX form. Score information is extracted from score file <in>.sc.

- **cmtexp <in>.sc**
  Presents phoneme confusion matrix in LaTeX form. Score information is extracted from score file <in>.sc.

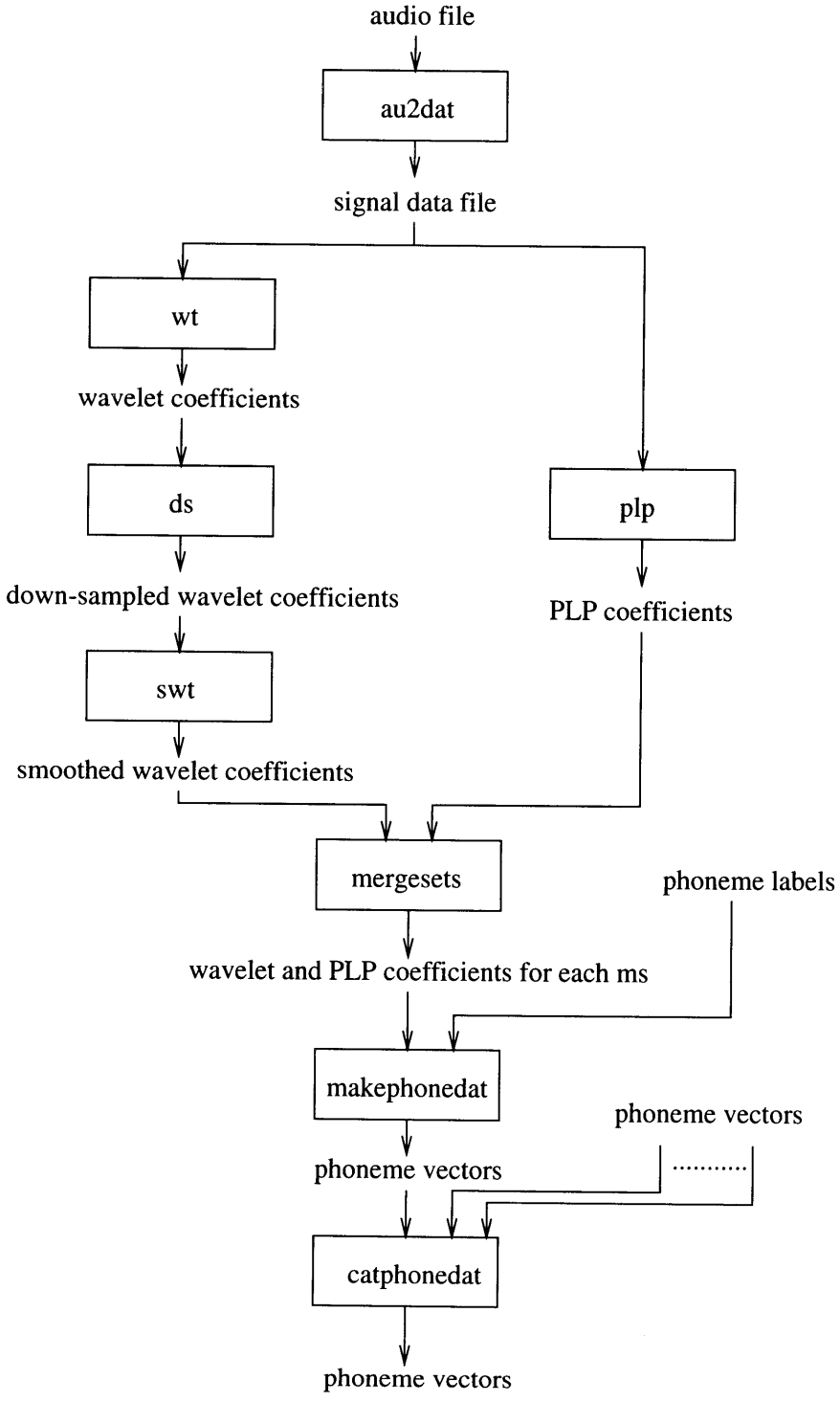

Figure A-1: Data flowchart for signal processing

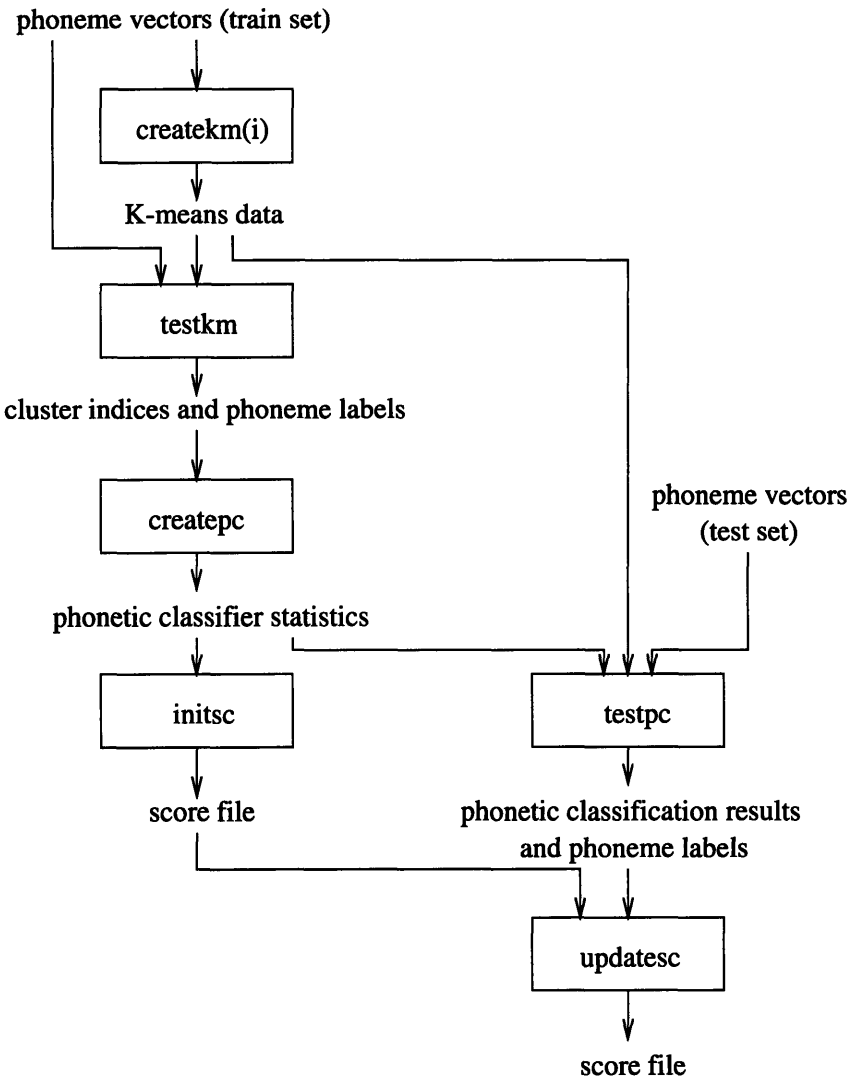

Figure A-2: Data flowchart for training and testing the phonetic classifier

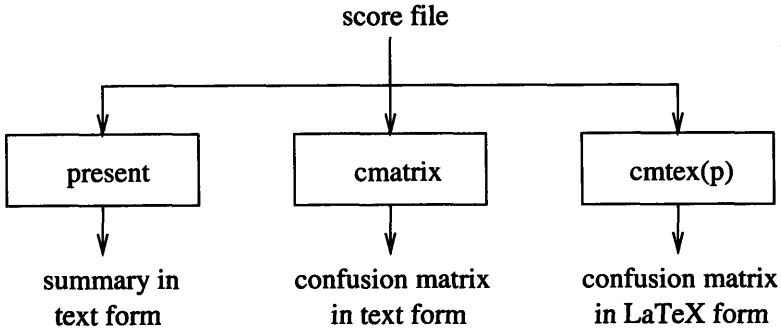

Figure A-3: Data flowchart for presenting classification results

Appendix B

Source Code

B.1 common.h

```
/*************************************************************************\
*                                                                         *
*                          Phonetic Classifier                            *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                  and                                    *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                 at the                                  *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                              May 15, 97                                 *
\*************************************************************************/


/* Common functions, includes and types */

#ifndef _COMMON_H
#define _COMMON_H

/* Set this to 1 if data is to be saved in MATLAB format */
#define MATLAB     0

/* Set this to 0 and 1 for machines using different integer formats */
#define BIGENDIAN  1

#include <stdio.h>
#include <string.h>
```

```
#include <math.h>
#include <limits.h>

#if MATLAB
#include "mat.h"
#endif

#define MAX_LBL    1000     /* Maximum no. of labels in label file */
#define MIN_GAMMA  0.0001   /* Minimum gamma for vector normalization */


/* Label structure */
typedef struct {
  int phone;  /* Phoneme label */
  int start;  /* Start time */
  int end;    /* End time */
} lbl_t;

/* K-means structure */
typedef struct {
  int K;         /* No. of clusters */
  int dim;       /* Vector dimension */
  float *mu;     /* Vector normalization parameter.  ith coefficient v[i] */
  float *gamma;  /* is normalized as (v[i] - mu[i]) / gamma[i] */
  float *meanv;  /* The centroid vectors */
} km_t;

/* Phoneme classifier structure */
typedef struct {
  int K;                 /* No. of clusters */
  int phoneN;            /* No. of phonemes */
  float *score;          /* Scores */
  unsigned long *count;  /* Token count */
} pc_t;

/* Score structure */
typedef struct {
  int phoneN;  /* No. of phonemes */
  int *count;  /* Score counts */
} sc_t;

/* Gene structure for genetic algorithm */
typedef struct {
  float fitness;  /* Fitness value */
  float *gamma;   /* The quantity to be optimized */
} gene_t;


int round(float x);

float square(float x);

#if BIGENDIAN
void flipint(int *x);
```

```
void flipfloat(float *x);
#endif

void readDat(char filename[], float **data, int *m, int *n);

void writeDat(char filename[], char datName[], float *data, int m, int n);

int readLbl(char filename[], lbl_t lbl[]);

void transpose(float **vector, float *data, int vectorN, int dim);

void clearVectors(float *vectors, int K, int dim);

void findNorm(km_t *km, float *vector, int vectorN, int dimp1, float gamma[]);

void normalize(float vector[], int dim, float mu[], float gamma[]);

float distance(float vector1[], float vector2[], int dim);

int nearestNeighbor(float vector[], float *meanv, int K, int dim);

void readKm(char filename[], km_t *km);

void writeKm(char filename[], km_t *km);

void readPc(char filename[], pc_t *pc);

void writePc(char filename[], pc_t *pc);

void readSc(char filename[], sc_t *sc);

void writeSc(char filename[], sc_t *sc);

#endif /* _COMMON_H */
```

B.2 common.c

```
/*********************************************************************\
*                                                                     *
*                          Phonetic Classifier                        *
*                                                                     *
*                    Speech Recognition by Clustering                 *
*                      Wavelet and PLP Coefficients                   *
*                                                                     *
*               Thesis submitted in partial fulfillment of            *
*                   the requirements for the degrees of               *
*        Bachelor of Science in Computer Science and Engineering      *
*                                  and                                *
*  Master of Engineering in Electrical Engineering and Computer Science *
*                                at the                               *
*                 Massachusetts Institute of Technology               *
*                                                                     *
```

```
*                  Chiangkai Er <changkai@alum.mit.edu>                 *
*                                May 15, 97                               *
\*************************************************************************/


/* Common functions, includes and types */

#include "common.h"


int round(float x) {
  /* Effects: Rounds x to the nearest integer. */
  /* Returns: x rounded to the nearest integer */
  int r;

  r = (int)(x + 0.5);
  if (x >= 0.0) {
    return r;
  }
  else {
    return --r;
  }
}


float square(float x) {
  /* Effects: Squares x. */
  /* Returns: Square of x */

  return x * x;
}


#if BIGENDIAN
void flipint(int *x) {
  /* Effects: Converts x between Big and Little Endian formats. */
  /* Modifies: x */
  char *byte;
  char temp;

  byte = (char *)x;

  temp = byte[0];
  byte[0] = byte[3];
  byte[3] = temp;

  temp = byte[1];
  byte[1] = byte[2];
  byte[2] = temp;
}


void flipfloat(float *x) {
  /* Effects: Converts x between Big and Little Endian formats. */
```

```
  /* Modifies: x */
  char *byte;
  char temp;

  byte = (char *)x;

  temp = byte[0];
  byte[0] = byte[3];
  byte[3] = temp;

  temp = byte[1];
  byte[1] = byte[2];
  byte[2] = temp;
}
#endif


#if MATLAB
void readDat(char filename[], float **data, int *m, int *n) {
  /* Effects: Reads MATLAB matrix file (filename) into *data, while
     storing the no. of rows and no. of columns in *m and *n respectively. */
  /* Modifies: *data, *m, *n */
  int i;
  double *ddata;  /* data as doubles */
  MATFile *fin;
  Matrix *mat;
  int mn;

  if (!(fin = matOpen(filename, "r"))) {
    fprintf(stderr, "readDat: Cannot open %s\n", filename);
    exit(-1);
  }
  mat = matGetNextMatrix(fin);
  *m = mxGetM(mat);
  *n = mxGetN(mat);
  mn = (*m) * (*n);
  ddata = (double *)malloc(mn * sizeof(double));
  memcpy(*data, mxGetPr(mat), mn * sizeof(double));
  matClose(fin);
  mxFreeMatrix(mat);

  /* Transfer ddata (doubles) to *data (floats) */
  *data = (float *)malloc(mn * sizeof(float));
  for (i=0; i<mn; i++) {
    (*data)[i] = ddata[i];
  }
  free(ddata);
}


void writeDat(char filename[], char datName[], float *data, int m, int n) {
  /* Effects: Writes data into MATLAB matrix file (filename).  No. of
     rows and no. of columns are given by m and n respectively.  Name of
     the matrix is given by datName. */
```

```
  int i;
  double *ddata;  /* data as doubles */
  MATFile *fout;
  Matrix *mat;

  /* Transfer data (floats) to ddata (doubles) */
  ddata = (double *)malloc(m * n * sizeof(double));
  for (i=0; i<m*n; i++) {
    ddata[i] = data[i];
  }

  if (!(fout = matOpen(filename, "w"))) {
    fprintf(stderr, "writeDat: Cannot create %s\n", filename);
    exit(-1);
  }
  mat = mxCreateFull(m, n, REAL);
  memcpy(mxGetPr(mat), ddata, m * n * sizeof(double));
  mxSetName(mat, datName);
  matPutMatrix(fout, mat);
  matClose(fout);
  mxFreeMatrix(mat);
  free(ddata);
}


#else /* !MATLAB */
void readDat(char filename[], float **data, int *m, int *n) {
  /* Effects: Reads data file (filename) created by writeDat into *data.
     No. of rows and no. of columns are stored in *m and *n respectively. */
  /* Modifies: *data, *m, *n */
  int i;
  FILE *fin;
  int fd;

  if (!(fin = fopen(filename, "r"))) {
    fprintf(stderr, "readDat: Cannot open %s\n", filename);
    exit(-1);
  }
  fd = fileno(fin);
  read(fd, m, sizeof(int));
  read(fd, n, sizeof(int));

#if BIGENDIAN
  /* Convert between Endian formats */
  flipint(m);
  flipint(n);
#endif

  *data = (float *)malloc((*m) * (*n) * sizeof(float));
  read(fd, *data, (*m) * (*n) * sizeof(float));
  fclose(fin);

#if BIGENDIAN
  /* Convert between Endian formats */
```

```
  for (i=0; i<(*m)*(*n); i++) {
    flipfloat(*data + i);
  }
#endif
}


void writeDat(char filename[], char datName[], float *data, int m, int n) {
  /* Effects: Writes data into data file (filename) which can be read by
     readDat.  No. of rows and no. of columns are given by m and n
     respectively.  datName is ignored. */
  int i;
  FILE *fout;
  int fd;
# if BIGENDIAN
  int mf, nf;  /* Flipped m and n */
#endif

  if (!(fout = fopen(filename, "w"))) {
    fprintf(stderr, "writeDat: Cannot open %s\n", filename);
    exit(-1);
  }
  fd = fileno(fout);

#if BIGENDIAN
  mf = m;
  nf = n;
  flipint(&mf);
  flipint(&nf);
  write(fd, &mf, sizeof(int));
  write(fd, &nf, sizeof(int));
  for (i=0; i<m*n; i++) {
    flipfloat(data + i);
  }
#else
  write(fd, &m, sizeof(int));
  write(fd, &n, sizeof(int));
#endif

  write(fd, data, m * n * sizeof(float));
  fclose(fout);
}
#endif /* MATLAB */


int readLbl(char filename[], lbl_t lbl[]) {
  /* Effects: Reads label file (filename) into lbl. */
  /* Modifies: lbl */
  /* Returns: No. of labels */
  FILE *lblFile;
  int lblN;
  int x;

  if (!(lblFile = fopen(filename, "r"))) {
```

```
    fprintf(stderr, "readLbl: Cannot open %s\n", filename);
    exit(-1);
  }
  lblN = 0;
  while ((lblN < MAX_LBL) &&
         (fscanf(lblFile, "%d", &x) != EOF)) {
    lbl[lblN].phone = x;
    if (fscanf(lblFile, "%d", &x) == EOF) {
      fprintf(stderr, "readLbl: Wrong file format\n");
      exit(-1);
    }
    lbl[lblN].start = x;
    if (fscanf(lblFile, "%d", &x) == EOF) {
      fprintf(stderr, "readLbl: Wrong file format\n");
      exit(-1);
    }
    lbl[lblN].end = x;
    lblN++;
  }
  fclose(lblFile);
  return lblN;
}


void transpose(float **vector, float *data, int vectorN, int dim) {
  /* Effects: Performs the transpose operation on data and stores the
     result in *vector.  No. of rows and columns in data are given
     by vectorN and dim respectively. */
  /* Modifies: *vector */
  int i, j;

  *vector = (float *)malloc(vectorN * dim * sizeof(float));
  for (i=0; i<vectorN; i++) {
    for (j=0; j<dim; j++) {
      (*vector + i*dim)[j] = (data + j*vectorN)[i];
    }
  }
}


void clearVectors(float *vectors, int K, int dim) {
  /* Effects: Clears all values in vectors.  K is the no. of vectors while
     dim is the vector dimension. */
  /* Modifies: vectors */
  int i, j;

  for (i=0; i<K; i++) {
    for (j=0; j<dim; j++) {
      (vectors + i*dim)[j] = 0.0;
    }
  }
}
```

```
void findNorm(km_t *km, float *vector,
              int vectorN, int dimp1, float gamma[]) {
  /* Effects: Performs normalization on vector by finding the normalization
     parameters mu and gamma to be used.  mu[i] is the mean of the ith
     coefficient while gamma[i] is the root-mean-square of the ith
     coefficient (if gamma is null).  However, if gamma is not null, then
     gamma[i] takes what is given.  Normalization parameters are stored
     in km. vectorN is the no. of vectors and dimp1 is the vector
     dimension plus 1. */
  /* Modifies: km, vector */
  int i, j;
  float sum, sumsqr, rms;  /* Sum, sum of squares, root-mean-square */

  /* For each vector coefficient */
  for (i=0; i<dimp1-1; i++) {
    /* Obtain mu */
    sum = 0.0;
    for (j=0; j<vectorN; j++) {
      sum += (vector + j*dimp1)[i];
    }
    km->mu[i] = sum / vectorN;
    /* Normalize vector with mu */
    for (j=0; j<vectorN; j++) {
      (vector + j*dimp1)[i] -= km->mu[i];
    }

    /* Obtain gamma */
    if (gamma) {
      /* Gamma is given */
      km->gamma[i] = gamma[i];
    }
    else {
      /* Gamma not given, so compute root-mean-square */
      sumsqr = 0.0;
      for (j=0; j<vectorN; j++) {
        sumsqr += square((vector + j*dimp1)[i]);
      }
      rms = sqrt(sumsqr / vectorN);
      /* Check if rms is too small */
      if (rms < MIN_GAMMA) {
        km->gamma[i] = MIN_GAMMA;
      }
      else {
        km->gamma[i] = rms;
      }
    }
    /* Normalize vector with gamma */
    for (j=0; j<vectorN; j++) {
      (vector + j*dimp1)[i] /= km->gamma[i];
    }
  }
}
```

```
void normalize(float vector[], int dim, float mu[], float gamma[]) {
  /* Effects: Performs vector normalization on vector, which has dim
     dimensions.  Normalization parameters are given by mu and gamma. */
  /* Modifies: vector */
  int i;

  for (i=0; i<dim; i++) {
    vector[i] = (vector[i] - mu[i]) / gamma[i];
  }
}


float distance(float vector1[], float vector2[], int dim) {
  /* Effects: Computes distance between vector1 and vector2, both having
     dim dimensions. */
  /* Returns: Distance between vector1 and vector2 */
  int i;
  float d;

  d = 0.0;
  for (i=0; i<dim; i++) {
    d += square(vector1[i] - vector2[i]);
  }
  return d;
}


int nearestNeighbor(float vector[], float *meanv, int K, int dim) {
  /* Effects: Find the centroid in meanv that is nearest to vector.
     meanv contains K vectors, and all vectors dim-dimensional. */
  /* Returns: The index corresponding to the nearest centroid in meanv */
  int i;
  int nn;
  float dist, mindist;

  nn = 0;
  mindist = FLT_MAX;
  for (i=0; i<K; i++) {
    dist = distance(vector, meanv + i*dim, dim);
    if (dist < mindist) {
      nn = i;
      mindist = dist;
    }
  }
  return nn;
}


void readKm(char filename[], km_t *km) {
  /* Effects: Reads K-means file (filename) and stores it in km. */
  /* Modifies: km */
  int i, j;
  FILE *kmFile;
```

```
  if (!(kmFile = fopen(filename, "r"))) {
    fprintf(stderr, "readKm: Cannot open %s\n", filename);
    exit(-1);
  }
  fscanf(kmFile, "%d", &(km->K));
  fscanf(kmFile, "%d", &(km->dim));
  /* Read mu values */
  km->mu = (float *)malloc(km->dim * sizeof(float));
  for (i=0; i<km->dim; i++) {
    fscanf(kmFile, "%f", &(km->mu[i]));
  }
  /* Read gamma values */
  km->gamma = (float *)malloc(km->dim * sizeof(float));
  for (i=0; i<km->dim; i++) {
    fscanf(kmFile, "%f", &(km->gamma[i]));
  }
  /* Read centroids */
  km->meanv = (float *)malloc(km->K * km->dim * sizeof(float));
  for (i=0; i<km->K; i++) {
    for (j=0; j<km->dim; j++) {
      fscanf(kmFile, "%f", &((km->meanv + i*km->dim)[j]));
    }
  }
  fclose(kmFile);
}


void writeKm(char filename[], km_t *km) {
  /* Effects: Writes contents of km into K-means file (filename). */
  int i, j;
  FILE *kmFile;

  if (!(kmFile = fopen(filename, "w"))) {
    fprintf(stderr, "writeKm: Cannot create %s\n", filename);
    exit(-1);
  }
  fprintf(kmFile, "%d\n", km->K);
  fprintf(kmFile, "%d\n", km->dim);
  /* Write mu values */
  for (i=0; i<km->dim; i++) {
    fprintf(kmFile, "%f ", km->mu[i]);
  }
  fprintf(kmFile, "\n");
  /* Write gamma values */
  for (i=0; i<km->dim; i++) {
    fprintf(kmFile, "%f ", km->gamma[i]);
  }
  fprintf(kmFile, "\n");
  /* Write centroids */
  for (i=0; i<km->K; i++) {
    for (j=0; j<km->dim; j++) {
      fprintf(kmFile, "%f ", (km->meanv + i*km->dim)[j]);
    }
    fprintf(kmFile, "\n");
```

```
  }
  fclose(kmFile);
}


void readPc(char filename[], pc_t *pc) {
  /* Effects: Reads phoneme classifier file (filename) and stores it
     in pc. */
  /* Modifies: pc */
  int i, p;
  int K, phoneN;
  FILE *pcFile;

  if (!(pcFile = fopen(filename, "r"))) {
    fprintf(stderr, "readPc: Cannot open %s\n", filename);
    exit(-1);
  }
  fscanf(pcFile, "%d\n", &K);
  pc->K = K;
  fscanf(pcFile, "%d\n", &phoneN);
  pc->phoneN = phoneN;
  pc->score = (float *)malloc(K * phoneN * sizeof(float));
  pc->count = (unsigned long *)malloc(K * sizeof(unsigned long));
  for (i=0; i<K; i++) {
    for (p=0; p<phoneN; p++) {
      fscanf(pcFile, "%f ", &((pc->score + i*phoneN)[p]));
    }
    fscanf(pcFile, "%lu ", &(pc->count[i]));
  }
  fclose(pcFile);
}


void writePc(char filename[], pc_t *pc) {
  /* Effects: Writes contents of pc into phoneme classifier
     file (filename). */
  int i, p;
  FILE *pcFile;

  if (!(pcFile = fopen(filename, "w"))) {
    fprintf(stderr, "writePc: Cannot create %s\n", filename);
    exit(-1);
  }
  fprintf(pcFile, "%d\n", pc->K);
  fprintf(pcFile, "%d\n", pc->phoneN);
  for (i=0; i<pc->K; i++) {
    for (p=0; p<pc->phoneN; p++) {
      fprintf(pcFile, "%f ", (pc->score + i*(pc->phoneN))[p]);
    }
    fprintf(pcFile, "%d\n", pc->count[i]);
  }
  fclose(pcFile);
}
```

```
void readSc(char filename[], sc_t *sc) {
  /* Effects: Reads score file (filename) and stores it in sc. */
  /* Modifies: sc */
  int i, j;
  int phoneN;
  FILE *scFile;

  if (!(scFile = fopen(filename, "r"))) {
    fprintf(stderr, "readSc: Cannot open %s\n", filename);
    exit(-1);
  }
  fscanf(scFile, "%d", &phoneN);
  sc->phoneN = phoneN;
  sc->count = (int *)malloc(phoneN * phoneN * sizeof(int));
  for (i=0; i<phoneN; i++) {
    for (j=0; j<phoneN; j++) {
      fscanf(scFile, "%d", &((sc->count + i*phoneN)[j]));
    }
  }
  fclose(scFile);
}


void writeSc(char filename[], sc_t *sc) {
  /* Effects: Writes contents of sc into score file (filename). */
  int i, j;
  int phoneN;
  FILE *scFile;

  if (!(scFile = fopen(filename, "w"))) {
    fprintf(stderr, "writeSc: Cannot create %s\n", filename);
    exit(-1);
  }
  phoneN = sc->phoneN;
  fprintf(scFile, "%d\n", phoneN);
  for (i=0; i<phoneN; i++) {
    for (j=0; j<phoneN; j++) {
      fprintf(scFile, "%d\t", (sc->count + i*phoneN)[j]);
    }
    fprintf(scFile, "\n");
  }
  fclose(scFile);
}
```

