Hidden Markov Models: Applications to Flash Memory data and Hospital Arrival times

MSci Individual Project Report

Tiberiu Chis (tc207)

Project supervisor: Prof. Peter Harrison

Second marker: Dr. William Knottenbelt

June 23, 2011

Abstract

A hidden Markov model (HMM) is a bivariate Markov chain which encodes information about the evolution of a time series. HMMs can faithfully represent workloads for discrete time processes and therefore be used as portable benchmarks to explain and predict the complex behaviour of these processes. This project introduces the main concepts of HMMs for discrete time series including a summary of HMM mathematical properties. A section of this report explains the motives behind cluster analysis and the most efficient selection of the clustering algorithm when creating workload models. In the case of this project, an explanation is provided into the benefits of the K-means clustering algorithm for data points in discrete time.

The main aims of this project are to: apply HMMs to two different scenarios to correctly analyse discrete time series; provide meaning to the underlying hidden states of the HMMs in each case; and recreate representative traces for each application. Firstly, the HMM is applied to Flash Memory data in the form of operation type traces to achieve a workload model. Secondly, the HMM is used to decode a data trace formed of hospital patient arrivals creating a Hospital Arrivals model. Both of these models will be validated using averages from the raw and HMM-generated traces and also by comparison of autocorrelation functions.

Another aim of the project is to create a novel adaptation of the Baum-Welch algorithm using Flash Memory data. It is known that discrete HMMs can effectively learn long sequences of observations such as workload access patterns in computer storage systems. However, there is now increasing demand for systems which handle higher density, additional loads as seen in storage workload modelling [1], where workloads can be characterized on-line. Thus, we derive a sliding version of the Baum-Welch algorithm, which constantly updates its observation set, discarding old data points in the time series as it inputs new ones. We refer to a HMM with this sliding Baum-Welch algorithm as a SlidHMM due to the fact that it slides across the time series, updating its parameters "on-the-fly". The benefit of this novel approach is to obtain a parsimonious model which updates its encoded information whenever more real time workload data becomes available. The SlidHMM is also efficient in keeping track of non-homogeneous processes because it updates the observation set at different stages of the analysis, therefore analysing only the current portion of the time series.

An analysis of an efficient process to identify the optimal number of hidden states for a HMM is also discussed, but left mostly as future work. Also reserved as extensions are: the choice of a different clustering algorithm for each model; and a new approximation for the backward variables in the Baum-Welch algorithm to seek an improvement on the SlidHMM.

Acknowledgements

I would firstly like to thank Professor Peter Harrison for his guidance, patience and support throughout this process.

I would also like to thank Dr William Knottenbelt for his advice and constructive comments at the beginning of this project.

Contents

Chapter 1

Introduction

The hidden Markov model (HMM) has become one of the simplest and most widely used statistical tools for modelling discrete times series. The HMM is not only efficient, through its parameter estimation algorithm, but it is also flexible, with applications ranging from patient waiting times in hospitals to dysarthric Speech Recognition and even automatic earthquake detection and classification. HMMs can often supply answers to the future behaviour of time series and provide representation for inputs of large, complex systems.

In this project, we explore the applicability of the HMM to two such systems, namely instructions arriving at a specified server and patients arriving at a hospital. In chapter 2 we define a HMM and discuss some of its important mathematical properties. Then, we describe the three fundamental problems associated with HMMs: firstly, to determine the likelihood of an observed sequence O given the model $\lambda = (A, B, \pi)$, which is written as $P(O \mid \lambda)$; secondly, to maximize $P(O \mid \lambda)$ by adjusting the parameters A, B, π; thirdly, to find the optimal state sequence reponsible for producing the observation set. We attempt to solve these three problems by using the Forward-Backward algorithm, the Baum-Welch algorithm and the Viterbi algorithm, respectively. Next, a straightforward example is considered to explain the formation of HMMs. Finally, after researching a wide range of applications of HMMs, we examined the potential of these models for efficient problem solving and information storing.

We dedicate chapter 3 to clustering algorithms as they provide the stepping stones to creating the HMMs through the formation of an observation set. We will discuss the purpose of clustering in general, summarising the advantages and disadvantages for each type of clustering. Then, we explain the motivation behind choosing to implement our K-means clustering when building our HMMs.

In chapter 4, we will apply HMMs to Flash Memory data to create a discrete Markov arrival process (MAP), also known as a Flash Memory workload model. We will describe how we use the raw trace of read and write commands to create a binned trace. We then discuss how we used the K-means clustering algorithm and the Baum-Welch algorithm to process the binned trace and obtain our MAP parameters.

An analysis of the results for our Flash HMM will be carried out, notably summary average statistics and autocorrelation functions.

One of the main contributions of this project with respect to novelty, is the creation of a sliding version of the Baum-Welch algorithm. This sliding HMM (which we refer to as **SlidHMM**) is formed using our MAP parameters and the Flash Memory trace and incorporates the idea of a moving average on the observations. Essentially, the SlidHMM parameters will be updated "on-the-fly" as more workload data becomes available and, as a result, SlidHMM can easily keep track of variable time-dependent processes. More generally, this type of model is desirable in industry for its potential of run time analysis and planning. As we shall see, an additional benefit of SlidHMM is that it only needs to update variables for the new incoming observations thus reducing the computational burden (space and time complexity) of the Baum-Welch algorithm, especially for very large observation sets.

The main challenge with this type of model comes in the dependency of its parameters on all preceding data. As we will described later in more detail, this problem requires an approximation on the new backward variables of the Baum-Welch algorithm to save computing the terms for the accumulated observation set. In fact, the difficulty in achieving an accurate approximation for the backward recurrence formula might explain why very little work has been done in this domain in the last ten years.

We continue this report by applying HMMs to hospital patient arrivals as will be seen in chapter 5. This consists of creating the Hospital Arrivals model with similar steps to our Flash Memory model, but focusing more on the use of the Viterbi algorithm to reveal the identity of the hidden states. We will analyse the results of the Baum-Welch and Viterbi algorithms by generating averages and also displaying autocorrelation functions to compare raw and HMM traces. Finally, we investigate the criteria needed to find the optimal number of hidden states for a HMM. This involves the set up of many hidden states and the gradual merging of these states until the HMM has optimal transition and emission probability matrices.

To complete this report, we evaluate our results in chapter 6 and provide justification of our models and choice of parameters. A comparison with exisiting work in the field is also included to scale our achievements against industry standards. Lastly, in chapter 7 we present a summary of our contributions along with possible future work as a continuation of this project.

Chapter 2

Background

In this chapter, we will go through what a hidden Markov model is, and define some of its important properties. Then, we shall briefly discuss the fundamental problems associated with hidden Markov models before moving on to a simple example explaining the main concepts of hidden Markov models. After that, we explain in detail the solutions of the three fundamental problems of hidden Markov models. Finally, we finish this chapter by explaining several applications where hidden Markov models are used in the real world.

2.1 What is a hidden Markov model?

A hidden Markov model (HMM) is a probabilistic model (more specifically, a bivariate Markov chain) which encodes information about the evolution of a time series. The HMM is made up of the following: a hidden Markov chain $\{C_k\}$ (where k is an integer) which has states that are not directly observable, and a discrete time stochastic process $\{O_k\}_{k\geq 0}$ which is observable. Putting those together, we get the bivariate Markov chain $\{(C_k, O_k)\}_{k\geq 0}$ where all the statistical inference is done on $\{O_k\}$, as $\{C_k\}$ is not observed. Another point worth mentioning is that C_k governs the distribution of the corresponding O_k, and thus we assume that C_k is the only variable of the Markov chain that affects the O_k distribution. We now give a formal definition of a HMM and show some of its important properties.

Definition 1.1 *Let $\{C_t\}_{t\in\mathbb{N}}$ be a stochastic process belonging to state space $S = \{1, \ldots, r\}$. Then $\{C_t\}_{t\in\mathbb{N}}$ is a* **Markov chain** *if*

$$P(C_{t+1} = c_{t+1} \mid C_t = c_t, C_{t-1} = c_{t-1}, \ldots, C_1 = c_1) = P(C_{t+1} = c_{t+1} \mid C_t = c_t)$$

where $c_1, c_2, \ldots, c_{t+1} \in S$.

Definition 1.2 *Suppose we have a Markov chain $\{C_t\}_{t\in\mathbb{N}}$ with state space $S = \{1, \ldots, r\}$, transition matrix $Q = (q_{cc'})_{c,c'\in S}$ and an initial distribution $\upsilon_c (c \in S)$. We also have a stochastic process $\{O_t\}_{t\in\mathbb{N}}$ which takes values in $J = \{1, \ldots, m\}$. Then, the bivariate stochastic process $\{(C_t, O_t)\}_{t\in\mathbb{N}}$ is a* **hidden Markov model** *if it is a Markov chain with transition probabilities*

$$P(C_t = c_t, O_t = o_t \mid C_{t-1} = c_{t-1}, O_{t-1} = o_{t-1}) = P(C_t = c_t, O_t = o_t \mid C_{t-1} = c_{t-1})$$

$$= q_{c_{t-1}c_t} g_{c_t o_t}$$

where $G = (g_{co})_{c \in S, o \in J}$ *is a stochastic matrix.*

We also use the property, which states that conditionally on $\{C_t\}_{t=0,1,\ldots}$ we have that $\{O_t\}_{t=0,1,\ldots}$ are independent. In other words,

$$P(O_0 = o_0, \ldots, O_n = o_n \mid C_0 = c_0, \ldots, C_n = c_n) = \prod_{i=0}^{n} P(O_i = o_i \mid C_i = c_i)$$

Now, we will prove the following proposition.

Proposition 1.2 $P(O_t = o_t \mid C_t = c_t) = g_{c_t, o_t}$

Proof

$$P(O_t = o_t \mid C_t = c_t) = \frac{P(O_t = o_t, C_t = c_t)}{P(C_t = c_t)}$$

$$= \frac{1}{P(C_t = c_t)} \sum_{c_{t-1}} P(O_t = o_t, C_t = c_t, C_{t-1} = c_{t-1})$$

$$= \frac{1}{P(C_t = c_t)} \sum_{c_{t-1}} P(O_t = o_t, C_t = c_t \mid C_{t-1} = c_{t-1}) P(C_{t-1} = c_{t-1})$$

$$= \frac{1}{P(C_t = c_t)} \sum_{c_{t-1}} p_{c_{t-1}c_t} g_{c_t, o_t} P(C_{t-1} = c_{t-1})$$

$$= \frac{g_{c_t, o_t}}{P(C_t = c_t)} \sum_{c_{t-1}} P(C_t = c_t, C_{t-1} = c_{t-1})$$

$$= \frac{g_{c_t, o_t}}{P(C_t = c_t)} P(C_t = c_t)$$

$$= g_{c_t, o_t}$$

which is the RHS as required. □

We continue by expanding on the proofs (seen in [2]) used in the derivation of the Baum-Welch algorithm. Note that, as the following proofs are longer and require more space for each line, we will simplify the notation and replace the event $O_t = o_t$ by O_t from now on.

Proposition 1.3 For all integers t and l such that $1 \leq t \leq l \leq T$

$$P(O_l, O_{l+1}, \ldots, O_T \mid C_t, \ldots, C_T) = P(O_l, O_{l+1}, \ldots, O_T \mid C_l, \ldots, C_T)$$

Proof From the definition of conditional probability, we begin by writing the LHS as:

$$P(O_l, O_{l+1}, \ldots, O_T \mid C_t, \ldots, C_T)$$

$$= \frac{1}{P(C_t,\ldots,C_T)} \sum_{c_1,\ldots,c_{t-1}} P(O_l,\ldots,O_T \mid C_1,\ldots,C_T)P(C_1,\ldots,C_T)$$

Now, using the fact that the random variables $O_1,\ldots,O_T$ are independent, given $C_1,\ldots,C_T$, and the distribution of any O_t only depends on C_t, we get

$$= \frac{1}{P(C_t,\ldots,C_T)} \sum_{c_1,\ldots,c_{t-1}} P(O_l \mid C_l)\ldots P(O_T \mid C_T)P(C_1,\ldots,C_T)$$

$$= \frac{1}{P(C_t,\ldots,C_T)} P(O_l \mid C_l)\ldots P(O_T \mid C_T) \sum_{c_1,\ldots,c_{t-1}} P(C_1,\ldots,C_T)$$

$$= \frac{1}{P(C_t,\ldots,C_T)} P(O_l \mid C_l)\ldots P(O_T \mid C_T)[P(C_t,\ldots,C_T)]$$

$$= P(O_l \mid C_l)\ldots P(O_T \mid C_T)$$

$$= P(O_l,\ldots,O_T \mid C_l,\ldots,C_T)$$

which is the RHS as required. □

Proposition 1.4 For $t = 1, 2, \ldots, T$, we have

$$P(O_1,\ldots,O_t \mid C_1,\ldots,C_T) = P(O_1,\ldots,O_t \mid C_1,\ldots,C_t)$$

Proof Using again the fact that, given $C_1,\ldots,C_T$, the random variables are independent and the distribution of each O_t only depends on C_t, we can write the LHS as follows:

$$P(O_1,\ldots,O_t \mid C_1,\ldots,C_T) = P(O_1 \mid C_1)\ldots P(O_t \mid C_t)$$

$$= P(O_1,\ldots,O_t \mid C_1,\ldots,C_t)$$

which is the RHS as required. □

Proposition 1.5 For $t = 1, 2, \ldots, T-1$, we have

$$P(O_{t+1},\ldots,O_T \mid C_1,\ldots,C_t) = P(O_{t+1},\ldots,O_T \mid C_t)$$

Proof We begin by writing the LHS as follows:

$$P(O_{t+1},\ldots,O_T \mid C_1,\ldots,C_t)$$

$$= \frac{1}{P(C_1,\ldots,C_t)} \sum_{c_{t+1},\ldots,c_T} P(C_1,\ldots,C_T)P(O_{t+1},\ldots,O_T \mid C_1,\ldots,C_T)$$

Applying **Proposition 1.3** we get

$$= \frac{1}{P(C_1,\ldots,C_t)} \sum_{c_{t+1},\ldots,c_T} P(C_1,\ldots,C_T)P(O_{t+1},\ldots,O_T \mid C_t,\ldots,C_T)$$

$$= \sum_{c_{t+1},\ldots,c_T} P(C_{t+1},\ldots,C_T \mid C_1,\ldots,C_t)P(O_{t+1},\ldots,O_T \mid C_t,\ldots,C_T)$$

Using the Markov property of $\{C_t\}$ we obtain

$$= \sum_{c_{t+1},\ldots,c_T} P(C_{t+1},\ldots,C_T \mid C_t) P(O_{t+1},\ldots,O_T \mid C_t,\ldots,C_T)$$

$$= \sum_{c_{t+1},\ldots,c_T} \frac{P(C_t,C_{t+1},\ldots,C_T)}{P(C_t)} \frac{P(O_{t+1},\ldots,O_T,C_t,\ldots,C_T)}{P(C_t,\ldots,C_T)}$$

$$= \sum_{c_{t+1},\ldots,c_T} \frac{P(O_{t+1},\ldots,O_T,C_t,\ldots,C_T)}{P(C_t)}$$

$$= \frac{P(O_{t+1},\ldots,O_T,C_t)}{P(C_t)}$$

$$= P(O_{t+1},\ldots,O_T \mid C_t)$$

which is the RHS as required. □

We shall use the propositions above to prove some very useful properties of the stochastic process O_t. These are listed below in **Proposition 1.6**:

Proposition 1.6

(i) For $t = 1, 2, \ldots, T$, we have

$$P(O_1,\ldots,O_T \mid C_t) = P(O_1 \ldots, O_t \mid C_t) P(O_{t+1},\ldots,O_T \mid C_t)$$

(ii) For $t = 1, 2, \ldots, T$, we have

$$P(O_t,\ldots,O_T \mid C_t) = P(O_t \mid C_t) P(O_{t+1},\ldots,O_T \mid C_t)$$

(iii) For $t = 1, 2, \ldots, T-1$, we have

$$P(O_1,\ldots,O_T \mid C_t, C_{t+1}) = P(O_1 \ldots, O_t \mid C_t) P(O_{t+1},\ldots,O_T \mid C_{t+1})$$

(iv) For all integers t and l such that $1 \le t \le l \le T$

$$P(O_l,\ldots,O_T \mid C_t,\ldots,C_l) = P(O_l,\ldots,O_T \mid C_l)$$

Proof

(i) We use the mutual independence of $O_1,\ldots,O_T$, given $C_1,\ldots,C_T$, to write the LHS as:

$P(O_1,\ldots,O_T \mid C_t)$

$$= \frac{1}{P(C_t)} \sum_{c_1,\ldots,c_{t-1}} \sum_{c_{t+1},\ldots,c_T} P(C_1,\ldots,C_T) P(O_1,\ldots,O_t \mid C_1,\ldots,C_T) \times P(O_{t+1},\ldots,O_T \mid C_1,\ldots,C_T)$$

Using the formula for conditional probability, we get

$$= \frac{1}{P(C_t)} \sum_{c_1,\ldots,c_{t-1}} \sum_{c_{t+1},\ldots,c_T} P(O_1,\ldots,O_t,C_1,\ldots,C_T) \times P(O_{t+1},\ldots,O_T \mid C_1,\ldots,C_T)$$

Using **Proposition 1.3**, we obtain

$$= \frac{1}{P(C_t)} \sum_{c_1,\ldots,c_{t-1}} \sum_{c_{t+1},\ldots,c_T} P(O_1, \ldots, O_t, C_1, \ldots, C_T) \times P(O_{t+1}, \ldots, O_T \mid C_t, \ldots, C_T)$$

Summing this expression over the sum that spans $\{c_{t+1}, \ldots, c_T\}$, we get

$$= \frac{1}{P(C_t)} \sum_{c_1,\ldots,c_{t-1}} P(O_1, \ldots, O_t, C_1, \ldots, C_t) P(O_{t+1}, \ldots, O_T \mid C_t)$$

Next, we evaluate the expression in the sum spanning $\{c_1, \ldots, c_{t-1}\}$ and we get

$$= \frac{1}{P(C_t)} P(O_1, \ldots, O_t, C_t) P(O_{t+1}, \ldots, O_T \mid C_t)$$

$$= P(O_1, \ldots, O_t \mid C_t) P(O_{t+1}, \ldots, O_T \mid C_t)$$

which is the RHS as required. □

(ii) We will sum the resulting expression of **Proposition 1.6 (i)** with respect to $\{o_1, \ldots, o_{t-1}\}$ and the LHS becomes:

$P(O_t, \ldots, O_T \mid C_t)$

$$= \sum_{o_1,\ldots,o_{t-1}} P(O_1, \ldots, O_t \mid C_t) P(O_{t+1}, \ldots, O_T \mid C_t)$$

$$= P(O_t \mid C_t) P(O_{t+1}, \ldots, O_T \mid C_t)$$

which is the RHS as required. □

(iii) We write the LHS as

$P(O_1, \ldots, O_T \mid C_t, C_{t+1})$

$$= \frac{1}{P(C_t, C_{t+1})} \sum_{c_1,\ldots,c_{t-1}} \sum_{c_{t+2},\ldots,c_T} P(C_1, \ldots, C_T) P(O_1, \ldots, O_t \mid C_1, \ldots, C_T) \times P(O_{t+1}, \ldots, O_T \mid C_1, \ldots, C_T)$$

Using **Proposition 1.3** and the formula for conditional probability, we get

$$= \frac{1}{P(C_t, C_{t+1})} \sum_{c_1,\ldots,c_{t-1}} \sum_{c_{t+2},\ldots,c_T} P(O_1, \ldots, O_t, C_1, \ldots, C_T) \times P(O_{t+1}, \ldots, O_T \mid C_{t+1}, \ldots, C_T)$$

