2009

MRCRAIG: MapReduce and Ensemble Classifiers for Parallelizing Data Classification Problems

Glenn Jahnke
San Jose State University

MRCRAIG: MAPREDUCE AND ENSEMBLE CLASSIFIERS FOR PARALLELIZING DATA CLASSIFICATION PROBLEMS

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Computer Science

by

Glenn Jahnke

May 2009

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Prof. Robert Chun

Prof. Cay Horstmann

Prof. Sami Khuri

APPROVED FOR THE UNIVERSITY

ABSTRACT

MRCRAIG: MAPREDUCE AND ENSEMBLE CLASSIFIERS FOR PARALLELIZING DATA CLASSIFICATION PROBLEMS

by Glenn Jahnke

In this paper, a novel technique for parallelizing data-classification problems is applied to finding genes in sequences of DNA. The technique involves various ensemble classification methods such as Bagging and Select Best. It then distributes the classifier training and prediction using MapReduce. A novel sequence classification voting algorithm is evaluated in the Bagging method, as well as compared against the Select Best method.

DEDICATION

To my mother and father for carrying me through school in so many ways, and to Randy whose friendship and mentoring for countless years will never be forgotten.

ACKNOWLEDGEMENTS

I would like to acknowledge many people who helped me in the process of making this thesis. My advisors Professor Chun, Professor Khuri, and Professor Horstmann deserve large credit for their excellence in teaching and inspiration. I would also especially like to acknowledge Tina for her kindness and hard work editing this document. Lastly, I would like to extend acknowledgement to all the other unmentioned friends and family who aided me on my academic journey, you all kept me on track in the times that I needed it the most. Thank you.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

Figure

CHAPTER 1

INTRODUCTION

In an age of plateauing single-threaded CPU performance, there is a growing importance of utilizing parallelization. Finding parallelism in algorithms can be challenging, but implementing it can be even more challenging. There are many hurdles attempting to parallelize code, from preventing deadlocks to ensuring a correct implementation. MapReduce, a recently-revived idea, makes parallelizing code much simpler if an algorithm can be formulated in terms of MapReduce's constituent functions.

Gene prediction is another challenging and current problem. Millions of base-pairs of DNA are uncovered at an astonishing pace creating a backlog of data to analyze. For instance, the Human Genome Project finished sequencing approximately 3 billion base-pairs and will require years to fully understand all the data revealed[6]. Yet determining what every portion of DNA does exactly in terms of protein production is a mediocre-at-best process; identifying gene sequences hovers just above 50 percent for even the state-of-the-art predictors[4]. This is also only including the genes which actually code for proteins. The function of the non-coding regions also has many mysteries to uncover. However, for the purposes of this paper, we will only consider protein-coding genes.

MrCRAIG's goal is to approach the gene-prediction problem in a manner that

can be easily expanded to cover other sequence classification problems, and potentially data-classification in general. This is accomplished using generally-applicable techniques without modifying the existing classifier, only the manner in which it is used. MrCRAIG starts with a highly-accurate, but slow, gene predictor named CRAIG. It then sees how far general parallelization techniques like MapReduce can be taken to increase the accuracy. It does this by utilizing properties of ensembles of classifiers. This increase in accuracy is done while attempting to keep the training and prediction time manageable.

CHAPTER 2

MOTIVATION

For years, the various CPU manufacturers enjoyed relatively easy success by simply increasing the frequency of the CPU to achieve higher performance. This continued for decades, but recent times have shown that this will not be the case in the future. CPU frequencies have plateaued, or in some cases declined, as manufacturers have reached hard physical limits on size of silicon transistors in their processors. Manufacturers have been forced to switch to a new method for increased performance: increasing the number of cores per processor while maintaining their frequencies. This has many paradigm-shifting effects.

One of the biggest effects of this new multicore era is that the majority of programming languages have been built around mostly single-threaded use-cases. Asking any programmer to write something using multiple threads in their favorite language will result in anywhere from discomfort to refusal. Further, many languages and techniques focused around parallelism that sat on the back burner for the past couple of decades have gained new support. Look at the new interest in functional languages like Erlang which has existed since the 1980's. Erlang's strengths are in its vast scalability across processors and networks, making parallel and distributed computing easier. Yet even those who use these languages built with parallelism in mind, state that the parallelism can be the most challenging part[1].

As greater performance is demanded in the coming years, matched with the growing number of processors, methods to use the many processors easily will become increasingly important. MapReduce is one such technique that allows vastly scalable processing with the guarantee of no synchronization issues. Modern implementations like Hadoop also provide other useful features like high fault tolerance, distributed logging, job queuing, and a distributed file-system. There still exists the burden of defining their problem in terms of the map and reduce functions. However, this can be a simpler task than manually managing synchronization issues and guaranteeing correctness.

A current problem in Bioinformatics is difficulty of determining where the coding regions of DNA are amongst the billions of base-pairs in human DNA. Finding the genes is important for biologists to understand the nature of how organisms function. Once an organism has been sequenced, the first step to understanding that organism is to locate all of its genes. While this will not give biologists any information about the actual function of the gene, it will tell them where to begin laboratory analysis, thus saving a great deal of expensive and time.

More precisely, gene prediction is the task of predicting the role of each nucleotide in a DNA sequence in the gene transcription/translation process. A protein-encoding gene is a segment of DNA that is ultimately used as a template for the production of a protein. Not all of an organism's DNA is composed of genes; there are large regions of unused sequences which are mostly ignored. This means genes must first be located. Next, there are many parts that make up genes, and those parts must be labeled. A gene predictor takes a DNA sequence as input and, as output, creates an annotation describing what role the different parts of the sequence play in protein production.

Biological processes can be messy and imprecise, following loose and often

broken rules. Mutations have allowed organisms to adapt and evolve over millions of years and arise solely from the fact that biological systems are not perfect. This makes searching DNA sequences difficult as the path is laden with oddities such as pseudo-genes and millennia of dormant genetic history stored within our genome.

The math behind gene prediction is also troublesome. Organisms have very large genomes with many billions of base-pairs. The percentage of genes that code for proteins can be as little as 0.5 percent which makes the problem of gene prediction even more challenging. This problem grows daily as more organisms are sequenced.

CHAPTER 3

BACKGROUND

3.1 Genetics

A gene is a sequence of DNA bases that carries information necessary to create a protein. The central dogma of molecular biology is a description of the process by which DNA is used to create proteins and is shown in Figure 3.1. Genes are extracted from DNA strands and transcribed into RNA strands which are translated into proteins.

Figure 3.1: Central Dogma of Molecular Biology

3.2 Genomes and Chromosomes

Most living organisms on Earth have genomes that contain many chromosomes which are sets of DNA wound together in the form of a double helix. The

chromosomes are located within the nucleus of cells (for higher order species) and contain all the DNA for the given species. There are anywhere from a few to a few tens of chromosomes (23 in *Homo sapiens*) and they are quite long. A chromosome can contain thousands of genes, up to 30,000 in *Homo sapiens* and amazingly this represents only a tiny fraction of the whole sequence. The coding sequences in *Homo sapiens* only takes up 1.5% of the entire sequence while the other 98% is almost completely ignored. This poses challenging problems when trying to find the genes amongst all of the non-coding regions.

3.3 Transcription

Transcription is the process in which a gene is copied from a DNA strand and an RNA molecule is produced. Each chromosome consists of two strands of DNA which are complementary to each other. When a gene is transcribed, the two complementary strands of DNA are separated and one strand is used as a template to copy the other. This copying process is called transcription, and the result is an RNA molecule which is a copy of a gene.

3.4 Translation

After transcription, there is an RNA molecule which usually is used as a template for the creation of a protein. As mentioned earlier, we will only consider genes that are used for creating proteins. The next step that must occur is translation which is the process that directly uses an RNA to build a protein.

A ribosome is the director of translation. A ribosome reads an RNA sequence one codon at a time, which is three consecutive bases, and each of these codons is associated with either an amino acid or a stop signal. The set of associations between

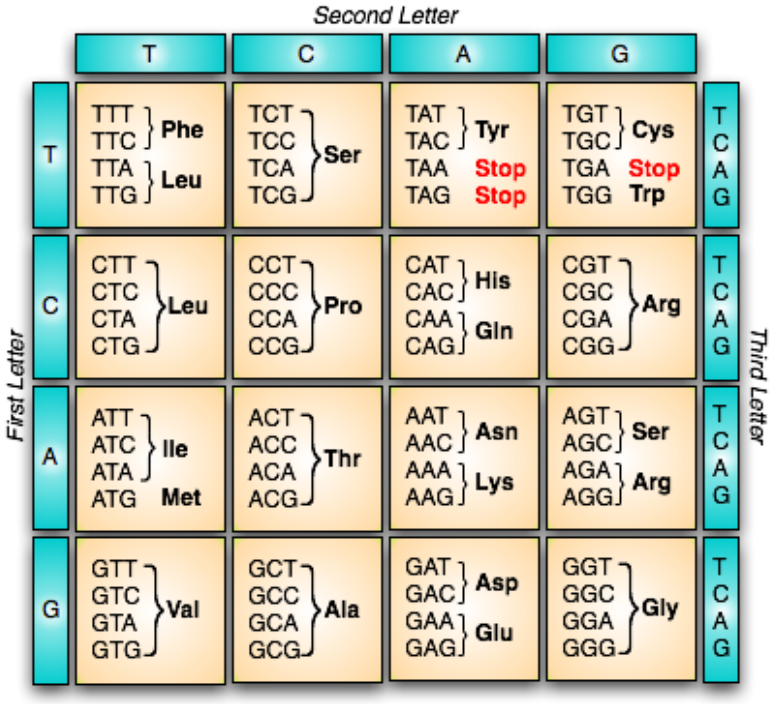


Figure 3.2: Genetic Code - source: http://plato.stanford.edu/entries/information-biological/

codons and amino acids is called the genetic code as seen in Figure3.2. The stop codon tells the ribosome when to stop translation and allows the newly created sequence of amino acids to separate and begin folding itself into the correct shape of the protein.

The whole process begins with DNA from the chromosomes from which the genes are extracted, then transcription and translation occur. This finally produces the protein that is a functional entity and can do work in the biological system. If it is possible to find the genes, it is possible to see how the proteins that make up species are made, which gives biologists a great deal of information.

3.5 Anatomy of a Gene

Genes are comprised of many parts that are all important in turning the string of nucleotides into protein. All genes consist of at least one exon followed by an alternation of introns and exons that must end in an exon. The exons in the gene contain the information about what amino acids should be generated. Introns do not explicitly contain information about the end result, but contain other information and are linked to how much of the protein should be created. This process, as seen in Figure3.3, begins with splicing which removes introns from the gene to prepare it for translation. Then, post processing occurs, followed by translation of codons into amino acids. Finally, the amino acid chain is released and constitutes the protein.

