San Jose State University

SJSU ScholarWorks

Master's Projects

Master's Theses and Graduate Research

Fall 2013

# Higher Order PWM for Modeling Transcription Factor Binding Sites

Dhivya Srinivasan
*San Jose State University*

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Computer Sciences Commons

Recommended Citation
Srinivasan, Dhivya, "Higher Order PWM for Modeling Transcription Factor Binding Sites" (2013). *Master's Projects*. 327.
DOI: https://doi.org/10.31979/etd.yvas-9f4b
https://scholarworks.sjsu.edu/etd_projects/327

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

# Higher Order PWM for Modeling Transcription Factor Binding Sites

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Dhivya Srinivasan

December 2013

© 2013

Dhivya Srinivasan

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Higher Order PWM for Modeling Transcription Factor Binding Sites

by

Dhivya Srinivasan

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2013

| | |
|---|---|
| Dr. Sami Khuri | Department of Computer Science |
| Dr. Mark Stamp | Department of Computer Science |
| Dr. Chris Tseng | Department of Computer Science |

## ABSTRACT

**Higher Order PWM for Modeling Transcription Factor Binding Sites**

**by Dhivya Srinivasan**

Traditional Position Weight Matrices (PWMs) that are used to model Transcription Factor Binding Sites (TFBS) assume independence among different positions in the binding site. In reality, this may not necessarily be the case. A better way to model TFBS is to consider the distribution of dinucleotides or trinucleotides instead of just mononucleotides, thus taking neighboring nucleotides into account. We can therefore, extend the single nucleotide PWM to a dinucleotide PWM or an even higher-order PWM to correctly estimate the dependencies among the nucleotides in a given sequence. The purpose of this project is to develop an algorithm to implement higher-order PWMs to detect the TFBS and other biological motifs in DNA, RNA, and proteins.

## ACKNOWLEDGEMENTS

I am extremely obliged to my project advisor, Dr. Sami Khuri, for his guidance, encouragement, and support throughout this project. I would also like to thank my committee members, Dr. Mark Stamp and Dr. Chris Tseng for their time and support.

My special thanks to Natalia Khuri for her invaluable insights and suggestions during the course of this project.

# TABLE OF CONTENTS

CHAPTER 1 ..... 1

Introduction ..... 1

1.1 Background ..... 2

CHAPTER 2 ..... 3

Representation Models ..... 3

2.1 Introduction ..... 3

2.2 Consensus Model ..... 3

CHAPTER 3 ..... 6

Position Weight Matrices ..... 6

3.1 Introduction ..... 6

3.2 Pseudocounts ..... 7

3.3 Scoring the PWM ..... 8

CHAPTER 4 ..... 9

Determining the Threshold ..... 9

4.1 Sensitivity and Specificity ..... 9

4.2 Receiver Operating Characteristic Curve ..... 9

4.3 Limitations of scoring using relative frequencies ..... 10

CHAPTER 5 ..... 11

Log-likelihood Ratios ..... 11

5.1 Log-likelihood ..... 11

5.2 Scoring the PWM ..... 12

5.3 Logo Representation and Information Content ..... 12

5.4 PWM Limitations ..... 14

CHAPTER 6 ..... 16

Higher-order PWM and MDD ..... 16

6.1 The MDD Algorithm ..... 18

CHAPTER 7 ..... 27

Implementation ..... 27

7.1 PWM Implementation ..... 27

7.2 Higher-order PWM Implementation ... 27
7.3 MDD Implementation... 28
7.4 MDD Stopping Criteria ... 28
7.5 MDD Traversal... 29
CHAPTER 8 ... 30
DATASETS AND RESULTS ... 30
8.1 Dataset and Testing ... 30
8.2 MDD Clustering ... 35
8.3 Extensions... 37
8.4 Comparative Analysis... 45
8.5 Conclusion and Future Work... 46
REFERENCES ... 47
APPENDIX... 49

**LIST OF FIGURES**

Figure 1: Ten Experimentally Determined Binding Sites for yeast TF Leu3 ........ 4

Figure 2: IUPAC Nucleotide Codes ........ 5

Figure 3: Position Weight Matrix Showing Relative Frequencies of the Nucleotides ........ 7

Figure 4: Receiver Operating Characteristic Curve ........ 10

Figure 5: PWM showing Log-likelihood Ratios ........ 12

Figure 6: Sequence Logo Representation of the Motif ........ 13

Figure 7: Sequences of a Hypothetical Binding Site ........ 14

Figure 8: Maximal Dependence Decomposition Modeling and Scoring ........ 17

Figure 9: Sequences of a Hypothetical Binding Site ........ 19

Figure 10: MDD approach applied on the example sequences ........ 26

Figure 11: ROC Curve of a Simple PWM for Yeast Dataset ........ 32

Figure 12: ROC Curve of Dinucleotide PWM for Yeast Dataset ........ 33

Figure 13: ROC Curve of MDD for Yeast Dataset ........ 34

Figure 14: Screenshot from Jaspar showing the TFs of *Zea mays* ........ 36

Figure 15: The MDD sequence logo tree showing the clustering of the TFs of *Zea mays* ........ 37

Figure 16: ROC Curve of a Simple PWM for Splice Sites (Exon-Intron) ........ 39

Figure 17: ROC Curve of Dinucleotide PWM for Splice Sites (Exon-Intron) ........ 40

Figure 18: ROC Curve of MDD for Splice Sites (Exon-Intron) ........ 41

Figure 19: ROC Curve of a Simple PWM for Splice Sites (Intron-Exon) ........ 43

Figure 20: ROC Curve of Dinucleotide PWM for Splice Sites (Intron-Exon) ........ 44

Figure 21: ROC Curve of MDD for Splice Sites (Intron-Exon) ........ 45

## LIST OF TABLES

Table 1: $\chi^2$ Distribution Table ........ 20

Table 2: Initial partition of the sequences ........ 21

Table 3: $\chi^2$ statistic showing dependencies of various positions ........ 22

Table 4: Table formed after dividing the sequences based on position 2 ........ 23

Table 5: $\chi^2$ distribution after partitioning the sequences based on position 2 ........ 23

Table 6: Table formed after dividing the sequences based on position 1 ........ 24

Table 7: $\chi^2$ distribution after partitioning the sequences based on position 1 ........ 24

Table 8: Table showing the relative performance of the three models over the three datasets tested ........ 45

# CHAPTER 1

# Introduction

Traditionally, two computational models have been used to summarize experimentally determined binding sites: consensus sequences and Position Weight Matrices (PWMs). The consensus sequence is a simple approach but it systematically ignores the rare bases at each position, which might represent biologically important instances of binding sites. Due to this restrictive nature, PWMs are preferred instead. PWMs are currently the most common representation of Transcription Factor Binding Sites (TFBS) and unlike the consensus approach; it captures all observed bases at each position.

A PWM is a probability matrix with four rows corresponding to the four nucleotide bases and K columns corresponding to each position in the binding site. The weight matrix gives the frequency distribution of each of the four nucleotides at every position in the binding site. While a PWM is simple, intuitive and most commonly used, the main drawback is that it assumes independence among different positions in the binding site, which may not be the case.

A better way to model the TFBS is by looking at the distribution of dinucleotides instead of just mononucleotides, thus taking neighboring nucleotides into account. We can therefore, extend the single nucleotide PWM with 4 rows and $K$ columns to a dinucleotide (or higher-order) PWM with 16 rows and $K-1$ columns.

The purpose of this project is to develop an algorithm to implement the higher-order PWM and an approach called Maximal Dependence Decomposition as better models to determine TFBS. The algorithm is implemented in Java, and tested on several input data sets. The results obtained from the program are compared to the ones that use the regular PWM with the same input data. The algorithm that we designed, implemented, and tested is not confined to the detection of Transcription Factors (TFs). With very minor modifications to our program, it can be used to detect other biological patterns and motifs in DNA, RNA, and proteins.

## 1.1 Background

Transcription is the process by which messenger RNA is synthesized from a DNA template. TFs are proteins that regulate transcription. These may facilitate or inhibit the recruitment of the RNA polymerase by binding to DNA near the gene that they regulate. The binding of the TFs to DNA requires specific short cis-regulatory sequences called binding sites, usually located upstream of the 5' end of the gene [1].

Different DNA binding sites share consensus base-pairs that "almost always" appear at the same position in every site. These consensus base-pairs then form a distinct pattern that can help the biologist to identify previously unknown recognition sites in other DNA sequences. Detection of these binding site motifs, therefore, is very important in the study and understanding of gene regulation. These motifs are however, very degenerate and hence it is very difficult to build reliable models to identify TFBS.

For a specific TF, there are several experimental techniques, such as SELEX and DNA Microarrays that are used to determine the binding sites. These binding site sequences are then used to construct a model for that specific TF binding. JASPAR [2] and TRANSFAC [3] are some of the very popular databases that store such experimentally determined binding sites.

In the next chapter, we will look into the representation models in detail.

# CHAPTER 2

# Representation Models

## 2.1 Introduction

As mentioned in the last chapter, there are several experimental techniques to determine the TFBS. Our focus here is on constructing a model using these binding site sequences, and to efficiently use that model to determine other unknown binding sites for that particular TF. Some of the common ways to summarize these binding sites is to use a *consensus* model or a *PWM*.

## 2.2 Consensus Model

The consensus representation is a simple and straightforward approach where we create a consensus string of length $K$ and place in position $j$ the consensus nucleotide which occurs with the highest frequency at position $j$ in $N$ binding sites [4].

**Example 1**

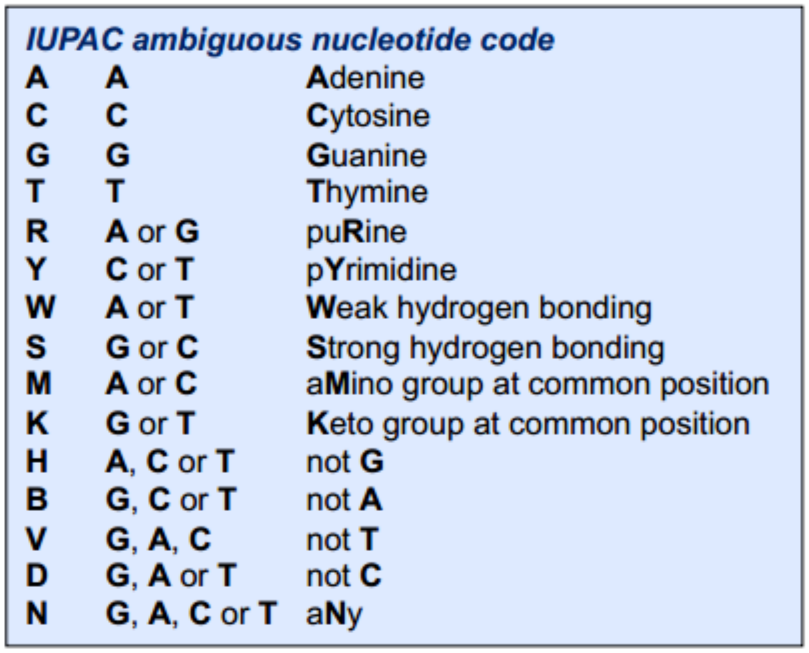
In Figure 1, there are 10 binding site sequences of length 10 each. On carefully observing the sequences, we notice that nucleotide *C* occurs most of the time (8 out of 10 times) at position 1. Nucleotide *C* is also the consensus at position 2 and the consensus at position 3 is *G*. Similarly, the consensus nucleotides at all the positions are noted and the consensus string is determined. As can be seen in Example 1, for the given ten binding sites, the consensus sequence is *CCGGTACCGG*.

|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $X_1$ | C | C | G | G | T | A | C | C | G | G |
| $X_2$ | C | C | T | G | T | A | C | C | G | G |
| $X_3$ | C | C | G | C | T | A | C | C | G | G |
| $X_4$ | C | C | G | G | A | A | C | C | G | G |
| $X_5$ | G | C | G | G | T | A | C | C | G | G |
| $X_6$ | C | C | G | T | T | A | C | C | G | G |
| $X_7$ | C | C | G | C | A | A | C | C | G | G |
| $X_8$ | C | C | T | G | A | A | C | C | G | G |
| $X_9$ | G | C | G | G | T | A | A | C | G | G |
| $X_{10}$ | C | C | G | C | T | A | C | A | G | G |

Figure 1: Ten Experimentally Determined Binding Sites for yeast TF Leu3

In Example 1, there is no ambiguity in any of the positions, i.e., there is only one nucleotide that occurs most of the times. If, on the other hand, we have cases where nucleotides C and G are equally likely at a particular position, then we can't clearly assign one of the nucleotides as the consensus base. In order to address this problem, we can use the IUPAC letter code (Figure 2 [5]) to denote the possible combinations between the 4 nucleotides [4]. Using IUPAC codes is not very effective as it gives a poor idea of the relative importance of the nucleotides. The consensus representation, though simple, is very restrictive as it ignores the rare bases at each position, which might represent biological important instances of binding sites.

PWMs, which will be discussed in the next chapter, are therefore preferred to overcome this drawback.



*IUPAC ambiguous nucleotide code*

| | | |
|---|---|---|
| **A** | **A** | **A**denine |
| **C** | **C** | **C**ytosine |
| **G** | **G** | **G**uanine |
| **T** | **T** | **T**hymine |
| **R** | **A** or **G** | pu**R**ine |
| **Y** | **C** or **T** | p**Y**rimidine |
| **W** | **A** or **T** | **W**eak hydrogen bonding |
| **S** | **G** or **C** | **S**trong hydrogen bonding |
| **M** | **A** or **C** | a**M**ino group at common position |
| **K** | **G** or **T** | **K**eto group at common position |
| **H** | **A**, **C** or **T** | not **G** |
| **B** | **G**, **C** or **T** | not **A** |
| **V** | **G**, **A**, **C** | not **T** |
| **D** | **G**, **A** or **T** | not **C** |
| **N** | **G**, **A**, **C** or **T** | a**N**y |

Figure 2: IUPAC Nucleotide Codes

# CHAPTER 3

# Position Weight Matrices

## 3.1 Introduction

Position Weight Matrices are the most commonly used representations of TFBS and can be described by a $4*K$ probability matrix with the $4$ rows representing the 4 nucleotides and the $K$ columns representing each position in the binding site. Let us denote the frequency of nucleotide $x$ occurring at position $j$ among the given binding sites as $f_{x,\,j.}$ For Example 1, the relative frequency of each nucleotide at a particular position is calculated as follows:

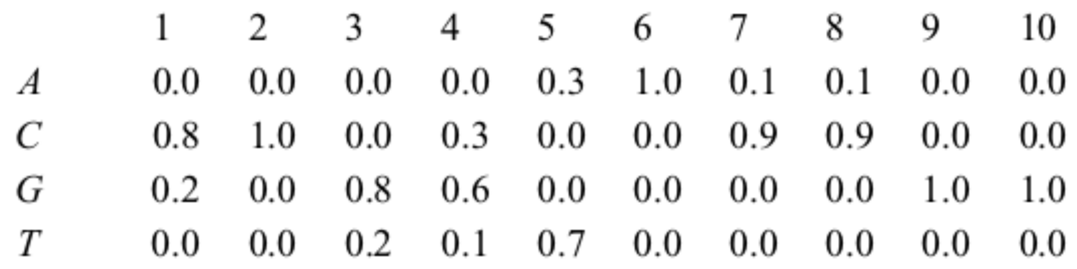
$$f_{x,\,j} = (N_{x,\,j})/\,N$$

where $N_{x,\,j}$ is the number of times nucleotide $x$ occurs at position $j$ and $N$ is the total number of nucleotides at position $j$ (i.e., the number of sequences).

Among the binding sites, nucleotide C occurs 8 out of 10 times, G occurs twice and nucleotides A and T do not appear at all, at position 1. Hence, the relative frequencies of these bases at position 1 are

$f_A$: 0/10 = 0 $\quad$ $f_C$: 8/10 = 0.8 $\quad$ $f_G$: 2/10 = 0.2 $\quad$ $f_T$: 0/10 = 0

Similarly, the relative frequencies of the four nucleotides in the remaining positions are calculated and tabulated in the Position Weight Matrix (PWM) of Figure 3 [4].



|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| *A* | 0.0 | 0.0 | 0.0 | 0.0 | 0.3 | 1.0 | 0.1 | 0.1 | 0.0 | 0.0 |
| *C* | 0.8 | 1.0 | 0.0 | 0.3 | 0.0 | 0.0 | 0.9 | 0.9 | 0.0 | 0.0 |
| *G* | 0.2 | 0.0 | 0.8 | 0.6 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| *T* | 0.0 | 0.0 | 0.2 | 0.1 | 0.7 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |

Figure 3: Position Weight Matrix Showing Relative Frequencies of the Nucleotides

**3.2 Pseudocounts**

As can be seen from Figure 3, some nucleotides do not appear at all (represented by zero in the weight matrix) at certain positions. This would mean that these nucleotides are not allowed to appear at these positions but in reality, this is not necessarily the case. The reason we observe zero nucleotides at a particular position can be attributed to the fact that our sample set is too small. This is caused by data insufficiency. This problem can be overcome by using "*Pseudocounts*" or "*Priors*" before computing the frequencies. By adding pseudocounts to the observed count of nucleotides at a position, we can account for the unobserved nucleotides at that position.

