2004

Design and analysis of gene prediction algorithms

Jimmy Y. Yu
San Jose State University

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

Recommended Citation

Yu, Jimmy Y., "Design and analysis of gene prediction algorithms" (2004). *Master's Theses*. 2642.
DOI: <https://doi.org/10.31979/etd.acjw-v7ek>
https://scholarworks.sjsu.edu/etd_theses/2642

This Thesis 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 Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

DESIGN AND ANALYSIS OF
GENE PREDICTION ALGORITHMS

A Thesis

Presented to

The Faculty of the Department of Biological Sciences

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Jimmy Y. Yu

August 2004

UMI Number: 1424513

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI Microform 1424513

Copyright 2005 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

© 2004

Jimmy Y. Yu

ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF BIOLOGY

A handwritten signature in cursive script, reading "Sami Khuri".

Dr. Sami Khuri

A handwritten signature in cursive script, reading "Leslie Parr".

Dr. Leslie Parr

A handwritten signature in cursive script, reading "Julio Soto".

Dr. Julio Soto

APPROVED FOR THE UNIVERSITY

A handwritten signature in cursive script, reading "Pamela C. Stuebe".

ABSTRACT

DESIGN AND ANALYSIS OF GENE PREDICTION ALGORITHMS

by Jimmy Y. Yu

In this study, Linear Discriminant Analysis and hidden Markov models are implemented and compared as algorithms used for gene prediction. The goal is to determine which algorithm is more accurate in predicting genes using a 12-nucleotide window surrounding each potential site. The sites considered are the start and stop codons and the donor and acceptor splice sites. Models are trained on mouse sequences and algorithms are tested with rat sequences, both from Genbank® release 136. These sequences are also run through two publicly available packages, Genscan and FGENES, which utilize Linear Discriminant Analysis and hidden Markov models, respectively. Sensitivity and specificity are used to measure the accuracy of gene predictions for each program. Comparison of these programs showed that hidden Markov models predict coding regions more accurately. The results from FGENES and Genscan also showed that hidden Markov models are more accurate than Linear Discriminant Analysis in gene prediction.

Acknowledgments

To my thesis committee members: Thank you for all the time you spent in helping me to prepare and polish this thesis. This work wouldn't have been possible without your help. Dr. Parr, thank you for all the time you spent with me ironing out the details, both big and small. Dr. Khuri, thank you for encouraging me to pursue this work and for opening the door for me to work with Dr. Parr.

To my parents: Thank you for pushing me to "just do it." I'm glad I pressed on and finally completed the work here at San José State.

To my wife: Thank you for your constant support through this busy and hectic time of my life. I couldn't have finished this without you.

To Jesus, my Lord and Savior: Thank you for putting all these people in my life and giving me the opportunity to be challenged, to learn, and to succeed. I'm thankful for all the ways that I was able to grow through this experience.

Table of Contents

Table of Contents	2
List of Figures	4
List of Tables	5
Introduction	6
Background	6
What is a Gene?	7
What is Gene Prediction?	8
Computer Science and Biology	9
Gene Prediction	9
Gene Prediction Rules	9
Computer Science and Gene Prediction	10
Algorithms Used in Gene Prediction	11
Linear Discriminant Analysis	11
Hidden Markov Model	12
Neural Networks	14
Decision Tree System	15
Linear Discriminant Analysis	15
Overview	15
Hypothetical Example	16
LDA Weight Tables	20
Advantages	22
Disadvantages	23
Hidden Markov Models	24
Markov Chains	24
Hidden Markov Models	29
Advantages	31
Disadvantages	32
Materials and Methods	34
Data Sources	34
Implementation of LDA Procedure	39
Weight Tables	40

Linear Discriminant Analysis	43
Adjustments to LDA Program	48
Implementation of HMM Procedure	48
Training the Hidden Markov Model	48
Utilizing HMMs to Predict Genes	53
Adjustments to HMM Program	58
Results and Discussion	60
Tests and Results	60
Overview	60
LDA Results	61
HMM Results	64
Discussion	66
Conclusion	68
References	70
Appendix A	73
Generate_lda_table.pl Code	73
Appendix B	90
Generate_hmm_table.pl Code	90
Appendix C	107
Lda.c Code	107
Appendix D	123
Hmm.c Code	123

List of Figures

Figure 1 – Hypothetical weight distribution of 100 people	17
Figure 2 – Hypothetical height distribution of 100 people	18
Figure 3 – Hypothetical height and weight distribution of 100 people	19
Figure 4 – Sample LDA Weight Table	21
Figure 5 – Markov chain representing a dog owner's decision	25
Figure 6 – Transition probabilities for owners with small and large dogs	26
Figure 7 – Markov chain representing the DNA alphabet (Durbin et al., 1999)	27
Figure 8 – Hidden Markov model representing the dog owner models	30
Figure 9 – Hidden Markov model, an extension of the Markov chain	31
Figure 10 – Length distribution of mouse sequences used in training set	37
Figure 11 – Length distribution of rat sequences used in testing set	39
Figure 12 – Characteristics recorded for each site	40
Figure 13 – Generating the LDA Weight Table process	41
Figure 14 – Top scoring codons from LDA weight table	43
Figure 15 – LDA program flowchart	44
Figure 16 – Sample output from LDA program	46
Figure 17 – LDA algorithm used to calculate maximal frame score	47
Figure 18 - Generated Markov model scores for start codon	50
Figure 19 – Hidden Markov model used for gene prediction	51
Figure 20 – Sample HMM table	52
Figure 21 – HMM program main process flowchart	53
Figure 22 – HMM program algorithm	55
Figure 23 – Example of all legitimate paths from Start Site 1	56
Figure 24 – Sample output from HMM program	58

List of Tables

Table 1 – Raw results for comparing accuracy of two programs	62
Table 2 – Sensitivity and Specificity of LDA program	63
Table 3 – Sn(true) and Sn(false) of LDA program	64
Table 4 – Sensitivity and Specificity of HMM program	65
Table 5 – Sn(true) and Sn(false) of HMM program	66
Table 6 – Sensitivity and Specificity of LDA and HMM algorithms	67

Introduction

Background

The completion of the human genome project has opened numerous doors for researchers in the area of drug discovery and development. These data have the potential to dramatically shorten the amount of time and money required to bring a drug to market. The challenge lies in developing the tools necessary to extract and organize the useful pieces of information from the large volume of data. Bioinformatics is the key to bridging the gap between raw data and useful information.

Bioinformatics is referred to as “biology in silico” because it utilizes computing power in place of wet lab work. Researchers in the biological sciences have long relied on lab experiments to support or refute hypotheses. Even with the advancements in lab equipment and decreasing costs of lab materials, the process is still time consuming and expensive. Thus, the advantage in utilizing bioinformatics is that it can substantially reduce research time and cost.

One major area of interest in the development of bioinformatics is gene prediction. Let us think of the human genome as being a book of random characters with no punctuation. Consider that in eukaryotic organisms (*Homo sapiens*, *Mus musculus*) a very small percentage of genome corresponds to regions which can code for proteins. In the human genome, only 1.1% to 1.4% of the genome codes for proteins (“The Human Genome,” 2001), which means that out of every 100 characters that are read, only one or two contribute to a word that you can comprehend. These “words” are the pieces of information in the genome referred to as exons, which many researchers are interested in.

A program which can accurately predict the locations of these exons from a DNA sequence would be very valuable.

What is a Gene? A gene is a stretch of genomic DNA comprised of the four nucleotides (Adenine, Thymine, Guanine, and Cytosine) that can be read by the complex molecular machinery within an organism to form a protein (Singer & Berg, 1991). A eukaryotic gene may contain multiple segments that are combined to form the complete coding region for the gene. It also includes segments which regulate the gene's transcription and expression (Brazma, Parkinson, Schlitt & Shojatalab, 2001).

To understand what a gene is, we must first understand the biology behind it. The building blocks of each organism are proteins. The blueprint for creating those proteins lies in genetic data that are contained within each organism, commonly referred to as the genome. In humans, the genome is roughly three billion base pairs long (Brazma et al., 2001). Genes are the regions of the genome that are involved in the expression of proteins. In eukaryotic organisms, roughly one percent ("The Human Genome," 2001) (Brazma et al., 2001) of the genome encodes for proteins. These encoding regions are referred to as exons.

In prokaryotic organisms, the majority of the genome consists of exons ("How BIG," 2002), which means that most of the genome encodes for proteins. Eukaryotic genomes, in particular, mammalian and vertebrate genomes, are more complex and may contain other elements such as introns and intergenic DNA. Introns are portions of the gene present in the genome found interspersed between exons. They are removed prior

to the translation of the gene into proteins. Intergenic DNA are regions between genes which may include transcription factors and promoters that affect the expression level of a particular gene.

What is Gene Prediction? Oftentimes in a research lab, parts of an organism's genome will be sequenced in an attempt to identify regions which code for a specific protein, namely, the genes. Gene prediction is the process of identifying the regions where genes exist in the genome. It is important in studying eukaryotic organisms because much of the genome does not play an active role in the expression of proteins. Though the intergenic regions of the genome are likely to play a significant role in the life of eukaryotes, much of the research and development in today's industry centers around genomic regions involved in coding for proteins.

Gene prediction is one of the most challenging tasks in bioinformatics. It is the process of taking a sequence of nucleotides whose coding information is unknown and determining whether it contains a sequence which encodes for a protein. The task is difficult in eukaryotic organisms because less than 1.5% ("The Human Genome," 2001) of the nucleotides in the genome encodes proteins and genes are often comprised of multiple smaller fragments called exons. Pre-spliced DNA sequence contains both exons and introns. Intron exons junctions are characterized by a GU at the 5 prime end and an AG in the 3 prime end. In order to obtain the actual portion of the DNA sequence which encodes for a specific protein, the introns must be spliced out of the sequence and the remaining exons stitched together.

Computer Science and Biology. The marriage between computer science and biology goes back as far as 1981, when the Smith-Waterman algorithm for sequence alignment was published (Richon, 2001). Since then, a number of companies, such as Incyte Pharmaceuticals (now Incyte Corporation), Celera, Affymetrix, and Gene Logic, have been formed to take advantage of the ability of computers to process biological information, something that was never possible before. These companies, and many others, utilize computers to put together the human genome, analyze data from microarrays, and develop tools to help scientists extract useful data from raw DNA sequences. Today, the use of bioinformatics is widespread in the biotechnology community. The number of companies which develop bioinformatics tools is still growing and larger pharmaceutical companies such as Roche and Pfizer have developed their own in house bioinformatics departments.

Gene Prediction

Gene Prediction Rules. For gene prediction, we are interested mainly in start, stop, acceptor, and donor sites. The start site represents the start of a gene, and is almost always the codon or triplet “ATG” (AUG in RNA) (Russell, 1996). The stop site signifies the end of a gene and is one of three possible codons, “TAA,” “TAG,” and “TGA” (UAA, UAG, and UGA in RNA). The donor site in Figure 2 (denoted by the splice donor site) marks the boundary between exons (coding regions) and introns (non-

coding regions). Most introns start with “GT” (GU in RNA). Similarly, the acceptor site marks the boundary between intron and exon. Most introns end with “AG.”

Gene prediction is a complex process because for each of these four regions it must discern the real sites from the false or pseudo sites. For example, nearly all genes begin with the sequence “ATG,” the start codon. Predicting genes would be simple if there were only one “ATG” for every gene in the genome, but there may be tens to thousands of these codons for any given gene. Predicting which “ATG” represents the real start codon can be difficult. Fortunately, additional characteristics exist which help identify the start codons and provide clues that help predict which of these codons is the actual start of the gene. There are a number of promoter elements such as the TATA box, CAAT box, and GC box, which are present upstream of the real start codon (Russell, 1996). Looking for these elements in addition to a start codon greatly improves the accuracy of the gene prediction algorithm.

Computer Science and Gene Prediction. There are a number of mathematical models used to predict the protein-coding regions of a gene. In his review article, Costa describes a number of methods algorithms used in gene prediction. These methods include Linear Discriminant Analysis and Quadratic Discriminant Analysis, Perceptron Method, hidden Markov models, Hexamer-Coding Measures, Weight Matrix Method and Weight Array Method, Maximal-Dependence Decomposition (MDD) Donor Matrices, Decision Tree, and Artificial Neural Networks (Costa, 2003). In most of these models, the program is trained on a known set of data and utilizes that knowledge to perform gene

prediction on an unknown set of data. We survey four of these methods in more detail in the next section.

Algorithms Used for Gene Prediction

Most gene prediction programs are hybrids. These programs utilize a combination of different algorithms and yield better results than programs that use only one algorithm. For example, regardless of the core algorithm used in the method, dynamic programming is commonly used to find the maximal path or score in a set of possible paths. In describing the next four gene prediction algorithms, it is important to keep in mind that though the algorithm may provide the core framework, other algorithms are commonly used in cooperation.

Linear Discriminant Analysis. Linear Discriminant Analysis (LDA) uses differences in the characteristics between the samples in a set to separate them into groups. The key to making LDA successful lies in choosing the criteria that will be used to discriminate the sample set into categories. LDA is used in evaluating clustering of gene expression data for microarrays. Given a certain clustering, LDA can be used to confirm the quality and validity of that clustering (Mendez, Hodar, Vulpe, Gonzalez & Cambiazo, 2002). LDA can also be used to identify patterns of gene expression (Moore, Parker, & Hahn, 2001). Sparse linear discriminant analysis has been used to help discriminate cancer tissue samples (Cox, 2002). In gene prediction, frequency of hexamers and codons are often used as the discriminating function. LDA uses pre-

computed probabilities for hexamers and codons from known samples to discriminate real sites from false ones. It then uses dynamic programming to find the optimal combination of real sites to predict the coding regions of a gene (Solovyev, Salamov, & Lawrence, 1995).

An example of a program that uses LDA in gene prediction is FGENES, written by Victor Solovyev (Solovyev et al., 1995). FGENES uses LDA to recognize different types of exons, promoters, and polyA signals then uses dynamic programming to find the optimal combination of those features. Originally developed at the Sanger Institute, FGENES has become a commercial product in the company Softberry. It is advertised as a program that distinguishes itself from most other gene finding programs because it uses LDA rather than hidden Markov models and can be used as a supplemental gene predictor for exhaustive annotation tasks ("Softberry," 2002).

Hidden Markov Model. The hidden Markov model (HMM) is comprised of smaller Markov models and has been utilized to solve problems in many areas. Each of these smaller models is trained on a certain type of data which can range from facial features to gene sites. Given a set of unknown data, the hidden Markov model uses the characteristics found in that data to match the smaller model in the HMM which its characteristics fit best. It uses the information to suggest what kind of data we have. Embedded hidden Markov models have been used in face recognition. The Markov models are trained on the different regions of the face, which are then used to help in facial recognition (Kim, Kim & Lee, 2003). It has also been used to recognize

handwritten Arabic manuscripts by training a single HMM with the structural features extracted from manuscript words (Khorsheed, 2003). In gene prediction, elements can refer to sites, promoter regions, and a number of other entities. Some examples of how HMMs are used in gene prediction include HMM sampling to find alternative splicing for genes, modeling and prediction transcriptional units of *E. Coli* genes, and detecting splicing junction sites in DNA (Cawley & Pachter, 2003; Yada, Nakao, Totoki & Nakai, 1999; Yin & Wang, 2001). In the gene prediction program, there are two models, the positive model and the negative model. The positive model represents the real sites and the negative model represents the false sites. These models are first trained with sequences whose site locations are known. They are then integrated together into a hidden Markov model. The goal for HMMs in gene prediction is to classify a potential site into the positive or negative model. Given a sequence of potential sites, the hidden Markov model will predict which sites are real and which are false by finding the optimal path for those sites through the HMM. All potential sites which go through the positive model are predicted as sites for that sequence.

There are a number of packages which use hidden Markov models in gene prediction. One such package is Genscan, written by Chris Burge while at Stanford and now at MIT ("New GENSCAN," 2002). Another such package is Genie, which was developed as a joint effort between the University of Santa Cruz Computational Biology Department and the Lawrence Berkeley Laboratory Genomics Informatics Group. It utilizes a generalized hidden Markov model (GHMM) as the core framework for the recognition of human genes. As with many packages, Genie also integrates other

algorithms such as dynamic programming, which are used to select the path of highest probability, and neural networks, which are used in site prediction (“GENIE,” 1996).

Neural Networks. Neural Networks utilize many pieces of data from a sequence. In gene prediction, Markov model scores, flanking region GC composition, candidate region GC composition and scores for splicing acceptor sites are among the types of data which are used (Mount, 2001). Groups of data points relate to each other by confirming or contradicting each other. For example, if the Markov model score is high, but the candidate region GC composition was low, these datapoints would indicate a conflict. High Markov model scores typically indicate the presence of an exon while low GC composition usually indicates an intron. The conflict would be resolved at the intermediate or hidden state and the resulting signal would be forwarded to the neuron. The neuron is the decision maker in the system. It takes all the input from hidden states and generates an overall score for the sequence, determining if it is an exon or not.

GRAIL (Gene Recognition and Analysis Internet Link) is a well-known package which utilizes neural networks. It was developed by the Computational Biology Program of the Life Sciences Division of Oak Ridge National Laboratory (“ORNL Grail,” 1996).

Decision Tree System. Decision Trees are used in gene prediction to improve the effectiveness of other algorithms such as dynamic programming. It begins by considering all possible start, acceptor, donor, and stop sites and utilizes dynamic programming to find the optimal combination. Since the number of potential sites in sequences is very large, the decision tree is used to pare down the number of possible choices that are examined (Salzberg, Delcher, Fasman & Henderson, 1998).

MORGAN (Multi-frame Optimal Rule-based Gene Analyzer) is one of the few available packages which utilize the decision tree system. It was developed by Salzberg, et al. at The Institute for Genomic Research (TIGR) ("The MORGAN," 1998).

Linear Discriminant Analysis

Overview. Discriminant Analysis is a general statistical method for classification. The problem can be stated as follows: "Given N objects, how can one assign each object into K known classes with minimum errors?" (Zhang, 2000) Linear Discriminant Analysis (LDA) is a dimensionality reduction technique which reduces each element to a single data point or score by utilizing the different characteristics of each element. For example, if we wanted to classify a set of bats into two categories, we would sample a set of characteristics such as wingspan, length of teeth, size of eyes, and pitch. Studies that have been performed use 12 acoustic variables in the discriminant function. Based on similarities and differences in these characteristics, bats were classified into their different cave colonies (Boughman, Wenrick & Gerald, 1998). The idea is similar in gene prediction where the objects are sites and the assignment is two-fold. First, sites are

categorized into one of the four possible types – start, stop, acceptor, and donor. Then sites must be categorized as real sites or pseudo sites. The most general function used for LDA is: $F(x) = a * X_i$, where $F(x)$ is the overall score we are trying to calculate. In the formula, “a” represents a regional coefficient score, and the “ X_i ’s” represent the key regions to be analyzed. The following example shows how Discriminant Analysis can be used to separate people in different diabetes risk categories from each other by examining the correlation between weight and height.

Hypothetical Example. The simplest thing to do is to look at one characteristic and plot it. Let us consider weight as an example. We can assume that the heavier the person, the more likely they are at risk for diabetes.

Figure 1 – Hypothetical weight distribution of 100 people.

The data in Figure 1 suggests that anyone over 250 lbs. has a high risk for diabetes, while someone who weighs less than 175 lbs. is at low risk. The claim, however, may not be true, as two people with the same weight may have dramatically different body types. In general, taller people will typically be heavier than shorter people so looking at weight only may not give an accurate measurement of risk.

Suppose we look at height as the sole factor in determining risk for diabetes. According to Figure 2, subjects who are short are at high risk for diabetes and those who are tall have low risk for diabetes. We could generalize and assume that people under 5'

are at higher risk than people taller than 6'. But in the same way as with weight, the assumption would probably be inaccurate. Two people who are 4' tall may have very different risks based on their weight. Looking at weight or height individually does not provide the discrimination between those that are at high and low risk.

Figure 2 – Hypothetical height distribution of 100 people.

The key to gaining a clear separation of ambiguous data in the experiment is to use both pieces of information together, which will help discriminate those at risk from those with very little risk.

Figure 3 – Hypothetical height and weight distribution of 100 people.

Figure 3 shows how using two characteristics together, rather than each individually, can help to discriminate two (or three) classes of people with more clarity. In a similar fashion, Linear Discriminant Analysis attempts to utilize various characteristics of genes and pseudo genes to separate the two categories. Dynamic programming is then utilized to determine the most probable arrangement of the exons coding for the gene.

In eukaryotic and prokaryotic organisms, there are start and stop sites which mark the beginning and end of a gene. Since genes in eukaryotic mammalian organisms often

consist of multiple exons, we can have many acceptor and donor sites. These are the four types of regions that we are interested in classifying. In classifying each region or site, we must determine whether it is an authentic site or a pseudo site. The task is difficult because rules that classify each of these sites are quite general and, as a result, will group real and pseudo sites together. For example, a donor site is characterized as having “GT” immediately downstream of the site. With only four possible bases for each site, there is a $1/16$ or 6.25% chance of getting “GT” randomly. With 99% of the genome not coding for any genes, it would be very unlikely that 6.25% of the genome consists of donor sites. In order to separate the real sites from the pseudo sites, Linear Discriminant Analysis and dynamic programming are utilized.

LDA Weight Tables. The purpose of weight tables is to provide a way to represent and store data from sequences whose site locations are known. Though the nomenclature used between programs may be different, the content is roughly the same. The data is utilized in Linear Discriminant Analysis to predict the potential sites in sequences for which the information is unknown.

In gene prediction, weight tables store information on the frequency of codons and hexamers found flanking key regions. Though the format of weight tables may vary depending on the program utilizing the tables, the type of data stored is similar. For example, Figure 4 is part of the weight table that was generated by the program. The first column describes the site being scored and the second column is the actual score or frequency. The key regions we are interested in are the start, stop, acceptor, and donor

sites. The start site is abbreviated “STRT,” the stop site is represented by “STOP,” acceptor sites are represented by “ACPT,” and donor sites are represented by “DONR.” We are interested in the regions surrounding each of these sites. “UP” stands for upstream, and represents the codon or hexamer preceding the site, while “DN” stands for downstream, which represents the codon or hexamer following the site. For each combination of site and relative location, the frequency of codons and hexamers, represented by “3” and “6” respectively, are recorded.