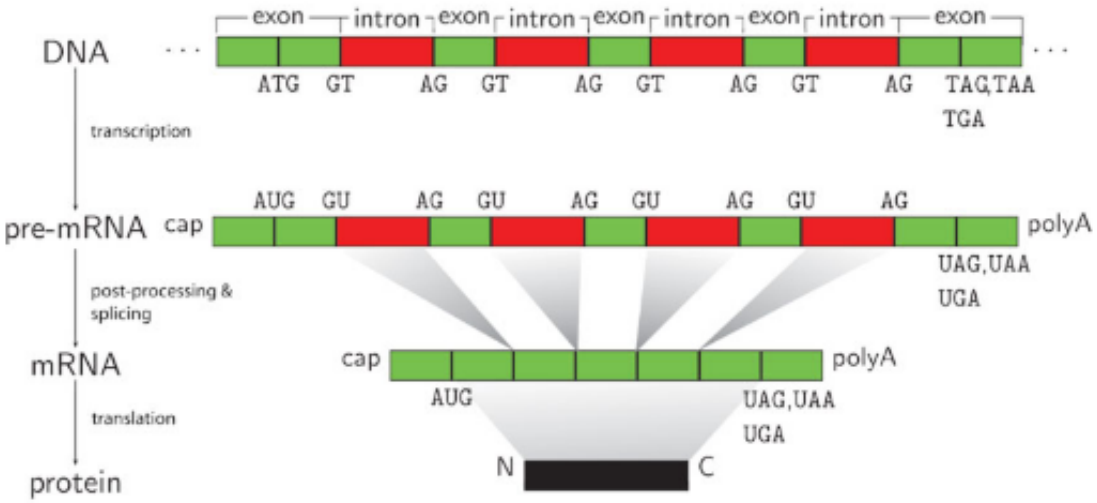


Figure 3.3: DNA to Protein - source: http://ida.first.fraunhofer.de/ida.gen/

3.6 Gene Prediction Problem

Nature has this strange way of hiding all the pertinent information amongst non-informative regions. Because of this, the best way to find genes is by identifying

attributes of the sequence that make them stand out. This is difficult because mutations may have turned what were genes into non-coding sequences called pseudo-genes, and otherwise wreak havoc on the whole process. It becomes a process of separating the noise from the signal.

Luckily, there are many hints within genes to help identify them. These are signals for which to look, many of which are used by our program to find the genes. For example, nearly all coding sequences start with the codon which is translated into the amino acid methionine, and all genes end with one of three codons that represent the stop signal. These are obvious signals that indicate the presence of genes. Other signals include the donor and acceptor splice sites which are the borders in between exons and introns and exhibit unique properties. One thing is for certain: if the process of transcription can determine where to cut up the DNA sequences, then there must be some way to predict where it will happen programmatically.

CHAPTER 4

RESEARCH

4.1 Parallelization Methods

Algorithms are primarily written in a sequential fashion because the task of parallelizing algorithms is difficult and prone to error[7]. Race conditions are difficult to detect as seemingly functioning code may break years later due to oddities in timing. There are many libraries being built to make these tasks easier, however the last mile of assuring correctness still lies in the hands of programmers. The problem worsens when jobs require more resources than a single computer can readily complete. Peak performance in clusters of computers is often difficult to achieve as heterogeneity across hardware, operating systems, and software make algorithms perform irregularly[2].

Parallelization has not severely hindered program development though, due mostly to the fact that processors have steadily been increasing in speed for decades because of Moore's Law. However, because of physical limitations of current processor design like heat dissipation, power consumption, and transistor geometry, many paradigms are changing. Improvements from Moore's Law are turning from higher clock-rate processors to an increased number of cores per processors. Simply waiting 18 months to get a significant performance boost will no longer work if code is not capable of being spread across multiple cores. This means programmers will have to

faced parallelizing their code if they desire faster execution of their programs.

4.2 MapReduce

MapReduce is an attempt to make parallelizing programs significantly simpler and more robust. It restricts the programming architecture, and by doing so alleviates the programmer of most parallelization issues like concurrency, error detection, and inter-computer communication[7]. Its map and reduce functions are founded in the Function Programming paradigm which has a history of enabling more readily parallelizable code because of the reduction in shared state.

The architecture of MapReduce works on many records which are key/value pairs that are all user-defined. The input data is split into a set of N key/value pairs, denoted by K_{in} and V_{in}, and are distributed to various machines. The records are processed by a user-defined map function where each of the N records pass through the map function. The output of the map function is an intermediate set of key/value pairs denoted K_{im} and V_{im} for a total of M output records. All of these M intermediate records are sorted by their key value, K_{im}, and redistributed amongst the machines in the cluster where all the records with key K_{im} go to the same machine. Each of the M records then passes through a reduce function, also a user-defined function, then emits a final key/value pair, K_{out} and V_{out} (usually either 0 or 1 records). Finally, an optional combine phase occurs that aggregates the results of the reduce phase output, and the algorithm is completed.[7]

A simple example of an implementation of MapReduce should clarify some details. Matrices are easy candidates for parallelization in general, and manipulating them with MapReduce is no exception. For example, matrix transposition can be parallelized in more than one way where transposition, D^T is defined by swapping

$$InputSplit : (data) \Rightarrow (K_{in}, V_{in})$$

$$Map : (K_{in}, V_{in}) \Rightarrow (K_{im}, V_{im})$$
$$Reduce : (K_{im}, V_{im}) \Rightarrow (K_{out}, V_{out})$$
$$Combine : (K_{out}, V_{out}) \Rightarrow (output)$$

Figure 4.1: MapReduce Functions

every element $D_{i,j}$ in a matrix D with the element at $D_{j,i}$. This can be done by breaking up the input matrix D into a set of row vectors where the input key is defined as the row index, $K_{in} = i$, and the input value is defined as the vector of data in the row, $V_{in} = D_i$. This can be seen in Figure 4.2. The first step of the MapReduce process is to split the matrix into rows during the input split phase. As the rows are now separate and will be moved to different computers, the row index, K_{in}, must be stored with the row data to keep track of its position in the original matrix.

$$D = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = inputsplit \Rightarrow \begin{array}{l} \left(K_{in} = 1, V_{in} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}\right) \\ \left(K_{in} = 2, V_{in} = \begin{bmatrix} 4 & 5 & 6 \end{bmatrix}\right) \\ \left(K_{in} = 3, V_{in} = \begin{bmatrix} 7 & 8 & 9 \end{bmatrix}\right) \end{array}$$

Figure 4.2: Input Split Phase

We define the map function to take in i and D_i and produce intermediate keys and values for each element in the row vector such that $K_{im} = j$ and $V_{im} = (i, (D_i)_j)$, meaning the column index is used as the key, and the value is the pair P where P_1 is the row index i and P_2 is the matrix element $(D_i)_j$ so $P = (i, (D_i)_j)$. Plainly, each input row is broken up into individual elements and enough information is stored

to track each cell's original position. This can be seen in Figure 4.3. All of the input splits are sent out to different mappers where they are each mapped from the input splits into the intermediate records. In this case, each mapper maps each input record to 3 output records, but this is not necessarily the case in general because map functions can produce any number of outputs, independent of each other.

$$\begin{aligned}
\left(K_{in} = 1, V_{in} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}\right) &= map \Rightarrow \quad \begin{array}{l} (K_{im} = 1, V_{im} = (1, [1])) \\ (K_{im} = 2, V_{im} = (1, [2])) \\ (K_{im} = 3, V_{im} = (1, [3])) \end{array} \\
\left(K_{in} = 2, V_{in} = \begin{bmatrix} 4 & 5 & 6 \end{bmatrix}\right) &= map \Rightarrow \quad \begin{array}{l} (K_{im} = 1, V_{im} = (2, [4])) \\ (K_{im} = 2, V_{im} = (2, [5])) \\ (K_{im} = 3, V_{im} = (2, [6])) \end{array} \\
\left(K_{in} = 3, V_{in} = \begin{bmatrix} 7 & 8 & 9 \end{bmatrix}\right) &= map \Rightarrow \quad \begin{array}{l} (K_{im} = 1, V_{im} = (3, [7])) \\ (K_{im} = 2, V_{im} = (3, [8])) \\ (K_{im} = 3, V_{im} = (3, [9])) \end{array}
\end{aligned}$$

Figure 4.3: Map Phase

There are now an equal number of intermediate key/value pairs as the number of elements in the original matrix. These key/value pairs are sorted by key, and all of the key/value pairs with the same key K_{im} are sent to the same machine for reducing as a list. The reduce function is defined as taking the key K_{im} which is the column index and building a column vector E from the list of aggregated values $V = P$ such that $E_{1,P_1=i} = (P_2 = (D_i)_j)$. This means the column vector is constructed by taking the list of values and putting each element in the index specified from the value pair. Now all the column vectors are output where $K_{out} = j$ and $V_{out} = D_j$. This is seen in Figure 4.4.

The Reduce Phase takes all the results of the Map Phase, sorts them by K_{im}, and sends all the results to individual reducers where it is guaranteed that all

of the records with the same K_{im} will be sent to the same reducer. This occurs in all MapReduce algorithms. For the matrix transposition, every reducer will get all the information to rebuild a column.

$$\begin{array}{l} (K_{im} = 1, V_{im} = (1, [1])) \\ (K_{im} = 1, V_{im} = (2, [4])) \\ (K_{im} = 1, V_{im} = (3, [7])) \end{array} \quad = reduce \Rightarrow \quad \left(K_{out} = 1, V_{out} = \begin{bmatrix} 1 & 4 & 7 \end{bmatrix}\right)$$

$$\begin{array}{l} (K_{im} = 2, V_{im} = (1, [2])) \\ (K_{im} = 2, V_{im} = (2, [5])) \\ (K_{im} = 2, V_{im} = (3, [8])) \end{array} \quad = reduce \Rightarrow \quad \left(K_{out} = 2, V_{out} = \begin{bmatrix} 2 & 5 & 8 \end{bmatrix}\right)$$

$$\begin{array}{l} (K_{im} = 3, V_{im} = (1, [3])) \\ (K_{im} = 3, V_{im} = (2, [6])) \\ (K_{im} = 3, V_{im} = (3, [9])) \end{array} \quad = reduce \Rightarrow \quad \left(K_{out} = 3, V_{out} = \begin{bmatrix} 3 & 6 & 9 \end{bmatrix}\right)$$

Figure 4.4: Reduce Phase

Finally, a combine function will take all output key/value pairs and arrange them by their key K_{out}, the column index which joins the columns back into a matrix of the correct, dimension-swapped size and the transposed matrix D^T is completed. This is seen in Figure 4.5. All the rows are merged back into one contiguous, transposed matrix, D^T. The K_{out} is used to insure correct ordering of the rows V_{out}.

The combine phase is only required if the results must be aggregated back to a single computer. For small matrices, this is easy. However, the use of MapReduce is usually indicative of massive data sets in which it is not practical to actually recombine the data. Usually the data is left in a distributed state for further processing or use.