Laplace prior [6] can be used to recalculate the relative frequencies of the nucleotides as follows:

$$f_{x,j} = (N_{x,j} + 1) / (N + 4)$$

Using this in our example to calculate the frequency distribution at position 1, we have

A: (0 + 1)/ (10 + 4) = 0.0714

C: (8 + 1)/ (10 + 4) = 0.642

G: (2 + 1)/ (10 + 4) = 0.214

T: (0 + 1)/ (10 + 4) = 0.0714

### 3.3 Scoring the PWM

Now that a PWM has been constructed, we can use it to score a given motif and see if it is a binding site for that particular TF or not. Let us consider the nucleotide sequence *CCGCTACAGG* and compute the score for this motif against the PWM we have created. From Figure 3, we know that the probabilities of nucleotide C occurring at position 1 is 0.8, nucleotide C occurring at position 2 is 1.0, and so on. The score for the given motif is computed by simply multiplying these values.

Score = 0.8 * 1.0 * 0.8 * 0.3 * 0.7 * 1.0 * 0.9 * 0.1 * 1.0 * 1.0 = 0.012096.

If the computed score is above a threshold value, then we can say that the motif is a potential binding site for that TF.

This threshold value is determined by plotting a Receiver Operating Characteristic (ROC) curve, which is explained in detail in the next chapter.

# CHAPTER 4

# Determining the Threshold

## 4.1 Sensitivity and Specificity

The threshold value for any test is determined using two factors: Sensitivity and Specificity [7]. Sensitivity is the proportion of actual positives, which are correctly identified, and Specificity is the proportion of negatives that are correctly identified. These two values describe how well a particular test discriminates between the two test conditions [7]. In the current context of TFBS, the test conditions are whether a given motif is a match or not.

## 4.2 Receiver Operating Characteristic Curve

An ROC curve [7] is a plot of the true positive rate against the false positive rate for different possible cut-points of a test. As can be seen from Figure 4, it is basically a graph of Sensitivity (y axis) vs. 1-Specificity (x axis). The ideal cut-off value is a trade-off between Sensitivity and Specificity. Usually, a higher Sensitivity is chosen at the cost of lower Specificity.

The closer the curve follows the left-hand border and the top border of the ROC space, the more accurate the test is. The Area under the Curve (AUC) is recognized as the measure of a test's discriminatory power. If the area is exactly 1, then the test is considered to be perfect (100% Sensitive and 100% Specific). If the AUC is 0.5, then the test is considered worthless as there is no discriminative value with the test being 50% Sensitive and 50% Specific [7]. Depending on the requirements, the Sensitivity and Specificity values can be adjusted to obtain an optimal threshold value for the PWM.

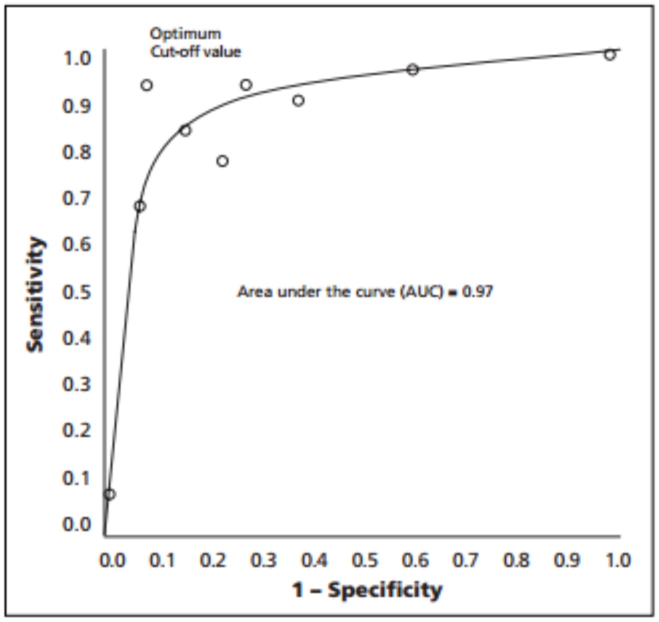


Figure 4: Receiver Operating Characteristic Curve

## 4.3 Limitations of scoring using relative frequencies

As discussed in the previous chapter, the score for a given motif is calculated by computing the product of the relative frequencies of the nucleotides. We can observe from the weight matrix that the values are very small and multiplication of such small numbers result in even smaller numbers and for very long sequences, might give an underflow. In order to overcome this, log-likelihood ratios are used to construct the Weight Matrix. By using logarithms, the score can be calculated by adding the values from the weight matrix instead of multiplying them. Depending on the log-likelihood value for a particular nucleotide at a particular position, we can deduce whether that nucleotide is the consensus base at that position or not.

The next chapter deals with likelihood ratios and explains why log-likelihood ratios are preferred while computing the score in PWMs.

# CHAPTER 5

# Log-likelihood Ratios

## 5.1 Log-likelihood

In order to make the computations simpler, we use log-likelihood ratios to construct the PWM. Likelihood ratio is defined as the ratio of observed frequency over the expected frequency. The logarithm of this ratio is the log-likelihood ratio. If the log-likelihood ratio is greater than one, it signifies that the nucleotide is more likely to occur at that particular position than it is to occur overall. Similarly, a value less than one indicate that the nucleotide is less likely to occur at that particular position than it is to occur overall.

For Example 1, we can compute the log-likelihood ratios as follows:

Log-likelihood ratio = Log (Observed Frequency / Expected Frequency).

Expected Frequency could be a prior probability or we could assume all the nucleotides to be equiprobable for a given position. Let us assume in this case that the nucleotides are equiprobable and hence we have the Expected Frequencies (E) of the nucleotides as

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

Observed Frequencies (O) of the nucleotides at position 1 is (assuming pseudocounts):

O (A) = O (T) = 0.1

O (C) = 0.9

O (G) = 0.3

Therefore, log-likelihood ratios for the nucleotides A, C, G, and T at position 1 are:

A - Log (0.0714/0.25) = -1.808

C - Log (0.6428/0.25) = 1.362

G - Log (0.2142/0.25) = -0.222

T - Log (0.0714/0.25) = -1.808

Similarly the log-likelihood ratios for all the other positions are calculated and shown in Figure 5.

| | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| **A** | -1.808 | -1.808 | -1.808 | -1.808 | 0.192 | 1.652 | -0.807 | -0.807 | -1.808 | -1.808 |
| **C** | 1.362 | 1.652 | -1.808 | 0.192 | -1.808 | -1.808 | 1.514 | 1.514 | -1.808 | -1.808 |
| **G** | -0.222 | -1.808 | 1.362 | 1.0 | -1.808 | -1.808 | -1.808 | -1.808 | 1.652 | 1.652 |
| **T** | -1.808 | -1.808 | -0.222 | -0.807 | 1.192 | -1.808 | -1.808 | -1.808 | -1.808 | -1.808 |

Figure 5: PWM showing Log-likelihood Ratios

## 5.2 Scoring the PWM

The score for the nucleotide sequence *CCGCTACAGG* (assuming a log base value of 2) can be computed as:

Score = 1.362 + 1.652 + 1.362 + 0.192 + 1.192 + 1.652 + 1.514 + (-0.807) + 1.652 + 1.652 = 11.423.

If this value is above the threshold that we choose for our test, then the sequence is considered a match for the given set of binding site sequences.

## 5.3 Logo Representation and Information Content

The motif in Example 1 can be represented in the form of a logo where the x-axis represents the binding site positions and the y-axis indicates the information content. A logo is a visual representation of the motif where the size or height of the base corresponds to the base's relative frequency at that position. The total height of each column is proportional to its information content [4].

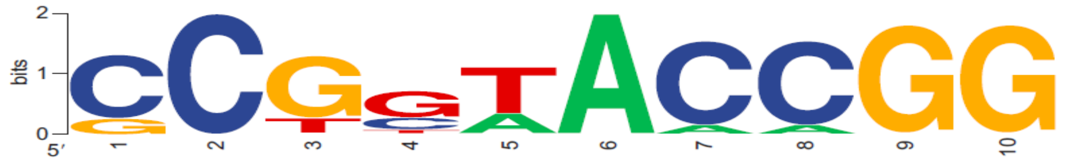


Figure 6: Sequence Logo Representation of the Motif

We can see in Figure 6 [4] that at position 2, nucleotide C is highly conserved across the entire given binding site sequences and hence, in the logo representation, the height of the character C is 2 bits. At position 1, C occurs 8 out of 10 times while nucleotide G occurs 2 out of 10 times. The height of the characters in the character stack at position 1 reflects this information, which can be quantified using the Information Content measured in bits. It is given by [4]:

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

where $I_j$ is the Information Content at position $j$ and $f_{x,\,j}$ is the relative frequency of nucleotide $x$ at position $j$. Notice that the information content achieves its highest value of 2 bits when a particular nucleotide appears at position $j$ in all the sequences and $I_j$ has the lowest value when all nucleotides are equally likely to appear ($f_{x,\,j}$ is 0.25) at position $j$.

In Figure 6, positions 2, 9, and 10 have a highly conserved C, G, and G, respectively. It is highly likely that these positions will be represented by the same consensus base in a different binding site for the same TF. This implies that there is no uncertainty at these positions and thus the information content is very high (2 bits in the sequence logo). On the other hand, at position 4, nucleotide G occurs most of the time but is closely followed by C and T. We can therefore say that, at position 4, there is high uncertainty as to which base would occur and consequently, the information content at position 4 is very low.

## 5.4 PWM Limitations

Although PWMs are simple, popular, and effective, they have certain limitations. We illustrate PWM's drawbacks in Example 2.

**Example 2**

Let us consider the set of binding sites in Figure 7 [4]:

| | | | | |
|---|---|---|---|---|
| $X_1$ | $C$ | $G$ | $G$ | $G$ |
| $X_2$ | $C$ | $G$ | $T$ | $G$ |
| $X_3$ | $C$ | $G$ | $G$ | $C$ |
| $X_4$ | $A$ | $T$ | $G$ | $G$ |
| $X_5$ | $A$ | $T$ | $G$ | $G$ |
| $X_6$ | $A$ | $T$ | $G$ | $T$ |

Figure 7: Sequences of a Hypothetical Binding Site

We can see that, at position 1, nucleotides C and A are equally likely and similarly at position 2, nucleotides G and T are equally likely. Based on these observations, we can infer that both sequences CGGG and CTGG are equally preferred. But, on careful observation, we can see that only when C occurs at position 1, we find a G at position 2. For nucleotide T to occur in position 2, the sequence should have an A at position 1. Thus we can infer that positions 1 and 2 are dependent on each other. This dependency is not captured by a PWM as it assumes independence among neighboring nucleotides. In order to address this limitation, higher-order PWMs or Position Weight Arrays are used.

In the upcoming chapter, higher-order PWMs and Maximal Dependence Decomposition (MDD) approach are explained in detail.

# CHAPTER 6

# Higher-order PWM and MDD

The limitations of a normal PWM can be addressed by using a Position Weight Array or a higher-order PWM. In the hypothetical example considered in Example 2, we know that there is a dependency between the first and the second positions in the binding sites. This dependency can be captured effectively by considering dinucleotide frequencies instead of mononucleotides. In our example, CG and AT are the most likely dinucleotides at the first two positions and if we want to incorporate possible dependencies between such nucleotides at every pair of adjacent positions, we can extend the single nucleotide PWM with *4* rows and *K* columns to a dinucleotide PWM with 16 rows (corresponding to all dinucleotide combinations) and *K-1* columns corresponding to all dinucleotide positions. This can be further expanded to form even higher-order (with trinucleotides or more) PWM.

Higher-order PWMs, though seem to identify the dependencies among nucleotides in adjacent positions of the binding sites, they fail to capture the dependencies among non-adjacent nucleotides. There is also the problem of insufficient dinucleotides (insufficient number of sequences considered) which makes the model unreliable. These limitations can be overcome in the "Maximal Dependence Decomposition" (MDD) approach, which "explicitly estimates the extent to which the nucleotide at position *j* depends on the nucleotide at position *i*. [4]" Here, the model specifically determines the extent to which a nucleotide at position *j* depends on a nucleotide at position *i*, based on whether the nucleotide at position *i* is the consensus or the non-consensus nucleotide at that position. Thus, dependencies between a consensus indicator $C_i$ and a nucleotide $X_j$ are determined instead of the $X_i$ vs $X_j$ comparisons.

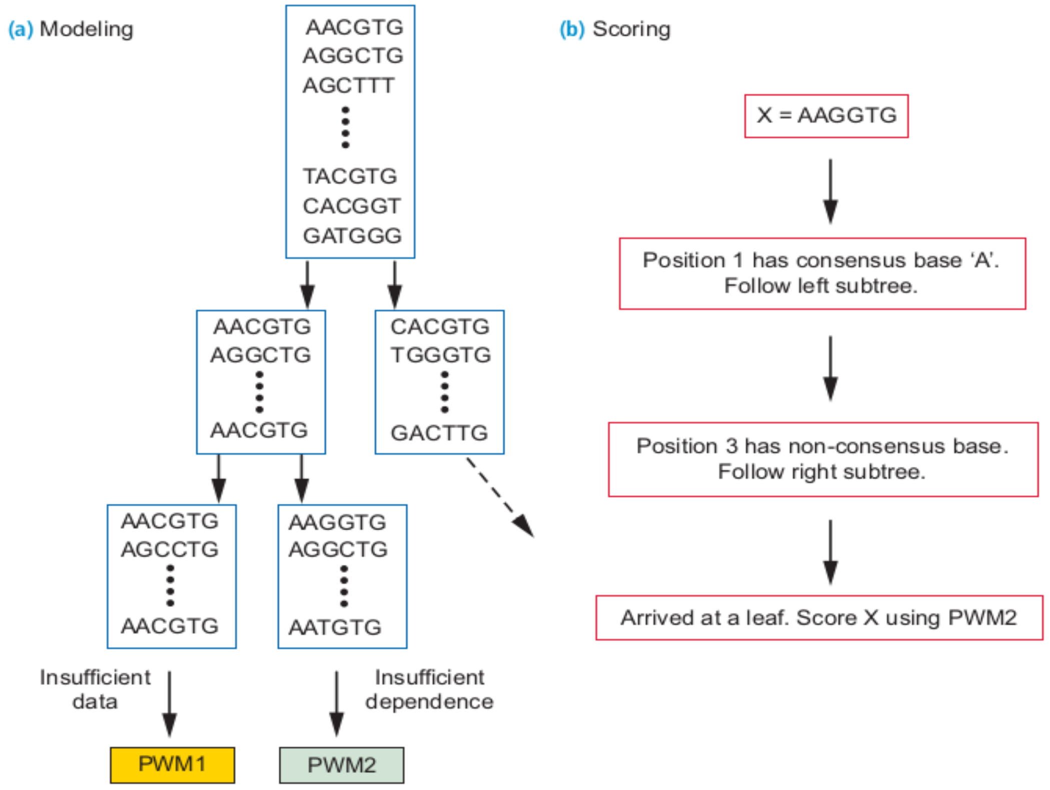


Figure 8: Maximal Dependence Decomposition Modeling and Scoring

Figure 8 [4] shows the MDD procedure adapted to find the dependencies among non-adjacent nucleotide position in a given binding site. As can be seen from the figure, a set of sequences are initially divided into 2 groups, $C_i$ and $\overline{C}_i$, based on whether $C_i$ is the consensus nucleotide at position $i$ or not. If one of the groups has fewer sequences than a pre-set number, then we ignore the group and do not divide it further. In Figure 8(a), it is determined that initially there is high dependence on the nucleotide A at position 1 and hence the sequences are partitioned based on whether nucleotide A is the consensus base at position 1 or not. Group $C_i$ is further divided into two groups based on whether nucleotide C is the consensus base at position 3 or not as maximal dependence is observed for nucleotide C at position 3. The resulting groups are divided again and again until the number of remaining sequences in the group falls beneath some pre-set number or there is insufficient dependency. In either case, the construction of the tree is halted and a leaf is created as can be seen in Figure 8(a). The PWMs are then computed for the leaf nodes, which are then used to score a new sequence, not originally part of the training set.

The maximal dependence over a nucleotide at a particular position by the nucleotides at other positions in the binding site is determined using chi-square ($\chi^2$) statistics. If the $\chi^2$ value is significantly different among the two groups ($C_i$ and $\overline{C}_{i}$), then we can say that there is a definite dependence on the consensus nucleotide based on which the groups are divided.

