5-14-2010

Hidden Markov Model with Binned Duration and Its Application

Zuliang Jiang
University of New Orleans

Hidden Markov Model with Binned Duration and Its Application

A Dissertation

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Engineering and Applied Science
Computer Science

by

Zuliang Jiang

B.S. Xiamen University, 2006

May 2010

Acknowledgments

I would like to express my deepest gratitude to my advisor Dr Stephen Winters-Hilt for his continuous support of my Ph.D. study and research. This dissertation is impossible without his guidance, patience and encouragement. His perceptive insight in research, perpetual enthusiam and big sense of humor always provide me an excellent and smooth enviroment for doing research. This would be one of the most important experience in my life.

My many thanks and appreciations go to my committee members, Dr. Dongxiao Zhu, Dr. Christopher Summa, Dr. Huimin Chen and Dr. Linxiong Li, whose invaluable feedback and comments greatly better my work. I am very grateful to my colleague Carl Baribault for his help throughout this work. I also would like to thank the faculty in the Computuer Science Department and Graudate School who provided a lot useful advice and discussions that helped solve any problems or concerns I had: Chairman Mahdi Abdelguerfi, Dr. Shengru Tu, Dr.Zella Huaracha, Ms. Amanda Athey, Ms. Jeanne Boudreaux and many others. In addition, I am really grateful to my parents for their love and support in my life.

Finally, my heartfelt thanks to those at the University of New Orleans whose names I didn't mention but contributed in any form towards the successful completion of the dissertation.

Contents

List of Figures

List of Tables

Abstract

Hidden Markov models (HMM) have been widely used in various applications such as speech processing and bioinformatics. However, the standard hidden Markov model requires state occupancy durations to be geometrically distributed, which can be inappropriate in some real-world applications where the distributions on state intervals deviate significantly from the geometric distribution, such as multi-modal distributions and heavy-tailed distributions. The hidden Markov model with duration (HMMD) avoids this limitation by explicitly incorporating the appropriate state duration distribution, at the price of significant computational expense. As a result, the applications of HMMD are still quited limited. In this work, we present a new algorithm - Hidden Markov Model with Binned Duration (HMMBD), whose result shows no loss of accuracy compared to the HMMD decoding performance and a computational expense that only differs from the much simpler and faster HMM decoding by a constant factor. More precisely, we further improve the computational complexity of HMMD from $\theta(TNN+TND)$ to $\theta(TNN+TND^*)$, where TNN stands for the computational complexity of the HMM, D is the max duration value allowed and can be very large and D^* generally could be a small constant value.

Chapter 1

HMM background

1.1 Introduction

A Markov chain is a sequence of random variables $S_1, S_2, S_3, \cdots$ with the Markov property: given the present state, the future and past states are independent [33]. Formally,

$$P(S_{n+1} = s|S_1 = s_1, S_2 = s_2, \cdots, S_n = s_n) = P(S_{n+1} = s|S_n = s_n).$$

The probability of a sequence $s = s_1, s_2, \cdots, s_n$ can be calculated by

$$P(s_1, s_2, \cdots, s_n) = P(s_1)P(s_2|s_1) \cdots P(s_{n-1}|s_{n-2})P(s_n|s_{n-1})$$

In the Markov model, the states are directly visible to the observer. Hidden Markov Models extend the Markov chains to include the case where the observation is a probabilistic function of the state. Now the states are not directly visible (they are hidden), but output observation symbols are visible. To be more precise, "Hidden Markov Model is a doubly embeded stochastic process with an underlying stochastic process that is not observable, but can only be observed through another set of stochastic process that produce the sequence of

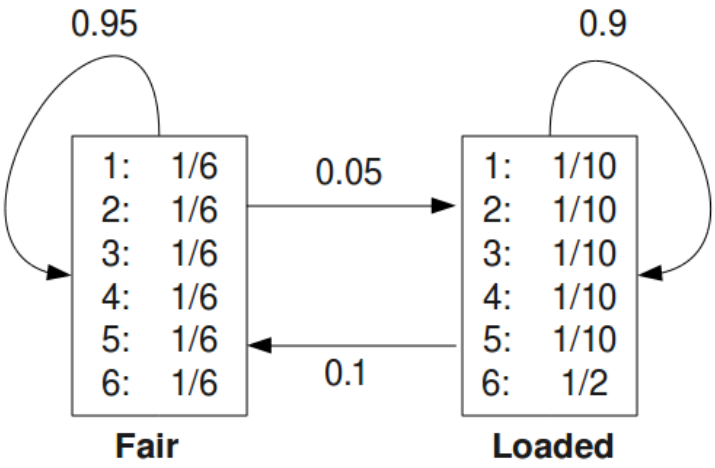


Figure 1.1: An illustration of a dishonest casino model from [13]

observations [38]."

To understand the concept of HMM, consider an occasionally dishonest casino example from [13]. In the casino, a fair die is used most of the time, but occasionally it will switch to a loaded die, and switch back later. Suppose the probability that a fair die will switch to a loaded die before each roll is 0.05, and the probability that a loaded die will switch to the fair die before each roll is 0.1. Also suppose that the loaded die will emit the digit "6" with probability 0.5 and other five digits each with probability 0.1. Figure 1.1 is the hidden Markov model we can draw [13].

Below is a sequence of 300 random rolls that are generated from the above described model. Each roll uses either the fair die (F) or the loaded one (L) as the generator based on the corresponding probability. We can see the sequence of rolls (the sequence of observations), but we don't know the underlying dies (F or L) that generate the rolls.

```
Rolls 3151162464466442453113216311641521336251445436316566266566666

Die   FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLLLLL


Rolls 6511664531326512456366646316366631623264552362666666625151631
```

```
Die   LLLLLLFFFFFFFFFFFLLLLLLLLLLLLLLLLFFFLLLLLLLLLLLLLLFFFFFFFFF

Rolls 22255544166656656356432436413151346514635341112641462625335 6
Die   FFFFFFFFLLLLLLLLLLLLLFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLL

Rolls 366163666466232534413661661163252562462255265252266435353336
Die   LLLLLLLLFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

Rolls 233121625364414432335163243633665562466662632666612355245242
Die   FFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLLLLLLLLLLLFFFFFFFFFFF
```

Now we turn to the formal definition of the hidden Markov model (HMM). We just recapitulate the elements of an HMM from [38]:

- N, the number of states in the model. We denote the individual states as $S = \{S_1, S_2, \cdots, S_N\}$, and the state at time t as q_t.
- M, the number of distinct observation symbols per state. We denote the individual symbols as $V = \{v_1, v_2, \cdots, v_M\}$.
- The state transition probability distribution $A = \{a_{ij}\}$ where

$$a_{ij} = P(q_{t+1} = S_j | q_t = S_i), \quad 1 \leq i, j \leq N.$$

- The observation symbol probability (emission probability) distribution in state j, $B = \{b_j(k)\}$, where

$$b_j(k) = P(v_k \text{ at } t | q_t = S_j), \quad i \leq j \leq N, 1 \leq k \leq M.$$

- The initial state distribution $\pi = \{\pi_i\}$ where

$$\pi_i = P(q_1 = S_i) \quad 1 \leq i \leq N$$

Given the values of the above terms N, M, A, B, and π, hidden Markov model can generate an observation sequence

$$O = O_1 O_2 \cdots O_T$$

(where O_t denotes the symbol observated at time t and T is the amount of observations of the sequence.) by

1. Choose an initial state $q_1 = S_i$ according to the initial state probability π_i and set $t = 1$.
2. Choose $O_t = v_k$ according to the symbol probability $b_i(k)$.
3. Transit to state $q_{t+1} = S_j$ according to the state transition probability a_{ij}.
4. Set $t = t + 1$; If $t < T$, go to step 2, else stop the procedure.

In the casino example, the number of states is two ($N = 2$), corresponding to the fair die and the loaded die; the number of distinct observation symbols per state is six ($M = 6$), corresponding to digit 1 to digit 6, which are the outcome alphabet of each roll; the transition probabilities and emission probabilities are already shown in Figure 1.1..

In hidden Markov model, there are three base problems to solve [38]:

- Problem 1 (Evaluation Problem): Given the observation sequence $O = O_1 O_2 \cdots O_T$ and a model $\lambda = (A, B, \pi)$, how to efficiently calculate the probability of the observation sequence, i.e. $P(O|\lambda)$?

- Problem 2 (Decoding Problem): Given the observation sequence $O = O_1 O_2 \cdots O_T$ and a model λ, how to choose a corresponding state sequence $Q = q_1 q_2 \cdots q_T$ that optimally "explains" the observations?

- Problem 3 (Learning Problem): How to estimate the model paramenter $\lambda = (A, B, \pi)$ that would maximize $P(O|\lambda)$?

We present mathematical solutions to each of the above three base problems of HMM in the next sections. In Section 1.2, we introduce the forward and backward algorithms of the standard HMM, which solve Problem 1. And the Baum-Welch algorithm of the standard HMM is shown in Section 1.3, which solves Problem 3. Then the Viterbi algorithm is introduced in Section 1.4, which solves Problem 2. Finally, in Section 1.5, we present the extensive applications of HMMs.

1.2 Forward and backward algorithm

Problem 1 is an evaluation problem. We want to calculate the probablity that the observed sequence $O = O_1 O_2 \cdots O_T$ was produced by the model, given a model λ. The brute force method of considering all possible state sequences and summing them altogether is impractical, since the the number of possible paths increases exponentially with the sequence length T. The forward algorithm and backward algorithm [38] described in this section efficiently solve this problem.

First consider the forward algorithm. Define the forware variable $\alpha_t(i)$ as

$$\alpha_t(i) = P(O_1 O_2 \cdots O_t, q_t = S_i|\lambda)$$

i.e., the probability of the partial observation sequence, $O_1 O_2 \cdots O_t$ and state S_i at time t, given the model parameter λ. We can compute $\alpha_t(i)$ by induction as follows

1. Initialization:

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

2. Induction:

$$\begin{aligned}
\alpha_t(j) &= P(O_1 O_2 \cdots O_t, q_t = S_j|\lambda) \\
&= \sum_{i=1}^{N} P(O_1 O_2 \cdots O_t, q_{t-1} = S_i, q_t = S_j|\lambda) \\
&= [\sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij}] b_j(O_t), \quad t = 2, 3, \cdots, T, \ 1 \leq j \leq N.
\end{aligned}$$

3. Termination:

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

Step (1) initializes the forward probabilities as the multiplication of the initial state probability of state S_i and the observation probability of O_1 at S_i. The induction step (2) calculates the $\alpha_t(j)$ by summing over all N possible state S_i at time t that are reachable to state S_j at time $t+1$ via state transitions, and then multiplying it by the observation probability $b_j(O_t)$. The termination step (3) gives the resulting calculation of $P(O|\lambda)$ as the sum of all N terminal forward variables $\alpha_T(i)$, since by definition,

$$\alpha_T(i) = P(O_1 O_2 \cdots O_T, q_T = S_i|\lambda)$$

and $P(O|\lambda)$ is based on the sum of $\alpha_T(i)$ over i. It is easy to see that the forward algorithm has a computional complexity of $\Theta(TNN)$.

The backward algorithm is presented in a very similar way. Define the backward variable $\beta_t(i)$ as the probability of the partial observation sequence from $t+1$ to the end, given state

S_i at time t and the model parameter λ

$$\beta_t(i) = P(O_{t+1}O_{t+2}\cdots O_T|q_t = S_i, \lambda)$$

Again we can calculate $\beta_t(i)$ by induction as follows

1. Initialization:

$$\beta_T(i) = 1, \quad 1 \leq i \leq N.$$

2. Induction:

$$\begin{aligned}
\beta_t(i) &= P(O_{t+1}O_{t+2}\cdots O_T|q_t = S_i, \lambda) \\
&= \sum_{j=1}^{N} P(O_{t+1}O_{t+2}\cdots O_T|q_t = S_i, q_{t+1} = S_j\lambda) \\
&= \sum_{j=1}^{N} a_{ij}b_j(O_{t+1})\beta_{t+1}(j), \quad t = T-1, T-2, \cdots, 1, \ 1 \leq i \leq N.
\end{aligned}$$

3. Termination:

$$P(O|\lambda) = \sum_{i=1}^{N} \pi_i\beta_1(i)b_i(O_1)$$

Step (1) arbitrarily initializes the all backward probabilities to 1. The induction step (2) calculates the $\beta_t(i)$ by summing over all N possible state S_j at time $t+1$ that are reachable from state S_i at time t via state transitions, accounting for the observation probability $b_j(O_{t+1})$ respectively. Again, the termination step (3) gives the resulting calculation of $P(O|\lambda)$, since by definition,

$$\beta_1(i) = P(O_2O_3\cdots O_T|q_1 = S_i, \lambda)$$

and $P(O|\lambda)$ is based on the sum of $\beta_1(i)$, accouting for initial state probability and obser-

vation $b_i(O_1)$ respectively. The computional complexity of the backward algorithm is also $\Theta(TNN)$.

To avoid the underflow error when the algorithms are implemented on a computer, we ususally multiply $\alpha_t(i)$ by a scaling coeffient c_t at each time index t, so that it will stay within a manageable numerical interval [13]. We usually define c_t as

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

Thus at each time index t, we first compute

$$\alpha_t(j) = [\sum_{i=1}^{N} \hat{\alpha}_{t-1}(i) a_{ij}] b_j(O_t)$$

Then the scaled version $\hat{\alpha}_t(j)$ is computed as

$$\hat{\alpha}_t(j) = \frac{[\sum_{i=1}^{N} \hat{\alpha}_{t-1}(i) a_{ij}] b_j(O_t)}{\sum_{j=1}^{N} \{[\sum_{i=1}^{N} \hat{\alpha}_{t-1}(i) a_{ij}] b_j(O_t)\}}$$

The backward variable has to be scaled with the same scaling coefficient at each time index, that is,

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

Since the magnitudes of the forward and backward variable are comparable, using the same scaling factor c_t at each time index is an efficent way of keeping their computations within manageable bounds.

1.3 Baum-Welch algorithm

Problem 3 is a parameter estimation problem. We want to optimize the model parameters so that it "makes most sense" that the given observation sequence(s) were generated by this model. We call the observation sequence(s) used to adjust the model parameter the "training sequence(s)". In this section, we introduce a standardly used method, called Baum-Welch algorithm [27] that will produce model parameters such that $P(O|\lambda)$ is locally maximized. The Baum-Welch algorithm is a kind of the Expectation Maximisation (EM) algorithm, which is a general algorithm for Maximum Likelihood (ML) estimation with "missing data". For HMMs the missing data is the unknown underlying state sequence, since we can only know the observations, but not the underlying state sequence that generates them.

The following is the Baum-Welch algorithm

1. Initialization: initialize the model parameters, i.e. λ.

2. Recurrence:

    - Calculate $\alpha_t(i)$ and $\beta_t(i)$ using the forward and backward algorithm in Section 1.2.

- Compute:

$$
\begin{aligned}
a_{ij}^{new} &= \frac{\text{expected number of transitions from state } S_i \text{ to state } S_j}{\text{expected number of transitions from state } S_i} \\
&= \frac{\sum_{t=1}^{T} \alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{\sum_{j=1}^{N} \sum_{t=1}^{T} \alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)} \\
b_j^{new}(k) &= \frac{\text{expected number of times in state } S_j \text{ and observing sysbol } v_k}{\text{expected number of times in state } S_j} \\
&= \frac{\sum_{\substack{t=1 \\ \text{s.t. } O_t = k}}^{T} \alpha_t(j) \beta_t(j)}{\sum_{k=1}^{M} \sum_{\substack{t=1 \\ \text{s.t. } O_t = k}}^{T} \alpha_t(j) \beta_t(j)} \\
\pi_i^{new} &= \text{expected number of times in state } S_i \text{ at time } t = 1 \\
&= \frac{\pi_i b_i(O_1) \beta_1(i)}{P(O|\lambda)}
\end{aligned}
$$

- Calculate the $P(O|\lambda)$ in log space.

3. Termination: Stop if the change in log likelihood is less than some predefined threshold or the maximum number of iterations is exceeded.

Step (1) provides reasonable intial values for the model parameters. In recursion step (2), the $\alpha_t(i)$s and $\beta_t(i)$s are calculated using the forward and backward algorithm from Section 1.2. $P(q_t = S_i, q_{t+1} = S_j|O, \lambda) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{P(O|\lambda)}$ is the probability of being in state S_i at time t, and state S_j at time $t+1$, given the model and the observation sequence. $P(q_t = S_j|O, \lambda) = \frac{\alpha_t(j)\beta_t(j)}{P(O|\lambda)}$ is the probability of being in state S_j at time t, given the model and the observation sequence. Each recursion helps converge the $P(O|\lambda)$ to a local maximum [13]. The termination step (3) stops the algorithm when the change in log likelihood is sufficiently small, or the predefined number of iterations is reached.

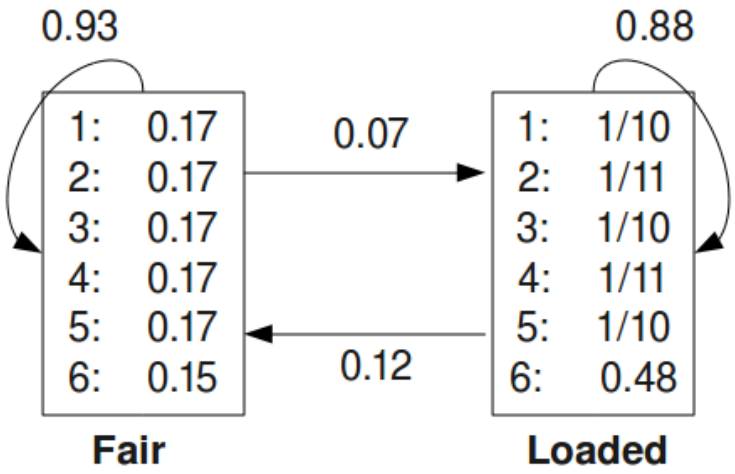


Figure 1.2: An illustration of Baum-Welch generated dishonest casino model from [13]

As shown in Section 1.2, the forward and backward algorithm use the scaling procedure to avoid underflow problem. Since the same set of coefficents appear in both the numerator and denominator of the above formulas, they cancel out altogether, so the formulas stay unchanged when we use the rescaled version of $\alpha_t(i)$ and $\beta_t(i)$. For a detailed proof, please refer to [38].

In the occasional dishonest casino example, if we have enough sequence(s) of rolls, then we can run the Baum-Welch algorithm to obtain a model closer to the correct one. Figure 1.2 is a model that is estimated by running Baum-Welch on 30,000 random rolls that are generated by the model in Figure 1.1 [13]. This resulting figure is close to its generator figure.

1.4 Viterbi algorithm

Problem 2 is a decoding problem. We attempt to uncover the hidden part of the model. We want to find out the most "reasonable" underlying state sequence for the obeservation sequence. There are several ways of doing this depending on how the "optimality criteria" is defined. In this section, we will describe the most common one, known as the Viterbi

algorithm, which is a dynamic programming algorithm. The Viterbi algorithm finds a single "best" state sequence (with highest probability), $Q = \{q_1 q_2 \cdots q_T\}$, for the given observation sequence $O = \{O_1 O_2 \cdots O_T\}$.

