Spring 5-13-2015

Using Hidden Markov Models to Detect DNA Motifs

Santrupti Nerli
San Jose State University

Using Hidden Markov Models to Detect DNA Motifs

A project

Presented to

The Faculty of the Department of Computer Science

San Jose Staté University

In Partial Fulfilment

Of the Requirements for the Degree

Master of Science

by

Santrupti Nerli

May 2015

The Designated Project Committee Approves the Project Titled

Using Hidden Markov Models to Detect DNA Motifs

by

Santrupti Nerli

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

San José State University

May 2015

Dr. Sami Khuri	Department of Computer Science
Dr. Thomas Austin	Department of Computer Science
Dr. H Chris Tseng	Department of Computer Science

ABSTRACT

Using Hidden Markov Models to Detect DNA Motifs

by Santrupti Nerli

During the process of gene expression in eukaryotes, mRNA splicing is one of the key processes carried out by a complex called spliceosome. Spliceosome guarantees proper removal of introns and joining of exons before the translation process. Precise splicing is essential for the production of functional proteins. Spliceosome detects specific sequence motifs within an mRNA sequence called splice sites. Two of the splice sites are the 5' and 3' sites that border all the introns. Normal splicing process if disrupted by mutation may lead to fatal diseases. In this work, we predict splice sites in a human genome using hidden Markov models (HMMs).

Prior to hidden Markov models, we tried to predict splice sites using higher order position weight matrices. Position Weight Matrix (PWM) is a conventional computational method used to represent splice sites or any sequence motif. In a set of aligned sequences, PWM captures the distribution of nucleotides at each position. The performance of simple PWMs in classifying authentic 5′ and 3′ splice sites and predicting cryptic splice sites in human genes is resonably well [1, 2, 3]. However, they are built by making a strong independence assumption between contiguous and non- contiguous nucleotide positions. Therefore, we developed a higher order PWM method that incorporates maximal dependence decomposition algorithm (MDD) [4] to successfully identify statistically significant splice sites.

Simple PWM also fails to capture sites that lie in both splice site and non-splice site regions. Therefore, we implemented HMMs to overcome this limitation of PWM.

We performed 10-fold cross validation of all the three methods for 5′ and 3′ authentic human splice sites from the *HS3D database* [5] and observed that MDD outperforms the other two methods with area under the Receiver Operating Characteristic curve (ROC) to be 0.96 and 0.93, respectively. Similarly, we performed classification of 5′ and 3′ putative cryptic splice sites in the beta-globin (HBB) and breast cancer type 1 susceptibility protein (BRCA1) genes. We observed that MDD performs very well in classifying both BRCA1 and HBB cryptic splice sites with area under ROC of 0.99, 0.95, 0.89 and 1.0 respectively. However, we also observed that HMMs perform fairly well in classifying splice sites and cryptic splice sites compared to traditional PWM method.

ACKNOWLEDGEMENTS

I want to sincerely thank my project advisor Dr. Sami Khuri, for his continuous support and encouragement throughout this project. I would also like to extend my thanks to my committee members, Dr. Thomas Austin and Dr. H Chris Tseng for their time and support.

My special thanks to Dr. Natalia Khuri for her guidance and valuable suggestions during the course of this project.

Lastly, I would like to thank my husband Prasannakumar Patil for his support at all times.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

Proteins are the building blocks of living organisms. In order to produce functional proteins in eukaryotes, messenger ribonucleic acid (mRNA) undergoes a process called alternative splicing. Alternative splicing is a process where some introns are removed and exons are fused to ready the mRNA for translation process. This process is carried out by a complex called spliceosome. Spliceosome is intelligent enough to identify the extremes of all introns called 5' and 3' splice sites. Spliceosome can sometimes be misled due to mutations at these sites causing it to splice unintended regions. This may trigger production of dysfunctional proteins in turn leading to malignant diseases.

Prediction of splice sites is essential as they play most significant role in protein production. In this work, we build a predictor in silico that will identify splice sites in the human genome.

1. 1 Background

Splice site oligomers can be represented computationally using a popular method known as position weight matrix (PWM). Position weight matrix identifies unknown sites by scoring them with a matrix that is constructed by taking into account the probabilities of observing specific nucleotides at specific positions of aligned sequences.

More precisely, a PWM is a four by K matrix. Four rows for nucleotides {A, C, G, T} and K is the size of an oligomer. Each cell represents a distribution of that nucleotide at position p, where $1 \leq p \leq K$. PWMs are known to perform fairly well in predicting splice sites, however they are built with a strong assumption. They assume that nucleotides at various positions are independent of one another which is rarely the case with splice sites. In order to overcome independence assumption, we constructed a sophisticated model that takes into account the

interdependencies between nucleotide positions. This model constructs a higher order position weight matrix and makes use of the maximal dependence decomposition (MDD) algorithm. The accuracy of prediction of splice sites using MDD is very high and can be observed in later chapters.

In the next chapter, we will study PWMs in detail.

CHAPTER 2

Position Weight Matrix

The position weight matrix (PWM) is one of the most popular computational methods to represent DNA motifs. For splice sites, a PWM is a *4xK* matrix where rows represent the four nucleotides and columns represents the size of each motif. To construct the PWM for splice sites, we have to align multiple splice site motifs, and then compute the frequency of every nucleotide at every position.

Let *x* represent a nucleotide and *j* the position, then frequency of observing *x* at position *j* is given by

$$f_{x,j} = \frac{N_{x,j}}{N}$$

where, $N_{x,j}$ is total number of times *x* is observed at position *j* and *N* is the total number of nucleotides at position *j*.

Consider the following set of aligned sequences.

```
AGTGTAAGT
TTCGTAAGT
AGGGTAAGA
CAGGTGGGG
GAGGTGAGT
ACGGTAACT
CTCGTAAGT
TAAGTAAGC
CTGGTGGGT
CAGGTGAGG
```

Example 1: Sample splice sites oligomers from HS3D [5]

These are ten splice site motifs from Homo Sapiens Splice Site Database (HS3D). Let us construct a PWM for the sample sequences.

Frequency of observing x = {A, C, G, T} at position 1 is given by:

$$f_{A,1} = \frac{3}{10} \qquad f_{C,1} = \frac{4}{10} \qquad f_{G,1} = \frac{1}{10} \qquad f_{T,1} = \frac{2}{10}$$

Similarly, we compute the frequencies of all nucleotides at all positions [6] and construct Table 1 for the sample sequences.

Table 1: Position weight matrix for the same sequences

Position / Nucleotide	1	2	3	4	5	6	7	8	9
A	0.3	0.4	0.1	0.0	0.0	0.6	0.8	0.0	0.1
C	0.4	0.1	0.2	0.0	0.0	0.0	0.0	0.1	0.1
G	0.1	0.2	0.6	1.0	0.0	0.4	0.2	0.9	0.2
T	0.2	0.3	0.1	0.0	1.0	0.0	0.0	0.0	0.6

2.1 Pseudocounts

In the above example sequences, some of the nucleotides at a few positions are not observed at all. For example, nucleotides C and T are not observed at position 8. However, this is seldom true in reality. The example sequences represent a small set of the entire population of sequences. Due to this data insufficiency, we might not observe some nucleotides at a few positions. This problem can be eliminated by taking pseudocounts into account before computing the probabilities. By adding pseudocounts, we are considering unobserved nucleotides at that position.

We can make use of Laplace smoothing [7] and add a pseudocount of 1.

We recalculate the frequencies of nucleotides as follows:

$$f_{x,j} = \frac{N_{x,j} + 1}{N + 4}$$

Frequencies with pseudocounts for position 1 are now recalculated as:

$$f_{A,1} = \frac{3+1}{10+4} \qquad f_{C,1} = \frac{4+1}{10+4} \qquad f_{G,1} = \frac{1+1}{10+4} \qquad f_{T,1} = \frac{2+1}{10+4}$$

Updating Table 1 with pseudocounts, we get Table 2.

Table 2: PWM for the example sequences with pseudocounts

Position / Nucleotide	1	2	3	4	5	6	7	8	9
A	0.286	0.357	0.143	0.071	0.071	0.500	0.643	0.071	0.143
C	0.357	0.143	0.214	0.071	0.071	0.071	0.071	0.143	0.143
G	0.143	0.214	0.500	0.786	0.071	0.143	0.214	0.714	0.214
T	0.214	0.286	0.143	0.071	0.786	0.071	0.071	0.071	0.500

2.2 Log Likelihood Ratios

The scores obtained previously i.e. the scores from Table 2, can sometimes be misleading. This is because, PWM constructed using training sequences will score the unknown sequence higher if the composition of unknown sequence is somewhat similar to the PWM. It may score another unknown sequence with a very low score if it deviates from a norm even a little but, still belongs to family of sequences used for training.

To overcome this bias, we take the ratio of observed frequency and the frequency of that nucleotide from a large population of sequences, also known as the expected frequency or background frequency.

Therefore,

$$Normalized\ score = \frac{observed\ frequency}{expected\ frequency}$$

In our case, we can assume that the expected frequency among all the nucleotides is equiprobable. So, expected frequencies E of nucleotides are

E(A) = E(C) = E(G) = E(T) = 0.25

Hence, normalized scores, say r of nucleotides at position 1 will be:

$$r_{A,1} = \frac{0.286}{E(A)} \qquad r_{C,1} = \frac{0.357}{E(C)} \qquad r_{G,1} = \frac{0.143}{E(G)} \qquad r_{T,1} = \frac{0.214}{E(T)}$$

We can further update the scoring tables by taking logs of the ratios we computed. Taking logs will help us avoid underflows and reduce round off errors propagated by multiplication of odds scores. Therefore, the log-odds scores of each nucleotide at position 1 will now become

Log-odds score for A = $\log_2\left(\frac{0.286}{0.25}\right)$

Log-odds score for C = $\log_2\left(\frac{0.357}{0.25}\right)$

Log-odds score for G = $\log_2\left(\frac{0.143}{0.25}\right)$

Log-odds score for T = $\log_2\left(\frac{0.214}{0.25}\right)$

Updating Table 2 with log-odds scores, we get Table 3.

Table 3: PWM with log-odds with log base 2 score and expected frequency of 0.25

Position / Nucleotide	1	2	3	4	5	6	7	8	9
A	0.194	0.514	-0.806	-1.816	-1.816	1.000	1.363	-1.816	-0.806
C	0.514	-0.806	-0.224	-1.816	-1.816	-1.816	-1.816	-0.806	-0.806
G	-0.806	-0.224	1.000	1.653	-1.816	-0.806	-0.224	1.514	-0.224
T	-0.224	0.194	-0.806	-1.816	1.653	-1.816	-1.816	-1.816	1.000

2.3 Scoring Sequences

Once we have finished the construction of PWM as shown in Table 3, we can score any unknown sequence and check if it is a splice site or not. To score a sequence motif, we have to add up the corresponding entries in the PWM. For example, to score the sequence `AGTGTAAGT`, we add up entries corresponding to nucleotides at that position highlighted in Table 4.

Table 4: PWM with highlighted score of sequence `AGTGTAAGT`

Position / Nucleotide	1	2	3	4	5	6	7	8	9
A	0.194	0.514	-0.806	-1.816	-1.816	1.000	1.363	-1.816	-0.806
C	0.514	-0.806	-0.224	-1.816	-1.816	-1.816	-1.816	-0.806	-0.806
G	-0.806	-0.224	1.000	1.653	-1.816	-0.806	-0.224	1.514	-0.224
T	-0.224	0.194	-0.806	-1.816	1.653	-1.816	-1.816	-1.816	1.000

Score of `AGTGTAAGT` = 0.194 + (-0.224) + (-0.806) + 1.653 + 1.653 + 1.000 + 1.363 + 1.514 + 1.000 = 7.347

The computed score is compared against a threshold. If the score is above the threshold, then we classify the sequence as a splice site otherwise we do not.

Thresholds are determined by using Receiver Operating Characteristic Curves (ROC). ROCs are explained in detail in Chapter 5.

2.4 Information Content and Sequence Logos

Information content is the amount of information about nucleotides at some position j and is represented in bits. It is given by [6]

$$I_j = 2 + \sum_{x \in \{A,C,G,T\}} f_{x,j} \log_2(f_{x,j})$$

where I_j is the information content at position j. $f_{x,j}$ is the relative frequency of nucleotide x at position j. If a nucleotide is very popular at any position specific position, the information content is very high. The highest value for information content is 2 bits. The information content is lowest if the nucleotides are equally distributed at any position j i.e. $f_{x,j}$ is 0.25. Information content can be represented as a logo as shown in Figure 1. In the sequence logo, the height of a character is proportional to its information content [6].

Figure 1: Sequence logo representing Example 1 [11]

In Figure 1, nucleotide G at position 4 and nucleotide T at position 5 are highly conserved and have the highest information content of 2 bits at their respective positions. Thus we can infer that most of the times, we will observe a G and a T in positions 4 and 5. However,

positions 1 and 2 do not show any nucleotides and hence the information content at these positions is very low.

2.5 Limitations of PWMs

PWMs are very simple and perform fairly well for the prediction of splice sites. However, they make a strong independence assumption between nucleotides at distinct positions which is rarely true. If we consider the example sequences, whenever there is a G in position 4, there is a T is position 5. It means, almost all the times, G and T appear together in splice sites. But, PWM fails to capture this interdependence. In order to overcome this limitation, we have used Maximal Dependence Decomposition (MDD) [4].

Lastly, PWMs also fails to recognize a sequence that contains part of splice site region and part of the non-splice site region. The percentage of sequences that belong to this category is very small, but these sites are also equally important. In order to score any sequence fairly, we have developed Hidden Markov Model (HMM).

In the next chapter, we will look into the working of MDD method that is used to address one of the drawbacks of PWMs.

CHAPTER 3

Maximal Dependence Decomposition

Maximal Dependence Decomposition (MDD) estimates the degree of dependence of nucleotide at position *j* on nucleotide at position *i*. More precisely, depending on nucleotide at position *i* being a consensus or not, MDD captures the dependence of nucleotide at position *j* on position *i*. Based on position *i*, splice site sequences are divided into two groups, C_i and $\overline{C_i}$ depending on whether the nucleotide at position *i* is a consensus nucleotide or not, respectively. In each group, we have to compute nucleotide frequencies at every position *j*. For any position *j*, we use chi-squared statistic to compare the two sets of frequencies. If there is dependence between positions *i* and *j*, then the frequencies are very different from one another. If there is no dependence between positions *i* and *j*, then the frequencies are very close. We can define chi-squared statistic as follows [6]

$$\frac{(N \times f_A - N_A)^2}{N \times f_A} + \frac{(N \times f_C - N_C)^2}{N \times f_C} + \frac{(N \times f_G - N_G)^2}{N \times f_G} + \frac{(N \times f_T - N_T)^2}{N \times f_T} \tag{3.1}$$

where,

N: Number of sequences in C_i

f_A, f_G, f_C, f_T: Frequencies of nucleotides at position *j* in sequences from $\overline{C_i}$.

N_A, N_G, N_C, N_T: Observed number of nucleotides at position *j* in sequences from C_i.

3.1 Algorithm

In this procedure, we iteratively quantify the dependence of position *j* on position *i* using chi-squared statistic, $\chi^2(j|i)$. Splice site sequences are divided at every iteration and the end

result is a tree with leaves representing a family of sequences which are alike. Each family of sequences is best represented with its own PWM that can be used to score unknown splice sites. The steps to carry out the MDD procedure are as follows [6]:

Step 1: Compute $S_i = \sum_{j \neq i} \chi^2(j, i)$ // Quantify the dependence on position *i*

Step 2: Select *i* such that S_i is maximum.

Step 3: Divide all the sequences into two groups C_i and $\overline{C_i}$ based on position *i*.

Step 4: Repeat steps 1, 2 and 3 for all the sequences in C_i and $\overline{C_i}$

Step 5: Terminate if there are less sequences or if there is not significant dependence between positions.

3.2 MDD Example

Let us consider a hypothetical example that consists of six sequences as shown below.

```
CGGG

CGTG

CGGC

ATGG

ATGT

ATGG

CGGG
```

Example 2: Hypothetical splice sites that show dependence between positions 1 and 2 [6]

We will now construct a table that captures dependencies between various nucleotide positions.

Table 5: χ^2 distribution table

Position *i*	Consensus	Position *j*				Sum
		1	2	3	4	
1	C	-				
2	G		-			
3	G			-		
4	G				-	

In Table 5, Position *i* represents distinct positions in a splice site. Consensus shows the consensus nucleotide at every position. Every cell is filled by computing the χ^2 value that shows the dependence between positions *i* and *j*. Sum column contains all the χ^2 values added up from that row. This completes the first step in MDD procedure.

For example: Let $i = 1$ and $j = 2$.

The consensus nucleotide at position 1 is C. Divide the sequences into groups C_i and $\overline{C_i}$ such that C_i has all the sequences that contain C in position 1 and $\overline{C_i}$ does not have C in position 1. Therefore, we have the following table that shows the sequences in C_i and $\overline{C_i}$

Table 6: Sequences in C_i and $\overline{C_i}$ after dividing based on C in position 1

C_i	$\overline{C_i}$
CGGG	ATGG
CGTG	ATGT
CGGC	ATGG
CGGG	

Let us now compute the χ^2 statistic as per the formula mentioned in 3.1. For the above example, when $i = 1$ and $j = 2$, we have:

$N = 8$ (Assuming Laplace correction/pseudocounts)

$$f_A = \frac{1}{7} \quad f_C = \frac{1}{7} \quad f_G = \frac{1}{7} \quad f_T = \frac{1}{7}$$

$$N_A = 1 \quad N_C = 1 \quad N_G = 5 \quad N_T = 1$$

$$\chi^2 = \frac{(8 \times 0.143 - 1)^2}{8 \times 0.143} + \frac{(8 \times 0.143 - 1)^2}{8 \times 0.143} + \frac{(8 \times 0.143 - 5)^2}{8 \times 0.143} + \frac{(8 \times 0.571 - 1)^2}{8 \times 0.571}$$

Therefore, $\chi^2 = 15.850$.

In Table 7, χ^2 statistic is high when $i = 1$ and $j = 2$ or $i = 2$ and $j = 1$. This shows that we do observe some dependence between positions 1 and 2 in the sequences from Example 2.

Table 7: χ^2 statistic table showing dependencies between positions *i* and *j*

Position *i*	Consensus	Position *j*				Sum
		1	2	3	4	
1	C	-	15.850	0.750	1.480	18.081
2	G	15.850	-	0.751	1.480	18.082
3	G	3.000	3.000	-	0.750	6.750
4	G	0.662	0.662	0.553	-	1.887

As per step 2, we select *i* such that Sum at *i* in Table 7 is the highest. In our case, $i = 2$. Based on position 2, we divide sequences into groups C_i and $\overline{C_i}$. Therefore, we will now have sequences in C_i that contain nucleotide G in position 2 and sequences in $\overline{C_i}$ that does not contain G in position 2, as shown in Table 8.

Table 8: Division of sequences into groups C_i and $\overline{C_i}$ based on nucleotide G in position 2

C_i	$\overline{C_i}$
CGGG	ATGG
CGTG	ATGT
CGGC	ATGG
CGGG	

We continue this iteratively until there is no significant dependence between nucleotide positions or if there is insufficient data. In our example, as seen in Table 8, the number of sequences in each group is less and hence there is insufficient data.

