UNLV | UNIVERSITY LIBRARIES

UNLV Theses, Dissertations, Professional Papers, and Capstones

5-2011

Implementation of numerically stable hidden Markov model

Usha Ramya Tatavarty
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/thesesdissertations

Part of the Statistics and Probability Commons, and the Theory and Algorithms Commons

Repository Citation

Tatavarty, Usha Ramya, "Implementation of numerically stable hidden Markov model" (2011). *UNLV Theses, Dissertations, Professional Papers, and Capstones*. 1018.
http://dx.doi.org/10.34917/2362226

This Thesis is protected by copyright and/or related rights. It has been brought to you by Digital Scholarship@UNLV with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in UNLV Theses, Dissertations, Professional Papers, and Capstones by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

IMPLEMENTATION OF NUMERICALLY STABLE HIDDEN MARKOV MODEL

by

Usha Ramya Tatavarty

Bachelor of Technology in Computer Science and Engineering
Jawaharlal Nehru Technological University, India
May 2009

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
May 2011

Copyright by Usha Ramya Tatavarty 2011
All Rights Reserved

The Graduate College

We recommend the thesis prepared under our supervision by

Usha Ramya Tatavarty

entitled

Implementation of Numerically Stable Hidden Markov Model

be accepted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
School of Computer Science

Kazem Taghva, Committee Chair

Ajoy K. Datta, Committee Member

Laxmi P. Gewali, Committee Member

Venkatesan Muthukumar, Graduate Faculty Representative

Ronald Smith, Ph. D., Vice President for Research and Graduate Studies
and Dean of the Graduate College

May 2011

ABSTRACT

Implementation of Numerically Stable Hidden Markov Model

by

Usha Ramya Tatavarty

Dr. Kazem Taghva, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

A Hidden Markov model (HMM) is a statistical Markov model in which the system being modeled is assumed to be a Markov process with unobserved (hidden) states. HMM is an extremely flexible tool and has been successfully applied to a wide variety of stochastic modeling tasks. One of the first applications of HMM is speech recognition. Later they came to be known for their applicability in handwriting recognition, part-of-speech tagging and bio-informatics.

In this thesis, we will explain the mathematics involved in HMMs and how to efficiently perform HMM computations using dynamic programming (DP) which makes it easy to implement HMM. We will also address the practical issues associated with the use of HMM like numerical scaling of conditional probabilities to model long sequences and smoothing of poor probability estimates caused by sparse training data.

ACKNOWLEDGEMENTS

I would like to express my gratitude to many people who helped and directed me to pursue my goal of getting an education in USA. Firstly, I would like to thank the faculty and staff of School of Computer Science, University of Nevada Las Vegas. I cannot thank enough to Dr. Kazem Taghva for being more than a mentor and an advisor to me. It is only because of his support and guidance I am able to finish my research work. I sincerely thank our graduate coordinator, Dr. Ajoy K Datta, for his help and invaluable support throughout my masters program. I also would like to thank other members of my committee, Dr. Laxmi P. Gewali and Dr. Venkatesan Muthukumar.

I would also like to extend my appreciation towards my parents, and my sisters, for being there for me through thick and thin and always encouraging me to strive for the best. Without their endless support, I would never be able to reach the place I am standing today in my life. Last but not the least; I thank my friends for their support in successful completion of this work.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGEMENTS iv

LIST OF FIGURES vii

CHAPTER 1 INTRODUCTION 1
1.1 Thesis Overview 2
1.2 Thesis Structure 3

CHAPTER 2 BACKGROUND 4
2. 1 Statistical Model 4
2.2 Markov Models 5
2.2.1 Markov Chain 6
2.2.2 Discrete Markov Model 9
2.2.3 First –order Markov Model 9
2.3 Discrete Markov Model Examples 10
2.3.1 Example 1 Single Fair Coin Tossing Experiment 11
2.3.2 Example 2: Stock Market Index 13
2.4 Extension to HMMs 15

CHAPTER 3 HIDDEN MARKOV MODELS 18
3.1 Elements of an HMM 18
3.2 Canonical Problems of HMM 21
3.2.1 Evaluation 24
3.2.1.1 Forward Algorithm 26
3.2.1.2 Backward Algorithm 28
3.2.2 Decoding 30
3.2.2.1 Viterbi Algorithm 30
3.2.3 Training 32
3.2.3.1 Supervised Learning 33
3.2.3.2 Unsupervised learning 34
3.3 HMM Examples 37
3.3.1 Example 1: Coin Tossing Experiment 37
3.3.1 Example 2: Stock Market Index 38

CHAPTER 4 IMPLEMENTATION OF HMM 40
4.1 Representation of HMM Model 40
4.1.1 HMM Representation on the disk file 41
4.3 Decoding 46
4.4 Training 48
4.4.1 Supervised Training 48
4.4.2 Unsupervised Training 50

CHAPTER 5 IMPLEMENTATION ISSUES OF HMM 54
5.1 Scaling... 54
5.1.1Viterbi underflow 54
5.1.2 Forward algorithm underflow 55
5.1.2.1 Normalized Forward Algorithm 56
5.2 Smoothing 57

CHAPTER 6 CONCLUSION AND FUTURE WORK 59

BIBLIOGRAPHY 60

VITA 61

LIST OF FIGURES

Figure 1 A Markov Chain with five states and state transitions 7
Figure 2 Observation Sequence for coin tossing experiment 11
Figure 3 Single Fair Coin tossing experiment-Markov Model.............. 12
Figure 4 Stock Market Index Markov Model 14
Figure 5 Operations for computing the forward variable α_j (t + 1) 26
Figure 6 Computing $\alpha_j(t)$ 27
Figure 7 Computing $\beta t(i)$.. 29
Figure 8 Operations for computing $\xi t(i,j)$ 35
Figure 9 HMM-Coin Tossing Experiment.. 38
Figure 10 Hidden Markov Model-Stock Market Index 38
Figure 11 Screenshot of the Stock Market Index HMM Model file. 42
Figure 12 Screen shot of an observation sequence file.......................... 43
Figure 13 Forward Algorithm-Initialization code snippet 45
Figure 14 code snippet to compute alpha values 45
Figure 15 Viterbi Algorithm-Initialization-code snippet......................... 46
Figure 16 Viterbi Algorithm Compute Step.. 47
Figure 17 Viterbi Algorithm-termination code snippet 47
Figure 18 screen shot of tagged sequence file...................................... 48
Figure 19 screen shot of Counting in MLE 49
Figure 20 code snippet for UpdateParameters() of MLE........................ 50
Figure 21 cone snippet of function train-BW Algorithm 51
Figure 22 code snippet of UpdateHMM function.................................. 52
Figure 23 source code to calculate scaling coefficients 56
Figure 24 source code to compute normalized alpha values 56
Figure 25 Step 2 of Normalized Forward Algorithm 57
Figure 26 Smoothing in HMM 58

CHAPTER 1

INTRODUCTION

A Hidden Markov Model (HMM) is simply a Markov Model in which the states are hidden. Hidden Markov Models (HMMs) are powerful statistical models for modeling sequential or time-series data. Hidden Markov Models were first introduced in a series of statistical papers by Leonard E. Baum and others in the late 1960s. Andrei Markov gave his name to the mathematical theory of Markov processes in the early twentieth century, but it was Baum and his colleagues that developed the theory of HMMs. One of the first applications of HMMs was speech recognition. Later they have been successfully used in many tasks such as computational sequence analysis, robot control, and information extraction. Hidden Markov modeling has become popular as it works very well in practice for several important applications when applied properly. Also it is very rich in mathematical structure and hence can provide a theoretical basis to a wide range of applications. In this thesis we attempt to understand the theoretical aspects of this type of statistical modeling.

Real-world processes usually produce observable outputs which can be characterized as signals [1]. These signals can be characterized in terms of signal models and with a good signal model we can stimulate the source that generated the signal. Signal Models can be broadly categorized into deterministic models and statistical models. In

deterministic model the specification of the signal model is normally straightforward. We need to determine values of the parameters of the signal model. In statistical models we characterize only the statistical properties of the signal. The signal is first illustrated as a parametric random process, and then the parameters of the stochastic process are estimated in a precise, well-defined manner. Some examples of statistical models include Gaussian processes, Markov processes and Hidden Markov processes, among others. The statistical model that is of interest to us is hidden Markov process, to be more specific discrete Hidden Markov Models.

1.1 Thesis Overview

Hidden Markov Model (HMM) is a finite state model that describes a probability distribution over an infinite number of possible sequences. It is a statistical Markov model in which the system being modeled is assumed to be a Markov process with unobserved (hidden) states [2]. The "hidden" in Hidden Markov Models comes from the fact that the observer does not know in which state the system may be in, but has only a probabilistic insight on where it should be [3].

In this thesis we will explain the mathematics involved in HMMs and how to perform efficient HMM computations using dynamic programming (DP). However, before going to the theoretical aspects of Hidden Markov Models we will first understand the theory behind Markov chains using some simple examples. The basic idea is to

characterize the theoretical aspects of Hidden Markov Model in terms of solving three fundamental problems. So we focus our attention on the three fundamental problems for HMM design, namely: the Forward and Backward algorithm for evaluating the likelihood of a sequence of observation given a specific HMM; Viterbi Algorithm to find the most likely explanation of a sequence; and Baum-Welch Algorithm and Maximum Likelihood Estimation (MLE) for training an HMM given sequence of observations. There are two practical issues that are associated with the implementation of Hidden Markov Models. We will also address those issues and solve them using numerical scaling and smoothing techniques.

1.2 Thesis Structure

This thesis is organized into different chapters starting from introduction in chapter 1 followed by a brief description about Markov Chains in Chapter 2. Then we extend the idea to the class of Hidden Markov Models in Chapter 3 using simple examples. Chapter 4 presents the implementation of theoretical aspects of HMM discussed in the previous chapter. The issues that arise during the implementation of HMM are addressed in the chapter 5. Chapter 6 concludes the thesis by giving a brief description about future proceedings.

