Structure Discovery in Hidden Markov Models

by

Ben Murrell

Submitted in partial fulfilment of the
academic requirements for the degree of
Master of Arts in Cognitive Science
in the School of Philosophy and Ethics,
University of KwaZulu-Natal

Durban South Africa, 2009

Declaration

These studies represent original work by the author and have not otherwise been submitted in any form for any degree or diploma to any tertiary institution. Where use has been made of the work of others it is duly acknowledged in the text.

Name:

Date:

Signed:

As the candidate's supervisor I have/have not approved this thesis for submission

Name:

Date:

Signed:

Abstract

The Baum-Welch algorithm for training hidden Markov models (HMMs) requires model topology and initial parameters to be specified, and iteratively improves the model parameters. Sometimes prior knowledge of the process being modeled allows such specification, but often this knowledge is unavailable. Experimentation and guessing are resorted to. Techniques for discovering the model structure from observation data exist but their use is not commonplace. We propose a state splitting approach to structure discovery, where states are split based on two heuristics: within-state autocorrelation and a measure of Markov violation in the state path. Statistical hypothesis testing is used to decide which states to split, providing a natural termination criterion and taking into account the number of observations assigned to each state, splitting states only when the data demands it.

Acknowledgements

I would like to thank Jules and David for making this thesis possible. Jules, my interactions with you over the past year have been career changing. Before our conversations last May I was a humanities student, and I had not heard of hidden Markov models. Thank you for always enthusiastically pointing me in the right direction.

David, you introduced me to research. Your patience and persistent encouragement allowed me to dabble in topics well beyond the reasonable boundaries of our discipline. I am very grateful for this.

I would also like to thank my parents for their support and encouragement and for patiently enduring many discussions about hidden Markov models and state splitting.

Finally, I would like to thank Sasha. For everything.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Hidden Markov models (HMMs) are tools for modeling time series. They are used for classification, clustering and prediction, commonly in speech recognition [68], biological sequence analysis [38] and stock market forecasting [96], and not so commonly in whale song analysis [16], forecasting turmoil in Indonesia [13], and in neural signal processing [31, 37]. The ubiquity of HMMs is due to their expressive power, and to the existence of tractable algorithms for HMM inference and parameter estimation. This short chapter will informally introduce HMMs, give a brief description of the structure discovery problem that this thesis seeks to address, and provide an outline of the rest of the thesis.

1.2 Hidden Markov Models - A Brief Informal Overview

Consider a series of observations of some process, sampled at regular intervals. Here are three examples:

1. The type of sandwiches in Johnny's lunchbox, either toasted cheese, or peanut butter.
2. The daily rainfall at Umzumbe in millimeters.
3. The (x, y, z) coordinates of someone's hand as they sign to their friend.

Note the different types of observation variable. 1 is categorical, 2 is real, and 3 is a vector of real values. In each case, there is some process underlying the production of the observations, but for now we will assume that these noisy observations are our only window to the state of the process behind their production.

Let's embellish example 1 a little. Johnny's mother is self employed. Some days she is busy with work, and others she is free. Johnny is a strange child, and he counts everything. Over the last year, Johnny noticed that the kind of sandwich in his lunchbox depends on whether or not his mother is busy. If she is free, she is more likely to spend the extra time making toasted cheese. Specifically, if she is free, he gets peanut butter 30% of the time, and toasted cheese 70%. If she is busy, he gets peanut butter 90% of the time, and toasted cheese 10%. As jobs tend to last longer than just a day, if Johnny's mother is busy the one day, the chances she will be busy the next are 60%, and 40% that she will be free. Owing to the recession, periods of unemployment last longer than periods of work, and if Johnny's mother was free on one day, she has a 90% chance of being free the next, and a 10% chance of being busy. Lacking Johnny's perfect numerical recall, we can benefit much from figure 1.1, which represents all this information compactly.

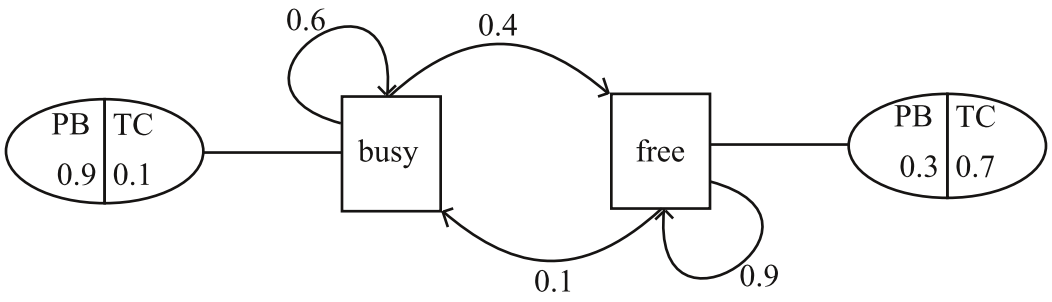

Figure 1.1: *Johnny's sandwiches: A Hidden Markov Process.* Squares represent the state of Johnny's mom, either 'busy' or 'free'. Arrows between states are labeled with the probability of transiting from one state to another (or to itself). Ellipses represent the output distributions of their associated state, in this case, the probabilities of Johnny getting PB (Peanut Butter) or TC (Toasted Cheese) when his mom is in that state.

This describes the process that determines Johnny's sandwiches. It is a generative model. Assuming some state at time $t = 1$, we can use this model to generate a sequence of states and observations from $t = 1$ onwards. Say we started at time $t = 1$ in state 'busy'. To generate the observation (the type of sandwich) at time $t = 1$, we refer to the observation probabilities associated with that state. We pick a random real number between 0 and 1. If it is less than 0.9 (look at the observation probabilities for state 'busy'), then the observation produced is PB, else it is TC. Then, to decide which state to move to, we generate another random number between 0 and 1. If its less than 0.6 (look at the arrows starting at 'busy'), the state at time $t = 2$ will be 'busy', else it will be 'free'. Proceeding in this fashion, we could generate a sample sequence produced by this process.

A few things need to be noticed. Firstly, the state at time $t+1$ depends only on the state at time t. The process governing the evolution of the *state* sequence is thus

called a Markov process, of which there will be more later. Now in the case where we don't observe the states of the process (Johnny doesn't know whether or not his mom is actually busy), but only the observations they stochastically produce, we say the Markov process is 'hidden'. Apart from representing the probabilities of the very first state, figure 1.1 represents a fully specified hidden Markov model.

There is a tension present above: HMMs deal with the case where the states are not directly observable, but, in this example, we somehow knew the dynamics governing how the system moved between states from one day to the next. How is this possible? For now, we will spin the yarn that Johnny used to know on what days his mother was busy but stopped paying attention after receiving a TV and so he no longer knows. Before, when he could view the entire system, he could simply count how often she changed from busy to free and *vice versa*, and how often he got certain sandwiches in certain states, and use these counts to estimate the frequencies represented by the HMM (called the HMM 'parameters'). This, however, is not typically the situation. In most applications of HMMs, we *never* get to see the states directly. How then, might we construct HMMs to model anything? A technique for constructing an HMM from the observations alone was presented in [42] in 1970, and is still the most commonly used today. It is considerably more involved than Johnny's simple counting strategy, so unless Johnny's surname happened to be von Neumann, he would most likely never have figured it out. In the canonical exposition [68], it is one of the three important problems associated with HMMs, and will be discussed after the other two.

The three problems canonically associated with HMMs [68] are:

- How can we efficiently compute the probability of a particular sequence of observations, given a particular model? This is the probability that a particular HMM, if used to generate an observation sequence of the appropriate length, would output that particular sequence, as opposed to some other sequence. This has a fairly simple analytical expression which can be efficiently computed with the Forward algorithm.

- Given a model and a sequence of observations, how can we compute the most likely sequence of states that produced that observation sequence? This is efficiently solved using the Viterbi algorithm.

- Given a model and a sequence of observations, how can we modify the model parameters to create a new model, such that the probability of the observation sequence given the new model is greater than the probability of the observation sequence given the old one? This is solved using the Baum-Welch algorithm.

The algorithms that solve these three problems are crucial to most HMM applications. We can use them to build models for prediction, perform supervised classification, and unsupervised clustering of sequences of observations. The formal specification of HMMs and the solutions to these three problems will be the next

chapter's goal. Before that, however, we will first informally state the structure discovery problem that this thesis seeks to address.

1.3 The Structure Discovery Problem - An Informal Statement

Armed with algorithms to solve the three canonical HMM problems, a practitioner proceeds by specifying an initial model, including all parameters, and iterating the Baum-Welch algorithm to improve how the model fits the data. If one knows much about the underlying process, one can use this to make informed decisions about model structure, such as how many states to use. As Baum-Welch training does not guarantee that globally optimal model parameters are found, specifying good initial model parameters is important in practice to avoid getting trapped in deleterious local optima. Without knowledge of the process, one is forced into guessing both model structure and initial parameters: a situation which is clearly not ideal. Structure discovery approaches to hidden Markov modeling attempt to ameliorate this by providing strategies to construct HMMs without specifying initial parameters and attempting to avoid local optima during training. This thesis proposes and tests a novel structure approach.

1.4 Thesis Outline

Chapter 2 introduces probability theory, which is followed by a formal presentation of HMMs. The rest of the chapter describes in detail the Viterbi algorithm for state-path inference, the Forward algorithm for likelihood computation, and the Baum-Welch algorithm for parameter learning.

Chapter 3 begins with a survey of HMM applications in a variety of domains: gesture recognition, the analysis of behavior, neural signal processing, bioinformatics, ecological modeling, medicine, weather, fault detection and music. After this, the HMM structure discovery problem is introduced, and criteria for model selection are briefly discussed. Finally, previous HMM structure discovery approaches, mostly based on state splitting or state merging, are surveyed.

Chapter 4 presents a novel HMM structure discovery approach. The notion of within-state autocorrelation is introduced, and arguments are made for why it is a useful heuristic for discovering the structure of an HMM. The problem of overlapping densities is discussed, and a further heuristic is introduced to solve it. A state splitting algorithm based on these heuristics is described, and it it tested on a classification task using synthetic univariate data, as well as real multivariate data, comparing classification accuracies with those from standard Baum-Welch training.

Chapter 5 begins with some reflections on the proposed algorithm and suggests some potential avenues for future research. These include adapting the algorithm to work for discrete observation data, extending the complexity of each state's observation distribution, and combining the algorithm with an already existing one to improve its efficiency.

The optimal reading path through this thesis depends on a reader's background. Someone completely unfamiliar with HMMs should read it as is, although if amidst the detail of chapter 2 they begin to wonder "What is it all for?" they might want to take a brief detour through any interesting looking applications in section 3.1 to help recruit motivation. A reader familiar with HMMs from a speech recognition background, and thus comfortable with vectors of continuous observations, could begin with section 3.2. One familiar with discrete-output HMMs, perhaps from a bioinformatics background, could read section 2.5.8, using the summary in section 2.3.1 to reconcile any notational differences, and then read from section 3.2 onwards.

Chapter 2

Theoretical Background

This chapter introduces the probability theory necessary to understand HMMs, which will form the analytical framework for the rest of this thesis. A formal treatment of HMMs and their associated algorithms follows this.

2.1 Probability Theory

As might be gathered from the previous chapter, understanding the techniques behind hidden Markov models requires familiarity with elementary probability theory, which we introduce here. The exposition in this section closely follows [12].

Consider a number of trials of some experiment, where each trial has a number of different outcomes. For our purposes, the probability of a particular outcome is the fraction of the times that outcome occurs over the total number of trials, as the number of trials approaches infinity. To clarify, and introduce some notation, we once again recycle our Johnny example. Concerning ourselves, for now, only with Johnny's sandwiches, we want to denote the probability that, on a randomly selected day, Johnny will get a particular kind of sandwich. We introduce the random variable T, for the type of sandwich Johnny gets, and the individual outcomes t_1, for a peanut butter sandwich, and t_2 for a toasted cheese sandwich. We denote the probability that, on a randomly selected day, Johnny gets a peanut butter sandwich $p(T = t_1)$, and that he gets toasted cheese $p(T = t_2)$. Notice Johnny can never get a particular type of sandwich more times than the total number of trials, or less than 0 times, and thus, by our definition, each probability must lie on the interval $[0, 1]$. Also, if we assume that these outcomes are exhaustive (that Johnny has no other lunch options), and mutually exclusive, the sum of these probabilities will be 1.

One can also consider combinations of variables. Let us denote the probability that, on a given day, ants crawl on Johnny's lunchbox $p(A = a_1)$, and $p(A = a_2)$ the probability that they don't. We can then denote the probability that Johnny has peanut butter sandwiches *and* that ants crawl on his lunchbox $p(T = t_1, A = a_1)$.

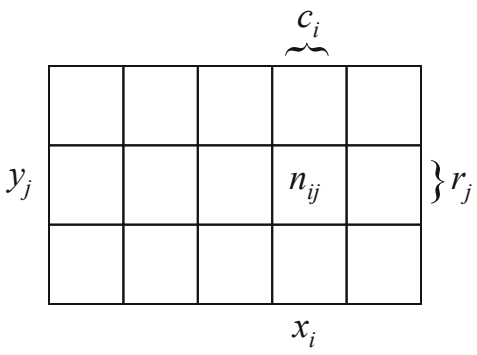

Figure 2.1: *Deriving the sum and product rules.* See text for explanation.

Forgetting Johnny, and moving to a general example, consider sampling from two random variables, X and Y, which can take the values $X = x_i$ for $i = 1, ..., M$ and $Y = y_j$ for $j = 1, ..., L$. Let N be the number of times we sample, and let the number of samples where $X = x_i$ and $Y = y_j$ be n_{ij}. Furthermore, c_i is the number of times $X = x_i$, regardless of the value of Y, and r_j is number of times $Y = y_i$, regardless of X. Figure 2.1 represents this situation. $p(X = x_i, Y = y_j)$ is the number of times X takes the value x_i *and* Y takes the value y_j (called the joint probability), and is simply the fraction n_{ij}/N, as N approaches infinity. Thus,

$$p(X = x_i, Y = y_j) = \frac{n_{ij}}{N} \tag{2.1}$$

Similarly, the probability that X takes the value x_i, regardless of the value of Y (called the *marginal* probability) can be denoted

$$p(X = x_i) = \frac{c_i}{N} \tag{2.2}$$

Examining figure 2.1, we see that the number of events in column i is simply the sum of instances over every cell in the column, so $c_i = \sum_j n_{ij}$. From equations 2.1 and 2.2, and moving the division by N inside the summation, we have the sum rule:

$$p(X = x_i) = \sum_{j=1}^{L} p(X = x_i, Y = y_j) \tag{2.3}$$

The probability of $Y = y_j$ given $X = x_i$, denoted $p(Y = y_j|X = x_i)$ is the number of times y_j and x_i jointly occur, over the total number of times x_i occurs.

$$p(Y = y_j|X = x_i) = \frac{n_{ij}}{c_i} \tag{2.4}$$

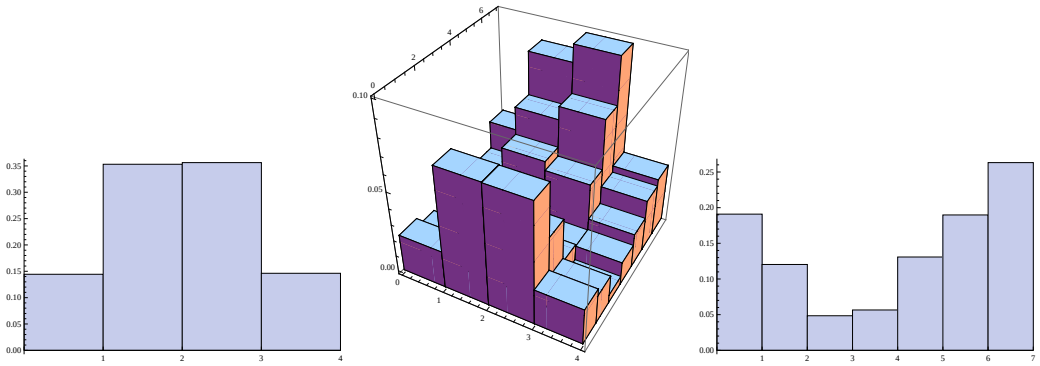

Figure 2.2: *Distributions.* Centre is the full joint distribution $p(X, Y)$. Here, X can take four values and Y seven. Left is the marginal distribution $p(X)$. Each x_i is equal to $\sum_{j=1}^{7} y_j$. The marginal distribution $p(Y)$ can be calculated in the same fashion, *mutatis mutandis*. A conditional distribution would be obtained by taking a particular row or column from the joint distribution, and normalizing it to ensure that it sums to 1.

From equations 2.1, 2.2, and 2.4, we have the product rule:

$$\begin{aligned} p(Y = y_j, X = x_i) &= \frac{n_{ij}}{N} = \frac{n_{ij}}{c_i} \cdot \frac{c_i}{N} \\ &= p(Y = y_j | X = x_i) p(X = x_i) \end{aligned} \tag{2.5}$$

To avoid clumsy notation, we will sometimes use $p(x_i)$ to denote $p(X = x_i)$, and $p(X)$ to denote the entire distribution over the random variable X, where context disambiguates. This allows for the compact expression of the sum and product rule:

$$p(X) = \sum_{Y} p(X, Y) \tag{2.6}$$

$$p(X, Y) = p(Y|X)p(X) \tag{2.7}$$

Figure 2.2 serves as a further illustration of the notion of joint and marginal distributions. It should be pointed out that these extend straightforwardly to more than just 2 variables. We can also straightforwardly extend these rules to continuous random variables, by replacing the summation signs with integrals. Conditional distributions then correspond to a normalized horizontal or vertical slice through the joint distribution.

2.1.1 Bayes' Theorem

From the product rule (equation 2.7) and noticing that $p(X, Y) = p(Y, X)$, we get

$$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)} \tag{2.8}$$

This is called Bayes' theorem. Applying the sum and product rules to the denominator, we can express Bayes' theorem in terms of the probabilities occurring in the denominator:

$$p(Y|X) = \frac{p(X|Y)p(Y)}{\sum_{Y} p(X|Y)p(Y)} \tag{2.9}$$

These are the rules we will use in this chapter. If you are not well acquainted with probability theory, having them ready to hand will almost certainly be helpful.

2.2 Sequential Data

This thesis will only be concerned with sequential data. This is where data points can be arranged so that they are ordered in some meaningful sense. One point follows another. Time series, where a series of measurements are taken at discrete intervals, are good examples of this, but not the only ones. Nucleotide sequences also qualify, as do horizontal rows of pixels on your screen. An important caveat is that the intervals between successive measurements must be the same size, although the formalism can be extended to handle irregular intervals. The observations can be categorical, as with nucleotide sequences, univariate real valued, as with the price of a single stock over time, or vectors of real values, as with the pixels on your (colour) screen. The following exposition will initially deal only with discrete observation symbols, but will later be extended to deal with real and vector valued observations.

We will first deal with Markov chains of states, and then hide those states behind observations, yielding HMMs. From now on, this exposition very closely follows [68], with a slight twist in the order, and an attempt to fill in some of the gaps in the derivations.

Consider a process which can be said to be in one of N discrete states, at discrete moments in time. We denote the set of states $S = \{S_1, ..., S_N\}$. In our Johnny example, N was 2 and the 2 states were 'busy' and 'free'. q_t denotes the state the process was in at time t. For example, $q_{31} = S_2$ says that the process was in state 2 at the 31^{st} time step. Without any assumptions about the dependence of states on previous states,

$$p(q_1, ..., q_T) = \prod_{t=1}^{T} p(q_t|q_1, ..., q_{t-1}) \tag{2.10}$$

where T is the last time step in the sequence. This is because, without restricting assumptions, q_t might depend on the whole history of states that have occurred so far. Inference and learning with such complicated joint distributions is usually intractable, so we need to make some simplifying assumptions about how the state sequence evolves from one time period to the next. The assumption present throughout this thesis is called the first-order Markov assumption.

2.2.1 The First-Order Markov Assumption

We assume q_t is independent of all previous states except q_{t-1}. This is equivalent to saying that the conditional probability of the state at time t given the entire history of states is equal to the conditional probability of the state at time t given only the previous state. It is a credit to the power of notation that this can be expressed as

$$p(q_t|q_1, ..., q_{t-1}) = p(q_t|q_{t-1}) \tag{2.11}$$

This means that the full joint distribution from equation 2.10 can be expressed as

$$p(q_1, ..., q_T) = p(q_1) \prod_{t=2}^{T} p(q_t|q_{t-1}) \tag{2.12}$$

If we further assume that $p(q_t|q_{t-1})$ does not depend on t, we can very compactly describe the Markov process with a transition matrix and an initial state distribution. We denote the transition probabilities with a matrix A, where a_{ij} is the probability of moving to state j, after being in state i, or $p(q_t = S_j|q_{t-1} = S_i)$. Ignoring the sandwiches for the moment, consider just the states of Johnny's mom in figure 1.1, and how they shift from one to another over time. If we say 'busy' is state 1, and 'free' is state 2, we have

$$A = \begin{pmatrix} 0.6 & 0.4 \\ 0.1 & 0.9 \end{pmatrix}$$

Notice that each row must obey stochastic constraints, having non-negative elements and summing to 1. If we specify the initial state probability distribution, π, where $\pi_i = p(q_1 = S_i)$, we have a fully described first-order Markov model.

$$\pi = \begin{pmatrix} 0.2 \\ 0.8 \end{pmatrix}$$

If we can observe a number of states sequences produced by some Markov process, we can estimate the initial and transition probabilities, simply by counting the starting states and transitions. Also, if we have a particular Markov model, we can calculate the probability that a particular sequence will be produced, using equation 2.12, π and A. For the state sequence $Q = \{2, 2, 1, 1, 2\}$ with the above A and π,

$$\begin{aligned} p(Q = \{2, 2, 1, 1, 2\}) &= \pi_2 a_{2,2} a_{2,1} a_{1,1} a_{1,2} \\ &= 0.8 \times 0.9 \times 0.1 \times 0.6 \times 0.4 \\ &= 0.01728 \end{aligned}$$

We can use a simulation as a sanity check on our calculations. We set up a Markov model with A and π as specified above, and use the sampling procedure outlined in the introduction (MATLAB code in A.1). We generate one million state sequences of length 5, and check for what proportion of them $Q = \{2, 2, 1, 1, 2\}$. Happily, the result is 0.017316, which is close to our calculated value of 0.01728. As the number of simulated trials increases, we would expect this proportion to converge to the analytically calculated value. The most common sequence of length 5 is $Q = \{2, 2, 2, 2, 2\}$, which has a probability of 0.52488, appearing just over half the time. Indeed, a simulation, also with one million trials, yields a value of 0.5252. We will use such sanity checks throughout this chapter, wherever appropriate. The MATLAB code for such sanity checks is in A.2.

2.3 Hidden Markov Models

The Markov process we could previously observe must now be hidden. To recapitulate the big picture, a Markov model consists of a set of states, a matrix of transition probabilities describing the evolution of the state sequence, and a vector of initial state probabilities. To create an HMM, we take such a Markov model and assign to each state a distribution of observations (also called outputs or emissions in the literature). The Markov model produces a sequence of states and, for each state, an observation is sampled from that state's observation distribution. This section will begin by introducing the notation used in the rest of the chapter. Some of it has been defined before but, due to its importance, will be presented again.

2.3.1 Notation

- N is the number of states in the HMM.
- $S = \{S_1, ..., S_N\}$ is the set of states.
- q_t denotes the state the model was in at time t, with Q denoting the entire state sequence.
- M is the number of distinct observation symbols produced by the HMM.
- $V = \{v_1, ..., v_M\}$ denotes the set of observation symbols.
- O_t denotes the observation symbol at time t, with O denoting the entire observation sequence.
- T is the total number of time steps.
- $A = \{a_{ij}\}$ denotes the state transition probability distribution, with $1 \leq i, j \leq N$ and,

$$a_{ij} = p(q_{t+1} = S_j | q_t = S_i) \tag{2.13}$$

- $B = \{b_j(k)\}$ denotes the observation symbol probability distribution in state j, where $1 \leq j \leq N$ and $1 \leq k \leq M$, and

$$b_j(k) = p(O_t = v_k | q_t = S_j) \tag{2.14}$$

- $\pi = \{\pi_i\}$ denotes the initial state distribution, where $1 \leq i \leq N$, and

$$\pi_i = p(q_1 = S_i) \tag{2.15}$$

- $\lambda = (A, B, \pi)$ is the compact notation denoting the full HMM.

Stated using this notation, an HMM generates a state sequence Q and an observation sequence O. The state sequence evolves as a Markov chain, with $p(q_1 = S_i) = \pi_i$, and $p(q_{t+1} = S_j | q_t = S_i) = a_{ij}$, until T is reached. For each t, $p(O_t = v_k | q_t = S_j) = b_j(k)$. Typically the observation sequence is visible but the state sequence is hidden, and needs to be inferred.

2.3.2 The Three HMM Problems, Formally Restated

Armed with this notation, we can now restate the three canonical HMM problems, and say why they are useful.

Problem 1 - Computing $p(O|\lambda)$

For this problem we are given a particular observation sequence O, and a model λ. We want to compute in what proportion of trials λ would produce O, as the number of trials tends to infinity. This is extremely useful, because when we are trying to match models to observations sequences, as in a classification task, $p(O|\lambda)$ is exactly what we need. The Forward algorithm solves this problem.

Problem 2 - Finding the Most Likely State Sequence

Here, we are given a model λ and an observation sequence O, and we want to find a state sequence Q that maximizes $p(Q|O, \lambda)$. This is efficiently solved using the Viterbi algorithm, and the corresponding Q is often referred to as the 'Viterbi path'. This is especially useful when the states of the model have some interesting real-world interpretation.

Problem 3 - Improving the HMM Parameters

Given a model λ and an observation sequence O, problem 3 involves modifying the parameters of λ to produce a new model $\overline{\lambda}$, such that $p(O|\overline{\lambda}) \geq p(O|\lambda)$. The Baum-Welch algorithm computes just such a $\overline{\lambda}$. This procedure can be iterated until $p(O|\overline{\lambda})$ reaches a (local) maximum, which brings us to the most common way HMMs are constructed for processes. First, obtain a collection of training observation sequences. Then, guess a number of states and the initial architecture and parameters for λ. Lastly, iterate the Baum-Welch algorithm until a local maximum is reached. The final $\overline{\lambda}$ is taken to be the HMM for those observation sequences. This procedure is problematic for reasons we will discuss in detail later, but is nevertheless the standard technique for constructing HMMs.

2.3.3 The Forward Procedure

For an observation sequence of length T, $O = \{O_1, ...O_T\}$, the forward procedure efficiently computes $P(O|\lambda)$, using an inductive procedure. As a way of better understanding the problem, we first present a very inefficient way to compute this.

Notice that if we were also given Q, a particular sequence of states, we could easily compute

$$p(O|Q, \lambda) = \prod_{t=1}^{T} p(O_t|q_t, \lambda) \tag{2.16}$$

because the observations are conditionally independent given the state sequence. Using the product rule,

$$p(O, Q|\lambda) = p(O|Q, \lambda)p(Q|\lambda) \tag{2.17}$$

We already know how to calculate $p(Q|\lambda)$ from equation 2.12. Finally, using the sum rule,

$$p(O|\lambda) = \sum_{\forall Q} p(O|Q, \lambda)p(Q|\lambda) \tag{2.18}$$

The problem with this calculation is that it requires summing over all possible state sequences. This is usually intractable, as the number of possible state sequences is N^T, and T is typically quite large. To mitigate this, we introduce the 'Forward' variable, α.

The Forward Variable

$$\alpha_t(i) = p(O_1O_2...O_t, q_t = S_i|\lambda) \tag{2.19}$$

denotes the joint probability of the partial observation sequence $O_1O_2...O_t$ from time 1 to t, and the model being in state S_i at time t, given the model λ.