We will terminate the MDD procedure and build PWMs for each leaf of MDD tree. MDD for Example 2 will be:

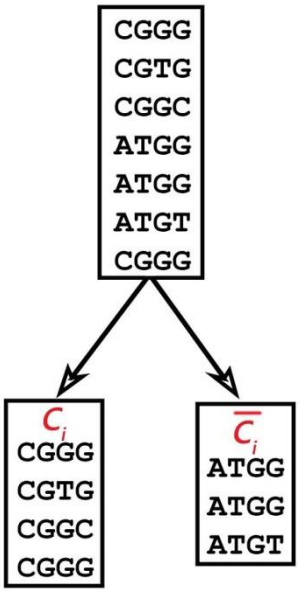

Figure 2: MDD tree for Example 2

PWMs for each leaf along with their sequence logos for Example 2:

Table 9: PWM for sequences in C_i

Position / Nucleotide	1	2	3	4
A	-1.163	-1.163	-1.163	-1.163
C	1.506	-0.816	-0.816	0.184
G	-0.816	1.506	1.184	1.184
T	-1.163	-1.163	-0.163	-1.163

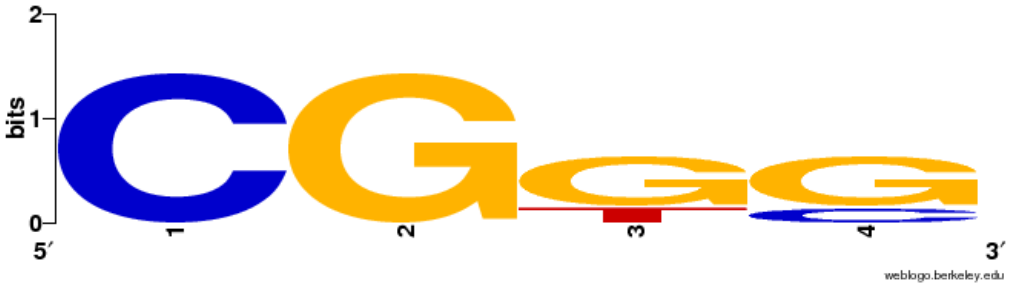

Figure 3: Sequence logo for sequences in C_i

Table 10: PWM for sequences in $\overline{C_i}$

Position / Nucleotide	1	2	3	4
A	1.029	-0.971	-0.971	-0.971
C	-0.623	-0.623	-0.623	-0.623
G	-0.623	-0.623	1.377	0.962
T	-0.971	1.029	-0.971	-0.029

Figure 4: Sequence logo for sequences in $\overline{C_i}$

3.3 Scoring Sequences

If an unknown sequence is given, then we can make the new sequence traverse down the tree until it reaches a leaf. Once the sequence reaches a leaf, we can use the PWM constructed for leaf to score the unknown sequence.

Consider an unknown sequence ATGC. In Figure 2, we divided the original sequences into two groups based on observing G at position 2. Therefore, while scoring an unknown sequence ATGC, check position 2 and assign it to one of the groups based on nucleotide observed at that position. Clearly, the unknown sequence does not belong to group C_i since it has T in position 2. So, we traverse to the right tree and check if it is a leaf. In our case, it is a leaf and hence we stop the traversal and score the unknown sequence with PWM of the leaf.

Hence,

Score of ATGC = 1.029 + 1.029 + 1.377 - 0.623 = 2.812

Again, we shall use ROC curves to determine a threshold to compare the score of unknown sequence against the threshold and classify it as a splice site or non-splice site.

MDDs work very well in capturing dependencies and overcoming the drawback of PWMs. In the next chapter, we shall address another drawback of PWMs by delving deeper into HMMs.

CHAPTER 4

Hidden Markov Models

Hidden Markov Model is a probabilistic model that contains a finite set of states. Each state is accompanied by a probability distribution. Transition probabilities are the probabilities that govern the transition among different states. Each state emits certain observation with some probability and this is termed as emission probability. For any observation sequence, the set of states that emitted each outcome is hidden.

We have used HMM to address the second drawback of PWMs. PWMs do not recognize the sequences that are neither splice sites nor non-splice sites. They will classify these kinds of sequences as either splice or non-splice site categories. Therefore, we shall use HMMs that capture splice sites, non-splice sites and sites that contain both splice and non-splice site regions.

4.1 Definition

N: Number of hidden states

Set of states $Q = \{1, 2, \dots, N\}$

M: Number of symbols

Set of symbols $V = \{1, 2, \dots, M\}$

A: State-transition probability matrix

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

B: Emission probability distribution; k is a symbol

$$b_j(k) = P(o_t = k | q_t = j) \qquad 1 \leq i, j \leq M$$

The initial state distribution π

$$\pi_i = P(q_1 = i)$$

The model is given by [14]

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

4.2 Algorithm

Given an observation sequence $O = \{o_1, o_2, o_3, \dots, o_T\}$ we shall estimate the model parameters $\lambda = (A, B, \pi)$ that maximize $P(O|\lambda)$ using Baum-Welch algorithm [13, 14].

Step 1: Initial condition

Set $\lambda = (A, B, \pi)$ such that the initial, emission and transition probabilities are chosen using the information from the data. Section 4.4 discusses how we set initial, emission and transition probabilities for our project.

Step 2: Compute forward variable, $\alpha_t(i) = P(o_1, o_2, o_3, \dots, o_T, q_t = i|\lambda)$ where $\alpha_t(i)$ is the probability of observing the partial sequence $\{o_1, o_2, o_3, \dots, o_t\}$ and landing in state *i* at stage *t*.

The forward variable is computed using the recurrence relation

Initialization:

$$\alpha_1(i) = \pi_i b_i(o_1) \qquad (4.1)$$

Recursion:

$$\alpha_{t+1}(j) = b_j(o_{t+1}) \left[\sum_{i=1}^{N} \alpha_t(i)\, a_{ij} \right] \tag{4.2}$$

Alternatively, one could use the backward algorithm.

Step 3: Compute backward variable, $\beta_t(i) = P(o_1, o_2, o_3, \dots, o_t | q_t = i, \lambda)$, where $\beta_t(i)$ is the probability of observing partial sequence $\{o_1, o_2, o_3, \dots, o_t\}$ at the end given that the starting state is *i* at time *t*.

Backward variable is computed using the recurrence relation

Initialization:

$$\beta_T(i) = 1 \tag{4.3}$$

Recursion:

$$\beta_i(t) = \sum_{j=1}^{N} a_{ij} b_j(o_{t+1}) \beta_{t+1}(j) \tag{4.4}$$

Step 4: We will use the forward and backward variables computed in steps 2 and 3 to compute temporary variable γ and ξ using Bayes' theorem

$$\gamma_i(t) = P(q_t = i | O, \lambda) = \frac{\alpha_i(t)\, \beta_i(t)}{\sum_{j=1}^{N} \alpha_j(t)\, \beta_j(t)} \tag{4.5}$$

where γ is the probability of being in state *i* given that the observed sequence is O and parameters are λ at time *t*.

If ξ is the probability of observing a sequence O with parameters λ such that you are in states *i* and *j* at times *t* and *t+1*, then we have:

$$\xi_{ij}(t) = \frac{\alpha_i(t)\, a_{ij}\, \beta_j(t+1)\, b_j(o_{t+1})}{\sum_{k=1}^{N} \alpha_k(t)\, \beta_k(t)} \tag{4.6}$$

Step 5: Update initial, transition and emission probabilities

$$\pi_i^* = \gamma_i(1) \tag{4.7}$$

Equation 4.7 represents the expected frequency in state *i* at time, t = 1:

$$a_{ij}^* = \frac{\sum_{t=1}^{T-1} \xi_{ij}(t)}{\sum_{t=1}^{T-1} \gamma_i(t)} \tag{4.8}$$

Equation 4.8 represents the expected number of transitions from state *i* to state *j* compared to expected total number of transitions away from state *i*

$$b_i^*(v_k) = \frac{\sum_{t=1}^{T} 1_{o_t = v_k} \gamma_i(t)}{\sum_{t=1}^{T} \gamma_i(t)} \tag{4.9}$$

where

$$1_{o_t = v_k} = \begin{cases} 1, & if\ y_t = q_k \\ 0, & otherwise \end{cases}$$

$b_i^*(v_k)$ is the expected number of times, we observe the nucleotide v_k in state *i* over the expected total number of times we observe all the nucleotides in state *i*.

Step 6: Repeat from step 2 until convergence is reached.

4.3 Multiple Observation Sequences

Multiple observation sequences are required for a model to have reliable estimates of all the model parameters. Hence we have to re-estimate formulas for multiple observation sequences [15]. Assuming that there are K observation sequences,

$$O = \{O(1), O(2), \dots, O(K)\}$$

Where $O(k) = \{o^1(k), o^2(k), o^3(k), \dots, o^T(k)\}$ is the k[th] observation sequence. Assuming that all the observation sequences are independent of one another, we can maximize the model parameter for all the observation sequences as follows:

$$P(O|\lambda) = \prod_{k=1}^{K} P(O(K)|\lambda)$$

Emission and transition probability formulas for re-estimation are given by

$$a_{ij}^{*} = \frac{\sum_{k=1}^{K}\sum_{t=1}^{T-1}\xi_{ij}(t)}{\sum_{k=1}^{K}\sum_{t=1}^{T-1}\gamma_i(t)}$$

$$b_i^{*}(v_k) = \frac{\sum_{k=1}^{K}\sum_{t=1}^{T} 1_{o_t = v_k}\gamma_i(t)}{\sum_{k=1}^{K}\sum_{t=1}^{T}\gamma_i(t)}$$

where

$$1_{o_t = v_k} = \begin{cases} 1, & if\ y_t = q_k \\ 0, & otherwise \end{cases}$$

Initial probabilities π_i is not re-estimated.

4.4 Initial Parameters

During the learning process of HMM, we assume initial estimates of all the probabilities based on the data. Assume we have M splice sites and N non-splice sites, each sequence is K nucleotides long. We construct a hidden Markov model with 2K states as shown below.

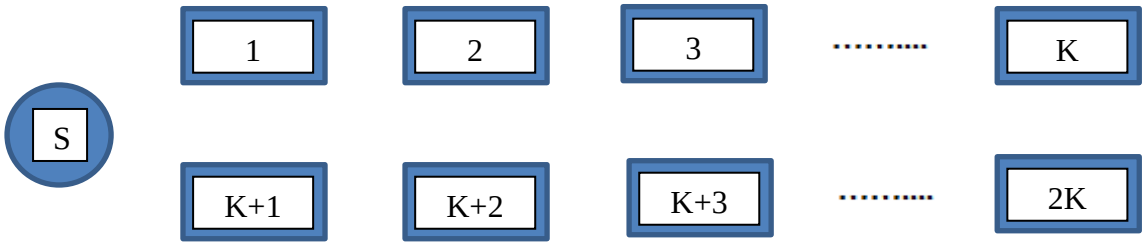

Figure 5: Model showing 2K states

The first K states represent all K positions in splice sites and the last K states represent all K positions in non-splice sites. Based on this assumption, we can compute initial, transition and emission probabilities.

Initial Probabilities, π:

Since any sequence can belong to either a splice site or non-splice site, so it can either start from state 1 or state K+1 i.e., the first nucleotide of splice site or non-splice site oligomer. Hence, we can update initial probabilities as follows:

Table 11: Initial probabilities for 2K states

States	1	2	……	K+1	……	2K
Π	0.5	0	……	0.5	……	0

In a human genome, we know the distribution of splice sites is much less compared to that of non-splice sites so, we can tune the initial probabilities from Table 11 even more by reducing the probability to state 1 and increasing the probability to state K+1.

Transition Probabilities, A:

From the data, we have 2K matching states. From any state k in Figure 5, we can go to the next state, k+1 in splice site region ($1 \leq k < k+1 \leq K$) or the next state, $K+k+1$ in the non-splice site region ($K+1 < K+k+1 \leq 2K$) only. But, the transition from a state in splice site region to splice site region is higher than the transition from a state in splice site region to non-splice site region and vice versa. Therefore, we can assume that the transitions from one region to itself and another follow the distribution as shown in Table 12.

Table 12: Transition probabilities from one region to another and itself

Region	Splice Site	Non-splice site
Splice Site	0.9	0.1
Non-splice site	0.1	0.9

Updating these probabilities in Figure 5, we get the model of Figure 6.

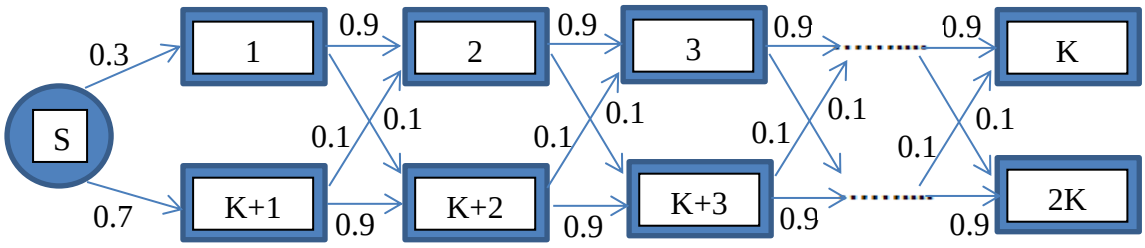

Figure 6: Model showing 2K states with initial probabilities and transition probabilities

Emission Probabilities, B:

Emission probabilities at specific state are initialized based on distribution of nucleotides at that position. For example, from a set of aligned splice sites, the distribution of nucleotides at position 1 constitutes the emission probabilities at state 1, distribution of nucleotides at position 2 constitute the emission probabilities at state 2 and so on. From a set of aligned non-

splice sites, distribution of nucleotides at position 1 constitutes the emission probabilities at state K+1, distribution of nucleotides at position 2 constitute the emission probabilities at state K+2 and so on.

4.5 Scoring Sequences

Once we learn the model using the Baum-Welch algorithm, we need to use it to score the unknown sequences in order to classify them as splice sites or non-splice sites.

In order to score an unknown sequence, O, we first need to find the optimal state sequence associated with O since the states are hidden. This is called decoding. Once the decoded states are obtained, we use them to compute P(O). This probability represents the score of an unknown sequence O.

To find the optimal state sequence, we use Viterbi algorithm. Instead of exhaustively searching for the most likely path of unknown sequence, we use dynamic programming to find the best path.

Viterbi Algorithm [14, 16]:

Let $\delta_t(i)$ be the highest probability path ending in state i at time t, we have to compute

$$\delta_t(i) = \max_{q_1,q_2,\dots,q_t} P(q_1, q_2, \dots, q_t = i, o_1, o_2, \dots, o_t|\lambda)$$

The recursion procedure for the Viterbi is as follows:

Initialization:

$$\delta^1(i) = \pi_i b_i(o_1) \qquad 1 \leq i \leq N$$

$$\psi^1(i) = 0$$

Recursion:

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

$$\psi_t(j) = \max_{1 \le i \le N} \left[\delta_{t-1}(j) a_{ij}\right] \qquad 1 \le i \le N, 2 \le t \le T$$

Termination:

$$P_T^* = \max_{1 \le i \le N} [\delta_T(i)]$$

$$q_T = \arg\max_{1 \le i \le N} [\delta_T(i)]$$

where

$$P_T^* = P\,(q^1, q^2, \dots, q^T | O, \lambda)$$

We can obtain the maximum likelihood path $q^* = (q_1^*, q_2^*, \dots, q_3^*)$ where

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

4.6 Using Logarithms to Avoid Underflow

Multiplication of probabilities in every step of training will lead to underflows. For our case, we observe underflow at the very first iteration of training. In order to avoid them, we have used logarithms and rewritten all the formulas mentioned in algorithm [17].

Using logarithms for equations 4.1 and 4.2 from forward procedure, we get:

$$\log_e \alpha_1(i) = \log_e \pi_i + \log_e b_i(o_1) \tag{4.10}$$

$$\log_e \alpha_{t+1}(j) = \log_e b_j(o_{t+1}) + \log_e \sum_{i=1}^{N} e^{\log_e \alpha_t(i) + \log_e a_{ij}} \tag{4.11}$$

Using logarithms for equations 4.3 and 4.4 from forward procedure, we get:

$$\log_e \beta_i(T) = 0 \tag{4.12}$$

$$\log_e \beta_i(t) = \log_e \sum_{j=1}^{N} e^{\log_e a_{ij} + \log_e b_j(o_{t+1}) + \log_e \beta_{t+1(j)}} \tag{4.13}$$

Using logarithms for equations 4.5 and 4.6 that are used to compute the temporary variables, we get:

$$\log_e \gamma_i(t) = \log_e \alpha_i(t) + \log_e \beta_i(t) - \log_e \sum_{j=1}^{N} e^{\log_e \alpha_j(t) + \log_e \beta_j(t)} \tag{4.14}$$

$$\log_e \xi_{ij}(t) = \log_e \alpha_i(t) + \log_e a_{ij} + \log_e \beta_j(t+1) + \log_e b_j(o_{t+1}) - \log_e \sum_{k=1}^{N} e^{\log_e \alpha_k(t) + \log_e \beta_k(t)} \tag{4.15}$$

Using logarithms for equations 4.7, 4.8 and 4.9 used to update HMM parameters, we get:

$$\log_e \pi_i^* = \log_e \gamma_i(1) \tag{4.16}$$

$$\log_e a_{ij}^* = \log_e \sum_{t=1}^{T-1} e^{\log_e \xi_{ij}(t)} - \log_e \sum_{t=1}^{T-1} e^{\log_e \gamma_i(t)} \tag{4.17}$$

$$\log_e b_i^*(v_k) = \log_e \sum_{t=1}^{T} e^{\log_e 1_{o_t = v_k} + \log_e \gamma_i(t)} - \log_e \sum_{t=1}^{T} e^{\log_e \gamma_i(t)} \tag{4.18}$$

The above approach works very well in avoiding underflows, but it is computationally expensive since it requires calculation of logarithms in each step.

4.7 HMM Example

In the previous subsections, we have seen how to set initial parameters and train the HMM for splice site and non-splice site oligomers. In this section, we assume that the sites in Example 3 are hypothetical splice sites and non-splice sites. By using them, we are going to construct the HMM and train them.

In a normal human genome, the number of splice sites is much less than those of non-splice sites. In order to replicate the same, we have considered skewed dataset consisting of less splice sites than those of the non-splice sites.

Splice Sites	Non-splice Sites
CGGG	ATGG
CGTG	ATGT
CGGC	ATGG
	ATTT
	ATGC
	ATGA

Example 3: Hypothetical splice site and non-splice site oligomers

Looking at the length of sequences, we can assume that there are 2K states where, K = 4 being the length of each sequence. Hence, the HMM without transition and emission probabilities will be:

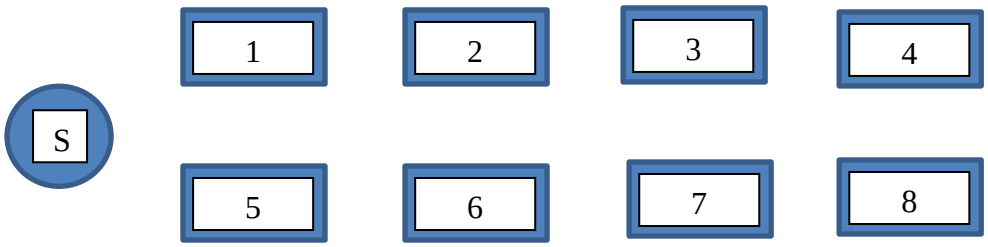

Figure 7: Model showing 8 states that represents sequences from Example 3

Step 1: Initial condition. Based on data, let us set the initial parameters

Initial Probabilities, π:

Initial probability to state 1 (first position of splice sites)

$$\log_e \pi_i = \log_e(0.3) = -1.204$$

Initial probability to state 5 (first position of non-splice sites)

$$\log_e \pi_i = \log_e(0.7) = -0.357$$

Assuming initial probabilities to other states is 0 and hence (assuming -*99* is infinity in our case)

$$\log_e(0) = -99$$

Table 13: Initial probabilities for 8 states for Example 3

States	1	2	3	4	5	6	7	8
π	-1.204	-99	-99	-99	-0.357	-99	-99	-99

Transition Probabilities, A:

From any state *k*, we are moving only to two states, *k+1* or *K+k+1* where $1 \leq k < K/2$ and *k* to *k+1* or *K-k+1* where $K/2 \leq k < K$ with probabilities 0.9 and 0.1, respectively.