Summing over the second sum (spanning $\{c_{t+2}, \ldots, c_T\}$), we obtain

$$= \frac{1}{P(C_t, C_{t+1})} \sum_{c_1,\ldots,c_{t-1}} P(O_1, \ldots, O_t, C_1, \ldots, C_{t+1}) P(O_{t+1}, \ldots, O_T \mid C_{t+1})$$

Summing over the remaining sum (spanning $\{c_1, \ldots, c_{t-1}\}$), we get

$$= \frac{1}{P(C_t, C_{t+1})} P(O_1, \ldots, O_t, C_t, C_{t+1}) P(O_{t+1}, \ldots, O_T \mid C_{t+1})$$

$$= P(O_1, \ldots, O_t \mid C_t, C_{t+1}) P(O_{t+1}, \ldots, O_T \mid C_{t+1})$$

Finally, using the property of conditional independence where the distribution of O_t depends only on C_t, we can evaluate our expression to

$$= P(O_1, \ldots, O_t \mid C_t) P(O_{t+1}, \ldots, O_T \mid C_{t+1})$$

which is the RHS as required. □

(iv) We write the LHS as

$P(O_l, \ldots, O_T \mid C_t, \ldots, C_l)$

$$= \frac{1}{P(C_t, \ldots, C_l)} \sum_{c_{l+1}, \ldots, c_T} \sum_{c_1, \ldots, c_{t-1}} P(O_l, \ldots, O_T \mid C_1, \ldots, C_T) P(C_1, \ldots, C_T)$$

Using **Proposition 1.3** we change our expression to

$$= \frac{1}{P(C_t, \ldots, C_l)} \sum_{c_{l+1}, \ldots, c_T} \sum_{c_1, \ldots, c_{t-1}} P(O_l, \ldots, O_T \mid C_l, \ldots, C_T) P(C_1, \ldots, C_T)$$

Evaluating the summation which spans $\{c_1, \ldots, c_{t-1}\}$, we get

$$= \frac{1}{P(C_t, \ldots, C_l)} \sum_{c_{l+1}, \ldots, c_T} P(O_l, \ldots, O_T \mid C_l, \ldots, C_T) P(C_t, \ldots, C_T)$$

Next, we bring in the denominator inside the summation:

$$= \sum_{c_{l+1}, \ldots, c_T} P(O_l, \ldots, O_T \mid C_l, \ldots, C_T) \frac{P(C_t, \ldots, C_l, C_{l+1}, \ldots, C_T)}{P(C_t, \ldots, C_l)}$$

Then, we use the formula for conditional probability and obtain

$$= \sum_{c_{l+1}, \ldots, c_T} P(O_l, \ldots, O_T \mid C_l, \ldots, C_T) P(C_{l+1}, \ldots, C_T \mid C_t, \ldots, C_l)$$

By the Markov property of $\{C_t\}$ we have

$$= \sum_{c_{l+1}, \ldots, c_T} P(O_l, \ldots, O_T \mid C_l, \ldots, C_T) P(C_{l+1}, \ldots, C_T \mid C_l)$$

Again applying our well-known formulas for conditional probability our expression becomes

$$= \sum_{c_{l+1}, \ldots, c_T} \frac{P(O_l, \ldots, O_T, C_l, \ldots, C_T)}{P(C_l, \ldots, C_T)} \frac{P(C_{l+1}, \ldots, C_T, C_l)}{P(C_l)}$$

which cancels out to give

$$= \sum_{c_{l+1}, \ldots, c_T} \frac{P(O_l, \ldots, O_T, C_l, \ldots, C_T)}{P(C_l)}$$

Summing give us

$$= \frac{P(O_l,\ldots,O_T,C_l)}{P(C_l)}$$

$$= P(O_l,\ldots,O_T \mid C_l)$$

which is the RHS as required. □

These properties have provided us with the mathematical background to carry out our analysis of the three fundamental problems surrounding HMMs. But before that, we define the joint probability and the likelihood functions.

The probability of joint processes $C_0, O_0, C_1, O_1, \ldots, C_n, O_n$, is vital to help us calculate the probabilities associated with HMMs. This joint probability function can be written as

$$\begin{aligned} J_{v,n}(c_0,o_0,\ldots,c_n,o_n) &= P(C_0=c_0,O_0=o_0,\ldots,C_n=c_n,O_n=o_n) \\ &= P(C_0=c_0,O_0=o_0) \quad \times \\ &\qquad P(C_1=c_1,O_1=o_1 \mid C_0=c_0)\ldots \\ &\qquad \ldots P(C_n=c_n,O_n=o_n \mid C_{n-1}=c_{n-1}) \\ &= v_{c_0} g_{c_0,o_0} \textstyle\prod_{i=1}^{n} q_{c_{i-1},c_i} g_{c_i,o_i} \end{aligned}$$

where the initial distribution of the chain $v_{c_0} = P(C_0 = c_0)$

In fact, this is the full likelihood function which belongs to all random variables (i.e. observed and unobserved). As defined in section 3.3 of [1], the likelihood function $L_{v,n}(o_0, o_1, \ldots, o_n)$ of observations $o_0, o_1, \ldots, o_n$ can be written as follows:

$$L_{v,n}(o_0,o_1,\ldots,o_n) = P(O_0=o_0,O_1=o_1,\ldots,O_n=o_n)$$

Therefore, we can write

$$\begin{aligned} L_{v,n}(o_0,o_1,\ldots,o_n) &= \textstyle\sum_{c_0,\ldots,c_n} J_{v,n}(c_0,o_0,\ldots,c_n,o_n) \\ &= \textstyle\sum_{c_0,\ldots,c_n} v_{c_0} g_{c_0,o_0} \prod_{i=1}^{n} q_{c_{i-1},c_i} g_{c_i,o_i} \end{aligned}$$

For the rest of this report, we shall use a different notation to represent the likelihood function, as explained in the forward-backward algorithm. Essentially, we will write $L_{v,n}(o_0, o_1, \ldots, o_n)$ as $P(O \mid \lambda)$ (where O is the observation set $o_0, o_1, \ldots, o_n$ and λ is the model) to better understand the terms of the algorithms used later on.

We have seen mathematically how to treat events where a certain observation occurs (e.g. $O_t = o_t$, for some t) given a state in the Markov chain. We will now illustrate how the Markov chain and its different hidden states interact with the possible observations:

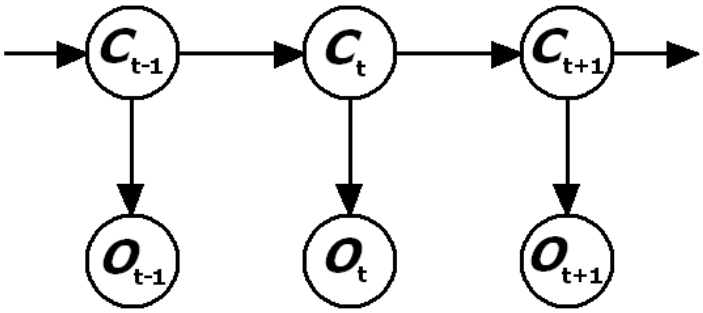

Figure 2.1: A directed acyclic graph (DAG) showing conditional independence relations for a HMM.

Above, 2.1 shows a directed acyclic graph specifying conditional independence relations for a HMM. At the top of the diagram, we have the Markov chain with its hidden states C_i. Each node in the Markov chain is conditionally independent from its non-descendants given its parents. For example, given $C_1, C_2, \ldots, C_{t-1}, C_t$, we have that C_{t+1} is independent of $C_1, C_2, \ldots, C_{t-1}$ (which is the first Markov property). At the bottom of the diagram, we have the observations O_i which are linked to C_i of the Markov chain. We are only able to observe the O_i as they are not hidden to us.

When constructing a HMM, there are three main problems that need to be addressed. First, given the parameters of the model, we compute the probabilities of a particular sequence of observations, which can be solved by the *Forward-Backward algorithm*. This helps us compute the probability of the HMM generating a particular sequence (i.e. the probability of that sequence under the model). Second, given a sequence of observations, we aim to find the most likely set of model parameters. This may be solved by statistical inference through the *Baum-Welch Algorithm*, which uses the Forward-Backward algorithm. Lastly, we need to find the path of hidden states that is most likely to generate a sequence of observations. This is solved using a posteriori statistical inference in the *Viterbi Algorithm*. We shall go through these three problems after we introduce a simple example involving HMMs.

2.2 An example involving hidden Markov models

Suppose we wanted to determine the average annual rainfall of a particular region in the UK over a period of time many years ago. Getting inspiration from a similar example used in [10], we can make the problem simpler by looking at two measurements, wet (W) and dry (D). Let us now assume that initially, in the first year of analysis, the probability we had a wet year was 0.55 and the probability we had a dry year was 0.45. A very simple matrix will show this

$$\begin{matrix} W & D \end{matrix}$$
$$\begin{pmatrix} 0.55 & 0.45 \end{pmatrix} \tag{2.1}$$

To help us progress down the time period, we also have evidence that the probability of two consecutive wet years is 0.8 and the probability of two consecutive dry years is 0.7. Then, we can summarise this information into the following matrix:

$$\begin{matrix} & W & D \\ W & 0.8 & 0.2 \\ D & 0.3 & 0.7 \end{matrix} \tag{2.2}$$

Notice how the matrix above has elements which are probabilities and the elements in each row sum up to 1. This type of matrix is called *row stochastic*.

Now, let us suppose that there is a relationship between the height of grass and the type of rainfall in those particular regions. For the purpose of this experiment, we can assume there are three types of grass heights: short (S), medium (M) and tall (T). Therefore, the correlation between rainfall and grass height can be summarised by

$$\begin{matrix} & S & M & T \\ W & 0.2 & 0.3 & 0.5 \\ D & 0.1 & 0.6 & 0.3 \end{matrix} \tag{2.3}$$

We denote the average annual rainfall as the *state*, which can be W or D. Notice that moving from one state to the next depends solely on the probabilities in 2.2 and on the current state. Therefore, the state transition is in fact a *Markov process*. Looking at 2.3, we cannot directly observe our average annual rainfall states, so we label (W and D) the "hidden" states. However, we can observe the height of grass, which provides us with selective information of the rainfall, indirectly. Hence, we denote this type of system as a *hidden Markov model* (HMM).

We continue to use the observed information to learn more about our Markov process, in an efficient manner. We now define some important notation for the system we have been describing. Firstly, 2.1 will be known as the *initial state distribution*, given by

$$\pi = \begin{pmatrix} 0.55 & 0.45 \end{pmatrix} \tag{2.4}$$

Secondly, 2.2 will be known as the *state transition matrix*

$$A = \begin{pmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{pmatrix} \tag{2.5}$$

Thirdly, 2.3 will be known as the *observation matrix*

$$B = \begin{pmatrix} 0.2 & 0.3 & 0.5 \\ 0.1 & 0.6 & 0.3 \end{pmatrix} \tag{2.6}$$

Let us now consider a five-year period which we can analyse. The sequence of grass heights are as follows: *S,M,T,M,T* and we let 0 represent S, 1 represent M and 2 represent T. The sequence we have observed over the five years is now

$$O = (0, 1, 2, 1, 2) \tag{2.7}$$

From the information given in equation 2.7, we can use HMMs to find the most likely state sequence of the Markov process. By "most likely", we mean the state sequence which maximises the expected number of correct states.

Let us now introduce some notation, which will help us find the probability of a possible state sequence. In the next section, we will use this notation to formulate the solutions to our three fundamental problems regarding HMMs. We define our terms as follows:

$O = (O_0, O_1, \ldots, O_{T-1})$ = observation sequence
T = length of the observation sequence
$Q = q_0, q_1, \ldots, q_{N-1}$ = states of the Markov chain
N = number of states in the model
$V = v_0, v_1, \ldots, v_{M-1}$ = set of possible observations
M = number of observation symbols
π = initial state distribution
A = state transition probabilities
B = observation probability matrix

Note that $O_t \in V$ for $t = 0, 1, \ldots, T-1$. Also, we elaborate the definition of π as

$$\pi = \{\pi_i\}, \pi_i = P(\text{state } q_i \text{ at } t = 0)$$

We also note that A is an $N \times N$, row stochastic matrix such that

$$A = a_{ij} = P(\text{state } q_j \text{ at } t + 1 \mid \text{state } q_i \text{ at } t)$$

and a_{ij} are independent of t.

Similarly, B is an $N \times M$ row stochastic matrix such that

$$B = b_j(k) = P(\text{observation } v_k \text{ at } t \mid \text{state } q_j \text{ at } t)$$

and $b_j(k)$ are independent of t.

Looking back at our DAG in Figure 2.1, we can use our newly defined matrices (A and B) to explain the transitions in the graph. The state transition matrix A helps us progress through the Markov chain from state C_t to state C_{t+1}. The observation matrix B gives us the observation O_t from its corresponding state C_t. Therefore, the matrices A and B can create our entire graph in 2.1 provided we are given an initial state distribution.

Now, we have gathered sufficient notation to define a HMM by $\lambda = (A, B, \pi)$. We shall use this definition to solve the three fundamental problems regarding HMMs (see next section). But first, suppose we were given the following generic information:

$X = (x_0, x_1, x_2, x_3, x_4)$ = state sequence of length five
$O = (O_0, O_1, O_2, O_3, O_4)$ = corresponding observations
π_{x_0} = probability of starting in state x_0
$b_{x_t}(O_t)$ = probability of observing O_t from state x_t
$a_{x_t, x_{t+1}}$ = probability of a transition from state x_t to state x_{t+1}

Therefore, what would the probability be of the given state sequence (i.e. $P(X)$)? To answer this question, we must start small to begin with. The probability of generating the initial state x_0 is given by

$$\pi_{x_0} b_{x_0}(O_0).$$

Then, the probability of generating the state sequence (x_0, x_1) is given by

$$\pi_{x_0} b_{x_0}(O_0) a_{x_0,x_1} b_{x_1}(O_1).$$

Continuing in this manner, the probability of the entire state sequence X is

$$P(X) = \pi_{x_0} b_{x_0}(O_0) a_{x_0,x_1} b_{x_1}(O_1) a_{x_1,x_2} b_{x_2}(O_2) a_{x_2,x_3} b_{x_3}(O_3) a_{x_3,x_4} b_{x_4}(O_4) \tag{2.8}$$

Let us now return to our rainfall example where we analysed a five year period, using the sequence from 2.7. We can formalise our specification for this problem using the new notation:

$O = (0, 1, 2, 1, 2)$
$T = 5$
$Q = W, D$
$N = 2$
$V = 0, 1, 2$
$M = 3$
π = matrix from 2.1
A = matrix from 2.2
B = matrix from 2.3

Furthermore, given the observation sequence $O = (0, 1, 2, 1, 2)$, we can use 2.8 to find

$$\begin{aligned} P(WDWDW) &= \pi_W b_W(0) a_{W,D} b_D(1) a_{D,W} b_W(2) a_{W,D} b_D(1) a_{D,W} b_W(2) \\ &= (0.55)(0.2)(0.2)(0.6)(0.3)(0.5)(0.2)(0.6)(0.3)(0.5) \\ &= 0.00003564 \end{aligned}$$

Similarly, we can calculate the probabilitiy of each of the possible state sequences

of length five ($WWWWW$, $WWWWD$, $WWWDW$, etc.). Note that there will be a total of $2^5 = 32$ possible state sequences of length five because there are only two choices for each state (W or D). We shall now move on to the three fundamental problems of HMMs and attempt to solve them generically.

2.3 Solution of the three fundamental problems

2.3.1 The first problem

The first fundamental problem can be described as follows: Suppose that we have a sequence of observations $O = (O_0, O_1, \ldots, O_{T-1})$ and the model $\lambda = (A, B, \pi)$. Our aim is to find $P(O \mid \lambda)$, the probability of the given sequence of observations given the model. In doing this, we want to determine the likelihood of the observed sequence O.

In a similar fashion to the solution in [5], we shall use the "forward" part of the *Forward-Backward algorithm*, which is called the α-pass. We can define $\alpha_t(i)$ as the probability of the observation sequence up to time t and of state q_i at time t, given our model λ. The mathematical notation is

$$\alpha_t(i) = P(O_0, O_1, \ldots, O_t, s_t = q_i \mid \lambda) \tag{2.9}$$

where $i = 0, 1, \ldots, N-1$ and $t = 0, 1, \ldots, T-1$

The solution of $\alpha_t(i)$ is an inductive one and proceeds as follows:

1. to initiate the forward probabilities, for $i = 0, 1, \ldots, N-1$, we have

   $$\alpha_0(i) = \pi_i b_i(O_0).$$

2. then, for $i = 0, 1, \ldots, N-1$ and $t = 0, 1, \ldots, T-2$ we have

   $$\alpha_{t+1}(i) = [\textstyle\sum_{j=0}^{N-1} \alpha_t(j) a_{ji}] b_i(O_{t+1})$$

   where $\alpha_t(j)a_{ji}$ is the probability of the joint event that $O_0, O_1, \ldots O_t$ are observed and there is a transition from state q_j at time t to state q_i at time $t+1$.

3. it follows that

   $$P(O \mid \lambda) = \textstyle\sum_{i=0}^{N-1} \alpha_{T-1}(i)$$

   where we use 2.9 to get $\alpha_{T-1}(i) = P(O_0, O_1, \ldots, O_{T-1}, s_{T-1} = q_i \mid \lambda)$

Similarly, we can define the backward variable, $\beta_t(i)$ as the probability of the observation sequence from time $t+1$ to the end, given state q_i at time t and the model λ. The mathematical notation is

$$\beta_t(i) = P(O_{t+1}, O_{t+2}, \ldots O_{T-1} \mid s_t = q_i, \lambda) \tag{2.10}$$

The solution of $\beta_t(i)$ is inductive and is given by:

1. to start with, for $i = 0, 1, \ldots, N-1$, we have

$$\beta_{T-1}(i) = 1.$$

2. then, for $i = 0, 1, \ldots, N-1$ and $t = T-2, T-3, \ldots, 0$ we have

$$\beta_t(i) = \sum_{j=0}^{N-1} a_{ij} b_j(O_{t+1}) \beta_{t+1}(j).$$

where we note that the observation O_{t+1} can be generated from any state q_j.

2.3.2 The second problem

Our second problem was, given the model $\lambda = (A, B, \pi)$ and the observation sequence $O = (O_0, O_1, \ldots, O_{T-1})$, attempt to maximise $P(O \mid \lambda)$ by adjusting the parameters A, B, π. This problem can be solved using the *Baum-Welch algorithm*, which is an iterative process.

Firstly, we define the probability of a path being in state q_i at time t and making a transition to state q_j at time $t+1$, given O and λ, as

$$\xi_t(i, j) = P(s_t = q_i, s_{t+1} = q_j \mid O, \lambda)$$

Now, we describe the different parts of computing $\xi_t(i, j)$. Firstly, the observations $O_0, O_1, \ldots O_t$ finishing in state q_i at time t are covered by $\alpha_t(i)$. Then, the transition from q_i to q_j, where O_{t+1} was observed at time $t+1$, is represented by the term $a_{ij} b_j(O_{t+1})$. Finally, the remaining observations $O_{t+2}, O_{t+3} \ldots O_{T-1}$ beginning in state q_j at time $t+1$ are covered by β_{t+1}. Putting those together, and dividing by a normalizing term ($P(O \mid \lambda)$) we have

$$\xi_t(i, j) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{P(O \mid \lambda)} \tag{2.11}$$

Let us now sum the terms in 2.11 over j and notice that we get the probability of being in state q_i at time t, given the observation sequence O and model λ. We define this probability as follows:

$$\gamma_t(i) = P(s_t = q_i \mid O, \lambda)$$

$$= \sum_{j=0}^{N-1} \xi_t(i, j)$$

If we sum the $\gamma_t(i)$ over time t up to $T-1$, then we get the number of times we expect to visit state q_i. This is the same, in fact, as the expected number of transitions made from q_i, summing up to $T-2$ this time because we need to save one time index for the transition. This is summarised below:

$$\sum_{t=0}^{T-1} \gamma_t(i) = \text{Expected number of times state } q_i \text{ is visited.}$$

$$\sum_{t=0}^{T-2} \gamma_t(i) = \text{Expected number of transitions made from state } q_i.$$

In a similar fashion, we sum $\xi_t(i, j)$ over t and the results are summarised as follows:

$\sum_{t=0}^{T-1} \xi_t(i,j)$ = Expected number of times state q_i, then state q_j are visited.

