COMPARISON OF ALTERNATIVE TRAINING ALGORITHMS FOR HIDDEN MARKOV MODELS

BY

SHAILESH S. INGALE

B.S., University of Illinois at Urbana-Champaign, 2002

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2004

Urbana, Illinois

ABSTRACT

The application of categorizing time-varying data by using hidden Markov models (HMMs) involves three basic procedures: evaluation, decoding, and learning. In evaluation, the goal is to find the probability of a certain observation sequence being produced by a particular HMM. In decoding, the goal is to find the optimal state sequence corresponding to an HMM and an observation sequence. In training, the HMM learns from observation sequences. These procedures are considered computationally expensive; however, for evaluation and decoding, there are already existing fast procedures. Training is performed by using the Baum-Welch algorithm, an iterative procedure to find model parameters in a maximum likelihood sense.

In this thesis, we investigate the training procedure. We study the Baum-Welch algorithm and compare its performance with two alternative training methods proposed by Tong Zhang and C.-C. Kuo. One of the methods is an ad-hoc noniterative algorithm that uses simple statistical measurements to define HMM parameters. The other is an iterative method that incorporates Viterbi decoding to further refine the model. These training methods are faster than the Baum-Welch algorithm.

In this thesis, we compare these three methods in performance and computational complexity in different noise conditions for the application of isolated digit recognition. In training these models, we alter the number of states within the model, and the type of initial model used for the iterative methods. In our tests, we found that alternative methods perform almost as well as the Baum-Welch algorithm in training the models.

To the unicorns.

ACKNOWLEDGEMENTS

This thesis represents the culmination of my six years at the University of Illinois, during which I have earned a bachelor's degree, and now a master's degree. Of course, there have been many people who have been influential to me in my long journey into adulthood at the University of Illinois.

I would first like to thank my advisor, Prof. Dr. Douglas Jones, and Dr. Nail Çadallı of Phonak Hearing Systems. During this entire process, they have spent countless hours helping me debug code, rewording my thesis, and providing context for the work I was trying to do. Without their assistance, this thesis would still be a work in progress. I would also like to thank Professor Dr. Dilip Sarwate, who has not only been my professor for two different classes, but also gave me my first research opportunity as a graduate student. I am grateful for the opportunities he provided me.

Next, I would never be in a position to write this thesis if it were not for the guidance I received from my parents, Satish and Shobha Ingale. For 18 years of my life, they provided me with the discipline and motivation to succeed in life, and my only hope is that for the six years since then, I've made them proud of their efforts.

Of course when the discipline and love they offered was too much to handle, I needed someone else to go, and for that I offer thanks to my brother-in-law Jim and sister Suneela Thompson. My sister, the original electrical engineer in our family, is often the lone voice of reason in the many moments of insanity we face on a daily basis.

Next I would like to thank my fellow graduate student and fraternity brother Matt Sztelle, who heavily influenced my decision to pursue graduate study. Being a graduate student, Matt

has always been understanding and patient as I described my latest adventures in academic politics.

It is people like this who have influenced, inspired, and contributed not only to my success in completing this thesis, but my success in life up to now. Unfortunately, this is far from the full list of people who deserve credit.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

BW Baum-Welch

BW-F Baum-Welch with flat initialization

BW-ZK Baum-Welch with ZKNI initialization

CR Confidence ratio

EM Expectation maximization

ER Error ratio

FAR False-alarm rate

HMM Hidden Markov model

HR Hit rate

MFCC Mel-frequency cepstral coefficients

OMC Observable Markov chain

PDF Probability distribution function

SNR Signal-to-noise ratio

ZKI Zhang and Kuo's iterative HMM training method

ZKNI Zhang and Kuo's noniterative HMM training method

CHAPTER 1

INTRODUCTION

The hidden Markov model (HMM) is a statistical tool used to parametrize observation data of a discrete-time process. Since the HMM can model time evolution, it has been used widely in pattern segmentation and classification. It is the major statistical tool for speech classification [1, 2] among others including neural networks [3, 4], classification and regression trees [5], self organizing trees [6], and logistic regression [7].

At each discrete-time observation instant, the HMM system is said to be in one of N states. At each observation instant, the HMM produces an observation associated with the current state. The observation at a time is a probabilistic function of the state of the system. There are three basic procedures associated with HMMs: evaluation, decoding, and training [2, 8]. Evaluation involves calculating the probability of a particular observation sequence given the parameters of the HMM. Decoding involves calculating the most probable state sequence given a particular observation sequence. In training, one adjusts the model to make a given set of training observation sequences most likely. This is the learning stage of the HMM.

In this thesis, we are particularly interested in the training procedure. Usually, HMMs are trained using the Baum-Welch (BW) algorithm, an iterative expectation-maximization (EM) algorithm for HMMs. The BW algorithm is considered to be computationally expensive compared to suboptimal methods [9]. Computational expense is not an issue for applications such as speech recognition, where training is performed offline and speech is evaluated in real

time. For other applications, faster training may become necessary for realtime training. One of these potential applications is classification of auditory scenes to perform noise reduction in hearing aids [10]. By classifying the feature vectors to the correct auditory scene where they were created, one can identify the auditory scene (such as a quiet room, a noisy room, or a concert) experienced by the hearing aid user and use the most appropriate noise-reduction technique.

This document studies alternate training methods of HMMs and compares their performance. This thesis is organized as follows: In Chapter 2, we review the theory of HMMs. We also review the technical details of evaluation, decoding, and training. In Chapter 3, we discuss alternative training methods for HMMs and implementational modifications to the three common procedures. In Chapter 4, we compare the performance of the training methods with respect to the application of isolated digit recognition. In Chapter 5, we provide conclusions of the work contained in this thesis and also propose further work.

CHAPTER 2

REVIEW OF HIDDEN MARKOV MODELS

2.1 Observable Markov Chains

A sequence of discrete-time realizations of a random variable X form a first-order Markov chain if they satisfy

$$P(X_t|X_{t-1}, X_{t-2}, \ ... \ , X_1) \ = \ P(X_t|X_{t-1}). \tag{2.1}$$

This means the probability of a system being in any state at time t is solely dependent on its state at time $t-1$.

The system can be thought of as being in one of N states. At each time interval, the state may change to any of the other $N-1$ states, or remain in its current state. The underlying assumption is that the system is a first-order Markov chain satisfying Equation (2.1). In an observable Markov chain (OMC), the state is directly observable. The observation is a deterministic function of the state.

We define the *transition probability*

$$a_{ij} \ = \ P(s_t = j|s_{t-1} = i), \quad 1 \leq i \leq N, \ 1 \leq j \leq N, \tag{2.2}$$

as the probability of going from state i to j. The transition probabilities are kept in an

$N \times N$ matrix $\mathbf{A}$ where each row represents one origin state and each column represents a destination state:

$$\mathbf{A} \;=\; \{a_{ij}\}_{i,j=1}^{N} \tag{2.3}$$

Since the system must either switch to a new state or stay in its current state at each time instant, each row of $\mathbf{A}$ must add to 1:

$$\sum_{j=1}^{N} a_{ij} \;=\; 1, \quad 1 \leq i \leq N. \tag{2.4}$$

Another parameter of a Markov chain is the *initial state distribution* $\boldsymbol{\pi}$. The vector $\boldsymbol{\pi}$ is of length N and contains the probability for each state of the system, for its being the initial state:

$$\pi_i \;=\; P(s_1 = i), \quad 1 \leq i \leq N. \tag{2.5}$$

Because the system must start at one of the N states, $\boldsymbol{\pi}$ satisfies the constraint given by

$$\sum_{i=1}^{N} \pi_i \;=\; 1. \tag{2.6}$$

2.2 Discrete-Observation Hidden Markov Model

In an HMM, the observation is a probabilistic function of the state of the system, as opposed to the OMC, where the observation is a deterministic function of the state of the system. The *observation sequence* is a function of a double-embedded stochastic process. The first process is the state transitions, and the second process produces the observations.

To model this, another variable is added to the existing Markov model. The observation probability b_{jk} is the probability of the system producing observation k while being in state j:

$$b_{jk} \;=\; P(X_t = o_k | s_t = j), \quad 1 \leq j \leq N,\ 1 \leq k \leq K, \tag{2.7}$$

where o_k comes from the *observation space* of a discrete HMM, given by

$$\mathcal{O} \;=\; \{o_1,\; o_2,\; \ldots\;,\; o_M\}. \tag{2.8}$$

We might also refer to the observation probability using b_{jX_t} or $b_{s_tX_t}$ without regard to the actual value of the observation or actual value of the state. This alternative notation is necessary for the continuous observation case described in the next section.

The *observation probabilities* are stored in an $N \times K$ matrix $\mathbf{B}$, where each row represents a state and each column represents an observation:

$$\mathbf{B} \;=\; \{b_{jk}\}_{j,k=1}^{j=N,\; k=K}. \tag{2.9}$$

Since each state must produce an observation,

$$\sum_{k=1}^{K} b_{jk} \;=\; 1, \quad 1 \leq j \leq N. \tag{2.10}$$

The HMM can now be represented by $\Phi \;=\; (\mathbf{A}, \mathbf{B}, \boldsymbol{\pi})$.

2.3 Continuous-Observation Hidden Markov Model

In the discrete HMM, the observation space is a discrete set. A variation of this case is the continuous-observation HMM, where the observations come from a continuous distribution. In this case, each state has a probability distribution function (p.d.f.) associated with its observations.

One way to model the p.d.f. of a state is to treat the distribution as the sum of multiple Gaussian distributions. Linear combinations of Gaussian mixtures are capable of representing many types of sample distributions. The Gaussian mixture model (GMM) is a statistical tool for forming smooth approximations of arbitrarily shaped densities [8]. A Gaussian dis-

tribution can be fully parametrized by its mean and variance. The p.d.f. for a state j can be written as

$$g_j(\boldsymbol{x}) \;=\; \sum_{k=1}^{M} c_{jk}\, \mathcal{N}(\boldsymbol{x}, \boldsymbol{\mu}_{jk}, \boldsymbol{\Sigma}_{jk}), \tag{2.11}$$

where $\mathcal{N}(\boldsymbol{x}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$ is the probability of observation vector $\boldsymbol{x}$ occuring within a Gaussian (normal) distribution with mean vector $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$. The *mixture weight* is denoted by c_{jk}, which corresponds to state j, mixture k. The *mean* is denoted by the vector $\boldsymbol{\mu}_{jk}$, corresponding to state j, mixture k. Finally the *covariance* is denoted by the matrix $\boldsymbol{\Sigma}_{jk}$, corresponding to state j, mixture k. To make the GMM a p.d.f., the mixture weights must satisfy the constraint

$$\sum_{k=1}^{M} c_{jk} \;=\; 1, \qquad 1 \leq j \leq N. \tag{2.12}$$

2.4 HMM Procedures

Here, we discuss the three HMM procedures: evaluation, decoding, and training. The evaluation procedure involves computing $P(\mathcal{X}|\Phi)$, the probability that the model generates the observation sequence. The observation sequence $\mathcal{X}$ is defined as

$$\mathcal{X} \;=\; (X_1,\; X_2,\; \ldots\;,\; X_T), \tag{2.13}$$

where X_t is the observation at time t. Note that the observations X_t can be scalar or vector.

In applications such as isolated-word recognition, each spoken word has its own HMM. When a word is spoken, it is evaluated with respect to each HMM. The word corresponding to the HMM with the highest probability score is chosen to be the spoken word.

In continuous-speech recognition, there are HMMs for each unit of speech. The size of the unit of speech is determined by the size of the recognizable vocabulary. These units can be as long as each word for a small vocabulary, or as short as a single phone for a larger vocabulary. As the units of speech are identified, they are reassembled into words and

eventually sentences with lexical decoding, syntactic analysis, and semantic analysis [1].

The decoding procedure involves computing the most likely state sequence for a given model and observation sequence. We define the state sequence $\mathcal{S}$ as

$$\mathcal{S} \ = \ (s_1, \ s_2, \ \ldots \ , \ s_T). \tag{2.14}$$

Here, the goal is to find $\mathcal{S}$ that maximizes the probability $P(\mathcal{S}|\mathcal{X}, \Phi)$. The result of decoding is the optimal state sequence $\mathcal{S}^*$.

In training, the goal is to find model parameters that maximize the evaluation score of one or more sets of observation sequences refered to as *training data*. Here the HMM can be considered as learning from the training data. Training provides the most interesting of the three procedures described. Current training methods are considered computationally expensive, rendering HMMs only marginally useful for some applications, where real-time training might be desirable. Finding a faster but slightly suboptimal training method might enable the use of HMMs for many new pattern-classification applications.

2.4.1 Evaluation

The goal of evaluation is to find the probability of a particular observation sequence $\mathcal{X}$ given the model Φ. There are two general ways to find this probability. In a brute-force approach, one would compute the probability of the observation sequence for every possible state sequence. For T observations and N states, this is N^T possible state sequences.

Another method is the forward algorithm. The forward method takes advantage of the redundancy of the brute-force approach to achieve a faster probability computation. At each time t the forward algorithm simply tracks the probability of producing the correct observations up to time t and being in each of the N states. Adding these N probabilities at time T produces the evaluation probability. We explain both methods in detail.

2.4.1.1 The brute-force method

The probability $P(\mathcal{X}|\Phi)$ can be expressed by the equation

$$\begin{aligned} P(\mathcal{X}|\Phi) &= \sum_{\text{all } \mathcal{S}} P(\mathcal{X}, \mathcal{S}|\Phi), \\ &= \sum_{\text{all } \mathcal{S}} P(\mathcal{X}|\mathcal{S}, \Phi) P(\mathcal{S}|\Phi), \end{aligned} \tag{2.15}$$

where "all $\mathcal{S}$" refers to the set of all possible state sequences. The probability of a particular state sequence given the model Φ, $P(\mathcal{S}|\Phi)$, can be expressed by

$$\begin{aligned} P(\mathcal{S}|\Phi) &= P(s_1|\Phi) \prod_{t=2}^{T} P(s_t|s_{t-1}, \Phi), \\ &= \pi_{s_1} \; a_{s_1 s_2} \; a_{s_2 s_3} \; ... \; a_{s_{T-1} s_T}. \end{aligned} \tag{2.16}$$

The probability $P(\mathcal{X}|\mathcal{S}, \Phi)$ is the probability of a particular output sequence occuring given particular HMM parameters and a particular state sequence:

$$\begin{aligned} P(\mathcal{X}|\mathcal{S}, \Phi) &= \prod_{t=1}^{T} P(X_t|s_t, \Phi), \\ &= b_{s_1 X_1} \; b_{s_2 X_2} \; ... \; b_{s_T X_T}. \end{aligned} \tag{2.17}$$

The probability $P(\mathcal{X}|\Phi)$ can now be expressed for a sequence of T observations as

$$P(\mathcal{X}|\Phi) = \sum_{\text{all } \mathcal{S}} \pi_{s_1} b_{s_1 X_1} \; a_{s_1 s_2} b_{s_2 X_2} \; ... \; a_{s_{T-1} s_T} b_{s_T X_T}. \tag{2.18}$$

2.4.1.2 The forward algorithm

The forward algorithm is a faster method for calculating $P(\mathcal{X}|\Phi)$. The forward algorithm simply computes the probability of being in each state and having the correct observations up to time t. This is the only information necessary to compute the same probabilities at time $t+1$. The *forward probability* $\alpha(t)$ is defined as the probability of being in a particular

state and having particular observations until time t:

$$\alpha_t(i) = P(\mathcal{X}_1^t, s_t = i|\Phi), \quad 1 \leq i \leq N. \tag{2.19}$$

Here, $\mathcal{X}_1^t$ is defined as

$$\mathcal{X}_1^t \ = \ (X_1, \ \ldots \ , \ X_t). \tag{2.20}$$

The intermediate variable $\alpha(t)$ is initialized with $\boldsymbol{\pi}$ and b_{ij} for the observation X_1 and then computed iteratively for $2 \leq t \leq T$. Initialization:

$$\alpha_1(i) = \pi_i b_{iX_1}. \tag{2.21}$$

Iteration:

$$\alpha_{t+1}(j) = \sum_{i=1}^{N} \alpha_t(i) \ a_{ij} \ b_{jX_{t+1}}. \tag{2.22}$$

The probability $P(\mathcal{X}|\Phi)$ can now be expressed as a summation of $\alpha_t(i)$ for each state i, at time T:

$$P(\mathcal{X}|\Phi) \ = \ \sum_{i=1}^{M} \alpha_T(i). \tag{2.23}$$

2.4.1.3 Computational complexity

The brute-force method computes the probability of N^T state sequences and performs $2T-1$ multiplications for each state sequence. A total of $N^T(2T-1)$ multiplications and $N^T - 1$ additions are performed in computing $P(\mathcal{X}|\Phi)$. The complexity of this method is $\mathcal{O}(TN^T)$. If, for instance, $N = 5$ and $T = 100$, the brute-force method of evaluation requires 1.6×10^{72} multiplications and 7.9×10^{69} additions.

The forward algorithm presents a more efficient evaluation method than the brute-force method. Computing $\alpha_t(i)$ by the forward algorithm requires $N + N(N+1)(T-1)$ multiplications and $N(N-1)(T-1) + (N-1)$ additions. The complexity of the forward algorithm is $\mathcal{O}(TN^2)$. For the case above, where $N = 5$ and $T = 100$, the forward algorithm requires

2975 multiplications and 1984 additions. The complexities of these two methods are shown in Table 2.1. All tables and figures appear in the last section of each chapter.

2.4.2 Decoding

The aim of the decoding procedure is to compute the optimal state sequence $\mathcal{S}$ for a given observation sequence $\mathcal{X}$. There are many ways to define the optimality criterion. One optimality criterion is the locally most likely state sequence. Here, the state sequence $\mathcal{S}^*$ consists of the most likely state at each time t, s_t^*. This definition, however, disregards the transition probabilities between various states and may pose a variety of problems in implementation. Suppose at time t, the locally optimal state s_t^* is state i. Suppose at time $t+1$, the locally optimal state s_{t+1}^* is state j. If $a_{ij} = 0$, then the most likely state sequence is actually impossible.

To get an accurate optimal state sequence, one must consider each sequence as a whole and choose the most likely sequence that produces the observation sequence. The Viterbi algorithm provides a fast solution to this procedure, using the principle of optimality within dynamic programming [11]. Note that the ability of an HMM to incorporate time evolution of a data sequence comes in at this stage.

2.4.2.1 Viterbi decoding

Decoding the observation sequence involves maximizing the probability $P(\mathcal{S}|\mathcal{X}, \Phi)$. This is equivalent to maximizing $P(\mathcal{S}, \mathcal{X}|\Phi)$. To use the Viterbi algorithm, the function $V_t(i)$ is defined as the maximum probability, at time t, of observing $\mathcal{X}_1^{t-1}$ with the state $s_t = i$ given the model Φ and written as

$$V_t(i) = \max_{\mathcal{S}_1^{t-1}} P(\mathcal{X}_1^{t-1}, \mathcal{S}_1^{t-1}, s_t = i|\Phi). \tag{2.24}$$

The Viterbi algorithm is a dynamic programming method. It uses the principle of opti-

mality, which states that if at time t, s_t is on the optimal path $\mathcal{S}_1^{T*}$, then the optimal path from time t to T, $\mathcal{S}_t^{T*}$, is the same as the portion of $\mathcal{S}_1^{T*}$ from time t to T.

Another function, $Z_t(j)$, is defined as the most likely previous state for each state j at time t. The values of $Z_t(j)$ will be an integer corresponding to the most likely previous state. $V_t(i)$ and $Z_t(j)$ are initialized at $t = 1$ and then iteratively computed for $2 \leq t \leq T$.
Initialization:

$$V_1(i) = \pi_i b_{iX_1}, \tag{2.25}$$

$$Z_1(j) = 0, \quad 1 \leq j \leq N. \tag{2.26}$$

Iteration:

$$V_t(j) = \max_{1 \leq i \leq N} V_{t-1}(i)\ a_{ij}\ b_{jX_t}, \tag{2.27}$$

$$Z_t(j) = \arg \max_{1 \leq i \leq N} \{V_{t-1}(i) a_{ij}\}. \tag{2.28}$$

The final step of the Viterbi algorithm is backtracking of $Z_t(j)$. The probability of the most likely state sequence is given by

$$P(\mathcal{S}^*|\Phi, \mathcal{X}) = \max_{1 \leq i \leq N} V_T(i). \tag{2.29}$$

The last state in the most likely state sequence is the argument of that probability:

$$s_T^* \ = \ \arg \max_{1 \leq i \leq N} V_T(i). \tag{2.30}$$

To find $s_1^*,\ s_1^*,\ \ldots\ ,\ s_{T-1}^*$, one uses the function $Z_t(j)$, which contains the most likely prior state:

$$s_t^* = Z_{t+1}(s_{t+1}^*), \quad 1 \leq t \leq T-1. \tag{2.31}$$

The most likely state sequence is $\mathcal{S}^* = (s_1^*,\ s_2^*,\ \ldots\ ,\ s_T^*)$.

2.4.2.2 Computational complexity

The Viterbi algorithm is more efficient than a brute-force method. For the brute-force method, there are N^T possible state sequences and $T(T-1)$ multiplications for each state sequence. To compute $P(\mathcal{S}, \mathcal{X}|\Phi) = P(\mathcal{S}|\Phi)P(\mathcal{X}|\mathcal{S}, \Phi)$ using Equations (2.16) and (2.17), the brute-force method requires $N^T T(T-1)$ multiplications and no additions. Its complexity is $\mathcal{O}(N^T T^2)$. For an observation sequence where $T = 100$ and an HMM where $N = 5$, the brute-force method requires 7.8×10^{73} multiplications.

The Viterbi algorithm requires $N + (T-1)N(N+1)$ multiplications and no additions. Its complexity is $\mathcal{O}(N^2 T)$. For the case above, where $N = 5$ and $T = 100$, the Viterbi algorithm requires 2975 multiplications. The complexities of these two methods are shown in Table 2.2.

2.4.3 Training (learning)

In the training procedure, the goal is to find Φ that maximizes the probability $P(\mathcal{X}|\Phi)$. The training procedure can be thought of as the learning procedure for the HMM. Here the HMM learns from the training observation sequences. There is no analytical method to maximize this probability. The training is accomplished using iterative methods.

The most common training method is the Baum-Welch (BW) algorithm, an EM algorithm for HMMs [9]. In the BW algorithm, the iterations begin with an initial model Φ and a set of training observations $\mathcal{X}$. The model Φ is iteratively updated until it converges to a model $\hat{\Phi}$ where the probability $P(\mathcal{X}|\hat{\Phi})$ is locally maximal with respect to $\hat{\Phi}$. The BW algorithm is discussed in detail in Appendix A. The implementational issues of the BW algorithm are discussed in Appendix B.

The BW algorithm finds a locally optimal model (in a maximum likelihood sense) in an iterative way. The location of the local optimum on the likelihood surface depends on the initial estimate. Zhang and Kuo propose an iterative and a noniterative suboptimal method

for training HMMs [9]. The noniterative method uses clustering and calculates the mean and variance of various clusters to form initial mixtures. The iterative method uses Viterbi decoding to redistribute states to new clusters and recalculates cluster means and variances to refine the model. These are discussed in detail in Chapter 3.

2.5 Tables

Table 2.1 In the HMM evaluation procedure, the forward algorithm provides considerable computational savings over the brute-force method.

Procedure	Multiplications	Additions	Complexity
Brute-Force	$N^T(2T-1)$	N^T-1	$\mathcal{O}(TN^T)$
Forward Algorithm	$N+N(N+1)(T-1)$	$N(N-1)(T-1)+(N-1)$	$\mathcal{O}(TN^2)$

Table 2.2 In the HMM decoding procedure, the Viterbi algorithm provides considerable computational savings over the brute-force method.

Procedure	Multiplications	Additions	Complexity
Brute-Force	$N^TT(T-1)$	0	$\mathcal{O}(N^TT^2)$
Viterbi Algorithm	$N+(T-1)N(N+1)$	0	$\mathcal{O}(N^2T)$

CHAPTER 3

ALTERNATIVE HMM TRAINING METHODS

This chapter describes two alternative training methods for HMMs. The first is a noniterative method proposed by Zhang and Kuo [9]. This learning method, in addition to training quickly, also does not require an initial model. The other method discussed here is an iterative method also proposed by Zhang and Kuo [9]. This method is a modification of the noniterative method.

3.1 An Alternative Training Method (Noniterative)

We will refer to Zhang and Kuo's noniterative method by the abbreviation ZKNI. This method uses clustering and mean and variance calculation to form an HMM from the training data. This process is outlined in Figure 3.1.

3.1.1 State clustering

ZKNI starts by clustering the entire set of observations into N clusters. Each of the clusters represents a state within the HMM. Clustering can be accomplished using a number of methods. Zhang and Kuo suggest using generalized Lloyd iteration [12].

The performance of generalized Lloyd iteration is dependent on the initial codebook used by the algorithm [13]. Using certain initial codebooks results in poorly formed clusters. To

avoid the added complexity of creating a robust codebook initialization method, we use the *modified K-means algorithm* for clustering our observation vectors [14]. The modified K-means algorithm does not require an initial codebook, so each attempt to cluster the same observations with the same number of clusters produces the same results. The K-means and modified K-means algorithms are discussed in detail in Appendix C.

3.1.2 Mixture clustering

Once each observation is assigned to a state cluster, the clusters are further split into M smaller clusters. These clusters are the basis for each of the M mixtures for each of the N states in the HMM. Zhang and Kuo again suggest using generalized Lloyd iteration to split the state clusters. We instead use the modified K-means algorithm to split our state clusters into mixture clusters.

3.1.3 Mean and variance calculation

With every observation assigned to a state, the parameters of the HMM can now be estimated. The mean vector $\boldsymbol{\mu}_{jk}$ and covariance matrix $\boldsymbol{\Sigma}_{jk}$ are calculated from the mean and variance of the observations that fall into each mixture cluster. These calculations are given by

$$\boldsymbol{\mu}_{jk} = \frac{\sum\limits_{X_t \in U_{jk}} X_t}{F_{jk}}, \tag{3.1}$$

$$\boldsymbol{\Sigma}_{jk} = \frac{\sum\limits_{X_t \in U_{jk}} (\boldsymbol{\mu}_{jk} - X_t)(\boldsymbol{\mu}_{jk} - X_t)^T}{F_{jk}}. \tag{3.2}$$

Here, U_{jk} is the set of all observations that fall into the cluster for state j and mixture k. F_{jk} is the number of observations contained in the set U_{jk}. The notation X_t refers to the transpose of the vector X_t. For the sake of simplicity, Zhang and Kuo assume that each dimension of the observations is independent. The covariance matrix $\boldsymbol{\Sigma}_{jk}$ is a diagonal matrix containing only the variance of each dimension. This calculation is repeated for each

of the M mixtures in each of the N states. This provides a fairly straightforward method to initially estimate $\boldsymbol{\mu}_{jk}$ and $\boldsymbol{\Sigma}_{jk}$.