Transition probability from state *k* to state *k+1* is 0.9,

$$\log_e(0.9) = -0.22$$

Transition probability to state *k* to *K+k+1* or *K-k+1* is 0.1,

$$\log_e(0.1) = -1.609$$

Transition probabilities to any other states from *k* is 0,

$$\log_e(0) = -99$$

Table 14: Transition probabilities that follow Table 12 for Example 3

States	1	2	3	4	5	6	7	8
1	-99	-0.223	-99	-99	-99	-99	-99	-99
2	-99	-99	-0.223	-99	-99	-99	-99	-99
3	-99	-99	-99	-0.223	-99	-99	-99	-99
4	-99	-99	-99	-99	-0.223	-99	-99	-99
5	-99	-1.609	-99	-99	-99	-0.223	-99	-99
6	-99	-99	-1.609	-99	-99	-99	-0.223	-99
7	-99	-99	-99	-1.609	-99	-99	-99	-0.223
8	-99	-99	-99	-99	-99	-99	-99	-99

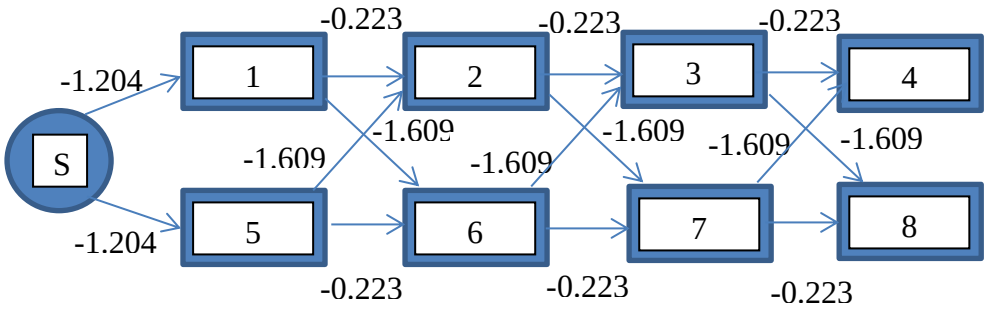

Figure 8: Model showing 8 states with initial and transition probabilities for Example 3

Emission Probability, B:

Emission probabilities are calculated using pseudocounts. Let us count the first entry in Table 15. Number of A's at first position in splice site data is 0 out of 3 sequences. Therefore, emission probability of A at state 1 is given by:

$$b_1(A) = \frac{0+1}{3+4} = 0.1428$$

$$\log_e(0.1428) = -1.946$$

Other entries are also calculated in a similar fashion and are used to fill Table 15.

Table 15: Emission probabilities of nucleotides table for all 8 states for Example 3

States / Nucleotides	1	2	3	4	5	6	7	8
A	-1.946	-1.946	-1.946	-1.946	-0.357	-2.303	-2.303	-1.609
C	-0.559	-1.946	-1.946	-1.253	-2.303	-2.303	-2.303	-1.609
G	-1.946	-0.559	-0.847	-0.847	-2.303	-2.303	-0.511	-1.204
T	-1.946	-1.946	-1.253	-1.945	-2.303	-0.357	-1.609	-1.204

Step 2: Take one training sequence, CGGG for example and for that training sequence, perform the following:

Compute forward probabilities

Using formulas 4.10 and 4.11, Table 16 is filled.

For example: When i = 1 (entry in the first cell of Table 16)

$$\log_e \alpha_1(i) = -1.204 - 0.559 = -1.764$$

Similarly, we compute the first column of Table 16 which is the initialization in the forward procedure.

We now compute the first entry in the second column. Therefore, *j = 1, t+1 = G*

$$\log_e \alpha_{t+1}(j) = -1.946 + \log_e(e^{-1.764-99} + e^{-100.95-99} + \ldots) = -102.367$$

Similarly, we fill the entries in column 2 and then use the column 2 to fill column 3 and so on.

Table 16: Forward probabilities of nucleotides for all 8 states for Example 3

Nucleotides / States	C	G	G	G
1	-1.764	-102.367	-103.332	-104.364
2	-100.946	-2.449	-101.608	-102.602
3	-100.946	-100.826	-3.504	-102.208
4	-100.253	-100.826	-101.197	-4.552
5	-2.659	-102.058	-102.813	-103.409
6	-101.303	-5.185	-103.346	-104.289
7	-101.303	-100.646	-5.919	-102.654
8	-100.609	-101.339	-101.605	-7.346

Step 3: Compute backward probabilities

We use formulas 4.12 and 4.13, to fill the entries of Table 17.

For example: When i = 1,

$$\log_e \beta_i(T) = 0$$

We fill the last column of Table 17 with 0 which is the initialization in backward procedure.

We now compute the first entry in the third column. Therefore, *i = 1, t = G*

$$\log_e \beta_i(t) = \log_e(e^{-99-1.946+0} + e^{-0.223-0.559+0} + \ldots) = -0.783$$

Similarly, we fill the entries in column 3 and then use the column 3 to fill column 2 and so on.

Table 17: Backward probabilities of nucleotides for all 8 states for Example 3

Nucleotides / States	C	G	G	G
1	-2.924	-1.853	-0.783	0.000
2	-4.667	-2.141	-1.070	0.000
3	-5.235	-3.596	-1.070	0.000
4	-4.997	-4.164	-2.526	0.000
5	-3.565	-2.472	-1.638	0.000
6	-5.177	-1.683	-0.569	0.000
7	-6.621	-4.982	-1.122	0.000
8	-100.972	-99.337	-98.017	0.000

Step 4: Calculate temporary variables γ and ξ

Using formula 4.14, we fill the entries of Table 18.

For example: When i = 1, we compute the first entry in the first column of Table 18.

Therefore, *i = 1*, *t = C*

$$\log_e \gamma_i(t) = -1.76 - 2.92 - \log_e(e^{-1.8-2.92} + e^{-100.95-4.67} + \ \ldots) = -0.195$$

Similarly, we fill each cell in Table 18 that forms the γ table used to update HMM parameters.

Table 18: Table representing temporary variable γ for all 8 states for Example 3

Nucleotide / States	C	G	G	G
1	-0.195	-99.728	-99.622	-99.871
2	-101.120	-0.098	-98.186	-98.109
3	-101.688	-99.929	-0.081	-97.716
4	-100.758	-100.498	-99.230	-0.059
5	-1.732	-100.037	-99.959	-98.916
6	-101.988	-2.376	-99.423	-99.797
7	-103.431	-101.136	-2.548	-98.162
8	-197.089	-196.184	-195.130	-2.854

Using formula 4.15, we fill the entries of Tables 19 and 20.

For example: When i = 1 and j = 1 (keeping j constant), we compute the first entry in the first column of Table 19. Therefore, *i = 1, j = 1, t = C*

$$\log_e \xi_{ij}(t) = -1.8 - 99 - 1.9 - 1.95 - \log_e(e^{-1.8-2.92} + e^{-100.95-4.67} + \ \ldots)$$

$$\log_e \xi_{ij}(t) = -100.07$$

Similarly, we fill each entry in Table 19 that forms the ξ table for 1 state (say j = 1) used to update HMM parameters.

Table 19: Table representing temporary variable ξ for 1st state for Example 3

Nucleotide / State	C	G	G	G
1	-100.070	-199.603	-199.786	0.000
2	-0.195	-99.728	-99.622	0.000
3	-100.715	-198.792	-198.687	0.000
4	-101.283	-200.248	-198.687	0.000
5	-101.045	-200.816	-200.142	0.000
6	-100.257	-199.747	-200.142	0.000
7	-101.764	-198.507	-198.351	0.000
8	-196.812	-296.096	-199.044	0.000

We perform the same for each state j from 1 to N. So, we will get 8 tables with entries as shown in Tables 19 and 20. These are two of the 8 tables we need to compute the temporary variable ξ.

Once the temporary variables are calculated, new HMM parameters are updated using formulas 4.16, 4.17 and 4.18. This is for a single observation sequence. The process from Step 2 needs to be performed for all the observation sequences and parameters are updated as per section 4.3. The training continues till convergence is reached.

Table 20: Table representing temporary variable ξ for 2nd state for Example 3

Nucleotide / State	C	G	G	G
1	-199.253	-99.685	-198.061	0.000
2	-198.154	-98.587	-196.675	0.000
3	-101.120	-0.098	-98.186	0.000
4	-200.465	-100.329	-196.963	0.000
5	-200.228	-100.898	-198.418	0.000
6	-199.439	-99.829	-198.418	0.000
7	-200.947	-98.589	-196.626	0.000
8	-295.994	-196.178	-197.319	0.000

Once the HMM is built, we use the Viterbi algorithm to decode the sequence and score it using the decoded sequence which represents the score of an unknown sequence. Once the score is computed, it is compared against the threshold to classify it as a splice site or not. The threshold is obtained a using ROC curve which is discussed in the next chapter.

CHAPTER 5

Receiver Operating Characteristic Curve

In our method, we need to compare the scores of unknown sequences to a threshold value. The threshold value is determined by two factors: sensitivity and specificity. Sensitivity is the true positive rate meaning the fraction of real positives identified correctly. Specificity is the true negative rate meaning the fraction of real negatives identified correctly [8]. These two values describe how well our model discriminates between match (with splice site) or not.

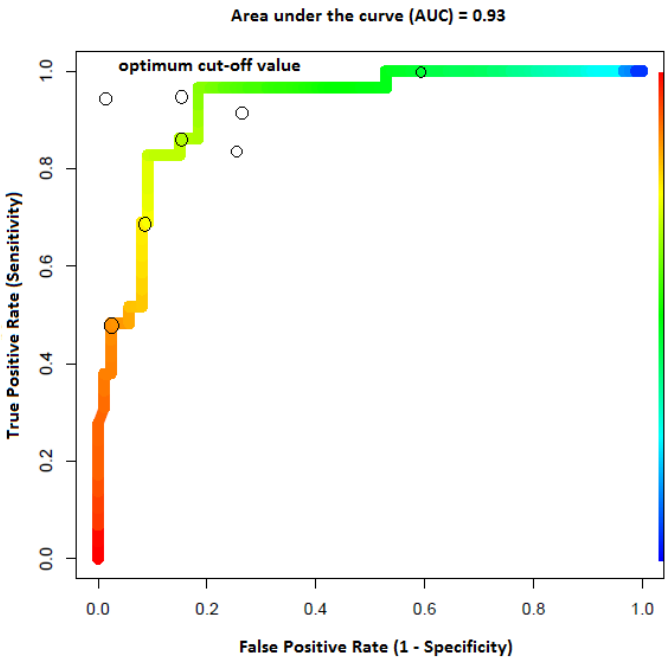

Figure 9: Receiver Operating Characteristic Curve

The receiver operating characteristic (ROC) curve is a plot of true positive rate vs the false positive rate for different scores of a test. As shown in Figure 9, ROC curve is a plot of sensitivity vs 1-specificity. A trade-off between sensitivity and specificity gives the score which we use as the threshold or the optimum cut-off value. So, we aim at maximizing the true positive rate with acceptable false positive rate.

Accuracy of the test is higher if in ROC space, we are closer to the upper left corner. The area under the curve (AUC) is one of the popular measures that determine the model's discrimination potential. Entire AUC is 1.0. If any test results in AUC = 1.0, then it is a perfect model for given set of training and testing data. However, if AUC = 0.5, then we will observe a diagonal line which indicates that it does not yield any meaningful result. It just means that we are performing the test by randomly guessing which is equivalent to tossing a coin.

If we see that the curve is closer to the lower right corner, then it is a perfectly worst test since it is performing worse than random guessing. In such cases, we just flip the data (i.e. we change the positive set to negative set and vice versa) so that it becomes a perfect test with high AUC.

In the next chapter, we use the ROC curves to evaluate our models and determine the threshold scores that we use to classify unknown sites as splice sites or not.

CHAPTER 6

Datasets and Results

We obtained the splice site data from Homo Sapiens Splice Site Database (HS3D) [5]. The database has a collection of authentic 5' and 3' splice sites. The database has data for both splice sites and non-splice sites. The dataset consisted of long sequences from which 5' and 3' splice sites were extracted for splice sites and non-splice sites. Once the sites are extracted, they are aligned. Aligned sites are used to train and test our methods. We also performed ten-fold cross validation to evaluate our models.

We also collected cryptic splice sites datasets for breast cancer type1 susceptibility (BRCA1) protein and beta globin gene (HBB) from the literature. We then trained and tested our models with cryptic splice sites and found interesting results.

6.1 Ten-Fold Cross Validation Procedure

In each round of cross-validation procedure, we divide the data into non-overlapping subsets. We train the model with one subset of data and validate it with the other subset. In order to reduce the bias of one round of cross-validation, we perform multiple rounds with distinct splits. The results of validation are then averaged over all the rounds [10].

Positive and negative scores are obtained after every round of ten-fold cross-validation and ROC curves are generated. We compute the average of all the ten area under ROC curves (AUC) which is considered as a final value.

The ROC curves are color coded with threshold scores shown on the right. Depending on trade-off between sensitivity and specificity, we can choose the best threshold for our dataset. Next we examine the ROCs of different methods for our dataset.

6.2 5’ Splice Site Dataset

Length of sites: 9

Training set containing true sites: 2516

Training set containing false sites: 7560

Testing set containing true sites in one fold: 280

Testing set containing false sites in one fold: 840

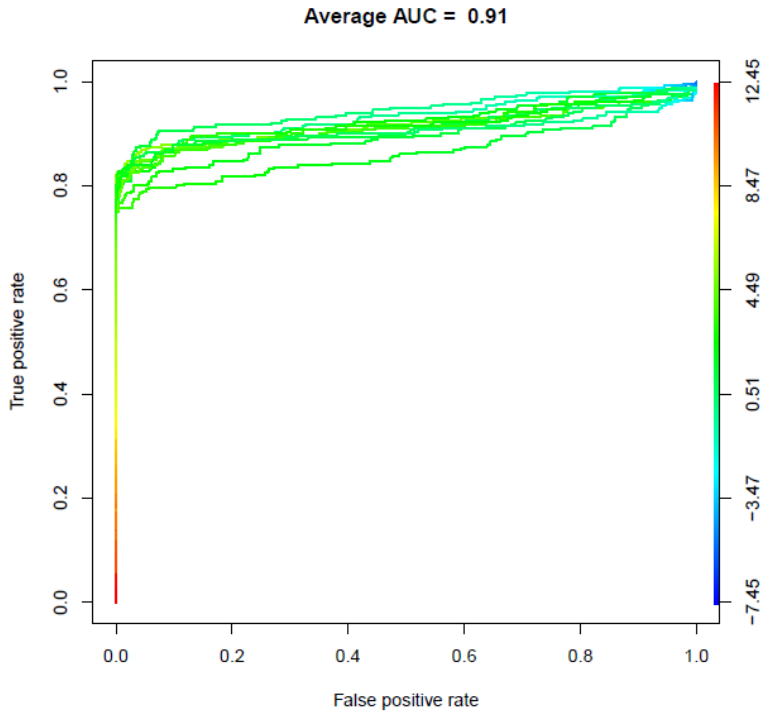

Figure 10: 10-fold ROC curve for simple PWM for 5’ splice site model

Figure 10 shows the performance of simple PWM for 5’ splice site model. We choose the threshold score such that the classification error is the lowest. With a threshold score of 12, we get high true positives and tolerable number of false positives. The prediction accuracy of PWM for 5’ splice site oligomers is 0.91 with a threshold score of 12 and is computed as an average of prediction accuracies of each fold in a 10-fold cross validation procedure.

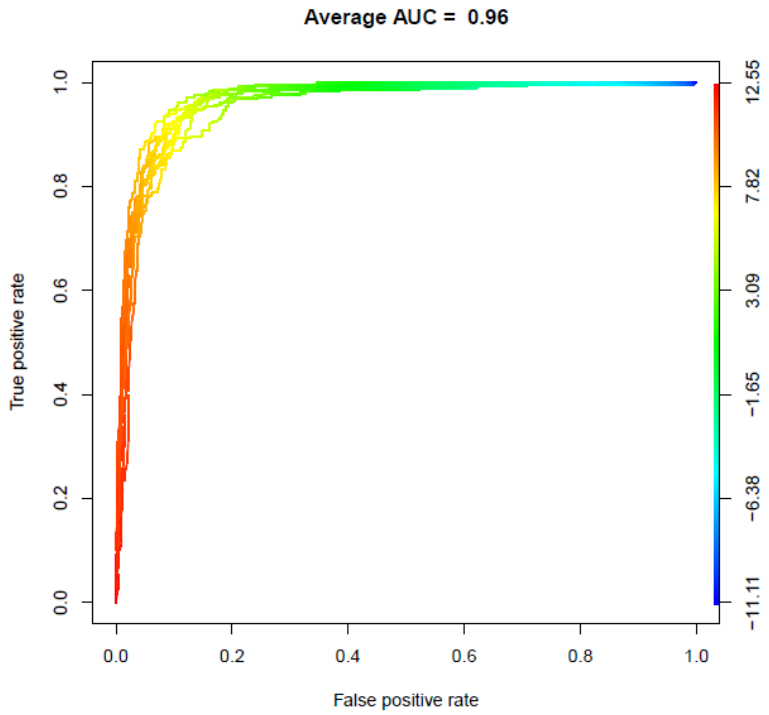

Figure 11: 10-fold ROC curve for MDD for 5’ splice site model

Figure 11 shows the performance of MDD for 5’ splice site model. With a threshold score of 12.2, we get a good balance of true positives and false positives. The prediction accuracy of MDD for 5’ splice site oligomers is 0.96 with a threshold score of 12.2 and is computed as an average of prediction accuracies of each fold in a 10-fold cross validation procedure.

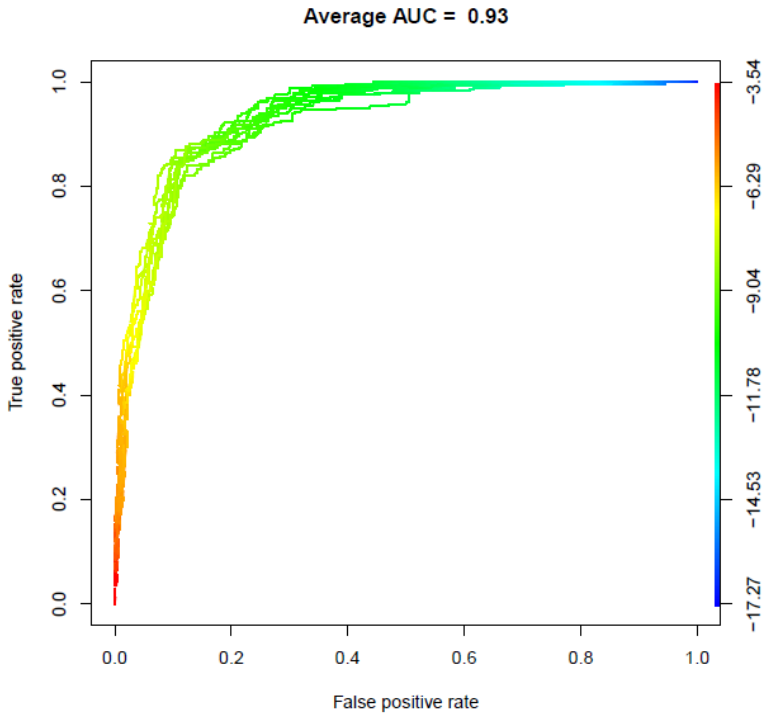

Figure 12: 10-fold ROC curve for HMM for 5' splice site model

Figure 12 shows the performance of HMM for 5' splice site model. The performance of HMM is greatly affected by varying initial probabilities. To achieve the best prediction accuracy, we trained the model with initial probabilities of 0.2 and 0.8 to splice site state and non-splice site state respectively. With a threshold score of -4.1, we get a good balance of true positives and false positives. The prediction accuracy of HMM for 5' splice site oligomers is 0.93 with a threshold score of -4.1 and is computed as an average of prediction accuracies of each fold in a 10-fold cross validation procedure.

As can be seen from Figures 10, 11 and 12, we can conclude that MDD performs very well with the AUC of 0.96 for 5' splice site dataset.