Solving for $\alpha_t(i)$ inductively:

Initialization

Given a model λ, with $1 \leq i \leq N$,

$$\alpha_1(i) = p(O_1, q_1 = S_i) \tag{2.20}$$

Applying the product rule,

$$\alpha_1(i) = p(O_1|q_1 = S_i)p(q_1 = S_i) \tag{2.21}$$

Which, from equations 2.14 and 2.15, is simply

$$\alpha_1(i) = b_i(O_1)\pi_i \tag{2.22}$$

Induction

Given a model λ, applying the product rule we have

$$\alpha_t(i)a_{ij} = p(O_1O_2...O_t, q_t = S_i, q_{t+1} = S_j) \tag{2.23}$$

Applying the sum rule,

$$\sum_{i=1}^{N} \alpha_t(i)a_{ij} = p(O_1O_2...O_t, q_{t+1} = S_j) \tag{2.24}$$

Extending the partial observation sequence to include O_{t+1}, relying on the fact that $p(O_{t+1})$ is conditioned only on q_{t+1},

$$b_j(O_{t+1}) \sum_{i=1}^{N} \alpha_t(i)a_{ij} = p(O_1O_2...O_{t+1}, q_{t+1} = S_j) \tag{2.25}$$

Now the right hand side is simply $\alpha_{t+1}(j)$, so

$$\alpha_{t+1}(j) = b_j(O_{t+1}) \sum_{i=1}^{N} \alpha_t(i)a_{ij} \tag{2.26}$$

Termination

$$\alpha_T(i) = p(O_1 O_2 ... O_T, q_T = S_i|\lambda) \tag{2.27}$$

using the sum rule,

$$p(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i) \tag{2.28}$$

Using equations 2.22, 2.26, and 2.28, we can thus compute $p(O|\lambda)$.

An Example

We once again return to Johnny and his sandwiches for an example. Let's encode peanut butter as 1, and toasted cheese as 2. We are given the observation sequence $O = \{1, 2, 2, 1, 1\}$. Table 2.1 shows the lattice of alpha values calculated for this example.

Table 2.1: Recursing the forward variable.

t	1	2	3	4	5
$\alpha_t(1)$	0.180000	0.013200	0.002808	0.013280	0.010377
$\alpha_t(2)$	0.240000	0.201600	0.130700	0.035627	0.011213
O_i	1	2	2	1	1

We will refer to this structure as the α-lattice. It is an $N \times T$ array, where the value of row i column t is $\alpha_t(i)$. For calculating $p(O|\lambda)$, we do not need to store the entire α-lattice, as only $\alpha_t(i)$ is needed to compute $\alpha_{t+1}(i)$. We can obtain $p(O|\lambda)$ by adding up the final values in the α-lattice $\alpha_5(1)$ and $\alpha_5(2)$ in this example, yielding 0.02159. As a sanity check, a simulation of one million trials estimates $p(O|\lambda)$ to be 0.02143, which is close enough. The MATLAB code for the Forward algorithm is in A.3.

Computational Complexity

From equation 2.26, and the example above, we can see that, in each time step, for every state we compute the sum of an $O(1)$ computation for all the states. Thus, for each time step, we have an $O(N^2)$, computation, implying that the entire computation time is $O(TN^2)$. This is much better than the naive $O(N^T)$ computation from 2.18.

Preventing Underflow

A serious problem for the forward algorithm in its present form is that the values of α get exponentially closer to 0 with increasing T, producing underflow errors. Intuitively, as the observation sequence length increases, the number of possible observation sequences increases exponentially, which means that the probability of any particular sequence must quickly approach 0.

One remedy is to re-scale the α values to keep them within some appropriate dynamic range, and using these scaling coefficients to compute $\log[p(O|\lambda)]$. This has the disadvantage of requiring the use of logarithms, but such use is necessary, as $p(O|\lambda)$ itself would underflow, for any reasonable T.[1]

Scaling occurs during the induction step of the forward procedure. We let $\alpha_t(i)$ refer to the unscaled α values, as in the above description of the forward procedure. We introduce $\hat{\alpha}_t(i)$ to refer to α values after scaling. We also require $\hat{\hat{\alpha}}_i(t)$ to refer to the α values that are computed from scaled alpha values at previous time steps, but are not yet themselves scaled. So, to modify equation 2.26,

$$\hat{\hat{\alpha}}_{t+1}(j) = b_j(O_{t+1}) \sum_{i=1}^{N} \hat{\alpha}_t(i) a_{ij} \tag{2.29}$$

For each time step $1 \leq t \leq T$, we now introduce a scaling coefficient c_t, with

$$c_t = \frac{1}{\sum_{i=1}^{N} \hat{\hat{\alpha}}_t(i)} \tag{2.30}$$

giving

$$\hat{\alpha}_t(i) = c_t \hat{\hat{\alpha}}_t(i) \tag{2.31}$$

The α-lattice must now consist of the scaled $\hat{\alpha}$ values. As we recurse through the computation of $\hat{\alpha}_t(i)$, building the α-lattice, at every step we multiply each $\hat{\hat{\alpha}}_t(i)$ by c_t. We thus scale the α values at t, ensuring $\sum_{i=1}^{N} \hat{\alpha}_t(i) = 1$, before we compute the values of the α-lattice for $t+1$. This keeps the values in the α-lattice within a sensible dynamic range. To obtain $p(O|\lambda)$ we can no longer simply sum over the last column in the α-lattice, as all the values have been scaled by multiplication with c_t. An inductive proof reveals that

$$\left(\prod_{t=1}^{T} c_t \right) \alpha_T(i) = \hat{\alpha}_T(i) \tag{2.32}$$

[1] There is an error in the Rabiner tutorial paper [68], and whenever we discuss scaling procedures we depart from that paper and refer to 'Correction to: 'A Tutorial on ...' [67]

Now, since $\sum_{i=1}^{N} \alpha_T(i) = p(O|\lambda)$, and $\sum_{i=1}^{N} \hat{\alpha}_T(i) = 1$,

$$p(O|\lambda) = \frac{1}{\prod_{t=1}^{T} c_t} \tag{2.33}$$

Logging both sides, and using that $\log(ab) = \log(a) + \log(b)$,

$$\log[p(O|\lambda)] = -\sum_{t=1}^{T} \log c_t \tag{2.34}$$

Code for the Forward algorithm with the above scaling procedure is in A.4.

2.3.4 The Viterbi Algorithm

The Viterbi algorithm solves problem 2. With an observation sequence $O = \{O_1...O_T\}$ and a model λ, we seek to find a $Q = \{q_1...q_T\}$ that maximizes $p(Q|O, \lambda)$. This is equivalent to maximizing $p(Q, O|\lambda)$, which can be seen using the product rule, noting that $p(O|\lambda)$ is a constant. The form of the algorithm is very similar to the Forward algorithm, and so we will focus only on the parts that differ. We introduce two new variables, δ and ψ. δ plays a similar role to α, but instead of denoting the cumulative sum of the probability of a partial observation sequence over all states ending in some particular state, it denotes the maximum probability of such a partial observation sequence. ψ denotes the partial state sequence that maximized that probability. The algorithm takes the form of a typical dynamic programming algorithm, where the δ values propagate forward along a lattice structure, with ψ values stored accordingly, and then a path is backtracked along the ψ lattice.

The Delta Variable

for $1 \leq i \leq N$

$$\delta_t(i) = \max_{q_1,...,q_{t-1}} p(q_1...q_{t-1}, q_t = S_i, O_1...O_t|\lambda) \tag{2.35}$$

$\delta_t(i)$ is the maximum joint probability of any sequence of states up until t ending in S_i, and the observation sequence up until t. Formally, $\psi_t(i)$ denotes the state that maximized $\delta_t(i)$, for a particular t and i.

Initialization

As there is no prior state sequence to maximize over, $\delta_1(i)$ is just the joint probability of the first state being s_i, and the first observation occurring whilst in that state, so

$$\delta_1(i) = b_i(O_1)\pi_i \tag{2.36}$$

$\psi_1(i)$ is set to a null value, 0, as no maximization has yet occurred.

$$\psi_1(i) = 0 \tag{2.37}$$

Induction

The induction step for the delta variable is similar to equation 2.26. For $2 \leq t \leq T$ and $1 \leq j \leq N$,

$$\delta_t(j) = b_j(O_t) \max_{1 \leq i \leq N} \delta_{t-1}(i)a_{ij} \tag{2.38}$$

$\psi_t(j)$ is then set to the argument that maximized $\delta_t(j)$,

$$\psi_t(j) = \arg\max_{1 \leq i \leq N} \delta_{t-1}(i)a_{ij} \tag{2.39}$$

Termination

We introduce P^* which represents the maximum of the final δ values,

$$P^* = \max_{1 \leq i \leq N} \delta_T(i) \tag{2.40}$$

and q_T^* which is the state that maximized the terminal δ values,

$$q_T^* = \arg\max_{1 \leq i \leq N} \delta_T(i) \tag{2.41}$$

Backtracking

We now backtrack to find the path through the ψ-lattice that led to q_t^*, so we begin there and work backwards. For $t = T-1, T-2, ..., 1$

$$q_t^* = \psi_{t+1}(q_{t+1}^*) \tag{2.42}$$

Q^* is then the path $\{q_1^*...q_T^*\}$ that maximized $p(Q|O, \lambda)$, henceforth, the Viterbi path.

Preventing Underflow

The Viterbi algorithm as stated above also suffers from underflow errors when T gets large enough, for the same reason the Forward algorithm does. We do not, however, need to implement such a complicated scaling procedure. Instead, we can use the fact that $\log(ab) = \log(a) + \log(b)$, and modify the initialization and recursion steps of the algorithm so that we compute the maximum of $\log[p(q_1...q_{t-1}, q_t = S_i, O_1...O_t|\lambda)]$.

The Log-Viterbi Algorithm

We introduce $\phi_t(i)$,

$$\phi_t(i) = \max_{q_1,...,q_{t-1}} \log[p(q_1...q_{t-1}, q_t = S_i, O_1...O_t|\lambda)] \tag{2.43}$$

The initialization, induction, and termination steps thus become

$$\phi_1(i) = \log b_i(O_1) + \log \pi_i \tag{2.44}$$

$$\phi_t(j) = \log b_j(O_t) + \max_{1\le i\le N}(\phi_{t-1}(i) + \log a_{ij}) \tag{2.45}$$

$$\log P^* = \max_{1\le i\le N} \phi_T(i) \tag{2.46}$$

The ψ variable is defined in terms of ϕ just as it was in terms of δ. This works because $\log(x)$ increases monotonically with x, and thus the argument that maximized $\delta_t(i)$ will also maximize $\phi_t(i)$. It should be pointed out that the values of $\log a_{ij}$, $\log \pi_i$, and $\log b_j(k)$ can be pre-computed, so, besides some initial overheads, the use of logarithms doesn't slow down the Viterbi algorithm at all.[2] Code for the Log-Viterbi algorithm is found in A.5. From now on, when we refer to the Viterbi algorithm, we will mean the Log-Viterbi algorithm.

2.4 Learning the Model Parameters

The solution to problem 3 is important, and will occupy the rest of this chapter. We will depart from the pedagogy in [68], and first describe a way to learn HMM parameters from the data that is conceptually simpler. After that, we will move onto the industry standard, the Baum-Welch algorithm.

[2] $\log b_j(k)$ can only be pre-computed in the case of discrete observations. When dealing with continuous outputs, we need $N \times T$ logarithms. This is true even in the case of vector valued observations. A version of the Viterbi algorithm with scaling similar to that used in the Forward algorithm would bring the number of logarithms down to T if the (log) probability of the most likely sequence is required, and 0 if just the sequence itself is required. This is at the expense of some additional additions and multiplications for normalization. This is not described in [68].

2.4.1 Overview

The problem is to find an HMM that 'fits' - in some sense of the word - one or many sequences of data. Many ways of doing this have been proposed [7, 34, 43, 56, 70]. We will begin with the simplest, called 'segmental k-means training' [34], and sometimes 'Viterbi training' [70].

2.4.2 Viterbi Training

Viterbi training fits an HMM to the data, O, by iteratively improving the parameters of some initial guess at a model, λ_G. Our initial guess, λ_G can be guided by knowledge of the process we are trying to model, if available, or it can be random. We then find the Viterbi path, Q, of λ_G through the data, maximizing $p(Q, O|\lambda_G)$. Q is used to recompute the parameters of the HMM, yielding a new HMM, $\overline{\lambda}$, with $\overline{A} = \{\overline{a}_{ij}\}$, $\overline{B} = \{\overline{b}_j(k)\}$. In the case where the data is a single sequence, O, the new probability of transiting to state j when in state i, $\overline{a}_{ij}$, is estimated by simply counting the number of times state j follows state i in the Viterbi path, and dividing this by the number of times state i is followed by any state. The new probability of symbol k occurring in state j, $\overline{b}_j(k)$, is obtained in a similar fashion, by counting the number of times state j occurs in the Viterbi path *and* observation k occurs at the same time in the observation sequence. This must, of course, be divided by the total number of times state j occurs. $\overline{\pi}_i$ can only be reliably estimated in this fashion when many training sequences are available, by simply dividing the number of times state i occurs at $t = 1$ in the Viterbi path by the number of sequences. Estimating $\overline{a}_{ij}$ and $\overline{b}_j(k)$ for multiple observation sequences doesn't change, except to note that the Viterbi paths for all sequences are computed from the same model, and new transition and emission probabilities are computed over the Viterbi paths of all sequences. All of the above can be accomplished with a single pass through each Viterbi path, as seen in the code in A.6, by keeping the various event counts in their respective matrices and vectors, normalizing $\overline{\pi}$ and ensuring that $\overline{A}$ and $\overline{B}$ obey stochastic constraints.

The procedure described above takes in a guess HMM, and finds an improved HMM. 'Improved' means something precise here, namely

$$\max_Q p(O, Q|\lambda_G) \leq \max_Q p(O, Q|\overline{\lambda}) \tag{2.47}$$

This is a particularly useful property. It can be exploited by iterating the procedure with the output $\overline{\lambda}$ becoming the 'guess' λ_G for the following iteration. $\max_Q p(O, Q|\lambda)$ will then increase monotonically with each new λ, until a local maximum is reached. We will not prove property 2.47, but a proof can be found in [34]. It is important to pay attention to what increases with every iteration - the objective function - as this is what differs between training algorithms. Here, it is the probability of the Viterbi path, $\max_Q p(O, Q|\lambda)$. This probability is also

called the Viterbi approximation, as is sometimes used to approximate $p(O|\lambda)$. In practice, we often limit the number of iterations, or set some convergence threshold and monitor $\max_Q p(O, Q|\lambda)$ between iterations, halting when the difference is less than the threshold.

An overview is informative. We have a model for some process, and some data. The model has hidden states, and we don't know which states produced which data points. What the Viterbi training procedure does is first estimate what the hidden states were, using the Viterbi algorithm, and then use that estimate of the state sequence to recalculate the model parameters. The similarity to the k-means clustering algorithm is so striking that the authors who invented it considered it to be an extension of k-means [34]. Clustering now occurs with each data point at each time step being assigned to a state, instead of each data point being assigned to a cluster. This similarity is made explicit in table 2.2, and is useful for understanding Viterbi training if one already understands k-means clustering.

Table 2.2: Similarity between k-means and Viterbi training.

	k-means	Viterbi training
Initialization	Guess cluster centres	Guess HMM parameters
Expectation	Assign data points to clusters with closest centres	Assign data points to states using the Viterbi algorithm
Maximization	Re-estimate cluster centres averaging over points assigned to each cluster	Re-estimate HMM parameters by averaging state transitions and outputs

2.4.3 A Classification Test

Before we move on to the Baum-Welch algorithm, we would like to run a sanity check on the Viterbi training method. For this purpose, we introduce the 'classification task', in which we try to decide which category a particular sequence of observations belongs to. With time series classification, we are given a number of category labels and some training sequences associated with each label. Our task is to assign labels to test sequences we have not seen before.

Many techniques using HMMs for classification exist [4, 72]. Throughout this thesis we will use a very simple one. First, train one HMM for each class using all the available training sequences. Then to classify a test sequence, compare each HMM to that test sequence, using a measure of how well the HMM 'fits' the sequence. The test sequence is then assigned to the category of the HMM with the best 'fit'. The particular measure of 'fitness' depends on what was maximized in the training procedure. In this section, we will use $\max_Q p(O, Q|\lambda)$, as we are testing Viterbi training.

We now encounter our first real set of data. Kudo et al. [52] recorded 9 native speakers of Japanese saying the /ae/ vowel sound a number of times each.

Identical to [52], 30 sequences per speaker were reserved as training data, and the rest (varying in number between speakers, but averaging about 41 sequences per speaker) were used as test data. The task is to identify which speaker produced each sequence of test data. Identifying a speaker from a single utterance of a single vowel sound is a simplified version of the more general speaker identification task, which seeks to tell speakers apart using samples of fluent speech.

Going from the continuous waveform of speech signals to a sequence of discrete symbols is a complicated topic, and such detail would distract from the task at hand, which is to test our Viterbi training procedure. Briefly, the data is supplied as sequences of 12 real valued Linear Prediction Coefficients (LPCs) per time step, with each sequence ranging from 7 to 29 time steps long. This is available at the UCI Machine Learning Archive under the name 'Japanese Vowels'. At each point in time, we have a vector of 12 real values, and we need to convert each such vector to a discrete symbol. An off-the-shelf Vector Quantization (VQ) MATLAB toolbox was used for this purpose. The number of symbols can be chosen. More symbols means less information lost during conversion, but more computation required, both during VQ and subsequent classification. The MATLAB function we used for VQ requires the number of symbols to be a power of 2.

So that it may be used effectively on such sequences, we first modify the Viterbi training procedure. With very many symbols, and quite short sequences, there are some symbols that will occur in the test sequences, but not the training sequences, of a particular subject. With Viterbi training, because the way the symbol probabilities are updated, any symbol u that doesn't appear in the training sequences will have all $b_j(u)$ for all states j set to 0, which means that the probability of any path through a sequence containing u will be 0. To mitigate such pathological behaviour, we add a small positive constant to each $b_j(k)$, and then normalize over all j. The unexpected appearance of u in a test sequence now reduces the probability of any state path through it, but it remains positive and non-zero.

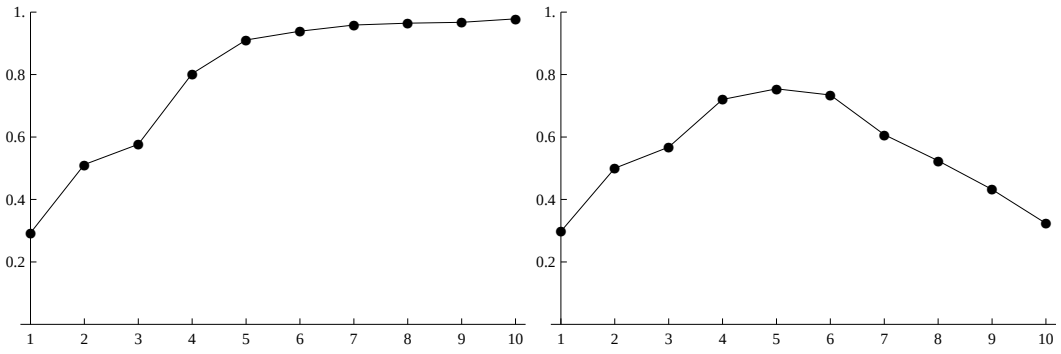

Figure 2.3: *Accuracy against the number of symbols.* Both plots show the accuracy of an 8 state HMM against the number of symbols. The y-axis shows the accuracy, while the x-axis shows n, where 2^n is the number of symbols. Thus 1 on the x-axis means there were just 2 symbols, and 10 means there were 1024 symbols. The left plot shows the results with the addition of a small positive constant to each $b_j(k)$, while the right plot is the same procedure without such addition.

For an objective measure of performance, we use classification accuracy, the number of correct classifications divided by the number of test sequences. Experimentation with different numbers of states showed an 8 state model provided the best results. The number of symbols chosen during the VQ process heavily influenced the accuracy. Figure 2.3 shows the performance of an 8 state HMM classifier as the codebook size increases. Note the difference made by the addition of a small positive constant to each $b_j(k)$. With such an addition, the accuracy increases monotonically with the number of symbols. Without it, the accuracy increases initially, closely matching the curve on the left, but begins to drop off as soon as the number of symbols gets great enough to ensure that some novel symbols occur in the test sequences.

For comparison, the original paper that the data is from [52] reports attaining 0.946 accuracy with their own classification technique, and 0.962 with a 5 state continuous HMM (most likely trained with Baum-Welch training, although they do not specify). Interestingly, we get 0.979 accuracy using more states and a discrete HMM trained with Viterbi training. We have already mentioned that this is averaged over 10 runs, and so it cannot be a result of lucky model initialization. Almost 98% of the time, we can tell which of 9 speakers uttered the Japanese vowel sound /ae/, from a single instance of that sound. Code for this classification test can be found in A.7

2.5 The Baum-Welch Algorithm

The final part of this chapter will deal with the Baum-Welch algorithm. The goal of the Baum-Welch algorithm is very similar to that of Viterbi training, but instead of finding a λ that maximizes $\max_Q p(O, Q|\lambda)$, we instead want to find a λ that maximizes $p(O|\lambda)$, the probability of the observation sequence given the model. The procedure has the same overall structure as Viterbi training. We first guess a model, and then use that model and the data to iteratively re-estimate the parameters of the model. The main difference between Viterbi training and the Baum-Welch algorithm is that in Viterbi training, we treat the Viterbi path as though it were the actual state path that produced the observations, and use that state path to compute the new model parameters. The Baum-Welch algorithm performs the same computation, but over all possible state paths, each weighted by its own probability. This cannot be done naively, as the number of state paths increases exponentially with T. This section describes the procedure for achieving such a computation efficiently.

2.5.1 Overview

We begin with a summary of how the rest of this chapter will proceed. We first define the 'Backward' variable β, and its recursive computation similar to α. We

then define $\gamma_t(i)$, the probability of state i occurring at time t, and show how this can be expressed in terms of α and β. We go on to define $\xi_t(i,j)$, the probability of being in state i at t, and state j at $t+1$, and show how this can be expressed in terms of α, β, and the parameters of the present HMM. We then show how γ and ξ can be used to re-estimate the parameters of λ, creating $\overline{\lambda}$. Discussions of scaling to avoid underflow and the use of Gaussian distributions to model real and vector valued observations will conclude the chapter.

2.5.2 The Backward Variable

$\beta_t(i)$ is the joint probability of the occurrence of the partial observation sequence from $t+1$ until T, given that the model is in state i at t.

$$\beta_t(i) = p(O_{t+1}...O_T|q_t = S_i, \lambda) \tag{2.48}$$

Similar to the Forward variable, we can derive inductive equations to compute β by recursing backwards from T.

Initialization

For $1 \leq i \leq N$,

$$\beta_T(i) = 1 \tag{2.49}$$

Induction

For $1 \leq i \leq N$ and $t = T-1, T-2, ...1$

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(O_{t+1}) \beta_{t+1}(j) \tag{2.50}$$

We can thus begin at T, and recurse backwards to compute $\beta_t(i)$.

2.5.3 The Gamma Variable

$\gamma_t(i)$ is the probability of being in state i at time t, given an observation sequence and a model,

$$\gamma_t(i) = p(q_t = S_i|O, \lambda) \tag{2.51}$$

Applying Bayes' rule,

$$\gamma_t(i) = \frac{p(O|q_t = S_i, \lambda) p(q_t = S_i|\lambda)}{p(O|\lambda)} \tag{2.52}$$

Separating O and relying on the fact that $O_{t+1}..O_T$ is conditionally independent of $O_1..O_t$ given q_t [12], we have

$$\gamma_t(i) = \frac{p(O_1...O_t|q_t = S_i, \lambda)p(O_{t+1}...O_T|q_t = S_i, \lambda)p(q_t = S_i|\lambda)}{p(O|\lambda)} \tag{2.53}$$

Using the product rule to combine the first and last terms in the numerator,

$$\gamma_t(i) = \frac{p(O_1...O_t, q_t = S_i|\lambda)p(O_{t+1}...O_T|q_t = S_i, \lambda)}{p(O|\lambda)} \tag{2.54}$$

Which allows expression in terms of α and β,

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{p(O|\lambda)} \tag{2.55}$$

This makes sense, as $\alpha_t(i)$ accounts for the partial observation sequence up until t, and $\beta_t(i)$ for the rest. Using the fact that γ is a correctly normalized distribution,

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{N} \alpha_t(i)\beta_t(i)} \tag{2.56}$$

2.5.4 The Xi Variable

$\xi_t(i, j)$ is the probability of being in state i at time t, and state j at $t + 1$, given an observation sequence and a model,

$$\xi_t(i, j) = p(q_t = S_i, q_{t+1} = S_j|O, \lambda) \tag{2.57}$$

Using the Bayes' rule,

$$\xi_t(i, j) = \frac{p(O|q_t = S_i, q_{t+1} = S_j, \lambda)p(q_t = S_i, q_{t+1} = S_j|\lambda)}{p(O|\lambda)} \tag{2.58}$$

Due to the conditional independence in HMMs, the first term in the numerator factorizes such that

$$p(O|q_t = S_i, q_{t+1} = S_j, \lambda) = p(O_1...O_t|q_t = S_i, \lambda)p(O_{t+1}|q_{t+1} = S_j, \lambda)p(O_{t+2}...O_T|q_{t+1} = S_j, \lambda) \tag{2.59}$$

Using this property, applying the product rule to the last term, and replacing some terms with their shorthand notation,

$$\xi_t(i,j) = \frac{p(O_1...O_t|q_t = S_i, \lambda)b_j(O_{t+1})\beta_{t+1}(j)p(q_{t+1} = S_j|q_t = S_i, \lambda)p(q_t = S_i|\lambda)}{p(O|\lambda)} \tag{2.60}$$

Combining the first and last terms in the numerator using the product rule,

$$\xi_t(i,j) = \frac{p(O_1...O_t, q_t = S_i|\lambda)b_j(O_{t+1})\beta_{t+1}(j)p(q_{t+1} = S_j|q_t = S_i, \lambda)}{p(O|\lambda)} \tag{2.61}$$

This can be rewritten as

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{p(O|\lambda)} \tag{2.62}$$

As $\xi_t(i,j)$ is a probability distribution,

$$\xi_t(i,j) = \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{\sum_{i=1}^{N}\sum_{j=1}^{N}\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)} \tag{2.63}$$

2.5.5 Re-estimation

As promised, γ and ξ can be used to re-estimate the parameters of the HMM, just as the Viterbi paths were used in Viterbi training.

The Initial State Probabilities

The expected number of times in state S_i at $t = 1$ is

$$\overline{\pi}_i = \gamma_1(i) \tag{2.64}$$

The State Transition Probability Matrix

The expected number of times transiting from state S_i to S_j over the expected number of times transiting from state S_i to any state is

$$\overline{a}_{ij} = \frac{\sum_{t=1}^{T-1}\xi_t(i,j)}{\sum_{t=1}^{T-1}\gamma_t(i)} \tag{2.65}$$

The Emission Probabilities

First we define $g_t(k) = \begin{cases} 1, \ if \ O_t = v_k \\ 0, \ otherwise \end{cases}$

The expected number of times in state S_j observing v_k over the expected number of times in state S_j gives us $\overline{b}_j(k)$

$$\overline{b}_j(k) = \frac{\sum_{t=1}^{T} \gamma_t(j) g_t(k)}{\sum_{t=1}^{T} \gamma_t(j)} \tag{2.66}$$

Convergence

$\overline{\lambda}$ is thus composed of $\overline{\pi}$, $\overline{A}$ and $\overline{B}$, each re-estimated from the old HMM λ and an observation sequence O. Just as with Viterbi training, this leads to an improvement in some objective function. For the Baum-Welch re-estimation formulae, the objective function is the probability of the observation sequence given the model, and

$$p(O|\overline{\lambda}) \geq p(O|\lambda) \tag{2.67}$$

This classic result was proven by Baum and colleagues in [42], and clearly implies that iterating this procedure will improve the quality of some model until convergence on a local maximum. This forms the key step in the learning procedure of most HMM applications.