$\sum_{t=0}^{T-2} \xi_t(i,j)$ = Expected number of transitions made from state q_i to state q_j.

Using these terms, we now define the re-estimation formulas for our HMM parameters (π, A, B):

1. Initially at $t = 0$, we have

   $$\pi_i' = \gamma_0(i)$$

   where $i = 0, 1, \ldots, N-1$

2. For A, we have

   $$a_{ij}' = \frac{\sum_{t=0}^{T-2} \xi_t(i,j)}{\sum_{j=0}^{N-1} \sum_{t=0}^{T-2} \xi_t(i,j)}$$

   where we notice that this formula is just the expected number of transitions from q_i to q_j divided by the expected number of transitions coming from q_i.

3. For B we have

   $$b_j(k)' = \frac{\sum_{t=0, O_t=k}^{T-1} \gamma_t(j)}{\sum_{t=0}^{T-1} \gamma_t(j)}$$

   where this is just the expected number of times state q_j is visited and k is observed divided by the number of times q_j is visited.

Using these re-estimation formulas on a given model, λ, we can re-estimate our model and obtain:

$$\lambda' = (A', B', \pi')$$

where $A' = \{a_{ij}'\}$, $B' = \{b_j(k)'\}$ and $\pi' = \{\pi_i'\}$

We can use our new re-estimation model to check if $P(O \mid \lambda') > P(O \mid \lambda)$, which would mean we found a model λ' that is more likely to produce the observation sequence. We continue like this, replacing λ with λ' if it gives us a higher probability, until we have reached a limit. This limit could be the number of iterations, which would tell us when to stop the re-estimation, or we find that the probability of observing O cannot be improved beyond some threshold.

2.3.3 The third problem

The third fundamental problem we had to solve was as follows: Suppose that we have a sequence of observations $O = (O_0, O_1, \ldots, O_{T-1})$ and the model $\lambda = (A, B, \pi)$. Our aim is to find an optimal state sequence for the underlying Markov chain and thus, reveal the hidden part of the HMM λ. In other words, the criterion is to find the best sequence of states (i.e. $S = (S_0, S_1, \ldots, S_{T-1})$) such that

$$S^* = argmax_S P(S \mid O, \lambda)$$

As $P(O \mid \lambda)$ is independent of S, we have

$$= argmax_S P(S \mid O, \lambda) P(O \mid \lambda)$$

$$= argmax_S P(S, O \mid \lambda)$$

The *Viterbi algorithm* will give us this optimal state sequence S^*. At each step (time t), the Viterbi algorithm allows S^* to retain all optimal paths that finish at the N states. At the next step (time $t + 1$), the N optimal paths will be updated and S^* continues to grow in this manner.

Let $S_t^*(i)$ be the optimal path ending in state S_i for the observations $O_0, O_1, \ldots, O_t$. Then we can define $\delta_t(i) = P(O_0, O_1, \ldots, O_t, S_t^*(i) \mid \lambda)$, which is the probability of generating observations $O_0, O_1, \ldots, O_t$ from path $S_t^*(i)$. Finally, we use an array $\psi_t(i)$ to keep track of each t and i that has maximized the last $\delta_t(i)$.

The steps of the Viterbi algorithm are described below:

1. We initialise the following variables

$$\delta_0(i) = \pi_i b_i(O_0) \text{ for } i = 0, 1, \ldots N - 1$$

$$\psi_0(i) = 0$$

2. We recurse for $j = 0, 1, \ldots, N - 1$ and $t = 1, 2, \ldots, T - 1$ on the variables as follows

$$\delta_t(j) = \max_{0 \leq i \leq N-1} [\delta_{t-1}(i) a_{ij}] b_j(O_t)$$

$$\psi_t(j) = argmax_{0 \leq i \leq N-1} [\delta_{t-1}(i) a_{ij}]$$

3. We terminate with

$$P^* = \max_{0 \leq i \leq N-1} [\delta_T(i)]$$

$$S_T = argmax_{0 \leq i \leq N-1} [\delta_T(i)]$$

4. We backtrack through the state sequence for $t = T - 2, T - 3, \ldots, 0$ as such:

$$S_t^* = \psi_{t+1}(i_{t+1}^*)$$

2.4 Underflow

The solutions to the three fundamental problems, seen earlier, could be implemented using any double point precision language (e.g. C) and produce convergent results for a small sequence of observations. However, as we increase the number of points in the sequence, the Baum-welch algorithm can succumb to *underflow*. This means that as our sequences are larger, the probabilistic values in the algorithm get increasingly small and after enough iterations become almost zero.

We will discuss how this problem can be solved for the Baum-Welch algorithm, and then do the same for the Viterbi algorithm.

2.4.1 Normalized Baum-Welch algorithm

We will show how to normalize the αs and the βs of the Forward-Backward algorithm and solve this issue of underflow. Note, the normalizing procedure which will be used in this section is adapted from [13]. Firstly, we normalize $\hat{\alpha}_t(j)$ so that all the terms (from 0 to $N-1$) sum to 1:

$$\sum_{i=0}^{N-1} \hat{\alpha}_t(i) = 1$$

In other words, we have

$$\begin{aligned}
\hat{\alpha}_t(i) &= \frac{\alpha_t(i)}{\sum_{i=0}^{N-1} \alpha_t(i)} \\
&= \frac{P(O_0, O_1, \ldots, O_t, s_t = q_i | \lambda)}{P(O_0, O_1, \ldots, O_t | \lambda)} \\
&= \frac{P(O_0, O_1, \ldots, O_t, s_t = q_i, \lambda)/P(\lambda)}{P(O_0, O_1, \ldots, O_t, \lambda)/P(\lambda)}
\end{aligned}$$

Cancelling out the $P(\lambda)$ in the fraction gives us

$$\begin{aligned}
&= \frac{P(O_0, O_1, \ldots, O_t, s_t = q_i, \lambda)}{P(O_0, O_1, \ldots, O_t, \lambda)} \\
&= P(s_t = q_i \mid O_0, O_1, \ldots, O_t, \lambda)
\end{aligned}$$

Hence, the solution of $\hat{\alpha}_t(i)$ is the following:

1. to initiate the forward probabilities, for $i = 0, 1, \ldots, N-1$, we have

$$\hat{\alpha}_0(i) = \frac{\pi_i b_i(O_0)}{\sum_{j=0}^{N-1} \pi_j b_j(O_0)}.$$

2. then, for $j = 0, 1, \ldots, N-1$ and $t = 0, 1, \ldots, T-2$ we have

$$\hat{\alpha}_{t+1}(i) = \frac{[\sum_{j=0}^{N-1} \hat{\alpha}_t(j) a_{ji}] b_i(O_{t+1})}{\sum_{k=0}^{N-1} [\sum_{j=0}^{N-1} \hat{\alpha}_t(j) a_{jk}] b_k(O_{t+1})}$$

For the $\hat{\beta}_t(i)$s, we use the same normalizers as those for the αs. However, unlike the αs, the βs do not sum to 1 at any time t. Thus, the solution of $\hat{\beta}_t(i)$ is given by:

1. to start with, for $i = 0, 1, \ldots, N-1$, we have

$$\hat{\beta}_{T-1}(i) = \beta_{T-1}(i) = 1.$$

2. then, for $i = 0, 1, \ldots, N-1$ and $t = T-2, T-3, \ldots, 0$ we have

$$\hat{\beta}_t(i) = \frac{\sum_{j=0}^{N-1} a_{ij} b_j(O_{t+1}) \hat{\beta}_{t+1}(j)}{\sum_{k=0}^{N-1} [\sum_{j=0}^{N-1} \hat{\alpha}_t(j) a_{jk}] b_k(O_{t+1})}.$$

Notice that we can write $\hat{\alpha}_t(i)\hat{\beta}_t(i)$ as:

$$\begin{aligned}\hat{\alpha}_t(i)\hat{\beta}_t(i) &= \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=0}^{N-1} \alpha_{T-1}(i)} \\ &= \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)}\end{aligned}$$

Therefore, the $\gamma_t(i)$s have the same formula because we can just divide by the normalizers without changing the fraction, shown as follows:

$$\begin{aligned}\gamma_t(i) &= \sum_{j=0}^{N-1} \xi_t(i, j) \\ &= \frac{\alpha_t(i)\beta_t(i)}{\sum_{j=0}^{N-1} \alpha_t(j)\beta_t(j)}\end{aligned}$$

Dividing the numerator and the denominator by $P(O \mid \lambda)$ we get

$$= \frac{\hat{\alpha}_t(i)\hat{\beta}_t(i)}{\sum_{j=0}^{N-1} \hat{\alpha}_t(j)\hat{\beta}_t(j)}$$

where throughout we have $t = 0, 1, \ldots, T-1$

However, the ξs are computed differently and are given by the following formula:

$$\xi_t(i, j) = \frac{\hat{\alpha}_t(i) a_{ij} b_j(O_{t+1}) \hat{\beta}_{t+1}(j)}{[\sum_{k=0}^{N-1} [\sum_{j=0}^{N-1} \hat{\alpha}_t(j) a_{jk}] b_k(O_{t+1})][\sum_{j=0}^{N-1} \hat{\alpha}_t(j)\hat{\beta}_t(j)]}$$

By definition of $\gamma_t(i)$, this gives us

$$= \frac{\gamma_t(i) a_{ij} b_j(O_{t+1}) \hat{\beta}_{t+1}(j)}{[\sum_{k=0}^{N-1} [\sum_{j=0}^{N-1} \hat{\alpha}_t(j) a_{jk}] b_k(O_{t+1})]\hat{\beta}_t(i)}$$

The re-estimation formulas for our HMM parameters (π, A, B) now use the new γs and ξs, but otherwise remain the same.

2.4.2 Logarithmic Viterbi algorithm

With the Viterbi algorithm, underflow can be avoided by using *logarithms*. As a consequence, we sum the logarithms of the terms instead of multiplying them using products. Note, that this could be used for the Baum-Welch algorithm. We could take logarithms of the α and β values, but computing the γ values would require dealing

with a sum of $\alpha_t(i)$, a sum not in the logarithm domain.

Therefore, taking logarithms of our terms, the enhanced Viterbi algorithm becomes:

1. We initialise the following variables

$$\delta_0(i) = log(\pi_i b_i(O_0)) \text{ for } i = 0, 1, \ldots N-1$$

$$\psi_0(i) = 0$$

2. We recurse for $j = 0, 1, \ldots, N-1$ and $t = 1, 2, \ldots, T-1$ on the variables as follows

$$\delta_t(j) = \max_{0 \leq i \leq N-1}[\delta_{t-1}(i) + log(a_{ij})] + log(b_j(O_t))$$

$$\psi_t(j) = argmax_{0 \leq i \leq N-1}[\delta_{t-1}(i) + log(a_{ij})]$$

3. We terminate with

$$P^* = \max_{0 \leq i \leq N-1}[\delta_T(i)]$$

$$S_T = argmax_{0 \leq i \leq N-1}[\delta_T(i)]$$

4. We backtrack through the state sequence for $t = T-2, T-3, \ldots, 0$ as such:

$$S_t^* = \psi_{t+1}(i_{t+1}^*)$$

2.5 Application of HMMs

HMMs were first used in the late 1960s in statistical papers by Leonard E. Baum for stastical inference of Markov chains (Baum and Petrie [7]) and also for statistical estimation of Markov process probability functions (Baum and Eagon [6]). As time went on, it became clear that HMMs could be used in a number of diverse fields ranging from Bioinformatics to Flash Memory. However, one of the first applications of HMMs was speech recoginition in the 1970s.

2.5.1 Application of HMMs to Speech Recognition

HMMs can be applied to Speech Recognition (SR) as they are an important part of statistically-based algorithms. They are computationally feasible to use and can be trained automatically. More importantly, HMMs are linked to SR because a speech signal is represented as a short-time stationary signal. Therefore, for stochastic processes, speech can be seen as a Markov model.

Rabiner published papers about using HMMs for Speech Recognition in the late 1980s, where in one such paper he adapted a HMM to build an isolated word recognizer [5]. In this paper, Rabiner and Juang use a fixed vocabulary of words, a training

set of fixed tokens and an independent testing set. Firstly, they built a HMM for each word in the vocabulary, and used token observations to estimate the parameters for each word. Secondly, for each unknown word in the test set, they calculate its probability given the word model. Finally, they find the word whose model probability is biggest. The calculations carried out were similar to those involved in solving the three fundamental solutions of HMMs. Therefore, either the Viterbi algorithm or the Forward-Backward algorithm can provide the required probability.

Another more recent application of HMMs to SR is where statistical-based SR software is used to model variations in speech with automatic learning procedures. Speech signals are viewed as short-time stationary signals meaning that speech can be seen as a model for stochastic processes. In fact, the HMM would output a sequence of n-dimensional real-valued vectors, which consist of cepstral coefficients (obtained from applying a Fourier transform on a sample of speech and then passing this sample through the cosine transform). So, the HMM will have a likelihood function for each vector and thus each word will have a different output distribution. An example is *Speaker Independent Urdu Speech Recognition using HMMs* [3], where an acoustic model was used to interpret Urdu sounds and then evaluated using Word Error Rate (WER). For example, a WER of 5% meant that there was an error of one out of every twenty words.

2.5.2 Application of HMMs to Flash Memory

Attempts have been made to sort workloads at a Flash storage system at different temporal scales. Researchers have aimed to create portable benchmarks in terms of Markov arrival processes (MAPs). Such a paper which looked at MAPs was *Storage Workload Modelling by Hidden Markov Models: Application to FLASH Memory* [1] and looked at Markov modulated Poisson processes (MMPPs) in particular. These benchmarks can correctly represent the correlation and burstiness in multi-application workloads and therfore optimize storage and access time. When analysed at the millisecond timescale, an aim is to represent the load at the Flash chip level (Figure 2.2) as input to a fluid model which reproduces transaction response time distributions [12].

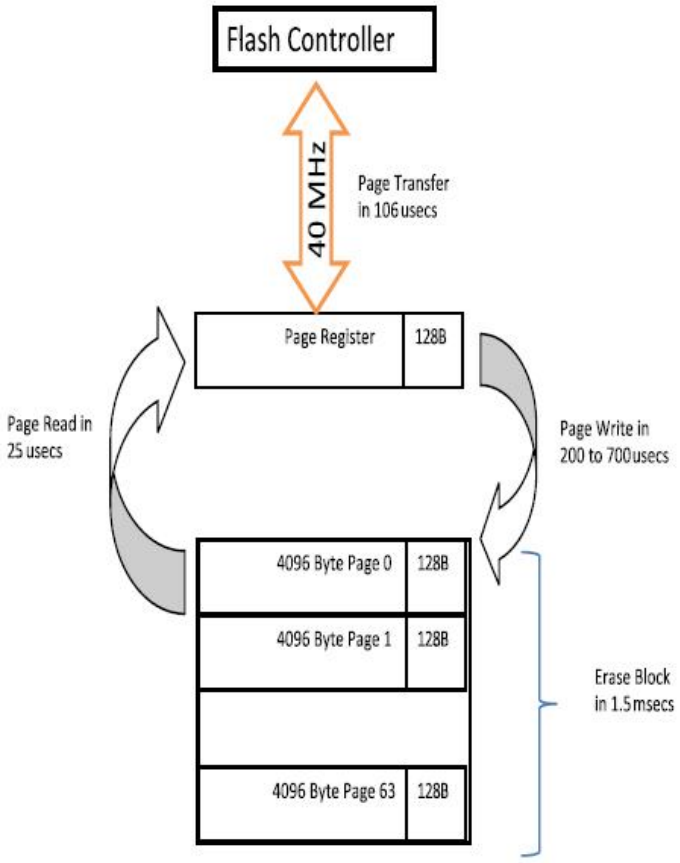

Figure 2.2: Flash chip and controller. [1]

We can see from Figure 2.2 that read and write rates change over time (even with constant input workload rates). This supports the idea that real system workloads contain sudden jumps in arrival rates.

In fact, a Markov chain can represent workload dynamics, and it's important to estimate its parameters and obtain the characteristics of the studied traces. HMMs can model these traces (or time series) mainly when it's known that the trace is subject to switching between modes (represented by the hidden states of the HMM). This is only a short description of the potential HMMs have for Flash Memory applications. We shall elaborate on this in further detail in chapter (4) of this report.

2.5.3 Application of HMMs to Biology

In the late 1980s, HMMs were used to analyse biological sequences and consequently their uses have stretched to many applications in Bioinformatics. One of the main uses of HMMs in Bionformatics has been the prediction of protein coding regions in genome sequences. Another use of HMMs has been to model common groups of protein sequences (see [18] for further information), an important topic in computational biology.

HMMs have been also utilised to locate genes given an uncharacterized DNA se-

quence. The GENSCAN HMM ([19]) has been used for Eukaryotic gene finding and we give the model in the following diagram:

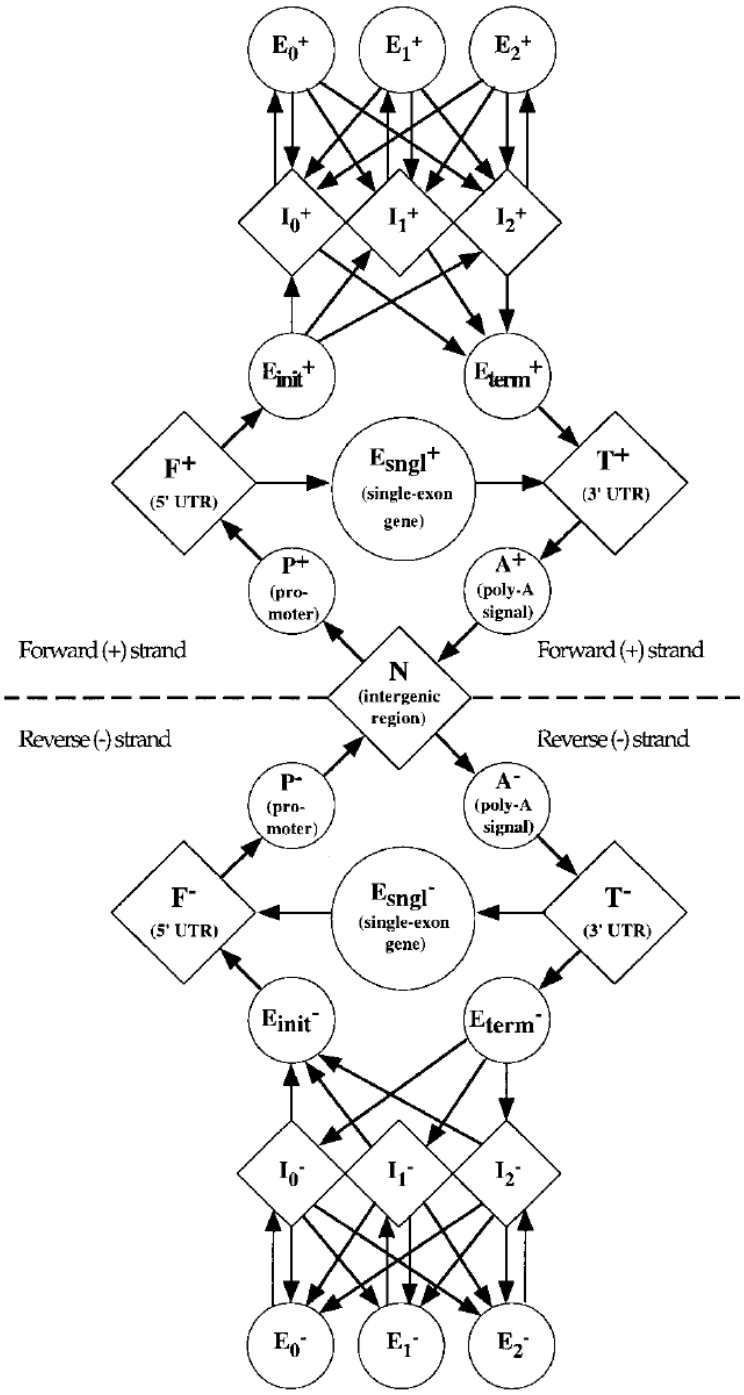

Figure 2.3: The general model of the structure of genomic sequences [19]. Note that the hidden states of the HMM are represented by circles and diamonds.