6.3 3’ Splice Site Dataset

Length of sites: 14

Training set containing true sites: 2592

Training set containing false sites: 7776

Testing set containing true sites in one fold: 288

Testing set containing false sites in one fold: 864

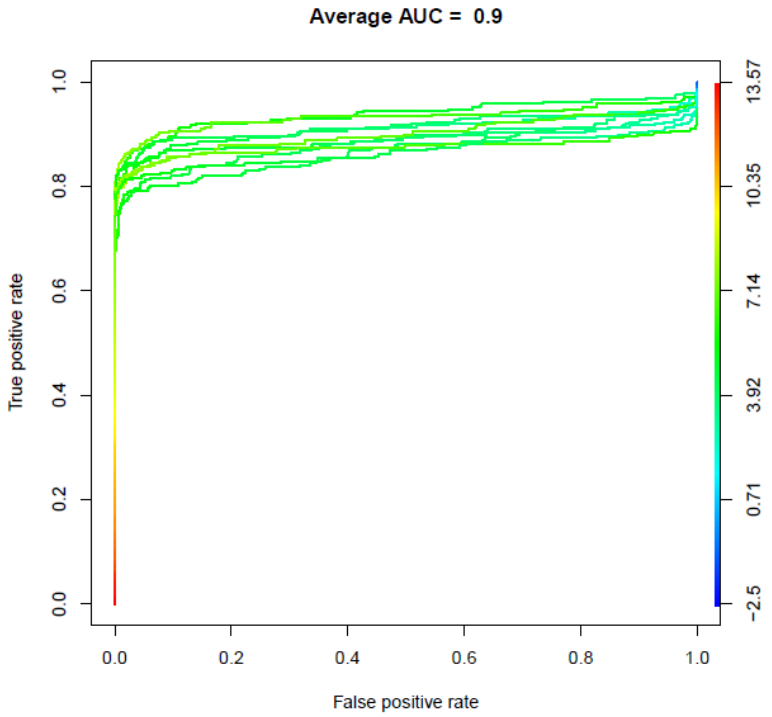

Figure 13: 10-fold ROC curve for simple PWM for 3’ splice site model

Figure 13 shows the performance of simple PWM for 3’ splice site model. With a threshold score of 13.5, we get high true positives and tolerable number of false positives. The prediction accuracy of PWM for 3’ splice site oligomers is 0.9 with a threshold score of 13.5 and is computed as an average of prediction accuracies of each fold in a 10-fold cross validation procedure.

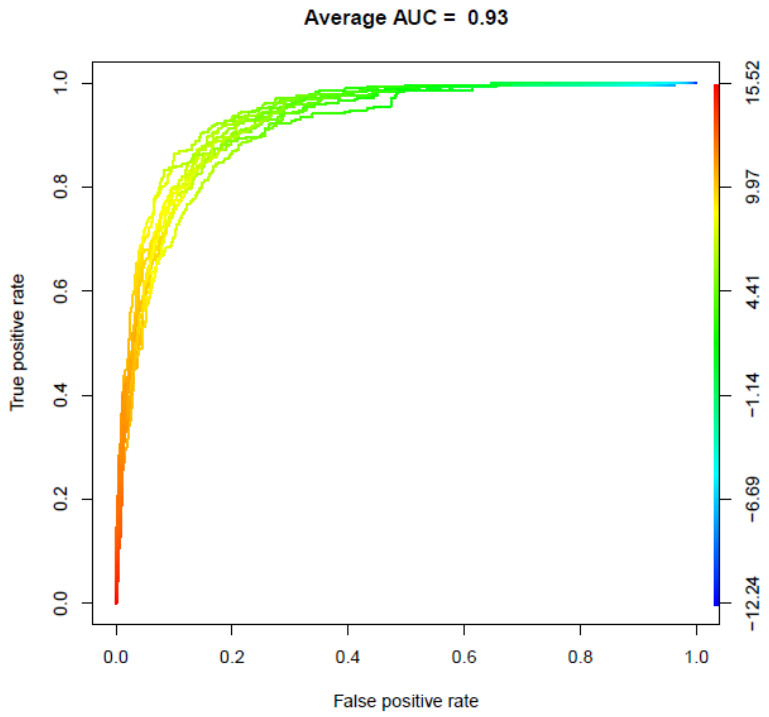

Figure 14: 10-fold ROC curve for MDD for 3’ splice site model

Figure 14 shows the performance of MDD for 3’ splice site model. With a threshold score of 15.1, we get a good balance of true positives and false positives. The prediction accuracy of MDD for 3’ splice site oligomers is 0.93 with a threshold score of 15.1 and is computed as an average of prediction accuracies of each fold in a 10-fold cross validation procedure.

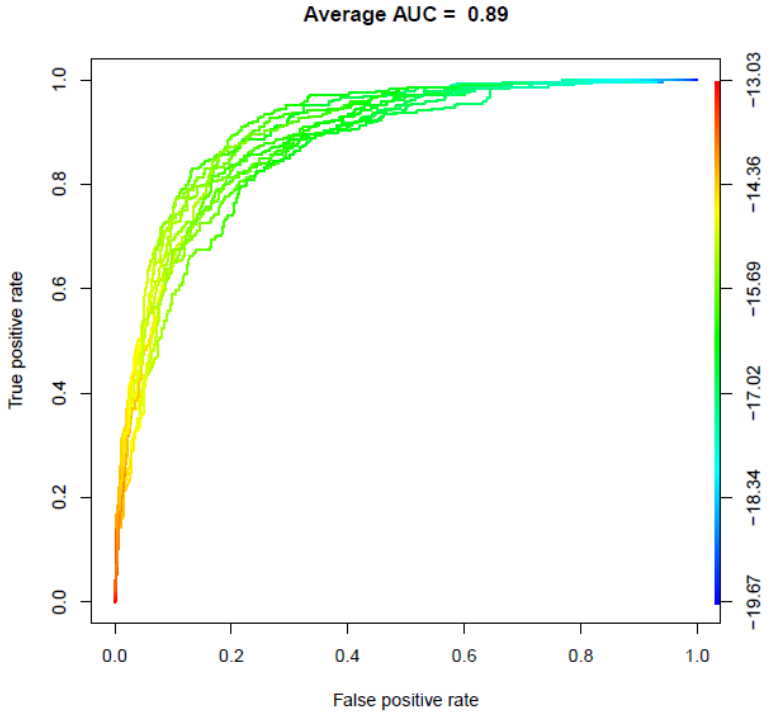

Figure 15: 10-fold ROC curve for HMM for 3' splice site model

Figure 15 shows the performance of HMM for 3' splice site model. To achieve the best prediction accuracy, we trained the model with initial probabilities of 0.3 and 0.7 to splice site state and non-splice site state respectively. With a threshold score of -13.5, we get a good balance of true positives and false positives. The prediction accuracy of HMM for 5' splice site oligomers is 0.89 with a threshold score of -13.5 and is computed as an average of prediction accuracies of each fold in a 10-fold cross validation procedure.

As can be seen from Figures 13, 14 and 15, we can conclude that MDD performs very well with the AUC of 0.93 for 3' splice site dataset given that the threshold score is above 15.1.

6.4 BRCA1 5’ Cryptic Splice Site Dataset

Length of sites: 9

Training set containing true sites: 29

Training set containing false sites: 87

Testing set containing true sites: 29

Testing set containing false sites: 87

For BRCA1 5’cryptic splice sites, the data used for training and testing the models are the same.

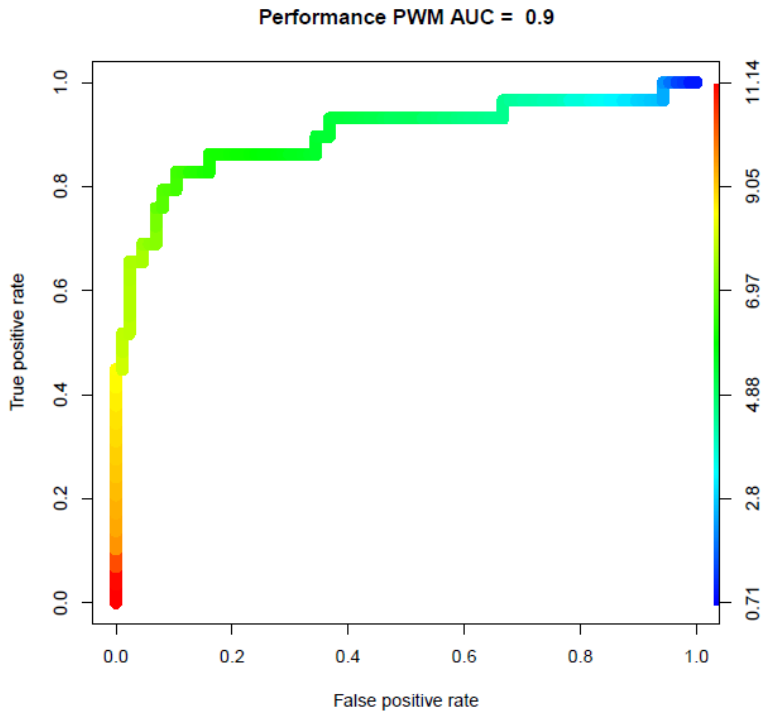

Figure 16: ROC curve for simple PWM for 5’ BRCA1 cryptic splice site model

Figure 16 shows the performance of simple PWM for BRCA1 5’ cryptic splice site model. With a threshold score of 10.8, we get high true positives and tolerable number of false

positives. The prediction accuracy of PWM for BRCA1 5’ cryptic splice site oligomers is 0.9 with a threshold score of 10.8.

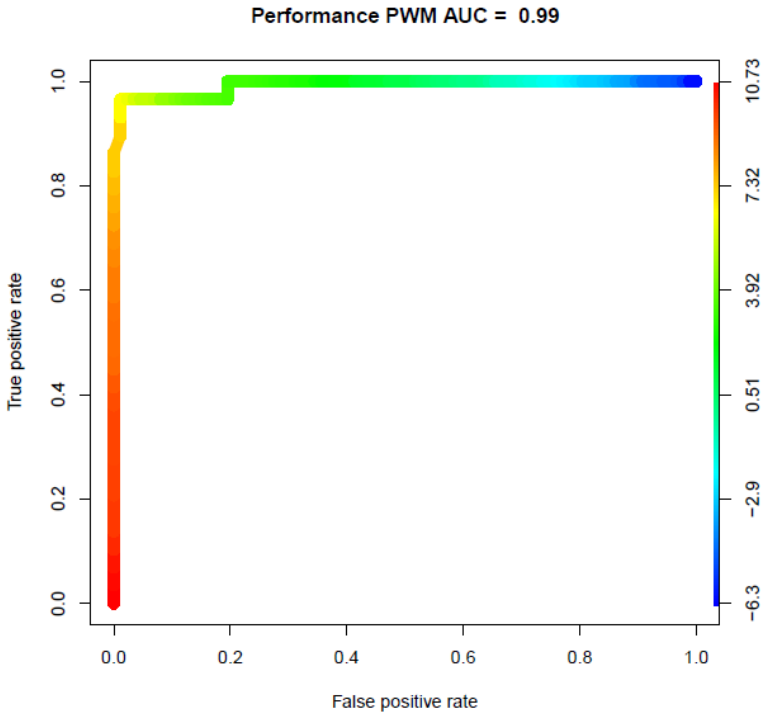

Figure 17: ROC curve for MDD for 5’ BRCA1 cryptic splice site model

Figure 17 shows the performance of MDD for BRCA1 5’ cryptic splice site model. With a threshold score of 10.7, we get a good balance of true positives and false positives. The prediction accuracy of MDD for BRCA1 5’ cryptic splice site oligomers is 0.99 with a threshold score of 10.7.

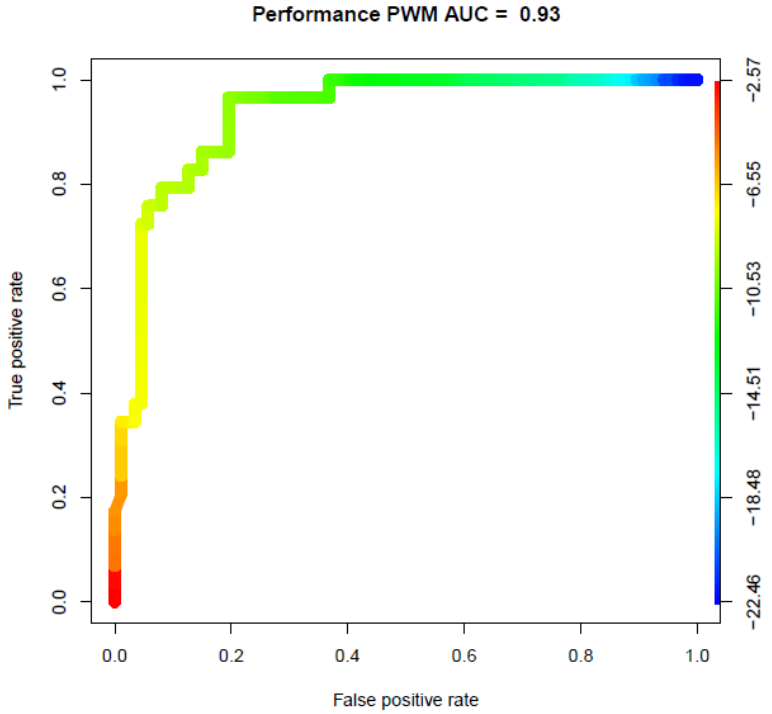

Figure 18: ROC curve for HMM for 5' BRCA1 cryptic splice site model

Figure 18 shows the performance of HMM for BRCA1 5' cryptic splice site model. To achieve the best prediction accuracy, we trained the model with initial probabilities of 0.3 and 0.7 to cryptic splice site state and non- cryptic splice site state respectively. With a threshold score of -3.1, we get a good balance of true positives and false positives. The prediction accuracy of HMM for BRCA1 5' cryptic splice site oligomers is 0.93 with a threshold score of -3.1.

As can be seen from Figures 16, 17 and 18, we can conclude that MDD performs well with the AUC of 0.99 for BRCA1 5' cryptic splice site dataset given that the threshold score is above 10.7.

6.5 BRCA1 3’ Cryptic Splice Site Dataset

Length of sites: 14

Training set containing true sites: 28

Training set containing false sites: 85

Testing set containing true sites: 28

Testing set containing false sites: 85

For BRCA1 3’cryptic splice sites, the data used for training and testing the models are the same.

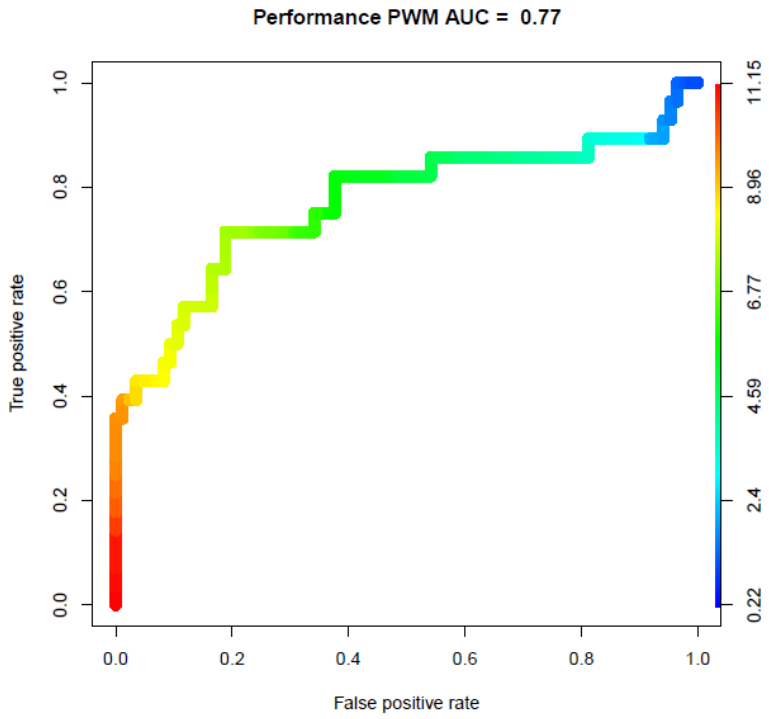

Figure 19: ROC curve for simple PWM for 3’ BRCA1 cryptic splice site model

Figure 19 shows the performance of simple PWM for BRCA1 3’ cryptic splice site model. With a threshold score of 7.5, we get high true positives and tolerable number of false

positives. The prediction accuracy of PWM for BRCA1 3' cryptic splice site oligomers is 0.77 with a threshold score of 7.5.

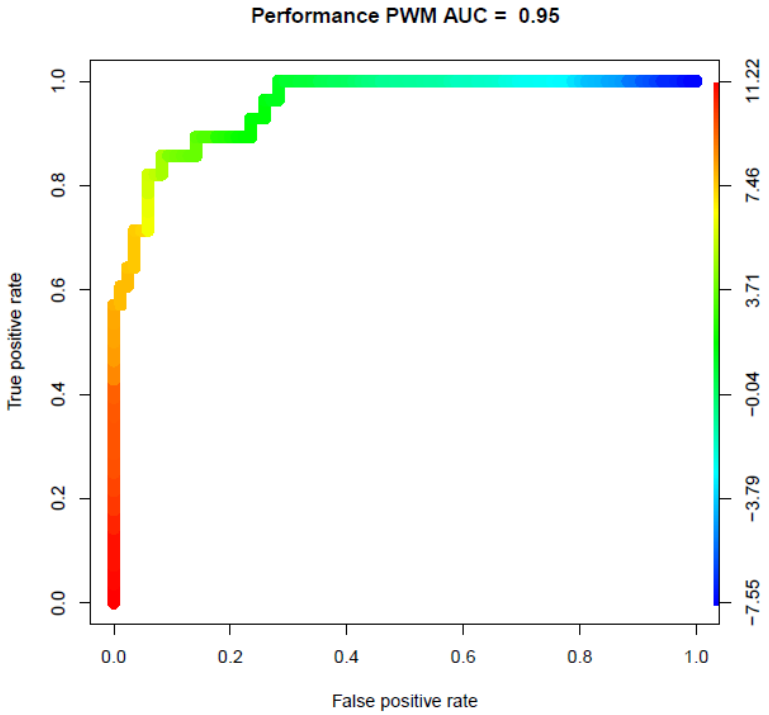

Figure 20: ROC curve for MDD for 3' BRCA1 cryptic splice site model

Figure 20 shows the performance of MDD for BRCA1 3' cryptic splice site model. With a threshold score of 10.6, we get a good balance of true positives and false positives. The prediction accuracy of MDD for BRCA1 3' cryptic splice site oligomers is 0.95 with a threshold score of 10.6.

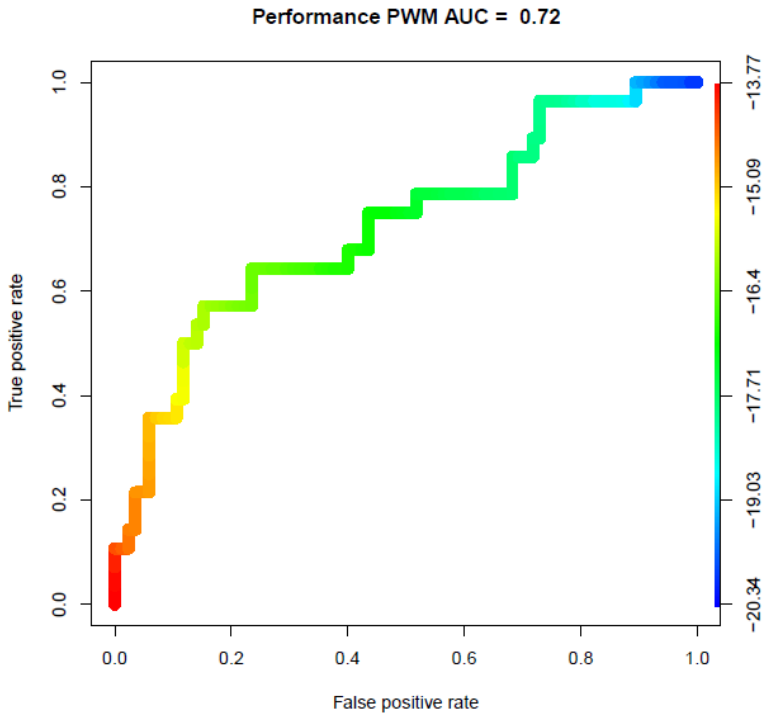

Figure 21: ROC curve for HMM for 3' BRCA1 cryptic splice site model

Figure 21 shows the performance of HMM for BRCA1 3' cryptic splice site model. To achieve the best prediction accuracy, we trained the model with initial probabilities of 0.3 and 0.7 to cryptic splice site state and non- cryptic splice site state respectively. With a threshold score of -16.1, we get a good balance of true positives and false positives. The prediction accuracy of HMM for BRCA1 3' cryptic splice oligomers is 0.72 with a threshold score of -16.1.