B.3 au2dat.c

```
/*************************************************************************\
*                                                                         *
*                          Phonetic Classifier                            *
*                                                                         *
*                   Speech Recognition by Clustering                      *
```

```
*                    Wavelet and PLP Coefficients                        *
*                                                                        *
*               Thesis submitted in partial fulfillment of               *
*                   the requirements for the degrees of                  *
*        Bachelor of Science in Computer Science and Engineering         *
*                                  and                                   *
*  Master of Engineering in Electrical Engineering and Computer Science  *
*                                at the                                  *
*                 Massachusetts Institute of Technology                  *
*                                                                        *
*                  Chiangkai Er <changkai@alum.mit.edu>                  *
*                              May 15, 97                                *
\************************************************************************/


/* Converts audio (au) file to data (dat) file */

#include "common.h"

#define MAX_LEN    1000000      /* Maximum length of audio file */
#define SCALE      1.0/32768


int etab[] = {0, 132, 396, 924, 1980, 4092, 8316, 16764};


float mu2lin(unsigned char mu) {
  /* Effects: Converts mu-law encoded 8-bit audio signal into a linear
     signal amplitude. */
  /* Returns: Linear signal amplitude */
  float y;
  int sig, e, f;

  mu = 255 - mu;
  sig = (mu > 127);
  e = mu/16 - 8*sig + 1;
  f = mu % 16;
  y = f * pow(2.0, e + 2);
  e = etab[e - 1];
  y = SCALE * (1 - 2*sig) * (e + y);
  return y;
}


void auRead(char filename[], float **data, int *len) {
  /* Effects: Reads audio file (filename) and converts it into linear signal
     amplitude stored in data.  The length of the signal is stored in len. */
  /* Modifies: *data, *len */
  int t;
  FILE *auFile;
  int fd;
  unsigned char audat[MAX_LEN];
  int aulen, k;
```

```
  if (!(auFile = fopen(filename, "r"))) {
    fprintf(stderr, "auRead: Cannot open %s\n", filename);
    exit(-1);
  }
  fd = fileno(auFile);
  aulen = read(fd, audat, MAX_LEN);
  fclose(auFile);

  /* Strip off file header */
  for (k=63; k>=0; k--) {
    if (audat[k] == 0) break;
  }
  k++;

  *len = aulen - k;
  *data = (float *)malloc((*len) * sizeof(float));
  /* Convert to mu-law audio signal to linear signal amplitude */
  for (t=0; t<(*len); t++) {
    (*data)[t] = mu2lin(audat[t + k]);
  }
}


void powerNorm(float data[], int len) {
  /* Effects: Power-normalizes speech waveform stored in data of
     length len. */
  /* Modifies: data */
  int t;
  float sumsqr, rms;  /* Sum of squares, Root-mean-square */

  sumsqr = 0.0;
  for (t=0; t<len; t++) {
    sumsqr += data[t] * data[t];
  }
  rms = sqrt(sumsqr / len);
  if (rms > 0) {
    for (t=0; t<len; t++) {
      data[t] = data[t] / rms;
    }
  }
}




main(int argc, char *argv[]) {
  float *data;
  int len;

  if (argc < 3) {
    fprintf(stderr, "Usage: %s <in>.au <out>.dat\n", argv[0]);
    exit(-1);
  }
  auRead(argv[1], &data, &len);
```

```
  powerNorm(data, len);
  writeDat(argv[2], "s", data, len, 1);
}
```

B.4 wt.c

```
/*************************************************************************\
*                                                                         *
*                          Phonetic Classifier                            *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                  and                                    *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                at the                                   *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                              May 15, 97                                 *
\*************************************************************************/


/* Performs wavelet transform */

#include "common.h"


int HaarTransform(float **wt, float *s, int len, int maxscales) {
  /* Effects: Performs Haar wavelet transform on signal s of length
     len.  maxscales is the maximum no. of scales needed.  The wavelet
     coefficients are stored in *wt. */
  /* Modifies: *wt */
  /* Returns: No. of scales */
  int scales;
  int i, j, t;
  int scale;
  float ave, detail;

  scales = (int)(log(len)/log(2));
  if (scales > maxscales) scales = maxscales;
  *wt = (float *)malloc(len * scales * sizeof(float));
  /* First pass (scale of 2) */
  for (t=0; t<len; t+=2) {
    ave = (s[t] + s[t+1]) / 2;
    detail = s[t] - ave;
    /* Average is stored temporarily, to be used by next scale */
    (*wt)[t] = ave;
    (*wt)[t+1] = detail;
```

```
  }
  scale = 2;
  /* Subsequent scales */
  for (i=1; i<scales; i++) {
    for (t=0; t<len; t+=2*scale) {
      ave = ((*wt + (i-1)*len)[t] + (*wt + (i-1)*len)[t + scale]) / 2;
      detail = (*wt + (i-1)*len)[t] - ave;
      /* Remove the averages in the previous scale */
      (*wt + (i-1)*len)[t] = (*wt + (i-1)*len)[t + 1];
      (*wt + (i-1)*len)[t + scale] = (*wt + (i-1)*len)[t + scale + 1];
      /* Average is stored temporarily, to be used by next scale */
      (*wt + i*len)[t] = ave;
      /* Repeat the detail coefficients */
      for (j=1; j<2*scale; j++) {
        (*wt + i*len)[t + j] = detail;
      }
    }
    scale *= 2;
  }
  /* Remove the last average */
  (*wt + (i-1)*len)[0] = (*wt + (i-1)*len)[1];
  return scales;
}



main(int argc, char *argv[]) {
  int i;
  float *s, *es;      /* Signal and extended signal */
  int m, n;
  int eslen, scales;  /* Extended length and no. of scales */
  int maxscales;
  float *wt;

  if (argc < 3) {
    fprintf(stderr, "Usage: %s <signal>.dat <wt>.dat [maxscales]\n", argv[0]);
    exit(-1);
  }
  if (argc >= 4) {
    maxscales = atoi(argv[3]);
  }
  else {
    maxscales = INT_MAX;
  }
  readDat(argv[1], &s, &m, &n);

  /* Extend the signal to make it a power of 2 */
  eslen = (int)pow(2, ceil(log(m * n)/log(2)));
  es = (float *)malloc(eslen * sizeof(float));
  for (i=0; i<m*n; i++) {
    es[i] = s[i];
  }
  for (i=m*n; i<eslen; i++) {
```

```
    es[i] = es[m*n - 1];
  }

  scales = HaarTransform(&wt, es, eslen, maxscales);
  writeDat(argv[2], "wt", wt, eslen, scales);
  free(s);
  free(es);
  free(wt);
}
```

B.5 ds.c

```
/*************************************************************************\
*                                                                         *
*                           Phonetic Classifier                           *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                    and                                  *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                  at the                                 *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                                May 15, 97                               *
\*************************************************************************/


/* Down-samples signal and wavelet transform */

#include "common.h"


int downSampleS(float **ds, float *s, int len, int step) {
  /* Effects: Down-samples signal s into *ds.  Length of s is len.
     Down-sampling step is given by step. */
  /* Modifies: *ds */
  /* Returns: Length of *ds */
  int t, i;

  *ds = (float *)malloc(len/step * sizeof(float));
  i = 0;
  for (t=0; t<=len-step; t+=step, i++) {
    (*ds)[i] = s[t];
  }
  return len/step;
}
```

```
void downSampleWt(float **dwt, float *wt,
                  int len, int wtlen, int scales, int step) {
  /* Effects: Down-samples wavelet transform wt into *dwt.  Length of
     wt is wtlen while length of original signal is len.  wt contains
     scales scales.  Down-sampling step is given by step. */
  /* Modifies: *dwt */
  int i, j;
  int t;
  float ave;

  *dwt = (float *)malloc(len/step * scales * sizeof(float));
  for (i=0; i<scales; i++) {
    for (t=0; t<len/step; t++) {
      ave = 0.0;
      for (j=0; j<step; j++) {
        ave += (wt + i*wtlen)[t*step + j];
      }
      (*dwt + i*(len/step))[t] = ave / step;
    }
  }
}



main(int argc, char *argv[]) {
  float *s, *wt;
  int len, temp, dlen;
  int wtlen, scales;
  float *ds, *dwt;

  if (argc < 6) {
    fprintf(stderr, "Usage: %s ", argv[0]);
    fprintf(stderr, "<s>.dat <wt>.dat <ds>.dat <dwt>.dat <step>\n");
    exit(-1);
  }
  readDat(argv[1], &s, &len, &temp);
  readDat(argv[2], &wt, &wtlen, &scales);
  dlen = downSampleS(&ds, s, len, atoi(argv[5]));
  downSampleWt(&dwt, wt, len, wtlen, scales, atoi(argv[5]));
  writeDat(argv[3], "ds", ds, dlen, 1);
  writeDat(argv[4], "dwt", dwt, dlen, scales);
  free(s);
  free(wt);
  free(ds);
  free(dwt);
}
```

B.6 swt.c

```
/*********************************************************************\
```

```
*                                                                     *
*                          Phonetic Classifier                        *
*                                                                     *
*                     Speech Recognition by Clustering                *
*                       Wavelet and PLP Coefficients                  *
*                                                                     *
*                 Thesis submitted in partial fulfillment of          *
*                     the requirements for the degrees of             *
*          Bachelor of Science in Computer Science and Engineering    *
*                                  and                                *
*  Master of Engineering in Electrical Engineering and Computer Science *
*                                 at the                              *
*                   Massachusetts Institute of Technology             *
*                                                                     *
*                    Chiangkai Er <changkai@alum.mit.edu>             *
*                                May 15, 97                           *
\*********************************************************************/


/* Converts wavelet coefficients into dB and performs smoothing operation */

#include "common.h"


int computeSwt(float **swt, float *wt, int len, int scales, int window) {
  /* Effects: Converts wavelet coefficients wt into dB and performs
     smoothing operation with window as the window size.  The result
     is stored in *swt.  The length of wt and the no. of scales are
     given by len and scales respectively. */
  /* Modifies: *swt */
  int i, j;
  int t, tj;
  int count;
  float sqr, ave;

  *swt = (float *)malloc(len * scales * sizeof(float));
  /* For each scale */
  for (i=0; i<scales; i++) {
    /* Convert to dB */
    for (t=0; t<len; t++) {
      sqr = (wt + i*len)[t] * (wt + i*len)[t];
      if (sqr > 1.0e-6) {
        (wt + i*len)[t] = 10 * log10(sqr);
      }
      else {
        /* If too small */
        (wt + i*len)[t] = -60.0;
      }
    }
    /* Perform smoothing operation */
    for (t=0; t<len; t++) {
      ave = 0.0;
      count = 0;
      for (j=-window/2; j<=window/2; j++) {
```

```
        tj = t + j;
        if ((tj >= 0) && (tj < len)) {
          ave += (wt + i*len)[tj];
          count++;
        }
      }
      (*swt + i*len)[t] = ave / count;
    }
  }
}



main(int argc, char *argv[]) {
  float *wt;
  int slen, scales;
  float *swt;

  if (argc < 4) {
    fprintf(stderr, "Usage: %s <wt>.dat <swt>.dat <window>\n", argv[0]);
    exit(-1);
  }
  readDat(argv[1], &wt, &slen, &scales);
  computeSwt(&swt, wt, slen, scales, atoi(argv[3]));
  writeDat(argv[2], "swt", swt, slen, scales);
  free(wt);
  free(swt);
}
```

B.7 plp.c

```
/*************************************************************************\
*                                                                         *
*                           Phonetic Classifier                           *
*                                                                         *
*                     Speech Recognition by Clustering                    *
*                       Wavelet and PLP Coefficients                      *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                    and                                  *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                   at the                                *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                                May 15, 97                               *
\*************************************************************************/
```

```
/* Performs PLP analysis */
/* Adapted from FORTRAN 77 code in
   "Perceptual linear predictive (PLP) analysis of speech"
   by H. Hermansky, J. Acoust. Soc. Am. 87(4):1738-1752, 1990 */

#include "common.h"

#define PLP_ORDER   5                   /* Order of PLP analysis */
#define CB_INIT     100                 /* Initial size of cb[] */
#define SAMP_FREQ   8000                /* Sampling frequency */
#define WINDOW_SIZE 160                 /* Window size */
#define DOWN_SAMP   (SAMP_FREQ / 1000)  /* Down-sampling step */


void audw(int npoint, int *nfilt,
          float **cb, int **ibegen, float sf) {
  /* Effects: Computes auditory weighting functions to be stored
     in *nfilt, *cb and *ibegen.  npoint is the no. of data points
     and sf is the sampling frequency. */
  /* Modifies: *nfilt, *cb, *ibegen */
  float fnqbar, f2samp, zdel, z0;
  float f0, fl, fh;
  float freq, x, z, fsq, rsss;
  int icount, i, j;
  int cbsize;

  fnqbar = 6.0 * log(sf/1200.0 + sqrt(square(sf/1200.0) + 1.0));
  *nfilt = round(fnqbar) + 2;
  cbsize = CB_INIT;
  *cb = (float *)malloc(cbsize * sizeof(float));
  *ibegen = (int *)malloc((*nfilt - 1) * 3 * sizeof(int));
  f2samp = (npoint - 1) / (sf/2.0);
  zdel = fnqbar / (*nfilt - 1);
  icount = 0;
  for (j=1; j<*nfilt-1; j++) {
    (*ibegen + j*3)[2] = icount;
    z0 = zdel * j;
    f0 = 600.0 * (exp(z0/6) - exp(-z0/6)) / 2.0;
    fl = 600.0 * (exp((z0 - 2.5)/6) - exp(-(z0 - 2.5)/6)) / 2.0;
    (*ibegen + j*3)[0] = round(fl * f2samp);
    if ((*ibegen + j*3)[0] < 0)
      (*ibegen + j*3)[0] = 0;
    fh = 600.0 * (exp((z0 + 1.3)/6) - exp(-(z0 + 1.3)/6)) / 2.0;
    (*ibegen + j*3)[1] = round(fh * f2samp);
    if ((*ibegen + j*3)[1] >= npoint)
      (*ibegen + j*3)[1] = npoint - 1;
    for (i=(*ibegen+j*3)[0]; i<=(*ibegen+j*3)[1]; i++) {
      freq = i / f2samp;
      x = freq / 600.0;
      z = 6.0 * log(x + sqrt(square(x) + 1.0));
      z = z - z0;
      if (icount >= cbsize) {
        cbsize += CB_INIT;
        *cb = (float *)realloc(*cb, cbsize * sizeof(float));
```

```
      }
      if (z <= -0.5) {
        (*cb)[icount] = pow(10.0, z + 0.5);
      }
      else if (z >= 0.5) {
        (*cb)[icount] = pow(10.0, -2.5 * (z - 0.5));
      }
      else {
        (*cb)[icount] = 1.0;
      }
      fsq = square(f0);
      rsss = square(fsq) * (fsq + square(1200.0)) /
        (square(fsq + square(400.0)) * (fsq + square(3100.0)));
      (*cb)[icount] = rsss * (*cb)[icount];
      icount++;
    }
  }
}


void hwind(float **weight, int npoint) {
  /* Effects: Computes Hamming window (for npoint points) weighting
     to be stored in *weight. */
  /* Modifies: *weight */
  int ii;

  *weight = (float *)malloc(npoint * sizeof(float));
  for (ii=0; ii<npoint; ii++) {
    (*weight)[ii] = 0.54 - 0.46*cos(2.0 * M_PI * ii / (npoint-1));
  }
}


void cosw(int m, int nfilt, float **wcos) {
  /* Effects: Computes cosine weightings for IDFT to be stored
     in *wcos. */
  /* Modifies: *wcos */
  int ii, jj;

  *wcos = (float *)malloc(nfilt * (m + 1) * sizeof(float));
  for (ii=0; ii<m+1; ii++) {
    for (jj=1; jj<nfilt-1; jj++) {
      (*wcos + jj*(m+1))[ii] =
        2.0 * cos(2.0 * M_PI * ii * jj / (2 * (nfilt-1)));
    }
    (*wcos + (nfilt-1)*(m+1))[ii] =
      cos(2.0 * M_PI * ii * jj / (2 * (nfilt-1)));
  }
}


void ffthr(float real[], float **power, int ll, int m) {
  /* Effects: Computes the power spectrum of real by FFT.  The
     spectrum is stored in *power. */
```

```
/* Modifies: *power */
int k1, n, n1, n2;
int l, le, le1;
int ii, i, j, id;
float t1, t2;
float u1, u2, u3, u4, w1, w2;
float *signal, *sigima;

n = (int)pow(2.0, m);
signal = (float *)malloc(n * sizeof(float));
sigima = (float *)malloc(n * sizeof(float));
*power = (float *)malloc(n * sizeof(float));
for (ii=0; ii<ll; ii++) {
  sigima[ii] = 0.0;
  signal[ii] = real[ii];
}
for (ii=ll; ii<n; ii++) {
  sigima[ii] = 0.0;
  signal[ii] = 0.0;
}
n2 = n / 2;
n1 = n - 1;
j = 1;
for (i=1; i<=n1; i++) {
  if (i < j) {
    t1 = signal[j - 1];
    t2 = sigima[j - 1];
    signal[j - 1] = signal[i - 1];
    sigima[j - 1] = sigima[i - 1];
    signal[i - 1] = t1;
    sigima[i - 1] = t2;
  }
  k1 = n2;
  while (k1 < j) {
    j = j - k1;
    k1 = k1 / 2;
  }
  j = j + k1;
}
for (l=1; l<=m; l++) {
  le = (int)pow(2.0, l);
  le1 = le / 2;
  u1 = 1.0;
  u2 = 0.0;
  w1 = cos(M_PI / le1);
  w2 = - sin(M_PI / le1);
  for (j=1; j<=le1; j++) {
    for (i=j; i<=n; i+=le) {
      id = i + le1;
      t1 = signal[id - 1]*u1 - sigima[id - 1]*u2;
      t2 = sigima[id - 1]*u1 + signal[id - 1]*u2;
      signal[id - 1] = signal[i - 1] - t1;
      sigima[id - 1] = sigima[i - 1] - t2;
      signal[i - 1] = signal[i - 1] + t1;
```

```
        sigima[i - 1] = sigima[i - 1] + t2;
      }
      u3 = u1;
      u1 = u1*w1 - u2*w2;
      u2 = u2*w1 + u3*w2;
    }
  }
  for (ii=0; ii<n; ii++) {
    (*power)[ii] = signal[ii]*signal[ii] + sigima[ii]*sigima[ii];
  }
  free(signal);
  free(sigima);
}


void plp(float speech[], int nwind, int m, float **a,
         float **rc, float *gain, float sf) {
  /* Effects: Computes the mth order PLP model, given by m+1 autoregressive
     coefficients *a or by m reflection coefficients *rc and model
     gain *gain.  The speech signal is given by speech, with nwind
     samples and a sampling frequency of sf. */
  /* Modifies: *a, *rc, *gain */
  static int icall = 0;
  static int nwind0, m0;
  static float sf0;
  static float *hwei, *cb, *wcos;
  static int *ibegen;
  static int nfft, npoint, nfilt;
  float *r, *spectr, *alp, *audspe;
  int ii, i, ib, ip, idx;
  int jfilt, kk, ll;
  int icb, nspt;
  int mct, mct2, mh;
  float aib, aip, alpmin;
  float s, rcmct;

  *a = (float *)malloc((m + 1) * sizeof(float));
  *rc = (float *)malloc(m * sizeof(float));
  r = (float *)malloc((m + 1) * sizeof(float));
  alp = (float *)malloc((m + 1) * sizeof(float));

  if (icall == 0) {
    hwind(&hwei, nwind);
    nfft = (int)ceil(log(nwind) / log(2.0));
    npoint = (int)(pow(2.0, nfft)) / 2 + 1;
    audw(npoint, &nfilt, &cb, &ibegen, sf);
    cosw(m, nfilt, &wcos);
    nwind0 = nwind;
    m0 = m;
    sf0 = sf;
    icall = 1;
  }
  else if ((nwind != nwind) || (m0 != m) || (sf0 != sf)) {
    free(hwei);
```

```
  free(cb);
  free(ibegen);
  free(wcos);
  hwind(&hwei, nwind);
  nfft = (int)ceil(log(nwind) / log(2.0));
  npoint = (int)(pow(2.0, nfft)) / 2 + 1;
  audw(npoint, &nfilt, &cb, &ibegen, sf);
  cosw(m, nfilt, &wcos);
  nwind0 = nwind;
  m0 = m;
  sf0 = sf;
}
for (ii=0; ii<nwind; ii++) {
  speech[ii] = hwei[ii] * speech[ii];
}
ffthr(speech, &spectr, nwind, nfft);
audspe = (float *)malloc(nfilt * sizeof(float));
for (jfilt=1; jfilt<nfilt-1; jfilt++) {
  audspe[jfilt] = 0.0;
  for (kk=(ibegen+jfilt*3)[0]; kk<=(ibegen+jfilt*3)[1]; kk++) {
    icb = (ibegen + jfilt*3)[2] - (ibegen + jfilt*3)[0] + kk;
    audspe[jfilt] = audspe[jfilt] + spectr[kk] * cb[icb];
  }
}
for (ii=1; ii<nfilt-1; ii++) {
  audspe[ii] = pow(audspe[ii], 0.33);
}
audspe[0] = audspe[1];
audspe[nfilt - 1] = audspe[nfilt - 2];
nspt = 2 * (nfilt - 1);
for (kk=0; kk<m+1; kk++) {
  r[kk] = audspe[0];
  for (ll=1; ll<nfilt; ll++) {
    r[kk] = r[kk] + audspe[ll] * (wcos + ll*(m+1))[kk];
  }
  r[kk] = r[kk] / nspt;
}
(*a)[0] = 1.0;
alp[0] = r[0];
(*rc)[0] = - r[1] / r[0];
(*a)[1] = (*rc)[0];
alp[1] = r[0] + r[1] * (*rc)[0];
for (mct=2; mct<=m; mct++) {
  s = 0.0;
  mct2 = mct + 2;
  alpmin = alp[mct - 1];
  for (ip=1; ip<=mct; ip++) {
    idx = mct2 - ip;
    s = s + r[idx - 1] * (*a)[ip - 1];
  }
  rcmct = - s / alpmin;
  mh = mct / 2 + 1;
  for (ip=2; ip<=mh; ip++) {
    ib = mct2 - ip;
```

```
      aip = (*a)[ip - 1];
      aib = (*a)[ib - 1];
      (*a)[ip - 1] = aip + rcmct * aib;
      (*a)[ib - 1] = aib + rcmct * aip;
    }
    (*a)[mct] = rcmct;
    alp[mct] = 0.0;
    alp[mct] = alpmin - alpmin * rcmct * rcmct;
    (*rc)[mct - 1] = rcmct;
  }
  *gain = alp[m];
  free(r);
  free(spectr);
  free(alp);
  free(audspe);
}


int computePLP(float s[], int len, int m, float **ceps) {
  /* Effects: Computes the mth order PLP cepstral coefficients from
     signal s of length len.  The coefficients are stored in *ceps. */
  /* Modifies: *ceps */
  int i, j, k;
  int windowN;      /* No. of windows */
  int plplen;       /* No. of sets of PLP coefficients */
  int rep;          /* No. of repetitions */
  float *awind;     /* PLP autoregressive coefficients */
  float *rc;        /* PLP reflection coefficients */
  float gain;       /* PLP model gain */
  float csum;       /* Cepstral sum (for cepstral computation) */

  plplen = len / (DOWN_SAMP);
  rep = (WINDOW_SIZE) / (DOWN_SAMP);
  windowN = plplen / rep;
  if (plplen % rep > 0) windowN++;
  *ceps = (float *)malloc(plplen * (m + 1) * sizeof(float));
  /* For each window */
  for (i=0; i<windowN; i++) {
    if (i < windowN - 1) {
      /* Not last window */
      plp(s + i*(WINDOW_SIZE), (WINDOW_SIZE),
          m, &awind, &rc, &gain, (SAMP_FREQ));
    }
    else {
      /* Last window */
      plp(s + i*(WINDOW_SIZE), len - i*(WINDOW_SIZE),
          m, &awind, &rc, &gain, (SAMP_FREQ));
    }

    /* Compute cepstral coefficients from autoregressive coefficients */
    (*ceps)[i*rep] = log(gain);
    (*ceps + plplen)[i*rep] = - awind[1];
    for (j=2; j<m+1; j++) {
      csum = 0.0;
```

```
      for (k=1; k<j; k++) {
        csum += k * (*ceps + k*plplen)[i*rep] * awind[j - k];
      }
      csum /= j;
      (*ceps + j*plplen)[i*rep] = - (awind[j] + csum);
    }

    /* Repeat the coefficients for every millisecond */
    if (i < windowN - 1) {
      /* Not last window */
      for (j=1; j<rep; j++) {
        for (k=0; k<m+1; k++) {
          (*ceps + k*plplen)[i*rep + j] = (*ceps + k*plplen)[i*rep];
        }
      }
    }
    else {
      /* Last window */
      for (j=1; j<plplen-i*rep; j++) {
        for (k=0; k<m+1; k++) {
          (*ceps + k*plplen)[i*rep + j] = (*ceps + k*plplen)[i*rep];
        }
      }
    }
    free(awind);
    free(rc);
  }
  return plplen;
}



main(int argc, char *argv[]) {
  float *s;
  float *plp;
  int slen, plplen;
  int temp;

  if (argc < 3) {
    fprintf(stderr, "Usage: %s <signal>.dat <plp>.dat\n", argv[0]);
    exit(-1);
  }
  readDat(argv[1], &s, &slen, &temp);
  plplen = computePLP(s, slen, PLP_ORDER, &plp);
  writeDat(argv[2], "plp", plp, plplen, PLP_ORDER + 1);
  free(s);
  free(plp);
}
```