Another thing to note about the matrix inversion algorithm is its usefulness is limited as the actually processing in the map and reduce phases is incredible small. The overhead of distributing the work and recombining the results is not justifiable if it is greater than the actual processing time spent on the processors. A simple

$$\begin{matrix}\left(K_{out}=1, V_{out}=\begin{bmatrix}1 & 4 & 7\end{bmatrix}\right)\\ \left(K_{out}=2, V_{out}=\begin{bmatrix}2 & 5 & 8\end{bmatrix}\right)\\ \left(K_{out}=3, V_{out}=\begin{bmatrix}3 & 6 & 9\end{bmatrix}\right)\end{matrix} = combine \Rightarrow \begin{bmatrix}1 & 4 & 7\\ 2 & 5 & 8\\ 3 & 6 & 9\end{bmatrix} = D^T$$

Figure 4.5: Combine Phase

modification can be made to increase the processing time per processor and reduce the overhead. Consider a larger input matrix Z. Now, instead of simple breaking Z up into individual elements and inverting the coordinates of each element, Z will be broken up into square sub-matrices which will be individually inverted, and have those sub-matrices positions inverted. This works because matrix inversion can be done recursively.

For example, the larger matrix could be of size 9 by 9 instead of 3 by 3. The matrix is subdivided into a 3 by 3 matrix where each element contains a 3 by 3 matrix like in Figure 4.6.

One of the original tasks involved receiving the element at (1,2) which is the upper-middle element. In the new method, the task would receive the 3 by 3 sub-matrix in the upper-middle position of the divided matrix which is labeled B. The task would now invert B and pass on its inverted sub-matrix coordinates, (2,1), to be inverted in the larger matrix context, Z', just as before. The sub-matrix and matrix inversions as well as their recombination into Z^T are shown in Figure 4.7.

MapReduce's setup is very abstract which allows it to be flexible, but restrictive enough that it can scale well across a very large number of computers. It has been applied to many hundreds of problems where the data sets and computational requirements are far larger than a single computer is capable of processing.[7]

$$Z = \begin{bmatrix} 01 & 02 & 03 & 04 & 05 & 06 & 07 & 08 & 09 \\ 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 & 32 & 33 & 34 & 35 & 36 \\ 37 & 38 & 39 & 40 & 41 & 42 & 43 & 44 & 45 \\ 46 & 47 & 48 & 49 & 50 & 51 & 52 & 53 & 54 \\ 55 & 56 & 57 & 58 & 59 & 60 & 61 & 62 & 63 \\ 64 & 65 & 66 & 67 & 68 & 69 & 70 & 71 & 72 \\ 73 & 74 & 75 & 76 & 77 & 78 & 79 & 80 & 81 \end{bmatrix} \Rightarrow Z' = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix}$$

where

$$A = \begin{bmatrix} 01 & 02 & 03 \\ 10 & 11 & 12 \\ 19 & 20 & 21 \end{bmatrix} \quad B = \begin{bmatrix} 04 & 05 & 06 \\ 13 & 14 & 15 \\ 22 & 23 & 24 \end{bmatrix} \quad C = \begin{bmatrix} 07 & 08 & 09 \\ 10 & 11 & 12 \\ 19 & 20 & 21 \end{bmatrix}$$

$$D = \begin{bmatrix} 28 & 29 & 30 \\ 37 & 38 & 39 \\ 46 & 47 & 48 \end{bmatrix} \quad E = \begin{bmatrix} 31 & 32 & 33 \\ 40 & 41 & 42 \\ 49 & 50 & 51 \end{bmatrix} \quad F = \begin{bmatrix} 34 & 35 & 36 \\ 43 & 44 & 45 \\ 52 & 53 & 54 \end{bmatrix}$$

$$G = \begin{bmatrix} 55 & 56 & 57 \\ 64 & 65 & 66 \\ 73 & 74 & 75 \end{bmatrix} \quad H = \begin{bmatrix} 58 & 59 & 60 \\ 67 & 68 & 69 \\ 76 & 77 & 78 \end{bmatrix} \quad I = \begin{bmatrix} 61 & 62 & 63 \\ 70 & 71 & 72 \\ 79 & 80 & 81 \end{bmatrix}$$

Figure 4.6: Splitting the Matrix

4.3 Ensemble Classifiers

Ensemble classification is the process of combining multiple classifiers to increase the overall accuracy of the classifications. They are effective for many problems like named entity recognition in Linguistics and gene prediction in Bioinformatics, as well as many others. It is a simple idea which leads to many variations, some of which are described as follows.

$$A^T = \begin{bmatrix} 01 & 10 & 19 \\ 02 & 11 & 20 \\ 03 & 12 & 21 \end{bmatrix} \quad B^T = \begin{bmatrix} 04 & 13 & 22 \\ 05 & 14 & 23 \\ 06 & 15 & 24 \end{bmatrix} \quad C^T = \begin{bmatrix} 07 & 10 & 19 \\ 08 & 11 & 20 \\ 09 & 12 & 21 \end{bmatrix}$$

$$D^T = \begin{bmatrix} 28 & 37 & 46 \\ 29 & 38 & 47 \\ 30 & 39 & 48 \end{bmatrix} \quad E^T = \begin{bmatrix} 31 & 40 & 49 \\ 32 & 41 & 50 \\ 33 & 42 & 51 \end{bmatrix} \quad F^T = \begin{bmatrix} 34 & 43 & 52 \\ 35 & 44 & 53 \\ 36 & 45 & 54 \end{bmatrix}$$

$$G^T = \begin{bmatrix} 55 & 64 & 73 \\ 56 & 65 & 74 \\ 57 & 66 & 75 \end{bmatrix} \quad H^T = \begin{bmatrix} 58 & 67 & 76 \\ 59 & 68 & 77 \\ 60 & 69 & 78 \end{bmatrix} \quad I^T = \begin{bmatrix} 61 & 70 & 79 \\ 62 & 71 & 80 \\ 63 & 72 & 81 \end{bmatrix}$$

$$Z'^T = \begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix}$$

$$Z^T = \begin{bmatrix} A^T & D^T & G^T \\ B^T & E^T & H^T \\ C^T & F^T & I^T \end{bmatrix} = \begin{bmatrix} 01 & 10 & 19 & 28 & 37 & 46 & 55 & 64 & 73 \\ 02 & 11 & 20 & 29 & 38 & 47 & 56 & 65 & 74 \\ 03 & 12 & 21 & 30 & 39 & 48 & 57 & 66 & 75 \\ 04 & 13 & 22 & 31 & 40 & 49 & 58 & 67 & 76 \\ 05 & 14 & 23 & 32 & 41 & 50 & 59 & 68 & 77 \\ 06 & 15 & 24 & 33 & 42 & 51 & 60 & 69 & 78 \\ 07 & 16 & 25 & 34 & 43 & 52 & 61 & 70 & 79 \\ 08 & 17 & 26 & 35 & 44 & 53 & 62 & 71 & 80 \\ 09 & 18 & 27 & 36 & 45 & 54 & 63 & 72 & 81 \end{bmatrix}$$

Figure 4.7: Inverting the Matrices

4.4 Select Best Ensemble Classifier

A simple method to make an ensemble is called Cross-Validation Selection or Select Best. The data is partitioned randomly into two sets, one for training and one for validation. The classifier is trained on the training set, then rated based on the validation set. This is repeated multiple times. After some fixed number of repetitions, simply pick the classifier that scores the highest. Despite its simplicity, Select Best works well.[10]

4.5 Bagging Ensemble Classifier

Another method of ensemble creation called Bootstrap Aggregating, or Bagging, partitions the data into random subsets of the training data. The classifiers then vote with equal weight and the decision is the result with the highest plurality. It has been found that the more varied the classifiers, the higher the accuracy of the ensemble, so adding randomness can increase the ensemble's performance. However, classifiers trained on more data work better than a larger number of classifiers trained on less data with randomness added. Also, weighted voting can be implemented by evaluating the performance of each classifier and giving it weight equal to its performance.[11]

4.6 Boosting Ensemble Classifier

Boosting is an ensemble technique where incremental phases train classifiers emphasizing the training data that the classifier from the previous phase misclassified. Boosting can have the problem of over fitting where the classifier models begin to reflect the training data too much, making the classifier perform poorly on new problems. This problem can be mitigated by utilizing results from the previous classifiers trained on the less emphasized data.[14]

4.7 Stacking Ensemble Classifier

Stacking involves combining the results of the individual classifiers using a meta classifier. A simple plurality of classifiers is used as a baseline to train the meta classifier. It works best with strong, heterogeneous classifiers, and the data may or may not be partitioned. However, even using Stacking with sophisticated meta classifiers has been shown to perform only slightly better than the Select Best

method[10].

4.8 Gene Prediction

There are several types of gene predictors. A main classification of the types puts them into two groups: those based only on genetic sequences, and those that use external information about the genes to help find it. An *extrinsic* gene predictor describes a gene prediction program that relies on information from outside, such as information derived from studying protein structure or related organisms. The terms *ab initio*, or from the beginning, and *de novo*, meaning starting anew, refer to the former of the types where gene prediction is based solely on the input sequence.

Generative methods, such as Hidden Markov Models (HMM), have been successfully used for gene prediction since the 1970s when they were used for speech recognition and later for object identification within images. More recently they have been used for gene prediction. Generative models learn the joint probability distributions between the predicted variables and the observed variables and use those distributions to find the most likely prediction. In sequence prediction, there are many observation and prediction variables, and the number of joint distribution parameters needed to be learned can quickly become intractable. In order to get around this limitation, independence assumptions need to be made which reduce the amount of joint distributions needed.[13]

A first order Hidden Markov Model uses the assumption that each observed variable is independent of all variables except one prediction variable and that each prediction variable is independent of all other variables except for the prediction variable preceding it in the sequence. These types of independence assumptions can create models that are an inaccurate representation of the underlying forces that reg-

ulate gene production. However, the number of dependencies allowed exponentially increases the number of probability parameters that need to be learned. This is the motivation for using discriminative learning for sequence prediction[13, 8]. A discriminative learning method would directly learn the conditional probabilities, which is information that is actually needed.

4.9 Introduction to CRAIG

CRAIG, the Conditional Random field Ab Initio Gene predictor, is an algorithm with substantially higher prediction capabilities than most of its ab initio, primarily HMM-based competition. It benefits from Conditional Random Fields or CRFs. These allow more dynamic probabilistic models and global optimization of predictive features using discriminative learning models. This makes CRAIG capable of better balancing gene signals than its HMM brethren.[4]

4.10 The CRAIG Algorithm

DNA sequences are represented in CRAIG as a string of many As, Cs, Gs, and Ts which represent any of the nucleotides adenine, cytosine, guanine, and thymine respectively. A segmentation of any given sequence is just a labeling of each nucleotide as either an intergenic region, an exon, or an intron. These are denoted with Ns, Es, and Is. Segmentations are then just a string of these three characters, one for each nucleotide in the DNA sequence, and are effectively a marking of where the genes are in a sequence. Segmentations can also be represented as a list of elements containing a label, start position, and length.

Segments are consecutive portions of segmentations that share the same label. This allows for a compact form of segmentation storage because all that is needed

to represent a segmentation is a list of segments. These segments only need to keep track of the label of the segment, the start position relative to the DNA sequence, and the length of the segment. A useful attribute of segment classification is that whole segments can be classified at a time in contrast to how Hidden Markov Models classify only typically between 1 and 5 nucleotides at any given step [4].