We define the Viterbi variable as

$$\delta_t(i) = \max_{q_1, q_2 \cdots q_{t-1}} P(q_1 q_2 \cdots q_t = i, O_1 O_2 \cdots O_t | \lambda)$$

i.e., $\delta_t(i)$ is the most probable path ending in state i at time t with observation O_t. We also define $\psi_t(i)$ to store the state that transits to state i in the most probable path, so as to keep track of the most probable paths. The Viterbi algorithm can also be calculated by induction as follows

1. Initialization:

$$\delta_1(i) = \pi_i b_i(O_1), \quad 1 \leq i \leq N.$$
$$\psi(i) = 0, \quad 1 \leq i \leq N.$$

2. Induction:

$$\delta_t(j) = b_i(O_t) \max_{i=1}^{N}(\delta_{t-1}(i) a_{ij}), \quad 2 \leq t \leq T, \ 1 \leq j \leq N.$$
$$\psi_t(j) = \operatorname{argmax}_{i=1}^{N}(\delta_{t-1}(i) a_{ij})$$

3. Termination:

$$P^* = \max_{i=1}^{N} \delta_T(i)$$
$$q_T^* = \operatorname{argmax}_{i=1}^{N} \delta_T(i)$$

4. Path backtracking:

$$q_t^* = \psi_{t+1}(q_{t+1}^*), \quad t = T-1, T-2, \cdots, 1.$$

The Viterbi algorithm is similar to the forward algorithm. The major difference is that the summing operation is now replaced by a maximization operation. To avoid overflow error on computing, we usually calculate Viterbi in log space, so the above steps become

1. Initialization:

$$\delta_1(i) = \log(\pi_i) + \log(b_i(O_1)), \quad 1 \leq i \leq N.$$
$$\psi(i) = 0, \quad 1 \leq i \leq N.$$

2. Induction:

$$\delta_t(j) = \log(b_i(O_t)) + \max_{i=1}^{N}(\delta_{t-1}(i) + \log(a_{ij})), \quad 2 \leq t \leq T, \ 1 \leq j \leq N.$$
$$\psi_t(j) = \operatorname{argmax}_{i=1}^{N}(\log(\delta_{t-1}(i)) + \log(a_{ij}))$$

3. Termination:

$$P^* = \max_{i=1}^{N}\delta_T(i)$$
$$q_T^* = \operatorname{argmax}_{i=1}^{N}\delta_T(i)$$

4. Path backtracking:

$$q_t^* = \psi_{t+1}(q_{t+1}^*), \quad t = T-1, T-2, \cdots, 1.$$

For the occasional dishonest casino example, we can run the Viterbi algorithm to find out the most probable path for a sequence of rolls, through the model obtained by the Baum-Welch algorithm in Section 1.3 (shown in Figure 1.2). Below is the result of running the Viterbi algorithm on a sequence of 300 random rolls that are generated from the correct model in Figure 1.1. Comparing this result with the real one in Section 1.2, it is apparent that the Viterbi algorithm has recovered the hidden state sequence quite well.

```
Rolls   315116246446644245311321631164152133625144543631656626566666
Die     FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLLLLL
Viterbi FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLL

Rolls   651166453132651245636664631636663162326455236266666625151631
Die     LLLLLLFFFFFFFFFFFLLLLLLLLLLLLLLLLFFFLLLLLLLLLLLLLFFFFFFFFF
Viterbi LLLLLLFFFFFFFFFFFLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLFFFFFFFF

Rolls   22255544166656656356432436413151346514635341112641462625335 6
Die     FFFFFFFLLLLLLLLLLLLLFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFLL
Viterbi FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFL

Rolls   366163666466232534413661661163252562462255265252266435353336
Die     LLLLLLLLFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
Viterbi LLLLLLLLLLLLFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

Rolls   233121625364414432335163243633665562466662632666612355245242
Die     FFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLLLLLLLLLLLFFFFFFFFFF
Viterbi FFFFFFFFFFFFFFFFFFFFFFFFFFFLLLLLLLLLLLLLLLLLFFFFFFFFFF
```

1.5 Applications

Hidden Markov model' greatly automatized training and learning ability and strong capabibiliy to model spatio-temporal time series make them the underlying communication, error-coding, and structure-identification algorithms used in cell-phone communications, deep-space satellite communications, voice recognition, and gene-structure identification [38] [3], with increasingly many areas such as image processing and channel current cheminformatics [49].

Since 1970s, hidden Markov models have been extensively used in speech recognition area. In an isolated word recognizer, a HMM model can be built for each word in the vocabulary using the training observation sequences, which are obtained by preprocessing a set of occurences of each spoken words. Then for each unkown word to be recognized, we pass it through all possible word models and calculate their respective model likelihoods by running the Viterbi algorithm. Finally we choose the word model with the highest likelihood. Based on the individual word models, we can build more complicated connected word recognizers. The purpose is to find an "optimal" word sequence that best matches the input observation sequence. The level building approach is a popular method wildly used. First, the observation sequence is matched against the individual word models using the Viterbi algorithm to generate candidiate word sequences ordered by the resulting probability scores. Then a postprocessing phrase does the validation job, eliminates the unlikely candidates, and chooses the most probable one. For large vocabulary speech recognition systems, the basic speech unit extends to sub-words, and can be much more complicated. Currently, most modern large vocabulary speech recognition systems are based on HMMs.

Hidden Markov models have become more and more popular in bioinformatics, especially in DNA sequence analysis [29]. The observation sequences here are nucleotide bases, adenine, thymine, cytosine, guanine, generally abbreviated as A, T, C, G and the hidden states are site

states such as exons, introns, etc. The hidden Markov models usually have to be expanded to include additional requirements such as the codon frame information, site state duration probability informations and must also follow some acception rules. For example, the start of an initial exon must begin with the canonic triplet sequence ATG and end with TAA, TAG or TGA; the introns generally follow the GT-AG rule. There are many popular gene finder application programs in this area such as GeneMark, GeneScan and etc. The statistical model employed by GeneMark.hmm is a HMM with duration or a hidden semi-Markov model. The state duration distributions are derived as approximation of the observed length distributions in the training set of sequences and they are characterized by the minimum and maximum duration length allowed. For example, the minimum and maximum durations of introns and intergenic seqeunces are set to 20 and 10,000 nts. The hidden Markov model with binned duration algorithm presented in this dissertation could be used as an improvement for the GeneMark to make it more efficient while still remaining the same powerful performance.

Gesture recognition is another area where hidden Markov models are often applied [47] [32]. Consider an automatic system that recognizes continuous hand motion for Arabic number from 0 to 9 [31]. In the segmentation and preprocessing stage, a Gaussian Mixture Model (GMM) is used for skin color detection, then hands are localized and tracked using blob analysis to generate their motion trajectors (gesture paths). Good features including location, orientation and velocity are extracted in the following feature extraction stage. The final stage is the HMM classification. Based on the complexity of the gesture, several states are generated for each isolated gesture by mapping each straight-line segment into a HMM state. A corresponding hidden Markov model is built for each isolated gesture. The Baum-Welch algorithm then is applied to train the HMMs and the Viterbi algorithm is used for identifying.

Hidden Markov models are also found in many other applications such as handwriting and text recognition [43] [46] [48], image processing [22] [23], computer vision [10], communication

[17], Climatology [14], Acoustics [40] [26] and so on. In many applications, the ability to incorporate the state duration into HMM is very important because the standard, HMM-based, the Viterbi and Baum-Welch algorithms are critically constrained in their modeling ability to distributions on state intervals that are geometric. This can lead to a significant decoding failure in noisy environments when the state-interval distributions are not geometric (or approximately geometric). The starkest contrast occurs for multimodal distributions and heavy-tailed distributions. Hidden Markov model with duration (HMMD) that will be introduced in Chapter 3 avoids this problem by the inclusion of explicit state duration density in HMMs. However, the computional cost brought is very expensive, if incorporating a maximum duration value of D, the original method will have a D^2 increase of computational cost using the original method [38], even the latest algorithms [56] still require a D fold computitional cost compared to the standared HMM. The above applications either not have duration information in the HMM at all (but instead have a postprocessing step to deal with durations on the Viterbi state sequecences, if needed), or have a quite limited duration ability and maybe have to sacrifice the performance of their applications. The method presented in this dissertation and just published in IEEE transactions on signal processing [54] helps solve the above problems by providing an efficient hidden Markov model with incorporated duration method (HMMBD).

Chapter 2

Distribution types

In HMMs, the state duration distribution is implicitly geometric (See Section 3.1), that is, at each time index t, the probability of remaining at the same state is a_{ii} (a_{ii} is the self transition probability), and the probability of transiting to other states is $1 - a_{ii}$. However, many real-world problems can have duration distribution quite different from the geomeric distribution. Non-geometric duration distributions occur in many familiar areas, such as the length of spoken words in phone conversation, as well as other areas in voice recognition. Even the most commonly used Gaussian distribution in many scientific fields to model data generations could be different from the geometric distribution, not to say there are huge number of other (skewed) types of distributions, such as heavy-tailed (or long-tailed) distributions, the multimodal distribution and so on.

Heavy-tailed distributions are widespread in describing phenomena across the sciences [15]. The log-normal and Pareto distributions are heavy-tailed distributions that are almost as common as the normal and geometric distributions in descriptions of physical phenomena or man-made phenomena and many other phenomena. Pareto distribution was originally used to describe the "allocation of wealth" of the society, known as the famous "80-20" rule, namely, about 80% of the wealth was owned by a small amount of people, while "the tail", the

large part of people only have the rest 20% wealth [30]. Pareto distribution has been extend to many other areas. For example, the internet traffic is of the long-tailed distribution, that is, there are only few large sized files but many small sized files to be transferred, while the large files are still the dominant components. This distribution assumption is an important factor much be considered to design a robust and reliable network and Pareto distribution could be a suitable choice to model the traffic generation. (The internet applications have found more and more heavy-tailed distribution phenomena.) Pareto distribution could also be found in a lot of other fields, such as economics, where it is often referred to as the Bradford distribution, to model the size of stock prices, to model the amount of oil reserves and so on.

The log-normal distribution is a good fitting to be used in many areas such as geology and mining, medicine, enviroment, atmosperic scicence and so on, where skewed distribution occurences are very common [15]. In Geology, the concentraion of elements and their radioactivity in the Earth's crust are ofter shown to be log-normal distributed. The infection latent period, the time from being infected to disease symptoms occurs, could usually be modeled as the log-nomal distribution. In enviroment, the distribution of particles, chemical and organisms is more or less log-normal distributed. Many atomospheric physical and chemical properities obey the log-normal distribution. The density of bacteria population often follows the log-normal distribution law. In linguistics, the number of letters per words and the number of words per sentence fit the log-normal distribution. The areas where log-normal distribution has extensive applications are too broad to enumerate.

In biology, differenct eukaryotic species may hold different length distributions. In many cases the length distribution for introns, in particular, has very strong support in an extended heavy-tail region, likewise for the length distribution on open reading frames (ORFs) in genomic DNA [3] [21]. For example, the exons and introns of human gene have a long-tailed in its distribution. In fact, the anomalously long-tailed aspect of the ORF distributions

could be the key distinguishing feature of this distribution, and has been the key attribute used by biologists to identify likely protein-coding regions in genomic DNA since the early days of (manual) gene structure identification. The length distributions on blockade states in channel current analysis can likewise be strongly skewed (engineered) towards having a heavy-tail, especially if the channel current environment is modulated to obtain an inverted population.

Poisson distribution could be used to model an extraordinarily large number of nutural and social phenomena. It is first used to model the accidental deaths of solders that are kicked to dealth by horses in Prussian Army by Ladislaus Bortkiewicz in his book titled "The Law of Small Numbers", where he noted that Possion distribution is a good fitting to model events with small varied probability but in a large population. Because of his contribution, Possion distribution sometimes also is referred to as "Bortkiewicz distribution". Bortkiewicz's initial application has then led to huge applications in many other areas such as analyzing traffic accidents, analyzing the typo error rate in the book page, counting the number of mails lost in one day, calculating the number of emergency cases called in a hospital in one day, computing the hit rate of a lightly loaded website and calculating the radiocative decay rate of an unstable substance.

Gaussian distribution or the normal distribution is a continuous probability distribution that describe the data that cluster around the mean. From the central limit theorem, the sum of a large number of independent radom variables is approximately Gaussian distributed. Guassian distribution may be the most commonly used distribution assumed by the random variables used as the data generator in many scientific areas. Gaussian distribution was first introduced by Abraham de Moivre in [12] to approximate binomial distribution with large n. Guassian distributions are ubiquitous in social and natural phenomena. Some examples but far from all are: the inaccuracy error rate of the length of the machine parts, the height of man in one area, the height of wave, the thermal noice in the semiconductor parts, the

underlying distribution assumed in information entropy and etc.

Gamma distribition is frequently used to model waiting times, for examples, the waiting time before death. Gaumma distribution are also applied to other areas, such as the size of insurance claims, the CRT correction, image enhancement, the amount of rainfalls and etc. Gamma distribution is sometimes used in HMM to model its state duration [28]. The memoryless property of expotential distribution makes it suitable to model the life time of component. The exponential distirbutions also have extensive usages in queuing theory, reliability engineering and so on.

Besides the above distributions, there are tons of other various distributions in real world, such as distributions that are convolutions of other distributions, multimodal distributions that have many peaks and so on. The shortcoming of the standard HMM – limiting HMM to only able to use the geometric distribution to model its state duration distribution – could greatly hinders HMM's application in many fields. The HMM with duration (HMMD) introduced in the next chapter overcomes this shortcoming by incorporating the state duration distribution explicitly into the model. As a result, HMMD can model any distributions directly no matter what type they are. In many applications, the use of HMMDs over HMMs is expected to offer a significant benefit.

Chapter 3

HMM with duration

This chapter will introduce the exact hidden Markov model with duration (HMMD). We will first show the differences between hidden Markov model (HMM) and hidden Markov model with binned duration (HMMD) and one of the original HMMDs presented by Rabiner in [38] in Section 3.1. Based on Rabiner's formulas, we derive our version of HMMD in Section 3.2 and Section 3.3 with an improved computational complexity from $\theta(TNN + TND^2)$ to $\theta(TNN + TND)$. (A similar improvement was achieved in 2003 in [56].) We apply this version of HMMD as the platform for incorporation of the side-information in our in-process paper [53]. The purpose of introducing this version of HMMD here is that it is the basis of Chapter 5, that is, the hidden Markov model with binned duration introduced in Chapter 5 is derived from the version of HMMD introduced in this chapter. In the last section– Section 3.4, we introduce the extensive real-world applications of HMMD.

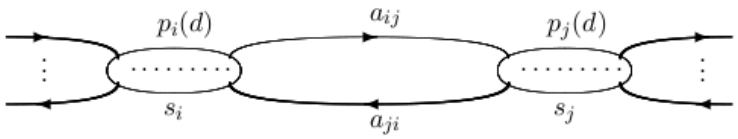
3.1 Introduction

In the standard HMM, when a state i is entered, that state is occupied for a period of time, via self-transitions, until transiting to another state j. If the state interval is given as d, the standard HMM's description of the probability distribution on state intervals is implicitly

given by

$$\begin{aligned} p_i(d) &= \text{probability of } d \text{ consecutive observations in state } S_i. \\ &= a_{ii}^{d-1}(1 - a_{ii}) \end{aligned} \tag{3.1}$$

where a_{ii} is the self-transition probablility of state i. This geometric distribution is inappropriate in many cases. For examples, if we know the system will stay on state 1 for about one miniute and later on state 2 for about two minites and so on, we just cannot represent this information in the standard HMM. We'd better need another paradigm – HMM with duration (HMMD) – that could directly incorporate the version of $p_i(d)$ that models the real duration distribution of state i into the hidden Markov models. If the explicit knowledge about the duration of states HMM is encoded, then the general HMMD can be illustrated as



When entered, state i will have a duration of d according to its duration density $p_i(d)$, it then transits to another state j according to the state transition probability a_{ij} (self-transitions, a_{ii}, are not permitted in this formalism). As shown, although the state transition between different states a_{ij} remains unchanged – still obeys the (first order) Markov property, the state self-transition a_{ii} now is replaced with the explicit $p_i(d)$. Since the Markov property is vialated, the new model is also often referred to as "Hidden Semi-Markov Model (HSMM)" [2]. It is easy to see that the HMMD will turn into a HMM if $p_i(d)$ is set to the geometric distribution shown in Eq.(3.1).

The first HMMD formulation was studied by Ferguson [16]. A detailed HMMD description was later given by Rabiner [38] (we follow much of the Rabiner notation in this

dissertation). The most intuitive method is to divide each state into substates based on the maximum allowed duration value D for that state. The bigger D is, the more substates need to be generated, and vice versa. The immediate consequece is a D^2 fold computation cost brought. There have been many efforts to improve the computational efficiency of the HMMD formulation given its fundamental utility in many endeavors in science and engineering. Notable amongst these also include the variable transition HMM methods for implementing the Viterbi algorithm introduced in [39], and the hidden semi-Markov model implementations of the forward-backward algorithm [56].

We now recapitulate the exact HMMD formalism in the notation introduced by Rabiner [38]. Based on it, we will later present our algorithms in Section 3.2 and Section 3.3. Elements (λ) of HMMD are as follows

- N, the number of states;
- M, the number of distinct observations;
- D, the maximum duration length;
- a_{ij}, the state transition probability;
- $b_j(k)$, the emission probability: probability of observing v_k in state i;
- π_i, the initial state probability.
- $p_i(d)$, the state duration density: the probability of having exactly d consecutive state i observations after state i is entered.

Given the above model (λ), an observation sequence

$$O = O_1 O_2 \cdots O_T$$

can be generated as follows

1. Choose an initial state $q_1 = S_i$ according to the initial state distribution π_i and set $t = 1$.

2. A duration d is chosen according to the state duration density $p_i(d)$.

3. Choose $O_t O_{t+1} \cdots O_{t+d-1}$ according to the joint symbol probability distribution in state S_i, i.e., $b_i(O_t) b_i(O_{t+1}) \cdots b_i(O_{t+d-1})$.

4. Transit to a new state $q_{t+d} = S_j$ according to the state transition probability distribution for state S_i, i.e. a_{ij}.

5. Set $t = t + d$; If $t < T$ return to step 2, else terminate the procedure.

The following forward-backward variables are used by HMMD

- $\alpha_t(i) = P(O_1 O_2 \cdots O_t, S_i \text{ ends at } t|\lambda)$
- $\beta_t(i) = P(O_{t+1} \cdots O_T | S_i \text{ ends at } t, \lambda)$
- $\alpha_t^*(i) = P(O_1 O_2 \cdots O_t, S_i \text{ begins at } t+1|\lambda)$
- $\beta_t^*(i) = P(O_{t+1} \cdots O_T | S_i \text{ begins at } t+1, \lambda)$

where $\alpha_t(i)$ can be calculated by

$$\alpha_t(i) = \sum_{j=1}^{N} \sum_{d=1}^{D} \alpha_{t-d}(j) a_{ji} p_i(d) \prod_{s=t-d+1}^{t} b_i(O_s)$$

Others can be calculated similarly. The relationships among α, α^*, β and β^* are

$$\begin{aligned}
\alpha_t^*(i) &= \sum_{j=1}^{N} \alpha_t(j) a_{ji} \\
\beta_t^*(i) &= \sum_{d=1}^{D} \beta_{t+d}(i) p_i(d) \prod_{s=t+1}^{t+d} b_i(O_s) \\
\alpha_t(i) &= \sum_{d=1}^{D} \alpha_{t-d}^*(i) p_i(d) \prod_{s=t-d+1}^{t} b_i(O_s) \\
\beta_t(i) &= \sum_{j=1}^{N} a_{ij} \beta_t^*(j)
\end{aligned}$$

Based on the above definitions and equations, Rabiner provided the following maximum likelihood re-estimation formulas (the Baum-Welch algorithm) for HMMD

$$\pi_i^{new} = \frac{\pi_i \beta_0^*(i)}{P(O|\lambda)} \tag{3.2}$$

$$a_{ij}^{new} = \frac{\sum_{t=1}^{T} \alpha_t(i) a_{ij} \beta_t^*(j)}{\sum_{j=1}^{N} \sum_{t=1}^{T} \alpha_t(i) a_{ij} \beta_t^*(j)} \tag{3.3}$$

$$b_i^{new}(k) = \frac{\sum_{\substack{t=1 \\ \text{s.t. } O_t=k}}^{T} \left\{ \sum_{r<t} \alpha_r^*(i)\beta_r^*(i) - \sum_{r<t} \alpha_r(i)\beta_r(i) \right\}}{\sum_{k=1}^{M} \sum_{\substack{t=1 \\ \text{s.t. } O_t=k}}^{T} \left\{ \sum_{r<t} \alpha_r^*(i)\beta_r^*(i) - \sum_{r<t} \alpha_r(i)\beta_r(i) \right\}} \tag{3.4}$$

$$p_i^{new}(d) = \frac{\sum_{t=1}^{T} \alpha_t^*(i) p_i(d) \beta_{t+d}(i) \prod_{s=t+1}^{t+d} b_i(O_s)}{\sum_{d=1}^{D} \sum_{t=1}^{T} \alpha_t^*(i) p_i(d) \beta_{t+d}(i) \prod_{s=t+1}^{t+d} b_i(O_s)} \tag{3.5}$$

More detailed explantions for the above formulas are shown in [38]. From the calculation of the $\alpha_t(i)$ term in the above equations, we can see that $\frac{D^2}{2}$ times more computational cost is required than the standard HMM. The following is the structure of this chapter. In the next two sections, derived from the above Rabiner's version and be consistent with his notation [38], we will introduce our more efficient implemention of HMMD. We begin in 3.2 that follows with a description of the Baum-Welch algorithm for HMMD. This is followed in 3.3 with a description of the Viterbi algorithm for HMMD. Finally in Section 3.4 we showed the wide applications of HMMD in various fields.

3.2 Baum-Welch algorithm for HMMD

Define the forward variable

$$\begin{aligned}\alpha_t(i,d) &= \begin{cases} b_i(O_t) \sum_{j=1,\, j\neq i}^{N} \hat{\alpha}_{t-1}(j) a_{ji} & \text{if } d = 1 \\ \alpha_{t-1}(i, d-1)\; b_i(O_t) & \text{if } 2 \le d \le D \end{cases} \\ &= \begin{cases} \check{\alpha}_{t-1}(i)\; b_i(O_t) & \text{if } d = 1 \\ \alpha_{t-1}(i, d-1)\; b_i(O_t) & \text{if } 2 \le d \le D \end{cases} \end{aligned} \tag{3.6}$$

where

$$\hat{\alpha}_t(i) = \sum_{d=1}^{D} \alpha_t(i,d) p_i(d) \tag{3.7}$$

$$\check{\alpha}_t(i) = \sum_{j=1,\, j\neq i}^{N} \hat{\alpha}_t(j)\; a_{ji} \tag{3.8}$$

It is easy to see the following relation

$$\alpha_t(i,d)\; p_i(d) = P(O_1 O_2 \cdots O_t, S_i \text{ ends at } t \text{ with duration of } d|\lambda)$$

A transition into state i with duration d takes place either from the same state i with duration $d-1$ in the pervious time index, if $d>1$; or from all other states that are allowed to transit to state i, if $d=1$. To save computational resource, the duration probability $p_i(d)$ is attached to the state only when the state is going to transit to other states, as shown is Eq.(3.7), where $\hat{\alpha}_t(i) = P(O_1 O_2 \cdots O_t, S_i \text{ ends at } t|\lambda)$.

