Spring 2017

Artificial Immune Systems: Applications, Multi-Class Classification, Optimizations, and Analysis

Brian Haroldo Schmidt
Western Michigan University, brian.schmidt27@gmail.com

ARTIFICIAL IMMUNE SYSTEMS: APPLICATIONS, MULTI-CLASS CLASSIFICATION, OPTIMIZATIONS, AND ANALYSIS

by

Brian Haroldo Schmidt

A dissertation submitted to the Graduate College
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
Computer Science
Western Michigan University
April 2017

Doctoral Committee:

Ala Al-Fuqaha, Ph.D., Chair
Robert Trenary, Ph.D
Ajay Gupta, Ph.D
Ikhlas Abdel-Qader, Ph.D

ARTIFICIAL IMMUNE SYSTEMS: APPLICATIONS, MULTI-CLASS CLASSIFICATION, OPTIMIZATIONS, AND ANALYSIS

Brian Haroldo Schmidt, Ph.D.

Western Michigan University, 2017

The focus of this research is the application of the Artificial Immune System (AIS) paradigm to a new research area along with the modifications necessary to adapt it to a new problem. In the past 10 years, there has been much research into the use of various Machine Learning (ML) algorithms in Network Flow Traffic Classification. AIS algorithms have thus far not been applied to this problem. Because AIS algorithms have been used extensively for Network Intrusion Detection applications, which is a similar area of research, the motivation to extend them to the network flow classification problem is clear.

This research also shows a technique for faster execution of the training and classification portions of an AIS algorithm, which are meant to speed-up the execution of the AIS algorithms and adapt them to resource-constrained environments. Additionally, the research performed for this study seeks to expand the knowledge available about the behavior of Artificial Immune System algorithms. Specifically, the effect of several different distance functions as well as different kernel functions on the accuracy of the AIS classifier. The optimization is also applied to the class of algorithms known as Negative Selection Algorithms (NSA).

This study includes a survey of the network traffic classification literature. It also contains a presentation of the history of Artificial Immune System algorithms, their inner

workings, and their previous applications. Furthermore, the reasoning for applying this type of algorithm to the network traffic classification problem is explained. Finally, the performance of the algorithm described in this study is analyzed by giving its big O complexity as well as a bound for its generalization error.

ACKNOWLEDGEMENTS

I would like to thank my parents for their support throughout my years in graduate school. I would also like to thank my advisor Dr. Ala Al-Fuqaha for his guidance and patience. Dr. Dionysios Kountanis also gave great support and it is only due to his help that I was able to finish the degree so quickly. I would like to thank Dr. Robert Trenary for reviewing several drafts of this document and his editing input. Lastly, I would like to thank the staff of the Computer Science department at Western Michigan for helping me through every stage of this degree program.

Brian Haroldo Schmidt

TABLE OF CONTENTS

Table of Contents—Continued

CHAPTER

Table of Contents—Continued

Table of Contents—Continued

LIST OF TABLES

LIST OF FIGURES

List of Figures—Continued

CHAPTER I

INTRODUCTION

The purpose of this work is to fulfill the requirements of the Computer Science PhD program at Western Michigan University. The subject matter of this research falls under the domain of biologically inspired computing.

The following subsections will provide a high-level introduction to the problem that will be explored in the dissertation. The motivation and importance of the work will also be discussed. Previous work in the field will be presented, as well as the work that will be included in the dissertation.

Scope

The research pursued for this dissertation will be the application of Artificial Immune System algorithms to networking problems. AIS algorithms are a biologically inspired class of algorithms which have been found to be appropriate for classification and optimization problems, particularly in binary classification problems. Because of their use in Network Intrusion Detection systems in previous research, it is now sought to expand the uses of AIS algorithms to include Network Flow Classification. To accomplish this, it is necessary to expand the range of possible classifications that can be done with AIS algorithms, to include multi-class classification.

Furthermore, it is also sought to adapt the algorithm to resource-constrained environments. To do this, two optimization techniques will be developed and tested with the algorithm. There is also a lack of basic research into some basic modifications to the algorithm that could increase the accuracy of its predictions. For example, different distance functions could be increase the

prediction accuracy of the algorithm. Hence, some basic techniques will be implemented within the AIS classifier in order to find out if they are useful.