The ability to pull out relevant classification data is necessary for both efficiency and accuracy. Feature vectors are vectors of attributes about a given segment from a sequence where each feature can vary widely. A feature can be something as simple as whether the segment starts with a codon representing methionine; a useful test because all genes start with methionine. They can also include things like the length of the segment; the frequency of the various nucleotides, as in the number of Ts; or the frequency of sub-sequences of nucleotides like the number of CGs or $CGCG$s.

Now that criteria has been laid out to describe the various features of a segment, these features must be weighted so that the more relevant descriptors of genes count for more than the less relevant features. For instance, the dinucleotide frequency for AA may not be that informative because AA may have a similar distribution compared with other dinucleotides. Therefore, it should not be counted highly as indicating a gene. However, CGs are known to be rare except right before genes in *Homo sapiens*, making it particularly indicative of an intergenic region.

Thus a weighting vector is used to describe how important each feature is. It is a vector of numbers equal in length to the feature vector where each index of the weight vector tells the relevance of the same indexed feature vector. This will allow easy overall scoring by simply taking the inner-product of the weight vector and feature vector to give a final score of a candidate segment.

The dynamic programming portion of the algorithm is what drives the us-

age of the segmentations, feature vectors, and weight vector. It uses a table with one dimension representing the number of states a segment can be labeled as (e.g. intergenic, intron, or exon), and the other dimension represents the sequence being searched where each cell of the table is one nucleotide. A Dynamic Programming table to label the (very short) sequence $CATG$ can be seen in Figure 4.8.

	-	C	A	T	G
N	0	0.5	0.6	0.2	0.4
E	0	0.1	1.1	1.4	1.7
I	0	0	0.1	0.3	0.8

Figure 4.8: CRAIG's use of Dynamic Programming

The recurrence relation for this dynamic programming problem fills the table by picking indexes over which to cut up the sequence. It iterates over the sequence by picking an end point to cut, and a candidate state which will be the label for that segment, then iterating over all possible start points and previous labels. Next, it takes the segment marked with the given start and end positions as well as the labels, and extracts the feature vector representing the various features of that segment. It then scores this segment by taking the inner product with the weight vector as described earlier.

Figure 4.9 shows the recurrence relation for the dynamic programming portion of the algorithm. M is the score table, i is the end index, l is the start index, y is the current state, y' is the previous state, w is the weight vector, f is the feature vector, and x is the sequence. [4]

After each candidate start position and label has been chosen, the highest score from all the candidates is taken and filled into the table with indexes at the end position and current label. Given that there are many possible start positions and

$$M(i,y) = \begin{cases} \max_{y',1\leq l\leq \min\{i,B\}} M(i-l,y') + \\ \quad \boldsymbol{w}\cdot\boldsymbol{f}(\langle i-l,l,y\rangle,y',\boldsymbol{x}) & \text{if } i>0 \\ 0 & \text{if } i=0 \\ -\infty & \text{otherwise} \end{cases}$$

Figure 4.9: Dynamic Programming Recurrence Relation for CRAIG

previous labels, it is best to keep back pointers to avoid many recomputations when back-tracking.

Now to pull out the best segmentation, start by taking the greatest score from the last column of the table, which represents the score of the whole sequence. Next, follow the back-pointers. Every time a back-pointer is followed, prepend the state and position to our final segmentation. Finding the length is found by taking the difference between successive states.

During training, a labeling is found for a given training sequence. Then a function is used to compare the predicted gene-segmentation to the actual gene-segmentation for that sequence, and return a score for how well the gene was labeled. A minimization algorithm is used to find the smallest change to the weight vector such that it scores this example correctly and hopefully does not break any existing training instance outcomes. The training instances are all fed through the system in this manner, and the whole operation is repeated until all the training instances are scored correctly, or until some threshold is reached.

After the many iterations of training all the instances, the weight vector is now precisely tuned given the assumption that the data is linearly separable. Now, running the dynamic programming portion of the algorithm with the tuned weight vector should give the best results possible for the chosen feature vectors and input training data.

The pseudo-code for the CRAIG algorithm is shown in Figure 4.10.

```
w=0  (initial discriminant line a.k.a. parameters we are training)
for i=1 to Tth
     x=i sequence in training set
     find most likely annotation of x using Dynamic Programming
     calculate diff between prediction and actual annotation
     adjust w to reflect difference with minimal change to w
```

Figure 4.10: CRAIG Algorithm

4.11 Running CRAIG

While CRAIG is a high performing gene predictor both in terms of accuracy and speed, training a model can takes a serious amount of time. An upper bound for the complexity of CRAIG is $O(T*L*A^2*F*S)$ where T is the number of iterations, L is the number of possible labels, A is the average length of a sequence, F is the number of features to compute, and S is the number of sequences. This very high order is due to CRAIG using Conditional Random Fields which makes the Dynamic Programming portion of CRAIG significantly slower. This is because CRAIG attempts to find the optimal starting position for a given label and the current stop position, so the sequence length is squared instead of just linear. The time complexity of the number of iterations is very imprecise as well. The author of CRAIG recommends running it until it appears to have converged based on monitoring the verbose output. This made automating the process and comparing results significantly more challenging.

The data set provided with CRAIG consists of over 3,000 genes. Training a model using the whole set takes a minimum of several days on modern hardware. CRAIG is heavily bounded by CPU speed as shown by consistent 100% CPU usage. Using only 120 randomly selected genes, CRAIG completed training in roughly 1.5

hours, and 375 genes took roughly 6.5 hours.

Not only does CRAIG scale poorly because of its time complexity, it has limited opportunity for parallelization. This is even in spite of the large number of variables affecting its runtime. Because CRAIG's very high order of complexity is based mainly on the number of training sequences, reducing those does has a large effect on time though. The only other controllable parameter is the number of iterations that CRAIG runs. However, since one iteration is sequentially tied to the previous, parallelization is not possible without modifying the source code.

4.12 NBLAST

Another important tool for biologists is BLAST, which stands for Basic Local Alignment Search Tool. It is an algorithm in Bioinformatics which performs sequence alignments of DNA to compare different species. Even though species diverged in their evolutionary path, many large portions of their DNA are conserved. For example, Hemoglobin is a protein found in blood that allows blood cells to bind to oxygen to transport it throughout the body. As the majority of animals thrive off of oxygen, this is an extremely important and well-conserved part of DNA. BLAST searches for these types of conservations and performs an alignment, allowing biologists to see how species have evolved comparatively, providing important insights into species evolution and ancestry.

An important feature of BLAST is that it emphasizes speed over accuracy. This is necessary because of the massive amount of data typically being considered. BLAST can achieve a 50 times speedup over standard dynamic programming approaches, but does have some accuracy penalties as a result[9]. The dynamic programming algorithm guarantees the highest scoring alignment, but does spend large

amounts of time on scoring alignments that are unlikely. BLAST takes advantage by pruning these unlikely alignments, which is an acceptable trade-off in most scenarios.

Even using BLAST instead of the dynamic programming algorithm, performing alignment on sequences of DNA and proteins can be a computationally intensive task. Furthermore, the number of sequences continues growing, so distributing the computation quickly becomes necessary. Therefore, the NBLAST algorithm was developed to calculate all N^2 combinations of N sequences in only $N^2/2$ time. It does this by only completing the upper triangle of the N by N matrix. It also distributes this work across many computers in potentially massive clusters to create a large single database of alignments.[9]

An online tool is shown in 4.11 which is used to show the sequence alignments from Neanderthal mitochondria. Many of the sequences have large amounts of conserved base-pairs, indicating a strong evolutionary relationship.

Sequence alignment relates in many ways to gene prediction. The amount of data to analyze can quickly overwhelm single computers. Both of the algorithms must deal with the mystifying nature of biological systems, making fuzzy calculations to approximate the systems. And finally, because of the fuzzy nature of the problems, there are many trade-offs between performance and speed, allowing a broad range of available solutions.

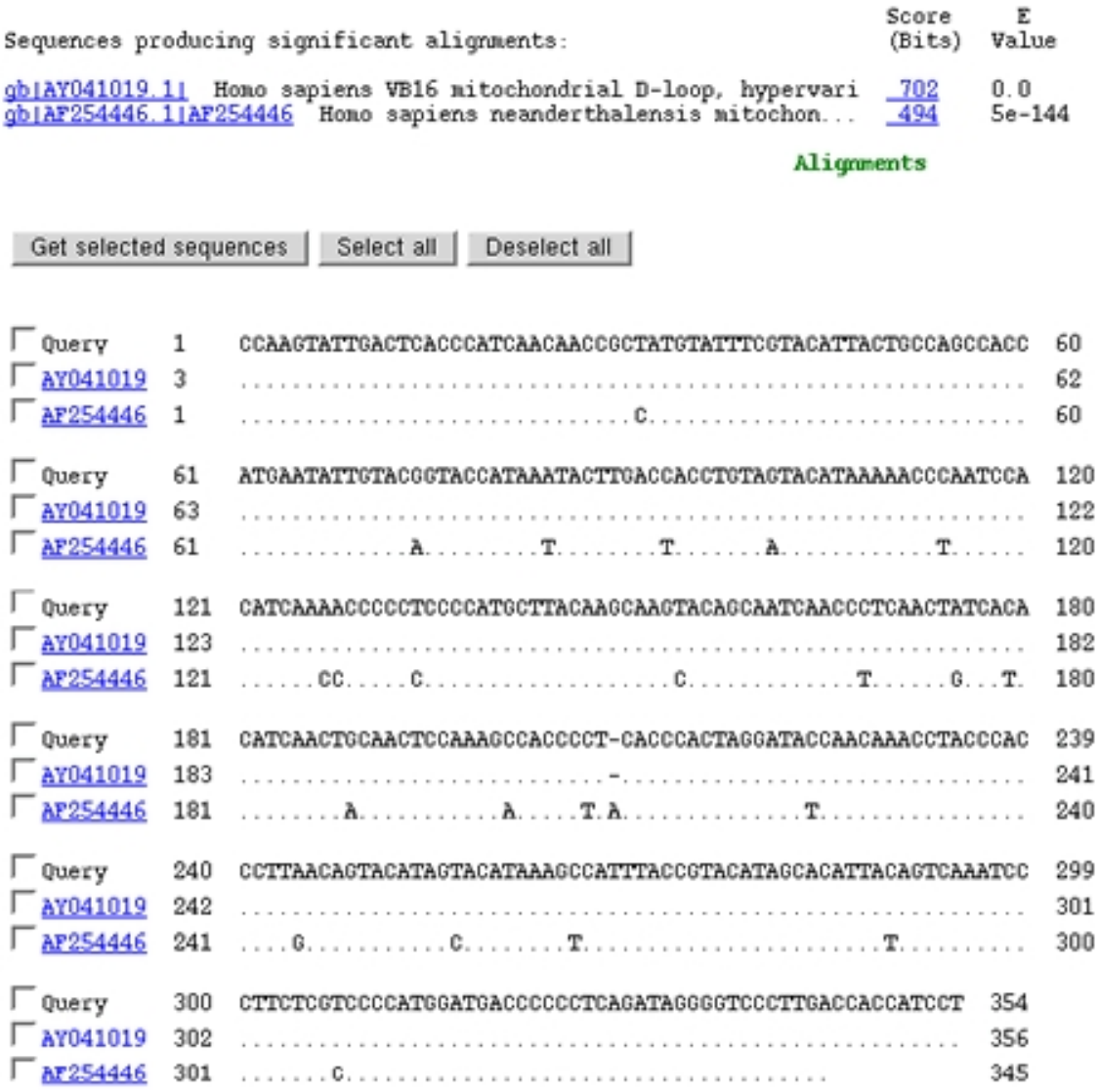


Figure 4.11: Sequence Alignment Tool

CHAPTER 5

ARCHITECTURE

A general framework for parallelizing classification problems will be described with a side-by-side overview of MrCRAIG's algorithm in light of the general framework. MrCRAIG follows the general layout of any MapReduce algorithm in terms of several phases. These phases include a setup phase where the data is partitioned, a first map phase that runs CRAIG's training program, a second map phase that evaluates the predictors and assigns them scores, and finally a reduce phase that accounts for the votes of all the predictors or selects the best one. They are further described in the following sections.