As can be seen from Figures 19, 20 and 21, we can conclude that MDD performs well with the AUC of 0.95 for BRCA1 3' cryptic splice site dataset given that the threshold score is above 10.6.

6.6 HBB 5' Cryptic Splice Site Dataset

Length of sites: 9

Training set containing true sites: 11

Training set containing false sites: 33

Testing set containing true sites: 11

Testing set containing false sites: 33

For HBB 5'cryptic splice sites, the data used for training and testing the models are the same.

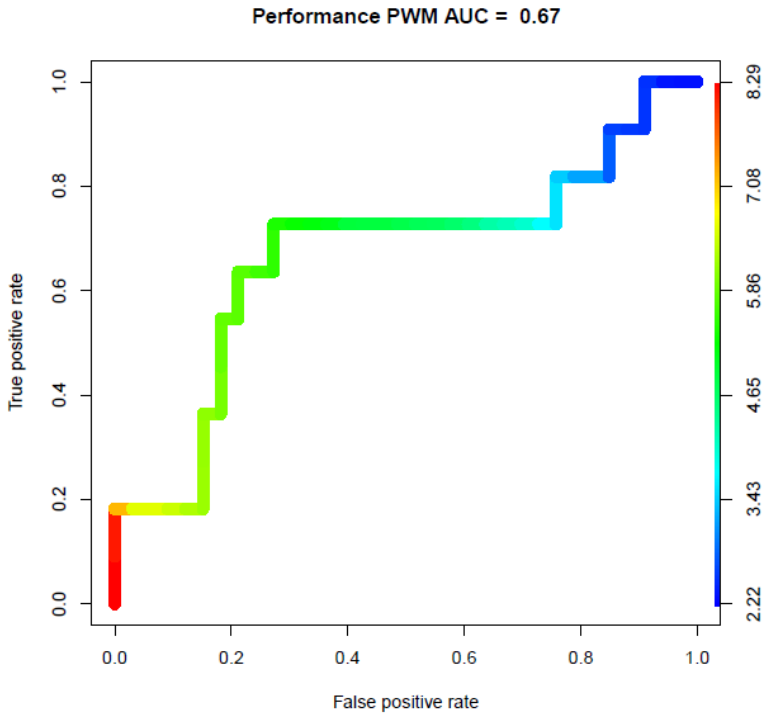

Figure 22: ROC curve for simple PWM for 5' HBB cryptic splice site model

Figure 22 shows the performance of simple PWM for HBB 5' cryptic splice site model. With a threshold score of 6.8, we get high true positives and tolerable number of false positives. The prediction accuracy of PWM for HBB 5' cryptic splice site oligomers is 0.67 with a threshold score of 6.8.

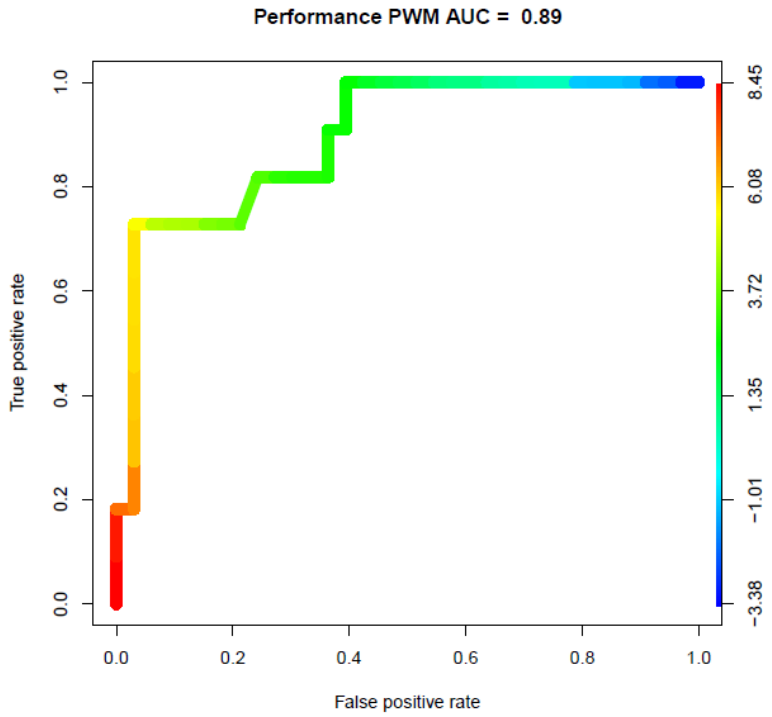

Figure 23: ROC curve for MDD for 5’ HBB cryptic splice site model

Figure 23 shows the performance of MDD for HBB 5’ cryptic splice site model. With a threshold score of 7.6, we get a good balance of true positives and false positives. The prediction accuracy of MDD for HBB 5’ cryptic splice site oligomers is 0.89 with a threshold score of 7.6.

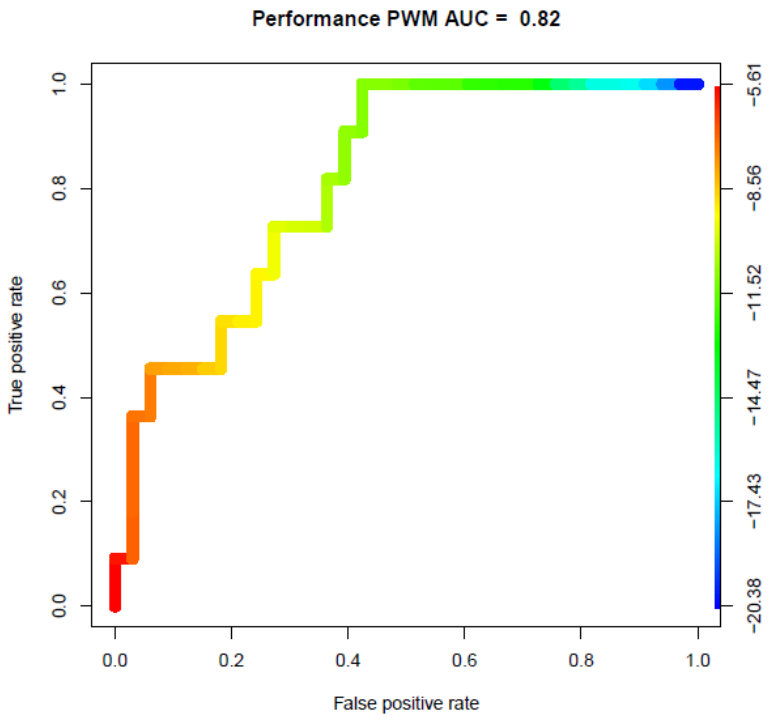

Figure 24: ROC curve for HMM for 5' HBB cryptic splice site model

Figure 24 shows the performance of HMM for HBB 5' cryptic splice site model. To achieve the best prediction accuracy, we trained the model with initial probabilities of 0.2 and 0.8 to cryptic splice site state and non- cryptic splice site state respectively. With a threshold score of -7.1, we get a good balance of true positives and false positives. The prediction accuracy of HMM for HBB 5' cryptic splice oligomers is 0.82 with a threshold score of -7.1.

As can be seen from Figures 22, 23 and 24, we can conclude that MDD performs very well with the AUC of 0.89 for HBB 5' cryptic splice site dataset given that the threshold score is above 7.6.

6.7 HBB 3' Cryptic Splice Site Dataset

Length of sites: 14

Training set containing true sites: 6

Training set containing false sites: 18

Testing set containing true sites: 6

Testing set containing false sites: 18

For HBB 3'cryptic splice sites, the data used for training and testing the models are the same.

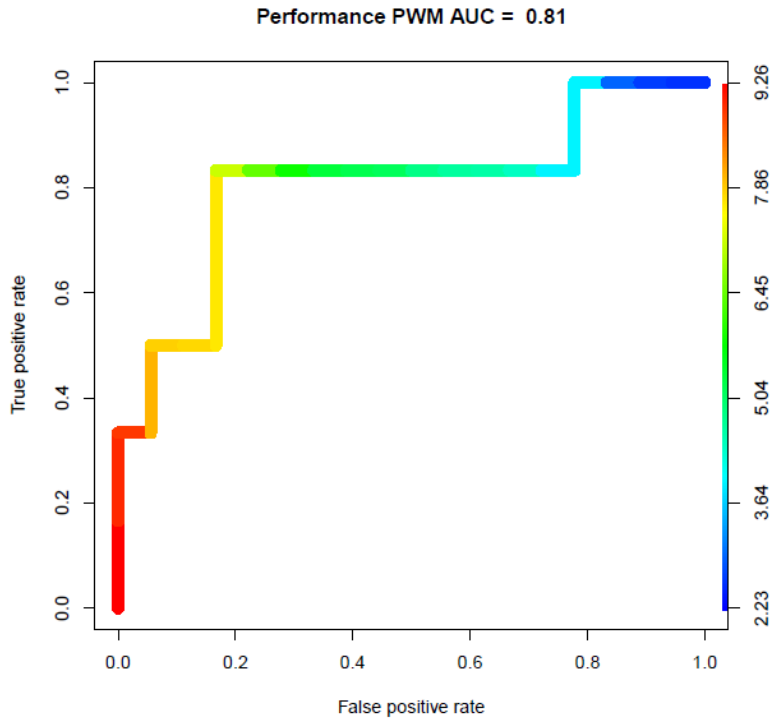

Figure 25: ROC curve for simple PWM for 3' HBB cryptic splice site model

Figure 25 shows the performance of simple PWM for HBB 3' cryptic splice site model. With a threshold score of 8.2, we get high true positives and tolerable number of false positives. The prediction accuracy of PWM for HBB 3' cryptic splice site oligomers is 0.81 with a threshold score of 8.2.

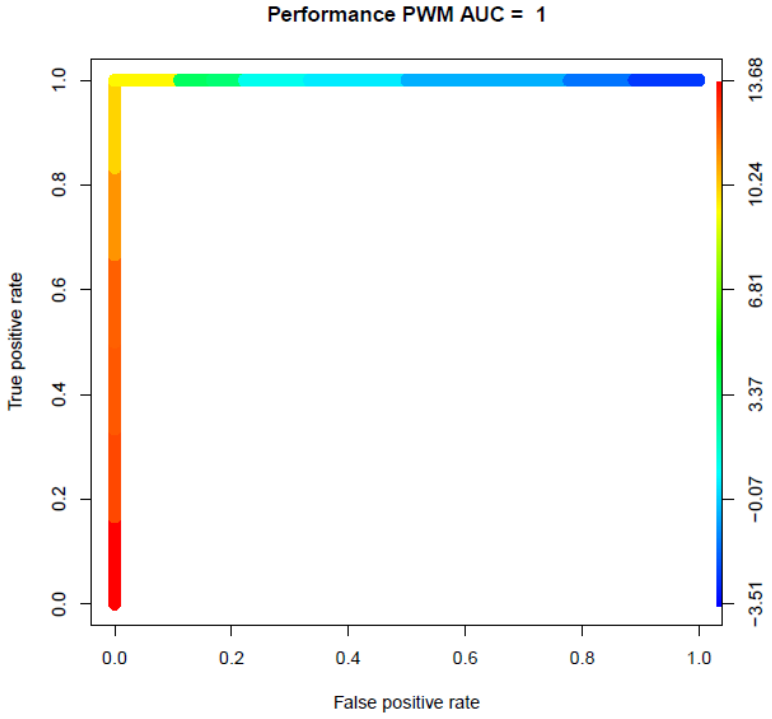

Figure 26: ROC curve for MDD for 3’ HBB cryptic splice site model

Figure 26 shows the performance of MDD for HBB 3’ cryptic splice site model. With a threshold score of 13.68, we get a good balance of true positives and false positives. The prediction accuracy of MDD for HBB 3’ cryptic splice site oligomers is 1 (perfect classification) with a threshold score of 13.68.

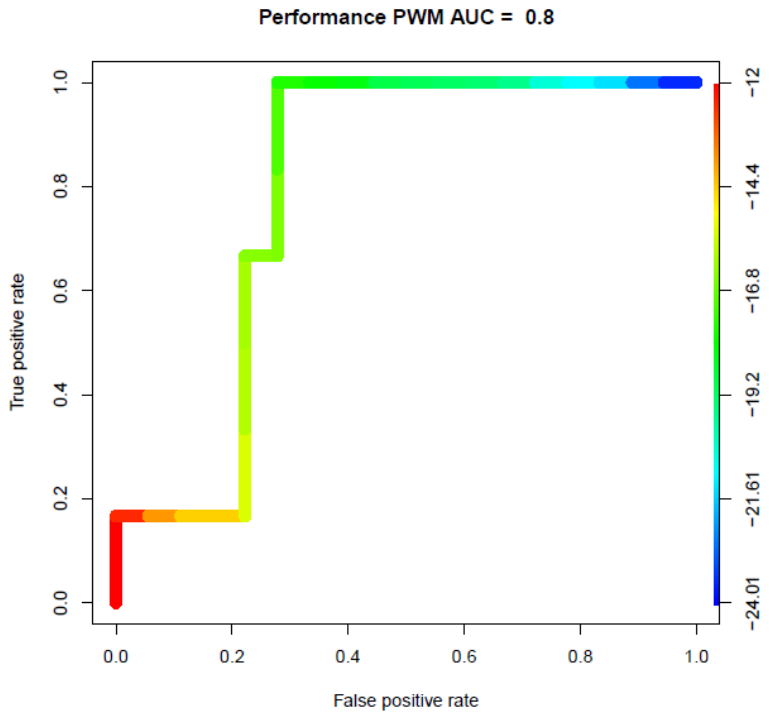

Figure 27: ROC curve for HMM for 3' HBB cryptic splice site model

Figure 24 shows the performance of HMM for HBB 3' cryptic splice site model. To achieve the best prediction accuracy, we trained the model with initial probabilities of 0.2 and 0.8 to cryptic splice site state and non- cryptic splice site state respectively. With a threshold score of -14.4, we get a good balance of true positives and false positives. The prediction accuracy of HMM for HBB 3' cryptic splice oligomers is 0.8 with a threshold score of -14.4.

As can be seen from Figures 25, 26 and 27, we can conclude that MDD performs well with the AUC of 1 for HBB 3' cryptic splice site dataset given that the threshold scores are above 13.68.

6.8 Comparative Analysis

Table 21: Performance of three models over six data sets

Method / Dataset	PWM	MDD	HMM
5’ Splice sites	0.91	**0.96**	0.93
3’ Splice sites	0.9	**0.93**	0.89
BRCA1 5’ cryptic splice sites	0.9	**0.99**	0.93
BRCA1 3’ cryptic splice sites	0.77	**0.95**	0.72
HBB 5’ cryptic splice sites	0.67	**0.89**	0.82
HBB 3’ cryptic splice sites	0.81	**1.0**	0.8

Table 21 gives a summary of the results of all the three models with six datasets. For all the datasets under consideration, MDD performs the best. We can observe that intelligent methods like MDD and HMM perform consistently better compared to simple PWM for the prediction of splice site and cryptic splice site oligomers.

In the next chapter, we conclude by discussing some of the findings in our project and the future directions.

CHAPTER 7

Conclusion and Future Work

All the methods that we analysed have their own advantages and disadvantages. The approach to be chosen depends heavily on the dataset. Splice site and cryptic splice site datasets exhibits a characteristic that shows dependence between nucleotide positions. Hence for our case, we need sophisticated models, such as MDD, that captures this property of the dataset. Also, from the results it is evident that MDD performs very well.

In this project, we also developed the HMM that represents positions in splice sites and cryptic splice sites and demonstrated that HMMs can be used to detect splice sites with good accuracy. It is very clear from the result that HMM outperforms the conventional methods, such as PWM, in most of the cases.

From our results, we can say that the MDD method can be used to successfully predict splice sites with good accuracy. However, there is a danger of overfitting for such models. Hence, it is always best to use robust and consistent models, such as HMMs, since they perform pretty well and without bias for most of the datasets.

As a future extension of this work, we might want to extend the HMM to include sequences that contain gaps. Currently, the HMM designed does not account for insertions or deletions. If we have sequences where we observe the gaps, HMM can be remodelled to accommodate insert and delete states.

REFERENCES

[1] Bortolazzo, A., Gaeke V., Khuri S. (2013) Predicting Cryptic Splice Sites in Human Genes; 2013 CSUPERB Symposium.

[2] Bortolazzo, A, Khuri N., Khuri S. (2013) Computational Modeling of Cryptic Splicing Events. Proceedings of the Great Lakes Bioinformatics Conference, Pittsburgh, PA, May 14-16, 2013.

[3] Bortolazzo, A., Khuri N., Khuri S. (2013) Profiling Cryptic Splice Sites in the Tumor-suppressor, BRCA1 gene. Proceedings of AAAS 2013 conference; pp.97.

[4] Burge C., Karlin S. (1997) Prediction of complete gene structures in human genomic DNA. Journal of Molecular Biology; 268:78-94.

[5] Pollastro P., Rampone S. (2003). HS3D: Homo Sapiens Splice Site Data Set, Nucleic Acids Research, 2003 Annual Database Issue.

[6] P. Pevzner and R. Shamir, *Bioinformatics for biologists*, first edition, Cambridge University Press, 2011.

[7] D. Poole and A. Mackworth. (n.d.). Learning Probabilities. Artificial Intelligence: Foundations of Computational Agents. Retrieved June 16, 2013, from http://artint.info/html/ArtInt_174.html.

[8] Interpreting Diagnostic Tests. Introduction to ROC Curves. Retrieved June 16, 2013, from http://gim.unmc.edu/dxtests/roc1.htm.

[9] J. Fan, S. Upadhye, and A. Worster, "Understanding receiver operating characteristic (ROC) curves," *CJEM*, vol. 8, no. 1, pp. 19-20, 2006.48

[10] Cross-validation (statistics). (n.d.). In *Wikipedia*. Retrieved October 2, 2013, from http://en.wikipedia.org/wiki/Cross-validation_(statistics).

[11] Lee, T., Lin, Z., Hsieh, S., Bretaña, N., & Lu, C. (2011). Exploiting maximal

dependence decomposition to identify conserved motifs from a group of aligned signal sequences. *Bioinformatics*, *27*(13), 1780-1787.

[12] WebLogo. Retrieved October 2, 2013, from http://weblogo.berkeley.edu/logo.cgi.

[13] Baum – Welch algorithm. (n.d). In *Wikipedia*. Retrieved February 23, 2015, from http://en.wikipedia.org/wiki/Baum-Welch_algorithm

[14] Biology/CS 123B and CS 223, Hidden Markov Model from Dr. Sami Khuri's lecture notes

[15] Lawrence R. Rabiner. "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proceedings of the IEEE* 77, no. 2 (February 1989), p. 257-86.