The research also involves the transfer of some of the knowledge gained from the optimizations developed for the AIS classifier to other applications and datasets. The algorithm is optimized by reducing the number of times the distance function is calculated as well as by reducing the number of comparisons done by the algorithm. This optimization was originally developed by us for our own AIS-inspired algorithm and it is adapted to the class of AIS algorithms known as Negative Selection algorithms.

The scope of the dissertation will be introduced in chapters 2 and 3 of this work. Chapter 2 will introduce the problems being examined and chapter 3 will give background information for Artificial Immune System algorithms.

Rationale

The field of Artificial Immune System algorithms is now 20 years old and has seen considerable research effort [1]. However, it has not been used in many practical applications. The purpose of this research is to develop an algorithm inspired by AIS principles, and make it into a workable solution for real-world problems.

One of the main applications of the algorithm is classification for embedded systems. Embedded systems are small computers, limited in size and the amount of power usage allowed. Algorithms designed for these computers must consider these special requirements. The Internet of Things is a very big part of the motivation for this research, since it relies on small, inexpensive computers that are limited in memory and processing power [2].

More information about the rationale behind the proposed research is shown in the third section of chapter 3, which gives a thorough review of previous applications AIS algorithms to networking problems. Chapter 4 is a literature review of all research related to the current research.

Review of the State of the Art

Network flow classification is the problem of classifying data flowing across a network into different categories. Some of the ways in which a network flow can be categorized are: by application, by the amount of data it holds, and whether a human or machine generated it. Network flow classification has become more and more important recently, and much research has been published in the application of Machine Learning techniques to it. Some ways to accomplish flow classification are: using port numbers, deep packet inspection, host behavior based approach, and using the statistical properties of the data passing over the network.

There have been many Machine Learning algorithms applied to the problem of network flow classification. Among them: SVM, Naïve Bayes, C4.5 trees, RIPPER, Multi-layer perceptrons, Bayes Net, and Radial Basis Function Neural Networks. However, none of these have been customized for execution on resource-constrained and embedded systems, as is the algorithm under study in this research. At the time of this research, there has not been any effort to apply Artificial Immune System algorithms to the problem of network flow classification.

A complete review of the state of the art of machine learning algorithms applied to network flow classification is given in the fourth section of chapter 3. Furthermore, Figure 7 gives a visual representation of the place of the current research in the field.

Significance of the Proposed Research

The research direction proposed in this dissertation is important because, although there has been much research in the field of Artificial Immune systems, there have not been many real-world applications for the algorithms developed. The research proposed here not only applies an AIS-inspired classification algorithm to a real and pertinent problem in networking, it will develop an optimization technique that allows the algorithm to compete with other, more well-known classification algorithms. Specifically, the AIS algorithm proposed here is very applicable to the new and flourishing field of algorithms for embedded systems, which are very important in the technology known as the Internet of Things (IoT).

The proposed research is inspired by previous work in the field of Artificial Immune Systems algorithms. However, because of challenges in the training and classification time required by this type of algorithms, as well as other considerations, the research breaks with some of the traditional techniques typically associated with AIS algorithms. Specifically, the training does not proceed randomly, as with other AIS algorithms, which makes the proposed algorithm much faster. Furthermore, there are no other AIS algorithms using the optimizations studied in this research. The use of kernel functions is also investigated, as well as different distance measures, both of which have been researched in the context of AIS algorithms before, but not on a real-world data set, as done here.

This research is important because it shows that it is possible to use AIS-like algorithms on real-world problems. Also, the algorithm developed and tested is able to approach the complexity of SVM and Naive Bayes algorithms, something that is not easily done with AIS algorithms. At the same time, the AIS-like algorithm proposed here is able to improve in several

ways on the SVM and Naïve Bayes algorithms, as it is able to improve on the accuracy of these when used with small training sets. The AIS-like algorithm also uses parameters that are easy to set up and understand. Lastly, through tests, it has been shown that the AIS algorithm does not require the use of kernel functions to achieve high accuracy, something that that the SVM algorithm cannot claim. Because of the lack of kernel functions as well as the simplicity of the algorithm itself, the algorithm does not require setting many parameters to function well, something that is very important in embedded systems.