CHAPTER 2

BACKGROUND

Some data mining techniques such as clustering assume that each data point in an observed input data is statistically independent from the observation (data point) that preceded it. But we often encounter sequences of observations, where each observation may depend on the observations which preceded it. One example that can explain this situation is a sequence of phonemes (fundamental sounds) in speech during the process of speech recognition [6]. In order to model such processes, we can use Hidden Markov Models.

Hidden Markov Model is one of the most important machine learning models in Information Extraction and Retrieval. Earlier, we have defined Hidden Markov Model (HMM) as a statistical model where the system being modeled is assumed to be a Markov process with unknown parameters, and the challenge is to determine the hidden parameters, from the observable parameters, based on this assumption [5]. To have a better understanding of what an HMM is, we will first focus our attention on what is meant by statistical models and Markov models and then on the concept of Hidden in Hidden Markov Models.

2. 1 Statistical Model

A statistical model is a formalization of relationships between variables in the form of mathematical equations. A statistical model describes how one or more random variables are related to each other. The model is

statistical as the variables are not deterministically but stochastically related. Stochastic means random. In stochastic or a random process instead of dealing with only one possible reality of how the process might evolve under time, there is some indeterminacy in its future evolution which is described by probability distributions [2].

In mathematical terms, a statistical model is frequently thought of as a pair (Y, P) where Y is the set of possible observations and P the set of possible probability distributions on Y. It is assumed that there is a distinct element of P which generates the observed data. Statistical inference enables us to make statements about which element(s) of this set are likely to be true [2].

Consider a simplest possible case of discrete time intervals. A stochastic process in this case amounts to a sequence of random variables known as a time series. A good example for this is Markov Chains.

2.2 Markov Models

To define Hidden Markov Model (HMM) properly, we need to first introduce the concept of Markov Chain, also referred to as an observed Markov Model. Markov chains and Hidden Markov models are both extensions of the finite automata. A finite automaton is defined by a set of states and a set of transitions between states. A weighted finite-state automaton is a simple augmentation of the finite automaton in which each arc is associated with a probability, indicating how likely that path

is to be taken. The probability on all the arcs leaving a node must sum to one [4].

2.2.1 Markov Chain

A Markov Chain is a particular case of a weighted automaton in which the input sequence uniquely determines the states through which the automaton traverses. Markov chains are sequences of random variables in which the future variable is determined by the present variable but is independent of the way in which the present state arose from its predecessors [8]. An important point to consider is that Markov chains can only assign probabilities to unambiguous sequences; it cannot represent problems that are inherently ambiguous.

We will now try to define Markov chains as probabilistic graphical models which are a way of representing probabilistic assumptions in a graph [4].

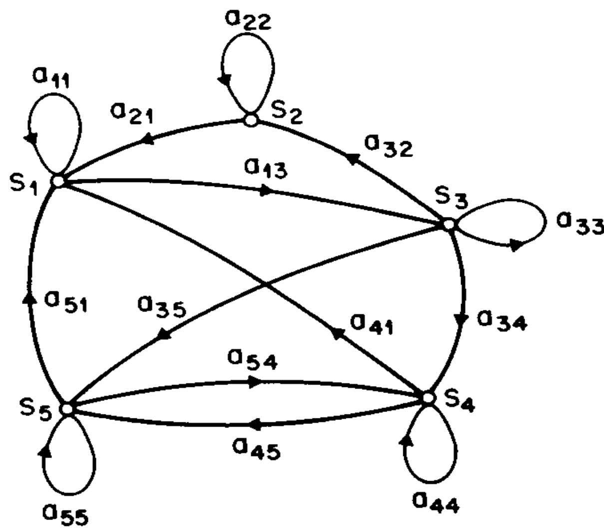

Figure 1 A Markov Chain with five states and state transitions

A Markov chain is specified by the following components:

State:

In Markov Model, at any time t, we consider the system to be in one of a set of N distinct states, S_1, S_2, S_3, …, S_n and we denote this distinct state occupied at state t as q_t.

Notation

A set of N states S= $\{ S_1, S_2, S_3, \ldots, S_{n-1}, S_n \}$

We can denote a sequence of successive states of length T as Q
Q = (q1, q2 , …, qt)

In a Markov model, we know what states the machine is passing through, so the state sequence or some deterministic function of it can be regarded as the output.

We will model the production of such a sequence using transition probabilities

Transition Probabilities:

A transition Probability denoted by aij is the probability that the system will be in state Sj at time t+1 given that it was in state Si at time t

Notation

$a_{ij} \equiv P(q_{t+1} = S_j / q_t = S_i)$ $a_{ij} \geq 0$

a_{11}, a_{12}, …, a_{nn} are the Transition Probabilities. All the transition probabilities together can be represented by a Transition Probability Matrix A.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & . & . & a_{1n} \\ a_{21} & a_{22} & a_{23} & . & . & a_{2n} \\ a_{31} & a_{32} & a_{33} & . & . & a_{3n} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ a_{n1} & a_{n2} & a_{n3} & . & . & a_{nn} \end{bmatrix}$$

In transition probability matrix A, each aij represents the probability of moving from state i to state j

$$\sum_{j=1}^{n} a_{ij} = 1 \text{ for all i}$$

As each a_{ij} represents the probability P (S_j/S_i), the laws of probability require that the values of the outgoing arcs from a given state must sum to one.

Initial Probability Distribution (π):

An initial probability denoted by Π_i is the probability that the Markov chain will start in state i. Some states j may have Π_j=0, meaning that they cannot be initial states.

Notation

$$\pi_i \equiv P(q_1 = S_i)$$

$\Pi_1, \Pi_2, \ldots, \Pi_n$ is the initial probability distribution over states

As each πi represents the probability of S_i being the start state, all the π must sum to one

$$\sum_{i}^{n} \Pi i = 1$$

A model of states and transition probabilities, such as the one we have just described, is called a Markov model. The above stochastic process could be called an observable Markov Model since the output of the process is the set of states at each instant of time, where each state corresponds to an observable event [1].

2.2.2 Discrete Markov Model

Discrete Markov chains model observation sequences which consist of symbols drawn from a discrete and a finite set of size N. This set of discrete observations is often referred to as a codebook.

The above Markov model considers the observations to be discrete; it has states that are distinct from one another. As the observations of a Markov Model are characterized as discrete symbols chosen from a fine alphabet it is a discrete Markov Model.

2.2.3 First –order Markov Model

A Markov Chain that we have defined above embodies an important assumption about transition probabilities. We assumed that transition

probabilities depend only on the previous state which makes it a first-order Markov model. In a first order Markov chain, the probability of a particular state depends only on the previous state. This assumption is called a Markov assumption.

Markov Assumption:

The probability of a certain observation at time n only depends on the observation q_{n-1} at time n-1

Markov Assumption:

$$P (q_t / q_1 \ldots q_{t-1}) = P (q_t / q_{t-1})$$

This is called first order Markov Assumption. A second order Markov assumption would have the probability of an observation at time n depend on q_{n-1} and q_{n-2}. Higher order Markov models are also possible but the model that is of concern to us is first order. In general when people talk about Markov assumption they usually mean the first-order Markov assumption [7].

2.3 Discrete Markov Model Examples

Markov models are used to model sequences of events (or observations) that occur one after another. These sequences of events can either be deterministic or non-deterministic. Deterministic Markov Models where one specific observation always follows another are easy to model. One good example to represent deterministic Markov Model is changes in traffic lights. Non-deterministic models are the ones where an event might be followed by one of several subsequent events, each with a

different probability. Some real time processes that come under non-deterministic Markov Models are daily changes in the weather, sequences of words and sequences of phonemes in spoken words [10].

To illustrate the concept of Markov chains we will consider an example of tossing coins. This example can later be extended to understand Hidden Markov Models.

2.3.1 Example 1 Single Fair Coin Tossing Experiment

Let us consider the following scenario. Assume that we are placed in a room which is divided into two sections with a curtain. Imagine that we are on one side of the curtain and there is a person on the other side of the curtain. The person on the other side has a single fair (un-biased) coin with him which he tosses to produce an observation. The person tosses the coin and tells us the outcome (H, T), after each trial. He does this several times and the outcome obtained after each trial is recorded as an observation.

This is how the observation sequence would look like

THTHHHTTTTHHHHHTHHTTHHTTHHTHHHHHHHHTTHTTHHHH

THTTTHHTHTTHHHHTHTHHTTHTHTTHHTHTHHHTHHTHT

Figure 2 Observation Sequence for coin tossing experiment

This sequence of Heads (H) and Tails (T) can be modeled as a Markov Chain. The two possible outcomes of each trial in the coin

tossing experiment, heads (H) and Tails (T) are represented as the two states of the Markov Model. In fact, we may describe the system with a deterministic model where the states are the actual observations [8]. These states can transition to themselves and as the experiment uses a fair coin, the transition probabilities are equally distributed. Here it is obvious that there is no concept of hidden as it is already known that a single fair coin is tossed every time which implies that the visible states correspond to the internal states.

Here is the graphical representation of the single fair coin tossing experiment.

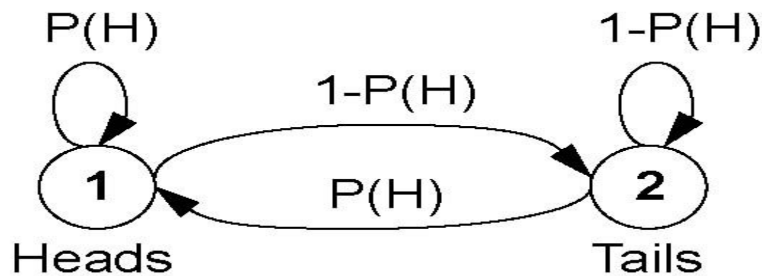

Figure 3 Single Fair Coin tossing experiment-Markov Model

States (S): Heads (1), Tails (2)

Transition Probabilities (A):

$$A = \begin{matrix} \text{Heads(1)} \\ \text{Tails (2)} \end{matrix} \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$$

Initial Probabilities (π):

$$\pi = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$$

O= H H H T T H…

Q = 1 1 1 2 2 1…

Observation sequence is O= H H H T T H… and the corresponding state sequence is Q = 1 1 1 2 2 1… .