STRT.DN.3.P.ATG	0.9775949242
ACPT.UP.3.P.CAG	0.6605731225
STOP.UP.3.P.TGA	0.4872583275
DONR.DN.3.P.GTA	0.4525691700
DONR.DN.3.P.GTG	0.4095849802
STOP.UP.3.N.TGA	0.4025178505
STOP.UP.3.N.TAA	0.3350527800
ACPT.UP.3.N.GTG	0.3262714871
DONR.UP.3.N.AGA	0.3176620521
ACPT.UP.3.N.GTT	0.2655672885

Figure 4 – Sample LDA Weight Table.

We begin by using sequence data for which we know the location of the sites, so we can calculate the probability that certain codons or hexamers appear at each of these key regions. The probabilities associated with these codons and hexamers are designated with a “P,” meaning that it suggests the presence of a real site. For example, according to Figure 4, in 98% of the time, the codon “ATG” appears downstream of real start site in sequences. Conversely, we also calculate the probability of false positives or pseudo-

sites. Pseudo-sites are regions of the sequences where false sites mimic real sites. For example, when we talk about start sites, we know that it must begin with the codon “ATG.” However, not every occurrence of “ATG” represents a start site. We find the probability by counting all occurrences of “ATG” in the known sequence, subtracting one for the real start site, then dividing by the total number of start codons that appear in the sequence. The probability is designated by an “N,” which stands for negative because it represents the cases where we have a pseudo or false site. Using these scores, each potential site is scored and the maximal overall score for the sequence is generated. The sites that contributed to the score are recorded and reported. The combination of these sites would represent the program’s best guess of the sites for the sequence.

Advantages. The main advantage to using Linear Discriminant Analysis (LDA) to predict genes is that the process is straightforward and relatively simple. Of all the algorithms used in gene prediction, LDA is probably the easiest to understand. As with many other algorithms that are used for gene prediction, LDA consists of two steps – training the program on a set of known data, then implementing an algorithm to predict genes on an unknown set of data. In generating the training data (weight tables), recall that the core of the process consists of:

1. using known coordinates for each of the sites to count codons and hexamers which are present upstream and downstream of those coordinates,
2. counting all possible sites which are not real (coordinates which do not exist in the set of known sites), and

3. dividing by the total sites counted at each site type.

The second step begins with an unknown set of sequences in FASTA format. Each sequence is read in a similar fashion to the first step. Each possible site encountered is identified and a table lookup for the codons and hexamers in the regions flanking that site returns a set of scores which are summed to give that site a regional score. The final step consists in determining an overall score for the whole sequence and can be accomplished in a number of ways. The most popular method uses dynamic programming to generate the score because it gives an exact solution efficiently. It is possible to use a greedy algorithm, which would be simpler to implement, but may not give the optimal solution.

Disadvantages. Though LDA is easy to understand and straightforward to implement, it is generally not as accurate in predicting sites in genes as some of its counterparts such as hidden Markov models (HMM). LDA utilizes the 12 nucleotides flanking each site to generate a regional score by looking at those nucleotides in sets of three and six. Using these codons and hexamers, it combines the individual scores for each of them to produce a score for that site. These scores are used to help predict which sites in the sequence are real and distinguish them from pseudo sites. In the model, a single nucleotide change in the 12 nucleotide region surrounding the site can dramatically change the score associated with that site. For example, in the sequence “ACCGCGATGACC,” if the first “G” is changed to an “A,” the upstream codon and hexamer scores will change. Originally these scores would be associated with the hexamer “ACCGCG” and the codon “GCG.” However, after changing one base, the

upstream scores are now associated with the hexamer “ACCACG” and the codon “ACG.”

The LDA method does not adapt to single nucleotide changes very well. Other methods utilizing models such as the hidden Markov model can utilize that same window of 12 nucleotides with more sensitivity. Rather than calculating scores based on groups of three or six nucleotides, it calculates scores for positions of each individual nucleotide and combines those scores. A single nucleotide change in the window of 12 nucleotides will likely have a much smaller effect on the overall score for that site than would be seen when using LDA. Hidden Markov models will be discussed in more depth in the following section.

Hidden Markov Models

Markov Chains. Before introducing hidden Markov models (HMM), Markov chains should be defined. They are the basic building blocks of the hidden Markov model. Markov chains are similar to state diagrams, for which the definition of each state depends on what is being described. Markov chains can be shown graphically as a collection of “states,” each of which corresponds to a residue, with arrows between the states (Durbin, Eddy, Krogh & Mitchison, 1999). The arrows between the states are called transitional probabilities. They represent the likelihood of going from one state to the next. Markov chains, which are also known as Markov models consist of states and transitions. The states can represent everything from a geographical location to a nucleotide depending on what is being modeled. Consider the example in Figure 5.

There is a Markov chain representing the decision that a dog owner must make about whether to let the dog in the house or not.

Figure 5 – Markov chain representing a dog owner’s decision.

In Figure 5, there are two possibilities, Yes and No. Between the possible choices, there are arrows connecting them. These are the transition probabilities. In the example, the arrow going from “Yes” to “No” represents the likelihood that the owner, who would initially let the dog in, would change his mind and decide against letting the dog in. Similarly, the arrow from “No” to “Yes” represents the likelihood that the owner who originally would not let the dog in would change his mind to let the dog in. Lastly, observe that there are arrows from “Yes” to itself and “No” to itself. These represent the likelihood that the owner will not change his mind after deciding to let to dog in or not.

The model will look the same regardless of the type of dog which is owned. However, the transition probabilities between the states (decisions) will change. If the dog is a small dog, which is predominately an indoor dog, then the owner will probably let it in the house more often. As a result, the transition probabilities from “Yes” to “No”

will be small and from “Yes” to “Yes” will be large. However, if the dog is a big dog, which is predominately an outdoor dog, then the owner will keep the dog outside most of the time. Consequently, the transition probability from “No” to “No” will be high and from “No” to “Yes” will be low. Figure 6 represents the Markov model for an owner with small dogs and an owner with large dogs.

Figure 6 – Transition probabilities for owners with small and large dogs.

Notice that the probabilities are not absolute. There is still a chance that a small dog owner will decide to leave the dog outside and that a big dog owner will let the dog in since some people do not mind having big dogs in the house and others do not like any dogs in the house (even small ones). It may also be due to the fact that some small dogs are very destructive and can cause a lot of damage to a house while some big dogs are very well behaved. Many factors play a role in whether a dog owner will or will not let a dog in the house, but on average, owners of big and small dog display different behaviors.

Markov chains can be powerful tools in representing complex models. They were first used for speech recognition and more recently have been used in the bioinformatics

community for a number of problems, including gene prediction, motif recognition, pairwise alignment, and multiple sequence alignment (Durbin et al., 1999).

Figure 7 is an example of a Markov chain for DNA. Since DNA consists of four possible nucleotides (A, T, G, C), the Markov chain has four states to represent each nucleotide. Arrows that connect the states represent the likelihood of going from one nucleotide to the next. For a sequence that is being read, if the last nucleotide read was an “A,” each arrow coming out of state “A” represents the probability of what the next nucleotide will be. For example, the arrow from “A” to “T” tells us the probability that “T” will be the next nucleotide read, given that “A” is the current nucleotide. When using a Markov chain to calculate the probability that a nucleotide may appear in a certain position, that probability is only dependent on the nucleotide in the previous position.

Figure 7 – Markov chain representing the DNA alphabet (Durbin et al., 1999).

One example of how Markov chains are used in bioinformatics is the identification of CpG islands. CpG islands are stretches of DNA with high

concentrations of the Cytosine and Guanine. In a DNA double helix chain, Cytosine and Guanine are complementary bases bonded to each other with hydrogen bonds (“Base Pairing,” 1999). They are commonly abbreviated CG or C-G, so the “p” in CpG islands was added to distinguish between them from the CG base pairing. CpG islands are rare in DNA because Cytosine undergoes methylation to a methyl-C, which has a high probability of being mutated to a Thymine. However, in the regions upstream of the start codon for a gene, methylation is suppressed. As a result, the upstream regions, which are also called promoter regions, have a higher than normal concentration of CpG islands.

The information can be used to model the regions upstream of a start site. A first order Markov chain is needed. In first order Markov chains, the current state depends on the previous state. (In a second order Markov chain, the current state would depend on the previous two states.) We are interested in the probabilities of seeing a Guanine after a Cytosine. In a Markov chain modeling the regions upstream of a start site, we expect the probability of seeing a Cytosine to be high relative to a Markov chain modeling other regions. In order to obtain these probabilities, the Markov model is trained on a set of sequences whose start site is known. Two Markov chain models are trained. One is trained on the regions upstream of a start site and the other is trained on regions with pseudo start sites providing the likelihood or transition probabilities for each nucleotide, given the nucleotide immediately preceding it. These data are then used on a sequence to predict which region or regions contain true CpG islands (Durbin et al., 1999).

Hidden Markov Models. Hidden Markov models (HMM) are built from Markov chains, but they differ in one important aspect. While Markov chains have one state for each symbol, hidden Markov models have more than one state for each symbol. In other words, HMMs do not have a one-to-one correspondence between states and symbols (Durbin et al., 1999). Given a current symbol, you may not know which state that symbol corresponds to because there may be more than one state with that symbol. The reason is because of the structure of the HMM. HMMs are constructed by combining two or more Markov models. Each state in each Markov model is interconnected with the states in the other Markov models. For example, Figure 8 shows a HMM which was created by combining the owner models with big and small dogs. If we only know that the dog owner decided to let the dog in, then we cannot determine the which “Yes” state is being referred to. The state is “hidden,” which is why these models are called hidden Markov models. The dotted lines between the top and bottom models represent the transitional probabilities between Markov models.

Figure 8 – Hidden Markov model representing the dog owner models.

The lines are dotted to show that these transitional probabilities are small, relative to the transitional probabilities within a model. In the example, a transition between Markov models represent the likelihood that one dog owner would change his behavior to mimic the behavior of the other type dog owner. It is unlikely that a person who owns a big dog would begin to treat his dog as like a small dog, which is why the transitional probabilities between the two states are very small. However, by integrating the two models into one, the need for two models is eliminated.

In the previous example with CpG islands, a hidden Markov model builds one model rather than the two built by the Markov chain. The way that it is accomplished is by creating two states for each nucleotide. One state represents the nucleotide in a real CpG island and the second state represents the nucleotide in a non-CpG island region. The model, shown in Figure 9, is the combination of two of the Markov models shown in

Figure 7, one for the positive and one for the negative. The positive (+) model represents the nucleotides in the CpG island region and the negative (-) model represents the nucleotides in the non-CpG island region. These two models are connected by transition probabilities between positive and negative nucleotide states. Generally, the transition probabilities between positive and negative nucleotide states are small relative to the transition probabilities of positive only or negative only states. As a result, the likelihood of an A+ transition to a C- is less likely than the A+ transitioning to C+.

Figure 9 – Hidden Markov model, an extension of the Markov chain.

Advantages. Using hidden Markov models to predict genes offers a number of advantages over other methods. The two main advantages are sensitivity and space. HMMs are more sensitive than other algorithms such as LDA because for each site in a sequence, two paths can be considered. They contrast methods such as LDA where one score is calculated for each site and the decision of whether to consider it is based largely on that score. HMMs allow you to choose the path which labels potential sites as a real site (positive model) or a false site (negative model). The differences between the scores in the positive model and negative model often indicate the quality of a site. If the score

for the positive model is much larger than the score for the negative model, the potential site is more likely to be a real site rather than a false site. Conversely, if the difference between the two scores is negligible or if the negative model score is greater, it suggests that it would be difficult to distinguish between a real and pseudo site.

The second advantage is the efficiency of using scores in the HMM table. While Linear Discriminant Analysis requires scores for each possible hexamer and codon at each site type, HMMs use scores for each position surrounding each site type. The difference in the amount of space used to store each table is substantial. LDA requires scores associated with each possible hexamer because in a sequence any combination of six nucleotides may appear, though the majority of them may never occur. One site corresponds to 4^6 or 4096 scores. HMMs on the other hand, require only 6×4 or 24 scores for that same window. There may be a few scores rarely used in the model, but the percentage of scores utilized is much greater than in the LDA model.

Disadvantages. HMMs are more sensitive in evaluating sites because all possible combinations of sites are considered. As the length of the sequence increases, the number of potential sites which are examined increases linearly and the number of possible paths increases exponentially. Consequently, there is a point where it is unreasonable to examine all sites because it would take too long for the program to run. There are optimization techniques that have been developed to address the problem, some of which will be discussed in later sections.

The other disadvantage is the use of one score to determine the best path through all potential sites in the sequence. This does not leave room for error in selecting the real sites. For example, the correct set of sites may be the second or third best score, but the method does not allow you to select them. The number of instances where the wrong sites are picked can be reduced by iteratively training the HMM, a method which is commonly used when utilizing the HMM in gene prediction. The next section will describe the data sources that were used to train and test these two algorithms.

Materials and Methods

Data Sources

Since each species is different, it is important to generate weight tables with known data from the same or similar species to obtain accurate results. Selections from Genbank® release 136, rodent files were used, specifically, the *Mus musculus* (mouse) and *Rattus norvegicus* (rat) cDNA sequences. Genbank® is the NIH (National Institute of Health) genetic sequence database; an annotated collection of all publicly available DNA sequences.

Data from Genbank® was parsed into two files, the FASTA file and the CDS (coding sequence) file. The FASTA file contains each complete nucleotide sequence and the CDS file contains the coordinates for the coding regions on each of those sequences. Each entry (sequence) in the FASTA file represents the genomic DNA corresponding to a protein. Some entries have more than one protein associated with them because of potential splice variants.

Genbank® was chosen because it is widely recognized by pharmaceutical companies and academia as the central comprehensive location to retrieve sequence information for various species, such as mouse and rat. Mouse and rat were chosen because they are closely related (both categorized under Rodent) and both have been studied extensively. Since mouse had a larger sample size (55K sequences versus 15K sequences for rat), it was chosen as the source to generate the calibration data (weight tables).

When the weight and HMM tables were generated, inconsistencies were found in the files from Genbank®. Since data from an external source (Genbank®) was used, the data was examined closely to ensure that results were accurate. The format and terminology used in the data had to be understood first so that the programs, which evaluate the data, could do so correctly. If the inconsistency affected a substantial amount of the data, the program was modified to accommodate the data. After analyzing the Genbank® CDS (coding sequence) file and comparing it to the Genbank® FASTA file, a number of adjustments were made. Points 1-4 were additions made to the programs that generated the LDA weight tables and the HMM tables. Points 5 and 6 were adjustments made to the datafiles.

1. Some genes were marked as “complement,” meaning they were sequenced in the complementary direction. The start site is represented by the last coordinate and the stop codon represented by the first. The sequence had to be reverse complemented which meant switching A’s and T’s and switching G’s and C’s. Functions were added to the programs to reverse complement these sequences.
2. Some nucleotide sequences had multiple proteins associated with them as a result of splice variants (the sequence can be spliced at different points and stitched together, resulting in different proteins). Each splice variant of the nucleotide was considered.
3. Some nucleotide sequences contained the character “N” which represents an ambiguous base call. Any codon or hexamer containing “N’s” was ignored and

not included in any calculations. For example, if there was a potential donor splice site with the codon “ATN,” the codon was omitted and not considered in the calculation of site scores.

4. Some nucleotide sequences contained the complete coding region of the gene which can occur if the sequence is short or if the coding region is long. There may be a start site, but the sequence stops before reaching the stop site. Only the start site was considered in these calculations.
5. Formatting anomalies were identified and omitted. The format of the coding regions file consisted of pairs of sites. The number of sites ranged from one pair to 20 or 30 pairs for any given sequence. One pair of sites defines a gene which contains no donor or acceptor sites. Sequences with more than one pair of sites defined genes with at least one set of donor and acceptor sites. Each pair of sites had the format “coordinate .. coordinate,” however, there were some cases in the mouse CDS file where the pair contained only the first coordinate. For example, the sequence g53114 had the following data:

```
g53114    g1334125          join(64,228..497,685..>951) .
```

In the example, the coordinate 64 was missing its coordinate pair.

6. Any sequences that were not in both the FASTA file and CDS file were also omitted because the sequences that did not have reported coding regions would skew the probabilities. For example, if a sequence had a start and stop site, which were not reported in the CDS file, those real sites would be counted as pseudo

sites and would have contributed to the negative scores for that site rather than the positive scores.

The filtering of the training set data reduced the number of sequences from roughly 55,000 to 33,000 sequences. Figure 10 shows a length distribution of these mouse sequences. Over 12,000 sequences were of lengths that range between 250 and 500 base pairs, while there were around 2,000 sequences whose lengths varied between 2,000 and 2,250 base pairs.

Figure 10 – Length distribution of mouse sequences used in training set.

Though the majority of the sequences were less than 500 base pairs long, some sequences were over 5,000 base pairs long. The longest mouse sequence was 461,875 base pairs long and the average length was 4,957 base pairs. The total number of bases of all mouse sequences was 273Mb (megabases).

Figure 11 shows the length distribution for the rat sequences that were used as the testing set. The graph has a similar shape to that of the mouse sequences, with the majority of sequences being greater than 500 base pairs, and some greater than 5,000 bases pairs. The longest rat sequence was 189,137 base pairs, and the average sequence length was 1,870 base pairs. The total number of bases of all rat sequences was 27Mb (megabases). Though the rat sequences show a similar profile to that of the mouse sequences, they are generally shorter as evidenced by the average length. The set of mouse sequences is a better candidate to use as a training set because there are more sequences and because those sequences are generally longer. Longer sequences tend to contain longer coding regions and consequently, more exons and sites. Using a greater number of sites to train the HMM and LDA tables generates more accurate tables. The next two sections will discuss the details of the implementation of the LDA and HMM algorithms.

Figure 11 – Length distribution of rat sequences used in testing set.

Implementation of LDA Procedure

The LDA algorithm consists of two parts. First, weight tables are generated. Weight tables are probability tables for each site that are based on a known set of data. Then, these tables are used to calculate scores for an unknown set of sequences and make a prediction on the presence and location of genes.

Weight Tables. The purpose of weight tables is to discriminate between real sites and pseudo sites. The first step is using a set of sequences for which the coding regions are known to generate the weight tables. Genbank® release 136 mouse sequences were used to generate the weight tables. Characteristics at each splicing region are recorded. Codons are strings of three nucleotides and hexamers are strings of six nucleotides. In LDA, these characteristics are upstream and downstream codons and hexamers. Upstream and downstream are terms that refer to the nucleotides before and after the site, respectively. The coding regions tell us the coordinates on the genomic DNA that will be cut, resulting in the pieces that stitch together to form the gene. The first coordinate corresponds to the start site and the last coordinate corresponds to the stop site. All sets of coordinates between the start and stop represent the acceptor and donor sites.

Figure 12 – Characteristics recorded for each site.

At each of these sites, the frequency of each codon (triplet) and hexamer are recorded. For example, in Figure 12, “TCC” is tallied for upstream codon and “ACCTCC” is tallied for upstream hexamer. The process is repeated for all the sequences in the known set and then the total number of each upstream and downstream

codon and hexamer is divided by the total number of each tallied type. The resulting data comprises the positive scores for the weight table. These scores represent the frequency of each possible codon and hexamer in a real gene. Figure 13 details the process by which the weight tables were created.

Figure 13 – Generating the LDA Weight Table process.

First, the positive scores are calculated. Coordinates for each sequence are taken from the CDS file and the codons and hexamers surrounding that site are scored. These scores contribute to the positive score table. After the positive scores have been tabulated, the negative scores are calculated. The remaining sites (those which are not in the CDS file) are taken and the codons and hexamers that surround each site are scored. These scores create the negative score matrix.

For example, at the start site, a sequence from the known coding region set is taken. Each occurrence of “ATG” would be examined except for the one that is specified as the start site by the coding region coordinate. At each of these start sites, the upstream and downstream codons and hexamers are recorded and tabulated. After all sequences are evaluated, the frequency of each codon and hexamer at each site is calculated, providing the negative scores for the weight table. These scores represent the frequency of each possible codon and hexamer in pseudo sites.

Originally, the process rounded the scores off to the nearest percent, however, many scores were too small to be detected at the established level of granularity, so the precision was extended to 10 decimal places.

After the weight tables are generated, they are evaluated to assure that they make sense biologically. Two facts that are known are that “ATG” is the start codon and “TAG, TAA, TGA” are the stop codons. It is expected that “ATG” appears downstream of the start site at a high frequency and “TAG, TAA, TGA” appears upstream of the stop site at high frequencies. In the generated weight tables, “ATG” appears as the codon downstream of the start site 98% of the time and the sum of the frequencies for which the

stop codons appear upstream of the stop site is 98%. Figure 14 shows the top scoring codons from the weight tables which were generated using a training set of 36,504 mouse sequences. The table had a total of 66,560 scores, representing 4,096 hexamers and 64 codons upstream and downstream at each of the four types of sites.

STRT.DN.3.N.ATG	1.0000000000
STRT.DN.3.P.ATG	0.9775938135
ACPT.UP.3.P.CAG	0.6605731225
STOP.UP.3.P.TGA	0.4872810287
DONR.DN.3.P.GTA	0.4525691700
STOP.UP.3.N.TGA	0.4248533329
DONR.DN.3.P.GTG	0.4095849802
ACPT.UP.3.N.GTG	0.3372163841
STOP.UP.3.N.TAA	0.3181582720
DONR.UP.3.N.AGA	0.3125055894

Figure 14 – Top scoring codons from LDA weight table.

Linear Discriminant Analysis. LDA was implemented in C because of its speed and widespread use in the UNIX environment. The implemented algorithm can be divided into five distinct steps, which are outlined in Figure 15:

- 1) Genomic Input
- 2) Key Region Analysis (Process Frames)
- 3) Regional Score Calculation
- 4) Frame Score Calculation (Find Maximal Scoring Path)
- 5) Overall Maximal Scoring Path

Figure 15 - LDA program flowchart.

Steps 1-4 are executed for each of the three reading frames. Step 5 is performed after all three reading frames have been processed. Step 1 is straightforward and the only requirement is that the file be in FASTA format. Once the genomic sequence has been

read, analysis can proceed. Step 2 involves recognizing all the key regions considered in the model, namely, the start (ATG), donor (GT), acceptor (AG), and stop (TAG, TGA, TAA) sites. These are used in later steps to find a maximal scoring path. Finding the maximal scoring path consists in scoring individual sites and determining if that site can improve the likelihood of that path being real. These are steps 3 and 4. Let us consider the following example for scoring a site:

Positive Upstream Codon = 0.75
Negative Upstream Codon = 0.25
Positive Upstream Hexamer = 0.40
Negative Upstream Hexamer = 0.15
Positive Downstream Codon = 0.50
Negative Downstream Codon = 0.45
Positive Downstream Hexamer = 0.60
Negative Downstream Hexamer = 0.60

The negative scores are subtracted from the corresponding positive scores and summed to yield a net score for the identified region. In this example:

$$(0.75 - 0.25) + (0.40 - 0.15) + (0.50 - 0.45) + (0.60 - 0.60) = \mathbf{0.80}$$

In the example, 0.80 is the regional score. These steps are repeated for the length of the genomic sequence.