5.1 Data Setup

A main assumption of the general framework is that the majority of the time spent by the predictor is iterating over training and validation data. Therefore, cutting the data into fractional parts should radically reduce the time it takes to train the predictor. This has a high probability of decreasing the accuracy of the classifiers[5], but that should be made up for by the combination of the classifiers in the ensemble.

The training data is broken down into chunks that should be equal in number to the number of processors, denoted N. Any more chunks than N and there will be

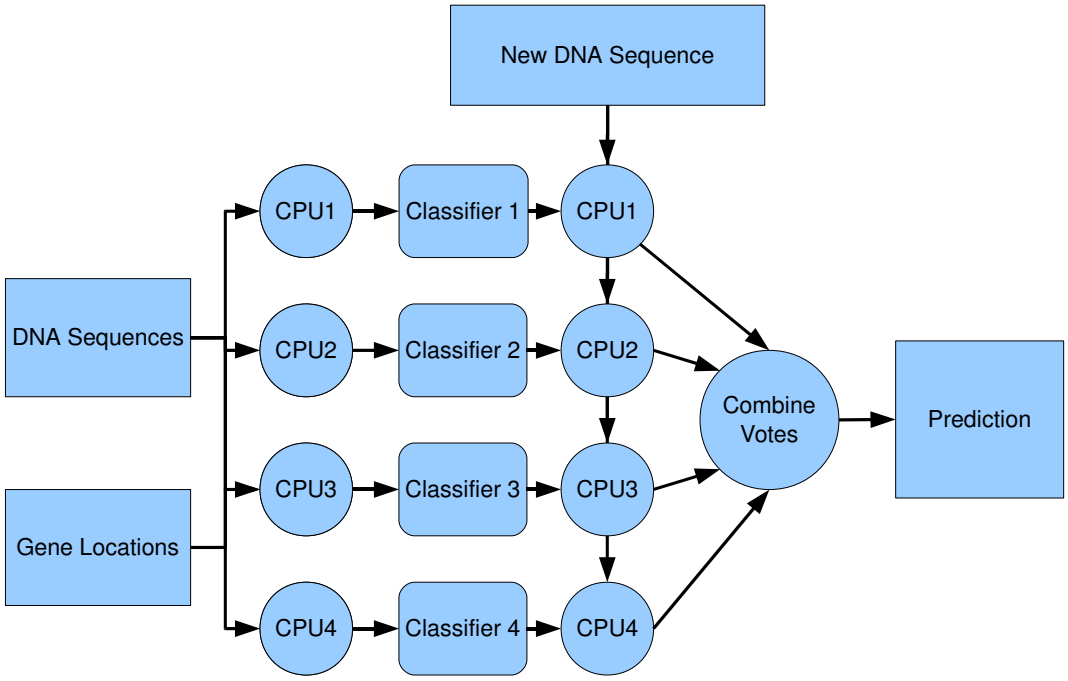


Figure 5.1: MrCRAIG Overview

at least double the amount of time spent in the otherwise fully parallel training phase. This is because at least one processor has to train on multiple chunks due to the Pigeon Hole Principle. Another downside is that the classifiers would be weakened with no gain in parallelization. Using fewer data chunks than processors is also undesirable because some processors would be left idle during the training phase.

In the gene prediction problem, the training input is both the sets of DNA sequences and the locations of the genes in those sequences. These come in the form of two input files, one in FASTA format, and the other in GTF format. The FASTA file consists of DNA region names and the actual sequences. The GTF file contains the actual gene locations matched by DNA region name. MrCRAIG splits these files into individual file pairs representing a single gene and region as in Figure 5.2. They are then distributed amongst N groups where N is the number of classifiers in the

ensemble.

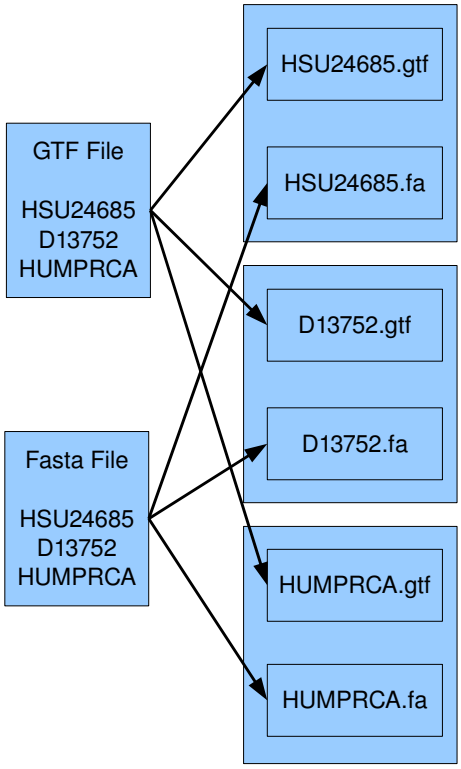


Figure 5.2: Data Setup

5.2 Building the Ensemble: The Training Map Phase

As mentioned previously, there are many methods for making an ensemble. The only exclusions from all the various methods are the ones that cannot be parallelized. Boosting, for example, cannot be parallelized in a course-grained manner because each iteration depends on the previous iteration's output[3]. However the Bagging method trains on independent subsets of the data and then votes, making it easily fit the MapReduce framework. Stacking works as long as the meta-classifier has a reasonable performance. Select Best is also easily suitable because of its training independence.

With MrCRAIG, when the Bagging method is used, multiple CRAIG instances are trained. They use a discrete subset of the data split evenly among the CPUs, and the data is reused as validation data on separate processors as in Figure 5.3. The data is never used twice by the same CPU. This is run in the training map phase. CRAIG also requires an ad hoc file format and some additional configuration files which are generated. After all the preparation work, the predictors are trained which generates many CRAIG models in the form of params files.

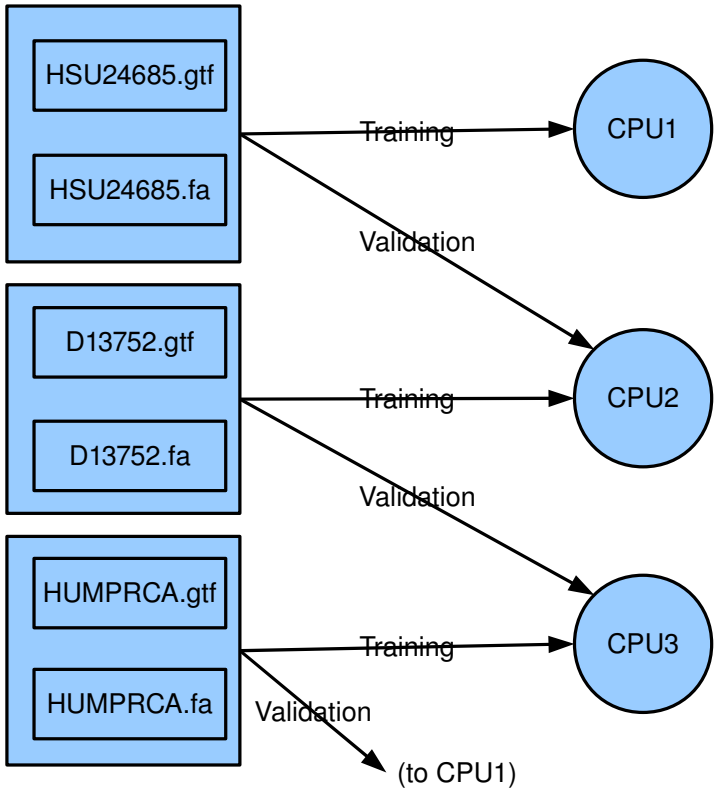


Figure 5.3: Building the Ensemble

After the predictors have been trained, a program called Eval[12] is run that provides detailed statistics on the performance of the classifier. This will allow for

both the weighted voting in the Bagging ensemble and to judge which classifier is used in the Select Best ensemble. Eval provides a plethora of useful statistics.

5.3 Generating Predictions: The Prediction Map Phase

Now that many predictors have been trained and scored, MrCRAIG can move on to predicting new genes. At this point, the Select Best method can select the highest performing predictor, and any predictions will be derived solely from that predictor. MrCRAIG will also generate predictions for the rest of the predictors and combine for use in the Voting ensemble. New test data is distributed to the classifiers on each processor and run. Each generates classifications for the various input sequences and outputs prediction and the sequence from which it came. This will allow all predictions for same input sequence to be reduced on the same processor in the coming reduce phase.

MrCRAIG generates predictions by taking the testing DNA sequences in FASTA file format, and producing a GTF file containing gene information. Each line in the GTF file represents either a gene start sequence, an exon sequence, or a gene stop sequence as in Figure 5.4. There are typically many exon sequences within the range of the start and stop signals with gaps in between exons indicating non-coding regions, or introns. The GTF format specifies many additional fields besides the start and stop locations. One such field represents the evaluation score that the predictor received.

As CRAIG outputs the lines from the GTF file, MrCRAIG's Prediction Map Phase uses the DNA sequence identifier as the map output key, and the rest of the line as the map value. Since there are many classifiers predicting over the same regions, there will be many overlapping predictions. There is a scoring field in the GTF format

```
Region Predictor     Type    Start Stop Score Dir Offset     (other info)
  |       |           |       \     \     |    /    /            |
HUMPCI  MrCRAIG1  start_codon  100  102  10  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";
HUMPCI  MrCRAIG1  CDS          100  200  10  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";
HUMPCI  MrCRAIG1  stop_codon   201  203  10  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";


HUMPCI  MrCRAIG2  start_codon  150  152  20  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";
HUMPCI  MrCRAIG2  CDS          150  300  20  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";
HUMPCI  MrCRAIG2  stop_codon   301  303  20  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";


HUMPCI  MrCRAIG3  start_codon  250  252  30  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";
HUMPCI  MrCRAIG3  CDS          250  350  30  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";
HUMPCI  MrCRAIG3  stop_codon   351  353  30  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";
```

Figure 5.4: GTF File Showing Three Predictions

which is not utilized by CRAIG. This can be seen in Figure 5.4 as indicated by a "." in the 6^{th} field. The score field is filled in with the gene predictor's evaluation score and will be used when combining predictions.