A more detailed description of the similarities and differences between the current research and previous published research is given in chapter 5, which also describes the original contribution that the current research brings to the field.

Delimitations

The research pursued for this dissertation will be the application of Artificial Immune System algorithms to networking problems only and does not seek to apply to other classification problems or data sets. Furthermore, it is also sought to adapt the algorithm to resource-constrained environments through the design of two optimizations, no other modifications to the basic algorithm are considered. This research also seeks to tests the effects of different distance functions and kernel functions on the performance of the classifier, no other variations on the basic algorithm will be investigated. The research also involves the transfer of some of the knowledge gained from the optimizations developed for the AIS classifier to other applications and datasets, this is limited to the Negative Selection algorithm only.

Limitations

Although every measure has been taken to make sure that the results in this dissertation are applicable in as many situations as possible there are some limitations. For example, only a few data sets are used to tests the algorithms. There could be some data sets for which the algorithms will give very different results. It is sought to describe these limitations whenever they may be found.

Assumptions

In the same way as the previous section, the assumptions of this dissertation are centered on the data sets used. For example, the flow classification data set used for some of the experiments. It is assumed that this data set is a fair and balanced sample of internet traffic and does not emphasize certain types of traffic over others. The nature of the internet has not changed considerably in the last 10 years, even as its size has increased exponentially. It is believed that assumptions are safe to make and the research is still relevant.

CHAPTER II

PROBLEM STATEMENT

The research making up this dissertation falls under the umbrella of Machine Learning as well as Network Traffic Classification. This chapter introduces these concepts.

Statistical Classification Problem

This section introduces the idea of a classifier, defines the notion of a classifier mathematically, and shows how to measure the performance of a classifier.

Within the field of statistics, a model is used to describe a real-world process; within the field of Machine Learning, statistical models are used to make predictions about the world. When statistical models are used to make predictions about the class membership of an entity they are called "classifiers." Creating a classifier is called "training," and using it to make predictions is called "predicting" or "classifying." There are many ways in which to train a classifier, all of which fall into one of these categories: supervised learning, unsupervised learning, reinforcement learning, and evolutionary learning. This dissertation will deal exclusively with supervised learning.

Supervised learning is done when there is data available that is correctly labeled, or when the desired output of the algorithm is packaged with the data in the data set. The goal is to model the underlying distribution that generated the samples. Unsupervised learning is done when there is no data available with the correct label or desired output. Instead, interesting patterns are gleaned from the data without human intervention. Reinforcement learning is used when the environment is dynamic and changing. It is useful when there is no label or category that can be learned, instead using the concept of a "reward" that needs to be maximized.

Machine learning algorithms can be divided into two categories: online learning algorithms and offline learning algorithms. Offline algorithms derive a model from the data available and do not change it after the initial training phase ends. Online algorithms learn one instance at a time, and do not use an entire data set to derive a model. Online learning algorithms often have correct class labels available to them after a pause, instead of being bundled with the data, as in offline machine learning. For example, online learning can be used to make predictions of stock market prices, updating its model as data becomes available.

Another distinction between learning algorithms is between eager learning algorithms, and lazy learning algorithms. Eager algorithms build a model from training data in an explicit step, and do not change the model without repeating the training step. Lazy algorithms, on the other hand, do not build a model until a prediction is requested from the algorithm. Instance-based learning is a type of lazy learning, in which a query is compared directly to the training data to make a prediction.

Classifiers are machine learning algorithms that assign a category or class label to a previously unseen observation. The assignment can happen either through supervised learning or unsupervised learning. When done with supervised learning, the correct category of training data is known, and when done with unsupervised learning, the correct category of training data is not known but is inferred from similarities between training observations.

This dissertation will deal with both binary classification and multi-class classification. Binary classification problems only classify instances into two classes. Multiple class classification problems involve more than two classes and are more complicated than binary

classification problems. Moreover, a multiple class classifier can be built using many binary classifiers.