The final step involves implementing a dynamic programming algorithm to find the maximal total score for the system. After establishing the first start site, the dynamic programming algorithm is used to find the maximum possible score. The algorithm is outlined in Figure 17. The sequence is processed by visiting all sites in their order of occurrence within each frame. The type of site that was last added determines the course of action for the next site encountered. For example, if a donor site was the last site added to the path, and another donor site is seen, the current donor site will replace the

previous donor site only if the current site's score is greater than the previous site's score. For each site encountered, the current site score is compared to scores of previously added sites to determine if adding the current site will increase the overall score. If it does increase the overall score, some previously added site may be removed and the current site will be added. Special weight is given to the stop site because without a stop site, the possible gene prediction would be invalid. If the maximal score was obtained with a set of sites that did not contain a stop site, a rollback function is utilized to adjust the score and sites to the last stop site encountered. The maximal scoring path for the reading frame is recorded. After the process is completed for each of the three reading frames, the maximal scoring paths from each frame are compared to each other to determine which prediction is selected. The prediction that covers the largest percentage of the sequence is selected. When none of the three reading frames make a prediction, " no genes found" is reported. Figure 16 shows a sample gene prediction generated from the LDA Gene Prediction Program. The first column is the sequence for which the prediction is being made. Column two is the score for the best prediction. Column three is the frame from which the prediction came. Column four is the predicted sites. If no predictions are made, "No Possible Genes" is returned.

g1001919	1.067927	2	20..223
g1001920	1.045804	2	50..130
g1001921	0.000000	1	No Possible Genes
g1001922	1.045421	3	66..185
g1001926	2.298060	3	288..1989 2025..2102
g1001928	1.060599	2	170..196

Figure 16 – Sample output from LDA program.

Figure 17 – LDA algorithm used to calculate maximal frame score.

Adjustments to LDA Program. Following the first run, the program was over-predicting sites, meaning that it was predicting more sites than the Genbank® CDS data listed. Threshold cutoffs were applied for each site type to filter out sites which were likely to be false. For example, in order for a start site to be considered, the local score had to be greater than 0.9. The adjustment greatly improved the over-prediction of the sites, however, the program still over-predicted sites.

Start sites at the beginning of the sequence and stop sites at the end of the sequence are treated as special cases. For the start site, it is not possible to add the upstream codon and hexamer to the regional score. For the stop site, it is not possible to add the downstream codon and hexamer to the regional score, which substantially decreases the score associated with the stop site. It does not however decrease the score of the start site by much.

Implementation of HMM Procedure

Using hidden Markov models to predict genes is basically a two-step process. The first step is to train the hidden Markov model and the second step is to use that model to evaluate all potential sites in a sequence to determine which ones are real.

Training the Hidden Markov Model. In order to use hidden Markov models to predict genes, the model must first be trained. A set of sequences that contain genes with known site locations is taken. *Mus musculus* (mouse) sequences from Genbank® release 136 were used to train the HMM. Originally, the dataset contained 55,151 sequences,

which was reduced down to 36,504 sequences after filtering. As mentioned in Section 6, sequences with no CDS (coding sequence) information are filtered out. A window of 12 nucleotides, which surrounded the start, stop, acceptor and donor sites, were used as the training region. The occurrence frequency was computed for each of the six nucleotides upstream and downstream of the site. For the positive model, known sites from the mouse sequences were used and the frequency of occurrence of each nucleotide at each position was counted. The remaining sites that resembled each site (ATG for start, TAG, TGA, and TAA for stop, GT* for donor, and *AG for acceptor) were used to train the negative model for each of the false sites. Figure 18 gives an example of how the scores for the Markov model are obtained from a set of four sequences. The example assumes that the dataset used to train the Markov models consists of four sequences. The 12 positions for which scores are calculated begin with the nucleotides that occur six bases upstream of the site and end with the nucleotides that occur six bases downstream of the site. In the example, the start site is identified by the “atg” immediately following the site in all four sequences. In position 1, there are two Cytosines, one Guanine, and one Thymine. The resulting scores for the position can be seen in the position 1 box. $A = 0.00$, $T = 0.25$, $G = 0.25$ and $C = 0.50$. Scores for the other 11 positions are calculated in the same fashion. Notice that in position 9, the scores for A, T, and C are 0.00 while the score for G is 1.00 which means that in these sequences, G appears 100% of the time at position 9 for start sites. The score makes sense because G is the third nucleotide of the start codon which starts with A at position 7 and continues to positions 8 and 9 for T and G, respectively.

Figure 18 - Generated Markov model scores for start codon.

After both the positive and negative Markov models were trained for start, stop, acceptor, and donor sites, they were integrated together by adding transition probabilities between the positive and negative models. Transition probabilities of 0.01 from positive to negative model and 0.02 from negative to positive model were used because the transition from a negative state to a positive state is more likely. The remaining probabilities represent the probability of transition between positive states (0.99) and negative states (0.98). The generated hidden Markov model is shown in Figure 19.

Figure 19 – Hidden Markov model used for gene prediction.

Two issues needed to be addressed before continuing. The first issue was that in some positions, one base was seen 100% of the time, resulting in the other three bases having scores of 0.00. If these scores were multiplied to give a score to a particular site, it would result in a score of 0.00. To address the issue, a value of one was added to the totals of each base (and four to the overall total) and divided by the new total to ensure that all scores were nonzero. This procedure is known as pseudocount (Durbin et al., 1999). The next adjustment made was the conversion of all scores to log base 10, which improved processing speed in subsequent steps by replacing multiplication of scores with addition of scores and prevented potential underflow. Part of the generated HMM table

is shown in Figure 20. The label for each score consists of four parts. The first part of the label describes the type of site: STRT = start, STOP = stop, DONR = donor, and ACPT = acceptor. The second part identifies which model the score is associated with: P = positive model and N = negative model. The third part tells which position of the 12 nucleotide window the score corresponds to. The last element tells which base is being scored. The table shows the nucleotide frequencies for positions 7, 8, and 9 surrounding the start codon. Since these scores are Log base 10 of the original frequencies, the highest possible value is 0 (Log (1.00)). Notice the scores for A at position 7, T at position 8, and G at position 9. These scores are substantially greater than the scores for any of the other nucleotides at those positions because these ATG is the start codon which starts at position 7 and ends at position 9. The data shows that seeing an A-T-G at positions 7-8-9 is the most likely combination that will be seen. The HMM table has a total of 384 entries, representing the 12 positions and 4 possible bases at each site for the positive and negative models.

STRT.P.9.T	-4.85646688361536
STRT.P.9.G	-0.0307096027543568
STRT.P.9.C	-4.22644078951201
STRT.P.9.A	-4.83021865754042
STRT.P.8.T	-0.0334009344512678
STRT.P.8.G	-4.32146772331529
STRT.P.8.C	-4.8671621727321
STRT.P.8.A	-4.42302792346479
STRT.P.7.T	-4.56745590731352
STRT.P.7.G	-4.29079047106824
STRT.P.7.C	-4.85646688361536
STRT.P.7.A	-0.0322893481888828

Figure 20 – Sample HMM table.

Utilizing HMMs to Predict Genes. The program consists of five major steps,

which are detailed in the Figure 21:

Figure 21 – HMM program main process flowchart.

The first step is relatively straightforward. It involves reading in the sequence and passing it to the next part of the program. The next step is processing the three reading frames. Here, all potential sites are identified, then all legitimate paths in each frame are identified and scored. If any predictions were made in any frame, the one that covers the greatest percentage of the sequence will be selected. If there were no predictions made, the program will report that no sites were found.

The number of possible paths that can be taken given N potential sites is 2^N since for each site, there are two possibilities – the site could go through the positive or negative model. As the length of the sequence increases, the number of sites will likely increase and the potential paths that need to be scored will grow exponentially. By scoring only the legitimate paths, the amount of processing is greatly reduced without the danger of losing legitimate predictions. The algorithm that is used to find and score all legitimate paths is shown in Figure 22. Finding all legitimate paths can be accomplished by following the algorithm shown in Figure 22. There are a few basic rules which it follows:

1. A Start site can be followed by a Stop or Donor site.
2. A Donor site must be followed by an Acceptor site.
3. An Acceptor site may be followed by a Donor or Stop site.
4. If a Stop site is reached, that path ends.

In addition to finding legitimate paths, the algorithm also keeps track of the maximum score for each frame and the best overall prediction. Within a frame, the prediction with the highest score will be selected. The final prediction is selected from

the best predictions in each frame. As previously mentioned, the criteria used for the prediction is the greatest percentage of sequence covered.

Figure 22 - HMM program algorithm.

After all paths are evaluated and the legitimate paths are found, each legitimate path is scored. A path consists of all potential sites in a particular reading frame and whether that site is included or omitted from the solution. In the previous example, the path Start 1 → Stop 8 (leftmost path in tree, in bold) means that Sites 1 and 8 are selected and Sites 2, 3, 4, 5, 6, 7, 9, and 10 are not selected. To generate a score associated with the example, the scores for Start Site 1 and Stop Site 8 are calculated in the positive Markov models for start and stop respectively. Then, the scores for the remaining sites are calculated by using the negative models for those sites. These scores are summed and transition probabilities are added, giving the overall score for the path. In the path, the transition probability between Start 1 and Start 2 and between Stop 8 and Stop9 and would be 0.01 because we are transitioning from the positive model to the negative model. The transition probabilities between Acceptor 7 and Stop 8 would be 0.02 because we are transitioning from the negative to positive model and the transitions between Sites 2-7 and 9-10 would be 0.98 because we are transitioning between negative models. The process is repeated for each of the three reading frames, which gives three possible predictions. Then, the best scoring paths from the three reading frames are compared to each other and the path that covers the most bases is selected as the one that contains the predicted sites for that particular sequence.

Figure 24 shows a sample output from the HMM Gene Prediction Program. The first column is the Genbank ID for the sequence for which a prediction is being made. The second column is the sites for the predicted gene. “No legitimate sites found” is displayed for sequences where no predictions are made.

g1000447	53..1591
g1000679	913..1519,1786..1897,3298..3375
g1001918	74..142
g1001919	293..334
g1001920	50..130
g1001921	No legitimate sites found
g1001922	41..58
g1001923	155..229
g1001924	152..238
g1001925	74..142

Figure 24 – Sample output from HMM program.

Adjustments to HMM Program. After completing the program and running it, one major problem was encountered. There were sequences in the dataset which were very long and as a result contained over 32,000 potential sites. The number of legitimate paths that resulted from these sequences would have taken a substantial amount of time for the program to evaluate. Two adjustments were made to optimize the number of paths the program would have to examine.

The first adjustment was to filter the potential sites by score. The adjustment was done by taking the difference between the scores generated by feeding the 12 nucleotide region surrounding the site through the positive model and the negative model. The premise is, being substantially greater in the positive model compared to the negative model suggests that the particular site is more likely real. A small difference between the models suggests that it would be difficult to determine if that site was real or not.

Though the first adjustment reduced the number of paths that the program had to evaluate, there were still cases where there were too many paths and it took too long to evaluate. The second adjustment that was made filtered each sequence based on the number of adjusted (from the first part) sites it had. Numerous tests were conducted to maximize the number of sites while allowing the program to complete in a reasonable amount of time. A threshold of 200 sites was implemented to allow the program to complete in a reasonable amount of time. The next section will focus on analysis of the data obtained from the various runs of each program and compare them against the data obtained from the Genbank® CDS files.

Results and Discussion

Tests and Results

Overview. A test set of *Rattus norvegicus* sequences from the Genbank® rodent division was used to test each program. The test set began with 14,571 sequences, but was reduced to 6,831 sequences after a series of filters were applied:

1. The site locations for each sequence had to be known and thus contained in the corresponding CDS file.
2. All sequences which were reverse complemented (as designated in the CDS file) were omitted because all the programs which were written or tested only analyzed sequences in the forward direction.
3. Sequences with incomplete coding regions (designated by symbols such as “<” and “>”) were omitted.

The test set generated two sets of data points for the four programs that were tested. The gene prediction program utilizing LDA was tested and compared against FGENES, a commercially available package which uses the LDA algorithm to predict genes. The gene prediction program using HMMs was also tested and compared against Genscan, another commercially available package using HMMs. For each program, the sensitivity (Sn) and specificity (Sp) was calculated by base, exon, and gene. Sensitivity (Sn) is defined as the percentage of coding nucleotides correctly labeled as coding. Specificity (Sp) is defined as the percentage of nucleotides labeled as coding that were actually coding (Henderson, Salzberg & Fasman, 1996). Calculating these results by

exons meant that the percentage of exons which were labeled correctly was calculated, and for genes, the percentage of genes which were correctly labeled was calculated.

The second set of statistics that were used to compare these four gene prediction programs deals with the accuracy of predicted sites. The two ratios of interest were S_n^{true} , which is the number of correctly classified sites (for example start sites) over the total number of sites and S_n^{false} , which is the ratio of correctly classified false sites over the total number of false sites (Yin et al., 2001).

LDA Results. The LDA program predicted the presence of genes in roughly 80% of the sequences which were examined, comparable to the commercial package, FGENES which also correctly predicted the presence of genes in roughly 80% of the sequences examined. Though FGENES should work with any species, the weight tables that it used were based on human sequences rather than rodent sequences. As a result, the predictions were less accurate than they would have been had the weight tables been generated based on rodent sequences. There were some cases where the expected start site according to the Genbank® files did not contain an “ATG” (start codon). In these cases, both the program and FGENES failed to identify the start site. These failures occur because the sequence may not contain the full gene. Without the ability to stitch together multiple sequences, the program cannot accurately predict the sites in these sequences.

		Overlap	Program	Genbank
Base	LDA GPP	5,649,657	8,290,383	10,155,948
	HMM GPP	3,853,766	4,630,297	9,963,245
Exon	LDA GPP	247	8,550	8,238
	HMM GPP	414	8,823	8,238
Gene	LDA GPP	243	6,813	6,813
	HMM GPP	414	6,813	6,813

Table 1 – Raw results for comparing accuracy of two programs.

Next, the accuracy of the predictions were compared on three levels – Base, Exon, and Gene. This is summarized in Table 1. At these levels, each program is evaluated by counting the number of bases, exons, and genes it correctly predicted as coding which is represented in the Overlap column. For example, our LDA program predicted 5,649,657 coding bases correctly out of a total over 10,155,948 coding bases. The Program column tells us how many total bases our LDA program predicted as coding. The number is important because it tells us how accurate our program is when it predicts coding regions. For example, though the HMM program only predicted 3,853,766 coding bases correctly, it only predicted a total of 4,630,297 total bases, which means that over 80% of the bases that it predicted were actually coding. The Genbank column gives us the number of bases, exons, and genes reported by the Genbank® CDS files. Dividing the Overlap by

the Genbank number gives us the percentage of total true coding bases, exons, and genes predicted by our programs.

		Sensitivity	Specificity
Base	LDA GPP	55.74%	68.16%
Exon	LDA GPP	2.84%	2.95%
Gene	LDA GPP	3.57%	3.57%

Table 2 – Sensitivity and Specificity of LDA program.

Table 2 shows the statistics generated from running the LDA gene prediction program (LDA GPP) against the aforementioned rat sequence test set. The program completed predictions on the 14,971 sequence test set in under 10 minutes. It predicted over 55% of the total coding bases correctly and of the bases that it predicted as coding over 68% of them were actually coding. The publicly available package FGENES was also run on the same dataset. It took over 12 hours to complete predictions on the same test set. However, it was more accurate in the predictions that it made, predicting over 84% of the total coding bases with a specificity of 98%. By utilizing more elements of the sequence, FGENES is able to make more accurate predictions, while taking longer to complete.

Table 3 represents the second set of statistics that were examined, S_n^{true} and S_n^{false} . The program predicted almost 18% of start sites correctly and 6% of stop sites correctly. Examining the number of potential start and stop sites (TN, True Negatives) helps to

illustrate difficulty of the task. For example, out of 227,000 potential start sites, only 6,000 (2.64%) of them were real. The program predicted over 99% of the false sites correctly. As expected, FGENES performs more accurately, predicting 63.91% of start sites correctly and 78% of stop sites correctly.

		TP	FN	TN	FP	Sn(true)	Sn(false)
Start	LDA GPP	1,209	5,604	221,643	0	17.75%	100.00%
Donor	LDA GPP	22	2,821	1,435,370	1,742	0.77%	99.88%
Acceptor	LDA GPP	0	2,843	1,869,400	1,464	0.00%	99.92%
Stop	LDA GPP	436	6,377	563,014	0	6.40%	100.00%

Table 3 – Sn(true) and Sn(false) of LDA program.

HMM Results. The same test set was run through both Genscan and the HMM gene prediction program and generated the same statistics to compare the two to each other.

		Sensitivity	Specificity
Base	HMM GPP	38.68%	83.23%
Exon	HMM GPP	5.03%	4.69%
Gene	HMM GPP	6.08%	6.08%

Table 4 – Sensitivity and Specificity of HMM program.

Table 4 shows how the HMM GPP performed on the same rat test set. The program completed predictions for the test set in under 10 minutes. It predicted over 38% of total coding bases correctly, and 83.23% of the bases that it predicted as coding were true coding bases. These percentages tell us that the HMM program is conservative in the regions which it predicts as coding, but does so very accurately. Genscan, a publicly available package which uses HMMs, was also run against the test set. Genscan took over 4 hours to complete its run. It predicted over 99% of total coding bases correctly with an accuracy of 99%.

Table 5 shows how accurate the HMM GPP performed at identifying real and false sites. It predicted 10.93% of start sites correctly and 35.36% of stop sites correctly. Again, it is important to consider the scope of these predictions. As previously mentioned, the number of real start sites is only 2.6% of the number of potential start sites. The number is even more dramatic when examining the stop sites. Real stop sites comprise only 1.05% of all potential stop sites. Considering that fact, 35.36% of real

stop sites predicted correctly is reasonable. Genscan performs very well, predicting 80% of start sites correctly and 85% of stop sites correctly.

		TP	FN	TN	FP	Sn(true)	Sn(false)
Start	HMM GPP	745	6,068	221,179	0	10.93%	100.00%
Donor	HMM GPP	1	647	1,434,573	3,216	0.15%	99.78%
Acceptor	HMM GPP	0	648	1,868,603	3,688	0.00%	99.80%
Stop	HMM GPP	2,409	4,404	564,987	0	35.36%	100.00%

Table 5 – Sn(true) and Sn(false) of HMM program.

Discussion

Examining the results of the runs between the two programs revealed that hidden Markov models are more accurate and conservative than Linear Discriminant Analysis. What is impressive about HMMs is that they predict fewer coding regions than LDA but predict stop sites correctly nearly six times more than LDA does. Consider the data presented in Table 6 which compares sensitivity and specificity of the two programs that were implemented against each other.

		Sensitivity	Specificity
Base	LDA GPP	55.74%	68.16%
	HMM GPP	38.68%	83.23%
Exon	LDA GPP	2.84%	2.95%
	HMM GPP	5.03%	4.69%
Gene	LDA GPP	3.57%	3.57%
	HMM GPP	6.08%	6.08%

Table 6 – Sensitivity and Specificity of LDA and HMM algorithms.

The HMM algorithm outperformed the LDA algorithm in every category except for Base sensitivity. HMMs perform better because they consider more paths and utilize the data at each site with more sensitivity than the LDA does. In scoring individual sites, the LDA algorithm takes scores calculated from four regions consisting of 12 nucleotides that surround a site (upstream/downstream codon/hexamer) which means that if one base changes, as many as half of those four scores may change. On the other hand, the HMM examines each base pair from the window independently, which means that if one base in that window changes, the effect on the overall score is considerably smaller.

A path through a sequence determines which sites are chosen as part of the gene prediction. In the LDA algorithm, if a site is not chosen as part of the path, its corresponding score is not included in the calculation. In essence, it is ignored.

However, in the HMM algorithm, all sites are considered, whether they are chosen in the path or not. If a site is not part of the predicted sites for a given sequence, it is not omitted from the calculations. The only difference is that the score is calculated in the negative model rather than the positive model. Ultimately, that site will either contribute or prevent a given path from being chosen as the more likely for a gene prediction.

The hidden Markov model performs better than the Linear Discriminant Analysis in gene prediction because it considers more of the elements that contribute to determining whether a set of sites is real or false.

Conclusion

Two popular algorithms that are used in gene prediction were analyzed and compared. Like many algorithms that are used in gene prediction in bioinformatics, there are generally two steps. The first is a training step and the second is a prediction step utilizing the trained data. In both cases, the prediction algorithm is only as good as the trained data that it used, which is why species-specific tables were used. When compared to the commercial packages, the programs were not as accurate in predicting exact sites. However, they did run considerably faster, which could be beneficial if one was interested in finding which sequences in a large dataset contained genes. These programs could be used as a pre-filter to reduce the number of sequences to run through commercially available programs such as Genscan and FGENES.

There are a number of changes that could be implemented to improve the accuracy of the programs:

1. Use promoter detection to improve accuracy of start site predictions.
2. Use elements in the 3' UTR region such as polyA tails to improve accuracy of stop site predictions.
3. Investigate current and past methods used to improve splice site predictions.
4. Allow for the program to predict genes in sequences with multiple genes.
5. Improve accuracy of transition probabilities for the HMM by doing iterative runs with training sequences until change between runs falls below a preset threshold.

It is worthwhile noting that packages such as Genscan and FGENES have taken into consideration some of the points mentioned above.

There are numerous methods that have been used to varying degrees of success in gene prediction. Most methods have been able to predict start and stop sites to a good degree of accuracy, but have difficulties with acceptor and donor sites. As a result, researchers have been working on modifying current algorithms or creating new ones to improve the accuracy of acceptor and donor site predictions (Stanke & Waack, 2003). Improving on current methods and integrating different algorithms is the key to improving the accuracy and reliability of the challenging task of gene prediction.