5.4 Combining Votes: The Prediction Reduce Phase

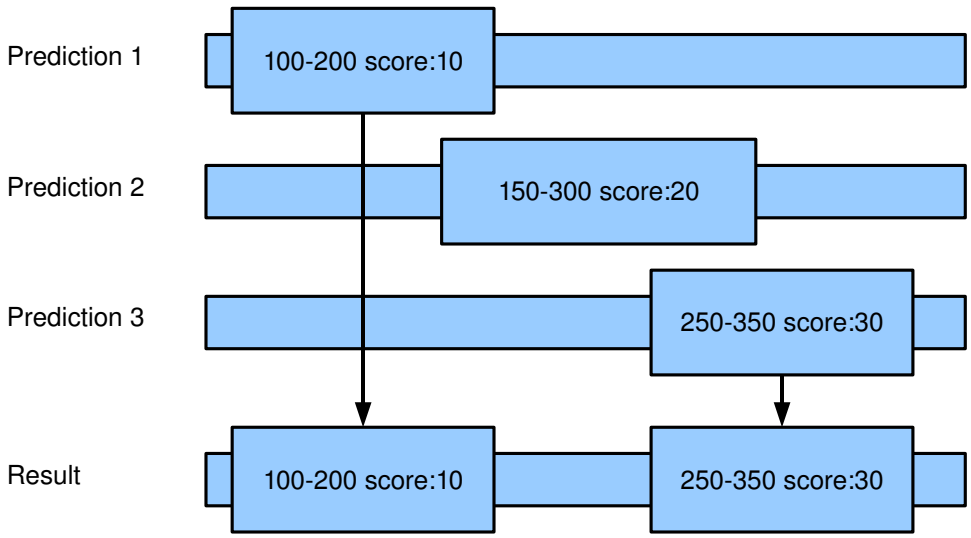
The MapReduce framework receives the output key-value pairs from the Prediction Map Phase, partitions the results into groups by the key, and distributes the groups to reducers. Since the sequence identifier is used as the key, all predictions from the same sequence are sent to the same reduce task. Now each reducer can combine the various predictions and associated evaluation scores into a single unified prediction for each sequence. Overlapping sequences with low scores are removed using a dynamic programming algorithm.

As plurality voting has been shown to perform poorly, MrCRAIG uses a unique algorithm to combine votes. This requires a scalar score which necessitates the combination of the various statistics that the Eval program returns. There are several

hundred values describing the sensitivities and specificities of the genes, as well as sub-regions like exons and base pairs. Identifying bases and exons as coding is useful as a stepping stone to finding whole genes, but finding the genes is of much greater relevance to Biologists and provides greater challenges to classification algorithms. Therefore, the overall score of a classifier used in MrCRAIG is the sum of the squares of the sensitivity and specificity of gene prediction, $Score = specificity^2 + sensitivity^2$. This gives the highest scores to predictors that achieve both high sensitivity and specificity, as both are required for good predictions.

Each reduce task will receive multiple, potentially overlapping predictions for a given input sequence as in Figure 5.4. MrCRAIG combines predictions by removing low-scoring regions that overlap with other higher-scoring ones as in Figure 5.5. This maximizes the overall prediction score and eliminates gene overlaps. Note that the 2^{nd} prediction from the Map Prediction Phase overlaps the 1^{st} and 3^{rd} predictions. There are two options to resolve the overlap: removing the 2^{nd} prediction, or both the 1^{st} and 3^{rd}. MrCRAIG removes the 2^{nd} because it results in a higher overall score including the scores 10 and 30 instead of just 20.

The Dynamic Programming algorithm that removes overlaps works by maximizing a global score. At each step, it attempts to add a prediction to the result, and chooses to either add that prediction and update a score array, S, or keep the existing score as in $S_i = max\left(S_i, S_{P_{start}} + P_{score}\right)$. The score array is equal in length to the total sequence length. Each value in the score array indicates the maximum score achievable by including only predictions with end indexes, i, less than the current index. So the value at the i^{th} index in the score array will only include predictions, P, that stop before i in the sequence. Backtracking is used to recover the predictions that were used to achieve the maximum score.



Result in GTF format:

```
HUMPCI  MrCRAIG1  start_codon  100  102  10  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";
HUMPCI  MrCRAIG1  CDS          100  200  10  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";
HUMPCI  MrCRAIG1  stop_codon   201  203  10  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";

HUMPCI  MrCRAIG3  start_codon  250  252  30  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";
HUMPCI  MrCRAIG3  CDS          250  350  30  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";
HUMPCI  MrCRAIG3  stop_codon   351  353  30  +  0  gene_id "HUMPCI"; transcript_id "HUMPCI.0";
```

Figure 5.5: GTF Result

5.5 Hardware Setup

A cluster of computers was assembled to run MrCRAIG. The cluster was configured with one master for task and distributed file system management running on a AMD Athlon 3500+ CPU at 2.4 GHz with 1 gigabyte of RAM. Four computers are used as slave nodes to both run the tasks and store the distributed file system data, which is the standard configuration. Three of the slaves are laptops with Intel Core2 Duo T9300 CPU running at 2.5 GHz with 3 gigabytes of RAM each. The fourth slave has a AMD Athlon X2 4600+ CPU running at 2.4 GHz with 3.5 gigabytes of RAM. All

computers are interconnected with a common 100 MB/s Ethernet connection using a fast rack-mount switch. As CRAIG requires significant CPU and RAM resources, many additional computers shown in the cluster in Figure 5.6 were not capable of completing any of the MrCRAIG tasks in any reasonable amount of time, or would simply crash due to overheating. The result is a 5 computer cluster with 1 master and 4 slaves shown in Figure 5.7.

Figure 5.6: MapReduce Cluster Photo

5.6 Software Setup

The cluster runs MapReduce programs using the Hadoop MapReduce implementation. This requires greater-than-usual SSH configuration. The setup of the distributed file system and Hadoop site configuration were very standard. All the computers in the cluster run Ubuntu Linux at version 8.04 or higher. Sun's version of Java 6 was used as recommended. CRAIG was compiled and installed individually

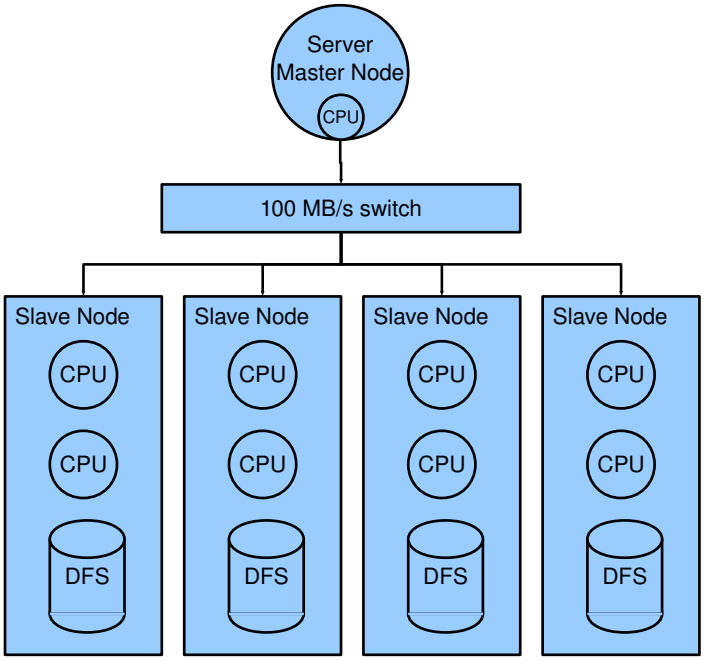


Figure 5.7: MapReduce Cluster Diagram

on each computer using GCC 4. Eval 2.2.8 is used both by MrCRAIG for scoring predictors and at the end to analyze predictions.

CHAPTER 6

EXPERIMENTS

6.1 Tests

MrCRAIG is tested changing several factors to analyze performance. The tests vary the number of processors available, N, as well as the ensemble method used, Select Best or Voting. The sensitivity and specificity of the gene predictions are measured.

The N predictors in MrCRAIG are each trained using 375 randomly selected genes from the CRAIG training set and scored based on the CRAIG development set containing 65 genes. The sizes were chosen as any smaller and the randomness of the genes would cause large variances in predictive results from predictors despite being trained on the same number of genes. Choosing the smallest reasonable size was necessary as more training genes increased the likelihood of task failure which already takes significant time: 6 hours and 28 minutes +/- 12 minutes per predictor on the AMD Athlon slave.

6.2 Results

After MrCRAIG is run, statistics for the Select Best method and Voting method were compiled. For comparison, the other classifiers from the Voting method were used as individual classifiers, not within an ensemble. Three major tests were

run where N is 4, 6, and 8, shown in Figures 6.1, 6.3, and 6.5 respectively. Also, the average, minimum, and maximum sensitivities and specificities are provided for each test in Figures 6.2, 6.4, and 6.6. It is important to note that the bar charts with aggregate statistics do not reflect actual predictors. For instance, the Best Case in 6.2 shows the highest sensitivity, and highest specificity, but they were not derived from the same predictor.