3.1.4 Mixture weight calculation

The next step in ZKNI is to use the number of observations in each state and mixture cluster to estimate the mixture weights, c_{jk}. The mixture weight c_{jk} is estimated as the ratio of the number of observations that fall within a particular mixture cluster and the total number of observations in the state cluster. We compute c_{jk} as

$$c_{jk} = \frac{F_{jk}}{\sum_{k=1}^{M} F_{jk}}. \tag{3.3}$$

Note that this calculation of c_{jk} satisfies the constraint given by Equation (2.12).

3.1.5 State transition probability calculation

The last step of ZKNI is to estimate the state transition probabilities, a_{ij}. To calculate a_{ij}, ZKNI counts the frequency of each state transition and normalizes that value by the number of transitions from each state. This is expressed as

$$a_{ij} = \frac{n_{ij}}{\sum_{j=1}^{N} n_{ij}}, \quad 1 \leq i, j \leq N. \tag{3.4}$$

Here, n_{ij} is the number of state transitions from state i to state j. Note that by the nature of this definition, this calculation satisfies the constraint given by Equation (2.4).

The final step is to compute $\boldsymbol{\pi}$. Since the initial state distribution is used only once in each evaluation, this is trivial. The simplest distribution for $\boldsymbol{\pi}$ is a uniform distribution. This is given by

$$\pi_i = \frac{1}{N}, \quad 1 \leq i \leq N. \tag{3.5}$$

This calcuation of π_i also satisfies the unity constraint given by Equation (2.5).

3.1.6 Comparison to BW algorithm

Because of its noniterative nature, ZKNI trains HMMs very quickly. This method also requires no initial Φ, meaning that it can form an initial estimate solely from the training data. Thus it can provide a starting point for any of the iterative methods.

The ZKNI algorithm, however, is not guaranteed to produce a result that is even a local maximum. Remember that the BW algorithm converges to a local maximum.

3.2 An Alternative Training Method (Iterative)

Zhang and Kuo's iterative (ZKI) training method improves upon ZKNI by incorporating the Viterbi algorithm into the training process [9]. Using each Viterbi decoding, ZKI reassigns observations into new state clusters and recomputes the GMMs for each state. When Viterbi produces an optimal state sequence, the state clusters are redefined by the observations corresponding to the occurence of each state within the optimal state sequence. If the optimal state at time t is $s_t = j$, then the observation X_t is now part of the state cluster j.

This continues until all of the observations converge into a state. This process is outlined in Figure 3.2.

3.2.1 Initialization

Like any iterative training method, ZKI requires an initial estimate and improves upon that estimate to train an HMM. Zhang and Kuo suggest using a ZKNI-trained model as an initial estimate for ZKI. An alternative initial model would be a flat model, where a_{ij} and c_{jk} are uniform, and $\boldsymbol{\mu}_{jk}$ and $\boldsymbol{\Sigma}_{jk}$ are randomly distributed.

3.2.2 Viterbi decoding

After a model is trained using ZKNI, the Viterbi algorithm decodes the observation sequence into a most-likely state sequence. With this state sequence, the ZKI training method reassigns observations to new states. With all of the observations in new states, it is necessary to recalculate the state-transition probabilities, the mixture weights, and means and variances of the model.

3.2.3 State elimination

One of the problems we encounter is that often a most-likely state sequence obtained by Viterbi decoding will not include some of the states. Since the model's parameters are computed from the observations corresponding to each state at each iteration, this will result in divide-by-zero errors. For instance, if state j is not contained in $\mathcal{S}^*$, then in Equations (3.3) and (3.4), the denominator will be zero. The ZKI uses these equations to reestimate a_{ij} and c_{jk} and an empty state will result in divide-by-zero errors.

Although this problem is not discussed in the original definition of ZKI in [9], it is a problem we encountered fairly often while attempting to train HMMs with ZKI. Our solution to this is to collapse the model by one state whenever a state is eliminated. For instance, if, while a training a 9-state HMM, one of the Viterbi-decoded state sequences only uses eight of the 9 states, then we simply collapse the HMM into an 8-state HMM. The 8-state HMM is essentially a special case of the 9-state HMM where one state is impossible to reach.

3.2.4 GMM recalculation

When the observations are reassigned to new state clusters, the mixture weight c_{ij}, the mean vector $\boldsymbol{\mu}_{jk}$, and covariance matrix $\boldsymbol{\Sigma}_{jk}$ are recalculated with a modification of the EM algorithm used for GMMs. This procedure is discussed in detail in Appendix D.

Since this procedure is an iterative method, it needs an initial GMM to optimize. Zhang

and Kuo suggest splitting the state cluster into M clusters and calculating the mean vectors and covariance matrices of each cluster as initial mean vectors and covariance matrices. The mixture weight can be expressed as a ratio of the number of observations within a mixture cluster and the total number of observations in the state cluster containing the mixture cluster. This is the same process used in estimating the mixture weights, means, and variances in ZKNI, described in Sections 3.1.3 and 3.1.4.

To save some training time, we propose a few changes to this procedure. First, instead of recomputing the GMM for every every state cluster, we only calculate GMMs for state clusters that have gained or lost observations since the previous iteration of ZKI. If a GMM is trained for a certain set of observations, it is redundant to retrain the GMM for the same set of observations. In the first few iterations of ZKI, many observations change states and this modification provides no benefit. In later iterations, as few as one or two observations change states. In these situations, we only train the GMMs for the few states that have gained or lost observations.

We also modified ZKI by changing the GMM initialization for each ZKI iteration. Zhang and Kuo suggest clustering state clusters into M mixtures at each iteration. We simply use the GMM for the previous iteration as an initial estimate for the current GMM to be trained. The theory behind this is that the number of observations leaving or joining a state is relatively small compared to the collection of observations that are already within that state. Since most of the training data is the same as in the previous iteration, this means the previously trained GMM is already a good estimate of the new GMM for each particular state. The GMM training process updates the parameters c_{jk}, $\boldsymbol{\mu}_{jk}$, and $\boldsymbol{\Sigma}_{jk}$.

3.2.5 State transition probability update

The state transition probabilities a_{ij} are updated the same way as in ZKNI. They are expressed as a function of the ratio of the frequency of a particular state-transition pair and the total number of transitions from the origin state. When updating $\mathbf{A}$, it is important

to remember to use the new state indices, based on the Viterbi decoding, rather than the original state indices, based on clustering. The formula for a_{ij} is given by Equation (3.4).

Once the HMM parameters are recalculated, the Viterbi algorithm once again decodes the observation sequence into a most-likely state sequence. Once again, we account for state elimination if such a condition exists.

3.2.6 State sequence comparison

The final step of ZKI is to compare the result of the most recent Viterbi decoding with that of the previous. If these decodings are not the same, then one or more observations have switched states. In this situation the algorithm returns to the GMM recalculation stage to recalculate the HMM parameters. If the previous two decodings are the same, then the HMM is said to have converged and the HMM training is complete.

3.2.7 Comparison to BW algorithm

Although ZKI is an iterative method, and slower than ZKNI, it still runs faster than the BW algorithm. A drawback to this algorithm is that, like the BW algorithm, ZKI is an iterative method. This means that the algorithm must be provided with an initial HMM to start the iterations. The trained model's ability to accurately represent the training observations is highly dependent on the accuracy of the initial model.

3.2.8 Flat initialization

For iterative training methods, it is necessary to provide an initial model Φ that will be iteratively improved upon. A common initial model is a *flat model*, where a_{ij} and c_{jk} are uniformly distributed. Here, $a_{ij} = \frac{1}{N}$ and $c_{jk} = \frac{1}{M}$. The mean vector $\boldsymbol{\mu}_{jk}$ is random as is the the covariance matrix $\boldsymbol{\Sigma}_{jk}$.

When training a model with the BW algorithm, such a model is a valid starting point.

The BW algorithm uses relative likelihoods with respect to each mixture to modify its parameters. When training with ZKI, however, such an initial model will result in a poor final model. The state reassignment is done through Viterbi decoding, which uses the probabilities a_{ij} and c_{jk} to compute an optimal state sequence. Because these entities are uniformly distributed, they play a trivial role in this calculation. The initial Viterbi decoding will map each observation to the mixture with the nearest mean vector. Since the mean vectors are initially random and may lie anywhere in the L-dimensional vector space (where L is the vector length of the observations), it is possible that all of the observation vectors may be mapped to a few state clusters. Thus, many of the states in the model will be eliminated during the first iteration of ZKI. The initial model for ZKI must contain some relevance to the observation data. Training the initial model with ZKNI seems sufficient for using ZKI.

3.3 Figures

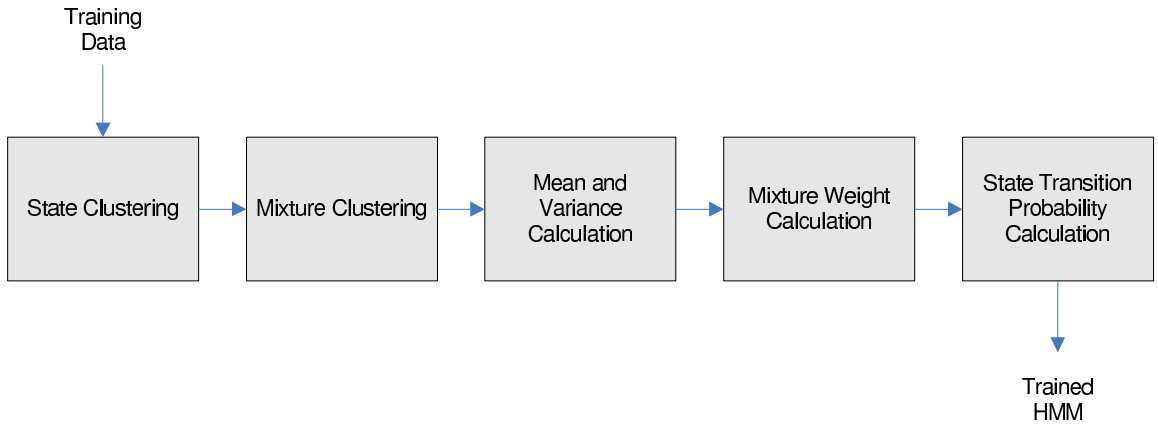


Figure 3.1 Zhang and Kuo's noniterative (ZKNI) training method uses clustering to assign observations into state and mixture clusters. Then it computes HMM parameters from the means and variances of the observations in the clusters, relative size of the clusters, and frequency of transitions between clusters.

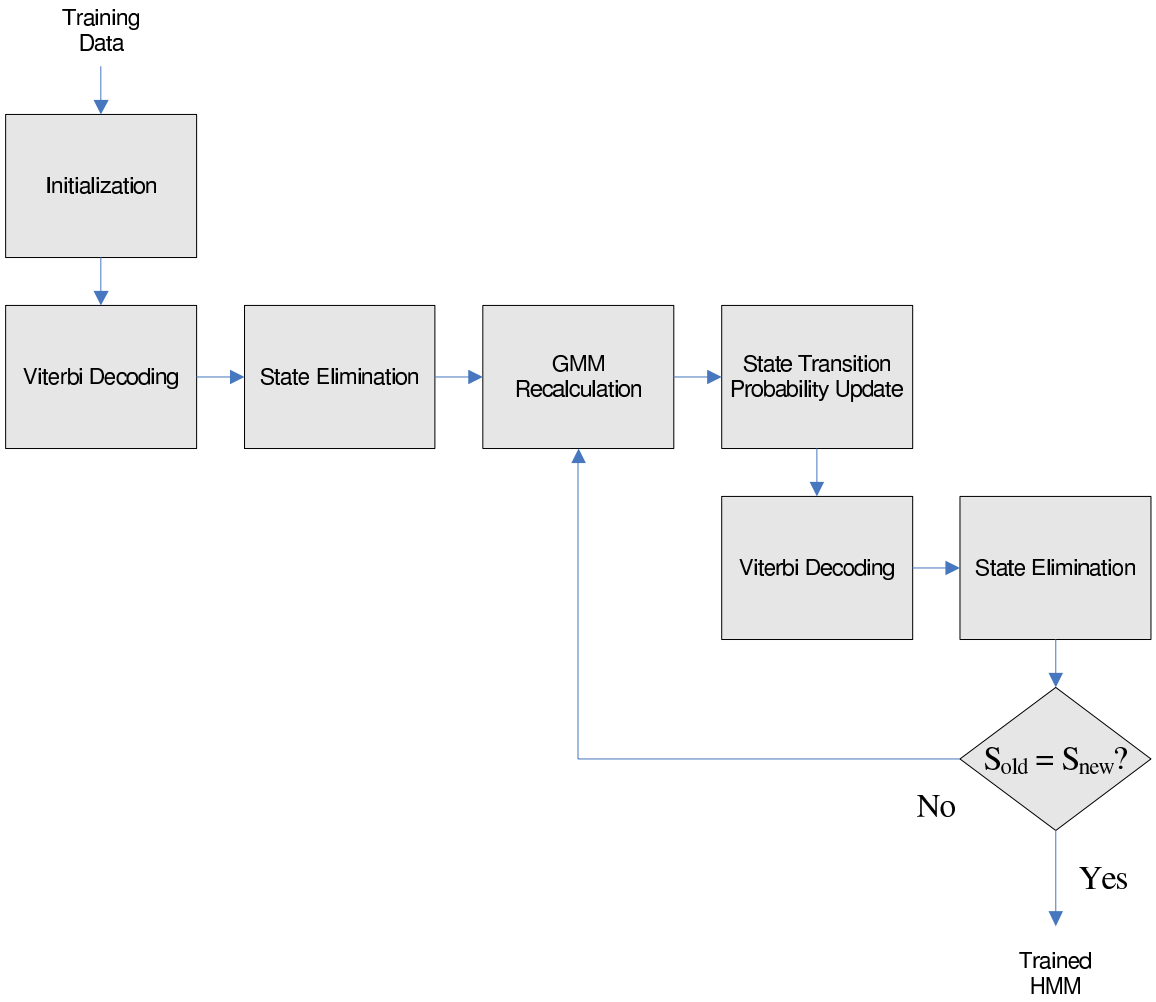


Figure 3.2 Zhang and Kuo's iterative (ZKI) training method expands upon ZKNI by using Viterbi decoding to reassign observations to states. Then it uses Gaussian mixture model (GMM) training algorithms and the methods from the noniterative method to recompute HMM parameters. This is done until none of the observations switch states after Viterbi decoding. Here, the test $\mathcal{S}_{old} = \mathcal{S}_{new}$? refers to comparing the previous two optimal state sequences to see if they are the same.

CHAPTER 4

TRAINING HMMS FOR DIGIT RECOGNITION

Here we compare the training methods BW, ZKNI, and ZKI on an isolated digit recognition application. To compare the performance of the training methods, we look at the recognition rate of the HMM models trained by the different training methods. We use sound files of 50 adult male speakers speaking the digits "one" through "nine," "oh," and "zero." These sound files are taken from the TI-DIGIT database. The TI-DIGIT database is a library created by Texas Instruments during the early 1980s that contains sound files in SPHERE format of various speakers saying the digits listed above in isolated conditions (one digit spoken per sound clip), as well as in series (multiple digits spoken per sound clip). The files are sampled at 20 kHz, with 16-bit resolution and recorded in mono. The database includes various types of speakers including men, women, and children. For this test, we used 50 male speakers speaking each of the 11 isolated digits. We had, in total, 550 total spoken digits, since 50 speakers each spoke the 11 digits. In each *realization* of this test, 40 of the 50 speakers were randomly chosen for training the HMMs, each of which corresponds to a digit. These speakers were the *training group*. The remaining 10 were used only for testing the models. These were the *test group*. We performed 100 realizations of this test, with a different set of 40 speakers used to train the models in each realization. We used the 13-dimensional mel-frequency cepstral features (MFCC) taken at 10-ms intervals as our

feature vectors for each sound file [15].

For each realization of the test described above, we trained 4 sets of 11 models. We have four training algorithms whose performance we analyzed. For each training algorithm, we trained one model for each of the 11 spoken digits. We compared the following training methods:

- Zhang and Kuo's noniterative method (ZKNI)
- Zhang and Kuo's iterative method with ZKNI initialization (ZKI)
- Baum-Welch algorithm with ZKNI initialization (BW-ZK)
- Baum-Welch algorithm with flat initialization (BW-F)

In training our models, we trained each as a 5-state and a 9-state model. This let us observe the relative benefits of having more states with each training method. Due to the state elimination problem with ZKI, training some models to be 9-state models resulted in 7-state or 8-state models. The state elimination condition is discussed in detail in Section 3.2.3.

After training our models, we attempted to use them to recognize the training and test groups. We also used a set of metrics to compare the performance of each training method. The four metrics used are the hit rate, the false-alarm rate, the confidence ratio, and the error ratio. The *hit rate* of a digit is the relative number of times a particular digit is identified correctly. The *false-alarm rate* of a digit is the relative number of times that other spoken digits are mistakenly identified as that particular digit. The *confidence ratio* of a digit is the ratio of the likelihood between a correctly identified digit and the second-most likely digit. The *error ratio* of a digit is the ratio of the likelihood between the identified digit and the actual spoken digit. Note that the confidence ratio applies only to correctly identified digits, while the error ratio applies only to misidentified digits. These metrics are described in detail in Section 4.2.

We also studied the effects of noise on our models. For the noise case, we still used the models trained with clean spoken digits. For each noisy case we computed all of the above metrics.

4.1 Convergence Criteria

To begin training HMMs for multiple digits, we first decided upon convergence criteria for the BW algorithm. To do this, we initially trained a single digit and tracked the evolution of the important variables within its HMM through multiple iterations. We trained a single digit through 1000 iterations of the BW algorithm, and for each iterative model Φ, we plotted $P(\mathcal{X}|\Phi)$, ΔA, ΔC, $\Delta\mu$, and $\Delta\Sigma$, as defined by Equations (A.21)-(A.24).

Here, $P(\mathcal{X}|\Phi)$ is the probability of the training data after each iteration of the BW algorithm. The variables ΔA, ΔC, $\Delta\mu$, and $\Delta\Sigma$ are the changes in $\boldsymbol{A}$, c_{jk}, $\boldsymbol{\mu}_{jk}$ and $\boldsymbol{\Sigma}_{jk}$ through each iteration of the BW algorithm. Each of the Δ-variables is essentially a norm of the difference between a component of Φ and its corresponding component of $\hat{\Phi}$. We plotted the $\log_{10} P(\mathcal{X}|\Phi)$ as a function of the number of iterations.

Figure 4.1 shows $\log_{10} P(\mathcal{X}|\Phi)$ for the first 100 iterations of the BW algorithm. The HMM essentially converges during this window, so we limited our future BW executions to no more than 50 iterations. Next, we plotted ΔA, ΔC, $\Delta\mu$, and $\Delta\Sigma$ as a function of the iteration. Figure 4.2 shows the logarithm of all four of these variables plotted for all 1000 iterations.

Figure 4.2 shows that the variables initially change significantly, but they converge logarithmically to zero. We can use this data to potentially stop the BW algorithm early if it convergences quickly. Here, $\boldsymbol{A}$ and c_{jk} converged faster than $\boldsymbol{\mu}_{jk}$ and $\boldsymbol{\Sigma}_{jk}$. For our training, we chose $\epsilon_A = \epsilon_C = 0.01$, and $\epsilon_\mu = \epsilon_\Sigma = 0.1$ is sufficient to ensure convergence. Thefore, our BW executions stopped either when the first four conditions in Table 4.1 were true, or when the fifth condition in Table 4.1 was false.

4.2 Performance Metrics

To compare the performance of models trained with different methods, we use metrics to quantify the general performance of a class of HMMs. The four metrics we use to compare performance of models are the *hit rate*, the *false-alarm rate*, the *confidence ratio*, and the *error ratio*. The hit rate and false-alarm rate are fairly well-known statistical tools, while the confidence ratio and error ratio are metrics we have devised to quantify how well a digit is identified, or how poorly a digit is misidentified. Because the goal of training an HMM is to make the training data as likely as possible, we expect that a well-trained HMM will not only correctly identify a spoken digit, but will identify it so well that the probability score of the correct digit will be many magnitudes greater than the next-best probability score. Likewise, we expect that when a digit is misidentified, the probability score of the correct digit will be very close but just short of the identified digit.

4.2.1 Confusion matrices

The confusion matrix is a simple tool used to track the frequency of combinations of each digit being spoken and each possible classification of that digit. For example, consider the sample 3×3 confusion matrix, given in Table 4.2. Here, the column represents the identified digit, while the row represents the actual spoken digit. For the sample confusion matrix, the spoken digit 'one' is identified as 'one' 9 times. It is identified as 'two' twice and as 'three' once. Likewise, the spoken digit 'two' is identified as 'one' twice, as 'two' 8 times, and as 'three' 3 times.

We define the *hit rate* (HR) as the relative frequency that a spoken digit is identified correctly:

$$HR_i = \frac{m_{ii}}{\sum_j m_{ij}}, \tag{4.1}$$

where m_{ij} refers to the value of row i, column j in the confusion matrix. For our sample

confusion matrix, the HR for 'one' is $HR_1 = \frac{9}{9+2+1} = 0.75$. Likewise, the HR for 'two' is $HR_2 = 0.62$, and the HR for 'three' is $HR_3 = 0.83$. An ideal isolated digit recognizer will identify every digit correctly and $HR_i = 1$ for all i.

We define the *false-alarm rate* (FAR) of a particular digit as the relative frequency that a different spoken digit is incorrectly identified as that digit:

$$FAR_j = \frac{\sum\limits_{i,\ i \neq j} m_{ij}}{\sum\limits_{i,\ i \neq j} \sum\limits_{k} m_{ik}}. \tag{4.2}$$

For our sample confusion matrix, the FAR for 'one' is $FAR_1 = \frac{2+0}{2+8+3+0+2+10} = 0.08$. Likewise, the FAR for 'two' is $FAR_2 = 0.18$ and the FAR for 'three' is $FAR_3 = 0.16$. In an ideal digit recognizer, no spoken digit will be misidentified, and $FAR_i = 0$ for all i.

The HR and FAR thus share an inverse relationship with each other. For instance, we could create a system where every spoken digit is recognized as 'five,' regardless of what is actually said. This would mean that $HR_5 = 1$; however, many other digits would also be falsely recognized as 5. Thus, $FAR_5 = 1$ also. We wish to maximize HR for each digit while minimizing the FAR. To demonstrate this relationship we can plot the HR as a function of the FAR for each of the spoken digits. This may give us great insight into the operation of the digit recognizer. For the isolated digit recognition test, we used 11 different spoken digits, so our confusion matrix was an 11×11 matrix with one row for each spoken digit and one column for each identified digit.

4.2.2 Likelihood analysis

The purpose of training an HMM is to maximize the probability of the training data, $P(\mathcal{X}|\Phi)$. Since we use the probability $P(\mathcal{X}|\Phi)$ as our probability score, we can assume that a well-trained set of models will not only correctly identify a spoken digit, but will have a probability score that is many magnitudes higher than the next-best probability score. Conversely, when

a well-trained set of HMMs misidentifies a spoken digit, the probability score of the model for the actual digit will be only a few magnitudes below the probability score of the incorrectly identified digit.

We define the *confidence ratio* as the logarithm of the ratio between the probability score for a correctly identified digit and the next-best probability score:

$$CR_i = \log_{10} \frac{P(\mathcal{H}_i|\Phi_i)}{\arg\max\limits_{j,\ j\neq i} P(\mathcal{H}_i|\Phi_j)}. \tag{4.3}$$

Here, $\mathcal{H}_i$ refers to the observation sequence corresponding to the spoken digit i. The model Φ_j is the model for spoken digit j. In an optimal digit recognizer CR_i is maximized.

For a misidentified digit, we define the *error ratio* (ER) as the logarithm of the ratio between the best probability score and the probability score of the actual digit:

$$ER_i = \log_{10} \frac{\arg\max\limits_{j,\ j\neq i} P(\mathcal{H}_i|\Phi_j)}{P(\mathcal{H}_i|\Phi_i)}. \tag{4.4}$$

In an optimal digit recognizer, when the recognizer fails, it will do so by a small magnitude. Thus ER_i should be minimized.

To express each digit's CR_i and ER_i for a set for HMMs, we compute the mean CR_i for each identified digit and the mean ER_i for all erroneously identified digits:

$$CR_{i,avg} = \frac{\sum\limits_{\text{Identified digits}} CR_i}{\#\text{ of correctly identified digits}}, \tag{4.5}$$

$$ER_{i,avg} = \frac{\sum\limits_{\text{Misidentified digits}} ER_i}{\#\text{ of incorrectly identified digits}}. \tag{4.6}$$

For each training method and noise case, we have 1000 spoken versions of each digit as test data and 4000 spoken versions of each digit as training data. The $CR_{i,avg}$ gives us the average confidence ratio for a particular digit being identified correctly. The $ER_{i,avg}$ gives

us the average error ratio for a particular digit being misidentified. Thus, we can compute $CR_{i,avg}$ and $ER_{i,avg}$ for each digit within each training condition, noise condition, and data set.

The metrics CR_i and ER_i should have an inverse relationship to each other. For a well-trained HMM, CR_i should be high, while ER_i should be low. Likewise, for a poorly trained HMM, CR_i will be low while ER_i will be high. Thus, we can learn much about the performance of our models by plotting CR_i as a function of ER_i.

4.3 Comparison of the Methods

Since the goal of digit recognition is to identify spoken digits, the most significant metric in measuring a digit recognizer's performance is the digit-averaged HR. The *digit-averaged hit rate* is given by the mean HR for all 11 digits:

$$HR_{avg} = \frac{\sum_{i=1}^{11} HR_i}{11}. \tag{4.7}$$

This is equal to the number of correctly identified digits divided by the number of total digits. We computed HR_{avg} for each training method, each noise condition, each number of states, and data set. The results are given in Tables 4.3 and 4.4.

From Tables 4.3 and 4.4, we can conclude the following: First, when comparing the results of BW-F and BW-ZK, we saw little or no difference in digit-averaged HR. We can say that the choice in initial models proved to be insignificant when using the BW algorithm for this particular application. Second, we saw that the ZKI-trained models achieved comparable digit-averaged HRs to their BW-trained counterparts. This led us to the conclusion that ZKI is a sufficient suboptimal training method for this application. Third, the digit-averaged HRs were comparable whether we used 5-state or 9-state models. Figures 4.3 and 4.4 plot the data contained within Tables 4.3 and 4.4.

From Figures 4.3 and 4.4, we can see that the BW-trained models achieve the highest HR_{avg}, followed by the ZKI-trained models and then finally the ZKNI-trained models. The differences between BW-F, BW-ZK, and ZKI are relatively small. The difference between ZKNI and the other three methods is a bit larger.