[16] Viterbi algorithm. (n.d). In *Wikipedia*. Retrieved March 5, 2015, from http://en.wikipedia.org/wiki/Viterbi_algorithm

[17] Statistical Genetics & Bioinformatics. Hidden Markov Models. Retrieved March 3, 2015, from ftp://statgen.ncsu.edu/pub/thorne/bioinf2/markov3.pdf

[18] Equation editor by codecogs. Retrieved March 31, 2015, from http://www.codecogs.com/latex/eqneditor.php

APPENDIX

Source Code

Formatting the input sequences if necessary (Perl)

format.pl

```
#
#
# @author: Santrupti Nerli, SJSU, January 2015
#
#
#!/usr/perl/bin -w

open (READ_FILE, "<",$ARGV[0]) or die("Couldn't open file:$!");
open (WRITE_FILE, ">",$ARGV[1]) or die("Couldn't open file:$!");

# This will convert the input sequence to all uppercase without any spaces
inbetween
sysread(READ_FILE,$_,1);
$flag = 0;
if(/>/) {
    while($_ ne "\n")
    {
        sysread(READ_FILE,$_,1);
    }
    $_ = '';
    $flag = 1;
}

$endFlag = 0;
do {
    if(/>/ && $flag == 1) {
        $endFlag = 1;
    }
    else {
        if(/[acgtACGT]/) {
            print(WRITE_FILE uc($_))
        }
        else {
            # Any character other than ACGT/acgt will be reported as a
warning to the user.
            if(/[bd-fh-su-zBD-FH-SU-Z]/) {
                print("**Warning: The input sequence contains character
$_\n");
                print(WRITE_FILE uc($_))
            }
        }
    }
}while(sysread(READ_FILE,$_,1) && !$endFlag);

close(READ_FILE) or die("Couldn't close file:$!");
close(WRITE_FILE) or die("Couldn't close file:$!");
exit;

# End of formatting
```

Extracting oligomers using sliding window (Perl)

sliding_window.pl

```
#
#
# @author: Santrupti Nerli, SJSU, January 2015
#
#

#!/usr/perl/bin -w

# Read file handles will read input sequence from sequence file.
open(READ_FILE, "<",$ARGV[0]) or die("Couldn't open file:$!");

my @arr;

# Write the extracted sequences into output file
open(WRITE_FILE, ">output.out") or die("Couldn't open file:$!");

# Read the sequence to identify the sites.
while($line = <READ_FILE>)
{
    chomp($line);
    push(@arr, split('',$line));
}

$limit = $ARGV[1];

for(my $i=0;$i<@arr-$limit+1;$i++)
{
    my @new_arr = @arr;

    # @sub_seq is the subsequence obtained by moving the window one at a
time.
    my @sub_seq = splice(@new_arr,$i,$limit);

    # for 5' splice site
    if( $limit == 9 && $arr[$i+3] eq 'G' && $arr[$i+4] eq 'T')
    {
        print WRITE_FILE @sub_seq;
        print WRITE_FILE "\n";
    }
    # for 3' splice site
    if( $limit == 14 && $arr[$i+10] eq 'A' && $arr[$i+11] eq 'G')
    {

        print WRITE_FILE @sub_seq;
        print WRITE_FILE "\n";
    }
}

close(READ_FILE) or die("Couldn't close file:$!");
close(WRITE_FILE) or die("Couldn't close file:$!");

exit;

# End of sliding window
```

Position Weight Matrix and Maximal Dependence Decomposition (C)

functions.h

```
/*
 *
 * @author: Santrupti Nerli, SJSU, March 2015
 *
 * */

/* Variables to hold alphabets {A, C, G, T} for nucleotides */
int no_groups;
char *groups;

/* condition for stopping the construction of tree*/
long long int limit;

/* laplace smooting */
long long int pseudo_n, pseudo_d;

/* degrees of freedom for chi-squared test */
static float df[2] = {45.5579,65.4750};

/* structure of a node in classification tree */
struct node
{
    struct node *left;
    struct node *right;
    char **seq;
    long long int position;
    char nucleotide;
    float **pwm;
    long long int no_ele;
    long long int *seq_no;
};
typedef struct node * NODE;

/* Definition of frees in file free.c */
void freenode(NODE set);

void free_ptr(float *row, float *N_X, float *f_X, char *consensus, long
long int *ci_row, long long int *ci_inv_row, long long int *prev, float
*total_check, long long int *stack, char *line);

void free_children(NODE cur);

/* Definition of computation in file computation.c */
float square(float x);

float compute_chi(float N,float *N_X,float *f_X);

void find_consensus(char *con, char **sequences, long long int row, long
long int column);

long long int check(float *arr, long long int row);

void create_pwm(NODE root, int no_groups);

int nucleotide(char ch);

NODE compute_weights(NODE root, int col, int no_groups);
```

```
/* Definition of tree operations in tree.c */
NODE getnode();

void display(NODE tree);

NODE split(NODE set, char nucleotide, long long int pos, long long int row,
long long int col, long long int *ci_row, long long int *ci_inv_row, char
max_nucltd, long long int max_pos, int is_root);

float traverse(NODE root, char *sequence);

void formation(NODE root, long long int n, float *row, float *N_X, float
*f_X, char *consensus, float pseudo_n, float pseudo_d,long long int
no_sequences, long long int N, long long int *ci_row, long long int
*ci_inv_row, long long int r, long long int ri,long long int *prev,float
*total_check, long long int *stack, long long int top, long long int dir,
int df_ind);
```

main.c

```
/*
 *
 * @author: Santrupti Nerli, SJSU, March 2015
 *
 * */

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include"functions.h"

/* main function */
int main(int argc, char *argv[])
{

    /* consensus: string to the store the consensus sequence.
       n        : Number of nucleotides in the consensus sequences (it is
the column).
       pos_i    : column in the chi-squared table.
       pos_j    : row in the chi-squared table.
       row      : row holds values of one row in a chi-squared table.
       max      : holds the max of total of rows of the chi-squared table.
       max_index    : index which is used to fetch the conserved
nucleotide.
       N        : Total number of sequences in ci.
       N_X      : Total number of sequences that have X in a particular
position (say j).
       f_X      : Relative frequency of nucloetide X in the sequences in
ci_inv.
       pseudo_n : pseudocount for the numerator.
       pseudo_d : pseudocount for the denominator.
       ci_row   : Number of sequences in ci set.
       ci_inv_row: Number of sequences in ci_inv set.
       no_sequences: number of sequences in the input file which has
training data.
    */
    char *consensus, *line;
    long long int *prev;
    long long int *stack, top = -1;

    float *row, N, *N_X, *f_X, *total_check;
```

```
    long long int *ci_row,*ci_inv_row;
    long long int no_sequences = 100000, n=30;
    int df_ind = -1;
    extern long long int pseudo_n, pseudo_d;
    extern int no_groups;

    char lines[n], filename[250];

    long long int r = 0,ri = 0;

    long long int i,j,k;
    extern long long int limit;
    char *method = (char*)malloc(sizeof(char)*4);

    if(argc < 4 || (!strcmp(argv[1],"-h")))
    {
        printf("Usage: <executable> <method: PWM|MDD> <groups>
<train_file_path> <test_file_path>\n");
        exit(0);
    }

    if(!strcmp(argv[1], "PWM")) {
        strcpy(method, argv[1]);
    }
    else {
        if(!strcmp(argv[1], "MDD")) {
            strcpy(method, argv[1]);
        }
        else {
            printf("Invalid method name: %s\n", argv[1]);
            exit(0);
        }
    }

    no_groups = argc-4;
    pseudo_n = 1;
    pseudo_d = no_groups;
    extern char *groups;
    groups = (char *)malloc(sizeof(char)*no_groups);
    for(i=1;i<no_groups+1;i++) {
        groups[i-1] = argv[i+1][0];
    }

    /* root: It is a root of a tree which has all the sequences from a
file.
       cur : It represents current which holds the tree while calculating
chi-squared table.
    */
    NODE root = NULL;

    /* open input file for reading the training data. */
    FILE *fp = fopen(argv[argc-2],"r");
    FILE *fptr = NULL, *fout = NULL;

    for(i=0; !feof(fp); i++)
    fscanf(fp,"%s",lines);

    no_sequences = i-1;
    limit = 0.15*no_sequences;
    fclose(fp);
    fp = fopen(argv[argc-2],"r");
```

```
    /* Get the root ready to store the training data. Read from the file
and store it in root. */
    root = getnode();
    root->seq = (char**)malloc(sizeof(char *)*no_sequences);
    root->seq_no = (long long int*)malloc(sizeof(long long
int)*no_sequences);
    if(root->seq == NULL)
        printf("Allocation failed\n");
    for(i=0; i<no_sequences; i++)
    {
        root->seq[i] = (char*)malloc(sizeof(char)*(n+1));
        if(root->seq[i] == NULL)
            printf("Allocation failed\n");
        fscanf(fp,"%s",root->seq[i]);
    root->seq_no[i] = i;
    }

    if(!strcmp(method, "PWM")) {
        root->position = -1;
        root->nucleotide = '\0';
    }

    root->left = NULL;
    root->right = NULL;

    /* Now, i has the actual number of sequences which is stored in
appropriate variable no_sequences. */

    root->no_ele = no_sequences;
    printf("no_sequences=%lld\n",no_sequences);


    n = strlen(root->seq[0]);
    consensus = (char*)malloc(sizeof(char)*(n+1));

    prev = (long long int *)malloc(sizeof(long long int)*n);
    stack = (long long int *)malloc(sizeof(long long int)*100);

    /* Allocate memory for row, N_X and f_X. */
    row = (float*)malloc(sizeof(float)*n);
    N_X = (float*)malloc(sizeof(float)*no_groups);
    f_X = (float*)malloc(sizeof(float)*no_groups);
    total_check = (float*)malloc(sizeof(float)*n);

    if(n == 9)
    df_ind = 0;
    else
    {
    if(n == 14)
        df_ind = 1;
    }

    for(i=0; i<n; i++)
    {
        prev[i] = -1;
        total_check[i] = 0.0;
    }

    /* Allocate memories to ci_row and ci_inv_row. */
    ci_row = (long long int*)malloc(sizeof(long long int));
    ci_inv_row = (long long int*)malloc(sizeof(long long int));
```

```
    if(!strcmp(method, "MDD")) {
        formation(root, n, row, N_X, f_X, consensus, pseudo_n, pseudo_d,
no_sequences, N, ci_row, ci_inv_row, r, ri, prev, total_check, stack, top,
0, df_ind);
    }

    create_pwm(root, no_groups);
    display(root);

    fptr = fopen(argv[argc-1],"r");
    fout = fopen("output","a");
    line = (char*)malloc(sizeof(char)*(n+1));

    while(!feof(fptr))
    {
        if(fscanf(fptr,"%s",line) != 1) break;
        fprintf(fout,"%s\t%f\n",line,traverse(root,line));
    }

    fclose(fp);
    fclose(fptr);
    fclose(fout);

    free_ptr(row, N_X, f_X, consensus, ci_row, ci_inv_row, prev,
total_check, stack, line);
    freenode(root);

    return 0;
}
```

tree.c

```
/*
 *
 * @author: Santrupti Nerli, SJSU, March 2015
 *
 * */

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include"functions.h"

/* getnode() is a function which allocates memory for a new node. */
NODE getnode()
{
    NODE temp;
    temp=(NODE)malloc(sizeof(struct node));
    if(temp==NULL)
    {
        printf("Memory allocation failed");
        return NULL;
    }
    return temp;
}

void print(NODE tree, FILE* fp, int noSeq) {
    int i = 0;
    for(i = 0; i < noSeq; i++) {
    fprintf(fp, "%s\n", tree->seq[i]);
    }
```

```
}

void display(NODE tree)
{
    int i, j, k, col;
    if(tree == NULL)
        return;
    FILE *fp = fopen("PWM1.txt", "a");
    if(tree->nucleotide != '\0' && tree->left != NULL && tree->right !=
NULL)
    {
        fprintf(fp, "\n%lld|%c|%lld\nPWM\n-------\n",tree->position+1,tree-
>nucleotide,tree->no_ele);
    for(i=0;i<no_groups;i++)
    {
        fprintf(fp, "%f\n",tree->pwm[i][0]);
    }
    //print(tree, fp, tree->no_ele);
    }
    else
    {
    if(tree->nucleotide != '\0')
    {
        fprintf(fp, "Node without one child\n");
        fprintf(fp, "\n%lld|%c|%lld\nPWM\n-------\n",tree->position+1,tree-
>nucleotide,tree->no_ele);
    }
    else
        fprintf(fp, "Leaf|%lld\n",tree->no_ele);
    col = strlen(tree->seq[0]);
    for(i=0;i<no_groups;i++)
    {
        for(j=0;j<col;j++)
            fprintf(fp, "%f\t",tree->pwm[i][j]);
        fprintf(fp, "\n");
    }
    }
    fclose(fp);
    display(tree->left);
    display(tree->right);
}

/* Give only one set to split that is only one node. Based on the
nucleotide at that position,
   sequences are split into two groups which are the children of set.

   Parameters:      set: It is a root for which we need to find children.
            nucleotide and pos: based on nucleotide and position, splitting
happens.
            row and col: rows and columns of set(i.e, no. of sequences and
no. of nucleotides in each sequence).
            ci_row and ci_inv_row: attributes which hold the count of
sequences in ci and ci_inv.
*/
NODE split(NODE set, char nucleotide, long long int pos, long long int row,
long long int col, long long int *ci_row, long long int *ci_inv_row, char
max_nucltd, long long int max_pos, int is_root)
{
    long long int i,j,k;
    /* lchild and rchild are the children of set that are created by split.
*/
    NODE lchild = NULL,rchild = NULL;
```

```
    long long int lflag = 0,rflag = 0;

    if(set==NULL)return;

    lchild = getnode();
    rchild = getnode();

    lchild->seq = (char**)malloc(sizeof(char*)*row);
    rchild->seq = (char**)malloc(sizeof(char*)*row);

    lchild->seq_no = (long long int*)malloc(sizeof(long long int)*row);
    rchild->seq_no = (long long int*)malloc(sizeof(long long int)*row);

    /* If the nucleotide matches the position for a particular sequence,
add it to left child else to the right child. */
    j = 0;
    k = 0;
    for(i=0; i<row; i++)
    {
        if(set->seq[i][pos] == nucleotide)
        {
            lflag = 1;
            lchild->seq[j] = (char*)malloc(col*sizeof(char));
            strcpy(lchild->seq[j], set->seq[i]);
        lchild->seq_no[j] = set->seq_no[i];
            j++;

        }
        else
        {
            rflag = 1;
            rchild->seq[k] = (char*)malloc(col*sizeof(char));
            strcpy(rchild->seq[k], set->seq[i]);
        rchild->seq_no[k] = set->seq_no[i];
            k++;
        }
    }

    lchild->right = NULL;
    lchild->left = NULL;

    rchild->right = NULL;
    rchild->left = NULL;

    if(lflag == 0)
    {
        free(lchild->seq);
        lchild->seq = NULL;
    free(lchild);
    lchild = NULL;
    }
    if(rflag == 0)
    {
        free(rchild->seq);
        rchild->seq = NULL;
    free(rchild);
    rchild = NULL;
    }

    if(is_root)
    {
    if((j != 0 && j <= limit) || (k != 0 && k <= limit))
```

```
        return set;
    }

    /* Attach the children to the parent and return the parent to the
calling function. */
    set->left = lchild;
    set->right = rchild;

    set->nucleotide = max_nucltd;
    set->position = max_pos;

    /* Imp: ci_row and ci_inv_row are pointers because we need the no. of
sequences in each
       of ci and ci_inv and the only way to get it is through this
function. It is because,
       this is the function that creates these sets. */

    if(lchild != NULL)
    {
        set->left->no_ele = j;
    set->left->nucleotide = '\0';
    set->left->position = -1;
    }
    if(rchild != NULL)
    {
    set->right->no_ele = k;
    set->right->nucleotide = '\0';
    set->right->position = -1;
    }
    *ci_row = j;
    *ci_inv_row = k;

    return set;
}

/* Rewrite traverse so that it scores the sequence */
float traverse(NODE root, char *sequence)
{
    float score = 0.0;
    int nuc = -1, i, j;
    char *temp_seq;
    if(root == NULL)
        return 0;

    temp_seq = (char *)malloc(sizeof(char)*(strlen(sequence)+1));
    strcpy(temp_seq, sequence);

    while(root != NULL)
    {
    for(j=0;j<no_groups;j++)
        if(root->nucleotide == groups[j])
            nuc = j;

        if(root->position != -1 && root->nucleotide == sequence[root-
>position])
    {
        if(root->left != NULL && root->right != NULL)
            score = score + root->pwm[nuc][0];
        else
        score = score + root->pwm[nuc][root->position];
        sequence[root->position] = 'X';
        root = root->left;
```

```
    }
        else
    {
        if(root->right == NULL || root->nucleotide == '\0')
        {
            for(i=0;i<strlen(sequence);i++)
            {
                if(sequence[i] == 'X')
                {
                    //Do nothing
                }
                else
                {
                    for(j=0;j<no_groups;j++)
                        if(sequence[i] == groups[j])
                            nuc = j;
                    score = score + root->pwm[nuc][i];
                }
            }
        }
        root = root->right;
    }
    }
    strcpy(sequence, temp_seq);
    free(temp_seq);
    return score;
}

/* formation() function will build a tree. */
void formation(NODE root, long long int n, float *row, float *N_X, float
*f_X, char *consensus, float pseudo_n, float pseudo_d,long long int
no_sequences, long long int N, long long int *ci_row, long long int
*ci_inv_row, long long int r, long long int ri,long long int *prev,float
*total_check, long long int *stack, long long int top, long long int dir,
int df_ind)
{
    long long int pos_i = 0,pos_j = 0, max_index = 0;
    long long int i,j,k,l;
    float max = -1, total;
    long long int flag = 0;
    NODE cur = NULL;

    if(root == NULL || root->seq == NULL)
        return;
    if(root->no_ele <= limit)
    {
    prev[stack[top--]] = -1;
    return;
    }
    find_consensus(consensus, root->seq, no_sequences, n);

    /* Loop through till you fill the entire chi-squared table. */
    for(pos_i=0; pos_i<n; pos_i++)
    {
    cur = root;
        /* based on consensus nucleotides, split each time for each row and
fill the table. */
        cur =
split(root,consensus[pos_i],pos_i,no_sequences,(n+1),ci_row,ci_inv_row,'\0'
,0,0);

        /* Add the pseudocount (pseudo_n) to N. */
```

```
        N = *ci_row + pseudo_d;

        /* This loop will calculate the row of the table one by one. */
        for(pos_j=0; pos_j<n; pos_j++)
        {
            if(pos_i == pos_j)
            {
                row[pos_j] = 0.0;
            }
            else
            {
                for(i=0; i<no_groups; i++)
                {
                    N_X[i] = 0;
                    f_X[i] = 0;
                }
                /* To find N_X. */
                for(i=0; i<*ci_row; i++)
                {
            for(l=0;l<no_groups;l++)
            if(cur->left->seq[i][pos_j] == groups[l])
                N_X[l]++;
                }
            for(l=0;l<no_groups;l++)
                    N_X[l] += pseudo_n;

                /* To find f_X. */
                for(i=0; i<*ci_inv_row; i++)
                {
            for(l=0;l<no_groups;l++)
            if(cur->right->seq[i][pos_j] == groups[l])
                f_X[l]++;
                }
            for(l=0;l<no_groups;l++)
                    f_X[l] = (f_X[l] + pseudo_n)/(*ci_inv_row + pseudo_d);

                row[pos_j] = compute_chi(N,N_X,f_X);
            }
        }
        total = 0.0;
        for(i=0; i<n; i++)
        {
            total += row[i];
        }
        if(max - total < 0.0)
        {
            if(prev[pos_i] != 1)
            {
                max = total;
                max_index = pos_i;
            }
        }
        free_children(cur);
    cur = NULL;
    }
    root =
split(root,consensus[max_index],max_index,no_sequences,(n+1),ci_row,ci_inv_
row,consensus[max_index],max_index,1);
    r = *ci_row;
    ri = *ci_inv_row;

    if((r <= limit && r != 0) || (ri <= limit && ri != 0))
```

```
    return;

    if(dir == 0)
    {
        stack[++top] = max_index;
    }
    total_check[max_index] = max;
    prev[max_index] = 1;

    if(r <= limit && ri <= limit && (max - df[df_ind]) < 0.0 )
    {
    prev[stack[top--]] = -1;
        return;
    }
    else
    {
        no_sequences = *ci_row;

        formation(root->left, n, row, N_X, f_X, consensus, pseudo_n,
pseudo_d, no_sequences, N, ci_row, ci_inv_row, r, ri, prev,total_check,
stack, top, 0, df_ind);
        *ci_row = r;
        *ci_inv_row = ri;
    }

    if(r <= limit && ri <= limit && (max - df[df_ind]) < 0.0 )
        return;
    else
    {
        no_sequences = *ci_inv_row;

        formation(root->right, n, row, N_X, f_X, consensus, pseudo_n,
pseudo_d, no_sequences, N, ci_row, ci_inv_row, r, ri, prev,total_check,
stack, top, 1, df_ind);
        *ci_row = r;
        *ci_inv_row = ri;
    }
}
```