2.5.6 Scaling

To prevent underflow, we rely on the scaling constants c_t computed for each $\alpha_t(i)$,

$$\hat{\beta}_t(i) = c_t \beta_t(i) \tag{2.68}$$

This works because the recursive computations that diminish α are the same as those that diminish β, and as c_t normalizes α, so will it serve to keep β within dynamic range. This scheme also has the useful property that, for all the re-estimation computations, the constants cancel out, and thus $\hat{\alpha}$ and $\hat{\beta}$ can be used instead of α and β. This is shown in [68], along with the errata available on Rabiner's webpage [67], and I will not repeat the exposition here.

2.5.7 Multiple Sequences

As with Viterbi training, we may want to train HMMs from multiple sequences. We call our collection of observations sequences O. Assuming we have K observation

sequences, $O = \{O^{(1)}, O^{(2)}, ..., O^{(K)}\}$, where $O^{(k)} = \{O_1^{(k)} O_2^{(k)} ... O_{T_k}^{(k)}\}$ is the k^{th} observation sequence containing T_k observations. The re-estimation formulae are presented in the following subsections.

The Initial State Probabilities - Multiple Sequences

$$\overline{\pi}_i = \sum_{k=1}^{K} \gamma_1^k(i) \tag{2.69}$$

The State Transition Probabilities - Multiple Sequences

$$\overline{a}_{ij} = \frac{\sum_{k=1}^{K} \sum_{t=1}^{T_k - 1} \xi_t^k(i,j)}{\sum_{k=1}^{K} \sum_{t=1}^{T_k - 1} \gamma_t^k(i)} \tag{2.70}$$

The Emission Probabilities - Multiple Sequences

First defining $g_t^k(h) = \begin{cases} 1, \ if \ O_t^k = v_h \\ 0, \ otherwise \end{cases}$

$$\overline{b}_j(h) = \frac{\sum_{k=1}^{K} \sum_{t=1}^{T_k} \gamma_t^k(j) g_t^k(h)}{\sum_{k=1}^{K} \sum_{t=1}^{T_k} \gamma_t^k(j)} \tag{2.71}$$

To incorporate scaling to prevent underflow, simply use $\hat{\alpha}$ and $\hat{\beta}$ when computing γ and ξ, and all scaling factors cancel appropriately. For a discussion see [68].

2.5.8 Gaussian Emission Densities

Up until now we have dealt only with discrete probability symbols. We now seek to extend this to the case where the observations from each state are vectors of real values. It is important to note that we do not have to change very much. As long as we can evaluate $p(O_t|q_t = S_j)$, we can compute α, β, γ, δ, ϕ, ψ and ξ. This includes the Forward algorithm, the Backward algorithm, the Viterbi algorithm (both standard and Log-Viterbi), and the re-estimation of π, A and B. The only thing that need change is how to model and evaluate $p(O_t|q_t = S_j)$, and how to re-estimate the parameters of such a model.

We will use a multivariate Gaussian distribution as our probability density function for each state. If the observations $\boldsymbol{O}_t$ are vectors of length D, then we will need a vector of D means $\boldsymbol{\mu}$ and an $D \times D$ covariance matrix $\boldsymbol{\sigma}$ for each state, in order to model the observation densities.

With the mean vector for state j denoted $\boldsymbol{\mu}_j$ and the covariance matrix for state j denoted $\boldsymbol{\sigma}_j$, we redefine $b_j(k)$ as

$$p(\boldsymbol{O}_t = k | q_t = S_j) = \frac{1}{(2\pi)^{D/2}|\boldsymbol{\sigma}_j|^{1/2}} \exp\left\{-\frac{1}{2}(\boldsymbol{O}_t - \boldsymbol{\mu}_j)'\boldsymbol{\sigma}_j^{-1}(\boldsymbol{O}_t - \boldsymbol{\mu}_j)\right\} \quad (2.72)$$

All that is required are the re-estimation formulae for $\boldsymbol{\mu}_j$ and $\boldsymbol{\sigma}_j$, which are computed in a similar fashion to their usual maximum likelihood solutions, but with each observation weighted by $\gamma_t(j)$, to accounts for the probability that a particular state produced that observation.

Weighted Mean

$$\overline{\boldsymbol{\mu}}_j = \frac{\sum_{t=1}^{T} \gamma_t(j) \cdot \boldsymbol{O}_t}{\sum_{t=1}^{T} \gamma_t(j)} \quad (2.73)$$

Weighted Covariance Matrix

$$\overline{\boldsymbol{\sigma}}_j = \frac{\sum_{t=1}^{T} \gamma_t(j) \cdot (\boldsymbol{O}_t - \boldsymbol{\mu}_j)(\boldsymbol{O}_t - \boldsymbol{\mu}_j)'}{\sum_{t=1}^{T} \gamma_t(j)} \quad (2.74)$$

These re-estimation formulae straightforwardly extend to handle multiple sequences and scaling to prevent underflow. We can thus model observation sequences where each observation is a vector of real values. More complicated observation models have been used, such as mixtures of Gaussians. These are extensively discussed in [68].

2.5.9 Chapter summary

We have presented the probability theory and computational details of HMMs, outlining solutions to the three canonical HMM problems. The Forward, Viterbi and Baum-Welch algorithms allow likelihood computation, state path estimation, and HMM training. The ability to choose observation distributions lets HMMs

model both discrete and continuous data. These methods and techniques will form the background to the discussion of numerous HMM applications, as well as the HMM structure discovery problem.

Chapter 3

Literature Survey

This chapter is divided into two sections. The first demonstrates the power and flexibility of HMMs. Through surveying numerous HMM applications, it shows both where and how they are applied to many different problems. The second section deals with structure discovery in HMMs, the primary topic of this thesis. It begins with a discussion of model selection criteria. After that, state merging and state splitting techniques are discussed, ending off with a brief glimpse of some other approaches.

3.1 Applications of Hidden Markov Models

The popularity of HMMs as pattern recognition tools grew through their success in speech recognition, and they are often introduced in this context. They have since been applied to many other domains, finding success almost wherever a process produces data that varies with time. This survey of HMM applications is intended to give the reader an idea of their scope of application, as well as the inventive ways HMMs have been used to tackle problems. We will attempt to describe how the model is applied to each domain, including a discussion of the feature vector, HMM type, and what the various model components represent. Often, however, the application being discussed requires specialized domain knowledge, and in such cases brevity prohibits a detailed treatment.

The reader should note that the applications surveyed here are not intended to be representative of the impact HMMs have made in each discipline, nor any factor besides the author's interest. Speech recognition should be conspicuous by its absence. This omission is intentional, as the literature abounds with such discussions.

3.1.1 Gesture Recognition

The domain of gesture recognition ranges from recognizing a few isolated hand signs - possibly useful for device control - right through to continuous sign language

recognition. The latter is a difficult task, involving thousands of words and sharing the complexities of speech recognition, from modeling the syntactic structure of the language through to handling co-articulation effects.

Gestures are mostly performed with the hands. Although facial expression does play an important semantic role in sign language, we will ignore such complexities here. With this restriction, gesture recognition requires learning a mapping from hand movements to words. The first step in constructing such a mapping is finding a good way to encode hand movements.

Encoding Hand Movement

Two main approaches to encoding hand movement appear in the literature. The first uses computer vision techniques to track the hands, which are either bare, or wearing gloves. This approach is exemplified in Starner et al. [79], who experiment with both desk mounted cameras, and wearable hat mounted cameras to track the hands. The hand blobs[1] are identified in the scene, and their positions, various moments and shape features are extracted, and the feature vector at each timestep is just a list of such values. Starner et al., for example, use a 16 element feature vector containing, for each hand, the x and y positions, the changes in x and y positions between successive frames, the area in pixels, the angle and length of the first eigenvector of the hand blob, and the eccentricity of the bounding ellipse.

The other way to track hands uses purpose built tracking hardware that directly measures the position of different parts of the arms or hands. Vogler et al. [91] use an Ascension Flock-of-Birds tracking device that uses body mounted sensors to measure position and orientation relative to a fixed magnetic field. The feature vector is just a list of these values. Kadous [35] uses a combination of Fifth Dimension Technologies data gloves to track individual finger movements, as well as two Ascension Flock-of-Birds position trackers, yielding a 22 element feature vector with, for each hand, x,y and z positions, pitch, roll and yaw orientations, as well as finger curl measurements for all five fingers.

Modeling Hand Movement Over Time

Since the elements in the feature vector are real valued, HMMs with Gaussian outputs are used. The most common strategy is to use left-to-right HMMs where the transition matrix is upper triangular; no state can transit to any previous state. Beyond this, there is much variation in how HMMs are used, including explicit modeling of co-articulation, and attempts to model smaller units than whole signs; the sign language equivalent of phoneme modeling.

[1] 'Blob' is a term of art in computer vision.

3.1.2 Analysis of Behaviour

HMMs have been employed to model the behaviour of both animals and humans in many different domains. The motivation for such modeling varies, but it is usually hoped that the parameters of the model as well as the inferred state paths will tell us something interesting about the properties of the behaviour.

Animal Behaviour

HMMs have been used to automatically segment mouse activities from video data [21]. Laboratory experiments involving mice are increasingly automated, with the positions and orientations of the mice being tracked. This produces a large amount of data, which is then typically manually segmented into a few discrete activities. Vetrov et al. train HMMs on some manually segmented data, and use these to automatically segment the rest of the data, yielding imperfect but promising agreement with hand-segmented data.

HMMs have been used to model the effects of feeding on locomotory behaviour in locusts [53]. Captive locusts were assigned to two groups, either fed or unfed. Their behaviour was examined at discrete intervals, and recorded as either locomoting or not. HMMs were used in different ways to model such observations. A single HMM per group of locusts was used, with multivariate binary observations for each state describing the group behaviour. An HMM with Poisson distributed outputs was also used, and the total number of locomoting locusts at each timestep was treated as count data.

There is a larger literature on HMMs applied to animal behaviour, but brevity prohibits a detailed discussion. Briefly, HMMs have been applied to stride segmentation from accelerometer data in thoroughbred racehorses [64], movements of caribou [25], pecking behaviour in pigeons [63], the clustering of whale song [16], and the prediction of wolf kill sites [26].

Human Behaviour - Eye Movement

Human eye movement patterns have been modeled with HMMs. In [75], participants performed three different reading related tasks: searching for words, finding a sentence that answers a question, and choosing an interesting topic from a list. Their saccades and fixations were recorded by an eye tracker, and were discretized to create time series. A separate model was trained for each task type, and the ability of the models to distinguish between task type was experimented with, yielding better results than logistic regression. The parameters were also examined to make inferences about processing differences between tasks, and the HMM states turned out to correspond to cognitive states discussed in the literature, such as 'scanning', 'reading' and 'decision' states.

Eye movements have also been modeled with HMMs in other settings. Pilots in flight simulators have had their eye movement behaviour modeled in an effort to understand the differences in instrument scanning patterns between pilots of different experience levels [29]. An eye tracker was used to identify which instrument the pilot was looking at, and an HMM with discrete observations was used to model the instrument sequences. HMMs were used in a similar study, this time on crew members in a space shuttle simulation [30]. As an interesting addition, a plot of the log probabilities of each observation was included, showing that these drop drastically during critical events, such as side rocket booster separation, or system malfunctions, indicating that the models fail during such events. This is not surprising, but it does serve to validate the methodology. It also suggests that such models of eye tracking might be useful for detecting pilot confusion and potential catastrophe.

Understanding Intention

Kelley et al. [36] use HMMs to understand intentions. The domain is restricted to the interaction between two agents, and the possible behaviours are limited to meeting, passing by, following, dropping off an object, and picking up an object. The goal is to accurately decide which activity will occur by monitoring the changes in angle and distance between two agents, but from the perspective of one of those agents. In an approach inspired by theory of mind work on intentions, a robot that can perform these activities is used to train the HMMs. The robot records changes in angle and distance between itself and another agent (a human) when performing each of these different activities. The observations are discretized into either increasing, decreasing, constant, or unknown angles and distances, and a discrete HMM is trained for each of the activities, capturing the temporal structure in the observations. The robot is then placed to observe the activity of two humans performing these same activities, and estimates the changes in angle and distance from one person's perspective through a geometric transformation. The HMMs trained from the robot's perspective while it was the actor are then used to successfully classify the human behaviour, deciding, for example, whether or not the two people will meet some time in advance of them actually meeting. This work contributes to human-robot interaction research, shows that HMMs can be used to model interactions between humans, and demonstrates the plausibility of the theory of mind that inspired it.

HMMs have also been used to classify human trajectories in video footage from a surveillance camera, detecting abnormal behaviour such as turnpike jumping or fighting [57]. They have been used to model intersection driving behaviour, providing better prediction of vehicle trajectories than regular car following models [97]. They have been used for human gait analysis, modeling the patterns in accelerometer and pressure sensor data, with the goal of detecting when someone walks with an abnormal gait [17]. Finally, the actions of player types in massively multiplayer online games have been modeled with HMMs in an attempt to classify players into

different categories [54].

3.1.3 Neural Signal Processing

Neuroscience produces many different kinds of time series. Electroencephalography (EEG) is the use of electrodes placed on the scalp to record electrical activity generated by the spiking of large numbers of neurons. EEG is routinely used in clinical settings to diagnose epilepsy type, but it is also a common tool in brain related research. The recordings of many electrodes form a vector of real values for each sampled time step. Electrocorticography (ECoG) is similar to an EEG, but with the electrodes placed directly on the surface of the brain, yielding better spatial resolution as the electrical activity from the brain doesn't have to pass through the skull. At a much finer spacial resolution, electrodes can also be inserted into the brain and the electrical activity of individual neurons can be recorded. Neurons have a characteristic 'spiking' behaviour, where pulses of electrical activity propagate from the soma along the axon. This property allows us to simply record the spike times rather than the entire signal. After discretizing the time axis, this leads to a time series of binary observations. Recordings of spike times from many neurons lead to a vector of binary values at each time step. Both the continuous signals from EEGs and ECoGs as well as the discrete signals from spiking neurons have been modeled with HMMs.

Spiking Neurons

One of the ways neuroscience progresses is by understanding how particular neurons in the brain relate to stimuli from the world [22]. These relationships are typically established through repeated trials in experiments where an animal is exposed to a particular stimulus whilst having some neurons in its brain recorded. There is much between-trial variation, and so the recordings of neural spikes from different trials are overlayed, using the stimulus onset as a reference, and the spiking activity is averaged out across trials, treating this variation as noise. The result is called a peristimulus time histogram (PSTH). The problem with this approach is that some of this variation might not be noise at all, and certain temporal structure in spike patterns can be lost in PSTHs. Jones et al. [33] show exactly this, using HMMs to model the underlying state dynamics of a population of neurons in the gustatory cortex of rats receiving taste stimuli in repeated trials. A multivariate discrete HMM is used, with each state specifying a firing rate for each neuron in the population. A separate HMM was used to model each taste stimulus: sucrose, quinine, citric acid, and salt. The results show that the neural spiking activity in the population switched between states, with the state sequence being conserved across most trials. Much effort went into showing that such behaviour was not just an artifact of applying HMMs to this kind of data. Furthermore, HMMs proved better than other commonly used approaches at predicting which tastant had been received by the rat. About 65% of the time, we can tell which of 4 substances the

rat received just by monitoring the spiking activity of 10 neurons from that rat's gustatory cortex.

HMMs have also been used to solve the spike sorting problem [31]. Neural activity is recorded with electrodes inserted into the brain. As neurons are close together, these electrodes typically pick up electrical activity from more than one neuron. This leads to the spike sorting problem: How can we tell which spikes came from which neurons? This problem is further complicated by the use of multi-electrode arrays, where many electrodes are used to record spikes from many different neurons, but the electrical activity for each neuron ends up affecting more than one of the electrodes. The good news is that each neuron has its own characteristic spiking waveform. An HMM approach to spike sorting involves building a generative model of the electrical activity from each neuron. A circular HMM is used, where the model remains in a quiescent state 1, until a spiking event begins. From then, the model progresses from state 2 to state K, with all transition probabilities set to 1, until the model returns to the quiescent state again. The outputs for each state are Gaussians. Figure 3.1 depicts this graphically. When modeling multiple neurons, the activity on each electrode is simply the sum of the activity from each neuron that affects that electrode, and this is captured by the use of a model known as a factorial HMM. Numerous assumptions and approximations are invoked to allow for tractable computation, and the results compare favorably to other state of the art spike sorting algorithms.

HMMs have also been applied to EEG and ECoG Brain-Computer Interfaces (BCIs). The goal is to allow the user to control a device without moving any parts of their body, by modulating only mental signals. Typically, electrodes are placed over the motor cortex and the user engages in 'movement imagery': imagining moving a particular body part, which causes activation in the motor cortex. The task is often set up much like an isolated word recognition task, with the participant repeatedly performing movement imagery of a few different kinds (eg. tongue or toe), while scalp electrodes record the electrical activity. The goal is to assign unlabeled recordings to the correct class. A few papers have used HMMs for this task [18, 44, 61], although the technique has not proven popular in the mainstream literature.

3.1.4 Bioinformatics

HMMs have shown incredible popularity in bioinformatics, with varied applications and success [11]. Krogh et al.[38] show how to use HMMs to model protein families. A discrete HMM is used, where individual proteins are amino acid sequences, with an alphabet of 20 possible amino acids. The model, called a profile HMM, has a specific structure and is constructed in a left-to-right fashion, with start and end states that produce no outputs. The rest of the model is made up of 'match', 'insert', and 'delete' states for each position in the amino acid sequence. A single profile HMM is built for a family of proteins, using the Baum-Welch algorithm on a

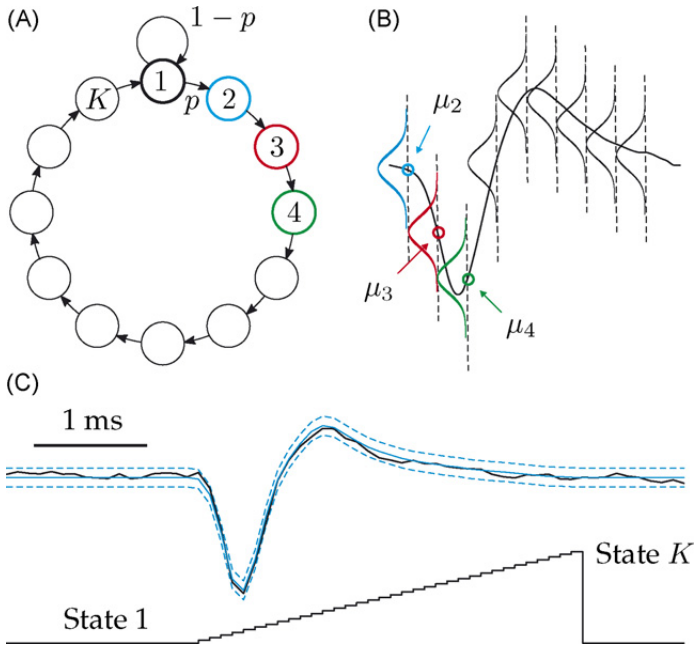

Figure 3.1: *An HMM model of a neural spike.* A) depicts the states of the generative model of the neural spike. Note that all state transitions are set to 1, except the self transition of state 1, and the transition from state 1 to state 2, which together must sum to 1. This means that the entire transition matrix can be described with a single parameter, p, which corresponds to the spike rate. B) shows the Gaussian outputs of each state. C) shows an example fit to the data. (Picture from [31])

number of training proteins. Individual proteins can then be 'aligned' to the model using the Viterbi algorithm, and the state path tells the details of the alignment. A 'match' state at a particular position indicates that the protein corresponds well with the model at that position. A 'delete' state (which is a state with no output) indicates that the protein being aligned to the model is missing an amino acid at that position, and an 'insert' state indicates that an extra amino acid is present in the protein in that position. Figure 3.2 depicts a (shortened) profile HMM.

HMMs have also been applied to the gene prediction problem. In a phenomenon known as pre-mRNA splicing, certain subsequences of RNA within a gene - called introns - are removed, and the remaining subsequences (exons) are fused such that their ordering is preserved. The gene prediction problem in bioinformatics is to classify which segments of genetic material are introns, and which are exons. Birney [11] briefly surveys the application of HMMs to this problem. Discrete HMMs are used to capture the statistical properties of the nucleotide subsequences.

HMMs have also been applied to the analysis of gene expression time series. Microarrays measure the expression levels of a particular gene, and time series of such measurements have been constructed. Such time series allow the identification of gene regulation relationships, where a gene's expression affects the expression of other genes. Multivariate Gaussian HMMs have been applied to this problem,

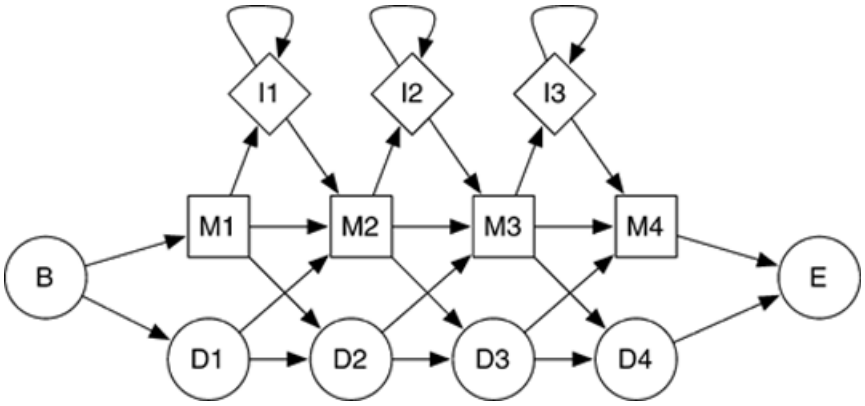

Figure 3.2: *A profile HMM.* Insertion (I), match (M) and deletion (D) states of a profile HMM. See text for details.

directly modeling the expression levels of pairs of genes over time [94].

3.1.5 Ecological Modeling

HMMs have found applications in ecological modeling. Viovy et al. [90] use HMMs to capture vegetation dynamics, modeling the normalized differential vegetation index (NDVI), which can be extracted from radiometric satellite measurements and is sensitive to changes in vegetation density. NDVI measurements have noise introduced by atmospheric conditions and cloud cover. They are partially determined by complicated underlying processes which govern vegetation growth. Lastly, these processes change seasonally, so NDVI time series have important temporal structure, making them plausible candidates for HMM analysis. HMMs were applied to NDVI data from the West African savannah collected weekly over 6 years, after some preprocessing steps. Using domain knowledge, the authors decided to use a 3 state cyclic HMM to model dormant periods with almost no vegetation, growth periods which usually occur during rainy seasons, and periods of senescence. Each state was intended to model a rate of change in NDVI, rather than NDVI itself, so the temporal derivative of NDVI was used as the observation sequence. State output distributions were histograms of NDVI values, effectively discretizing a continuous variable. Various model verification steps were performed, including demonstrating that the estimated state sequences correspond to realistic seasonal fluctuations. When the same basic model was trained on data from different regions, parametric variation in the model corresponded to known variation in savannah type.

Li et al. [49] use clustering and structure discovery techniques in a preliminary analysis of the dynamics of various indices of plant growth gathered from a collection of marsh sites. Given variation in the marsh sites, one might expect different models to be appropriate for different sites, hence the clustering requirement. Indeed, for some configurations of data, two clusters with quite different dynamics were identified in the data. More recently, Tucker et al. [87] examine the difference

between Markov chain and HMM approaches to modeling vegetation dynamics. Using mostly synthetic data, they identify certain dynamics where Markov chains are insufficient, but HMMs perform well.

3.1.6 Medicine

We will divide the use of HMMs in medicine into two branches. One uses so-called 'structured' HMMs as stochastic epidemiological models [55], and the other uses regular HMMs for time series analysis. Structured HMMs usually require information about the underlying process being modeled, and this is used to parameterize the model. This can yield a drastic drop in the number of model parameters, allowing a model with an arbitrarily large number of states to have, in some cases, as few as 4 parameters. This cannot be achieved without extending the HMM framework beyond what has been presented so far and we present an overview of such an extension below.

Epidemiology

McBryde et al. [55] use structured HMMs to model outbreaks of vancomycin-resistant enterococci (VRE). VRE can occur sporadically, but is typically contracted through nosocomial transmission, which can lead to epidemic outbreaks. A structured HMM was used to model VRE occurrence in a ward with 68 patients at any given time, where the state the model was in, I, represented the number of infected patients in that ward. The model thus had 68 states. Normally, estimating the parameters of a 68 state HMM requires large amounts of data. Instead, structured HMMs use a well understood model of transmission to parameterize the HMM. Consider a hospital ward with N patients. Assume that VRE can arise under three circumstances, either through (a) an infected patient entering the ward, (b) VRE originating in a patient already in the ward, or (c) VRE being transmitted from one patient in the ward to another. We group (a) and (b) together and assume they occur with a constant rate per unit time, proportional to the number of uninfected patients in the ward, $v(N - I)$. (c) occurs with a constant rate β proportional to the number of infected patients, and proportional to the number of uninfected patients, $\beta I(N - I)$. We assume that the ward can only lose an infected patient through discharging them and replacing them with an uninfected patient, and that this occurs at a rate μ. These three parameters actually describe the entire transition matrix of the HMM. From them, we can compute the probability of I increasing, decreasing, or staying the same[2]. For a ward of 68 patients, instead of 68×67 parameters to estimate for the transition matrix, we have just 3. Of these 3, the per unit time probability of patient discharge can be estimated directly from the patient data, and does not require re-estimation. So far, we have described a Markov model. To make it a full HMM, we allow for imperfect detection of VRE,

[2] [55] use a small positive 'fudge factor' to allow for jumps in infection numbers > 1.

and the inclusion of a parameter D, the probability of being known to be infected given that the patient is actually infected. We can thus take hospital infection data and, through fitting the HMM, obtain estimates of different infection rate parameters, even when we can only imperfectly detect the number of infected patients in a ward. Other methods to calculate such parameters require expensive genotyping to separate spontaneous infections from transmitted ones. McBryde et al. [55] show that the HMM method returns exactly the same results as glycopeptide resistance genotyping. Other examples of structured HMMs as epidemiological models also occur in the literature [19, 77].

Other Medical Applications

HMMs are frequently used to model time-series data in medicine. Rasku et al. [69] use HMMs to model the balance signals from subjects standing on a tilting force platform, which measures x and y swaying, as well as mass. A group of 32 healthy controls and 32 patients with balance affecting disorders stood on the force platform while it changed tilt angle, and the balance signals were recorded. HMMs were used to model these signals, and the resulting models were used to successfully discriminate between healthy subjects and patients. Al-Ani et al. [2] use HMMs to model polysomnography data to diagnose sleep disordered breathing. Such diagnosis typically involves careful expert inspection of an entire polysomnogram, which comes in 10 minute segments for up to 8 hours of recording. The observations used by the HMM were upper airway flow, oesophageal pressure and gastric pressure. HMMs trained on data from different classes of sleep disordered breathing were reported to effectively discriminate between classes. Van den Hout et al. [88] use HMMs to estimate life expectancy by modeling transitions between normal states, impaired states, and death. Altman et al. [3] use a 2-state HMM to model multiple sclerosis lesion counts, intended to capture the dynamics of chronic diseases which transit between relapsing and remitting states. Finally, [5] use HMMs to model the response of migraines to treatment.