We also noticed that the results, in each case, were highly sensitive to white noise. We investigated this further by plotting the HR-FAR relationship for each digit within each noise condition. In Figures 4.5-4.8 we plot the HR as a function of the FAR for the 5-state ZKI models with test data in four noise cases: clean, 40-dB SNR, 20-dB SNR, and 0-dB SNR. The digits are represented by different symbols on the plots.

From Figures 4.5-4.8, we see that as the SNR was lowered, a few digit models had very high HRs and FARs. The rest had very low HRs and FARs. This indicates that some digit models are actually very responsive to white noise, leading to a low HR_{avg}. For the sake of completeness we plot these figures for ZKNI, BW-ZK, and BW-F, but only for the clean case and the 20-dB case, with both 5-state and 9-state models. This is done in Figures 4.9-4.22.

In Figures 4.9-4.22 we see that in low-SNR conditions our digit models behave the same for ZKNI, BW-ZK, and BW-F training as they did for ZKI training. When we plotted the HR-FAR characteristics for 9-state models or with respect to training data, we experienced similar results. The noise performance we observed seems to occur regardless of the training method and number of states used to train the models.

From the HR-FAR plots, we concluded that the poor performance at low SNRs was due to a few of the 11 digit models being particularly responsive to white noise. To study this further, we tabulated the HR and FAR of each training method for each digit in the clean case and the 20-dB SNR case. This is done in Tables 4.5 and 4.6. In Table 4.5, we can see that in the clean case, the HRs are almost one (the smallest being for the digit 'nine'). Meanwhile, the FARs are almost zero, with the model for the spoken digit 'one' having the highest FAR. In Table 4.6, however, we see that in the 20-dB SNR case, the noise has affected the overall performance of the digit recognizer, as compared to Table 4.5, but the

drop in digit-averaged HR is not a result of an all-around drop in HR for each digit. Instead, some of the digits have extremely low HRs, while others still achieve very high HRs. This is true for all training methods regardless of the number of states. The spoken digits 'three,' 'four,' 'five,' and 'six' each have HRs above 0.75. Meanwhile, the digits 'one,' 'nine,' and 'oh' achieve HRs of 0.00. Based on these results, we concluded that the models for the spoken digits 'three,' 'four,' 'five,' and 'six,' are particularly responsive to white noise. To explain this, we hypothesize that the spoken sounds for 'f,' 'th,' and 'x' are wideband phonemes that sound much like white noise. The HMMs trained to recognize digits with these sounds will dedicate one more more states to the cepstral features of these sounds. When presented with white noise, these models may confuse the noise to be one of these sounds. Note that the digits with high HRs and FARs are the same in each type of model. Likewise, the digits with low HRs and low FARs are the same ones, regardless of the training method or number of states.

We also look at the CR and the ER plots. Since these two metrics normally have an inverse relationship, we plotted the CR as the function of the ER for each digit at varying SNRs. In Figures 4.23 and 4.24, we plot this relationship for 5-state ZKI models with respect to testing data in four noise conditions: clean, 40-dB, 20-dB, and 0-dB SNR. Notice that as the SNR gets lower, the range of ERs gets larger. In the higher SNRs, the range of CRs is varied, but the range of ERs is very tight. At the lowest SNRs, the CRs still exhibit a wide range of values, but now the range of ERs is also large. The low-SNR plots also show the inverse relationship between CR and ER. Some digit models have very high CRs, but low ERs. Others are the opposite. This once again reveals that some of the digit models are more attuned to the noise with respect to our cepstral features. Figures 4.25-4.31 plot the CR-ER relationship for the remaining methods, with both 5-state and 9-state models, in the clean and 20-dB SNR conditions. Once again, the models trained by every training method display the same type of behavior under similar noise conditions.

To identify the models performing well under white noise, we tabulated the CR and ER

for each training method in the clean case and the 20-dB SNR case. Tables 4.7 and 4.8 show this data for the case of 9-state models with test data. In the clean case, the ERs assume a small range of values, while the CRs have a wider range of values. In the 20-dB case, however, we can see that some of the ERs reach values over 200 while others are less than 15. In this analysis, we identified digits with extremely high ERs and extremely low CRs. These were considered the poorest performers. We also identified digits with high CRs and low ERs. These were considered to have the best noise immunity. In the 20-dB case, the best performers were the digits 'three,' 'four,' 'five,' and 'six.' The worst performers were the digits 'one,' 'two,' 'nine,' 'oh,' and 'zero.' In each of the training conditions methods, the same digits were the best and worst performers. Recall from our HR-FAR analysis, that the digits 'three,' 'four,' 'five,' and 'six' achieved HRs above 0.75 at 20 dB, while the digits 'one,' 'nine,' and 'oh' achieved HRs of 0.00. The digits with the highest CRs and lowest ERs have the highest HRs. The best-performing digits from our HR-FAR analysis are also the best performers in our CR-ER analysis. This is also true of the worst-performing digits in both analyses. Further, the best and worst performers are the same digits regardless of the training method of number of states.

As a final analysis, we plotted the digit-averaged CR and digit-averaged ER as a function of the SNR. This is done in Figures 4.32-4.33. Here, we see that the CR drops as the SNR drops. The ER, meanwhile, increases as the SNR drops. The exceptions to this trend occur at the 0-dB and 10-dB cases for the CR. From this observation, we concluded that there is enough noise energy with respect to the signal energy that the digit model most responsive to white noise will almost always obtain the highest probability score. To understand how this is possible, assume temporarily that that model Φ_j responds to white noise very well. When the digit j is spoken, the resulting high likelihood ratio with respect to other models will be reflected in the CR. If our hypothesis is correct, then when a digit $i \neq j$ is spoken, it will be misidentified as j due to the white noise in the signal. The resulting high relative likelihood will be reflected in the ER. Thus, if our hypothesis is true – that some of the digit

models are simply better attuned to white noise – then we expect the low-SNR cases to have high ERs and CRs. This holds true in Figure 4.32-4.35.

The analysis of our results shows that ZKNI and ZKI are indeed usable suboptimal alternatives to the BW algorithm. Although our results showed a very high sensitivity to white noise, our analysis shows that this shortcoming is most likely a result of the particular features and application we chose to use. Our analysis shows that the various training methods achieve comparable results at each SNR level. Thus, these training methods also exhibit comparable noise immunity.

4.4 Tables and Figures

Table 4.1 The Baum-Welch algorithm converges when either the first four criteria are true, or when the last criterion listed is false.

Variable	**Criterion**
ΔA	≤ 0.01
ΔC	≤ 0.01
$\Delta \mu$	≤ 0.1
$\Delta \Sigma$	≤ 0.1
Number of Iterations	< 50

Table 4.2 A sample confusion matrix for the case of three digits.

Spoken ↓, Estimate →	'One'	'Two'	'Three'
'One'	9	2	1
'Two'	2	8	3
'Three'	0	2	10

Table 4.3 Digit-averaged HRs for 5-state HMMs. The HRs fall as the SNR decreases. At 50-dB SNR, the speech is essentially clean. At 0-dB SNR, the digit recognition is essentially random, since the HR converges to $\frac{1}{11}$.

SNR	**Test**				**Train**			
	ZKNI	*ZKI*	*BW-ZK*	*BW-F*	*ZKNI*	*ZKI*	*BW-ZK*	*BW-F*
0	0.086	0.094	0.093	0.088	0.088	0.095	0.094	0.087
10	0.138	0.196	0.210	0.191	0.139	0.199	0.215	0.192
20	0.391	0.426	0.417	0.422	0.404	0.439	0.432	0.440
30	0.456	0.501	0.504	0.507	0.474	0.536	0.538	0.557
40	0.751	0.805	0.834	0.854	0.804	0.874	0.905	0.925
50	0.939	0.952	0.953	0.950	0.981	0.990	0.994	0.994
clean	0.961	0.960	0.959	0.954	0.991	0.994	0.997	0.996

Table 4.4 Digit-averaged HRs for 9-state HMMs. The HRs fall as the SNR decreases. At 50-dB SNR, the speech is essentially clean. At 0-dB SNR, the digit recognition is essentially random, since the HR converges to $\frac{1}{11}$.

SNR	**Test**				**Train**			
	ZKNI	*ZKI*	*BW-ZK*	*BW-F*	*ZKNI*	*ZKI*	*BW-ZK*	*BW-F*
0	0.086	0.097	0.093	0.090	0.086	0.099	0.095	0.091
10	0.158	0.179	0.182	0.112	0.159	0.182	0.185	0.113
20	0.384	0.387	0.396	0.377	0.398	0.410	0.416	0.401
30	0.450	0.504	0.538	0.495	0.478	0.568	0.607	0.564
40	0.768	0.821	0.827	0.863	0.844	0.923	0.938	0.962
50	0.943	0.949	0.952	0.939	0.989	0.996	0.999	0.996
clean	0.957	0.956	0.959	0.947	0.999	0.998	1.000	0.997

Table 4.5 HRs and FARs for 9-state models with test data in the clean condition. Here, every model achieves a high HR and a low FAR, an optimal result.

Spoken Digit	**ZKNI**		**ZKI**		**BW-ZK**		**BW-F**	
	HR	*FAR*	*HR*	*FAR*	*HR*	*FAR*	*HR*	*FAR*
'one'	0.991	0.015	0.991	0.015	0.999	0.016	0.996	0.018
'two'	0.945	0.001	0.936	0.000	0.934	0.000	0.922	0.000
'three'	1.000	0.005	1.000	0.003	1.000	0.003	0.998	0.007
'four'	0.990	0.002	0.985	0.003	0.985	0.002	0.992	0.002
'five'	0.952	0.002	0.991	0.003	0.952	0.002	0.940	0.004
'six'	0.969	0.005	0.986	0.007	0.982	0.006	0.947	0.005
'seven'	0.975	0.006	0.969	0.005	0.978	0.005	0.975	0.007
'eight'	0.938	0.000	0.945	0.000	0.949	0.000	0.943	0.000
'nine'	0.856	0.003	0.865	0.003	0.853	0.002	0.829	0.002
'oh'	0.907	0.001	0.908	0.001	0.917	0.001	0.882	0.001
'zero'	1.000	0.007	0.992	0.009	0.995	0.008	0.982	0.013

Table 4.6 HRs and FARs for 5-state models with test data in the 20-dB SNR condition. In low-SNR conditions, five of the spoken digits achieve high HRs and FARs, while the remaining six achieve low HRs and FARs. This indicates that some of the models are very receptive to white noise.

Spoken Digit	**ZKNI**		**ZKI**		**BW-ZK**		**BW-F**	
	HR	*FAR*	*HR*	*FAR*	*HR*	*FAR*	*HR*	*FAR*
'one'	0.000	0.000	0.001	0.000	0.001	0.000	0.000	0.000
'two'	0.137	0.001	0.023	0.000	0.017	0.000	0.198	0.005
'three'	0.881	0.210	0.997	0.354	0.999	0.388	0.950	0.319
'four'	0.795	0.050	0.943	0.084	0.985	0.075	0.963	0.073
'five'	0.844	0.047	0.908	0.099	0.902	0.087	0.939	0.118
'six'	0.874	0.268	0.862	0.044	0.765	0.028	0.929	0.105
'seven'	0.497	0.091	0.764	0.049	0.829	0.062	0.514	0.016
'eight'	0.254	0.004	0.138	0.002	0.063	0.002	0.104	0.001
'nine'	0.000	0.000	0.034	0.000	0.022	0.000	0.016	0.000
'oh'	0.000	0.000	0.007	0.000	0.005	0.000	0.025	0.000
'zero'	0.015	0.000	0.000	0.000	0.000	0.000	0.001	0.000

Table 4.7 CRs and ERs for 9-state models with test data in the clean condition. In this example, the ERs display a fairly small range of values.

Spoken Digit	ZKNI		ZKI		BW-ZK		BW-F	
	CR	*ER*	*CR*	*ER*	*CR*	*ER*	*CR*	*ER*
'one'	45.5	7.9	56.0	8.7	62.4	5.0	58.1	3.2
'two'	54.0	24.0	55.6	20.8	60.0	21.6	55.9	N/A
'three'	69.6	N/A	85.9	N/A	93.6	N/A	70.0	7.5
'four'	75.9	4.5	84.0	8.1	89.4	12.1	83.6	5.2
'five'	52.3	9.6	57.1	13.5	58.1	14.2	58.7	17.1
'six'	49.0	18.1	61.2	12.4	60.4	12.1	58.7	10.1
'seven'	64.1	28.5	63.7	23.7	68.6	23.9	70.1	19.4
'eight'	37.6	10.5	48.5	12.1	49.8	8.9	45.2	14.4
'nine'	34.5	11.4	41.1	12.9	43.6	13.9	44.9	13.0
'oh'	50.4	31.9	54.6	28.8	62.3	32.0	58.5	26.7
'zero'	102.5	N/A	118.9	10.5	128.8	5.2	104.6	N/A

Table 4.8 CRs and ERs for 9-state models with test data in the 20-dB SNR condition. Here, the ERs much wider range of values than in the clean condition.

Spoken Digit	ZKNI		ZKI		BW-ZK		BW-F	
	CR	*ER*	*CR*	*ER*	*CR*	*ER*	*CR*	*ER*
'one'	N/A	202.4	24.0	116.8	N/A	113.7	3.0	144.0
'two'	15.6	48.4	12.2	73.3	13.3	80.3	6.6	N/A
'three'	42.8	16.1	58.5	14.5	58.5	10.3	78.8	N/A
'four'	52.5	26.6	48.1	20.3	53.7	17.9	36.1	13.8
'five'	58.0	23.5	60.6	21.5	68.1	19.3	51.8	14.3
'six'	46.7	13.5	50.3	16.2	52.2	18.1	28.9	25.6
'seven'	21.5	22.9	19.3	24.5	19.8	25.9	23.5	20.7
'eight'	15.1	33.8	12.7	34.3	13.4	37.5	10.3	48.1
'nine'	21.8	176.6	20.9	99.9	17.5	82.8	9.3	96.5
'oh'	11.5	121.2	18.1	91.6	16.3	87.3	11.8	91.0
'zero'	12.6	99.2	12.9	96.7	10.5	96.2	N/A	N/A

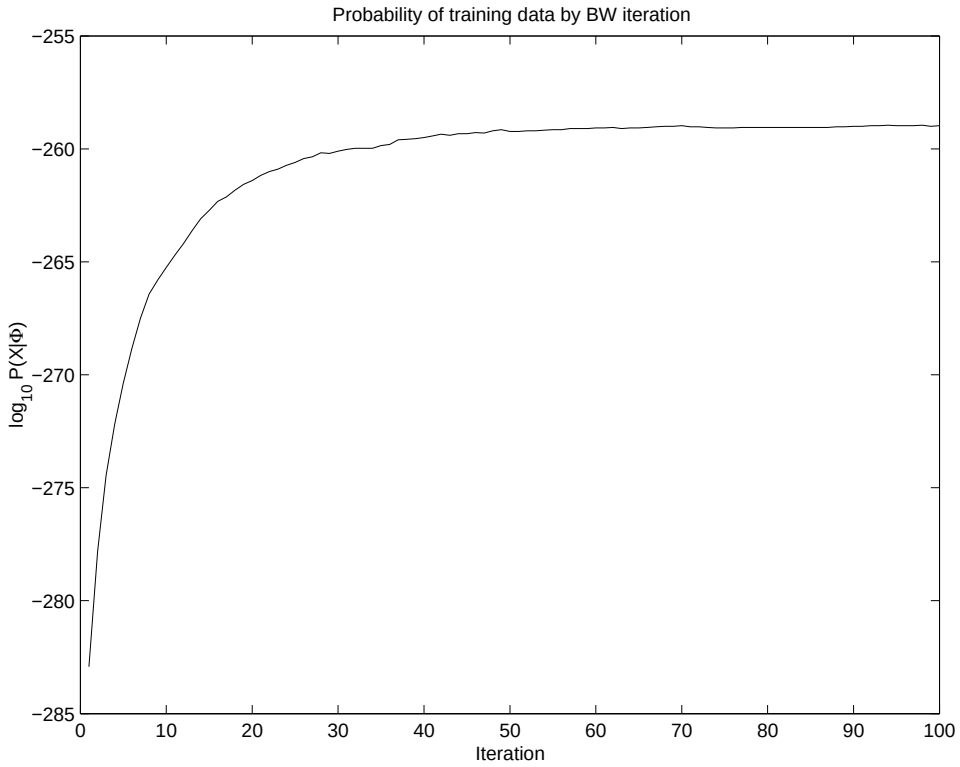


Figure 4.1 With successive BW iterations, the probability $P(\mathcal{X}|\Phi)$ converges to a local maximum with respect to Φ. This plot shows $P(\mathcal{X}|\Phi)$ for the first 100 BW iterations.

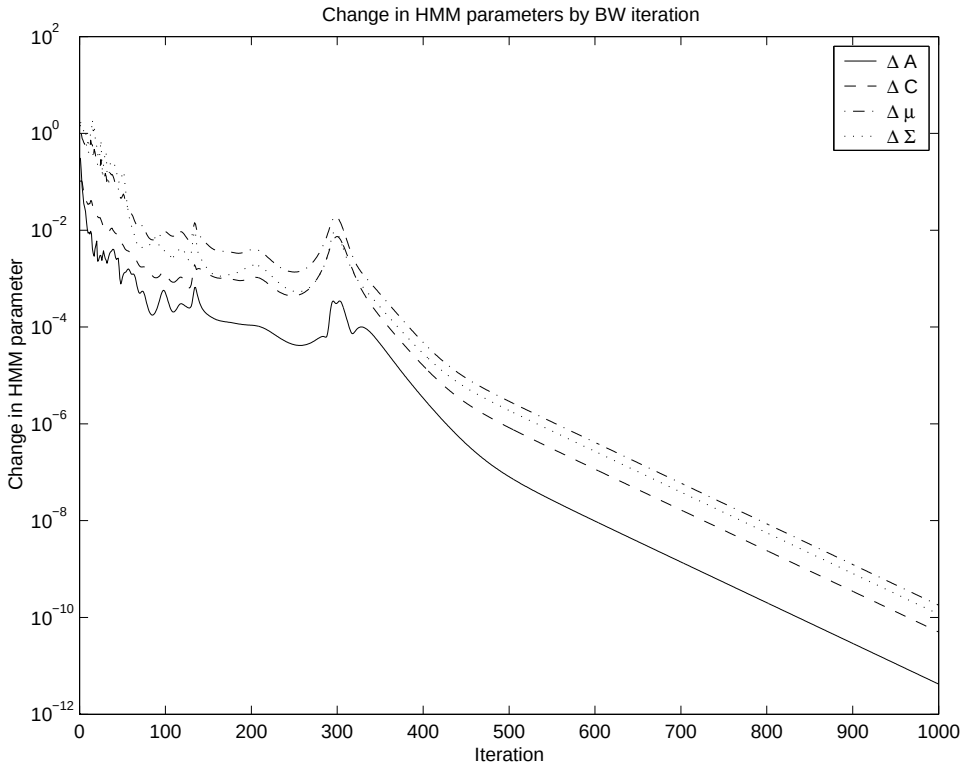
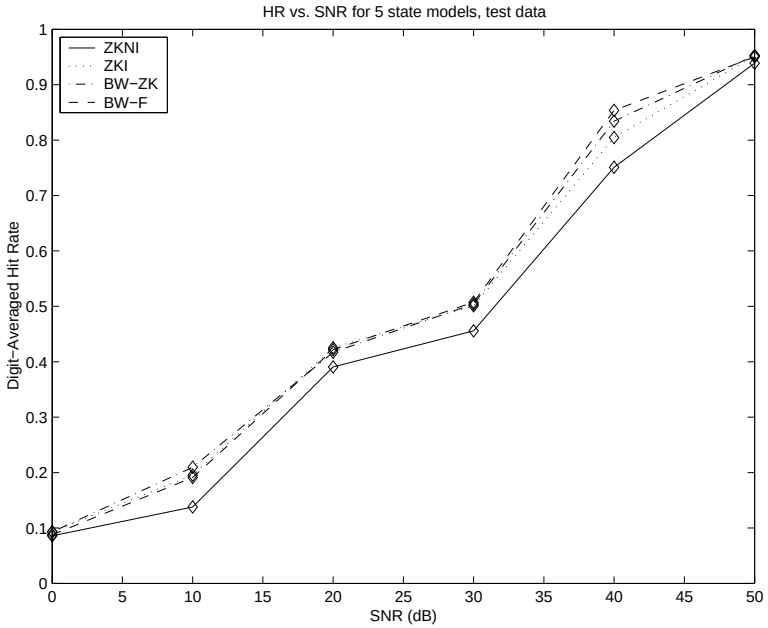
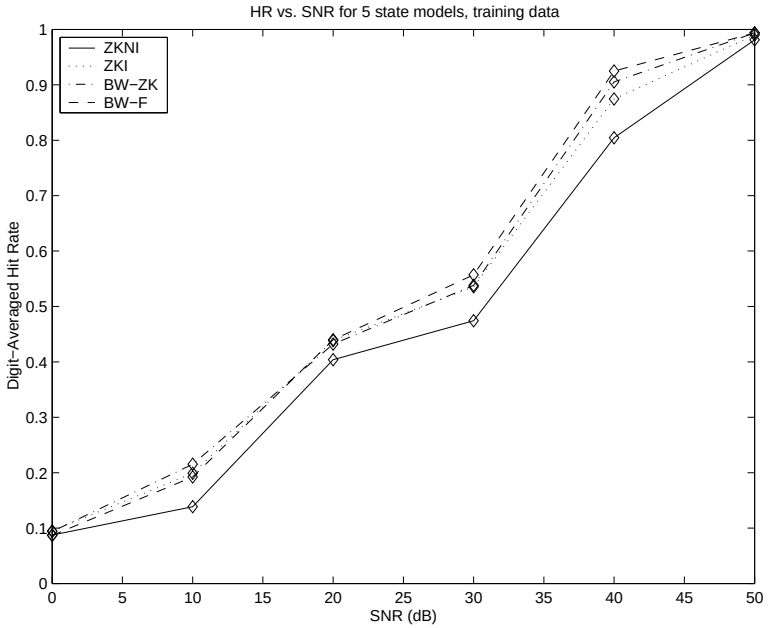


Figure 4.2 With successive BW iterations, the model Φ converges logarithmically. This plot shows the change in the parameters within Φ for the first 1000 BW iterations.

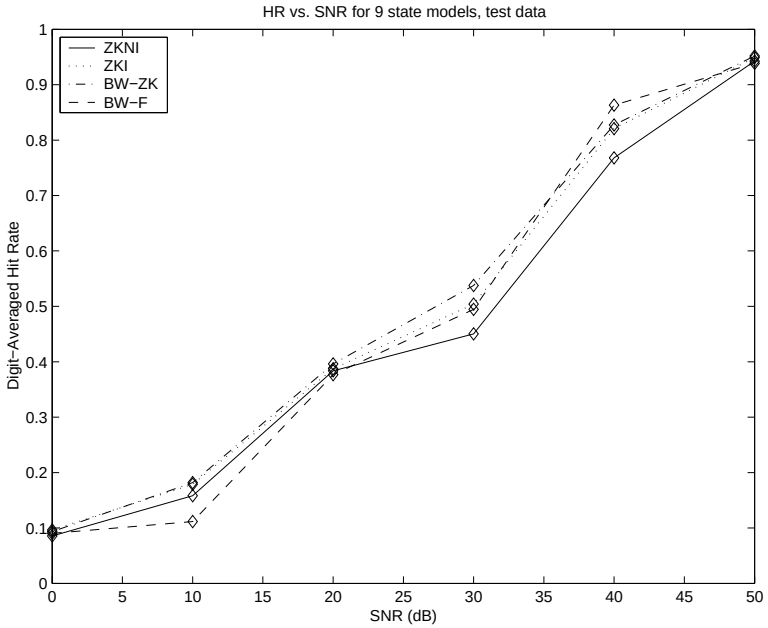


(a)

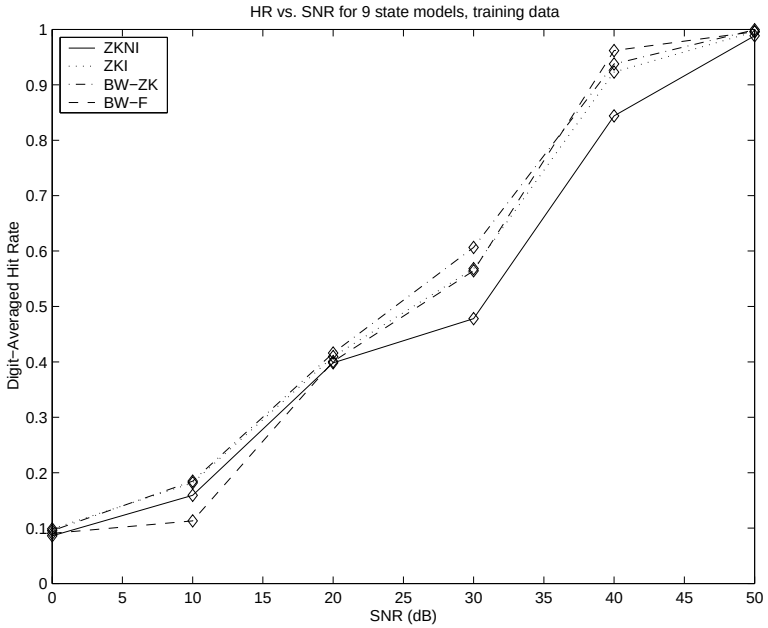


(b)

Figure 4.3 Digit-averaged HR vs. SNR for the 5-state testing and training cases. In both, we see that the two BW-trained models achieved the highest digit-averaged hit rate. The ZKI-trained models perform the next best and the ZKNI-trained models perform the worst. (a) Test data. (b) Training data.