The GENSCAN HMM models the length distribution and sequence composition

for each sequence type. To find the most probable path through the model for the sequence, we use the Viterbi algorithm. Note that the path which is returned by Viterbi will contain the coordinates of the predicted genes. The accuracy of the GENSCAN HMM is tested using metrics such as SENSITIVITY and SPECIFICITY of the data examples used. These accuracy metrics are given below:

$$\text{SENSITIVITY} = \frac{TP}{TP+FN} \qquad \text{SPECIFICITY} = \frac{TN}{TN+FN}$$

where TP = True Positives, TN = True Negatives and FN = False Negatives.

These are just some possibilities of the uses of HMMs in Biology, which we have covered here. Possible extensions include using classifying proteins given an amino-acid sequence, modelling multiple sequences with pair HMMs, etc.

Chapter 3

Clustering Algorithms

In this section, we will discuss the purpose of clustering in general and also with respect to our Flash Memory and Hospital models. We will examine various clustering algorithms, summarising the advantages and disadvantages firstly for each type of algorithm and then secondly for cluster analysis in general. After this, we elaborate on the motivation behind choosing to implement our K-means clustering for both of our models. Finally, we discuss at the end of this chapter the possible ways that clustering can provide extensions to this project, mainly through further statistical data analysis.

3.1 What is Clustering?

Clustering can be defined as the process of partitioning data items into groups with common attributes. Essentially, clustering is an unsupervised learning problem which seeks to obtain a structure to some apparently randomly distributed set of data. It is deemed ”unsupervised” because we are unaware of the class labels nor can we be sure of the number of classes. One of the first goals of clustering is to ensure we have assigned each point in our collection to a group (or *cluster*).

A cluster can be seen as one of these groups, where all the data points belonging to it share a common feature. Also, all the data points belonging to a specific cluster are different from any other point outside this cluster (i.e. points belonging to the other clusters). In summary, we are looking for clusters where *intra-cluster* similarity is high and *inter-cluster* similarity is low. Looking below at the two figures we can see the effect that clustering can have on the same set of random points, giving a clearer structure to the data after grouping.

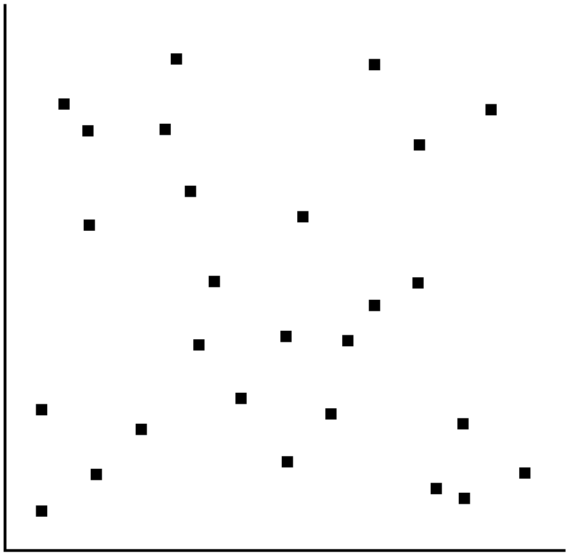

Figure 3.1: Before Clustering: A set of seemingly random data points without clear structure.

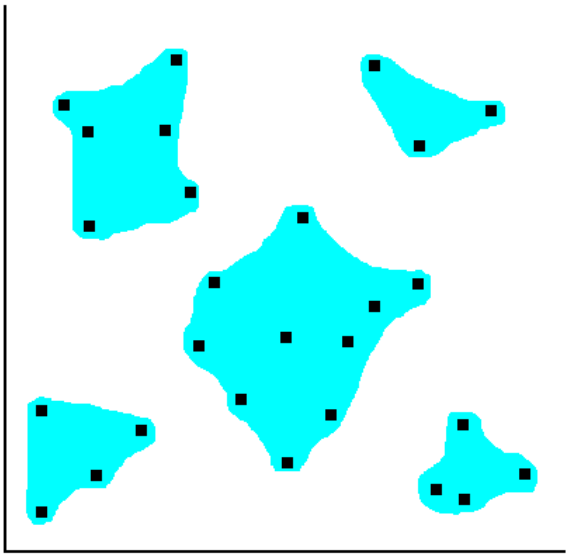

Figure 3.2: After Clustering: A structured collection of points, organized into 5 distinct clusters.

The similarity criteria for the clusters in Figure 3.2 is geometrical distance, so that each cluster contains points which are close to each other. In other words, it is an example of *distance-based clustering*. Note that the similarity criteria can include other measurements other than distance, such as *conceptual clustering*, and is another important feature of cluster analysis.

3.2 Why do we need Clustering Analysis?

Clustering enables us to give some logical grouping to unlabeled data. It can be used to solve a number of large scale or small scale analytical problems. Below, we give several examples of the various *applications* of clustering:

1. In *Ecology*, flora and fauna can be classified by using their features across different communities in heterogenous environments.
2. In *Bioinformatics*, clustering algorithms can be used to assign genotypes (i.e. the genetic makeup of an organism).
3. In *Marketing*, clustering can be used to discover distinct groups in the firm's customer databases and use this knowledge to develop their future marketing programs.
4. In *Seismology*, the earthquake epicentres can be clustered to detect the most likely zones which are to be affected.
5. In *Criminology*, "hot spots" (i.e. areas where incidents of crime happen more frequently) can be identified over a period of time using cluster analysis.

3.3 Clustering Algorithm Classification

The three main classifications of clustering algorithms we will explore are listed below. Each classification type will have its own section where we elaborate on their methodologies and possible drawbacks.

1. Hierarchical algorithms
2. Density-based algorithms
3. Partitioning algorithms

3.3.1 Hierarchical Clustering

Hierarchical clustering forms new clusters from the union of two existing clusters which are closest to each other. The two types of hierarchical clustering algorithms include:

1. **Agglomerative** (aka "bottom-up")- Clusters are merged iteratively until all objects are in one large cluster, which can be the termination condition. AGNES (or AGlomerative NESting) was introduced by Rousseeuw and Kaufmann in 1990 [21] and merges nodes which have most in common with each other, until all nodes belong to the same cluster. AGNES is implemented in statistical analysis packages such as S+.

2. **Divisive** (aka "top-down")- Clusters are split iteratively and successively smaller clusters are formed until each cluster has one point in it. DIANA (or DIvisive ANAlysis) [21] is essentially the inverse of AGNES and terminates when each node forms a cluster comprising of itself. It is also implemented in the S+ package.

If we integrate hierarchical clustering and distance-based approach, we obtain a data clustering method known as BIRCH [16] (Balanced Iterative Reducing and Clustering using Hierarchies) which was written about in 1996. BIRCH uses Clustering Feature tree and incrementally improves the quality of the sub-clusters.

One disadvantage of agglomerative clustering is that the time complexity for n objects is $O(n^2)$ which means it does not scale well. Another is that once a step in the algorithm has been carried out, it cannot be undone.

3.3.2 Density-based Clustering

Density-based clustering algorithms are essentially based on density-connected points. It uses clusters as areas where the density of data points goes beyond a boundary. Some important features of density-based clustering is the discovery of arbitrary-shaped clusters and the abilitiy to handle noise. However, in order for the algorithm to terminate, density parameters are needed as a termination condition.

DBSCAN (Density Based Spatial Clustering of Application with Noise) was proposed by Sander, Kriegel, Xu and Sander in 1996 [20]. It is essentially a clustering algorithm which uses the idea of density-reachability (i.e. a data point is within a boundary distance or neighborhood of another point). A cluster is defined as a maximal set of density-connected points (i.e. data points x and y are density-connected if they have a common point p such that x and p and also y and p are density-reachable). If a data point is density-connected to any point of the cluster, then it also belongs to that cluster.

DBSCAN has a runtime complexity of $O(nlog(n))$, for n objects, if an efficient index structure is used. However, without this structure, the complexity becomes $O(n^2)$, which is worse still. Below, we run through some advantages of DBSCAN:

1. DBSCAN takes noise into consideration.

2. Unlike *K-means* (see next section below), DBSCAN does not require the number of clusters as input.

3. DBSCAN can find clusters which are surrounded by other clusters and other such arbitrarily shaped clusters.

A disadvantage of DBSCAN includes the fact that it relies heavily on the function $getNeighbours(P, \epsilon)$ which contains the distance measure. With data given in many dimensions, the distance metric becomes useless and thus it is difficult to find an accurate value to assign to ϵ.

3.3.3 Partitioning Clustering

The name "partitioning" helps explain the concept of this type of clustering as data is split between clusters. The idea of splitting can be seen on a two-dimensional plane, where the points can be separated by a line, for example. Each data point in the collection belongs to a definite cluster and therefore cannot be included in another cluster. Another name for partitioning clustering is *exclusive* clustering.

The main type of partitioning clustering which we will discuss is *K-means* clustering. First developed by MacQueen [17] in 1967, it assumes we have k clusters and for each of the clusters we define a *centroid*. The centroid is just the mean point of the cluster. The K-means algorithm follows these four simple steps:

1. Arrange the k initial centroids among the data points.
2. Assign each data point to the group with the closest centroid. Note that the Euclidean distance measure is used here to find the closest centroid.
3. Once all the data points have been assigned, recalculate the positions of the k centroids.
4. Repeat steps 2 and 3 until the centroids converge (i.e. stay in the same position). Termination.

The strengths of K-means lie in its efficiency of $O(ikn)$ where i is the number of iterations, k is the number of clusters and n are the number of data points in the collection. By increasing the number of iterations, we are also ensuring that the initial randomly selected centroids do not affect the outcome of the algorithm. It is important, nonetheless, to strategically place the initial centroids as far as possible from each other.

Some disadvantages of K-means include: The number k must be specified beforehand. It can be quite difficult to find the optimal k from the first try, thus requiring an initial run of trial and error with the algorithm. K-means does not handle noisy data, and is also quite sensitive to outliers.

3.4 Advantages of Clustering

The benefits of clustering are summarised below:

1. Clustering helps to review the data, revealing in the best cases any trends found in the collection.
2. It provides a process of efficiently grouping a data set of variable size.
3. Clustering is parsimonious in the sense that it leads to fewer parameters when applied to HMMs, for example.
4. A profile of individual data points can be created, classifying data in a clear and logic manner.

3.5 Disadvantages of Clustering

We have also compiled a list of the general negatives of clustering:

1. When dealing with a large set of data points, the process of clustering can be a problem in terms of time complexity.

2. It is difficult to find that optimal number of clusters to give as input, which yields the best results for your data, as is needed with k-means clustering.

3. Data points can only be characterized in one way because each point can only belong to one cluster.

3.6 K-means for Flash and Hospital Models

The decision to use the K-means clustering algorithm for the two models was made early on in the project timeline. Firstly, K-means was a relatively simple algorithm to implement compared to the other types we discussed above. Secondly, the idea of the Euclidean distance being a standard metric meant we could use it to distinguish between a pair of data points, in our case the number of reads and writes per bin. Thirdly, we could understand how to create the clusters from the collection of data points by the simple iterative steps of the algorithm. By implementing the K-means algorithm in Java, we were able to achieve this.

The only drawback we had to deal with was assigning the value for k, the number of clusters. This required a degree of trial and error which included comparing the raw and cluster-generated standard deviations to identify the best results given the set of data points. The choice of k also depended on the size of the collection data we used. For example, for the Flash Memory raw trace, there were several entries with 1000+ reads per bin which meant our range of values was large. This required more clusters and thus we decided to use $k = 7$ for the Flash HMM. For the Hospital patient arrival data, we only used 3 clusters because the count of arrivals per bin only ranged between 0 and 9.

In the Future Work section of this report, we discuss the possibility of using a different clustering algorithm (e.g. DBSCAN) and then compare results with our K-means algorithm. If we had more time, it would be feasible to investigate the suitability of each type of clustering algorithm for each of our models (i.e. Flash and Hospital) and summarise the results for the best clustering algorithm in each case.

Chapter 4

Flash Memory Workload Model

In this section, we describe in great detail, how we obtained our Flash Memory workload model. Each subsection below is ordered in chronological fashion, beginning with the raw trace. We then discuss how we obtained our binned trace, the set up of our K-means clustering algorithm used to create the sequence of observations and finally displaying the HMM parameters given by Baum-Welch algorithm. Finally, we analyse the results of the Viterbi-generated state sequence and finish this section with the results validating our HMM, including summary average statistics and autocorrelation functions.

4.1 Raw Trace

We begin by analysing the *raw trace* that has been provided by my supervisor Peter Harrison. This large raw trace has hundreds of thousands of entries, and was taken from NetApp storage servers. A CIFS (Common Internet File System) network trace (of about 750 GB) was collected from file servers at the NetApp headquarters, where the servers were accessed mainly by Windows desktops and laptops using various applications.

The trace we analysed for this HMM formed only a part of this network trace (about 12 GB) and is made up of I/O commands (single CIFS *reads* and *writes*). Each entry of the collected trace is of the form:

Cmd: Write
Timestamp: 2.4294339401
PID: 2520
IP: 10.58.48.58
Filename: 2C4F6E8245688F384DFEE31DE815D6850BC4A707
Size: 72
Offset: 0

For each of these entries displayed above, we check whether the command (Cmd) is a ”Read” or ”Write” and also its ”Timestamp” (i.e. the time in seconds when the command was made). The data itself was stored on a web page, which we transferred into read and write arrays using an *InputStreamReader* by reading each line and de-

ciding if it was a read or write command. Below, we go into more detail how this was done and particularly how the Timestamp value was used to assign each entry to a specific subsection of the time series or "bin".

4.2 Binned Trace

We partition the entries of the raw trace into uniform bins of a pre-defined size. These bins are intervals of a constant size which split up the raw data into a discrete time series. Each bin is made up of two values: the number of read entries and the number of write entries which occur at that given time interval.

Choosing the size of the bin is important and depends on the timescale required for the modelling exercise. For example, if the raw trace spans a time period of several days, then we expect much larger bin sizes than if we had a raw trace spanning a couple of hours. Also, the level of detail at which the raw trace is operating (e.g. at the Application level) is also an important factor in determining the bin size.

After experimenting with the raw trace, we found that the best bin size was 1 second. This was appropriate given the observation time of the raw trace (lasting about 6000 seconds). If the bin size was less, having tried 100 milliseconds, then we would have too many time intervals that were empty. On the other hand, with a larger time interval (i.e. 5 seconds) there were issues of missing out low-level, operation sequence characteristics such as mode transitions. Also, we noticed a cyclical pattern where every 4th second of the 5 was empty (i.e. had no reads or writes in that time interval).

Therefore, we used this 1 second bin to our advantage. The number of reads and writes were stored in arrays and each index of these arrays represented a unique bin. In other words, each index held information about that particular second from the data trace of I/O operations. For example, if the two arrays store values *reads[3] = 20* and *writes[3] = 2* then it means that in the 3rd second we counted 20 reads and 2 writes. So, using the Timestamp value when counting the number of reads or writes per bin, we incremented the value at the corresponding index in our arrays. If no Timestamp was present for that bin, then the value at that index was left empty, as all the array entries were already initiated to 0. After this process was finished, we obtained the complete binned trace in the form of our *reads* and *writes* arrays.

4.3 Clustering Algorithm

The next step was to apply a clustering algorithm to the *binned trace* and further reduce the trace to a more manageable length (i.e. the *observation* trace). We implemented the *K-means clustering algorithm*, which essentially grouped the data into K clusters. Each cluster contains a pair of values representing the *centroid* (i.e. the mean number of reads and mean number of writes for that group of points) and all the data points belonging to that cluster (i.e. all the reads and writes in the group). For each cluster, we also carry the standard deviation of all the data points, which will be used later in the validation of the model.

The formation of the clusters was done by using a Euclidean-distance itervative

algorithm which calculates cluster centroids over and over again until they become fixed. As we inputted K manually, we chose a value of 7 clusters. This value was agreed to be not too large (which gives surplus or even empty clusters) or too small (missing out significant differences among clusters) for our data trace.

The seven clusters are listed below as vectors, with the centroid written as a pair of values, the first value representing reads and the second value for writes. These are essentially our seven observation values:

$$\begin{pmatrix} 964.53 & 2.18 \\ 221.87 & 0.35 \\ 1.49 & 0.69 \\ 160.92 & 0.78 \\ 637.5 & 0.37 \\ 394.08 & 0.2 \\ 77.35 & 2.95 \end{pmatrix} \tag{4.1}$$

As we can see above in 4.1, observation values represent low writes with increasing reads. The read values start off very low and progress to quite low, then medium, high, and very high. It is expected that there are not many varying writes (i.e. medium or high writes) in this read-dominated trace.

4.4 Baum-Welch algorithm

4.4.1 Initialization

We use a sequence comprising of observation values as defined in 4.1 to give to the Baum-Welch algorithm as input. Choosing to use the I/O trace which observed the system for 3000 seconds, our sequence will have a length of 3000. Initially, we set the following parameters for the Baum-Welch algorithm:

1. We shall start by having two hidden states for our HMM (read and write).
2. For the initial state distribution, we assume an equiprobable distribution:

$$\pi_0 = (0.5, 0.5)$$

3. For the transition probabilities, we shall assume the following distribution based on the information from the read-dominated raw trace. Most of the time, there will be a good chance of staying in the current state. The initial transition probability matrix is given by:

$$A_0 = \begin{pmatrix} 0.8 & 0.2 \\ 0.4 & 0.6 \end{pmatrix}$$

4. For the emission probabilities, we assume again an equiprobable distribution:

$$B_0 = \begin{pmatrix} 0.14286 & 0.14286 & 0.14286 & 0.14286 & 0.14286 & 0.14286 & 0.14286 \\ 0.14286 & 0.14286 & 0.14286 & 0.14286 & 0.14286 & 0.14286 & 0.14286 \end{pmatrix}$$

4.4.2 Results

From the sequence of 3000 observation values that we used, the Baum-Welch algorithm produced the following **transition probability matrix** (approximated to 4 decimal places):

$$A = \begin{pmatrix} 0.9763 & 0.0237 \\ 0.0774 & 0.9226 \end{pmatrix} \tag{4.2}$$

The **emission probability matrix** can be seen below (to 4 decimal places):

$$B = \begin{pmatrix} 0.0006 & 0.0397 & 0.9185 & 0.0034 & 0.0083 & 0.0215 & 0.0080 \\ 0.2341 & 0.0957 & 0.0638 & 0.0451 & 0.3357 & 0.1909 & 0.0347 \end{pmatrix} \tag{4.3}$$

We can also calculate the **initial state distribution**:

$$\pi = (0.0, 1.0) \tag{4.4}$$

From the results above, we can observe that initially, there is a certainty we will start in the *read state* (state 2). The probability that we move into the *write state* (state 1) is 0.0774 and the probability that we stay in the read state is therefore 0.9226 (as the rows in the transition probability matrix must sum up to 1). Once we are in the write state, the probability we stay in this state is 0.9763 and therefore the probability that we move back to the read state is 0.0237. Overall, matrix A 4.2 shows us that once we find ourselves in a specific state, we will most likely stay in that state for some time.