3.1.7 Weather

Weather is another domain where noisy observations are generated by a complex underlying process. Betro et al. [9] use HMMs to model rainfall in Central East Sardinia. The area is affected by extratropical cyclones, and thus rainfall has large variability. Weibull distributions are used to model the outputs of the HMM, as these flexibly cater for extreme variation. Greene et al. [28] use HMMs to model the daily monsoon rainfall in India. The observation model is a mixture of a dirac delta function at 0, and two exponential distributions. This allows the proportion of non-rainfall days to have a separate mixing parameter to the two exponential models. A 4 state HMM was used, and the resulting states corresponded to distinct periods in the seasonal progression of the monsoon.

3.1.8 Fault Detection

HMMs have been used to detects faults in industrial processes; from nuclear power plants [39] to pasteurization [85]. Typically one models a vector of measurements from the process, training separate models corresponding to specific kinds of fault. These models can then be used for online fault detection. Ge et al. [27] mount a strain sensor on a particular part of a stamping press and monitor the strain over time. The machine can undergo several different kinds of faults. Each fault a specific problem with the material being fed into the stamping press (it has run out, is misaligned, is too thick etc...). HMMs are trained to model signals from the strain sensor during such failures, and they are shown to classify novel sequences with accuracies between 80 and 90%. The HMMs were so-called auto-regressive HMMs, which model serial dependence in the observations for each state. Much of the literature on fault detection in machinery follows very similar lines [1, 39, 83, 85, 95]. Often simulations of the processes are used, rather than the processes themselves. This is understandable. Any attempt to gather enough data to train an HMM for identifying faults in nuclear power plants is probably ill advised. Whether the HMMs trained on simulated data will succeed in the real world is an open question.

3.1.9 Music

HMMs have been applied to many different problems in music. They have been used for pitch tracking; the identification of notes in acoustic data [6]. Flexer et al. [24] use HMMs to measure the similarity between two pieces of music, a measure required for content based retrieval and clustering systems. Noland et al. [60] use HMMs to estimate the key of a piece of music. Pikrakis et al. [66] classify raw audio recordings of monophonic instruments into 12 distinct 'musical patterns' in traditional Greek music.

A very novel application is presented by Simon et al. [76]. They use HMMs to automatically generate chord sequences for recorded vocal melodies. An HMM is constructed where the states correspond to particular chords, and an abstracted representation of melody constitutes the observations. Users record a vocal segment, and the Viterbi algorithm extracts the most likely chord path corresponding to that melody. The way the HMM is trained is instructive, as it is trained from a corpus of lead sheets consisting of chord lists and melody score, rather than raw audio data. First, some simplifying assumptions are introduced. All music pieces are transposed into the key of C, and all chords converted into their corresponding 3 note triads. This leaves 5 chords per pitch. All octave information is discarded, leaving 12 distinct pitches, and thus 60 chords. The lead sheet training corpus is then divided into 2 musical modes, major and minor. Within each mode, a chord transition matrix is constructed by simply counting the number of times one particular chord is followed by another. This describes the Markov chain governing chord evolution. The observations produced by each chord are simply histograms of the frequencies of melody notes produced while that chord is played. This is also just

counted off the lead sheet data. Note that this discards all temporal melody information during each chord, which is another simplifying assumption. The user's recorded vocal line is then broken into regular measures, and the frequency histogram of each measure is constructed. This will create the series of observations to which the HMM must be fit. After some bookkeeping to account for key differences, running the Viterbi algorithm over this sequence of observations using the HMMs estimated directly from the lead sheet data produces audibly pleasing chord sequences. The quality of such chord annotations were rated by listeners and compared favorably to expert annotations. In addition, the user can select which mode they want the chord sequence to come from, in a way that allows for blending, which is simply the averaging of the transition matrices for the two modes. Finally, they included a parameter that weighted the importance of the observations over the transition matrices in the likelihood calculation. They call this the 'jazz' factor. One can adjust this to only take the observation for each measure into account, producing chord sequences that disregard chord transition structure, sounding more surprising, or one can adjust it to totally disregard the observations and produce chords directly from the transition matrix, resulting in common chord sequences that are not sensitive to the melody. Gradations between these extremes produce the most musically interesting results.

3.1.10 Other

A few other applications do not fall into any of the above categories, but nevertheless deserve brief mention. HMMs have been used to model dynamic backgrounds from video footage, in order to accurately segment foreground regions [80]. Srinivasan et al. [78] use HMMs to recognize behaviour of opposing agents in a game of robot soccer, and another HMM to select appropriate actions. Beyreuther et al. [10] use HMMs to automatically detect and classify earthquakes and other seismographic activity into distinct classes from seismogram data. Wang et al. [92] use HMMs to automate Chinese business card recognition, demonstrating superior performance to commercial optical character recognition for this task. Toreyin et al. [86] detect flames in video sequences using HMMs. Senior [71] uses HMMs to classify fingerprints into different classes, an activity that should be useful for indexing large fingerprint databases. Lee et al. [45] tackle the spam deobfuscation problem with HMMs. To avoid spam filters, many spam messages involve corrupted words which are interpretable to humans, but not recognized by some spam blocking software. Spam deobfuscation requires building word models to reconstruct words from corrupted words. Finally, to conclude our survey of HMM applications, Bond et al. [13] use HMMs to forecast political turmoil in Indonesia.

3.2 Structure Discovery Techniques

We now come the structure discovery problem, the topic of this thesis. Baum-Welch and Viterbi training both adjust the parameters of a model to improve its fit to the data. An initial model is specified and, through iteration, a local maximum in the appropriate objective function is reached. The quality of models trained by these iterative training procedures is hostage to the vagaries of the initial model specification, both because they only guarantee local maxima, and because they can only change the model parameters, not the structure of the model itself. Neither of these are trivial problems. Models in low likelihood local maxima degrade classification and prediction performance. Simple models with too few states cannot adequately capture the detail in the data, but too complex models can result in over-fitting, capturing the idiosyncrasies of the training set and leading to poor generalization to novel sequences.

Despite these difficulties, the creation of an initial model is usually left to the skill of the individual researcher. Sometimes knowledge of the underlying process can inform the design of the initial model. In speech recognition, the order of phonemes and the dynamics of the sub-phonemic units are usually well understood, so plausible initial models can be constructed. This is better than guessing but it is time consuming, requires expert knowledge, and allows human preconception to intrude on model creation. Most importantly, it is often not an option in domains where there is little knowledge about the process generating the data. Solutions to the structure discovery problem - as we use the phrase - both discover the appropriate model size and tackle the problem of model initialization. This thesis deals with automating the creation of HMMs from the data alone, removing the need to specify an initial HMM. Previous research into HMM structure discovery will be surveyed in this chapter.

Discovering HMM structure is an area of active research. The problem has been approached with many different strategies, each with their own benefits and restrictions. This chapter will give an overview of these attempts, without going into too much detail. We will first discuss some criteria for model selection, and then outline some algorithms for structure discovery in the literature.

3.3 Model Selection Criteria

Before discussing techniques for finding good models, we need to say what we mean by 'good'. We cannot just use the regular model likelihood, $p(O|\lambda)$, as more complex models typically fit the data better. We need a tradeoff between the fit of the model to the data and the number of model parameters; a statistical Occam's razor. We also need to take the number of data points into account, as more fine grained complexities in the data are only discovered as the amount of data grows [15]. In this thesis, we will use two such criteria from the literature, Akaike's Information

Criterion (AIC) and the Bayesian Information Criterion (BIC). Both of these are minimized by greater model likelihoods and fewer parameters.

3.3.1 AIC

AIC has its roots in information theory. It is an asymptotically unbiased estimator of relative Kullback-Leibler divergence [15].

$$AIC = -2 \ln L + 2k \tag{3.1}$$

where L is the maximized log-likelihood, and k is the number of free parameters in the model. Because of stochastic constraints with some HMM parameters, not all parameters are free. Rows in the transition matrix must sum to 1, for example, and as such, each row has only $N - 1$ free parameters. A second order bias correction improves AIC for smaller samples,

$$AICc = -2 \ln L + 2k + \frac{2k(k+1)}{n-k-1} \tag{3.2}$$

where n is the number of data points. As n increases, the second order correction term tends to 0, so AICc can replace AIC, no matter what the sample size. From now on, when we say AIC, we will mean AICc, always including the correction for small samples, as recommended by [15].

3.3.2 BIC

BIC is also a penalized model selection criterion, optimal under different assumptions to AIC.

$$BIC = -2 \ln L + k \ln n \tag{3.3}$$

There is a large and confusing discussion in the literature over which selection criterion is appropriate to which situation [15]. BIC penalizes extra parameters more heavily than AIC, so its use results in simpler models. We will circumvent this discussion entirely, treating AIC and BIC as useful heuristics, both ultimately accountable to test data classification accuracy. There are also computational reasons why we might prefer simpler models in domains where speed is important, such as real-time speech recognition.

3.4 A Naive Approach

As an introduction to the problem, a naive approach will first be discussed. We simply choose a range of model sizes, train one model for each each possible model

size, and use the model with the best AIC or BIC score. This is often done in practice [50]. The main problem with this approach is that regular Baum-Welch training is very sensitive to initial model parameters, and the model with the best number of states will often not be found. Discovering the structure of an HMM is not just about finding the correct number of states. It also requires avoiding local maxima during parameter optimization. Structure discovery techniques often achieve better likelihoods than Baum-Welch training even when the number of states is decided in advance [74].

3.5 Structure Discovery

Discovering the structure of an HMM amounts to searching the structure topology and parameter space for good models. For data of reasonable complexity, a full enumerative search is intractable, so various heuristics, approximations and greedy strategies are resorted to. A very important distinction to make in the HMM structure discovery literature is between state merging and state splitting approaches. Briefly, state merging approaches begin with a very complex model and iteratively merge states to reduce the model complexity, until a turning point in some model selection criterion is reached. The details of when and how states are merged distinguish the approaches. We will begin our literature survey with one state merging algorithm, from [82], and then move onto state splitting approaches. State splitting approaches begin with a very simple model, and increase the model size by iteratively selecting a state, and splitting it into two distinct states. Again, there is large variation in deciding which states to split, and how to adjust the model parameters after the split. Splitting is typically stopped when extra splits no longer improve model selection criteria. We will discuss many techniques of this kind, as they are more akin to the technique we will propose in chapter 4. Lastly, some techniques combine state splitting and state merging. We will treat these primarily as state splitting techniques, as they begin with simple models and introduce complexity through state splitting, while state merging only comes into play later.

3.6 State Merging

One early attempt at discovering the structure of HMMs from the data is due to Stolcke and Omohundro [81] and [82]. Working in a Nondeterministic Finite Automata framework, their HMMs have some superficial differences to those typically encountered in a Bayesian networks framework. They propose a state merging approach to HMM structure discovery, which works for discrete observation data only, but could potentially be generalized to handle continuous observations. They begin with an HMM that describes the data exactly. There is a separate state for each data point. Each sequence is represented by a chain of states, with each state only transiting to the very next state in the chain. π_i for the first state in each chain

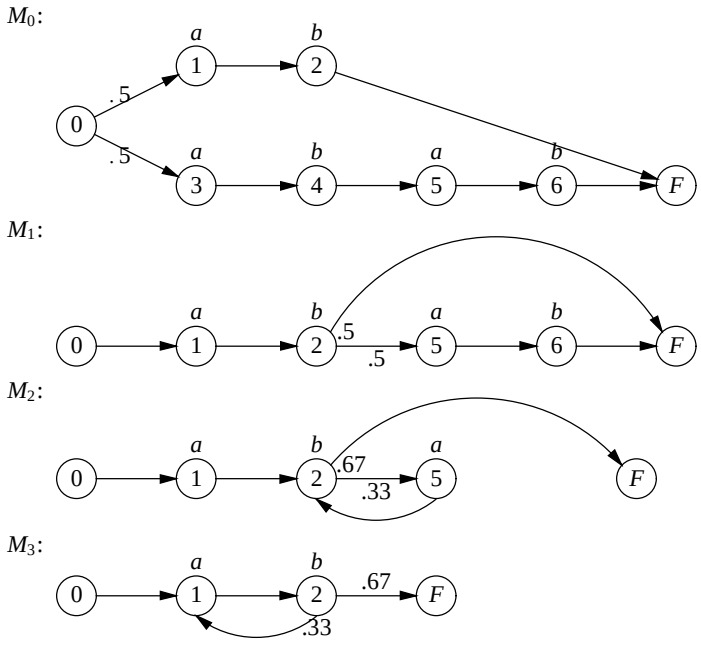

Figure 3.3: *Model Merging.* M_i is the i^{th} iteration of the model merging procedure. The data was just 2 sequences, *ab* and *abab*. The initial model produces these sequences exactly. The subsequent iterations show how states are merged to discover the structure of the process. Note that each state in each of the above models deterministically produces a specific output symbol. This is a peculiarity of this data, rather than a feature of the procedure. The figure is taken from [81].

is set to $1/S$ where S is the number of sequences. For all other states π_i is set to 0. In addition, there is a terminal state, which the last state in each chain transits to. We lack a terminal state in our HMM framework, but it can be thought of as a state that outputs a null symbol with probability 1, and only transits to itself. If allowed to run, this initial model is equivalent to randomly selecting one of the training sequences, and generating it. This is represented in the M_0 model in figure 3.3.

The idea is to iteratively merge states to reduce the complexity of the initial model. To merge two states, a new state is created whose parameters are an average of the parameters of the two states being merged, weighted by the expected number of times each state occurs. Instead of asymptotic model selection criteria such as AIC or BIC, a fully Bayesian approach is used, where priors are specified over model structures and parameters, and the posterior probability of the model is maximized. To select which states to merge at each step, they adopt a greedy 'best-first' heuristic, each time selecting the two states whose merging will maximize the posterior. Figure 3.3 shows the initial model, and three iterations of the merging procedure for a toy data set with just two short sequences.

A number of approximations are adopted to keep the computation within rea-

sonable limits, including the Viterbi approximation, treating the Viterbi path as if it were the only path. Most importantly, it is infeasible to consider all possible merges, as this would, for each step, be quadratic in the number of states, and the initial model has the same number of states as observations. To circumvent this an online version is adopted, where an initial model is built from a small number of data points, usually around 20, and extra data points are merged into the new model in small batches, between 1 and 5, until all the data has been exhausted. This allows the procedure to run in time linear to the number of data points. Despite the fact that all these greedy strategies, approximations, and online computation schemes remove any guarantee of the optimality of the final model, the merging procedure still outperforms Baum-Welch training, both in speed and model quality, on the toy and real data presented in [82].

3.7 State Splitting

State splitting approaches typically begin with a single state. At each iteration, states are selected, split, and the parameters are re-estimated. Splitting a state involves adding two new states to the model, and removing the original one. We will refer to the state to be split as the 'split candidate'. Algorithms differ greatly on how the split candidate is chosen at each iteration. We divide state splitting algorithms into two classes: 'exhaustive' state splitting algorithms, and 'heuristic' state splitting algorithms. Exhaustive state splitting algorithms split every state at each iteration, retraining the parameters each time. The split candidate is then selected as the state whose splitting achieved the best improvement in some model selection criterion. The number of splits that require consideration grows with the model, and as each split requires iterative parameter re-estimation, various approximations are used, including constraining re-estimation to a subset of the parameters, and re-estimating the parameters from subsets of the data. Note that 'exhaustive' here does not mean that every architecture is considered. Rather all possible splits are tried at each iteration. Heuristic approaches use other means for selecting split candidates, such as properties of the output distribution. Only one model re-estimation per split iteration is required. We will order the rest of this literature survey by conceptual simplicity, starting with two prominent heuristic state splitting approaches, and moving on to two exhaustive state splitting approaches, which are somewhat more complicated.

3.7.1 Heuristic State Splitting Approaches

Li and Biswas

Li and Biswas [48] discover the structure of HMMs using a combination of state splitting and state merging. Their presentation of the algorithm does not go into very much detail, as they are using HMMs for unsupervised clustering, and they

spend more time describing their clustering approach than HMM structure discovery. Briefly, they model continuous features with a single Gaussian observation distribution per state. Each state thus has an associated mean and variance. With each iteration, they split the state with the largest variance, creating two new states, and retrain the whole model. In the same iteration, from the base model before the split, they also merge the two states with the least distance between the means, and re-estimate the model parameters. At each iteration, they have to decide between two models, the larger model resulting from splitting, and the smaller model resulting from merging. They choose the model that produced the largest improvement in model selection criteria, specifically BIC and the Cheeseman-Stutz approximation. Thus the model can grow and shrink, until no further improvement can be found by either merging or splitting states according to the above rules. This is clearly a heuristic structure discovery approach, as the splitting and merging is guided by the properties of the observation distribution, with no attempt to ensure that the selected split (or merge) is the best one.

Takami and Sayagama

Takami and Sayagama [84] propose the Successive State Splitting (SSS) algorithm to discover the structure of an HMM with partially constrained topology, called a Hidden Markov Network (HM-Net), used for speech recognition. They model the output for each state with a mixture of 2 Gaussian distributions. The initial model is a single state, and grows through state splitting. SSS uses a measure of divergence between the two components of the Gaussian mixture model as the criterion for state splitting. The state with the largest such divergence between its two Gaussians is selected as the split candidate. Each new state after the split takes one of the Gaussian mixture components as its output distribution. Due to the constraints on the structure of an HM-Net, state splitting can be of two kinds, either 'temporal', or 'contextual'. A temporal split involves chaining the two states in series, such that all inputs to the split candidate feed into one of the new states, and all outputs from the split candidate feed out from the other. A contextual split places the two states in parallel, as shown in Figure 3.4. From the base model with the selected split candidate, two models are re-trained, one with a temporal split, and the other with a contextual split. The split that yielded the largest increase in likelihood is retained. The two new states in the model now each have a single Gaussian output. These states are then re-trained with a mixture of two Gaussians, allowing them to be split candidates in future iterations of the splitting process. SSS does not employ model selection criteria, and the state splitting process is terminated when the number of states reaches a predetermined maximum.

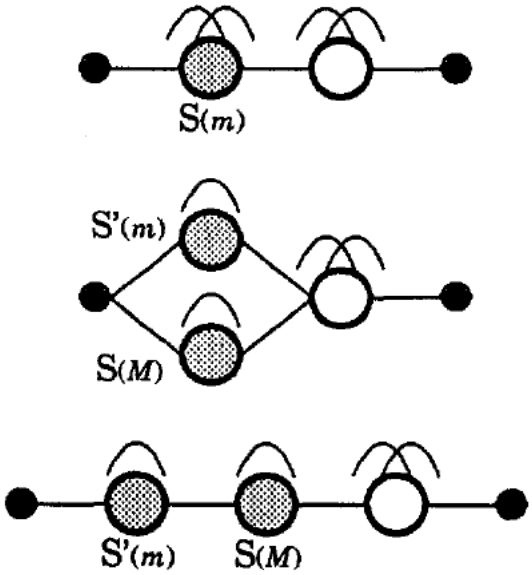

Figure 3.4: *Contextual and Temporal splits.* Top depicts the a two state model, with the split candidate shaded dark. The two curves above each states represent the mixture model with two Gaussian components. Middle shows an example of a contextual split, and bottom shows a temporal split. Note that each state newly created by the split has a single Gaussian output distribution, as each inherits one of its parent's two Gaussians.

3.7.2 Exhaustive State Splitting Approaches

Maximum Likelihood Successive State Splitting

Maximum Likelihood Successive State Splitting (ML-SSS) is proposed by Ostendorf and Singer in [62]. Like SSS, it is also restricted to HM-Net topologies. The key idea behind ML-SSS is to try every possible split at each iteration, and retain the split that yielded the best improvement. Under our categorization scheme, it is thus an exhaustive state splitting technique. This 'try each option' approach requires retraining the model for every possible split, which might seem computationally expensive. In fact, ML-SSS exploits a variety of restrictions of the HM-Net topology and employs a constrained version of the Baum-Welch algorithm to make it more efficient than SSS. Specifically, the parameters of a single state's observation distribution can be optimized independently while still guaranteeing a global likelihood increase. With contextual splits, only the observation distributions of the new states are optimized. For temporal splits, all parameters require optimization, but the number of these is limited by restrictions on HM-Net topology, amortizing their cost. We will not entertain further details of such efficiency gains here as they are not general enough to extend usefully beyond HM-Net topologies.

Simultaneous Temporal and Contextual Splitting

Siddiqi et al. [74] propose Simultaneous Temporal and Contextual Splitting (STACS), which does away with the idea of restricting splits to either temporal or contextual kinds, allowing the algorithm to discover HMM structure without topological constraints. They adopt an exhaustive approach, trying each state as a split candidate and retraining the model each time. Importantly, like ML-SSS, they adopt a constrained re-estimation procedure designed to train only the parameters of the newly created states. Their constrained re-estimation procedure works by only considering observations for sections of the Viterbi path 'belonging' to the split candidate. This dramatically increases the speed of re-estimation, and STACS can even find the structure of a model and its parameters faster than Baum-Welch training finds the parameters alone, with the model structure pre-specified. STACS employs BIC as a model selection criterion, and splitting stops when no split yields any increase in BIC.

3.8 Other Approaches

We conclude this survey of HMM structure discovery techniques with a brief discussion of approaches that are interesting enough to warrant mention, but differ from the present work sufficiently to prohibit detailed discussion.

Shalizi et al. [73] describe a technique called Causal State Splitting and Reconstruction (CSSR) to infer the structure of HMMs from data. The algorithm is based in a computational mechanics framework, exploiting a particular definition of 'causal state'. The properties of such states are derived from this definition, and a state splitting algorithm is created that discovers these causal states.

Brand [14] proposes a structure discovery approach based on an entropic model prior, and a maximum a posteriori HMM training approach. A more-complex-than-necessary initial model is specified. Parameter re-estimation with the proposed entropic prior drives parameters whose presence hardly affects the model likelihood to 0, producing sparser models. Explicit parameter pruning is also incorporated, to speed convergence. The result is sparse, easily interpretable models that perform well when compared to Baum-Welch.

Krogh et al. [38] use HMMs for protein modeling and alignment. The topology of the HMM has the form of a profile model, described above, intended to capture matches, substitutions, insertions and deletions at particular positions in amino acid sequences. A kind of structure discovery, called 'model surgery', is implemented where if too many of the training sequences pass through a particular state that represents a deletion at a particular position, all states corresponding to that position are removed. Similarly, if too many sequences pass through a particular insertion state, extra states are added, lengthening the model.

Kwong et al. [40] use a hybrid genetic algorithm to optimize the structure and parameters of an HMM. The model structure is encoded in a chromosome, and a

population of models is maintained. Model fitness is simply the log likelihood computed with the forward algorithm. The fittest models are retained, interbred, and mutated. The reason the algorithm is a 'hybrid' genetic algorithm is that Baum-Welch training is applied every 10 generations to improve the speed of convergence.

Lee et al. [46] apply HMMs to handwritten character recognition. They decompose the characters into discrete line segments, and design the initial models based on this decomposition, such that each line segment has its own state. The model structure discovery process is thus driven by the data. They show improved recognition results over regular HMMs.

Finally, an approach that attempts to circumvent the problem of structure discovery entirely is the so-called 'Infinite Hidden Markov Model' (IHMM) [8, 89]. The number of states is countably infinite, and so sampling techniques must be employed for inference in IHMMs. Both Gibbs sampling [8] and, more recently, Beam sampling [89] have been explored. The IHMM is a non-parametric alternative to HMMs.

Chapter 4

Structure Discovery - Contributions

In this chapter[1] we propose a state splitting approach to structure discovery, where states are split based on two heuristics: 1) within-state autocorrelation and 2) transition dependence. Both rely on detecting violations of conditional independence assumptions in the fit between the model and the data. Statistical hypothesis testing provides a natural termination criterion, and takes into account the number of observations assigned to each state, splitting states only when the data demands it. With synthetic data, we demonstrate the algorithms ability to recover the structure of hidden Markov models from their observation samples. We also show how it outperforms regular Baum-Welch training both in achieving lower training set AIC and BIC scores, and in a classification task. This superior performance is despite the fact that in both tasks, Baum-Welch training had the advantage of being initialized with the number of states of the HMM that actually generated the data. We then generalize the approach to vector valued observations again showing improvement over Baum-Welch with synthetic data, as well as data from an online handwriting character recognition task. We also use our algorithm as a platform for comparing the performance of the larger models favored by AIC against smaller ones favored by BIC, showing that BIC tends to underfit. We postpone a discussion of the computational complexity of the algorithm until chapter 5.

4.1 Rethinking the Reasons for State Splitting

The heuristics for state splitting discussed in section 3.7.1 are based on the observation distributions of states, paying no attention to the temporal structure of the observations. This is problematic for two reasons. Firstly, it requires assumptions

[1]The primary contributions of this chapter, roughly up to the end of section 4.5.3, were presented at the Annual Symposium of the Pattern Recognition Association of South Africa [59] in November 2008.

about the observation distributions that preclude the use of more complex observation models. For speech recognition, for example, Rabiner [68] recommends using a mixture of as many as 9 Gaussians. It is not clear how to generalize such approaches to mixtures of many Gaussians. This is closely related to the second reason. When it is possible to model state outputs with multi-modal distributions, it is not clear why we should split states based on the shape of the distribution. Some states in the underlying process might have genuinely multi-modal, high variance distributions, and splitting such states would be a mistake. This would be like approximating a multi-modal distribution with many states, imposing arbitrary temporal structure, rather than a single state with a multi-model observation distribution. This effectively wastes many transition parameters. Rather than split states based on the shape of their observation distributions, we propose two heuristics based on the temporal structure of the observation and state sequences.

4.2 Autocorrelation as a Heuristic for State Splitting

We suggest that one good reason for splitting a state is if successive observations produced by that state have temporal structure. If the output at t for state s is not independent of previous outputs in a run of state s, then a better approximation can often be found with more states. This is analogous to checking the residuals of a regression for significant autocorrelation, and increasing the model complexity in an attempt to remove it. We are concerned with continuous valued outputs, and we use autocorrelation as an indicator of serial dependence. Before describing how this can be used to find the structure of HMMs, we first briefly review autocorrelation, giving a definition as well as some visualization to aid intuition.

4.2.1 Autocorrelation

The autocorrelation of a process is the correlation of that process with a time shifted version of itself. The time shift is typically called the lag, and the autocorrelation at lag k is defined as:

$$R(k) = \frac{E[(O_i - \mu)(O_{i+k} - \mu)]}{\sigma^2} \tag{4.1}$$

where σ^2 is the variance of the process and μ its mean. For simplicity, we only consider the autocorrelation at lag 1, although the technique could easily be extended to a range of lags if such a thing proves useful. Restricting to lag 1, the autocorrelation can be rewritten as:

$$R = \frac{E[(O_i - \mu)(O_{i+1} - \mu)]}{\sigma^2} \tag{4.2}$$

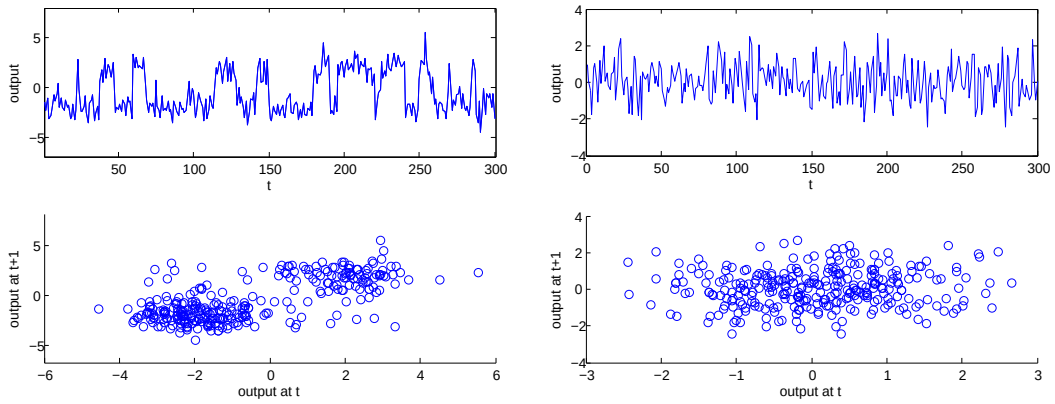

Figure 4.1: *Autocorrelation.* Left and right are time series (top), and lag 1 (bottom) plots for two different observation sequences. Left is an observation sequence from a 2 state HMM with one observation distribution mean at -2 and the other at 2. The positive relationship in the lag 1 plot below is evident. If these observations were being credited to a single state, significant autocorrelation would suggest that state be split. Right is an independent and identically distributed observation sequence with mean 0. Its lag 1 plot shows no correlation.