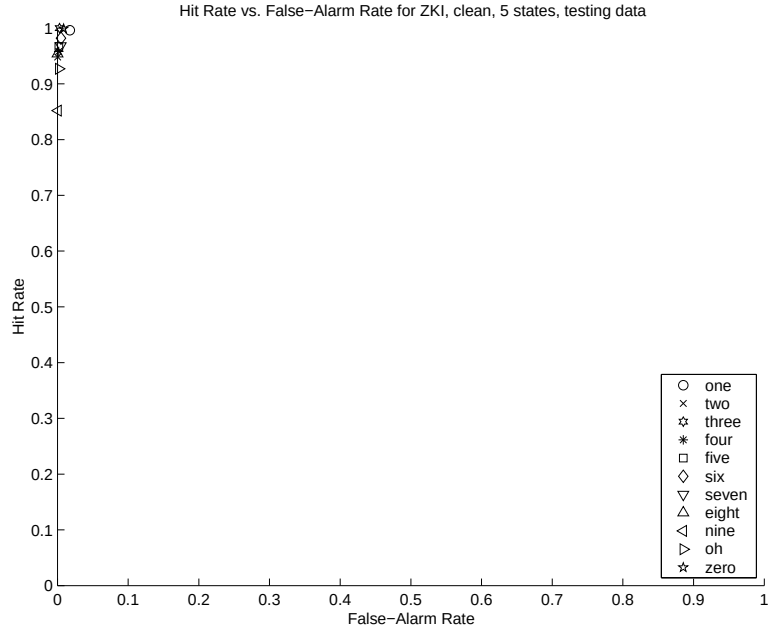


(a)

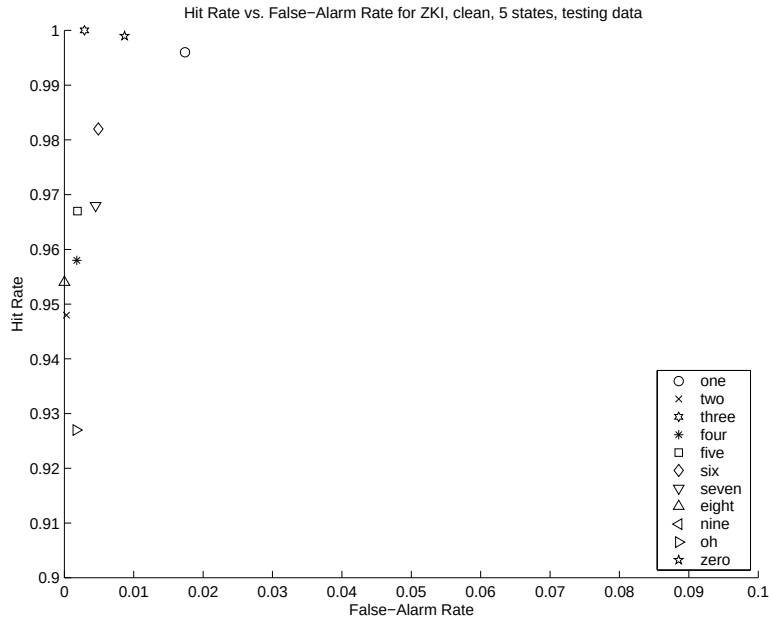


(b)

Figure 4.4 Digit-averaged HR vs. SNR for the 9-state testing and training cases. In both, we see that the two BW-trained models achieved the highest digit-averaged hit rate. The ZKI-trained models perform the next best and the ZKNI-trained models perform the worst. (a) Test data. (b) Training data.

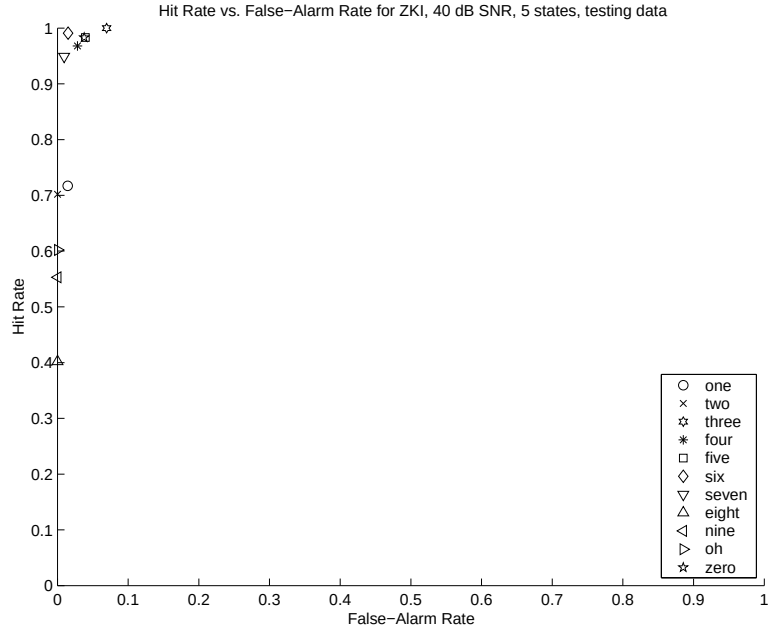


(a)

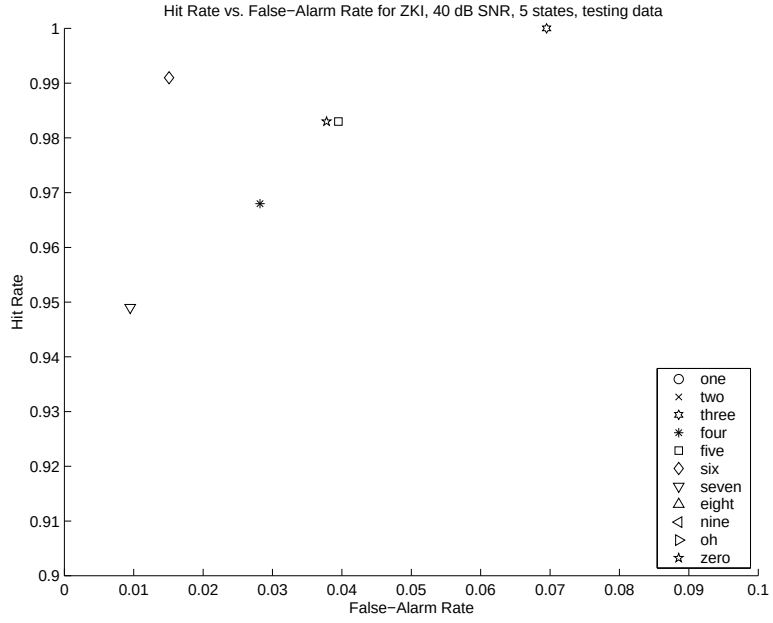


(b)

Figure 4.5 HR-FAR characteristics for each digit in the 5-state, ZKI model condition, with testing data. (a) Clean. (b) Clean, zoomed in.

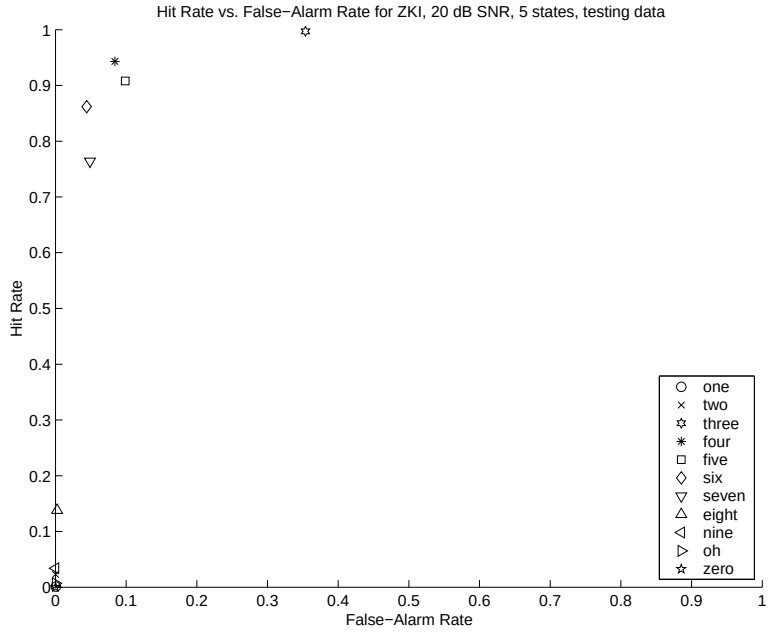


(a)

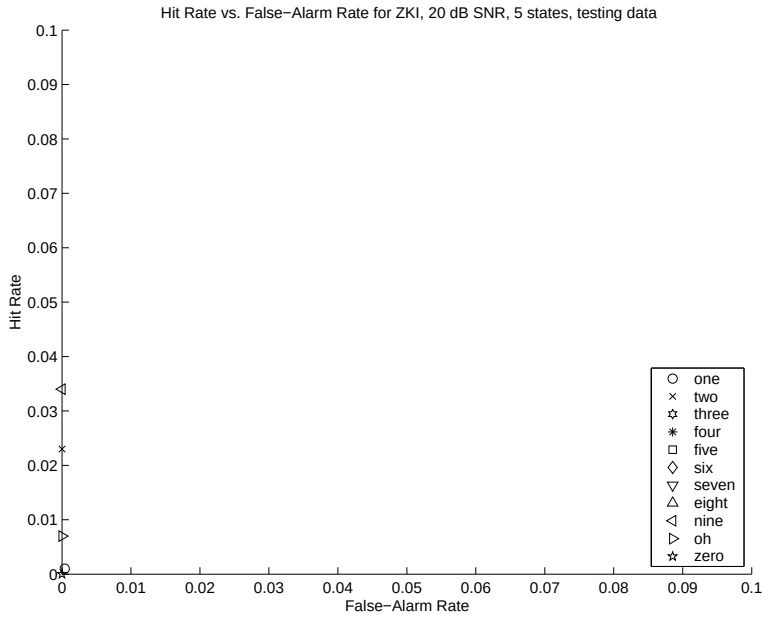


(b)

Figure 4.6 HR-FAR characteristics for each digit in the 5-state, ZKI model condition, with testing data. (a) 40-dB SNR. (b) 40-dB SNR, zoomed in.

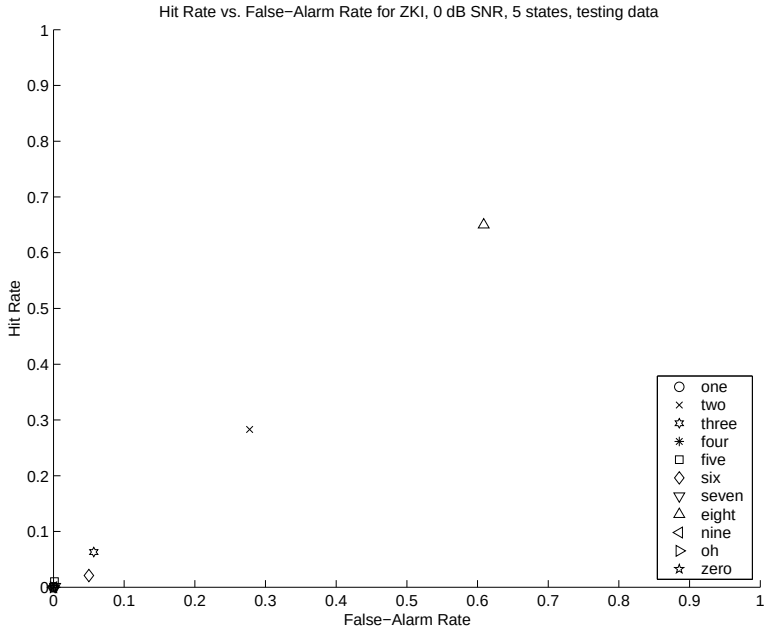


(a)

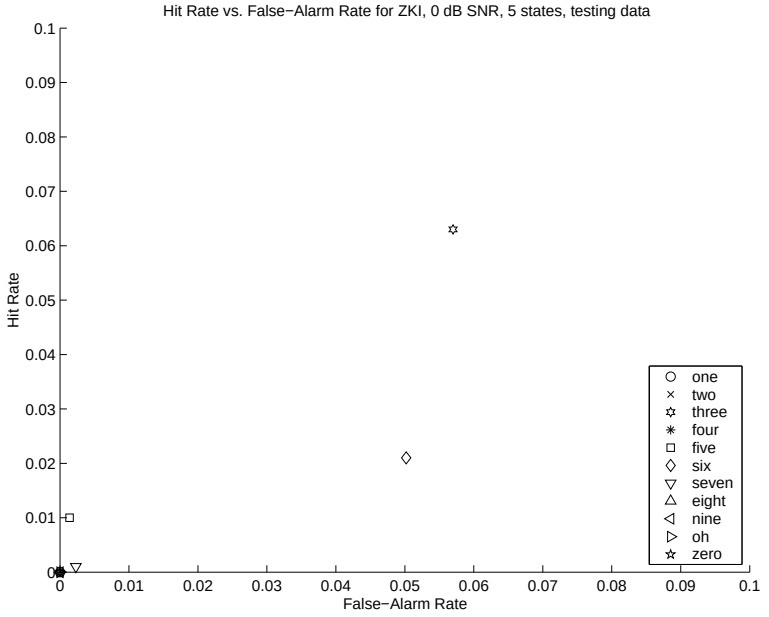


(b)

Figure 4.7 HR-FAR characteristics for each digit in the 5-state, ZKI model condition, with testing data. (a) 20-dB SNR. (b) 20-dB SNR, zoomed in.

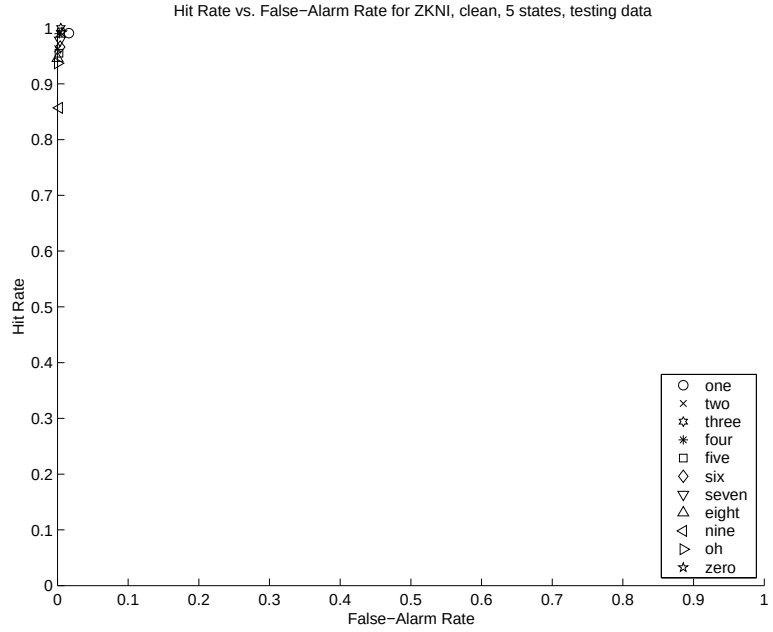


(a)

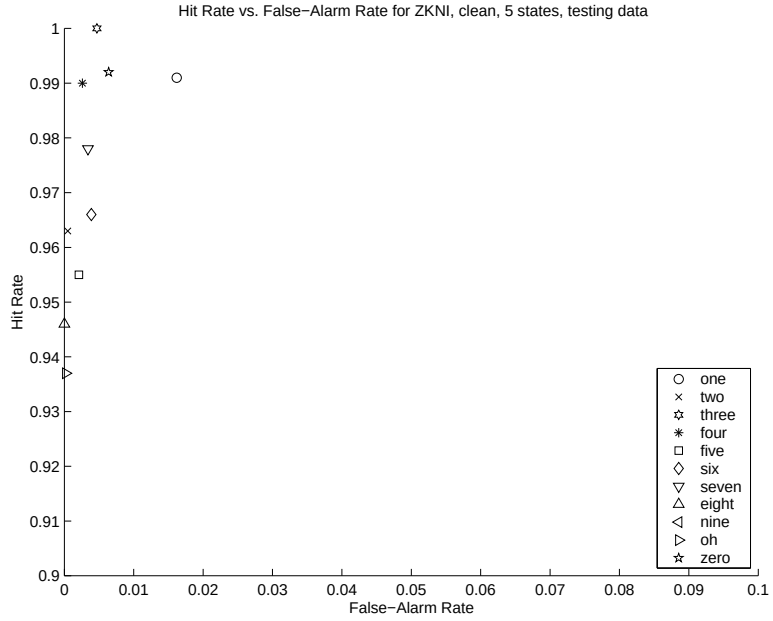


(b)

Figure 4.8 HR-FAR characteristics for each digit in the 5-state, ZKI model condition, with testing data. (a) 0-dB SNR. (b) 0-dB SNR, zoomed in.

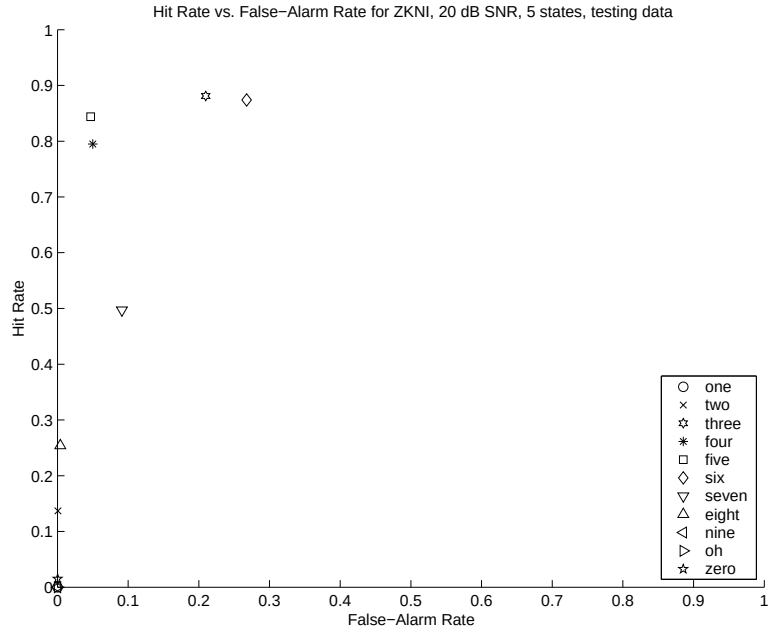


(a)

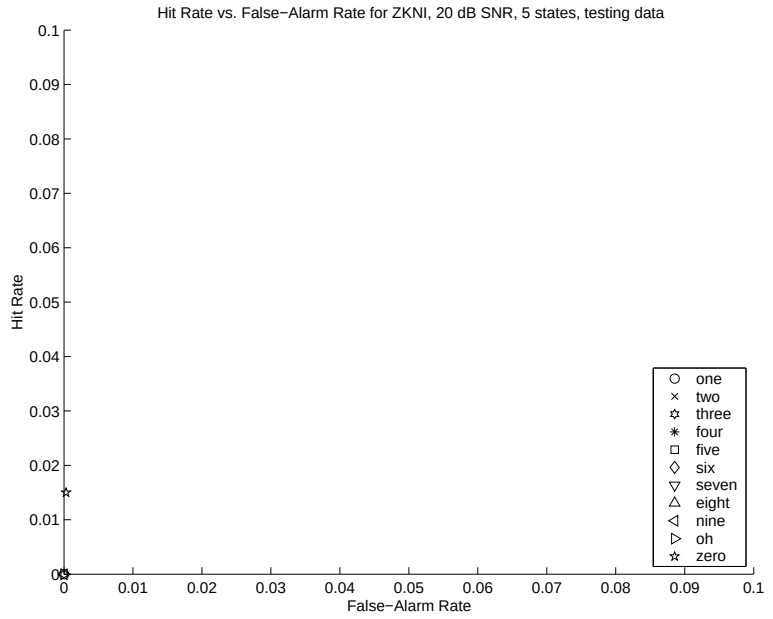


(b)

Figure 4.9 HR-FAR characteristics for each digit in the 5-state, ZKNI model condition, with testing data. (a) Clean. (b) Clean, zoomed in.

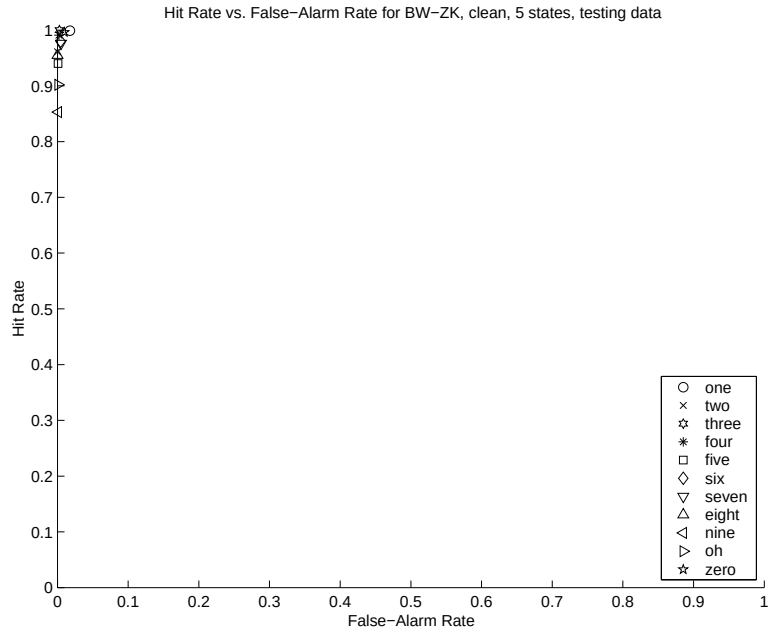


(a)

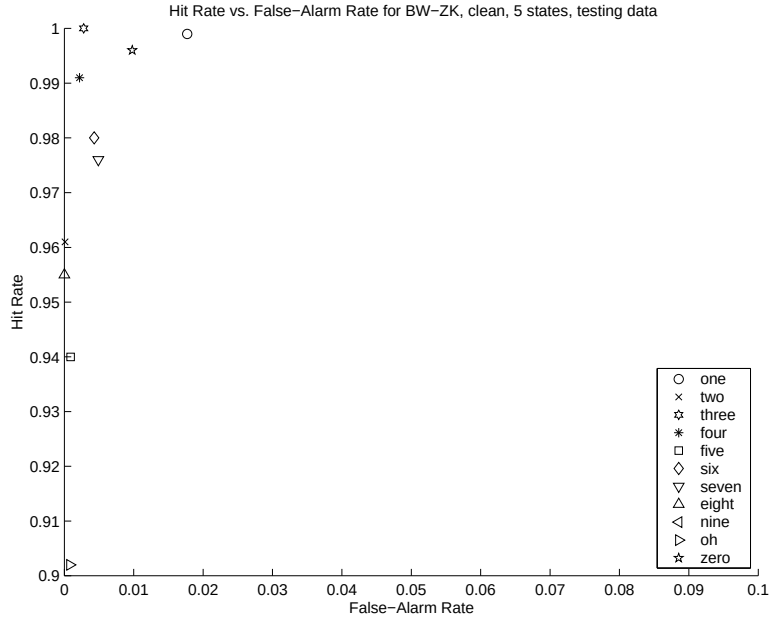


(b)

Figure 4.10 HR-FAR characteristics for each digit in the 5-state, ZKNI model condition, with testing data. (a) 20-dB SNR. (b) 20-dB SNR, zoomed in.

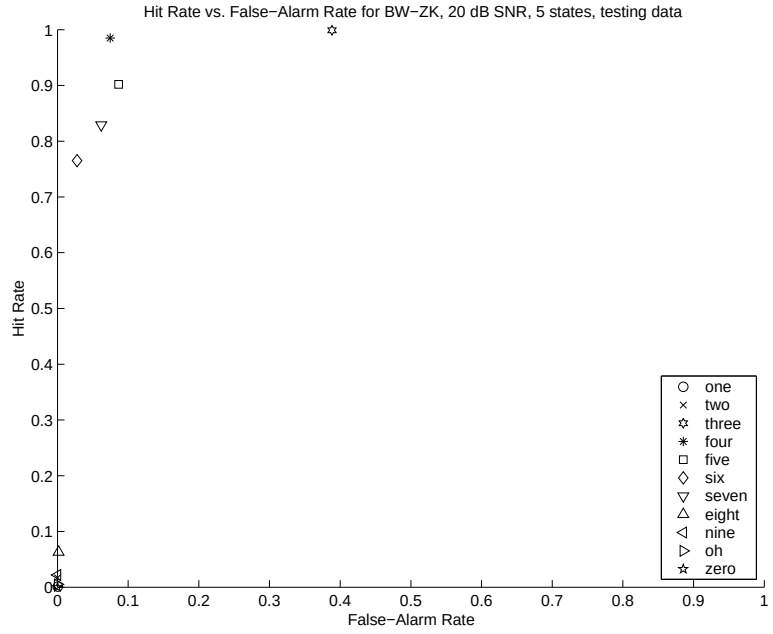


(a)

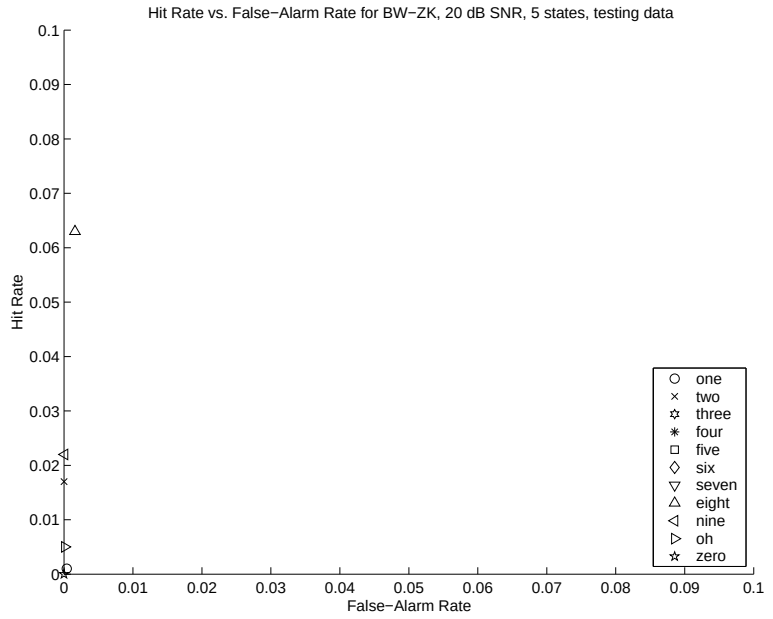


(b)

Figure 4.11 HR-FAR characteristics for each digit in the 5-state, BW-ZK model condition, with testing data. (a) Clean. (b) Clean, zoomed in.