It is obvious from the Markov Model that these are the states traversed to obtain the above observation sequence.

2.3.2 Example 2: Stock Market Index

Let us consider another example to get a better understanding of Markov Processes. Figure 4 depicts a simple example of Markov process. It describes a simple model for a stock market index. It has three stocks, Bull, Bear and Even, that represents states and three index observations up, down, unchanged that represent the variations of stock in model. We associated Bull to the variation Up, Bear to the variation Down and Even to the variation Unchanged.

Graphical Representation of Example 2

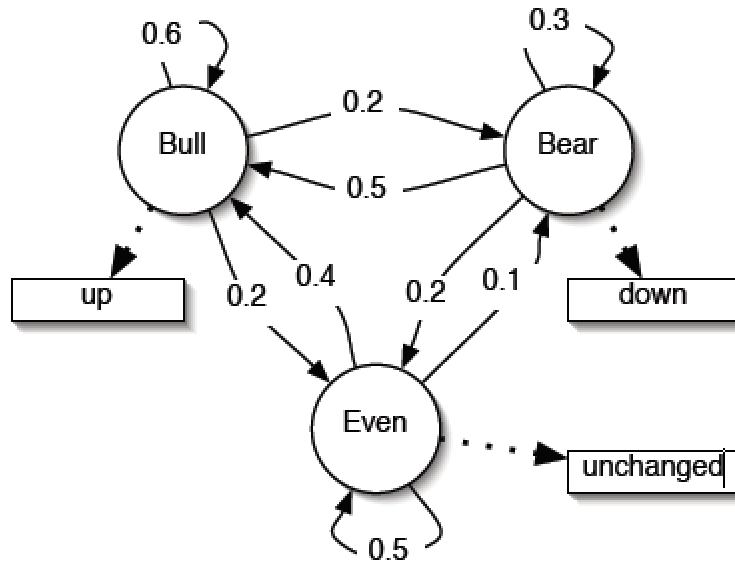

Figure 4 Stock Market Index Markov Model

Parameters of this Markov Model are

States:

Bull, Bear and Even

These are the internal states corresponding to the three stocks of the example

Transition Probabilities:

It is a finite state automaton, with probabilistic transitions between states.

$$
\begin{array}{cc} & \begin{array}{ccc} Bull & Bear & Even \end{array} \\ \mathrm{A} = \begin{array}{r} Bull \\ Bear \\ Even \end{array} & \begin{bmatrix} 0.6 & 0.2 & 02 \\ 0.5 & 0.3 & 0.2 \\ 0.4 & 0.1 & 0.5 \end{bmatrix} \end{array}
$$

Initial Probabilities (π):

$$\pi = \begin{matrix} Bull \\ Bear \\ Even \end{matrix} \begin{bmatrix} 0.33 \\ 0.33 \\ 0.33 \end{bmatrix}$$

Given a sequence of observations we can easily find the state sequence that produced these observations.

Observations: up-down-down-unchanged

States: Bull-Bear-Bear-Even

Probability for the above sequence can now be calculated

$P(O) = \pi_{Bull} * A_{Bull, Bear} * A_{Bear, Bear} * A_{Bear, Even}$

$= 0.33 * 0.2 * 0.3 * 0.2$

$= 0.00396$

2.4 Extension to HMMs

A Markov Chain is useful when we need to compute probability for a sequence of events that we can observe in the real world. In many cases however the events are not observable. We will see how the above mentioned examples can be improved to represent more realistic problems and how this improvisation leads to the concept of Hidden Markov Models.

In example 1 we considered a coin tossing experiment where there is a person on one side of the curtain who tosses a single fair coin. Here once an observation is made, the state of the system is trivially retrieved. But this model is too restrictive as it is limited to one un-biased coin. However this is not always the case. There can be a case where the person behind the curtain has several coins, some of them

being biased. This situation cannot be modeled using Markov Chains as there is a concept of hidden in this situation.

Now consider example 2 and how it is too restrictive to be of any practical use. The model presented in example 2 describes a simple model for a stock market index with three states and three observations. We assumed in our example that a bull market has only good days. Similarly, we assumed that Bear market always goes down and Even market remains Unchanged. Our goal here is to represent a Stock Market index and we are not doing a good job by restricting the stocks to a specific variation.

To make our example more expressive and realistic we have to consider the possibility of bull market not only going Up but also the possibility of it going down and remaining unchanged. Similar possibilities should also be considered for other stocks, Bear and Even. By including these prospects to the Model we are associating a state (Stocks) to all the observations (Variations in Price) as opposed to a state being indexed to a particular observation making in more realistic. By extending example 2 to include these new prospects we come up with Hidden Markov Models.

To conclude, in order to make a Markov Model more flexible, we assume that the outcomes or observations of the model are a probabilistic function of each state [8]. This makes the Markov Model a Hidden Markov Model. The "hidden" in Hidden Markov Models comes

from the fact that the observer does not know in which state the system may be in, but has only a probabilistic insight on where it should be.

CHAPTER 3

HIDDEN MARKOV MODELS

Hidden Markov Models (HMMs) separate the observations from the states; the observations (outputs) are visible, but the state sequences that led to them are hidden. It's "Markov" because the next state is determined solely from the current state. It is "Hidden" because the actual state sequences are hidden [10]. In this thesis we talk about discrete Hidden Markov Models. This type of HMM has discrete observation symbols. In this thesis the term HMM implies Discrete HMM. A set of five elements can be used to describe an HMM.

In the following section the elements of an HMM and their notation are defined.

3.1 Elements of an HMM

We define Q to be a fixed state sequence of length T, and corresponding observations O:

$Q = q1, q2, \cdots, qT$

$O = o1, o2, \cdots, oT$

T is the number of observations in the sequence

We can define an HMM as a 5-tuple (S, V, π, A, B)

HMM Notation: $\lambda = (A, B, \pi)$

- N: Number of states in the Model.

There are a finite set of states in a model. The states in a HMM are hidden but there is a lot of significance to these states in defining an HMM. We denote the individual states as $S_1, S_2, S_3, \ldots, S_n$.

$S = \{ S_1, S_2, S_3, \ldots, S_n \}$

- M: Number of distinct symbols observable in states.

  These symbols correspond to the observable output of the system that is being modeled. We denote the individual symbols as $v_1, v_2, v_3, \ldots, v_M$

  $V = \{ v_1, v_2, v_3, \ldots, v_M \}$

- A: State transition probability distribution

  A is transition array that store the state transition probabilities, $A=\{ a_{ij} \}$

  $a_{ij}= P(q_t=S_j/q_{t-1}=S_i)$, $i \geq 1$ and $j \geq N$

  aij, the probability of moving from state Si to Sj at time t

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & . & . & a_{1n} \\ a_{21} & a_{22} & a_{23} & . & . & a_{2n} \\ a_{31} & a_{32} & a_{33} & . & . & a_{3n} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ a_{n1} & a_{n2} & a_{n3} & . & . & a_{nn} \end{bmatrix}$$

At each time t, a new state is entered which depends on the transition probability distribution of the state at time t – 1. Transition to the same state is also possible. An important point about transition probabilities is that they are independent of time; the probability of moving from state Si to state Sj is independent of time t

- B: Observation symbol probability distribution

  $B = \{ b_j(k) \}$ is the output symbol array that stores the probability of an observation V_k being produced from the state j, independent of time t. Observation symbol probability or Output Emission Probability estimates are also independent of time; the probability of a state emitting a particular output symbol does not vary with time.

  $$B = \{ b_j(k) \} \ , \ b_i(k) = P(x_t = v_k / q_t = S_j) \quad 1 \le j \le N \text{ and } 1 \le k \le M$$

  $b_i(k)$, the probability of emitting symbol v_k when state Sj is entered at time t

  After each transition is made a symbol is outputted based on the output probability distribution which depends only on the current state.

- π: Initial state distribution

  $\pi = \{ \Pi_i \}$ is the initial probability array that stores the probability of the system starting at state i in an observation. It is the probability of state Si being the start state in an observation sequence.

  $\pi = \{ \Pi_i \}, \Pi_i = P(q_1 = S_i), \quad 1 \le i \le N$

  Π_i , the probability of being in state I at time t=1

A complete specification of an HMM consists of the above five elements, {S, V, π, A, B}. We usually use a compact notation $\lambda=(A, B, \pi)$ to represent the above complete parameter set of HMM.

The following are obvious constraints on the elements of an HMM.

$\sum_{I=1}^{n} \pi_i = 1$

$\sum_{j=1}^{n} a_{ij} = 1$ for all i

$\sum_{j=1}^{n} b_i (v_k) = 1$ for all i

As each a_{ij} represents the probability P (S_j/S_i), the laws of probability require that the values of the outgoing arcs from a given state must sum to one. Same laws of probability apply to Initial Probabilities and Output Emission Probabilities.

3.2 Canonical Problems of HMM

The operations of an HMM are characterized by state sequence Q and the observation sequence O.

$Q = q_1, q_2, \cdots, q_T$

$O = O_1, O_2, \cdots, O_T$

Q is a fixed state sequence of length T, and corresponding observation sequence is O. T is total number of observation in the observation sequence and O_t is one of the symbols from V (Output variables). Using an HMM, we can generate an observation sequence $O = O_1, O_2, \cdots, O_T$. We can also estimate the most probable state sequence $Q= (q_1, q_2, \cdots, q_T)$ given the set of observations $O =(O_1, O_2, \cdots, O_T)$. Here the observations are assumed to have statistical independence; the Model has discrete observation (output) symbols. For an HMM to perform all these we need appropriate values of N, M, A, B and π. These values can be obtained by a learning process.