Similarly define the backward variable

$$\begin{aligned}\beta_t(i,d) &= \begin{cases} b_i(O_t) \displaystyle\sum_{j=1,\, j\neq i}^{N} a_{ij}\hat{\beta}_{t+1}(j) & \text{if } d=1 \\ b_i(O_t)\ \beta_{t+1}(i,d-1) & \text{if } 2\leq d\leq D \end{cases} \\ &= \begin{cases} b_i(O_t)\ \check{\beta}_{t+1}(i) & \text{if } d=1 \\ b_i(O_t)\ \beta_{t+1}(i,d-1) & \text{if } 2\leq d\leq D \end{cases} \end{aligned} \tag{3.9}$$

where

$$\hat{\beta}_t(i) = \sum_{d=1}^{D} \beta_t(i,d)\ p_i(d) \tag{3.10}$$

$$\check{\beta}_t(i) = \sum_{j=1,\, j\neq i}^{N} a_{ij}\hat{\beta}_t(j) \tag{3.11}$$

and we have the following relation

$$\beta_t(i,d)\ p_i(d) = P(O_t O_{t+1}\cdots O_T|S_i \text{ has a remaining duration of } d \text{ at } t,\ \lambda)$$

A state i with a remaining lifetime of d durations at the current time index will stay at the same state i with a remaining lifetime of $d-1$ durations at the next time index, if its duration value $d>1$. Otherwise, if the duration $d=1$ at the current time index, it has to transit to other transition-allowable states at the next time index, as shown in Eq.(3.11).

Given the forward and backward variables as shwon above, now α, α^*, β and β^* in the

Rabiner's formulas can be expressed as

$$\alpha_t^*(i) = \check{\alpha}_t(i)$$

$$\beta_t^*(i) = \hat{\beta}_{t+1}(i)$$

$$\beta_t(i) = \check{\beta}_{t+1}(i)$$

$$\alpha_t(i) = \hat{\alpha}_t(i)$$

For convenience in what follows we now define

$$\begin{aligned}
\omega(t,i,d) &= P(O_1...O_T, q_{t-1} \neq S_i, q_t...q_{t+d-1} = S_i, q_{t+d} \neq S_i|\lambda) \\
&= \check{\alpha}_{t-1}(i)\ \beta_t(i,d)\ p(d) \qquad (3.12) \\
\mu_t(i,j) &= P(O_1...O_T, q_t = S_i, q_{t+1} = S_j|\lambda) \\
&= \hat{\alpha}_t(i) a_{ij} \hat{\beta}_{t+1}(j) \qquad (3.13) \\
\varphi(i,j) &= \sum_{t=1}^{T-1} \mu_t(i,j) \qquad (3.14) \\
\nu_t(i) &= P(O_1...O_T, q_t = S_i|\lambda)
\end{aligned}$$

In what follows use is made of the following relation (as in [56])

$$\begin{aligned}
&P(O_1...O_T,\ q_t = S_i, q_{t+1} = S_i|\lambda) \\
&\qquad = P(O_1...O_T, q_t = S_i|\lambda) - P(O_1...O_T, q_t = S_i, q_{t+1} \neq S_i|\lambda) \\
&\qquad = P(O_1...O_T, q_{t+1} = S_i|\lambda) - P(O_1...O_T, q_t \neq S_i, q_{t+1} = S_i|\lambda)
\end{aligned}$$

from which we get the following recursion formula

$$\nu_t(i) = \begin{cases} \hat{\alpha}_T(i) & \text{if } t = T \\ \nu_{t+1} + \sum_{j \neq i}^{N} (\mu_t(i,j) - \mu_t(j,i)) & \text{if } 1 \leq t \leq T-1 \end{cases} \tag{3.15}$$

Using the above equations, Eq. (3.2)-(3.5) can be expressed as

$$\pi_i^{new} = \frac{\pi_i \hat{\beta}_1(i)}{P(O|\lambda)} \tag{3.16}$$

$$a_{ij}^{new} = \frac{\varphi(i,j)}{\sum_{j=1}^{N} \varphi(i,j)} \tag{3.17}$$

$$b_i^{new}(k) = \frac{\sum_{t=1 \text{ s.t. } O_t=k}^{T} \nu_t(i)}{\sum_{t=1}^{T} \nu_t(i)} \tag{3.18}$$

$$p_i(d) = \frac{\sum_{t=1}^{T} \omega(t,i,d)}{\sum_{d=1}^{D} \sum_{t=1}^{T} \omega(t,i,d)} \tag{3.19}$$

A summary of the Baum-Welch training algorithm is as follows

1. initialize elements(λ) of HMMD.

2. calculate $\alpha_t(i,d)$ using *Eq.* (3.6) − (3.8).
   (save two tables: $\hat{\alpha}_t(i)$ and $\check{\alpha}_t(i)$)

3. calculate $\beta_t(i)$ using *Eq.* (3.9) − (3.11).

4. reestimate elements(λ) of HMMD using *Eq.* (3.16) − (3.19).

5. terminate if stop condition is satisfied, else goto step 2.

The memory complexity of this method is $O(TN)$. As shown above, the algorithm first does forward computing (step (2)), and saves two tables: one is $\hat{\alpha}_t(i)$, the other is $\check{\alpha}_t(i)$. Then at every time index t, the algorithm can group the computation in step (3) and (4) together. So no backward table needs to be saved. We can do a rough estimation of HMMD's computation cost by counting multiplications inside the loops of $\sum^T \sum^N$(which corresponds to standard HMM's computation cost.) and $\sum^T \sum^D$(the additional computational cost incurred by the HMMD). The computation complexity is $O(TNN + TND)$.

To avoid underflow problem, we apply a dynamic scaling process to keep the forward and backward variables within a manageable numerical interval. Basically, we keeps two set of emissions $b_i(k)$: one is scaled, the other is not. At every time index, if the numerical values become too small (test $\hat{\alpha}_t(i)$, for example), we make the emission pointer point to the scaled version, otherwise use the unscaled version. In this way no additional computation complexity is brought by the scaling.

3.3 Viterbi algorithm for HMMD

Define

$$\delta_t(i,d) = \begin{cases} b_i(O_t) \max\limits_{j=1, j\neq i}^{N} \{V_{t-1}(j)\ a_{ji}\} & \text{if } d = 1 \\ \delta_{t-1}(i, d-1)\ b_i(O_t) & \text{if } 2 \leq d \leq D \end{cases}$$

where

$$V_t(i) = \max_{d=1}^{D} \{\delta_t(i,d)\ p_i(d)\}$$

for which we have a useful recursive definition

$$\delta_t(i,d)\ p_i(d) = \text{the probability of the most probable path}$$
$$\text{that ends at state } i \text{ with duration of } d \text{ at time } t$$

The algorithm's goal is to find

$$\text{argmax}_{\,i,\,d}\,\{\delta_T(i,d)\ p_i(d)\}$$

Since a logarithm scaling can be applied to the terms $a_{ij}, b_i(k)$ and $p_i(d)$ in advance, the above equations become

$$\delta_t(i,d) = \begin{cases} \log b_i(O_t) + \max\limits_{j=1,\,j\neq i}^{N}\{V_{t-1}(j) + \log a_{ji}\} & \text{if } d = 1 \\ \delta_{t-1}(i,d-1) + \log b_i(O_t) & \text{if } 2 \leq d \leq D \end{cases} \tag{3.20}$$

where

$$V_t(i) = \max_{d=1}^{D}\{\delta_t(i,d) + \log p_i(d)\}$$

The goal simplifies to

$$\text{argmax}_{\,i,\,d}\,\{\delta_T(i,d) + \log p_i(d)\}$$

The complexity of Viterbi algorithm for HMMD is $O(TNN + TND)$. Because the logarithm scaling can be performed in advance, the Viterbi procedure consists only of additions, yielding a very fast computation.

3.4 Applications of HMMD

Hidden Markov models have been applied successfully to speech recognition field. Most modern automatic speech recognition (ASR) systems are based on modeling the phonemes with hidden Markov models. However, a major weakness is that the geometric duration distribution assumed by HMM for speech events may not be appropriate to represent the speech characteristics. In many scenarios, the duration of speech units (phonemes) plays an important role in speech recognition [5]. For examples, in many word pairs, such as "beat" vs "bit" or "ship" vs "sheep", the duration information is the only discriminating feature that can be used by the speech recognizer to differentiate these confusable words. Many researches have found that using HMM with explicit duration could improve the recogniton accuracy [2] [5] [41] [28]. Chen utilized the explicit duration HMM to model a prosody dependent speech recognizer with a resulting improved recognition accuracy [25]. Although the duration lengthening in phrase final syllable rhymes improved the acoustic modeling, the chosen Ferguson's HMM with duration algorithm as the actual implementer immediately brought a quadratic factor (D^2) of speed slow down. It is exact the purpose of this dissertation (details in Chapter 5) to present an efficent HMM with duration model– HMMBD– that has the same asymptotical computational complexity as the the standard HMM. As a result, the performance of the whole ASR system should not be affected when HMM with duration, instead of HMM, is used.

In the stock market, the durations of the regimes and the factors that affect these durations are of particular interest. Ntantanmis applied a Duration Hidden Markov Model to model regimes switches in the stock market, where typical regimes were "bull" regime and "bear" regime [37]. The duration of each state was set to be a random variable that depended on a set of exogenous explanatory variables, typically the short term interest rate and the interest rate spread. That is, the duration of each state was a function whose value

varies across times. The observations' emission distributions were assumed to be a finite mixture of Normal distribution, instead of only a single Normal distribution as generally used, in order to capture more potential complexities, such as asymmetrics. The model's governing parameters were estimated and then used to analyze and identify regimes in the market. The results showed that the bull market was identified as the state with a higher mean and a lower coefficient of variability of the observations' distribution; whereas, the bear market was identified as the state with a lower mean but a higher coefficent of variability of the observations' distribution. The factors' impacts to the duration of the stock regimes were also analyzed. For example, a higher short-term interest rate could has a negative effect for the duration of the state, when it was in a bull market regime. And a higher interest rate spread could stimulate the duration of the bull market. The quantitive esitimates of these impacts were presented. All these results could be valuable to policy makers before the decisions are issued.

Hauberg and Sloth modeled duration in hidden Markov models with a continuous variable to help control the actions of autonomous computer actors in a theatrical play [20]. The implementation of the resulting model was aided with the optimal particle filter. The purpose was to produce a theatrical play where an autonomous computer acts together with a human actor. The gesture recognition system based on "Motion History Images" was used to determine what the human actor was doing. The system then had an estimation of its current process position of the play in the human actor's manuscript. This information was then used to determine which action the computer controlled actor should follow by comparing the attained position information with the manuscript of the computer controlled actor. In the model, an human action was considered as a state. The order and duration of the state could be gained from the detailed description of the manuscript. Since the certainty of the duration of states could vary in the theatrical setting, a tuning parameter relating with the certainty was used to adjust the algorithm's sensitiveness to the input. In order to

determine which action the computer controlled actor should perform, the probability representing the likelihood was calculated. If the probability of this action was over the pre-set threshold (60%), it was performed. The result of this method showed that the computers knows very well when an action was to be performed.

Variable Duration HMMs are also used in Handwritting recognition [8]. First a Morphological segmentation algorithm used operations such as conditional operations, iterative operations , to translate the 2-D image into 1-D sequence of sub-charactor sysmbols. After the proper feature extraction, there was a vector of features, including moment features, zero-crossing features, regional and global topological features and major features representing the spatial distribution of pixels. The 26 letters of the alphabet were defined as the states of the model. The duration distributions were based on the number of times that a state was split into some number of parts in the segmentation procedure. The emission distribution could use the Mixture Gaussian distribution, whose training may use the support of a K-mean clustering algorithm that clusters the samples into groups for the states. The final Viterbi phase found the optimal state sequence given the sequence of obervations and parameters λ. Based on a likelihood criterion for identifying the words, the model classified the learning words. Experimental performed showed that a substantial improvement of the recognition accurate was achieved when using HMM with state duration, compared to HMM without duration incorporated.

Luhr used explicit state duration HMM for abnormality dection in sequences of human activity [44]. Activity duration plays an important cue in the accurate modeling of human behaviour, especitially among sequences where the order of observations is similar but the duration of activities are quite different. Video sequences of normal activities recorded in the kitchen scenario were classified into five classes: preparing cereal, making toast for breakfast, preparing dinner and cooking a bacon and eggs breakfast. A robust tracker segmented the motions and a Kalman filter was used to track moving objects between frames. The

position of subjects and relative boundaries of interst were calculated. The duration of the video sequences recorded were ranged from 30 to 300 seconds, with the mean 90 seconds. Another set of abnormal behaviours that had different activity durations were also recorded. Experiment results showed explicit state duration modelling were necessary for identifying the abnormality in activity duration. The explicit duration HMM used by the author was from [38], which requires a D^2 fold of computional cost, compared to the standard HMM. Use our version of HMMBD described in Chapter 5 , we can have a version of HMM with duration that has asymtotical the same computational complexity, while the performance of the system stays the same.

HMM with duration has also been applied to model Musical patterns. Pikrakis used a variable duration hidden Markov model as the classifer for the musical patterns [4]. The first stage of the processing was feature extraction. A fundamental frequencey tracking algorithm was utilized to extract a sequence of music intervals from the raw audio data. Each fundamental frequency was then quantized to the closest quarter-tone frequency and the difference was calculated. After the preprocessing, the resulting sequence was given as the input to the variable duration hidden Markov model. The note duration was models using a Gaussian function. Each of the variable duration hidden Markov model was trained to recognize the corresponding predefined class of music pattern. A modified Viterbi algorithm was used to classify the unknown music sequences based on the highest recognition probability. The classification results demonstrated the outperformance of HMM with duration over the conventional HMM.

Plasma etch is a critical process in semiconductor manufacturing. Ge extended the standard hidden Markov model to explict state duration semi-Markov model to investigate two statistical detection problems in Plasma etch endpoint detection: change-point detection and pattern matching [18]. The implicit duration in the standard HMM is geometric distribution, which is improper, since in reality other distributions such as log-normal may be

more realistic, so an HMM with duration incorporated is preferred. For the "change-point" detection, a 2-state segmental semi-Markov model was used: state 1 represents "before the change point" and state 2 represents "after the change point". The state duration distribution was set to reflect the prior knowledge about when the change-point would occur. For the pattern-based end-point detection, instead of a "change-point", a waveform indicated the endpoint. The piecewise linear segments were regarded as the states, the duration distribution of each state was modeled as a truncated normal distribution. The results showed that the proposed model was more accurate than other non-probabilitstic alternatives such as dynamic time-warping technique.

The original description of an explicit HMMD required computation of order $O(TNN + TND^2)$ [16], where T is the period of observations, N is the number of states, and D is the maximum duration of state transitions to self (where D is typically > 500 in gene-structure identification and channel current analysis [49]). This is generally too prohibitive (computationally expensive) in practical operations, and introduced a severe maximum-interval constraint on the interval-distribution model. Improvements via hidden semi-Markov models to computations of order $O(TNN+TND)$ were described in [39] [56], where the Viterbi and Baum-Welch algorithms were implemented, the latter improvement only obtained as of 2003. In these derivations, however, the maximum-interval constraint is still present (comparisons of these methods were subsequently detailed in [24]). Other HMM generalizations include Factorial HMMs [19] and hierarchical HMMs [42]. For the latter, inference computations scaled as $O(T^3)$ in the original description, and have since been improved to $O(T)$ by [36].

The above shown HMMD variants all have the computational inefficiency problem which limits their applications in real world. In this dissertation, we present a new algorithm: Hidden Markov Model with Binned Duration (HMMBD). Our HMMBD has its computation complexity of $O(TNN + TND^*)$, where D^* (typically < 50, and can be as small as 4 or 5) is the number "bins" used to group up consecutive durations. These "bins" are generated

by analyzing the state duration distribution and grouping together neighboring durations if their values are approaximate in measure. In this way, we take back computational resource from those insignificant parts and focus on those more significant ones. As a result, we now have an efficient HMM with duration model that can be applied in many areas that were originally thought impractical.

Chapter 4

The hidden Markov model with binned duration algorithm (HMMBD)

In this chapter, we will introduce our new method–Hidden Markov model with binned duration (HMMBD). The original idea of this method was proposed by my Ph.D. thesis advisor, Dr. Winters-Hilt, who also provided a lot of critical detail considerations such as the important push/pop operations used by HMMBD. With his extensive guidance through the process, I was able to have the details of HMMBD here and had a complete implementation of the algorithm. This HMMBD algorithm has just been published in IEEE transactions on signal processing [54].

We now start to introduce our new efficient HMM with duration method by analyzing the duration distribution, which is not incorporated into many common applications (automatic speech recognition system, for example) due to computational complexity if the explict duration is introduced [35]. The duration distribution of a state consists of rapidly changing probability regions and slowly changing probability regions. In the standard HMMD all regions share an equal computation resource (represented as D substates of a given state) − this can be very inefficient in practice. In this chapter, we describe a way to recover

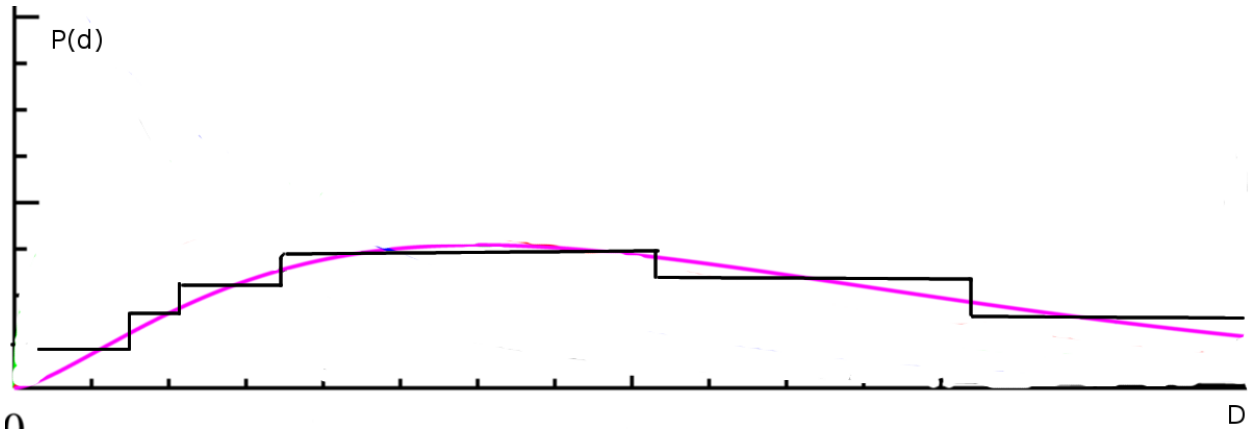


Figure 4.1: Use the step curve to simulate the real distribution curve

computational resources, during the training process, from the slowly changing probability regions. As a result, the computation complexity can be reduced to $O(TNN + TND^*)$, where D^* is the number of "bins" used to represent the final, coarse-grained, probability distribution. A "bin" of a state is a group of substates with consecutive durations. For example, $\alpha(i,d), \alpha(i,d+1), ...\alpha(i,d+\Delta d)$ can be grouped into one bin if their values meet the binning requirement. The bin size is a measure of the granularity of the evolving length distribution approximation. A fine-granularity is retained in the active regions,perhaps with only one length state per bin, while a coarse-granularity is adopted in weakly changing regions, with possibly hundreds of length states per bin. As shown in the below illustration, we use only six "bins" to represent for the whole duration distribution

Starting from this binning idea, for substates in the same bin, a reasonable approximation is applied

$$\sum_{d=d'}^{d'+\Delta d} \alpha_t(i,d)\ p_i(d) = p_i(\bar{d}) \sum_{d=d'}^{d'+\Delta d} \alpha_t(i,d)$$

where $\bar{d}$ is the duration representative for all substates in that bin. For different bins, the values of $\bar{d}$ and Δd may be differnt. From now on whenever they are mentioned, we mean those values associated with its bins.

We begin in Section 4.1 that follows with a description of the Baum-Welch using the

HMMBD algorithm This is followed by Section 4.2 with a description of the Viterbi using the HMMBD algorithm. And the heuristics for the training procedure are presented in Section 4.3. In Section 4.4, we provide an explanation of the code implementation of the algorithm. Finally the experimental results about the performance of HMMBD compared with HMM and HMMD are shown in Section 4.5.