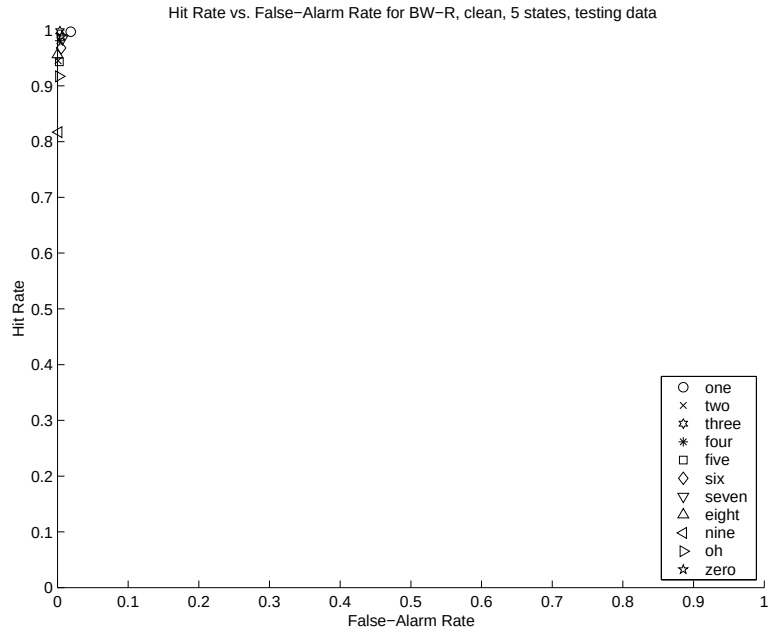


(a)

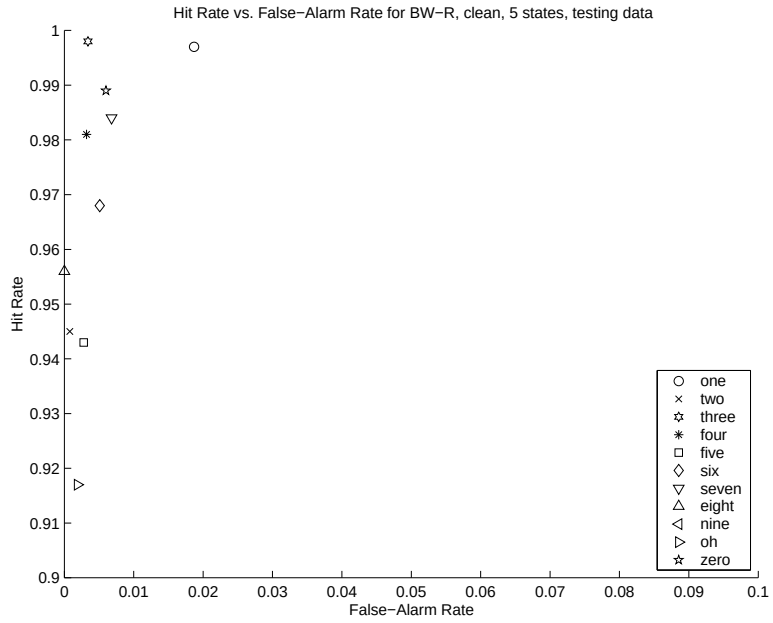


(b)

Figure 4.12 HR-FAR characteristics for each digit in the 5-state, BW-ZK model condition, with testing data. (a) 20-dB SNR. (b) 20-dB SNR, zoomed in.

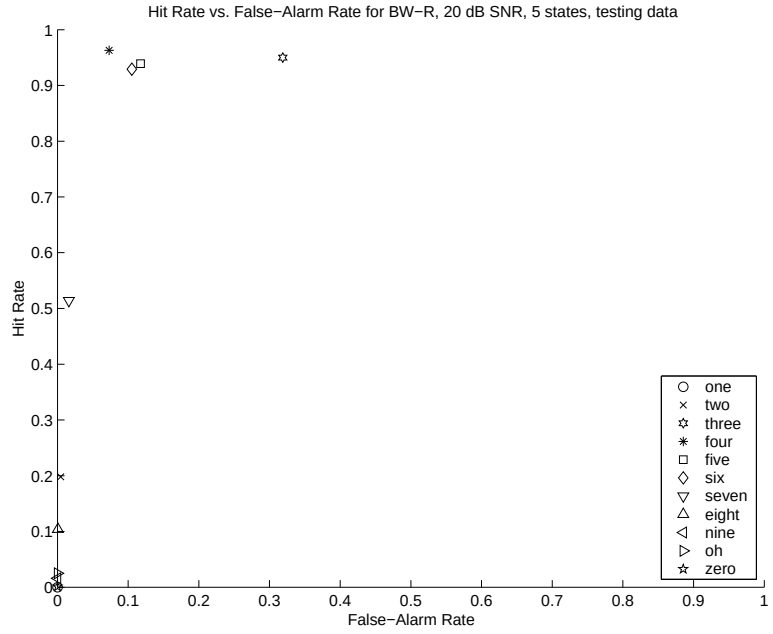


(a)

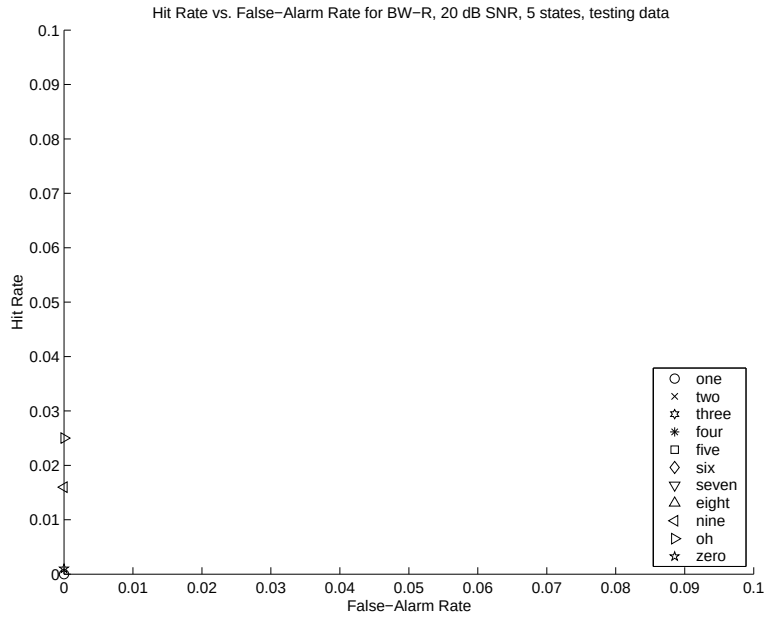


(b)

Figure 4.13 HR-FAR characteristics for each digit in the 5-state, BW-R model condition, with testing data. (a) Clean. (b) Clean, zoomed in.

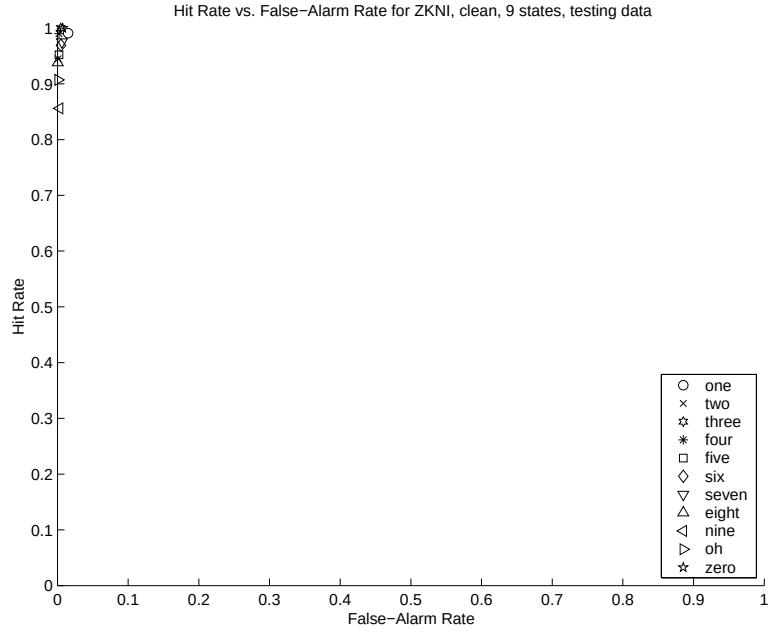


(a)

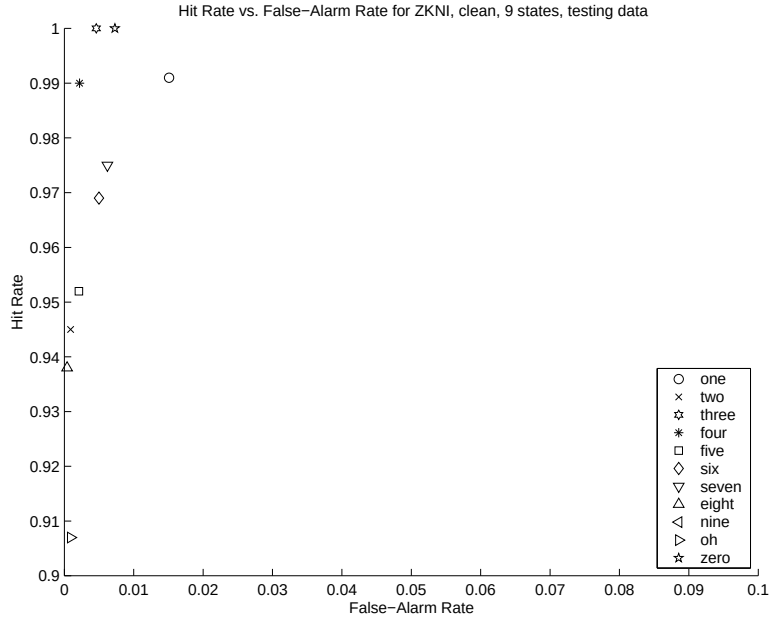


(b)

Figure 4.14 HR-FAR characteristics for each digit in the 5-state, BW-R model condition, with testing data. (a) 20-dB SNR. (b) 20-dB SNR, zoomed in.

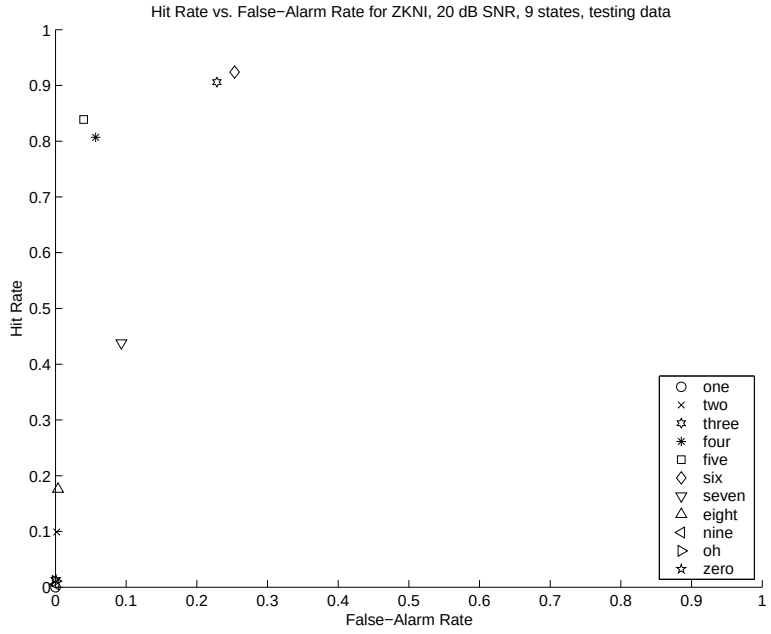


(a)

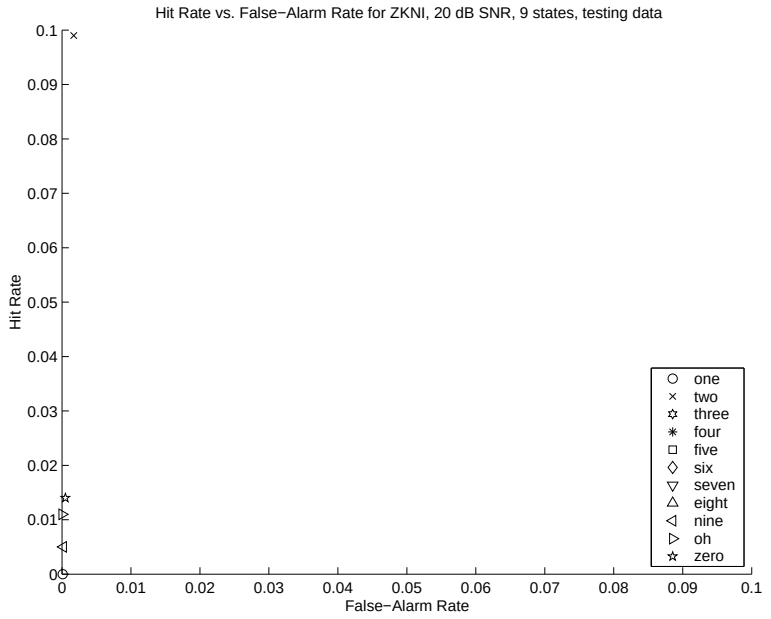


(b)

Figure 4.15 HR-FAR characteristics for each digit in the 9-state, ZKNI model condition, with testing data. (a) Clean. (b) Clean, zoomed in.

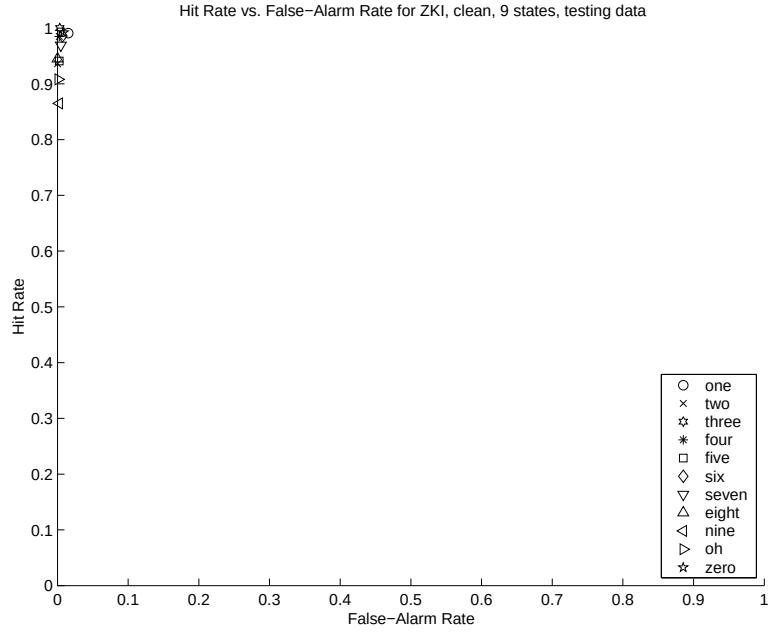


(a)

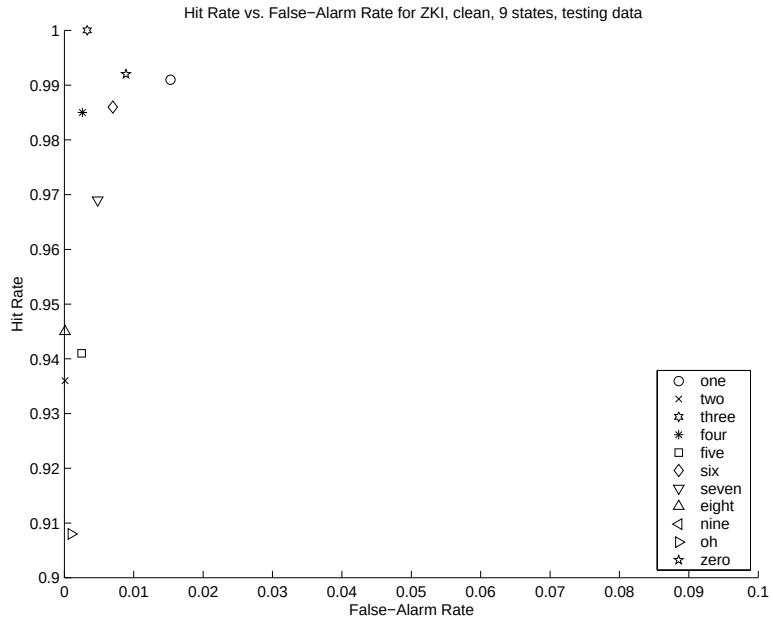


(b)

Figure 4.16 HR-FAR characteristics for each digit in the 9-state, ZKNI model condition, with testing data. (a) 20-dB SNR. (b) 20-dB SNR, zoomed in.

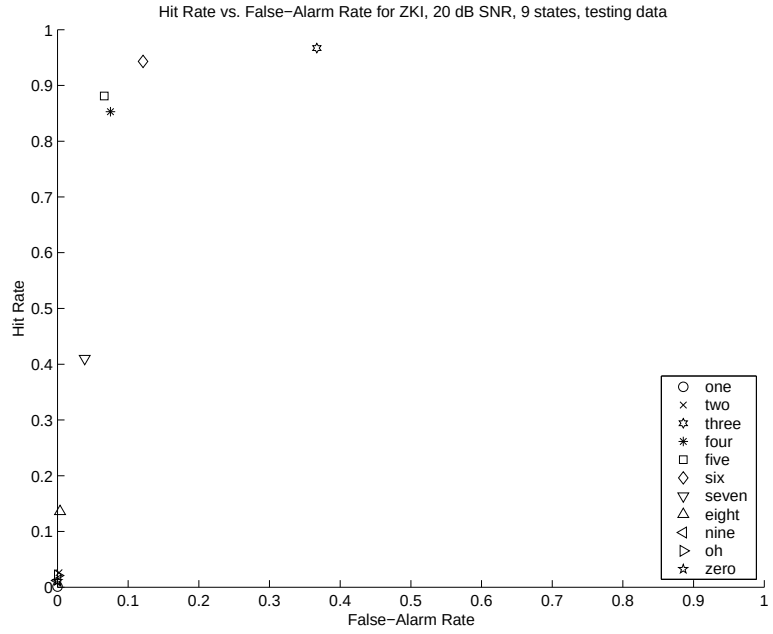


(a)

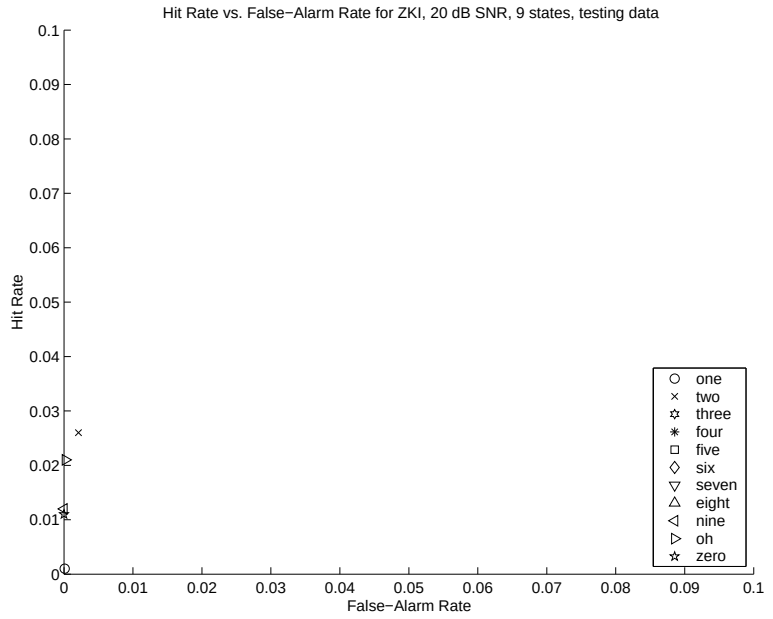


(b)

Figure 4.17 HR-FAR characteristics for each digit in the 9-state, ZKI model condition, with testing data. (a) Clean. (b) Clean, zoomed in.

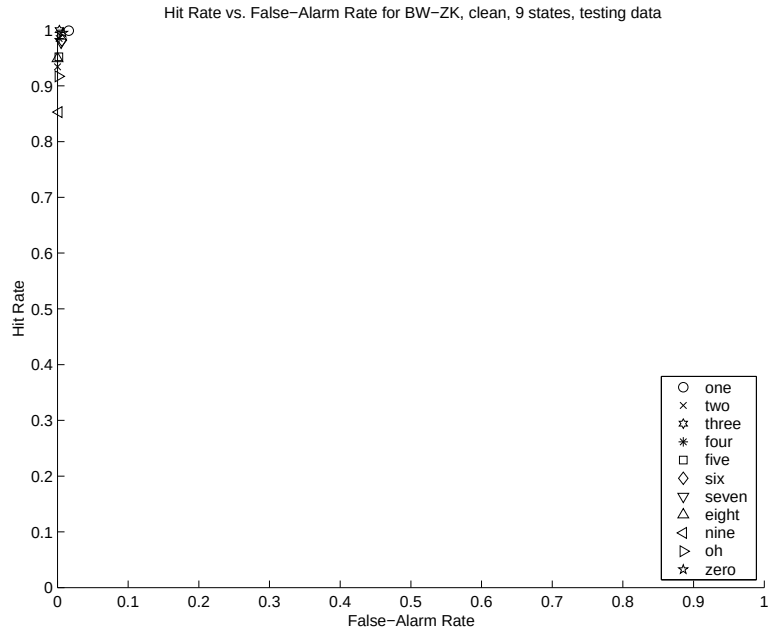


(a)

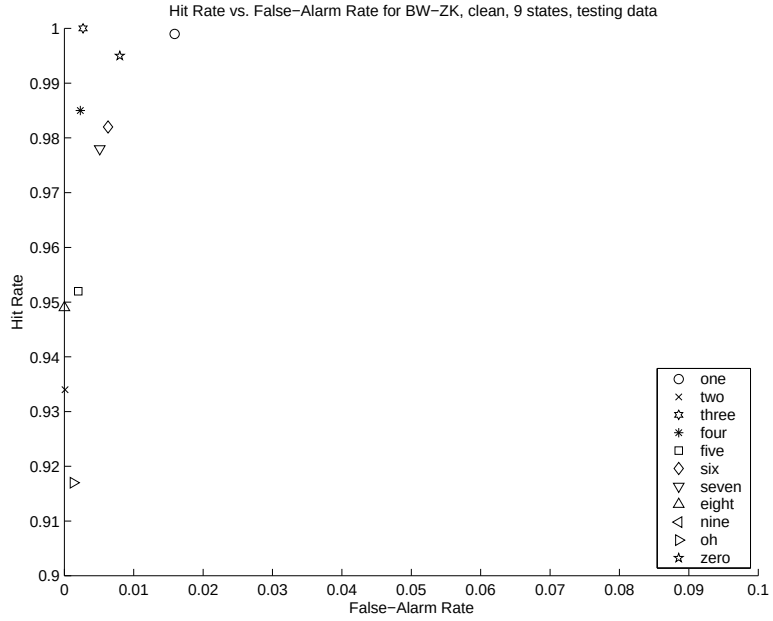


(b)

Figure 4.18 HR-FAR characteristics for each digit in the 9-state, ZKI model condition, with testing data. (a) 20-dB SNR. (b) 20-dB SNR, zoomed in.

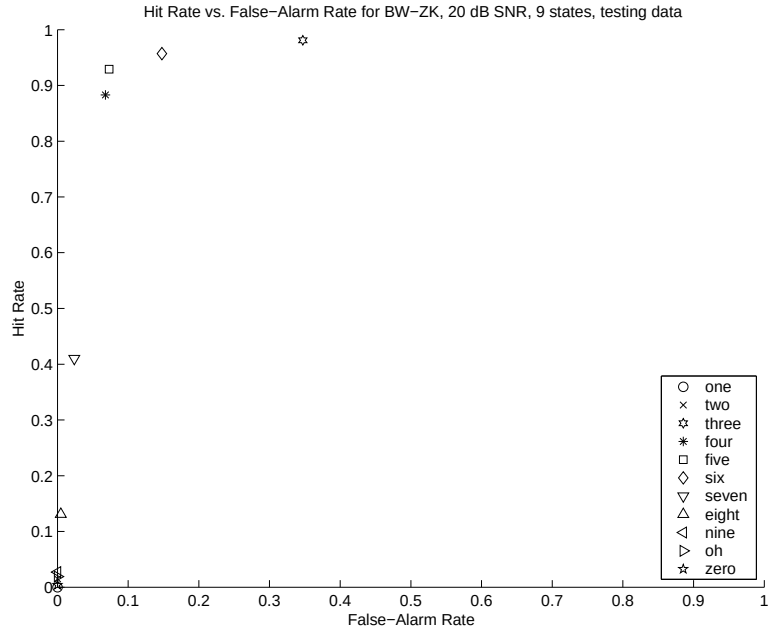


(a)

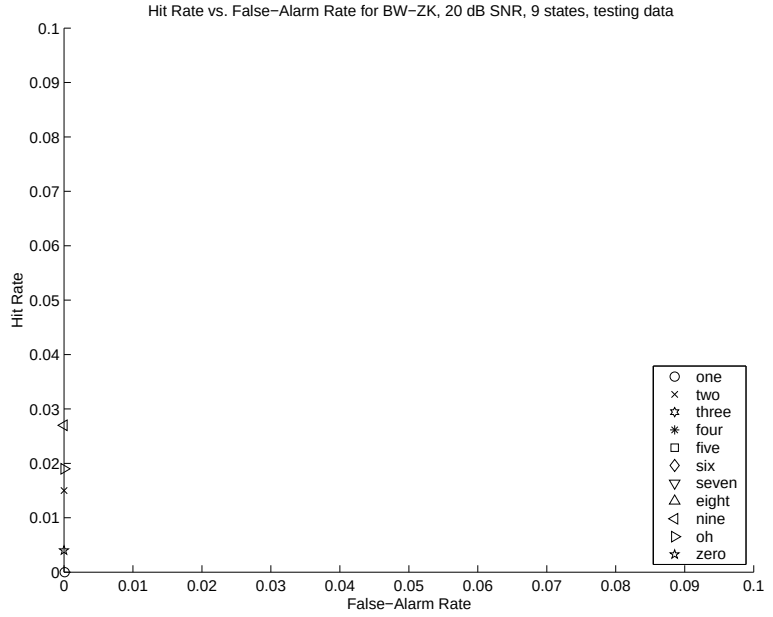


(b)

Figure 4.19 HR-FAR characteristics for each digit in the 9-state, BW-ZK model condition, with testing data. (a) Clean. (b) Clean, zoomed in.