If μ and σ^2 are unknown, then we will have to estimate them from the data. Analogous to the standard definition of correlation, we will consider two subsequences of the observation sequence, $O^a = (O_1...O_{T-1})$ and $O^b = (O_2...O_T)$, and use estimates of their means and variances. O^a is just the sequence minus the last observation, and O^b the sequence minus the first. Autocorrelation then becomes

$$AC = \frac{E[(O_t - E[O^a])(O_{t+1} - E[O^b])]}{\sqrt{Var[O^a]Var[O^b]}} \tag{4.3}$$

Visualizing Autocorrelation

A lag plot can provide some insight into the behaviour of autocorrelation [20]. A lag n plot displays an observation sequence O as a scatter plot, where each data point has O_t as its x value and O_{t+n} as its y value. The lag n autocorrelation would be the correlation coefficient associated with this bivariate data. Most of the plots from this section use lag 1 autocorrelation.

Figure 4.1 demonstrates lag plots for two processes. Each is composed of the observation sequence on top, and the corresponding lag 1 plot below. The observation sequence from the left plot is generated by a two state HMM, with Gaussian observation distributions with means -2 and 2. There is clearly a positive relationship in its associated lag plot, as the density of the points is mostly concentrated in two clusters centered about $(-2, -2)$ and $(2, 2)$. The lag plot on the right for independently and identically distributed (i.i.d.) data, on the other hand, shows no such trend.

What if the observation sequences aren't produced by an HMM-like process? The collection of lag plots in figure 4.2 on page 56 shows the autocorrelation for a few different processes. As can be seen, any process where successive observations are closer than distant ones exhibits positive autocorrelation. This includes the noisy sin curve in the top left plot, the clean sin curve in the top right plot, and the linear trend in the middle left plot. Generally, the less noise in such a process, the more impressive the correlation.

Negative autocorrelation can also be produced. This occurs when successive observations are further apart than distant ones. One example of this is the plot on the middle right. The process produces observations that alternate between 1 and -1, with added Gaussian noise. The lag plot shows the strong negative correlation. It is possible for observation sequences that have serial dependence to exhibit no lag 1 autocorrelation. An example of this is the process on the bottom left. The lag 1 plot shows no correlation. A lag 2 plot, however, shows strong negative correlation for the same process (bottom right).

4.2.2 Using Autocorrelation to Build HMMs

If the observations belonging to a single state exhibit serial dependence, then a conditional independence property is being violated. In particular,

$$p(O_t|q_t) = p(O_t|q_1...q_{t-1}, O_1...O_{t-1}) \tag{4.4}$$

implies that O_t is conditionally independent of O_{t-1}, given q_t. In addition, O_t is independent of t, and thus the observations produced by any state should not exhibit any serial dependence. In fact, the observations produced by each state should be independently and identically distributed according to that state's output distribution. The presence of autocorrelation shows that this i.i.d. property is being violated. The major contribution of this thesis is the demonstration that we can use this to iteratively grow HMMs, adding states wherever serial dependence is detected. Autocorrelation is a just a simple yet powerful way to detect serial dependence, and other approaches might prove equally successful.

The above discussion assumes that we know which states produced which observations, yet we cannot assign observations to states with certainty. Instead we employ a frequently adopted useful fiction and associate each observation with the corresponding state in the most likely state path, using the Viterbi algorithm[2]. We thus have an observation sequence and its corresponding state sequence. We need to be able to compute the autocorrelation of the observations for a particular state, s. A number of runs[3] of state s occur within our observation sequence O, and the autocorrelation associated with state s must consider only successive observations within a run. In other words, the last observation of the j^{th} run of state s should

[2]This is discussed in more detail in chapter 5.

[3]A 'run' of state s is a maximal non-empty subsequence consisting of adjacent s elements.

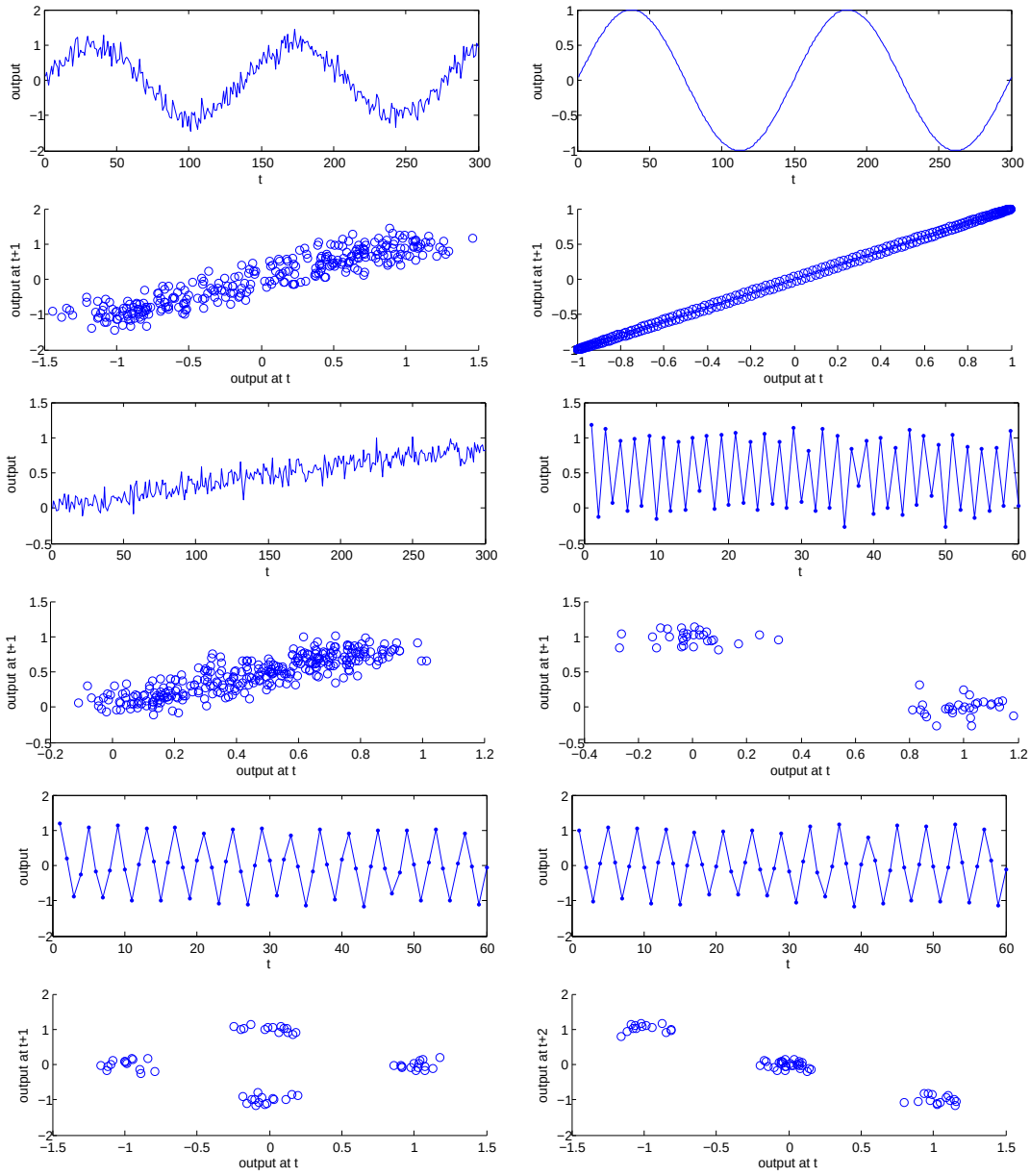

Figure 4.2: *Time series and lag plots to visualize autocorrelation.* Top left: A noisy sin curve. Top right: A smooth sin curve. Middle left: A noisy linear trend. Middle right: Noisy alternating points. Bottom left: A periodic process. Bottom right: A lag 2 plot of the same periodic process.

not be paired with the first observation of the $j+1^{th}$ run of state s, as these are separated by some other state and must be omitted from the computation.

Let A^s be a subsequence of O where the observations of all runs of state s are concatenated after dropping the *last* observation in each run. Similarly, let B^s be a subsequence of O where the observations of all runs of state s are concatenated after dropping the *first* observation in each run. After concatenation, the index of these subsequences no longer refers to time, and as such we index each sequence with i rather than t. A^s_i and B^s_i are, by this definition, both from the same run of s, and if A^s_i is O_t then B^s_i is O_{t+1}. The within-state autocorrelation for state s is then

$$AC^s = \frac{E[(A^s_i - E[A^s])(B^s_i - E[B^s])]}{\sqrt{Var[A^s]Var[B^s]}} \tag{4.5}$$

In a state splitting algorithm, we need to decide which state, if any, to split. We need to convert an autocorrelation coefficient for each state to such a decision. One approach would be to split the state with the greatest autocorrelation. This is problematic, as very short sequences can produce very large autocorrelations by chance. Splitting states that produce very few observations would be extremely counterproductive.

Instead, the frequentist hypothesis testing paradigm provides a natural way to make such a decision. To evaluate whether or not the observations for a particular state exhibit significant autocorrelation, we use a statistical hypothesis test under the null hypothesis that there is no autocorrelation, and reject that null hypothesis if the p-value for the test is below a chosen ϵ[4]. We employ the standard test for significant correlation using the Student's T distribution[5] If there is more than one state that exhibits significant autocorrelation, we split the state with the lowest p-value.

4.3 Transition Dependence as a Heuristic for State Splitting

Before describing the state splitting algorithm in more detail, we introduce a second heuristic. It is based on a different conditional independence violation, and is designed to suggest splits where autocorrelation cannot. The final algorithm will combine both heuristics. Before introducing this heuristic, we will introduce the problem it seeks to address.

[4]Typically called α in the statistics literature, but we use ϵ to avoid confusion with the forward variable

[5]The Ljung-Box test [51] for autocorrelation would probably be more appropriate here, and would certainly be essential when testing a range of lags. We chose the simple test as it yields good results. It also has a rank based non-parametric version which can be used if odd distributions are skewing the significance.

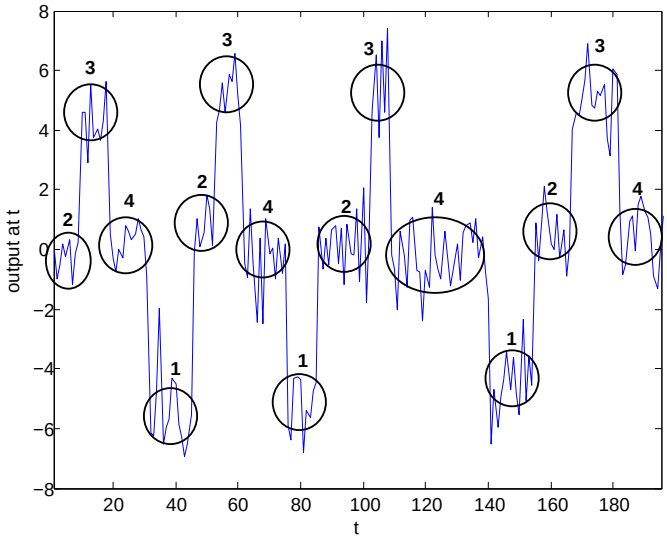

Figure 4.3: *A process with overlapping densities.* A 4 state HMM, with a single Gaussian output per state. Labeled ellipses denote states that produced corresponding observations. Outputs for states 1 and 3 have means -5 and 5 respectively, but outputs for both states 2 and 4 have mean 0. We can tell states 2 and 4 apart, however, because 2 always transits to itself or 3, and 4 always transits to itself or 1.

4.3.1 The Problem of Overlapping Densities

Siddiqi et al. [74] identify a particular situation where structure discovery techniques have trouble correctly recovering the model. This occurs when two states have indistinguishable observation distributions, but different transition parameters. An example of such a process is in figure 4.3. States 2 and 4 have identical distributions, but state 2 always transits to itself or 3, and 4 always transits to itself or 1. Discovering such structure is challenging. Even when Baum-Welch is given the correct number of states, it fails to identify the correct model structure.

No three state model can capture the dynamics exhibited by the process in figure 4.3. A state splitting approach using significant autocorrelation to guide split decisions results in a three state model, with states 1 and 3 correctly identified, but only a single state where states 2 and 4 should be. This is because these observations exhibit no autocorrelation. We thus need a way to tell such states apart. This is the goal of our second heuristic.

4.3.2 Transition Dependence

Consider all runs of state s in a state sequence. We use 'transition dependence' to refer to the case where knowing the state that preceded a run of s improves one's guess of the state that will follow that run. Put differently, where a state transits to depends on where it came from. Formally, let pre_s be the state preceding a

run of s and $post_s$ the state following it. Transition dependence describes the situation where the conditional distribution $p(post_s|pre_s)$ differs from the marginal distribution $p(post_s)$,

$$p(post_s|pre_s) \neq p(post_s) \tag{4.6}$$

If this is the case, then the Markov property of the state sequence is being violated. Recall that

$$p(q_t|q_1, ..., q_{t-1}) = p(q_t|q_{t-1}) \tag{4.7}$$

implies that q_t, given q_{t-1}, must be conditionally independent of any other state. The dependence in 4.6 clearly violates this, indicating that there must be more than a single state underlying s.

Once again we use the statistical hypothesis testing framework to detect such dependence. Let s denote the state under consideration, and N the total number of states. An $N \times N$ matrix T is constructed where each entry $T(r, c)$ represents the number of times a run of state s transitioned to state c, after being preceded by state r. Note that the s^{th} row and column are empty (as we are considering runs of s, and by the definition of a run, s cannot precede or follow a run of s), and are removed from the matrix leaving a $(N-1) \times (N-1)$ matrix. The differences in the relative frequencies between these rows is a measure of how much a state's successor depends on its predecessor. If there is only one state producing the observations we have attributed to s, then we should expect these frequencies to differ only by chance. We use a 2 sample chi-square test and compare each row of frequencies to the frequencies averaged over all the other rows, under the null hypothesis that they come from the same distribution.

This is $N-1$ different tests, and in order to control the false positive rate we use Bonferroni correction [20] for multiple tests and scale our ϵ accordingly. If any of these $N-1$ tests reject the null hypothesis, then state q is a candidate for splitting. This procedure is repeated for each state, and if more than one state shows significant transition dependence, then the one with the lowest p-value is selected as the split candidate.

4.4 The Algorithm

This section describes the computational flow of our heuristic state splitting algorithm, which we name 'Discover'. We adopt a standard heuristic state splitting approach as described in section 3.7.1. We first give an overview of the algorithm, then proceed to describe each step in more detail. We model each state's output with a single Gaussian, although the approach makes no assumptions about the observation distributions, and could easily be extended to work with other observation models. We begin with a single state. If significant autocorrelation is detected,

that state is replaced with two states, and the model is retrained. Within-state autocorrelation is then calculated for each state, and the state with the lowest p-value is split. This is continued until there is no significant autocorrelation in any state. Transition dependence is then calculated, and if it is detected, the state with the lowest p-value from the chi-square test is split, and the model is retrained. This process in continued until neither heuristic detects any significant conditional independence violations. Rather than rely on our own (potentially buggy) HMM algorithm implementations, all algorithms presented here use the basic HMM routines from Kevin Murphy's Bayes Net ToolBox for MATLAB [58].

4.4.1 Initialization

The initial state has only two parameters: the mean and variance of its output distribution. These are set to the maximum likelihood estimates for the data over the entire sequence. As there is only a single state, the Viterbi path is trivially a sequence of 1s.

4.4.2 Splitting Due to Autocorrelation

If significant autocorrelation has been detected within the observations attributed to a single state, we need to split that state. This requires specifying how to initialize a new model with an extra state. We cannot simply add an extra state to the model with random parameters. Baum-Welch is being used for model retraining and its sensitivity to initial parameters requires a good initial guess. Nor can we just add an extra state with exactly the same parameters as the split candidate (after normalizing to satisfy stochastic constraints), as Baum-Welch training is deterministic, and if two states have identical parameters, Baum-Welch would never separate them. To remedy this, we simply separate the observation distributions slightly.

Let s denote the split candidate. Let s' and s^* denote the two new states. The initial state probabilities for the two new states are set to half that of the split candidate.

$$\left.\begin{matrix}\pi_{s'} \\ \pi_{s^*}\end{matrix}\right\} \leftarrow \tfrac{1}{2}\pi_s$$

The transition probabilities between the two new states, as well as their self transitions are set to half the self transition probabilities of the split candidate.

$$\left.\begin{matrix}a_{s's^*} \\ a_{s^*s'} \\ a_{s's'} \\ a_{s^*s^*}\end{matrix}\right\} \leftarrow \tfrac{1}{2}a_{ss}$$

The transition probabilities from the two new states to every other state are set to the corresponding transition probabilities from the split candidate.

$$\left.\begin{array}{c} a_{s'j} \\ a_{s^*j} \end{array}\right\} \leftarrow a_{sj}$$

The transition probabilities from every other state to the two new states are set to half the transition probabilities from every other state to the split candidate.

$$\left.\begin{array}{c} a_{is'} \\ a_{is^*} \end{array}\right\} \leftarrow \frac{1}{2} a_{is}$$

The variances for the new states are set to that of the split candidate.

$$\left.\begin{array}{c} \sigma_{s'} \\ \sigma_{s^*} \end{array}\right\} \leftarrow \sigma_s$$

The only case where s' and s^* have different parameters are the means. They are both shifted a single standard deviation either side of the mean for the split candidate, forcing them to account for different observations.

$$\begin{array}{ll} \mu_{s'} & \leftarrow \mu_s + \sqrt{\sigma_s} \\ \mu_{s^*} & \leftarrow \mu_s - \sqrt{\sigma_s} \end{array}$$

4.4.3 Splitting Due to Transition Dependence

We do not use the same procedure when splitting states due to transition dependence. Perturbing the means makes no sense, as the observation distributions of the underlying states are similar. Instead, we rely on the fact that our test for transition dependence identifies the particular way in which the underlying transition distributions differ. The means and variances for s' and s^* are set to those of s. As above, the initial state probabilities are set to half those of s. As the means, variances, and initial state probabilities are identical, we use the transition matrix to allow Baum-Welch to distinguish the two new states. Recall that the test for transition dependence identifies not just the state to split, but which particular row in the T matrix differed from the others. We initialize $a_{s'i}$ with the normalized transition counts from the row of the T matrix that differed most significantly from the rest, and a_{s^*i} is initialized to the normalized sum of the other rows.

4.4.4 Termination

Using hypothesis testing to guide state splitting provides a natural stopping criterion. We could use the minimum AIC or BIC model if our heuristic state splitting lead to overfitting, but in our empirical tests it does not. It does sometimes lead to

quite large models. If this is unfavorable for computational reasons then one could use the minimum BIC model, which is typically much smaller. The frequentist statistical hypothesis testing framework is also useful when combining many different heuristics, as a single ϵ parameter can be set for all of them.

Algorithm 1 :Discover

```
Initialize: Create a single state model.
repeat
  Run Baum-Welch re-estimation until convergence. Compute p-values for autocorrelation in each state(p-valAC)
  if min(p-valAC) < ε then
    Split state argmin(p-valAC)
  else
    Compute p-values for transition dependence in each state (p-valTD)
    if min(p-valTD) < ε then
      Split state argmin(p-valTD)
    end if
  end if
until (min(p-valAC) ≥ ε)) && (min(p-valTD)) ≥ ε)
```

4.5 Performance

In this section we first show how the algorithm behaves on some simple test cases, and some real data. We then test the model on synthetic data in a classification task. The use of synthetic data here is simply because we did not have access to univariate continuous time series data for classification tasks. In the next section, we generalize the technique to handle multivariate data and demonstrate its performance on real data.

4.5.1 The Overlapping Observation Distribution Process

Discover can successfully recover the structure of the 4 state process in figure 4.3. We use this process as a demonstration of the algorithm's behaviour. First, we must be precise about the parameters of the model that we are trying to recover.

The model only starts in state 1, so the initial parameters are

$$\pi = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The transition matrix is

$$A = \begin{pmatrix} 0.95 & 0.05 & 0 & 0 \\ 0 & 0.95 & 0.05 & 0 \\ 0 & 0 & 0.95 & 0.05 \\ 0.05 & 0 & 0 & 0.95 \end{pmatrix}$$

The vector of means is

$$\mu = \begin{pmatrix} 1 \\ 0 \\ -1 \\ 0 \end{pmatrix}$$

All the variances are set to 0.05, so the vector of variances is

$$\sigma = \begin{pmatrix} 0.05 \\ 0.05 \\ 0.05 \\ 0.05 \end{pmatrix}$$

10 sequences of length 400 were generated by this model. Discover was run on these sequences. This is a good time to introduce two kinds of plots used in this section. The first we will call a 'state sequence plot'. Each state sequence plot is divided by a vertical black bar. To the right of the bar is a plot of the observation sequence, with the colour of the data points determined by the Viterbi path of the model through the data. To the left of the bar is a representation of the model output distributions for each state, coloured and scaled to match the observation sequence to the right. Overall, this plot allows one to visualize part of the model, as well as how it fits to the data. Figure 4.4 shows the progression of the algorithm using 4 different state sequence plots, from the initial single state model, to the final 4 state model. At the bottom of figure 4.4 is the second kind of plot, which we call the 'score plot'. This is divided into two sections. The top plot shows a plot of a candidate observation sequence, and the bottom shows the AIC and BIC values over the course of Discover's state splitting progression.

The model parameters discovered by the algorithm are listed below. It should be pointed out that multiplying each of these parameter matrices and vectors by some row exchange matrix (ie. relabeling the states) would produce a model with parameters very close to the original model. The log-likelihood of the data for the recovered model was -505.192259.

$$\pi = \begin{pmatrix} 0.0000 \\ 0.0000 \\ 1.0000 \\ 0.0000 \end{pmatrix}$$

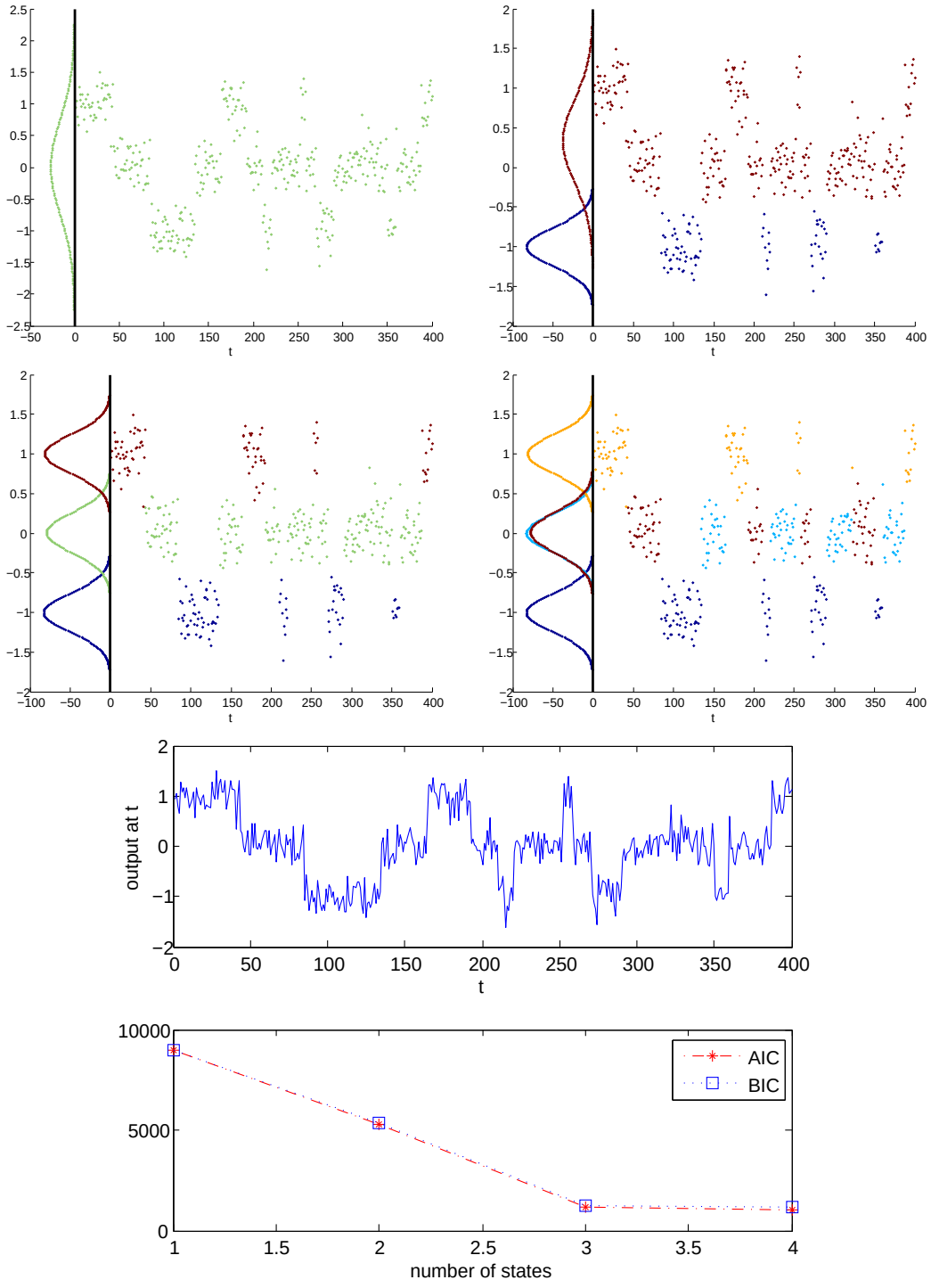

Figure 4.4: *Discovering the 4 state process.* Discover reconstructing the HMM for the 4 state process from figure 4.3. 4 'state sequence plots' (above) show the progression of the model over the course of the discovery process. The 'score plot' (below) shows the progression of AIC and BIC scores. See text for further details.

$$A = \begin{pmatrix} 0.9523 & 0.0477 & 0.0000 & 0.0000 \\ 0.0000 & 0.9457 & 0.0543 & 0.0000 \\ 0.0000 & 0.0000 & 0.9484 & 0.0516 \\ 0.0427 & 0.0000 & 0.0000 & 0.9573 \end{pmatrix}$$

$$\mu = \begin{pmatrix} -1.0059 \\ -0.0027 \\ 0.9970 \\ 0.0069 \end{pmatrix}$$

$$\sigma = \begin{pmatrix} 0.0583 \\ 0.0581 \\ 0.0604 \\ 0.0659 \end{pmatrix}$$

Running Baum-Welch training until convergence on the same data sequence does not recover the original model, even when you tell it the correct number of states in advance. Baum-Welch achieved a much lower log-likelihood of -570.256613 with the following transition matrix:

$$A = \begin{pmatrix} 0.3087 & 0.0839 & 0.0000 & 0.6074 \\ 0.0158 & 0.9531 & 0.0225 & 0.0086 \\ 0.0000 & 0.0505 & 0.9495 & 0.0000 \\ 0.3927 & 0.0244 & 0.0000 & 0.5829 \end{pmatrix}$$

The model can be seen on the left of figure 4.5. The failure is evident. Another run of Baum-Welch training on a sequence generated by the same original model yielded a log likelihood of -589.236377, with the following transition matrix

$$A = \begin{pmatrix} 0.3525 & 0.0272 & 0.5964 & 0.0238 \\ 0.0365 & 0.9388 & 0.0247 & 0.0000 \\ 0.5408 & 0.0262 & 0.4113 & 0.0217 \\ 0.0247 & 0.0000 & 0.0266 & 0.9487 \end{pmatrix}$$

The state sequence plot for this attempt can be seen to the right of figure 4.5. Even though the output distributions are roughly in the correct place, Baum-Welch training fails to learn the correct transition probabilities. Neither transition matrix can be row exchanged to produce anything near the original transition matrix. Repeated runs revealed that Baum-Welch never finds the correct model, whereas Discover always does.