4.1 Baum-Welch algorithm for HMMBD

First define a variable associated with each bin (which can be calculated recursively)

$$\hat{b}_t(i,n) = \prod_{t-\Delta d}^{t} b_i(O_t) \tag{4.1}$$

As mentioned above, Δd is the number of substates inside the bin. $\hat{b}_t(i,n)$ is the product of consecutive emission probabilities for state i, bin n at time t. In an actual implementation a scaling procedure will be applied on $b_t(O_t)$, so the underflow problem can be avoided. Based on the above approximation, the forward equations in (3.6)-(3.8) can be replaced by

$$\alpha_t(i,n) = \begin{cases} [\alpha_{t-1}(i,n) - \rho_t(i,n) + \breve{\alpha}_{t-1}(i)]\ b_i(O_t) & \text{if } n = 1 \\ [\alpha_{t-1}(i,n) - \rho_t(i,n) + \rho_t(i,n-1)]\ b_i(O_t) & \text{if } 1 < n \leq D^* \end{cases} \tag{4.2}$$

where

$$\hat{\alpha}_t(i) = \sum_{n=1}^{D^*} \alpha_t(i,n)\ p_i(\bar{d}) \tag{4.3}$$

$$\breve{\alpha}_t(i) = \sum_{j=1,\, j\neq i}^{N} \hat{\alpha}_t(j) a_{ji} \tag{4.4}$$

$$\rho_t(i,n) = Q(i,n).pop() * \hat{b}_{t-1}(i,n) \tag{4.5}$$

$$Q(i,n).push(\rho_t(i,n-1)) \tag{4.6}$$

The explanation begins with associating every bin with a queue $Q(i,n)$. The queue's size is equal to the number of substates grouped by this bin. At every time index, the oldest substate: $\alpha(i, d+\Delta d)$ will be popped out of its current bin and pushed into its next bin, as shown in (4.6), where $Q(i,n)$ stores the original probability of each substate in that bin when they were pushed in. So when one substate becomes old enough to move to the next bin, its current probability can be recovered by: first pop out its original probability, then multiply it by $\hat{b}_{t-1}(i,n)$, as shown in (4.5). On the other hand, as long as substates stay inside a bin, their invidual computations can be grouped into only one in the α term, as shown in (4.2).

Similarly, the backward equations in (3.9)-(3.11) can be replaced by

$$\beta_t(i,n) = \begin{cases} b_t(O_t)\ [\beta_{t+1}(i,n) - \rho_t(i,n) + \check{\beta}_{t+1}(i)] & \text{if } n = 1 \\ b_t(O_t)\ [\beta_{t+1}(i,n) - \rho_t(i,n) + \rho_t(i,n-1)] & \text{if } 1 < n \leq D^* \end{cases} \tag{4.7}$$

where

$$\hat{\beta}_t(i) = \sum_{n=1}^{D^*} \beta_t(i,n)\ p_i(\bar{d}) \tag{4.8}$$

$$\check{\beta}_t(i) = \sum_{j=1,\, j\neq i}^{N} a_{ij}\hat{\beta}_t(j) \tag{4.9}$$

$$\rho_t(t,n) = Q(i,n).pop() * \hat{b}_{t+\Delta d+1}(i,n) \tag{4.10}$$

$$Q(i,n).push(\rho_t(i,n-1)) \tag{4.11}$$

The explanation for backward procedure is similar to the one for forward procedure, since they are symmetric procedures. HMMBD uses the same re-estimating formulas as HMMD

except that ω term. For convenience, the re-estimating formulas are provided as follows

$$\omega(t,i,\bar{d}) = \breve{\alpha}_{t-1}(i)\ \beta_t(i,\bar{d})\ p(\bar{d}) \tag{4.12}$$

$$\begin{aligned}\mu_t(i,j) &= P(O_1...O_T, q_t = S_i, q_{t+1} = S_j|\lambda) \\ &= \hat{\alpha}_t(i)a_{ij}\hat{\beta}_{t+1}(j)\end{aligned} \tag{4.13}$$

$$\varphi(i,j) = \sum_{t=1}^{T-1} \mu_t(i,j) \tag{4.14}$$

$$\nu_t(i) = \begin{cases} \hat{\alpha}_T(i) & \text{if } t = T \\ \nu_{t+1} + \sum_{j\neq i}^{N}(\mu_t(i,j) - \mu_t(j,i)) & \text{if } 1 \leq t \leq T-1 \end{cases} \tag{4.15}$$

$$\pi_i^{new} = \frac{\pi_i \hat{\beta}_1(i)}{P(O|\lambda)} \tag{4.16}$$

$$a_{ij}^{new} = \frac{\varphi(i,j)}{\sum_{j=1}^{N}\varphi(i,j)} \tag{4.17}$$

$$b_i^{new}(k) = \frac{\sum_{t=1 \text{ s.t. } O_t=k}^{T} \nu_t(i)}{\sum_{t=1}^{T}\nu_t(i)} \tag{4.18}$$

$$p_i(d) = \frac{\sum_{t=1}^{T}\omega(t,i,d)}{\sum_{d=1}^{D}\sum_{t=1}^{T}\omega(t,i,d)} \tag{4.19}$$

It is easy to see that the computation complexity of HMMBD now is $O(TNN + TND^*)$. Again, we apply a dynamic scaling process to keep the forward and backward variables within a manageable numerical interval in real implementations. More explanation of the implementation details for the Baum-Welch algorithm are shown in Section 4.4.

4.2 Viterbi algorithm for HMMBD

Similarly the process of Viterbi using the HMMBD algorithm (with computation complexity $O(TNN + TND^*)$) is

$$\hat{b}_t(i) = \begin{cases} 0 & \text{if } t = 1 \\ \hat{b}_{t-1}(i) + \log b_i(O_t) & \text{if } 1 < t \leq T \end{cases} \tag{4.20}$$

$$\zeta_t(i) = \max_{j=1, j\neq i}^{N} \{m_{t-1}(j) + \hat{b}_{t-1}(j) + \log a_{ji}\} \tag{4.21}$$

$$\zeta_t(i) = \zeta_t(i) - \hat{b}_{t-1}(i) \tag{4.22}$$

$$\text{if } (B(i,n).front() \text{ is too old}) \ \{B(i,n).pop();\} \tag{4.23}$$

$$\text{while } (B(i,n).back() < \zeta_t(i)) \ \{B(i,n).removeback();\} \tag{4.24}$$

$$B(i,n).push(\zeta_t(i)) \quad 1 \leq n \leq D^* \tag{4.25}$$

$$t' = t + d(i,n) - 1 \quad 1 \leq n \leq D^* \tag{4.26}$$

$$\text{if } (B(i,n).front() + log(p_i(\bar{d})) > m_{t'}(i)) \ \{m_{t'}(i) = B(i,n).front() + log(p_i(\bar{d}));\} \tag{4.27}$$

The explanation and usage for the above relations are as follows

- First define the term $\hat{b}_t(i)$ as shown in (4.20). When a new substate transits (from one of other states) to state i at time t, we store (by pushing into the bin) the difference between this substate's current path probability and the term $\hat{b}_{t-1}(i)$, as shown in (4.22) and (4.25). Then if this substate's probability is needed at any future time $t' > t$, its path probability at time t' can be computed as "the originally stored difference" plus $\hat{b}_{t'}(i)$, as shown in the term $m_{t-1}(j) + \hat{b}_{t-1}(j)$ in (4.21). The benefit is that the add-computations on $\log b_i(O_t)$) in (3.20) are saved (in (4.20)).

- If a substate at $t-1$ has a duration $d-1$, then at time t, its duration increases to d. So at every time index, we check to see if there is a substate too old to stay in the

bin-$B(i,n)$, if so, it is be removed from the bin, as shown in (4.23).

- Before a new substate is pushed into the bin, we first scan the bin top-down and delete all substates whose path probability is less than the new substate. Because those deleted substates will never have a chance to be chosen as the "max" Viterbi path, as shown in (4.24) and (4.25). As a result, paths inside a bin are always kept sorted, and the "max" one is always the front element of the bin. Thus, in (4.27) we can directly apply $front()$ on each bin to get its "max". The advantage of (4.24) is that the complexity of the "while" loop is only $O(1)$. (Suppose if more than one comparison is needed on average for every time index, then the bins will soon shrink to empty, a contradiction.)

- The max probability path of $B_t(i,n)$ can be pre-determined at time $t - d(i,n) + 1$, where $d(i,n)$ is the duration length of first substate of $B_t(i,n)$. This is because all substates of the same state i get the same increment of $b_i(O_t)$ at very time index. Detailed steps are shown in (4.25), (4.26) and (4.27).

4.3 Training heuristics on length distribution representation

A variety of heuristics have been explored for the HMMBD training process. In initial efforts, for simplicity, we use a heuristic where the first 1/1000 of training on the fine-granularity D-states is performed and used to estimate the length distribution crudely, but enough to estimate some very slowly changing regions (where counts are sufficiently high for this to be trusted), we then conduct the next round of training on another 1/1000 of the training data, but now with slightly fewer states because a very small amount of granularity in states is introduced (where not all bins simply have a single length state). We then iterate this

granularization process, such that by the time 1/100 of the data has been examined the granularity reduction is accomplished, and the remaining 99% of training is performed at the optimized (dynamic mesh) granularity. All sums, recursions, max operations, etc., shift to expressions according to the granularity. In this way we get the best of all possibilities, an unconstrained-D HMMD that dynamically shifts to a strong representation of the length distribution information using only $D^* \ll D$ bins.

The next heuristic considered (and used in the experiment results section – Section 4.5) is based on evaluations of the mean-probability differences between adjacent length-distribution bins, i.e., a '*min_gap*' cutoff is introduced between the average probabilities of adjacent bins. If adjacent bins have mean bin probabilities that differ by less than '*min_gap*', then those bins are combined. For monotonically decreasing distributions, with no bin-gaps less than $min\_gap = 0.05$, this results in less than 20 bins in a solution, and for single-peaked distributions would give rise to less than $2 * (1/0.05) = 40$ bins in a solution. (In the Results we find that we typically obtain reductions to 4 or 5 bins, however, well below the bin-limit artifacts indicated.) The '*min_gap*' heuristic begins with the full length distribution and adiabatically imposes the '*min_gap*' constraint by incrementally imposing the '*min_gap*' cutoff. In the experiments described in Section 4.5 we start from $min\_gap * (1/10)$ and increment by tenths until the full $min\_gap * (10/10)$ cutoff is imposed. The original formulation of the heuristic followed the structured learning process indicated above, by alternating incremental '*min_gap*' coarsening of the length distributions with each round of the HMMBD learning process (with possible EM re-estimation of the coarse-grained length distributions). This incremental approach in the batch learning process is then trivially extendable to on-line learning (part of the motivation for its formulation), for use in situations where where slow drift in experimental conditions may result in a slowly changing set of the emission, transition, and length distribution variables (as occurs in channel current analysis [15]). Details about on-line HMMBD learning and its applications will not be discussed further in this

paper.

4.4 Implementation of HMMBD

A full implementation of the core HMMBD in the C programming language is presented in the Appendix. First we show some common parameters used in the algorithm

- seq[size]: this sequence array stores a data sequence of length- "size".
- nstate: this parameter sets the number of distinct states in the model.
- nvalue: this parameter sets the number of distinct observations per state.
- MAX_D[i]: this parameter defines the max duration length allowed for state i
- nb[i]: this parameter denotes the number of bins owned by state i
- len[i][n]: this array stores the number of substates inside "bin n of state i". That is, the number of individual (consecutive) durations this bin contains, which corresponds to Δd term in Section 4.1.
- pi[i]: this array defines the initial state probability of state i
- a[i][j]: this array defines the probability that state i transits to state j, no self-transition is allowed (i.e. $i \neq j$), which is represented in the below by the duration terms.
- p[i][n]: the duration probability for "bin n of state i".
- e[i][k]: this array defines the probability that state i emits observation k

In the forward algorithm, we first define the following arrays

- queue[i][d]]: this array store the original probability of the substate– "state i with duration d" – when it enters the a bin. This corresponds to $Q(i, n)$ in Section 4.1.

- bin[i][n]: this array represents "bin n of state i", which correponds to $\alpha(i, n)$ in Section 4.1.

- E[i][n]: this array memories the emissions for "bin n of state i", which corresponds to the equation (4.1) in Section 4.1.

- pos[i][n]: this pointer array denotes the position of the last substate of "bin n of state i" in the queue "queue[i][n]". This term helps implement the operation "pop()" in the algorithm in Section 4.1.

- hat_alpha[t][i]: This term corresponds to $\hat{\alpha}[t][i]$ in Section 4.1.

- check_alpha[t][i]: This term correponds to $\check{\alpha}[t][i]$ in Section 4.1.

- e2[i][k]: this is a transformation of $e[i][k]$ needed in the scaling process, i.e. e2[i][k] = e[i][k] * SCALOR

- scale[t]: this is a flag recording whether or not the scaling precedure is applied at time t

After initialization, the forward procedure runs the following big loop with three steps to build the forward table

```
for (t = 1; t < size; t++) {
            ...
    /* (1) calculate f_bins */
            ...
    /* (2) update E[i][n] */
            ...
    /* (3) scaling procedure */
            ...
```

```
}
```

In step (1), for each state, we first calculate check_alpha[t][i]

```
for(j = 0; j < nstate; j++)
    check_alpha[t][i] += hat_alpha[t-1][j] * a[j][i];
```

This corresponds to Equation (4.4) in Section 4.1. Then we push the new substate with duration of 1 into the queue

```
d = pos[i][nb[i]-1]+1;
if(d > MAX_D[i]) d = 0;
queue[i][d] = check_alpha[t][i];
```

$queue[i][d]$ corresponds to the $Q(i, n)$ in Section 4.1. Note this is a circular array, so one index pass the end of the array goes to the beginning the array. Then for each bin, we do the following calculations

```
for (n = 0; n < nb[i]; n++) {
    bin[i][n] += queue[i][d];
    d = pos[i][n];
    queue[i][d] *= E[i][n];
    bin[i][n] -= queue[i][d];
    bin[i][n] *= emission[i][k];
    hat_alpha[t][i] += bin[i][n]* p[i][n];
}
```

Here is the explantion. For each bin, the oldest substate (queue[i][d]) that has been popped out from the previous bin will be push into current bin

```
bin[i][n] += queue[i][d];
```

For the first bin, which doesn't have the previous bin, the new substate with duration of 1 as shown above will be pushed into it instead. When the oldest state is being popped out, its value will be recovered by

```
d = pos[i][n];
queue[i][d] *= E[i][n];
```

which corresponds to the Equation (4.5) in Section 4.1. Note $pos[i][n]$ denotes the position of the oldest state for that bin. With the following operations to remove the contribution of the popped out state and add the new emission contribution

```
    bin[i][n] -= queue[i][d];
    bin[i][n] *= emission[i][k];
```

Equation (4.2) in Section 4.1 now is done. Finally, the operation

```
for (n = 0; n < nb[i]; n++) {
    hat_alpha[t][i] += bin[i][n]* p[i][n];
}
```

corresponds to Equation (4.3) in Section 4.1, which is the probability that state i end at t with partial obeservations up to time t.

Step (2) is to update the emission cumulant array $E[i][n]$ for each bin

```
/* update E[i][n] */
for (i = 0; i < nstate; i++)
    for (n = 0; n < nb[i]; n++) {
        E[i][n] *= emission[i][k];
        if ((j = t-len[i][n]) >= 0) {
            int kk = seq[j];
            E[i][n] /= ((scale[j] == 0) ? e[i][kk] : e2[i][kk]);
```

```
        }
    }
```

As the time index t increses, the new emission should be added

```
        ...
E[i][n] *= emission[i][k];
        ...
```

And the old emission should be removed

```
if ((j = t-len[i][n]) >= 0) {
    int kk = seq[j];
    E[i][n] /= ((scale[j] == 0) ? e[i][kk] : e2[i][kk]);
}
```

Step(3) is the scaling process.

```
 /* scaling  test */
if (emission == e2) scale[t] = 1;
for (i = 0; i < nstate; i++)
    if (hat_alpha[t][i] > TOOSMALL) break;
emission = (i == nstate) ? e2 : e;
```

If a scaling is applied at this time column, then turn on the the scaling flag

```
if (emission == e2) scale[t] = 1;
```

This information will be needed later by the backward procedure. If all values of hat_alpha[t][i] is below the threshold "TOOSMALL", then a scaling will be applied at the next time index, otherwise no scaling will be applied

```
for (i = 0; i < nstate; i++)
    if (hat_alpha[t][i] > TOOSMALL) break;
emission = (i == nstate) ? e2 : e;
```

To save memory, the backward and Baum-Welch procedures are combined together in the function "backward_baumwelch". Similar to the forward procedure, terms such as $queue[i][d]$, $bin[i][n]$, $E[i][n]$ and $pos[i][n]$ are defined. After initialization and precomputing the forward procedure, the backward_baumwelch procedure runs the following loop with four steps to train the model.

```
   for (t = size-2; t >= 0; t--) {
        /* step (1): Baum-Welch: transition and emission */
        /* step (2): calculate b_bins */
        /* step (3): update E[i][n] and hat_beta[i] */
        /* step (4): Baum-Welch: duration */
    }
```

Step (1) calculates the expected number of times each transition and emission is used

```
for (i = 0; i < nstate; i++)
    for (j = 0; j < nstate; j++) {
        u[i][j] = hat_alpha[t][i] * a[i][j] * hat_beta[j];
        new_a[i][j] += u[i][j];
    }
```

corresponds to Equations (4.13) and (4.14).

```
for (i = 0; i < nstate; i++) {
    for (j = 0; j < nstate; j++)
        v[i] += (u[i][j] - u[j][i]);
```

```
    new_e[i][k] += v[i];
}
```

corresponds to Equation (4.15).

Step (2) is the backward procedure, similar to the forward procedure. For each state,

```
double check_beta = 0.0;
for(j = 0; j < nstate; j++)
    check_beta += hat_beta[j] * a[i][j];
```

corresponds to Equation (4.9), which is the substate with remaining duration of 1. Similar to the one in the forward procedure, inside each bin of state i,

```
d = pos[i][nb[i]-1]+1;
if(d > MAX_D[i]) d = 0;
queue[i][d] = check_beta;
```

pushes the new substate with remaining duration of 1 into the queue.

```
bin[i][n] += queue[i][d];
```

adds the substate that has been popped out of the previous bin. This corresponds to Equation (4.11) in Section 4.1.

```
d = pos[i][n];
queue[i][d] *= E[i][n];
```

recovers the substate that is going to be popped out by the bin. This corresponds to Equation (4.10) in Section 4.1.

```
bin[i][n] -= queue[i][d];
bin[i][n] *= emission[i][k];
```

removes the contribution of the popped out substate and adds the new emission contribution. Now the corresponding equation (4.7) in Section 4.1 is done. As time index t increases, the position of the last substate inside each bin should be updated

```
for (i = 0; i < nstate; i++) {
    for (n = 0; n < nb[i]; n++) {
        pos[i][n]--;
        if (pos[i][n] < 0) pos[i][n] = MAX_D[i];
    }
}
```

Note that the queue is a circular array.

Step (3) update the $E[i][n]$ as in the forward procedure,

```
E[i][n] *= emission[i][k];
if ((j = t + len[i][n]) < size) {
    int kk = seq[j];
    E[i][n] /= ((scale[j] == 0) ? e[i][kk] : e2[i][kk]);
}
```

And

```
for (i = 0; i < nstate; i++) {
    hat_beta[i] = 0.0;
    for (n = 0; n < nb[i]; n++) {
        ...
        hat_beta[i] += bin[i][n] * p[i][n];
        ...
    }
}
```

corresponds to Equation (4.8) in Section 4.1.

Finally, step (4) update the expected number of times each duration is used

```
for (i = 0; i < nstate; i++)
    for (n = 0; n < nb[i]; n++)
        new_p[i][n] += check_alpha[t][i] * bin[i][n];
```

corresponds to Equation (4.12) in Section 4.1. (The common duration factor $p(d)$ is not calculated yet, but will be computed later in the below.)

After the main loop, a new set of model parameters is computed (re-estimated)

```
    double sum = 0.0;
    for (i = 0; i < nstate; i++)
        sum += v[i];
    for (i = 0; i < nstate; i++) {
        pi[i] = v[i] / sum;
        if (pi[i] < MINIMUM) pi[i] = MINIMUM;
    }
```

corresponds to Equation (4.16) in Section 4.1, re-estimating the state priors.

```
    for (i = 0; i < nstate; i++) {
        sum = 0.0;
        for (j = 0; j < nstate; j++)
            sum += new_a[i][j];
        for (j = 0; j < nstate; j++) {
            a[i][j] = new_a[i][j] / sum;
            if (a[i][j] < MINIMUM) a[i][j] = MINIMUM;
        }
    }
```

```
for (i = 0; i < nstate; i++)
    a[i][i] = 0.0;  // no self-transition is allowed.
```

corresponds to Equation (4.17) in Section 4.1, re-estimating the state transitions.

```
for (i = 0; i < nstate; i++) {
    sum = 0.0;
    for (n = 0; n < nb[i]; n++) {
        new_p[i][n] *= p[i][n];
        sum += new_p[i][n];
    }
    for (n = 0; n < nb[i]; n++) {
        p[i][n] = new_p[i][n] / sum;
        if (p[i][n] < MINIMUM) p[i][n] = MINIMUM;
    }
}
```

corresponds to Equation (4.19) in Section 4.1, re-estimating the duration densities.

```
for (i = 0; i < nstate; i++) {
    sum = 0.0;
    for (k = 0; k < nvalue; k++)
        sum += new_e[i][k];
    for (k = 0; k < nvalue; k++) {
        e[i][k] = new_e[i][k] / sum;
        if (e[i][k] < MINIMUM) e[i][k] = MINIMUM;
    }
}
```

corresponds to Equation (4.18) in Section 4.1, re-estimating the emission probabilities.