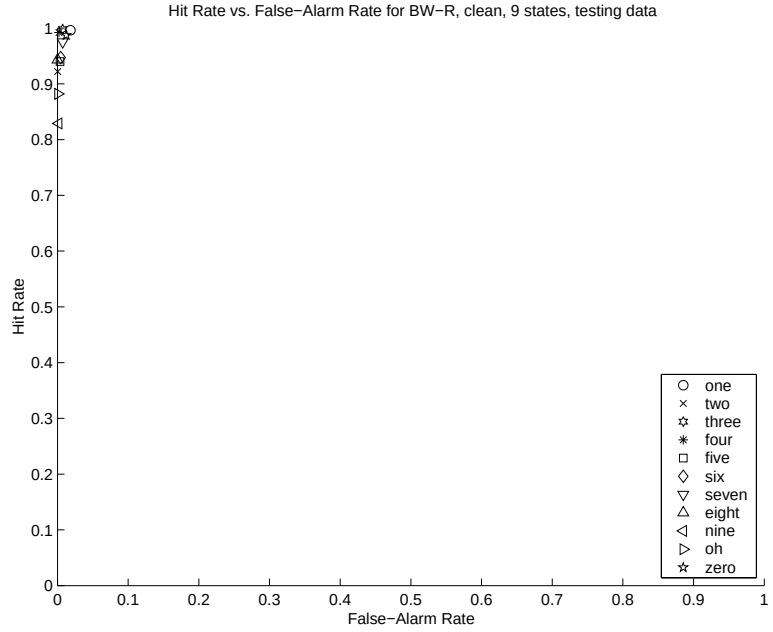


(a)

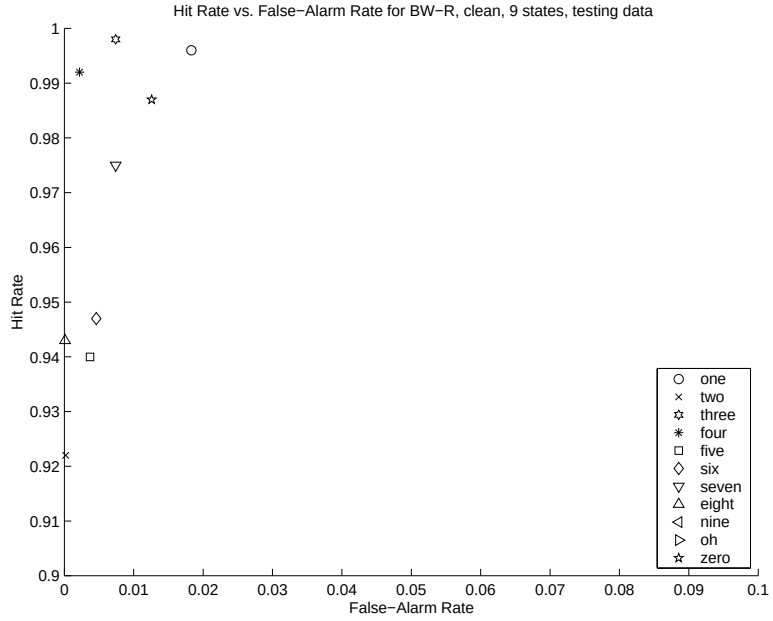


(b)

Figure 4.20 HR-FAR characteristics for each digit in the 9-state, BW-ZK model condition, with testing data. (a) 20-dB SNR. (b) 20-dB SNR, zoomed in.

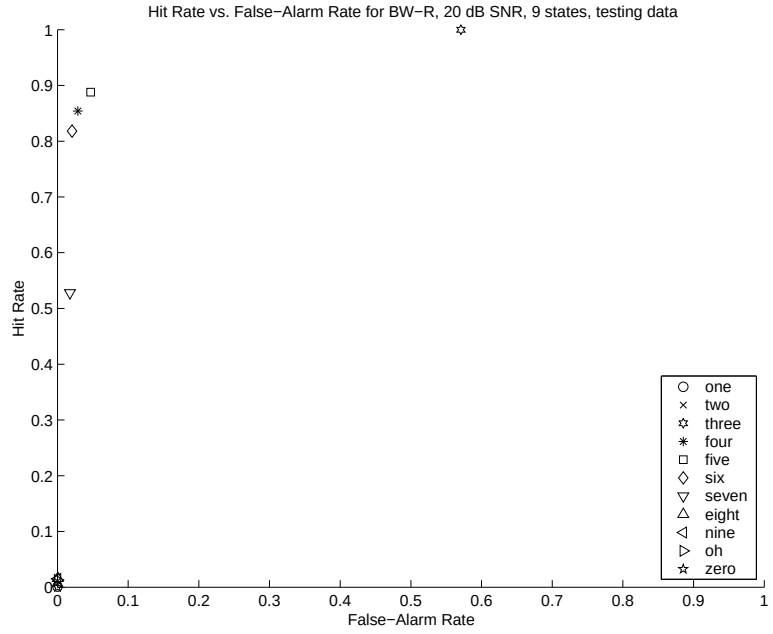


(a)

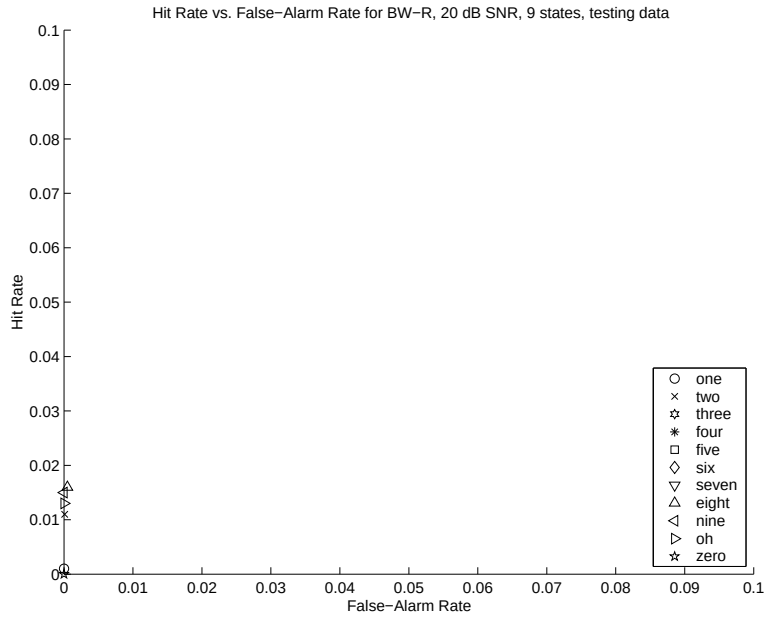


(b)

Figure 4.21 HR-FAR characteristics for each digit in the 9-state, BW-R model condition, with testing data. (a) Clean. (b) Clean, zoomed in.

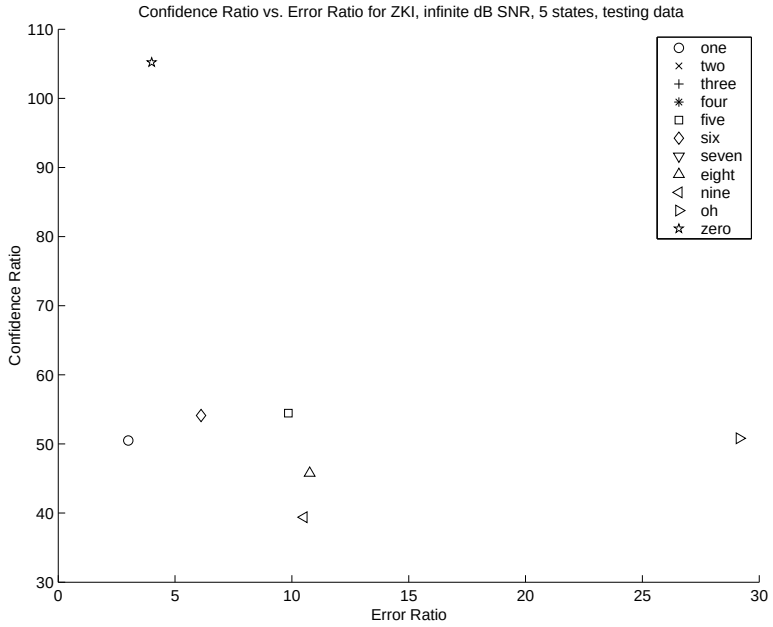


(a)

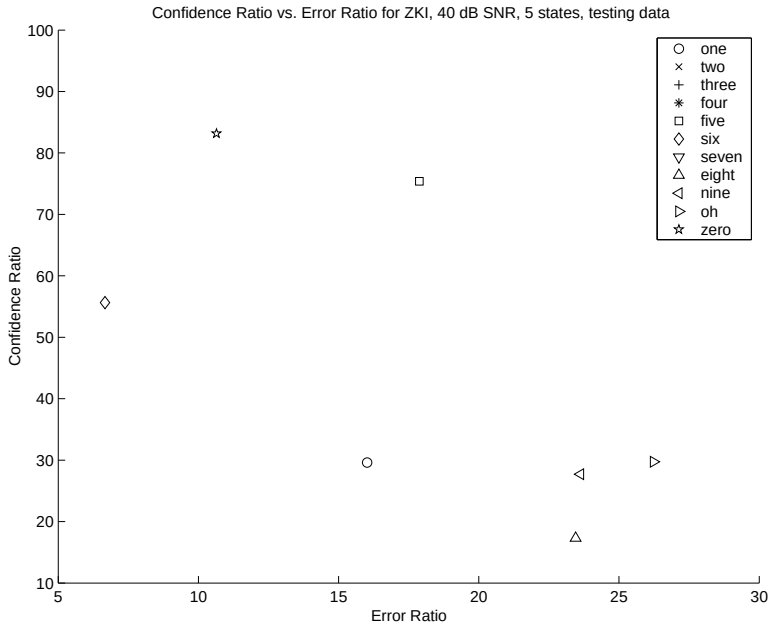


(b)

Figure 4.22 HR-FAR characteristics for each digit in the 9-state, BW-R model condition, with testing data. (a) 20-dB SNR. (b) 20-dB SNR, zoomed in.

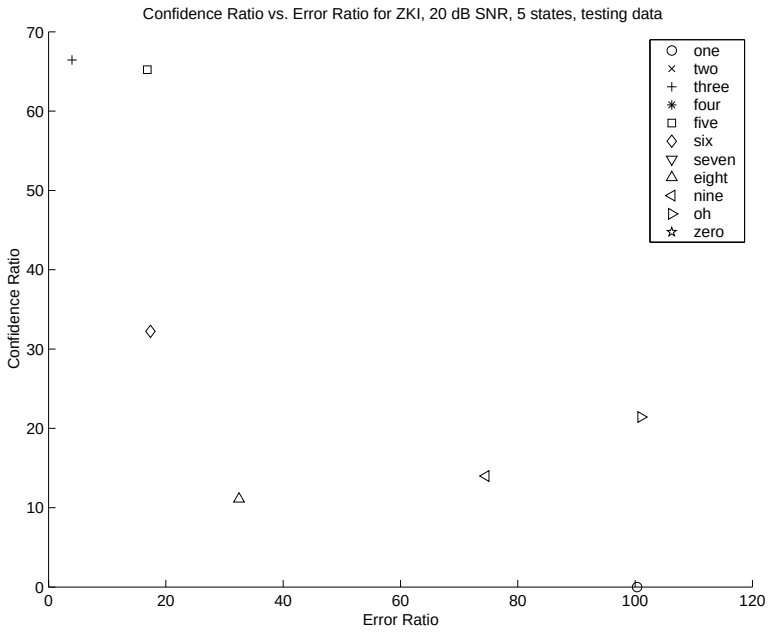


(a)

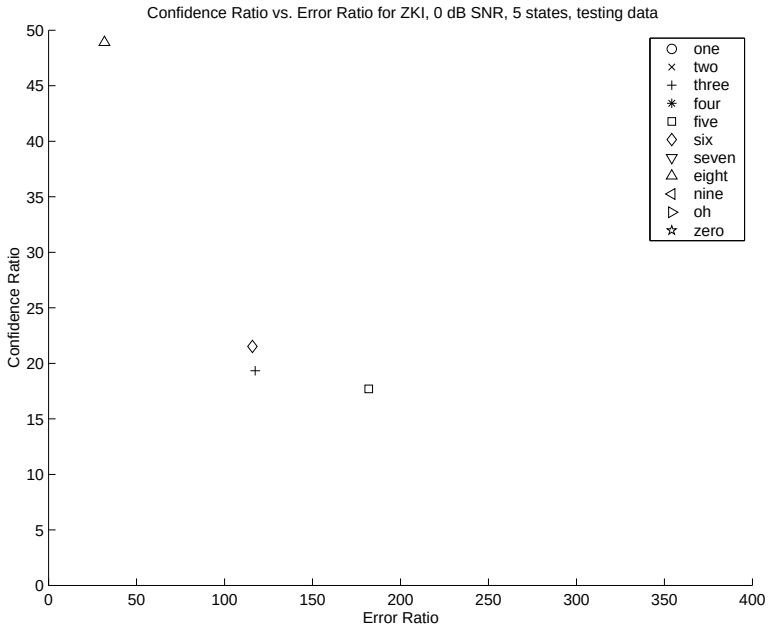


(b)

Figure 4.23 CR-ER characteristics for each digit in the 5-state, ZKI condition with testing data. (a) Clean. (b) 40-dB SNR.

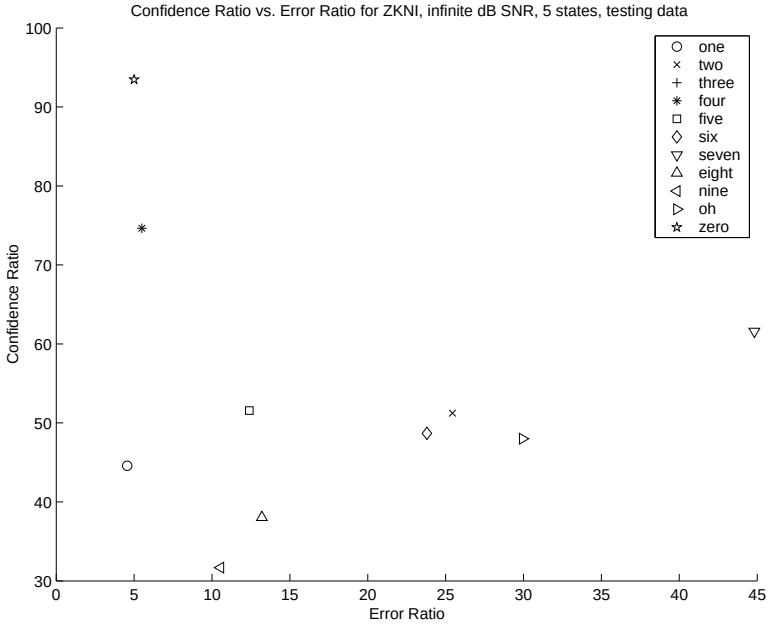


(a)

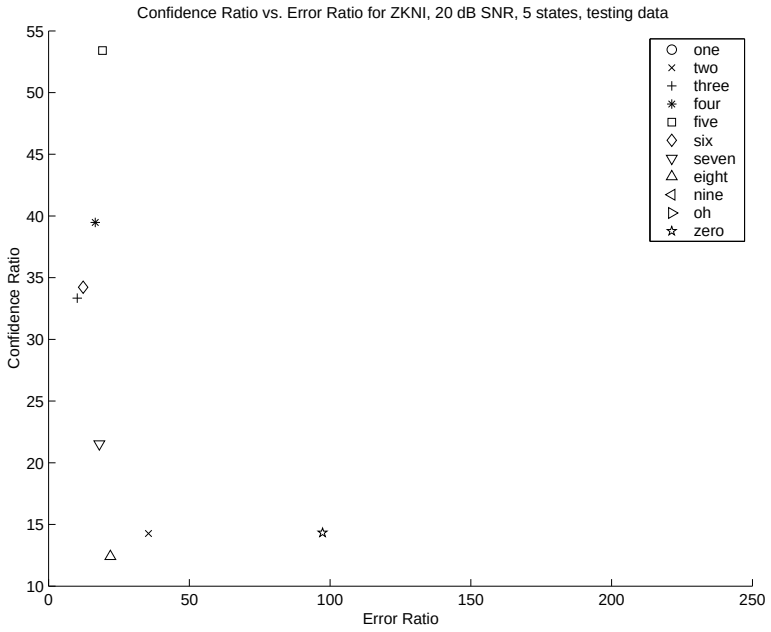


(b)

Figure 4.24 CR-ER characteristics for each digit in the 5-state, ZKI condition with testing data. (a) 20-dB SNR. (b) 0-dB SNR.

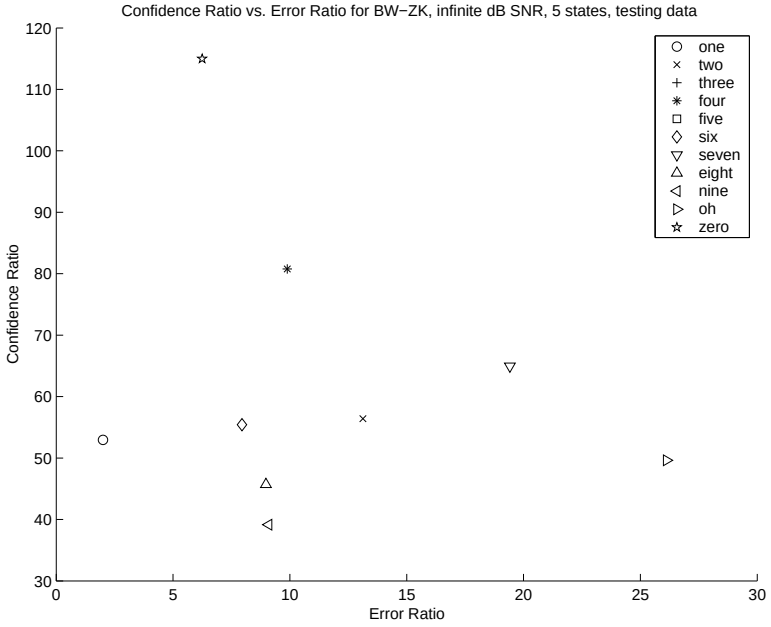


(a)

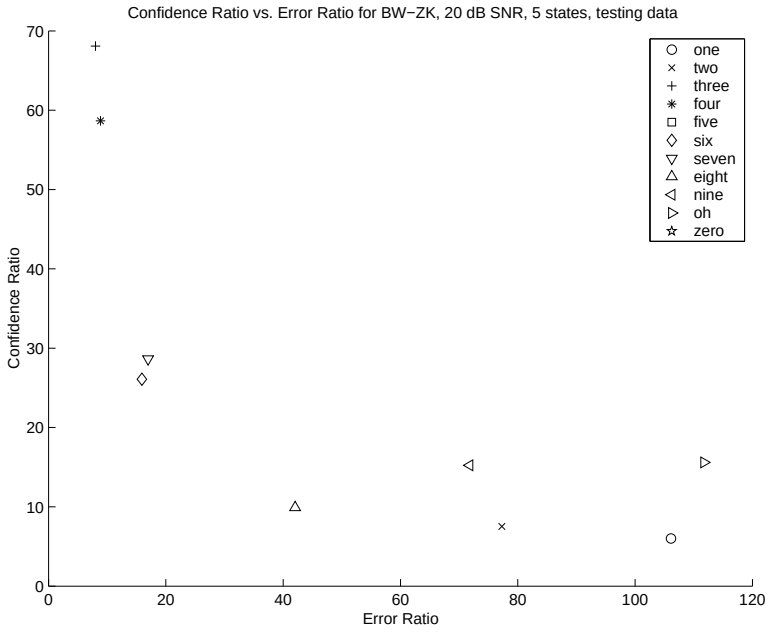


(b)

Figure 4.25 CR-ER characteristics for each digit in the 5-state, ZKNI condition with testing data. (a) Clean. (b) 20-dB SNR.

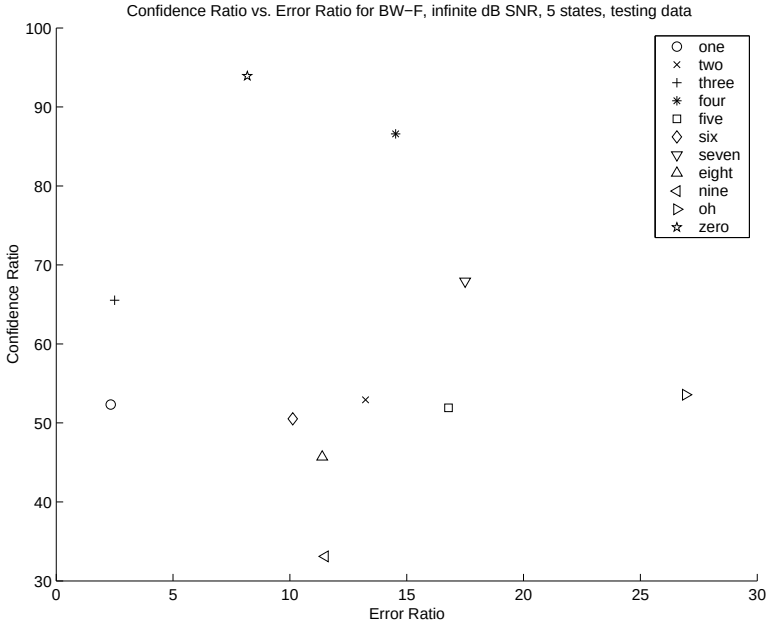


(a)

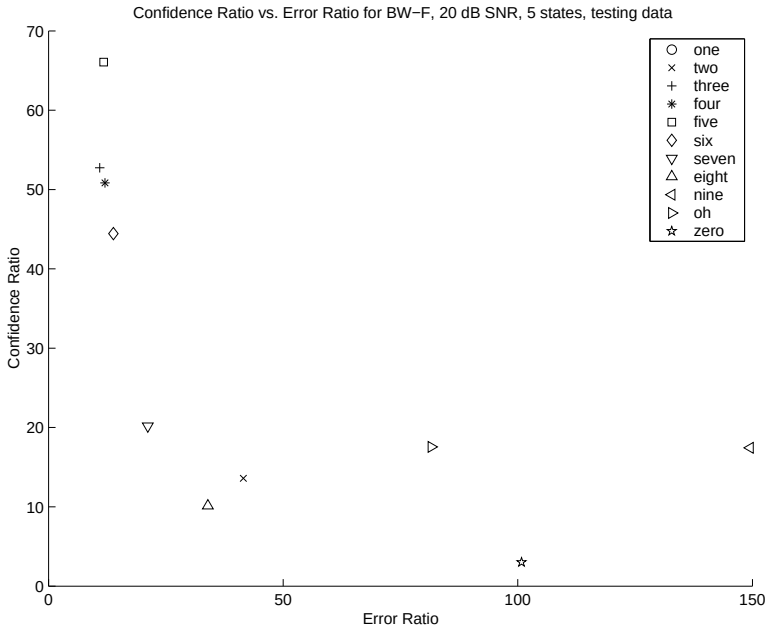


(b)

Figure 4.26 CR-ER characteristics for each digit in the 5-state, BW-ZK condition with testing data. (a) Clean. (b) 20-dB SNR.

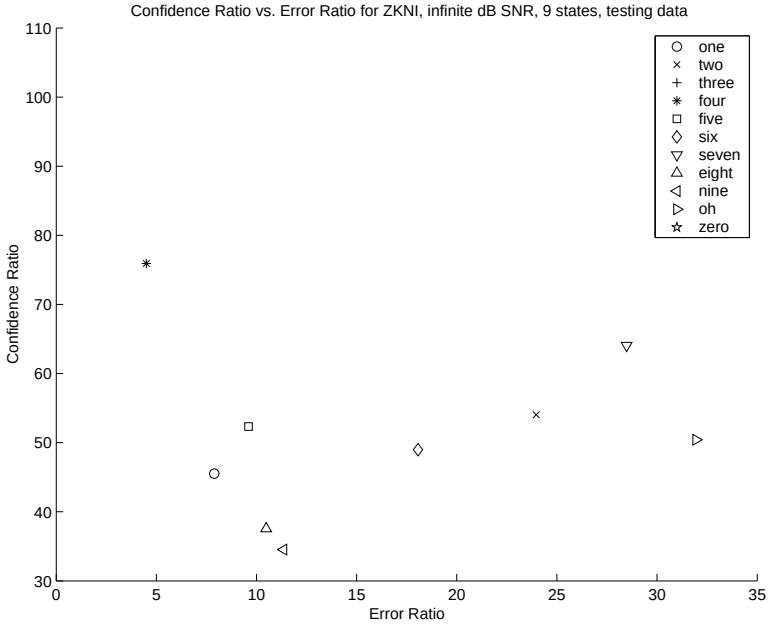


(a)

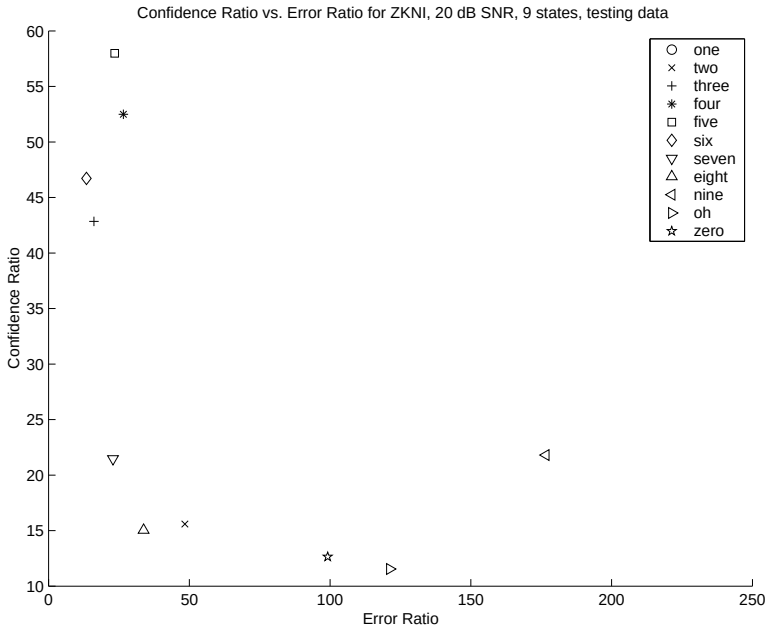


(b)

Figure 4.27 CR-ER characteristics for each digit in the 5-state, BW-F condition with testing data. (a) Clean. (b) 20-dB SNR.