computation.c

```
/*
 *
 * @author: Santrupti Nerli, SJSU, March 2015
 *
 * */

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include"functions.h"

/* compares floating point values */
long long int check(float *arr, long long int row)
{
    while(--row>0 && abs(arr[row]-arr[0]) > 0.00001);
    return row!=0;
}

/* square() is a function which squares the given number and returns the
result. */
```

```
float square(float x)
{
    return (x*x);
}

/*
   compute_chi() is a function which will compute the chi value as per the
formula
   chi = Summation(square(observed-expected)/expected)
*/
float compute_chi(float N,float *N_X,float *f_X)
{
    float res = 0;
    long long int i;
    for(i=0; i<no_groups; i++)
    {
        res += (square((N*f_X[i])-N_X[i]))/(N*f_X[i]);
    }
    return res;
}

/* find_consensus() function will determine the consensus sequence given a
set of sequences. */
void find_consensus(char *con, char **sequences, long long int row, long
long int column)
{
    /* counter : Keeps the count of each nucleotide ACGT in a column in
that order.
       max     : Keeps the maximum count of the nucleotide in a particular
column.
       index   : stores the index of the consensus nucleotide.
    */
    long long int counter[no_groups];
    long long int i,j,k,max = 0,index = 0, l;
    if(sequences == NULL)
    {
        con = NULL;
        return;
    }

    /* Count the ACGT occurance in each column. */
    for(i=0; i<column; i++)
    {
    for(l=0;l<no_groups;l++)
            counter[l] = 0;
        for(j=0; j<row; j++)
        {
        for(l=0;l<no_groups;l++)
                if(sequences[j][i] == groups[l])
                    counter[l]++;
        }
        /* Find the consensus nucleotide based on the count. */
        max = 0;
        index = 0;
        for(k=0; k<no_groups; k++)
        {
            if(max < counter[k])
            {
                max = counter[k];
                index = k;
            }
        }
```

```
    con[i] = groups[index];
    }
    con[i] = '\0';
}

/* constructs a PWM for a given node */
NODE compute_weights(NODE root, int col, int no_groups)
{
    extern long long int pseudo_n, pseudo_d;
    float exp, denominator;
    float group_count[no_groups];
    long long int i, j, temp, k, l;
    char consensus = '\0';
        if(no_groups != 0)
            exp = 1.0/no_groups;

    root->pwm = (float**)malloc(sizeof(float*) * no_groups);
    for(i=0;i<no_groups;i++)
    {
        root->pwm[i] = (float *) malloc(sizeof(float) * col);
    }

    for(j=0;j<col;j++)
    {
        for(k=0;k<no_groups;k++)
            group_count[k] = 0.0;

        temp = j;
        for(i=0;i<root->no_ele;i++)
        {
            if(root->position != -1 && root->left != NULL && root->right !=
NULL)
            {
                j = root->position;
            }
            consensus = root->seq[i][j];
            for(l=0;l<no_groups;l++)
                if(consensus == groups[l])
                    group_count[l]++;
        }
        denominator = root->no_ele+pseudo_d;

        j = temp;
        for(l=0;l<no_groups;l++)
            root->pwm[l][j] =
(log(((group_count[l]+pseudo_n)/(denominator))/exp))/log(2);
    }
    return root;
}

/* Traverses tree and calls compute_weights to construct PWM for
    leaf or parent without a child */
void create_pwm(NODE root, int no_groups)
{
    int col = 0;
    if(root == NULL)
        return;
    if(root->position != -1 && root->left != NULL && root->right != NULL)
        col = 1;
    else
        col = strlen(root->seq[0]);
    compute_weights(root, col, no_groups);
```

```
    create_pwm(root->left, no_groups);
    create_pwm(root->right, no_groups);
}
```

free.c

```
/*
 *
 * @author: Santrupti Nerli, SJSU, March 2015
 *
 * */

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#include"functions.h"

/* freenode() will free the memory of a node. */
void freenode(NODE set)
{
    long long int i;

    if(set == NULL) return;
    freenode(set->left);
    freenode(set->right);
    for(i=0; i<set->no_ele; i++)
    {
        free(set->seq[i]);
    set->seq[i] = NULL;
    }
    for(i=0;i<4;i++)
    {
        free(set->pwm[i]);
    set->pwm[i] = NULL;
    }
    free(set->pwm);
    set->pwm = NULL;
    free(set->seq);
    set->seq = NULL;
    free(set);
    set = NULL;
}

/* Traverse through the tree and keep freeing the nodes */
void free_children(NODE cur)
{
    long long int i;
    if(cur->left != NULL)
    {
        if(cur->left->seq != NULL)
            {
                    for(i=0; i<cur->left->no_ele; i++)
                    {
                        if(cur->left->seq[i] != NULL)
                                free(cur->left->seq[i]);
                    }
                    if(cur->left->seq != NULL)
                        free(cur->left->seq);
            }
        if(cur->left->seq_no != NULL)
        {
```

```
            free(cur->left->seq_no);
            cur->left->seq_no = NULL;
        }
            if(cur->left != NULL)
        {
                    free(cur->left);
            cur->left = NULL;
        }
    }
    if(cur->right != NULL)
    {
        if(cur->right->seq != NULL)
            {
                    for(i=0; i<cur->right->no_ele; i++)
                    {
                        if(cur->right->seq[i] != NULL)
                                free(cur->right->seq[i]);
                    }
                    if(cur->right->seq != NULL)
                        free(cur->right->seq);
            }
        if(cur->right->seq_no != NULL)
        {
            free(cur->right->seq_no);
            cur->right->seq_no = NULL;
        }
            if(cur->right != NULL)
        {
                    free(cur->right);
            cur->right = NULL;
        }
    }
}


/* free_ptr() frees all the pointers. */
void free_ptr(float *row, float *N_X, float *f_X, char *consensus, long
long int *ci_row, long long int *ci_inv_row, long long int *prev, float
*total_check, long long int *stack, char *line)
{
    if(N_X != NULL)
    {
        free(N_X);
        N_X = NULL;
    }
    if(f_X != NULL)
    {
        free(f_X);
        f_X = NULL;
    }
    if(consensus != NULL)
    {
        free(consensus);
        consensus = NULL;
    }
    if(ci_row != NULL)
    {
        free(ci_row);
        ci_row = NULL;
    }
    if(ci_inv_row != NULL)
    {
```

```
        free(ci_inv_row);
        ci_inv_row = NULL;
    }
    if(prev != NULL)
    {
        free(prev);
        prev = NULL;
    }
    if(total_check != NULL)
    {
        free(total_check);
        total_check = NULL;
    }
    if(line != NULL)
    {
        free(line);
        line = NULL;
    }
    if(stack != NULL)
    {
        free(stack);
        stack = NULL;
    }
    if(row != NULL)
    {
        free(row);
        row = NULL;
    }
}
```

makefile

```
#
#
# @author: Santrupti Nerli, SJSU, March 2015
#
#

.phony:clean compile link debug run leakcheck all

TRAIN = "F:/SJSU/Fall_2014/CS_280/MDD/testFiles/EI_train.txt"
TEST = "F:/SJSU/Fall_2014/CS_280/MDD/testFiles/EI_positives.txt"
METHOD = "MDD"
clean:
    if [ -a *_pwm ]; \
    then \
        rm *_pwm; \
    fi;
    rm *.o *.out output
compile:
    gcc -c main.c free.c tree.c computation.c
link:
    gcc main.o free.o tree.o computation.o -lm
run:
    a.exe $(METHOD) A C G T $(TRAIN) $(TEST)
debug:
    gdb a.exe
leakcheck:
    valgrind --tool=memcheck --leak-check=yes --show-reachable=yes --track-
origins=yes a.exe
all:
    make compile link run
```

Hidden Markov Model (Java)

HMMEntry.java

```
/*
 *
 * @author: Santrupti Nerli, SJSU, March 2015
 *
 * */

package hmm_main;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class HMMEntry {

    // members to store sequences, their count along with
    // various other parameters
    public String[] ssSeq;
    public String[] nssSeq;
    public String method;
    private static int noSsSeq;
    private static int noNssSeq;
    private static int states;
    private static int vocabSize;
    public static int[] codeSeq = {0, 1, 2, 3};
    public static int T;
    public static final int MAX_ITERATIONS = 20;


    // HMM Parameters
    private double[][] emissionProb;
    private double[][] transitionProb;
    private double[] initialProb;

    // Old HMM Parameters (required to check for convergence)
    private double[][] emissionProbOld;
    private double[][] transitionProbOld;

    //Keep track of convergence
    double transitionDiff;
    double emissionDiff;

    // For updating multiple sequence observations
    public double[][] emissionProbNum;
    public double[][] emissionProbDen;
    public double[][] transitionProbNum;
    public double[][] transitionProbDen;

    // constructor
    public HMMEntry() {
        ssSeq = null;
        method = "HMM";
        noSsSeq = 0;
        noNssSeq = 0;
        vocabSize = 4;
        transitionDiff = 0.0;
        emissionDiff = 0.0;
```

```
    }

    // open the training file to count the length of each sequence and
total number of sequences
    public void setSsNssSeqNo(String ssFile, String nssFile) throws
FileNotFoundException, IOException {

        BufferedReader brSs = new BufferedReader(new FileReader(ssFile));
        BufferedReader brNss = new BufferedReader(new FileReader(nssFile));

        try{
            String line = brSs.readLine();
            int ctr = 0;

            T = line.length();

            while(line != null) {
                if(line.charAt(0) != '>') {
                    ctr++;
                }
                line = brSs.readLine();
            }

            noSsSeq = ctr;

            ctr = 0;
            line = brNss.readLine();

            while(line != null) {
                if(line.charAt(0) != '>') {
                    ctr++;
                }
                line = brNss.readLine();
            }

            noNssSeq = ctr;

        } catch(Exception e){
            System.out.println("setSsSeqNo(): " + e);
        } finally {
            brSs.close();
            brNss.close();
        }
    }

    // Read the splice site set into ssSeq and non-splice site set into
nssSeq
    public void ReadSsNssFile(String ssFile, String nssFile) throws
FileNotFoundException, IOException  {


        BufferedReader brSs = new BufferedReader(new FileReader(ssFile));
        BufferedReader brNss = new BufferedReader(new FileReader(nssFile));

        try{
            String line;
            int ctr = 0;

            line = brSs.readLine();

            setStates(line.length() * 2);
            System.out.println("No. of states: " + states);
```

```
            ssSeq = new String[noSsSeq];

            while(line != null) {
                ssSeq[ctr] = new String(line.toUpperCase());
                ctr++;
                line = brSs.readLine();
            }

            line = brNss.readLine();
            ctr = 0;
            nssSeq = new String[noNssSeq];

            while(line != null) {
                nssSeq[ctr] = new String(line.toUpperCase());
                ctr++;
                line = brNss.readLine();
            }

        } catch(Exception e){
            System.out.println("ReadSsNssFile(): " + e);
        } finally {
            brSs.close();
            brNss.close();
        }
    }

    // parse cli arguments, fetch the file names to look for splice site
and non splice site training sequences
    public void parseCliArgs(String[] args) {
        try {
            if(args.length < 4 || args[0].equalsIgnoreCase("-h")) {
                System.out.println("Usage: java <class_name>
<splice_site_train_file_path> <non_splice_site_train_file_path>
<test_file_path> <output_file_path>");
                System.exit(0);
            }
            String ssFilePath = args[0];
            String nssFilePath = args[1];

            System.out.println("Splice Site File - " + ssFilePath);
            System.out.println("Non-splice Site File - " + nssFilePath);

            this.setSsNssSeqNo(ssFilePath, nssFilePath);
            this.ReadSsNssFile(ssFilePath, nssFilePath);

            // these are variables required to compute probabilities for
multiple observation sequences
            emissionProbNum = new double[getVocabSize()][getStates()];
            emissionProbDen = new double[getVocabSize()][getStates()];
            transitionProbNum = new double[getStates()][getStates()];
            transitionProbDen = new double[getStates()][getStates()];

            emissionProbOld = new
double[HMMEntry.getVocabSize()][HMMEntry.getStates()];
            transitionProbOld = new
double[HMMEntry.getStates()][HMMEntry.getStates()];

        } catch (Exception e) {
            System.out.println("parseArgs(): " + e);
        }
    }
```

```
    // print sequences
    public void printSeq() {
        for(int i = 0; i < ssSeq.length; i++) {
            System.out.println(ssSeq[i]);
        }

        System.out.println("Splice sites end.");
    }

    // setters
    public void setEmissionProb(double[][] matrix) {
        emissionProb = matrix;
    }

    public void setTransitionProb(double[][] matrix) {
        transitionProb = matrix;
    }

    public void setIntialProb(double[] matrix) {
        initialProb = matrix;
    }

    public void setStates(int s) {
        states = s;
    }

    public void setHMMParameters(double[][] eP, double[][] tP, double[] iP)
{
        this.emissionProb = eP;
        this.transitionProb = tP;
        this.initialProb = iP;
    }

    // getters
    public double[][] getEmissionProb() {
        return emissionProb;
    }

    // getter to fetch total number of splice site and non-splice site
sequences
    public int getTotalSeq() {
        return (noSsSeq + + noNssSeq);
    }

    public double[][] getTransitionProb() {
        return transitionProb;
    }

    public double[] getIntialProb() {
        return initialProb;
    }

    public static int getStates() {
        return states;
    }

    public static int getVocabSize() {
        return vocabSize;
    }

    public static int getNoSeq() {
```

```
        return noSsSeq;
    }

    // nucleotides are coded to integers {A, G, C, T} = {0, 1, 2, 3}
    public static int[] computeCode(String seq) {

        int[] code = new int[seq.length()];

        for(int i = 0; i < seq.length(); i++) {
            switch(seq.charAt(i)) {
            case 'A': code[i] = 0; break;
            case 'C': code[i] = 1; break;
            case 'G': code[i] = 2; break;
            case 'T': code[i] = 3; break;
            }
        }
        return code;
    }

    // storing current HMM parameters before the next iteration starts
    public void storeCurrentValues() {

        for(int i = 0; i < getStates(); i++) {
            for(int j = 0; j < getStates(); j++) {
                transitionProbOld[i][j] = transitionProb[i][j];
            }
        }

        for(int i = 0; i < HMMEntry.getVocabSize(); i++) {
            for(int j = 0; j < getStates(); j++) {
                emissionProbOld[i][j] = emissionProb[i][j];
            }
        }
    }

    // check if convergence has reached
    public boolean checkConvergence(double[][] tP, double[][] eP, int
iteration) {
        double transitionDiff = 0.0;
        double emissionDiff = 0.0;
        for(int i = 0; i < getStates(); i++) {
            for(int j = 0; j < getStates(); j++) {
                transitionDiff += Math.abs(tP[i][j]-
transitionProbOld[i][j]);
            }
        }

        for(int i = 0; i < HMMEntry.getVocabSize(); i++) {
            for(int j = 0; j < getStates(); j++) {
                emissionDiff += Math.abs(eP[i][j]-emissionProbOld[i][j]);
            }
        }

        if(iteration != 0 && (transitionDiff > this.transitionDiff ||
emissionDiff > this.emissionDiff)) {
            return true;
        }

        this.transitionDiff = transitionDiff;
        this.emissionDiff = emissionDiff;

        return false;
```

```
    }

    // print probability tables
    public static void printMatrix(double[][] matrix, int row, int col) {
        for(int i = 1; i <= col; i++) {
            System.out.print("\t" + i + "\t");
        }
        System.out.println();

        for(int i = 0; i < row; i++) {
            System.out.print(i+1 + "\t");
            for(int j = 0; j < col; j++) {
                System.out.printf("%f\t", matrix[i][j]);
            }
            System.out.println();
        }
    }

    // print probability tables of row x 1
    public static void printArray(double[] matrix, int row) {
        for(int i = 0; i < row; i++) {
                System.out.printf("%f\t", matrix[i]);
        }
    }

    // main method
    public static void main(String[] args) throws IOException {

        HMMEntry hmmObject = new HMMEntry();
        hmmObject.parseCliArgs(args);

        TrainHMM train = new TrainHMM(0, args[2], args[3]);
        train.startTraining(hmmObject);
    }

}
```

InitialSetting.java

```
/*
 *
 * @author: Santrupti Nerli, SJSU, March 2015
 *
 * */

package initializeHMM;

import hmm_main.HMMEntry;

public class InitialSetting {

    // Laplace smoothing
    public static int pseudocountN;
    public static int pseudocountD;

    // constructor
    public InitialSetting() {
        pseudocountN = 1;
        pseudocountD = 4;
    }

    // computes emission probabilities for splice sites and non-splice
sites
```

```
    public double[][] formEmissionProbTable(String[] Seq) {

        double[][] emissionProb = new
double[HMMEntry.getVocabSize()][Seq[0].length()];

        try {
            for(int k = 0; k < Seq[0].length(); k++) {
                int ctrA = 0, ctrC = 0, ctrG = 0, ctrT = 0;
                for(int j = 0; j < Seq.length; j++) {
                    switch(Seq[j].charAt(k)) {
                    case 'A': ctrA++; break;
                    case 'C': ctrC++; break;
                    case 'G': ctrG++; break;
                    case 'T': ctrT++; break;
                    }
                }

                emissionProb[0][k] = Math.log((double)(ctrA +
pseudocountN)) - Math.log((Seq.length + pseudocountD));
                emissionProb[1][k] = Math.log((double)(ctrC +
pseudocountN)) - Math.log((Seq.length + pseudocountD));
                emissionProb[2][k] = Math.log((double)(ctrG +
pseudocountN)) - Math.log((Seq.length + pseudocountD));
                emissionProb[3][k] = Math.log((double)(ctrT +
pseudocountN)) - Math.log((Seq.length + pseudocountD));
            }

        } catch(Exception e) {
            System.out.println("formEmissionProbTable(): " + e);
        }

        return emissionProb;
    }

    // combines both splice site and non-splice site emission probabilities
to form emission probability matrix
    public double[][] computeEmissionProb(String[] ssSeq, String[] nssSeq)
{

        double[][] emissionProb = new
double[HMMEntry.getVocabSize()][HMMEntry.getStates()];
        double[][] eProbss;
        double[][] eProbnss;

        eProbss = formEmissionProbTable(ssSeq);
        eProbnss = formEmissionProbTable(nssSeq);

        for(int i = 0; i < HMMEntry.getVocabSize(); i++) {
            for(int j = 0; j < ssSeq[0].length(); j++) {
                emissionProb[i][j] = eProbss[i][j];
                emissionProb[i][j+ssSeq[0].length()] = eProbnss[i][j];
            }
        }

        return emissionProb;
    }

    // compute transition probabilities, defaults are 0.8 and 0.2 from
position k to k+1 and K+k+1
    public double[][] computeTransitionProb() {

        final int DIMENSION = HMMEntry.getStates();
```

```
        double[][] transitionProb = new double[DIMENSION][DIMENSION];

        double highProb = Math.log(0.8);
        double lowProb = Math.log(0.2);

        for(int i = 0; i < DIMENSION; i++) {
            for(int j = 0; j < DIMENSION; j++) {
                if(i == j-1 && i < (DIMENSION/2)) {
                    transitionProb[i][j] = highProb;
                    transitionProb[i][j+(DIMENSION/2)-1] = lowProb;
                }
                else if(i == j-1 && i >= (DIMENSION/2)) {
                    transitionProb[i][j] = highProb;
                    transitionProb[i][j-(DIMENSION/2)] = lowProb;
                }
                else {
                    transitionProb[i][j] = -99;
                }
            }
        }
        return transitionProb;
    }

    // compute initial probabilities, defaults are 0.3 to splice site
region and 0.7 to non-splice site region
    public double[] computeInitialProb() {

        final int DIMENSION = HMMEntry.getStates();

        double[] initialProb = new double[DIMENSION];

        initialProb[0] = Math.log(0.3);

        for(int i = 1; i < DIMENSION; i++) {
            initialProb[i] = -99;
        }

        initialProb[4] = Math.log(0.7);
        return initialProb;
    }
}
```