References

- Base Pairing. (1999). Retrieved January 17, 2004, from <http://users.rcn.com/jkimball.ma.ultranet/BiologyPages/B/BasePairing.html>
- Brazma, A., Parkinson, H., Schlitt, T., Shojatalab, M. (2001). A quick introduction to elements of biology - cells, molecules, genes, functional genomics, microarrays. Retrieved January 17, 2004, from http://www.ebi.ac.uk/microarray/biology_intro.html#Genomes
- Boughman, J., Wenrick, W., Gerald S. (1998). Greater spear-nosed bats discriminate group mates by vocalizations. *Animal Behavior*, **55**, 1717–1732.
- Cawley, S. L., Pachter, L. (2003). HMM sampling and applications to gene finding and alternative splicing. *Bioinformatics*, **19**(2), ii36-ii41.
- Costa, R. (2003). Gene Prediction Algorithms. *Computational Biology*, **May 2003 Review**, 1-7.
- Cox, J. (2002). Comparative Study of Classification Algorithms and Gene Selection Methods for the Discrimination of Cancer Tissue Samples using Microarray Expression Data. 3rd International Conference on Systems Biology. December 13-15, 2002. Karolinska Institutet, Stockholm, Sweden.
- Durbin, R., Eddy, S., Krogh, A., Mitchison, G. (1999). *Biological Sequence Analysis: Probabilistic models of proteins and nucleic acids*. (1st ed.). Cambridge, United Kingdom: Cambridge University Press.
- GENIE, a UCSC/LBNL Gene Finding Program. (1996). Retrieved January 17, 2004, from <http://www.fruitfly.org/~martinr/doc/genie.html>
- Henderson, H., Salzberg, S., Fasman, K. (1996). Finding Genes in Human DNA with a Hidden Markov Model. *Journal for Computational Biology*, **4**(2), 127-141.
- How BIG are Prokaryotic Genomes? (2002) Retrieved January 17, 2004 from <http://www.sci.sdsu.edu/~smaloy/MicrobialGenetics/topics/chroms-genes-prots/genomes.html>
- The Human Genome Discovery Page. (2001). Retrieved January 17, 2004 from <http://www.jupiterscientific.org/sciinfo/genome.html>
- Kim, M., Kim, D., Lee, S. (2003) Face recognition using the embedded HMM with second-order block-specific observations. *Pattern Recognition*, **36**, 2723-2735.

- Khorsheed, M. S. (2003). Recognising handwritten Arabic manuscripts using a single hidden Markov model. *Pattern Recognition Letters*, **24**, 2235-2242.
- Mendez, M. A., Hodar, C., Vulpe, C., Gonzalez, M., Cambiazo, V. (2002). Discriminant analysis to evaluate clustering of gene expression data. **FEBS Letters**, **522**, 24-28.
- Moore, J. H., Parker, J. S., Hahn, L. W. (2001). Symbolic Discriminant Analysis for Mining Gene Expression Patterns. In L. De Raedt, P. Flach (Ed.), *Lecture Notes in Artificial Intelligence* (pp. 372-381), Berlin, Springer-Verlag.
- The MORGAN gene finding system. (1998). Retrieved January 17, 2004 from <http://www.tigr.org/~salzberg/morgan.html>
- Mount, D. (2001). *Bioinformatics: Sequence and Genome Analysis*. (1st ed.). Cold Spring Harbor, New York: Cold Spring Harbor Laboratory Press.
- New GENSCAN Web Server at MIT. (2002). Retrieved January 17, 2004 from <http://genes.mit.edu/GENSCAN.html>
- ORNL Grail Form (v1.3). (1996). Retrieved January 17, 2004 from <http://compbio.ornl.gov/Grail-1.3/>
- Richon, A. (2001). A Short History of Bioinformatics. Retrieved January 17, 2004 from <http://www.netsci.org/Science/Bioinform/feature06.html>
- Russell, P. (1996) *Genetics*, (4th ed.). New York, New York: HarperCollins College Publishers.
- Salzberg, S., Delcher, A. L., Fasman, K. H., Henderson, J. (1998). A Decision Tree System for Finding Genes in DNA. *Journal for Computational Biology*, **5**(4), 667-680.
- Singer, M., Berg, P. (1991) *Genes and Genomes: A Changing Perspective*. (1st ed.). Mill Valley: University Science Books.
- Softberry. (2002). Retrieved January 17, 2004 from <http://www.softberry.com/pdf/broshura.pdf>
- Solovyev, V. V., Salamov, A. A., Lawrence, C. B. (1995). Identification of human gene structure using linear discriminant functions and dynamic programming. Proceedings of the Third International Conference on Intelligent Systems for Molecular Biology. July 16-19, 1995. Cambridge, United Kingdom.

- Stanke, M., Waack, S. (2003) Gene prediction with a hidden Markov model and a new intron submodel. *Bioinformatics*, **19**, 215-225.
- Yada, T., Nakao, M., Totoki, Y., Nakai, K. (1999) Modeling and predicting transcriptional units of Escherichia coli genes using hidden Markov models. *Bioinformatics*, **15**(12), 987-993.
- Yin, M., Wang, J. (2001) Effective hidden Markov models for detecting splicing junction sites in DNA sequences. *Information Sciences*, **139**, 139-163.
- Zhang, M. (2000) Discriminant Analysis and its application in DNA sequence motif recognition. *Briefing in Bioinformatics*, **1**(4), 1-12.

Appendix A

Generate_lda_table.pl Code

```
#!/usr/bin/perl -w
# *****
# generate_lda_table.pl
# purpose: to generate linear discriminant analysis
#         score table:
#         1. parse cds file
#         2. find corresponding cds regions and score
#         3. find remaining possible splice sites and score
# inputs: 2 files
#         1. cds file
#         2. fasta file
# *****

die "Usage: $0 <cds file> <fasta file>\n" if (@ARGV != 2);

## GLOBAL VARIABLES ##
use vars qw(%seqs $seqidx @start @end @complement @numExons %dups);
$main::seqidx = 0;

## MAIN PROGRAM ##

## PARSE CDS FILE ##
open (CDS, $ARGV[0]) ||
    die "Cannot open $ARGV[0] for reading!!\n";

print STDERR "Reading in CDS data...\n";
while (<CDS>)
{
    chomp;
    parseLine($_);
    $main::seqidx++;
}

close (CDS);
print STDERR "Done.\n";

## SET UP LDA MATRIX INITIAL VALUES ##

## GENERATE ALL POSSIBLE TRIPLETS AND HEXONS
print STDERR "Setting up triplet and hexon tables...\n";
@nuc = qw(A T G C);
foreach $n (@nuc)
{
    foreach $n2 (@nuc)
    {
        foreach $n3 (@nuc)
        {
            $triplet = "$n$n2$n3";
            $triplets{$triplet} = 1;
            foreach $n4 (@nuc)
            {
                foreach $n5 (@nuc)
                {
                    foreach $n6 (@nuc)
                    {
```

```

        $hexon = "$n$n2$n3$n4$n5$n6";
        $hexons{$hexon} = 1;
    }
}
}
}
}

use vars qw(%str %stp %acpt %donor);
use vars qw(%strN %stpN %acptN %donorN);
use vars qw($allstart3 $allstart6 $allstop3 $allstop6);
use vars qw($allaccept3 $allaccept6 $alldonor3 $alldonor6);
use vars qw($allstartN $allstopN $allacceptN $alldonorN);
use vars qw(%stopC %acceptS %donorS %startC %triplets %doublets);

$main::stopC{TGA} = 1;
$main::stopC{TAA} = 1;
$main::stopC{TAG} = 1;

$main::acceptS = "GT";
$main::donorS = "AG";

$main::startC{ATG} = 1;

$main::triplets{TGA} = 1;
$main::triplets{TAA} = 1;
$main::triplets{TAG} = 1;
$main::triplets{ATG} = 1;

$main::doublets{"GT"} = 1;
$main::doublets{AG} = 1;

## ZERO COUNTS FOR ALL TRIPLETS TO ZERO
foreach $t (keys(%triplets))
{
    ## POSITIVE SCORES
    $main::str->{UP}->{3}->{$t} = 0;
    $main::str->{DN}->{3}->{$t} = 0;
    $main::stp->{UP}->{3}->{$t} = 0;
    $main::stp->{DN}->{3}->{$t} = 0;
    $main::acpt->{UP}->{3}->{$t} = 0;
    $main::acpt->{DN}->{3}->{$t} = 0;
    $main::donor->{UP}->{3}->{$t} = 0;
    $main::donor->{DN}->{3}->{$t} = 0;

    ## NEGATIVE SCORES
    $main::strN->{UP}->{3}->{$t} = 0;
    $main::strN->{DN}->{3}->{$t} = 0;
    $main::stpN->{UP}->{3}->{$t} = 0;
    $main::stpN->{DN}->{3}->{$t} = 0;
    $main::acptN->{UP}->{3}->{$t} = 0;
    $main::acptN->{DN}->{3}->{$t} = 0;
    $main::donorN->{UP}->{3}->{$t} = 0;
    $main::donorN->{DN}->{3}->{$t} = 0;
}

## ZERO COUNTS FOR ALL HEXONS TO ZERO
foreach $h (keys(%hexons))
{
    ## POSITIVE SCORES

```



```

$main::str->{UP}->{6}->{$h} = 0;
$main::str->{DN}->{6}->{$h} = 0;
$main::stp->{UP}->{6}->{$h} = 0;
$main::stp->{DN}->{6}->{$h} = 0;
$main::acpt->{UP}->{6}->{$h} = 0;
$main::acpt->{DN}->{6}->{$h} = 0;
$main::donor->{UP}->{6}->{$h} = 0;
$main::donor->{DN}->{6}->{$h} = 0;

## NEGATIVE SCORES
$main::strN->{UP}->{6}->{$h} = 0;
$main::strN->{DN}->{6}->{$h} = 0;
$main::stpN->{UP}->{6}->{$h} = 0;
$main::stpN->{DN}->{6}->{$h} = 0;
$main::acptN->{UP}->{6}->{$h} = 0;
$main::acptN->{DN}->{6}->{$h} = 0;
$main::donorN->{UP}->{6}->{$h} = 0;
$main::donorN->{DN}->{6}->{$h} = 0;
}
print STDERR "Done.\n";

$main::allstart3 = $main::allstart6 = $main::allstop3 = $main::allstop6 = 0;
$main::allaccept3 = $main::allaccept6 = $main::alldonor3 = $main::alldonor6 = 0;

$main::allstartN = $main::allstopN = $main::allacceptN = $main::alldonorN = 0;

open (FASTA, $ARGV[1]) ||
    die "Cannot open $ARGV[1] for reading!!\n";

$fasta = $seq = "";

print STDERR "Reading fasta file\n";

open (LOG, ">logfile");

while (<FASTA>)
{
    chomp;
    if ($_ =~ />/) ## TITLE LINE
    {
        ## SCORE PREVIOUS SEQUENCE
        $fasta =~ tr/atgcn/ATGCN/;
        print STDERR "$seq\n";
        scoreSeq($seq, $fasta);

        negScore($seq, $fasta);
        ## PARSE NEW SEQUENCE
        ($seq) = split('\t');
        $seq =~ s/>\/;
        $fasta = "";
    }
    else
    {
        $fasta .= $_;
    }
}

close (FASTA);
print STDERR "Done\n";

```

```

print STDERR "Tabulating Scores....\n";

if (!$main::allstart3) { $main::allstart3 = 1; }
if (!$main::allstart6) { $main::allstart6 = 1; }
if (!$main::allstop3) { $main::allstop3 = 1; }
if (!$main::allstop6) { $main::allstop6 = 1; }
if (!$main::allaccept3) { $main::allaccept3 = 1; }
if (!$main::allaccept6) { $main::allaccept6 = 1; }
if (!$main::alldonor3) { $main::alldonor3 = 1; }
if (!$main::alldonor6) { $main::alldonor6 = 1; }
if (!$main::allstartN) { $main::allstartN = 1; }
if (!$main::allstopN) { $main::allstopN = 1; }
if (!$main::allacceptN) { $main::allacceptN = 1; }
if (!$main::alldonorN) { $main::alldonorN = 1; }

foreach $t (keys(%triplets))
{
    ## POSITIVE SCORES
    print sprintf "STRT.UP.3.P.$t\t%.10f\n", ($main::str->{UP}->{3}->{$t})/$main::allstart3;
    print sprintf "STRT.DN.3.P.$t\t%.10f\n", ($main::str->{DN}->{3}->{$t})/$main::allstart3;
    print sprintf "STOP.UP.3.P.$t\t%.10f\n", ($main::stp->{UP}->{3}->{$t})/$main::allstop3;
    print sprintf "STOP.DN.3.P.$t\t%.10f\n", ($main::stp->{DN}->{3}->{$t})/$main::allstop3;
    print sprintf "ACPT.UP.3.P.$t\t%.10f\n", ($main::acpt->{UP}->{3}->{$t})/$main::allaccept3;
    print sprintf "ACPT.DN.3.P.$t\t%.10f\n", ($main::acpt->{DN}->{3}->{$t})/$main::allaccept3;
    print sprintf "DONR.UP.3.P.$t\t%.10f\n", ($main::donor->{UP}->{3}->{$t})/$main::alldonor3;
    print sprintf "DONR.DN.3.P.$t\t%.10f\n", ($main::donor->{DN}->{3}->{$t})/$main::alldonor3;

    ## NEGATIVE SCORES
    print sprintf "STRT.UP.3.N.$t\t%.10f\n", ($main::strN->{UP}->{3}->{$t})/$main::allstartN;
    print sprintf "STRT.DN.3.N.$t\t%.10f\n", ($main::strN->{DN}->{3}->{$t})/$main::allstartN;
    print sprintf "STOP.UP.3.N.$t\t%.10f\n", ($main::stpN->{UP}->{3}->{$t})/$main::allstopN;
    print sprintf "STOP.DN.3.N.$t\t%.10f\n", ($main::stpN->{DN}->{3}->{$t})/$main::allstopN;
    print sprintf "ACPT.UP.3.N.$t\t%.10f\n", ($main::acptN->{UP}->{3}->{$t})/$main::allacceptN;
    print sprintf "ACPT.DN.3.N.$t\t%.10f\n", ($main::acptN->{DN}->{3}->{$t})/$main::allacceptN;
    print sprintf "DONR.UP.3.N.$t\t%.10f\n", ($main::donorN->{UP}->{3}->{$t})/$main::alldonorN;
    print sprintf "DONR.DN.3.N.$t\t%.10f\n", ($main::donorN->{DN}->{3}->{$t})/$main::alldonorN;
}

foreach $h (keys(%hexons))
{
    ## POSITIVE SCORES
    print sprintf "STRT.UP.6.P.$h\t%.10f\n", ($main::str->{UP}->{6}->{$h})/$main::allstartN;

```

```

    print sprintf "STRT.DN.6.P.$h\t%.10f\n", ($main::str->{DN}->{6}-
>{$h})/$main::allstartN;
    print sprintf "STOP.UP.6.P.$h\t%.10f\n", ($main::stp->{UP}->{6}-
>{$h})/$main::allstopN;
    print sprintf "STOP.DN.6.P.$h\t%.10f\n", ($main::stp->{DN}->{6}-
>{$h})/$main::allstopN;
    print sprintf "ACPT.UP.6.P.$h\t%.10f\n", ($main::acpt->{UP}->{6}-
>{$h})/$main::allacceptN;
    print sprintf "ACPT.DN.6.P.$h\t%.10f\n", ($main::acpt->{DN}->{6}-
>{$h})/$main::allacceptN;
    print sprintf "DONR.UP.6.P.$h\t%.10f\n", ($main::donor->{UP}->{6}-
>{$h})/$main::alldonorN;
    print sprintf "DONR.DN.6.P.$h\t%.10f\n", ($main::donor->{DN}->{6}-
>{$h})/$main::alldonorN;

    ## NEGATIVE SCORES
    print sprintf "STRT.UP.6.N.$h\t%.10f\n", ($main::strN->{UP}->{6}-
>{$h})/$main::allstartN;
    print sprintf "STRT.DN.6.N.$h\t%.10f\n", ($main::strN->{DN}->{6}-
>{$h})/$main::allstartN;
    print sprintf "STOP.UP.6.N.$h\t%.10f\n", ($main::stpN->{UP}->{6}-
>{$h})/$main::allstopN;
    print sprintf "STOP.DN.6.N.$h\t%.10f\n", ($main::stpN->{DN}->{6}-
>{$h})/$main::allstopN;
    print sprintf "ACPT.UP.6.N.$h\t%.10f\n", ($main::acptN->{UP}->{6}-
>{$h})/$main::allacceptN;
    print sprintf "ACPT.DN.6.N.$h\t%.10f\n", ($main::acptN->{DN}->{6}-
>{$h})/$main::allacceptN;
    print sprintf "DONR.UP.6.N.$h\t%.10f\n", ($main::donorN->{UP}->{6}-
>{$h})/$main::alldonorN;
    print sprintf "DONR.DN.6.N.$h\t%.10f\n", ($main::donorN->{DN}->{6}-
>{$h})/$main::alldonorN;
}

## SUBROUTINES ##

sub errorCheck
{
    foreach $seq (keys(%main::seqs))
    {
        my $idx = $main::seqs{$seq};
        if (exists $main::dups{$seq})
        {
            $main::dups{$seq} .= $main::seqs{$seq};
            @line = split('\t', $main::dups{$seq});
            $numDups = @line;
            for ($i = 0; $i < $numDups; $i++)
            {
                $idx = $line[$i];
                print "$idx\t".
                    "$seq\t".
                    "$main::complement[$idx]\t".
                    "$main::numExons[$idx]\t";
                for ($j = 0; $j < $main::numExons[$idx]; $j++)
                {
                    print "$main::start[$idx][$j]\t".
                        "$stop[$idx][$j]\t";
                }
                print "\n";
            }
        }
    }
}

```

```

        else
        {
            print "$idx\t".
                "$seq\t".
                "$main::complement[$idx]\t".
                "$main::numExons[$idx]\t".
                "$main::start[$idx][0]\t".
                "$stop[$idx][0]\n";
        }
    }
}

sub parseLine
{
    my $idx = $main::seqidx;

    ## SPLIT LINE, GET DATA THAT YOU NEED ##
    @line = split('\t', $_[0]);
    my ($seq, $cds) = @line[0,2];

    ## INDEX SEQUENCES TO ASSOCIATE THEM WITH ARRAY INDEX ##
    if (exists $main::seqs{$seq}) ## THERE IS A DUPLICATE ##
    {
        $main::dups{$seq} .= "$main::seqs{$seq}\t";
    }
    $main::seqs{$seq} = $idx;

    ## PARSE CDS ##
    parseCDS($cds);
}

sub parseCDS
{
    my $cds = $_[0];
    if ($cds =~ /complement/) ## ALL CDS NEED TO BE COMPLEMENTED
    {
        $cds =~ s/\\/g;
        push(@main::complement, 1);
        ($temp, @rest) = split('\(', $cds);

        if ($rest[0] =~ /join/) ## MULTIPLE CDS REGIONS
        {
            setMultiCDS(@rest);
        }
        else # SINGLE CDS
        {
            setSingleCDS($rest[0]);
        }
    }
    else ## NO COMPLEMENT
    {
        push(@main::complement, 0);

        if ($cds =~ /join/) ## MULTIPLE CDS REGIONS
        {
            $cds =~ s/\\/g;
            (@rest) = split('\(', $cds);
            setMultiCDS(@rest);
        }
        else # SINGLE CDS
    }
}

```

```

        {
            setSingleCDS($cds);
        }
    }
}

sub setSingleCDS
{
    my $idx = $main::seqidx;
    ($begin, $end) = split('\.\.', $_[0]);
    $main::start[$idx][0] = $begin;
    $main::stop[$idx][0] = $end;
    push (@main::numExons, 1);
}

sub setMultiCDS
{
    my $idx = $main::seqidx;
    ($temp, $exons) = @_;
    @pairs = split(',', $exons);
    my $num = @pairs;
    push (@main::numExons, $num);
    for ($i = 0; $i < $num; $i++)
    {
        ($begin, $end) = split('\.\.', $pairs[$i]);
        $main::start[$idx][$i] = $begin;
        $main::stop[$idx][$i] = $end;
    }
}

sub scoreSeq
{
    ($seq, $fasta) = @_;
    if (exists $main::seqs{$seq})
    {
        # score sequence
        # FIND ALL SPLICE SITES

        # CHECK IF SPLICE SITE IS A CDS
        # ARE THERE MULTIPLE ENTRIES FOR THIS SEQUENCE?

        if (exists $main::dups{$seq})
        {
            $main::dups{$seq} .= $main::seqs{$seq};
            $save = $main::dups{$seq};
            @line = split('\t', $save);
            $numDups = @line;
            for ($x = 0; $x < $numDups; $x++)
            {
                $idx = $line[$x];
                $numE = $main::numExons[$idx];
                $comp = $main::complement[$idx];

                if ($numE == 1)
                {
                    $s1 = $main::start[$idx][0];
                    $s2 = $main::stop[$idx][0];
                    scoreOuter($s1, $s2, $comp, $fasta);
                }
                else
                {

```

```

        $s1 = $main::start[$idx][0];
        $s2 = $main::stop[$idx][$numE-1];
        scoreOuter($s1, $s2, $comp, $fasta);

        for ($e = 1; $e < $numE; $e++)
        {
            $i1 = $main::stop[$idx][$e-1];
            $i2 = $main::start[$idx][$e];
            scoreInner($i1, $i2, $comp, $fasta);
        }
    }
}
else ## NO DUPLICATES
{
    $idx = $main::seqs{$seq};
    $numE = $main::numExons[$idx];

    $comp = $main::complement[$idx];

    if ($numE == 1)
    {
        $s1 = $main::start[$idx][0];
        $s2 = $main::stop[$idx][0];
        scoreOuter($s1, $s2, $comp, $fasta);
    }
    else
    {
        $s1 = $main::start[$idx][0];
        $s2 = $main::stop[$idx][$numE-1];
        scoreOuter($s1, $s2, $comp, $fasta);

        for ($e = 1; $e < $numE; $e++)
        {
            $i1 = $main::stop[$idx][$e-1];
            $i2 = $main::start[$idx][$e];
            scoreInner($i1, $i2, $comp, $fasta);
        }
    }
}
}

sub scoreOuter
{
    ($st, $sp, $c, $fasta) = @_;
    $length = length($fasta);

    if ($st !~/>|</)
    {
        $st--;
        if ( ($st >= 3) and ($length - $st > 3) ) ## ENOUGH NUCL TO GET UPSTREAM
        {
            $upTri = substr($fasta, $st - 3, 3);
            $dnTri = substr($fasta, $st, 3);
            if (!defined $upTri)
            {
                print LOG "ST: $st = NOT DEFINED\n";
            }
        }
    }
}