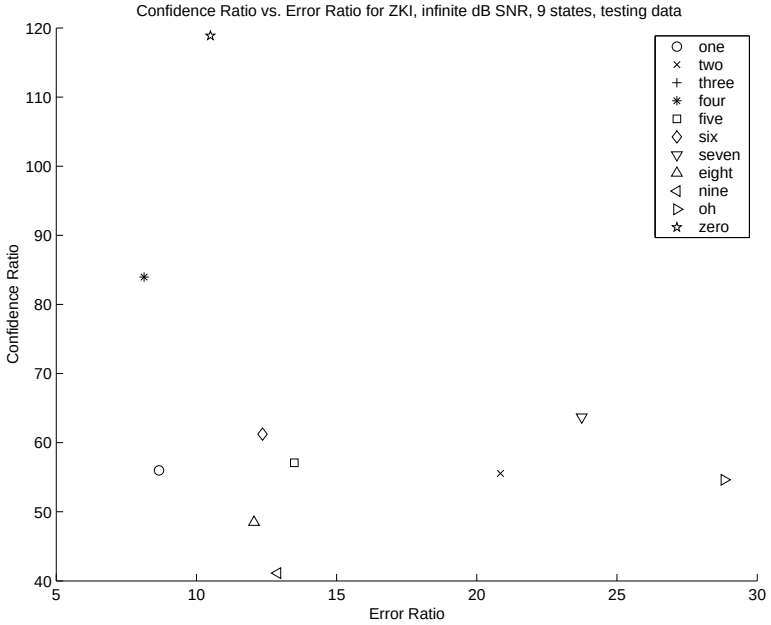


(a)

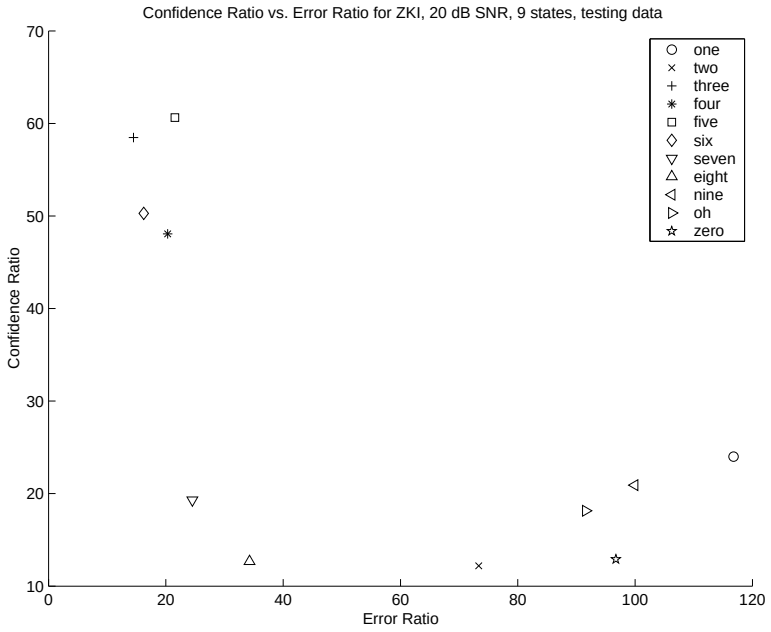


(b)

Figure 4.28 CR-ER characteristics for each digit in the 9-state, ZKNI condition with testing data. (a) Clean. (b) 20-dB SNR.

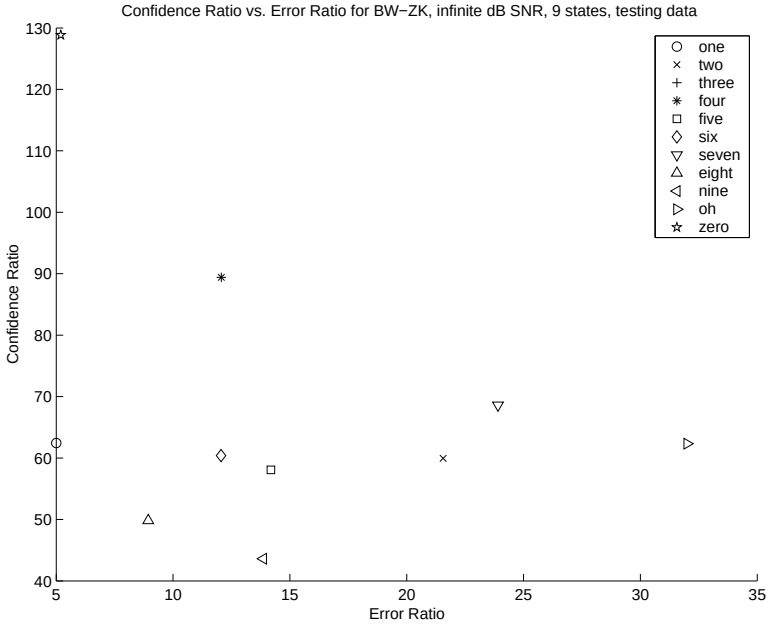


(a)

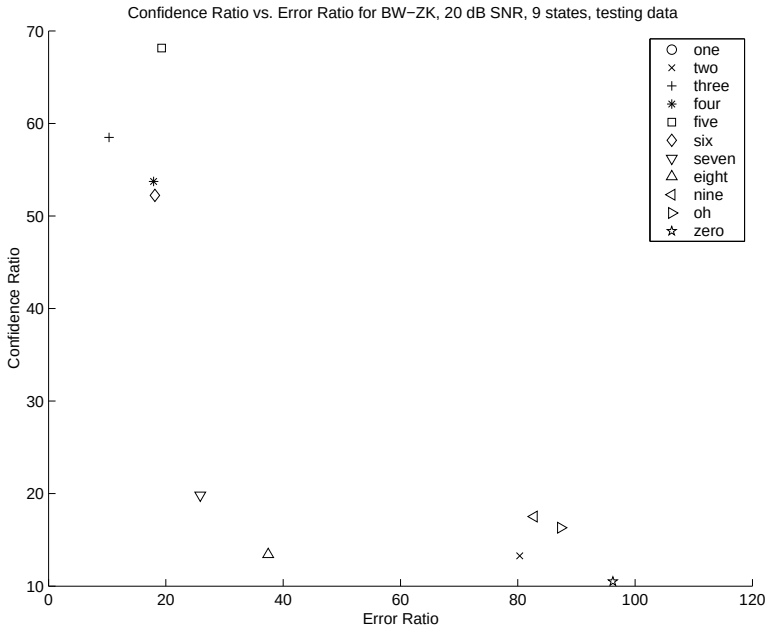


(b)

Figure 4.29 CR-ER characteristics for each digit in the 9-state, ZKI condition with testing data. (a) Clean. (b) 20-dB SNR.

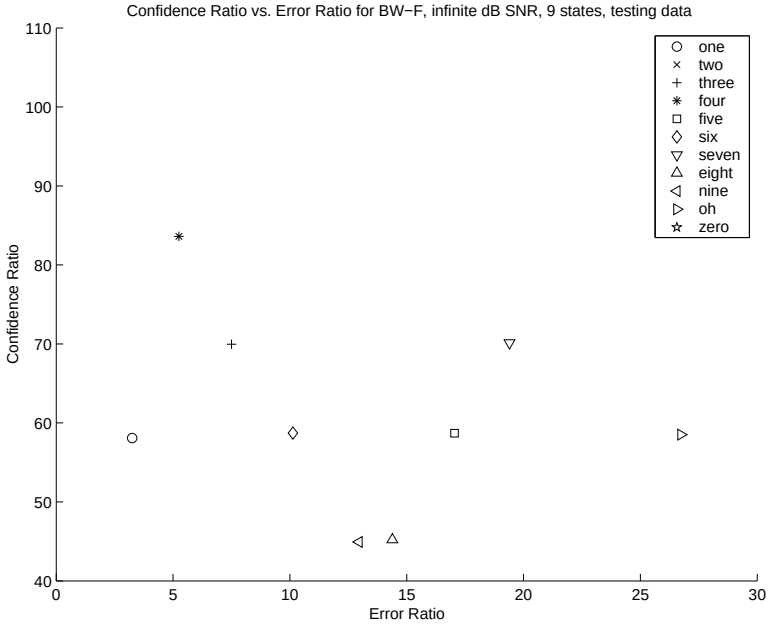


(a)

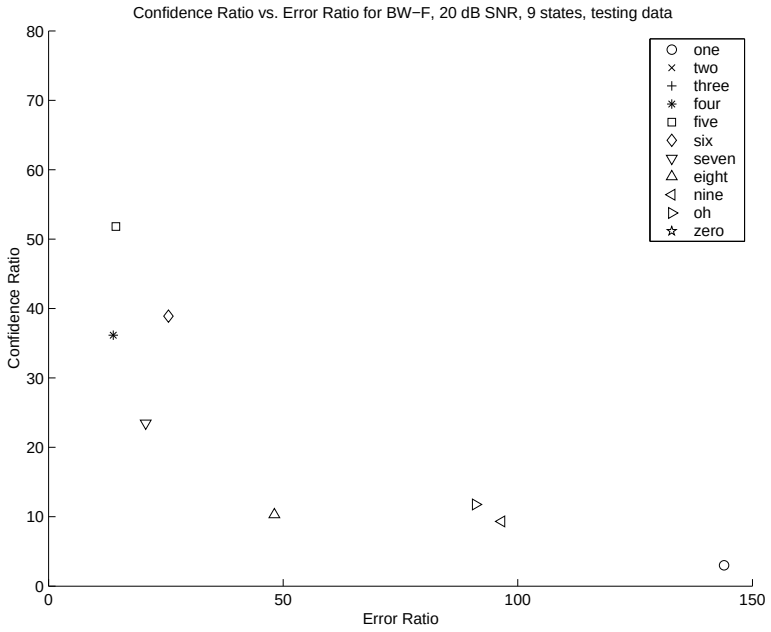


(b)

Figure 4.30 CR-ER characteristics for each digit in the 9-state, BW-ZK condition with testing data. (a) Clean. (b) 20-dB SNR.

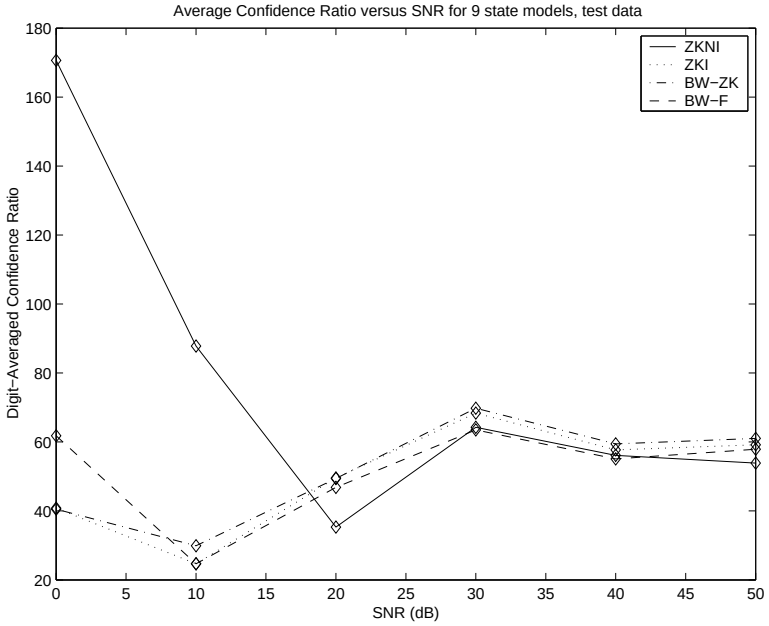


(a)

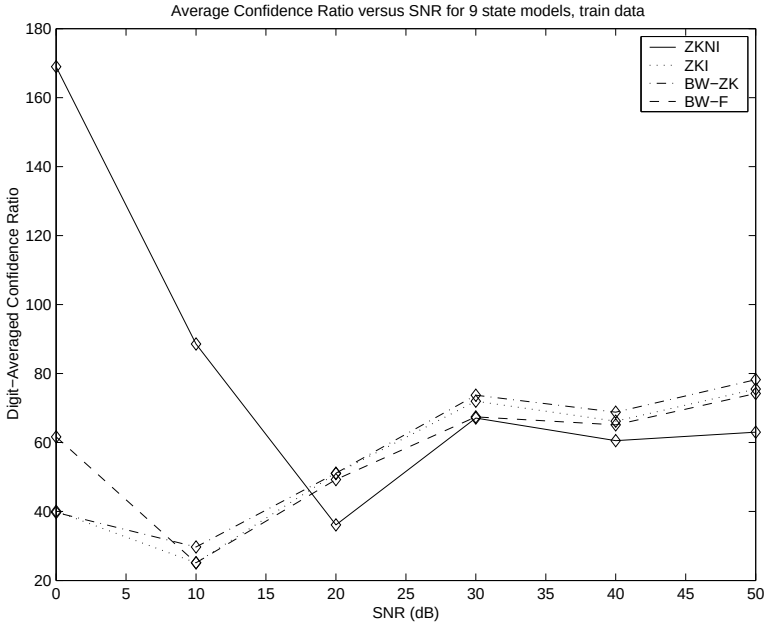


(b)

Figure 4.31 CR-ER characteristics for each digit in the 9-state, BW-R condition with testing data. (a) Clean. (b) 20-dB SNR.

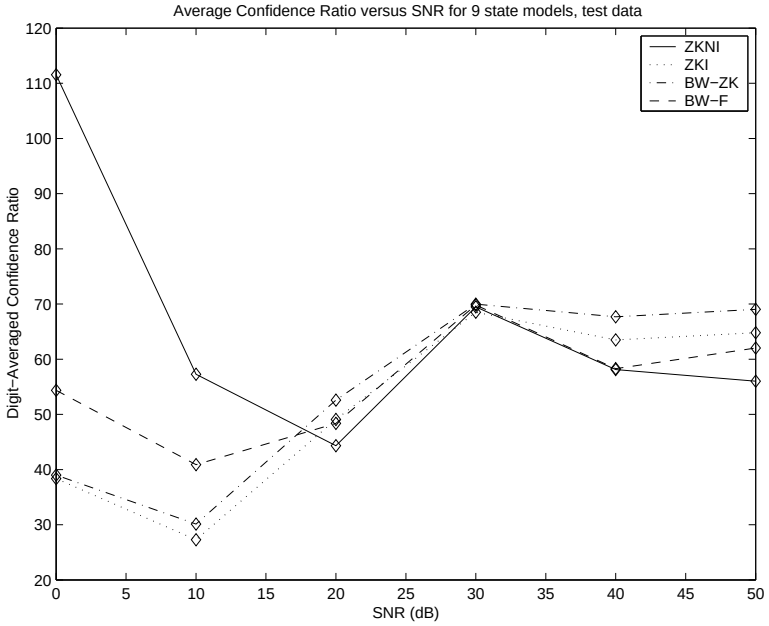


(a)

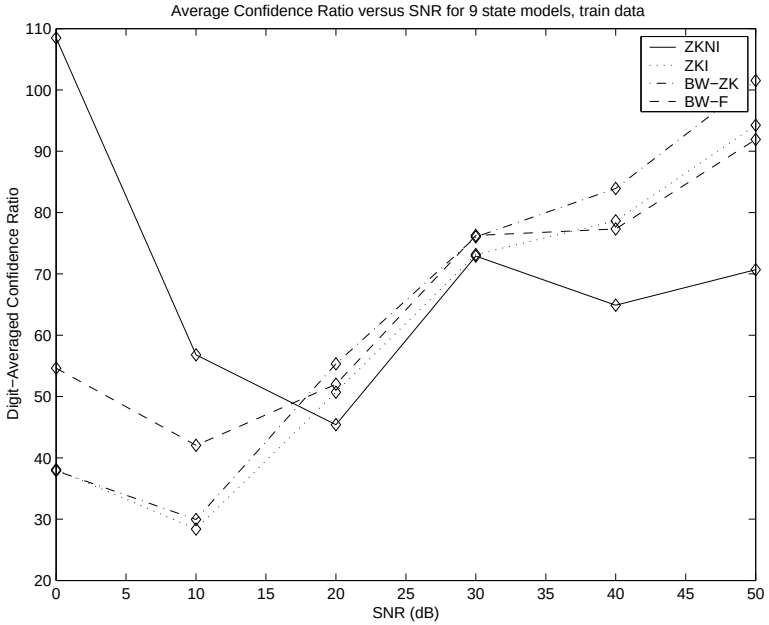


(a)

Figure 4.32 The digit-averaged CR for 5-state models as a function of the SNR. The CR falls with the SNR except in the 0-dB and 10-dB SNR cases. (a) Test data. (b) Training data.

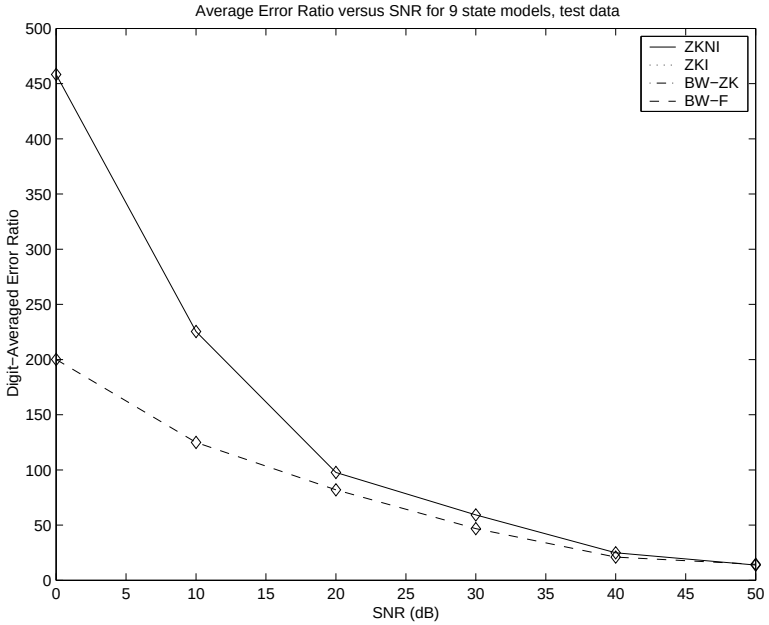


(a)

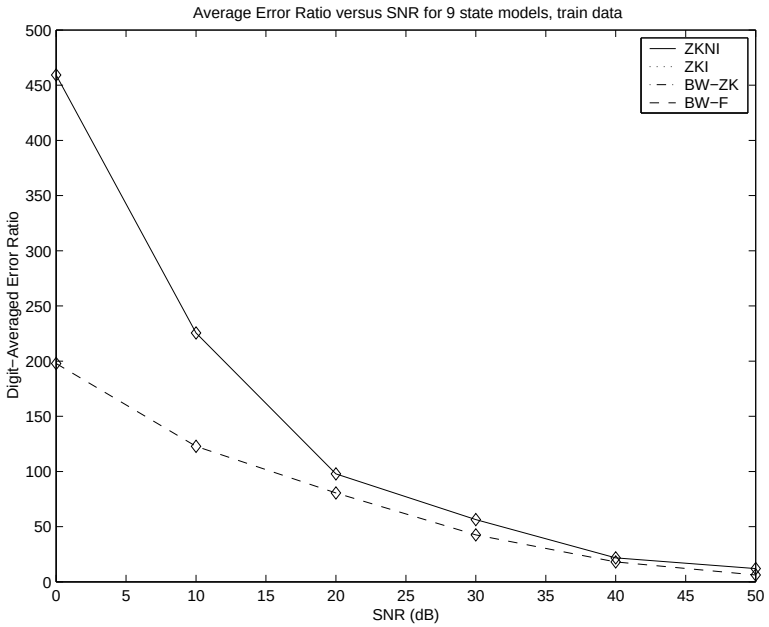


(a)

Figure 4.33 The digit-averaged CR for 9-state models as a function of the SNR. The CR falls with the SNR except in the 0-dB and 10-dB SNR cases. (a) Test data. (b) Training data.

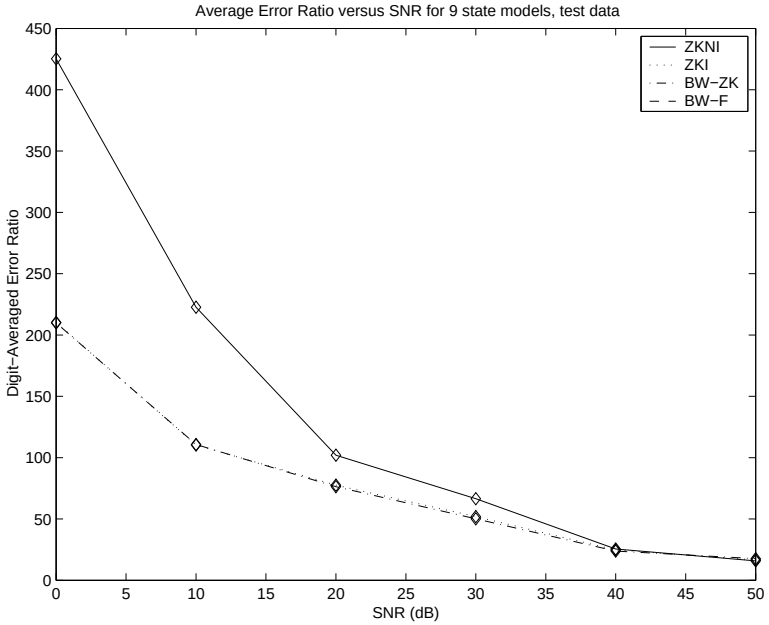


(a)

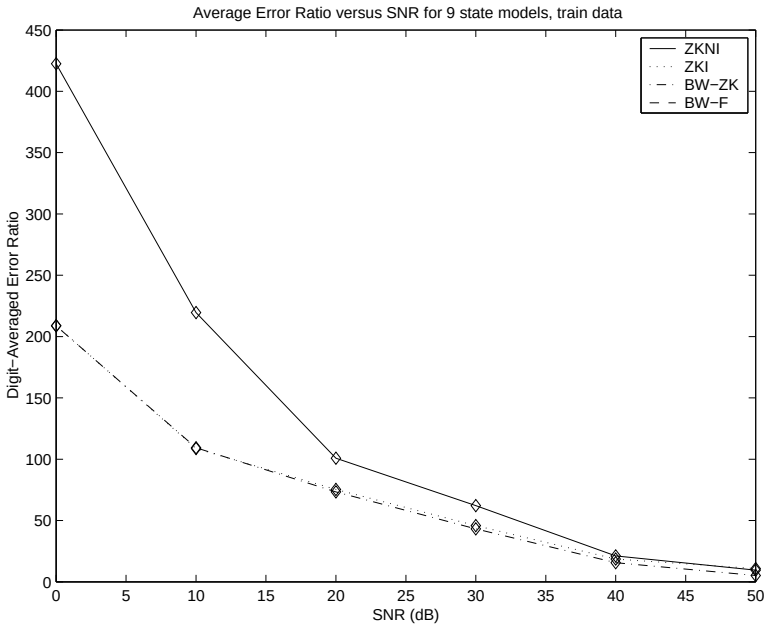


(a)

Figure 4.34 The digit-averaged ER for 5-state models as a function of the SNR. The ER rises as the SNR falls. (a) Test data. (b) Training data.



(a)



(a)

Figure 4.35 The digit-averaged ER for 9-state models as a function of the SNR. The ER rises as the SNR falls. (a) Test data. (b) Training data.

CHAPTER 5

CONCLUSIONS

To compare the performance of the training methods discussed in this thesis, we trained hidden Markov models (HMMs) corresponding to each digit using the training methods, and attempted to identify the digits at varying SNR conditions. It appears that Zhang and Kuo's iterative method (ZKI), as described and implemented in Chapter 3, is a reasonable suboptimal alternative to the Baum-Welch (BW) algorithm for training HMMs. The models trained by ZKI perform comparably with the those trained by BW, with substantial savings in computational expense. Zhang and Kuo's noniterative method (ZKNI), a one-pass training algorithm, also trains remarkably accurately with very little computational expense.

Although our results exhibited very low noise immunity, our analysis showed that this was a result of the mel-frequency cepstral (MFCC) features, and not of the training method. The HMMs trained with the BW algorithm achieved slightly higher recognition rates and confidence ratios than the models trained with ZKI. The models trained by BW did, however, exhibit the same degradation in recognition rate as a function of SNR.

Because of the noise performance we encountered, we believe that it will be beneficial to study the feasibility of using ZKI and ZKNI in a more noise-robust environment. There is already much research devoted to using HMMs for noisy speech [16, 17, 18]. It may be necessary to apply gain-adaption and noise-compensation to ZKNI and ZKI for the purpose of training HMMs with better noise immunity. There is also a possibility of using more robust features than MFCC. There is currently much research devoted to the extraction of

different features better noise immunity than MFCC [19].

Of course, isolated digit recognition represents only one application of HMMs. Although we have shown that ZKNI and ZKI are adequate suboptimal training methods for this application, it may be necessary to perform a similar set of tests on a different application. One such application is continuous speech recognition, where multiple digits or words may be spoken in a single sound file. We can also attempt these training methods in training HMMs to perform scene-classification, where BW-trained HMMs have been used effectively [10].

APPENDIX A

THE BAUM-WELCH ALGORITHM

The Baum-Welch algorithm (also called the forward-backward algorithm) is a modified expectation-maximation (EM) algorithm used for training hidden Markov models [8]. In the EM algorithm, the parameters of a statistical model are iteratively updated until the probability of the training data reaches a local maximum with respect to the model parameters. Because the EM algorithm only guarantees convergence to a local maximum, the initial model becomes an important consideration. This also holds true in the BW algorithm.

The BW algorithm also takes into account the constraints that the HMM parameters being updated must satisfy. The constraints are given in Equations (2.4), (2.6), and (2.10). For continuously distributed HMMs the BW algorithm is constrained by Equation (2.12) instead of (2.10). The probability $P(\mathcal{X}|\Phi)$ increases with each iteration of the BW algorithm. However, the probability only increases to a local maximum with respect to the parameters of the HMM. The initial parameters are therefore very important in the BW algorithm.

The basic procedure of the Baum-Welch algorithm is to iteratively change the HMM parameters $\Phi = (\boldsymbol{\pi}, \boldsymbol{A}, \boldsymbol{B})$ to maximize $P(\mathcal{X}|\Phi)$. For the case of continuously distributed observations, the HMM parameters include $\Phi = (\boldsymbol{\pi}, \boldsymbol{A}, \boldsymbol{c}_{jk}, \boldsymbol{\mu}_{jk}, \boldsymbol{\Sigma}_{jk})$.

A.1 The EM Algorithm

Let us denote by $\hat{\Phi}$ the updated version of Φ after an iteration of the BW algorithm. Since

$$P(\mathcal{X},\ \mathcal{S} \mid \hat{\Phi}) \ = \ P(\mathcal{S} \mid \mathcal{X},\ \hat{\Phi})\ P(\mathcal{X} \mid \hat{\Phi}), \tag{A.1}$$

we can write

$$\log P(\mathcal{X}, \mathcal{S}|\hat{\Phi}) \ = \ \log P(\mathcal{S}|\mathcal{X}, \hat{\Phi}) \ + \ \log P(\mathcal{X}|\hat{\Phi}). \tag{A.2}$$

Taking the expectation of Equation (A.2) yields

$$\log P(\mathcal{X}|\hat{\Phi}) = E_{\Phi}[\log P(\mathcal{S}, \mathcal{X}|\hat{\Phi})]_{\mathcal{S}|\mathcal{X}} - E_{\Phi}[\log P(\mathcal{S}|\mathcal{X}, \hat{\Phi})]_{\mathcal{S}|\mathcal{X}}, \tag{A.3}$$

since

$$\begin{aligned} E_{\Phi}[\log P(\mathcal{X}|\hat{\Phi})]_{\mathcal{S}|\mathcal{X}} \ &= \ \sum_{all\ \mathcal{S}} P(\mathcal{S}|\mathcal{X}, \Phi) \log P(\mathcal{X}|\hat{\Phi}), \\ &= \ \log P(\mathcal{X}|\hat{\Phi}). \end{aligned} \tag{A.4}$$

To simplify this expression, we express the first and second terms on the right-hand side by the functions $Q(\Phi, \hat{\Phi})$ and $H(\Phi, \hat{\Phi})$, respectively, as

$$Q(\Phi, \hat{\Phi}) \ = \ \sum_{all\ \mathcal{S}} P(\mathcal{S}|\mathcal{X}, \Phi) \log P(\mathcal{S}, \mathcal{X}|\hat{\Phi}), \tag{A.5}$$

$$H(\Phi, \hat{\Phi}) \ = \ \sum_{all\ \mathcal{S}} P(\mathcal{S}|\mathcal{X}, \Phi) \log P(\mathcal{S}|\mathcal{X}, \hat{\Phi}). \tag{A.6}$$

Now, the training data is the most likely if $Q(\Phi, \hat{\Phi})$ is maximized while $H(\Phi, \hat{\Phi})$ is minimized. Suppose $\hat{\Phi}$ is chosen so that $Q(\Phi, \hat{\Phi}) \geq Q(\Phi, \Phi)$. Then,

$$\log P(\mathcal{X}|\hat{\Phi}) \geq \log P(\mathcal{X}|\Phi) \tag{A.7}$$

because

$$H(\Phi, \hat{\Phi}) \leq H(\Phi, \Phi). \tag{A.8}$$

This can be proved by using the nonnegativity of the Kullback-Leibler distance [20], which is given as

$$D(p||q) \;=\; \sum_x p(x) \frac{q(x)}{p(x)} \tag{A.9}$$

and using the Jensen's inequality given as

$$E\{f(\mathcal{X})\} \geq f(E[\mathcal{X}\ ) \tag{A.10}$$

for a convex function $f()$ and a random variable X.