In the Viterbi algorithm, we first define the following parameters

- E[i]: this is a cumulant array, storing all emissions of state i up to current time index.
- qbin[i]: this array records the bin that has the largest number of substates of state i.
- qsize[i]: the value of the above "largest number of substate".
- queue[i][p]: the queue for state i that is big enough to contain all substates of the largest bin of state i.
- qtime[i][p]: this array records the beginning time index of each substate in the queue[i][].
- pointer[i][n]: this array denotes the position of the "max" value of each bin.
- shift[i][n]: this array denotes the duration length of the first substate of each bin plus two.
- state[t][i]: this is the Viterbi state table.
- start[t][i]: this array records the start time (duration = 1) for state i

After initialization, Viterbi algorithm use the following loop to build the Viterbi table

```
for (t = 1; t < size; t++) {
    for (i = 0; i < nstate; i++) {
           ...
        /* Step (1): generate a new substate with duration=1 */
           ...
        /* Step (2): insert this new substate */
           ...
        /* Step (3): adjust pointers if necessary */
```

```
            ...
        /* Step (4): update "the most probable" pathMAX_D
            ...
    }
            ...
    /* Step (5) update E[i]
            ...
}
```

Step (1) generates a new substate with duration by choosing one from all other states that are able to transit to the current state

```
double max = -DBL_MAX;
int s =0;
for (j = 0; j < nstate; j++) {
    double tmp = state[t+1][j] + E[j] + log_a[j][i];
    if (tmp > max) {
        max = tmp;
        s = j;
    }
}
```

This corresponds to Equation (4.21) in Section 4.2. Note in the "state" array, the index is $t+1$, not state $t-1$;. This is because the "shift" array defined above adds two more units, in order to use the not yet unused memory at the time $t+1$ column.

```
state[t][i]  = s;
start[t][i] = start[t+1][s];
```

records the state transition information for backtracking uages.

```
max -= E[i];
```

calculates the relative differece between "max" and the current $E[i]$. This corresponds to Equation (4.22) in Section 4.2.

Step (2) insert this new substate into each bin. Since here all bin's queues are presented by one queue, only one pass of insertion is required.

```
int p = qtail[i];
while (p != pointer[i][qbin[i]] && queue[i][p] < max)
    if (--p < 0) p = qsize[i] - 1;
if (++p == qsize[i]) p = 0;
queue[i][p] = max;
qtime[i][p] = t;
qtail[i] = p;
```

The queue is always sorted in an increasing order. First find the tail of the queue, which contains the smallest value

```
int p = qtail[i];
```

Scan from the tail to head of bin, as long as the indexed value is smaller that the new value, remove that value. This is because the new value is "younger" than all other values, so all other values that are smaller than the new value will never have a chance to be chosen as the max by the Viterbi algorithm

```
while (p != pointer[i][qbin[i]] && queue[i][p] < max)
    if (--p < 0) p = qsize[i] - 1;
```

Note the queue is a circular array.

```
if (++p == qsize[i]) p = 0;
```

is a remedy to the "over-scanning". The above operations correspond to Equation (4.24) in Section 4.2.

```
queue[i][p] = max;
qtime[i][p] = t;
qtail[i] = p;
```

adds the new value at the proper position, updates its time index and resets the new tail to point to the new value. This corresponds to Equation (4.25) in Section 4.2.

Step (3) help adjust pointers that pointer to the "max" vaule for each bin.

```
if (queue[i][pointer[i][n]] < max) {
    pointer[i][n] = p;
}
```

If the new value is larger than the "max" of bin[i][n], the new value now becomes the "max" of the bin. This corresponds to Equation (4.24) in Section 4.2.

```
if (t - qtime[i][pointer[i][n]] >= len[i][n]) {
    pointer[i][n]++;
    if (pointer[i][n] == qsize[i]) pointer[i][n] = 0;
}
```

If the "max" value is too old, then since the queue is always kept sorted, the new "max" is just after the old "max" value. This corresponds to Equation (4.23) in Section 4.2.

Step (3) update the most probable path.

```
for (n = 0; n < nb[i]; n++) {
    int index = t + shift[i][n];
    if (index < size+2) {
        double tmp = queue[i][pointer[i][n]] + log_p[i][n];
```

```
        if (tmp > state[index][i]) {
            state[index][i] = tmp;
            start[index][i]  = qtime[i][pointer[i][n]];
        }
    }
}
```

where queue[i][pointer[i][n]] is the "max" value of bin[i][n] and log_p[i][n] is the duration probability in logarithm. The "state" and "start" array are for backtracking purpose. The "max" value of bin(i,n) at time t can be pre-determined at time $t - d(i, n) + 1$, where d(i, n) is the duration length of first substate of Bt (i, n). This is because all substates in bin(i,n) of time t get the same emission increment since time $t - d(i, n) + 1$. This corresponds to Equation (4.27) in Section 4.2.

Step (4) is to update E[i], which corresponds to Equation (4.20) in Section 4.2.

```
for (i = 0; i < nstate; i++)
    E[i] += log_e[i][seq[t]];
```

4.5 Experiment Results

For the experiments described here we consider scenarios that occur in nanopore detector channel current analysis [51] [49] [9] [45]. In preliminary explorations three parameterized distributions were examined: geometric, Gaussian, and Poisson. Distributions both segmented and "messy" were also examined. In all cases HMMD and HMMBD performed comparably. In many tests HMMBD, with $min\_gap = 0.05$ coarsening on length distributions, arrived at very few states, often just $D^* = 3$ or 4, with performance still comparable to the exact HMMD (96.6% correct decoding for the HMMBD with $D^* = 3$ compared to 96.8% correct decoding for the exact HMMD). This remarkably improved performance compared

to the HMM performance (61% correct) is attributed to the benefit of having a better fit to the "tail" of the distribution (i.e., the tail bin described earlier).

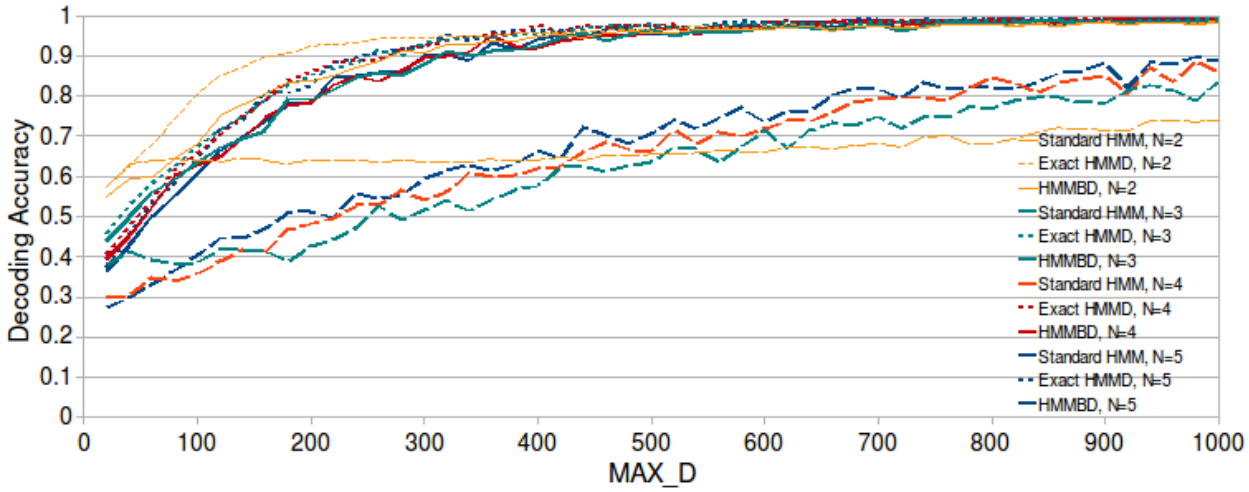


Figure 4.2: Decoding Accuracy of Exact HMMD, HMMBD, and Standard HMM. The N=2 case is a special degenerate case (when you leave one state, there is only one other state to go to), so its behavior is notably different (the light orange lines). For N=3, 4, or 5, the clear decoding improvements of HMMD/HMMBD over HMMs is readily apparent (where long-tailed distributions are used in the data generation described in (4.3). The behavior for N=10, 25, and 50 was also examined (not shown) and was found to be very similar to that shown above where the comparisons on the now small test-sets described in (4.3) could be trusted − i.e., there was good comparison for the lower MAX_D values, while the higher N higher MAX_D experiments require larger datasets for true comparative studies and are not pursued further in this study.

The preliminary results described above motivated the set of tests shown in Figures 4.2-4.4. In this set of experiments a much larger dataset is considered: Sequences are generated that are 100,000 samples long, with 100 sequences generated for training, 5 for testing. This data generation is repeated for each specification of MAX_D or of state number, etc. The number of states considered ranges from 2 to 50 (with only 2-5 shown in the figures). For the three-state system, the states are described as occupying blockade levels with means at 11, 12, and 13 pA. All states have Gaussian emission with standard deviation 6 pA. All states have as length distribution the student-t distribution (a heavy-tail distribution) with mean=$MAX\_D/10$. For the four-state system we have means set at 11, 12, 13, and 14,

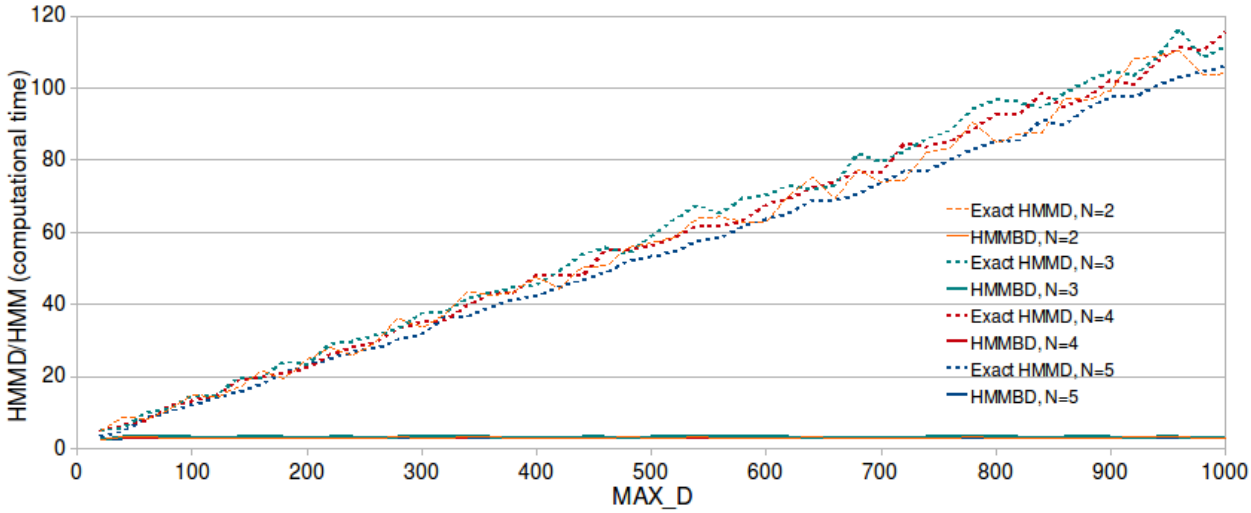


Figure 4.3: Baum-Welch (EM) Processing Speed. The computational time required to complete the Baum-Welch algorithm for the Exact HMMD and HMMBD algorithms, in terms of comparison to the HMM decoding time on the same data. The HMMBD algorithm is shown to not scale with MAX_D, and has asymptote at 2.4 times the computational time of the standard HMM.

generalizing similarly for the other multi-state experiments.

In Figure 4.2 we show that the decoding accuracy of the HMMBD algorithm, that is using the simple *min_gap* heuristic, is comparable to the much more computationally expensive exact HMMD. Both HMMD and HMMBD methods improve from the HMM's 60% decoding accuracy to approximately 95%, and this holds true for the 2, 3, 4, and 5-state systems shown, and over a large range of MAX_D (100 to 500). Overall, a 15% to 35% improvement is observed when switching from HMM modeling to HMMD. This vast improvement is critically needed in applications such as channel current experiments, where the decoding directly tracks molecular state and, thereby, directly confers kinetic information [16]. The need for an HMMD is also important in gene structure efforts as well. For our new method it is critical that the HMMD method be fast, comparable to HMM processing speeds, and this is what is demonstrated for the HMMBD algorithm in Figures 4.3 and 4.4.

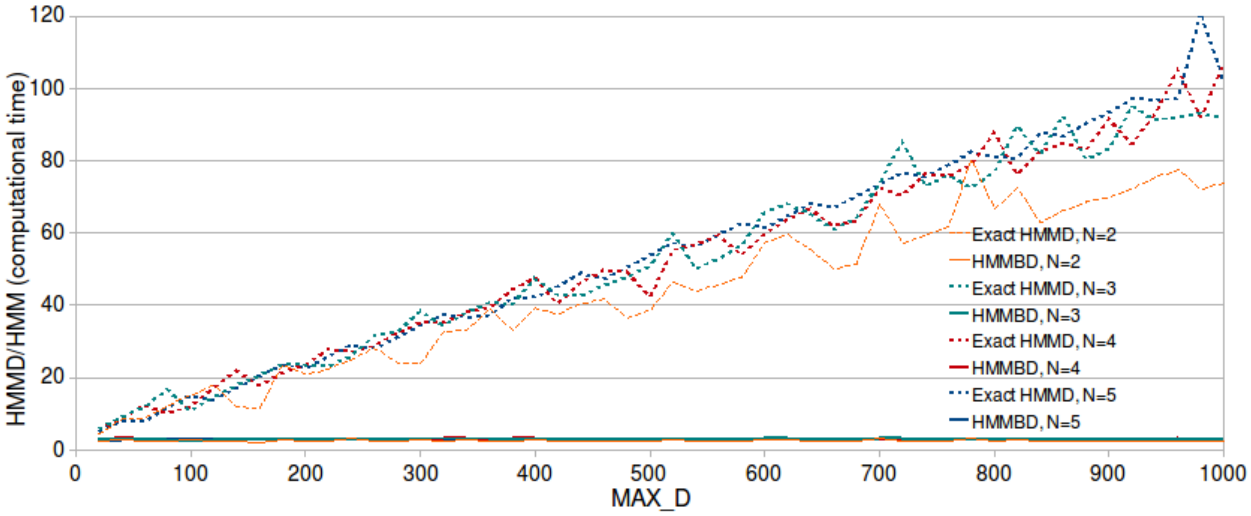


Figure 4.4: Viterbi Processing Speed. The computational time required to complete the Viterbi algorithm for the Exact HMMD and HMMBD algorithms, in terms of comparison to the HMM decoding time on the same data. The HMMBD algorithm is shown to not scale with MAX_D, and has asymptote at 2.5 times the computational time of the standard HMM.

Chapter 5

Application of HMMBD to Gene identification

Hidden Markov model is a stochastic model that has a strong capability to capture the statistical properties of large amount of observed data even though their exact theory may be unknown. Nowadays, large amount of gnome sequences are become avaiable. To extract biological knowledge from the data so as to explain the protein function, interaction between cells and many other mechanisms make the gene finding one of the most important research work in computational biology. During the past years, HMM-based approaches have gained a big success in solving the problems of gene finding. Many effective tools have been developed for the purpose of gene finding. Among these successful ones are Genie, GeneID, GeneMark and so on. Currently, gene finding tools have a good performance at indentifying regions as coding and non-coding, but are still don't have a high accuracy at finding the exact boundaries of exons.

This chapter presents the implementation of HMMBD to the gene identification application. The problem with gene finding is that given a sequence of DNA, determine the locations of genes, which are the regions that contains coding information for proteins. We

first privides a basic background in molecular biology in Section 5.1. Then we lay out the specification details for HMMBD in the application in Section 5.2. Following the suggestions from my Ph.D. thesis advisor, Dr. Winters-Hilt, three kinds of different emission mechanisms are used: the normal emission regions, the position-dependent emission regions and the hash-interpolated regions (or called zone regions when the order is fixed). Our in-process paper [52] use HMMD as the platform as the platform for incorporation of these position and zone dependent emissions. Finally we present the experiment results and discuss the future work in Section 5.3 and Section 5.4.

5.1 biology background

In molecular biology, deoxyribonucleic acid (DNA) is often considered as the blueprint of life, which contains the genetic intructions specifying the develpment and functioning of living organisms and some viruses. DNA is a long polymer made of nucleotides. A nuceotides is composed of a nucleobase, a five-carbon sugar and one to three phospate groups. Thare are four nucleobases: adenine (A), cytosine (C), guanine (G) and thymine (T). The four bases are classified into two types: adenine and guanine are purines while cytosine and thymine are pyrimidines. In living organisms, a pair of DNA molecules is ususally entwined together to form the shape of a double helix. The asymmetry of phosphodiester bonds (linkage between the $3'$ carbon atom of one sugar molecule and the $5'$ carbon of another) can be used to denote the direction of a DNA strand. The two direction-opposite strands have the complementary base pairing property. Adenine can only binds to thymine (and vice versa), and guanine can only binds to cytosine (and vice versa). As a result, one base strand can be deduced from the other, so conventionally when recording a DNA sequence, we only write down one strand left to right, from $5'$ to $3'$. We usually refer to the $5'$ side as the "upstream" side and the $3'$ side as the "downstream" side. In reality, the twisted and coiled three-dimentional structure

of the double-stranded DNA is so complex that research work in molecular biology generally focuses on the nucleotides sequences.

There are three major information-carrying biopolymers in the living organisms: DNA, RNA, and protein. Similar to DNA, Ribonucleic acid (RNA) is also nucleic acids, but is usually single-stranded. RNA nucleotides contains ribose not deoxyribose that DNA contains and RNA has the base uracil (U) in place of thymine (T) of DNA. While DNA is the genetic information carrier, it is the proteins (which are organic compounds made of amino acids) who perform the real work for the cell functions or serve as the building blocks. The central dogma of molecular biology laid out by Francis Crick states the relationship between DNA, RNA and proteins. There are total $3 * 3 = 9$ conceivable direct transfers of information flow between these the above three biopolymers. The dogma classifies them into 3 groups

- general transfers: DNA $\rightarrow$ DNA, DNA $\rightarrow$ RNA, RNA $\rightarrow$ protein
- special transfers: RNA $\rightarrow$ DNA, RNA $\rightarrow$ RNA, DNA $\rightarrow$ protein
- unknown transfers: protein $\rightarrow$ DNA, protein $\rightarrow$ RNA, protein $\rightarrow$ protein

The unknow transfers are believed never to occur, which means that information cannot be transferred back from protein to either protein or nucleic acid. Special transfers only occur at rare cases, which generally can be neglected. It is the general transfers that describe the normal biological infromation flows. That is, DNA can be copied to DNA (DNA replication), DNA information can go to messenger RNA (transcription), and messenger RNA can be used a template to synthesize proteins (translation).

The process to code a protein follows the below information flow

$$\text{DNA} \rightarrow \text{RNA} \rightarrow \text{protein}$$

In the first step, the two strands of DNA containing the protein coding information unzip

from each other and a single strand of messenger RNA (mRNA) is manufactured by pairing up mRNA bases with the exposed DNA bases. This process is called transcription ($DNA \rightarrow RNA$). After the mRNA is assembled, it finds its way to a ribosome, where each set of three mRNA bases (called codon) specifies an amino acid through the help of a transfer RNA (tRNA). (Some codons are only the indicators of the start or end of the coding process and so on.) After the process, the protein is generated. This step ($RNA \rightarrow protein$) is called translation.

A DNA molecule has many genes. A gene is a segment of DNA that codes for the synthesis of a specific protein as gene. A gene for a protein starts with a start codon, then has a number of condons coding for amino acids, and ends with a stop codon. In eukaryotes, genes consist of coding segments (exons) which are delimited internally by special, intragenic, non-coding segments (introns). In the splicing process, these introns will be removed and the resting exons are re-connected together, which then can be used for protein translation. The intergenic, non-coding regions of bases outside the genes are generally referred to as junks. The problem of gene finding is to indentifying those coding regions from non-coding regions.

5.2 Specifications of HMMBD in gene identification

Hidden Markov model is very common in gene identification nowadays. Churchill are among the first to introduce HMMs to analysis the genome structure [1]. Different regions of DNA sequences had very different nucleotide occurrence frequecies. These statistical features are very suitable for HMMs to model with the different emission probabilities for each region types. Reese [34] used the HMM divided the DNA sequences into segment states such as exons, introns, junk and so on. We will start with our specificiations for these different regions. Exons has 3-base codons, so a gene's cumulative exon string length must be divisible

by 3. The term "reading frame" is used to denote one of the 3 possible position – 0, 1 or 2 by our convention – relative to the start of a codon. Introns can interrupt the exons at any frame and introns don't have the notion of framing. But for the tracking convenience, we also associate the framing to introns, depending on the previous exon's ending frame information. The purpose is that when the current intron meets the next exon, this next exon is able to know which frame to continue with based on the intron's framing information. There is no framing notion for junks, since we know the exons of gene start with frame 0 and end with 2. The following are some illustrating examples when the above notations are used

$$jjj\cdots je_0e_1e_2\cdots e_0i_0i_0i_0\cdots i_0i_0e_1e_2\cdots e_0e_1e_2jjj\cdots j$$

$$jjj\cdots je_0e_1e_2\cdots e_1i_1i_1i_1\cdots i_1i_1e_2e_0\cdots e_0e_1e_2jjj\cdots j$$

$$jjj\cdots je_0e_1e_2\cdots e_2i_2i_2i_2\cdots i_2i_2e_0e_1\cdots e_0e_1e_2jjj\cdots j$$

Encoding for proteins can be found in both DNA strands, we follow the settings in [6] and [50] to identify genes in both directions simultaneously in one pass. The advantage is that it helps avoid the interpretation of the coding segmentation from the other direction as the junk, which is may be very possible if a two-pass method is used instead. The junks are the common region shared by both directions. We also take the settings in [6] and [50] to use the same set of estimated probabilities of occurrence of base sequences for both the forward and reverse strands' segment types and so on–exon, intron or junk, since the differences among such probabilities are negligible. We further break the one emission table generally used by a state into several different emission tables based on the relative distance of the bases to the state transition point. The following explains the purpose and details.