```

```

    }
    if ($c)
    {
        $main::allstop3++;
        $upTri = revComp($upTri);
        $dnTri = revComp($dnTri);
        $main::stp->{UP}->{3}->{$dnTri}++;
        $main::stp->{DN}->{3}->{$upTri}++;
    }
    else
    {
        $main::allstart3++;
        $main::str->{UP}->{3}->{$upTri}++;
        $main::str->{DN}->{3}->{$dnTri}++;
    }
}

if ( ($st >= 6) and ($length - $st > 6) )## ENOUGH NUCL TO GET UPSTREAM 6
{
    $upHex = substr($fasta, $st - 6, 6);
    $dnHex = substr($fasta, $st, 6);
    if ($c)
    {
        $main::allstop6++;
        $upHex = revComp($upHex);
        $dnHex = revComp($dnHex);
        $main::stp->{UP}->{6}->{$dnHex}++;
        $main::stp->{DN}->{6}->{$upHex}++;
    }
    else
    {
        $main::allstart6++;
        $main::str->{UP}->{6}->{$upHex}++;
        $main::str->{DN}->{6}->{$dnHex}++;
    }
}

if ($sp !~/>|</)
{
    if ($length - $sp > 3) ## ENOUGH NUCL TO GET DOWNSTREAM 3
    {
        $upTri = substr($fasta, $sp - 3, 3);
        $dnTri = substr($fasta, $sp, 3);
        if ($c)
        {
            $main::allstart3++;
            $upTri = revComp($upTri);
            $dnTri = revComp($dnTri);
            $main::str->{UP}->{3}->{$dnTri}++;
            $main::str->{DN}->{3}->{$upTri}++;
        }
        else
        {
            $main::allstop3++;
            $main::stp->{UP}->{3}->{$upTri}++;
            $main::stp->{DN}->{3}->{$dnTri}++;
        }
    }
}

```

```

    }
}

if ($length - $sp > 6)
{
    $upHex = substr($fasta, $sp - 6, 6);
    $dnHex = substr($fasta, $sp, 6);
    if ($c)
    {
        $main::allstart6++;
        $upHex = revComp($upHex);
        $dnHex = revComp($dnHex);
        $main::str->{UP}->{6}->{$dnHex}++;
        $main::str->{DN}->{6}->{$upHex}++;
    }
    else
    {
        $main::allstop6++;
        $main::stp->{UP}->{6}->{$upHex}++;
        $main::stp->{DN}->{6}->{$dnHex}++;
    }
}
}

sub scoreInner
{
    ($istop, $istart, $c, $fasta) = @_;

    if ($istop !~>|</)
    {
        $upTri = substr($fasta, $istop - 3, 3);
        $dnTri = substr($fasta, $istop, 3);
        if ($c)
        {
            $main::allaccept3++;
            $upTri = revComp($upTri);
            $dnTri = revComp($dnTri);
            $main::acpt->{UP}->{3}->{$dnTri}++;
            $main::acpt->{DN}->{3}->{$upTri}++;
        }
        else
        {
            $main::alldonor3++;
            $main::donor->{UP}->{3}->{$upTri}++;
            $main::donor->{DN}->{3}->{$dnTri}++;
        }
    }

    $upHex = substr($fasta, $istop - 6, 6);
    $dnHex = substr($fasta, $istop, 6);
    if ($c)
    {
        $main::allaccept6++;
        $upHex = revComp($upHex);
        $dnHex = revComp($dnHex);
        $main::acpt->{UP}->{6}->{$dnHex}++;
    }
}

```



```

        $main::acpt->{DN}->{6}->{$upHex}++;
    }
    else
    {
        $main::main::alldonor6++;
        $main::donor->{UP}->{6}->{$upHex}++;
        $main::donor->{DN}->{6}->{$dnHex}++;
    }
}

if ($istart !~/>|</)
{
    $istart--;
    $upTri = substr($fasta, $istart - 3, 3);
    $dnTri = substr($fasta, $istart, 3);
    if ($c)
    {
        $main::alldonor3++;
        $upTri = revComp($upTri);
        $dnTri = revComp($dnTri);
        $main::donor->{UP}->{3}->{$dnTri}++;
        $main::donor->{DN}->{3}->{$upTri}++;
    }
    else
    {
        $main::allaccept3++;
        $main::acpt->{UP}->{3}->{$upTri}++;
        $main::acpt->{DN}->{3}->{$dnTri}++;
    }
}

$upHex = substr($fasta, $istart - 6, 6);
$dnHex = substr($fasta, $istart, 6);
if ($c)
{
    $main::alldonor6++;
    $upHex = revComp($upHex);
    $dnHex = revComp($dnHex);
    $main::donor->{UP}->{6}->{$dnHex}++;
    $main::donor->{DN}->{6}->{$upHex}++;
}
else
{
    $main::allaccept6++;
    $main::acpt->{UP}->{6}->{$upHex}++;
    $main::acpt->{DN}->{6}->{$dnHex}++;
}
}

}

sub revComp
{
    my $reverse = "";
    $f = $orig = $_[0];
    $f =~ tr/ATGC/TACG/;
    for ($i = length($f) - 1; $i >= 0; $i--)
    {
        $reverse .= substr($f, $i, 1);
    }
    return $reverse;
}

```

```

sub negScore
{
    ($seq, $fasta) = @_ ;
    $slength = length($fasta);

    if (exists $main::seqs{$seq})
    { ## SUBTRACT EXISTING KNOWN EXON SITES
        for ($a = 0; $a < $slength; $a++)
        {
            $codon = substr($fasta, $a, 3);
            $dimer = substr($fasta, $a, 2);
            if ( (exists $main::stopC{$codon}) or (exists $main::startC{$codon})
        )
        {
            if (exists $main::stopC{$codon})
            {
                if (exists $main::dups{$seq})
                {
                    # check all dups
                    $save2 = $main::dups{$seq};
                    @dupdata = split('\t', $save2);
                    $numDup = @dupdata;
                    for ($y = 0; $y < $numDup; $y++)
                    {
                        $six = $dupdata[$y];
                        if (!sameStop($a, $seq))
                        {
                            $main::allstopN++;
                            $triU = substr($fasta, $a, 3);
                            $triD = substr($fasta, $a - 3, 3);
                            $hexU = substr($fasta, $a, 6);
                            $hexD = substr($fasta, $a - 6, 6);

                            $main::stpN->{UP}->{3}->{$triU}++;
                            $main::stpN->{DN}->{3}->{$triD}++;
                            $main::stpN->{UP}->{6}->{$hexU}++;
                            $main::stpN->{DN}->{6}->{$hexD}++;
                        }
                    }
                }
            }
        }
        else # NO DUPS, ONLY ONE SEQUENCE TO CHECK
        {
            if (!sameStop($a, $seq))
            {
                if ($slength - $a >= 6)
                {
                    $main::allstopN++;
                    $triU = substr($fasta, $a, 3);
                    $triD = substr($fasta, $a - 3, 3);
                    $hexU = substr($fasta, $a, 6);
                    $hexD = substr($fasta, $a - 6, 6);

                    $main::stpN->{UP}->{3}->{$triU}++;
                    $main::stpN->{DN}->{3}->{$triD}++;
                    $main::stpN->{UP}->{6}->{$hexU}++;
                    $main::stpN->{DN}->{6}->{$hexD}++;
                }
            }
        }
    }
}

```

```

}
else ## START CODON
{
  if (exists $main::dups{$seq})
  {
    # check all dups
    $save2 = $main::dups{$seq};
    @dupdata = split('\t', $save2);
    $numDup = @dupdata;
    for ($y = 0; $y < $numDup; $y++)
    {
      $ix = $dupdata[$y];
      if (!sameStart($a, $seq))
      {
        $main::allstartN++;
        $triU = substr($fasta, $a, 3);
        $triD = substr($fasta, $a - 3, 3);
        $hexU = substr($fasta, $a, 6);
        $hexD = substr($fasta, $a - 6, 6);

        $main::strN->{UP}->{3}->{$triU}++;
        $main::strN->{DN}->{3}->{$triD}++;
        $main::strN->{UP}->{6}->{$hexU}++;
        $main::strN->{DN}->{6}->{$hexD}++;
      }
    }
  }
  else # NO DUPS, ONLY ONE SEQUENCE TO CHECK
  {
    if (!sameStart($a, $seq))
    {
      if ( ($a >= 6) and ($slength - $a >= 6) )
      {
        $main::allstartN++;
        $triU = substr($fasta, $a, 3);
        $triD = substr($fasta, $a - 3, 3);
        $hexU = substr($fasta, $a, 6);
        $hexD = substr($fasta, $a - 6, 6);

        $main::strN->{UP}->{3}->{$triU}++;
        $main::strN->{DN}->{3}->{$triD}++;
        $main::strN->{UP}->{6}->{$hexU}++;
        $main::strN->{DN}->{6}->{$hexD}++;
      }
    }
  }
}

if (exists $doublets{$dimer})
{
  if ($dimer eq $acceptS)
  {
    if (exists $main::dups{$seq})
    {
      # check all dups
      $save2 = $main::dups{$seq};
      @dupdata = split('\t', $save2);
      $numDup = @dupdata;
      for ($y = 0; $y < $numDup; $y++)
      {

```

```

$ix = $dupdata[$y];
if (!contained($a, $seq))
{
    $main::allacceptN++;
    $triU = substr($fasta, $a, 3);
    $triD = substr($fasta, $a - 3, 3);
    $hexU = substr($fasta, $a, 6);
    $hexD = substr($fasta, $a - 6, 6);

    $main::acptN->{UP}->{3}->{$triU}++;
    $main::acptN->{DN}->{3}->{$triD}++;
    $main::acptN->{UP}->{6}->{$hexU}++;
    $main::acptN->{DN}->{6}->{$hexD}++;
}
}
}
else
{
    if (!contained($a, $seq))
    {
        if ( ($a >= 6) and ($slength - $a >= 6) )
        {
            $main::allacceptN++;
            $triU = substr($fasta, $a, 3);
            $triD = substr($fasta, $a - 3, 3);
            $hexU = substr($fasta, $a, 6);
            $hexD = substr($fasta, $a - 6, 6);

            $main::acptN->{UP}->{3}->{$triU}++;
            $main::acptN->{DN}->{3}->{$triD}++;
            $main::acptN->{UP}->{6}->{$hexU}++;
            $main::acptN->{DN}->{6}->{$hexD}++;
        }
    }
}
}
else ## DIMER = DONOR SITE
{
    if (exists $main::dups{$seq})
    {
        # check all dups
        $save2 = $main::dups{$seq};
        @dupdata = split('\t', $save2);
        $numDup = @dupdata;
        for ($y = 0; $y < $numDup; $y++)
        {
            $ix = $dupdata[$y];
            if (!contained($a, $seq))
            {
                $main::alldonorN++;
                $triU = substr($fasta, $a, 3);
                $triD = substr($fasta, $a - 3, 3);
                $hexU = substr($fasta, $a, 6);
                $hexD = substr($fasta, $a - 6, 6);

                $main::donorN->{UP}->{3}->{$triU}++;
                $main::donorN->{DN}->{3}->{$triD}++;
                $main::donorN->{UP}->{6}->{$hexU}++;
                $main::donorN->{DN}->{6}->{$hexD}++;
            }
        }
    }
}
}

```



```

    }
  }
  return $same;
}

sub sameStart
{
  ($coord, $s) = @_;
  $same = 0;
  $localidx = $main::seqs{$s};
  $localstart = $main::start[$localidx][0];
  if ($localstart !~/>|</)
  {
    if ($localstart == $coord)
    {
      $same = 1;
    }
    else
    {
      $same = 0;
    }
  }
  return $same;
}

sub contained
{
  ($coord, $se) = @_;
  $in = 0;
  $cidx = $main::seqs{$se};
  if ($main::numExons[$cidx] == 1)
  {
    $in = 0;
  }
  else
  {
    for ($t = 0; $t < $main::numExons[$cidx]; $t++)
    {
      $lstart = $main::start[$cidx][$t];
      $lstop = $main::stop[$cidx][$t];
      if ( ($lstart !~/>|</) and ($lstart == $coord) )
      {
        $in = 1;
      }
      if ( ($lstop !~/>|</) and ($lstop == $coord) )
      {
        $in = 1;
      }
    }
  }
  return $in;
}

```

Appendix B

Generate_hmm_table.pl Code

```
#!/usr/bin/perl -w
# *****
# generate_hmm_table.pl
# purpose: to generate hidden Markov model score table
#         1. parse cds file
#         2. find corresponding cds regions and score
#         3. find remaining possible splice sites and score
# inputs: 2 files
#         1. cds file
#         2. fasta file
# *****

die "Usage: $0 <cds file> <fasta file>\n" if (@ARGV != 2);

## GLOBAL VARIABLES ##
use vars qw(%seqs $seqidx @start @end @complement @numExons %dups);
$main::seqidx = 0;

## MAIN PROGRAM ##

## PARSE CDS FILE ##
open (CDS, $ARGV[0]) ||
    die "Cannot open $ARGV[0] for reading!!\n";

print STDERR "Reading in CDS data...\n";
while (<CDS>)
{
    chomp;
    parseLine($_);
    $main::seqidx++;
}

close (CDS);
print STDERR "Done.\n";

## SET UP HMM MATRIX INITIAL VALUES ##
# (STRT|STOP|DONR|ACPT).(1-12).(ATGC)<tab>probability (or log) score

use vars qw(%strt %stop %acpt %donr);
use vars qw(%strtN %stopN %acptN %donrN);
use vars qw(%allStrtP %allStrtN %allDonrP %allDonrN %allAcptP %allAcptN
%allStopP %allStopN);
use vars qw($allstart3 $allstart6 $allstop3 $allstop6);
use vars qw($allaccept3 $allaccept6 $alldonor3 $alldonor6);
use vars qw($allstartN $allstopN $allacceptN $alldonorN);
use vars qw(%stopC %acceptS %donorS $startC %triplets %doublets %bases);

$main::bases{A} = 1;
$main::bases{T} = 1;
$main::bases{G} = 1;
$main::bases{C} = 1;

$main::stopC{TGA} = 1;
$main::stopC{TAA} = 1;
$main::stopC{TAG} = 1;
```



```

$main::acceptS = "AG";
$main::donorS = "GT";

$main::startC{ATG} = 1;

$main::triplets{TGA} = 1;
$main::triplets{TAA} = 1;
$main::triplets{TAG} = 1;
$main::triplets{ATG} = 1;

$main::doublets{"GT"} = 1;
$main::doublets{AG} = 1;

## ZERO COUNTS FOR ALL BASES TO ZERO

for ($i = 1; $i <= 12; $i++)
{
    $main::allStrtP->{$i} = 0;
    $main::allStrtN->{$i} = 0;
    $main::allDonrP->{$i} = 0;
    $main::allDonrN->{$i} = 0;
    $main::allAcptP->{$i} = 0;
    $main::allAcptN->{$i} = 0;
    $main::allStopP->{$i} = 0;
    $main::allStopN->{$i} = 0;

    foreach $b (keys(%bases))
    {
        $main::strt->{$i}->{$b} = 0;
        $main::stop->{$i}->{$b} = 0;
        $main::donr->{$i}->{$b} = 0;
        $main::acpt->{$i}->{$b} = 0;

        $main::strtN->{$i}->{$b} = 0;
        $main::stopN->{$i}->{$b} = 0;
        $main::donrN->{$i}->{$b} = 0;
        $main::acptN->{$i}->{$b} = 0;
    }
}

print STDERR "Done.\n";

open (FASTA, $ARGV[1]) ||
    die "Cannot open $ARGV[1] for reading!!\n";

$fasta = $seq = "";

print STDERR "Reading fasta file\n";

while (<FASTA>)
{
    chomp;
    if ($_ =~ />/) ## TITLE LINE
    {
        ## SCORE PREVIOUS SEQUENCE
        $fasta =~ tr/atgcn/ATGCN/;
        scoreSeq($seq, $fasta);

        print "Neg Score\n";
        negScore($seq, $fasta);
    }
}

```

```

        ## PARSE NEW SEQUENCE
        ($seq) = split('\t');
        $seq =~ s/>//;
        $fasta = "";
    }
    else
    {
        $fasta .= $_;
    }
}

close (FASTA);
print STDERR "Done\n";

print STDERR "Tabulating Scores....\n";

for ($i = 1; $i <= 12; $i++)
{
    if (!$main::allStrtP->{$i}) { $main::allStrt->{$i} = 1; }
    if (!$main::allStopP->{$i}) { $main::allStrt->{$i} = 1; }
    if (!$main::allDonrP->{$i}) { $main::allStrt->{$i} = 1; }
    if (!$main::allAcptP->{$i}) { $main::allStrt->{$i} = 1; }

    if (!$main::allStrtN->{$i}) { $main::allStrt->{$i} = 1; }
    if (!$main::allStopN->{$i}) { $main::allStrt->{$i} = 1; }
    if (!$main::allDonrN->{$i}) { $main::allStrt->{$i} = 1; }
    if (!$main::allAcptN->{$i}) { $main::allStrt->{$i} = 1; }
}

open (TAB, ">hmm_table");
for ($i = 1; $i <= 12; $i++)
{
    foreach $b (keys(%bases))
    {
        # POSTIVE SCORES
        print TAB sprintf "STRT.P.$i.$b\t%.10f\t%.2f\t%.2f\n", $main::strt->{$i}-
>{$b}/$main::allStrtP->{$i}, $main::strt->{$i}->{$b}, $main::allStrtP->{$i};
        print TAB sprintf "STOP.P.$i.$b\t%.10f\t%.2f\t%.2f\n", $main::stop->{$i}-
>{$b}/$main::allStopP->{$i}, $main::stop->{$i}->{$b}, $main::allStopP->{$i};
        print TAB sprintf "DONR.P.$i.$b\t%.10f\t%.2f\t%.2f\n", $main::donr->{$i}-
>{$b}/$main::allDonrP->{$i}, $main::donr->{$i}->{$b}, $main::allDonrP->{$i};
        print TAB sprintf "ACPT.P.$i.$b\t%.10f\t%.2f\t%.2f\n", $main::acpt->{$i}-
>{$b}/$main::allAcptP->{$i}, $main::acpt->{$i}->{$b}, $main::allAcptP->{$i};

        # NEGATIVE SCORES
        print TAB sprintf "STRT.N.$i.$b\t%.10f\t%.2f\t%.2f\n", $main::strtN-
>{$i}->{$b}/$main::allStrtN->{$i}, $main::strtN->{$i}->{$b}, $main::allStrtN-
>{$i};
        print TAB sprintf "STOP.N.$i.$b\t%.10f\t%.2f\t%.2f\n", $main::stopN-
>{$i}->{$b}/$main::allStopN->{$i}, $main::stopN->{$i}->{$b}, $main::allStopN-
>{$i};
        print TAB sprintf "DONR.N.$i.$b\t%.10f\t%.2f\t%.2f\n", $main::donrN-
>{$i}->{$b}/$main::allDonrN->{$i}, $main::donrN->{$i}->{$b}, $main::allDonrN-
>{$i};
        print TAB sprintf "ACPT.N.$i.$b\t%.10f\t%.2f\t%.2f\n", $main::acptN-
>{$i}->{$b}/$main::allAcptN->{$i}, $main::acptN->{$i}->{$b}, $main::allAcptN-
>{$i};
    }
}

```

```

## SUBROUTINES ##

sub errorCheck
{
    foreach $seq (keys(%main::seqs))
    {
        my $idx = $main::seqs{$seq};
        if (exists $main::dups{$seq})
        {
            $main::dups{$seq} .= $main::seqs{$seq};
            @line = split('\t', $main::dups{$seq});
            $numDups = @line;
            for ($i = 0; $i < $numDups; $i++)
            {
                $idx = $line[$i];
                print "$idx\t".
                    "$seq\t".
                    "$main::complement[$idx]\t".
                    "$main::numExons[$idx]\t";
                for ($j = 0; $j < $main::numExons[$idx]; $j++)
                {
                    print "$main::start[$idx][$j]\t".
                        "$stop[$idx][$j]\t";
                }
                print "\n";
            }
        }
        else
        {
            print "$idx\t".
                "$seq\t".
                "$main::complement[$idx]\t".
                "$main::numExons[$idx]\t".
                "$main::start[$idx][0]\t".
                "$stop[$idx][0]\n";
        }
    }
}

sub parseLine
{
    my $idx = $main::seqidx;

    ## SPLIT LINE, GET DATA THAT YOU NEED ##
    @line = split('\t', $_[0]);
    my ($seq, $cds) = @line[0,2];

    ## INDEX SEQUENCES TO ASSOCIATE THEM WITH ARRAY INDEX ##
    if (exists $main::seqs{$seq}) ## THERE IS A DUPLICATE ##
    {
        $main::dups{$seq} .= "$main::seqs{$seq}\t";
    }
    $main::seqs{$seq} = $idx;

    ## PARSE CDS ##
    parseCDS($cds);
}

sub parseCDS
{

```

```

my $cds = $_[0];
## filter out bad cds lines ##
#if ($_ =~ /\t(complement){0,1}(\() {0,1}(join){0,1}(\() {0,1}<{0,1}([0-9]+\.\.){1}>){0,1}[0-9]+(\,[0-9]+\.\.[0-9]+)*(\)) {0,1}/)
#{
  if ($cds =~ /complement/) ## ALL CDS NEED TO BE COMPLEMENTED
  {
    $cds =~ s/\\/g;
    push(@main::complement, 1);
    ($temp, @rest) = split('\(', $cds);

    if ($rest[0] =~ /join/) ## MULTIPLE CDS REGIONS
    {
      setMultiCDS(@rest);
    }
    else # SINGLE CDS
    {
      setSingleCDS($rest[0]);
    }
  }
  else ## NO COMPLEMENT
  {
    push(@main::complement, 0);

    if ($cds =~ /join/) ## MULTIPLE CDS REGIONS
    {
      $cds =~ s/\\/g;
      (@rest) = split('\(', $cds);
      setMultiCDS(@rest);
    }
    else # SINGLE CDS
    {
      setSingleCDS($cds);
    }
  }
}

sub setSingleCDS
{
  my $idx = $main::seqidx;
  ($begin, $end) = split('\.\.', $_[0]);
  $main::start[$idx][0] = $begin;
  $main::stop[$idx][0] = $end;
  push (@main::numExons, 1);
}

sub setMultiCDS
{
  my $idx = $main::seqidx;
  ($temp, $exons) = @_;
  @pairs = split(',', $exons);
  my $num = @pairs;
  push (@main::numExons, $num);
  for ($i = 0; $i < $num; $i++)
  {
    ($begin, $end) = split('\.\.', $pairs[$i]);
    $main::start[$idx][$i] = $begin;
    $main::stop[$idx][$i] = $end;
  }
}