Algorithms that perform classification work with data sets which are made up of vectors of data. A vector is a collection of ordered data. A labeled data set includes a class label with each vector, an unlabeled data set does not. Each element in a vector is called a "feature" in machine learning. Features can be binary (when it can only take on two values), categorical (when it can only take on a fixed number of possible values), ordinal (when the values can be compared in an ordinal scale, i.e. they can be ranked), integer-valued, or real-valued. These types of features can be combined to describe complicated entities in the world, such as speech, text, or pictures. The vector space is also called the feature space, which can be thought of as the space in which all possible feature vectors can exist. [3]

There are four main areas in which a classifier can be evaluated:

1. Accuracy: the dependability of the classifier, defined as the proportion of successful classifications out of all attempted classifications. Some similar measurements are: recall, precision, and f-measure.
2. Speed: the amount of time needed for the classifier to evaluate the data given to it and make a prediction.
3. Comprehensibility: describes a quality that allows a human to understand the inner workings of the classifier. Some classifiers can easily explain their decisions to humans, others cannot.
4. Training time: the amount of time that the classifier takes to build a model or develop a rule that will be used to make predictions. [4]

In this research, the amount of time a classifier takes to make a prediction will also be investigated.

Formal Definition

As stated above, this research deals primarily with supervised learning. The result of supervised learning is a model, which can then be used to make classifications. Classification can be seen as a mathematical function, which takes a data point as an input and outputs the category or class identifier. Such a function can be very simple or complicated, and may or may not be representable in human-understandable terms. An implementation of the function is called a classifier. The simplest type of classifier is a binary classifier, therefore a binary classifier is described formally in this section.

Although there are more types of variables (e.g. categorical variables) that can be used by a classifier, the feature space is described as continuous here, for the sake of simplicity:

$$X = \mathbb{R}^d \tag{1}$$

where the number of dimensions is d. Furthermore, a vector belonging to X is denoted:

$$x = [x_1, x_2, x_3, \ldots, x_d] \in \mathbb{R} \tag{2}$$

The job of a classifier is to map every vector x in vector space X to a class y_i, where:

$$y_i \in Y \tag{3}$$

since binary classification is being described:

$$Y = \{1, -1\} \tag{4}$$

The mathematical definition of the supervised learning classification problem follows. Given a set of learning data S, consisting of pairs of inputs and outputs:

$$S = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), \ldots, (x_{|S|}, y_i)\} \tag{5}$$

where each $x \in X$ and each $y \in Y$. The goal of a learning algorithm is to build a function h that approximates a function y, where y is the true function that classifies an input vector x to a category y, being described like this:

$$h(x): X \rightarrow Y \tag{6}$$

mapping a member of set X to set Y.

To test a classifying function h created by the training process, a set of test data is used, which is of the same structure are S:

$$T = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_{|T|}, y_i)\} \tag{7}$$

The quality of the predictions of a binary classifier can be measured in this way:

$$\sum_{i=1}^{|T|} \frac{I(h(x_i) = y_i)}{|T|} \tag{8}$$

where I is the indicator function, which is equal to 1 when the statement is true and 0 when it is false. The goal of a training algorithm is to produce an accurate model, so that, when used to make predictions it will maximize the above function.

Introduction to Multi-Class Classification

Multi-class classification is a natural extension of the binary classification problem. Simply, the definition above can be extended by making the set Y equal to:

$$Y = \{y_1, y_2, \dots, y_{|Y|}\} \tag{9}$$

There are two ways in which multi-class classification can be accomplished. A classifier built specifically for multi-class classification can be designed, or the problem can be decomposed so that it can be handled by a set of binary classifiers.

There are several common classifiers that can handle multi-class datasets. Artificial Neural Networks are classifiers inspired by the nervous systems of mammals and other animals. They consist of collections of interconnected neurons. Individual neurons are capable of making simple choices. With enough interconnected neurons, more complicated decisions can be made. Information storage is accomplished by changing the value of "weights," which affect how a single neuron is connected to its neighbors. Training is accomplished by changing the weights of the connections between neurons. Neural networks are one of the most common approaches to classification.

When dealing with real-valued data, the k-nearest neighbors algorithm is a very simple and useful classifier, since it does not require a training process. To do a classification, the vector to be classified is compared against all vectors in the dataset, and a distance function is calculated. The k "nearest" neighbors class label is selected to be predicted class label of the vector to be classified, where k is an integer. If k is equal to more than 1, then a vote is taken and the majority class is assigned to the example. This classifier is very easy to implement, but it does not scale well to larger data sets.