The emission probability matrix (4.3) shows us that in the write state (i.e. the first row), we are most likely to obtain observation 3 (low reads and low writes), and least likely to see any other observations. This supports our expected behaviour of our read-dominated trace, where it is unlikely that we will observe medium or high number of reads from the write state.

However, from the read state, there are lots of possible observations which could occur. In this state, there is a more even spread than what we see in the write state. Looking at the second row in matrix B, the most likely observations (or clusters) are $1, 5$, and 6 which all contain medium to high reads. So, as expected given the read-dominated trace, we observe various levels of reads in this state.

4.5 Viterbi-generated Sequence of States

The next procedure we attempted was to generate a sequence of the states responsible for producing our observation sequence (i.e. sequence with values 1-7). To produce this sequence of states, we implemented the Viterbi algorithm, and gave as inputs the HMM and the observations, expecting a list of states as output.

From our results (sequence of 3000 states is not shown here), we notice that we initially are in the read state and continue to remain in this state until the 13th observation when we move into the write state. We continue to oscillate between these two states with the majority of the observations coming from the read state. One

reason for this behaviour might be the lack of write entries seen in the I/O trace. When the HMM was trained, it was given a read-dominated trace, and therefore missed a sufficient level of variety of write values to analyse. In fact, most of the write entries were empty, which left the reads to set the trends for the time series. Our HMM generates more reads due to the emissions probability matrix (see 4.3) which informs us that the read state is more likely to generate non-empty observations (i.e. observing at least one read or one write). We can also deduce that the empty obervations most likely were generated from the write state. Therefore, one expects to see more non-empty observations than empty ones, meaning we are likely to observe more read states than writes. Quite fittingly, the Viterbi algorithm generated 2286 reads and 714 writes out a total of 3000 observations.

4.6 HMM-generated Trace

Once we have obtained the three parameters for our HMM (i.e. initial state distribution, transition probability matrix and emission probability matrix), we can use a type of random simulation to produce our own sequence of observations. These observations will contain a value 1-7 as they will be based on the observation set we obtained from clustering the binned trace.

With any simulation, there must be an element of randomness so that it is a fair experiment. We used random numbers in our algorithm to decide two matters: firstly, how the next state was chosen in the transition probability matrix; and secondly, how the observation was chosen from that state in the emission probability matrix. After we obtained the HMM-generated trace (of 3000 points, each belonging to one of the 7 clusters), we compared the means and standard deviations of the HMM and original trace (i.e. the binned trace) to validate our model. Our results use the cluster centroids to calculate the means and standard deviations and are summarised in the table below:

Reads/bin	Writes/bin
Raw Mean: 149.217	Raw Mean: 0.732
HMM Mean: 148.250	HMM Mean: 0.734
Raw Std Dev: 278.258	Raw Std Dev: 0.476
HMM Std Dev: 279.013	HMM Std Dev: 0.480

Figure 4.1: Statistics for raw and HMM Flash Memory traces of 3000 points

We can see from the means, these are very accurate results considering our relatively large set of data of 3000 points. It appears the random distribution of the HMM-generated trace has matched the raw trace well. The mean values for the raw and HMM reads are in excellent agreement, as are the mean writes, which are equal in value for 2 decimal places.

The standard deviations produce even more satisfying results, beginning with the reads. The standard deviations for the raw and HMM-generated reads are almost double of the means, but are closer to each other. The standard deviations for the writes match as well, with the HMM producing an almost identical figure to match the raw trace.

4.7 Autocorrelation

The name *autocorrelation* comes from correlating data with itself (see [24]). More specifically, autocorrelation is a computational method for comparing two time series, where the second series is a lagged version of the first time series over a number of time periods. Hence, another name for autocorrelation is *lagged correlation*.

The result of autocorrelation between the time series and its lagged version is a number between -1 and $+1$. If the result is -1 then we have a perfect negative correlation, and if it is $+1$ then we have a perfect positive one. As there are similarities between *autocovariance* and the formulas used to compute autocorrelation, these two terms are sometimes used interchangeably in industry.

The autocorrelation function (ACF) for observations $y_1, y_2, \ldots, y_N$ can be defined as follows:

$$p_k = \frac{\sum_{t=1}^{N-k}(y_t - \bar{y})(y_{t+k} - \bar{y})}{\sum_{t=1}^{N}(y_t - \bar{y})^2}$$

where $\bar{y}$ is the mean of the observations $y_1, y_2, \ldots, y_N$.

One of the main purposes of the ACF is to find a desired time series model, assuming the data is not random. Time series analysis helps us investigate the data, looking specifically for some internal structure. Thus, another use of ACFs is to find trends or cycles in the autocorrelated time series. In this section, we apply the ACF (defined above) on the Flash data traces as follows: firstly, apply the ACF on the raw, unclustered traces; secondly, apply the ACF on the HMM-generated traces. We then compare the results for both of these time series and and attempt to explain our findings.

We have chosen an observation set of 3500 points, which gives us a total of 1750 lags for our ACF. These lags will be used to find an appropriate time series model. The first pairs of graphs below show how the autocorrelation of reads behaved for increasing lags when comparing the raw trace with the HMM-generated trace.

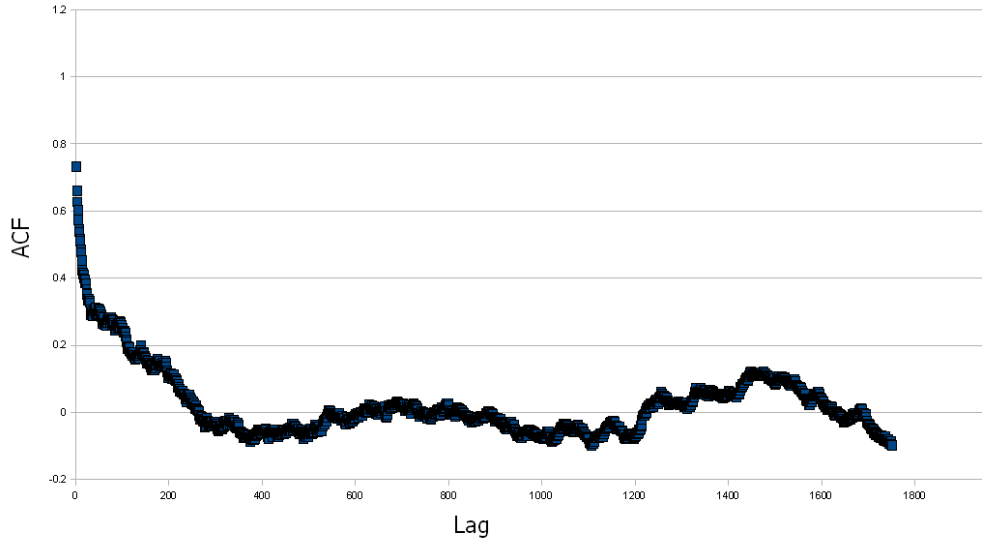

Figure 4.2: ACF for raw reads.

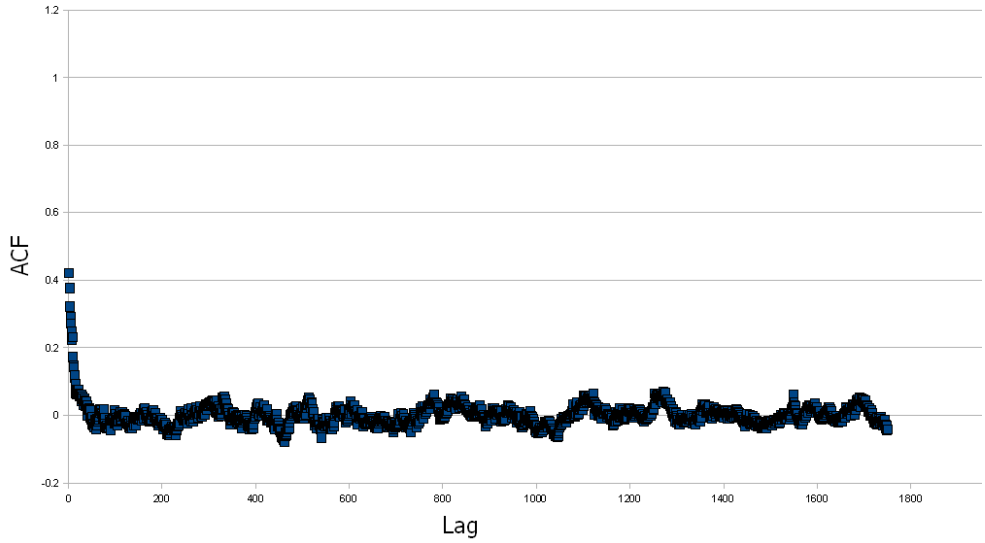

Figure 4.3: ACF for HMM-generated reads.

First of all, we can see that there is significant autocorrelation in both the raw trace and the HMM-generated trace. The oscillations in the autocorrelation are quite similar in both graphs, but are longer with greater magnitude in the raw reads (ranging from values of 0.7 to -0.1). For the ACF of the HMM reads, the oscillations have smaller magnitudes and occur more frequently, almost like bursts of correlation about every 200 lags. An explanation of this difference might come from the smoothing which is assumed by our clustering algorithm.

Below, we see the autocorrelation graphs for raw writes versus HMM-generated writes.

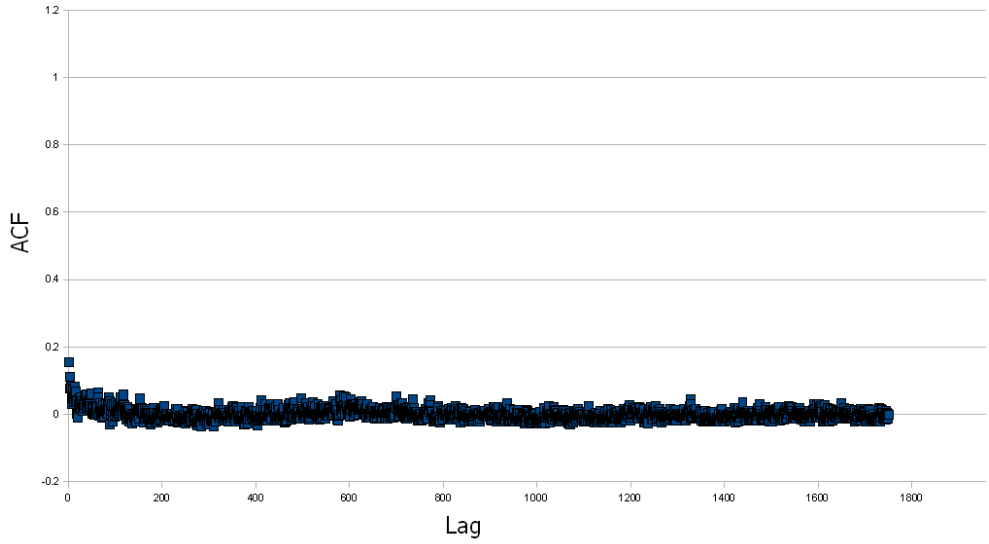

Figure 4.4: ACF for raw writes.

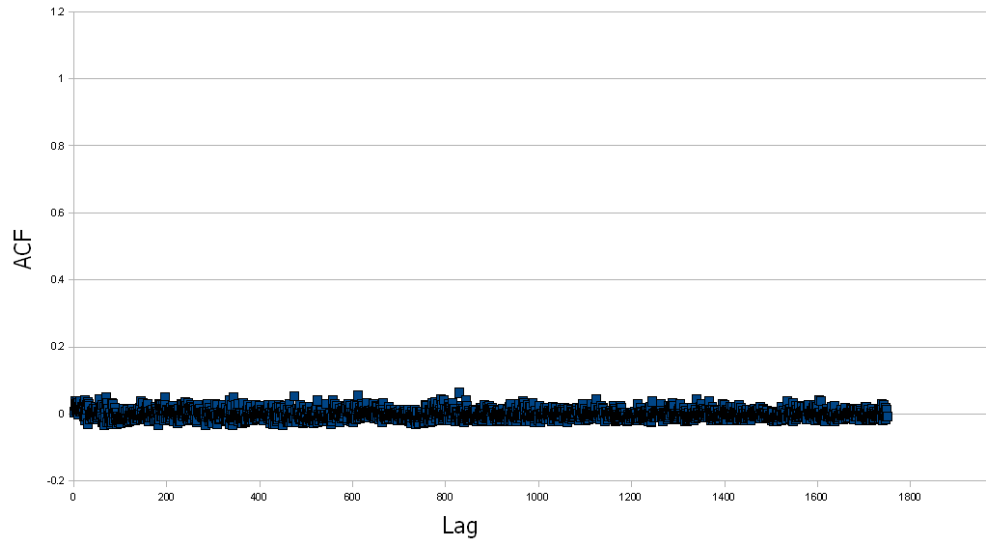

Figure 4.5: ACF for HMM-generated writes.

For the writes, there is very little correlation as most lags have a value very close to zero. Perhaps this characteristic is brought about by the read-dominated trace which we gave as input, meaning there were insufficient non-zero write entries to correlate.

Given that our Flash HMM has given satisfying results through accurate statistics (seen in 4.1) and ACFs, we now proceed to one of the main contributions of this project.

4.8 Sliding version of the HMM

4.8.1 Motivation behind Sliding version

The inspiration for a *sliding* HMM came from my supervisor, Peter Harrison, in the FLASH paper [1]. In this paper, the authors discuss the possibility of handling infrequent, higher density, additional loads because they would like to use their methodology for on-line characterization of workloads. Therefore, the availability of a HMM that has its parameters updated "on-the-fly" as more real time workload data becomes available would help them achieve this. Another advantage of a sliding HMM is to keep track of processes that change with time (i.e. changing the observation set at different stages of the analysis). Yet another benefit is that the constant size of the observation set helps reduce the space and time complexity of the Baum-Welch algorithm, which now must only deal with the new observations.

Realising the benefits this would have on most HMM applications, we decided to attempt to find an approximate sliding version of the HMM. Using our Flash HMM Baum-Welch algorithm, we experimented the possibility of adding new data points to our input trace without re-calculating all the parameters again, whilst simultaneously discarding any "outdated" observations points. The success of this could cut processing times significantly, making HMMs more efficient and therefore workloads in general computationally more cost effective.

4.8.2 Moving Average

A *moving average* or *running average* is a statistical technique where a set of data points is split into subsets and averages are calculated on each of these subsets. For a simple moving average (SMA), we select a fixed subset size (n) and keep shifting along, subtracting old points from the summation as we add new points to it. For example, if we begin with the data points $\{x_t, x_{t+1}, \ldots, x_{t+n}\}$, then we can work out an average of these points:

$$ave = \frac{x_t + x_{t+1} + \cdots + x_{t+n}}{n} \tag{4.5}$$

Then from 4.5 we can create a SMA when we add one more data point (x_{t+n+1}):

$$\begin{aligned} sma &= \tfrac{x_t + x_{t+1} + \cdots + x_{t+n} + x_{t+n+1} - x_t}{n} \\ &= \tfrac{x_t + x_{t+1} + \cdots + x_{t+n}}{n} + \tfrac{x_{t+n+1}}{n} - \tfrac{x_t}{n} \\ &= ave + \tfrac{x_{t+n+1}}{n} - \tfrac{x_t}{n} \end{aligned}$$

We shall apply this idea of SMA to HMMs for observation sets of fixed length. We replace the data points in the simple example above by our model recurrence terms such as αs, βs, etc. To begin with, an incremental version of the HMM (and the Baum-welch algorithm in particular) is proposed, where we intend to a modify the Forward-Backward algorithm to achieve this.

4.8.3 Incremental version of the Baum-Welch algorithm

In order to create any form of incremental Baum-Welch algorithm, we must adopt a new technique of storing existing α and β values and just calculate the new α and β values for the new set of observations. For example, if we are given the observation set $\{O_{T+1}, O_{T+2}, \ldots, O_{2T}\}$ having an existing HMM defined on the observations $\{O_1, O_2, \ldots, O_T\}$, then the αs for the new set of observations will be updated as follows:

For $T \leq t \leq 2T$, we have

$$\alpha_{t+1}(i) = b_i(O_{t+1}) \sum_{j=1}^{N} \alpha_t(j) a_{ji}$$

However, we cannot compute the new β values incrementally for the new observation set $\{O_{T+1}, O_{T+2}, \ldots, O_{2T}\}$ without working out all the β values for the aggregate observation set $\{O_1, O_2, \ldots, O_{2T}\}$. Unlike the α values, the β values use a backward recursion where we need to set $\beta_{2T}(i)$ to 1 (as O_{2T} is our latest observation) and therefore modify all the β values before it using our backward recurrence formulas.

There exists a solution, or more specifically an approximation, of these unknown β values for the new observation set $\{O_{T+1}, O_{T+2}, \ldots, O_{2T}\}$. The technique used by Stenger et al. in 2001 ([23]) assumes the following simple approximation:

For $1 \leq i \leq N$, we have

$$\beta_T(i) = \beta_{T+1}(i) = \beta_{T+2}(i) = \cdots = \beta_{2T}(i) = 1 \tag{4.6}$$

However, from the knowledge of the traditional backward recurrence formula for the β values, we can deduce that the sequence $\beta_{2T}(i), \beta_{2T-1}(i), \ldots, \beta_T(i)$ decreases in value, where $\beta_{2T-1}(i)$ is significantly less than $\beta_{2T}(i)$. Eventually, this decreasing sequence of β values should tend exponentially to zero. Therefore, setting all the new β values to 1 as seen in 4.6 is not the most efficient solution.

We can attempt a more accurate approximation for the β values by assuming that only $\beta_{2T}(i) = 1$ and then use the normal β recurrence formula to update the terms: $\beta_{2T-1}(i), \beta_{2T-2}(i), \ldots \beta_{T+1}(i), \beta_T(i)$. Note that we change $\beta_T(i)$ too as it can no longer be equal to 1, because now we have that $\beta_{2T}(i) = 1$.

So, we can write $\beta_{2T-1}(i)$ as follows:

$$\begin{aligned}\beta_{2T-1}(i) &= \sum_{j=1}^{N} a_{ij} b_j(O_{2T}) \beta_{2T}(j) \\ &= \sum_{j=1}^{N} a_{ij} b_j(O_{2T})\end{aligned}$$

And using this we can write $\beta_{2T-2}(i)$ as:

$$\begin{aligned}\beta_{2T-2}(i) &= \sum_{k=1}^{N} a_{ik} b_k(O_{2T-1}) \beta_{2T-1}(k) \\ &= \sum_{k=1}^{N} a_{ik} b_k(O_{2T-1}) [\sum_{j=1}^{N} a_{kj} b_j(O_{2T})]\end{aligned}$$

We continue in this manner until we write $\beta_T(i)$ in terms of all the new β values we calculated above. Once we have an approximate estimation of the αs and the βs, we can work out the ξ values and the γ values for the observation set $\{O_{T+1}, O_{T+2}, \ldots, O_{2T}\}$:

For $T+1 \leq t \leq 2T-1$ we have

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

and for $T+1 \leq t \leq 2T$ we have

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

However, we can only add these incrementally (i.e. one at a time) and therefore we change the Baum-Welch algorithm firstly by incrementally adding one new term for each new observation (in T separate steps). Hence, for each new observation that is added, we can define the modified re-estimation formulas for our incremental HMM parameters $(\hat{\pi}, \hat{A}, \hat{B})$:

Initially at $t = 1$, we have

$$\hat{\pi}'_i = \gamma_1(i)$$

where $i = 1, \ldots, N$

For $\hat{A}$, we have

$$\begin{aligned}
\hat{a}_{ij}^{T+1} &= \frac{\sum_{t=1}^{T}\xi_t(i,j) + \xi_{T+1}(i,j)}{\sum_{j=1}^{N}\sum_{t=1}^{T}\xi_t(i,j) + \sum_{j=1}^{N}\xi_{T+1}(i,j)} \\
&= \frac{\sum_{t=1}^{T}\gamma_t(i)}{\sum_{t=1}^{T+1}\gamma_t(i)}\frac{\sum_{t=1}^{T}\xi_t(i,j)}{\sum_{t=1}^{T}\gamma_t(i)} + \frac{\xi_{T+1}(i,j)}{\sum_{t=1}^{T+1}\gamma_t(i)} \\
&= \frac{\sum_{t=1}^{T}\gamma_t(i)}{\sum_{t=1}^{T+1}\gamma_t(i)}\hat{a}_{ij}^{T} + \frac{\xi_{T+1}(i,j)}{\sum_{t=1}^{T+1}\gamma_t(i)}
\end{aligned}$$

Therefore we only need to compute the new $\xi_{T+1}(i,j)$ and $\gamma_{T+1}(i)$ for the new observation. This is because we already store the $\xi_t(i,j)$ values for $1 \leq t \leq T$ in the $\hat{a}_{ij}^{T}$ entry.