B.8 catsets.c

```
/*************************************************************************\
*                                                                         *
*                          Phonetic Classifier                            *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                    and                                  *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                  at the                                 *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                               May 15, 97                                *
\*************************************************************************/


/* Concatenates two data sets */

#include "common.h"


void concatenate(float *data1, int m1, int n1,
                 float *data2, int m2, int n2,
                 float **data3, int *m3, int *n3) {
  /* Effects: Concatenates data1 and data2 to form *data3.  data1 has
     dimensions of m1 and n1; data2 has dimensions of m2 and n2;
     *data3 has dimensions of *m3 and *n3. */
  /* Modifies: *data3, *m3, *n3 */
  int i, j;

  if (n1 != n2) {
    fprintf(stderr, "concatenate: Sets must have the same width\n");
    exit(-1);
  }
  *m3 = m1 + m2;  /* Length is sum of lengths */
  *n3 = n1;       /* Scale remains the same */
  *data3 = (float *)malloc((*m3) * (*n3) * sizeof(float));
  for (i=0; i<n1; i++) {
    for (j=0; j<m1; j++) {
      (*data3 + i*(*m3))[j] = (data1 + i*m1)[j];
    }
  }
  for (i=0; i<n2; i++) {
    for (j=0; j<m2; j++) {
      (*data3 + i*(*m3))[j + m1] = (data2 + i*m2)[j];
    }
  }
}
```

```
main(int argc, char *argv[]) {
  float *data1;
  int len1, scales1;
  float *data2;
  int len2, scales2;
  float *data3;
  int len3, scales3;

  if (argc < 4) {
    fprintf(stderr, "Usage: %s ", argv[0]);
    fprintf(stderr, "<set1>.dat <set2>.dat <concatenated>.dat\n");
    exit(-1);
  }
  readDat(argv[1], &data1, &len1, &scales1);
  readDat(argv[2], &data2, &len2, &scales2);
  concatenate(data1, len1, scales1,
              data2, len2, scales2,
              &data3, &len3, &scales3);
  writeDat(argv[3], "set", data3, len3, scales3);
  free(data1);
  free(data2);
  free(data3);
}
```

B.9 mergesets.c

```
/*************************************************************************\
*                                                                         *
*                           Phonetic Classifier                           *
*                                                                         *
*                     Speech Recognition by Clustering                    *
*                       Wavelet and PLP Coefficients                      *
*                                                                         *
*                 Thesis submitted in partial fulfillment of              *
*                     the requirements for the degrees of                 *
*         Bachelor of Science in Computer Science and Engineering         *
*                                    and                                  *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                  at the                                 *
*                   Massachusetts Institute of Technology                 *
*                                                                         *
*                    Chiangkai Er <changkai@alum.mit.edu>                 *
*                                 May 15, 97                              *
\*************************************************************************/


/* Merges two data sets */
```

```
#include "common.h"


void merge(float *data1, int m1, int n1,
           float *data2, int m2, int n2,
           float **data3, int *m3, int *n3) {
  /* Effects: Merges data1 and data2 to form *data3.  data1 has m1
     rows and n1 columns; data2 has m2 rows and n2 columns.  The
     no. of rows and columns of *data3 are stored in *m3 and *n3
     respectively. */
  /* Modifies: *data3, *m3, *n3 */
  int i, j;

  *m3 = m1;
  *n3 = n1 + n2;
  *data3 = (float *)malloc((*m3) * (*n3) * sizeof(float));
  for (i=0; i<n1; i++) {
    for (j=0; j<m1; j++) {
      (*data3 + i*m1)[j] = (data1 + i*m1)[j];
    }
  }
  for (i=0; i<n2; i++) {
    for (j=0; j<m2; j++) {
      (*data3 + (i+n1)*m1)[j] = (data2 + i*m2)[j];
    }
  }
}




main(int argc, char *argv[]) {
  float *data1;
  int len1, dim1;
  float *data2;
  int len2, dim2;
  float *data3;
  int len3, dim3;

  if (argc < 4) {
    fprintf(stderr, "Usage: %s <set1>.dat <set2>.dat <merged>.dat\n", argv[0]);
    exit(-1);
  }
  readDat(argv[1], &data1, &len1, &dim1);
  readDat(argv[2], &data2, &len2, &dim2);
  if (len1 != len2) {
    fprintf(stderr, "%s: Sets must have the same length\n", argv[0]);
    exit(-1);
  }
  merge(data1, len1, dim1,
        data2, len2, dim2,
        &data3, &len3, &dim3);
  writeDat(argv[3], "set", data3, len3, dim3);
  free(data1);
```

```
  free(data2);
  free(data3);
}
```

B.10 makephonedat.c

```
/*************************************************************************\
*                                                                         *
*                           Phonetic Classifier                           *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                  and                                    *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                 at the                                  *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                              May 15, 97                                 *
\*************************************************************************/


/* Makes phoneme data for clustering and classification */

#include "common.h"


void phoneData(float *data1, int len1, int dim1,
               float **data2, int *len2, int *dim2,
               lbl_t lbl[], int lblN) {
  /* Effects: Creates phoneme data from millisecond-data (data1)
     and phoneme labels (lbl).  data1 has length len1 and dim1
     dimensions.  lbl has lblN labels.  The phoneme data is
     stored in *data2 with length *len2 and *dim2 dimensions.
     The last coefficient in each set of phoneme data is the
     phoneme label */
  /* Modifies: *data2, *len2, *dim2 */
  int i, t, s;
  int phonelbl;    /* Phoneme label */
  int phonedur;    /* Phoneme duration */
  int currdur;     /* Current duration */

  *len2 = lblN;
  *dim2 = dim1 * 3 + 1;
  *data2 = (float *)malloc((*len2) * (*dim2) * sizeof(float));
  i = 0;
  phonelbl = -1;
  /* For each millisecond */
```

```
for (t=0; t<len1; t++) {
  if (i < lblN) {
    /* More labels to go */
    if (phonelbl == -1) {
      /* Not in phoneme yet */
      if (t >= lbl[i].start) {
        /* New phoneme */
        phonelbl = lbl[i].phone;
        phonedur = lbl[i].end - lbl[i].start;
        currdur = 0;
        (*data2 + (*dim2 - 1)*(*len2))[i] = phonelbl;
      }
    }
    else {
      /* Currently in phoneme */
      if (t >= lbl[i].end) {
        /* End of phoneme? */
        phonelbl = -1;
        i++;
      }
    }
  }
  if (phonelbl >= 0) {
    /* Currently in phoneme */
    if (currdur < phonedur / 3) {
      /* Initial third */
      for (s=0; s<dim1; s++) {
        (*data2 + s*(*len2))[i] =
          ((*data2 + s*(*len2))[i] * currdur + (data1 + s*len1)[t]) /
          (currdur + 1);
      }
    }
    else if (currdur < phonedur * 2/3) {
      /* Middle third */
      for (s=0; s<dim1; s++) {
        (*data2 + (dim1 + s)*(*len2))[i] =
          ((*data2 + (dim1 + s)*(*len2))[i] * (currdur - phonedur/3) +
           (data1 + s*len1)[t]) /
          (currdur - phonedur/3 + 1);
      }
    }
    else {
      /* Final third */
      for (s=0; s<dim1; s++) {
        (*data2 + (2*dim1 + s)*(*len2))[i] =
          ((*data2 + (2*dim1 + s)*(*len2))[i] *
           (currdur - phonedur * 2/3) +
           (data1 + s*len1)[t]) /
          (currdur - phonedur*2/3 + 1);
      }
    }
    currdur++;
  }
}
```

```
}



main(int argc, char *argv[]) {
  float *data1;
  int len1, dim1;
  float *data2;
  int len2, dim2;
  lbl_t lbl[MAX_LBL];
  int lblN;

  if (argc < 4) {
    fprintf(stderr, "Usage: %s ", argv[0]);
    fprintf(stderr, "<set>.dat <label>.lbl <out>.dat\n");
    exit(-1);
  }
  readDat(argv[1], &data1, &len1, &dim1);
  lblN = readLbl(argv[2], lbl);
  phoneData(data1, len1, dim1,
            &data2, &len2, &dim2,
            lbl, lblN);
  writeDat(argv[3], "phone", data2, len2, dim2);
  free(data1);
  free(data2);
}
```

B.11 catphonedat.c

```
/*************************************************************************\
*                                                                         *
*                          Phonetic Classifier                            *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                  and                                    *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                  at the                                 *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                                May 15, 97                               *
\*************************************************************************/


/* Concatenates phoneme data for clustering and classification */
```

```
#include "common.h"

#define MAX_LINELEN  1000    /* Maximum length of a line of text */


void getDimensions(char listfilename[], char setdir[],
                   int *len, int *dim) {
  /* Effects: Obtains the dimensions of the concatenated file by adding up
     the lengths of the files in listfilename.  Each vector file in
     listfilename should have the same number of dimensions.  setdir is
     the directory in which the files can be found.  The length and no. of
     dimensions found are stored in *len and *dim respectively. */
  /* Modifies: *len, *dim */
  FILE *listFile, *dataFile;
  int fd;
  char datfile[MAX_LINELEN];
  char datfilename[MAX_LINELEN];
  int m, n;

  *len = 0;
  *dim = 0;
  if (!(listFile = fopen(listfilename, "r"))) {
    fprintf(stderr, "getDimensions: Cannot open %s\n", listfilename);
    exit(-1);
  }
  while (fgets(datfile, MAX_LINELEN, listFile)) {
    /* Remove newline character */
    datfile[strlen(datfile) - 1] = '\0';
    /* Form full filename */
    strcpy(datfilename, setdir);
    strcat(datfilename, "/");
    strcat(datfilename, datfile);
    strcat(datfilename, ".dat");
    printf("%s\n", datfilename);
    /* Open file */
    if (!(dataFile = fopen(datfilename, "r"))) {
      fprintf(stderr, "getDimensions: Cannot open %s\n", datfilename);
      exit(-1);
    }
    fd = fileno(dataFile);
    read(fd, &m, sizeof(int));
    read(fd, &n, sizeof(int));
    fclose(dataFile);
    if (*dim == 0) *dim = n;
    if (n != *dim) {
      fprintf(stderr, "getDimensions: Sets must have the same width\n");
      exit(-1);
    }
    *len += m;
  }
  close(listFile);
}
```

```
void copyData(float *data1, int len1, int base,
              float *data2, int len2, int dim) {
  /* Effects: Copies data from data2 to data1.  data2 has length len2 and
     dim dimensions.  data1 has length len1, and data is copied to data1
     starting from index base. */
  /* Modifies: data1 */
  int i, j;

  for (i=0; i<dim; i++) {
    for (j=0; j<len2; j++) {
      (data1 + i*len1)[j + base] = (data2 + i*len2)[j];
    }
  }
}




main(int argc, char *argv[]) {
  float *data;
  int len, dim;
  float *catdata;
  int catlen, catdim;
  int currbase;
  FILE *listFile;
  char datfile[MAX_LINELEN];
  char datfilename[MAX_LINELEN];

  if (argc < 4) {
    fprintf(stderr, "Usage: %s ", argv[0]);
    fprintf(stderr, "<datlist> <setdir> <concatenated>.dat\n");
    exit(-1);
  }
  getDimensions(argv[1], argv[2], &catlen, &catdim);
  catdata = (float *)malloc(catlen * catdim * sizeof(float));
  if (!(listFile = fopen(argv[1], "r"))) {
    fprintf(stderr, "%s: Cannot open %s\n", argv[0], argv[1]);
    exit(-1);
  }
  currbase = 0;
  while (fgets(datfile, MAX_LINELEN, listFile)) {
    datfile[strlen(datfile) - 1] = '\0';
    strcpy(datfilename, argv[2]);
    strcat(datfilename, "/");
    strcat(datfilename, datfile);
    strcat(datfilename, ".dat");
    printf("%s\n", datfilename);
    readDat(datfilename, &data, &len, &dim);
    copyData(catdata, catlen, currbase,
             data, len, dim);
    currbase += len;
    free(data);
  }
  close(listFile);
```

```
  writeDat(argv[3], "data", catdata, catlen, catdim);
  free(catdata);
}
```

B.12 createkm1.c

```
/*************************************************************************\
*                                                                         *
*                           Phonetic Classifier                           *
*                                                                         *
*                     Speech Recognition by Clustering                    *
*                       Wavelet and PLP Coefficients                      *
*                                                                         *
*                 Thesis submitted in partial fulfillment of              *
*                     the requirements for the degrees of                 *
*          Bachelor of Science in Computer Science and Engineering        *
*                                    and                                  *
*   Master of Engineering in Electrical Engineering and Computer Science  *
*                                    at the                               *
*                   Massachusetts Institute of Technology                 *
*                                                                         *
*                    Chiangkai Er <changkai@alum.mit.edu>                 *
*                                 May 15, 97                              *
\*************************************************************************/


/* Performs K-means clustering */
/* Binary split without phoneme weights */

#include "common.h"

#define ERR_THRES    0.001    /* Error threshold for centroids */
#define EPSILON      0.04     /* Epsilon used in splitting */


/* gamma parameter for vector normalization */
float kmgamma[] = {
  25.172358, 19.218266, 21.742237, 24.898365, 18.966608, 25.621672,
  33.172913, 37.476593, 1.096734, 0.230770, 0.337857, 0.225993,
  0.357310, 0.067926, 23.546637, 17.280016, 11.299080, 11.924601,
  17.581976, 24.429148, 32.420090, 36.827389, 1.184915, 0.251185,
  0.257460, 0.269493, 0.185793, 0.071770, 27.508614, 13.651873,
  10.648049, 15.503012, 11.959195, 27.137836, 34.634777, 38.795593,
  0.951037, 0.217823, 0.321228, 0.218384, 0.186709, 0.071181
};


void performClustering(km_t *km, int K, int phoneN, float *vector,
                       int vectorN, int dimp1, float gamma[]) {
  /* Effects: Performs clustering on vector, which contains vectorN
     vectors of (dimp1 - 1) dimensions.  K clusters are to be found
     and there are phoneN phonemes.  If gamma is not null, it is used
```

```
   for vector normalization.  vector is normalized before clustering
   begins.  Clustering results are stored in km */
/* Modifies: km, vector */
int i, j;
int dim;          /* No. of dimensions */
int N;            /* Current no. of centroids */
int nn;           /* Nearest neighbor */
float *centroid;  /* Current centroids */
float *count;     /* No. of vectors corresponding to each centroid */
int phoneme;      /* Current phoneme */
int iter;         /* No. of iterations */
float newval;     /* New value */
float error;

/* Initializations */
dim = dimp1 - 1;
km->K = K;
km->dim = dim;
km->mu = (float *)malloc(dim * sizeof(float));
km->gamma = (float *)malloc(dim * sizeof(float));
km->meanv = (float *)malloc(K * dim * sizeof(float));
centroid = (float *)malloc(K * dim * sizeof(float));
count = (float *)malloc(K * sizeof(float));
findNorm(km, vector, vectorN, dimp1, gamma);

N = 1;
do {
  if (N == 1) {
    for (i=0; i<dim; i++) {
      (km->meanv)[i] = 0.0;
    }
  }
  else {
    /* Binary split each centroid */
    for (i=N/2-1; i>=0; i--) {
      for (j=0; j<dim; j++) {
        if (N/2 + i >= K) {
          (km->meanv + (K - N/2 + i)*dim)[j] =
            (1 - EPSILON) * (km->meanv + i*dim)[j];
        }
        else {
          (km->meanv + 2*i*dim)[j] =
            (1 - EPSILON) * (km->meanv + i*dim)[j];
        }
      }
    }
    for (i=0; i<N/2; i++) {
      if (N/2 + i < K) {
        for (j=0; j<dim; j++) {
          (km->meanv + (2*i + 1)*dim)[j] =
            (1 + EPSILON) / (1 - EPSILON) * (km->meanv + 2*i*dim)[j];
        }
      }
    }
```

```
    }

    if (N > K) N = K;
    printf("Set Size = %d\n", N);
    /* Perform iterations */
    iter = 1;
    do {
      clearVectors(centroid, N, dim);
      for (i=0; i<N; i++) count[i] = 0.0;
      /* Find nearest centroids and compute new centroids */
      for (i=0; i<vectorN; i++) {
        phoneme = (int)(vector + i*dimp1)[dimp1 - 1];
        if (phoneme >= 0) {
          nn = nearestNeighbor(vector + i*dimp1, km->meanv, N, dim);
          for (j=0; j<dim; j++) {
            (centroid + nn*dim)[j] += (vector + i*dimp1)[j];
          }
          count[nn] += 1.0;
        }
      }
      /* Update centroids and compute error */
      error = 0.0;
      for (i=0; i<N; i++) {
        if (count[i] > 0) {
          for (j=0; j<dim; j++) {
            newval = (centroid + i*dim)[j] / count[i];
            error += square(newval - (km->meanv + i*dim)[j]);
            (km->meanv + i*dim)[j] = newval;
          }
        }
      }
      printf("  Iteration %d Error = %f\n", iter, error);
      iter++;
    } while (error > ERR_THRES);
    N *= 2;
  } while (N < 2*K);

  free(count);
  free(centroid);
}



main(int argc, char *argv[]) {
  float *data;
  float *vector;
  int vectorN, dimp1;
  km_t km;

  if (argc < 5) {
    fprintf(stderr, "Usage: %s <out>.km <train>.dat <K> <phoneN>\n", argv[0]);
    exit(-1);
  }
```

```
  readDat(argv[2], &data, &vectorN, &dimp1);
  transpose(&vector, data, vectorN, dimp1);
  performClustering(&km, atoi(argv[3]), atoi(argv[4]),
                    vector, vectorN, dimp1, kmgamma);
  writeKm(argv[1], &km);
  free(data);
  free(vector);
}
```

B.13 createkm2.c

```
/*************************************************************************\
*                                                                         *
*                          Phonetic Classifier                            *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                    and                                  *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                  at the                                 *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                                May 15, 97                               *
\*************************************************************************/


/* Performs K-means clustering */
/* Binary split with phoneme weights */

#include "common.h"

#define ERR_THRES    0.001    /* Error threshold for centroids */
#define EPSILON      0.04     /* Epsilon used in splitting */


/* gamma parameter for vector normalization */
float kmgamma[] = {
  25.172358, 19.218266, 21.742237, 24.898365, 18.966608, 25.621672,
  33.172913, 37.476593, 1.096734, 0.230770, 0.337857, 0.225993,
  0.357310, 0.067926, 23.546637, 17.280016, 11.299080, 11.924601,
  17.581976, 24.429148, 32.420090, 36.827389, 1.184915, 0.251185,
  0.257460, 0.269493, 0.185793, 0.071770, 27.508614, 13.651873,
  10.648049, 15.503012, 11.959195, 27.137836, 34.634777, 38.795593,
  0.951037, 0.217823, 0.321228, 0.218384, 0.186709, 0.071181
};
```

```
void performClustering(km_t *km, int K, int phoneN, float *vector,
                       int vectorN, int dimp1, float gamma[]) {
  /* Effects: Performs clustering on vector, which contains vectorN
     vectors of (dimp1 - 1) dimensions.  K clusters are to be found
     and there are phoneN phonemes.  If gamma is not null, it is used
     for vector normalization.  vector is normalized before clustering
     begins.  Clustering results are stored in km */
  /* Modifies: km, vector */
  int i, j;
  int dim;          /* No. of dimensions */
  int N;            /* Current no. of centroids */
  int nn;           /* Nearest neighbor */
  float *centroid;  /* Current centroids */
  float *count;     /* No. of vectors corresponding to each centroid */
  int phoneme;      /* Current phoneme */
  int *tokencount;  /* No. of vectors representing each phoneme */
  int iter;         /* No. of iterations */
  float newval;     /* New value */
  float error;

  /* Initializations */
  dim = dimp1 - 1;
  km->K = K;
  km->dim = dim;
  km->mu = (float *)malloc(dim * sizeof(float));
  km->gamma = (float *)malloc(dim * sizeof(float));
  km->meanv = (float *)malloc(K * dim * sizeof(float));
  centroid = (float *)malloc(K * dim * sizeof(float));
  count = (float *)malloc(K * sizeof(float));
  tokencount = (int *)malloc(phoneN * sizeof(int));
  findNorm(km, vector, vectorN, dimp1, gamma);

  /* Count phonemes */
  for (i=0; i<phoneN; i++) {
    tokencount[i] = 0;
  }
  for (i=0; i<vectorN; i++) {
    phoneme = (int)(vector + i*dimp1)[dimp1 - 1];
    if (phoneme >= 0) {
      tokencount[phoneme]++;
    }
  }

  N = 1;
  do {
    if (N == 1) {
      for (i=0; i<dim; i++) {
        (km->meanv)[i] = 0.0;
      }
    }
    else {
      /* Binary split each centroid */
      for (i=N/2-1; i>=0; i--) {
        for (j=0; j<dim; j++) {
```

```
      if (N/2 + i >= K) {
        (km->meanv + (K - N/2 + i)*dim)[j] =
          (1 - EPSILON) * (km->meanv + i*dim)[j];
      }
      else {
        (km->meanv + 2*i*dim)[j] =
          (1 - EPSILON) * (km->meanv + i*dim)[j];
      }
    }
  }
  for (i=0; i<N/2; i++) {
    if (N/2 + i < K) {
      for (j=0; j<dim; j++) {
        (km->meanv + (2*i + 1)*dim)[j] =
          (1 + EPSILON) / (1 - EPSILON) * (km->meanv + 2*i*dim)[j];
      }
    }
  }
}

if (N > K) N = K;
printf("Set Size = %d\n", N);
/* Perform iterations */
iter = 1;
do {
  clearVectors(centroid, N, dim);
  for (i=0; i<N; i++) count[i] = 0.0;
  /* Find nearest centroids and compute new centroids */
  for (i=0; i<vectorN; i++) {
    phoneme = (int)(vector + i*dimp1)[dimp1 - 1];
    if ((phoneme >= 0) && (tokencount[phoneme] > 0)) {
      nn = nearestNeighbor(vector + i*dimp1, km->meanv, N, dim);
      /* Weighted centroid */
      for (j=0; j<dim; j++) {
        (centroid + nn*dim)[j] +=
          (vector + i*dimp1)[j] / tokencount[phoneme];
      }
      /* Weighted count */
      count[nn] += 1.0 / tokencount[phoneme];
    }
  }
  /* Update centroids and compute error */
  error = 0.0;
  for (i=0; i<N; i++) {
    if (count[i] > 0) {
      for (j=0; j<dim; j++) {
        newval = (centroid + i*dim)[j] / count[i];
        error += square(newval - (km->meanv + i*dim)[j]);
        (km->meanv + i*dim)[j] = newval;
      }
    }
  }
  printf("  Iteration %d Error = %f\n", iter, error);
  iter++;
```

```
    } while (error > ERR_THRES);
    N *= 2;
  } while (N < 2*K);

  free(count);
  free(centroid);
  free(tokencount);
}



main(int argc, char *argv[]) {
  float *data;
  float *vector;
  int vectorN, dimp1;
  km_t km;

  if (argc < 5) {
    fprintf(stderr, "Usage: %s <out>.km <train>.dat <K> <phoneN>\n", argv[0]);
    exit(-1);
  }
  readDat(argv[2], &data, &vectorN, &dimp1);
  transpose(&vector, data, vectorN, dimp1);
  performClustering(&km, atoi(argv[3]), atoi(argv[4]),
                    vector, vectorN, dimp1, kmgamma);
  writeKm(argv[1], &km);
  free(data);
  free(vector);
}
```