As shown in Figure 8(b), once the model is built, we can determine whether a given sequence conforms to the model or not by traversing through the MDD tree until a leaf node is reached at which point, we just compute the score for the sequence with the PWM of that leaf node. As usual, a score above the threshold indicates a match and a score below the threshold indicates a non-match.

## 6.1 The MDD Algorithm

In Example 3, we show how to use MDD with the sequences of Figure 9. As explained earlier, we need to compute the $\chi^2$ distribution for the given set in order to determine the dependency of a nucleotide at a particular position over the nucleotides at other positions in the binding site. $\chi^2$ can be represented in simple terms as:

$$\chi^2 = (\text{Observed Frequency} - \text{Expected Frequency})^2 / \text{Expected Frequency}$$

| | |
|---|---|
| X1 | C G G G |
| X2 | C G T G |
| X3 | C G G C |
| X4 | A T G G |
| X5 | A T G G |
| X6 | A T G T |
| X7 | C G G G |

Figure 9: Sequences of a Hypothetical Binding Site

We used the approach mentioned in [4] to formulate the MDD tree. The steps are as follows:

1. Compute $S_i = \sum_{j \neq i} \chi^2 (j, i)$ to capture the total dependence on position $i$.
2. Among all $K$ positions, select position $i$ with the maximum value of $S_i$, and partition all sequences into two parts based on whether they have $C_i$ or $\overline{C}_i$ at position $i$.
3. Repeat steps 1 and 2 separately for each of the two sets of sequences obtained in step 2.
4. Stop if there is no significant dependence, or if there is an insufficient number of sequences in the current subset. In either case, construct a standard PWM for the remaining subset of sequences.

**Example 3**

Let us consider the set of sequences in Figure 9 and apply the MDD approach. We use Table 1 to determine the dependencies among the nucleotide positions. Position *i* represents the various positions in the binding site and the consensus column indicates the consensus nucleotide determined at that particular position. We need to fill in the table with the $\chi^2$ values that we obtain from calculating the dependencies of position *j* on position *i*. Computing the final column "Sum" which simply calculates the sum of all the $\chi^2$ values at position *i,* is the first step in the MDD approach. Table 1 which is initially empty will be filled as we iterate through the MDD steps.

Table 1: $\chi^2$ Distribution Table

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

Case 1: Position $i = 1, j = 2$

At position 1, nucleotide C is the consensus base and hence we need to divide the binding site sequences into two groups $C_i$ and $\overline{C}_i$ where $C_i$ contains all the sequences that have the consensus base C at position 1 and $\overline{C}_i$ contains the remaining sequences that do not have a C at position 1, as shown in Table 2.

Table 2: Initial partition of the sequences

| $C_i$ | $\overline{C}_\mathrm{i}$ |
|---|---|
| CGGG | ATGG |
| CGTG | ATGG |
| CGGC | ATGG |
| CGGG | |

Next we compute the $\chi^2$ statistic as proposed in [4]:

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

where,

$N$ is the total number of sequences in $C_i$.

$f_A, f_C, f_G, f_T$ are the normalized frequencies (number of each base divided by the total number of sequences) of the four bases at position $j$ among the sequences in $\overline{C}_i$.

$N_A$, $N_C$, $N_G$, $N_T$ are the observed number of the four bases at position $j$ among the sequences in $C_i$.

Using this formula for Case 1, we get:

$f_A$ = 4/7 [Total number of sequences in $\overline{C}_i$ is 3 but to account for insufficient data, pseudocounts are introduced and hence the total is 7]

$f_C$ = 1/7 $f_G$ = 1/7 $f_T$ = 4/7 $N$ = 8

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

$\chi^2 = [(8 * 0.1428 - 1)^2 / (8*0.1428)] + [(8 * 0.1428 - 1)^2 / (8*0.1428)] + [(8 * 0.1428 - 5)^2 / (8*0.1428)] + [(8 * 0.5714 - 1)^2 / (8*0.5714)]$

Hence $\chi^2 = 15.8514$.

Similarly, we have computed the $\chi^2$ values for other positions as shown in Table 3.

Table 3: $\chi^2$ statistic showing dependencies of various positions

| Position *i* | Consensus | Position *j* | | | | Sum |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | |
| 1 | C | - | 15.8514 | 0.75067 | 1.4801 | 18.0821 |
| 2 | G | 15.8517 | - | 0.75067 | 1.4801 | **18.0824** |
| 3 | G | 3 | 3 | - | 0.75 | 6.75 |
| 4 | G | 0.6662 | 0.6662 | 0.5553 | - | 1.8877 |

From Table 3, we can see that the $\chi^2$ value is the highest when *i = 1, j = 2 and i = 2, j = 1*. This indicates that there is some dependency between positions 1 and 2 in the binding site sequences.

The highest value for the sum (last column of Table 3) is 18.0824 at position 2. This indicates that we need to partition the given set at position 2 where group $C_i$ will contain the sequences that have the consensus nucleotide G at position 2 and group $\overline{C_i}$ will contain the remaining sequences (that do not have nucleotide G at position 2).

The binding site sequences are partitioned as shown in Table 4.

Table 4: Table formed after dividing the sequences based on position 2

| $C_i$ | $\overline{C}_i$ |
|---|---|
| CGGG | ATGG |
| CGTG | ATGG |
| CGGC | ATGG |
| CGGG | |

The $\chi^2$ statistic is again computed as done previously and the results are tabulated in Table 5.

Table 5: $\chi^2$ distribution after partitioning the sequences based on position 2

| Position *i* | Consensus | Position *j* | | | Sum |
|---|---|---|---|---|---|
| | | 1 | 3 | 4 | |
| 1 | C | - | 3 | 3 | 6 |
| 3 | G | 0.8571 | - | 0.4999 | 1.3571 |
| 4 | G | 0.8571 | 0.4999 | - | 1.3571 |

As $S_1$ has the highest value, we need to partition the sequences at position 1 into two groups, $C_i$ containing sequences with consensus base C at position 1 and $\overline{C}_i$ containing the remaining sequences. See Table 6 for the partitioned sequences.

Table 6: Table formed after dividing the sequences based on position 1

| $C_i$ | $\overline{C}_i$ |
|---|---|
| CGGG | ATGG |
| CGTG | ATGG |
| CGGC | ATGG |
| CGGG | |

The $\chi^2$ statistic is again computed and the results are shown in Table 7.

Table 7: $\chi^2$ distribution after partitioning the sequences based on position 1

| Position $i$ | Consensus | Position $j$ | | Sum |
|---|---|---|---|---|
| | | 3 | 4 | |
| 3 | G | - | 0.4999 | 0.4999 |
| 4 | G | 0.4999 | - | 0.4999 |

We can now see that, there is no significant difference in the $\chi^2$ value and there is insufficient data. At this point, we stop partitioning the sequences further and build the individual PWMs. Any new sequence that is not part of the initial training set is then scored using these PWMs. If the final score is above a predefined threshold value, then we consider the sequence to be a match for that binding site.

Figure 10 depicts the final MDD tree obtained for the sequences after partitioning the sequences on different positions based on their $\chi^2$ values. When a new sequence is given, we can walk down the MDD tree till we reach a leaf node and score the new sequence using the PWM of that leaf node.

In the next chapter, we discuss the implementation details of the PWM, higher-order PWM and, Maximal Dependence Decomposition approaches.

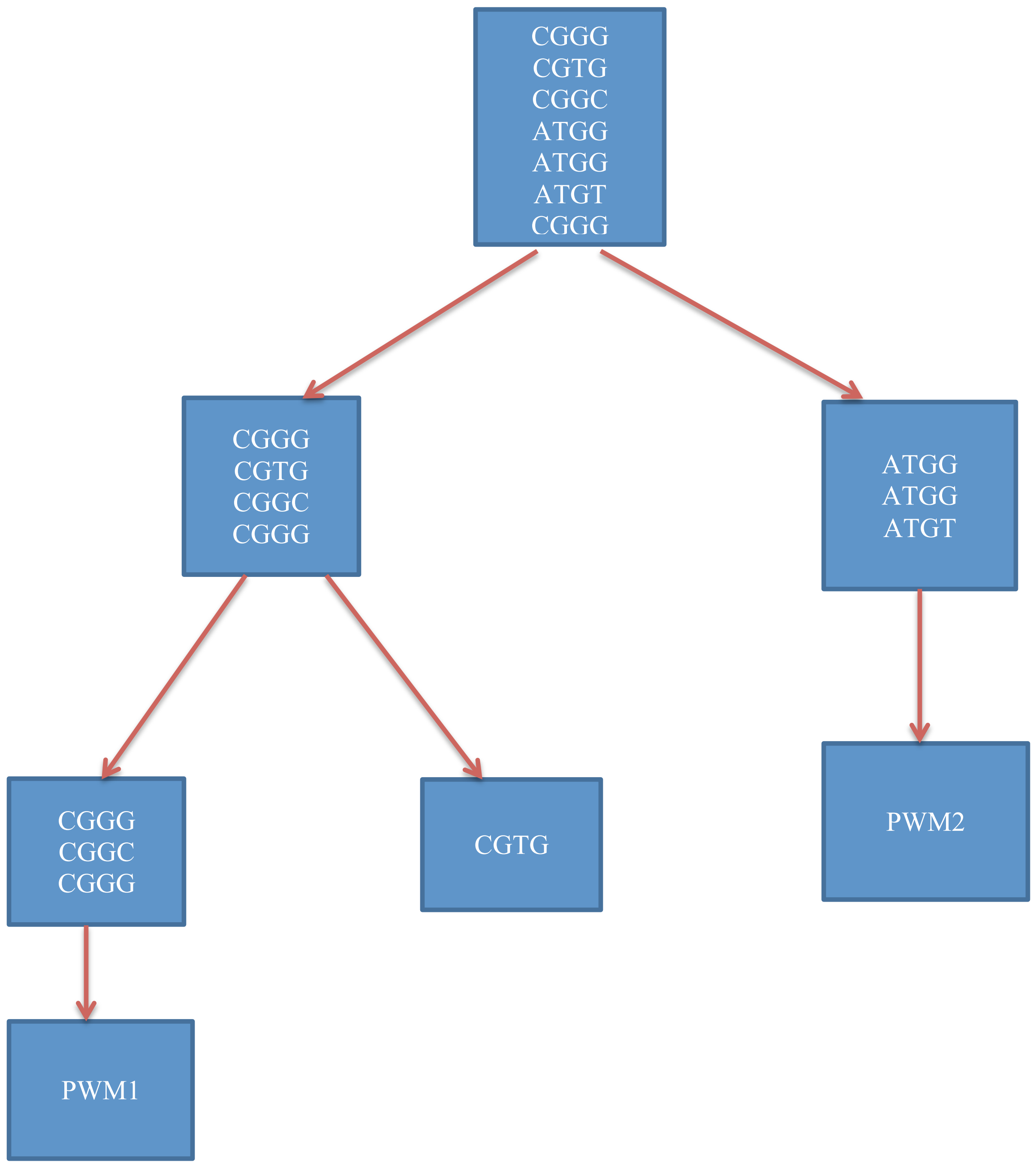


Figure 10: MDD approach applied on the example sequences

# CHAPTER 7

# Implementation

In the previous chapter, we discussed how an MDD is constructed with a simple example. In this chapter we discuss the implementation of MDD and some of the heuristics that were considered to evaluate various sample sets.

## 7.1 PWM Implementation

A simple PWM was implemented in *JAVA* and the source code is attached in the appendix. The PWM program reads the datasets from a specific location on disk and stores them in a memory ArrayList structure, which forms the input for the PWM scoring function. The PWM scoring function calculates the frequency of occurrence of the nucleotides A, C, G, and T in each of the positions. Logarithms of these values are considered and a Laplace prior is introduced to account for pseudocounts as well. The sequences are then scored using the frequency matrix calculated above. Thresholds are determined using the Receiver Operating Characteristic (ROC) curves mentioned in the earlier chapters. A program in *R* is used to generate these ROC curves for the dataset and a threshold is decided. During testing, when a new sequence has to be classified, it is scored with the simple PWM and the score is compared with the threshold. If the score is less than the threshold, the sequence is considered to be a non-match and a score higher than the threshold indicates a match.

## 7.2 Higher-order PWM Implementation

Implementation of higher-order PWM is very similar to the simple PWM implementation except that two adjacent nucleotides are considered for every position instead of one. The overall positions considered run from 1 through *N -1* where *N* is the length of the sequence. The nucleotide pairs AA through TT were stored in a TreeMap for an extremely quick lookup while maintaining the order of the nucleotides within the data structure. Laplace prior and log ratios are included in the higher-order PWM calculation as was done with the simple PWM. The code for the implementation is attached in the appendix. An ROC curve is plotted with the results obtained from the program and compared with the simple PWM. We can see the improvement in the scores while using the higher-order PWM from the simple PWM, but this improvement depends on the dataset under consideration, as will be seen in the next chapter.

## 7.3 MDD Implementation

To build the MDD tree, the MDD algorithm discussed in the previous chapter was implemented. The PWM scores obtained from the simple PWM scoring program above is used as a basis to build the MDD tree. The dependency of one position over all the remaining adjacent and non-adjacent positions are calculated using the chi-square statistics and the nucleotide position with the maximum sum that correlates with the highest dependency position is selected for splitting the tree. All sequences in the node that have the splitting nucleotide at the maximum sum position are grouped into a $C_i$ MDD node and all other sequences are grouped into a new $\overline{C}_i$ MDD node. This process is continued until the stopping criteria are reached.

## 7.4 MDD Stopping Criteria

If at any node, the number of sequences to be considered is less than 15% of the total number of input sequences at that point, the program will terminate the splitting and score that MDD node as a leaf. In addition, the PWM score is computed at every $C_i$ MDD node and the children of a particular $C_i$ MDD node will score the position on which they were split using the parent's PWM. MDD also maintains a p-value table and it stops proceeding if the maximum sum obtained from the chi-square test at any node is less than the expected p-value. The p-value is calculated based on the degrees of freedom. In this case, it is the product of the number of un-split positions at a given node minus 1 and the number of classes (A, C, G, and T) minus 1. The MDD tree construction is also stopped when the sequences are split on *N-1* positions, where *N* is the length of the sequence.

## 7.5 MDD Traversal

After creating the MDD tree using the above program, a new sequence can be scored by traversing the tree. A new sequence will take a MDD $C_i$ path if it matches the splitting nucleotide at the splitting position or MDD $\overline{C_i}$ otherwise, until it reaches a leaf node in the tree. At this point, the new sequence will be scored based on the PWM calculated at this MDD node. We partitioned the datasets into training and test sets and all the test sequences are scored based on the MDD tree obtained from the training sequences. The code for the MDD program is attached in the appendix.

In the next chapter, we discuss in detail the datasets used to train and to test the above implementations and compare the results obtained.

# CHAPTER 8

# DATASETS AND RESULTS

## 8.1 Dataset and Testing

The dataset for this project was gathered from the JASPAR database, which has a collection of experimentally determined binding sites [2]. The TFBS data for yeast was considered because we could obtain both the positive and the negative sequences (those that are not binding sites). Data from JASPAR was parsed and converted to the format required by the program. The dataset is split into true sites and false sites. The PWM, higher-order PWM and MDD models are trained and tested using the ten-fold cross validation method.

### Ten-Fold Cross Validation

One round of cross-validation involves a sample of data into complementary subsets, performing the analysis on one subset (called the training set), and validating the analysis on the other subset (called the validation set or testing set). To reduce variability, multiple rounds of cross-validation are performed using different partitions, and the validation results are averaged over the rounds [9].

For each round of the ten-fold validation, the positive and the negative scores are obtained and are input to the ROC program, which generates the resulting ROC curve. The average of the ten AUC values is considered to be the final value.