For $\hat{B}$ we have

$$\begin{aligned}
\hat{b}_j(k)^{T+1} &= \frac{\sum_{t=1,O_t=k}^{T}\gamma_t(j) + \sum_{t=T+1,O_t=k}^{T+1}\gamma_t(j)}{\sum_{t=1}^{T}\gamma_t(j) + \gamma_{T+1}(j)} \\
&= \frac{\sum_{t=1}^{T}\gamma_t(i)}{\sum_{t=1}^{T+1}\gamma_t(i)}\hat{b}_j(k)^{T} + \frac{\sum_{t=T+1,O_t=k}^{T+1}\xi_{T+1}(i,j)}{\sum_{t=1}^{T+1}\gamma_t(i)}
\end{aligned}$$

where we only are required to update $\gamma_{T+1}(j)$ (such that $O_{T+1} = k$) as we store all the previously calculated γ values in the $b_j(k)^T$ entries.

Under these modified parameters which are similar to those seen in [23], we can create a HMM that has this incremental version of the Baum-Welch algorithm (referred to as **IncHMM**). It requires only a partial computation of the forward and backward variables and thus can converge to fixed results much quicker than the

traditional Baum-welch algorithm (which has time and space complexity $O(N^2T)$, where T is the number of observations and N is the number of states [22]). We shall test this hypothesis using various observation sets from the Flash Memory data described earlier in this chapter.

4.8.4 Results of Incremental Baum-Welch algorithm

We shall take an observation set of 3000 points and incrementally add 100 new points. The execution in Java for the IncHMM will be done as follows:

```
// Create a HMM with 2 hidden states and 7 distinct observations for 3000 points
HMM hmmFlash = new HMM(2, 7, 3000);

// Initialise the HMM by running the Baum-welch algorithm
hmmFlash.initFlash();

// Create an Incremental HMM that uses the previous HMM and adds 100 new points
IncHMM incHmm = new IncHMM(100, hmmFlash);

// Initialise the IncHMM by running the Incremental Baum-welch algorithm
incHmm.initIncFlash();
```

The results we obtained from the IncHMM Baum-Welch algorithm are as follows:

$$A = \begin{pmatrix} 0.9777 & 0.0223 \\ 0.0762 & 0.9238 \end{pmatrix} \tag{4.7}$$

$$B = \begin{pmatrix} 0.0005 & 0.0387 & 0.9199 & 0.0047 & 0.0087 & 0.0201 & 0.0075 \\ 0.2208 & 0.1163 & 0.0621 & 0.0385 & 0.3660 & 0.1587 & 0.0375 \end{pmatrix} \tag{4.8}$$

$$\pi = (0.0, 1.0) \tag{4.9}$$

As we can see the initial distribution will stay the same (e.g. initially we are in the read state.). The transition and emission probability matrices are very similar to the ones generated by our standard Baum-Welch algorithm in our normal HMM (see 4.2 and 4.3). The means and standard deviations of the raw and IncHMM-generated traces are shown below:

Reads/bin	Writes/bin
Raw Mean: 151.038	Raw Mean: 0.782
IncHMM Mean: 147.905	IncHMM Mean: 0.788
Raw Std Dev: 279.507	Raw Std Dev: 0.436
IncHMM Std Dev: 277.136	IncHMM Std Dev: 0.445

Figure 4.6: Statistics for raw and IncHMM Flash Memory traces of 3000 points

The results above in Figure 4.6 show that our IncHMM is a very good approximation of the raw means for both the reads and the writes. The raw standard deviations are very well approximated by our IncHMM, with the reads being very similar and the writes having almost identical values.

Let us for now refer back to the simple approximation of the β values discussed earlier, where the new observation set is $\{O_{T+1}, O_{T+2}, \ldots, O_{2T}\}$ and for $1 \leq i \leq N$, we have

$$\beta_T(i) = \beta_{T+1}(i) = \beta_{T+2}(i) = \cdots = \beta_{2T}(i) = 1$$

Note that for this approximation, the α values are calculated incrementally only for the new observations, as mentioned before. We ran a simulation of a HMM having this type of incremental Baum-Welch algorithm (referred to as **IBW**) using the same inputs as above (i.e. an observation set of 3000 points and 100 new points). Below, we present the means and standard deviations of IBW:

Reads/bin	Writes/bin
Raw Mean: 151.038	Raw Mean: 0.782
IBW Mean: 118.333	IBW Mean: 0.763
Raw Std Dev: 279.507	Raw Std Dev: 0.436
IBW Std Dev: 249.409	IBW Std Dev: 0.379

Figure 4.7: Statistics for raw and IBW Flash Memory traces of 3000 points

The IBW mean for the reads does not match the raw mean, nor do the standard deviations match for the reads. This is sufficient evidence to show that our IncHMM has a more accurate β recurrence formula for the new set of observations than the simple approximation presented in IBW. Once we established the most suitable recurrence formulas for the Forward-Backward algorithm, our next step is to modify our IncHMM into a Sliding HMM using the concept of simple moving average, as explained earlier. We shall modify the Baum-Welch algorithm yet again to incorporate the sliding in the observation set and present our results in the usual format.

4.8.5 Sliding version of the Baum-Welch algorithm

One major difference between IncHMM and this new Sliding HMM (referred to as **SlidHMM**) is the size of the observation set. For IncHMM, the observation set grows because we are adding new data points for the model to train on. However, SlidHMM will always keep the same size of observations as it shifts along any group of points like a moving average. In a way, it is similar to a IncHMM which takes away (from the front) as many observations as it adds (to the end).

For example, if we had an initial observation set $\{O_1, O_2, \ldots, O_{400}\}$, we could apply SlidHMM (of length 300) twice. The first pass would be a like a normal HMM which receives the observation set $\{O_1, O_2, \ldots, O_{300}\}$. The second pass (aka the 1st slide) is a SlidHMM which receives the observation set $\{O_{101}, O_{102}, \ldots, O_{400}\}$. We would then have two sets of parameters, from each MAP. Ideally, we would take the best set of results (i.e. one with the best match of means and standard deviations when

compared to the raw trace). Alternatively, we take the most recent set of parameters if the latest segment of the time series is of interest.

One obvious advantage of the SlidHMM approach is the time of computation is decreased because each model has an observation set which is of smaller size (300 points) than the size of the initial set (400 points). It also stores the results of the computations done on the first observation set, passing the information on to the next model and its observation set, etc. We must also remember at each slide to discard a fixed amount of old observations and with it the old α and β values. This is done precisely to maintain a constant length of observations as we slide along, thus helping to create a new updated model each time.

4.8.6 Results of the Sliding Baum-Welch algorithm

We shall take an observation set of 3000 points and slide across once, thus incrementally adding 200 new points whilst discarding the first 200 points. The execution in Java is:

```
// Create a HMM with 2 hidden states and 7 distinct observations for 2800 points
HMM hmmFlash = new HMM(2, 7, 2800);

// Initialise the HMM by running the Baum-welch algorithm
hmmFlash.initFlash();

// Create SlidHMM using hmmFlash and do 2 slides on a total of 3000 points
SlidHMM slidHmm = new SlidHMM(hmmFlash, 2, 3000);

// Initialise SlidHMM by running the Sliding Baum-welch algorithm
slidHmm.initSlidFlash();
```

The results we obtained from **SlidHMM** are as follows:

$$A = \begin{pmatrix} 0.9703 & 0.0297 \\ 0.1009 & 0.8991 \end{pmatrix} \tag{4.10}$$

$$B = \begin{pmatrix} 0.0042 & 0.0044 & 0.9568 & 0.0163 & 0.0 & 0.0164 & 0.0018 \\ 0.2663 & 0.1716 & 0.1155 & 0.0399 & 0.2081 & 0.1594 & 0.0392 \end{pmatrix} \tag{4.11}$$

$$\pi = (0.0, 1.0) \tag{4.12}$$

The transition matrix has expected entries, very close to the original HMM. The emission matrix is also well approximated for this new observation set. From these new parameters, the SlidHMM generated a new observation trace. The means and standard deviations of this SlidHMM-generated trace can be seen in the table below:

Reads/bin	Writes/bin
Raw Mean: 149.505	Raw Mean: 0.731
SlidHMM Mean: 149.993	SlidHMM Mean: 0.735
Raw Std Dev: 277.434	Raw Std Dev: 0.478
SlidHMM Std Dev: 276.140	SlidHMM Std Dev: 0.507

Figure 4.8: Statistics for raw and SlidHMM Flash Memory traces of 2800 points

As we can see above in Figure 4.8, there is sufficient evidence that the **SlidHMM** behaves as expected and with excellent averages. Note that with two slides, the computation involved is not too demanding for the 200 new observation points. In the Further Work section of the report, we discuss extending the observation set and attempting three or four slides over this set.

Chapter 5

Hospital Arrival Model

In this section, we will explore the steps which helped us create our Hospital Arrivals Model. We begin by describing the process of obtaining an accurate and realistic raw trace of patient arrivals. Then, we discuss how we used the clustering algorithm and the Baum-Welch algorithm to produce our model parameters. As with the Flash workload model, we will analyse the results of the Viterbi-generated state sequence and also display autocorrelation functions to compare raw and HMM traces.

5.1 Collecting the Hospital Arrival Trace

We obtained data for our patient arrival times from an internal DoC PostgreSQL database called *aesop_artery*. Within this database, we accessed the *arrivals* table to extract the arrival times (from the "time_arrival" column and other information) for a period of four weeks. This gave us a raw trace of about a thousand entries and provided sufficient data to partition into bins and use this binned trace for our clustering algorithm.

We extracted the data from time_arrival using a simple SQL query (with a limit of $10,000$) and unloaded the entries into a file called *hospitalData.csv*. After importing this file into an Excel worksheet, we then saved the patient arrival times into a text file called *TimePatientArrived.txt*. Below we delve into the process of transforming the raw trace into the binned trace.

5.2 Binned Trace

As with the Flash memory data, we used a *InputStreamReader* to read each line of our *TimePatientArrived.txt* file. We begin with an array called *arrivals* which has size 1000 and all its entries set to zero initially. This array will store the number of patient arrivals per hour (where each index represents an hour). We create a simple while loop and set the termination condition to be the size of our array (i.e. 1000). As we read each line of our text file, we check the time, focusing on the hour especially, and increment our arrivals by 1 for that index. If the next line reads a different time, we change our index to match the change in time, else we keep the same index as we are in the same hour slot. Note that when iterating past midnight, we need to change our

index by adding 24 hours to the difference (e.g. change in index from arrival (23:52) to arrival (02:23) is 24 + 2 - 23 = 3). Once this process was done, we obtained our complete array containing the number of patient arrivals per hour for the entire four weeks we chose to observe.

We now analyse the number of patient arrivals per 60 minute interval which we observed over our four-week period. The frequency of patient arrivals that we collected for our observation period are presented in the bar chart below:

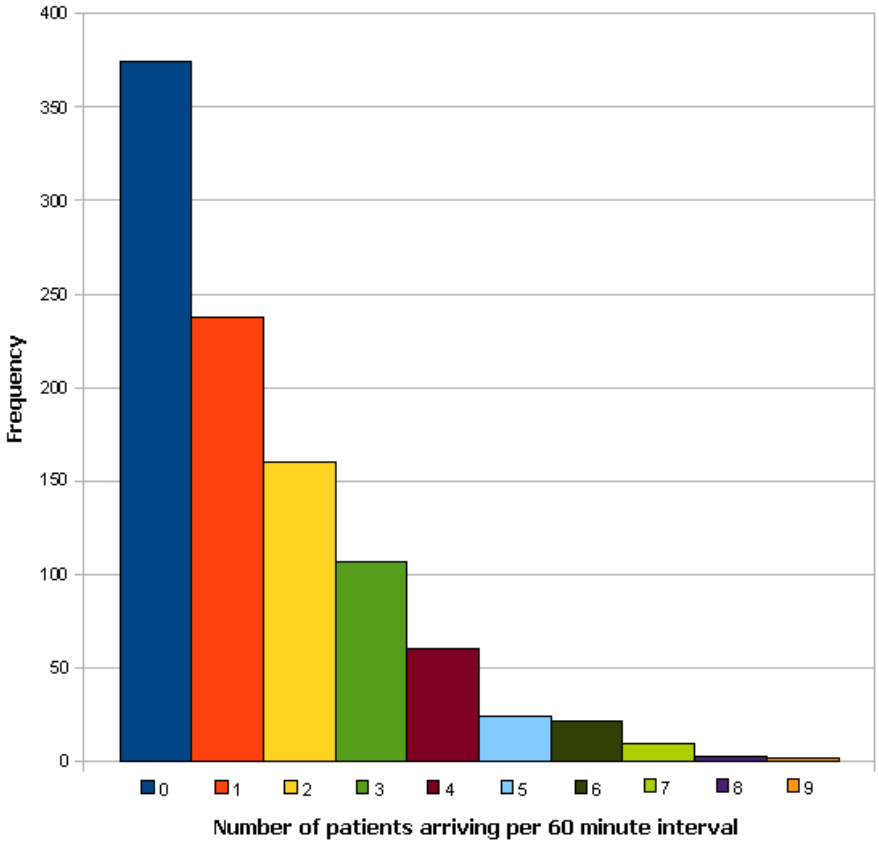

Figure 5.1: Frequency of various patient arrivals in 60 minute intervals.

Looking at the data above it is clear that more than a third of intervals we observed were empty (i.e. no patients checked in). About a quarter of intervals experienced 1 patient, while about a sixth of all intervals received 2 patients. As we increase the number of patients per interval we decrease the frequency and observe that only a handful of intervals received more than 8 patients. We will see in the next section how we can use our data distribution in 5.1 to help us choose the number of clusters for our clustering algorithm.

5.3 Clustering Algorithm

As before we will use the well-known *K-means* clustering algorithm to apply to our binned trace and further reduce it to the ”observation” trace. The centroid for each

cluster in this case would be the mean number of patient arrivals for all the data points belonging to that cluster. As stated earlier, the number of clusters K was chosen with the help of some analytical background information from 5.1. The range of values is only 9, so we do not expect many clusters to be formed. After trying 6 clusters, we find that we had two empty clusters (i.e. with centroids 0.0) so we decided to aim for a number smaller than 6.

Analysing the pie chart below (see 5.2) might help us understand why we must seek less than 6 clusters for our clustering algorithm. We can see that the different arrival patterns are divided into four main sections in the diagram, which suggests that we may only need at most 5 clusters.

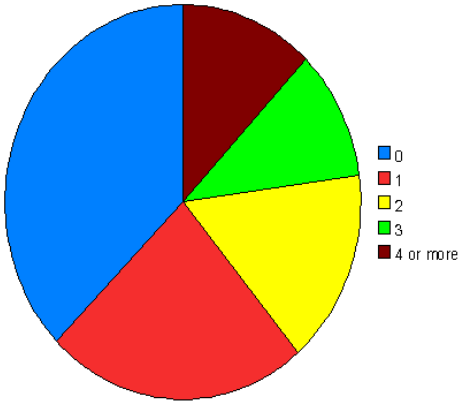

Figure 5.2: Division of the different frequency of arrivals based on same patient arrival data.

As we inputted the value of K manually, we decided to use a value of 3 clusters, as it gave closer means to the raw data when we compared them with our HMM-generated data (see below). The three clusters are listed here and are essentially our observation values:

$$\begin{pmatrix} 4.99 \\ 0.39 \\ 2.4 \end{pmatrix} \tag{5.1}$$

As we can see above in 5.1, observation values from top to bottom represent: very frequent arrivals, very few arrivals and moderately frequent arrivals. Comparing these values to the different sections in diagram 5.2, we see that the first cluster (with centroid 4.99) represents more than 3 arrivals (i.e. the brown and green sections on the pie chart). The cluster with centroid 0.39 will represent the blue section and some of the red section. Finally, the last cluster in 5.1 with centroid 2.4 will use the yellow section and some of the green section also. The observation trace is now represented in terms of these three values and is inputted into the Baum-Welch algorithm.

5.4 Baum-Welch algorithm

5.4.1 Initialization

We observe patient arrivals for 3000 hours and therefore can input an observation trace of 3000 data points into the Baum-Welch algorithm. Note that we use the sequence of observation values as defined in 5.1 to populate this observation trace. As with the Flash HMM, we initially set the following parameters for the algorithm:

1. We shall start by having two hidden states for our HMM. The meaning of these states will become clear when we analyse the output of the Viterbi algorithm further on.

2. For the initial hidden state distribution, we have an equiprobable distribution:

$$\pi_0 = (0.5, 0.5)$$

3. For the transition probabilities, we shall assume the following distribution based on the patient arrival times seen in the raw trace. Most of the time, the transition will not move to a different state. The initial transition probability matrix is given as follows:

$$A_0 = \begin{pmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{pmatrix}$$

4. For the emission probabilities, we assume again an equiprobable distribution:

$$B_0 = \begin{pmatrix} 0.333 & 0.333 & 0.333 \\ 0.333 & 0.333 & 0.333 \end{pmatrix}$$

5.4.2 Results

Using the observation trace made up of 3000 points as input, the Baum-Welch algorithm produced the following **transition probability matrix** for the HMM (approximated to 4 decimal places):

$$A = \begin{pmatrix} 0.8772 & 0.1228 \\ 0.1348 & 0.8652 \end{pmatrix}$$

The **emission probability matrix** can be seen below (to 4 decimal places):

$$B = \begin{pmatrix} 0.0 & 0.9544 & 0.0456 \\ 0.2445 & 0.2625 & 0.4930 \end{pmatrix}$$

We can also calculate the **initial probability distribution**:

$$\pi = (0.0, 1.0)$$

From the results above, we can observe that initially, there is a certainty we will start in state 2. The probability that we move into state 1 is 0.1348 and the probability that we stay in state 2 is 0.8652 (as the rows in the transition probability matrix must sum up to 1). Once we are in state 1, the probability we stay in this state is 0.8772 and therefore the probability that we move back to the other state is 0.1228. Overall, matrix A shows us that once we find ourselves in a specific state, we will most likely stay in that state for some time. This is confirmed by our Viterbi-generated sequence of hidden states (see the section below).

The emission probability matrix (B) shows us that from state 1 (i.e. the first row), we are most likely to obtain observation 2, and unlikely to see observations 1 or 3. Therefore, from this state we expect to observe sparse patient arrivals, even considering seeing no arrivals in the one hour interval. Seeing more than one arrival from this state is unlikely as supported by the emission probabilities. We can gather that this state is quite a sparse state where patient arrivals are rare. For now, we will label state 1 as the *sparse* state.

On the other hand, looking in the second row of our emission matrix, we observe various levels of patient arrivals which are all likely to occur. It seems that this state (state 2) is more active in general, when it comes to receiving patients. On average, we have a 75% chance of seeing at least 2.4 patients per hour in this state, which represents a more dense distribution of patient arrivals than state 1. Therefore, we label state 2 as the *dense* state.