TrainHMM.java

```
/*
 *
 * @author: Santrupti Nerli, SJSU, March 2015
 *
 * */

package hmm_main;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

import forwardAlgo.ForwardProcedure;
import initializeHMM.InitialSetting;
import update.CalcTempVar;
import update.UpdateParameters;
import viterbi.ViterbiDecoding;
```

```
import backwardAlgo.BackwardProcedure;

public class TrainHMM {

    // variables to store HMM parameters
    double[][] emissionProb;
    double[][] transitionProb;

    // stores iteration number
    int i;

    // file paths
    String outFile;
    String testFile;

    // constructor
    public TrainHMM(int i, String outFile, String testFile) {

        this.outFile = outFile;
        this.testFile = testFile;

        transitionProb = new
double[HMMEntry.getStates()][HMMEntry.getStates()];
        emissionProb = new
double[HMMEntry.getVocabSize()][HMMEntry.getStates()];
        this.i = i;
    }

    // update transition probabilities
    public void updateTP(double[][] tpN, double[][] tpD) {
        for(int i = 0; i < HMMEntry.getStates(); i++) {
            for(int j = 0; j < HMMEntry.getStates(); j++) {
                transitionProb[i][j] = Math.log(tpN[i][j]) -
Math.log(tpD[i][j]);
            }
        }
    }

    // update emission probabilities
    public void updateEP(double[][] epN, double[][] epD) {
        for(int i = 0; i < HMMEntry.getVocabSize(); i++) {
            for(int j = 0; j < HMMEntry.getStates(); j++) {
                emissionProb[i][j] = Math.log(epN[i][j]) -
Math.log(epD[i][j]);
            }
        }
    }

    // start training for every observation sequence in every iteration
    public void startTraining(HMMEntry hmmObject) throws IOException {

        InitialSetting randomSet = new InitialSetting();

hmmObject.setEmissionProb(randomSet.computeEmissionProb(hmmObject.ssSeq,
hmmObject.nssSeq));
        hmmObject.setIntialProb(randomSet.computeInitialProb());
        hmmObject.setTransitionProb(randomSet.computeTransitionProb());

        System.out.println("\nInitial Prob");
        HMMEntry.printArray(hmmObject.getIntialProb(),
HMMEntry.getStates());
        System.out.println("\nEmission Prob");
```

```
        HMMEntry.printMatrix(hmmObject.getEmissionProb(),
HMMEntry.getVocabSize(), HMMEntry.getStates());
        System.out.println("\nTransition Prob");
        HMMEntry.printMatrix(hmmObject.getTransitionProb(),
HMMEntry.getStates(), HMMEntry.getStates());

        int iteration;

        for(iteration = 0; iteration < HMMEntry.MAX_ITERATIONS;
iteration++) {

            hmmObject.storeCurrentValues();

            for(int steps = 0; steps < hmmObject.ssSeq.length +
hmmObject.nssSeq.length; steps++) {

                int[] codedSeq;
                if(steps < hmmObject.ssSeq.length) {
                    codedSeq =
HMMEntry.computeCode(hmmObject.ssSeq[steps]);
                }
                else {
                    codedSeq = HMMEntry.computeCode(hmmObject.nssSeq[steps-
hmmObject.ssSeq.length]);
                }

                ForwardProcedure f = new ForwardProcedure();
                f.alphaRecurrence(hmmObject, codedSeq);

                BackwardProcedure b = new BackwardProcedure();
                b.betaRecurrence(hmmObject, codedSeq);

                CalcTempVar c = new CalcTempVar(hmmObject, f, b, codedSeq);
                c.computeGamma();
                c.computeEta();

                UpdateParameters u = new UpdateParameters(hmmObject, c,
codedSeq, steps);
                u.updateInitialProb();
                u.updateTransitionProb(hmmObject);
                u.updateEmissionProb(hmmObject);

            }

            updateTP(hmmObject.transitionProbNum,
hmmObject.transitionProbDen);
            updateEP(hmmObject.emissionProbNum, hmmObject.emissionProbDen);

            if(hmmObject.checkConvergence(transitionProb, emissionProb,
iteration)) {
                break;
            }

            hmmObject.setEmissionProb(emissionProb);
            hmmObject.setTransitionProb(transitionProb);

        }

        System.out.println("\nInitial Prob");
        HMMEntry.printArray(hmmObject.getIntialProb(),
HMMEntry.getStates());
        System.out.println("\nEmission Prob");
```

```
        HMMEntry.printMatrix(hmmObject.getEmissionProb(),
HMMEntry.getVocabSize(), HMMEntry.getStates());
        System.out.println("\nTransition Prob");
        HMMEntry.printMatrix(hmmObject.getTransitionProb(),
HMMEntry.getStates(), HMMEntry.getStates());

        System.out.println("Converging after iterations: " + iteration);

        BufferedWriter brw = new BufferedWriter(new FileWriter(outFile));
        BufferedReader brr = new BufferedReader(new FileReader(testFile));
        String line = brr.readLine();
        while(line != null) {
            ViterbiDecoding v = new ViterbiDecoding(hmmObject, line);
            double score = v.decode(hmmObject);
            brw.write(line + "\t" + score + "\n");
            line = brr.readLine();
        }
        brw.close();
        brr.close();
    }
}
```

ForwardProcedure.java

```
/*
 *
 * @author: Santrupti Nerli, SJSU, March 2015
 *
 * */

package forwardAlgo;

import hmm_main.HMMEntry;

public class ForwardProcedure {

    // variable that holds forward values
    private static double[][] alpha;

    // HMM parameters
    double[][] emissionProb;
    double[][] transitionProb;
    double[] initialProb;
    int[] seq;

    // constructor
    public ForwardProcedure() {
        alpha = new double[HMMEntry.getStates()][HMMEntry.T];
    }

    // getter that returns forward variable
    public double[][] getAlpha() {
        return alpha;
    }

    // compute forward values using dynamic programming technique
    public void computeAlpha() {

        for(int i = 0; i < HMMEntry.getStates()/2; i++) {
            alpha[i][0] = initialProb[i] + emissionProb[seq[0]][i];
            alpha[i + HMMEntry.T][0] = initialProb[i + HMMEntry.T] +
emissionProb[seq[0]][i + HMMEntry.T];
        }
```

```
        for(int t = 0; (t+1) < HMMEntry.T; t++) {
            for(int j = 0; j < HMMEntry.getStates()/2; j++) {
                double evaluate1 = 0.0, evaluate2 = 0.0;
                for(int i = 0; i < HMMEntry.getStates(); i++) {
                    evaluate1 = evaluate1 + Math.pow(Math.E, (alpha[i][t] +
transitionProb[i][j]));
                    evaluate2 = evaluate2 + Math.pow(Math.E, (alpha[i][t] +
transitionProb[i][j + HMMEntry.T]));
                }
                alpha[j][t+1] = emissionProb[seq[t+1]][j] +
Math.log(evaluate1);
                alpha[j + HMMEntry.T][t+1] = emissionProb[seq[t+1]][j +
HMMEntry.T] + Math.log(evaluate2);
            }
        }
    }

    // API that calls interface to compute forward variable
    public void alphaRecurrence(HMMEntry hmmObject, int[] codedSeq) {

        emissionProb = hmmObject.getEmissionProb();
        initialProb = hmmObject.getIntialProb();
        transitionProb = hmmObject.getTransitionProb();
        seq = codedSeq;

        computeAlpha();
    }
}
```

BackwardProcedure.java

```
/*
 *
 * @author: Santrupti Nerli, SJSU, March 2015
 *
 * */

package backwardAlgo;

import hmm_main.HMMEntry;

public class BackwardProcedure {

    // beta stores the backward variable
    private static double[][] beta;

    // HMM parameters
    double[][] emissionProb;
    double[][] transitionProb;
    double[] initialProb;
    int[] seq;

    // constructor
    public BackwardProcedure() {
        beta = new double[HMMEntry.getStates()][HMMEntry.T];
    }

    // getter to get backward variable
    public double[][] getBeta() {
        return beta;
    }
```

```
    // computes backward variable using dynamic programming technique
    public void computeBeta() {

        for(int i = 0; i < HMMEntry.getStates(); i++) {
            beta[i][HMMEntry.T-1] = 0;
        }

        for(int t = HMMEntry.T-2; t >= 0; t--) {
            for(int i = 0; i < HMMEntry.getStates()/2; i++) {
                double sum1 = 0, sum2 = 0;
                for(int j = 0; j < HMMEntry.getStates(); j++) {
                    sum1 = sum1 + Math.pow(Math.E, (beta[j][t+1] +
transitionProb[i][j] + emissionProb[seq[t+1]][j]));
                    sum2 = sum2 + Math.pow(Math.E, (beta[j][t+1] +
transitionProb[i + HMMEntry.T][j] + emissionProb[seq[t+1]][j]));
                }
                beta[i][t] = Math.log(sum1);
                beta[i + HMMEntry.T][t] = Math.log(sum2);
            }
        }
    }

    // API that calls interface to compute backward variable
    public void betaRecurrence(HMMEntry hmmObject, int[] codedSeq) {

        emissionProb = hmmObject.getEmissionProb();
        initialProb = hmmObject.getIntialProb();
        transitionProb = hmmObject.getTransitionProb();
        seq = codedSeq;

        computeBeta();
    }
}
```

CalcTempVar.java

```
/*
 *
 * @author: Santrupti Nerli, SJSU, March 2015
 *
 * */

package update;

import backwardAlgo.BackwardProcedure;
import forwardAlgo.ForwardProcedure;
import hmm_main.HMMEntry;

public class CalcTempVar {

    // store temporary variables gamma and eta
    double[][] gamma;
    double[][][] eta;

    // we need forward and backward variables to compute gamma and eta
    double[][] alpha;
    double[][] beta;

    // represents sequence of observation
    int[] seq;

    // HMM parameters
    double[][] emissionProb;
```

```
    double[][] transitionProb;
    double[] initialProb;

    // constructor
    public CalcTempVar(HMMEntry hmmObject, ForwardProcedure f,
BackwardProcedure b, int[] codedSeq) {
        alpha = f.getAlpha();
        beta = b.getBeta();

        gamma = new double[HMMEntry.getStates()][HMMEntry.T];
        eta = new
double[HMMEntry.getStates()][HMMEntry.getStates()][HMMEntry.T];

        emissionProb = hmmObject.getEmissionProb();
        initialProb = hmmObject.getIntialProb();
        transitionProb = hmmObject.getTransitionProb();

        seq = codedSeq;
    }

    // getters
    public double[][] getGamma() {
        return gamma;
    }

    public double[][][] getEta() {
        return eta;
    }

    // denominators while computing gamma and eta are same
    public double denominator(int t) {

        double value = 0;
        for(int j = 0; j < HMMEntry.getStates(); j++) {
            value = value + Math.pow(Math.E, (alpha[j][t] + beta[j][t]));
        }
        return Math.log(value);
    }

    // compute gamma
    public void computeGamma() {
        for(int t = 0; t < HMMEntry.T; t++) {
            double den = denominator(t);
            for(int i = 0; i < HMMEntry.getStates()/2; i++) {
                gamma[i][t] = alpha[i][t] + beta[i][t] - den;
                gamma[i + HMMEntry.T][t] = alpha[i + HMMEntry.T][t] +
beta[i + HMMEntry.T][t] - den;
            }
        }
    }

    // compute eta
    public void computeEta() {

        for(int t = 0; t < HMMEntry.T-1; t++) {
            double den = denominator(t);
            for(int i = 0; i < HMMEntry.getStates()/2; i++) {
                for(int j = 0; j < HMMEntry.getStates()/2; j++) {
                    eta[i][j][t] = alpha[i][t] + transitionProb[i][j] +
beta[j][t+1] + emissionProb[seq[t+1]][j] - den;
```

```
                    eta[i][j + HMMEntry.T][t] = alpha[i][t] +
transitionProb[i][j + HMMEntry.T] + beta[j + HMMEntry.T][t+1] +
emissionProb[seq[t+1]][j + HMMEntry.T] - den;
                    eta[i + HMMEntry.T][j][t] = alpha[i + HMMEntry.T][t] +
transitionProb[i + HMMEntry.T][j] + beta[j][t+1] +
emissionProb[seq[t+1]][j] - den;
                    eta[i + HMMEntry.T][j + HMMEntry.T][t] = alpha[i +
HMMEntry.T][t] + transitionProb[i + HMMEntry.T][j + HMMEntry.T] + beta[j +
HMMEntry.T][t+1] + emissionProb[seq[t+1]][j + HMMEntry.T] - den;
                }
            }
        }

        System.out.println("Eta");
        for(int i = 0; i < HMMEntry.getStates(); i++) {
            System.out.println("State: " + i);
            HMMEntry.printMatrix(eta[i], HMMEntry.getStates(), HMMEntry.T);
        }

        System.exit(0);
    }
}
```

UpdateParameters.java

```
/*
 *
 * @author: Santrupti Nerli, SJSU, March 2015
 *
 * */

package update;

import hmm_main.HMMEntry;

public class UpdateParameters {

    // observation sequence
    int[] seq;

    // temporary variables
    double[][] gamma;
    double[][][] eta;

    // number of observations
    int K;

    // HMM parameters
    double[] initialProb;

    // constructor
    public UpdateParameters(HMMEntry hmmObject, CalcTempVar c, int[]
codedSeq, int steps) {

        final int DIMENSION = HMMEntry.getStates();

        initialProb = new double[DIMENSION];

        gamma = c.getGamma();
        eta = c.getEta();

        seq = codedSeq;
```

```
        K = steps;
    }

    // update initial probability after training
    public void updateInitialProb() {

        for(int i = 0; i < HMMEntry.getStates(); i++) {
            initialProb[i] = gamma[i][1];
        }
    }

    // update transition probability after training
    public void updateTransitionProb(HMMEntry hmmObject) {
        for(int i = 0; i < HMMEntry.getStates()/2; i++) {
            for(int j = 0; j < HMMEntry.getStates()/2; j++) {
                double etaSum1 = 0, etaSum2 = 0, etaSum3 = 0, etaSum4 = 0;;
                for(int t = 0; t < HMMEntry.T-1; t++) {
                    etaSum1 += Math.pow(Math.E, eta[i][j][t]);
                    etaSum2 += Math.pow(Math.E, eta[i][j + HMMEntry.T][t]);
                    etaSum3 += Math.pow(Math.E, eta[i + HMMEntry.T][j][t]);
                    etaSum4 += Math.pow(Math.E, eta[i + HMMEntry.T][j +
HMMEntry.T][t]);
                }
                hmmObject.transitionProbNum[i][j] += etaSum1;
                hmmObject.transitionProbNum[i][j + HMMEntry.T] += etaSum2;
                hmmObject.transitionProbNum[i + HMMEntry.T][j] += etaSum3;
                hmmObject.transitionProbNum[i + HMMEntry.T][j + HMMEntry.T]
+= etaSum4;

                double sum1 = sumGamma(i);
                double sum2 = sumGamma(i + HMMEntry.T);
                hmmObject.transitionProbDen[i][j] += sum1;
                hmmObject.transitionProbDen[i][j+ HMMEntry.T] += sum1;
                hmmObject.transitionProbDen[i+ HMMEntry.T][j] += sum2;
                hmmObject.transitionProbDen[i+ HMMEntry.T][j+ HMMEntry.T]
+= sum2;
            }
        }
    }

    // computes summation of temporary variable gamma required to update
both transition and emission probabilities
    public double sumGamma(int i) {
        double sum = 0;
        for(int t = 0; t < HMMEntry.T-1; t++) {
            sum = sum + Math.pow(Math.E, gamma[i][t]);
        }
        return sum;
    }

    // update emission probability after training
    public void updateEmissionProb(HMMEntry hmmObject) {

        for(int k = 0; k < HMMEntry.getVocabSize(); k++) {
            for(int i = 0; i < HMMEntry.getStates()/2; i++) {
                double numerator1 = 0, numerator2 = 0;
                for(int t = 0; t < HMMEntry.T-1; t++) {
                    if(seq[t] == k) {
                        numerator1 += Math.pow(Math.E, gamma[i][t]);
                        numerator2 += Math.pow(Math.E, gamma[i +
HMMEntry.T][t]);
                    }
```

```
                }
                hmmObject.emissionProbNum[k][i] += numerator1;
                hmmObject.emissionProbNum[k][i + HMMEntry.T] += numerator2;

                hmmObject.emissionProbDen[k][i] += sumGamma(i);
                hmmObject.emissionProbDen[k][i + HMMEntry.T] += sumGamma(i
+ HMMEntry.T);
            }
        }
    }
}
```

ViterbiDecoding.java

```
/*
 *
 * @author: Santrupti Nerli, SJSU, March 2015
 *
 * */

package viterbi;

import hmm_main.HMMEntry;

public class ViterbiDecoding {

    // HMM parameters
    private double[][] emissionProb;
    private double[][] transitionProb;
    private double[] initialProb;

    // unknown sequence for which we will find the most likely path
    private String unknownSeq;

    // states in HMM
    int states;
    int[] state;

    // constructor
    public ViterbiDecoding(HMMEntry hmmObject, String scoreSeq) {
        emissionProb = hmmObject.getEmissionProb();
        transitionProb = hmmObject.getTransitionProb();
        initialProb = hmmObject.getIntialProb();

        states = HMMEntry.getStates();
        unknownSeq = scoreSeq;

        state = new int[states];
        for(int i = 0; i < states; i++) {
            state[i] = i;
        }
    }

    // decode the unknown sequence using Viterbi algorithm
    public double decode(HMMEntry hmmObject) {

        double[][] T1 = new double[states][unknownSeq.length()];
        int[][] T2 = new int[states][unknownSeq.length()];
        int[] z = new int[unknownSeq.length()];
        int[] x = new int[unknownSeq.length()];

        int[] codedSeq = HMMEntry.computeCode(unknownSeq);
```

```
        for(int i = 0; i < states; i++) {
            T1[i][0] = initialProb[i] + emissionProb[codedSeq[0]][i];
            T2[i][0] = 0;
        }

        double intermediate;
        for(int i = 1; i < unknownSeq.length(); i++) {

            intermediate = 0.0;
            for(int j = 0; j < states; j++)
            {
                T1[j][i] = (T1[0][i-1] + transitionProb[0][j] +
emissionProb[codedSeq[i]][j]);
                for(int k = 1; k < states; k++)
                {
                    intermediate = (T1[k][i-1] + transitionProb[k][j] +
emissionProb[codedSeq[i]][j]);
                    if(T1[j][i] < intermediate)
                    {
                        T1[j][i] = intermediate;
                        T2[j][i] = k;
                    }
                }
            }
        }

        double max = -999.99;
        for(int k = 0; k < states; k++)
        {
            if(max < T1[k][unknownSeq.length()-1])
            {
                max = T1[k][unknownSeq.length()-1];
                z[unknownSeq.length()-1] = k;
            }
        }

        x[unknownSeq.length()-1] = state[z[unknownSeq.length()-1]];

        for(int i = unknownSeq.length()-1; i > 0; i--)
        {
            z[i-1] = T2[z[i]][i];
            x[i-1] = state[z[i-1]];
        }

        System.out.println("\nDecoded Sequence:");
        for(int i = 0; i < x.length; i++) {
            System.out.print((x[i]+1) + " ");
        }
        System.out.println();

        ScoreSequences s = new ScoreSequences(hmmObject, x,
unknownSeq.length(), codedSeq);
        double score = s.score();
        return score;
    }
}
```

ScoreSequences.java

```
/*
 *
 * @author: Santrupti Nerli, SJSU, March 2015
 *
```

```
 * */

package viterbi;

import hmm_main.HMMEntry;

public class ScoreSequences {

    // HMM parameters
    private double[][] emissionProb;
    private double[][] transitionProb;
    private double[] initialProb;

    // sequence to score
    private int[] Seq;
    private int[] codedSeq;
    private int seqLen;

    // constructor
    public ScoreSequences(HMMEntry hmmObject, int[] scoreSeq, int len,
int[] coded) {
        emissionProb = hmmObject.getEmissionProb();
        transitionProb = hmmObject.getTransitionProb();
        initialProb = hmmObject.getIntialProb();

        Seq = scoreSeq;
        codedSeq = coded;
        seqLen = len;
    }

    // score any sequence using initial, transition and emission
probabilities
    public double score() {
        double score = initialProb[Seq[0]];
        for(int i = 1; i < seqLen; i++) {
            score += emissionProb[codedSeq[i]][i] + transitionProb[i-1][i];
        }
        return score;
    }

}
```