We now describe the characteristics of the yeast dataset:

**Yeast Dataset**

Type: Transcription Factor Binding Sites

Length of the sequences: 14

True Sites: 137

False Sites: 10000

Training Set: 123

Testing Set – True: 14

Testing Set – False: 50

**Ten-Fold Cross Validation Results with the AUC values**

| | PWM | Higher-order PWM | MDD Approach |
|---|---|---|---|
| **Test 1** | 1 | 1 | 1 |
| **Test 2** | 1 | 1 | 1 |
| **Test 3** | 1 | 1 | 1 |
| **Test 4** | 1 | 1 | 1 |
| **Test 5** | 1 | 1 | 1 |
| **Test 6** | 1 | 1 | 1 |
| **Test 7** | 1 | 1 | 1 |
| **Test 8** | 1 | 1 | 1 |
| **Test 9** | 1 | 1 | 1 |
| **Test 10** | 1 | 1 | 1 |
| **Average** | **1** | **1** | **1** |

**PWM Results**

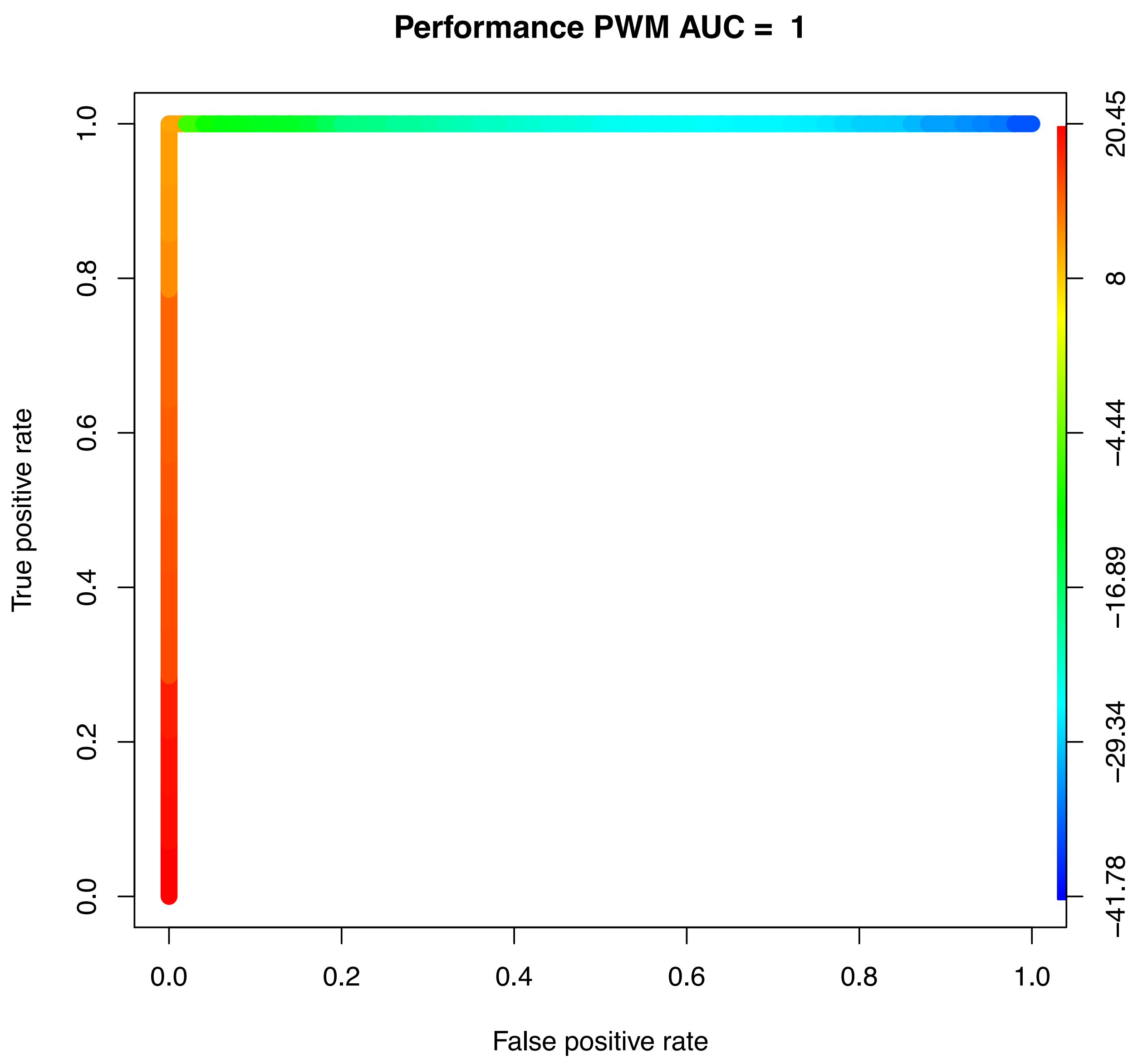


Figure 11: ROC Curve of a Simple PWM for Yeast Dataset

**Higher-order (Dinucleotide) PWM Results**

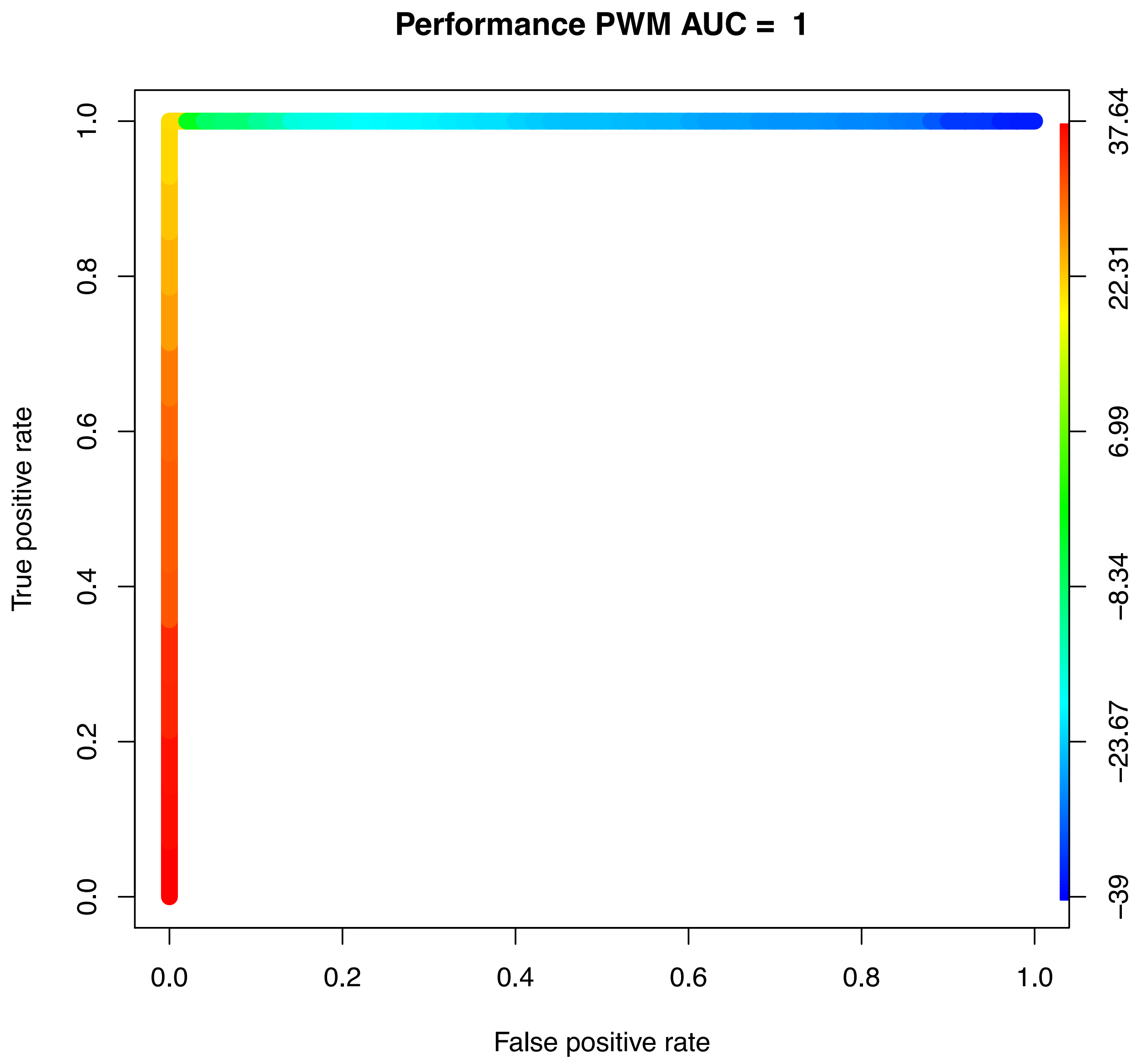


Figure 12: ROC Curve of Dinucleotide PWM for Yeast Dataset

## Maximal Dependence Decomposition Results

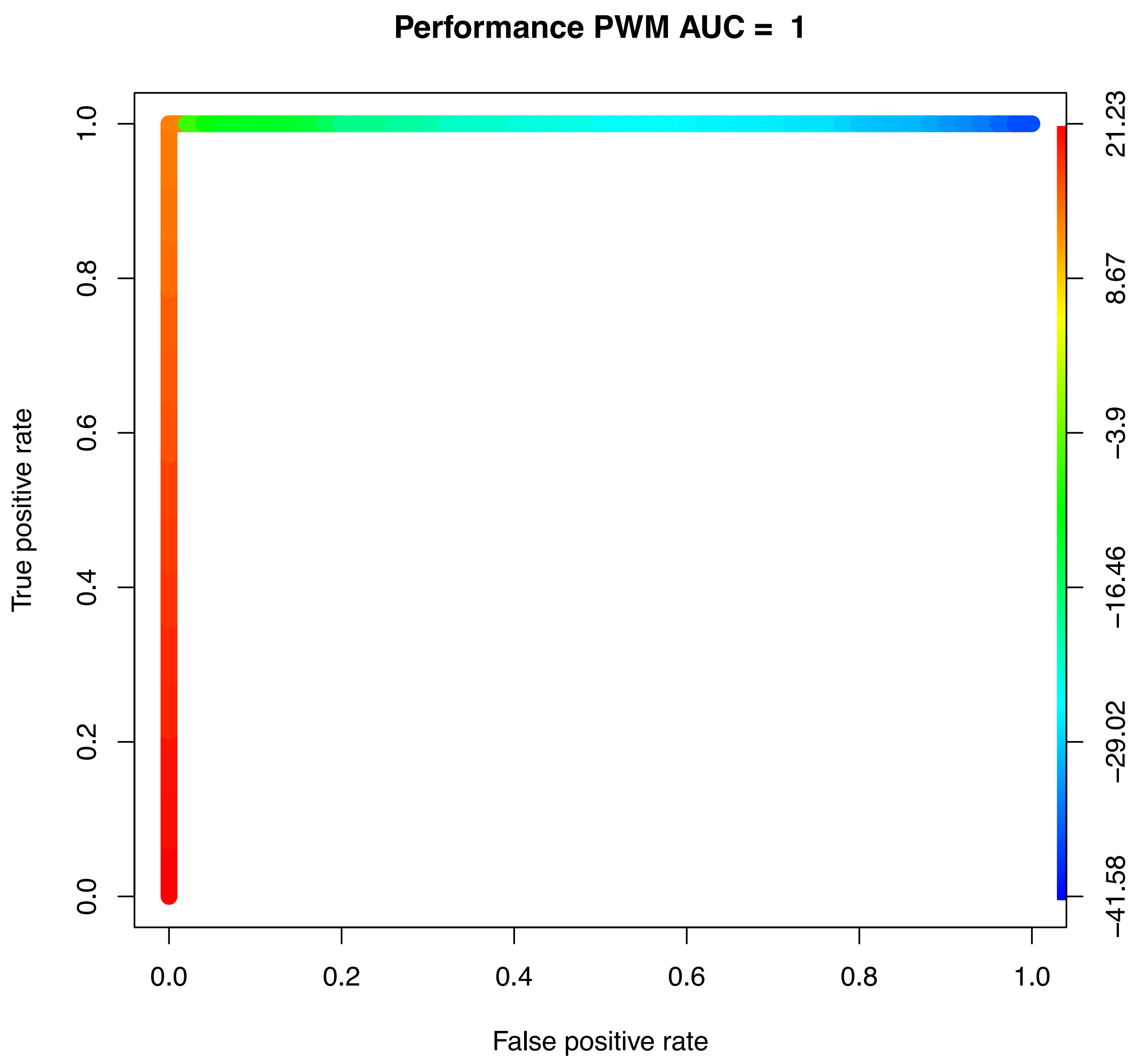
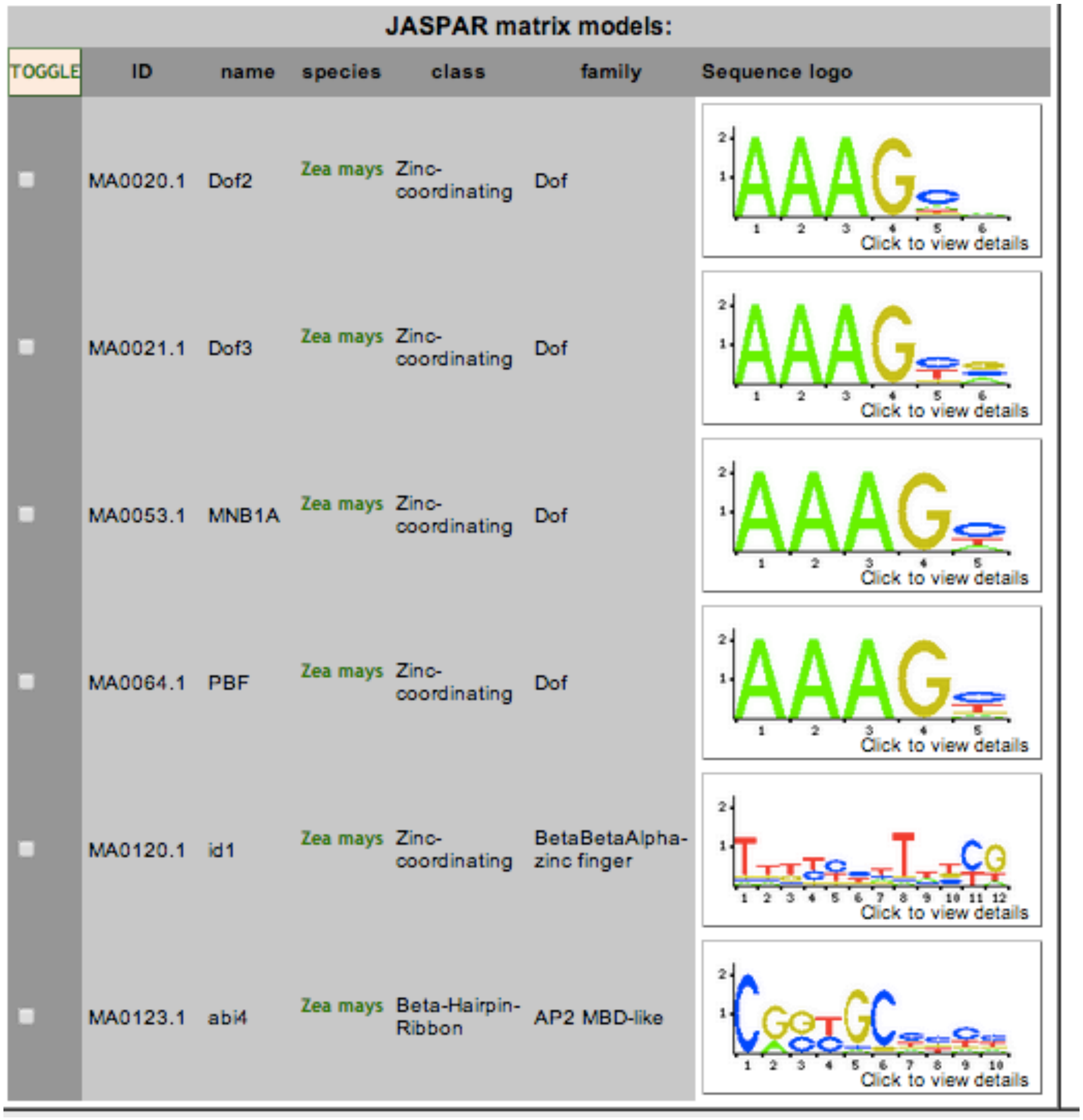


Figure 13: ROC Curve of MDD for Yeast Dataset

The ROC curves are color coded with the legend described on the right side border of the curve. The ROC curve in Figure 13 has a false positive value of -41.58 and a true positive value of 21.23. Depending on whether we want high sensitivity or high specificity, we can choose the threshold accordingly. We see that in the case of the Yeast dataset, all the three models seem to be performing very well. This is indicated by the AUC (Area under the curve) value of 1.

## 8.2 MDD Clustering

As mentioned in the previous section, the MDD approach could be used for predicting TFBSs. It is also interesting to note that it could also be used to cluster binding sites. This method clusters all sequences into subgroups that have statistically significant motifs [10]. As part of this experiment, different Transcription Factors (TFs) of the species *Zea mays* (Maize) were obtained from the Jaspar database [2]. These individual TFBSs were consolidated into one single dataset and were input to the MDD program. Figure 14 shows a screenshot obtained from Jaspar with the details of the different TFs of *Zea mays* and as can be seen from Figure 15; the MDD program correctly grouped these sequences into appropriate clusters. The sequence logos were generated for the groups of sequences using WebLogos [11].