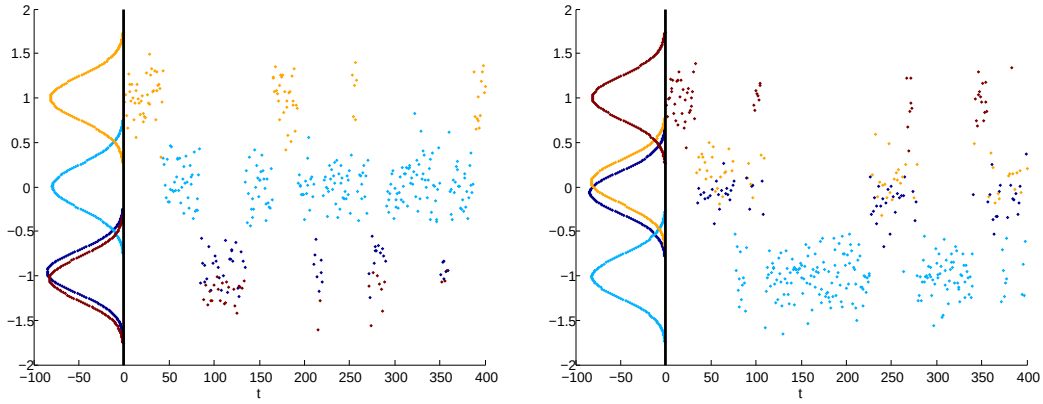

Figure 4.5: *Baum-Welch on the 4 state process.* Baum-Welch fails on two different runs on data from the same 4 state process. See text for details.

4.5.2 Well Log Data

We also demonstrate Discover's performance on real univariate data. The well log data comes from lowering a device to measure the nuclear magnetic response of rock strata down a bore hole [23]. This data set has 4050 measurements. The signal appears piecewise constant, with some severe outliers. As we don't have anything corresponding to ground truth for this data, we will compare log likelihoods for models from Discover with those from Baum-Welch initialized with the same number of states found by Discover. This is intended more as a demonstration of the behaviour of the algorithm than of its superior performance.

We show the results of Discover with two different ϵ values in figure 4.6. With a very sensitive ϵ of 10^-2, Discover learns a 15 state model with a log likelihood of -37559.84 from the well log data. With a less sensitive ϵ, Discover learns an 8 state model with a log likelihood of -37765.57. The models and their fit to the data can be seen in figure 4.6, as well as the AIC and BIC curves. Interestingly, the well log data actually exhibits more temporal structure than meets the eye. Even where the segments appear constant Discover detects significant autocorrelation. Also notice the difference between BIC and AIC as the state splitting progresses. BIC penalizes extra parameters more stringently than AIC, biasing heavily against large models, while AIC does not. When Baum-Welch is applied to the same data, initialized with 15 and 8 states, not only are the log likelihoods lower (-37752.95 and -38173.32), but the models just don't seem to correspond well to the data, as can be seen in figure 4.12. The striations from the 15 state Baum-Welch model show pathologically poor performance, most likely due to sensitivity to initialization.

4.5.3 Synthetic Data

An objective test of Discover's performance is a classification task. A 20 'word' vocabulary was created, with each word corresponding to a randomly generated

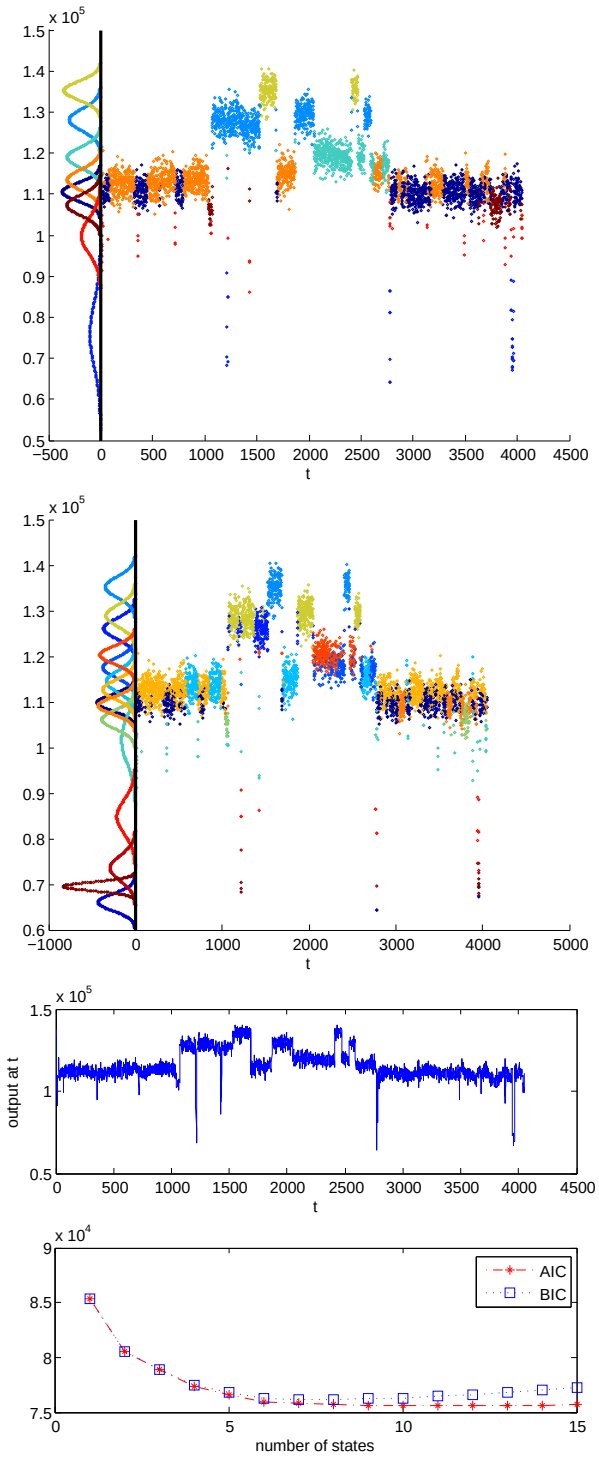

Figure 4.6: *Well log data.* Discover's performance on the Well log data. Top $\epsilon = 10^{-12}$, middle and bottom $\epsilon = 10^{-2}$. See text for further details.

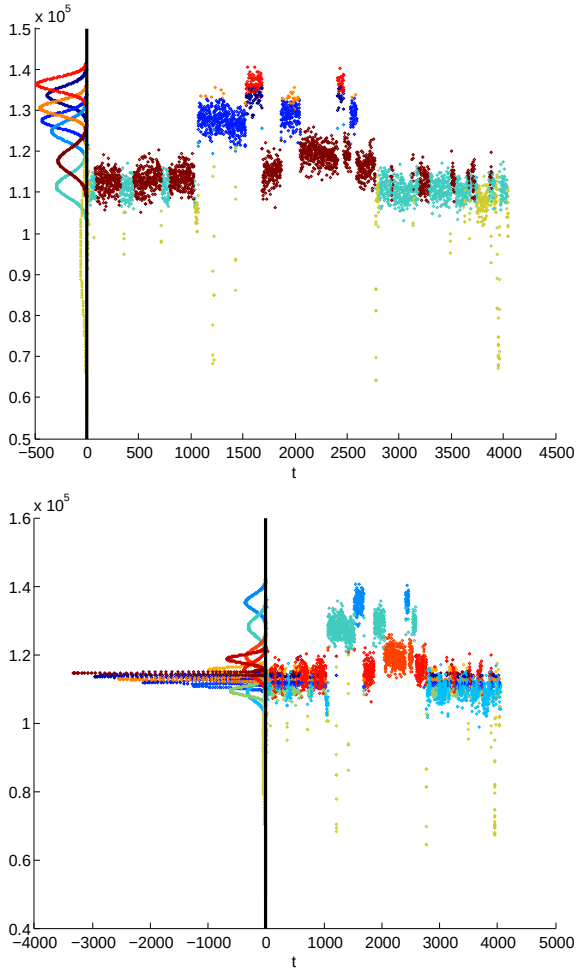

Figure 4.7: *Baum-Welch on the well log data.* Baum-Welch initialized with 8 states (top) and 15 states (bottom). See text for details.

HMM. 30 observation sequences, each with 400 observations, were generated by each word model; 15 for its training set and 15 for its test set. For both Baum-Welch and Discover, one model was trained on the training sequences of each word, with Baum-Welch training once again being initialized with the same number of states as the model that generated the sequences. This yields 3 models per word, one from Discover, one from Baum-Welch training, and the original model that generated the sequences. The classification performance of the original model is included as an upper-bound for the classification accuracy, which will change from one iteration of the experiment to the next, as the similarity between the randomly created models will determine the similarity of the sequences generated and thus the difficulty of the classification task. Some experiment iterations will result in more similarity between words than others, and we use the generating model to obtain an estimate of the difficulty of the classification task.

Classification accuracies using Discover, Baum-Welch training, and the original models are reported in Tables 4.1, 4.2, 4.3 and 4.4 for words with 4, 7, 10 and 20 states, with each classification experiment being performed 10 times. Remarkably, the performance of Discover was extremely close to that of the upper bound set by the original model for each iteration. Baum-Welch training performed much worse[6], even though it was initialized with the correct number of states. Besides classification tasks, we also tested Discover's ability to recover the actual structure of the generating model. Discover very often found the true model structure, but we postpone a discussion of this until the next section.

4.6 Multivariate Observation Data

This section extends Discover to multivariate observations, and demonstrates its performance on synthetic and real data. The transition dependence heuristic relies on the Viterbi path alone and needs no modification. The extension to multivariate data thus requires detecting significant autocorrelation for multivariate time series. We adopt a very simple strategy. Recall that selecting a split candidate using the significant autocorrelation heuristic requires calculating a p-value for each state. With multivariate observations, to compute a p-value for each state, we simply compute individual p-values separately for every observation dimension, and the smallest of these is taken to be the p-value for that state. We then scale these p-values by multiplying them by the total number of observation dimensions, effectively implementing Bonferonni correction for multiple tests to control the false discovery rate.

The only other difference made to the state splitting procedure is how the new model is initialized after state splitting. Analogous to the univariate case, we separate the means (which are now vectors) for each new state by some function of the covariance matrix. We rely on the diagonal of the covariance matrix:

[6]Statistical analysis of such results is somewhat gratuitous, but for all 4 experiments, differences between Discover and Baum-Welch were significant on paired samples t-tests ($p < 0.00005$).

Table 4.1: *Classification accuracies on 4 state HMMs.*

iteration	1	2	3	4	5	6	7	8	9	10	mean
Generating Model	0.86	0.93	0.93	0.81	0.82	0.82	0.84	0.87	0.93	0.94	0.87
Structure Discovery	0.86	0.92	0.93	0.81	0.82	0.82	0.82	0.87	0.92	0.94	0.87
Baum-Welch Training	0.70	0.68	0.71	0.68	0.53	0.69	0.75	0.69	0.80	0.66	0.69

Table 4.2: *Classification accuracies on 7 state HMMs.*

iteration	1	2	3	4	5	6	7	8	9	10	mean
Generating Model	0.78	0.92	0.89	0.87	0.95	0.90	0.81	0.91	0.86	0.80	0.87
Structure Discovery	0.78	0.90	0.90	0.87	0.96	0.89	0.82	0.90	0.85	0.79	0.87
Baum-Welch Training	0.71	0.61	0.61	0.66	0.82	0.65	0.65	0.64	0.63	0.70	0.67

Table 4.3: *Classification accuracies on 10 state HMMs.*

iteration	1	2	3	4	5	6	7	8	9	10	mean
Generating Model	0.92	0.88	0.87	0.75	0.94	0.73	0.93	0.95	0.80	0.85	0.86
Structure Discovery	0.91	0.84	0.85	0.75	0.93	0.73	0.92	0.95	0.79	0.84	0.85
Baum-Welch Training	0.75	0.64	0.69	0.66	0.68	0.62	0.75	0.80	0.59	0.65	0.68

Table 4.4: *Classification accuracies on 20 state HMMs.*

iteration	1	2	3	4	5	6	7	8	9	10	mean
Generating Model	0.85	0.84	0.90	0.87	0.85	0.95	0.84	0.76	0.84	0.89	0.86
Structure Discovery	0.77	0.80	0.87	0.82	0.80	0.85	0.79	0.71	0.82	0.88	0.81
Baum-Welch Training	0.58	0.52	0.61	0.59	0.53	0.77	0.59	0.59	0.51	0.71	0.60

$$\mu^d_{s'} \leftarrow \mu^d_s + \sqrt{\sigma^{dd}_s}$$
$$\mu^d_{s*} \leftarrow \mu^d_s - \sqrt{\sigma^{dd}_s}$$

where μ^d refers to the d^{th} element of the mean vector $\boldsymbol{\mu}$, and σ^{dd} refers to the element at the d^{th} row and d^{th} column of the covariance matrix $\boldsymbol{\sigma}$, which is simply the d^{th} element in the diagonal of $\boldsymbol{\sigma}$. This is the variance along the d^{th} observation dimension for that state's output distribution. The covariance matrices for the two new states just take the values of the split candidate's covariance matrix. The multivariate state splitting procedure is otherwise identical to the univariate one. The MATLAB code for Discover is in A.8 and below.

4.6.1 Synthetic Data

The multivariate extension of Discover can adequately recover model structure when multivariate synthetic data is used. Qualitatively[7], model recovery ability increased with the number of dimensions. We hypothesize that this is because most model recovery errors on univariate data occur when different states have similar observation distributions. In the multivariate case, the probability of this occurring gets exponentially lower as the number of dimensions increases. A high dimensional space space requires a conspiracy of chance for two uniformly sampled random points to be proximal [8] and since this is how our state's output distribution means are generated, there should be less overlap in higher dimensional spaces. This property depends on the distribution of the randomly generated models, and might not extend to real data.

Figure 4.8 shows the score plot for a set of multivariate sequences, synthetically generated from a randomly constructed HMM with 7 states, and a 3 dimensional observation vector. To evidence that the model has been accurately recovered, figure 4.8 includes a 3D scatter plot of the means of each state, for both the model that generated the data and the model recovered by Discover. As can be seen, there is very little difference between the means from the generating and discovered models. While such performance is generally representative, Discover does occasionally fail to identify the number of states in the generating model. This could be a result of the generating model having statistically indistinguishable observation distributions, resulting in fewer states than the original model, or either of the two heuristics returning false positives, leading to extra states. Besides such errors, one genuine problem we identified involved occasionally poor initialization of the two new states, but we postpone a discussion of this until chapter 5.

[7] We do not attempt to quantify this, as we don't have measures of model distance that are comparable across dimensions.

[8] Consider the average distance between two uniformly sampled points on a line, two points in a square, a cube etc...

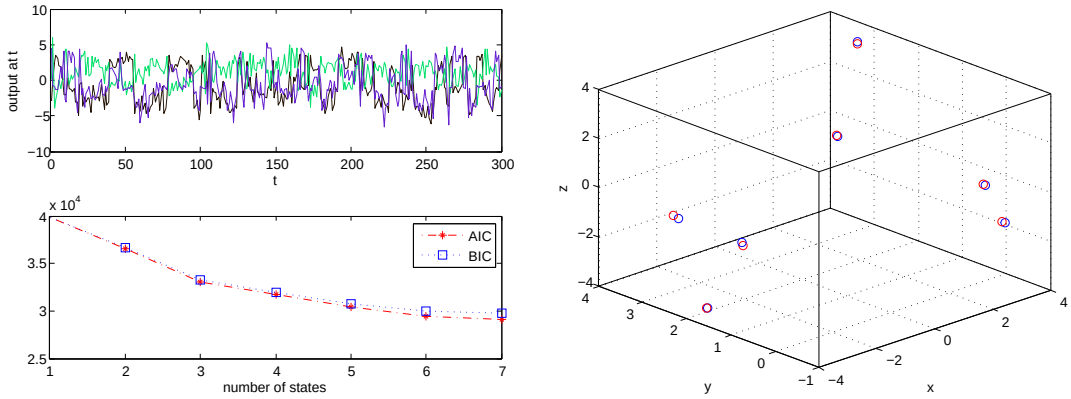

Figure 4.8: *Discovering the structure of a 7 state HMM with a 3D observation vector.* Left shows the score plot for Discover on a 7 state HMM with a 3D observation vector. The sample output is shown in the top half of the score plot. Right shows a 3D scatter plot of the means for each state. Blue depicts means from the HMM that generated the data, red from the HMM found by Discover.

4.6.2 Character Trajectory Data

We conclude this chapter by discussing the performance of Discover in a classification task on multivariate data. The data comes from the 'character trajectories' data set in the UCI machine learning archive [32]. The data comprise 2858 online handwritten characters recorded on a WACOM tablet. The data set is supplied after some preprocessing, which includes numerical differentiation, Gaussian smoothing, and normalization. 20 character types are included in the data set, and each character instance is a time series of x, y, and pen force measurements. In most of the following analysis we discarded pen force, as it only marginally improved performance. As a result of the differentiation, we do not not model the pen position through time, but rather the pen velocity. Figure 4.9 shows the pen velocity trajectories, as well as their numerical integrals which correspond to the actual characters.

There were different numbers of characters in each class, and we randomly selected 30 from each to comprise our training set. The rest formed the test set. The training set was thus just over 20% of the data. As supplied, the data was pathologically noiseless, probably due to the Gaussian smoothing. This presents problems for HMM construction. A state's output distribution can shrink to a delta function as it tries to fit many data points with exactly the same value, which makes the Baum-Welch re-estimation numerically unstable. We experimented with a few solutions to this problem, including adding a prior to the covariance matrix to discourage such shrinkage. The final best results for both Baum-Welch and Discover were obtained by simply adding small amounts of Gaussian noise to the training data.

We used Baum-Welch training as a comparison. Unlike with the synthetic data above, we do not know the best number of states to use in advance. We thus tested

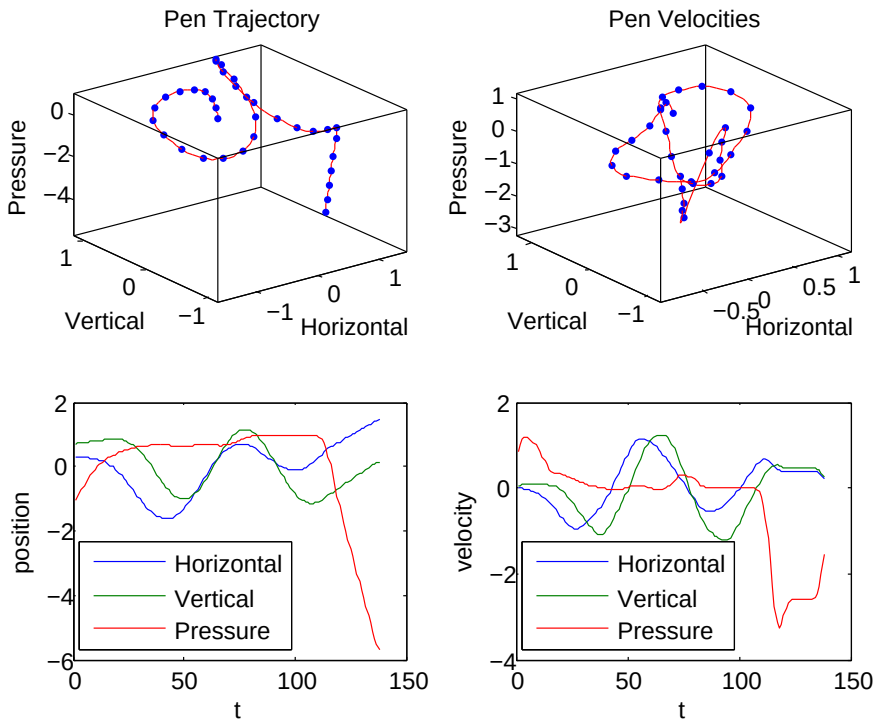

Figure 4.9: *The character 'a', as supplied by the UCI.* A plot of the pen tip position (left) and velocity (right), showing the paths through space (top) and time (bottom). Note that the velocities, without the pressure dimension, are what is actually modeled.

Baum-Welch with a number of different states. The performance varied for each iteration due to sensitivity to model initialization. Average results are plotted in figure 4.11. Baum-Welch always hovered below the 0.8 accuracy mark.

Discover fared far better than Baum-Welch. Accuracies ranged from about 0.88 to 0.91, depending on the value of ϵ, the only parameter. The number of states discovered decreases with the value of ϵ, as does the accuracy. This allows ϵ to weigh computational efficiency against accuracy.

4.6.3 BIC vs AIC for Classification Tasks

AIC and BIC select models of different sizes. BIC selects smaller models, and is more commonly used in the HMM structure discovery literature [47, 74]. As our algorithm terminates naturally without the need for model selection criteria, we can ask how it would behave if we did use model selection criteria. Recall that, at each stage in the state splitting process, a fully trained HMM is produced. This allows us to set our ϵ parameter to be sensitive enough to produce models more complex than either AIC or BIC would recommend, store the models that maximized AIC and BIC during the discovery process and use those to attempt classification. This enables comparison between AIC and BIC as model selection

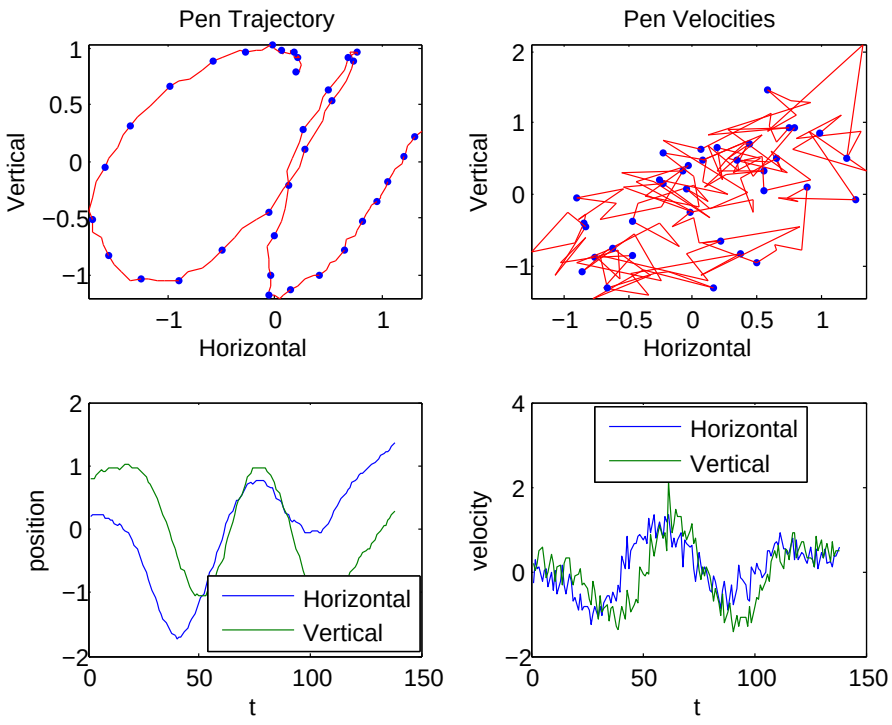

Figure 4.10: *The character 'a', after discarding pressure and adding noise.* This is an example of the data we use to build our HMMs.

criteria for HMMs used for classification tasks. On the characters dataset, models selected by BIC tended to be small, with a mean of 11 states (standard deviation: 2.1764), while models selected by AIC averaged 17.85 states (standard deviation: 3.031). AIC outperformed BIC in classification accuracy, achieving 0.8897 against BIC's 0.8667.

This result should be considered in context. Burnham et al. [15] assert that the performance of AIC and BIC depend on the kind of data, as well the set of models from which selection occurs. Specifically, BIC performs better when there are a few major effects, and AIC better when there are many tapering effects. Now, as HMMs are only intended as an approximation - modeling smooth curves with small piecewise constant segments - we believe the situation is analogous to the case of tapering effects, where more data allows more model structure to be found. Burnham et al. [15] claim that BIC tends to underfit in these cases, and this is consistent with our results above. What is surprising is that the models discovered with quite sensitive ϵ parameters are larger than - and outperform - even the best AIC models. This could be taken as (a small piece of) evidence that AIC also underfits, but we favor an alternative hypothesis. Speculatively, a state splitting search through the model space might introduce a kind of implicit parameter regularization. At each split iteration, the entire model is retrained. This means that when two new states are introduced through splitting, all the other states are already occupying very good locations in their parameter space,

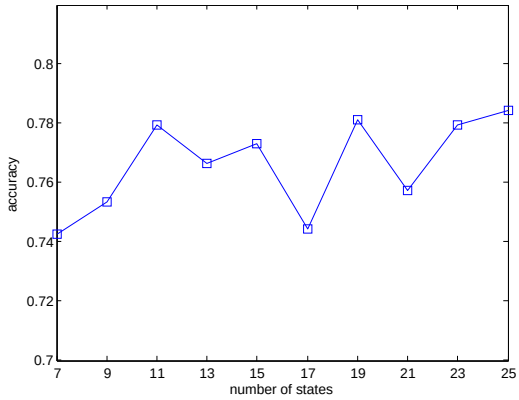

Figure 4.11: *Baum-Welch on the character data.* The performance of Baum-Welch varied between 74% and 78%, and is plotted against the number of states for each experiment.

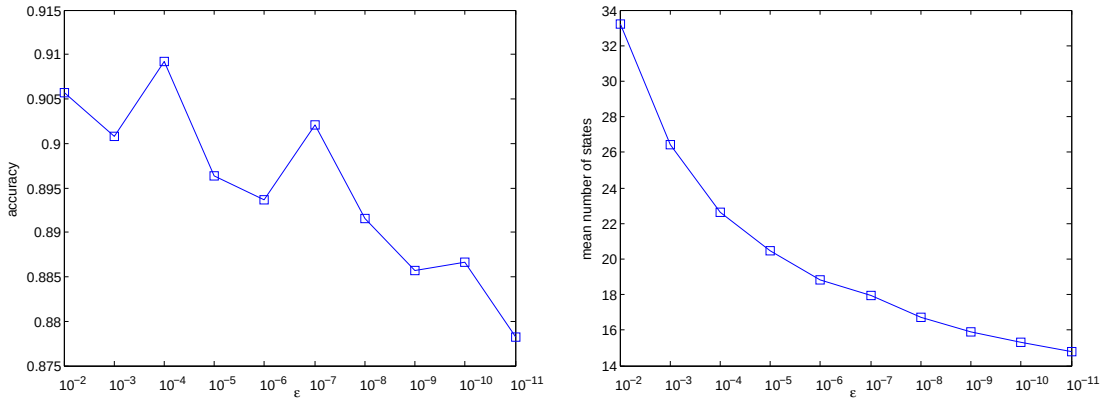

Figure 4.12: *Discover on the character data.* Plots above show the effect of ϵ on accuracy (left) and the mean number of states discovered (right).

effectively restricting the parameter space of the newly introduced states. While this hypothesis is entirely speculative, it would explain why extremely large models with more parameters than there are data points in the training set still produce good classification accuracies. This is atypical; overly large models usually learn the idiosyncracies in the training data and generalize very poorly.