B.14 createkm3.c

```
/*************************************************************************\
*                                                                         *
*                          Phonetic Classifier                            *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                  and                                    *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                  at the                                 *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                              May 15, 97                                 *
\*************************************************************************/
```

```
/* Performs K-means clustering */
/* Phoneme split without phoneme weights */

#include "common.h"

#define ERR_THRES    0.001    /* Error threshold for centroids */
#define EPSILON      0.04     /* Epsilon used in splitting */


/* gamma parameter for vector normalization */
float kmgamma[] = {
  25.172358, 19.218266, 21.742237, 24.898365, 18.966608, 25.621672,
  33.172913, 37.476593, 1.096734, 0.230770, 0.337857, 0.225993,
  0.357310, 0.067926, 23.546637, 17.280016, 11.299080, 11.924601,
  17.581976, 24.429148, 32.420090, 36.827389, 1.184915, 0.251185,
  0.257460, 0.269493, 0.185793, 0.071770, 27.508614, 13.651873,
  10.648049, 15.503012, 11.959195, 27.137836, 34.634777, 38.795593,
  0.951037, 0.217823, 0.321228, 0.218384, 0.186709, 0.071181
};


void performClustering(km_t *km, int K, int phoneN, float *vector,
                       int vectorN, int dimp1, float gamma[]) {
  /* Effects: Performs clustering on vector, which contains vectorN
     vectors of (dimp1 - 1) dimensions.  K clusters are to be found
     and there are phoneN phonemes.  If gamma is not null, it is used
     for vector normalization.  vector is normalized before clustering
     begins.  Clustering results are stored in km */
  /* Modifies: km, vector */
  int i, j, k;
  int dim;            /* No. of dimensions */
  int N;              /* Current no. of centroids */
  int nn;             /* Nearest neighbor */
  float *centroid;    /* Current centroids */
  float *count;       /* No. of vectors corresponding to each centroid */
  int phoneme;        /* Current phoneme */
  int *nearmean;      /* Record of nearest centroids */
  int *phonecount;    /* Phoneme counts in each cluster */
  float *pcentroid;   /* Phoneme centroids in cluster */
  int maxphone;       /* Maximum phoneme count in cluster */
  int secondphone;    /* Next to maximum phoneme count in cluster */
  float score;
  float maxscore;     /* Maximum score in cluster */
  float secondscore;  /* Next to maximum score in cluster */
  int iter;           /* No. of iterations */
  float newval;       /* New value */
  float error;

  /* Initializations */
  dim = dimp1 - 1;
  km->K = K;
  km->dim = dim;
  km->mu = (float *)malloc(dim * sizeof(float));
```

```
km->gamma = (float *)malloc(dim * sizeof(float));
km->meanv = (float *)malloc(K * dim * sizeof(float));
centroid = (float *)malloc(K * dim * sizeof(float));
count = (float *)malloc(K * sizeof(float));
nearmean = (int *)malloc(vectorN * sizeof(int));
phonecount = (int *)malloc(phoneN * sizeof(int));
pcentroid = (float *)malloc(phoneN * dim * sizeof(float));
findNorm(km, vector, vectorN, dimp1, gamma);

N = 1;
do {
  if (N == 1) {
    for (i=0; i<dim; i++) {
      (km->meanv)[i] = 0.0;
    }
  }
  else {
    /* Phoneme split each centroid */
    for (i=0; i<vectorN; i++) {
      nearmean[i] = nearestNeighbor(vector + i*dimp1, km->meanv, N/2, dim);
    }
    for (i=0; i<N/2; i++) {
      if (N/2 + i < K) {
        /* Count phonemes in cluster and compute phoneme centroids */
        for (j=0; j<phoneN; j++) phonecount[j] = 0;
        clearVectors(pcentroid, phoneN, dim);
        for (j=0; j<vectorN; j++) {
          if (nearmean[j] == i) {
            phoneme = (int)(vector + j*dimp1)[dimp1 - 1];
            if (phoneme >= 0) {
              for (k=0; k<dim; k++) {
                (pcentroid + phoneme*dim)[k] += (vector + j*dimp1)[k];
              }
              phonecount[phoneme]++;
            }
          }
        }
        /* Obtain maximum score */
        maxphone = -1;
        maxscore = 0.0;
        for (j=0; j<phoneN; j++) {
          score = (float)phonecount[j];
          if (score > maxscore) {
            maxphone = j;
            maxscore = score;
          }
        }
        /* Obtain next to maximum score */
        secondphone = -1;
        secondscore = 0.0;
        for (j=0; j<phoneN; j++) {
          score = (float)phonecount[j];
          if ((score > secondscore) && (j != maxphone)) {
            secondphone = j;
```

```
          secondscore = score;
        }
      }
      /* Now perform splitting */
      if (secondphone == -1) {
        /* Only 0 or 1 phoneme in cluster, so perform binary split */
        for (j=0; j<dim; j++) {
          (km->meanv + (N/2 + i)*dim)[j] =
            (1 + EPSILON) * (km->meanv + i*dim)[j];
        }
        for (j=0; j<dim; j++) {
          (km->meanv + i*dim)[j] *= (1 - EPSILON);
        }
      }
      else {
        /* Split into phoneme centroids */
        for (j=0; j<dim; j++) {
          newval = (pcentroid + maxphone*dim)[j] /
            phonecount[maxphone];
          (km->meanv + i*dim)[j] = newval;
          newval = (pcentroid + secondphone*dim)[j] /
            phonecount[secondphone];
          (km->meanv + (N/2 + i)*dim)[j] = newval;
        }
      }
    }
  }
}

if (N > K) N = K;
printf("Set Size = %d\n", N);
/* Perform iterations */
iter = 1;
do {
  clearVectors(centroid, N, dim);
  for (i=0; i<N; i++) count[i] = 0.0;
  /* Find nearest centroids and compute new centroids */
  for (i=0; i<vectorN; i++) {
    phoneme = (int)(vector + i*dimp1)[dimp1 - 1];
    if (phoneme >= 0) {
      nn = nearestNeighbor(vector + i*dimp1, km->meanv, N, dim);
      for (j=0; j<dim; j++) {
        (centroid + nn*dim)[j] += (vector + i*dimp1)[j];
      }
      count[nn] += 1.0;
    }
  }
  /* Update centroids and compute error */
  error = 0.0;
  for (i=0; i<N; i++) {
    if (count[i] > 0) {
      for (j=0; j<dim; j++) {
        newval = (centroid + i*dim)[j] / count[i];
        error += square(newval - (km->meanv + i*dim)[j]);
```

```
            (km->meanv + i*dim)[j] = newval;
          }
        }
      }
      printf("  Iteration %d Error = %f\n", iter, error);
      iter++;
    } while (error > ERR_THRES);
    N *= 2;
  } while (N < 2*K);

  free(count);
  free(centroid);
  free(nearmean);
  free(phonecount);
  free(pcentroid);
}



main(int argc, char *argv[]) {
  float *data;
  float *vector;
  int vectorN, dimp1;
  km_t km;

  if (argc < 5) {
    fprintf(stderr, "Usage: %s <out>.km <train>.dat <K> <phoneN>\n", argv[0]);
    exit(-1);
  }
  readDat(argv[2], &data, &vectorN, &dimp1);
  transpose(&vector, data, vectorN, dimp1);
  performClustering(&km, atoi(argv[3]), atoi(argv[4]),
                    vector, vectorN, dimp1, kmgamma);
  writeKm(argv[1], &km);
  free(data);
  free(vector);
}
```

B.15 createkm4.c

```
/*************************************************************************\
*                                                                         *
*                          Phonetic Classifier                            *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                  and                                    *
```

```
*  Master of Engineering in Electrical Engineering and Computer Science  *
*                                at the                                  *
*                  Massachusetts Institute of Technology                 *
*                                                                        *
*                   Chiangkai Er <changkai@alum.mit.edu>                 *
*                              May 15, 97                                *
\************************************************************************/


/* Performs K-means clustering */
/* Phoneme split with phoneme weights */

#include "common.h"

#define ERR_THRES    0.001    /* Error threshold for centroids */
#define EPSILON      0.04     /* Epsilon used in splitting */


/* gamma parameter for vector normalization */
float kmgamma[] = {
  25.172358, 19.218266, 21.742237, 24.898365, 18.966608, 25.621672,
  33.172913, 37.476593, 1.096734, 0.230770, 0.337857, 0.225993,
  0.357310, 0.067926, 23.546637, 17.280016, 11.299080, 11.924601,
  17.581976, 24.429148, 32.420090, 36.827389, 1.184915, 0.251185,
  0.257460, 0.269493, 0.185793, 0.071770, 27.508614, 13.651873,
  10.648049, 15.503012, 11.959195, 27.137836, 34.634777, 38.795593,
  0.951037, 0.217823, 0.321228, 0.218384, 0.186709, 0.071181
};


void performClustering(km_t *km, int K, int phoneN, float *vector,
                       int vectorN, int dimp1, float gamma[]) {
  /* Effects: Performs clustering on vector, which contains vectorN
     vectors of (dimp1 - 1) dimensions.  K clusters are to be found
     and there are phoneN phonemes.  If gamma is not null, it is used
     for vector normalization.  vector is normalized before clustering
     begins.  Clustering results are stored in km */
  /* Modifies: km, vector */
  int i, j, k;
  int dim;            /* No. of dimensions */
  int N;              /* Current no. of centroids */
  int nn;             /* Nearest neighbor */
  float *centroid;    /* Current centroids */
  float *count;       /* No. of vectors corresponding to each centroid */
  int phoneme;        /* Current phoneme */
  int *tokencount;    /* No. of vectors representing each phoneme */
  int *nearmean;      /* Record of nearest centroids */
  int *phonecount;    /* Phoneme counts in each cluster */
  float *pcentroid;   /* Phoneme centroids in cluster */
  int maxphone;       /* Maximum phoneme count in cluster */
  int secondphone;    /* Next to maximum phoneme count in cluster */
  float score;
  float maxscore;     /* Maximum score in cluster */
  float secondscore;  /* Next to maximum score in cluster */
```

```
int iter;           /* No. of iterations */
float newval;       /* New value */
float error;

/* Initializations */
dim = dimp1 - 1;
km->K = K;
km->dim = dim;
km->mu = (float *)malloc(dim * sizeof(float));
km->gamma = (float *)malloc(dim * sizeof(float));
km->meanv = (float *)malloc(K * dim * sizeof(float));
centroid = (float *)malloc(K * dim * sizeof(float));
count = (float *)malloc(K * sizeof(float));
tokencount = (int *)malloc(phoneN * sizeof(int));
nearmean = (int *)malloc(vectorN * sizeof(int));
phonecount = (int *)malloc(phoneN * sizeof(int));
pcentroid = (float *)malloc(phoneN * dim * sizeof(float));
findNorm(km, vector, vectorN, dimp1, gamma);

/* Count phonemes */
for (i=0; i<phoneN; i++) {
  tokencount[i] = 0;
}
for (i=0; i<vectorN; i++) {
  phoneme = (int)(vector + i*dimp1)[dimp1 - 1];
  if (phoneme >= 0) {
    tokencount[phoneme]++;
  }
}

N = 1;
do {
  if (N == 1) {
    for (i=0; i<dim; i++) {
      (km->meanv)[i] = 0.0;
    }
  }
  else {
    /* Phoneme split each centroid */
    for (i=0; i<vectorN; i++) {
      nearmean[i] = nearestNeighbor(vector + i*dimp1, km->meanv, N/2, dim);
    }
    for (i=0; i<N/2; i++) {
      if (N/2 + i < K) {
        /* Count phonemes in cluster and compute phoneme centroids */
        for (j=0; j<phoneN; j++) phonecount[j] = 0;
        clearVectors(pcentroid, phoneN, dim);
        for (j=0; j<vectorN; j++) {
          if (nearmean[j] == i) {
            phoneme = (int)(vector + j*dimp1)[dimp1 - 1];
            if (phoneme >= 0) {
              for (k=0; k<dim; k++) {
                (pcentroid + phoneme*dim)[k] += (vector + j*dimp1)[k];
              }
```

```
            phonecount[phoneme]++;
          }
        }
      }
      /* Obtain maximum score */
      maxphone = -1;
      maxscore = 0.0;
      for (j=0; j<phoneN; j++) {
        if (tokencount[j] > 0) {
          score = (float)phonecount[j] / tokencount[j];
          if (score > maxscore) {
            maxphone = j;
            maxscore = score;
          }
        }
      }
      /* Obtain next to maximum score */
      secondphone = -1;
      secondscore = 0.0;
      for (j=0; j<phoneN; j++) {
        if (tokencount[j] > 0) {
          score = (float)phonecount[j] / tokencount[j];
          if ((score > secondscore) && (j != maxphone)) {
            secondphone = j;
            secondscore = score;
          }
        }
      }
      /* Now perform splitting */
      if (secondphone == -1) {
        /* Only 0 or 1 phoneme in cluster, so perform binary split */
        for (j=0; j<dim; j++) {
          (km->meanv + (N/2 + i)*dim)[j] =
            (1 + EPSILON) * (km->meanv + i*dim)[j];
        }
        for (j=0; j<dim; j++) {
          (km->meanv + i*dim)[j] *= (1 - EPSILON);
        }
      }
      else {
        /* Split into phoneme centroids */
        for (j=0; j<dim; j++) {
          newval = (pcentroid + maxphone*dim)[j] /
            phonecount[maxphone];
          (km->meanv + i*dim)[j] = newval;
          newval = (pcentroid + secondphone*dim)[j] /
            phonecount[secondphone];
          (km->meanv + (N/2 + i)*dim)[j] = newval;
        }
      }
    }
  }
}
```

```
    if (N > K) N = K;
    printf("Set Size = %d\n", N);
    /* Perform iterations */
    iter = 1;
    do {
      clearVectors(centroid, N, dim);
      for (i=0; i<N; i++) count[i] = 0.0;
      /* Find nearest centroids and compute new centroids */
      for (i=0; i<vectorN; i++) {
        phoneme = (int)(vector + i*dimp1)[dimp1 - 1];
        if ((phoneme >= 0) && (tokencount[phoneme] > 0)) {
          nn = nearestNeighbor(vector + i*dimp1, km->meanv, N, dim);
          /* Weighted centroid */
          for (j=0; j<dim; j++) {
            (centroid + nn*dim)[j] +=
              (vector + i*dimp1)[j] / tokencount[phoneme];
          }
          /* Weighted count */
          count[nn] += 1.0 / tokencount[phoneme];
        }
      }
      /* Update centroids and compute error */
      error = 0.0;
      for (i=0; i<N; i++) {
        if (count[i] > 0) {
          for (j=0; j<dim; j++) {
            newval = (centroid + i*dim)[j] / count[i];
            error += square(newval - (km->meanv + i*dim)[j]);
            (km->meanv + i*dim)[j] = newval;
          }
        }
      }
      printf("  Iteration %d Error = %f\n", iter, error);
      iter++;
    } while (error > ERR_THRES);
    N *= 2;
  } while (N < 2*K);

  free(count);
  free(centroid);
  free(tokencount);
  free(nearmean);
  free(phonecount);
  free(pcentroid);
}


main(int argc, char *argv[]) {
  float *data;
  float *vector;
  int vectorN, dimp1;
  km_t km;

  if (argc < 5) {
```

```
    fprintf(stderr, "Usage: %s <out>.km <train>.dat <K> <phoneN>\n", argv[0]);
    exit(-1);
  }
  readDat(argv[2], &data, &vectorN, &dimp1);
  transpose(&vector, data, vectorN, dimp1);
  performClustering(&km, atoi(argv[3]), atoi(argv[4]),
                    vector, vectorN, dimp1, kmgamma);
  writeKm(argv[1], &km);
  free(data);
  free(vector);
}
```

B.16 createkm5.c

```
/*************************************************************************\
*                                                                         *
*                           Phonetic Classifier                           *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                    and                                  *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                    at the                               *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                                May 15, 97                               *
\*************************************************************************/


/* Performs K-means clustering */
/* Independent clustering with equal phoneme emphasis */

#include "common.h"

#define ERR_THRES    0.001    /* Error threshold for centroids */
#define EPSILON      0.04     /* Epsilon used in splitting */


/* gamma parameter for vector normalization */
float kmgamma[] = {
  25.172358, 19.218266, 21.742237, 24.898365, 18.966608, 25.621672,
  33.172913, 37.476593, 1.096734, 0.230770, 0.337857, 0.225993,
  0.357310, 0.067926, 23.546637, 17.280016, 11.299080, 11.924601,
  17.581976, 24.429148, 32.420090, 36.827389, 1.184915, 0.251185,
  0.257460, 0.269493, 0.185793, 0.071770, 27.508614, 13.651873,
  10.648049, 15.503012, 11.959195, 27.137836, 34.634777, 38.795593,
  0.951037, 0.217823, 0.321228, 0.218384, 0.186709, 0.071181
```

```
};


void performClustering(km_t *km, int K, int phoneN, float *vector,
                       int vectorN, int dimp1, float gamma[]) {
  /* Effects: Performs clustering on vector, which contains vectorN
     vectors of (dimp1 - 1) dimensions.  K clusters are to be found
     and there are phoneN phonemes.  If gamma is not null, it is used
     for vector normalization.  vector is normalized before clustering
     begins.  Clustering results are stored in km */
  /* Modifies: km, vector */
  int i, j;
  int dim;          /* No. of dimensions */
  int N;            /* Current no. of centroids */
  int nn;           /* Nearest neighbor */
  float *centroid;  /* Current centroids */
  int *count;       /* No. of vectors corresponding to each centroid */
  int phoneme;      /* Current phoneme */
  int meanbase;     /* Current base index for meanv */
  int phoneK;       /* No. of clusters for current phoneme */
  int iter;         /* No. of iterations */
  float newval;     /* New value */
  float error;

  /* Initializations */
  dim = dimp1 - 1;
  km->K = K;
  km->dim = dim;
  km->mu = (float *)malloc(dim * sizeof(float));
  km->gamma = (float *)malloc(dim * sizeof(float));
  km->meanv = (float *)malloc(K * dim * sizeof(float));
  centroid = (float *)malloc(K * dim * sizeof(float));
  count = (int *)malloc(K * sizeof(int));
  findNorm(km, vector, vectorN, dimp1, gamma);

  meanbase = 0;
  /* For each phoneme */
  for (phoneme=0; phoneme<phoneN; phoneme++) {
    printf("Clustering for phoneme %d....\n", phoneme);
    phoneK = round((float)(phoneme + 1)/phoneN * K) - meanbase;
    printf("K = %d\n", phoneK);
    /* Perform clustering for current phoneme */
    N = 1;
    do {
      for (i=0; i<N/2; i++) {
        if (N/2 + i < phoneK) {
          /* Binary split each centroid */
          for (j=0; j<dim; j++) {
            (km->meanv + (meanbase + N/2 + i)*dim)[j] =
              (1 + EPSILON) * (km->meanv + (meanbase + i)*dim)[j];
          }
          for (j=0; j<dim; j++) {
            (km->meanv + (meanbase + i)*dim)[j] *= (1 - EPSILON);
          }
```

```
        }
      }

      if (N > phoneK) N = phoneK;
      printf("Set Size = %d\n", N);
      /* Perform iterations */
      iter = 1;
      do {
        clearVectors(centroid, N, dim);
        for (i=0; i<N; i++) count[i] = 0;
        /* Find nearest centroids and compute new centroids */
        for (i=0; i<vectorN; i++) {
          if (phoneme == (int)(vector + i*dimp1)[dimp1 - 1]) {
            nn = nearestNeighbor(vector + i*dimp1,
                                 km->meanv + meanbase*dim, N, dim);
            for (j=0; j<dim; j++) {
              (centroid + nn*dim)[j] += (vector + i*dimp1)[j];
            }
            count[nn]++;
          }
        }
        error = 0.0;
        /* Update centroids and compute error */
        for (i=0; i<N; i++) {
          if (count[i] > 0) {
            for (j=0; j<dim; j++) {
              newval = (centroid + i*dim)[j] / count[i];
              error += square(newval - (km->meanv + (meanbase + i)*dim)[j]);
              (km->meanv + (meanbase + i)*dim)[j] = newval;
            }
          }
        }
        printf("  Iteration %d Error = %f\n", iter, error);
        iter++;
      } while (error > ERR_THRES);
      N *= 2;
    } while (N < 2*phoneK);

    meanbase += phoneK;
  }

  free(count);
  free(centroid);
}



main(int argc, char *argv[]) {
  float *data;
  float *vector;
  int vectorN, dimp1;
  km_t km;
```

```
  if (argc < 5) {
    fprintf(stderr, "Usage: %s <out>.km <train>.dat <K> <phoneN>\n", argv[0]);
    exit(-1);
  }
  readDat(argv[2], &data, &vectorN, &dimp1);
  transpose(&vector, data, vectorN, dimp1);
  performClustering(&km, atoi(argv[3]), atoi(argv[4]),
                    vector, vectorN, dimp1, kmgamma);
  writeKm(argv[1], &km);
  free(data);
  free(vector);
}
```

B.17 createkm6.c

```
/*************************************************************************\
*                                                                         *
*                          Phonetic Classifier                            *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                    and                                  *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                  at the                                 *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                  Chiangkai Er <changkai@alum.mit.edu>                   *
*                               May 15, 97                                *
\*************************************************************************/


/* Performs K-means clustering */
/* Independent clustering with proportional phoneme emphasis */

#include "common.h"

#define ERR_THRES    0.001    /* Error threshold for centroids */
#define EPSILON      0.04     /* Epsilon used in splitting */


/* gamma parameter for vector normalization */
float kmgamma[] = {
  25.172358, 19.218266, 21.742237, 24.898365, 18.966608, 25.621672,
  33.172913, 37.476593, 1.096734, 0.230770, 0.337857, 0.225993,
  0.357310, 0.067926, 23.546637, 17.280016, 11.299080, 11.924601,
  17.581976, 24.429148, 32.420090, 36.827389, 1.184915, 0.251185,
  0.257460, 0.269493, 0.185793, 0.071770, 27.508614, 13.651873,
  10.648049, 15.503012, 11.959195, 27.137836, 34.634777, 38.795593,
```

```
  0.951037, 0.217823, 0.321228, 0.218384, 0.186709, 0.071181
};


void performClustering(km_t *km, int K, int phoneN, float *vector,
                       int vectorN, int dimp1, float gamma[]) {
  /* Effects: Performs clustering on vector, which contains vectorN
     vectors of (dimp1 - 1) dimensions.  K clusters are to be found
     and there are phoneN phonemes.  If gamma is not null, it is used
     for vector normalization.  vector is normalized before clustering
     begins.  Clustering results are stored in km */
  /* Modifies: km, vector */
  int i, j;
  int dim;            /* No. of dimensions */
  int N;              /* Current no. of centroids */
  int nn;             /* Nearest neighbor */
  float *centroid;    /* Current centroids */
  int *count;         /* No. of vectors corresponding to each centroid */
  int phoneme;        /* Current phoneme */
  int meanbase;       /* Current base index for meanv */
  int phoneK;         /* No. of clusters for current phoneme */
  float *proportion;  /* Phoneme proportion */
  int total;          /* Total no. of phonemes */
  int iter;           /* No. of iterations */
  float newval;       /* New value */
  float error;

  /* Initializations */
  dim = dimp1 - 1;
  km->K = K;
  km->dim = dim;
  km->mu = (float *)malloc(dim * sizeof(float));
  km->gamma = (float *)malloc(dim * sizeof(float));
  km->meanv = (float *)malloc(K * dim * sizeof(float));
  centroid = (float *)malloc(K * dim * sizeof(float));
  count = (int *)malloc(K * sizeof(int));
  proportion = (float *)malloc(phoneN * sizeof(float));
  findNorm(km, vector, vectorN, dimp1, gamma);

  /* Compute phoneme proportions */
  for (i=0; i<phoneN; i++) {
    proportion[i] = 0.0;
  }
  total = 0;
  for (i=0; i<vectorN; i++) {
    phoneme = (int)(vector + i*dimp1)[dimp1 - 1];
    if (phoneme >= 0) {
      proportion[phoneme] += 1.0;
      total++;
    }
  }
  for (i=0; i<phoneN; i++) {
    proportion[i] /= total;
  }
```

```
meanbase = 0;
/* For each phoneme */
for (phoneme=0; phoneme<phoneN; phoneme++) {
  printf("Clustering for phoneme %d....\n", phoneme);
  if (phoneme < phoneN - 1) {
    phoneK = round(proportion[phoneme] * K);
  }
  else {
    phoneK = K - meanbase;
  }
  printf("K = %d\n", phoneK);
  /* Perform clustering for current phoneme */
  N = 1;
  do {
    for (i=0; i<N/2; i++) {
      if (N/2 + i < phoneK) {
        /* Binary split each centroid */
        for (j=0; j<dim; j++) {
          (km->meanv + (meanbase + N/2 + i)*dim)[j] =
            (1 + EPSILON) * (km->meanv + (meanbase + i)*dim)[j];
        }
        for (j=0; j<dim; j++) {
          (km->meanv + (meanbase + i)*dim)[j] *= (1 - EPSILON);
        }
      }
    }

    if (N > phoneK) N = phoneK;
    printf("Set Size = %d\n", N);
    /* Perform iterations */
    iter = 1;
    do {
      clearVectors(centroid, N, dim);
      for (i=0; i<N; i++) count[i] = 0;
      /* Find nearest centroids and compute new centroids */
      for (i=0; i<vectorN; i++) {
        if (phoneme == (int)(vector + i*dimp1)[dimp1 - 1]) {
          nn = nearestNeighbor(vector + i*dimp1,
                               km->meanv + meanbase*dim, N, dim);
          for (j=0; j<dim; j++) {
            (centroid + nn*dim)[j] += (vector + i*dimp1)[j];
          }
          count[nn]++;
        }
      }
      error = 0.0;
      /* Update centroids and compute error */
      for (i=0; i<N; i++) {
        if (count[i] > 0) {
          for (j=0; j<dim; j++) {
            newval = (centroid + i*dim)[j] / count[i];
            error += square(newval - (km->meanv + (meanbase + i)*dim)[j]);
            (km->meanv + (meanbase + i)*dim)[j] = newval;
```

```
          }
        }
      }
      printf("  Iteration %d Error = %f\n", iter, error);
      iter++;
    } while (error > ERR_THRES);
    N *= 2;
  } while (N < 2*phoneK);

  meanbase += phoneK;
 }

 free(count);
 free(centroid);
 free(proportion);
}



main(int argc, char *argv[]) {
 float *data;
 float *vector;
 int vectorN, dimp1;
 km_t km;

 if (argc < 5) {
  fprintf(stderr, "Usage: %s <out>.km <train>.dat <K> <phoneN>\n", argv[0]);
  exit(-1);
 }
 readDat(argv[2], &data, &vectorN, &dimp1);
 transpose(&vector, data, vectorN, dimp1);
 performClustering(&km, atoi(argv[3]), atoi(argv[4]),
                   vector, vectorN, dimp1, kmgamma);
 writeKm(argv[1], &km);
 free(data);
 free(vector);
}
```