The Naïve-Bayes classifier is based on Baye's rule, which shows the relationship between the probabilities of two events. By using this rule, the probability of an example belonging to a certain class is calculated and the class with the highest probability is the returned as the class to which the example belongs to. The Naïve Bayes classifier is called naïve because it assumes that each feature is independent from every other feature in the dataset, something which may not always be true. This assumption, however, is usually true enough and very useful, since it cuts

down on a large amount of processing. The Naïve Bayes classifier is widely implemented and used.

Since their introduction, Support Vector Machine (SVM) classifiers have been the gold standard in binary classification, and can be used in multi-class classification with the use of specialized problem decomposition techniques. SVMs work by simply drawing a hyper plane within the vector space that separates the data points of the classes present in the data. The hyper plane is drawn so that the distance between the classes is maximized. Samples to be classified are compared to the hyper plane drawn by the classifier. If the classes in a data set cannot be separated by a hyper-plane the data set is said to be non-linearly separable. When the dataset is not linearly separable, a kernel trick can be used. This involves the use of a kernel function which maps all of the data points into a higher dimensional space, in which the data points can be linearly separated.

There are several ways to solve a multi-class classification problem with binary classifiers. The one-versus-all (OVA) approach uses K binary classifiers, where K is equal to the number of classes present in the dataset. One classifier is trained on each class, artificially dividing the data set into two subsets, where one set of examples is the class that the binary classifier currently being trained will recognize, and the other set is made up of all other examples in the data set. This creates a binary classification problem from the multi-class problem. This approach creates an ensemble classifier, since instead of having one output, now the outputs of K classifiers have to be dealt with. To deal with the problem, the class of the classifier with the highest confidence in its prediction is considered to be the correct class of the sample.

The all-versus-all (AVA) decomposition is more complicated than the one-versus-all decomposition, and it requires more classifiers to be implemented. This type of decomposition

compares all classes against all other classes, meaning that K(K-1)/2 classifiers have to be trained. For each pair of classes, a dataset is created using only examples from the two classes being compared, and a classifier is trained on this data. To classify a pattern, all classifiers are run on it and a vote is taken, the class with the highest number of votes is considered the class the pattern belongs to. [5]

The techniques mentioned here are general approaches to multi-class classification. There are many more that can be mentioned, and this section is only meant as an introduction.

Statistical Network Flow Classification Problem

The classification of network traffic has become important in recent years, since it is useful for network operators to have information about the information that flows over their networks. Some uses for this information are: for the prioritization of the packets of applications that might require Quality of Service (QoS), or disabling the network access of users that are committing illegal acts. A few current issues in this field are: network neutrality, the sharing of copyrighted material over networks, and the conflict between malicious users and network administrators [6]. There are a few ways to partition network traffic into different groups. This dissertation deals with the problem of categorizing network traffic by the type of application that generates the data.

Network traffic classification techniques have been used for many years, with growing complexity required, as techniques for hiding information about the identity of data have been developed as well. The simplest approach is to use the port number associated with the TCP connection to identify the application. This is a very fast approach that uses well-known port numbers. This approach has several weaknesses. Furthermore, certain applications are able to

negotiate a dynamic port number to exchange data, thus avoiding the use of standard port numbers. The efficacy of this technique for traffic classification has gone down recently [7].

Deep packet inspection is a technique for classifying network traffic that involves examining the contents of packets. To accomplish this, the contents of packets are compared against stored patterns of known applications. This technique produces accurate results, but is computationally expensive. Moreover, the legality of examining the data of users is questionable. This technique is also easily sidestepped by the use of encryption, which makes it impossible to view the data directly.

Another way in which the application that is generating the data might be identified is by identifying the patterns of interactions between a user and an application server. Applications interact with servers in known ways, and this can be used to identify them. This is called the host behavior based approach.