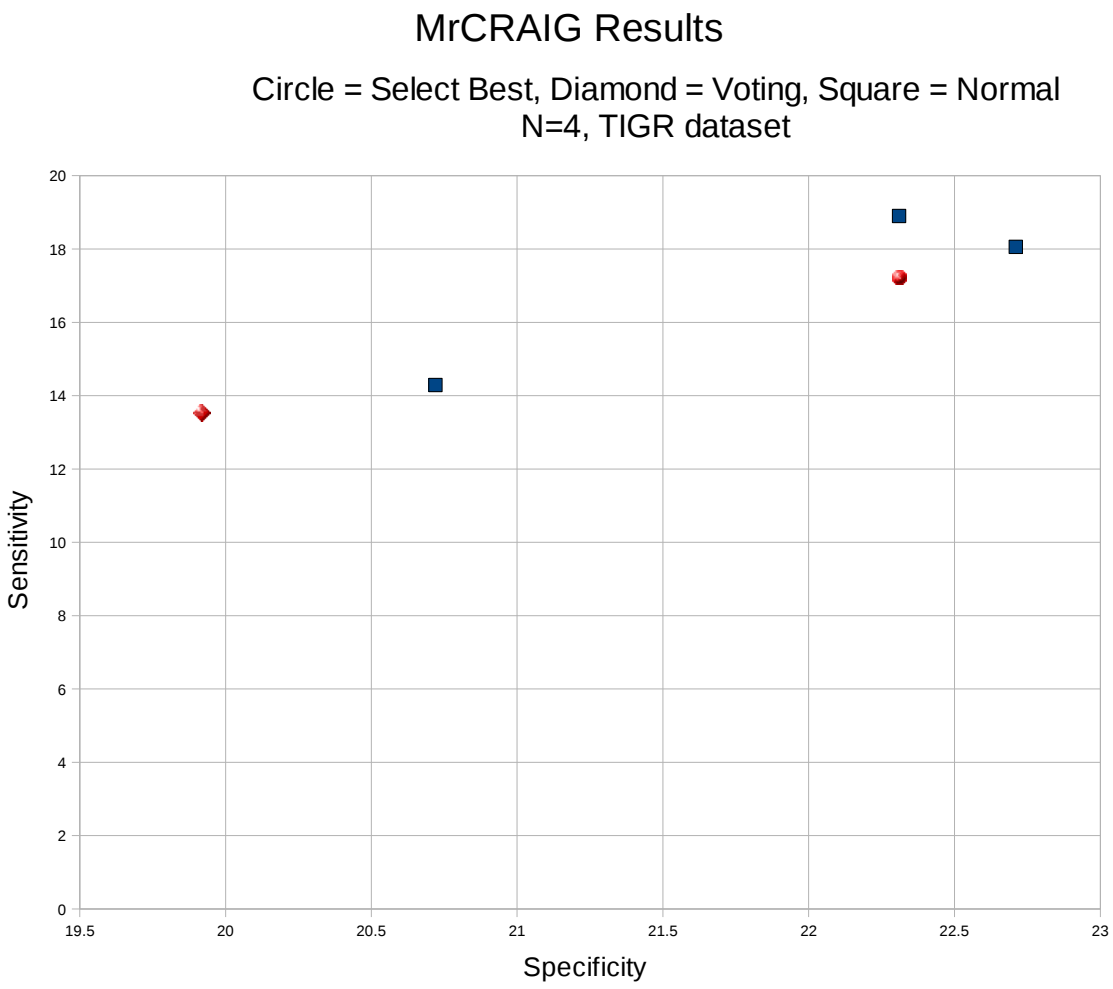


Figure 6.1: Results Graph for TIGR using N=4

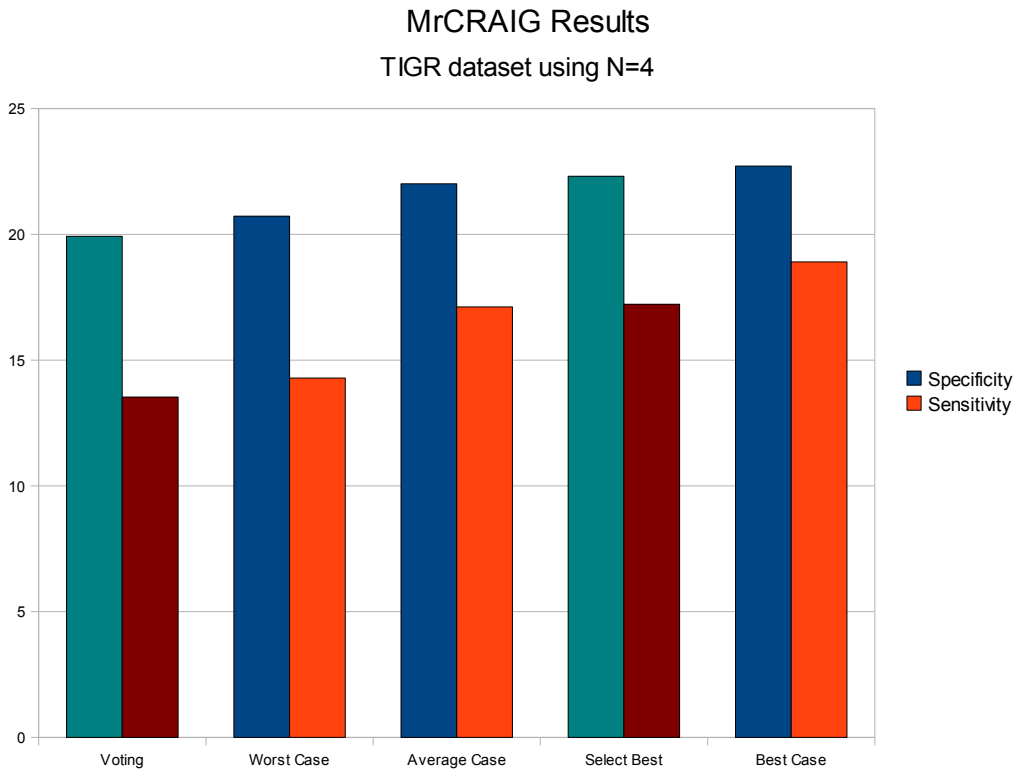


Figure 6.2: Bar Graph for TIGR using N=4

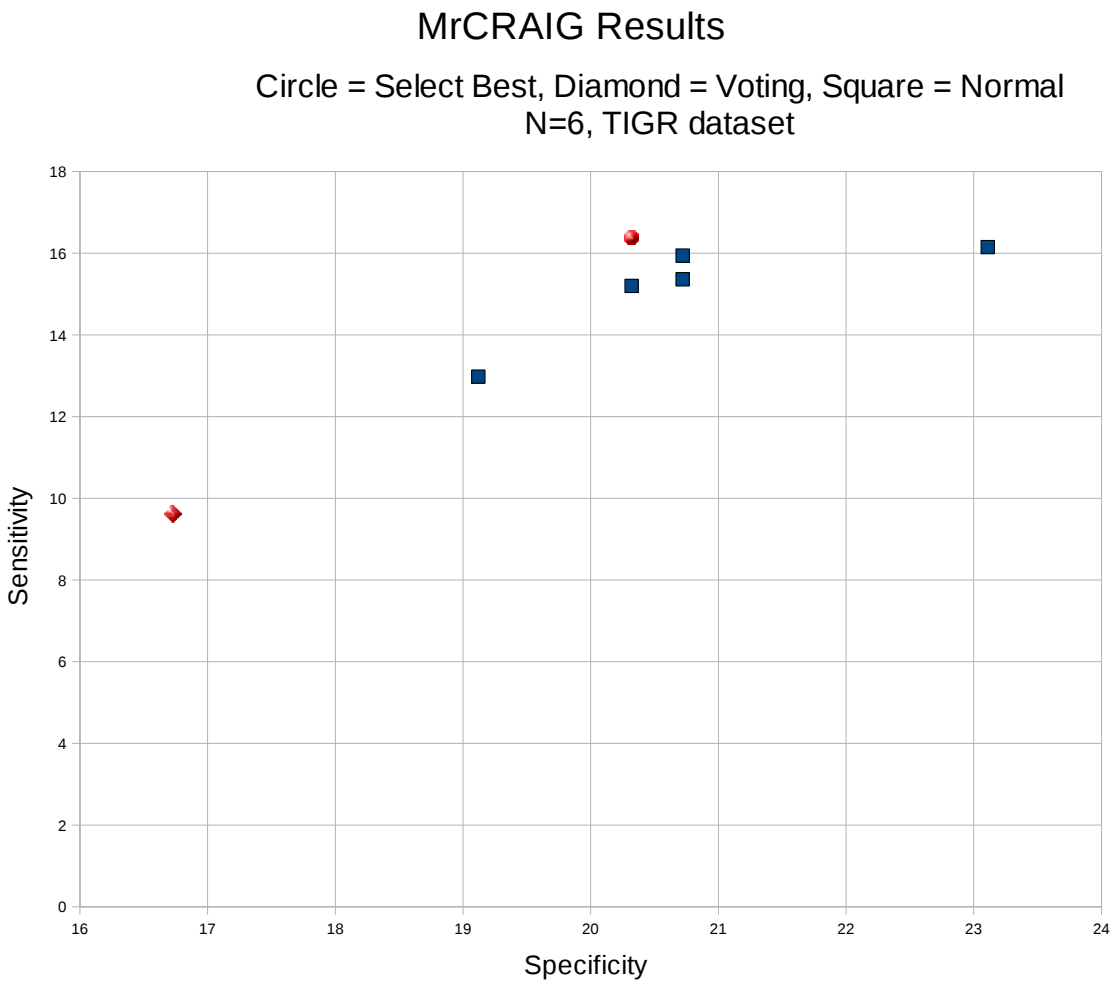


Figure 6.3: Results Graph for TIGR using N=6

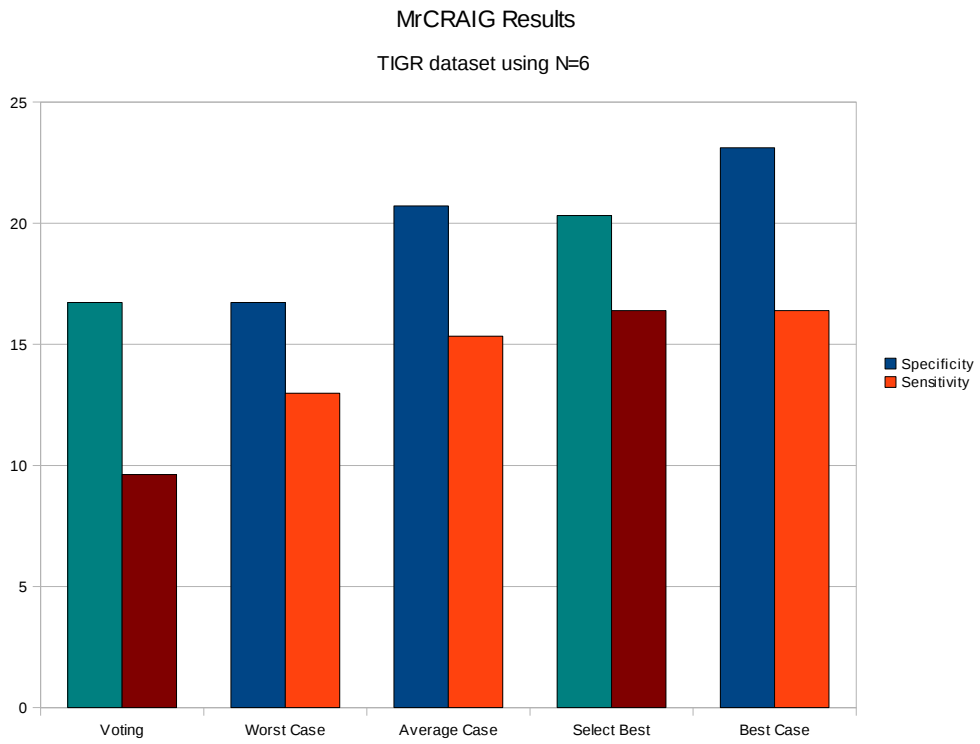


Figure 6.4: Bar Graph for TIGR using N=6

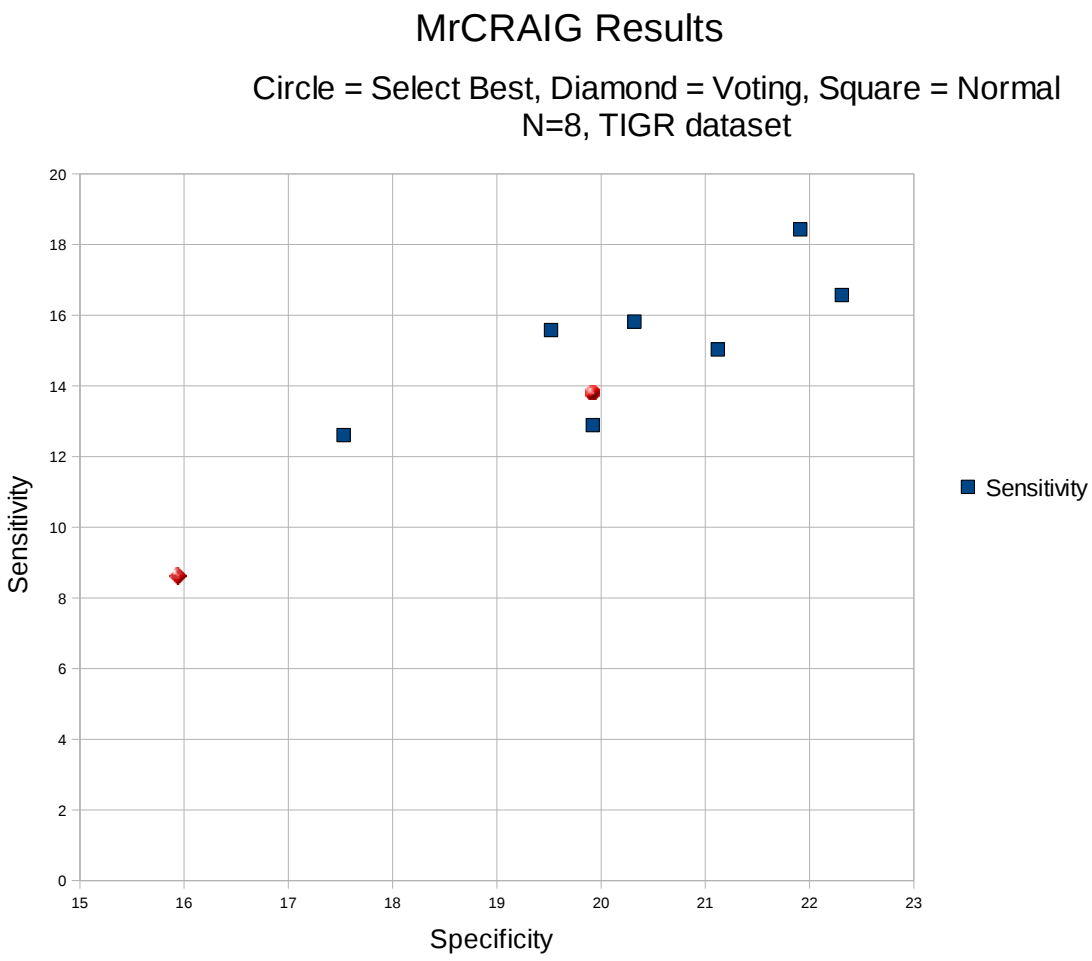


Figure 6.5: Results Graph for TIGR using N=8

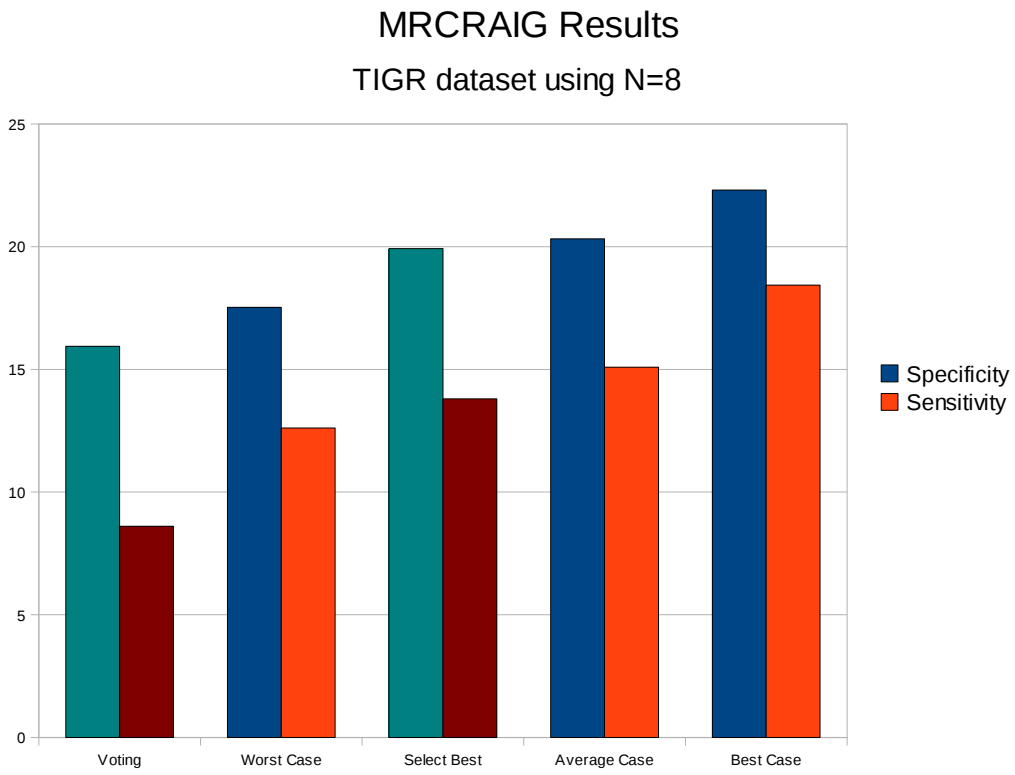


Figure 6.6: Bar Graph for TIGR using N=8

6.3 Analysis

While the results are not overwhelming using the ensemble predictors, there are some promising things to note. The Select Best method tended to stay within the top echelons of the predictors, and for $N = 6$ picked the predictor with the highest sensitivity of 16.39% and a very competitive specificity of 20.32%. Select Best also stayed notable above average for $N = 4$ and $N = 6$ indicating that its use pull up minimum prediction accuracy would be a good use.

The novel Voting technique did not fair as well. Despite the large modification to the simple voting, the novel Voting algorithm achieved very poor sensitivities and specificities. Looking into the evaluation data, there were 251 actual genes in the TIGR data set, but the voting algorithm predicted 377, 425, and 453 genes when N is 4, 6, and 8 respectively. This is likely due to a number of poor predictors that predict many genes in unlikely places. Since these unlikely "genes" are not overlapped by any other predictions, they remain, resulting in a large number of additional genes found. This significantly impacted the sensitivity. This also affected the specificity as well, where multiple poor predictions could achieve larger combined scores than single high scoring predictions and overriding them.

CHAPTER 7

FUTURE WORK

There are many opportunities for future work in the MrCRAIG project. For instance, experimenting with different scoring rules could improve voting. Utilizing different gene predictors instead of just CRAIG also has interesting possibilities. Also, analyzing more time versus accuracy trade-offs and other gene prediction variables would provide valuable insight to the Bioinformatics community. Finally, studying the distributed ensemble classification techniques employed by MrCRAIG in other contexts such as Natural Language Processing could be a useful endeavor.

Voting improvement has many potential directions. Changing voting so that it emphasizes fewer good predictions as opposed to many poor predictions would be one clear direction. Not incorporating predictors that score below some threshold could easily eliminate many bad predictions. Giving predictions that align closely a higher score may also increase performance, so they together could out-weigh outliers with overlapping predictions.

Incorporating different types of predictors like Augustus, Genezilla, and Genscan++ could provide an excellent direction for further work if incorporated with the stacking ensemble method as shown by Dzeroski et al[10]. This is especially true since gene predictors are typically biased towards certain types of problems. For instance, CRAIG is particularly good at identifying excessively long introns. Other predictors

would have other unique strengths that might be used to enhance the predictions for the whole system as it would benefit from better diversity than the Bagging method alone can provide.