For an HMM to perform the above mentioned tasks we characterize it in terms of solving three fundamental problems. We can use HMMs to solve real problems with real data by solving these three problems. So much being said about the three problems let us now discuss what these problems are.

Problem 1 Evaluation: Given an observation sequence O and a model λ, what is the probability of the observation sequence, $P(O|\lambda)$?

$P(O|\lambda) = P(O1, O2, \dots, OT \mid \lambda) = ?$

Problem 2 Decoding: Given an observation sequence O and the model λ, what is the most probable state transition sequence Q for O?

$Q^* = \arg\max_{Q=(q1, q2, \dots, qT)} P(Q, O \mid \lambda) = ?$

Problem 3 Training: Given a training sequence O, find a model λ, specified by parameters (A, B, π) to maximize $P(O|\lambda)$ (we assume for now that Q and V are known).

$P(O \mid \lambda = (A, B, \pi)) < P(O \mid \lambda' = (A', B', \pi'))$

$\lambda^* = \arg\max_{\lambda} P(O|\lambda)$

Among the three problems only the evaluation problem has a direct solution. The other problems are harder and involve optimization techniques like dynamic programming. There are specific algorithms for each problem that explain a best way to solve them. The problem of

evaluation is solved using the Forward and Backward iterative algorithms. The second problem is solved using the Viterbi Algorithm, also an iterative algorithm that output best path by sequentially considering each observation symbol of O. The last problems which deals with training an HMM can be solved by using Baum- Welch or Maximum Likelihood Estimation (MLE). The decision between these two algorithms can be made based on the training data available for the learning process.

Problem	**Solution**
Evaluation	1) Forward Algorithm 2) Backward Algorithm
Decoding	Viterbi Algorithm
Training	1) Supervised - Maximum Likelihood Estimation (MLE) 2) Unsupervised- Baum-Welch Algorithm

Table 1: Problems of HMM

We will now discuss in detail how these algorithms solve the three problems associated with HMM. The section is divided based on the problem that has to be addressed. Each algorithm is explained mathematically using equations that make use of HMM Notation, $\lambda = (A, B, \pi)$.

3.2.1 Evaluation

Problem 1 is the evaluation problem. Given a model and a sequence of observation the question is how to compute the probability that the observation sequence is produced by the model. Here we are trying to see how well a particular observation sequence matches the given model. This is an extremely important point. If there is a situation where a choice has to be made among several competing HMMs, the model which best suits the observation sequence can be found using Evaluation.

The simplest way to solve the evaluation problem is by following the Brute-Force approach. In this approach we enumerate every state sequence of length T (the number of observations) and calculate the probability of each state sequence producing the given observation sequence.

Consider one such fixed state sequence

$$Q = q_1, q_2, \ldots, q_T$$

Here q1 is the initial state of the sequence. Now find the probability of the observation sequence O for the selected state sequence of length T

$$P(O \mid Q, \lambda) = \prod_{t=1}^{T} P(O_t \mid q_t, \lambda)$$

The below equation is valid for our HMM as we are dealing with discrete HMM.

$$P(O \mid Q, \lambda) = b_{q_1}(O_1) \; . \; b_{q_1}(O_2) \; . \; . \; . \; b_{q_n}(O_T)$$

The probability for the model to emit our fixed state sequence is

$$P(Q \mid \lambda) = \pi_{q_1} \; . \; a_{q_1 q_2} \; . \; a_{q_2 q_3} \; . \; . \; . \; a_{q_{T-1} q_T}$$

The joint probability that O and Q occur simultaneously is simply the product of the two terms.

$P(O, Q | \lambda) = P(O | Q, \lambda) * P(Q | \lambda)$

This is just one state sequence. Our goal is to obtain $P (O | \lambda)$, not $P(O, Q | \lambda)$

So, to obtain the probability of the model λ emitting O we sum $P(O, Q | \lambda)$ over all the possible state sequences.

$P(O | \lambda) = \sum_{all\ Q}(P(O | Q,\lambda) * P(Q| \lambda))$

$=\sum_{q1,q2,\ldots,qT} \pi_{q_1} . \; b_{q_1}(O_1) \; . \; a_{q_1 q_2} . b_{q_2}(O_2) \; \ldots \; a_{q_{T-1} q_T} . b_{q_T}(O_T)$

This approach can be followed but the computations get very long. There is a problem of computational complexity as the calculations to be done are exponential. This is because our model has N states and at every t= 1, 2, …, T, there are N possible states which can be reached. So there are N^T possible state sequences of length T and for each such state sequence about 2T calculations are required. To be precise we need $(2T-N^T)$ multiplications and N^T-1 additions. The computational complexity is $O(N^T T)$. Even if we have small N and T, this is not feasible: for N = 5 and T = 100, there are ~10^{72} computations needed.

There is a more efficient procedure to solve Problem 1. It is called Forward- Backward Procedure.

3.2.1.1 Forward Algorithm

Luckily, there is no need for so many computations. We can perform a recursive evaluation, using an auxiliary variable $\alpha_t(i)$, called the forward variable.

$$\alpha_t(i) = P(O_1, O_2, ..., O_t, i(t) = q_i \mid \lambda)$$

$\alpha_t(i)$, the probability of the partial observation sequence until time t and internal state q_t = Si given the model λ.

How does $\alpha_t(i)$ help?

$\alpha_t(i)$ makes a recursive calculation possible because in a first-order HMM, the transition and emission probabilities only depend on the current state [6]. These recursive calculations reduce the numbers of calculations needed to obtain P (O| λ).

We can solve $\alpha_t(i)$ inductively as follows

Step 1: Initialization

$\alpha_1(i) = \pi_i \ . \ b_i(O_1)$, $1 \leq i \leq N$

Step 2 : Induction

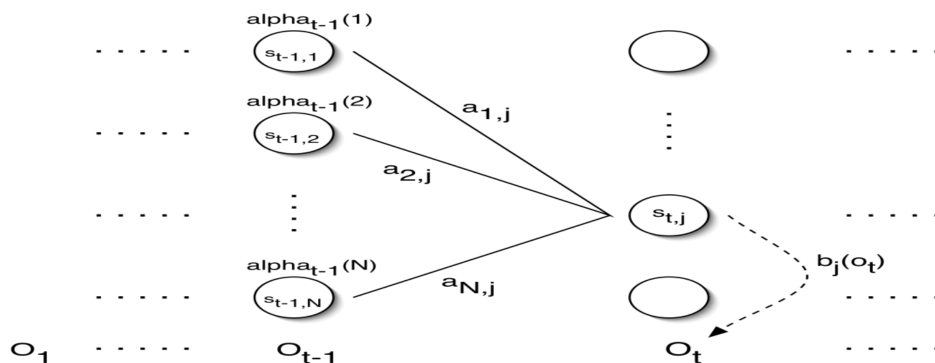

Figure 5: Operations for computing the forward variable α_j (t + 1)

Here we calculate $\alpha_{t+1}(j)$, the next step, using the previous one $\alpha_t(i)$. $\alpha_{t+1}(j)$ represents the probability of the observation sequence up to time t + 1 and being in state S_j at time t + 1.

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

$$1 \le t \le T-1,\ 1 \le j \le N$$

According to the above equation $\alpha_{t+1}(j)$ is the probability of observing symbol O_{t+1} when in state S_j ($b_j(O_{t+1})$), times the sum of the probabilities of getting to state S_j from state S_i times the probability of the observation sequence up to time t and being in state S_i [1].Note that we have to keep track of $\alpha_t(i)$,for all N possible internal states. These values are used in the termination step.

Step 3: Termination:

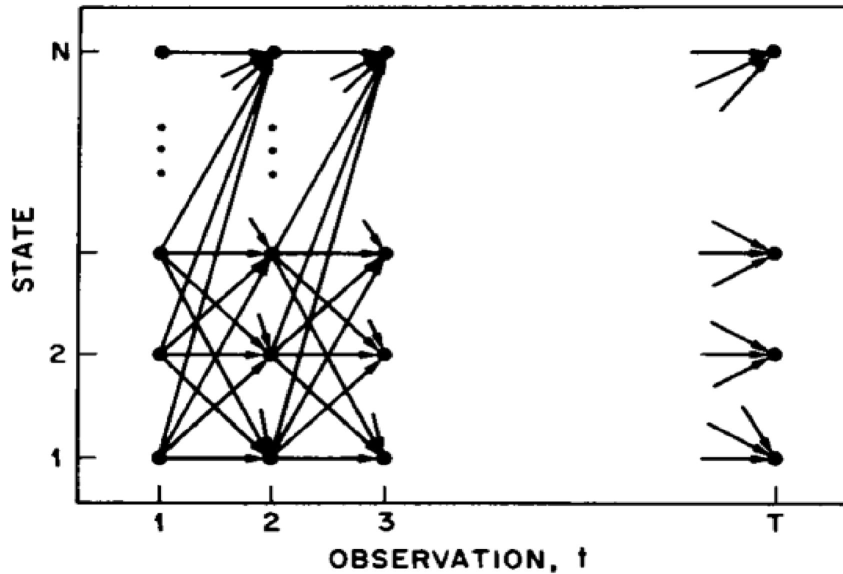

Figure 6: Computing $\alpha_j(t)$

If we know $\alpha_T(i)$ for all the possible states, we can calculate the overall probability of the sequence given the model, P(O | λ)

$$P(O \mid \lambda) = \sum_{i=1}^{N} \alpha_T(i)$$

The forward algorithm allows us to calculate P(O|λ). As it can be seen from the above algorithm it has a computational complexity $O(N^2T)$. This is linear in T, rather than exponential as compared to the Brute Force approach. This means that it is feasible. Apart from calculating P(O|λ) this algorithm is also used in Baum Welch Algorithm for unsupervised learning.