B.18 createkm7.c

```
/*************************************************************************\
*                                                                         *
*                          Phonetic Classifier                            *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                  and                                    *
```

```
*  Master of Engineering in Electrical Engineering and Computer Science  *
*                                  at the                                 *
*                 Massachusetts Institute of Technology                   *
*                                                                         *
*                  Chiangkai Er <changkai@alum.mit.edu>                   *
*                               May 15, 97                                *
\*************************************************************************/


/* Performs K-means clustering */
/* Clustering after independent clustering with equal phoneme emphasis */

#include "common.h"

#define ERR_THRES    0.001    /* Error threshold for centroids */
#define EPSILON      0.04     /* Epsilon used in splitting */


/* gamma parameter for vector normalization */
float kmgamma[] = {
  25.172358, 19.218266, 21.742237, 24.898365, 18.966608, 25.621672,
  33.172913, 37.476593, 1.096734, 0.230770, 0.337857, 0.225993,
  0.357310, 0.067926, 23.546637, 17.280016, 11.299080, 11.924601,
  17.581976, 24.429148, 32.420090, 36.827389, 1.184915, 0.251185,
  0.257460, 0.269493, 0.185793, 0.071770, 27.508614, 13.651873,
  10.648049, 15.503012, 11.959195, 27.137836, 34.634777, 38.795593,
  0.951037, 0.217823, 0.321228, 0.218384, 0.186709, 0.071181
};


void performClustering(km_t *km, int K, int phoneN, float *vector,
                       int vectorN, int dimp1, float gamma[]) {
  /* Effects: Performs clustering on vector, which contains vectorN
     vectors of (dimp1 - 1) dimensions.  K clusters are to be found
     and there are phoneN phonemes.  If gamma is not null, it is used
     for vector normalization.  vector is normalized before clustering
     begins.  Clustering results are stored in km */
  /* Modifies: km, vector */
  int i, j;
  int dim;             /* No. of dimensions */
  int N;               /* Current no. of centroids */
  int nn;              /* Nearest neighbor */
  float *centroid;     /* Current centroids */
  int *count;          /* No. of vectors corresponding to each centroid */
  float *countf;       /* float version of count */
  int phoneme;         /* Current phoneme */
  int meanbase;        /* Current base index for meanv */
  int phoneK;          /* No. of clusters for current phoneme */
  float *proportion;   /* Phoneme proportion */
  int total;           /* Total no. of phonemes */
  int iter;            /* No. of iterations */
  float newval;        /* New value */
  float error;
```

```
/* Initializations */
dim = dimp1 - 1;
km->K = K;
km->dim = dim;
km->mu = (float *)malloc(dim * sizeof(float));
km->gamma = (float *)malloc(dim * sizeof(float));
km->meanv = (float *)malloc(K * dim * sizeof(float));
centroid = (float *)malloc(K * dim * sizeof(float));
count = (int *)malloc(K * sizeof(int));
countf = (float *)malloc(K * sizeof(int));
proportion = (float *)malloc(phoneN * sizeof(float));
findNorm(km, vector, vectorN, dimp1, gamma);

/* Compute phoneme proportions */
for (i=0; i<phoneN; i++) {
  proportion[i] = 0.0;
}
total = 0;
for (i=0; i<vectorN; i++) {
  phoneme = (int)(vector + i*dimp1)[dimp1 - 1];
  if (phoneme >= 0) {
    proportion[phoneme] += 1.0;
    total++;
  }
}
for (i=0; i<phoneN; i++) {
  proportion[i] /= total;
}

meanbase = 0;
/* For each phoneme */
for (phoneme=0; phoneme<phoneN; phoneme++) {
  printf("Clustering for phoneme %d....\n", phoneme);
  phoneK = round((float)(phoneme + 1)/phoneN * K) - meanbase;
  printf("K = %d\n", phoneK);
  /* Perform clustering for current phoneme */
  N = 1;
  do {
    for (i=0; i<N/2; i++) {
      if (N/2 + i < phoneK) {
        /* Binary split each centroid */
        for (j=0; j<dim; j++) {
          (km->meanv + (meanbase + N/2 + i)*dim)[j] =
            (1 + EPSILON) * (km->meanv + (meanbase + i)*dim)[j];
        }
        for (j=0; j<dim; j++) {
          (km->meanv + (meanbase + i)*dim)[j] *= (1 - EPSILON);
        }
      }
    }

    if (N > phoneK) N = phoneK;
    printf("Set Size = %d\n", N);
    /* Perform iterations */
```

```
    iter = 1;
    do {
      clearVectors(centroid, N, dim);
      for (i=0; i<N; i++) count[i] = 0;
      /* Find nearest centroids and compute new centroids */
      for (i=0; i<vectorN; i++) {
        if (phoneme == (int)(vector + i*dimp1)[dimp1 - 1]) {
          nn = nearestNeighbor(vector + i*dimp1,
                               km->meanv + meanbase*dim, N, dim);
          for (j=0; j<dim; j++) {
            (centroid + nn*dim)[j] += (vector + i*dimp1)[j];
          }
          count[nn]++;
        }
      }
      error = 0.0;
      /* Update centroids and compute error */
      for (i=0; i<N; i++) {
        if (count[i] > 0) {
          for (j=0; j<dim; j++) {
            newval = (centroid + i*dim)[j] / count[i];
            error += square(newval - (km->meanv + (meanbase + i)*dim)[j]);
            (km->meanv + (meanbase + i)*dim)[j] = newval;
          }
        }
      }
      printf("  Iteration %d Error = %f\n", iter, error);
      iter++;
    } while (error > ERR_THRES);
    N *= 2;
  } while (N < 2*phoneK);
  meanbase += phoneK;
}

/* Clustering after independent clustering */
printf("Clustering for all phonemes....\n");
N = K;
printf("Set Size = %d\n", N);
iter = 1;
do {
  clearVectors(centroid, N, dim);
  for (i=0; i<N; i++) countf[i] = 0.0;
  /* Find nearest centroids and compute new centroids */
  for (i=0; i<vectorN; i++) {
    phoneme = (int)(vector + i*dimp1)[dimp1 - 1];
    if ((phoneme >= 0) && (proportion[phoneme] > 0)) {
      nn = nearestNeighbor(vector + i*dimp1, km->meanv, N, dim);
      /* Weighted centroid */
      for (j=0; j<dim; j++) {
        (centroid + nn*dim)[j] +=
          (vector + i*dimp1)[j] / proportion[phoneme];
      }
      /* Weighted count */
      countf[nn] += 1.0 / proportion[phoneme];
```

```
      }
    }
    error = 0.0;
    /* Update centroids and compute error */
    for (i=0; i<N; i++) {
      if (countf[i] > 0) {
        for (j=0; j<dim; j++) {
          newval = (centroid + i*dim)[j] / countf[i];
          error += square(newval - (km->meanv + i*dim)[j]);
          (km->meanv + i*dim)[j] = newval;
        }
      }
    }
    printf("  Iteration %d Error = %f\n", iter, error);
    iter++;
  } while (error > ERR_THRES);

  free(count);
  free(countf);
  free(centroid);
  free(proportion);
}



main(int argc, char *argv[]) {
  float *data;
  float *vector;
  int vectorN, dimp1;
  km_t km;

  if (argc < 5) {
    fprintf(stderr, "Usage: %s <out>.km <train>.dat <K> <phoneN>\n", argv[0]);
    exit(-1);
  }
  readDat(argv[2], &data, &vectorN, &dimp1);
  transpose(&vector, data, vectorN, dimp1);
  performClustering(&km, atoi(argv[3]), atoi(argv[4]),
                    vector, vectorN, dimp1, kmgamma);
  writeKm(argv[1], &km);
  free(data);
  free(vector);
}
```

B.19 createkm8.c

```
/*************************************************************************\
*                                                                         *
*                           Phonetic Classifier                           *
*                                                                         *
*                     Speech Recognition by Clustering                    *
```

```
*                     Wavelet and PLP Coefficients                      *
*                                                                       *
*               Thesis submitted in partial fulfillment of              *
*                   the requirements for the degrees of                 *
*        Bachelor of Science in Computer Science and Engineering        *
*                                  and                                  *
*  Master of Engineering in Electrical Engineering and Computer Science *
*                                 at the                                *
*                 Massachusetts Institute of Technology                 *
*                                                                       *
*                  Chiangkai Er <changkai@alum.mit.edu>                 *
*                              May 15, 97                               *
\***********************************************************************/


/* Performs K-means clustering */
/* Clustering after independent clustering with
   proportional phoneme emphasis */

#include "common.h"

#define ERR_THRES    0.001    /* Error threshold for centroids */
#define EPSILON      0.04     /* Epsilon used in splitting */


/* gamma parameter for vector normalization */
float kmgamma[] = {
  25.172358, 19.218266, 21.742237, 24.898365, 18.966608, 25.621672,
  33.172913, 37.476593, 1.096734, 0.230770, 0.337857, 0.225993,
  0.357310, 0.067926, 23.546637, 17.280016, 11.299080, 11.924601,
  17.581976, 24.429148, 32.420090, 36.827389, 1.184915, 0.251185,
  0.257460, 0.269493, 0.185793, 0.071770, 27.508614, 13.651873,
  10.648049, 15.503012, 11.959195, 27.137836, 34.634777, 38.795593,
  0.951037, 0.217823, 0.321228, 0.218384, 0.186709, 0.071181
};


void performClustering(km_t *km, int K, int phoneN, float *vector,
                       int vectorN, int dimp1, float gamma[]) {
  /* Effects: Performs clustering on vector, which contains vectorN
     vectors of (dimp1 - 1) dimensions.  K clusters are to be found
     and there are phoneN phonemes.  If gamma is not null, it is used
     for vector normalization.  vector is normalized before clustering
     begins.  Clustering results are stored in km */
  /* Modifies: km, vector */
  int i, j;
  int dim;            /* No. of dimensions */
  int N;              /* Current no. of centroids */
  int nn;             /* Nearest neighbor */
  float *centroid;    /* Current centroids */
  int *count;         /* No. of vectors corresponding to each centroid */
  float *countf;      /* float version of count */
  int phoneme;        /* Current phoneme */
  int meanbase;       /* Current base index for meanv */
```

```
int phoneK;         /* No. of clusters for current phoneme */
float *proportion;  /* Phoneme proportion */
int total;          /* Total no. of phonemes */
int iter;           /* No. of iterations */
float newval;       /* New value */
float error;

/* Initializations */
dim = dimp1 - 1;
km->K = K;
km->dim = dim;
km->mu = (float *)malloc(dim * sizeof(float));
km->gamma = (float *)malloc(dim * sizeof(float));
km->meanv = (float *)malloc(K * dim * sizeof(float));
centroid = (float *)malloc(K * dim * sizeof(float));
count = (int *)malloc(K * sizeof(int));
countf = (float *)malloc(K * sizeof(int));
proportion = (float *)malloc(phoneN * sizeof(float));
findNorm(km, vector, vectorN, dimp1, gamma);

/* Compute phoneme proportions */
for (i=0; i<phoneN; i++) {
  proportion[i] = 0.0;
}
total = 0;
for (i=0; i<vectorN; i++) {
  phoneme = (int)(vector + i*dimp1)[dimp1 - 1];
  if (phoneme >= 0) {
    proportion[phoneme] += 1.0;
    total++;
  }
}
for (i=0; i<phoneN; i++) {
  proportion[i] /= total;
}

meanbase = 0;
/* For each phoneme */
for (phoneme=0; phoneme<phoneN; phoneme++) {
  printf("Clustering for phoneme %d....\n", phoneme);
  if (phoneme < phoneN - 1) {
    phoneK = round(proportion[phoneme] * K);
  }
  else {
    phoneK = K - meanbase;
  }
  printf("K = %d\n", phoneK);
  /* Perform clustering for current phoneme */
  N = 1;
  do {
    for (i=0; i<N/2; i++) {
      if (N/2 + i < phoneK) {
        /* Binary split each centroid */
        for (j=0; j<dim; j++) {
```

```
          (km->meanv + (meanbase + N/2 + i)*dim)[j] =
            (1 + EPSILON) * (km->meanv + (meanbase + i)*dim)[j];
        }
        for (j=0; j<dim; j++) {
          (km->meanv + (meanbase + i)*dim)[j] *= (1 - EPSILON);
        }
      }
    }

    if (N > phoneK) N = phoneK;
    printf("Set Size = %d\n", N);
    /* Perform iterations */
    iter = 1;
    do {
      clearVectors(centroid, N, dim);
      for (i=0; i<N; i++) count[i] = 0;
      /* Find nearest centroids and compute new centroids */
      for (i=0; i<vectorN; i++) {
        if (phoneme == (int)(vector + i*dimp1)[dimp1 - 1]) {
          nn = nearestNeighbor(vector + i*dimp1,
                               km->meanv + meanbase*dim, N, dim);
          for (j=0; j<dim; j++) {
            (centroid + nn*dim)[j] += (vector + i*dimp1)[j];
          }
          count[nn]++;
        }
      }
      error = 0.0;
      /* Update centroids and compute error */
      for (i=0; i<N; i++) {
        if (count[i] > 0) {
          for (j=0; j<dim; j++) {
            newval = (centroid + i*dim)[j] / count[i];
            error += square(newval - (km->meanv + (meanbase + i)*dim)[j]);
            (km->meanv + (meanbase + i)*dim)[j] = newval;
          }
        }
      }
      printf("  Iteration %d Error = %f\n", iter, error);
      iter++;
    } while (error > ERR_THRES);
    N *= 2;
  } while (N < 2*phoneK);
  meanbase += phoneK;
}

/* Clustering after independent clustering */
printf("Clustering for all phonemes....\n");
N = K;
printf("Set Size = %d\n", N);
iter = 1;
do {
  clearVectors(centroid, N, dim);
  for (i=0; i<N; i++) countf[i] = 0.0;
```

```
    /* Find nearest centroids and compute new centroids */
    for (i=0; i<vectorN; i++) {
      phoneme = (int)(vector + i*dimp1)[dimp1 - 1];
      if ((phoneme >= 0) && (proportion[phoneme] > 0)) {
        nn = nearestNeighbor(vector + i*dimp1, km->meanv, N, dim);
        /* Weighted centroid */
        for (j=0; j<dim; j++) {
          (centroid + nn*dim)[j] +=
            (vector + i*dimp1)[j] / proportion[phoneme];
        }
        /* Weighted count */
        countf[nn] += 1.0 / proportion[phoneme];
      }
    }
    error = 0.0;
    /* Update centroids and compute error */
    for (i=0; i<N; i++) {
      if (countf[i] > 0) {
        for (j=0; j<dim; j++) {
          newval = (centroid + i*dim)[j] / countf[i];
          error += square(newval - (km->meanv + i*dim)[j]);
          (km->meanv + i*dim)[j] = newval;
        }
      }
    }
    printf("  Iteration %d Error = %f\n", iter, error);
    iter++;
  } while (error > ERR_THRES);

  free(count);
  free(countf);
  free(centroid);
  free(proportion);
}




main(int argc, char *argv[]) {
  float *data;
  float *vector;
  int vectorN, dimp1;
  km_t km;

  if (argc < 5) {
    fprintf(stderr, "Usage: %s <out>.km <train>.dat <K> <phoneN>\n", argv[0]);
    exit(-1);
  }
  readDat(argv[2], &data, &vectorN, &dimp1);
  transpose(&vector, data, vectorN, dimp1);
  performClustering(&km, atoi(argv[3]), atoi(argv[4]),
                    vector, vectorN, dimp1, kmgamma);
  writeKm(argv[1], &km);
  free(data);
```

```
  free(vector);
}
```

B.20 testkm.c

```
/*************************************************************************\
*                                                                         *
*                           Phonetic Classifier                           *
*                                                                         *
*                     Speech Recognition by Clustering                    *
*                       Wavelet and PLP Coefficients                      *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                    and                                  *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                  at the                                 *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                                May 15, 97                               *
\*************************************************************************/


/* Classifies vectors into their clusters */

#include "common.h"


void classify(float **class, km_t *km, float *vector, int vectorN) {
  /* Effects: Classifies a set of vectorN vectors given by vector into
     their corresponding clusters (defined by km).  The cluster indices
     and phoneme labels are stored in *class. */
  /* Modifies: *class */
  int i, j;
  int nn;

  *class = (float *)malloc(vectorN * 2 * sizeof(float));
  for (i=0; i<vectorN; i++) {
    normalize(vector + i*(km->dim+1), km->dim, km->mu, km->gamma);
    nn = nearestNeighbor(vector + i*(km->dim+1), km->meanv, km->K, km->dim);
    (*class)[i] = nn;
    (*class + vectorN)[i] = (vector + i*(km->dim+1))[km->dim];
  }
}




main(int argc, char *argv[]) {
  km_t km;
```

```
  float *data;
  float *vector;
  int vectorN, dim;
  float *class;

  if (argc < 4) {
    fprintf(stderr, "Usage: %s <in>.km <test>.dat <class-lbl>.dat\n", argv[0]);
    exit(-1);
  }
  readKm(argv[1], &km);
  readDat(argv[2], &data, &vectorN, &dim);
  if (dim != km.dim + 1) {
    fprintf(stderr, "%s: Vector dimensions do not match\n", argv[0]);
    exit(-1);
  }
  transpose(&vector, data, vectorN, dim);
  classify(&class, &km, vector, vectorN);
  writeDat(argv[3], "class", class, vectorN, 2);
  free(data);
  free(vector);
  free(class);
}
```

B.21 createpc.c

```
/*************************************************************************\
*                                                                         *
*                           Phonetic Classifier                           *
*                                                                         *
*                     Speech Recognition by Clustering                    *
*                       Wavelet and PLP Coefficients                      *
*                                                                         *
*                 Thesis submitted in partial fulfillment of              *
*                     the requirements for the degrees of                 *
*          Bachelor of Science in Computer Science and Engineering        *
*                                    and                                  *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                  at the                                 *
*                   Massachusetts Institute of Technology                 *
*                                                                         *
*                    Chiangkai Er <changkai@alum.mit.edu>                 *
*                                May 15, 97                               *
\*************************************************************************/


/* Creates Phoneme Classifier statistics file */

#include "common.h"


void createPc(pc_t *pc, float *data, int len, int K, int phoneN) {
  int i, p;
```

```
  int phone;
  int class;

  pc->K = K;
  pc->phoneN = phoneN;
  pc->score = (float *)malloc(K * phoneN * sizeof(float));
  pc->count = (unsigned long *)malloc(K * sizeof(unsigned long));
  for (i=0; i<K; i++) {
    for (p=0; p<phoneN; p++) {
      (pc->score + i*phoneN)[p] = 0.0;
    }
    pc->count[i] = 0;
  }
  for (i=0; i<len; i++) {
    class = (int)data[i];
    phone = (int)(data + len)[i];
    for (p=0; p<phoneN; p++) {
      if (p == phone) {
        (pc->score + class*phoneN)[p] =
          ((pc->score + class*phoneN)[p] * pc->count[class] + 1.0) /
          (pc->count[class] + 1);
      }
      else {
        (pc->score + class*phoneN)[p] =
          ((pc->score + class*phoneN)[p] * pc->count[class]) /
          (pc->count[class] + 1);
      }
    }
    (pc->count[class])++;
  }
}


main(int argc, char *argv[]) {
  float *data;
  int len, scales;
  pc_t pc;

  if (argc < 5) {
    fprintf(stderr, "Usage: %s ", argv[0]);
    fprintf(stderr, "<out>.pc <class-lbl>.dat <K> <phoneN>\n");
    exit(-1);
  }
  readDat(argv[2], &data, &len, &scales);
  if (scales != 2) {
    fprintf(stderr, "%s: %s is not a class vector\n",
            argv[0], argv[2]);
    exit(-1);
  }
  createPc(&pc, data, len, atoi(argv[3]), atoi(argv[4]));
  writePc(argv[1], &pc);
  free(data);
}
```

B.22 testpc.c

```
/*************************************************************************\
*                                                                         *
*                           Phonetic Classifier                           *
*                                                                         *
*                     Speech Recognition by Clustering                    *
*                       Wavelet and PLP Coefficients                      *
*                                                                         *
*                 Thesis submitted in partial fulfillment of              *
*                     the requirements for the degrees of                 *
*          Bachelor of Science in Computer Science and Engineering        *
*                                    and                                  *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                  at the                                 *
*                   Massachusetts Institute of Technology                 *
*                                                                         *
*                    Chiangkai Er <changkai@alum.mit.edu>                 *
*                                 May 15, 97                              *
\*************************************************************************/


/* Performs phonetic classification on test vectors */

#include "common.h"


void nearestNeighbors(float vector[], float *meanv,
                      int K, int dim, int nn[], int Knn) {
  /* Effects: Identifies the Knn centroids in meanv that are nearest
     to vector.  K is the no. of centroids in meanv and dim is the
     no. of dimensions.  The centroid indices are stored in nn. */
  /* Modifies: nn */
  int m, i, j;
  float dist;
  float *nndist;

  nndist = (float *)malloc(Knn * sizeof(float));
  /* Initialize */
  for (i=0; i<Knn; i++) {
    nndist[i] = FLT_MAX;
    nn[i] = -1;
  }
  /* Test each centroid and sort the Knn nearest ones */
  for (m=0; m<K; m++) {
    dist = distance(vector, meanv + m*dim, dim);
    /* Look for insertion point */
    for (i=Knn-1; i>=0; i--) {
      if (dist >= nndist[i]) break;
    }
    i++;
    if (i < Knn) {
      /* Insert */
      for (j=Knn-1; j>i; j--) {
```

```
        nndist[j] = nndist[j - 1];
        nn[j] = nn[j - 1];
      }
      nndist[i] = dist;
      nn[i] = m;
    }
  }
  free(nndist);
}


void getKnn(int **class, km_t *km, float *vector, int vectorN, int Knn) {
  /* Effects: Obtains Knn nearest centroids for each vector in the set
     of vectorN vectors (vector).  Centroids are given by km.  The
     centroid indices for each vector are stored in *class. */
  /* Modifies: *class */
  int i;

  *class = (int *)malloc(vectorN * Knn * sizeof(int));
  for (i=0; i<vectorN; i++) {
    normalize(vector + i*(km->dim+1), km->dim, km->mu, km->gamma);
    nearestNeighbors(vector + i*(km->dim+1), km->meanv,
                     km->K, km->dim, *class + i*Knn, Knn);
  }
}


void classify(pc_t *pc, int *class, int len, int Knn, float **score) {
  /* Effects: Performs phonetic classification using Knn centroid
     indices for each phoneme, given in class (contains len phonemes).
     Phoneme statistics are given by pc and the summed statistics
     are stored in *score. */
  /* Modifies: *score */
  int t, p, i;
  int phoneN;
  int currclass;

  phoneN = pc->phoneN;
  *score = (float *)malloc(phoneN * len * sizeof(float));
  for (t=0; t<len; t++) {
    for (p=0; p<phoneN; p++) {
      (*score + p*len)[t] = 0.0;
      for (i=0; i<Knn; i++) {
        currclass = (class + t*Knn)[i];
        if (currclass >= 0) {
          (*score + p*len)[t] += (pc->score + currclass*phoneN)[p];
        }
      }
    }
  }
}


void writeResult(char filename[], int len, int phoneN,
```

```
                float *score, float *vector, int dim) {
  /* Effects: Writes the result of phonetic classification on vector
     into result file (filename). The correct label is written along
     with the classified label.  len is the length of vector; dim is
     the no. of dimensions; and phoneN is the no. of phonemes.  score
     contains the summed phoneme statistics for each vector. */
  int t, p;
  FILE *outFile;
  int maxphone;
  float maxscore;

  if (!(outFile = fopen(filename, "w"))) {
    fprintf(stderr, "writeResult: Cannot create %s\n", filename);
    exit(-1);
  }
  /* Output most-represented phoneme for each vector */
  for (t=0; t<len; t++) {
    maxscore = -1.0;
    for (p=0; p<phoneN; p++) {
      if ((score + p*len)[t] > maxscore) {
        maxphone = p;
        maxscore = (score + p*len)[t];
      }
    }
    fprintf(outFile, "%d %d\n", maxphone, (int)(vector + t*dim)[dim - 1]);
  }
  fclose(outFile);
}



main(int argc, char *argv[]) {
  int Knn;
  km_t km;
  float *data;
  float *vector;
  int vectorN, dim;
  int *class;
  pc_t pc;
  float *score;

  if (argc < 6) {
    fprintf(stderr, "Usage: %s ", argv[0]);
    fprintf(stderr, "<in>.km <in>.pc <test>.dat <out>.dat <Knn>\n");
    exit(-1);
  }
  Knn = atoi(argv[5]);
  readKm(argv[1], &km);
  readPc(argv[2], &pc);
  readDat(argv[3], &data, &vectorN, &dim);
  if (dim != km.dim + 1) {
    fprintf(stderr, "%s: Vector dimensions do not match\n", argv[0]);
    exit(-1);
```

```
  }
  transpose(&vector, data, vectorN, dim);
  getKnn(&class, &km, vector, vectorN, Knn);
  classify(&pc, class, vectorN, Knn, &score);
  writeResult(argv[4], vectorN, pc.phoneN, score, vector, dim);
  free(data);
  free(vector);
  free(class);
  free(score);
}
```

B.23 initsc.c

```
/*************************************************************************\
*                                                                         *
*                           Phonetic Classifier                           *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                  and                                    *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                  at the                                 *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                              May 15, 97                                 *
\*************************************************************************/


/* Initializes the score file */

#include "common.h"


void initSc(char filename[], int phoneN) {
  /* Effects: Creates score file (filename) with zero entries. */
  int i, j;
  FILE *scFile;

  if (!(scFile = fopen(filename, "w"))) {
    fprintf(stderr, "initSc: Cannot create %s\n", filename);
    exit(-1);
  }
  fprintf(scFile, "%d\n", phoneN);
  for (i=0; i<phoneN; i++) {
    for (j=0; j<phoneN; j++) {
      fprintf(scFile, "0\t");
    }
```

```
    fprintf(scFile, "\n");
  }
  fclose(scFile);
}



main(int argc, char *argv[]) {
  pc_t pc;

  if (argc < 3) {
    fprintf(stderr, "Usage: %s <in>.pc <out>.sc\n", argv[0]);
    exit(-1);
  }
  readPc(argv[1], &pc);
  initSc(argv[2], pc.phoneN);
}
```