Siddiqi et al. [74] also encountered the fact that extremely large models (around 55 states), discovered by a state splitting approach using BIC as a stopping criterion, produced the best classification results. They ventured that the large state numbers were a result of a coarse 'hard-updates' algorithm being less prone to local maxima. This might explain their large state numbers, but it doesn't explain why such large models showed superior classification performance. Our results shed some light on this, indicating that state splitting algorithms involved in classification tasks using real world data might benefit from using a model selection criterion that penalizes additional parameters less heavily than BIC. Additionally, if the implicit regularization hypothesis holds, even AIC might be too restrictive. Discover, with

its natural stopping point, can avoid such issues, continuing to split states wherever temporal structure is detected.

Chapter 5

Discussion and Future Research

This chapter reflects on the Discover's behaviour and some of its current limitations, and then describes potential directions for future research. We begin by discussing the computational details of the algorithm. Following that, we describe how to generalize the algorithm to handle discrete observations, trading the autocorrelation heuristic for a discrete measure of serial dependence. We also discuss relaxing the reliance on the Viterbi path when computing within-state autocorrelation, showing how to compute a weighted autocorrelation that takes uncertainty in the estimated state sequence into account. A hybridization of Discover with STACS, from [74], is proposed in order to decrease the computation time. Finally, we describe how to incorporate more complex observation distributions; Gaussian mixture models and kernel density estimation.

5.1 Discussion

5.1.1 Why State Splitting Works

One can view a state splitting algorithm as a smart multi-stage initialization procedure. With every split, we only attempt to re-estimate a small piece of the model. While the rest of the model is not explicitly constrained, as it is in STACS, it usually does not change much. This is one potential explanation for the fact that state splitting performs better than Baum-Welch training, even when the model size is known in advance. Perhaps adjusting fewer parameters at a time causes the likelihood function to have fewer local maxima. Another explanation is that state redundancy is usually avoided. When the number of states is specified in advance, regular initialization procedures are forced to commit output distributions to locations in the output parameter space right at the beginning of parameter optimization, from which they may not be able to escape. Adding structure incrementally avoids this kind of problem. Further experimentation with different types of incremental model training might be able to elucidate the mechanisms behind the performance of state splitting procedures.

5.1.2 Running Time

We have made almost no attempt to optimize Discover. Currently, the algorithm requires running Baum-Welch training after every split. Thus each split takes $O(TN^2)$, and N goes from 1 to the number of states finally discovered, which we will call F. This is $TF(F+1)(2F+1)/6$, which is $O(TF^3)$. This is slower than Baum-Welch, and one direction for future research is working on speed increases.

Many HMM applications involve off-line learning, in which case the model training time is not a great concern. The final model size does affect online processing time though, and Discover typically results in larger models than Baum-Welch training. If there is such a computational limit on model size, this can easily be incorporated through early stopping. Discover trained with an upper limit on model size returns better models, on average, than Baum-Welch training initialized with models that size.

5.1.3 Initialization Failures

One of the ways Discover can go wrong is when an extra underlying state is detected, but the two new states created as a result of the initialization procedure fail to appropriately capture the extra structure. This can lead to redundant model structure; states which account for almost no data, or many states accounting for data where just one would do. This could be remedied by a state merging step somewhere in the algorithm, or by coming up with better post-split initialization procedures. Another option would be to include some threshold for model improvement, and if the split, after training, does not improve the model fit beyond this threshold, backtrack to the previous model and try the next best split candidate.

5.2 Extending Discover

Discover, in its present form, admits numerous extensions. We describe them below, in no particular order.

5.2.1 Discrete Observations

The main idea behind Discover, using serial dependence within runs of a single state as a heuristic for state splitting, can be extended to discrete observation sequences. Autocorrelation would need to be abandoned, and some measure of serial dependence for discrete observations employed. Perhaps most closely analogous to first order autocorrelation, one could construct histograms for each observation conditional on the preceding observation, and use a chi-square test to check if these conditional distributions differ from the marginal distributions. This is similar to our transition dependence heuristic, but applied to successive observations rather than runs of states.

5.2.2 Smooth AC Computation

Currently, the computation of within-state autocorrelation relies on treating the Viterbi path as if it were the true path. This is a convenient fiction which clearly serves quite well, but it neglects the probabilistic nature of the model. To fully capture uncertainty about transitions between states, we would need to compute something akin to a within-state autocorrelation over all possible paths, weighted by the probability of each path. We outline a procedure for doing such a thing below. We first define weighted autocorrelation in general, and then describe where the weights come from.

Weighted Correlation and Autocorrelation

Weighted correlation requires a weighted mean and a weighted covariance. If x is a vector of data, and w is a vector of weights of the same length as x, then the weighted mean, $m(x; w)$ is simply

$$m(x; w) = \frac{\sum_i w_i x_i}{\sum_i w_i} \tag{5.1}$$

With an additional vector y of the same length, the weighted covariance between x and y is

$$cov(x, y; w) = \frac{\sum_i w_i (x_i - m(x; w))(y_i - m(y; w))}{\sum_i w_i} \tag{5.2}$$

Weighted correlation is

$$corr(x, y; w) = \frac{cov(x, y; w)}{\sqrt{cov(x, x; w) cov(y, y; w)}} \tag{5.3}$$

Weighted autocorrelation is then just the weighted correlation of a signal with a time shifted version of itself. We now describe a suitable choice of weights.

Weighting by ξ

Recall $\xi_t(i, j)$, the probability of being in state i at t and j at $t + 1$, given an observation sequence and a model:

$$\xi_t(i, j) = p(q_t = S_i, q_{t+1} = S_j | O, \lambda) \tag{5.4}$$

We define $\xi^i = \{\xi_1(i, i), ..., \xi_{T-1}(i, i)\}$. This is simply a vector of self transition probabilities for a particular state. This will be our weight vector for state i. Our

definition of weighted autocorrelation for state i is simply the weighted correlation of the vectors O^a and O^b (defined in section 4.2.1), with ξ^i as the vector of weights:

$$AC^i = \frac{cov(O^a, O^b; \xi^i)}{\sqrt{cov(O^a, O^a; \xi^i) cov(O^b, O^b; \xi^i)}} \tag{5.5}$$

Computing the autocorrelation for state i thus weighs each pair of successive observations (O_t, O_{t+1}) by the probability that the model was in state i for both. Where there is a very low probability that the model was in both states at successive times, the pair of observations will contribute very little to the autocorrelation, but when that probability is very high, the pair will contribute a lot. It is also worth pointing out that this definition reduces to the previous definition when the Viterbi path is the only path, as the old definition is equivalent to ξ_t^i being 1 when the Viterbi path at t is in state i, and 0 otherwise.

This procedure is asymptotically as efficient as using the most likely path to approximate within-state autocorrelation for every state, as the entire $\xi_t(i, j)$ lattice can be computed in $O(N^2T)$, which is asymptotically as efficient as the Viterbi algorithm. We have not yet tested this weighted within-state autocorrelation, and issues related to statistical tests for weighted autocorrelation still need to be resolved.

5.2.3 Incorporation of Other Heuristics

The two heuristics employed in this thesis are not exhaustive, and there are many cases where they will fail to detect underlying structure. For example, if two states have the same transition probabilities, and their distributions have the same mean but different variances, our present algorithm would fail to tell them apart. To deal with such a case one could introduce a heuristic, similar to within-state autocorrelation, that keeps track of the deviation of observations from the mean and tests for temporal structure in this sequence. Such temporal structure would indicate different variances hiding within the same state. The details of the particular statistical test would need some thought though.

There are other cases where Discover would fail to detect underlying model structure. The statistical hypothesis testing framework allows incorporation of many different heuristics, without adding additional parameters. An algorithm with a large number of heuristics begins to look inelegant, however, and, in the next section we describe a way to combine Discover with a different algorithm to produce an efficient hybrid structure discovery procedure.

5.2.4 Combining Discover with STACS

The exhaustive state splitting algorithm, STACS, described in [74] requires testing a split for each state in the present model and retraining a new model once for

each state at every split iteration. To improve the efficiency of such a procedure, the retraining for each split test is constrained so that only the observations which belong to that state on the Viterbi path are considered. This drastically improves the training time. We propose a marriage of our heuristic procedure with STACS. This has two benefits: computational efficiency and thoroughness. We first describe the combination, and then describe the potential performance gains.

The Hybrid Algorithm

We propose to select split candidates initially using heuristics, until all heuristic splits have been exhausted. To retrain the split candidate, we would use the STACS constrained re-estimation procedure. Once heuristic splits can no longer be identified, we resort to the usual STACS training procedure, trying every possible split to see which results in the greatest likelihood increase, with stopping dictated by a model selection criterion.

To see why this might be a good idea, consider the course of the STACS algorithm discovering a 10 state model. Assuming most of the work is done by the split selection procedure, retraining new models for every candidate tested, the STACS re-estimation procedure would have to be called $1 + 2 + ... + 10$ times. Fortunately, the STACS re-estimation procedure gets faster per state as the number of states increases, as each state is responsible for fewer data points. This is precisely what keeps it efficient. Specifically, with N states, each STACS re-estimation for a single split is proportional to T/N, causing the above sum to be $1T/1+2T/2+...+10T/10$, or just $10T$. With heuristic state splitting, we no longer need to retrain every state, so, using STACS constrained re-estimation the above sum becomes $T/1+T/2+...+T/10$. Adding an extra state is faster when there are more states! The resulting efficiency gains should be very large, especially when very complex models are discovered.

Running the regular STACS 'try every state' procedure after all heuristic splits have been exhausted would serve to explore possible splits that might not otherwise be identified by our heuristics. This would circumvent the need to add any new heuristics to deal with cases where heuristic splitting might fail, instead relying only on a few strong heuristics, and falling back on the general structure discovery power of STACS when these heuristics fail.

5.2.5 Beyond Single Gaussian outputs

One potential issue for Discover, as well as other structure discovery techniques, is the use of a single Gaussian to model state output distributions. It might be the case that there is an underlying model with relatively little dynamical structure, say 2 hidden states, but with each state having quite complex observation distributions. One could of course use a model with many states to approximate the complexity of the output distributions, while the simplicity of the dynamic structure would

be reflected in the similarity of the state transitions. This is far from optimal, as it wastes many transition parameters, and destroys the interpretability of the resulting model. A better idea would be to use less restrictive output models, such as mixtures of Gaussians or kernel density estimation. Creating a structure discovery algorithm using such complex observation distributions would require ironing out some nontrivial kinks. Two proposals for such algorithms, possible problems, and potential solutions are described below.

Mixture of Gaussian Outputs

The use of Gaussian mixture models[1] as observation distributions is common in the speech recognition literature [68]. Unlike other heuristic structure discovery algorithms, the principles behind state splitting in Discover are agnostic about the observation models, and so allow for complex output distributions. Initial experiments with mixture model outputs have highlighted post split initialization as a problem for such an extension. In our experiments, we tried keeping our post-split initialization procedure as is, and applying it to each Gaussian component in the mixture. Interestingly, this causes the re-estimation procedure to become unstable, as all the data points for certain mixture components migrate to one of the two states, leaving that component in the other state with very little data points. Smarter post-split initialization procedures are required if such techniques are to be effective.

One possibility would be to dynamically add mixture components where the data requires them. A goodness-of-fit test, or some other measure of distance from the empirical distribution, could be used to grow output complexity for each state, and mixtures could be pruned when their mixture weights become too small. A naive post-split initialization procedure in this framework could simple ignore the structure of the split candidate and regrow the observation distributions for each new state. This might be too slow to be useful. Other options would be to take all the mixture components used in the split candidate and, before introducing transitional structure, probabilistically assign each data point to a component. A transition matrix between all components could then be constructed, and components with similar transition probabilities could then be iteratively merged. This leaves open the option of splitting a single state into more than two states, if the transitions between the components indicate such structure. One potential benefit of such an initialization procedure is that it might also reduce the number of Baum-Welch iterations required for convergence, as the transitional structure should be near optimal. The constrained parameter optimization used by STACS should also work with mixture models, speeding up the re-estimation after splitting.

[1] Brevity prevents us from reviewing Gaussian mixture models here, but see [68] for a full introduction to their use in HMMs. The basic idea is to model each state's output distribution as a weighted sum of individual Gaussians, with weights summing to 1 to preserve stochastic constraints.

Kernel Density Estimation

Another option for more complex observation models would be to use an entirely non-parametric kernel density estimation (KDE) output model. This would eliminate the need for adaptive mixture components, but it introduces a host of computational issues. The Baum-Welch algorithm would need to be adapted for KDE based HMMs. This has been attempted in [65], but is not widely used. It is unclear how stable expectation maximization is for KDE based HMMs, and the following set of ideas assumes good behaviour.

HMM structure discovery aims to approximate the structure of the underlying process. Having a KDE based output distribution allows the entire focus of the algorithm to be on finding the transitional structure between states, leaving KDE to handle the output distributions. More concretely, this reduces to probabilistically assigning data points to states, as the output for each state is implicitly created with such an assignment[2]. The hope is that state splitting schemes need uncover relatively fewer states when each state allows a complex observation distribution.

One of the most significant computational issues when using KDEs is that evaluating $p(O_t|q_t)$, the probability of a single observation given a state, is linear in the number of observations, T. This means that the standard HMM algorithms (forward, Viterbi etc.) become quadratic in T, which is prohibitive for long sequences. A number of options for coping with such computational explosions exist, such as the fast Gauss transform [93] and KD-tree based methods [41]. The ease with which these admit expectation maximization is an open question.

One option for handling such issues is to once again rely on Viterbi optimism, the useful fiction that treats the Viterbi path as the only path. The cost of evaluating $p(O_t|q_t)$ would decrease as the number of states increased. Re-estimation would be like the Viterbi training procedure described in chapter 2, simply hard assigning observations to states, and each observation distribution would just be a collection of observations. The degree to which this would harm model performance is an empirical question. We believe that incorporating heuristic structure discovery into the fast Viterbi based state splitting re-estimation procedure, like V-STACS, is a promising direction for research.

5.2.6 Encouraging Sparse Models Through Entropic Priors

Finally, sparse models could be encouraged through the use of an entropic prior, as in [14]. Combining state splitting with such *maximum a posteriori* parameter estimation could add structure while still encouraging sparse models. The pruning schemes discussed in [14] would take care of any states that have become redundant over the course of state splitting. Both state splitting and entropic priors have proven useful for structure discovery, and there is no impediment to combining them. The usefulness of such an endeavor remains to be seen.

[2]The global kernel variance also needs to be estimated, but see [65] for a discussion.

5.3 Conclusion

This thesis introduced the theory of HMMs, surveyed a range of HMM applications and proposed a novel state splitting approach to discovering HMM structure. The algorithm was first tested on univariate synthetic data, extended to multivariate observation vectors, and its performance was demonstrated with an online handwriting recognition task, showing superiority over Baum-Welch training alone. Some potential avenues for future research were suggested, including adapting the algorithm to work for discrete observation data, extending the complexity of each state's observation distribution, and combining the algorithm with STACS to improve its efficiency.

Appendix A

Code

A.1 The Sequence Generator

```
%The Sequence Generator
%hmm.N is the number of states
%hmm.M is the number of observations
%hmm.pi is a row vector denoting the initial state distribution
%hmm.A is a matrix denoting the state transition probabilities, where a_ij
%= P(q_t = S_i,q_{t+1}=S_j)
%hmm.B is a row vector denoting the state output distribution, where b_jk
%denotes b_j(k) = P(O_t = v_k|q_t = S_j)

%returns a randomly generated sequence of observation symbols and the state
%sequence that generated them. Input an HMM and the length of the sequence
%required. Outputs: O - The vector of observations, Q - The state sequence
function [O,Q] = generator(hmm,seqLength)

%initialise outputs
O = zeros(1,seqLength);
Q = zeros(1,seqLength);

%sample from pi to get the starting state, Q_1
Q(1) =  sample(hmm.pi);
%sample from the output distribution of the starting state to get the first
%observation, O_1
O(1) = sample(hmm.B(Q(1),:));

%iterate
for t = 2:seqLength
    %sample from the state transition distribution of state Q_{t-1},
    %A_{Q_{t-1},:} to get the state t.
```

```
    Q(t) = sample(hmm.A(Q(t-1),:));
    %sample from Q_t to get O_t
    O(t) = sample(hmm.B(Q(t),:));
end
```

A.2 The Sanity Checker

```
%The Sanity Checker
%This is useful for checking p(Q) and p(O) given some model. To check p(Q)
%for a Markov model, just set up a full HMM, but ignore the observation
%output.

%setting up the HMM from the Johnny example. See the text for details.
%For the states, 1 encodes 'busy' and 2 encodes 'free'. For the
%observations, 1 encodes 'peanut butter' and 2 encodes 'toasted cheese'
hmm.N = 2;
hmm.M = 2;
%       busy free
hmm.pi= [0.2 0.8];

%       busy free
hmm.A = [0.6 0.4; %busy
         0.1 0.9];%free

%         PB   TC
hmm.B = [ 0.9 0.1; %busy
          0.3 0.7];%free

%number of simulations
trials = 1000000;
%the sequence whose probability you are trying to estimate
testSeq = [2 2 1 1 2];
%count will store the number of times the sequence occurs
count = 0;

%run a number of simulations
for i=1:trials
    %generate observation and state sequences
    [O,Q] = generator(hmm,5);
    %Compare Q to testSeq for p(Q), and O for p(O)
    if Q == testSeq
        %if the sequence in question appears, inc count
        count = count + 1;
```

```
    end
end
%display the ratio, which should accord with the analytical calculation
count/trials

%Ben Murrell
```

A.3 The Forward Algorithm

```
%The Forward Algorithm
%returns p(O|model), and the lattice of alphas produced by that computation
%Inputs: an HMM and, and observation sequence O
function [prob,alpha] = forward(hmm,O)

%create alpha
alpha = zeros(hmm.N,length(O));

%initialise alpha
alpha(:,1) = hmm.pi'.*hmm.B(:,O(1));

%induction step
for t = 2:length(O)
    for j = 1:hmm.N
        alpha(j,t) =hmm.B(j,O(t)) * sum(alpha(:,t-1).*hmm.A(:,j));
    end
end

%termination
prob = sum(alpha(:,length(O)));
```

A.4 The Forward Algorithm, with Scaling

```
%The Forward Algorithm, with Scaling
%returns log[p(O|model)], and the lattice of (normalized) alphas produced by that
%computation
%Inputs: an HMM and, and observation sequence O
function [logProb,alpha] = forward(hmm,O)

%create alpha and c
alpha = zeros(hmm.N,length(O));
c = ones(1,length(O));
```

```
%initialise alpha and c
alpha(:,1) = hmm.pi'.*hmm.B(:,O(1));
%calculation the scaling coefficient c(1)
c(1) = 1/sum(alpha(:,1));
%multiplying alpha(:,1) by c to normalize
alpha(:,1) = alpha(:,1) * c(1);

%induction step
for t = 2:length(O)
    for j = 1:hmm.N
        alpha(j,t) =hmm.B(j,O(t)) * sum(alpha(:,t-1).*hmm.A(:,j));
    end
    %calculating the scaling coefficient c(t), and normalizing alpha(:,t)
    c(t) = 1/sum(alpha(:,t));
    alpha(:,t) = alpha(:,t) * c(t);
end

%termination
logProb = - sum(log(c));
```

A.5 The Log-Viterbi Algorithm

```
%The Log-Viterbi Algorithm
%Returns the most likely state sequence, and the log probability of that
%sequence, called the viterbi approximation. Inputs: an HMM and, and
%observation sequence O
function [Q,Pstar] = viterbi(hmm,O)

%precompute logs of distributions
logpi = log(hmm.pi);
logA = log(hmm.A);
logB = log(hmm.B);

%create Q, phi, and psi
Q = zeros(1,length(O));
phi = zeros(hmm.N,length(O));
psi = zeros(hmm.N,length(O));

%initialise phi
phi(:,1) = logpi' + logB(:,O(1));

%induction step
```

```
for t = 2:length(O)
    for j = 1:hmm.N
        [maxPhi,psi(j,t)] = max(phi(:,t-1)+logA(:,j));
        phi(j,t) = logB(j,O(t)) + maxPhi;
    end
end

%termination
[Pstar,Q(length(O))] = max(phi(:,length(O)));

%backtracking
for t = (length(O)-1):-1:1 %note decreasing loop var
    Q(t) = psi(Q(t+1),t+1);
end
```

A.6 Viterbi Training

```
%Viterbi Training
%Takes in an initial HMM, and a list of observation sequences (An array of
%structures, where sequences(i).seq refers to the ith sequence. This allows
%sequences of various length) Returns the improved HMM, and the Viterbi
%approximation of the old HMM.
function [hmm2,viterbiApprox] = viterbiTrain(hmm,sequences)

numSeqs = length(sequences);
viterbiApprox = 0;

for i=1:numSeqs
    [Q,pStar] = viterbi(hmm,sequences(i).seq);
    stateSeq(i).Q = Q;
    viterbiApprox = viterbiApprox + pStar;
end

hmm2 = hmm;
%initialise parameters
pi = zeros(1,hmm.N);
A = zeros(hmm.N,hmm.N);
B = zeros(hmm.N,hmm.M);
for i=1:numSeqs
    pi(stateSeq(i).Q(1)) = pi(stateSeq(i).Q(1))+1;
    %increments B for the first state-obs pair
    B(stateSeq(i).Q(1),sequences(i).seq(1)) = B(stateSeq(i).Q(1), ...
    sequences(i).seq(1)) + 1;
```

```
    for j=2:length(stateSeq(i).Q)
        %increments the appropriate entry in the transition matrix
        A(stateSeq(i).Q(j-1),stateSeq(i).Q(j)) = A(stateSeq(i).Q(j-1), ...
        stateSeq(i).Q(j)) + 1;
        %increments the appropriate entry in the observation matrix
        B(stateSeq(i).Q(j),sequences(i).seq(j)) = B(stateSeq(i).Q(j), ...
        sequences(i).seq(j)) + 1;
    end
end
hmm2.pi = normalize(pi);
hmm2.A = mk_stochastic(A);
hmm2.B = mk_stochastic(B);
```

A.7 Viterbi Training - Classification Test

```
%Runs a classification test for the discrete HMM with vector quantization
%and Viterbi training, on the 'Japanese vowel' data set
clear all
symbols = 128; %number of observation symbols
states = 5; %number of HMM states
numClasses = 9; %number of categories for classification

%read training data from file
trainData = multiLoad('C:\Hidden Markov Models\Data\Japanese Vowels\...
As Downloaded\ae.train',12);

%concatenating data for Vector Quantization(VQ)
dataCat = trainData(1).seq;
for i = 2:length(trainData)
    dataCat = horzcat(dataCat,trainData(i).seq);
end
%train a VQ codebook on the training data, with  symbols
[codeBook, P, DH]=vqsplit(dataCat,symbols);

%arrange training data by subjects, with 30 observation sequences per
%subject
sub = 1;
for i = 1:length(trainData)
    if sub*30<i
        sub = sub+1;
    end
    [subjectTrain(sub).sequences(i-(30*(sub-1))).seq, dst]= ...
    VQIndex(trainData(i).seq,codeBook);
```

```
end

%load the test data. Note that the vector quantization was learned from the
%training data only!
testData = multiLoad('C:\Hidden Markov Models\Data\Japanese Vowels\As Downloaded\...
ae.test',12);
%arrange test data by subjects, with subjects 1 to 9 having 31 35 88 44 29
%24 40 50 29 test sequences respectively
sequencesPerSubject = cumsum([31 35 88 44 29 24 40 50 29]);
seqsPerSubjectWithZero = horzcat([0],sequencesPerSubject);
sub = 1;
for i = 1:length(testData)
    if i > sequencesPerSubject(sub)
        sub = sub+1;
    end
    [subjectTest(sub).sequences(i-seqsPerSubjectWithZero(sub)).seq, dst]...
    =VQIndex(testData(i).seq,codeBook);
end

%train the hmms for each class on the training data
maxRuns = 20;
thresh = 5;
for j=1:numClasses
        [newHmms(j),vitApprox] = viterbiConverge(randomHMM(states,symbols),...
        subjectTrain(j).sequences,maxRuns,thresh);
end

%Add a small positive quantity to stop some probabilities going to 0.
for j=1:numClasses
    newHmms(j).B = mk_stochastic(newHmms(j).B + (1/symbols)*0.05);
end

%calculate the viterbi approximations for each hmm for each sequence
for i=1:numClasses
    for j=1:length(subjectTest(i).sequences)
        for k=1:numClasses
            %vitApsSubject(i).vitAp(j,k) is the viterbi approximation for the kth
            %model,
            %tested on the jth test sequence from the ith subject
            [Q,vitApsSubject(i).vitAp(j,k)] = viterbi(newHmms(k),...
            subjectTest(i).sequences(j).seq);

        end
    end
end
```

```
%construct confusion matrix
confMatrix = zeros(numClasses,numClasses);
for i=1:numClasses
    for j=1:length(subjectTest(i).sequences)
        [dontCare,I] = max(vitApsSubject(i).vitAp(j,:));
        confMatrix(i,I) = confMatrix(i,I)+1;
    end
end
%display results
confMatrix
confMatrix = mk_stochastic(confMatrix)
accuracy = sum(diag(confMatrix))/numClasses
```

A.8 Discover

```
%Discover takes a collection of sequences, a null hypothesis rejection
%threshold, and a maximum number of states and returns the resulting model.
%Also returned are: the array of AIC, BIC, and log probabilities, as well
%as the array of models explored over the course of splitting. This code
%relies on the basic functions in Kevin Murphy's BNT.
function [model,AICArr,BICArr,logProb,modelArray] = discover(obs,alphaThresh,cap)
AICArr = zeros(1,cap);
BICArr = zeros(1,cap);
format('short')
Q = 1; %Number of states to begin with.
M = 1; %The number of mixtures. 1 for now!
model = init_L2R(obs,Q,M,'full'); %Creates a Q state model.
split = true; %A Flag
while split && (Q < cap) %Outer loop, with a cap on the number of states
    numStates = Q %for display purposes
    %Uses Baum-Welch to re-estimate model parameters
    [LL, model.prior, model.transmat, model.mu, model.Sigma, model.mixmat] = ...
        mhmm_em(obs, model.prior, model.transmat, model.mu, model.Sigma,...
         model.mixmat, 'max_iter', 20);
    [O Q M] = size(model.mu); %Retrieves the appropriate model size
    modelArray{Q} = model;
    %Calculates number of parameters for AIC and BIC.
    %        mu       sigma                 mixmat   transmat  prior
    params = O*Q*M + ((O*O*Q*M)+(O*Q*M))/2 + Q*(M-1) + Q*(Q-1) + (Q-1);
    logProb = mhmm_logprob(obs, model.prior, model.transmat, model.mu,...
     model.Sigma, model.mixmat);
    %Calculates the total number of observations
```

```
numObs = 0;
for i = 1:length(obs)
    numObs = numObs + length(obs{i});
end
%For display purposes
disp('number of observations')
disp(numObs)
disp('number of parameters:')
disp(params)
disp('AIC penalty:')
disp(2*params + (2*params*(params + 1))/(numObs-params-1))
disp('BIC penalty:')
disp(params*log(numObs))
AICArr(Q) = - 2*logProb(end) + 2*params + (2*params*(params + 1))/...
(numObs-params-1);
if (numObs-params-1)<0
    AICArr(Q) = Inf;
end
BICArr(Q) = - 2*logProb(end)+params*log(numObs);
%Uses viterbi to gather the data by states, and then calculates their
%autocorralation p values.
ACpVals = autoCpVals(model,obs);
%If any alphas are below the thresh, returns the state with the smallest,
%and re-inits the model to have an extra state
if min(ACpVals) < alphaThresh
    candidate = find(ACpVals == min(ACpVals),1) %returns the state to be split
    Q = Q+1;
    disp('Split due to significant autocorrelation')
    %Adds a state to the model
    model = newModelDueToAC(model,candidate);
else %only check for transitional structure if there is no significant AC
    if Q > 2 %transitional structure only makes sense if there are more than 2
    %states
        [pVals,counts] = getTransitPVals(model,obs); %computes pVals for
        %transition dependence
        if min(min(pVals)) < (alphaThresh/(Q-1))
            Q = Q+1;
            disp('Split due to transitional structure')
            candidate = find(min(pVals) == min(min(pVals)),1); %returns the
            %state to be split
            [fromStR,fromStC] = find(pVals == min(min(pVals)),1);
            model = newModelDueToTransits(model,candidate,...
            counts(:,:,candidate),fromStR); %adds a state
        else
            split = false; %Allows the main loop to exit, if there are no
```

```
                %further splits made
            end
        else split = false; %Allows the main loop to exit, if there are no further
        %splits made
        end
    end
end
%A Final BW refinement, just to see if the last split update had sufficient
%training
[LL, model.prior, model.transmat, model.mu, model.Sigma, model.mixmat] = ...
    mhmm_em(obs, model.prior, model.transmat, model.mu, model.Sigma, ...
    model.mixmat, 'max_iter', 20);
logProb = LL(end);
%Displays and plots after training
AICArr = AICArr(AICArr > 0);
BICArr = BICArr(BICArr > 0);
AICArr
BICArr
subplot(2,1,1)
hold off
for i = 1:O
    plot(obs{1}(i,1:length(obs{1})),'Color',[rand rand rand])
    hold on
end
subplot(2,1,2)
hold off
plot(AICArr,'-.r*')
hold on
plot(BICArr,':bs')
hold off
```