3.2.1.2 Backward Algorithm

The α values computed using the forward algorithm are sufficient for solving the first problem, P(O | λ). However, in order to solve the third problem, we will need another set of probabilities, the β values. In the same manner as forward variable we define a backward variable, $\beta_t(i)$. We denote the backward variable $\beta_t(i)$ as the probability of the partial observation sequence after time t, given state S_i at time t.

$\beta_t(i) = P(O_{t+1}, O_{t+2}, \ldots, O_T \mid qt = Si, \lambda)$, $1 \le t \le T$, $1 \le i \le N$

Just like α's, β's can also be computed using the following backward recursive procedure:

Step 1: Initialization

The initialization step arbitrarily defines $\beta_T(i)$ to be 1 for all i.

$\beta_T(i) = 1$, $1 \le i \le N$

This algorithm is backwards in the sense that the time interval t are from T to one.

Step 2: Induction

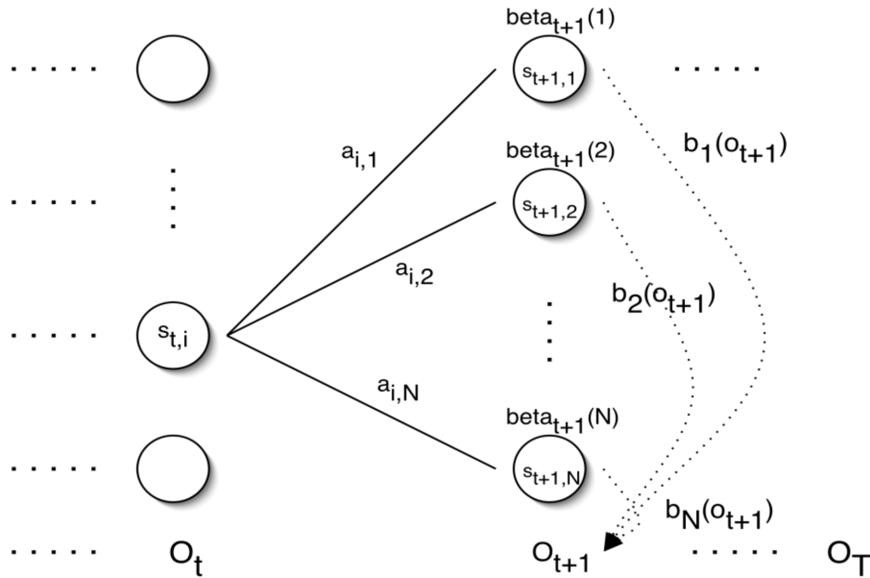

Figure 7: Computing $\beta_t(i)$

Here we calculate $\beta_t(i)$, the next step, that makes use of the previous one $\beta_{t+1}(j)$

$\beta_t(i) = \sum_{j=i}^{N} a_{ij} \,.\, b_j(O_{t+1})\, \beta_{t+1}(j)$, t=T-1, T-2, . . . , 1 $1 \leq i \leq N$

In order to have been in state S_i at time t and to account for the observation sequence from time t+1 on, you have to consider all possible states S_j at time t+1, accounting for the transition from S_i to S_j (a_{ij}), as well as observation O_{t+1} in state j ($b_j(O_{t+1})$), and then account for the remaining partial observation sequence from state j ($\beta_{t+1}(j)$) [1].

Step 3: Termination

$$p(O|\lambda)=\sum_{i=1}^{N}\beta_1(i)$$

Again the computation of $\beta_t(i)$, $1 \le t \le T$, $1 \le i \le N$ require N^2T calculations.

We will see later how the backward as well as the forward algorithms are used to solve problems 2 and 3 of HMM.

3.2.2 Decoding

Problem 2 deals with decoding. In decoding we attempt to uncover the hidden part of the HMM. In other words we try to find the optimal state sequence for a given observation sequence. Unlike evaluation, in decoding there is no single optimal sequence.

One possible solution to this problem is to choose states which are individually most likely and then find the single best state sequence that guarantees that the uncovered observation sequence is valid. This solution has several drawbacks. The most common solution to the decoding problem is the Viterbi algorithm which also uses partial sequences and recursion.

3.2.2.1 Viterbi Algorithm

The Viterbi algorithm is a dynamic programming algorithm that computes the most likely state transition path given an observed sequence of symbols. It is actually very similar to the forward algorithm, except that we will be taking a “max”, rather than a “ $\sum$ “, over all the possible ways to arrive at the current state under consideration.

However, the formal description of the algorithm involves some cumbersome notations [11].

We need to define the following quantity for solving the problem using Viterbi Algorithm.

$$\delta_t(i) = \max_q P(q_1, q_2, ..., q_t = i, o_1, o_2, ...o_t \mid \lambda)$$

$\delta_t(i)$ is the probability of the most probable path ending in state S_i at time t

By induction we have

$$\delta_{t+1}(j) = \max_i (\delta_t(i) a_{ij}) b_j(o_{t+1})$$

To retrieve the state sequence we need to keep track of the argument which maximize $\delta_{t+1}(i)$, for each t and j. We use an array $\psi_t(j)$ for back tracking the state sequence.

The Viterbi Algorithm is as follows

Step 1: Initialization

$$\delta_1(i) = \pi_i b_i(o_1) \qquad 1 \le i \le N$$

$$\psi_1(i) = 0$$

Step 2: Recursion

$$\delta_t(j) = \max_{1 \le i \le N} (\delta_{t-1}(i) a_{ij}) b_j(o_t)$$

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

$$2 \le t \le T, 1 \le j \le N$$

Step 3: Termination

$$P^* = \max_{1 \le i \le N} \delta_T(i)$$

P* gives the state-optimised probability

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

Q* is the optimal state sequence (Q* = {q1*,q2*,…,qT*})

Step 4: Backtrack State Sequence

$$q_t^* = \psi_{t+1}(q_{t+1}^*) \qquad t + T - 1, T - 2, \ldots, 1$$

Viterbi algorithm is similar to Forward algorithm except for the backtracking step and the maximization over previous states instead of summation. So, the time complexity here is $O(N^2T)$. We can use a trellis structure is clearly explain the Viterbi Algorithm

3.2.3 Training

The problem 3 in HMM is training an HMM to obtain the most likely parameters that best models a system, given a set of sequences originated from this system.

There is no known way to analytically solve for the model which maximizes the probability of the observation sequence(s). So we come up with models λ = (A, B, π) which locally maximizes P(O).

HMM = Topology + Statistical parameters

During the training process we compute the statistical parameters of the HMM. The topology is already designed. So the input to a training algorithm would be a database of sample HMM behaviour and output is the transition, emission and initial probability distribution of HMM. Thus we can conclude that given a set of examples from a process, we should

be able to estimate the model parameters $\lambda = (A, B, \pi)$ that best describe that process.

There are two standard approaches to the learning task based on the form of the examples (database available for learning process), supervised and unsupervised training. If the training examples contain both the inputs and outputs of a process, we can perform supervised training. It is done by equating inputs to observations, and outputs to states, but if only the inputs are provided in the training data then we must use unsupervised training. Unsupervised training guesses a model that may have produced those observations. Maximum Likelihood Estimation (MLE) comes under supervised training and Baum-Welch Algorithm comes under supervised training.

3.2.3.1 Supervised Learning

The easiest solution for creating a model λ is to have a large corpus of training examples, each annotated with the correct classification. If we having such tagged training data we use the approach of supervised training.

Maximum Likelihood Estimation (MLE):

MLE is a supervised learning algorithm. In MLE, we estimate the parameters of the model by counting the events in the training data. This is possible because the training examples for a MLE contain both the inputs and outputs of a process. We equate inputs to observations and outputs to states and easily obtain the counts of emissions and

transitions. These counts can be used to estimate the model parameters that represent the process.

$$aij = \frac{\# \, of \, transitions \, from \, i \, to \, j \, in \, the \, sample \, data}{total \, \# \, of \, transition \, from \, the \, state \, i \, in \, sample \, data}$$

$$b_i \, (v_k) = \frac{\# \, of \, emissions \, of \, the \, symbol \, v_k \, from \, i \, in \, the \, sample \, data}{total \, \# \, of \, emissions \, from \, the \, state \, i \, in \, sample \, data}$$

There is a possibility of aij or b_i (v_k) being zero. For example consider the case where state i is not visited by the sample training data then aij=0. In practice when estimating a HMM from counts it is normally necessary to apply smoothing in order to avoid zero counts and improve the performance of the model on data not appearing in the training set.

3.2.3.2 Unsupervised learning

Key idea of unsupervised learning is iterative improvement of model parameters. We can use iterative expectation-maximization algorithm, Baum-Welch to find local maximum of P (O | λ).

Baum- Welch Algorithm

Baum-Welch algorithm uses the forward and backward algorithms to calculate the auxiliary variables α, β.

B-W algorithm is a special case of the EM algorithm:

- E-step: calculation of ξ and γ
- M-step: iterative calculation of λ’

E-step :

In order to describe the procedure for solving the problem we need to first define $\xi_t(i,j)$

$$\xi_t(i,j) = P\left(q_t = S_i, q_{t+1} = S_j \mid O, \lambda\right)$$

$\xi_t(i,j)$ is the probability of being in state S_i at time t, and state S_j at time t+1, given λ, O.