B.24 updatesc.c

```
/************************************************************************\
*                                                                        *
*                          Phonetic Classifier                           *
*                                                                        *
*                    Speech Recognition by Clustering                    *
*                      Wavelet and PLP Coefficients                      *
*                                                                        *
*                Thesis submitted in partial fulfillment of              *
*                    the requirements for the degrees of                 *
*         Bachelor of Science in Computer Science and Engineering        *
*                                  and                                   *
*  Master of Engineering in Electrical Engineering and Computer Science  *
*                                 at the                                 *
*                  Massachusetts Institute of Technology                 *
*                                                                        *
*                  Chiangkai Er <changkai@alum.mit.edu>                  *
*                             May 15, 97                                 *
\************************************************************************/


/* Updates the score file */

#include "common.h"

#define MAX_RESULT 100000    /* Maximum size of result file */


int readResult(char filename[], int phoneme[][2]) {
  /* Effects: Reads results file (filename) into phoneme. */
  /* Modifies: phoneme */
  FILE *resFile;
```

```
  int len;
  int x;

  if (!(resFile = fopen(filename, "r"))) {
    fprintf(stderr, "readResult: Cannot open %s\n", filename);
    exit(-1);
  }
  len = 0;
  while ((len < MAX_RESULT) &&
         (fscanf(resFile, "%d", &x) != EOF)) {
    phoneme[len][0] = x;
    fscanf(resFile, "%d", &x);
    phoneme[len][1] = x;
    len++;
  }
  fclose(resFile);
  return len;
}


void updateSc(sc_t *sc, int phoneme[][2], int len) {
  /* Effects: Updates score structure sc using classification results
     in phoneme.  len is the no. of phonemes in phoneme. */
  /* Modifies: sc */
  int i;
  int phoneN;
  int phonelbl;

  phoneN = sc->phoneN;
  for (i=0; i<len; i++) {
    phonelbl = phoneme[i][1];
    ((sc->count + phonelbl*phoneN)[phoneme[i][0]])++;
  }
}




main(int argc, char *argv[]) {
  int phoneme[MAX_RESULT][2];
  int len;
  sc_t sc;

  if (argc < 3) {
    fprintf(stderr, "Usage: %s <in/out>.sc <result>.dat\n", argv[0]);
    exit(-1);
  }
  len = readResult(argv[2], phoneme);
  readSc(argv[1], &sc);
  updateSc(&sc, phoneme, len);
  writeSc(argv[1], &sc);
}
```

B.25 ga.c

```
/*************************************************************************\
*                                                                         *
*                          Phonetic Classifier                            *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                  and                                    *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                 at the                                  *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                              May 15, 97                                 *
\*************************************************************************/


/* Performs genetic algorithm to determine optimal weights of coefficients */

#include "common.h"

#undef  MIN_GAMMA

#define ERR_THRES    0.001    /* Error threshold for centroids */
#define EPSILON      0.04     /* Epsilon used in splitting */
#define MIN_GAMMA    0.01     /* Minimum gamma for vector normalization */
#define MAX_GAMMA    100.0    /* Maximum gamma for vector normalization */

#define POPSIZE      4        /* Population size */
#define OFFSPRINGS   2        /* No. of offsprings per generation */
#define PARENT_N     (POPSIZE - OFFSPRINGS)  /* No. of parents */
#define MAXGEN       100000   /* Maximum no. of generations */
#define MUTATE_AMP   1.0      /* Mutation amplitude */
#define CROSSOVER_P  0.1      /* Crossover probability */
#define MUTATE_P     0.05     /* Mutation probability */


/* Initial gamma */
float initgamma[42] = {
  25.172358, 19.218266, 13.333363, 13.331610, 18.966608, 25.621672,
  33.172913, 37.476593, 1.096734,  0.230770,  0.337857,  0.225993,
  0.182156,  0.067926,  23.546637, 17.280016, 11.299080, 11.545583,
  17.581976, 24.429148, 32.420090, 36.827389, 1.184915,  0.251185,
  0.355919,  0.230769,  0.185793,  0.069614,  27.508614, 21.645977,
  15.939310, 15.503012, 20.481485, 27.137836, 34.634777, 38.795593,
  0.951037,  0.217823,  0.321228,  0.218384,  0.186709,  0.071181
};
```

```
void initKm(km_t *km, int K, int phoneN, int dim) {
  /* Effects: Initializes km with K, phoneN and dim. */
  /* Modifies: km */

  km->K = K;
  km->dim = dim;
  km->mu = (float *)malloc(dim * sizeof(float));
  km->gamma = (float *)malloc(dim * sizeof(float));
  km->meanv = (float *)malloc(K * dim * sizeof(float));
}


float fitnessKm(km_t *km, int K, int phoneN,
                float *vector, int vectorN, int dimp1, float gamma[]) {
  /* Effects: Computes fitness of gamma by performing clustering on
     vector, which contains vectorN vectors of (dimp1 - 1) dimensions.
     K clusters are to be found and there are phoneN phonemes.  vector
     is normalized before clustering begins.  Clustering results are
     stored in km. */
  /* Modifies: km, vector */
  /* Returns: Fitness value */
  int i, j;
  int dim;            /* No. of dimensions */
  int N;              /* Current no. of centroids */
  int nn;             /* Nearest neighbor */
  float *centroid;    /* Current centroids */
  int *count;         /* No. of vectors corresponding to each centroid */
  int phoneme;        /* Current phoneme */
  int meanbase;       /* Current base index for meanv */
  int phoneK;         /* No. of clusters for current phoneme */
  float *proportion;  /* Phoneme proportion */
  int total;          /* Total no. of phonemes */
  int iter;           /* No. of iterations */
  float newval;       /* New value */
  float error;

  /* Phoneme classification variables */
  int p;
  int class;                    /* Cluster */
  float *pcscore;               /* Phoneme classifier score */
  unsigned long *pccount;       /* Phoneme classifier count */
  unsigned long totalpccount;  /* Total phoneme classifier count */
  float topscore;               /* Top-scoring phoneme */
  float fitness;                /* Fitness value */

  /* Initializations */
  dim = dimp1 - 1;
  centroid = (float *)malloc(K * dim * sizeof(float));
  count = (int *)malloc(K * sizeof(int));
  proportion = (float *)malloc(phoneN * sizeof(float));
  findNorm(km, vector, vectorN, dimp1, gamma);

  /* Compute phoneme proportions */
  for (i=0; i<phoneN; i++) {
```

```
  proportion[i] = 0.0;
}
total = 0;
for (i=0; i<vectorN; i++) {
  phoneme = (int)(vector + i*dimp1)[dimp1 - 1];
  if (phoneme >= 0) {
    proportion[phoneme] += 1.0;
    total++;
  }
}
for (i=0; i<phoneN; i++) {
  proportion[i] /= total;
}

meanbase = 0;
/* For each phoneme */
for (phoneme=0; phoneme<phoneN; phoneme++) {
  if (phoneme < phoneN - 1) {
    phoneK = round(proportion[phoneme] * K);
  }
  else {
    phoneK = K - meanbase;
  }
  /* Perform clustering for current phoneme */
  N = 1;
  do {
    for (i=0; i<N/2; i++) {
      if (N/2 + i < phoneK) {
        /* Binary split each centroid */
        for (j=0; j<dim; j++) {
          (km->meanv + (meanbase + N/2 + i)*dim)[j] =
            (1 + EPSILON) * (km->meanv + (meanbase + i)*dim)[j];
        }
        for (j=0; j<dim; j++) {
          (km->meanv + (meanbase + i)*dim)[j] *= (1 - EPSILON);
        }
      }
    }

    if (N > phoneK) N = phoneK;
    /* Perform iterations */
    iter = 1;
    do {
      clearVectors(centroid, N, dim);
      for (i=0; i<N; i++) count[i] = 0;
      /* Find nearest centroids and compute new centroids */
      for (i=0; i<vectorN; i++) {
        if (phoneme == (int)(vector + i*dimp1)[dimp1 - 1]) {
          nn = nearestNeighbor(vector + i*dimp1,
                               km->meanv + meanbase*dim, N, dim);
          for (j=0; j<dim; j++) {
            (centroid + nn*dim)[j] += (vector + i*dimp1)[j];
          }
          count[nn]++;
```

```
        }
      }
      error = 0.0;
      /* Update centroids and compute error */
      for (i=0; i<N; i++) {
        if (count[i] > 0) {
          for (j=0; j<dim; j++) {
            newval = (centroid + i*dim)[j] / count[i];
            error += square(newval - (km->meanv + (meanbase + i)*dim)[j]);
            (km->meanv + (meanbase + i)*dim)[j] = newval;
          }
        }
      }
      iter++;
    } while (error > ERR_THRES);
    N *= 2;
  } while (N < 2*phoneK);

  meanbase += phoneK;
}

free(count);
free(centroid);
free(proportion);

/* Phoneme classification */
pcscore = (float *)malloc(K * phoneN * sizeof(float));
pccount = (unsigned long *)malloc(K * sizeof(unsigned long));
/* Initialize */
for (i=0; i<K; i++) {
  for (p=0; p<phoneN; p++) {
    (pcscore + i*phoneN)[p] = 0.0;
  }
  pccount[i] = 0;
}
/* Collect statistics */
for (i=0; i<vectorN; i++) {
  class = nearestNeighbor(vector + i*dimp1, km->meanv, km->K, km->dim);
  phoneme = (int)(vector + i*dimp1)[dimp1 - 1];
  for (p=0; p<phoneN; p++) {
    if (p == phoneme) {
      (pcscore + class*phoneN)[p] =
        ((pcscore + class*phoneN)[p] * pccount[class] + 1.0) /
        (pccount[class] + 1);
    }
    else {
      (pcscore + class*phoneN)[p] =
        ((pcscore + class*phoneN)[p] * pccount[class]) /
        (pccount[class] + 1);
    }
    (pccount[class])++;
  }
}
```

```
  /* Compute fitness */
  fitness = 0.0;
  totalpccount = 0;
  for (i=0; i<K; i++) {
    topscore = -1.0;
    for (j=0; j<phoneN; j++) {
      if ((pcscore + i*phoneN)[j] > topscore) {
        topscore = (pcscore + i*phoneN)[j];
      }
    }
    fitness += topscore * pccount[i];
    totalpccount += pccount[i];
  }
  fitness /= totalpccount;

  free(pcscore);
  free(pccount);

  return fitness;
}


float randuniform(float amp) {
  /* Effects: Generates a uniformly-distributed random number between
     -amp and amp. */
  /* Returns: Random number */
  float r;

  r = (float)(random() % 10000) / 5000 - 1.0;
  r = r * amp;
  return r;
}


int randbernoulli(float prob) {
  /* Effects: Generates a Bernoulli random variable with prob as
     probability of returning a 1. */
  /* Returns: Random variable */
  float r;

  r = random() % 10000;
  return (r < 10000*prob);
}


void performGa(char filename[], int K, int phoneN,
               float *vector, int vectorN, int dimp1) {
  /* Effects: Performs genetic algorithm to optimize the gamma parameter
     for vector normalization.  Results of each generation are stored in
     file (filename).  Clustering is performed on vector, which contains
     vectorN vectors of (dimp1 - 1) dimensions.  K clusters are to be found
     and there are phoneN phonemes. */
  int i, j;
  int dim;              /* Vector dimension */
```

```
int gen;              /* Generation number */
FILE *gaFile;
km_t km;
float *currvector;    /* Current vector set */
gene_t pop[POPSIZE];  /* Population gene pool */
gene_t tempgene;      /* Temporary gene */
int p1, p2;           /* Parents */

/* Initializations */
initKm(&km, K, phoneN, dimp1 - 1);
dim = dimp1 - 1;
for (i=0; i<POPSIZE; i++) {
  pop[i].gamma = (float *)malloc(dim * sizeof(float));
  for (j=0; j<dim; j++) {
    pop[i].gamma[j] = initgamma[j % 42];
  }
}

if (!(gaFile = fopen(filename, "w"))) {
  fprintf(stderr, "performGa: Cannot open %s\n", filename);
  exit(-1);
}
currvector = (float *)malloc(vectorN * dimp1 * sizeof(float));
/* For each generation */
for (gen=1; gen<=MAXGEN; gen++) {
  printf("Processing generation %d\n", gen);
  /* Compute fitness for each individual */
  for (i=0; i<POPSIZE; i++) {
    for (j=0; j<vectorN*dimp1; j++) {
      currvector[j] = vector[j];
    }
    pop[i].fitness =
      fitnessKm(&km, K, phoneN, currvector, vectorN, dimp1, pop[i].gamma);
  }

  /* Sort the individuals by fitness */
  for (i=0; i<POPSIZE-1; i++) {
    for (j=i+1; j<POPSIZE; j++) {
      if (pop[i].fitness < pop[j].fitness) {
        tempgene = pop[i];
        pop[i] = pop[j];
        pop[j] = tempgene;
      }
    }
  }

  /* Output results to file */
  fprintf(gaFile, "%d\n", gen);
  for (i=0; i<PARENT_N; i++) {
    fprintf(gaFile, "%f\n", pop[i].fitness);
    for (j=0; j<dim; j++) {
      fprintf(gaFile, "%f ", pop[i].gamma[j]);
    }
    fprintf(gaFile, "\n");
```

```
    }
    fprintf(gaFile, "\n");
    fflush(gaFile);

    /* Generating offsprings */
    for (i=PARENT_N; i<POPSIZE; i++) {
      for (j=1; j<dim; j++) {
        p1 = random() % PARENT_N;
        if (randbernoulli(CROSSOVER_P)) {
          /* Perform cross-over */
          p2 = random() % PARENT_N;
          pop[i].gamma[j] = (pop[p1].gamma[j] + pop[p2].gamma[j]) / 2;
        }
        else if (randbernoulli(MUTATE_P)) {
          /* Perform mutation */
          pop[i].gamma[j] =
            pop[p1].gamma[j] * pow(2.0, randuniform(MUTATE_AMP));
          if (pop[i].gamma[j] < MIN_GAMMA) {
            pop[i].gamma[j] = MIN_GAMMA;
          }
          else if (pop[i].gamma[j] > MAX_GAMMA) {
            pop[i].gamma[j] = MAX_GAMMA;
          }
        }
        else {
          /* Inherit from parent */
          pop[i].gamma[j] = pop[p1].gamma[j];
        }
      }
    }
  }
  free(currvector);
  close(gaFile);
}



main(int argc, char *argv[]) {
  float *data;
  float *vector;
  int vectorN, dimp1;
  km_t km;

  if (argc < 5) {
    fprintf(stderr, "Usage: %s <out>.ga <train>.dat <K> <phoneN>\n", argv[0]);
    exit(-1);
  }
  srandom(getpid());
  readDat(argv[2], &data, &vectorN, &dimp1);
  transpose(&vector, data, vectorN, dimp1);
  performGa(argv[1], atoi(argv[3]), atoi(argv[4]),
            vector, vectorN, dimp1);
  free(data);
```

```
  free(vector);
}
```

B.26 present.c

```
/*************************************************************************\
*                                                                         *
*                          Phonetic Classifier                            *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                  and                                    *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                 at the                                  *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                              May 15, 97                                 *
\*************************************************************************/


/* Presents score in summarized text form */

#include "common.h"


void summarize(sc_t *sc) {
  /* Effects: Summarizes the score data (sc) to show only classification
     accuracies.  The summary is printed in stdout. */
  int i, j;
  int phoneN;
  float fraction;
  int count;
  int hit, total;

  hit = 0;
  total = 0;
  /* Classification accuracy for each phoneme */
  for (i=0; i<phoneN; i++) {
    count = 0;
    for (j=0; j<phoneN; j++) {
      count += (sc->count + i*phoneN)[j];
    }
    fraction = (float)(sc->count + i*phoneN)[i] / count;
    printf("%d\t%d\t%d\t%f\n",
           i, (sc->count + i*phoneN)[i], count, fraction);
    total += count;
    hit += (sc->count + i*phoneN)[i];
```

```
  }
  /* Average classification accuracy */
  fraction = (float)hit / total;
  printf("%d\t%d\t%f\n", hit, total, fraction);
}



main(int argc, char *argv[]) {
  sc_t sc;

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <in>.sc\n", argv[0]);
    exit(-1);
  }
  readSc(argv[1], &sc);
  summarize(&sc);
}
```

B.27 cmatrix.c

```
/*************************************************************************\
*                                                                         *
*                           Phonetic Classifier                           *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                    and                                  *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                  at the                                 *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                                May 15, 97                               *
\*************************************************************************/


/* Presents Confusion Matrix in text form */

#include "common.h"


void confusionMatrix(sc_t *sc) {
  /* Effects: Prints confusion matrix corresponding to sc in stdout. */
  int i, j;
  int phoneN;
  float fraction;
```

```
  int count;
  int hit, total;

  phoneN = sc->phoneN;
  hit = 0;
  total = 0;
  for (i=0; i<phoneN; i++) {
    /* Count no. of tokens of phoneme i */
    count = 0;
    for (j=0; j<phoneN; j++) {
      count += (sc->count + i*phoneN)[j];
    }
    /* Present phoneme classification accuracy */
    fraction = (float)(sc->count + i*phoneN)[i] / count;
    printf("%2d: %5d %5d %2.0f    ",
           i, (sc->count + i*phoneN)[i], count, fraction * 100);
    /* Present row i of confusion matrix */
    for (j=0; j<phoneN; j++) {
      if ((sc->count + i*phoneN)[j] == 0) {
        printf(" . ");
      }
      else {
        printf("%2.0f ", (float)(sc->count + i*phoneN)[j] / count * 100);
      }
    }
    printf("\n");
    total += count;
    hit += (sc->count + i*phoneN)[i];
  }
  /* Present average accuracy */
  fraction = (float)hit / total;
  printf("    %5d %5d %4.1f\n", hit, total, fraction * 100);
}




main(int argc, char *argv[]) {
  sc_t sc;

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <in>.sc\n", argv[0]);
    exit(-1);
  }
  readSc(argv[1], &sc);
  confusionMatrix(&sc);
}
```

B.28 cmtexv.c

```
/*************************************************************************\
*                                                                         *
```

```
*                          Phonetic Classifier                          *
*                                                                       *
*                    Speech Recognition by Clustering                   *
*                      Wavelet and PLP Coefficients                     *
*                                                                       *
*                Thesis submitted in partial fulfillment of             *
*                    the requirements for the degrees of                *
*         Bachelor of Science in Computer Science and Engineering       *
*                                  and                                  *
*  Master of Engineering in Electrical Engineering and Computer Science *
*                                 at the                                *
*                  Massachusetts Institute of Technology                *
*                                                                       *
*                   Chiangkai Er <changkai@alum.mit.edu>                *
*                                May 15, 97                             *
\***********************************************************************/


/* Presents Confusion Matrix (vowels) in LaTeX form */

#include "common.h"

#define PHONE_N 16    /* No. of phonemes */


/* Confusion matrix phoneme to label mapping */
int ndx[PHONE_N] = {
   0,  1,  2,  3,
   4,  7, 10,  5,
   6,  9,  8, 15,
  11, 12, 13, 14
};


void confusionMatrix(sc_t *sc, int ndx[], char *phones[]) {
  /* Effects: Prints confusion matrix corresponding to sc in stdout.  The
     confusion matrix is a LaTeX table with phonemes obtained from phones[].
     ndx is the mapping from confusion matrix phoneme to label. */
  int i, j;
  int phoneN;
  float fraction;
  int count;
  int hit, total;

  phoneN = sc->phoneN;
  printf("\\begin{tiny}\n");
  printf("\\begin{table}[htbp]\n");
  printf("\\caption{Confusion matrix}\n");
  printf("\\centerline{\n");
  printf("\\begin{tabular}{|c||c|c|c|c||");
  for (i=0; i<phoneN; i++) {
    printf("c|");
  }
  printf("}\n");
```

```
  printf("\\hline\n");
  printf(" & {\\bf Hit} & {\\bf Total} & {\\bf \\%%}\n");
  for (i=0; i<phoneN; i++) {
    printf(" & {\\bf %s}", phones[i]);
  }
  printf("\\\\\n");
  printf("\\hline\n\\hline\n");
  hit = 0;
  total = 0;
  /* For each phoneme */
  for (i=0; i<phoneN; i++) {
    /* Count no. of tokens of phoneme i */
    count = 0;
    for (j=0; j<phoneN; j++) {
      count += (sc->count + ndx[i]*phoneN)[ndx[j]];
    }
    /* Present phoneme classification accuracy */
    fraction = (float)(sc->count + ndx[i]*phoneN)[ndx[i]] / count;
    printf("{\\bf %s} & ", phones[i]);
    printf("%d & %d & %2.0f",
           (sc->count + ndx[i]*phoneN)[ndx[i]], count, fraction * 100);
    /* Present row i of confusion matrix */
    for (j=0; j<phoneN; j++) {
      if ((sc->count + ndx[i]*phoneN)[ndx[j]] == 0) {
        printf(" & .");
      }
      else {
        printf(" & %2.0f ",
               (float)(sc->count + ndx[i]*phoneN)[ndx[j]] / count * 100);
      }
    }
    printf("\\\\\n");
    printf("\\hline\n");
    total += count;
    hit += (sc->count + ndx[i]*phoneN)[ndx[i]];
  }
  /* Present average accuracy */
  printf("\\hline\n");
  fraction = (float)hit / total;
  printf(" & {\\bf %d} & {\\bf %d} & {\\bf %4.1f}",
         hit, total, fraction * 100);
  printf(" & \\multicolumn{%d}{c|}{}\\\\\n", phoneN);
  printf("\\hline\n");
  printf("\\end{tabular}\n");
  printf("}\n");
  printf("\\end{table}\n");
  printf("\\end{tiny}\n");
}



main(int argc, char *argv[]) {
  sc_t sc;
```

```
  char *phones[PHONE_N];

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <in>.sc\n", argv[0]);
    exit(-1);
  }
  readSc(argv[1], &sc);
  if (sc.phoneN != PHONE_N) {
    fprintf(stderr, "%s: Score file should contain %d phonemes\n",
            argv[0], PHONE_N);
    exit(-1);
  }
  phones[0] = strdup("iy");
  phones[1] = strdup("ih");
  phones[2] = strdup("ey");
  phones[3] = strdup("eh");
  phones[4] = strdup("ae");
  phones[5] = strdup("aa");
  phones[6] = strdup("ow");
  phones[7] = strdup("ah");
  phones[8] = strdup("ao");
  phones[9] = strdup("uw");
  phones[10] = strdup("uh");
  phones[11] = strdup("ux");
  phones[12] = strdup("er");
  phones[13] = strdup("ay");
  phones[14] = strdup("oy");
  phones[15] = strdup("aw");
  confusionMatrix(&sc, ndx, phones);
}
```

B.29 cmtexc.c

```
/************************************************************************\
*                                                                        *
*                          Phonetic Classifier                           *
*                                                                        *
*                    Speech Recognition by Clustering                    *
*                      Wavelet and PLP Coefficients                      *
*                                                                        *
*                Thesis submitted in partial fulfillment of              *
*                    the requirements for the degrees of                 *
*         Bachelor of Science in Computer Science and Engineering        *
*                                  and                                   *
*  Master of Engineering in Electrical Engineering and Computer Science  *
*                                at the                                  *
*                  Massachusetts Institute of Technology                 *
*                                                                        *
*                  Chiangkai Er <changkai@alum.mit.edu>                  *
*                              May 15, 97                                *
\************************************************************************/
```

```
/* Presents Confusion Matrix (consonants) in LaTeX form */

#include "common.h"

#define PHONE_N 24    /* No. of phonemes */
#define LINE_N  12    /* No. of phonemes per line of confusion matrix */


/* Confusion matrix phoneme to label mapping */
int ndx[PHONE_N] = {
   0,  1,  2,  3,
   4,  5,  6,  7,
   8,  9, 10, 11,
  12, 13, 14, 15,
  16, 17, 18, 19,
  20, 21, 22, 23
};


void confusionMatrix(sc_t *sc, int ndx[], char *phones[]) {
  /* Effects: Prints confusion matrix corresponding to sc in stdout.  The
     confusion matrix is a LaTeX table with phonemes obtained from phones[].
     ndx is the mapping from confusion matrix phoneme to label. */
  int i, j;
  int t, tN;      /* Table index and no. of tables */
  int lastphone;  /* Last phoneme of table */
  int phoneN;
  float fraction;
  int count;
  int hit, total;

  phoneN = sc->phoneN;
  tN = (phoneN - 1) / LINE_N + 1;
  /* For each table */
  for (t=0; t<tN; t++) {
    lastphone = (t + 1) * LINE_N;
    if (lastphone > phoneN) lastphone = phoneN;
    printf("\\begin{tiny}\n");
    printf("\\begin{table}[htbp]\n");
    printf("\\caption{Confusion matrix (%d of %d)}\n", t + 1, tN);
    printf("\\centerline{\n");
    printf("\\begin{tabular}{|c||c|c|c||");
    for (i=t*LINE_N; i<lastphone; i++) {
      printf("c|");
    }
    printf("}\n");
    printf("\\hline\n");
    printf(" & {\\bf Hit} & {\\bf Total} & {\\bf \\%%}\n");
    for (i=t*LINE_N; i<lastphone; i++) {
      printf(" & {\\bf %s}", phones[i]);
    }
    printf("\\\\\n");
    printf("\\hline\n\\hline\n");
```

```
    hit = 0;
    total = 0;
    /* For each phoneme */
    for (i=0; i<phoneN; i++) {
      /* Count no. of tokens of phoneme i */
      count = 0;
      for (j=0; j<phoneN; j++) {
        count += (sc->count + ndx[i]*phoneN)[ndx[j]];
      }
      /* Present phoneme classification accuracy */
      fraction = (float)(sc->count + ndx[i]*phoneN)[ndx[i]] / count;
      printf("{\\bf %s} & ", phones[i]);
      printf("%d & %d & %2.0f",
             (sc->count + ndx[i]*phoneN)[ndx[i]], count, fraction * 100);
      /* Present row i of confusion matrix */
      for (j=t*LINE_N; j<lastphone; j++) {
        if ((sc->count + ndx[i]*phoneN)[ndx[j]] == 0) {
          printf(" & .");
        }
        else {
          printf(" & %2.0f ",
                 (float)(sc->count + ndx[i]*phoneN)[ndx[j]] / count * 100);
        }
      }
      printf("\\\\\n");
      printf("\\hline\n");
      total += count;
      hit += (sc->count + ndx[i]*phoneN)[ndx[i]];
    }
    /* Present average accuracy */
    printf("\\hline\n");
    fraction = (float)hit / total;
    printf(" & {\\bf %d} & {\\bf %d} & {\\bf %4.1f}",
           hit, total, fraction * 100);
    printf(" & \\multicolumn{%d}{c|}{}\\\\\n", lastphone - t*LINE_N);
    printf("\\hline\n");
    printf("\\end{tabular}\n");
    printf("}\n");
    printf("\\end{table}\n");
    printf("\\end{tiny}\n\n");
  }
}



main(int argc, char *argv[]) {
  sc_t sc;
  char *phones[PHONE_N];

  if (argc < 2) {
    fprintf(stderr, "Usage: %s <in>.sc\n", argv[0]);
    exit(-1);
  }
```

```
  readSc(argv[1], &sc);
  if (sc.phoneN != PHONE_N) {
    fprintf(stderr, "%s: Score file should contain %d phonemes\n",
            argv[0], PHONE_N);
    exit(-1);
  }
  phones[0] = strdup("p");
  phones[1] = strdup("b");
  phones[2] = strdup("t");
  phones[3] = strdup("d");
  phones[4] = strdup("k");
  phones[5] = strdup("g");
  phones[6] = strdup("f");
  phones[7] = strdup("v");
  phones[8] = strdup("th");
  phones[9] = strdup("dh");
  phones[10] = strdup("s");
  phones[11] = strdup("z");
  phones[12] = strdup("sh");
  phones[13] = strdup("zh");
  phones[14] = strdup("hh");
  phones[15] = strdup("ch");
  phones[16] = strdup("jh");
  phones[17] = strdup("m");
  phones[18] = strdup("n");
  phones[19] = strdup("ng");
  phones[20] = strdup("w");
  phones[21] = strdup("l");
  phones[22] = strdup("r");
  phones[23] = strdup("y");
  confusionMatrix(&sc, ndx, phones);
}
```