```

```

sub scoreSeq
{
    ($seq, $fasta) = @_ ;
    print "$seq\n";
    if (exists $main::seqs{$seq})
    {
        # score sequence
        # FIND ALL SPLICE SITES

        # CHECK IF SPLICE SITE IS A CDS
        # ARE THERE MULTIPLE ENTRIES FOR THIS SEQUENCE?

        if (exists $main::dups{$seq})
        {
            print "DUP $seq\n";
            $main::dups{$seq} .= $main::seqs{$seq};
            $save = $main::dups{$seq};
            @line = split('\t', $save);
            $numDups = @line;
            for ($x = 0; $x < $numDups; $x++)
            {
                $idx = $line[$x];
                $numE = $main::numExons[$idx];
                $comp = $main::complement[$idx];

                if ($numE == 1)
                {
                    $s1 = $main::start[$idx][0];
                    $s2 = $main::stop[$idx][0];
                    scoreOuter($s1, $s2, $comp, $fasta);
                }
                else
                {
                    $s1 = $main::start[$idx][0];
                    $s2 = $main::stop[$idx][$numE-1];
                    scoreOuter($s1, $s2, $comp, $fasta);
                    print "START: $s1 STOP: $s2\n";
                    for ($e = 1; $e < $numE; $e++)
                    {
                        $i1 = $main::stop[$idx][$e-1];
                        $i2 = $main::start[$idx][$e];
                        print "SI istop = $i1, istart = $i2\n";
                        scoreInner($i1, $i2, $comp, $fasta);
                    }
                }
            }
        }
    }
    else ## NO DUPLICATES
    {
        print "NO DUP: $seq\n";
        $idx = $main::seqs{$seq};
        $numE = $main::numExons[$idx];

        $comp = $main::complement[$idx];
        print "NumE: $numE Idx: $idx Comp: $comp\n";
        if ($numE == 1)
        {
            $s1 = $main::start[$idx][0];
            $s2 = $main::stop[$idx][0];
            scoreOuter($s1, $s2, $comp, $fasta);
        }
    }
}

```

```

else
{
    $s1 = $main::start[$idx][0];
    $s2 = $main::stop[$idx][$numE-1];
    scoreOuter($s1, $s2, $comp, $fasta);
    print "START: $s1 STOP: $s2 NumE: $numE Idx: $idx\n";
    for ($e = 1; $e < $numE; $e++)
    {
        print "$e STOP $main::stop[$idx][$e]\n";
        $i1 = $main::stop[$idx][$e-1];
        $i2 = $main::start[$idx][$e];
        print "SI istop = $i1, istart = $i2\n";
        scoreInner($i1, $i2, $comp, $fasta);
    }
}
}
}

sub scoreOuter
{
    ($st, $sp, $c, $fasta) = @_ ;
    print "SO: Start: $st Stop: $sp \n";

    $length = length($fasta);

    if ($st !~/>|</)
    {
        $st--;

        print "Start: $st\n";
        if ( ($st >= 6) and ($length - $st > 6) )## ENOUGH NUCL TO GET UPSTREAM
        {
            print "Regular Start - Stop\n";
            $score_index = 1;
            if ($c)
            {
                print "Complement\n";
                for ($i = $st + 5; $i >= $st - 6; $i--)
                {
                    $base = substr($fasta, $i, 1);
                    $base =~ tr/ATGC/TACG/;
                    print "Reverse Stop - Base = $base, $i $score_index\n";
                    $main::allStopP->{$score_index}++;
                    $main::stop->{$score_index}->{$base}++;
                    $score_index++;
                }
            }
            else
            {
                print "Regular\n";
                for ($i = $st - 6; $i < $st + 6; $i++)
                {
                    $base = substr($fasta, $i, 1);
                    print "ScoreOuter - Base = $base, $i $score_index\n";
                    $main::allStrtP->{$score_index}++;
                    $main::strt->{$score_index}->{$base}++;
                    $score_index++;
                }
            }
        }
    }
}

```

```

    }
}
else
{
    print "Short Start - Stop = $st\n";
    if ($c)
    {
        print "Complement\n";
        if ($st >= 6)
        {
            print "Complement start at end of sequence, $length $st\n";
            $score_index = 7 - ($length - $st);
            $str = $length;
            $stp = $st - 6;
        }
        else
        {
            print "Complement start not at end of sequence, $length $st\n";
            $score_index = 7 - $st;
            $str = $st + 6;
            $stp = 0;
        }

        for ($i = $str; $i > $stp; $i--)
        {
            $base = substr($fasta, $i, 1);
            $base =~ tr/ATGC/TACG/;
            print "Reverse Start - Base = $base, $i $score_index\n";
            $main::allStopP->{$score_index}++;
            $main::stop->{$score_index}->{$base}++;
            $score_index++;
        }
    }
    else
    {
        print "Regular\n";
        if ($st >= 6) # at end of sequence
        {
            $score_index = 7 - ($length - $st);
            $stp = $length;
            $str = $st - 6;
        }
        else # at beginning of sequence
        {
            $score_index = 7 - $st;
            $stp = $st + 6;
            $str = 0;
        }
        for ($i = $str; $i < $stp; $i++)
        {
            $base = substr($fasta, $i, 1);
            print "ScoreOuter short - Base = $base, $i $score_index\n";
            $main::allStrtP->{$score_index}++;
            $main::strt->{$score_index}->{$base}++;
            $score_index++;
        }
    }
}
}
}

```

```

if ($sp !~/>|</)
{
    print "Stop: $sp\n";
    if ($length - $sp > 6) ## ENOUGH NUCL TO GET DOWNSTREAM 6
    {
        print "Regular Stop - Start\n";
        $score_index = 1;
        if ($c)
        {
            print "Complement\n";
            for ($i = $sp + 5; $i >= $sp - 6; $i--)
            {
                $base = substr($fasta, $i, 1);
                $base =~ tr/ATGC/TACG/;
                print "Reverse Strt - Base = $base, $i $score_index\n";
                $main::allStrtP->{$score_index}++;
                $main::strt->{$score_index}->{$base}++;
                $score_index++;
            }
        }
        else
        {
            print "Regular\n";
            for ($i = $sp - 6; $i < $sp + 6; $i++)
            {
                $base = substr($fasta, $i, 1);
                print "ScoreOuter SP - Base = $base, $i $score_index\n";
                $main::allStopP->{$score_index}++;
                $main::stop->{$score_index}->{$base}++;
                $score_index++;
            }
        }
    }
    else
    {
        print "Short Stop - Start = $sp\n";
        if ($c)
        {
            print "Complement\n";
            $score_index = 7 - ($length - $sp);
            $str = $length;
            $stp = $sp - 6;

            for ($i = $str; $i > $stp; $i--)
            {
                $base = substr($fasta, $i, 1);
                $base =~ tr/ATGC/TACG/;
                print "Reverse Start short - Base = $base, $i $score_index\n";
                $main::allStrtP->{$score_index}++;
                $main::strt->{$score_index}->{$base}++;
                $score_index++;
            }
        }
        else
        {
            print "Regular\n";
            $score_index = 1;
            $str = $sp - 6;
            $stp = $length;

```



```

        for ($i = $str; $i < $stp; $i++)
        {
            $base = substr($fasta, $i, 1);
            print "ScoreOuter SPS - Base = $base, $i $score_index\n";
            $main::allStopP->{$score_index}++;
            $main::stop->{$score_index}->{$base}++;
            $score_index++;
        }
    }
}

sub scoreInner
{
    ($istop, $istart, $c, $fasta) = @_ ;

    if ($istop !-/>|</)
    {
        print "Inner ";
        if ($c)
        {
            print "RevDonor\n";
            $score_index = 1;
            $str = $istop + 5;
            $stp = $istop - 6;
            for ($i = $str; $i >= $stp; $i--)
            {
                $base = substr($fasta, $i, 1);
                $base =~ tr/ATGC/TACG/;
                print "$base, $i $score_index\n";
                $main::allAcptP->{$score_index}++;
                $main::acpt->{$score_index}++;
                $score_index++;
            }
        }
        else
        {
            print "Donor\n";
            $score_index = 1;
            $str = $istop - 6;
            $stp = $istop + 6;
            for ($i = $str; $i < $stp; $i++)
            {
                $base = substr($fasta, $i, 1);
                print "$base, $i $score_index\n";
                $main::allDonrP->{$score_index}++;
                $main::donr->{$score_index}->{$base}++;
                $score_index++;
            }
        }
    }

    if ($istart !-/>|</)
    {
        $istart--;
        print "Inner ";
        if ($c)
        {
            print "RevAcptr\n";

```

```

$score_index = 1;
$str = $istart + 5;
$stp = $istart - 6;
for ($i = $str; $i >= $stp; $i--)
{
    $base = substr($fasta, $i, 1);
    $base =~tr/ATGC/TACG/;
    print "$base, $i $score_index\n";
    $main::allDonrP->{$score_index}++;
    $main::donr->{$score_index}->{$base}++;
    $score_index++;
}
}
else
{
    print "Acptr\n";
    $score_index = 1;
    $str = $istart - 6;
    $stp = $istart + 6;
    for ($i = $str; $i < $stp; $i++)
    {
        $base = substr($fasta, $i, 1);
        print "$base, $i $score_index\n";
        $main::allAcptP->{$score_index}++;
        $main::acpt->{$score_index}->{$base}++;
        $score_index++;
    }
}
}

sub revComp
{
    my $reverse = "";
    $f = $orig = $_[0];
    $f =~tr/ATGC/TACG/;
    for ($i = length($f) - 1; $i >= 0; $i--)
    {
        $reverse .= substr($f, $i, 1);
    }
    return $reverse;
}

sub negScore
{
    ($seq, $fasta) = @_ ;
    $slength = length($fasta);

    if (exists $main::seqs{$seq})
    { ## SUBTRACT EXISTING KNOWN EXON SITES
        print "NegScore: $seq\n";
        for ($a = 0; $a < $slength - 9; $a++)
        {
            $codon = substr($fasta, $a, 3);
            $dimer = substr($fasta, $a, 2);
            if (exists $triplets{$codon})
            {
                if (exists $stopC{$codon})
                {
                    print "Stop Codon ";
                    if (exists $main::dups{$seq})

```

```

{
    print "DUP\n";
    # check all dups
    $save2 = $main::dups{$seq};
    @dupdata = split('\t', $save2);
    $numDup = @dupdata;
    for ($y = 0; $y < $numDup; $y++)
    {
        if (!sameStop($a, $seq))
        {
            $score_index = 1;
            $str = $a - 3;
            $stp = $a + 9;
            for ($i = $str; $i < $stp; $i++)
            {
                $base = substr($fasta, $i, 1);
                print "$base, $i, $score_index\n";
                $main::allStopN->{$score_index}++;
                $main::stopN->{$score_index}->{$base}++;
                $score_index++;
            }
        }
    }
}
else # NO DUPS, ONLY ONE SEQUENCE TO CHECK
{
    print "No DUP\n";
    if (!sameStop($a, $seq))
    {
        if ($slength - $a >= 6)
        {
            $score_index = 1;
            $str = $a - 3;
            $stp = $a + 9;
            for ($i = $str; $i < $stp; $i++)
            {
                $base = substr($fasta, $i, 1);
                print "$base, $i, $score_index\n";
                $main::allStopN->{$score_index}++;
                $main::stopN->{$score_index}->{$base}++;
                $score_index++;
            }
        }
    }
}

}
else ## START CODON
{
    print "Start Codon ";
    if (exists $main::dups{$seq})
    {
        print "DUP\n";
        # check all dups
        $save2 = $main::dups{$seq};
        @dupdata = split('\t', $save2);
        $numDup = @dupdata;
        for ($y = 0; $y < $numDup; $y++)
        {
            if (!sameStop($a, $seq))
            {

```

```

        $score_index = 1;
        $str = $a - 6;
        $stp = $a + 6;
        for ($i = $str; $i < $stp; $i++)
        {
            $base = substr($fasta, $i, 1);
            print "$base, $i, $score_index\n";
            $main::allStrtN->{$score_index}++;
            $main::strtN->{$score_index}->{$base}++;
            $score_index++;
        }
    }
}
else # NO DUPS, ONLY ONE SEQUENCE TO CHECK
{
    print "No DUP\n";
    if (!sameStart($a, $seq))
    {
        if ( ($a >= 6) and ($slength - $a >= 6) )
        {
            $score_index = 1;
            $str = $a - 6;
            $stp = $a + 6;
            for ($i = $str; $i < $stp; $i++)
            {
                $base = substr($fasta, $i, 1);
                print "$base, $i, $score_index\n";
                $main::allStrtN->{$score_index}++;
                $main::strtN->{$score_index}->{$base}++;
                $score_index++;
            }
        }
    }
}

if (exists $doublets{$dimer})
{
    if ($dimer eq $acceptS)
    {
        print "Acceptor ";
        if (exists $main::dups{$seq})
        {
            print "DUP\n";
            # check all dups
            $save2 = $main::dups{$seq};
            @dupdata = split('\t', $save2);
            $numDup = @dupdata;
            for ($y = 0; $y < $numDup; $y++)
            {
                if (!contained($a, $seq))
                {
                    $score_index = 1;
                    $str = $a - 4;
                    $stp = $a + 8;
                    for ($i = $str; $i < $stp; $i++)
                    {
                        $base = substr($fasta, $i, 1);
                        print "$base, $i, $score_index\n";

```

```

        $main::allAcptN->{$score_index}++;
        $main::acptN->{$score_index}->{$base}++;
        $score_index++;
    }
}
}
else
{
    print "No DUP\n";
    if (!contained($a, $seq))
    {
        if ( ($a >= 6) and ($slength - $a >= 6) )
        {
            $score_index = 1;
            $str = $a - 4;
            $stp = $a + 8;
            for ($i = $str; $i < $stp; $i++)
            {
                $base = substr($fasta, $i, 1);
                print "$base, $i, $score_index\n";
                $main::allAcptN->{$score_index}++;
                $main::acptN->{$score_index}->{$base}++;
                $score_index++;
            }
        }
    }
}
}
else ## DIMER = DONOR SITE
{
    print "Donor ";
    if (exists $main::dups{$seq})
    {
        print "DUP\n";
        # check all dups
        $save2 = $main::dups{$seq};
        @dupdata = split('\t', $save2);
        $numDup = @dupdata;
        for ($y = 0; $y < $numDup; $y++)
        {
            if (!contained($a, $seq))
            {
                $score_index = 1;
                $str = $a - 6;
                $stp = $a + 6;
                for ($i = $str; $i < $stp; $i++)
                {
                    $base = substr($fasta, $i, 1);
                    print "$base, $i, $score_index\n";
                    $main::allDonrN->{$score_index}++;
                    $main::donrN->{$score_index}->{$base}++;
                    $score_index++;
                }
            }
        }
    }
}
else
{
    print "No DUP\n";
    if (!contained($a, $seq))

```



```

        $main::allStrtN->{$score_index}++;
        $main::strtN->{$score_index}->{$base}++;
        $score_index++;
    }
}
}
}
if (exists $doublets{$dimer})
{
    if ($dimer eq $acceptS)
    {
        print "Acceptor\n";
        if ( ($a >= 6) and ($slength - $a >= 6) )
        {
            $score_index = 1;
            $str = $a - 4;
            $stp = $a + 8;
            for ($i = $str; $i < $stp; $i++)
            {
                $base = substr($fasta, $i, 1);
                print "$base, $i, $score_index\n";
                $main::allAcptN->{$score_index}++;
                $main::donrN->{$score_index}->{$base}++;
                $score_index++;
            }
        }
    }
    else ## DIMER = DONOR SITE
    {
        print "Donor\n";
        if ( ($slength - $a >= 6) and ($a >= 6) )
        {
            $score_index = 1;
            $str = $a - 6;
            $stp = $a + 6;
            for ($i = $str; $i < $stp; $i++)
            {
                $base = substr($fasta, $i, 1);
                print "$base, $i, $score_index\n";
                $main::allDonrN->{$score_index}++;
                $main::donrN->{$score_index}->{$base}++;
                $score_index++;
            }
        }
    }
}
}
}
}

sub sameStop
{
    ($coord, $s) = @_ ;
    $same = 0;
    $localidx = $main::seqs{$s};
    $localstop = $main::stop[$localidx][0];
    if ($localstop !~/>|</)
    {
        if ($localstop == $coord)
    }

```

```

        {
            $same = 1;
        }
        else
        {
            $same = 0;
        }
    }
    return $same;
}

sub sameStart
{
    ($coord, $s) = @_;
    $same = 0;
    $localidx = $main::seqs{$s};
    $localstart = $main::start[$localidx][0];
    if ($localstart !~/>|</)
    {
        if ($localstart == $coord)
        {
            $same = 1;
        }
        else
        {
            $same = 0;
        }
    }
    return $same;
}

sub contained
{
    ($coord, $se) = @_;
    $in = 0;
    $cidx = $main::seqs{$se};
    if ($main::numExons[$cidx] == 1)
    {
        $in = 0;
    }
    else
    {
        for ($t = 0; $t < $main::numExons[$cidx]; $t++)
        {
            $lstart = $main::start[$cidx][$t];
            $lstop = $main::stop[$cidx][$t];
            if ( ($lstart !~/>|</) and ($lstart == $coord) )
            {
                $in = 1;
            }
            if ( ($lstop !~/>|</) and ($lstop == $coord) )
            {
                $in = 1;
            }
        }
    }
    return $in;
}

```


Appendix C

Lda.c Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define DATA "mouse_lda_table"
#define FASTA "fasta_file"

#define START_THRESHOLD 0.9
#define DONOR_THRESHOLD 0.6
#define ACCEPTOR_THRESHOLD 0.6
#define STOP_THRESHOLD 0.0

// codon and hexon coordinates do not overlap.  only one 2D array is needed
float scores[32][4096];

float* seqscore;
int* seqtype;
int* activePos;

int raisePower (int values[], int num, int power);
int typeVal(char* t);
int locationVal(char l[2]);

int raisePower (int values[], int num, int power)
{
    int mult, result, i;
    mult = 1;
    result = 0;

    for (i = 0; i < num; i++)
    {
        result += values[i]*mult;
        mult *= power;
    }

    return result;
}

int typeVal(char* t)
{
    int val = 0;

    if (strcmp(t, "STRT") == 0)
    {
        val = 0;
    }
    else if (strcmp(t, "STOP") == 0)
    {
        val = 1;
    }
    else if (strcmp(t, "ACPT") == 0)
    {
        val = 2;
    }
    else // type = DONR
```

```

    {
        val = 3;
    }

// printf("calc: %i\n", val);
return val;
}

int locationVal(char l[2])
{
    int val = 0;

    if (strcmp(l, "UP") == 0)
    {
        val = 0;
    }
    else
    {
        val = 1;
    }

    return val;
}

int lengthVal(int len)
{
    int val = 0;
    if (len == 3)
    {
        val = 0;
    }
    else
    {
        val = 1;
    }
    return val;
}

int pnVal(char pn[1])
{
    int val = 0;
    if (strcmp(pn, "P") == 0)
    {
        val = 0;
    }
    else
    {
        val = 1;
    }
    return val;
}

int returnCol(char seq[6], int len)
{
    int i, col;
    int val[6];
    col = 0;

    for (i = 0; i < len; i++)

```

```

{
    if ( (seq[i] == 'A') || (seq[i] == 'a') )
    {
        val[i] = 0;
    }
    else if ( (seq[i] == 'T') || (seq[i] == 't') )
    {
        val[i] = 1;
    }
    else if ( (seq[i] == 'G') || (seq[i] == 'g') )
    {
        val[i] = 2;
    }
    else if ( (seq[i] == 'C') || (seq[i] == 'c') )
    {
        val[i] = 3;
    }
}
col = raisePower(val, len, 4);
return col;
}

void returnLocation (char label[20], int* row, int* col, int* type)
{
    char *tok;
    char location[2];
    char typ[4];
    int length, v;
    char posneg[1];
    char str[6];
    int valConvert[3];

    tok = strtok(label, ".");

    strcpy(typ, tok);
    v = typeVal(typ);
    *row = v * 8;

    tok = strtok(NULL, ".");
    strcpy(location, tok);
    v = locationVal(location);

    valConvert[0] = v;

    tok = strtok(NULL, ".");
    length = atoi(tok);
    v = lengthVal(length);
    valConvert[1] = v;

    tok = strtok(NULL, ".");
    strcpy(posneg, tok);
    v = pnVal(posneg);
    valConvert[2] = v;

    v = raisePower(valConvert, 3, 2);
    *row += v;

    tok = strtok(NULL, ".");
    strcpy(str, tok);

```

```

    *col = returnCol(str, length);
    *type = length;
}

int checkN (char* fasta, int len)
{
    int i, pos;
    pos = -1;
    for (i = 0; i < len; i++)
    {
        if ( (fasta[i] == 'N') || (fasta[i] == 'n') )
        {
            pos = i;
            fasta[i] = 'a';
        }
    }

    return pos;
}

/* purpose: given a position in the sequence, process upstream and downstream
codon scores
   receives: sequence, position, type (strt, stop, acpt, donr)
   returns: codon score for that position
*/

float processCodon(char* sequence, int pos, char* type)
{
    float pscore, nscore, score = 0;
    int prow, nrow, col, val;
    char fasta[3];
    int valConvert[4];
    /* lookup 4 scores and add them up:
       + positive codon upstream
       + position codon downstream
       - negative codon upstream
       - negative codon downstream
    */

    // DOWNSTREAM
    // CODON
    strncpy(fasta, sequence + pos, 3);
    fasta[3] = '\0';

    // COL
    checkN(fasta, 3);
    col = returnCol(fasta, 3);

    // POSITIVE ROW
    val = typeVal(type);
    prow = nrow = val * 8;

    val = locationVal("DN");
    valConvert[0] = val;

    val = lengthVal(3);
    valConvert[1] = val;

    val = pnVal("P");

```

```

valConvert[2] = val;

val = raisePower(valConvert, 3, 2);
prow += val;

pscore = scores[prow][col];