At each iteration, the BW algorithm chooses the $\hat{\Phi}$ that maximizes $Q(\Phi, \hat{\Phi})$ and then iterates. The four steps are as follows:

1. Initiation: Choose an initial Φ.
2. Expectation: Compute $Q(\Phi, \hat{\Phi})$ based on Φ.
3. Maximization: Compute $\hat{\Phi} \;=\; \arg\max_{\hat{\Phi}} Q(\Phi, \hat{\Phi})$
4. Iterate from Step 2.

A.2 The Backwards Probability

Using the BW algorithm requires introducing a new intermediate variable. The *backwards probability* $\beta_t(i)$ is the probability

$$\beta_t(i) \;=\; P(\mathcal{X}_{t+1}^T | s_t = i, \Phi). \tag{A.11}$$

This is the probability of generating the observations $\mathcal{X}_{t+1}^T$ from state i at time t. The probability $\beta_t(i)$ is initialized at time T and computed in reverse time:

Initialization:

$$\beta_T(i) \;=\; \frac{1}{N}, \quad 1 \le i \le N. \tag{A.12}$$

Induction:

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij}\; b_{jX_{t+1}} \beta_{t+1}(j), \quad t = T-1, T-2, ..., 1,\; 1 \le i \le N. \tag{A.13}$$

A.3 Implementation

The BW algorithm computes a new model $\hat{\Phi}$ from intermediate variables $\gamma_t(i,j)$ and $\zeta_t(i,j)$. The intermediate probability $\gamma_t(i,j)$ is defined as the probability of going from state i at time $t-1$ to state j at time t given the observation sequence $\mathcal{X}_1^T$ and the model Φ:

$$\begin{aligned} \gamma_t(i,j) &= P(\mathcal{S}_{t-1} = i, \mathcal{S}_t = j | \mathcal{X}_1^T, \Phi), \\ &= \frac{P(\mathcal{S}_{t-1} = i, \mathcal{S}_t = j, \mathcal{X}_1^T | \Phi)}{P(\mathcal{X}_1^T | \Phi)}, \\ &= \frac{\alpha_{t-1}(i) a_{ij} b_j(X_t) \beta_t(j)}{\sum_{k=1}^{N} \alpha_T(k)}. \end{aligned} \tag{A.14}$$

For continuously distributed observations, $\zeta_t(j,k)$ is used to recalculate mixture means and variances:

$$\begin{aligned} \zeta_t(j,k) &= \frac{P(\mathcal{X},\; s_t = j,\; X_t = k \mid \Phi)}{P(\mathcal{X} \mid \Phi)}, \\ &= \frac{\sum\limits_{i=1}^{N} \alpha_{t-1}(i)\; a_{ij}\; c_{jk}\; b_{jk}(X_t)}{\sum\limits_{i=1}^{N} \alpha_T(i)}. \end{aligned} \tag{A.15}$$

Here, $b_{jk}(X_t)$ refers to the probability of the feature vector X_t occuring within the normal (Gaussian) distribution with mean $\boldsymbol{\mu}_{jk}$ and variance $\boldsymbol{\Sigma}_{jk}$.

Using this, the reestimation equations for a_{ij} and b_{jk} are given by

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T} \gamma_t(i,j)}{\sum_{t=1}^{T} \sum_{k=1}^{N} \gamma_t(i,k)}, \tag{A.16}$$

$$\hat{b}_{jk} = \frac{\sum_{t \in \mathcal{X}_t = o_k} \sum_{t} \gamma_t(i,j)}{\sum_{t=1}^{T} \sum_{t} \gamma_t(i,j)}. \tag{A.17}$$

When using continuously distributed features, $\boldsymbol{\mu}_{jk}$, $\boldsymbol{\Sigma}_{jk}$, and c_{jk} are also restimated during each iteration. These are all functions of $\zeta_t(j,k)$. Note that in this case, the reestimation equation for $\mathbf{B}$ becomes unnecessary. The reestimation equations for $\boldsymbol{\Sigma}_{jk}$, $\boldsymbol{\mu}_{jk}$, and c_{jk} are given by

$$\hat{\boldsymbol{\mu}}_{jk} = \frac{\sum_{t=1}^{T} \zeta_t(j,k) X_t}{\sum_{t=1}^{T} \zeta_t(j,k)}, \tag{A.18}$$

$$\hat{\boldsymbol{\Sigma}}_{jk} = \frac{\sum_{t=1}^{T} \zeta_t(j,k)(X_t - \boldsymbol{\mu}_{jk})(X_t - \boldsymbol{\mu}_{jk})^T}{\sum_{t=1}^{T} \zeta_t(j,k)}, \tag{A.19}$$

$$\hat{c}_{jk} = \frac{\sum_{t=1}^{T} \zeta_t(j,k)}{\sum_{t=1}^{T} \sum_{k=1}^{M} \zeta_t(j,k)}. \tag{A.20}$$

A.4 Implementational Issues

When programming the BW algorithm, there are a number of implementation issues to consider. One such consideration is initialization. Because the BW algorithm converges to a local maximum, the initial Φ becomes an important consideration in implementing the BW algorithm. Another such consideration is the convergence criterion. The BW algorithm does not have a formal stopping point, so it is necessary to choose a convergence criterion that maximizes performance but limits the time it takes to run the BW algorithm.

A.4.1 Initial parameters

Because the BW algorithm is iterative, the performance of the training will be dependent on the initial Φ provided to the BW algorithm. The BW algorithm will simply find a local maximum probability for the training data within the space of Φ. It is not guaranteed that the local maximum $P(\mathcal{X}|\Phi)$ will also be the absolute maximum. Currently, we use Zhang and Kuo's noniterative method to find an initial Φ.

A.4.2 Baum-Welch convergence

The BW algorithm will run infinitely with the parameters changing less and less upon each iteration. For this reason, it is important to define a convergence criterion so that we may decide the HMM has converged sufficiently. A way to do this is to limit the number of BW iterations. By plotting the probability of the training data as a function of the number of iterations, we can perform an ad-hoc analysis to find a sufficient number of iterations for convergence.

Another way to define convergence is to find the change in Φ after each iteration. After each iteration, we find ΔA, ΔC, $\Delta\mu$, and $\Delta\Sigma$ as defined by

$$\Delta A = \sqrt{\sum_i \sum_j (\hat{a}_{ij} - a_{ij})^2}, \tag{A.21}$$

$$\Delta C = \sqrt{\sum_j \sum_k (\hat{c}_{jk} - c_{jk})^2}, \tag{A.22}$$

$$\Delta\mu = \sqrt{\sum_i \sum_j \sum_l (\hat{\mu}_{jk}(l) - \mu_{jk}(l))^2}, \tag{A.23}$$

$$\Delta\Sigma = \sqrt{\sum_i \sum_j \sum_l (\hat{\Sigma}_{jk}(l) - \Sigma_{jk}(l))^2}. \tag{A.24}$$

Here, l is used to ensure that all dimensions of the feature vectors are accounted for. We can choose an ϵ to be the convergence criterion for the BW algorithm. When $\Delta A \leq \epsilon$,

$\Delta C \leq \epsilon$, $\Delta\mu \leq \epsilon$, and $\Delta\Sigma \leq \epsilon$, then the BW algorithm is said to have converged.

Generally, ΔA and ΔC will converge much faster than $\Delta\mu$ or $\Delta\Sigma$. Since the values in A and C must be within the probability contstraints 0 and 1, they will change much less than the values in μ or Σ, which have no such constraint. The convergence will be dependent on how quickly $\Delta\mu$ and $\Delta\Sigma$ converge.

A slight modification of this method is to use different ϵ's to define convergence. Since ΔA and ΔC converge faster than $\Delta\mu$ and $\Delta\Sigma$, we can use a lower threshold for ΔA and ΔC than for $\Delta\mu$ and $\Delta\Sigma$. We can also combine the two criteria. For instance, we can require a minumum number of iterations and then stop training once our Δ-functions fall below our designated threshold. Conversely, we can stop training once our Δ-functions fall below our designate threshold, but also set a maximum number of iterations, so that training will stop after a certain number of iterations regardless of the values of the Δ-functions.

APPENDIX B

CODING THE BW ALGORITHM

The Baum-Welch algorithm is an iterative EM solution to the training problem discussed in Chapter 2. Although the formulas are fairly well-defined, there are a number of other considerations when implementing the Baum-Welch Algorithm. This appendix walks through a single iteration of the BW algorithm and explains implementational issues using pseudocode.

B.1 Organizing Variables

In order for a single BW iteration to execute, we must provide training data, and an initial Φ. This is a sample MATLAB function declaration for a BW iteration:

```
function [ oHMM, oMu, oSig ] = bwiter( X, Xind, iHMM, iMu, iSig )
```

The variables `X` and `Xind` can be used to hold all of the training data. In this organization, `X` contains the observations and `Xind` simply lists the time instant of the first observation for each observation sequence. The variable `iHMM` contains the initial state distribution π, the state-transition matrix $\mathbf{A}$, and the mixture weights c_{jk}. The variables `iMu` and `iSig`, respectively, contain the mixture mean vectors $\boldsymbol{\mu}_{jk}$ and covariance matrices $\boldsymbol{\Sigma}_{jk}$.

In general, the number of types of data in the BW algorithm is large, so it is advisible to try to minimize the number of variables in function calls. This can be done by organizing multiple variables into matrices (as shown above) or organizing data into structures where

possible. In our code we create a function `hmmdata()` within the BW-iteration to reextract our data from the matrices:

```
[iA, iC, iisd, nStates, nMixtures, vLength] = hmmdata( iHMM, iMu, iSig );
```

B.2 Limiting the Variance

In some cases of the BW algorithm, if the amount of observation data is small enough, some mixture variances will converge to zero. Since calculating the probability in a Gaussian requires inverting the variance, this will eventually result in divide-by-zero errors. To avoid this problem, we introduce a minimum variance to the algorithm:

```
varmin = 0.0001;
```

B.3 Memory Management

A single HMM iteration requires a lot of memory, so it becomes important to organize memory correctly. Generally, the update equation for each of our parameters has a numerator containing a summation and a denominator containing a summation. We simply create a matrix for each numerator and denominator and add the proper values to these matrices. This allows us to extract all of our update information from the training data at once, and then calculate all of the new parameters at once:

```
NumA = zeros(nStates, nStates);
NumC = zeros(nStates, nMixtures);
DenA = zeros(nStates, nStates);
DenC = zeros(nStates, nMixtures);
DentMutSigma = zeros(nStates, nMixtures);
NumtMu = zeros(nStates, nMixtures);
NumtSigma = zeros(nStates, nMixtures);
```

Note that the update equations for μ and Σ have the same denominator. We can save some memory by using one variable, `DentMutSigma`, to express both denominators.

B.4 Analyzing the Training Data

The next step in the BW algorithm is to use the HMM parameters to calculate $\alpha_t(i)$ and $\beta_t(i)$. In our implementation, we simply go through all of the training data at once. At each time instant, we calculate $\alpha_t(i)$ and $\beta_t(i)$.

First we must calculate the probability of the observation at our current time instant with respect to each mixture. We use a function `makeB()` that simply returns an $N \times M$ matrix containing the probability of an observation within each mixture of each state.

We compare our current time instant to the values in `Xind` to see whether we should use the standard iteration for $\alpha_t(i)$ and $\beta_t(i)$. If our current time instant is the first observation of a new sequence, we use the initialization formula for $\alpha_t(i)$. Otherwise, we use the iteration formula for $\alpha_t(i)$. Likewise, if the current time instant is the last observation of an observation sequence, we use the initialization for $\beta_t(i)$. Otherwise, we use the iteration formula for $\beta_t(i)$:

```
for i=1:N,
  if any(i==Xind)
    [Use initializization for alpha]
  else
    [Use iteration for alpha]
  end
end
for i=N:-1:1
  if any(i==(Xind-1))
    [Use initialization for beta]
  else
    [Use iteration for beta]
  end
end
```

Here, the equations for $\alpha_t(i)$ and $\beta_t(i)$ are given by Equations (2.21) and (2.22), Equations

(A.12) and (A.13). Unfortunately, because $\alpha_t(i)$ is calculated forward in time and $\beta_t(i)$ is calculated backwards in time, the two variables must be computed separately.

B.5 Scaling

If our training sequences are long enough, the probabilities in $\alpha_t(i)$ and $\beta_t(i)$ will become extremely small. This poses a major problem to floating-point units that have minimum decimals. In MATLAB, for instance, any number below 1.0×10^{-320} becomes a 0. To count for this problem, we store the variables with scaling values for each time instant. The introduced vectors of length N are called `alphascale` and `betascale`. For this we change the code above slightly:

```
for i=1:N,
  if any(i==Xind)
    [Use initialization for alpha]
    alphascale(i) = floor(max(log10(alpha(:,i))));
    alpha(:,i) = alpha(:,i} * 10^(-alphascale(i));
  else
    [Use iteration for alpha]
    alphascale(i) = floor(max(log10(alpha(:,i))));
    alpha(:,i) = alpha(:,i} * 10^(-alphascale(i));
    alphascale(i) = alphascale(i) + alphascale(i-1);
  end
end
for i=N:-1:1
  if any(i==(Xind-1))
    [Use initialization for beta]
    betascale(i) = floor(max(log10(beta(:,i))));
    beta(:,i) = beta(:,i} * 10^(-betascale(i));
  else
    [Use iteration for beta]
    betascale(i) = floor(max(log10(beta(:,i))));
    beta(:,i) = beta(:,i} * 10^(-betascale(i));
    betascale(i) = betascale(i) + betascale(i+1);
  end
end
```

We essentially store $\alpha_t(i)$ and $\beta_t(i)$ in scientific notation so they can be tracked for extremely small values.

B.6 Calculation of Baum-Welch Terms

Next, we calculate $\gamma_{ij}(t)$ and $\zeta_{jk}(t)$ as directed by the Baum-Welch algorithm. Here, we can simply move forward through the observations in the training data and apply Equations (A.14) and (A.15). These terms are undefined for the first observation of each set of training data. We can implement this code as follows:

```
for i=1:N,
  if any(i==Xind)
    [do nothing]
  else
    [perform gamma calculation]
    [perform zeta calculation]
    gamma = gamma * 10^(alphascale(i-1) + betascale(i) - alphascale(N));
    zeta = zeta * 10^(alphascale(i-1) + betascale(i) - alphascale(N));
  end
end
```

Note that the calculations for $\gamma_{ij}(t)$ and $\zeta_{jk}(t)$ are slightly modified to account for the scaling factors.

B.7 Updating the HMM Parameters

With $\gamma_{ij}(t)$ and $\zeta_{jk}(t)$ computed, we can now update our numerators and denominators and continue to the next set of training data:

```
for i=1:nStates,
  for j=1:nStates,
    NumA(i,j)=NumA(i,j)+sum(HMMgamma(:,i,j));
```

```
    DenA(i,j)=DenA(i,j)+sum(sum(HMMgamma(:,i,:)));
  end
  for k=1:nMixtures,
    NumC(i,k)=NumC(i,k)+sum(HMMzeta(:,i,k));
    DenC(i,k)=DenC(i,k)+sum(sum(HMMzeta(:,i,:)));
    DentMutSigma(i,k)=DentMutSigma(i,k)+sum(HMMzeta(:,i,k));
    NumtMu(i,k)=NumtMu(i,k)+HMMzeta(:,i,k)'*thisX;
    NumtSigma(i,k)=NumtSigma(i,k)+HMMzeta(:,i,k)'*((thisX-iMu(i,k)).^2);
  end
end
```

When all training data is analyzed, $\hat{\Phi}$ is found by dividing the numerators by the denominators:

```
A = NumA ./ DenA;
C = NumC ./ DenC;
oMu = NumtMu ./ DentMutSigma;
oSig = max(varmin, NumtSigma ./ DentMutSigma);
```

As a final step we repackage A, c_{jk}, and $\boldsymbol{\pi}$ into the HMM format discussed above. π is simply set as a uniform distribution.

APPENDIX C

CLUSTERING ALGORITHMS

To understand the K-means and modified K-means clustering algorithms, we must first introduce two new terms that are used by these algorithms.

The centroid c_i is the average position of all the feature vectors contained within a cluster:

$$c_i = \frac{\sum_{\mathcal{X}_t \in \mathcal{C}_i} \mathcal{X}_t}{\sum_{\mathcal{X}_t \in \mathcal{C}_i} 1}. \tag{C.1}$$

Here, $\mathcal{C}_i$ refers to cluster i, and c_i refers to the centroid of $\mathcal{C}_i$.

The distortion D_i is the total distance from all of the members of $\mathcal{C}_i$ to its centroid c_i. Distortion is calculated by

$$D_i = \sqrt{\sum_{\mathcal{X}_t \in \mathcal{C}_i} (\mathcal{X}_t - c_i)(\mathcal{X}_t - c_i)'}, \tag{C.2}$$

where D_i refers to the distortion of $\mathcal{C}_i$. The distortion is often a good metric for choosing a cluster when it is necessary to split a cluster. In these situations, the cluster with the largest distortion is the best one to split into multiple clusters.

C.1 K-Means Clustering

In the K-means algorithm, the first step is to randomly map K points to K clusters. Then the remaining points are mapped to the nearest cluster. The cluster centroids are recalculated

and all points are remapped to the cluster with the nearest centroid. This continues until there is an iteration where no data points switch clusters. This process is given as follows:

1. Pick any K data points at random to be the initial centroids.
2. Assign the remaining data points to the cluster with the nearest centroid.
3. Recalculate the centroid for each cluster.
4. If the centroids have moved, go to step 2.

C.2 The Modified K-Means Clustering Algorithm

A drawback of the K-means algorithm is that it is highly dependent on the initial data points chosen as centroids. The strength of the K-means algorithm lies in splitting a single cluster into two. The modified K-means algorithm splits clusters in half iteratively until there are exactly K clusters. The steps involved in the modified K-means algorithm are as follows:

1. Calculate the centroid of the entire set of data points.
2. Calculate the distortion of the cluster.
3. Pick any data point within the cluster with the largest distortion. This point is the first member of the newest cluster.
4. Apply K-means iterations until all clusters are formed.
5. Calculate the distortion of each cluster.
6. If there are less than K clusters, return to step 3.

APPENDIX D

THE EM ALGORITHM AND GAUSSIAN MIXTURE MODELS

The Gaussian mixture model (GMM) is a method of characterizing a continuous probability density function as a sum of scaled Gaussian distributions. The benefits of doing this are two-fold. First, the Gaussian distribution is continuous and nonzero, so it does a sufficient job of modeling other distributions over short periods. Secondly, the k-th Gaussian distribution within a GMM is completely defined by two parameters, the mean vector $\boldsymbol{\mu}_k$, and the covariance matrix $\boldsymbol{\Sigma}_k$. A third parameter p_k is called the mixture weight and contains the scaling factor of a particular Gaussian within the mixture. For compactness, the GMM is refered to by $\lambda = (\boldsymbol{p}, \boldsymbol{\mu}, \boldsymbol{\Sigma})$, where $\boldsymbol{p}$ is the collection of all mixture weights p_k, $\boldsymbol{\mu}$ is the collection of all mean vectors $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}$ is the collection of all covariance matrices $\boldsymbol{\Sigma}_k$.

Thus the probability of a certain feature vector within a GMM is given by the sum of its probability in each mixture:

$$p(x|\lambda) = \sum_{k=1}^{M} p_k b_k(x), \qquad 1 \leq k \leq M. \tag{D.1}$$

Here, p_k represents each mixture weight, and $b_k(x)$ is the probability of the feature vector within each mixture of the GMM. We also know that since the area of each Gaussian is 1, the p_k must add to 1:

$$\sum_{k=1}^{M} p_k = 1. \tag{D.2}$$

D.1 Training the GMM

To train a GMM, we want to maximize the probability of the training data within the p.d.f. defined by the GMM. The training process begins with an initial Gaussian model and the model is updated with a slightly modified version of the EM algorithm. Due to the constraint given in Equation (D.2), the sum of the mixture weights must be calculated differently. This is much like the constraints faced by the BW algorithm for training HMMs. In training the GMM, there are three parameters that can be changed: p_k, $\boldsymbol{\mu}_k$ and $\boldsymbol{\Sigma}_k$.

For a set of T observations, $\mathcal{X} = (X_1, X_2, \ ..., \ X_T)$, the GMM update equations are given by

$$p_k = \frac{1}{T}\sum_{t=1}^{T} p(i|X_t, \lambda), \qquad 1 \leq k \leq M, \tag{D.3}$$

$$\boldsymbol{\mu}_k = \frac{\sum_{t=1}^{T} p(i|X_t, \lambda) X_t}{\sum_{t=1}^{T} p(i|X_t, \lambda)}, \tag{D.4}$$

$$\boldsymbol{\Sigma}_k = \frac{\sum_{t=1}^{T} p(k|X_t, \lambda)(X_t - \boldsymbol{\mu}_i)(X_t - \boldsymbol{\mu}_i)^T}{\sum_{t=1}^{T} p(i|X_t, \lambda)}. \tag{D.5}$$

As with other EM algorithms, this modified EM algorithm is an iterative process. Each updated GMM will be slightly better than the previous one. This algorithm runs iteratively while converging to an eventual locally maximal probability with respect to the GMM parameters. The performance of the final GMM is highly dependent on the initial GMM.

D.2 Initial Estimation

This algorithm needs an initial estimate to run. The initial estimate used most often is found by clustering the observations into M mixtures. For each mixture, the mean and variance are used as initial means and variances. The mixture weights are determined by finding the number of observations within each mixture and then dividing it by the total number of

observations. The equations for these calculations are given by

$$\mu_k = \frac{\sum\limits_{X_t \in S_k} X_t}{W_k}, \tag{D.6}$$

$$\Sigma_k = \frac{\sum\limits_{X_t \in S_k} (X_t - \boldsymbol{\mu}_k)(X_t - \boldsymbol{\mu}_k)^T}{W_k}, \tag{D.7}$$

$$p_k = \frac{W_k}{T}. \tag{D.8}$$

Here, W_k refers to the number of observation vectors within each mixture cluster. Note that Equation (D.8) by definition satisfies the constraint given in Equation (D.2).

REFERENCES

[1] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proc. IEEE*, vol. 77, pp. 257–286, Feb. 1989.

[2] L. Rabiner and B.-H. Juang, *Fundamentals of Speech Recognition*. Englewood Cliffs, NJ: Prentice Hall, 1993.

[3] C. Bishop, *Neural Networks for Pattern Recognition*. New York: Oxford University Press, 1995.

[4] R. P. Lippmann, "Review of neural networks for speech recognition," *Neural Computation*, vol. 1, no. 1, pp. 1–38, 1989.

[5] L. Brieman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Belmont, CA: Wadsworth, Inc., 1984.

[6] B. Allen and J. Munro, "Self-organizing search trees," *J. ACM*, vol. 25, no. 4, pp. 526–535, 1978.

[7] D. Hosmer and S. Lemeshow, *Applied Logistic Regression*. New York: John Wiley and Sons, 1989.

[8] Z. Huang, A. Acero, and H.-W. Han, *Spoken Language Processing: A Guide to Theory, Algorithm and System Devlopment*. Upper Saddle River, NJ: Prentice Hall, 2001.

[9] T. Zhang and C.-C. J. Kuo, *Content-Based Audio Classification and Retrieval For Audiovisual Data Parsing*. Los Angeles: Kluwer Academic Publishers, 2001.

[10] N. Çadallı, "A tutorial on hidden Markov models," Phonak internal report PUC-2004-03-NC-02, Phonak Hearing Systems, University of Illinois Research Park, Champaign, IL, Mar. 2004.

[11] R. Boudarel, J. Delmas, and P. Guichet, *Dynamic Programming and its Application to Optimal Control*. New York: Academic Press, 1971.

[12] A. Gersho and R. M. Gray, *Quantization and Signal Compression*. Los Angeles: Kluwer Academic Publishers, 1992.

[13] I. Katsavounidis, C.-C. J. Kuo, and T. Zhang, "A new initialization technique for generalized Lloyd iteration," *IEEE Signal Process. Let.*, vol. 1, pp. 144–146, Oct. 1994.

[14] S. Theodoridis and K. Koutroumbas, *Pattern Recognition*. San Diego: Academic Press, 1999.

[15] B. Gold and N. Morgan, *Speech and Audio Signal Processing*. New York: John Wiley and Sons, 2000.

[16] Y. Ephraim, "Gain-adapted hidden Markov models for recognition of clean and noisy speech," *IEEE Trans. Signal Processing*, vol. 40, pp. 1303–1316, June 1992.

[17] I. Sanches, "Noise-compensated hidden Markov models," *IEEE Trans. Speech and Audio Processing*, vol. 8, pp. 533–540, Sept. 2000.

[18] Y. Ephraim, D. Malah, and B.-H. Juang, "On the application of hidden Markov models for enhancing noisy speech," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 37, pp. 1846–1856, Dec. 1989.

[19] S. Sharma, D. Ellis, S. Kajarekar, P. Jain, and H. Hermansky, "Feature extraction using no-linear transformation for robust speech recognition on the aurora database," in *ICASSP 2000*, vol. 2, pp. 1117–1120, 2000.

[20] T. M. Cover and J. A. Thomas, *Elements of Information Theory*. New York: John Wiley and Sons, 1991.