5.5 Viterbi-generated Sequence of Hidden States

The next procedure was to generate a sequence of the hidden states responsible for producing our observation sequence (i.e. sequence with values 1-3). To produce this sequence of states, we implemented the Viterbi algorithm, giving the HMM and the observations as inputs and expecting a sequence of states as output. Our aim was to give meaning to what the hidden states could represent based on the observations each state produced.

Analysing the sequence of 4800 states, we begin in the sparse state (state 1) and after several observations in this state, we switch to state 2, the dense state. We continue to oscillate between these two states until the end of the sequence. An explanation can be formed from this result which could shed light on our oscillating Viterbi state sequence as well as explain the different distributions (i.e. state 1 produced many arrivals, while state 2 produced very few). Our prediction is that state 1 represents *day* and state 2 represents *night*.

To test this claim, we analysed the Viterbi sequence of 4800 states, by counting the number of times each state occured in this long sequence. Doing the computation, we labelled state 1 as day and state 2 as night. The following results were obtained:

As we can see from Figure 5.3, our Viterbi algorithm has generated 52.77% of the day states and 47.23% of night states. This is expected because a single 24 hour day is split into 12 hours of day (e.g. 7am to 7pm) and 12 hours of night (e.g. 7pm to 7am). Therefore we can be pleased for several reasons: firstly, the labelling of the

Viterbi Sequence of 4800 States
Total number of Days: 2533
Total number of Nights: 2267

Figure 5.3: Distribution of Days and Nights produced by the Viterbi algorithm for a sequence of 4800 states.

hidden states as ”day” and ”night” has proved accurate for this sequence of states; secondly, the Viterbi algorithm has supported this theory based on the results in the table above.

5.6 HMM-generated Trace

Once we have obtained the initial state distribution, transition probability matrix and emission probability matrix, randomly generate produce our own sequence of observations using these three HMM parameters. Note, these observations will be contain a value 1-3 as they will be based on the observation set. Similarly to the Flash model, we ran a simulation, and obtained the HMM-generated trace of 3000 points. Then, we compared the means and standard deviations of the HMM and original traces to validate our model with the results presented in Figure 5.4 below:

Patient Arrivals per bin
Raw Mean: 1.483
HMM Mean: 1.461
Raw Standard Deviation: 1.565
HMM Standard Deviation: 1.551

Figure 5.4: Statistics for raw and HMM patient arrival traces of 3000 points

We can see from the table above that the bin-means match well, and more pleasingly, the standard deviations are very similar indeed. We can conclude, from these statistics alone, that our HMM faithfully reproduces meaningful representations of out patient arrival times.

5.7 Autocorrelation

This section shows the results of autocorrelation functions (ACFs) carried out on the raw, unclustered traces and then on the HMM-generated traces. The graphs below show how the autocorrelation of patient arrival times behaved for increasing lags when we compared traces.

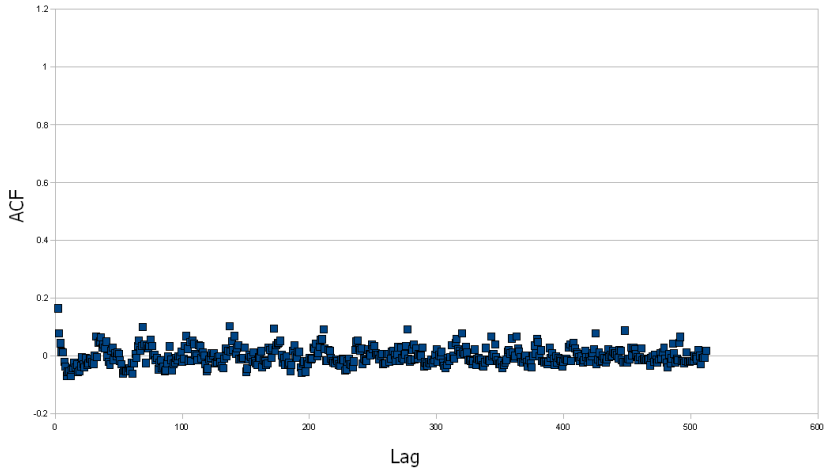

Figure 5.5: ACF for raw patient arrival times.

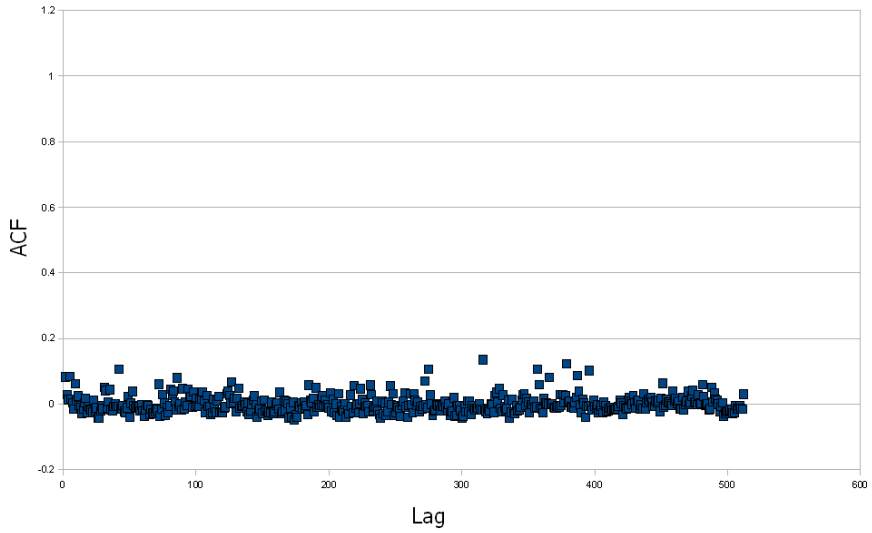

Figure 5.6: ACF for HMM-generated patient arrival times.

As we can see, the two graphs match well as they both show little autocorrelation. The HMM-generated ACF shows less variation than the raw ACF, possibly due to the clustering algorithm.

5.8 Optimal Number of Hidden States

In this section, we will investigate the general process of allocating an optimal number of hidden states to a HMM. As discussed in [14] there are two methods for finding the optimal number of hidden states. The first is called *top-down* and is a binary split scheme, where initially we have few states and iteratively split each state into two new states, etc. We carry on in this fashion until no further improvement can be made.

The second method is called *bottom-up* and can be seen in [15]. In this scheme, we initialize the state number with a large value and during training, we merge states together. Eventually, we will end up with a small, optimal number of states.

Therefore, we can choose between these two methods of finding the best number of states to initialize the Baum-Welch algorithm with. There are ways of checking whether we have an optimal amount of states if a HMM is already set up. One such way being to observe the emission probabilities that are outputted from the Baum-Welch algorithm. For example, if two rows in the emission matrix are very similar (i.e. almost identical set of entries) then we have one too many states for our HMM. Below, is an example of our emission matrix that was produced for the hospital patient arrival times which we gave to the Baum-Wlech algorithm. We used 3 d.p. for the entries of the matrix:

$$\begin{pmatrix} 0.003 & 0.422 & 0.575 \\ 0.437 & 0.157 & 0.406 \\ 0.02 & 0.941 & 0.039 \\ 0.0 & 0.996 & 0.004 \end{pmatrix}$$

Notice that the third and fourth rows are too similar to be a coincidence. Either our algorithm has not converged fully or, more likely, we have too many hidden states and thus our HMM had not been optimally set up. It is easy to deduce now that the Hospital HMM must take either two or three hidden states as input. Therefore, in this case, the use of the *top-down* or *bottom-up* methods are not necessary for such simple models.

After we apply our own method of analysing rows in the emission matrix, we can compare our two remaining cases to find which number of hidden states helps the HMM perform better. The comparison lies in the analysis of the entries in the emission matrix, of course. In [14], a graph of directed accuracies is produced for different hidden states, to pinpoint the exact moment when convergence is achieved:

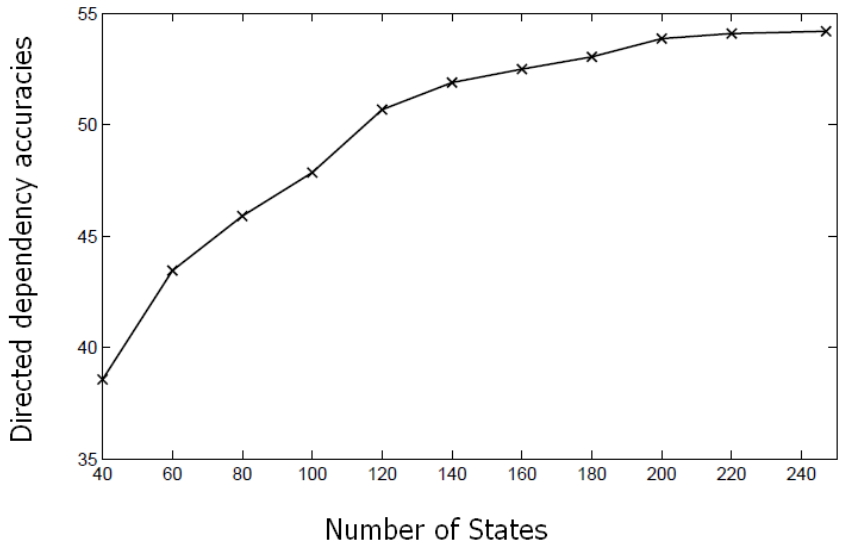

Figure 5.7: Directed dependency accuracies given number of states [14].

It seems that for this project the use of top-down or bottom-up techniques were too complex for the simplicity of our HMM parameters. For example, the small number of states and observation traces we used meant we could achieve the optimal set up of our HMM without the need to merge states. Nonetheless, in the Further Work section of this report, we explore an efficient bottom-up method of merging hidden states for a HMM.

Chapter 6

Evaluation

We evaluate the project in terms of our two models, giving justification for the use of parameters in each case and assessing the success of the execution of the final version. A comparison between our achievements in this project and the existing work in this field will conclude this section.

6.1 Flash Model

6.1.1 Size of Bins

To begin with, the choice of 1 second for the *bin size* proved to be an advantage to this model. This bin size was not too small, leaving too many intervals empty; nor was it too large, missing out operation characteristics such as mode transitions. Another advantage of this bin size was in the storing of the reads and writes in arrays. Each index of the array was used as a bin and the number of reads and writes for each bin translated to an integer in the appropriate index of each array. Therefore, if we observed the system for 3000 seconds then we had 3000 intervals and corresponding reads and writes arrays, each of size 3000.

6.1.2 Choice of Clustering algorithm

The use of the *K-means* clustering algorithm had the advantage of being relatively simple to implement in Java. The algorithm used Euclidean distance as a means of creating the clusters and then assigning each data point to a cluster, which seemed the logical approach of organising integers ranging from 0 to over 1000 (representing the number of reads/writes per bin). As an extension of this project, we discuss the possibility of implementing a different clustering algorithm in the Further Work section in chapter 7.

One important decision for the K-means clustering algorithm was the input parameter: K (i.e. the number of clusters). An ideal number was required to correctly classify all the data points in our observation set, whilst simultaneously grouping similar points efficiently. With too many clusters, the danger becomes that the algorithm assigns two similar data points to two different clusters, therefore losing trends

in the data set. At the same time, if we have too few clusters for our collection of data points, then we might miss out individual differences due to inefficient grouping.

Our choice of setting K=7 resulted in excellent results as simulated for the Flash workload model for 3000 data points. Originally, we had set K to be 10 and observed the system for 3000 seconds, which produced the following cluster centroids (for the reads and writes, respectively) along with the means and standard deviations of the raw and HMM-generated traces:

$$\begin{pmatrix} 967.41 & 2.22 \\ 465.55 & 0.12 \\ 658.17 & 0.42 \\ 213.93 & 0.31 \\ 332.13 & 0.34 \\ 146.15 & 1.54 \\ 176.62 & 0.46 \\ 31.15 & 13.92 \\ 103.81 & 0.0 \\ 0.91 & 0.28 \end{pmatrix}$$

Reads/bin	Writes/bin
Raw Mean: 149.216	Raw Mean: 0.732
HMM Mean: 132.530	HMM Mean: 0.846
Raw Std Dev: 278.258	Raw Std Dev: 2.146
HMM Std Dev: 263.920	HMM Std Dev: 2.041

Figure 6.1: Clustered trace for 10 clusters and statistics for raw and HMM Flash Memory traces.

We notice that for one of the cluster pairs $(103.81, 0.0)$, we have an empty centroid for the writes. This reflects the lack of write commands in the raw trace. It also indicates that there are sufficient data points with many reads, but no writes, and these points will be assigned to this cluster. However, this might be a case of creating too many clusters, bearing in mind that an aim is to have all centroids of each cluster with positive, non-zero values.

Despite the averages in Figure 6.1 being quite similar when we compare raw values to HMM-generated values, they do not perform as well as our results in chapter 4, where we used 7 clusters. The results above perform quite poorly in the means where the HMM mean differs by more than 11% from the raw mean. The standard deviation results were also quite disappointing in this case. Perhaps a value of 10 clusters gave us too many different groups of data points for which to assign the reads to, explaining in the significantly lower values of the mean and standard deviation produced by the HMM.

With a value of 5 clusters, we obtained poorer results than before. The set of 5 cluster centroids and the corresponding table of averages are presented below:

$$\begin{pmatrix} 964.53 & 2.18 \\ 637.5 & 0.37 \\ 391.13 & 0.25 \\ 200.59 & 0.36 \\ 2.35 & 0.74 \end{pmatrix}$$

Reads/bin	Writes/bin
Raw Mean: 149.217	Raw Mean: 0.732
HMM Mean: 192.762	HMM Mean: 0.746
Raw Std Dev: 278.057	Raw Std Dev: 0.392
HMM Std Dev: 309.603	HMM Std Dev: 0.453

Figure 6.2: Clustered trace for 5 clusters and statistics for raw and HMM Flash Memory traces.

In this case, we could deduce that with so few clusters, data points with high centroid values were grouped together with lower centroid values. Therefore, when the HMM generated its observation values, it's possible each value was assigned to a cluster (one of 5) with a much higher mean than its own. This might explain why the HMM means and standard deviations in Figure 6.2 are so high.

Thus, we concluded that a value of 7 clusters was the optimal number to use in the K-means algorithm. It certainly produced the closest averages to the raw trace averages.

6.1.3 Read-dominated trace

Another issue we had to deal with in the Flash workload model was the using a read-dominated trace. It is evident that this affected the structure of our set of observation values and cluster centroids. Due to the lack of writes present in the raw trace, we could only match varying reads with low writes. As a comparison to what our observation values might have looked like if we had a greater variety of writes, we look at Peter Harrison's paper on Flash Memory [1]. Below, we see their observed values along with their centroids ((reads,writes)) for each cluster.

$$\begin{pmatrix} 5.7 & 0.28 \\ 4.45 & 31.3 \\ 5.11 & 82.1 \\ 4.49 & 183.0 \\ 23.7 & 0.217 \\ 24.5 & 31.7 \\ 27.3 & 81.0 \\ 25.8 & 165.8 \end{pmatrix}$$

Figure 6.3: Eight centroids defining the set of observations values for the *update-mix* [1].

As we can see in Figure 6.3, the first four observation values represent low reads and increasing writes. The last four observation values are high reads and increasing writes. This symmetric variation in the read and write values contrasts our own observation values, where the centroids had very low write values. As we will discuss in chapter 7, if we had an I/O trace with more write commands, we could reproduce a more balanced set of observations as seen above.

6.1.4 Baum-Welch algorithm parameters

The parameters which the Baum-Welch algorithm outputs includs the initial transition distribution (π), the transition probability matrix (A) and the emission probability matrix (B). The outcome of these depend slightly on the initial transition probabilities matrix (A_0) which was inputted into the Baum-Welch algorithm and we defined as:

$$A_0 = \begin{pmatrix} 0.8 & 0.2 \\ 0.4 & 0.6 \end{pmatrix}$$

After experimenting with the entries of A_0, we decided that these values were the best set-up to our Baum-Welch algorithm for the Flash workload model. However, the following transition matrix was used as input and the same transition matrix was outputted:

$$A_0 = \begin{pmatrix} 0.8 & 0.2 \\ 0.2 & 0.8 \end{pmatrix} = A$$

Also, the emission matrix that was ouputted had identical rows, meaning the states were treated as being identical too. Therefore, we had to differentiate somehow between our two hidden states. We did this through our distribution of the transition matrix. This was enough information for the Baum-Welch algorithm to then identify the different states and represent each of their behaviour in the emission matrix.

6.1.5 Three hidden states

When the decision was made to use two hidden states for the HMM, we chose to label one hidden state as the *read state* and the other as the *write state*. However, this did not imply that if we were in the read state, for example, then we only observed reads and no writes. It was more an indication of what we *expected* to observe whilst in that state. A decision was made to use the reads and writes labels since the entries in the trace we analysed were only composed of these two commands. If there were a third command in our raw trace (e.g. *delete*) then we might try running our HMM with three hidden states. In fact, trying three hidden states gave us quite interesting results. We initialised the initial distributions and emission probability matrix to equiprobable entries (e.g. each matrix had the same value in every entry) and inputted the following transition probability matrix:

$$A_0 = \begin{pmatrix} 0.8 & 0.1 & 0.1 \\ 0.2 & 0.6 & 0.2 \\ 0.1 & 0.2 & 0.7 \end{pmatrix}$$

As before, we used different probability distributions for each row, making sure that there was a likely chance that a transition would not change the state (i.e. if we were in state 1 there was an 80% chance we would stay in state 1, there was a 60% chance we would not leave state 2, etc.). Given these inputs, the HMM produced the following results:

$$\pi = \begin{pmatrix} 0.0 & 1.0 & 0.0 \end{pmatrix}$$

$$A = \begin{pmatrix} 0.9744 & 0.0248 & 0.0007 \\ 0.0607 & 0.9127 & 0.0266 \\ 0.0031 & 0.0425 & 0.9544 \end{pmatrix}$$

$$B = \begin{pmatrix} 0.0005 & 0.0016 & 0.9858 & 0.0004 & 0.0065 & 0.0011 & 0.0042 \\ 0.0197 & 0.1913 & 0.4975 & 0.0472 & 0.0409 & 0.1611 & 0.0422 \\ 0.3296 & 0.0292 & 0.0208 & 0.0088 & 0.4668 & 0.1362 & 0.0087 \end{pmatrix}$$

We notice that, initially, we will always be in state 2. The emission probability matrix (B) reveals that given we are in state 1 (i.e. in the 1st row), we are almost 99% sure to produce observation 3, which is the cluster with small reads and small writes. This can be interpreted as the *write state* as it does not produce many reads. If we are in state 2 (i.e. the 2nd row) then we are likely to observe mostly small reads and small writes, but also medium reads and small writes. This can be interpreted as a *weak-read state*, where we expect to see moderate writes and reads, but not too many reads. Finally, the last state (row 3) shows that we are either expecting high reads or very high reads, so we call this the *strong-read state*.

The use of three hidden states would fit this Flash workload model just as efficiently as our current two state model. It is quite acceptable to split our read state into two different states: a weak-read state and a strong-read state. We can now distinguish between intensities of reads, brought about by this read-dominated I/O trace.

A disadvantage of having these three hidden states is that the states are not so distinct as they were previously. There is also the issue of assigning boundary values to certain states, which raises questions. For example, what is the acceptable boundary for deciding if an observation should be in the weak-read state or in the strong-read state? Also, might some of the blame for this problem belong to the clustering algorithm for not efficiently assigning observation values to each state? The solution to these issues are left as possible extensions of this project.