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

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

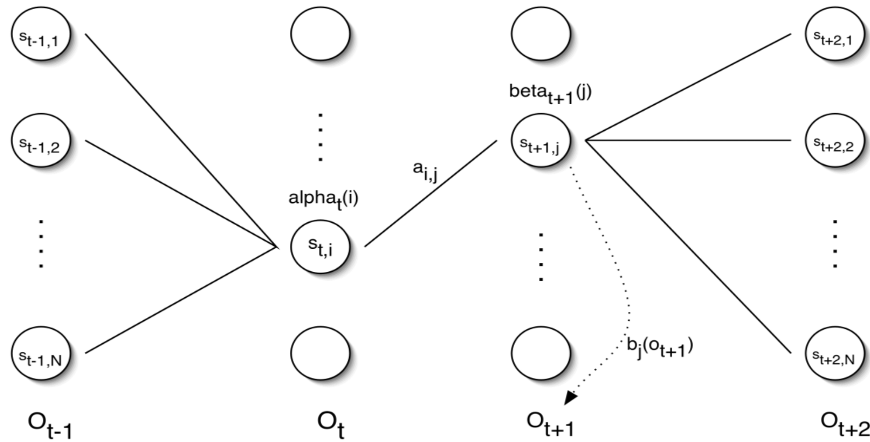

Figure 8: Operations for computing $\xi_t(i,j)$

After defining $\xi_t(i,j)$ define $\gamma_t(i)$

$\gamma_t(i) = P(q_t = S_i \mid O, \lambda)$

$\gamma_t(i)$ is the probability of being in state Si at time t for a given observation sequence O and model λ.

We can relate $\gamma_t(i)$ and $\xi_t(i,j)$ by summing over j

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

$\sum_{t=1}^{T-1} \gamma_t(i)$ = expected number of transitions from state Si to Sj

$\sum_{t=1}^{T-1} \xi_t(i,j)$ = expected number of transitions from Si to Sj.

Using these formulas we can re-estimate the parameters of an HMM.

M- Step:

A set of reasonable re-estimation formulas for a π, A and B are

- $\hat{\pi}$

  $\hat{\pi} = \gamma_1(i)$, the expected frequency of state i at time t=1

- $\hat{a}_{i,j}$

  $$\hat{a}_{i,j} = \frac{\text{expected number of transitions from states}_\text{i} \text{ to states}_\text{j}}{\text{expected number of transitions from states}_\text{i}}$$

  $$\hat{a}_{ij} = \frac{\sum \xi_t(i,j)}{\sum \gamma_t(i)}$$

- $\hat{b}_j(k)$

  $$\hat{b}_i(k) = \frac{\text{expected number of times in states}_\text{i} \text{ and observe symbol v}_\text{k}}{\text{expected number of times in states}_\text{i}}$$

  $$\hat{b}_j(k) = \frac{\sum_{t,o_t=k} \gamma_t(j)}{\sum \gamma_t(j)}$$

It we define the current model as $\lambda = (A, B, \pi)$, and use these values on the right hand side of the above equations we get the re-estimated model $\bar{\lambda}=(\bar{A},\bar{B},\bar{\pi})$.

It has been established by Baum and his colleagues that

$$P(O \mid \bar{\lambda}) > P(O \mid \lambda)$$

If we iteratively use $\bar{\lambda}$ in place of λ and repeat the re-estimation calculation we can improve the probability of O being observed from the model. This process is continued until some threshold value is reached that is when there is not much difference between $\bar{\lambda}$ and λ.

3.3 HMM Examples

3.3.1 Example 1: Coin Tossing Experiment

Consider the coin tossing experiment of Markov Models but here the person on the other side of the curtain has several coins both biased and un-biased with him. He selects one of his several coins and tosses it. Then he tells us the outcome (H, T), but not the coin selected. He does this several times and the outcome obtained after each trial is recorded as an observation. Here the coins will be the hidden states and H, T are the observations.

We make an assumption that the person has three coins and chooses among these three based on some probabilistic event [1]. This is the graphical representation of the example 1 model that has 3 coins.

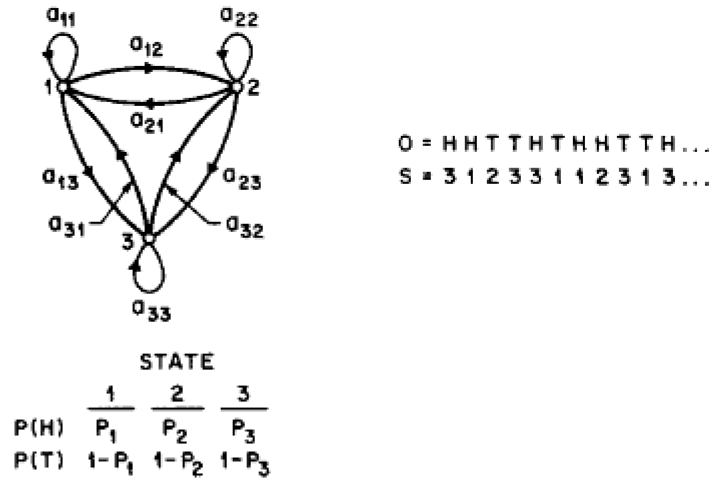

Figure 9 : HMM-Coin Tossing Experiment

3.3.1 Example 2: Stock Market Index

Consider the second example of Stock Market Index. It has three states representing stocks in the stock market which are associated with all the observations that represent the variations in these stocks.

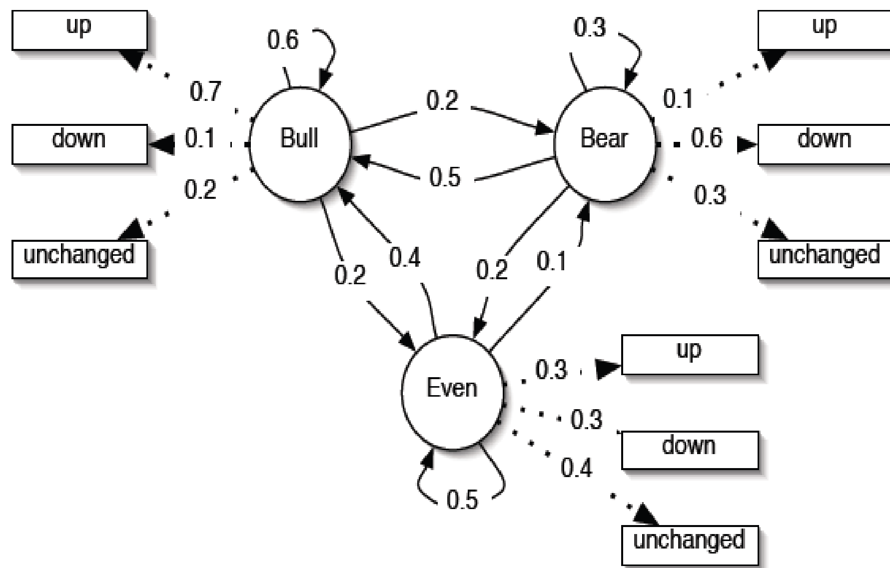

Figure 10 : Hidden Markov Model-Stock Market Index

Here the variations in stock market, up, down and unchanged are output variables and Bull, Bear and Even are hidden states. Given an observation sequence up-down-down, we cannot say exactly what state sequence produced these observations. We use some probabilistic functions that determine the most probable state sequence.

Parameters of this Model are

States:

Bull, Bear and Even

These are the internal states corresponding to the three stocks of the example

Transition Probabilities:

$$A = \begin{array}{c} \text{Bull} \\ \text{Bear} \\ \text{Even} \end{array} \overset{\begin{array}{ccc} \text{Bull} & \text{Bear} & \text{Even} \end{array}}{\begin{bmatrix} 0.7 & 0.1 & 0.2 \\ 0.1 & 0.6 & 0.3 \\ 0.4 & 0.1 & 0.5 \end{bmatrix}}$$

Initial Probabilities (π):

$$\pi = \begin{array}{c} \text{Bull} \\ \text{Bear} \\ \text{Even} \end{array} \begin{bmatrix} 0.33 \\ 0.33 \\ 0.33 \end{bmatrix}$$

Output Emission Probabilities (B):

$$A = \begin{array}{c} \text{Bull} \\ \text{Bear} \\ \text{Even} \end{array} \overset{\begin{array}{ccc} \text{Up} & \text{Down} & \text{Unchanged} \end{array}}{\begin{bmatrix} 0.6 & 0.2 & 02 \\ 0.5 & 0.3 & 0.2 \\ 0.3 & 0.3 & 0.4 \end{bmatrix}}$$

CHAPTER 4

IMPLEMENTATION OF HMM

In this thesis we implemented a numerically stable HMM in C++ programming language. In order to test the working of the HMM, we considered a simple real-time example and modelled it as an HMM. To keep the model very simple and easy to understand we consider an example of Stock Market Index with three stocks. There can be three variations possible to the values of these stocks.

We know that HMM can be characterized as solving three fundamental problems evaluation, decoding and training. Before getting to the implementation of the solution for these three problems, I will explain a major aspect in the understanding of HMM, the representation of a model file in source code.

4.1 Representation of HMM Model

For the Stock Market Index example we assume a fixed vocabulary that consists of three ascii characters u, d and n representing up, down and unchanged variations of stock value. When a symbol is not observed in the data, its count is automatically set to zero, effectively excluded from the model.

The number of states is stored in a variable N. This example has 3 states 0, 1 and 2 each representing *Bull*, *Bear* and *Even* stocks respectively.

An N-state HMM is represented by the following arrays:

1. Initial state probability (I)

   I is an array of length N (0-indexed), with I[s] representing the initial probability of being at state s.

2. State transition probability matrix (A)

   A is a 0-indexed two-dimensional array (NxN), with A[i][j] representing the probability of going to state j from state i.

3. Output probability matrix (B)

   B is a 0-indexed two-dimensional array (MxN), with B[o][s] representing the probability of generating output symbol "o" at state "s". M is the number of unique symbols (currently 3, stored in SYMNUM).

Note that the observed character “o” cannot be used directly as an index to access the entries of matrix B; it must be normalized by subtracting the lowest character “ ! “ [11].

4.1.1 HMM Representation on the disk file

An HMM can be encoded as a text file. The syntax is very simple. So, it will be explained by means of an example.

```
3
InitPr 3
0 0.34
1 0.33
2 0.33
OutputPr 9
0 u 0.7
0 d 0.1
0 n 0.2
1 u 0.1
1 d 0.6
1 n 0.3
2 u 0.3
2 d 0.3
2 n 0.4
TransPr 9
0 0 0.6
0 1 0.2
0 2 0.2
1 0 0.5
1 1 0.3
1 2 0.2
2 0 0.4
2 1 0.1
2 2 0.5
```

Figure 11: Screenshot of the Stock Market Index HMM Model file.