B.30 cmtexp.c

```
/*************************************************************************\
*                                                                         *
*                          Phonetic Classifier                            *
*                                                                         *
*                    Speech Recognition by Clustering                     *
*                      Wavelet and PLP Coefficients                       *
*                                                                         *
*                Thesis submitted in partial fulfillment of               *
*                    the requirements for the degrees of                  *
*         Bachelor of Science in Computer Science and Engineering         *
*                                    and                                  *
*  Master of Engineering in Electrical Engineering and Computer Science   *
*                                  at the                                 *
*                  Massachusetts Institute of Technology                  *
*                                                                         *
*                   Chiangkai Er <changkai@alum.mit.edu>                  *
*                              May 15, 97                                 *
```

```
\*************************************************************************/


/* Presents Confusion Matrix (phonemes) in LaTeX form */

#include "common.h"

#define PHONE_N 39    /* No. of phonemes */
#define LINE_N  13    /* No. of phonemes per line of confusion matrix */


/* Confusion matrix phoneme to label mapping */
int ndx[PHONE_N] = {
   0,  1,  2,  3,
   4,  5,  6,  7,
   8,  9, 10, 11,
  12, 13, 14, 15,
  16, 17, 18, 19,
  20, 21, 22, 23,
  24, 25, 26, 27,
  28, 29, 30, 31,
  32, 33, 34, 35,
  36, 37, 38
};


void confusionMatrix(sc_t *sc, int ndx[]) {
  /* Effects: Prints confusion matrix corresponding to sc in stdout.  The
     confusion matrix is a LaTeX table.  ndx is the mapping from
     confusion matrix phoneme to label. */
  int i, j;
  int t, tN;      /* Table index and no. of tables */
  int lastphone;  /* Last phoneme of table */
  int phoneN;
  float fraction;
  int count;
  int hit, total;

  phoneN = sc->phoneN;
  tN = (phoneN - 1) / LINE_N + 1;
  /* For each table */
  for (t=0; t<tN; t++) {
    lastphone = (t + 1) * LINE_N;
    if (lastphone > phoneN) lastphone = phoneN;
    printf("\\begin{tiny}\n");
    printf("\\begin{table}[htbp]\n");
    printf("\\caption{Confusion matrix (%d of %d)}\n", t + 1, tN);
    printf("\\centerline{\n");
    printf("\\begin{tabular}{|c||c|c|c||");
    for (i=t*LINE_N; i<lastphone; i++) {
      printf("c|");
    }
    printf("}\n");
    printf("\\hline\n");
```

```
    printf(" & {\\bf Hit} & {\\bf Total} & {\\bf \\%%}\n");
    for (i=t*LINE_N; i<lastphone; i++) {
      printf(" & {\\bf %d}", i + 1);
    }
    printf("\\\\\n");
    printf("\\hline\n\\hline\n");
    hit = 0;
    total = 0;
    /* For each phoneme */
    for (i=0; i<phoneN; i++) {
      /* Count no. of tokens of phoneme i */
      count = 0;
      for (j=0; j<phoneN; j++) {
        count += (sc->count + ndx[i]*phoneN)[ndx[j]];
      }
      /* Present phoneme classification accuracy */
      fraction = (float)(sc->count + ndx[i]*phoneN)[ndx[i]] / count;
      printf("{\\bf %d} & ", i + 1);
      printf("%d & %d & %2.0f",
             (sc->count + ndx[i]*phoneN)[ndx[i]], count, fraction * 100);
      /* Present row i of confusion matrix */
      for (j=t*LINE_N; j<lastphone; j++) {
        if ((sc->count + ndx[i]*phoneN)[ndx[j]] == 0) {
          printf(" & .");
        }
        else {
          printf(" & %2.0f ",
                 (float)(sc->count + ndx[i]*phoneN)[ndx[j]] / count * 100);
        }
      }
      printf("\\\\\n");
      printf("\\hline\n");
      total += count;
      hit += (sc->count + ndx[i]*phoneN)[ndx[i]];
    }
    /* Present average accuracy */
    printf("\\hline\n");
    fraction = (float)hit / total;
    printf(" & {\\bf %d} & {\\bf %d} & {\\bf %4.1f}",
           hit, total, fraction * 100);
    printf(" & \\multicolumn{%d}{c|}{}\\\\\n", lastphone - t*LINE_N);
    printf("\\hline\n");
    printf("\\end{tabular}\n");
    printf("}\n");
    printf("\\end{table}\n");
    printf("\\end{tiny}\n\n");
  }
}



main(int argc, char *argv[]) {
  sc_t sc;
```

```
  if (argc < 2) {
    fprintf(stderr, "Usage: %s <in>.sc\n", argv[0]);
    exit(-1);
  }
  readSc(argv[1], &sc);
  if (sc.phoneN != PHONE_N) {
    fprintf(stderr, "%s: Score file should contain %d phonemes\n",
            argv[0], PHONE_N);
    exit(-1);
  }
  confusionMatrix(&sc, ndx);
}
```

B.31 Makefile

```
###############################################################################
#                                                                             #
#                           Phonetic Classifier                               #
#                                                                             #
#                    Speech Recognition by Clustering                         #
#                      Wavelet and PLP Coefficients                           #
#                                                                             #
#                Thesis submitted in partial fulfillment of                   #
#                    the requirements for the degrees of                      #
#         Bachelor of Science in Computer Science and Engineering             #
#                                    and                                      #
#  Master of Engineering in Electrical Engineering and Computer Science       #
#                                  at the                                     #
#                  Massachusetts Institute of Technology                      #
#                                                                             #
#                   Chiangkai Er <changkai@alum.mit.edu>                      #
#                                May 15, 97                                   #
###############################################################################

CC              = gcc

# Uncomment these if MATLAB is used
#MATLAB         = /mit/matlab/Matlab4.2
#ARCH           = sol2
#INCLUDES       = -I$(MATLAB)/extern/include
#LIBRARIES      = $(MATLAB)/extern/lib/$(ARCH)/libmat.a -lm

# Comment these if MATLAB is used
INCLUDES        =
LIBRARIES       = -lm

PROGRAMS =      au2dat wt ds swt plp \
                catsets mergesets \
                makephonedat catphonedat \
                createkm1 createkm2 createkm3 createkm4 \
                createkm5 createkm6 createkm7 createkm8 \
```

```
                testkm \
                createpc testpc \
                initsc updatesc \
                ga \
                present cmatrix cmtexv cmtexc cmtexp

all: $(PROGRAMS)

.c.o:
        $(CC) -c $(INCLUDES) $*.c

au2dat: au2dat.o common.o
        $(CC) -o au2dat au2dat.o common.o $(LIBRARIES)

wt: wt.o common.o
        $(CC) -o wt wt.o common.o $(LIBRARIES)

ds: ds.o common.o
        $(CC) -o ds ds.o common.o $(LIBRARIES)

swt: swt.o common.o
        $(CC) -o swt swt.o common.o $(LIBRARIES)

plp: plp.o common.o
        $(CC) -o plp plp.o common.o $(LIBRARIES)

catsets: catsets.o common.o
        $(CC) -o catsets catsets.o common.o $(LIBRARIES)

mergesets: mergesets.o common.o
        $(CC) -o mergesets mergesets.o common.o $(LIBRARIES)

makephonedat: makephonedat.o common.o
        $(CC) -o makephonedat makephonedat.o common.o $(LIBRARIES)

catphonedat: catphonedat.o common.o
        $(CC) -o catphonedat catphonedat.o common.o $(LIBRARIES)

createkm1: createkm1.o common.o
        $(CC) -o createkm1 createkm1.o common.o $(LIBRARIES)

createkm2: createkm2.o common.o
        $(CC) -o createkm2 createkm2.o common.o $(LIBRARIES)

createkm3: createkm3.o common.o
        $(CC) -o createkm3 createkm3.o common.o $(LIBRARIES)

createkm4: createkm4.o common.o
        $(CC) -o createkm4 createkm4.o common.o $(LIBRARIES)

createkm5: createkm5.o common.o
        $(CC) -o createkm5 createkm5.o common.o $(LIBRARIES)

createkm6: createkm6.o common.o
```

```
	$(CC) -o createkm6 createkm6.o common.o $(LIBRARIES)

createkm7: createkm7.o common.o
	$(CC) -o createkm7 createkm7.o common.o $(LIBRARIES)

createkm8: createkm8.o common.o
	$(CC) -o createkm8 createkm8.o common.o $(LIBRARIES)

testkm: testkm.o common.o
	$(CC) -o testkm testkm.o common.o $(LIBRARIES)

createpc: createpc.o common.o
	$(CC) -o createpc createpc.o common.o $(LIBRARIES)

testpc: testpc.o common.o
	$(CC) -o testpc testpc.o common.o $(LIBRARIES)

initsc: initsc.o common.o
	$(CC) -o initsc initsc.o common.o $(LIBRARIES)

updatesc: updatesc.o common.o
	$(CC) -o updatesc updatesc.o common.o $(LIBRARIES)

ga: ga.o common.o
	$(CC) -o ga ga.o common.o $(LIBRARIES)

present: present.o common.o
	$(CC) -o present present.o common.o $(LIBRARIES)

cmatrix: cmatrix.o common.o
	$(CC) -o cmatrix cmatrix.o common.o $(LIBRARIES)

cmtexv: cmtexv.o common.o
	$(CC) -o cmtexv cmtexv.o common.o $(LIBRARIES)

cmtexc: cmtexc.o common.o
	$(CC) -o cmtexc cmtexc.o common.o $(LIBRARIES)

cmtexp: cmtexp.o common.o
	$(CC) -o cmtexp cmtexp.o common.o $(LIBRARIES)

clean:
	rm -f $(PROGRAMS) *.o
```

Appendix C

Confusion Matrices

The confusion matrices for experiments done in chapter 6 are presented in this appendix. They are classified under vowel, consonant and phoneme classifications. Please refer to table 6.2 for the abbreviations used for the various clustering algorithms.

C.1 Vowel Classification

Table C.1: Confusion matrix for BS (vowel classification)

	Hit	Total	%	iy	ih	ey	eh	ae	aa	ow	ah	ao	uw	uh	ux	er	ay	oy	aw
iy	195	243	80	80	11	5	0	0	.	.	.	.	.	.	2	0	.	.	.
ih	112	203	55	19	55	8	7	1	.	1	4	1	.	.	2	1	.	.	.
ey	52	114	46	16	29	46	7	1	.	.	.	.	.	.	.	.	2	.	.
eh	71	189	38	1	19	5	38	17	5	1	8	1	.	.	.	2	4	.	1
ae	56	105	53	3	8	5	17	53	6	1	1	.	.	.	.	1	6	.	.
aa	79	131	60	.	.	.	2	9	60	2	8	15	.	.	.	1	4	.	.
ow	46	89	52	.	.	.	6	2	4	52	13	16	1	.	.	3	1	.	1
ah	47	135	35	1	7	1	13	4	7	16	35	7	.	.	.	3	4	1	.
ao	66	97	68	.	.	.	4	1	18	3	5	68	.	.	.	.	1	.	.
uw	10	22	45	.	23	.	.	.	.	.	5	.	45	.	14	14	.	.	.
uh	0	29	0	3	52	.	7	.	.	17	.	3	3	.	.	10	3	.	.
ux	10	52	19	40	29	2	.	.	.	.	.	.	6	.	19	4	.	.	.
er	48	93	52	.	19	.	8	5	1	3	1	4	.	.	2	52	3	1	.
ay	49	89	55	.	2	3	3	10	18	.	7	.	.	.	.	.	55	.	1
oy	5	16	31	.	.	.	.	.	6	6	6	19	.	.	.	19	12	31	.
aw	5	30	17	.	.	.	7	20	33	7	7	3	.	.	.	.	7	.	17
	851	1637	52.0																

Table C.2: Confusion matrix for BSW (vowel classification)

	Hit	Total	%	iy	ih	ey	eh	ae	aa	ow	ah	ao	uw	uh	ux	er	ay	oy	aw
iy	184	243	76	76	14	6	0	0	.	.	.	.	.	.	3	.	.	.	.
ih	108	203	53	18	53	10	6	1	.	2	2	1	.	.	3	2	.	.	0
ey	53	114	46	18	26	46	9	.	.	.	.	.	.	.	.	.	1	.	.
eh	68	189	36	1	19	4	36	16	6	2	8	1	.	.	.	1	6	.	1
ae	52	105	50	1	7	5	19	50	5	.	3	.	.	.	.	.	9	.	3
aa	69	131	53	.	1	.	2	7	53	3	8	15	.	.	.	.	10	.	2
ow	43	89	48	.	2	.	6	2	2	48	11	24	1	.	.	.	1	.	2
ah	39	135	29	.	10	1	13	5	10	17	29	7	1	.	.	2	4	2	.
ao	66	97	68	.	.	.	.	2	19	3	3	68	.	.	.	1	1	2	1
uw	12	22	55	.	18	.	.	.	.	.	5	.	55	.	9	14	.	.	.
uh	1	29	3	3	48	.	3	3	.	14	3	3	7	3	.	7	3	.	.
ux	13	52	25	40	23	.	.	.	.	4	.	.	8	.	25	.	.	.	.
er	48	93	52	.	17	.	9	5	.	3	2	4	2	.	2	52	2	1	.
ay	47	89	53	.	1	3	6	10	17	.	9	.	.	.	.	.	53	1	.
oy	8	16	50	.	.	.	.	.	.	6	6	31	.	.	.	.	6	50	.
aw	6	30	20	.	.	.	7	20	37	7	3	3	.	.	.	.	3	.	20
	817	1637	49.9																

Table C.3: Confusion matrix for PS (vowel classification)

	Hit	Total	%	iy	ih	ey	eh	ae	aa	ow	ah	ao	uw	uh	ux	er	ay	oy	aw
iy	195	243	80	80	12	5	0	.	.	.	.	.	.	.	2	0	.	.	.
ih	115	203	57	18	57	7	6	1	.	1	3	1	0	.	3	1	.	.	.
ey	56	114	49	19	23	49	7	1	.	.	.	.	.	.	.	.	1	.	.
eh	80	189	42	1	17	4	42	15	6	1	8	1	.	.	.	2	4	.	.
ae	51	105	49	1	9	4	22	49	5	1	2	.	.	.	.	1	7	.	1
aa	77	131	59	.	1	.	3	9	59	2	8	13	.	.	.	.	5	.	1
ow	44	89	49	.	2	.	8	1	3	49	10	21	1	.	.	.	1	.	2
ah	51	135	38	1	10	.	12	4	8	13	38	8	1	.	.	2	4	.	.
ao	64	97	66	.	1	.	2	1	20	5	3	66	.	.	.	1	.	.	1
uw	9	22	41	.	27	.	.	.	.	5	5	5	41	.	5	14	.	.	.
uh	0	29	0	3	52	.	3	.	.	14	3	7	3	.	.	7	7	.	.
ux	13	52	25	42	25	.	.	.	.	.	.	.	4	.	25	4	.	.	.
er	51	93	55	.	18	.	10	5	.	4	.	2	1	.	2	55	1	1	.
ay	52	89	58	.	2	2	1	9	16	.	11	.	.	.	.	.	58	.	.
oy	4	16	25	.	.	.	.	.	.	6	19	25	.	.	.	12	12	25	.
aw	5	30	17	.	.	.	10	23	30	7	7	3	.	.	.	.	3	.	17
	867	1637	53.0																

Table C.4: Confusion matrix for PSW (vowel classification)

	Hit	Total	%	iy	ih	ey	eh	ae	aa	ow	ah	ao	uw	uh	ux	er	ay	oy	aw
iy	185	243	76	76	14	6	0	.	.	.	.	.	.	.	4	.	.	.	.
ih	109	203	54	17	54	8	7	1	.	1	3	1	.	0	3	2	.	.	0
ey	58	114	51	18	21	51	10	.	.	.	.	.	.	.	.	.	1	.	.
eh	69	189	37	1	18	6	37	14	6	1	6	2	.	.	.	2	7	.	1
ae	53	105	50	2	9	3	23	50	6	.	1	.	.	.	.	1	5	.	1
aa	75	131	57	.	1	.	1	8	57	1	7	16	.	.	.	1	5	.	3
ow	52	89	58	.	1	.	6	2	3	58	10	13	1	.	.	1	.	.	3
ah	44	135	33	1	7	1	10	6	10	16	33	7	.	1	.	4	2	2	1
ao	64	97	66	.	1	.	1	2	22	2	2	66	.	.	.	2	.	1	1
uw	12	22	55	.	23	.	.	.	5	.	5	.	55	.	5	9	.	.	.
uh	1	29	3	3	41	3	3	.	.	7	14	3	7	3	.	10	3	.	.
ux	17	52	33	37	23	.	.	.	.	.	.	.	6	.	33	2	.	.	.
er	48	93	52	.	17	.	8	4	1	5	1	4	3	.	2	52	1	1	.
ay	54	89	61	.	1	4	2	9	12	.	7	2	.	.	.	.	61	.	1
oy	10	16	62	.	.	.	.	.	.	6	.	12	.	.	.	6	12	62	.
aw	7	30	23	.	.	.	3	20	27	7	7	.	.	.	.	.	13	.	23
	858	1637	52.4																

Table C.5: Confusion matrix for IC (vowel classification)

	Hit	Total	%	iy	ih	ey	eh	ae	aa	ow	ah	ao	uw	uh	ux	er	ay	oy	aw
iy	183	243	75	75	16	5	.	.	.	.	.	.	.	.	3	.	0	.	.
ih	122	203	60	15	60	7	7	1	.	2	3	1	0	.	2	1	.	.	.
ey	55	114	48	13	26	48	11	.	.	.	.	.	.	.	.	.	2	.	.
eh	76	189	40	1	18	6	40	11	4	1	12	1	.	.	.	2	4	.	1
ae	50	105	48	2	8	4	22	48	6	1	2	.	.	.	.	2	5	.	2
aa	73	131	56	.	1	.	4	5	56	1	9	15	.	.	.	1	6	.	4
ow	43	89	48	.	3	.	7	1	3	48	16	17	1	.	.	.	1	.	2
ah	49	135	36	1	7	1	13	4	10	12	36	8	.	.	.	3	4	1	.
ao	65	97	67	.	.	.	1	4	19	2	5	67	.	.	.	.	1	1	.
uw	12	22	55	.	23	.	5	.	.	.	5	5	55	.	.	9	.	.	.
uh	2	29	7	3	45	3	7	.	3	10	.	7	7	7	.	7	.	.	.
ux	15	52	29	25	33	2	.	.	.	.	.	.	8	.	29	4	.	.	.
er	51	93	55	.	22	.	6	4	.	2	3	1	1	1	2	55	1	1	.
ay	53	89	60	.	.	3	1	4	18	.	10	.	.	.	.	.	60	3	.
oy	11	16	69	.	.	6	.	.	.	6	.	12	.	.	.	6	.	69	.
aw	8	30	27	.	.	3	7	20	23	7	7	.	.	.	.	.	7	.	27
	868	1637	53.0																

Table C.6: Confusion matrix for ICP (vowel classification)

	Hit	Total	%	iy	ih	ey	eh	ae	aa	ow	ah	ao	uw	uh	ux	er	ay	oy	aw
iy	201	243	83	83	9	5	1	.	.	.	.	.	.	.	2	.	0	.	.
ih	127	203	63	14	63	6	7	0	.	2	3	1	0	.	0	2	.	.	.
ey	56	114	49	18	21	49	11	.	.	.	.	.	.	.	.	.	1	.	.
eh	84	189	44	1	16	4	44	14	3	1	9	1	.	.	.	2	4	.	1
ae	53	105	50	2	7	3	22	50	5	.	3	.	.	.	.	2	6	.	1
aa	79	131	60	.	1	.	5	5	60	2	8	11	.	.	.	1	5	.	1
ow	45	89	51	.	4	.	7	.	3	51	13	17	1	.	.	.	1	.	2
ah	50	135	37	.	6	1	13	7	6	13	37	9	1	1	.	3	4	.	.
ao	68	97	70	.	.	.	1	2	15	1	6	70	.	.	.	2	1	.	1
uw	8	22	36	.	18	.	9	.	.	9	9	5	36	5	.	9	.	.	.
uh	2	29	7	3	52	.	3	3	.	7	10	3	7	7	.	3	.	.	.
ux	12	52	23	38	29	.	.	.	.	2	.	.	6	.	23	2	.	.	.
er	55	93	59	.	15	.	9	4	1	2	3	2	1	.	1	59	1	1	.
ay	56	89	63	.	.	3	2	7	11	.	11	2	.	.	.	.	63	.	.
oy	6	16	38	.	.	6	.	.	.	6	12	25	.	.	.	.	12	38	.
aw	5	30	17	.	.	3	10	17	27	7	17	.	.	.	.	.	3	.	17
	907	1637	55.4																

Table C.7: Confusion matrix for CAIC (vowel classification)

	Hit	Total	%	iy	ih	ey	eh	ae	aa	ow	ah	ao	uw	uh	ux	er	ay	oy	aw
iy	185	243	76	76	15	6	0	.	.	.	.	.	.	.	2	0	.	.	.
ih	126	203	62	14	62	7	5	1	.	1	4	1	0	.	2	1	0	.	.
ey	55	114	48	15	27	48	7	1	.	.	.	.	.	.	.	.	2	.	.
eh	73	189	39	1	16	5	39	14	5	1	9	1	.	.	.	3	6	.	1
ae	51	105	49	1	10	6	16	49	7	.	3	.	.	.	.	.	8	.	2
aa	70	131	53	.	.	.	3	8	53	3	9	16	.	.	.	1	5	.	2
ow	48	89	54	.	1	.	3	2	3	54	13	16	1	.	.	1	1	.	3
ah	43	135	32	1	7	1	14	6	10	14	32	7	.	.	.	3	4	1	1
ao	66	97	68	.	.	.	2	2	16	4	4	68	.	.	.	1	.	1	1
uw	12	22	55	.	23	.	.	.	.	.	5	.	55	.	5	14	.	.	.
uh	1	29	3	.	45	3	3	.	3	14	3	3	7	3	.	7	3	3	.
ux	16	52	31	31	29	2	.	.	.	2	.	.	4	.	31	2	.	.	.
er	50	93	54	.	18	.	9	4	1	4	1	3	1	.	1	54	1	2	.
ay	50	89	56	.	1	3	2	6	21	1	8	.	.	.	.	.	56	1	.
oy	10	16	62	.	.	.	.	.	.	6	.	19	.	.	.	6	6	62	.
aw	9	30	30	.	.	.	10	10	27	7	7	3	.	.	.	.	7	.	30
	865	1637	52.8																

Table C.8: Confusion matrix for CAICP (vowel classification)

	Hit	Total	%	iy	ih	ey	eh	ae	aa	ow	ah	ao	uw	uh	ux	er	ay	oy	aw
iy	193	243	79	79	12	5	0	.	.	.	.	.	.	.	4	0	.	.	.
ih	123	203	61	18	61	7	5	1	.	1	3	1	.	.	2	1	.	.	.
ey	51	114	45	16	28	45	10	.	.	.	.	.	.	.	.	.	2	.	.
eh	77	189	41	1	17	5	41	15	5	1	7	1	.	.	.	2	5	.	1
ae	52	105	50	2	8	3	19	50	6	.	2	1	.	.	.	1	9	.	1
aa	79	131	60	.	1	.	2	8	60	4	5	14	.	.	.	1	6	.	.
ow	48	89	54	.	2	.	4	2	6	54	9	17	1	.	.	1	1	.	2
ah	53	135	39	1	7	1	13	5	8	11	39	5	1	1	.	4	4	1	1
ao	69	97	71	.	1	.	.	2	16	3	4	71	.	.	.	1	.	1	.
uw	10	22	45	.	23	.	5	.	.	.	9	.	45	.	5	14	.	.	.
uh	1	29	3	.	55	3	10	.	.	10	3	3	3	3	.	7	.	.	.
ux	16	52	31	38	25	.	.	.	.	.	.	.	2	.	31	4	.	.	.
er	54	93	58	.	16	1	9	4	.	1	2	4	.	.	2	58	1	1	.
ay	53	89	60	.	1	3	8	4	16	.	8	.	.	.	.	.	60	.	.
oy	9	16	56	.	.	6	.	.	.	6	.	31	.	.	.	.	.	56	.
aw	6	30	20	.	.	.	7	27	23	7	7	.	.	.	.	.	10	.	20
	894	1637	54.6																

C.2 Consonant Classification

Please note that each confusion matrix is split into two tables due to the limited width of the page.