The vicinity around the transitions between exon, intron and junk usually contains rich

information for gene identification. For example, junk to exon transition ususally starts with a start codon ATG. Similarly in the exon to junk transition, one of the stop codons such as TAA, TAG and TGA generally could be detected. Nearly all eukaryotic nuclear introns start with GT and end with with AG (the AG-GT rule). Such exon-intron and intron-exon transitions are ususally referred to as the splice donor site and the splice acceptor site. To capture the information at those transition areas, we especially build a position-dependent emission (pde) table for these small amount of bases around each type of the transition point. It is called "position-dependent" since we make estimation of occurrence of the bases (emission probabilities) in this area according to their relative distances to the respective state transition point. For example, the start codon – ATG – is the first three bases at the junk-exon transition, not at other shifted positions. The size of this small region is determined by a window size parameter. We use four transition states to collect such position-dependent emission probabilities

$$ie, je_0, ei, e_2j.$$

Considering the framing information, we can expand the above four transition into eight transitions

$$i_2e_0, i_0e_1, i_1e_2, je_0, e_0i_0, e_1i_1, e_2i_2, e2j.$$

We make i_2e_0, i_0e_1, i_1e_2 share the same ie emission table and e_0i_0, e_1i_1, e_2i_2 share the same ei emission tables, since they only have the notation differences. Since we process both the forward-strand and reverse-strand gene identifications simultaneously in one pass, there are another corresponding set of eight state transitions for the reverse strand when running the Viterbi decoding procedure. Also there are anther eight position-dependent emission tables for the states of the reverse strand. Based on the training sequences' properties and

the size of the training data set, we may adjust the windown size and use different Markov high orders to calculate the estimated occurrence probabilities for different bases inside the window.

Regions a litter far way from the state transition point but not too far away may contain some special kinds of signals such as the promoter that plays an important cue in gene finding. (A promoter is a region of DNA that facilitates the transcription of the gene, which are typically located near the genes they regulate.) So the statistical properties in these regions may be different from other common regions, it may be benificial to separately build emission tables for these regions. The signals in these areas can be very diverse and their exact relative positions may be hard to nail down. We apply a "hash-interpolated" mechanism to to train the emission probabilies in these area. We use the hash table data structure to store the emission occurrences this region. The size of the region extends from the end of the last paragraph's "position-dependent" emission table's coverage to "distance" specified by a paramenter. The emission training starts from a low Markov order to collect the base sequence occurrence probabilies. Most occurrences of the train bases will have a moderate occurrence frequencies when an appropriate Markov order is reached. The emission training for these cases then stops when the number of their occurrences in the training samples drops under a preset threshold. But those strong signal in the regions may still keep having the intensed occurrence frequencies. We then push the Markov order of these signal to even higher Markov order, and start the next round of emission training for these strong signals until their occurrence counts also drop down the threshold value. In this way, we can push the Markov order of the signals to higher order based on their "strength" while keeping those common occurrences of base sequence in a low Markov order. Through the use of hash-interpolated process, we have the flexibility to assign different appropriate Markov order to different occurrence of base sequence based on their properties and the size of the traning data set. We build eight such hash tables to contain the corresponding emission

informations

- ieeeee: exon emission hash table for the right area that are some distance away from the intron-exon transition point.
- jeeeee: exon emission hash table for the right area that are some distance away from the junk-exon transition point.
- eeeeei: exon emissions hash table for the left area that are some distance away from the exon-intron transition point.
- eeeeej: exon emissions hash table for the left area that are some distance away from the exon-junk transition point.
- eiiiii: intron emissions hash table for the right area that are some distance away from the exon-intron transition point.
- iiiiie: intron emissions hash table for the left area that are some distance away from the intron-exon transition point.
- ejjjjj: junk emissions hash table for the right area that are some distance away from the exon-junk transition point.
- jjjjje: junk emissions hash table for the left area that are some distance away from the junk-exon transition point.

Note that the above "distance" is determined by the parameters. Again, we build another corresponding set of eight hash tables for state of the reverse strand. Our experiment results showned about 2% performance improvement when the above "hash-interrpolating" regions are separated from the common regions.

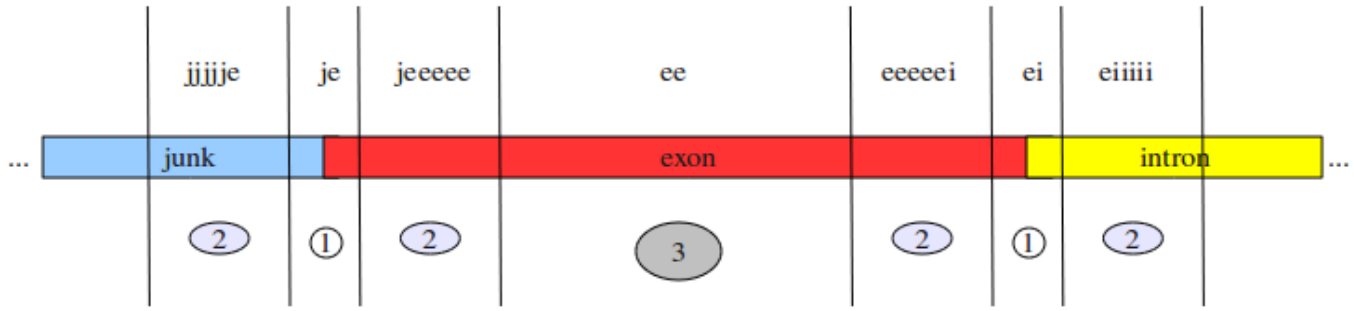


Figure 5.1: Three kinds of emission mechanisms: (1) position-dependent emission; (2) hash-interpolated emission; (3) normal emission.

Finally, for those regions that are far away from state transition point, that is, in the middle of a state, we build the normal emission tables with a suitable Markov order. (Basically we use the array data structure.) There are three such emission tables for both the forward and reverse strand states, corresponding to the normal exon emission table, the normal intron emission table and the normal junk emission table.

As a summary of these three kinds of emission processing, we provide an example of the illustration in Figure 5.1. As shown in the figure, based on the relative distance from the state transition point, we first encounter the position-dependent emissions (1), then we use the hash-interpolated emissions (2), finall travel to the normal state emissions (3).

Matching to the above different emission tables, The model contains the following 27 states in total for each strand

- Three ieeeee states: corresponding to the ieeeee emission table, with different reading frames at the time index t.
- Three jeeeee states: corresponding to the jeeeee emission table, with different reading frames at the time index t.
- Three eeeeei states: corresponding to the eeeeei emission table, with different reading frames at the time index t.

- Three eeeeej states: corresponding to the eeeeej emission table, with different reading frames at the time index t.
- Three eeeeee states: corresponding to the eeeeee emission table, with different reading frames at the time index t.
- Three eiiiii states: corresponding to the eiiiii emission table, with different reading frames of the intron. Here we associate the frames to intron for the purpose of tracking convenience.
- Three iiiiie states: corresponding to the iiiiie emission table, with different reading frames of the introns. Here we associate the frame to intron for the purpose of tracking convenience.
- Three iiiiii states: corresponding to the intron emission table, with different reading frames of the introns. Here we associate the frame to intron for the purpose of tracking convenience.
- one ejjjjj state: corresponding to the ejjjjj emission table.
- one jjjjje state: corresponding to the jjjjje emission table.
- one jjjjjj state: corresponding to the jjjjjj emission table.

Also there are another set of correponding reverse-strand states, with junk as the shared state. We don't assign corresponding states to the position-dependent emission table. When a state transion happened (junk to exon, for example), The positional-dependent emissions inside the window (je) will be absorbed first, then the state travels to the hash-interpolated emission area (jeeeee), then travels to the state of the normal emission region (eeeee), finally travels to another state of hash-interpolated emission region (eeeeei or eeeeej), and finally

may transit to another state. As usual, the duration information of each state is represented by the corresponding bin assigned by the algorithm. For convenience of the emission calculating in the Viterbi decoding, we precompute a cumulant emission tables for each of 54 substates (states of the forward and reverse strand) above, then as the state travels, its emission contritions at any span can be added up with a difference between two index of the cumulant array.

The occurrence of a stop codon (TAA, TAG or TGA) that is in reading frame 0 and located inside an exon, or across two exons because of the intron interruption, is called as an "in-frame" stop. In general the occurrences of in-frame stops are considered very rare in living organisms. We designed a in-frame stop filter to penalize those Viterbi paths containing stop codons. A DNA sequence has totally six reading frames (read in six ways based on frames), three for the forward strand and three for the reverse strand. When precomputing the emission tables in the above for the substates, for those substates related to exon, we consider the occcurrences of the in-frame stop codons in the six reading frame possibilities. For each reading frame possibility, we scan the DNA sequence from left to the right, whenever a stop codon is encounterred in that reading frame, we add to the emission probability of that position a user defined log_stop_penalty factor. In this way, the in-frame stop filter procedure is incorporated into the emission table building process and does not bring the addtional computational complexity to the whole program.

The algorithmic complexity of the whole program is $\theta(TND^*)$ where $N = 54$ substates and D^* is the the number of bins for each substates. The memory complexity is $\theta(TN)$. These complexities follow from the complexities of the general HMMBD algorithms.

5.3 Experiment results

The whole program for this application is written in the C programming language. GNU Compiler Collection (GCC) is used to compile the codes. The Operating system used is Ubuntu/Linux, runing on the lab's server with 8GB RAM. In general, the measure of predication performance is taken at both individual necleotide level and the full exon level, according to the specification by Burset and Guigo [7]. Following [7], we calculate both sensitivity (Sn) and specificity (Sp) of the predication and take their average as our final accuracy rate. The following paragraph taken from the authors' paper very well explains the intent of measuring at both levels

"Evaluation at the exonic structure level provides complementary information about the accuracy of the programs compared to that provided by evaluation at the coding nucleotide level. At the coding nucleotide level we are measuring how well the sequence coding regions are locatedthe search by content component of the gene structure prediction programswhile at the exonic structure level, we are measuring how well the sequence atomic signals (splice sites and start and stop codons) are identifiedthe search by signal component of the programs. High accuracy at the coding nucleotide level does not necessarily imply high accuracy at the exonic structure level. For instance, a program can have high sensitivity at the coding nucleotide level, because most of the coding nucleotides have been identified, but very low sensitivity at the exonic structure level, because most splicing signals have been incorrectly predicted [7]."

For accuracy at the base or nucleotide level, Sn is the proportion of actual coding nuecleotides that have been correctly predicated as coding, and Sp is the proportion of predicted

coding nucleotides that are the actual coding nucleotides

$$Sn = \frac{TP}{TP + FN}$$
$$Sp = \frac{TP}{TP + FP}$$
$$SnSp\_avg = \frac{sn + sp}{2}$$

where

TP = true positive count (the actual coding that are correctly predicted)

FN = false negative count (the actual coding that are falsely predicated as non-coding)

FP = false positive count (the actual non-coding that are falsely predicated as coding)

For accuracy at the full exon level,

$$Sn = \frac{\text{number of correct exons}}{\text{number of actual exons}}$$
$$Sp = \frac{\text{number of correct exons}}{\text{number of predicted exons}}$$
$$SnSp\_avg = \frac{Sn + Sp}{2}$$

Note that a correct exon means that both ends of the predicated exon match exactly with the corresponding actual exon. DNA sequences usually contains a large amount of junks, the above formulas avoid using the information from junks, which may overwhelm the contributions of other factors. Following [6] and [50], we use the mothod of measure of expections. That is, we count the occurrences of true positive, false negative and so on over all sequences and then use the above formulas to calculate the expections.

The data we use in the experiment are Chromosoems I-V of C.elegans that were obtained from release WS200 of Wormbbase [55]. The following data preparation work were processed

in [6] and [50]. “

- The data was scanning to detecting in-frame stops, and no in-frame stops were detected. This is also comfirmed by Wormbase curator Paul Davis [11]

- The data was scanned for alternative splicing, and 2974 (14.5%) out of a total of 20515 sequences represent alternative splicing including some forward encoded alternative splicings interfering with other, reverse encoded alternative splicings.

- In order to avoid the complexities involved in the prediction of alternative splicings, the transitive closure with respect to overlap of all alternative splicings was deleted from the data and the remaining annotation was appropriately offset in compensation for the deletions.

- In order to avoid both the complexity of segmented prediction as well as any bias toward any specific subset of chromosomes during cross-validation, the following were performed: a. Both data and annotation files for all 5 chromosomes were divided into a total of 70 autonomous chunks of nominal size 1Mb and minimum size 500kb. b. The resulting 70 chunks were then evenly distributed into five (5) groups for 5-fold cross-validation.

”

The details and properties of the data are show in the tables 5.1 and 5.2.

For verification purpose, we do the experiments that make the training and testing using the same data set, the results are very good–99%-100% accuracy rate. Then we seperate the training and testing data sets. [6] and [50] divided the whole data into five groups and combined groups 1-4 for training to form the probability estimates used to test group 5, then trained on 2-5 and tested on 1, and so on. In this way, a five-fold cross-validation was performed. In the results as shown in Figure 5.2 and Figure 5.3, the performance is

Summary of data reduction in *C. elegans*, Chromosomes I-V						
File	# sequences	# alt.	% alt.	# exons	# alt.	% alt.
CHROMOSOME_I	3537	619	17.50%	24295	5251	21.61%
CHROMOSOME_II	4161	629	15.12%	25427	5106	20.08%
CHROMOSOME_III	3277	590	18.00%	21541	4400	20.43%
CHROMOSOME_IV	3886	560	14.41%	24390	4478	18.36%
CHROMOSOME_V	5654	576	10.19%	32135	4201	13.07%
Total	20515	2974	14.50%	127788	23436	18.34%

Table 5.1: Summary of data reduction in C. elegans, Chromosomes I-V [6] [50]

# Bases	Coding Density	Sequences			Introns			Exons		
		Total	BP	Avg. Len.	Total	BP	Avg. Len.	Total	BP	Avg. Len.
69024243	0.20	14813	28391591	1916.67	67093	14742264	219.73	81906	13647511	166.62

Table 5.2: Properties of data set C. elegans, Chromosomes I-V (reduced) [6] [50]

very stable among each fold. So here we skip the cross-fold validation, and use four groups of the data for the training and the rest group for the testing and justify the matching result corresponding to the fold in [6] and [50]. In the experiments, we take advantage of the analysis in [6] and [50] to take the tuned parameters such as the Markov high order, window size and so on as our starting effort of the experiments. The following Figures 5.4 and 5.5 show the results of the experiments where we tune the Markov order and window size parameters to try to reach a local maximum in the predication performance for both the full exon level and the individual nucleotide level. In another set of experiments, we fix the the order of emssions in the hash areas to 5th order and now they are called "zone dependent emissions (zde)". We compare the results of three kinds of different settings. The first setting is the normal HMMBD setting, that is, we have HMM with binned duration with normal emissions, position-dependent emissions (pde) and zone dependent emissions (HMMBD+pde+zde). In the second setting, we turn off the zone dependent emissions (HMMBD+pde), the result's performance has about 1.5% drop as shown in Figures 5.6

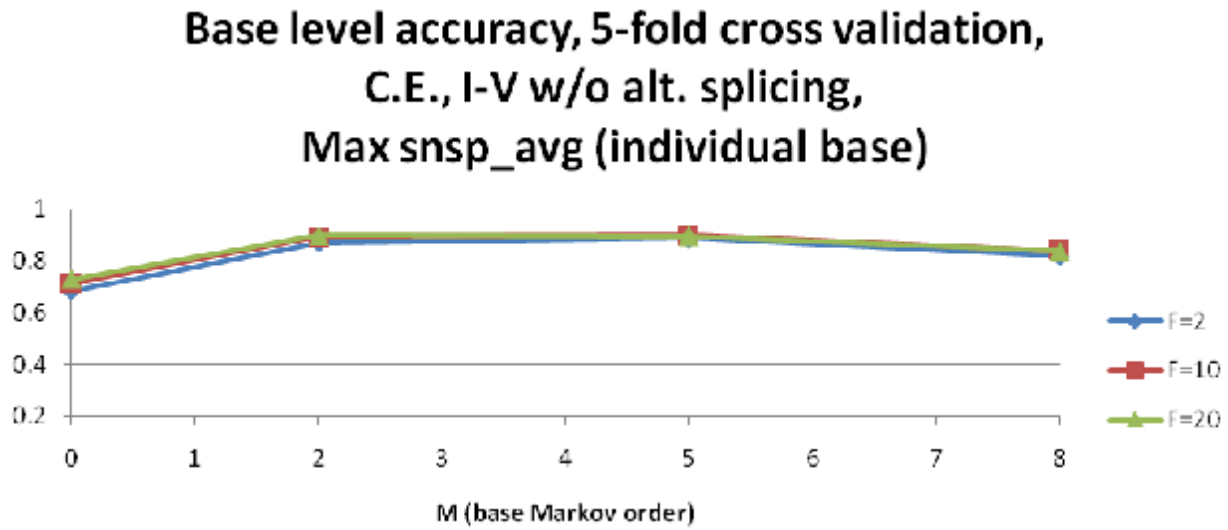


Figure 5.2: (Nucleotide) Base level accuracy for C. elegans 5-fold cross-validation [6] [50].

and 5.7. In the third setting, we use the same setting as the first setting except that we now use the geometric distribution that is implicitly incorporated by HMM as the duration distribution input to the HMMBD (HMMBD+pde+zde+Geometric). The purpose is have an approximation of the performance of the standard HMM. As show in Figures 5.6 and 5.7, the performance of the result has about 3% to 4% drop. When the window size becomes 0, that is, when we turn off the setting of position-dependent emissions, the performances of the results drop sharply as shown in Figures 5.6 and 5.7. This is because those strong informations at the transition points such as the start codon with ATG, stop codons with TAA, TAG or TGA and so on are now "polluted" by other general occurence informations.

5.4 Future development

The group's Gap interpolating Markov Model (GIMM) [49] idea may be incorporated into this model. For the n-th Markov order, we generally assume the previous n observations have the biggest influence on current observation. GIMM uses Mutual information to determine the n positions before the current position that have the most dependent information with

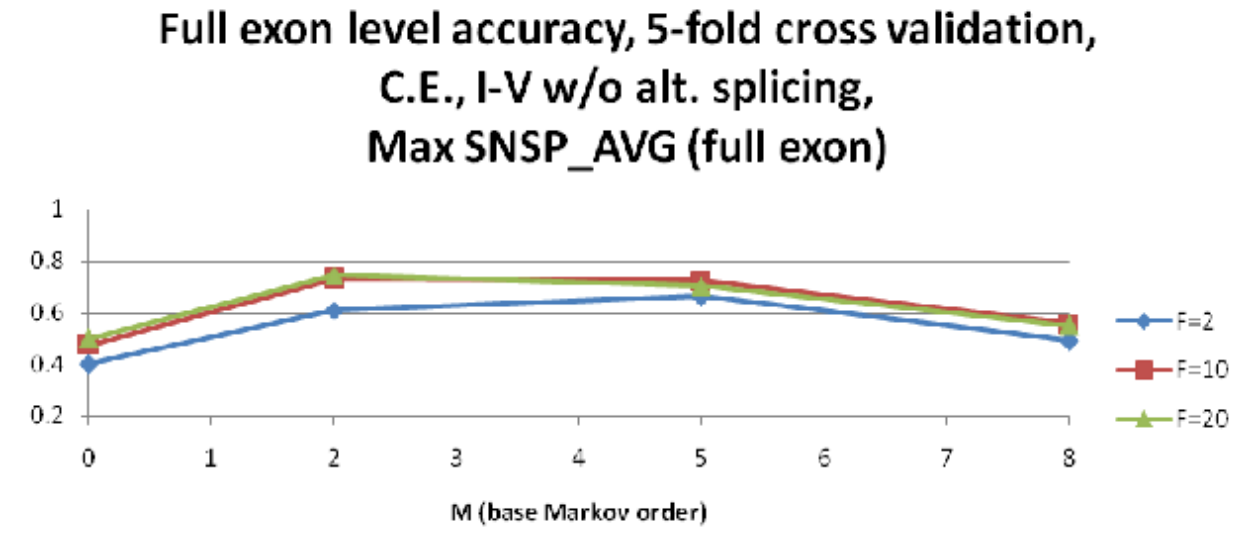


Figure 5.3: Full exon level accuracy for C. elegans 5-fold cross-validation [6] [50].