These are some of the rules associated with writing a model file. Failing to following any of these rules will fail the document from being considered as a model file. There could be any number of spaces between the numbers or words. All the white space characters are treated the same and if the model file has several white spaces they are treated as a single one.

But keywords "InitPr", "OutputPr", and "TransPr" is strict There should not be any variations in how they are written.

In the Model file

- The first number is the number of states
- The number after each keyword is the number of entries following it that should be associated with this keyword
- The entries not specified imply that the values are zero.

The keyword “InitPr” represents Initial Probabilities, “TransPr” represents Transition Probabilites and “OutputPr” represents Output Emission Probabilities [11].

Also let us see how an observation sequence would look like on a disk file.

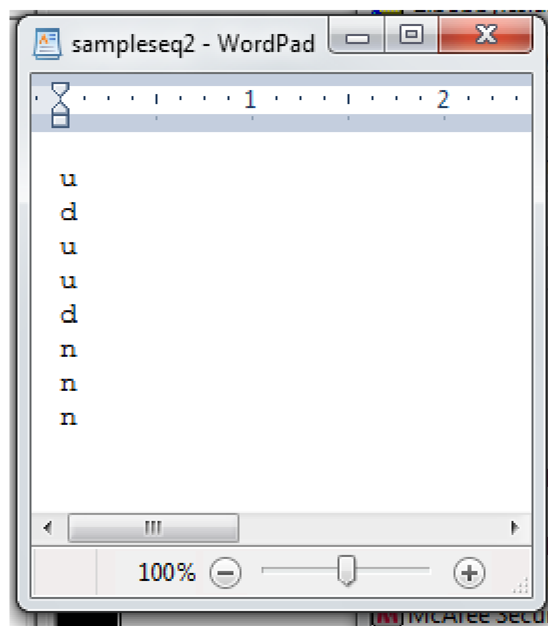

Figure 12 Screen shot of an observation sequence file

The sequence file contains a list of observations represented by their ascii values.

Model file and sequence files are the inputs, outputs of HMM.

The implementation of HMM is divided based on the three problems of HMM

4.2 Evaluation Problem

Problem: Given an observation sequence O and a model λ, what is the probability of the observation sequence, $P(O|\lambda)$?

Solution: Forward Algorithm and Backward Algorithm

Functions:

void ComputeAlpha(int *seq, int seqLength);

void ComputeBeta(int *seq, int seqLength);

Variables:

Array alpha[][]

alpha[t][i] = prob. of generating observations up to time t and being in state i at time t

Parameters:

- Observations

  *seq- the sequence of observed symbols, and is an array of length seqLength. It is 0-indexed, and seq[t] is the index of the symbols at time t, which can be used directly to access matrix B, B[seq[t]][i].

- HMM model file

Output: The probability that the given sequence has been generated by this model

Implementation:

Step 1:

```
// the following code computes alpha[0][i] for i =0, 1,...,N-1
for (i=0; i<N; i++) {
  alpha[t][i] = I[i]*B[seq[t]][i];
 }
}
```

Figure 13 Forward Algorithm-Initialization code snippet

Step 2:

```
// the following code will compute alpha[t][i] for t=1,..., seqLength-1.
while (t<seqLength) {

  for (i= 0; i<N; i++) {
    alpha[t][i] = 0;

    for (j=0; j<N; j++) {
      alpha[t][i] +=alpha[t-1][j]*A[j][i] ;
    }
    alpha[t][i]*=B[seq[t]][i];
   }
}
  t++;
}
```

Figure 14 code snippet to compute alpha values

Similarly Backward Algorithm has been implemented. One of the algorithms can be used to compute $P(O \mid \lambda)$

4.3 Decoding

Problem: This problem is the discovery of the most likely sequence of states that generated a given output sequence.

Solution: This can be computed efficiently using the *Viterbi algorithm*. A trackback is used to detect the maximum probability path travelled by the algorithm. The probability of travelling such sequence is also computed in the process.

Functions:

void Decode(char *seqFile);

ComputeStep(char *seq, int t, int i)

Parameters: model file and an observations sequence.

Variables:

delta[T][N], psy[T][N], stateSequence[T], lnProbability

Output: The sequence of states that most likely produced the sequence - a tagged sequence file

Implementation:

Step 1:

```
for (int i = 0; i <N; i++) {
    delta[0][i] = -Math.log(I[i]) - Math.log(B[cIndex][i]);
    psy[0][i] = 0;
}
```

Figure 15 Viterbi Algorithm-Initialization-code snippet

Step 2:

Compute Step

```java
/*
 * Computes delta and psy[t][j] (t > 0)
 */

public void computeStep(String obs, int t, int j) {
    // TODO Auto-generated method stub
    double minDelta = Double.MAX_VALUE;
    int min_psy = 0;

    for (int i = 0; i < N; i++) {
        double thisDelta = delta[t-1][i] - Math.log(A[i][j]);

        if (minDelta > thisDelta) {
            minDelta = thisDelta;
            min_psy = i;
        }
    }
    //char c=obs.charAt(0);
    int cIndex=GlobalVars.Vocab.get(obs);
    delta[t][j] = minDelta - Math.log(B[cIndex][j]);
    System.err.println("delta of ["+t+" ]"+"["+j+"] is.........." +delta[t][j] ).
    psy[t][j] = min_psy;
}
```

Figure 16 Viterbi Algorithm Compute Step

In this function we compute δ and Ψ values. Using these values we compute the most probable state sequence.

Step 3: Termination

```java
lnProbability = Double.MAX_VALUE;
for (int i = 0; i < N; i++) {
    double thisProbability = delta[size-1][i];

    if (lnProbability > thisProbability) {
        lnProbability = thisProbability;
        stateSequence[size - 1] = i;
    }
}
lnProbability = -lnProbability;

for (int t2 = size - 2; t2 >= 0; t2--)
    stateSequence[t2] = psy[t2+1][stateSequence[t2+1]];
```

Figure 17 Viterbi Algorithm-termination code snippet

StateSequence array stores the most probable state sequence.

The output of a HMM id a tagged sequence file which looks like this

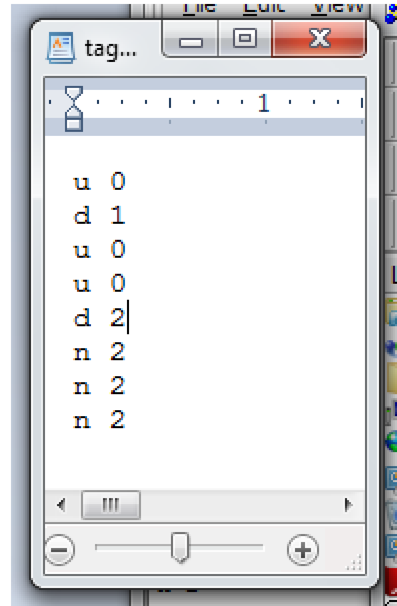

Figure 18: screen shot of tagged sequence file

4.4 Training

Problem: Find the most likely parameters that best models a system given a set of sequences that originated from this system.

4.4.1 Supervised Training

Solution: MLE

Functions:

void CountSequence(char *seqFile);

void UpdateParameter();

Parameters: tagged sequence file

Output: Model file

Implementation:

Accumulate the following counts

- count how many times it starts with state i
- count how many times a particular transition happens
- count how many times a particular symbol would be generated from a particular state

```
// Supervised training: You need to finish six assignment statements corresponding to
//            counting three different events.
void Model::CountSequence(char *seqFile)
{
  char c;  // to store the symbol
  int s; // to store the tagged state
  int prevState= -1;
  bool first=true;
  while (ifs >> c >> s) {
    int cIndex = c-baseChar; // convert the character to an index that can be used for BCounter.
    if (first) { // this is the very first observed symbol

      ICounter[s] +=1 ;
      INorm += 1;

      first = false;
    }
      BCounter[cIndex][s] += 1;
       BNorm[s] += 1;

    if (prevState>=0) { // the current symbol is not the first one; prevState has the previous state.

      // the state transition event,
       ACounter[prevState][s] +=1;
       ANorm[prevState] +=1;
    }
    prevState =s;
  }

}
```

Figure 19 screen shot of Counting in MLE

Using these counts relative frequencies are computed to obtain parameters of an HMM. This is done in the function UpdateParameters.

```
// Using the counts stored in the counters to estimate
// the HMM parameters
void Model::UpdateParameter()
{
  double smoothConstant = 0.00001; // smoothing constant
  int i, j;
  for (i=0; i< N; i++) {
    I[i] = (smoothConstant+ICounter[i])/(N*smoothConstant+INorm);

    for (j=0; j<N; j++ ) {
      A[i][j] = (smoothConstant+ACounter[i][j])/(N*smoothConstant+ANorm[i]);
    }
    for (j=0; j<SYMNUM; j++) {
      B[j][i] = (smoothConstant+BCounter[j][i])/(SYMNUM*smoothConstant+BNorm[i]);
    }
  }
}
```

Figure 20: code snippet for UpdateParameters() of MLE

4.4.2 Unsupervised Training

Solution: Baum-Welch Algorithm

Functions:

- Train(char *seqFile); train an HMM with an untagged sequence using Baum-Welch algorithm
- RandomInit(); initialize parameters of HMM to some random values
- double UpdateHMM(int *data, int seqLength);re-estimates the parameters of the HMM.

Parameters: A sequence file, Number of States and Output Variables

Output: Model file

Implementation:

We first initial probabilities values to the matrices A, B and π to random values using a function *RandomInit.*

- We train for many epochs.

  As we train, we keep track of how well the current HMM models the data (P(O | λ)). Additionally, we keep track of a history of how the model has fit the data in previous epochs (meanFit). As the HMM settles into a stable state, currentFit will asymptote to a particular value. The time-averaged meanFit will asymptote to this value also (though more slowly).