// NEGATIVE ROW
val = pnVal("N");
valConvert[2] = val;

val = raisePower(valConvert, 3, 2);
nrow += val;

nscore = scores[nrow][col];
score = pscore - nscore;

// UPSTREAM
// CODON
if (pos >= 3)
{
    *fasta = '\0';
    strncpy(fasta, sequence + pos - 3, 3);
    fasta[3] = '\0';

    // COL
    col = returnCol(fasta, 3);

    // POSITIVE ROW
    val = typeVal(type);
    prow = nrow = val * 8;

    val = locationVal("UP");
    valConvert[0] = val;

    val = lengthVal(3);
    valConvert[1] = val;

    val = pnVal("P");
    valConvert[2] = val;

    val = raisePower(valConvert, 3, 2);
    prow += val;

    pscore = scores[prow][col];

    // NEGATIVE ROW
    val = pnVal("N");
    valConvert[2] = val;

    val = raisePower(valConvert, 3, 2);
    nrow += val;

    nscore = scores[nrow][col];
    score = score + pscore - nscore;
}

return score;
}

```

```

float processHexon(char* sequence, int pos, char* type)
{
    float pscore, nscore, score = 0;
    int prow, nrow, col, val;
    char fasta[6];
    int valConvert[4];
    /* lookup 4 scores and add them up:
       + positive codon upstream
       + position codon downstream
       - negative codon upstream
       - negative codon downstream
    */

    // DOWNSTREAM
    // CODON
    strncpy(fasta, sequence + pos, 6);
    fasta[6] = '\0';

    // COL
    checkN(fasta, 6);
    col = returnCol(fasta, 6);

    // POSITIVE ROW
    val = typeVal(type);
    prow = nrow = val * 8;

    val = locationVal("DN");
    valConvert[0] = val;

    val = lengthVal(6);
    valConvert[1] = val;

    val = pnVal("P");
    valConvert[2] = val;

    val = raisePower(valConvert, 3, 2);
    prow += val;

    pscore = scores[prow][col];

    // NEGATIVE ROW
    val = pnVal("N");
    valConvert[2] = val;

    val = raisePower(valConvert, 3, 2);
    nrow += val;

    nscore = scores[nrow][col];
    score = pscore - nscore;

    // UPSTREAM
    // CODON
    *fasta = '\0';
    strncpy(fasta, sequence + pos - 6, 6);
    fasta[3] = '\0';

    // COL
    col = returnCol(fasta, 6);

```

```

// POSITIVE ROW
val = typeVal(type);
prow = nrow = val * 8;

val = locationVal("UP");
valConvert[0] = val;

val = lengthVal(6);
valConvert[1] = val;

val = pnVal("P");
valConvert[2] = val;

val = raisePower(valConvert, 3, 2);
prow += val;

pscore = scores[prow][col];

// NEGATIVE ROW
val = pnVal("N");
valConvert[2] = val;

val = raisePower(valConvert, 3, 2);
nrow += val;

nscore = scores[nrow][col];
score = score + pscore - nscore;

return score;
}

void processFrame(int pos, char* sequence, char* fasta, int seqlength, int
frame)
{
    int start, middle, modifier;
    start = frame - 1;
    middle = frame + 2;
    modifier = frame * 2 + 1;

    for (pos = start; pos < middle; pos += 3)
    {
        strncpy(fasta, sequence + pos, 3);
        fasta[3] = '\0';

        if (strcmp("atg", fasta, 3) == 0)
        {
            seqscore[pos] = processCodon(sequence, pos, "STRT");
            seqtype[pos] = 1;
        }
        else if ( (strcmp("taa", fasta, 3) == 0) ||
                  (strcmp("tag", fasta, 3) == 0) ||
                  (strcmp("tga", fasta, 3) == 0) )
        {
            seqscore[pos] = processCodon(sequence, pos + 3, "STOP");
            seqtype[pos] = 4;
        }
        else if (strcmp("ag", fasta + 1, 2) == 0)
        {
            seqscore[pos] = processCodon(sequence, pos + 3, "ACPT");
            seqtype[pos] = 3;
        }
    }
}

```

```

    }
    else if (strncmp("gt", fasta, 2) == 0)
    {
        seqscore[pos] = processCodon(sequence, pos, "DONR");
        seqtype[pos] = 2;
    }

    else
    {
        seqscore[pos] = 0;
        seqtype[pos] = 0;
    }
}

for (pos = middle; pos < seqlength - modifier; pos += 3)
{
    strncpy(fasta, sequence + pos, 3);
    fasta[3] = '\0';

    if (strncmp("atg", fasta, 3) == 0)
    {
        seqscore[pos] = processCodon(sequence, pos, "STRT");
        seqscore[pos] += processHexon(sequence, pos, "STRT");
        seqtype[pos] = 1;
    }
    else if ( (strncmp("tag", fasta, 3) == 0) ||
              (strncmp("taa", fasta, 3) == 0) ||
              (strncmp("tga", fasta, 3) == 0) )
    {
        seqscore[pos] = processCodon(sequence, pos + 3, "STOP");
        seqscore[pos] += processHexon(sequence, pos, "STOP");
        seqtype[pos] = 4;
    }
    else if (strncmp("ag", fasta + 1, 2) == 0)
    {
        seqscore[pos] = processCodon(sequence, pos + 3, "ACPT");
        seqscore[pos] += processHexon(sequence, pos, "ACPT");
        seqtype[pos] = 3;
    }
    else if (strncmp("gt", fasta, 2) == 0)
    {
        seqscore[pos] = processCodon(sequence, pos, "DONR");
        seqscore[pos] += processHexon(sequence, pos, "DONR");
        seqtype[pos] = 2;
    }

    else
    {
        seqscore[pos] = 0;
        seqtype[pos] = 0;
    }
}

for (pos = seqlength - modifier; pos < seqlength; pos += 3)

```



```

{
    strncpy(fasta, sequence + pos, 3);
    fasta[3] = '\0';

    if (strncmp("atg", fasta, 3) == 0)
    {
        seqscore[pos] = processCodon(sequence, pos, "STRT");
        seqtype[pos] = 1;
    }
    else if ( (strncmp("taa", fasta, 3) == 0) ||
              (strncmp("tag", fasta, 3) == 0) ||
              (strncmp("tga", fasta, 3) == 0) )
    {
        seqscore[pos] = processCodon(sequence, pos + 3, "STOP");
        seqtype[pos] = 4;
    }
    else if (strncmp("ag", fasta + 1, 2) == 0)
    {
        seqscore[pos] = processCodon(sequence, pos + 3, "ACPT");
        seqtype[pos] = 3;
    }
    else if (strncmp("gt", fasta, 2) == 0)
    {
        seqscore[pos] = processCodon(sequence, pos, "DONR");
        seqtype[pos] = 2;
    }

    else
    {
        seqscore[pos] = 0;
        seqtype[pos] = 0;
    }
}
}

int scoreFrame(int frame, int seqlength, int printing)
{
    float saveScore, score;
    int index, pos, type, stopPos, lastStop, start, stop, maxLength;
    saveScore = 0;
    score = 0;
    index = 0;
    lastStop = 0;
    stopPos = 0;
    maxLength = 0;

    for (pos = frame - 1; pos < seqlength; pos +=3)
    {
        if (seqtype[pos] != 0)
        {
            /* 1. look for start codon
               2. look for zero or more donor/acceptor pair
               3. look for stop
            */

            if (seqtype[pos] == 1) // START
            {
                if (score == 0)
                { // FIRST START SEEN
                    if (seqscore[pos] > START_THRESHOLD) // restrict to score > .9
                        for start

```

```

        {
            score = seqscore[pos];
            activePos[index] = pos;
            index++;
        }
    }
}
else if (index > 0) // if start site already found
{
    if (seqtype[pos] == 2) // DONOR
    {
        if (seqscore[pos] > DONOR_THRESHOLD) // restrict to scores >
.45 for donor
        {
            type = seqtype[activePos[index-1]];
            if (type == 1) // START
            {
                score += seqscore[pos];
                activePos[index] = pos;
                index++;
            }
            else if (type == 2) // DONOR
            {
                if (seqscore[pos] > seqscore[activePos[index-1]])
                { // this donor score > previously added one, replace
previous
                    score = score - seqscore[activePos[index-1]] +
seqscore[pos];
                    activePos[index-1] = pos;
                }
            }
            else if (type == 3) // ACCEPTOR
            {
                score += seqscore[pos];
                activePos[index] = pos;
                index++;
            }
            else if (type == 4) // STOP
            { /* compare score of previous stop added with current
donor
                if donor score > stop score, replace stop with donor */
                if (seqscore[pos] > seqscore[activePos[index-1]])
                {
                    stopPos = activePos[index-1];
                    score = score - seqscore[activePos[index-1]] +
seqscore[pos];
                    activePos[index-1] = pos;
                    lastStop = index - 1;
                }
            }
        }
    }
}
else if (seqtype[pos] == 3) // ACCEPTOR
{
    if (seqscore[pos] > ACCEPTOR_THRESHOLD) // restrict to scores >
.6 for acceptor
    {
        printf("accept\n");
        type = seqtype[activePos[index-1]];
        // type == 1 (start) is not possible, we will ignore this
case

```

```

        if (type == 2) // DONOR
        { /* no adjustments needed, just add */
            score += seqscore[pos];
            activePos[index] = pos;
            index++;
        }
        else if (type == 3) // ACCEPTOR
        { /* compare previous acpt score to current acpt score
            if current > previous, replace previous with current */
            if (seqscore[pos] > seqscore[activePos[index-1]])
            {
                score = score - seqscore[activePos[index-1]] +
seqscore[pos];
                activePos[index-1] = pos;
            }
        }
        else if (type == 4) // STOP
        { /* compare current score to previously added (acceptor +
stop)
            if current > previous, replace acceptor+stop with
current*/
            if (seqscore[pos] > seqscore[activePos[index-2]] +
seqscore[activePos[index-1]])
            {
                lastStop = index - 1;
                stopPos = activePos[index-1];
                score = score - seqscore[activePos[index-2]]
                    - seqscore[activePos[index-1]]
                    + seqscore[pos];
                activePos[index-2] = pos;
                index--;
            }
        }
    }
}
else if (seqtype[pos] == 4) // STOP
{
    if ( (pos == seqlength - 3) || (seqscore[pos] > STOP_THRESHOLD)
) // restrict to positive stop site score > 0
    {
        type = seqtype[activePos[index-1]];
        if (type != 4)
        { // if there is currently no stop site, add it
            if (type == 1) // START
            { /* simplest case. start-->stop
                add stop, update savedScore, lastStop, score */
                score += seqscore[pos];
                saveScore = score;
                lastStop = index;
                stopPos = pos;
                activePos[index] = pos;
                index++;
            }
            else if (type == 2) // DONOR
            { /* compare stop to previous donor score
                if stop > donor, replace donor with stop, update all
*/
                stopPos = activePos[index-1];
                score = score - seqscore[activePos[index-1]] +
seqscore[pos];

```

```

        activePos[index-1] = pos;
        saveScore = score;
        lastStop = index - 1;
    }
    else if (type == 3) // ACCEPTOR
    { /* do nothing, just add stop */
        score += seqscore[pos];
        saveScore = score;
        lastStop = index;
        stopPos = pos;
        activePos[index] = pos;
        index++;
    }
}
else // (type == 4) // STOP
{ /* compare current stop to previous stop
   if current > previous, replace stop */
    if (seqscore[pos] > seqscore[activePos[index-1]])
    {
        stopPos = activePos[index-1];
        score = score - seqscore[activePos[index-1]] +
seqscore[pos];

        activePos[index-1] = pos;
        saveScore = score;
        lastStop = index - 1;
    }
}
}
}
}
}
}

    if (index >= 2)
    {
        if (seqtype[activePos[index-1]] != 4) // if last position isn't a stop,
revert
        {
            score = saveScore;
            activePos[lastStop] = stopPos;
            index = lastStop + 1;
        }

        if (index < 2)
        {
            if (printing == 1)
                printf("%f\t%i\tNo Possible Genes\n", 0.0, frame);
        }
        else
        {
            if (printing == 1)
                printf("%f\t%i\t", score, frame);

            start = activePos[0];
            stop = activePos[index - 1];
            maxLength = stop - start;
            for (pos = 0; pos < index; pos += 2)
            {
                start = activePos[pos] + 1;

```

```

        stop = activePos[pos+1] + 1;
        if (pos+2 == index)
        {
            stop += 2;
        }
        if (printing == 1)
            printf(" %i..%i", start, stop);
    }
    if (printing == 1)
        printf("\n");
}
else
{
    if (printing == 1)
        printf("%f\t%i\tNo Possible Genes\n", 0.0, frame);
}
return maxLength;
}

void processSeq(char* sequence) //, int &sites[], int &num, int score)
{
    /* STEP THROUGH SEQUENCE
    1. in 3's and 6's
        - need to step through in frame, so in groups of 3 (ex. position 012,
then 345, 678, etc. then go back and start at position 123, 456, etc. and
finally 234, 567, etc.
        - search for first start site, then continue as specified
    2. lookup corresponding score for each
    3. generate legitimate scores
    4. find max via dynamic programming
    5. record positions for max score
    */

    int seqlength, pos, index, frame, currLength, length, selectedFrame;
    float score, saveScore;
    char fasta[7];
    seqlength = strlen(sequence);

    seqscore = malloc(seqlength * sizeof(float));
    seqtype = malloc(seqlength * sizeof(int));
    activePos = malloc(seqlength * sizeof(int));

    *fasta = '\0';

    /***** PROCESS FRAMES 1-3 *****/

    frame = 1;
    processFrame(pos, sequence, fasta, seqlength, frame);

    frame = 2;
    processFrame(pos, sequence, fasta, seqlength, frame);

    frame = 3;
    processFrame(pos, sequence, fasta, seqlength, frame);

    /***** FRAMES 1-3 SCORING *****/

```

```

    saveScore = 0;
    score = 0;
    index = 0;
        selectedFrame = 1;
        currLength = 0;

    frame = 1;
    length = scoreFrame(frame, seqlength, 0);
    if (length > currLength)
    {
        selectedFrame = 1;
    }

    frame = 2;
    length = scoreFrame(frame, seqlength, 0);
    if (length > currLength)
    {
        selectedFrame = 2;
    }

    frame = 3;
    length = scoreFrame(frame, seqlength, 0);
    if (length > currLength)
    {
        selectedFrame = 3;
    }

    length = scoreFrame(selectedFrame, seqlength, 1);

    free(seqtype);
    free(seqscore);
    free(activePos);

}

int main (int argc, const char * argv[])
{
    FILE *fp, *fasta;
    char str[20];
    float score;
    int row, col, type, len; //, x, y;
    char line[200];
    char seq[12], prevseq[12];
    char *fastaseq;

    if (argc < 3)
    {
        printf("Usage: %s <LDA Weight Table> <Fasta File>\n", argv[0]);
        return EXIT_FAILURE;
    }

    fp = fopen(argv[1], "r");
    if (fp == NULL)
    {
        printf("Cannot open %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }

```

```

/* READ IN GENERATED SCORES */
while (fscanf(fp, "%s\t%f", str, &score) != 1)
{
    if (ferror(fp))
    {
        fclose(fp);
        break;
    }
    if (feof(fp))
    {
        fclose(fp);
        break;
    }
    returnLocation(str, &row, &col, &type);

    // type will tell you which array to put data in - triplet or hexamer
    // row & column tell you where to put it

    scores[row][col] = score;

}

fclose(fp);

/* OPEN FASTA FILE, GENERATE SCORES FOR EACH SEQUENCE */

/* 1. read in sequence
   2. look for all possible splice sites
       ATG - start, AG - 3' splice, GT 5' splice, () - stop sites
   3. using dynamic programming, find the maximum possible score from all
splice site combinations
*/

/* READ IN SEQUENCE */

fasta = fopen(argv[2], "r");
if (fasta == NULL)
{
    printf("Cannot open %s\n", argv[2]);
    exit(EXIT_FAILURE);
}

fastaseq = malloc(200000 * sizeof(char));
*fastaseq = '\0';
*prevseq = '\0';
while(fgets(line, 160, fasta))
{
    if (sscanf(line, ">%s", seq))
    {
        /* PROCESS SEQUENCE */
        len = strlen(fastaseq);
        if (len)
        {
            printf("%s\t", prevseq);
            processSeq(fastaseq);
        }
    }
}

```

```

        /* RESET SEQUENCE */
        *fastaseq = '\0';
    }
    else
    {
        strcpy(prevseq, seq);
        sscanf(line, "%s\n", line);
        strcat(fastaseq, line);
    }
}

free(fastaseq);
if (ferror(fasta))
{
    fclose(fasta);
}
if (feof(fasta))
{
    fclose(fasta);
}

fclose(fasta);

return 0;
}

```


Appendix D

Hmm.c Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define START_THRESHOLD 0.9
#define DONOR_THRESHOLD 0.6
#define ACCEPTOR_THRESHOLD 0.6
#define STOP_THRESHOLD 0.0

typedef enum{ FALSE, TRUE } Boolean;

// codon and hexon coordinates do not overlap.  only one 2D array is needed
float scores[13][32];

float* seqscore;
int* seqtype;
int* activePos;
float* pScores;
float* nScores;
int* startSites;
int* stopSites;
int* donorSites;
int* acceptorSites;
int* solutionVector;
int* maxSolutionVector;
int* filteredVector;
int* overallMaxVector;

int raisePower (int values[], int num, int power);

int spliceValue(char ss[4]);
int posNegValue(char pn[1]);
int baseValue(char b[1]);
void processWindow(char* sequence, int pos, char* type);
Boolean donorSitesLeft(int position, int numDonors);
Boolean stopSitesLeft(int position, int numStops);
void addStop();
void addDonor();
void addAcceptor();

int raisePower (int values[], int num, int power)
{
    int mult, result, i;
    mult = 1;
    result = 0;

    for (i = 0; i < num; i++)
    {
        result += values[i]*mult;
        mult *= power;
    }

    return result;
}
```

```

int spliceValue(char ss[4])
{
    int val = 0;

    if (strcmp(ss, "STRT") == 0)
    {
        val = 0;
    }
    else if (strcmp(ss, "STOP") == 0)
    {
        val = 1;
    }
    else if (strcmp(ss, "ACPT") == 0)
    {
        val = 2;
    }
    else // type = DONR
    {
        val = 3;
    }

    return val;
}

```

```

int locationVal(char l[2])
{
    int val = 0;

    if (strcmp(l, "UP") == 0)
    {
        val = 0;
    }
    else
    {
        val = 1;
    }

    return val;
}

```

```

int lengthVal(int len)
{
    int val = 0;
    if (len == 3)
    {
        val = 0;
    }
    else
    {
        val = 1;
    }
    return val;
}

```

```

int posNegValue(char p[1])
{
    int val = 0;

    if (p[0] == 'P')

```

```

    {
        val = 0;
    }
    else
    {
        val = 1;
    }

    return val;
}

int baseValue(char b[1])
{
    int val = 0;

    if ( (b[0] == 'A') || (b[0] == 'a') )
    {
        val = 0;
    }
    else if ( (b[0] == 'T') || (b[0] == 't') )
    {
        val = 1;
    }
    else if ( (b[0] == 'G') || (b[0] == 'g') )
    {
        val = 2;
    }
    else if ( (b[0] == 'C') || (b[0] == 'c') )
    {
        val = 3;
    }
    else
    {
        //printf("baseValue - unknown base: %s\n", b);
        val = 0;
    }
    return val;
}

void returnLocation (char label[20], int* row, int* col)
{
    char *tok;
    char splice_site[4];
    char pos_neg[1];
    char nucleotide[1];
    int offset, local_value;
    int convert_array[3];

    tok = strtok(label, ".");

    strcpy(splice_site, tok);
    local_value = spliceValue(splice_site);
    convert_array[0] = local_value;

    tok = strtok(NULL, ".");
    strcpy(pos_neg, tok);
    local_value = posNegValue(pos_neg);
    *col = local_value * 16;
}

```

```

    tok = strtok(NULL, ".");
    offset = atoi(tok);
    *row = offset;

    tok = strtok(NULL, ".");
    strcpy(nucleotide, tok);
    local_value = baseValue(nucleotide);
    convert_array[1] = local_value;

    local_value = raisePower(convert_array, 2, 4);
    *col += local_value;
}

int checkN (char* fasta, int len)
{
    int i, pos;
    pos = -1;
    for (i = 0; i < len; i++)
    {
        if ( (fasta[i] == 'N') || (fasta[i] == 'n') )
        {
            pos = i;
            fasta[i] = 'a';
        }
    }

    return pos;
}

/* purpose: given a position in the sequence, process window of 12 nucleotides
   receives: sequence, position, type (strt, stop, acpt, donr)
   returns: nothing, score updated in global pScores & nScores arrays
*/

void processWindow(char* sequence, int pos, char* type)
{
    char local_base[2];
    int local_start, local_stop, i, local_row, local_col, positive_col,
    negative_col, local_value;
    int convert_array[2];
    int seqlength = strlen(sequence);
    float positiveScore, negativeScore;
    int modified_count = 0;

    positiveScore = negativeScore = 0.0;

    if ( (type == "STOP") || (type == "ACPT") )
    {
        pos += 3;
    }

    /* for bases 1-6, process special, 12 nucleotide window
       not possible. Process as much as possible */

    if (pos <= 6)
    {
        local_start = 8 - pos;
        local_stop = 12;
    }
}

```

```

/* for bases seqlength - 6 -> seqlength, process special
   12 window nucleotide not possible */
else if (pos >= seqlength - 6)
{
    local_start = 1;
    local_stop = 6 + (seqlength - pos);
}

/* process normally */
else
{
    local_start = 1;
    local_stop = 12;
}

for (i = local_start; i <= local_stop; i++)
{
    /* look up values for positive and negative models */
    strncpy(local_base, sequence + pos - (7 - i), 1);
    local_base[1] = '\0';
    local_row = i;

    local_value = spliceValue(type);
    convert_array[0] = local_value;
    local_value = baseValue(local_base);
    convert_array[1] = local_value;

    local_value = raisePower(convert_array, 2, 4);
    local_col = local_value;

    positive_col = local_col + 16;
    negative_col = local_col;

    positiveScore += scores[local_row][positive_col];
    negativeScore += scores[local_row][negative_col];
}

if ( (type == "STOP") || (type == "ACPT") )
{
    pos -= 3;
}

pScores[pos] = positiveScore;
nScores[pos] = negativeScore;
}

void processFrames(char* sequence, int seqlength)
{
    int pos, numStart, numStop, numDonor, numAcceptor, totalSites;
    char fasta[4];
    numStart = numStop = numDonor = numAcceptor = totalSites = 0;

    *fasta = '\0';

    /* Processing for special case - beginning of sequence
       Upstream nucleotides cannot be calculated in full
       for positions 1-6 --> this is for later when you are
       calculating scores
    */

    for (pos = 0; pos <= seqlength - 3; pos++)