**JASPAR matrix models:**

| TOGGLE | ID | name | species | class | family | Sequence logo |
|---|---|---|---|---|---|---|
| ☐ | MA0020.1 | Dof2 | *Zea mays* | Zinc-coordinating | Dof | Click to view details |
| ☐ | MA0021.1 | Dof3 | *Zea mays* | Zinc-coordinating | Dof | Click to view details |
| ☐ | MA0053.1 | MNB1A | *Zea mays* | Zinc-coordinating | Dof | Click to view details |
| ☐ | MA0064.1 | PBF | *Zea mays* | Zinc-coordinating | Dof | Click to view details |
| ☐ | MA0120.1 | id1 | *Zea mays* | Zinc-coordinating | BetaBetaAlpha-zinc finger | Click to view details |
| ☐ | MA0123.1 | abi4 | *Zea mays* | Beta-Hairpin-Ribbon | AP2 MBD-like | Click to view details |

Figure 14: Screenshot from Jaspar showing the TFs of *Zea mays*

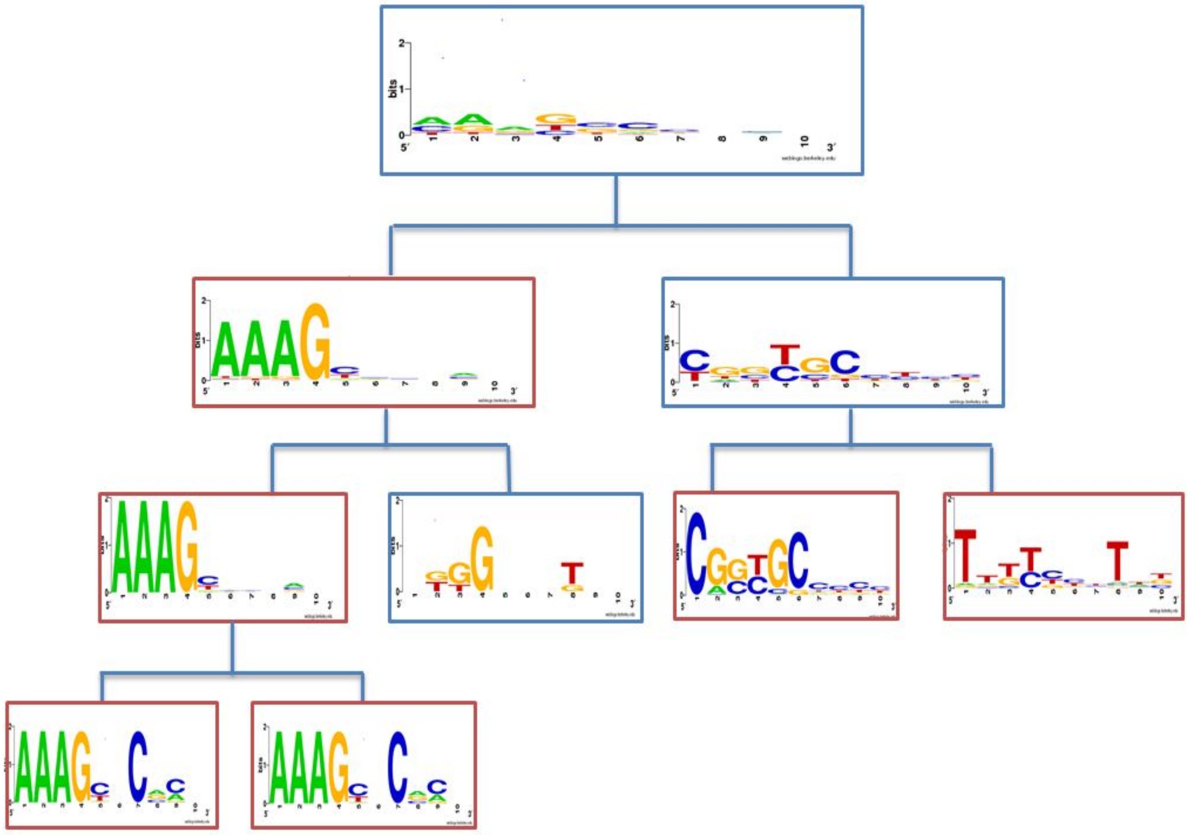


Figure 15: The MDD sequence logo tree showing the clustering of the TFs of *Zea mays*

## 8.3 Extensions

These prediction models are not specific to just TFBSs. These can be applied to the prediction of several other biological motifs. We tested our models on the Splice Site data obtained from [12], analyzed our results and performed a comparative study. Ten-Fold cross validation was used to test the dataset and the results were averaged.

We now describe the characteristics of the splice site dataset:

**Splice Sites**

Type: Splice Sites – Exon-Intron

Length of the sequences: 9

True Sites: 2796

False Sites: 10000

Training Set: 2516

Testing Set – True: 280

Testing Set – False: 900

**Ten-Fold Cross Validation Results with the AUC values**

| | **PWM** | **Higher-order PWM** | **MDD Approach** |
|---|---|---|---|
| **Test 1** | 0.9562 | 0.9614 | 0.9609 |
| **Test 2** | 0.9537 | 0.958 | 0.9622 |
| **Test 3** | 0.9609 | 0.9651 | 0.9682 |
| **Test 4** | 0.9593 | 0.9623 | 0.9602 |
| **Test 5** | 0.9511 | 0.9557 | 0.9639 |
| **Test 6** | 0.9488 | 0.9546 | 0.9585 |
| **Test 7** | 0.9646 | 0.9655 | 0.9635 |
| **Test 8** | 0.9664 | 0.968 | 0.9702 |
| **Test 9** | 0.9553 | 0.9572 | 0.9566 |
| **Test 10** | 0.9633 | 0.9673 | 0.9625 |
| **Average** | **0.9579** | **0.9615** | **0.9626** |

**PWM Results**

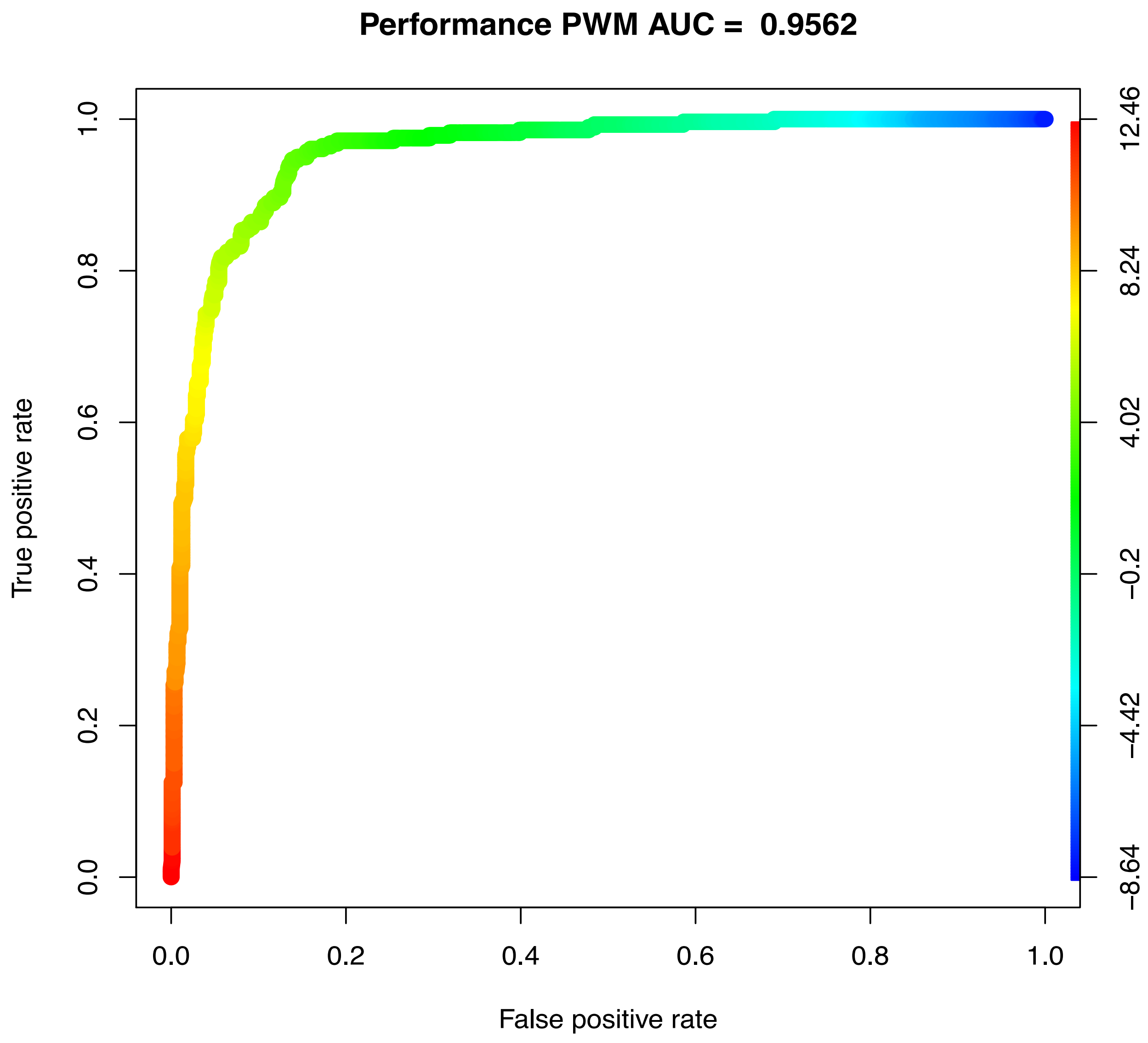


Figure 16: ROC Curve of a Simple PWM for Splice Sites (Exon-Intron)

## Higher-order (Dinucleotide) PWM Results

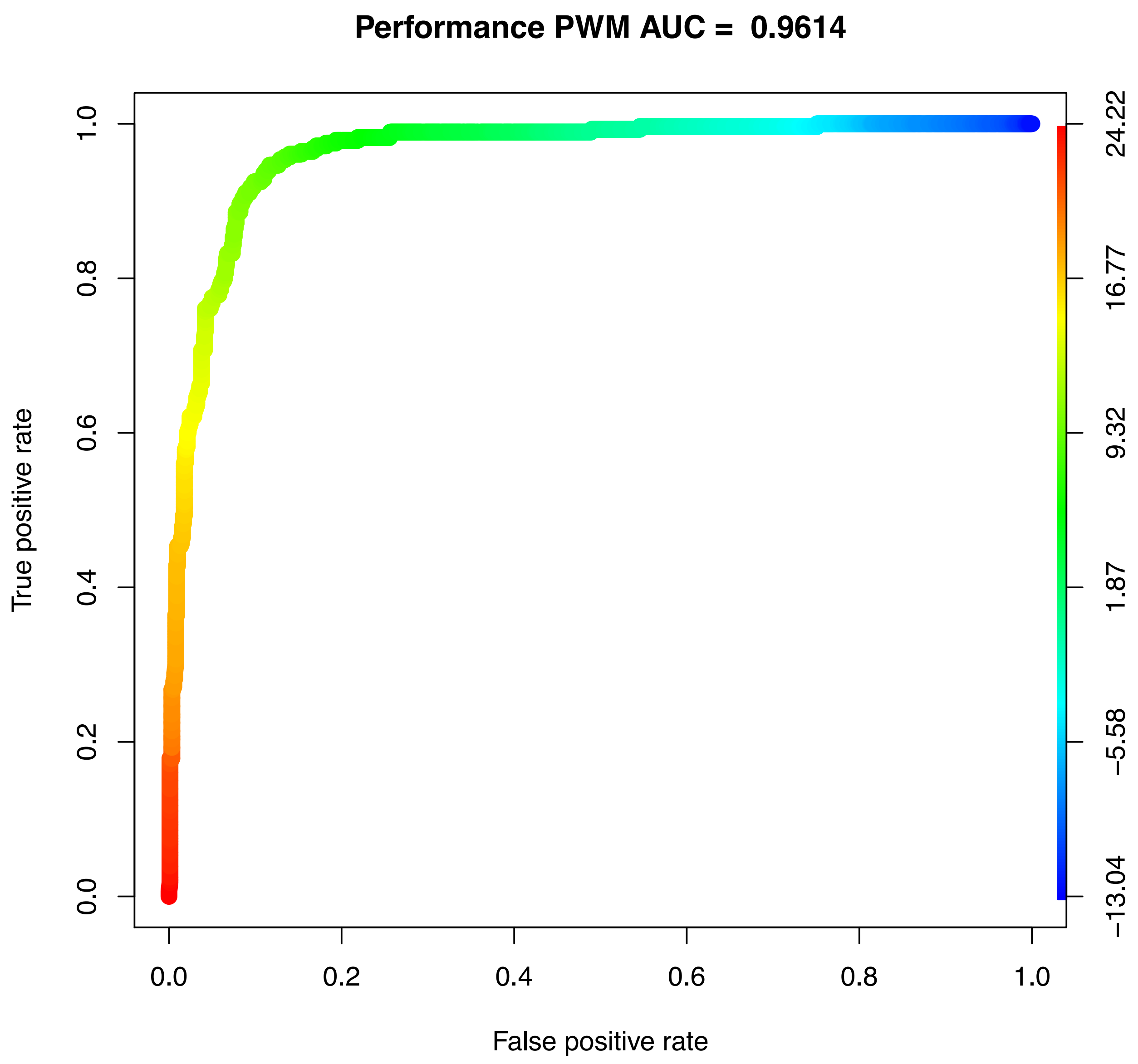


Figure 17: ROC Curve of Dinucleotide PWM for Splice Sites (Exon-Intron)

## Maximal Dependence Decomposition Results

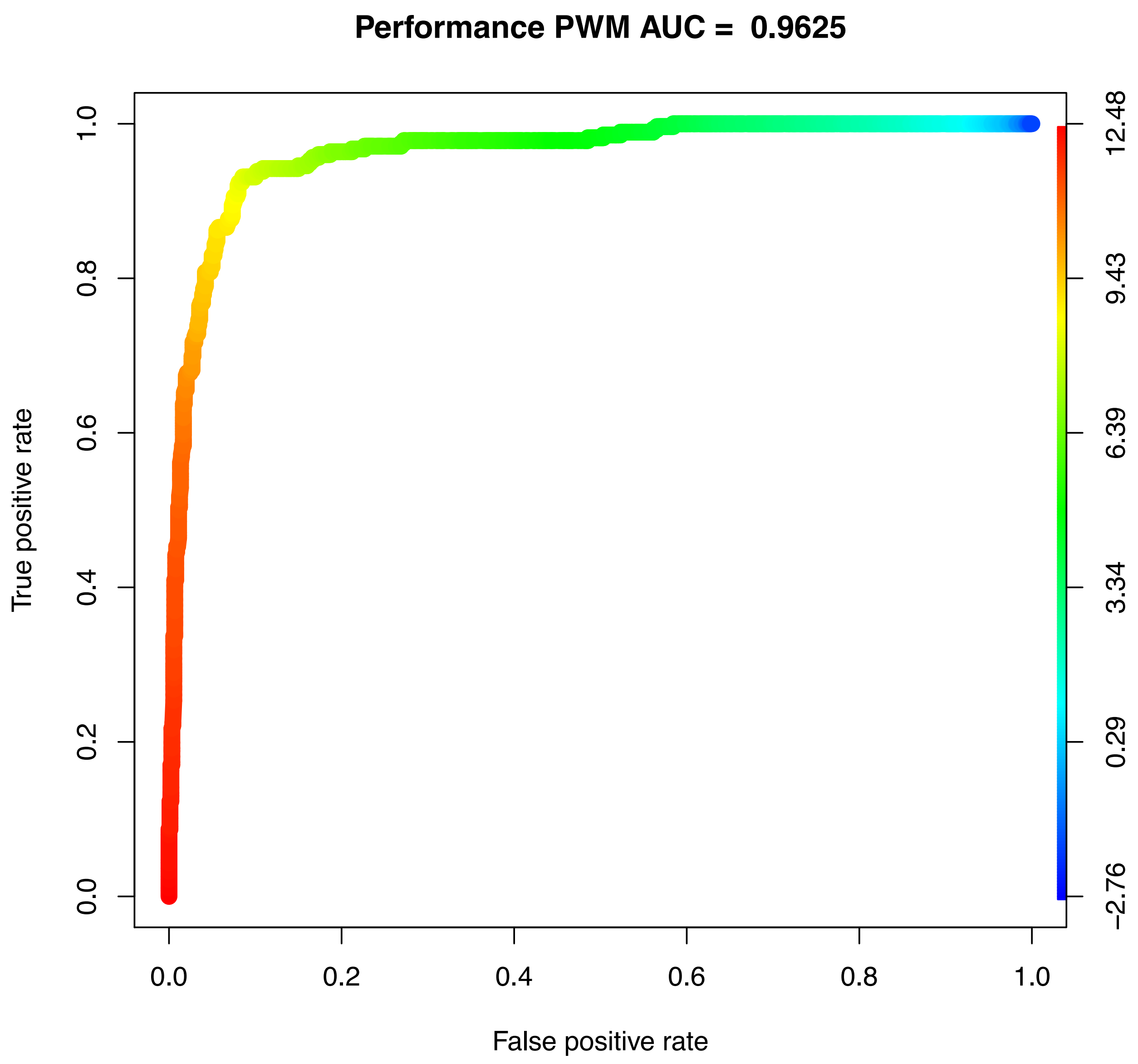


Figure 18: ROC Curve of MDD for Splice Sites (Exon-Intron)