As earlier stated, the goal of MrCRAIG was not only to find performance improvements in gene prediction, but also to find uses for the technique in general problems. There are many fields in need of performance boosts and parallelization, especially when faced with declining CPU frequencies. Pursuing other applications of the MrCRAIG data classification technique could reap many interesting results. Prediction problems that are troubled by low numbers of positives could clearly gain from MrCRAIG's tendencies.

CHAPTER 8

CONCLUSION

The future of parallel and distributed computing is imminent. Finding ways to utilize this breadth of hardware will prove a challenging task for many generations of programmers to come. The relatively new field of Bioinformatics also has many challenges to overcome. Gene prediction is still under research, and utilizing this newly gathered gene data will provide many new problems. Looking to the past for parallelization techniques as well as designing new methods will hopefully make the coming paradigm shift in software development more tolerable for the industry, and ease the difficult task of gene prediction.

BIBLIOGRAPHY

[1] Armstrong, J. Making reliable distributed systems in the presence of software errors. *ACM Queue September* (2008).

[2] Arpaci-dusseau, R. H., Anderson, E., Treuhaft, N., Culler, D. E., Hellerstein, J. M., Patterson, D., and Yelick, K. Cluster i/o with river: making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems* (1999), ACM Press, pp. 10–22.

[3] Bauer, E., and Kohavi, R. An empirical comparison of voting classification, algorithms: Bagging, boosting, and variants. *Machine Learning 36* (1999), 105–139.

[4] Bernal, A., Crammer, K., Hatzigeorgiou, A., and Pereira, F. Global discriminative learning for higher accuracy computational gene prediction. *PLoS Computational Biology 3* (2007), 488–497.

[5] Chan, P., and Stolfo, S. A comparative evaluation of voting and meta-learning on partitioned data. *Machine Learning-International Workshop* (1995).

[6] Collins, F. Medical and societal consequences of the human genome project. *The New England Journal of Medicine 341* (1999), 28–37.

[7] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *Operating Systems Design and Implementation 2004* (2004), 137–150.

[8] Decaprio, D., Vinson, J. P., Pearson, M. D., Montgomery, P., Doherty, M., and Galagan, J. E. Conrad: Gene prediction using conditional random fields. *Genome Research 17* (2007), 1389–1398.

[9] Dumontier, M., and Hogue, C. Nblast: a cluster variant of blast for nxn comparisons. *BMC Bioinformatics 3*, 1 (2002), 13.

[10] Dzeroski, S., and Zenko, B. . Is combining classifiers better than selecting the best one? In *Machine Learning* (2004), pp. 255–273.

[11] Hu, Z.-H., Li, Y.-G., Cai, Y.-Z., and Xu, X.-M. An empirical comparison of ensemble classification algorithms with support vector machines. *Machine Learning and Cybernetics 6* (2004), 3520–3523.

[12] Keibler, E., and Brent, M. R. Eval: A software package for analysis of genome annotations. *BMC Bioinformatics 4:50* (2003).

[13] Lafferty, J., McCallum, A., and Pereira, F. Conditional random fields - probabilistic models for segmenting and labeling sequence data. In *Proceedings of the Eighteenth International Conference on Machine Learning* (2001).

[14] Shapire, R. E. The boosting approach to machine learning, an overview. *MSRI Workshop on Nonlinear Estimation and Classification* (2001).