```

```

{
    strncpy(fasta, sequence + pos, 3);
    fasta[3] = '\0';

    /* Find and label locations of each of the 4 splice site types */
    if (strncmp("atg", fasta, 3) == 0)
    {
        seqtype[pos] = 1;
        solutionVector[pos] = 0;    /* initialize solutions to 0 */
        numStart++;
        totalSites++;
        processWindow(sequence, pos, "STRT");
    }
    else if ( (strncmp("taa", fasta, 3) == 0) ||
              (strncmp("tag", fasta, 3) == 0) ||
              (strncmp("tga", fasta, 3) == 0) )
    {
        seqtype[pos] = 4;
        solutionVector[pos] = 0;    /* initialize solutions to 0 */
        numStop++;
        totalSites++;
        processWindow(sequence, pos, "STOP");
    }
    else if (strncmp("ag", fasta + 1, 2) == 0)
    {
        seqtype[pos] = 3;
        solutionVector[pos] = 0;    /* initialize solutions to 0 */
        numAcceptor++;
        totalSites++;
        processWindow(sequence, pos, "ACPT");
    }
    else if (strncmp("gt", fasta, 2) == 0)
    {
        seqtype[pos] = 2;
        solutionVector[pos] = 0;    /* initialize solutions to 0 */
        numDonor++;
        totalSites++;
        processWindow(sequence, pos, "DONR");
    }
    else
    {
        seqtype[pos] = -1;
        solutionVector[pos] = -1;    /* for cases where solutions are not
valid, initialize solutions to -1 */
    }
}

/* allocate space to store splice site coordinates */
startSites = malloc(numStart * sizeof(int));
stopSites = malloc(numStop * sizeof(int));
donorSites = malloc(numDonor * sizeof(int));
acceptorSites = malloc(numAcceptor * sizeof(int));
}

Boolean stopSitesLeft(int position, int numStops)
{
    int i;
    Boolean left = FALSE;
    for (i = 0; i < numStops; i++)
    {

```

```

        if (stopSites[i] > position)
        {
            left = TRUE;
        }
    }

    return left;
}

Boolean donorSitesLeft(int position, int numDonors)
{
    int i;
    Boolean left = FALSE;
    for (i = 0; i < numDonors; i++)
    {
        if (donorSites[i] > position)
        {
            left = TRUE;
        }
    }

    return left;
}

Boolean acceptorSitesLeft(int position, int numAcceptors)
{
    int i;
    Boolean left = FALSE;
    for (i = 0; i < numAcceptors; i++)
    {
        if (acceptorSites[i] > position)
        {
            left = TRUE;
        }
    }

    return left;
}

void addStop(int stopIndex, int seqlength, float* maxscore)
{
    float local_score, t1;
    int i, previousFlag;

    previousFlag = 2;

    // add stop site
    solutionVector[stopSites[stopIndex]] = 1;
    // calculate score
    local_score = 0.0;
    for (i = 0; i <= seqlength - 3; i += 3)
    {
        if (solutionVector[i] != -1)
        {
            if (solutionVector[i] == 1)
            {
                local_score += pScores[i];
                if (previousFlag != 2)
                {
                    if (previousFlag != 1)
                    { // negative to positive

```

```

        local_score += -1.7;
        previousFlag = 1;
    }
    else
    {
        local_score += -0.0044;
    }
}
}
else
{
    local_score += nScores[i];
    if (previousFlag != 2)
    {
        if (previousFlag != 0)
        {
            local_score += -2;
            previousFlag = 0;
        }
        else
        {
            local_score += -0.0088;
        }
    }

    }
    previousFlag = solutionVector[i];
}
}

// compare score to max

t1 = *maxscore;

if (*maxscore != 0.0)
{
    if (*maxscore < local_score)
    {
        *maxscore = local_score;
        memcpy(maxSolutionVector, solutionVector, seqlength * sizeof(int));
        // save current splice sites
    }
}
else
{
    *maxscore = local_score;
    memcpy(maxSolutionVector, solutionVector, seqlength * sizeof(int));
}

solutionVector[stopSites[stopIndex]] = 0;
}

void addDonor(int donorIndex, int numDonors, int numAcceptors, int numStops,
int seqlength, float* maxscore)
{
    int acceptorIndex = 0;
    int donorPos = donorSites[donorIndex];
    // add donor
    solutionVector[donorSites[donorIndex]] = 1;
}

```



```

        // add all possible acceptors
        while ( (acceptorSitesLeft(donorPos, numAcceptors) == TRUE) &&
(acceptorIndex < numAcceptors) )
        {
            while (acceptorSites[acceptorIndex] < donorPos)
            {
                acceptorIndex++;
            }
            if (filteredVector[acceptorSites[acceptorIndex]] == 1)
            {
                addAcceptor(acceptorIndex, numAcceptors, numStops, numDonors,
seqlength, maxscore);
            }
            acceptorIndex++;
        }

        solutionVector[donorSites[donorIndex]] = 0;
    }

void addAcceptor(int acceptorIndex, int numAcceptors, int numStops, int
numDonors, int seqlength, float* maxscore)
{
    int stopIndex = 0;
    int donorIndex = 0;
    int acceptorPos = acceptorSites[acceptorIndex];
    float t1 = *maxscore;
    // add acceptor
    solutionVector[acceptorSites[acceptorIndex]] = 1;

    // add all possible stops
    while ( (stopSitesLeft(acceptorPos, numStops) == TRUE) && (stopIndex <
numStops) )
    {
        while (stopSites[stopIndex] < acceptorPos)
        {
            stopIndex++;
        }
        t1 = *maxscore;
        if (filteredVector[stopSites[stopIndex]] == 1)
        {
            addStop(stopIndex, seqlength, maxscore);
        }
        stopIndex++;
    }

    // then add all possible donors for this acceptor site
    while ( (donorSitesLeft(acceptorPos, numDonors) == TRUE) && (donorIndex <
numDonors) )
    {
        while (donorSites[donorIndex] < acceptorPos)
        {
            donorIndex++;
        }
        if (filteredVector[donorSites[donorIndex]] == 1)
        {
            addDonor(donorIndex, numDonors, numAcceptors, numStops, seqlength,
maxscore);
        }
        donorIndex++;
    }
}

```

```

        solutionVector[acceptorSites[acceptorIndex]] = 0;
    }

    /* purpose: given the location of splice sites, find all legitimate
    combinations
        then score each of the possibilities, keeping the greatest value
    receives: ?? frame?
    returns: ?? best scoring frame? coordinates
    */

    int findLegitimateParses(int seqlength, int frame)
    {
        int i, numStarts, numStops, numDonors, numAcceptors, allIdx;
        int st, sp, dn, ac, all;
        int startPos, stopPos, donorPos, acceptorPos, allPos;
        int startIndex, stopIndex, donorIndex, acceptorIndex;
        int minStart, maxStop, maxLength;
        float overall_score = 0.0;
        float max_overall = 0.0;

        minStart = maxStop = maxLength = 0;
        for (i = 0; i < seqlength; i++)
        {
            maxSolutionVector[i] = 0;
        }

        numStarts = numStops = numDonors = numAcceptors = allIdx = 0;
        /* populate arrays with positions of start/stop/acceptor/donors */

        for (i = frame - 1; i <= seqlength - 3; i += 3)
        {
            /* position is a splice site */
            if (seqtype[i] != -1)
            {
                if (seqtype[i] == 1)
                {
                    if (filteredVector[i] == 1)
                    {
                        startSites[numStarts] = i;
                        numStarts++;
                    }
                }
                else if (seqtype[i] == 2)
                {
                    if (filteredVector[i] == 1)
                    {
                        donorSites[numDonors] = i;
                        numDonors++;
                    }
                }
                else if (seqtype[i] == 3)
                {
                    if (filteredVector[i] == 1)
                    {
                        acceptorSites[numAcceptors] = i;
                        numAcceptors++;
                    }
                }
                else if (seqtype[i] == 4)
                {
                    if (filteredVector[i] == 1)

```

```

        {
            stopSites[numStops] = i;
            numStops++;
        }
    }
}

if (numStarts > 0)
{
    startPos = startSites[0];    // initial value
    st = sp = dn = ac = all = 0;
    while (st < numStarts)
    {
        /* case - current start position is already included in vector
        remove current start position from solution and add next one */
        if (solutionVector[startPos] == 1)
        {
            solutionVector[startPos] = 0;
        }

        startPos = startSites[st];
        if (filteredVector[startPos] == 1)
        {
            solutionVector[startPos] = 1;
            stopIndex = 0;
            donorIndex = 0;
            acceptorIndex = 0;

            // add all possible stops for this start site
            while ( (stopSitesLeft(startPos, numStops) == TRUE) &&
(stopIndex < numStops) )
            {
                while (stopSites[stopIndex] < startPos)
                {
                    stopIndex++;
                }
                if (filteredVector[stopSites[stopIndex]] == 1)
                {
                    addStop(stopIndex, seqlength, &max_overall);
                }
                stopIndex++;
            }

            // then add all possible donors for this start site

            while ( (donorSitesLeft(startPos, numDonors) == TRUE) &&
(donorIndex < numDonors) )
            {
                while (donorSites[donorIndex] < startPos)
                {
                    donorIndex++;
                }
                if (filteredVector[donorSites[donorIndex]] == 1)
                {
                    addDonor(donorIndex, numDonors, numAcceptors, numStops,
seqlength, &max_overall);
                }
                donorIndex++;
            }
            // removeLastAdded();

```

```

        st++;
    }
}

for (i = frame - 1; i <= seqlength - 3; i += 3)
{
    if (maxSolutionVector[i] == 1)
    {
        if ( (seqtype[i] == 1) && (minStart == 0) )
        {
            minStart = i;
        }
        if ( (seqtype[i] == 4) && (i > maxStop) )
        {
            maxStop = i;
        }
    }
}

maxLength = maxStop - minStart;

for (i = 0; i < seqlength - 3; i++)
{
    if (solutionVector[i] != -1)
        solutionVector[i] = 0;
}

return maxLength;
}

int scoreFrame(int frame, int seqlength, int printing)
{
    float saveScore, score;
    int index, pos, type, stopPos, lastStop, start, stop, maxLength;
    saveScore = 0;
    score = 0;
    index = 0;
    lastStop = 0;
    stopPos = 0;
    maxLength = 0;

    for (pos = frame - 1; pos < seqlength; pos +=3)
    {
        if (seqtype[pos] != 0)
        {
            /* 1. look for start codon
               2. look for zero or more donor/acceptor pair
               3. look for stop
            */

            if (seqtype[pos] == 1) // START
            {
                if (score == 0)
                { // FIRST START SEEN
                    if (seqscore[pos] > START_THRESHOLD) // restrict to score > .9
                    for start
                    {
                        score = seqscore[pos];
                        activePos[index] = pos;
                    }
                }
            }
        }
    }
}

```

```

        index++;
    }
}
else if (index > 0) // if start site already found
{
    if (seqtype[pos] == 2) // DONOR
    {
        if (seqscore[pos] > DONOR_THRESHOLD) // restrict to scores >
.45 for donor
        {
            type = seqtype[activePos[index-1]];
            if (type == 1) // START
            {
                score += seqscore[pos];
                activePos[index] = pos;
                index++;
            }
            else if (type == 2) // DONOR
            {
                if (seqscore[pos] > seqscore[activePos[index-1]])
                { // this donor score > previously added one, replace
previous
                    score = score - seqscore[activePos[index-1]] +
seqscore[pos];
                    activePos[index-1] = pos;
                }
            }
            else if (type == 3) // ACCEPTOR
            {
                score += seqscore[pos];
                activePos[index] = pos;
                index++;
            }
            else if (type == 4) // STOP
            { /* compare score of previous stop added with current
donor
                if donor score > stop score, replace stop with donor */
                if (seqscore[pos] > seqscore[activePos[index-1]])
                {
                    stopPos = activePos[index-1];
                    score = score - seqscore[activePos[index-1]] +
seqscore[pos];
                    activePos[index-1] = pos;
                    lastStop = index - 1;
                }
            }
        }
    }
}
else if (seqtype[pos] == 3) // ACCEPTOR
{
    if (seqscore[pos] > ACCEPTOR_THRESHOLD) // restrict to scores >
.6 for acceptor
    {
        type = seqtype[activePos[index-1]];
        // type == 1 (start) is not possible, we will ignore this
case
        if (type == 2) // DONOR
        { /* no adjustments needed, just add */
            score += seqscore[pos];
            activePos[index] = pos;

```

```

        index++;
    }
    else if (type == 3) // ACCEPTOR
    { /* compare previous acpt score to current acpt score
      if current > previous, replace previous with current */
      if (seqscore[pos] > seqscore[activePos[index-1]])
      {
          score = score - seqscore[activePos[index-1]] +
seqscore[pos];
          activePos[index-1] = pos;
      }
    }
    else if (type == 4) // STOP
    { /* compare current score to previously added (acceptor +
stop)
      if current > previous, replace acceptor+stop with
current*/
      if (seqscore[pos] > seqscore[activePos[index-2]] +
seqscore[activePos[index-1]])
      {
          lastStop = index - 1;
          stopPos = activePos[index-1];
          score = score - seqscore[activePos[index-2]]
- seqscore[activePos[index-1]]
+ seqscore[pos];
          activePos[index-2] = pos;
          index--;
      }
    }
}
}
else if (seqtype[pos] == 4) // STOP
{
    if ( (pos == seqlength - 3) || (seqscore[pos] > STOP_THRESHOLD)
) // restrict to positive stop site score > 0
    {
        type = seqtype[activePos[index-1]];
        if (type != 4)
        { // if there is currently no stop site, add it
            if (type == 1) // START
            { /* simplest case. start-->stop
              add stop, update savedScore, lastStop, score */
              score += seqscore[pos];
              saveScore = score;
              lastStop = index;
              stopPos = pos;
              activePos[index] = pos;
              index++;
            }
            else if (type == 2) // DONOR
            { /* compare stop to previous donor score
              if stop > donor, replace donor with stop, update all
*/
              // if (seqscore[pos] > seqscore[activePos[index-1]])
              // {
                  stopPos = activePos[index-1];
                  score = score - seqscore[activePos[index-1]] +
seqscore[pos];
                  activePos[index-1] = pos;
                  saveScore = score;
                  lastStop = index - 1;

```

```

        // }
    }
    else if (type == 3) // ACCEPTOR
    { /* do nothing, just add stop */
        score += seqscore[pos];
        saveScore = score;
        lastStop = index;
        stopPos = pos;
        activePos[index] = pos;
        index++;
    }
}
else // (type == 4) // STOP
{ /* compare curent stop to previous stop
   if current > previous, replace stop */
   if (seqscore[pos] > seqscore[activePos[index-1]])
   {
       stopPos = activePos[index-1];
       score = score - seqscore[activePos[index-1]] +
seqscore[pos];
       activePos[index-1] = pos;
       saveScore = score;
       lastStop = index - 1;
   }
}
}
}
}
}
}
}
}
}
}

    if (index >= 2)
    {
        if (seqtype[activePos[index-1]] != 4) // if last position isn't a stop,
revert
        {
            score = saveScore;
            activePos[lastStop] = stopPos;
            index = lastStop + 1;
        }

        if (index < 2)
        {
            if (printing == 1)
                printf("%f\t%i\tNo Possible Genes\n", 0.0, frame);
        }
        else
        {
            if (printing == 1)
                printf("%f\t%i\t", score, frame);

            start = activePos[0];
            stop = activePos[index - 1];
            maxLength = stop - start;
            for (pos = 0; pos < index; pos += 2)
            {
                start = activePos[pos] + 1;
                stop = activePos[pos+1] + 1;
            }
        }
    }
}

```

```

        if (pos+2 == index)
        {
            stop += 2;
        }
        if (printing == 1)
            printf(" %i..%i", start, stop);
    }
    if (printing == 1)
        printf("\n");
}
}
else
{
    if (printing == 1)
        printf("%f\t%i\tNo Possible Genes\n", 0.0, frame);
}
return maxLength;
}

void processSeq(char* sequence)
{
    /* STEP THROUGH SEQUENCE
    1. in 3's and 6's
        - need to step through in frame, so in groups of 3 (ex. position 012,
then 345, 678, etc. then go back and start at position 123, 456, etc. and
finally 234, 567, etc.
        - search for first start site, then continue as specified
    2. lookup corresponding score for each
    3. generate legitimate scores
    4. find max via dynamic programming
    5. record positions for max score
    */

    int seqlength = strlen(sequence);
    int total, modified, i, maxLength, currentLength;
    total = modified = 0;
    int maxDonor, maxAcceptor, maxStart, maxStop;
    maxDonor = maxAcceptor = maxStart = maxStop = 0;
    float temp_scores;
    int firstDonor, firstAcceptor, firstStart, firstStop;
    firstDonor = firstAcceptor = firstStart = firstStop = 0;
    maxLength = 0;

    seqscore = malloc(seqlength * sizeof(float));
    seqtype = malloc(seqlength * sizeof(int));
    activePos = malloc(seqlength * sizeof(int));
    pScores = malloc(seqlength * sizeof(float));
    nScores = malloc(seqlength * sizeof(float));
    solutionVector = malloc(seqlength * sizeof(int));
    filteredVector = malloc(seqlength * sizeof(int));
    overallMaxVector = malloc(seqlength * sizeof(int));
    maxSolutionVector = malloc(seqlength * sizeof(int));

    /***** PROCESS FRAMES 1-3 *****/

    processFrames(sequence, seqlength);

    for (i = 0; i <= seqlength - 3; i++)
    {
        if (seqtype[i] != -1)
        {

```



```

total++;
temp_scores = pScores[i] - nScores[i];
filteredVector[i] = 0;

if (seqtype[i] == 1)
{
    if ( (temp_scores > 0.0) && (i <= seqlength - 6) )
    {
        maxStart++;
        filteredVector[i] = 1;
    }
}
else if (seqtype[i] == 2)
{
    if (temp_scores > 5.0)
    {
        maxDonor++;
        filteredVector[i] = 1;
    }
}
else if (seqtype[i] == 3)
{
    if (temp_scores > 68.5)
    {
        maxAcceptor++;
        filteredVector[i] = 1;
    }
}
else if (seqtype[i] == 4)
{
    if (temp_scores > -1.0)
    {
        maxStop++;
        filteredVector[i] = 1;
    }
}

modified = maxStart + maxDonor + maxAcceptor + maxStop;
}
}

if (modified < 200)
{
    maxLength = findLegitimateParses(seqlength, 1);
    if (maxLength != 0)
    {
        memcpy(overallMaxVector, maxSolutionVector, seqlength * sizeof(int));
    }
    currentLength = findLegitimateParses(seqlength, 2);
    if (currentLength > maxLength)
    {
        maxLength = currentLength;
        memcpy(overallMaxVector, maxSolutionVector, seqlength * sizeof(int));
    }
    currentLength = findLegitimateParses(seqlength, 3);
    if (currentLength > maxLength)
    {
        maxLength = currentLength;
        memcpy(overallMaxVector, maxSolutionVector, seqlength * sizeof(int));
    }
}
}

```

```

// print results
if (maxLength != 0)
{
    for (i = 0; i <= seqlength - 3; i++)
    {
        if (overallMaxVector[i] == 1)
        {
            if (seqtype[i] == 1)
            {
                printf("%i..", i+1);
            }
            else if (seqtype[i] == 4)
            {
                printf("%i", i+3);
            }
            else if (seqtype[i] == 2)
            {
                printf("%i,", i+1);
            }
            else if (seqtype[i] == 3)
            {
                printf("%i..", i+1);
            }
        }
    }
    printf("\n");
}
else
{
    printf("No legitimate splice sites found\n");
}

free(seqtype);
free(seqscore);
free(activePos);
free(pScores);
free(nScores);
}

int main (int argc, const char * argv[])
{
    FILE *file_pointer, *fasta;
    char description[20];
    float score;
    int row, col, len; //, x, y;
    char line[200];
    char seq[12], prevseq[12];
    char *fastaseq;

    if (argc < 3)
    {
        printf("Usage: %s <HMM Table> <Fasta File>\n", argv[0]);
        return EXIT_FAILURE;
    }

    file_pointer = fopen(argv[1], "r");
    if (file_pointer == NULL)

```

```

{
    printf("Cannot open %s\n", argv[1]);
    exit(EXIT_FAILURE);
}

/* READ IN GENERATED SCORES */
while (fscanf(file_pointer, "%s\t%f", description, &score) != 1)
{
    if (ferror(file_pointer))
    {
        fclose(file_pointer);
        break;
    }
    if (feof(file_pointer))
    {
        fclose(file_pointer);
        break;
    }

    returnLocation(description, &row, &col);
    scores[row][col] = score;
    //      printf("Row: %i Col: %i Score: %f\n", row, col,
scores[row][col]);
}

fclose(file_pointer);

/* OPEN FASTA FILE, GENERATE SCORES FOR EACH SEQUENCE */

/* 1. read in sequence
   2. look for all possible splice sites
       ATG - start, AG - 3' splice, GT 5' splice, () - stop sites
   3. using dynamic programming, find the maximum possible score from all
splice site combinations
*/

/* READ IN SEQUENCE */

fasta = fopen(argv[2], "r");
if (fasta == NULL)
{
    printf("Cannot open %s\n", argv[2]);
    exit(EXIT_FAILURE);
}

fastaseq = malloc(200000 * sizeof(char));
*fastaseq = '\0';
*prevseq = '\0';
while(fgets(line, 160, fasta))
{
    if (sscanf(line, ">%s", seq))
    {
        /* PROCESS SEQUENCE */
        len = strlen(fastaseq);
        if (len)
        {
            printf("%s\t", prevseq);
            processSeq(fastaseq);
        }
        /* RESET SEQUENCE */
    }
}

```

```

        *fastaseq = '\0';
    }
    else
    {
        strcpy(prevseq, seq);
        sscanf(line, "%s\n", line);
        strcat(fastaseq, line);
    }
}

free(fastaseq);

if (ferror(fasta))
{
    fclose(fasta);
}
if (feof(fasta))
{
    fclose(fasta);
}

fclose(fasta);

return 0;
}

```