**Type: Splice Sites – Intron-Exon**

Length of the sequences: 14

True Sites: 2880

False Sites: 90915

Training Set: 2592

Testing Set – True: 288

Testing Set – False: 900

**Ten-Fold Cross Validation Results with the AUC values**

| | PWM | Higher-order PWM | MDD Approach |
|---|---|---|---|
| **Test 1** | 0.9224 | 0.9276 | 0.9231 |
| **Test 2** | 0.9347 | 0.938 | 0.9405 |
| **Test 3** | 0.9468 | 0.9482 | 0.9461 |
| **Test 4** | 0.9370 | 0.9400 | 0.9393 |
| **Test 5** | 0.9455 | 0.9468 | 0.9394 |
| **Test 6** | 0.9328 | 0.9358 | 0.9315 |
| **Test 7** | 0.9409 | 0.9410 | 0.9404 |
| **Test 8** | 0.9237 | 0.9269 | 0.9234 |
| **Test 9** | 0.9317 | 0.9341 | 0.9331 |
| **Test 10** | 0.9407 | 0.9423 | 0.9382 |
| **Average** | **0.9356** | **0.9380** | **0.9355** |

**PWM Results**

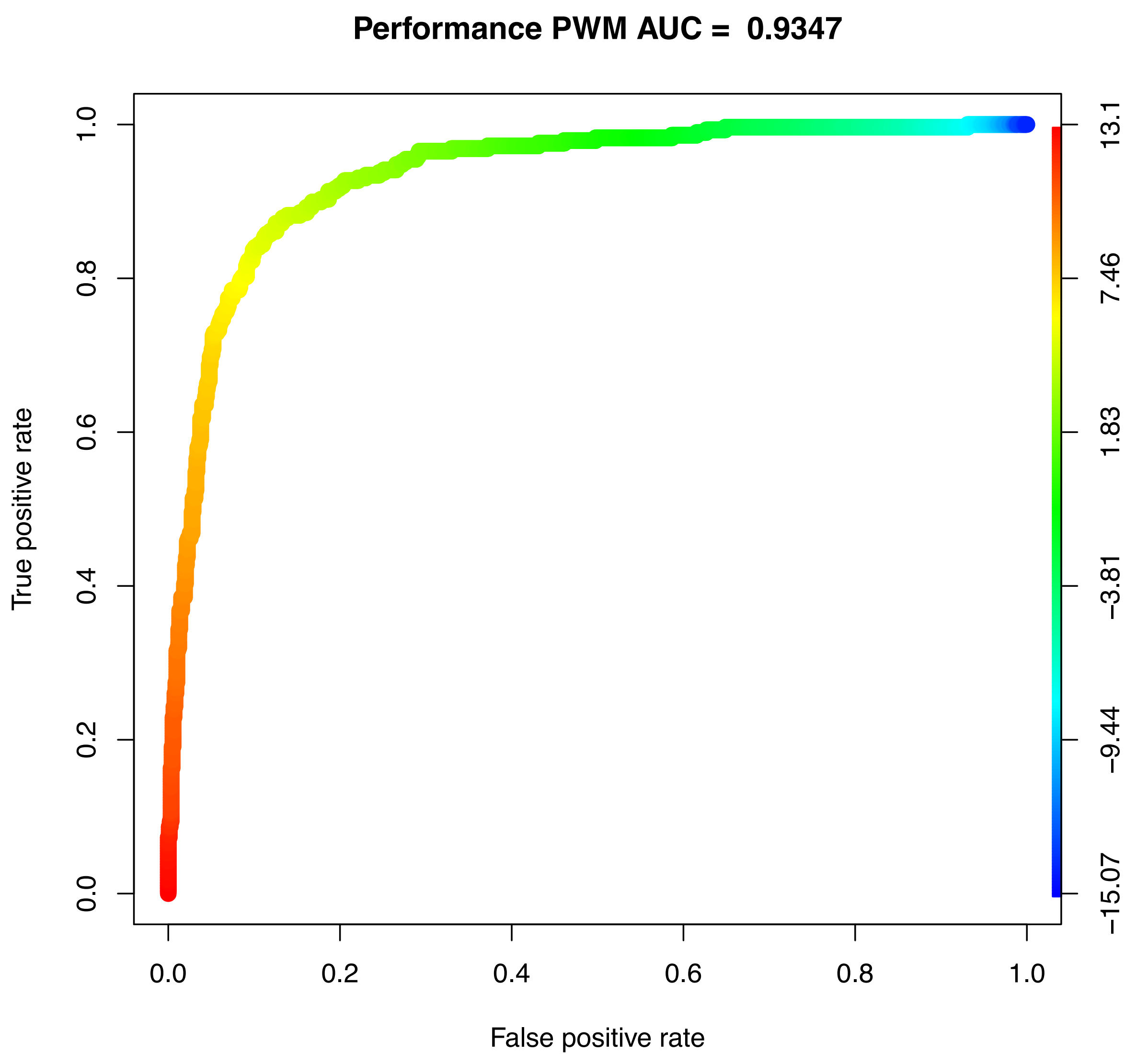


Figure 19: ROC Curve of a Simple PWM for Splice Sites (Intron-Exon)

**Higher-order (Dinucleotide) PWM Results**

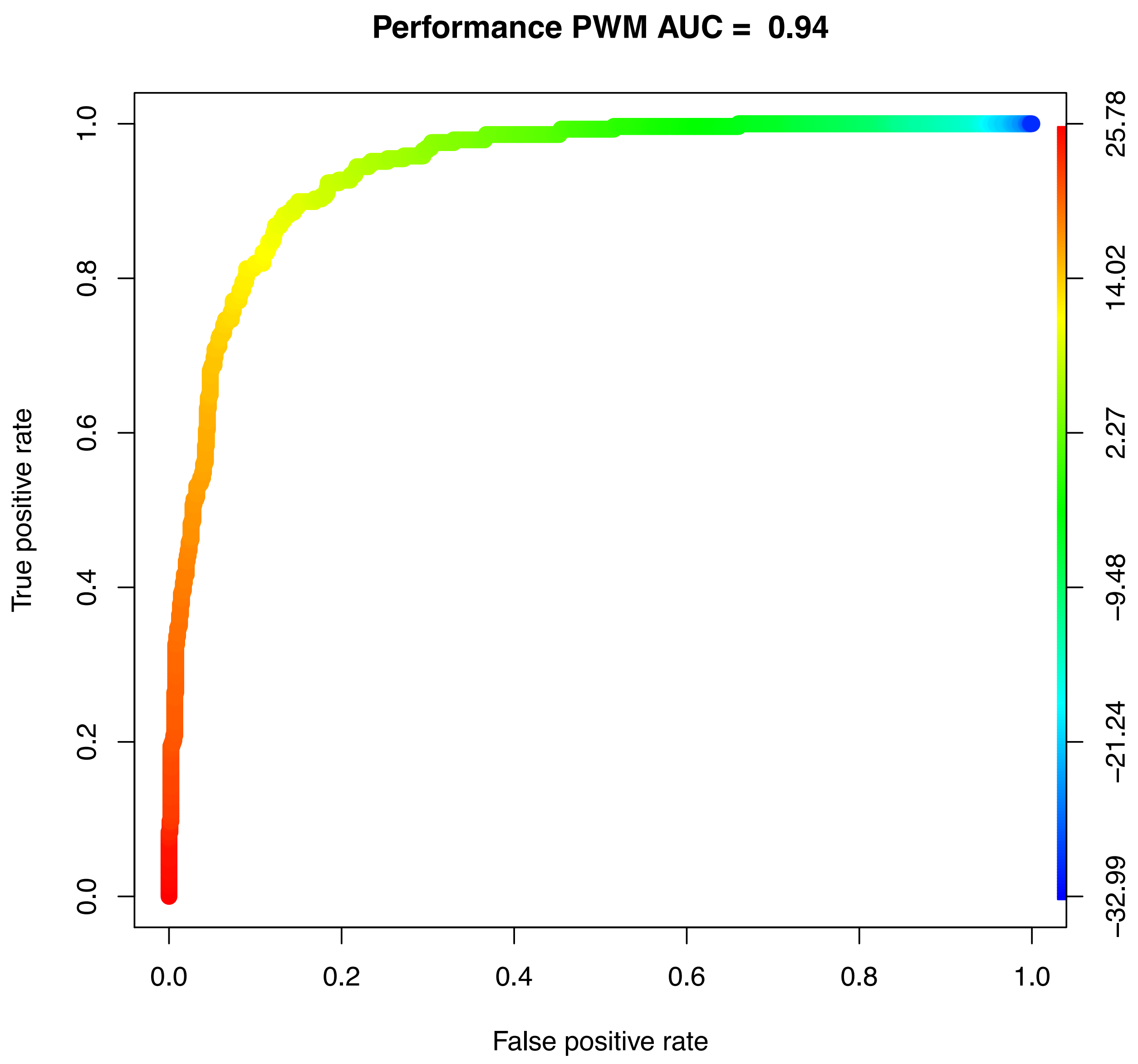


Figure 20: ROC Curve of Dinucleotide PWM for Splice Sites (Intron-Exon)

**Maximal Dependence Decomposition Results**

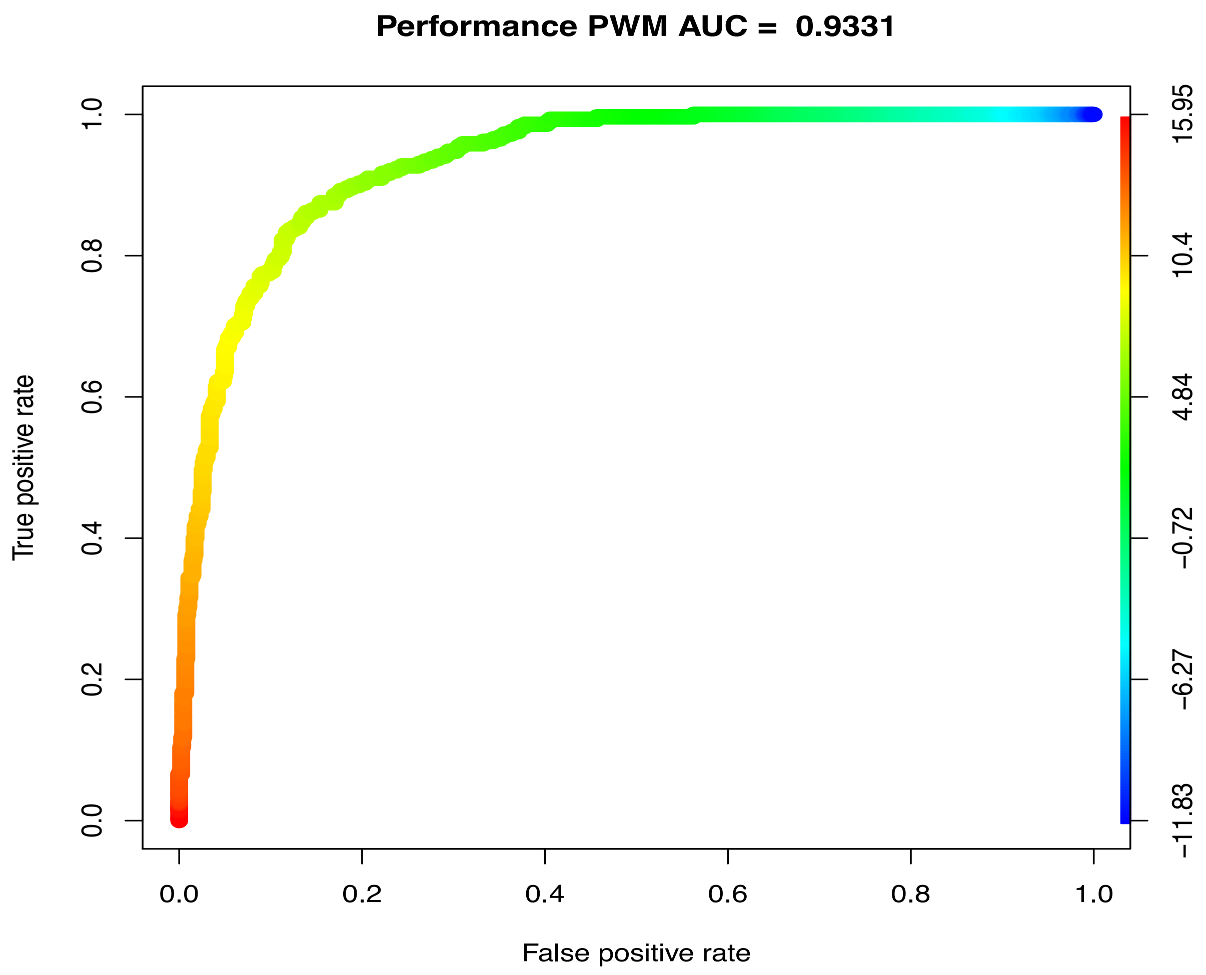


Figure 21: ROC Curve of MDD for Splice Sites (Intron-Exon)

## 8.4 Comparative Analysis

Table 8: Table showing the relative performance of the three models over the three datasets tested

| | PWM | Dinucleotide PWM | MDD |
|---|---|---|---|
| **Yeast Data** | 1 | 1 | 1 |
| **SpliceSet: Exon-Intron** | 0.9579 | 0.9615 | **0.9627** |
| **SpliceSet: Intron-Exon** | 0.9356 | **0.9380** | 0.9355 |

Table 8 summarizes the results of the three models with the three different datasets. We can see that for the yeast dataset, all the models seem to be performing equally well. For the Splice Set (Exon-Intron), the MDD approach seems to perform better than the other two and for the Splice Set (Intron-Exon), the dinucleotide PWM seems to outdo the other models. A small difference in the Area Under the Curve (AUC) could imply a significant improvement in the accuracy of TFBS prediction.

## 8.5 Conclusion and Future Work

The important question now, is to address the choice of the approach, that is, when to use which method? The answer to this question depends on the dataset under consideration. If the dataset is such that, all the positions seem to be independent from one another, then a simple PWM could be used. If we know that there are dependencies between adjacent nucleotide positions, a Weight Array Model (higher-order PWM) can be used. In cases where dependencies exist between non-adjacent nucleotide positions, one should use the MDD approach. It is better to go beyond a simple PWM when building models for binding site prediction. The results, in the worst case, may remain unchanged but in most cases, we can see a significant improvement in the performance while using the higher-order models.

We could also extend these models to predict other biological motifs in DNA, RNA, and proteins and some of these models, such as the MDD approach can be used both as a predicting tool as well as a clustering tool.

## REFERENCES

[1] S. Nandi and I.  Ioshikhes, “Optimizing the GATA-3 position weight matrix to improve the identification of novel binding sites,” *BMC Genomics*, vol. 13, no. 1, pp. 416-432, 2012.

[2] A. Sandelin, W.Alkema, P.Engstrom, W. W. Wasserman, and B. Lenhard. JASPAR: An open-access database for eukaryotic transcription factor binding profiles. Nucleic Acids Res., 32: D91-D94, 2004.

[3] V. Matys, O. V. Kel-Margoulis, E. Frickle, et al. TRANSFAC and its module TRANSCOMPEL: Transcriptional gene regulation in eukaryotes. Nucleic Acids Res., 34: D108-D10, 2006.

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

[5] RSA-tools-Tutorials-Position-specific-scoring matrices. Regulatory Sequence Analysis Tool. Retrieved June 16, 2013, from http://rsat.ulb.ac.be/tutorials/tut_PSSM.html.

[6] 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.

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

[8] J. Fan, S. Upadhye, and A. Worster, “Understanding receiver operating characteristic (ROC) curves,” *CJEM*, vol. 8, no. 1, pp. 19-20, 2006.

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

[10] 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.

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

[12] Homo Sapiens Splice Sites Datasets. Retrieved October 2, 2013, from http://www.sci.unisannio.it/docenti/rampone/.