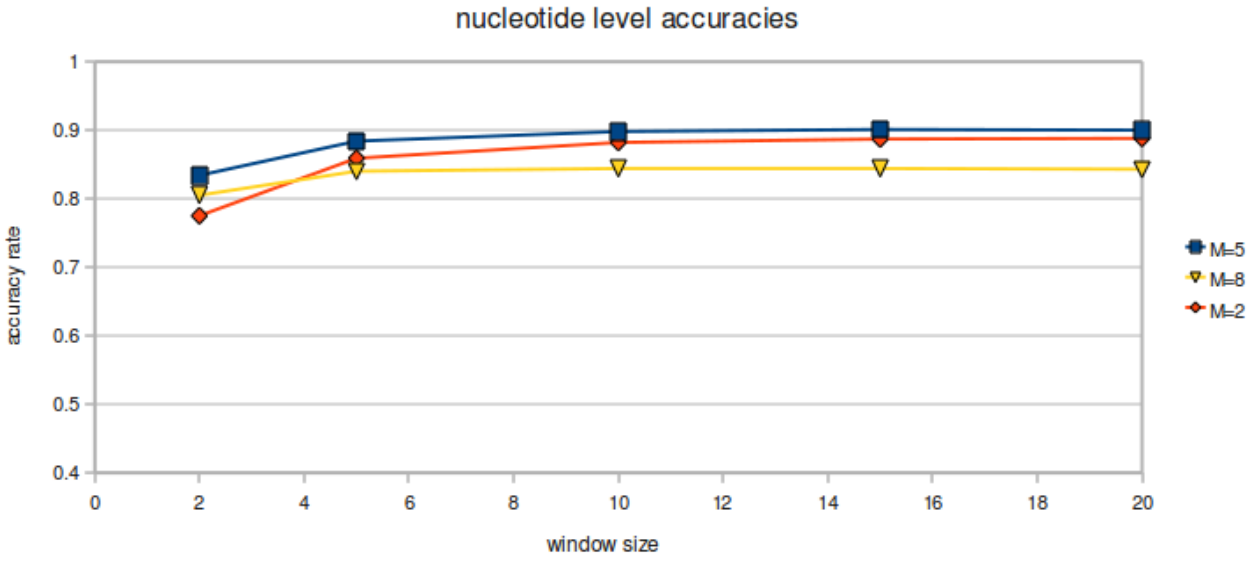


Figure 5.4: Nucleotide level accuracy rate results with Markov order of 2, 5, 8 respectively for C. elegans, Chromosomes I-V.

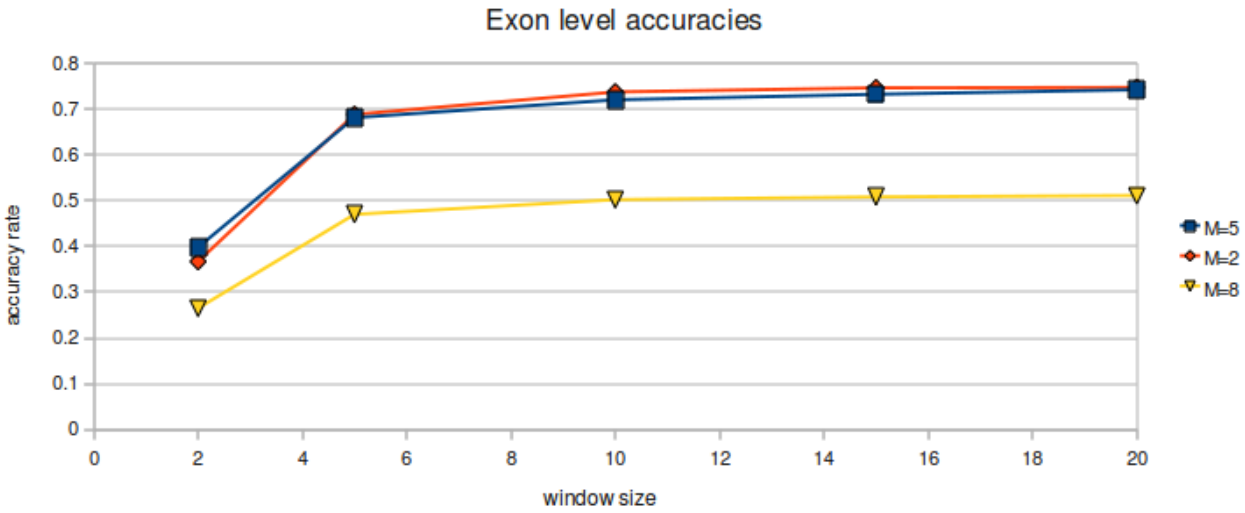


Figure 5.5: Exon level accuracy rate results with Markov order of 2, 5, 8 respectively for C. elegans, Chromosomes I-V.

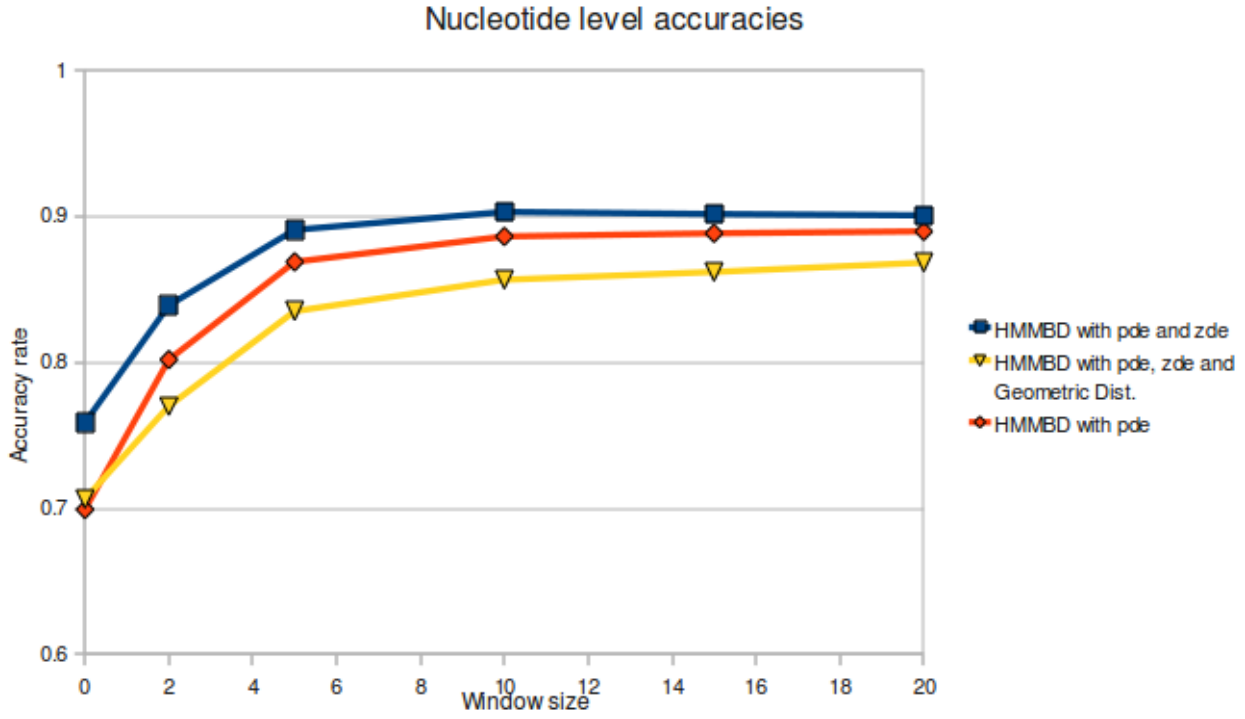


Figure 5.6: Nucleotide level accuracy rate results for three different kinds of settings.

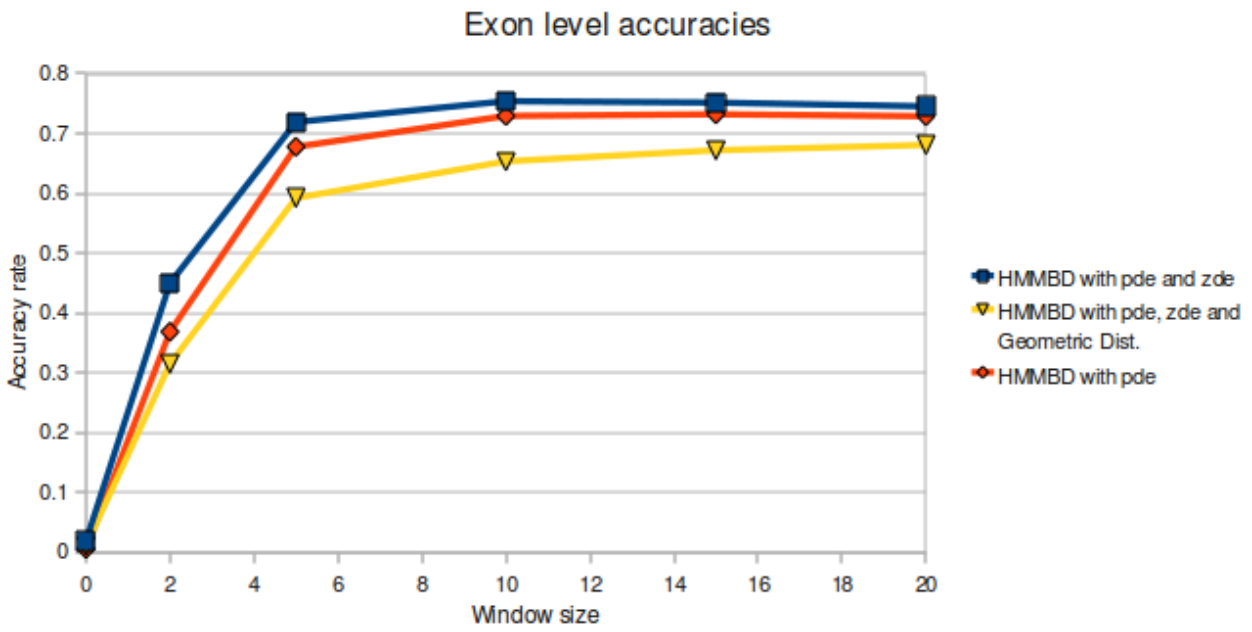


Figure 5.7: Exon level accuracy rate results for three different kinds of settings.

current position. This may be a good tool for searching the regulatory motifs.

The group developed high order HMM with Meta States [6] [50], whose HMM factorization is deviated from the traditional HMM. Current priliminary effort to incorporate the formula in the test doesn't show the performance improvment. It may be beneficial to develop an equivalence of the footprint meta states in the new model.

Chapter 6

Conclusions

Hidden Markov model with duration has been considered computationally expensive though it is usually more powerful in many applications compared to those without an explicit duration hmm. In this paper we present the hidden Markov model with binned duration algorithm (HMMBD), whose result showes no loss of accuracy compared to the HMMD decoding performance, but with computational expense that only differs from the much simpler and faster HMM decoding by a constant factor. What results (shown in the Chapter 4: Experiment Result), is an approximately 100-fold speedup over the fastest known HMMD approaches to Viterbi algorithm [39] and Baum-Welch algorithm [56]. Our approach also avoids the need to impose a maximum same-state duration, so improves upon the implementations described in [39] [56] in that manner as well. The HMMBD algorithm described here is, thus, an HMM-based signal processing improvement that makes HMMD processing only marginally more computationally expensive than standard HMM processing in a variety of situations where still remaining the HMMD decoding accuracy. The situations where HMMD can be of great benefit involve heavy-tailed duration distributions (since the standard HMM imposes a 'light-tailed' distribution – a geometric distribution). A great variety of heavy-tailed distributions can be found in bioinformatics (e.g., gene structure identifica-

tion), biology, channel current cheminformatics (e.g., nanopore detection), network packet modeling, and numerous other areas of science or engineering. In Chapter 5, we applied the new efficient algorithm to the problem of gene identification with additional position-dependent and hash-interpolated emission set up and showed the experiment results with tuned parameters.