A.9 Auxiliary Routines - Splitting States from Transition Dependence

```
%Takes a model, the state to be split, the counts from the transition
%dependence test, and the appropriate row identified during the transition
%dependence test and returns a new model with one extra state
function [model] = newModelDueToTransits(model,candidate,counts,fromStR)
transmat = model.transmat;
[O Q M] = size(model.mu);
peturb = 0.01;
new = zeros(Q+1);
new(1:Q,1:Q) = transmat;
```

```
selfTransit = transmat(candidate,candidate);
newState = zeros(1,Q+1);
%Set self-transition to the value of the old states self-transition
newState(Q+1) = selfTransit;
%scales the new transition probabilities according to the space left by the
%state self transition prob
scaled = horzcat((counts(fromStR,:)/sum(counts(fromStR,:)))*(1-selfTransit),0);
%Combines the state self transition prob and the other (scaled) transition
%probs. The transition prob between the new and the old state is set to 0
new(Q+1,1:Q+1) = normalize((newState+scaled)+peturb);
%countsWithoutNewState
countsWNS = vertcat(counts(1:fromStR-1,:),counts(fromStR+1:Q,:));
oldState = zeros(1,Q+1);
%Set self-transition to the value of the old states self-transition
oldState(candidate) = selfTransit;
%scales the new transition probabilities according to the space left by
%the state self transition prob
oldScaled = horzcat((sum(countsWNS)/sum(sum(countsWNS)))*(1-selfTransit),0);
%Combines the state self transition prob and the other (scaled) transition
%probs. The transition prob between the new and the old state is set to 0 + peturb
new(candidate,1:Q+1) = normalize(oldState+oldScaled+peturb);
for i = 1:Q
    for j = 1:Q
        if ((j==candidate) && (i ~= candidate))
            new(i,j) = transmat(i,j)/2;
            new(i,Q+1) = transmat(i,j)/2;
        end
    end
end
%Normalizes rows in case of numerical error
model.transmat = mk_stochastic(new);
Q = Q+1;
oldMean = model.mu(:,candidate,1);
oldVar = model.Sigma(:,:,candidate,1);
model.mu(:,candidate,1) = oldMean;
model.mu = cat(2,model.mu,oldMean);
model.Sigma = cat(3,model.Sigma,oldVar);
model.mixmat = ones(Q,1);
model.prior(candidate) = model.prior(candidate)/2;
model.prior = normalise(vertcat(model.prior,model.prior(candidate)));
```

A.10 Auxiliary Routines - Splitting States from Autocorrelation

```
%Takes a model and a state to split, and returns a new model.
function [model] = newModelDueToAC(model,candidate)
[O Q M] = size(model.mu);
Q = Q+1;
%Adds a state to the model
model.transmat = newTransmat(model.transmat,candidate);
%Assigns appropriate distributions
oldMean = model.mu(:,candidate,1);
oldVar = model.Sigma(:,:,candidate,1);
oldStd = diag(sqrt(oldVar));
model.mu(:,candidate,1) = oldMean-(oldStd);
model.mu = cat(2,model.mu,oldMean+(oldStd));
model.Sigma = cat(3,model.Sigma,oldVar);
model.mixmat = ones(Q,1);
model.prior(candidate) = model.prior(candidate)/2;
model.prior = normalise(vertcat(model.prior,model.prior(candidate)));

%appends a new state at pos Q+1 of transmat, with transitions to the new
%and the candidate state = 1/2 the transitions to the candidate state
function [transmat] = newTransmat(transmat,state)
[Q Q] = size(transmat);
new = zeros(Q+1);
new(1:Q,1:Q) = transmat;
new(Q+1,1:Q) = transmat(state,:);
new(:,Q+1) = new(:,state)/2;
new(:,state) = new(:,Q+1);
transmat = mk_stochastic(new);
```

A.11 Auxiliary Routines - Computing p-Values for Autocorrelation

```
%Takes a model and some data and returns a list of p-values for within
%state autocorrelation
function [ACpVals] = autoCpVals(model0,data)
[O Q M] = size(model0.mu);
numSeqs = length(data);
%Calculating the paths using the viterbi algorithm
paths = cell(numSeqs);
for i = 1:numSeqs
```

```
    B = mixgauss_prob(data{i}, model0.mu, model0.Sigma, model0.mixmat);
    paths{i} = viterbi_path(model0.prior, model0.transmat, B);
end
%Initialising the data by state. We need 2, because the first is shifted
%one way, and the second the other
datByState = cell(Q,2);
for j = 1:Q
    datByState{j,1} = [];
    datByState{j,2} = [];
end
%Gathers the outputs by state, but removes one element from either side of
%each current run before concatenating it with the each accumulation. This
%is to produce the shift required for autocorrelation.
for i = 1:numSeqs
    seqLength = length(paths{i});
    pointer = 1;
    %Iterates through the entire sequence
    while pointer < seqLength + 1
        %Stores the outputs for the current run of repeating states
        currentRun = [];
        state = paths{i}(pointer);
        %paths{i}(t) is the estimated state of the model at timestep t, for
        %sequence i
        while (pointer < seqLength + 1) && (paths{i}(pointer) == state)
            currentRun = horzcat(currentRun,data{i}(:,pointer));
            pointer = pointer + 1;
        end
        datByState{state,1} = horzcat(datByState{state,1},currentRun(:,2:end));
        datByState{state,2} = horzcat(datByState{state,2},currentRun(:,1:end-1));
    end
end
%Extracts autocorrelations of data by state
ACpVals = ones(Q,O);
for i = 1:Q
    for feat = 1:O
        if ((not(isempty(datByState{i}))) && (length(datByState{i,1}(feat,:)) > 2))
            [R,P] = corr(datByState{i,1}(feat,:)',datByState{i,2}(feat,:)');
            if (~isnan(P))
                ACpVals(i,feat) = P;
            end
        end
    end
end
%Multiplied by the number of observations to correct for multiple tests
ACpVals = min(ACpVals,[],2)*O;
```

A.12 Auxiliary Routines - Computing p-Values for Transition Dependence

```
function [pVals,counts] = getTransitPVals(model0,data)
[O Q M] = size(model0.mu);
numSeqs = length(data);
%Calculating the paths using the viterbi algorithm
paths = cell(numSeqs);
for i = 1:numSeqs
    B = mixgauss_prob(data{i}, model0.mu, model0.Sigma, model0.mixmat);
    paths{i} = viterbi_path(model0.prior, model0.transmat, B);
end
%Create a paths with duplicate elements removed
for i = 1:numSeqs
    noDupes{i}(1) = paths{i}(1);
    for t = 2:length(paths{i})
        if noDupes{i}(end)~= paths{i}(t)
            noDupes{i} = horzcat(noDupes{i},paths{i}(t));
        end
    end
end
%Extract expected transitions
counts = zeros(Q,Q,Q); %[from Q, to Q, for each Q]
for i = 1:numSeqs
    for t = 2:(length(noDupes{i})-1)
        counts(noDupes{i}(t-1),noDupes{i}(t+1),noDupes{i}(t)) = ...
        counts(noDupes{i}(t-1),noDupes{i}(t+1),noDupes{i}(t)) + 1;
    end
end
%refSt is the state you are checking for splitting potential. fromSt is the
%histogram of toState counts to check against the transmat entry for refSt
pVals = ones(Q,Q); %[fromSt,refState]
for refSt = 1:Q
    for fromSt = 1:Q
        if refSt ~= fromSt
            bins = 1:Q-1;
            obsCountsWithSt = counts(fromSt,:,refSt);
            obsCounts = horzcat(obsCountsWithSt(1:refSt-1),...
            obsCountsWithSt(refSt+1:Q));
            n = sum(obsCounts);
            allOtherCounts = vertcat(counts(1:fromSt,:,refSt),...
            counts(fromSt:Q,:,refSt));
            sumOtherCounts = sum(allOtherCounts);
            expCounts = normalise(horzcat(sumOtherCounts(1:refSt-1),...
```

```
            sumOtherCounts(refSt+1:Q)));
            expCounts = n*expCounts;
            [h,p] = chi2gof(bins,'ctrs',bins,...
                'frequency',obsCounts, ...
                'expected',expCounts,...
                'emin',0);
            pVals(fromSt,refSt) = p;
        end
    end
end
```

List of References

[1] T. K. Abdel-Galil, E. F. El-Saadany, A. M. Youssef, and M. M. A. Salama. Disturbance classification using hidden Markov models and vector quantization. *IEEE Transactions on Power Delivery*, 20(3):2129–2135, 2005.

[2] T. Al-Ani, Y. Hamam, R. Fodil, F. Lofaso, and D. Isabey. Using hidden Markov models for sleep disordered breathing identification. *Simulation Modelling Practice and Theory*, 12(2):117–128, 2004.

[3] R. M. K. Altman and A. J. Petkau. Application of hidden Markov models to multiple sclerosis lesion count data. *Statistics in Medicine*, 24(15):2335–2344, 2005.

[4] Y. Altun, I. Tsochantaridis, and T. Hofmann. Hidden Markov support vector machines. In *Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003)*, volume 20, 2003.

[5] V. V. Anisimov, H. J. Maas, M. Danhof, and O. Della Pasqua. Analysis of responses in migraine modelling using hidden Markov models. *Statistics in Medicine*, 26(22):4163–4178, 2007.

[6] F. R. Bach and M. I. Jordan. Discriminative training of hidden Markov models for multiple pitch tracking. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, 2005.

[7] P. Baldi and Y. Chauvin. Smooth on-line learning algorithms for hidden Markov models. *Neural Computation*, 6(2):307–318, 1994.

[8] M.J. Beal, Z. Ghahramani, and C.E. Rasmussen. The infinite hidden Markov model. *Advances in Neural Information Processing Systems*, 1:577–584, 2002.

[9] B. Betro, A. Bodini, and Q. A. Cossu. Using a hidden Markov model to analyse extreme rainfall events in Central-East Sardinia. *Environmetrics*, 19(7):702–713, 2008.

[10] M. Beyreuther, R. Carniel, and J. Wassermann. Continuous hidden Markov models: Application to automatic earthquake detection and classification at Las Candas caldera, Tenerife. *Journal of Volcanology and Geothermal Research*, 176(4):513–518, 2008.

[11] E. Birney. Hidden Markov models in biological sequence analysis. *IBM Journal of Research and Development*, 45(3-4):449–454, 2001.

[12] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 1st edition, October 2007.

[13] J. Bond, V. Petroff, S. OBrien, and D. Bond. Forecasting turmoil in indonesia: An application of hidden markov models. In *Annual Meeting of the International Studies Association, Montreal*, 2004.

[14] M. Brand. An entropic estimator for structure discovery. *Advances in Neural Information Processing Systems*, pages 723–729, 1999.

[15] K.P. Burnham and D.R. Anderson. Multimodel Inference: Understanding AIC and BIC in Model Selection. *Sociological Methods Research*, 33(2):261–304, November 2004.

[16] M. Butler. *Hidden Markov Model Clustering of Acoustic Data*. PhD thesis, University of Edinburgh, 2003.

[17] M. Chen, B. Huang, and Y. Xu. Human abnormal gait modeling via hidden Markov model. In *International Conference on Information Acquisition. ICIA'07.*, pages 517–522, 2007.

[18] S. Chiappa and S. Bengio. HMM and IOHMM modeling of EEG rhythms for asynchronous BCI systems. In *European Symposium on Artificial Neural Networks, ESANN*, pages 193–204, 2004.

[19] B. Cooper and M. Lipsitch. The analysis of hospital infection data using hidden Markov models. *Biostatistics*, 5(2):223–237, 2004.

[20] C. Croarkin and P. Tobias. *NIST/SEMATECH e-handbook of statistical methods*. NIST Gaithersburg, MD, USA, 2008.

[21] A. Konushin E. Lomakina-Rumyantsev I. Zarayskaya D. Vetrov, D. Kropotov and K. Anokhin. Automatic segmentation of mouse behavior using hidden Markov model. In *6th International Conference on Methods and Techniques in Behavioral Research*, 2008.

[22] P. Dayan and L.F. Abbott. *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press, 2001.

[23] P. Fearnhead. Exact and efficient Bayesian inference for multiple changepoint problems. *Statistics and Computing*, 16(2):203–213, 2006.

[24] A. Flexer, E. Pampalk, and G. Widmer. Hidden Markov models for spectral similarity of songs. In *Proceedings of the 8th International Conference on Digital Audio Effects*, 2005.

[25] A. Franke, T. Caelli, and R. J. Hudson. Analysis of movements and behavior of caribou (Rangifer tarandus) using hidden Markov models. *Ecological Modelling*, 173(2-3):259–270, 2004.

[26] A. Franke, T. Caelli, G. Kuzyk, and R. J. Hudson. Prediction of wolf (Canis lupus) kill-sites using hidden Markov models. *Ecological modelling*, 197(1-2):237–246, 2006.

[27] M. Ge, R. Du, and Y. Xu. Hidden Markov model based fault diagnosis for stamping processes. *Mechanical Systems and Signal Processing*, 18(2):391–408, 2004.

[28] A.M. Greene, A.W. Robertson, and S. Kirshner. Analysis of Indian monsoon daily rainfall on subseasonal to multidecadal time-scales using a hidden Markov model. *Quarterly Journal of the Royal Meteorological Society*, 134(633):875–888, 2008.

[29] M. Hayashi. Hidden Markov models to identify pilot instrument scanning and attention patterns. In *IEEE International Conference on Systems Man and Cybernetics*, volume 3, pages 2889–2896, 2003.

[30] M. Hayashi, B. Beutter, and R. S. McCann. Hidden Markov model analysis for space shuttle crewmembers' scanning behavior. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 2, 2005.

[31] J. A. Herbst, S. Gammeter, D. Ferrero, and R. H. R. Hahnloser. Spike sorting with hidden Markov models. *Journal of Neuroscience Methods*, 174(1):126–134, 2008.

[32] S. Hettich and SD Bay. The UCI KDD Archive. Irvine, CA: University of California, Department of Information and Computer Science, 1999.

[33] L. M. Jones, A. Fontanini, B. F. Sadacca, P. Miller, and D. B. Katz. Natural stimuli evoke dynamic sequences of states in sensory cortical ensembles. *Proceedings of the National Academy of Sciences*, 104(47):18772, 2007.

[34] B. H. Juang and L. Rabiner. The segmental K-means algorithm for estimating parameters of hidden Markov models. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 38(9):1639–1641, 1990.

[35] M.W. Kadous. *Temporal classification: Extending the classification paradigm to multivariate time series*. PhD thesis, The University of New South Wales, 2002.

[36] R. Kelley, A. Tavakkoli, C. King, M. Nicolescu, M. Nicolescu, and G. Bebis. Understanding human intentions via hidden Markov models in autonomous mobile robots. In *Proceedings of the 3rd ACM/IEEE International Conference on Human Robot Interaction*, pages 367–374, 2008.

[37] C. Kemere, G. Santhanam, B. M. Yu, A. Afshar, S. I. Ryu, T. H. Meng, and K. V. Shenoy. Detecting Neural-State transitions using hidden markov models for motor cortical prostheses. *Journal of Neurophysiology*, 100(4):2441, 2008.

[38] A. Krogh, M. Brown, I. S. Mian, K. Sjolander, and D. Haussler. Hidden markov models in computational biology. applications to protein modeling. *Journal of Molecular Biology*, 235(5):1501–1531, 1994.

[39] K. C. Kwon and J. H. Kim. Accident identification in nuclear power plants using hidden Markov models. *Engineering Applications of Artificial Intelligence*, 12(4):491–501, 1999.

[40] S. Kwong, C. W. Chau, K. F. Man, and K. S. Tang. Optimisation of HMM topology and its model parameters by genetic algorithms. *Pattern Recognition*, 34(2):509–522, 2001.

[41] D. Lang, M. Klaas, and N. de Freitas. Insights on fast kernel density estimation algorithms. *Department of Computer Science, University of Toronto, Technical Report*, 2004.

[42] G. Soules L.E. Baum, T. Petrie and N. Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *The Annals of Mathematical Statistics*, 41(1):164–171, February 1970.

[43] D. Lee and Y. Nakamura. Relaxed Viterbi training: Fast and reliable symbol acquisition of mimesis model. *Nippon Robotto Gakkai Gakujutsu Koenkai Yokoshu*, 24:2–22, 2006.

[44] H. Lee and S. Choi. PCA+ HMM+ SVM for EEG pattern classification. In *Seventh International Symposium on Signal Processing and its Applications*, volume 1, 2003.

[45] H. Lee and A. Y. Ng. Spam deobfuscation using a hidden Markov model. In *Conference on Email and Anti-Spam*, 2005.

[46] J. J. Lee, J. Kim, and J. H. Kim. Data-driven design of HMM topology for online handwriting recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 15(1):107–121, 2001.

[47] C. Li. A bayesian approach for structural learning with hidden markov models. *Scientific Programming*, 10(3):201–219, 2002.

[48] C. Li and G. Biswas. Temporal pattern generation using hidden Markov model based unsupervised classification. *Lecture Notes in Computer Science*, pages 245–256, 1999.

[49] C. Li, G. Biswas, M. Dale, and P. Dale. Building Models of Ecological Dynamics Using HMM Based Temporal Data Clustering-A Preliminary Study. In *Proceedings of the 4th International Conference on Advances in Intelligent Data Analysis*, pages 53–62, 2001.

[50] J. Li, J. Wang, Y. Zhao, and Z. Yang. Self-adaptive design of hidden Markov models. *Pattern Recognition Letters*, 25(2):197–210, 2004.

[51] G.M. Ljung and G.E.P. Box. On a measure of lack of fit in time series models. *Biometrika*, 65(2):297–303, 1978.

[52] J. Toyama M. Kudo and M. Shimbo. Multidimensional curve classification using passing-through regions. *Pattern Recognition Letters*, 20(11–13):1103–1111, 1999.

[53] I. L. Macdonald and D. Raubenheimer. Hidden Markov models and animal behaviour. *Biometrical Journal*, 37(6):701–712, 1995.

[54] Y. Matsumoto and R. Thawonmas. MMOG player classification using hidden Markov models. *Lecture Notes in Computer Science*, pages 429–434, 2004.

[55] E. S. McBryde, A. N. Pettitt, B. S. Cooper, and D. L. McElwain. Characterizing an outbreak of vancomycin-resistant enterococci using hidden Markov models. *Journal of The Royal Society Interface*, 4(15):745–754, 2007.

[56] C. A. Micchelli and P. Olsen. Penalized maximum-likelihood estimation, the Baum-Welch algorithm, diagonal balancing of symmetric matrices and applications to training acoustic data. *Journal of Computational and Applied Mathematics*, 119(1-2):301–331, July 2000.

[57] J. Mlıch and P. Chmelar. Trajectory classification based on hidden Markov models. pages 101–105, 2008.

[58] K. Murphy. The Bayes Net Toolbox for Matlab. In *Computing Science and Statistics: Proceedings of Interface*, volume 33, 2001.

[59] B. Murrell and J. Tapamo. Heuristics for state splitting in hidden Markov models. In *The 19th Annual Symposium for the Pattern Recognition Association of South Africa*, pages 3–8, 2008.

[60] K. Noland and M. Sandler. Key estimation using a hidden Markov model. In *Seventh International Symposium on Music Information Retrieval*, volume 10, 2006.

[61] D. Novak, Y. H. T. Al-Ani, and L. Lhotska. Electroencephalogram processing using hidden Markov models. *Transdisciplinary Biomedical Engineering Research Report*, 2003.

[62] M. Ostendorf and H. Singer. HMM topology design using maximum likelihood successive state splitting. *Computer Speech & Language*, 11(1):17–41, 1997.

[63] J. R. Otterpohl, J. D. Haynes, F. Emmert-Streib, G. Vetter, and K. Pawelzik. Extracting the dynamics of perceptual switching from noisy behaviour: An application of hidden Markov modelling to pecking data from pigeons. *Journal of Physiology-Paris*, 94(5-6):555–567, 2000.

[64] T. Pfau, M. Ferrari, K. Parsons, and A. Wilson. A hidden Markov model-based stride segmentation technique applied to equine inertial sensor trunk movement data. *Journal of Biomechanics*, 41(1):216–220, 2008.

[65] M. Piccardi and O. Perez. Hidden Markov models with kernel density estimation of emission probabilities and their use in activity recognition. In *IEEE Conference on Computer Vision and Pattern Recognition. CVPR'07*, pages 1–8, 2007.

[66] A. Pikrakis, S. Theodoridis, and D. Kamarotos. Classification of musical patterns using variable duration hidden Markov models. *IEEE Transactions on Audio, Speech, and Language Processing*, 14(5):1795–1807, 2006.

[67] L. Rabiner. Correction to:"a tutorial on hidden Markov models and selected applications in speech recognition". `http://www.cs.iastate.edu/~honavar/rabiner-errata.pdf`.

[68] L. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77:257—286, 1989.

[69] J. Rasku, M. Juhola, E. Toppila, and I. Pyykk. Recognition of balance signals between healthy subjects and otoneurological patients with hidden Markov models. *Biomedical Signal Processing and Control*, 2(1):1–8, 2007.

[70] L. J. Rodriguez and I. Torres. Comparative study of the Baum-Welch and Viterbi training algorithms applied to read and spontaneous speech recognition. *Lecture Notes in Computer Science*, 2652:847–857, 2003.

[71] A. Senior. A hidden Markov model fingerprint classifier. In *Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 306–310. Computer Society Press, 1997.

[72] F. Sha and L. Saul. Large-margin training of continuous-density hidden Markov models. *Neural Information Processing Systems*, 2006.

[73] C.R. Shalizi, K.L. Shalizi, and J.P. Crutchfield. Pattern discovery in time series, Part I: Theory, algorithm, analysis, and convergence. *Journal of Machine Learning Research*, pages 2–10, 2002.

[74] S. M. Siddiqi, G. J. Gordon, and A. W. Moore. Fast state discovery for HMM model selection and learning. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 2007.

[75] J. Simola, J. Salojrvi, and I. Kojo. Using hidden Markov model to uncover processing states from eye movements in information search tasks. *Cognitive Systems Research*, 9(4):237–251, 2008.

[76] I. Simon, D. Morris, and S. Basu. MySong: automatic accompaniment generation for vocal melodies. 2008.

[77] T. Smith and P. Vounatsou. Estimation of infection and recovery rates for highly polymorphic parasites when detectability is imperfect, using hidden Markov models. *Epidemiology*, 10:11–12, 2003.

[78] T. Srinivasan, S. P. Srinivasan, S. K. Alur, and P. Chandrasekaran. An efficient goalie strategy using twin hidden Markov models. In *7th IEEE International Conference on Computer and Information Technology*, pages 157–164, 2007.

[79] T. Starner, J. Weaver, and A. Pentland. Real-time American Sign Language recognition using desk and wearable computer based video. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(12):1371–1375, 1998.

[80] B. Stenger, V. Ramesh, N. Paragios, F. Coetzee, and J. M. Buhmann. Topology free hidden Markov models: Application to background modeling. In *Eighth IEEE International Conference on Computer Vision*, volume 1, pages 297–301, 2001.

[81] A. Stolcke and S. Omohundro. Hidden Markov model induction by Bayesian model merging. *Advances in Neural Information Processing Systems*, pages 11–18, 1992.

[82] A. Stolcke and S. M. Omohundro. Best-first model merging for hidden Markov model induction. *Arxiv preprint cmp-lg/9405017*, 1994.

[83] A. H. Tai, W. K. Ching, and L. Y. Chan. Detection of machine failure: Hidden Markov model approach. *Computers & Industrial Engineering*, 57(2):608–619, 2009.

[84] J. Takami and S. Sagayama. A successive state splitting algorithm for efficient allophone modeling. In *IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP-92.*, volume 1, 1992.

[85] F. K. Tokatli and A. Cinar. Fault detection and diagnosis in a food pasteurization process with hidden Markov models. *The Canadian Journal of Chemical Engineering*, 82(6):1252–1262, 2004.

[86] B. U. Toreyin, Y. Dedeoglu, and A. E. Cetin. Flame detection in video using hidden Markov models. In *IEEE International Conference on Image Processing*, volume 2, pages 1230–1233, 2005.

[87] B. C. Tucker and M. Anand. On the use of stationary versus hidden Markov models to detect simple versus complex ecological dynamics. *Ecological Modelling*, 185(2-4):177–193, 2005.

[88] A. van den Hout, C. Jagger, and F. E. Matthews. Estimating life expectancy in health and ill health by using a hidden Markov model. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 58(4):449–465, 2009.

[89] J. Van Gael, Y. Saatci, Y.W. Teh, and Z. Ghahramani. Beam sampling for the infinite hidden Markov model. *Proceedings of the 25th international conference on Machine learning*, pages 1088–1095, 2008.

[90] N. Viovy and G. Saint. Hidden markov models applied to vegetation dynamics analysis usingsatellite remote sensing. *IEEE Transactions on Geoscience and Remote Sensing*, 32(4):906–917, 1994.

[91] C. Vogler and D. Metaxas. Adapting hidden Markov models for ASL recognition by using three-dimensional computer vision methods. In *IEEE International Conference on Systems, Man, and Cybernetics: Computational Cybernetics and Simulation*, volume 1, pages 156–161, 1997.

[92] Y. K. Wang, K. C. Fan, Y. T. Juang, and T. H. Chen. Using hidden Markov model for Chinese business card recognition. In *Proceedings of the International Conference on Image Processing*, volume 1, 2001.

[93] C. Yang, R. Duraiswami, N.A. Gumerov, and L. Davis. Improved fast Gauss transform and efficient kernel density estimation. In *IEEE International Conference on Computer Vision*, pages 464–471, 2003.

[94] O. N. Yogurtcu, E. Erzin, and A. Gursoy. Extracting gene regulation information from microarray time-series data using hidden Markov models. *Lecture Notes in Computer Science*, 4263:144, 2006.

[95] H. Zhang, Z. Jiang, J. Y. Pi, H. K. Xu, and R. Du. On-line monitoring of pharmaceutical production processes using hidden Markov model. *Journal of Pharmaceutical Sciences*, 98(4):1487, 2009.

[96] Yingjian Zhang. *Prediction of Financial Time Series with Hidden Markov Models*. PhD thesis, Shandong University, 2004.

[97] X. Zou and D.M. Levinson. Modeling pipeline driving behaviors: Hidden Markov model approach. *Transportation Research Record: Journal of the Transportation Research Board*, 1980(1):16–23, 2006.