**APPENDIX**

**Source Code (Java)**

**Position Weight Matrix**

```
import java.text.*;
import java.io.*;
import java.util.*;

/**
 * @author Dhivya Srinivasan, SJSU December 2013
 *
 */

public class PWM {
	static int len;
	boolean CalcPwm = true;
	int i, k;
	float a, c, g, t;
	String pwm[][] = new String[4][len];

	char cons[] = new char[len];
	char base[] = { 'A', 'C', 'G', 'T' };
	String absFreq[][] = new String[4][len];
	String relFreq[][] = new String[4][len];
	ArrayList<String> arr = new ArrayList<String>();
	static ArrayList<String> arr_scoreseq = new ArrayList<String>();
	static String train_sequence_path = "";
	static String score_sequence_path = "";
	static String output_file_path = "";
	static BufferedWriter bw = null;

	public static void main(String args[]) {
		PWM.train_sequence_path = args[0];
		PWM.score_sequence_path = args[1];
		PWM.output_file_path = args[2];
		PWM.len = Integer.parseInt(args[3]);
		PWM pwm = new PWM();
		pwm.OpenFile();
		pwm.doPwm();
		try {
			FileWriter fw = new FileWriter(PWM.output_file_path);
			bw = new BufferedWriter(fw);
			for (int i = 0; i < PWM.arr_scoreseq.size(); i++) {
				bw.append(PWM.arr_scoreseq.get(i) + "	"
				+ pwm.scorePWM(pwm.pwm, PWM.arr_scoreseq.get(i)) + "\n");
			}

			bw.close();

		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	public ArrayList<String> OpenFile() {
		try {
			FileReader fr = new FileReader(train_sequence_path);
			BufferedReader br = new BufferedReader(fr);
			int NoOfLines = LineCount(train_sequence_path);
			for (int i = 0; i < NoOfLines; i++) {
				arr.add(br.readLine());
			}
			br.close();
		} catch (IOException e) {
			System.out.println("File not found");
		}
		// /New
		try {
			FileReader fr = new FileReader(score_sequence_path);
			BufferedReader br = new BufferedReader(fr);
			int NoOfLines = LineCount(score_sequence_path);
			for (int i = 0; i < NoOfLines; i++) {
				arr_scoreseq.add(br.readLine());
			}
			br.close();
		} catch (IOException e) {
			System.out.println("File not found");
		}
		return arr;
	}

	int LineCount(String path) throws IOException {
		String newPath = path;
		FileReader f = new FileReader(newPath);
		BufferedReader br = new BufferedReader(f);
		int Num = 0;
		String str;
		while ((str = br.readLine()) != null) {
```

```
                Num++;
            }
            br.close();
            return Num;
    }

    public void doPwm() {
            calConsensus(arr, CalcPwm, len);
            System.out.print("Consensus string for the PWM is ");
            System.out.println(cons);
            for (i = 0; i < len; i++) {
                    System.out.print("\t" + (i + 1));
            }
            System.out.print("\n");
            for (int l = 0; l < 4; l++) {
                    System.out.print(base[l] + "\t");
                    for (int m = 0; m < len; m++) {
                            System.out.print(pwm[l][m]);
                            System.out.print("\t");
                    }
                    System.out.print("\n");
            }
    }

    PWM calConsensus(ArrayList<String> array, boolean CalcPwm, int length) {
            char ch;
            for (i = 0; i < length; i++) {
                    a = c = g = t = 1;
                    for (int j = 0; j < array.size(); j++) {
                            ch = array.get(j).toString().charAt(i);
                            if (ch == 'A')
                                    a++;
                            else if (ch == 'C')
                                    c++;
                            else if (ch == 'G')
                                    g++;
                            else
                                    t++;
                    }
                    calPwm(array, a, c, g, t);
                    float max = Math.max(Math.max(a, c), Math.max(g, t));
                    if (a == max)
                            cons[i] = 'A';
                    else if (c == max)
                            cons[i] = 'C';
                    else if (g == max)
                    cons[i] = 'G';
                else
                    cons[i] = 'T';
            }
            return this;
        }
        void calPwm(ArrayList<String> array, float a, float c, float g, float t) {
            float base[] = { a, c, g, t };
            DecimalFormat df = new DecimalFormat("#.####");
            for (k = 0; k < 4; k++) {
                absFreq[k][i] = new Float(base[k]).toString();
                if (absFreq[k][i] == null) {
                    System.out.println("Here");
                }
                relFreq[k][i] = df.format(base[k] / (array.size() + 4));
                if (CalcPwm) {
                    pwm[k][i] = df.format((Math.log((base[k] / (array.size() + 4)) /
                                                        0.25) / Math.log(2)));
                }
            }
        }

        float scorePWM(String[][] pwm, String str) {
            float score = 0;
            for (int i = 0; i < str.length(); i++) {
                char c = str.charAt(i);
                switch (c) {
                case 'A':
                    score = score + Float.parseFloat(pwm[0][i]);
                    break;
                case 'C':
                    score = score + Float.parseFloat(pwm[1][i]);
                    break;
                case 'G':
                    score = score + Float.parseFloat(pwm[2][i]);
                    break;
                case 'T':
                    score = score + Float.parseFloat(pwm[3][i]);
                    break;
                default:
                    break;
                }
            }
            return score;
        }
    }
```

**Higher-order PWM (Dinucleotides)**

```
import java.text.*;
import java.io.*;
import java.util.*;

/**
* @author Dhivya Srinivasan, SJSU
* December 2013
*
*/

public class HigherOrderPWM {
	static int len;
	boolean CalcPwm = true;
	int i, k;
	float a, c, g, t;
	HashMap<String, Integer> nim = null;
	TreeMap<String, ArrayList<Integer>> tm = null;
	String pwm[][] = new String[16][len];
	String cons[] = new String[len];
	char base[] = { 'A', 'C', 'G', 'T' };

	static String path = null;
	static String output_file_path = null;
	static String scoreseqpath = null;

	String absFreq[][] = new String[4][len];
	String relFreq[][] = new String[4][len];
	ArrayList<String> arr = new ArrayList<String>();
	ArrayList<String> arr_scoreseq = new ArrayList<String>();
	static BufferedWriter bw = null;

	public static void main(String args[]) {
		HigherOrderPWM.path = args[0];
		HigherOrderPWM.scoreseqpath = args[1];
		HigherOrderPWM.output_file_path = args[2];
		HigherOrderPWM.len = Integer.parseInt(args[3]);
		HigherOrderPWM pwm = new HigherOrderPWM();
		pwm.OpenFile();
		pwm.fillIndexOfNucleotide();
		pwm.doPwm();

		try {
			FileWriter fw = new FileWriter(HigherOrderPWM.output_file_path);
			bw = new BufferedWriter(fw);
			for (int i = 0; i < pwm.arr_scoreseq.size(); i++) {
				bw.append(pwm.arr_scoreseq.get(i) + "        "
						+ pwm.scorePWM(pwm.arr_scoreseq.get(i)) +"\n");
			}
			bw.close();

		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	public ArrayList<String> OpenFile() {
		try {
			FileReader fr = new FileReader(path);
			BufferedReader br = new BufferedReader(fr);
			int NoOfLines = LineCount(path);
			for (int i = 0; i < NoOfLines; i++) {
				arr.add(br.readLine());
			}
			br.close();
		} catch (IOException e) {
			System.out.println("File not found");
		}

		try {
			FileReader fr = new FileReader(scoreseqpath);
			BufferedReader br = new BufferedReader(fr);
			int NoOfLines = LineCount(scoreseqpath);
			for (int i = 0; i < NoOfLines; i++) {
				arr_scoreseq.add(br.readLine());
			}
			br.close();
		} catch (IOException e) {
			System.out.println("File not found");
		}
		return arr;
	}

	int LineCount(String path) throws IOException {
		String newPath = path;
		FileReader f = new FileReader(newPath);
		BufferedReader br = new BufferedReader(f);
		int Num = 0;
		String str;
		while ((str = br.readLine()) != null) {
			Num++;
		}
		br.close();
		return Num;
	}

	public void doPwm() {
		calConsensus(arr, CalcPwm);
		for (i = 0; i < len - 1; i++) {
			System.out.print("\t" + (i + 1));
		}
		System.out.print("\n");
		Object[] keylist = tm.keySet().toArray();

		for (int l = 0; l < 16; l++) {
			System.out.print(keylist[l].toString() + "\t");
			for (int m = 0; m < len - 1; m++) {
				System.out.print(pwm[l][m]);
				System.out.print("\t");
			}
			System.out.print("\n");
		}
	}

	ArrayList<Integer> fillInitialTreeMap() {
		ArrayList<Integer> al = new ArrayList<Integer>();

		for (int i = 0; i < len - 1; i++) {
			al.add(new Integer(1));
		}

		return al;

	}

	void fillIndexOfNucleotide() {
		nim = new HashMap<String, Integer>();
		nim.put("AA", new Integer(0));
		nim.put("AC", new Integer(1));
		nim.put("AG", new Integer(2));
		nim.put("AT", new Integer(3));
		nim.put("CA", new Integer(4));
		nim.put("CC", new Integer(5));
		nim.put("CG", new Integer(6));
		nim.put("CT", new Integer(7));
		nim.put("GA", new Integer(8));
		nim.put("GC", new Integer(9));
```

```
        nim.put("GG", new Integer(10));
        nim.put("GT", new Integer(11));
        nim.put("TA", new Integer(12));
        nim.put("TC", new Integer(13));
        nim.put("TG", new Integer(14));
        nim.put("TT", new Integer(15));

}

HigherOrderPWM calConsensus(ArrayList<String> array, boolean CalcPwm) {
        String dinuceleotide = "";

        tm = new TreeMap<String, ArrayList<Integer>>();

        tm.put("AA", fillInitialTreeMap());
        tm.put("AC", fillInitialTreeMap());
        tm.put("AG", fillInitialTreeMap());
        tm.put("AT", fillInitialTreeMap());
        tm.put("CA", fillInitialTreeMap());
        tm.put("CC", fillInitialTreeMap());
        tm.put("CG", fillInitialTreeMap());
        tm.put("CT", fillInitialTreeMap());
        tm.put("GA", fillInitialTreeMap());
        tm.put("GC", fillInitialTreeMap());
        tm.put("GG", fillInitialTreeMap());
        tm.put("GT", fillInitialTreeMap());
        tm.put("TA", fillInitialTreeMap());
        tm.put("TC", fillInitialTreeMap());
        tm.put("TG", fillInitialTreeMap());
        tm.put("TT", fillInitialTreeMap());

        int max = 1;
        String maxcons = "";

        for (i = 0; i < len - 1; i++) {

                for (int j = 0; j < array.size(); j++) {
                        dinuceleotide = array.get(j).toString().charAt(i) + ""
                                        + array.get(j).toString().charAt(i + 1);
                        int curval = tm.get(dinuceleotide).get(i).intValue();
                        curval++;

                        if (max <= curval) {
                                maxcons = dinuceleotide;
                                max = curval;
                        }
                ArrayList<Integer> al = tm.get(dinuceleotide);
                al.remove(i);
                al.add(i, curval);
            }
            calPwm(array, i);
            cons[i] = maxcons;

        }
        return this;
    }

    void calPwm(ArrayList<String> array, int i) {
        DecimalFormat df = new DecimalFormat("#.####");

        int k = 0;
        for (Object key : tm.keySet()) {
            if (CalcPwm) {

                pwm[k][i] = df.format((Math.log((tm.get(key).get(i)
                        .floatValue() / (array.size() + 16)) / 0.0625) / Math
                        .log(2)));
                k++;
            }
        }
    }

    float scorePWM(String str) {
        float score = 0;
        for (int i = 0; i < str.length() - 1; i++) {
            String st = str.charAt(i) + "" + str.charAt(i + 1);
            score += Float.parseFloat(pwm[nim.get(st).intValue()][i]);
        }
        return score;
    }
}
```

**Maximal Dependence Decomposition (MDD)**

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;
import java.util.Map.Entry;

/**
* @author Dhivya Srinivasan, SJSU
* December 2013
*
*/
public class MDD {
        static int LENGTH = 0;
        static int THRESHOLD = 0;
        static double pvalue = 0.0;
        char cons[] = new char[4];
        char base[] = { 'A', 'C', 'G', 'T' };
        char cons_new;
        boolean Initial = true, has_pwm = true;
        int len = LENGTH, max_pos, count = 0;
        int SplitPosition = -1;
        int actualSplitPosition = -1;
        char SplitChar = ' ';
        int length = LENGTH;
        int col_len = LENGTH;
        float sum, sum_value;
        float value;
        boolean isLeaf = false;
        PWM_MDD mdd_pwm1, mdd_pwm2;
        String relFreq[][] = new String[4][len];
        String absFreq[][] = new String[4][len];
        ArrayList<String> IniGroup = new ArrayList<String>();
        ArrayList<String> Group1 = new ArrayList<String>();
        ArrayList<String> Group2 = new ArrayList<String>();
        ArrayList<Float> chi_sum = new ArrayList<Float>();
        ArrayList<ArrayList<String>> al = new ArrayList<ArrayList<String>>();
        ArrayList<String> DisplayGroup = new ArrayList<String>();
        HashMap<Integer, ArrayList<Float>> hmap =
        new HashMap<Integer,ArrayList<Float>>();

        StringBuilder nodeName = new StringBuilder("");
        StringBuilder nodeName1 = new StringBuilder("");

        PWM_MDD PWMAtConsesus;
```

```
String PWMAtConsensusArray[][] = new String[4][len];

ArrayList<Integer> FilledPositions = new ArrayList<Integer>();
String leafPWM[][] = new String[4][len];
MDD MDDCi;
MDD MDDCiBar;

static BufferedWriter bw = null;

/*
 * Program begins here. This Program builds an MDD tree based out of
 * sequences in the input path file and builds a Maximal Dependence
 * Decomposition tree. It later walks down a new sequence with this tree and
 * scores the new sequence with the leaf PWM.
 */
public static void main(String args[]) throws Exception {
        PWM_MDD.train_sequence_path = args[0];
        PWM_MDD.score_sequence_path = args[1];
        PWM_MDD.output_file_path = args[2];
        MDD.LENGTH = Integer.parseInt(args[3]);
        MDD.THRESHOLD = Integer.parseInt(args[4]);
        MDD.pvalue = Double.parseDouble(args[5]);
        MDD root_mdd = new MDD();
        PWM_MDD pwm = new PWM_MDD();
        root_mdd.len = pwm.len;
        root_mdd.IniGroup = pwm.arr;
        pwm.OpenFile();

        for (int i = 0; i < pwm.arr.size(); i++) {
                root_mdd.DisplayGroup.add(pwm.arr.get(i));
        }

        for (int i = 0; i < root_mdd.len; i++) {
                root_mdd.FilledPositions.add(new Integer(-1));
        }
        root_mdd.nodeName.append("All TFBS");
        root_mdd.nodeName1.append("RT");
        root_mdd.doMDD();

        System.out.println("WALKING THE MDD TREE !!!");
        BTreePrinter.printBinaryTree(root_mdd, 0);
        //
        root_mdd.scoreSequences();

}
```

```
void scoreSequences()
{
        try {
                FileWriter fw = new FileWriter(PWM_MDD.output_file_path);
                bw = new BufferedWriter(fw);

                for (int i = 0; i < PWM_MDD.arr_scoreseq.size(); i++) {
                        bw.append(PWM_MDD.arr_scoreseq.get(i) + "        "
                        + this.Traverse(PWM_MDD.arr_scoreseq.get(i), this, false) +"\n");
                }

                bw.close();

        } catch (Exception e) {
                e.printStackTrace();
        }
}

/*
 * Displays the entire MDD tree in a modular fashion
 */
void display(String Parenttype) {

        System.out.println(this.nodeName + "  " + "Num sequences ="
                        + this.DisplayGroup.size());

        if (this.isLeaf) {
                DisplayPWM(this.leafPWM);
        }

        if (MDDCi != null) {
                String type = Parenttype + "Ci->";
                MDDCi.display(type);
        }

        if (MDDCiBar != null) {
                String type = Parenttype + "CiBar->";
                MDDCiBar.display(type);
        }

}

void markPosition(int max_pos) {

        int i;
        for (i = 0; i < LENGTH; i++) {
                if (FilledPositions.get(i).intValue() == 1) {
                        max_pos++;
                } else {
                        if (i == max_pos) {
                                FilledPositions.set(i, 1);
                                break;
                        }
                }
        }
        this.actualSplitPosition = i;
        System.out.println("Changed position = " + i);

}

void doMDD() throws Exception {
        pvalue p = new pvalue();
        col_len = col_len - 1;
        if ((IniGroup.size() <= THRESHOLD) || (IniGroup.get(0).length() == 1) ) {
                this.isLeaf = true;

                PWM_MDD doPWM = new PWM_MDD();
                this.leafPWM = doPWM.calConsensus(this.DisplayGroup, true,
                LENGTH).pwm;
                Set<Entry<Integer, ArrayList<Float>>> set = hmap.entrySet();
                for (Map.Entry<Integer, ArrayList<Float>> me : set) {
                        ArrayList<Float> al = hmap.get(me.getKey());

                        for (int k = 0; k < 4; k++) {
                                this.leafPWM[k][me.getKey()] = al.get(k).toString();
                        }