Appendix

```
/* hidden Markov model with binned duration algorithm implementation.


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <float.h>
#include <time.h>
#define SCALOR    100000000
#define TOOSMALL  1.0         // underflow
#define MINIMUM   0.000000001



/* HMMBD algorithm:
 * seq[size]:    input sequence
 * nstate:       number of distinct states
 * nvalue:       number of distinct observations
 * MAX_D[i]:     max duration length of state i
 * nb[i]:        number of bins owned by state i
```

```
 * len[i][n]:    number of substates inside "bin n of state i"
 * pi[i]:        initial state prob. of state i
 * a[i][j]:      prob. that state i transits to state j
 * e[i][k]:      prob. that state i emits observation k
 * e2[i][k]:     e2[i][k] = e[i][k] * SCALOR
 * p[i][n]:      duration prob. for "bin n of state i"
 * scale[t]:     record whether or not scaling is applied at time t
 * hat_alpha[t][i]:      prob. that state i ends at time t
 * check_alpha[t][i]: prob. that state i starts at time t
 * queue[i][d]:  prob. of "substate i with duration=d"
 *               when it entered its current bin
 * E[i][n]:      emissions memoried for "bin n of state i"
 * bin[i][n]:    "bin n of state i"
 * pos[i][n]:    pointer to the last substate of "bin n of state i"
 * log_pi[i]:    initial state prob. of state i         (in log)
 * log_a[i][j]:  prob. that state i transits to state j (in log)
 * log_e[i][k]:  prob. that state i emits observation k (in log)
 * log_p[i][n]:  duration prob. for "bin n of state i"  (in log)
 * path[t]:      viterbi path
 * E[i]:         emissions memoried for state i
 * pointer[i][n]:pointer to the "max" substate of "bin n of state i"
 */
void forward(int *seq, int size, int nstate, int nvalue, int *MAX_D,
    double *pi, double **a, double **e, double **e2, double **p,
    int *scale, double **hat_alpha, double **check_alpha, int *nb, int **len) {
```

```
// allocate memory
int i, j, k, d, n, t;
double **emission = NULL;

double **queue = (double **) malloc(nstate * sizeof(double *));
for (i = 0; i < nstate; i++)
    queue[i] = (double *) malloc((MAX_D[i]+1) * sizeof(double));

double **bin = (double **) malloc(nstate * sizeof(double *));
for (i = 0; i < nstate; i++)
    bin[i] = (double *) malloc(nb[i] * sizeof(double));

double **E = (double **) malloc(nstate * sizeof(double *));
for (i = 0; i < nstate; i++)
    E[i] = (double *) malloc(nb[i] * sizeof(double));

int **pos = (int **) malloc(nstate * sizeof(int *));
for (i = 0; i < nstate; i++)
    pos[i] = (int *) malloc(nb[i] * sizeof(int));

// initialization
for (i = 0; i < nstate; i++) {
    queue[i][0] = pi[i];
    for (d = 1; d <= MAX_D[i]; d++)
        queue[i][d] = 0.0;
}
```

```
for (i = 0; i < nstate; i++) {
    bin[i][0] = pi[i] * e2[i][seq[0]];
    for (n = 1; n < nb[i]; n++)
        bin[i][n] = 0.0;
}

for (i = 0; i < nstate; i++)
    for (n = 0; n < nb[i]; n++)
        E[i][n] = e2[i][seq[0]];

for (i = 0; i < nstate; i++) {
    d = -1;
    for (n = 0; n < nb[i]; n++) {
        d += len[i][n];
        pos[i][n] = d;
    }
}

for (t=0; t < size; t++)
    scale[t] = 0;
scale[0] =1;

for (i = 0; i < nstate; i++)
    hat_alpha[0][i] = bin[i][0] * p[i][0];
for (t=1; t < size; t++)
```

```
    for (i = 0; i < nstate; i++)
        hat_alpha[t][i] = 0.0;

for (i = 0; i < nstate; i++)
    check_alpha[0][i] = pi[i];
for (t=1; t < size; t++)
    for (i = 0; i < nstate; i++)
        check_alpha[t][i] = 0.0;

// main body
emission = e;
for (t = 1; t < size; t++) {
    k=seq[t];
    /* calculate f_bins */
    for (i = 0; i < nstate; i++) {
        for(j = 0; j < nstate; j++)
            check_alpha[t][i] += hat_alpha[t-1][j] * a[j][i];
        d = pos[i][nb[i]-1]+1;
        if(d > MAX_D[i]) d = 0;
        queue[i][d] = check_alpha[t][i];
        for (n = 0; n < nb[i]; n++) {
            bin[i][n] += queue[i][d];
            d = pos[i][n];
            queue[i][d] *= E[i][n];
            bin[i][n] -= queue[i][d];
            bin[i][n] *= emission[i][k];
```

```
            hat_alpha[t][i] += bin[i][n]* p[i][n];
        }
        for (n = 0; n < nb[i]; n++) {
            pos[i][n]--;
            if (pos[i][n] < 0) pos[i][n] = MAX_D[i];
        }
    }
    /* update E[i][n] */
    for (i = 0; i < nstate; i++)
        for (n = 0; n < nb[i]; n++) {
            E[i][n] *= emission[i][k];
            if ((j = t-len[i][n]) >= 0) {
                int kk = seq[j];
                E[i][n] /= ((scale[j] == 0) ? e[i][kk] : e2[i][kk]);
            }
        }
    /* scaling  test */
    if (emission == e2) scale[t] = 1;
    for (i = 0; i < nstate; i++)
        if (hat_alpha[t][i] > TOOSMALL) break;
    emission = (i == nstate) ? e2 : e;
}

// Free memory
for(i = 0; i < nstate; i++)
    free(pos[i]);
```

```
    free(pos);
    for(i = 0; i < nstate; i++)
        free(E[i]);
    free(E);
    for(i = 0; i < nstate; i++)
        free(bin[i]);
    free(bin);
    for(i = 0; i < nstate; i++)
        free(queue[i]);
    free(queue);
}

void backward_baumwelch(int *seq, int size, int nstate, int nvalue,
    int* MAX_D, double *pi, double **a, double **e, double **p,
    int* nb, int** len) {

    // allocate memory
    int i, j, k, d, n, t;
    double **emission = NULL;

    double **queue = (double **) malloc(nstate * sizeof(double *));
    for (i = 0; i < nstate; i++)
        queue[i] = (double *) malloc((MAX_D[i]+1) * sizeof(double));

    double **bin = (double **) malloc(nstate * sizeof(double *));
    for (i = 0; i < nstate; i++)
```

```
    bin[i] = (double *) malloc(nb[i] * sizeof(double));

double **E = (double **) malloc(nstate * sizeof(double *));
for (i = 0; i < nstate; i++)
    E[i] = (double *) malloc(nb[i] * sizeof(double));

int **pos = (int **) malloc(nstate * sizeof(int *));
for (i = 0; i < nstate; i++)
    pos[i] = (int *) malloc(nb[i] * sizeof(int));

double *hat_beta = (double *) malloc(nstate * sizeof(double));
double *v = (double *) malloc(nstate * sizeof(double));

double **u = (double **) malloc(nstate * sizeof(double *));
for (i = 0; i < nstate; i++)
    u[i] = (double *) malloc(nstate * sizeof(double));

double **new_a = (double **) malloc(nstate * sizeof(double *));
for (i = 0; i < nstate; i++)
    new_a[i] = (double *) malloc(nstate * sizeof(double));

double **new_p = (double **) malloc(nstate * sizeof(double *));
for (i = 0; i < nstate; i++)
    new_p[i] = (double *) malloc(nb[i] * sizeof(double));

double **new_e = (double **) malloc(nstate * sizeof(double *));
```

```
    for (i = 0; i < nstate; i++)
        new_e[i] = (double *) malloc(nvalue * sizeof(double));

    double **e2 = (double **) malloc(nstate * sizeof(double *));
    for (i = 0; i < nstate; i++)
e2[i] = (double *) malloc(nvalue * sizeof(double));

    int *scale = (int *) malloc(size * sizeof(int));

    double **hat_alpha = (double **) malloc(size * sizeof(double *));
    for (i = 0; i < size; i++)
        hat_alpha[i] = (double *) malloc(nstate * sizeof(double));

    double **check_alpha = (double **) malloc(size * sizeof(double *));
    for (i = 0; i < size; i++)
        check_alpha[i] = (double *) malloc(nstate * sizeof(double));



    // initialization
    for (i = 0; i < nstate; i++)
        for (j = 0; j < nstate; j++)
            new_a[i][j] = 0.0;
    for (i = 0; i < nstate; i++)
        for (n = 0; n < nb[i]; n++)
            new_p[i][n] = 0.0;
    for (i = 0; i < nstate; i++)
```

```
    for (k = 0; k < nvalue; k++)
        new_e[i][k] = 0.0;

for (i = 0; i < nstate; i++)
    for (k = 0; k < nvalue; k++)
        e2[i][k] = e[i][k] * SCALOR;

forward(seq, size, nstate, nvalue, MAX_D, pi, a,
        e, e2, p, scale, hat_alpha, check_alpha, nb, len);

for (i = 0; i < nstate; i++) {
    queue[i][0] = 1.0;
    for (d = 1; d <= MAX_D[i]; d++)
        queue[i][d] = 0.0;
}

emission = scale[size-1] ? e2 : e;
for (i = 0; i < nstate; i++) {
    bin[i][0] = emission[i][seq[size-1]];
    for (n = 1; n < nb[i]; n++)
        bin[i][n] = 0.0;
}

for (i = 0; i < nstate; i++)
    for (n = 0; n < nb[i]; n++)
        E[i][n] = emission[i][seq[size-1]];
```

```
for (i = 0; i < nstate; i++) {
    d = -1;
    for (n = 0; n < nb[i]; n++) {
        d += len[i][n];
        pos[i][n] = d;
    }
}

for(i = 0; i < nstate; i++)
    hat_beta[i] = bin[i][0] * p[i][0];

for(i = 0; i < nstate; i++) {
    new_e[i][seq[size-1]] = v[i] = hat_alpha[size-1][i];
    new_p[i][0] = check_alpha[size-1][i] * bin[i][0];
}

// main body
for (t = size-2; t >= 0; t--) {
    k=seq[t];
    /* baumwelch: transition and emission */
    for (i = 0; i < nstate; i++)
        for (j = 0; j < nstate; j++) {
            u[i][j] = hat_alpha[t][i] * a[i][j] * hat_beta[j];
            new_a[i][j] += u[i][j];
        }
```

```
for (i = 0; i < nstate; i++) {
    for (j = 0; j < nstate; j++)
        v[i] += (u[i][j] - u[j][i]);
    new_e[i][k] += v[i];
}
/* calculate b_bins */
emission = scale[t] ? e2 : e;
for (i = 0; i < nstate; i++) {
    double check_beta = 0.0;
    for(j = 0; j < nstate; j++)
        check_beta += hat_beta[j] * a[i][j];
    d = pos[i][nb[i]-1]+1;
    if(d > MAX_D[i]) d = 0;
    queue[i][d] = check_beta;
    for (n = 0; n < nb[i]; n++) {
        bin[i][n] += queue[i][d];
        d = pos[i][n];
        queue[i][d] *= E[i][n];
        bin[i][n] -= queue[i][d];
        bin[i][n] *= emission[i][k];
    }
    for (n = 0; n < nb[i]; n++) {
        pos[i][n]--;
        if (pos[i][n] < 0) pos[i][n] = MAX_D[i];
    }
}
```

```
    /* update E[i][n] and hat_beta[i] */
    for (i = 0; i < nstate; i++) {
        hat_beta[i] = 0.0;
        for (n = 0; n < nb[i]; n++) {
            E[i][n] *= emission[i][k];
            if ((j = t + len[i][n]) < size) {
                int kk = seq[j];
                E[i][n] /= ((scale[j] == 0) ? e[i][kk] : e2[i][kk]);
            }
            hat_beta[i] += bin[i][n] * p[i][n];
        }
    }
    /* baumWelch: duration */
    for (i = 0; i < nstate; i++)
        for (n = 0; n < nb[i]; n++)
            new_p[i][n] += check_alpha[t][i] * bin[i][n];

}



 double num_scale = 0.0;
 for (i = 0; i < size; i++)
     num_scale += scale[i];
 double aa=0.0;
 for (i = 0; i < nstate; i++)
     aa += hat_alpha[size-1][i];
```

```
 double bb=0.0;
 for (i = 0; i < nstate; i++)
     bb += pi[i]* hat_beta[i];
 double log_prob = log(aa) - num_scale * log(SCALOR);
 printf("aa = %f, bb = %f, log_prob=%f\n", aa, bb, log_prob);


// update initial-state-prob., transition, duration and emission.
double sum = 0.0;
for (i = 0; i < nstate; i++)
    sum += v[i];
for (i = 0; i < nstate; i++) {
    pi[i] = v[i] / sum;
    if (pi[i] < MINIMUM) pi[i] = MINIMUM;
}

for (i = 0; i < nstate; i++) {
    sum = 0.0;
    for (j = 0; j < nstate; j++)
        sum += new_a[i][j];
    for (j = 0; j < nstate; j++) {
        a[i][j] = new_a[i][j] / sum;
        if (a[i][j] < MINIMUM) a[i][j] = MINIMUM;
    }
}
for (i = 0; i < nstate; i++)
```

```
    a[i][i] = 0.0;

for (i = 0; i < nstate; i++) {
    sum = 0.0;
    for (n = 0; n < nb[i]; n++) {
        new_p[i][n] *= p[i][n];
        sum += new_p[i][n];
    }
    for (n = 0; n < nb[i]; n++) {
        p[i][n] = new_p[i][n] / sum;
        if (p[i][n] < MINIMUM) p[i][n] = MINIMUM;
    }
}

for (i = 0; i < nstate; i++) {
    sum = 0.0;
    for (k = 0; k < nvalue; k++)
        sum += new_e[i][k];
    for (k = 0; k < nvalue; k++) {
        e[i][k] = new_e[i][k] / sum;
        if (e[i][k] < MINIMUM) e[i][k] = MINIMUM;
    }
}

// Free memory
for (i = 0; i < size; i++) free(check_alpha[i]);
```

```
free(check_alpha);
for (i = 0; i < size; i++) free(hat_alpha[i]);
free(hat_alpha);
free(scale);
for(i = 0; i < nstate; i++) free(e2[i]);
free(e2);
for(i = 0; i < nstate; i++)
    free(new_e[i]);
free(new_e);
for(i = 0; i < nstate; i++)
    free(new_p[i]);
free(new_p);
for(i = 0; i < nstate; i++)
    free(new_a[i]);
free(new_a);
for(i = 0; i < nstate; i++)
    free(u[i]);
free(u);
free(v);
free(hat_beta);
for(i = 0; i < nstate; i++)
    free(pos[i]);
free(pos);
for(i = 0; i < nstate; i++)
    free(E[i]);
free(E);
```

```
    for(i = 0; i < nstate; i++)
        free(bin[i]);
    free(bin);
    for(i = 0; i < nstate; i++)
        free(queue[i]);
    free(queue);
}

void viterbi(int *seq, int size, int nstate, double *log_pi,
   double **log_a, double **log_e, double **log_p, int *path,
   int* nb, int** len) {

    // allocation  and initialization
    int i, j, p, n, t;

    double *E = (double *) malloc(nstate * sizeof(double));
    for (i = 0; i < nstate; i++)
        E[i] = log_e[i][seq[0]];

    int *qhead  = (int *) malloc(nstate * sizeof(int));
    for (i = 0; i < nstate; i++)
        qhead[i] = 0;

    int *qbin  = (int *) malloc(nstate * sizeof(int));
    int *qsize = (int *) malloc(nstate * sizeof(int));
    for (i = 0; i < nstate; i++) {
```

```
     qsize[i] = 0;
     for (n = 0; n < nb[i]; n++)
        if (len[i][n] > qsize[i]) {
            qbin[i] = n;
            qsize[i] = len[i][n];
        }
     ++qsize[i];
}

double **queue = (double **) malloc(nstate * sizeof(double *));
for (i = 0; i < nstate; i++)
    queue[i] = (double *) malloc(qsize[i] * sizeof(double));

for (i = 0; i < nstate; i++) {
    queue[i][0] = log_pi[i];
    for (p = 1; p < qsize[i]; p++)
        queue[i][p] = 0.0;
}

int **qtime = (int **) malloc(nstate * sizeof(int *));
for (i = 0; i < nstate; i++)
    qtime[i] = (int *) malloc(qsize[i] * sizeof(int));

for (i = 0; i < nstate; i++)
    for (p = 0; p < qsize[i]; p++)
        qtime[i][p] = 0;
```

```
int **pointer = (int **) malloc(nstate * sizeof(int *));
for (i = 0; i < nstate; i++)
    pointer[i] = (int *) malloc(nb[i] * sizeof(int));

for (i = 0; i < nstate; i++)
    for (n = 0; n < nb[i]; n++)
        pointer[i][n] = 0;

int **shift = (int **) malloc(nstate * sizeof(int *));
for (i = 0; i < nstate; i++)
    shift[i] = (int *) malloc(nb[i] * sizeof(int));

for (i = 0; i < nstate; i++) {
    shift[i][0] = 2;
    for (n = 1; n < nb[i]; n++)
        shift[i][n] = shift[i][n-1] + len[i][n-1];
}

double **state = (double **) malloc((size+2) * sizeof(double *));
for (t = 0; t < size+2; t++)
    state[t] = (double *) malloc(nstate * sizeof(double));

for (t = 0; t < size+2; t++)
    for (i = 0; i < nstate; i++)
        state[t][i] = -DBL_MAX;
```

```
int **start = (int **) malloc((size+2) * sizeof(int *));
for (t = 0; t < size+2; t++)
    start[t] = (int *) malloc(nstate * sizeof(int));

for (t = 0; t < size+2; t++)
    for (i = 0; i < nstate; i++)
        start[t][i] = 0.0;



// initialization
for (i = 0; i < nstate; i++)
    for (n = 0; n < nb[i]; n++) {
        int index = 0 + shift[i][n];
        if (index < size+2) {
            double tmp = queue[i][pointer[i][n]] + log_p[i][n];
            if (tmp > state[index][i]) {
                state[index][i] = tmp;
                start[index][i]  = qtime[i][pointer[i][n]];
            }
        }
    }

// main body
for (t = 1; t < size; t++) {
    for (i = 0; i < nstate; i++) {
```

```
// generate a new substate with duration=1
double max = -DBL_MAX;
int s =0;
for (j = 0; j < nstate; j++) {
    double tmp = state[t+1][j] + E[j] + log_a[j][i];
    if (tmp > max) {
        max = tmp;
        s = j;
    }
}
state[t][i]  = s;
start[t][i] = start[t+1][s];
max -= E[i];


// insert this new substate
int p = qhead[i];
while (p != pointer[i][qbin[i]] && queue[i][p] < max)
    if (--p < 0) p = qsize[i] - 1;
if (++p == qsize[i]) p = 0;
queue[i][p] = max;
qtime[i][p] = t;
qhead[i] = p;
// adjust pointers if necessary
for (n = 0; n < nb[i]; n++) {
    if (queue[i][pointer[i][n]] < max) {
```

```
                pointer[i][n] = p;
            }
            if (t - qtime[i][pointer[i][n]] >= len[i][n]) {
                pointer[i][n]++;
                if (pointer[i][n] == qsize[i]) pointer[i][n] = 0;
            }
        }
        // update "the most probable" pathMAX_D
        for (n = 0; n < nb[i]; n++) {
            int index = t + shift[i][n];
            if (index < size+2) {
                double tmp = queue[i][pointer[i][n]] + log_p[i][n];
                if (tmp > state[index][i]) {
                    state[index][i] = tmp;
                    start[index][i]  = qtime[i][pointer[i][n]];
                }
            }
        }
        // update "the most probable" pathMAX_D
    }
    // update E[i]
    for (i = 0; i < nstate; i++)
        E[i] += log_e[i][seq[t]];

}
```

```
// backtrack
double log_prob = -DBL_MAX;
int s = 0;
double tmp;
for (i = 0; i < nstate; i++)
    if ((tmp = state[size+1][i] + E[i] ) > log_prob) {
        log_prob = tmp;
        s = i;
    }
int start_time = start[size+1][s];

t = size;
do {
    if (start_time < 0) printf("Error on backtracking!\n");
    for (i = start_time; i < t; i++)
        path[i] = s;
    t = start_time;
    start_time = (int)start[t][s];
    s = (int)state[t][s];
} while (t > 0);

// Free memory
for (t = 0; t <= size; t++)
    free(start[t]);
free(start);
for (t = 0; t <= size; t++)
```

```
        free(state[t]);
    free(state);
    for (i = 0; i < nstate; i++)
        free(shift[i]);
    free(shift);
    for(i = 0; i < nstate; i++)
        free(pointer[i]);
    free(pointer);
    for(i = 0; i < nstate; i++)
        free(qtime[i]);
    free(qtime);
    for(i = 0; i < nstate; i++)
        free(queue[i]);
    free(queue);
    free(qsize);
    free(qbin);
    free(qhead);
    free(E);
}
```

---------- END ----------

Bibliography

[1] Churchill G. A. Hidden markov chains and the analysis of genome structure. *Computers and Chemistry*, 16:107–115, 1992.

[2] X. Ros A. Bonafonte and J. B. Marino. An efficient algorithm to find the best state sequence in hsmm. *Proceedings of Eurospeech*, pages 1547–1550, 1993.

[3] S. Mian A. Krogh and D. Haussler. A hidden markov model that finds genes in e. coli dna. *Nucleic Acids Research*, 22:68–78, 1994.

[4] S. Theodoridis A. Pikrakis and D. Kamarotos. Classification of musical patterns using variable duration hidden markov models. *IEEE Transactions on Speech and Audio Processing*, 46:5, 2006.

[5] S. E. Levinson B. H. Juang, L. R. Rabiner and M. M. Sondhi. Recent developments in the application of hidden markov models to speakerindependent isolated word recognition. *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 9–12, 1985.

[6] Carl Edward Baribault. *Meta State Generalized HMM for Eukaryotic Gene Structure Identification*. PhD thesis, University of New Orleans, 2009.

[7] Moises Burset and Roderic Guigo. Evaluation of gene structure prediction programs. *Genomics*, 34:353–367, 1996.

[8] Mou-Yen Chen and Amlan Kundu. Variable duration hidden markov model and morphological segmentation for handwritten word recognition. *IEEE Transactions on Image Processing*, 4:12, 1995.

[9] A. Churbanov and S. Winters-Hilt. Implementing em and viterbi algorithms for hidden markov model in linear memory. *BMC Bioinformatics*, 9:228, 2009.

[10] M.S. Wu C.L. Huang and S.H. Jeng. Gesture recognition using the multi-pdm method and hidden markov model. *Image and Vision Computing*, 18:865, 2000.

[11] Paul Davis. http://ws200.wormbase.org/, 2009. pad@sanger.ac.uk.

[12] Abraham de Moivre. Approximatio ad summam terminarum binomii (a + b), 1773.

[13] Krogh A. Durbin R., Eddy S. and Mitchison. *Biological Sequence Analysis*. Cambridge University Press, Cambridge, MA, 1998.

[14] J.P. Hughes E. Bellone and P. Guttorp. A hidden markov model for downscaling synoptic atmospheric patterns to precipitation amounts. *Climate Research*, 2000.

[15] W. A. Stahel E. Limpert and M. Abbt. Log-normal distributions across the sciences: Keys and clues. *BioScience*, 51:341–352, 2001.

[16] J.D. Ferguson. Variable duration models for speech. *Proceedings of Symposium on the Application of Hidden Markov models to Text and Speech*, pages 143–179, 1980.

[17] J. Garcia-Frias and P. M. Crespo. Hidden markov models for burst error characterization in indoor radio channels. *IEEE Transactions on Vehicular Technology*, 1997.

[18] X. Ge and P. Smyth. Hidden markov models for endpoint detection in plasma etch processes. Technical report, Departement of Information and Computer Science, University of California, Irvine, 2001.

[19] Z. Ghahramani and M. Jordan. Factorial hidden markov models. *Machine Learning*, 29:245–273, 1997.

[20] Sren Hauberg and Jakob Sloth. An efficient algorithm for modelling duration in hidden markov models, with a dramatic application. *Journal of Mathematical Imaging and Vision*, 10, 2008.

[21] Scofield DG Hong X and Lynch M. Intron size, abundance, and distribution within untranslated regions of genes. *Molecular Biology and Evolution*, 23:2392–404, 2006.

[22] A. Najmi J. Li and R.M. Gray. Image classification by a two-dimensional hidden markov model. *IEEE Transactions on Signal Processing*, 48, 2000.

[23] R. A. Olshen J. Li, R. M. Gray. Multiresolution image classification by hierarchical modeling with two-dimensional hidden markov models. *IEEE Transactions on Information Theory*, 2000.

[24] M.T. Johnson. Capacity and complexity of hmm duration modeling techniques. *IEEE Signal Processing Letters*, 12:407–410, 2005.

[25] M. Hasegawa-Johnson K. Chen and S. Borys. Prosody dependent speech recognition with explicit duration modeling at intonational phrase boundaries. *Proceedings of Eurospeech*, 2003.

[26] Joseph A. Kogan and Daniel Margoliash. Automated recognition of bird song elements from continuous recordings using dynamic time warping and hidden markov models: A comparative study. *Journal of the Acoustical Society of America*, 1998.

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

[28] SE. Levinson. Continuously variable duration hidden markov models for automatic speech recognition. *Comput. Speech Language*, 1:29–45, 1986.

[29] Chernoff Y. Lomsadze A., Ter-Hovhannisyan V. and Borodovsky M. Gene identification in novel eukaryotic genomes by self-training algorithm. *Nucleic Acids Research*, 33:6494–6506, 2005.

[30] Lorenz and M. O. Methods of measuring the concentration of wealth. *Publications of the American Statistical Association*, 1905.

[31] J. Appenrodt M. Elmezain, A. Al-Hamadi and B. Michaelis. A hidden markov model-based isolated and meaningful hand gesture recognition. *International Journal of Electrical, Computer, and Systems Engineering*, 3:156–163, 2009.

[32] Jorg Appenrodt Mahmoud Elmezain, Ayoub Al-Hamadi and Bernd Michaelis. A hidden markov model-based continuous gesture recognition system for hand motion trajectory. *IEEE Conference Proceeding*, 2008.

[33] Andrey Andreyevich Markov. *Theory of Algorithms*. Academy of Sciences of the USSR, 1954.

[34] Hari Tammana Martin G. Reese, David Kulp and David Haussler. Geniegene finding in drosophila melanogaster. *Gnome Research*, 10:529–538, 2000.

[35] Harper Mitcell C. and M. Jamieson. on the complexity of explicit duration hmm's. *IEEE Transactions on Speech and Audio Processing*, 3:213–217, 1995.

[36] K. Murphy and M. Paskin. Linear time inference in hierarchical hmms. *Proceedings of Neural Information Processing Systems*, 2001.

[37] Christos Ntantamis. A duration hidden markov model for the identification of regimes in stock market returns. *Journal of Economic Literature*, 2009.

[38] L.R. Rabiner. A tutorial on hidden markov models and selected application in speech recognition. *Proceedings of the IEEE*, 77:257–286, 1989.

[39] P. Ramesh and J.G. Wilpon. Modeling state durations in hidden markov models for automatics speech recognition. *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, 1:381–384, 1992.

[40] C. Raphael. Automatic segmentation of acoustic musical signals using hidden markov models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21:1998, 360.

[41] M.J. Russel and R.K. Moore. Explicit modeling of state occupancy in hidden markov models for automatic speech recognition. *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 5–8, 1985.

[42] Y. Singer S. Fine and N. Tishby. The hierarchical hidden markov model: Analysis and applications. *Machine Learning*, 32:41, 1998.

[43] E. Augustin S. Knerr and D. Price. Hidden markov model based word recognition and its application to legal amount reading on french checks. *Computer Vision and Image Understanding*, page 404, 1998.

[44] G. W. West S. Lhr, S. Venkatesh and H. H. Bui. Explicit state duration hmm for abnormality detection in sequences of human activity. *Proceedings of the 8th Pacific Rim International Conference*, 2004.

[45] V.S. DeGuzman S. Winters-Hilt, W. Vercoutere and D. Deamer. Highly accurate classification of watson-crick base-pairs on termini of single dna molecules. *Biophysical Journal*, 84:967, 2003.

[46] M. Schenkel and M. Jabri. Low resolution, degraded document recognition using neural networks and hidden markov models. *Pattern Recognition Letters*, 3:365–371, 1998.

[47] Perry A. Stoll and Jun Ohya. Applications of hmm modeling to recognizing human gestures in image sequences for a man-machine interface. *IEEE International Workshop on Robot and Human Communication*, pages 129–134, 1995.

[48] J. Vlontzos and S. Kung. Hidden markov models for character recognition. *IEEE Transactions on Image Processing*, 1992.

[49] Stephen Winters-Hilt. Hidden markov model variants and their application. *BMC Bioinformatics*, 7:22, 2006.

[50] Stephen Winters-Hilt and Carl Baribault. A meta-state hmm with application to gene-structure identification in eukaryotes. In process.

[51] Stephen Winters-Hilt and Carl Baribault. A novel, fast, hmm-with-duration implementation - for application with a new, pattern recognition informed, nanopore detector. *BMC Bioinformatics*, 8:S19, 2007.

[52] Stephen Winters-Hilt and Zuliang Jiang. Eukaryotic gene-finding using hmm-with-duration as platform for incorporation of position and zone dependent emission. In process.

[53] Stephen Winters-Hilt and Zuliang Jiang. Using hmm-with-duration as platform for incorporation of side-information. In process.

[54] Stephen Winters-Hilt and Zuliang Jiang. A hidden markov model with binned duration algorithm. *IEEE Tranctions on Signal Processing*, 58:948–952, 2010.

[55] wormbase. http://www.wormbase.org, 2009.

[56] SZ Yu and H. Kobayashi. An efficient forward-backward algorithm for an explicit-duration hidden markov model. *IEEE Signal Processing Letters*, 10:11–14, 2003.

Vita

Zuliang Jiang was born in Fuzhou, China and received his B.S. in Computer Science from Xiamen University, China. He then went on to pursue his Ph.D. study in Computer Science Department, University of New Orleans.