Table C.9: Confusion matrix (1 of 2) for ICP (consonant classification)

	Hit	Total	%	p	b	t	d	k	g	f	v	th	dh	s	z
p	63	137	46	46	7	16	1	18	.	4	.	.	1	5	.
b	30	132	23	17	23	6	10	.	.	.	2	1	8	.	2
t	132	187	71	3	.	71	1	9	.	3	.	.	1	12	1
d	29	113	26	5	3	28	26	4	.	2	1	.	6	4	3
k	99	165	60	5	1	22	2	60	.	1	.	.	1	8	.
g	1	66	2	5	8	8	8	53	2	.	.	.	.	.	.
f	64	131	49	3	.	5	.	1	.	49	.	.	2	36	4
v	22	93	24	.	4	.	2	.	.	1	24	1	2	.	15
th	5	38	13	5	3	16	3	.	.	32	.	13	5	11	11
dh	27	128	21	2	5	13	5	.	.	4	5	.	21	1	3
s	294	321	92	.	.	2	.	1	.	2	.	.	.	92	3
z	94	181	52	.	.	.	.	.	.	3	2	.	.	38	52
sh	43	67	64	.	.	4	.	1	.	.	.	.	.	21	.
zh	0	10	0	.	.	.	.	.	.	.	.	.	.	.	40
hh	7	76	9	9	.	18	.	24	.	3	.	1	3	1	1
ch	11	40	28	.	.	22	.	.	.	3	.	.	.	20	.
jh	14	42	33	.	.	14	.	5	.	.	.	.	.	12	5
m	63	200	31	1	4	.	1	.	1	.	1	.	1	.	.
n	270	319	85	.	1	.	1	.	.	.	0	.	1	.	1
ng	0	51	0	.	.	.	.	.	.	.	.	.	.	.	.
w	57	144	40	1	3	.	.	.	.	.	.	.	.	.	.
l	165	234	71	.	2	.	.	.	.	.	.	.	1	.	.
r	220	270	81	.	1	.	0	.	.	.	0	.	0	.	.
y	34	50	68	.	.	.	6	.	.	.	.	.	2	.	.
	1744	3195	54.6												

Table C.10: Confusion matrix (2 of 2) for ICP (consonant classification)

	Hit	Total	%	sh	zh	hh	ch	jh	m	n	ng	w	l	r	y
p	63	137	46	.	.	.	.	.	.	2	.	.	.	1	.
b	30	132	23	.	.	.	1	.	10	10	.	3	4	3	.
t	132	187	71	.	.	1	1	.	.	.	.	.	.	.	.
d	29	113	26	.	.	2	.	1	.	10	.	1	.	3	2
k	99	165	60	.	.	1	1	.	.	.	.	.	.	.	.
g	1	66	2	.	.	.	.	.	3	3	.	.	.	6	6
f	64	131	49	.	.	1	.	.	.	.	.	.	.	.	.
v	22	93	24	.	.	.	.	.	10	34	.	1	3	2	.
th	5	38	13	.	.	.	.	.	.	3	.	.	.	.	.
dh	27	128	21	.	.	1	.	.	9	23	.	1	3	4	.
s	294	321	92	2	.	.	.	.	.	.	.	.	.	.	.
z	94	181	52	2	.	.	.	1	.	3	.	.	.	.	.
sh	43	67	64	64	.	.	7	1	.	.	.	.	.	.	.
zh	0	10	0	40	.	.	.	10	.	.	.	.	.	.	10
hh	7	76	9	.	.	9	.	.	5	20	.	.	3	1	1
ch	11	40	28	22	.	.	28	5	.	.	.	.	.	.	.
jh	14	42	33	17	.	.	10	33	.	2	.	.	.	.	2
m	63	200	31	.	.	.	.	.	32	55	.	2	3	3	.
n	270	319	85	.	.	.	.	.	9	85	.	1	1	1	0
ng	0	51	0	.	.	2	.	.	4	88	.	2	.	.	4
w	57	144	40	.	.	1	.	.	.	5	.	40	42	10	.
l	165	234	71	.	.	.	.	.	3	3	.	9	71	12	.
r	220	270	81	.	.	.	.	.	2	2	.	4	7	81	1
y	34	50	68	.	.	4	.	.	.	10	.	2	2	6	68
	1744	3195	54.6												

C.3 Phoneme Classification

Please note that each confusion matrix is split into three tables due to the limited width of the page. The phonemes listed here are the 39 Kai-Fu Lee phoneme classes and the numbering scheme is shown in table 2.3.

Table C.11: Confusion matrix (1 of 3) for ICP (phoneme classification)

	Hit	Total	%	1	2	3	4	5	6	7	8	9	10	11	12	13
1	141	245	58	58	37	0	.	.	.	.	.	2	.	.	.	.
2	423	580	73	5	73	1	1	5	.	.	0	1	.	.	.	.
3	34	189	18	1	37	18	12	12	.	.	10	3	3	.	.	.
4	54	105	51	.	16	15	51	2	.	.	5	1	7	.	.	.
5	105	307	34	.	22	3	3	34	.	.	9	0	1	.	.	2
6	1	74	1	9	65	.	.	5	1	.	.	.	.	.	.	.
7	0	29	0	.	66	7	.	10	.	.	.	.	.	.	.	.
8	140	228	61	.	2	0	4	8	.	.	61	.	3	.	.	.
9	31	114	27	7	60	3	3	.	.	.	.	27	1	.	.	.
10	38	89	43	.	9	3	11	10	.	.	21	1	43	.	.	.
11	0	16	0	.	6	.	.	19	.	.	19	.	6	.	.	.
12	0	30	0	.	.	10	27	17	.	.	37	.	3	.	.	7
13	6	89	7	.	6	2	3	26	.	.	15	.	.	.	.	7
14	196	291	67	0	4	.	.	8	.	.	7	.	0	.	.	0
15	111	270	41	0	10	1	3	9	.	.	10	1	3	.	.	.
16	2	50	4	58	30	.	.	2	.	.	.	.	.	.	.	.
17	50	144	35	.	4	.	.	1	.	.	7	.	.	.	.	.
18	75	234	32	.	32	3	2	2	.	.	1	.	1	.	.	.
19	42	204	21	.	7	.	0	3	.	.	0	.	.	.	.	.
20	262	385	68	1	14	.	.	1	.	.	0	.	.	.	.	.
21	0	52	0	2	17	.	.	4	.	.	.	.	.	.	.	.
22	8	40	20	.	.	.	.	.	.	.	.	.	.	.	.	.
23	6	42	14	.	.	.	.	.	.	.	.	.	.	.	.	.
24	6	128	5	.	7	.	.	3	.	.	1	.	.	.	.	.
25	26	132	20	.	6	.	.	4	.	.	1	.	.	.	.	.
26	22	113	19	2	15	.	.	.	.	.	.	.	.	.	.	.
27	1	90	1	10	44	1	.	4	.	.	.	2	.	.	.	.
28	0	66	0	3	3	.	.	.	.	.	.	.	.	.	.	.
29	59	136	43	.	.	.	.	.	.	.	.	.	.	.	.	.
30	123	187	66	.	.	.	.	.	.	.	.	.	.	.	.	.
31	89	165	54	.	.	.	.	.	.	.	.	.	.	.	.	.
32	61	181	34	.	2	.	.	.	.	.	.	.	.	.	.	.
33	49	77	64	1	3	.	.	.	.	.	.	.	.	.	.	.
34	7	93	8	.	9	.	.	3	.	.	.	.	.	.	.	.
35	43	131	33	.	.	.	.	.	.	.	.	.	.	.	.	.
36	0	38	0	.	.	.	.	.	.	.	.	.	.	.	.	.
37	287	321	89	.	.	.	.	.	.	.	.	.	.	.	.	.
38	1	76	1	4	4	.	3	3	.	.	.	.	.	.	.	.
39	970	1076	90	0	0	.	.	0	.	.	.	.	.	.	.	.
	3469	6817	50.9													

Table C.12: Confusion matrix (2 of 3) for ICP (phoneme classification)

	Hit	Total	%	14	15	16	17	18	19	20	21	22	23	24	25	26
1	141	245	58	.	.	.	.	.	0	1	.	.	.	.	.	0
2	423	580	73	1	1	.	.	1	0	8	.	.	.	.	0	0
3	34	189	18	.	4	.	.	2	.	.	.	.	.	.	.	.
4	54	105	51	.	2	.	.	1	.	.	.	.	.	.	.	.
5	105	307	34	13	2	.	.	2	1	7	.	.	.	.	0	.
6	1	74	1	5	1	.	.	4	1	7	.	.	.	.	.	.
7	0	29	0	10	.	.	.	7	.	.	.	.	.	.	.	.
8	140	228	61	16	2	.	1	0	.	0	.	.	.	.	.	.
9	31	114	27	.	.	.	.	.	.	.	.	.	.	.	.	.
10	38	89	43	.	1	.	.	.	.	.	.	.	.	.	.	.
11	0	16	0	38	12	.	.	.	.	.	.	.	.	.	.	.
12	0	30	0	.	.	.	.	.	.	.	.	.	.	.	.	.
13	6	89	7	39	.	.	.	2	.	.	.	.	.	.	.	.
14	196	291	67	67	1	.	5	1	1	3	.	.	.	.	1	.
15	111	270	41	3	41	.	2	14	1	2	.	.	.	.	1	.
16	2	50	4	.	.	4	.	.	.	4	.	.	.	.	.	.
17	50	144	35	40	4	.	35	1	1	4	.	.	.	.	3	.
18	75	234	32	3	18	.	.	32	.	4	.	.	.	.	.	.
19	42	204	21	3	0	.	.	.	21	52	.	.	.	1	3	.
20	262	385	68	1	0	0	0	1	5	68	.	.	.	1	1	0
21	0	52	0	2	.	.	.	.	.	69	.	.	.	.	.	.
22	8	40	20	.	.	.	.	.	.	.	.	20	.	.	.	.
23	6	42	14	.	.	2	.	.	.	.	.	12	14	.	.	.
24	6	128	5	2	.	.	.	.	8	27	.	.	.	5	2	8
25	26	132	20	3	2	.	1	.	5	14	.	.	.	5	20	9
26	22	113	19	.	.	.	1	1	1	8	.	.	.	4	3	19
27	1	90	1	1	1	.	.	2	1	18	.	.	.	.	.	2
28	0	66	0	.	3	.	.	3	2	5	.	.	.	2	5	6
29	59	136	43	.	.	.	.	.	.	2	.	.	.	1	6	3
30	123	187	66	.	.	.	.	.	.	.	.	1	.	.	.	1
31	89	165	54	.	.	.	.	.	.	.	.	1	1	1	1	1
32	61	181	34	.	.	.	.	.	1	2	.	.	.	.	.	.
33	49	77	64	.	.	.	.	.	.	.	.	1	3	.	.	.
34	7	93	8	1	1	.	2	1	5	30	.	.	.	1	1	2
35	43	131	33	.	.	.	.	.	.	.	.	.	.	.	.	1
36	0	38	0	.	.	.	.	.	.	.	.	.	.	.	3	.
37	287	321	89	.	.	.	.	.	.	.	.	.	.	.	.	.
38	1	76	1	3	.	.	.	1	1	16	.	.	.	3	.	.
39	970	1076	90	0	.	.	0	0	0	4	.	.	.	0	0	.
	3469	**6817**	**50.9**													

Table C.13: Confusion matrix (3 of 3) for ICP (phoneme classification)

	Hit	Total	%	27	28	29	30	31	32	33	34	35	36	37	38	39
1	141	245	58	.	.	.	.	.	.	.	.	.	.	.	.	1
2	423	580	73	.	.	.	.	.	0	.	.	.	.	.	0	1
3	34	189	18	.	.	.	.	.	.	.	.	.	.	.	.	.
4	54	105	51	.	.	.	.	.	.	.	.	.	.	.	.	.
5	105	307	34	.	.	.	.	.	.	.	0	.	.	.	.	0
6	1	74	1	.	.	.	.	.	.	.	.	.	.	.	.	.
7	0	29	0	.	.	.	.	.	.	.	.	.	.	.	.	.
8	140	228	61	.	.	.	.	.	.	.	.	.	.	.	.	.
9	31	114	27	.	.	.	.	.	.	.	.	.	.	.	.	.
10	38	89	43	.	.	.	.	.	.	.	.	.	.	.	.	.
11	0	16	0	.	.	.	.	.	.	.	.	.	.	.	.	.
12	0	30	0	.	.	.	.	.	.	.	.	.	.	.	.	.
13	6	89	7	.	.	.	.	.	.	.	.	.	.	.	.	.
14	196	291	67	.	.	.	.	.	0	.	.	.	.	.	.	.
15	111	270	41	.	.	.	.	.	.	.	.	.	.	.	.	.
16	2	50	4	.	.	.	.	.	2	.	.	.	.	.	.	.
17	50	144	35	.	.	1	.	.	.	.	.	.	.	.	.	.
18	75	234	32	.	.	.	.	.	.	.	.	.	.	.	.	1
19	42	204	21	.	.	0	.	.	.	.	.	.	.	.	.	8
20	262	385	68	.	.	.	.	.	.	.	.	.	.	.	.	7
21	0	52	0	.	.	.	.	.	.	.	.	.	.	.	.	6
22	8	40	20	.	.	.	22	.	.	20	.	.	.	35	.	3
23	6	42	14	.	.	.	21	5	2	24	.	.	.	14	.	5
24	6	128	5	.	.	3	12	.	1	.	2	5	1	.	.	16
25	26	132	20	.	.	19	4	2	1	1	2	1	.	.	.	2
26	22	113	19	.	.	6	22	5	1	.	1	1	.	5	.	5
27	1	90	1	1	.	.	.	.	2	.	.	.	.	.	2	7
28	0	66	0	.	.	9	12	42	.	.	2	.	.	.	.	5
29	59	136	43	.	.	43	18	12	.	.	.	4	.	6	1	3
30	123	187	66	.	.	3	66	6	.	1	.	5	.	16	.	3
31	89	165	54	.	.	5	21	54	.	.	.	2	.	8	1	5
32	61	181	34	.	.	.	1	.	34	2	1	1	.	48	.	10
33	49	77	64	.	.	.	3	.	1	64	.	.	.	25	.	.
34	7	93	8	1	.	.	.	.	5	.	8	.	.	.	.	29
35	43	131	33	.	.	3	7	1	4	.	.	33	.	38	.	14
36	0	38	0	.	.	3	18	.	5	.	.	16	.	13	.	42
37	287	321	89	.	.	.	2	0	2	2	.	2	.	89	.	2
38	1	76	1	.	.	7	16	28	.	.	.	3	.	1	1	8
39	970	1076	90	0	.	1	.	1	0	0	0	0	.	2	0	90
	3469	6817	50.9													

Appendix D

Corpus

This appendix lists the TIMIT speakers for the various NIST data sets. The 5 "*sx*" and 3 "*si*" sentences spoken by each speaker are used.

D.1 NIST TRAIN Speakers

faem0	fajw0	falk0	falr0	fapb0	fbas0	fbcg1	fbch0
fbjl0	fblv0	fbmh0	fbmj0	fcag0	fcaj0	fcdr1	fceg0
fcjf0	fcjs0	fcke0	fclt0	fcmg0	fcmm0	fcrz0	fcyl0
fdas1	fdaw0	fdfb0	fdjh0	fdkn0	fdml0	fdmy0	fdnc0
fdtd0	fdxw0	feac0	fear0	fecd0	feeh0	feme0	fetb0
fexm0	fgcs0	fgdp0	fgmb0	fgrw0	fhlm0	fhxs0	fjdm2
fjen0	fjhk0	fjkl0	fjlg0	fjlr0	fjrb0	fjrp1	fjsk0
fjsp0	fjwb1	fjxm0	fjxp0	fkaa0	fkde0	fkdw0	fkfb0
fkkh0	fklc0	fklc1	fklh0	fksr0	flac0	flag0	fleh0
flet0	flhd0	flja0	fljd0	fljg0	flkm0	flma0	flmc0
flmk0	flod0	fltm0	fmah1	fmbg0	fmem0	fmjb0	fmjf0
fmju0	fmkc0	fmkf0	fmmh0	fmpg0	fnkl0	fntb0	fpab1
fpac0	fpad0	fpaf0	fpaz0	fpjf0	fpls0	fpmy0	freh0
frjb0	frll0	fsag0	fsah0	fsak0	fsbk0	fscn0	fsdc0
fsdj0	fsgf0	fsjg0	fsjk1	fsjs0	fsjw0	fskc0	fskl0
fskp0	fsls0	fsma0	fsmm0	fsms1	fspm0	fsrh0	fssb0
ftaj0	ftbr0	ftbw0	ftlg0	ftmg0	fvfb0	fvkb0	fvmh0
mabc0	madc0	madd0	maeb0	maeo0	mafm0	majp0	makb0
makr0	mapv0	marc0	marw0	mbar0	mbbr0	mbcg0	mbef0
mbgt0	mbjv0	mbma0	mbma1	mbml0	mbom0	mbsb0	mbth0
mbwp0	mcae0	mcal0	mcdc0	mcdd0	mcdr0	mcef0	mcew0

mchl0	mclk0	mclm0	mcpm0	mcre0	mcss0	mcth0	mctm0
mcxm0	mdac0	mdas0	mdbb1	mdbp0	mdcd0	mdcm0	mddc0
mded0	mdef0	mdem0	mdhl0	mdhs0	mdjm0	mdks0	mdlb0
mdlc0	mdlc1	mdlc2	mdlh0	mdlm0	mdlr0	mdlr1	mdma0
mdmt0	mdns0	mdpb0	mdpk0	mdps0	mdrd0	mdsj0	mdss0
mdss1	mdtb0	mdwd0	mdwh0	mdwm0	meal0	medr0	mefg0
megj0	mejl0	mejs0	mesg0	mesj0	mewm0	mfer0	mfmc0
mfrm0	mfwk0	mfxs0	mfxv0	mgaf0	mgag0	mgak0	mgar0
mgaw0	mges0	mgjc0	mgrl0	mgrp0	mgsh0	mgsl0	mgxp0
mhbs0	mhit0	mhjb0	mhmg0	mhmr0	mhrm0	mhxl0	milb0
mjac0	mjae0	mjai0	mjbg0	mjda0	mjdc0	mjde0	mjdg0
mjdm0	mjeb0	mjeb1	mjee0	mjfh0	mjfr0	mjhi0	mjjb0
mjjj0	mjjm0	mjkr0	mjlb0	mjlg1	mjls0	mjma0	mjmd0
mjmm0	mjpg0	mjpm0	mjpm1	mjra0	mjrg0	mjrh0	mjrh1
mjrk0	mjrp0	mjsr0	mjwg0	mjws0	mjwt0	mjxa0	mjxl0
mkag0	mkah0	mkaj0	mkam0	mkdb0	mkdd0	mkdt0	mkes0
mkjo0	mkln0	mklr0	mkls0	mkls1	mklw0	mkrg0	mkxl0
mlbc0	mlel0	mljc0	mljh0	mlns0	mlsh0	mmaa0	mmab1
mmag0	mmam0	mmar0	mmbs0	mmcc0	mmdb0	mmdg0	mmdm0
mmdm1	mmds0	mmea0	mmeb0	mmgc0	mmgg0	mmgk0	mmjb1
mmlm0	mmpm0	mmrp0	mmsm0	mmvp0	mmwb0	mmws0	mmws1
mmxs0	mnet0	mntw0	mpar0	mpeb0	mpfu0	mpgh0	mpgr0
mpgr1	mpmb0	mppc0	mprb0	mprd0	mprk0	mprt0	mpsw0
mrab0	mrab1	mrai0	mram0	mrav0	mrbc0	mrcg0	mrcw0
mrdd0	mrdm0	mrds0	mree0	mreh1	mrem0	mrew1	mrfk0
mrfl0	mrgm0	mrgs0	mrhl0	mrjb1	mrjh0	mrjm0	mrjm1
mrjt0	mrkm0	mrld0	mrlj0	mrlj1	mrlk0	mrlr0	mrmb0
mrmg0	mrmh0	mrml0	mrms0	mrpc1	mrre0	mrso0	mrsp0
mrtc0	mrtj0	mrvg0	mrwa0	mrws0	mrxb0	msah1	msas0
msat0	msat1	msdb0	msdh0	msds0	msem1	mses0	msfh0
msfv0	msjk0	msmc0	msmr0	msms0	msrg0	msrr0	mstf0
msvs0	mtab0	mtas0	mtat0	mtat1	mtbc0	mtcs0	mtdb0
mtdp0	mter0	mtjg0	mtjm0	mtjs0	mtju0	mtkd0	mtkp0
mtlb0	mtlc0	mtml0	mtmn0	mtmt0	mtpf0	mtpg0	mtpp0
mtpr0	mtqc0	mtrc0	mtrr0	mtrt0	mtwh1	mtxs0	mvjh0
mvlo0	mvrw0	mwac0	mwad0	mwar0	mwch0	mwdk0	mwem0
mwgr0	mwre0	mwrp0	mwsb0	mwsh0	mzmb0		

D.2 NIST DEV Speakers

fadg0	faks0	fcal1	fcmh0	fdac1	fdms0	fdrw0	fedw0
fgjd0	fjem0	fjmg0	fjsj0	fkms0	fmah0	fmml0	fnmr0

frew0	fsem0	majc0	mbdg0	mbns0	mbwm0	mcsh0	mdlf0
mdls0	mdvc0	mers0	mgjf0	mglb0	mgwt0	mjar0	mjfc0
mjsw0	mmdb1	mmdm2	mmjr0	mmwh0	mpdf0	mrcs0	mreb0
mrjm4	mrjr0	mroa0	mrtk0	mrws1	mtaa0	mtdt0	mteb0
mthc0	mwjg0						

D.3 NIST CORE TEST Speakers

fdhc0	felc0	fjlm0	fmgd0	fmld0	fnlp0	fpas0	fpkt0
mbpm0	mcmj0	mdab0	mgrt0	mjdh0	mjln0	mjmp0	mklt0
mlll0	mlnt0	mnjm0	mpam0	mtas1	mtls0	mwbt0	mwew0

Bibliography

[1] R. Carlson and J. Glass. Vowel classification based on analysis-by-synthesis. In *Proc. Int. Conf. Spoken Language Processing*, pages 575–578, Banff, Canada, October 1992.

[2] R. T. Chun. A hierarchical feature representation for phonetic classification. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, March 1996.

[3] R. A. Cole and Y. K. Methusamy. Perceptual studies on vowels excised from continuous speech. In *Proc. Int. Conf. Spoken Language Processing*, pages 1091–1094, Banff, Canada, October 1992.

[4] I. Daubechies. *Ten Lectures on Wavelets.* Society for Industrial and Applied Mathematics, 1992.

[5] I. Daubechies, editor. *Different Perspectives on Wavelets.* American Mathematical Society, 1993.

[6] R. F. Favero and R. W. King. Wavelet parameterization for speech recognition. *ICSPAT*, pages 1447–1450, 1993.

[7] W. D. Goldenthal. *Statistical Trajectory Models for Phonetic Recognition.* PhD thesis, Massachusetts Institute of Technology, Department of Aeronautics and Astronautics, September 1994.

[8] A. K. Halberstadt and J. R. Glass. Heterogeneous acoustic measurements for phonetic classification. In *Proceedings of Eurospeech97*, Rhodes, Greece, September 1997. To appear in the conference in September.

[9] J. A. Hartigan, editor. *Clustering Algorithms.* John Wiley and Sons, 1975.

[10] H. Hermansky. Perceptual linear predictive (PLP) analysis of speech. *J. Acoust. Soc. Am.*, 87(4):1738–1752, 1990.

[11] S. Kadambe and G. F. Boudreaux-Bartels. Application of the wavelet transform for pitch detection of speech signals. *IEEE Trans. Info. Th.*, 38(2):917–924, 1992.

[12] K. F. Lee and H. W. Hon. Speaker-independent phone recognition using hidden Markov models. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(11):1641–1648, November 1989.

[13] H. Leung. *The Use of Artificial Neural Networks for Phonetic Recognition.* PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1989.

[14] H. Leung, B. Chigier, and J. Glass. A comparative study of signal representations and classification techniques for speech recognition. In *ICASSP*, pages 680–683, Minneapolis, April 1993.

[15] J. Makhoul and L. Cosell. LPCW: An LPC vocoder with linear predictive spectral warping. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, pages 466–469, 1976.

[16] J. D. Markel and Jr. A. H. Gray. *Linear Prediction of Speech.* Springer-Verlag, 1976.

[17] R. K. Martinet. The wavelet transform for analysis, synthesis, and processing of speech and music sounds. *Computer Music J.*, 12(4):11–20, 1988.

[18] R. K. Martinet, J. Morlet, and A. Grossmann. Analysis of sound patterns through wavelet transforms. *Int. J. Pattern Recognition and Artificial Intelligence*, 1(2):273–302, 1987.

[19] H. Meng. The use of distinctive features for automatic speech recognition. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, September 1991.

[20] D. J. Pepper and M. A. Clements. Phonemic recognition using a large hidden Markov model. *IEEE Transactions on Signal Processing*, 40(6):1590–1595, June 1992.

[21] O. Rioul and M. Vetterli. Wavelets and signal processing. *IEEE SP Magazine*, pages 14–38, October 1991.

[22] D. W. Robinson and R. S. Dadson. A redetermination of the equal-loudness relations for pure tones. *Br. J. Appl. Phys. 7*, pages 166–181, 1956.

[23] M. R. Schroeder. Recognition of complex acoustic signals. In T. H. Bullock, editor, *Life Sciences Research Report 5*, page 324. Abakon Verlag, Berlin, 1977.

[24] S. S. Stevens. On the psychophysical law. *Psychol. Rev. 64*, pages 153–181, 1957.

[25] B. T. Tan, R. Lang, H. Schroder, A. Spray, and P. Dermody. Applying wavelet analysis to speech segmentation and classification. In Harold H. Szu, editor, *Wavelet Applications*, pages 750–761. SPIE, Vol. 2242, 1994.

[26] S. Watanabe. *Pattern Recognition: Human and Mechanical.* John Wiley and Sons, 1985.

[27] M. V. Wickerhauser. Acoustic signal compression with wave packets. Technical report, Yale University, August 1989.

[28] S. A. Zahorian, P. Silsbee, and X. Wang. Phone classification with segmental features and a binary-pair partitioned neural network classifier. In *ICASSP*, pages 1011–1014, Munich, Germany, April 1997.

[29] V. W. Zue, J. R. Glass, D. Goodine, H. Leung, M. Phillips, J. Polifroni, and S. Seneff. Speech database development at MIT: TIMIT and beyond. *Speech Communication*, 9:351–356, 1990.