                }
                System.out.println("************ LEAF PWM  for " + this.nodeName
                +"******************");
                for(int i=0; i<this.DisplayGroup.size(); i++)
                {
                        System.out.println(this.DisplayGroup.get(i));
                }
                DisplayPWM(this.leafPWM);

                for (int i = 0; i < this.DisplayGroup.size(); i++) {
                        if (this.DisplayGroup.get(i).length() > MDD.LENGTH) {
                                System.out.println("BAD LENGTH !!!!"
                                                + this.DisplayGroup.get(i).length());
                        }
                        float score = doPWM.scorePWM(this.leafPWM,
                                this.DisplayGroup.get(i));
                }

                return;
        }


        System.out.println("************sequences in Non - LEAF Node  for " +
        this.nodeName + "num seq = " +
        this.DisplayGroup.size()+"******************");
        for(int i=0; i<this.DisplayGroup.size(); i++)
        {
                System.out.println(this.DisplayGroup.get(i));
        }

        PWM_MDD pwm1;
        PWM_MDD pwm = new PWM_MDD();
        pwm.len = len;
        pwm1 = pwm.calConsensus(this.IniGroup, false, len);
        this.cons = pwm1.cons;
        for (int i = 0; i < len; i++) {
                SplitGroup(i, IniGroup, false);
                sum = (float) 0.0;
                sum_value = (float) 0.0;
                PWM_MDD pwm2 = new PWM_MDD();
                PWM_MDD pwm3 = new PWM_MDD();
                PWM_MDD pwm4;
                pwm2.len = len;
                pwm3.len = len;

                // calculate consensus and split group based on the consensus position

                pwm4 = pwm2.calConsensus(Group1, false, len);
                absFreq = pwm4.absFreq;
                pwm4 = pwm3.calConsensus(Group2, false, len);
                relFreq = pwm4.relFreq;
                al.add(new ArrayList<String>());
                Integer h = i + 1;
                al.get(i).add(h.toString());
                al.get(i).add(String.valueOf(cons[i]));
                for (int k = 0; k < len; k++) {
                        calChiSq(i, k);
                        sum = sum + sum_value;
                }
                chi_sum.add(sum);
                al.get(i).add(String.valueOf(sum));
```

```
}
float max_val = Collections.max(chi_sum);

if( max_val < p.pvalue_lookup(pvalue, this.IniGroup.get(0).length()* 3)) {
        //LEAF PWM
        this.isLeaf = true;

        PWM_MDD doPWM = new PWM_MDD();
        this.leafPWM = doPWM.calConsensus(this.DisplayGroup, true,
        LENGTH).pwm;
        // DisplayPWM(this.leafPWM);
        Set<Entry<Integer, ArrayList<Float>>> set = hmap.entrySet();
        for (Map.Entry<Integer, ArrayList<Float>> me : set) {
                ArrayList<Float> al = hmap.get(me.getKey());

                for (int k = 0; k < 4; k++) {
                        this.leafPWM[k][me.getKey()] = al.get(k).toString();
                }

        }
        System.out.println("************ LEAF PWM  for " + this.nodeName
 +"******************");
        for(int i=0; i<this.DisplayGroup.size(); i++)
        {
                System.out.println(this.DisplayGroup.get(i));
        }
        DisplayPWM(this.leafPWM);

        for (int i = 0; i < this.DisplayGroup.size(); i++) {
                if (this.DisplayGroup.get(i).length() > MDD.LENGTH) {
                        System.out.println("BAD LENGTH !!!!"
                                        + this.DisplayGroup.get(i).length());
                }
                float score = doPWM.scorePWM(this.leafPWM,
                                this.DisplayGroup.get(i));
        }

        return;
}

for (int i = chi_sum.size() - 1; i >= 0; i--) {
        if (max_val == chi_sum.get(i)) {

                max_pos = i;

                break;
```

```
		}
	}
	 System.out.println(al);
	this.SplitPosition = max_pos;
	this.SplitChar = al.get(max_pos).get(1).charAt(0);
	markPosition(max_pos);
	System.out.println("Splitting at position " + SplitPosition
				+ " on char " + this.SplitChar);

	ArrayList<String> tempList = new ArrayList<String>();
	for (int idx = 0; idx < IniGroup.size(); idx++) {
		tempList.add("" + IniGroup.get(idx).charAt(max_pos));
	}

	PWM_MDD tempPWM = new PWM_MDD();
	this.PWMAtConsesus = tempPWM.calConsensus(tempList, true, 1);
	this.PWMAtConsensusArray = this.PWMAtConsesus.pwm;

	ArrayList<Float> al = new ArrayList<Float>();
	for (int k = 0; k < 4; k++) {
		al.add(k, Float.parseFloat(PWMAtConsensusArray[k][0]));
	}
	Integer val = this.actualSplitPosition;

	hmap.put(val, al);

	/*
	 * Group1 is all elements that have a particular nucleotide in the
	 * maximum consensus position.
	 */
	if (Group1.size() >= 1) {
		MDDCi = new MDD();
		SplitGroupAndDisplayGroup(max_pos, IniGroup, false,
					MDDCi.DisplayGroup, true);
		MDDCi.len = len - 1;

		MDDCi.Remove_Col(Group1, max_pos);

		for (int i = 0; i < Group1.size(); i++) {
			MDDCi.IniGroup.add(Group1.get(i));
		}
	}

	// All other sequences that do not have consensus
	if (Group2.size() >= 1) {
        MDDCiBar = new MDD();
        SplitGroupAndDisplayGroup(max_pos, IniGroup, false,
                MDDCiBar.DisplayGroup, false);

        MDDCiBar.len = len;

        for (int i = 0; i < Group2.size(); i++) {
            MDDCiBar.IniGroup.add(Group2.get(i));
        }
    }

    // Recursively build the tree for both Group1 and 2

    if (MDDCi != null) {

        for (int i = 0; i < this.FilledPositions.size(); i++) {
            MDDCi.FilledPositions.add(new Integer(this.FilledPositions
                    .get(i)));
        }

        MDDCi.hmap.putAll(hmap);
        String temp = this.nodeName.toString();
        if (!temp.equalsIgnoreCase("All TFBS")) {
            MDDCi.nodeName.append(this.nodeName);

        }
        MDDCi.nodeName1.append(this.SplitChar + ""
                + (this.actualSplitPosition+1));

        MDDCi.nodeName.append(this.SplitChar + ""
                + (this.actualSplitPosition+1));

        MDDCi.doMDD();
    }

    if (MDDCiBar != null) {

        for (int i = 0; i < this.FilledPositions.size(); i++) {
            MDDCiBar.FilledPositions.add(new Integer(this.FilledPositions
                    .get(i)));
        }
        MDDCiBar.FilledPositions.set(this.actualSplitPosition, -1);
        MDDCiBar.hmap.putAll(hmap);
        MDDCiBar.hmap.remove(this.actualSplitPosition);

        String temp = this.nodeName.toString();
        if (!temp.equalsIgnoreCase("All TFBS")) {
            MDDCiBar.nodeName.append(this.nodeName);

        }
        MDDCiBar.nodeName.append("~" + this.SplitChar + ""
                + (this.actualSplitPosition+1));
        MDDCiBar.nodeName1.append("~" + this.SplitChar + ""
                + (this.actualSplitPosition+1));

        MDDCiBar.doMDD();
    }

}

void SplitGroup(int m, ArrayList<String> group, boolean act) {
    Group1.clear();
    Group2.clear();
    for (int j = 0; j < group.size(); j++) {
        if (cons[m] == group.get(j).charAt(m)) {
            Group1.add(group.get(j));

        } else {
            Group2.add(group.get(j));

        }
    }
}

void SplitGroupAndDisplayGroup(int m, ArrayList<String> group, boolean act,
        ArrayList<String> dispGroup, boolean Ci) {
    Group1.clear();
    Group2.clear();
    for (int j = 0; j < group.size(); j++) {
        if (cons[m] == group.get(j).charAt(m)) {
            Group1.add(group.get(j));
            if (Ci) {
                dispGroup.add(this.DisplayGroup.get(j));
            }
        } else {
            Group2.add(group.get(j));
            if (!Ci) {
                dispGroup.add(this.DisplayGroup.get(j));
            }
        }
    }
}
```

```
/*
 * Utility method to keep track of Consensus column already considered by
 * removing it from original sequence.
 */
ArrayList Remove_Col(ArrayList<String> arr, int pos) {
        for (int a = 0; a < arr.size(); a++) {
                StringBuilder str1 = new StringBuilder(arr.get(a));
                str1.delete(pos, pos + 1);
                arr.remove(a);
                arr.add(a, str1.toString());
        }
        return arr;
}

/*
 * Calculates the Chi Square dependence between two nucleotides position.
 */

void calChiSq(int i, int k) {
        if (k == i) {
                al.get(i).add(null);
                sum_value = 0;
        } else {
                value = ChiVal(i, k);
                al.get(i).add(String.valueOf(value));
        }
}

 /* Uses relative and absolute frequency to score the dependence between two nucleotides
*/
float ChiVal(int i, int k) {
        value = 0;
        sum_value = 0;
        int Total = Group1.size() + 4;
        for (int j = 0; j < 4; j++) {
                if (i == k) {
                        sum_value = 0;
                } else {
                        value = value
                                        + ((Total * Float.parseFloat(relFreq[j][k])) - Float
                                                        .parseFloat(absFreq[j][k] + 4))
                                        * ((Total * Float.parseFloat(relFreq[j][k])) - Float
                                                        .parseFloat(absFreq[j][k] + 4))
                                        / (Total * Float.parseFloat(relFreq[j][k]));
                        sum_value = value;
                }
        }
        if (Group2.size() == 0) {
        }
        return value;
}

float constructPWMDynamically(MDD temp, String str) {
        PWM_MDD doPWM = new PWM_MDD();
        MDD dyn_mdd = new MDD();
        ArrayList<String> temp_al = new ArrayList<String>();
        temp_al.add(str);
        dyn_mdd.leafPWM = doPWM.calConsensus(temp.DisplayGroup, true,
        LENGTH).pwm;

        Set<Entry<Integer, ArrayList<Float>>> set = temp.hmap.entrySet();
        for (Map.Entry<Integer, ArrayList<Float>> me : set) {
                ArrayList<Float> al = temp.hmap.get(me.getKey());

                for (int k = 0; k < 4; k++) {
                        dyn_mdd.leafPWM[k][me.getKey()] = al.get(k).toString();
                }

        }
        System.out.println("************ LEAF PWM  for " + str
        +"******************");
        DisplayPWM(dyn_mdd.leafPWM);

        float score = doPWM.scorePWM(dyn_mdd.leafPWM, str);
        return score;

}

/*
 * Traverse the tree for the new string
 */
float Traverse(String str, MDD root,boolean shouldPrint) {
        String original = str;
        MDD temp;
        temp = root;
        while (temp != null) {
                if (temp.MDDCi == null && temp.MDDCiBar == null) {
                        if(shouldPrint) {
                        System.out.println("************ LEAF PWM  for " +
                        temp.nodeName +"******************");
                        DisplayPWM(temp.leafPWM);
					}
					PWM_MDD pwm = new PWM_MDD();

					// Reached required node. Now score this new string with the
					PWM in that leaf node.
					float score = pwm.scorePWM(temp.leafPWM, original);
					if(shouldPrint) {
					System.out.println("Score is " + score);
					}
					return score;
				}
				if (str.charAt(temp.SplitPosition) == temp.SplitChar) {
					StringBuilder sb = new StringBuilder(str);
					sb.deleteCharAt(temp.SplitPosition);
					str = sb.toString();
					temp = temp.MDDCi;
				} else {
					if (temp.MDDCiBar != null) {
						temp = temp.MDDCiBar;
					} else {
						float score = constructPWMDynamically(temp, original);
						if(shouldPrint) {
						System.out.println("Score for " + original + "= " + score);
						}
						return score;
					}
				}
			}

			return 0;
		}

		void DisplayPWM(String[][] pwm) {
			for (int l = 0; l < 4; l++) {
				System.out.print(base[l] + "\t");
				for (int m = 0; m < length; m++) {
					System.out.print(pwm[l][m]);
					System.out.print("\t");
				}
				System.out.print("\n");
			}
		}
	}
```

**PWM used with MDD**

```
import java.text.*;
import java.io.*;
import java.util.*;

/**
* @author Dhivya Srinivasan, SJSU
* December 2013
*
*/

public class PWM_MDD {
	int len = MDD.LENGTH;
	boolean CalcPwm = true;
	int i, k;
	float a, c, g, t;
	String pwm[][] = new String[4][len];

	char cons[] = new char[len];
	char base[] = { 'A', 'C', 'G', 'T' };
	String absFreq[][] = new String[4][len];
	String relFreq[][] = new String[4][len];
	ArrayList<String> arr = new ArrayList<String>();
	static ArrayList<String> arr_scoreseq = new ArrayList<String>();
	static String train_sequence_path = "";
	static String score_sequence_path = "";
	static String output_file_path = "";
	static BufferedWriter bw = null;

	public static void main(String args[]) {
		PWM_MDD pwm = new PWM_MDD();
		pwm.OpenFile();
		pwm.doPwm();
		try {
			FileWriter fw = new FileWriter(PWM_MDD.output_file_path);
			bw = new BufferedWriter(fw);

			for (int i = 0; i < PWM_MDD.arr_scoreseq.size(); i++) {
				bw.append(PWM_MDD.arr_scoreseq.get(i) + "	"
			+ pwm.scorePWM(pwm.pwm, PWM_MDD.arr_scoreseq.get(i)) + "\n");
			}

			bw.close();

		} catch (Exception e) {
```

```
                e.printStackTrace();
            }
    }

    public ArrayList<String> OpenFile() {
            try {
                FileReader fr = new FileReader(train_sequence_path);
                BufferedReader br = new BufferedReader(fr);
                int NoOfLines = LineCount(train_sequence_path);
                for (int i = 0; i < NoOfLines; i++) {
                    arr.add(br.readLine());
                }
                br.close();
            } catch (IOException e) {
                System.out.println("File not found");
            }
            // New
            try {
                FileReader fr = new FileReader(score_sequence_path);
                BufferedReader br = new BufferedReader(fr);
                int NoOfLines = LineCount(score_sequence_path);
                for (int i = 0; i < NoOfLines; i++) {
                    arr_scoreseq.add(br.readLine());
                }
                br.close();
            } catch (IOException e) {
                System.out.println("File not found");
            }
            return arr;
    }

    int LineCount(String path) throws IOException {
            String newPath = path;
            FileReader f = new FileReader(newPath);
            BufferedReader br = new BufferedReader(f);
            int Num = 0;
            String str;
            while ((str = br.readLine()) != null) {
                Num++;
            }
            br.close();
            return Num;
    }

    public void doPwm() {
            calConsensus(arr, CalcPwm, len);
```

```
        System.out.print("Consensus string for the PWM is ");
        System.out.println(cons);
        for (i = 0; i < len; i++) {
                System.out.print("\t" + (i + 1));
        }
        System.out.print("\n");
        for (int l = 0; l < 4; l++) {
                System.out.print(base[l] + "\t");
                for (int m = 0; m < len; m++) {
                        System.out.print(pwm[l][m]);
                        System.out.print("\t");
                }
                System.out.print("\n");
        }
}

PWM_MDD calConsensus(ArrayList<String> array, boolean CalcPwm, int length) {
        char ch;
        for (i = 0; i < length; i++) {
                a = c = g = t = 1;
                for (int j = 0; j < array.size(); j++) {
                        ch = array.get(j).toString().charAt(i);
                        if (ch == 'A')
                                a++;
                        else if (ch == 'C')
                                c++;
                        else if (ch == 'G')
                                g++;
                        else
                                t++;
                }
                calPwm(array, a, c, g, t);
                float max = Math.max(Math.max(a, c), Math.max(g, t));
                if (a == max)
                        cons[i] = 'A';
                else if (c == max)
                        cons[i] = 'C';
                else if (g == max)
                        cons[i] = 'G';
                else
                        cons[i] = 'T';
        }
        return this;
}
```

```
void calPwm(ArrayList<String> array, float a, float c, float g, float t) {
                float base[] = { a, c, g, t };
                DecimalFormat df = new DecimalFormat("#.####");
                for (k = 0; k < 4; k++) {
                        absFreq[k][i] = new Float(base[k]).toString();
                        if (absFreq[k][i] == null) {
                                System.out.println("Here");
                        }
                        relFreq[k][i] = df.format(base[k] / (array.size() + 4));
                        if (CalcPwm) {
                                pwm[k][i] = df
                                                .format((Math.log((base[k] / (array.size() + 4)) /
                                                                0.25) / Math.log(2)));
                        }

                }
        }

        float scorePWM(String[][] pwm, String str) {
                float score = 0;
                for (int i = 0; i < str.length(); i++) {
                        char c = str.charAt(i);
                        switch (c) {
                        case 'A':
                                score = score + Float.parseFloat(pwm[0][i]);
                                break;
                        case 'C':
                                score = score + Float.parseFloat(pwm[1][i]);
                                break;
                        case 'G':
                                score = score + Float.parseFloat(pwm[2][i]);
                                break;
                        case 'T':
                                score = score + Float.parseFloat(pwm[3][i]);
                                break;
                        default:
                                break;
                        }
                }
                return score;
        }
}
```