6.1.6 Variations of Sliding HMM

There were a number of choices for approximating the β values with the sliding Baum-Welch algorithm. One solution was to only update the α values for the new observations, and to work out the β values all over again for the accumulated observation set. In other words, suppose we have the old observation set $\{O_1, O_2, \ldots, O_T\}$ and the new observation set is $\{O_{T+1}, O_{T+2}, \ldots, O_{2T}\}$. When updating the β values (for $i = 1, \ldots, N$), we use the standard backward algorithm setting $\beta_{2T}(i) = 1$ and applying the recurrence formula

$$\beta_{T-1}(i) = \sum_{j=1}^{N} a_{ij} b_j(O_T) \beta_T(j)$$

to calculate all the β values down to and including $\beta_1(i)$.

This would save about half of the computation time (only for the α values) of our Sliding Baum-Welch algorithm. We attempted a simulation of this version of the Sliding Baum-Welch algorithm similar to that seen at the end of chapter 4 and achieved the following results:

$$A = \begin{pmatrix} 0.9681 & 0.0319 \\ 0.1269 & 0.8731 \end{pmatrix} \tag{6.1}$$

$$B = \begin{pmatrix} 0.0 & 0.0137 & 0.9564 & 0.0 & 0.0 & 0.0272 & 0.0027 \\ 0.1984 & 0.1786 & 0.1485 & 0.0100 & 0.0725 & 0.0813 & 0.3112 \end{pmatrix} \tag{6.2}$$

$$\pi = (0.0, 1.0) \tag{6.3}$$

Reads/bin	Writes/bin
Raw Mean: 149.505	Raw Mean: 0.731
SlidHMM Mean: 150.677	SlidHMM Mean: 0.764
Raw Std Dev: 277.434	Raw Std Dev: 0.478
SlidHMM Std Dev: 283.318	SlidHMM Std Dev: 0.492

Figure 6.4: Statistics for raw and SlidHMM Flash Memory traces for fully computed β values

As we can see from the table in Figure 6.4, the HMM averages are very good approximations for the raw Flash Memory trace. Despite receiving higher accuracy by calculating the β values all over again, it added more recurrence in our algorithm and therefore made it computationally more expensive. Nonetheless, it is still an improved version of the incremental Baum-welch algorithm as suggested by Stenger et al. in [23] which loses significant accuracy of the approximated β values for the new observation set (as explained in chapter 4).

6.2 Hospital Model

6.2.1 Size of bins

For our Hospital model, we focused on hourly intervals when counting patient arrivals. Logically, this made sense because choosing 30 minute intervals left too many intervals empty (increasing the number of data points belonging to the empty cluster). Also choosing less frequent intervals, for example 2 hour intervals, held too many arrivals and mixed busy times with sparse times. For example, in one hour, no patients arrived, then in the next hour, four patients arrived. With a 2 hour interval, four patients would seem quite an average outcome (same as two patients per hour), and thus we would lose the important trend of the sparse hour and the busy hour. We also felt that 2 hours was too long a time to obtain sufficient empty intervals.

6.2.2 Number of Clusters

From our statistical diagrams in chapter 5, we gathered that our trace of patient arrivals has about 5 distinct groups or clusters at most. When first attempting to use 4 clusters for our K-means clustering algorithm, we obtained the following observation values:

$$\begin{pmatrix} 1.0 \\ 4.99 \\ 2.4 \\ 0.0 \end{pmatrix} \tag{6.4}$$

As we can see, there is now a cluster with a zero centroid. This observation set mimics the different sections of diagram 5.2 where at least one third of all arrival intervals are empty. Two clusters (with centroids 1.0 and 2.4) represent the low and medium arrivals per interval, respectively. Finally, the last cluster (with centroid 4.99) is most likely representing intervals with more than 3 arrivals.

When we ran the Baum-Welch algorithm for this observation set, the following results were produced:

Patient Arrivals per bin
Raw Mean: 1.483
HMM Mean: 1.539
Raw Standard Deviation: 1.611
HMM Standard Deviation: 1.687

Figure 6.5: Statistics for raw and HMM patient arrival traces with 4 clusters

We have good results when we compare the HMM-generated statistics to the raw ones. However, these are not as accurate as our results obtained in chapter 5 for 3 clusters. Above in Figure 6.5, we see the HMM mean is somewhat higher than expected, possibly due to an overestimation of frequent arrivals by our model.

Attempting the same process with 5 clusters, we obtained the following observation set:

$$\begin{pmatrix} 1.0 \\ 4.99 \\ 0.0 \\ 0.0 \\ 2.4 \end{pmatrix} \tag{6.5}$$

We immediately notice that two clusters have the same centroid values (0.0). This is an indication that we have set too many clusters for our possible range of observations, which in this case is 0 to 9 arrivals per hour. Therefore, our hospital HMM with 3 clusters seems to be the most efficient out of all our tried models.

6.3 Day and Night as hidden states

Based on our Viterbi algorithm results in chapter 5, we decided to label our two hidden states as ”day” and ”night.” As mentioned in that chapter, the number of arrivals categorized as day and night in the raw trace was compared to the sequence of hidden states produced by the Viterbi algorithm. The results of the comparison were very good, which proved that the day and night approximation was well made.

However, this is not the only possibility for the given arrival data. We could also label our hidden states as ”busy” and ”idle” referring to many arrivals or very few arrivals, respectively. One method of deciding the best label for the hidden states is to analyse trends in the Viterbi algorithm. The following is a segment of 40 states taken from a total sequence of 1000 states:

$$0000000000000011111111111110000000000000 \tag{6.6}$$

We can split the sequence in 6.6 into three distinct groups of lengths 14, 13 and 13, respectively. One can now notice that each sequence is roughly the same length as half a day (i.e. 12 hours) because each entry represents an interval of one hour. Then, our interpretation of sequence 6.6 is now as follows: we are in state 0 for 14 hours, then we are in state 1 for 13 hours and finally we stay in state 0 for 13 hours. If we treat state 0 as ”day” and state 1 as ”night”, our earlier interpretation is translated to: we observe day for 14 hours, then move into night for 13 hours, then back into day for 13 hours. This is a realistic description given that the hospital arrivals were observed continuously over days and nights.

If we labelled state 0 as ”busy” and state 1 as ”idle”, then the sequence in 6.6 would not be a realistic segment of state transitions. For example, over the first 14 hours of observations we are in the busy state. Then, we notice a *cyclic* busy period that lasts about 13 hours and reappears about every 13 hours. This is unrealistic because in any hospital the day brings both busy periods and idle periods, just as the night is expected to have some patients arrive at random times.

Let us now analyse our Viterbi sequence later down the timeline. We notice an interesting segment made up of the following sequence of states:

$$1111110001111111100000000 \tag{6.7}$$

This segment in 6.7 lasts a total of 25 hours, but has four distinct periods of states: we are in state 1 for 6 hours, then move to state 0 for 3 hours; after that we go back to state 1 for 8 hours and finally we move to state 0 for 8 hours. It seems that labelling the hidden states as day and night would not work here very well. However, using the busy and idle labels we can make more sense of this sequence of states. Indeed, the first 9 hours are made up of 6 busy periods then 3 idle periods, which can represent a day that is initially busy then becomes less busy. The next 8 hours are all busy so can represent a day and finally the last 8 hours are idle, so maybe can represent night.

Therefore, as a whole, our initial interpretation of day and night has become somewhat unclear when analysing sequence 6.7. Despite this different result in our

state sequence, we can still rely on the accurate results presented in chapter 5 for the Viterbi sequence of length 4800. The entire sequence was approximately divided in two parts, one for each state. Nonetheless, further work could be done on more observations to explore which interpretation best matches our Viterbi sequence. Perhaps the result will be a mixture of labelling as shown in sequence 6.7, or we need to add more states. For example, we could now have four mixed states: busy day, idle day, busy night and idle night.

6.4 Comparisons with existing work

In the domain of HMMs in Flash Memory, the inspiration for the Flash workload model was taken from my supervisor Peter Harrison and papers such as [1], which help set the tone for constructing the HMM. Currently, a wide amount of research is being done on applying HMMs to Flash Memory data, as HMMs are becoming a popular method of building portable benchmarks. HMMs are just as accepted as other time series analysis methods (e.g. Box-Jenkins [24]) and have the advantage of using its hidden states to represent "mode switching." Indeed, further work has been done on Markov modulated fluid processes, where a continuous version of the Baum-welch algorithm was derived for two hidden states (see [26]).

The novelty of our Flash HMM lies in the derivation of an approximate *sliding verison* of the Baum-Welch algorithm. The recurrence equation for the β values in our sliding HMM is an improvement from the simple approximation presented by Stenger et al. in [23]. The sliding HMM has not been attempted elsewhere in the same way as we have presented in this project. It certainly adds to the potential of the incremental HMM obtained by Hansen et al. in [22]. The constant slide of input keeps up with the changing observation set as time goes on and is ideal for discovering trends in time series. We obtain satisfying results for a two state sliding HMM given our observation set, bearing in mind our read-dominated trace had limited capacity for analysis of trends for the write entries.

For our Hospital Arrivals model, we follow a similar approach based on the Flash workload model, but focus more heavily on the Viterbi algorithm. Given the correct MAP parameters, as validated by the techniques seen in chapter 5, we focus on the trends highlighted by the Viterbi algorithm when outputting its sequence of hidden states. By simply counting the number of occurences of each state in the sequence and assigning each state to a category, we can make a judgement of the meaning of the hidden states. The power, yet simplicity, of this process can lead to vast amounts of decoding about the time series which we want answers from. Other applications of the Viterbi algorithm seen in industry include speech recognition ([8]), biology([19]), etc. These papers have tried similar decoding techniques using the Viterbi algorithm as seen in this project, but for different applications.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The main aims we set at the beginning of this project were to apply HMMs to Flash Memory data and hospital patient arrivals to correctly analyse discrete time series, give meaning to the hidden states and help recreate representative traces.

To achieve this, we built portable benchmarks through our HMMs by firstly creating a binned trace of the raw time series. Through the use of the K-means clustering algorithm, we then constructed an observation trace which helped generate the MAP parameters using the Baum-Welch algorithm. Finally, the Baum-Welch algorithm generates a sequence of observations comparable with the original binned trace. The results we achieved in chapters 4 and 5 were very good and therefore the aims of this project were met.

An approximation of a sliding version of the Baum-Welch algorithm was achieved and produced meaningful results. For such a large observation set that was inputted as a binned trace, the accuracy of the HMM-generated mean and standard deviation compared to those of the raw trace was excellent. The power of this sliding HMM can now be used to save computation time in the Baum-Welch algorithm and more importantly, provide analysis for processes which change behaviour over discrete time.

For the Hospital HMM, we focused on the use of the Viterbi algorithm to reveal the identity of the hidden states and found that ”day” and ”night” provided an accurate description. This supported our efficient set up of the HMM with optimal parameters helping to generate a realistic trace of patient arrivals as observed over many days.

In conclusion, we showed that HMMs can be applied effectively to Flash Memory data and hospital patient arrivals to facilitate run time analysis and planning. Nonetheless, more can be achieved with these models and we present this as possible extensions below.

7.2 Future Work

7.2.1 New Raw Trace

Given more time, we would test our Flash workload model on a new set of observation values (i.e. use a different raw trace). More specifically, we would aim to use a raw trace with more frequent writes. This would then give us a set of cluster centroid pairs which have increasing reads and increasing writes. Since the I/O trace that was inputted into our Flash workload model was very read dominated, our cluster centroid pairs had very small write values. Thus we could not effectively identify the trends of the behaviour of the writes, compared to the reads. Using a more balanced I/O trace would give us different entries for our emission probability matrix as these reflect the probability of getting an observation from a cluster for each state.

A possibility is to use a different I/O trace from [25] to create another Flash workload model. Then we could compare the HMM-generated traces produced by these two different models with their respective raw traces.

7.2.2 Different Clustering Algorithm

One extension can be done on creating our binned traces for each model with a different clustering algorithm and then compare with existing results. The DBSCAN algorithm has the advantage of being able handle noise and, unlike K-means, it does not need the number of clusters as an input. If we use this density-based algorithm to try and improve the results in this project, we must take care in implementing the *getNeighbours* function, which includes the distance-measure for the algorithm. Saying this, because we are clustering points with at most two dimensions (i.e. reads and writes) this will not be a major obstacle for implementing and executing the DBSCAN algorithm.

7.2.3 Sliding HMM with new β values

With our sliding version of the HMM, we approximated the unknown β values for the new observations by using the backward algorithm recurrence formula. Another possibility for approximating the β values was to use a similar technique from the 2005 paper by Hansen et al. ([22]). In this paper, the authors assumed that decay functions (i.e. $\omega(T-t, j)$) could be used to approximate the β values in such a way that for large sequence, these decay functions are equal. Therefore, when calculating γ the decay functions cancel out in the fraction to give:

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

for all values of t which form the new observations $O_t, O_{t+1}, \ldots$

We know that be definition, γ can be written as

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

Therefore, we have the equation which calculates $\beta_t(i)$ for the new values of t:

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

As stated by Hansen et al. this would make the Backward algorithm a *fixed-lag smoothing* algorithm. Also, it seems that this would a better approximation for the β values than assuming that $\beta_t(i) = \beta_{t+1}(i) = \cdots = \beta_T(i) = 1$. Therefore we can modify our Sliding HMM to fit this β approximation and compare results with our original Sliding model for convergence and for similarity of parameters.

7.2.4 Sliding HMM for continuous time

Our sliding HMM works for discrete time series and therefore the next step would be to derive a sliding HMM for continuous time. This would require a continuous version of the Baum-Welch algorithm, as seen in [26], which would then behave as a sliding algorithm for a continuous time series. The intervals would not be observation points as in the discrete case, but rather fixed time intervals which slide along the continuous time series. Therefore, a degree of accuracy is needed in choosing the time intervals for the sliding HMM, perhaps to the nearest millisecond of the timestamp to represent each boundary value, for example.

7.2.5 Merging version of the Baum-Welch algorithm

As mentioned in chapter 5, an optimal way of choosing hidden states in a HMM is the *bottom up* technique. This approach starts off with many states and merges similar states together until the optimal number is achieved. Ideally, an extension to this project would be to derive a systematic approach for this method which works in any given scenario (e.g. Flash Memory, patient arrivals, etc.) and produces the optimal number of hidden states in that scenario. This would involve analysis of the emission probability matrix as a starting point to identify the effect of each state on the observation set. Then, a modification of the Baum-Welch algorithm needs to be executed to merge the chosen states whilst keeping any information of transition probabilities in the process. This would give us an approximation to the transition probability matrix for optimal states. Finally, we can complete the Baum-Welch algorithm by using the new transition matrix to construct the new emission proability matrix and calculate the initial probability distribution.

Although our merging technique as described above, seems computationally expensive, it is advantageous to achieving an HMM with optimal states when contextual information of the time series is not given. The power of this technique lies in the identification of the ”most efficient set up” just by using the original HMM parameters of a less efficient set up. Therefore, this technique is entirely ”self-contained” in the sense that the computations are limited within the bounds of the Baum-Welch algorithm.

Bibliography

[1] Harrison, P. G., Leyton, S. K., Patel N. M., Zertal, S. *Storage Workload Modelling by Hidden Markov Models: Application to FLASH Memory*, In: *Biometrika*, 2010

[2] MacDonald I. L., Zucchini, W., *Hidden Markov and Other Models for Discrete-valued Time Series*, Chapman and Hall, UK, 1997

[3] Ashraf, J., Iqbal, N., Khattak, N. S., Zaidi, A. M., *Speaker Independent Urdu Speech Recognition Using HMM*, 2010.

[4] Beyreuther, Moritz, Carniel, Roberto, Wassermann, Joachim, *Automatic earthquake detection and classification with Continuous Hidden Markov Models: a possbiel tool for monitoring Las Canadas caldera in Tenerife*, In: *IOP Conf. Series:* Earth and Environmental Science *3*, 2008

[5] Rabiner, L. R., Juang, B. H., *An Introduction to Hidden Markov Models*. In *IEEE ASSP Magazine*, **3**, pp. 4-16, January, 1986

[6] Baum, L. E., Eagon, J. A., *An Inequality with Applications to Statistical Estimation for Probabilistic Functions of a Markov Process and to a Model for Ecology*, In *Bulletin of the American Mathematical Society*, **73**, pp. 360-3, 1967.

[7] Baum, L. E., Petrie, T, *Stastical Inference for Probabilistic Functions of Finite Markov Chains*. In *The Annals of Mathematical Statistics*, **37**, pp. 1554-63, 1966.

[8] Rabiner, L. R., *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*. In *Proceedings of the IEEE*, **77**, pp. 257-86, 1989

[9] Durin, R., et al., *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. In *Cambridge: Cambridge University Press*, 1999

[10] Stamp, Mark, *A Revealing Introduction to Hidden Markov Models*, January 18, 2004

[11] Sewell, Martin, *Hidden Markov Models*, Department of Computer Science, UCL, August, 2008

[12] Harrison, P. G., Patel, N. M., Zertal, S, *Response Time Distribution of Flash Memory Accesses*, In *Valuetools*, 2008

[13] Zhai, C. X., *A Brief Note on the Hidden Markov Models (HMMs)*, Department of Computer Science, University of Illinois at Urbana-Champaign, IL, USA, March, 2003

[14] Zhang, L., Chan, K. P., *Bigram HMM with Context Distribution Clustering for Unsupervised Chinese Part-of-Speech tagging*, Department of Computer Science, Hong Kong, 2010

[15] Brand, Matthew, *An entropic estimator for structure discovery.*, In *Proceedings of the 1998 conference on Advances in neural information processing systems II*, pp. 723-29, Cambridge, MA, USA, MIT Press, 1999

[16] Zhang, T., Ramakrishnan, R., Livny, M., *BIRCH: An Efficient Data Clustering Method for Very Large Databases*, Computer Science Department, University of Wisconsin-Madison, USA, 1996

[17] MacQueen, J. B., *Some Methods for classification and Analysis of Multivariate Observations, Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability*, pp. 281-297, Berkeley, University of California Press, USA, 1967

[18] Krough, A., Brown, M., Mian, S., Sjolander, K., Haussler, D., *Hidden Markov Models in Computational Biology*, pp. 1501-31, Journal of Molecular Biology, Computer and Information Sciences, University of California, Santa Cruz, USA, 1994

[19] Burge, C., Karlin, S., *Prediction of cmplete gene structures in human genomic DNA*, pp. 78-94, Journal of Molecular Biology, Computer and Information Sciences, Stanford University, California, USA, April 1997

[20] Ester, M., Kriegel, H. P., Sander, J., Xu, X., *A density-basaed algorithm for discovering clusters in large spatial databases with noise*, pp. 226-31, Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), AAAI Press, 1996

[21] Kaufman, L., Rousseeuw, P. J., *Finding groups in data: An introduction to cluster analysis*, John Wiley and Sons, Inc., New York, USA, 1990

[22] Florez-Larrahondo, G., Bridges, S., Hansen, E. A., *Incremental Estimation of Discrete Hidden Markov Models on a New Backward Procedure*, Department of Computer Science and Engineering, Mississippi State University, Mississippi, USA, 2005

[23] Stenger, B., Ramesh, V., Paragois, N., Coetzee, F., Buhmann, J. M., *Topology free Hidden Markov Models: Application to background modeling*, pp. 297-301, Proceedings of the International Conference on Computer Vision, 2001

[24] Box G. E. P., Jenkins, G. M., *Time series analysis: forecasting and control*, pp 575, Holden-Day, San Francisco, CA, 1976

[25] Leung, A. W., Pasupathy S., Goodson, G., Miller, E. L., *Measurement and Analysis of Large-Scale File Network File System Workloads*, In *Proceedings of the 2008 USENIX Annual Technical Conference*, Boston, MA, June 2008

[26] Zraiaa, M., *Hidden Markov Models: A Continuous-Time Version of the Baum-Welch Algorithm*, Department of Computing, Imperial College London, London, September 2010