- When the currentFit becomes very similar to the meanFit, the model isn't changing, so we stop training. The code in function Train shows how this is done

```
    RandomInit(sym,maxT);

  double currentFit, meanFit = -1e-80;
      for(int epoch = 0; fabs( (meanFit - currentFit) / meanFit ) > 1e-6; epoch++) {
    /* Train for many epochs */
    cout << "\nEpoch " <<  epoch << endl;
    //cout << "before calling the update hmm method " <<endl;
    currentFit = UpdateHMM(sym, maxT);
    cout << " log-likelihood = "<< currentFit << endl;
    if (fabs(currentFit)< 0.1) {
      break;
    }
    meanFit = 0.2 * currentFit + 0.8 * meanFit;
  }
  cerr << "############## Final likelihood: " << currentFit << endl;
}
```

Figure 21 cone snippet of function train-BW Algorithm

The train function has the following steps in it

Step1:Initial probabilities values of the matrices A, B and π are set to random values using a function RandomInit.

Step 2:UpdateHMM function re-estimates the λ values to obatain λ'

```
double Model::UpdateHMM(int *data, int seqLength)
{
  ComputeAlpha(data, seqLength);
  ComputeBeta(data, seqLength);
  // compute data likelihood
  double prData = 1;
  int t;
  // now scale back
  for (t=0; t<seqLength; t++) {
    cout << "eta values are " << eta[t]<<endl;
    prData =  prData * eta[t];
  }
  prData = prData * 0.00001;
  prData = 1.0/prData;
    ResetCounter();  // Initialize all the counters and the normalizers
AccumulateCounts(data, seqLength);

  UpdateParameter();

}
```

Figure 22 code snippet of UpdateHMM function

AccumulateCounts function in UpdateHMM accumaulates counts using training data. These counts are needed for updating the parameters of HMM. It first computes the gamma's based on the alpha's and beta's and uses these to compute the following counts.

- counting for initial state distribution
- counting for output probabilities
- count for state transition probabilities

Step 3: Calculate Mean-fit. It the meant fit is less than or equal to a certain threshold value stop re-estimating.

The output of Baum-Welch Algorithm is a Model File that represents that best represents the observation sequence.

CHAPTER 5

IMPLEMENTATION ISSUES OF HMM

There are two practical issues associated with the implementation of HMM.

1) numerical scaling of conditional probabilities to model long sequences
2) smoothing of poor probability estimates caused by sparse training data

5.1 Scaling

When implementing a HMM, long observation sequences often result in the computation of extremely small probabilities. These values are usually smaller in magnitude than the smallest value a normal floating point number in a system can hold. This results in a significant problem called floating-point underflow.

This numerically instability is seen Viterbi and Forward Algorithms of HMM. When Viterbi and forward algorithms are applied to long sequences it results in extremely small probability values that could underflow on most machines. We solve this problem differently for each algorithm:

5.1.1Viterbi underflow

As the Viterbi algorithm only multiplies probabilities, a simple solution to underflow problem is to log all the probability values and then add those values instead of multiplying them.

If all the values in the model matrices (A, B, λ) are stored logged, then at runtime only addition operations are needed. But in our implementation model matrices are not logarithmic values.

This is how we convert floating numbers to log and instead of multiplying we add them.

delta[0][i] = Math.log(I[i]) + Math.log(B[cIndex][i]);

In the code as you can see in order to compute the delta value we just added the log values of I[i] and B[cIndex][i] instead of multiplying them. Similar scaling is applied to other computations of Viterbi Algorithm.

5.1.2 Forward algorithm underflow

The forward algorithm sums probability values, so using log values here will not help in preventing underflows. The most common solution to this problem is to use scaling coefficients that will keep the probability values in the dynamic range of the machine. These scaling coefficients should not be dependent on anything except for time t.

The scaling coefficient in our implementation is:

$$\prod_{k=1}^{t} \eta_k = \frac{1}{\sum_{i=1}^{N} \alpha_t(i)} = \frac{1}{p(O(t)|\lambda)}$$

This is how the scaled forward variable looks like

$$\hat{\alpha}_t(i) = \prod_{k=1}^{t} \eta_k \alpha_t(i)$$

In our implementation array eta (eta[t]) is the scaling coefficient. The code below shows the calculation of scaling coefficients and normalizing the alpha values.

```
for (i=0; i<N; i++) {
  alpha[t][i] = I[i]*B[seq[t]][i];
  eta[t]+= alpha[t][i];  // compute the normalizer
}
eta[t] = 1.0/eta[t];
for (i=0; i<N; i++) {
  alpha[t][i] *= eta[t]; //normalize all alpha values
}
```

Figure 23: source code to calculate scaling coefficients

5.1.2.1 Normalized Forward Algorithm

Step 1:

$$\hat{\alpha}_1(i) = \frac{\pi_i b_i(o_1)}{\sum_{k=1}^{N} \pi_k b_i(o_1)}$$

The code below shows the calculation of scaling coefficients and normalization of alpha values in step 1 of forward algorithm.

```
for (i=0; i<N; i++) {
  alpha[t][i] = I[i]*B[seq[t]][i];
  eta[t]+= alpha[t][i];  // compute the normalizer
}
eta[t] = 1.0/eta[t];
for (i=0; i<N; i++) {
  alpha[t][i] *= eta[t]; //normalize all alpha values
}
```

Figure 24 : source code to compute normalized alpha values

Step 2:

$$1 \leq i < T,\ \hat{\alpha}_{t+1}(i) = \frac{b_i(o_{t+1})\sum_{j=1}^{N}\hat{\alpha}_t(j)a_{ji}}{\sum_{k=1}^{N}b_k(o_{t+1})\sum_{j=1}^{N}\hat{\alpha}_t(j)a_{jk}}$$

This is how the above equation is implemented in code.

```
  t++;
  // the following code will compute alpha[t][i] for t=1,..., seqLength-1.
  while (t<seqLength) {
    eta[t]=0;
    for (i= 0; i<N; i++) {
      alpha[t][i] = 0;
      for (j=0; j<N; j++) {
        alpha[t][i] +=alpha[t-1][j]*A[j][i] ;
      }
      alpha[t][i]*=B[seq[t]][i];
      eta[t] += alpha[t][i]; // compute the normalizer for alpha[t][i], i=0,1,...,N-1.
    }
    eta[t] = 1.0/eta[t];
    for (i=0; i<N; i++) {
      alpha[t][i] *= eta[t]; // normalize alphas.
    }
    t++;
  }

}
```

Figure 25: Step 2 of Normalized Forward Algorithm

Similar coefficients and Normalization is down in Backward Algorithm.

5.2 Smoothing

Smoothing technique is used to solve the problem that occurs during the learning process. The output of learning in HMM is the estimated probabilities for vocabularies and transitions. Sparse training data causes poor probability estimates. Unseen words have emission probabilities of zero. Smoothing is the process of flattering probability

distribution so that all word sequences can occur with some probability. This often involves redistributing weight from high probability regions to zero probability regions.

In practice when estimating a HMM from counts it is normally necessary to apply smoothing in order to avoid zero counts and improve the performance of the model on data not appearing in the training set. This is how we implemented the concept of smoothing in our HMM.

```
void Model::UpdateParameter()
{
  double smoothConstant = 0.00001; // smoothing constant
  int i, j;
  for (i=0; i< N; i++) {
    I[i] = (smoothConstant+ICounter[i])/(N*smoothConstant+INorm);

    for (j=0; j<N; j++ ) {
      A[i][j] = (smoothConstant+ACounter[i][j])/(N*smoothConstant+ANorm[i]);
    }
    for (j=0; j<SYMNUM; j++) {
      B[j][i] = (smoothConstant+BCounter[j][i])/(SYMNUM*smoothConstant+BNorm[i]);
    }
  }
}
```

Figure 26: Smoothing in HMM

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, the theoretical aspects of a discrete HMM are implemented by dividing it into three canonical problems. The practical issues that arise during the implementation are also addressed which makes the HMM numerically stable. This Numerical stable HMM can be used in making application without worrying about genome length observation sequences or training data.

This thesis considered the output variables of the HMM to be discrete symbols but can be expanded to continuous output variables. Using this HMM as a basis and adding several features specific to the application under consideration, application specific HMM can be developed.

BIBLIOGRAPHY

[1] Lawrence R. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition

[2] Wikipedia

http://en.wikipedia.org/wiki/Hidden_Markov_model

[3] Hidden Markov Model in C#

http://crsouza.blogspot.com/2010/03/hidden-markov-models-in-c.html

[4] Daniel Jurafsky, James H. Martin. Speech and language processing

[5] Definition of HMM

http://www.wordiq.com/definition/Hidden_Markov_model

[6]Data Mining- Hidden Markov Models

http://www.csse.monash.edu.au/courseware/cse5230/2004/assets/week09.pdf

[7] Barbara Resch. Hidden Markov Models - A Tutorial for the Course Computational Intelligence

[8] Hidden Markov Models by Marc Sobel

[9] Hidden Markov Models by Phil Blunsom

[10] Hidden Markov Models by John Fry, San Jose State University

[11] A Brief Note on the Hidden Markov Models (HMMs) by ChengXiang Zhai

[12] Chapter 4: Hidden Markov Models by Prof. Yechiam Yemini, Columbia University

VITA

Graduate College
University of Nevada, Las Vegas

Usha Ramya Tatavarty

Degrees:

Bachelor of Technology in Computer Science, 2009
Jawaharlal Nehru Technological University, India

Thesis Title: Implementation of Numerically Stable Hidden Markov Model

Thesis Examination Committee:

Chairperson, Dr. Kazem Taghva, Phd.
Committee Member, Dr. Ajoy K. Datta, Phd.
Committee Member, Dr. Laxmi P. Gewali, Phd.
Graduate College Representative, Dr. Venkatesan Muthukumar, Phd.