The approach to network flow classification studied in this dissertation is based on the statistical properties of the data passing on the network. The information is gathered without looking at the contents of packets, and is therefore avoid legal dilemmas. Some of the information used is: port numbers, inter-packet delay, packet counts, as well as calculated features such as the averages and medians of these values. The information used can also come from packet headers, but never from packet payloads. Although it is not the only way to do it, the statistical classification approach to packet flow classification uses techniques from data mining and Machine Learning (ML) to build models of the data.

CHAPTER III

BACKGROUND

This section presents some of the background of the domain dealt with in this research, including an introduction to biologically inspired algorithms, natural immune systems, artificial immune systems, and network flow classification.

Biologically Inspired Computing

Over the last seven decades, computer scientists have been taking inspiration from biological systems to develop techniques to solve problems with computers. This approach is based on the idea that biological systems have been solving complicated problems long before computers were invented, and therefore have important insights into very hard problems. The problems solved by biological systems are usually very complex and studying them has given computer scientists original and useful ideas.

An important idea within biologically inspired computing is emergence. Emergence is used to describe the way in which complex behaviors arise within systems of simple agents that act according to simple rules. Almost all biological systems display emergent behaviors, either in the macro or micro levels. Thus, biologically inspired algorithms sometimes use emergent properties to solve problems. Emergent behaviors can also be found within the disciplines of economics, physics, chemistry, and even psychology. For example, emergence is used to study the food foraging activities of bee colonies. This research has inspired bee colony algorithms, which are useful in solving optimization problems.

Another way in which biologically inspired computing can be understood is through the idea of connectionism. Connectionism is used to understand systems of interconnected systems of

simple units. As with emergence, biological systems that are sufficiently complex also display the property of connectionism. An area of research in which connectionism is important is artificial neural networks.

Another way to understand biologically inspired computing is through swarm intelligence. Examples of swarm intelligence in nature can be found in the herding behavior of certain animals, the growth of bacteria colonies, the activities of ant colonies, the foraging behavior of bees, and many other examples. Swarm intelligence is not a well-defined field of study, but does include elements of connectionism and emergence. Swarm intelligence methods are characterized by populations of simple agents that interact with each other and their environment. The population solves a problem by acting locally, without global knowledge. The field of ant colony optimization algorithms is directly inspired by the swarming behavior of ants, and has proven to be a valuable technique for any problem that involves finding paths through graphs.

Biologically inspired computing is related to Artificial Intelligence (AI) and can be considered a sub-field of it. The approach used by biologically-inspired algorithms is very different from most other AI algorithms. Many AI algorithms take a top-down approach to solving problems, where the programmer sets up the algorithms and all of its parameters. Alternatively, biologically inspired algorithms do not require the programmer to set up the initial conditions perfectly, or add all necessary knowledge to the algorithm at the start of execution. The execution of biologically inspired algorithms is often non-deterministic, since they frequently have randomness built-in. Another feature of biologically inspired algorithms is their simplicity, since the components themselves are simple, and the mathematics required to describe them are also

simple. Lastly, another feature of biologically inspired algorithms is their amenability to execution on SIMD and MIMD processors.

Ant-Hill Algorithms

Ant hill algorithms are a type of a biologically-inspired algorithms that are well suited to finding the best path between two nodes in a graph or network. They are derived from the behavior of ants who are able to reliably find the shortest path between their colony and a source of food, and to communicate this information to other ants.

The basic elements of the algorithm are: ants, which are agents that travel along graph edges according to a set of rules, and pheromones, which are left behind by the ants to communicate with other ants. Like many other biologically inspired algorithms, ant hill algorithms run until a certain number of iterations are executed, or when an ending condition is met. Ants choose a move between vertices in the graph according to two parameters: the attractiveness of the move (computed by some desired heuristic), and the pheromone trail left behind by other ants that made the same move in the past. Good solutions leave a stronger pheromone trail and weaker solutions leave behind a weaker pheromone trail. At the end of the algorithm the best solution, according to a pre-defined heuristic, is selected. The basic steps of the algorithm are shown in Figure 1.

Ant hill algorithms have been applied to many problems that can be represented as graphs. For example: the traveling salesman problem, protein folding, scheduling problems, and the set cover problem. Another feature of ant hill algorithms is their ability to adapt to changing network conditions in real time, which allow the algorithms to run continuously.

```
Initiate the population of ants
WHILE !terminated
    FOREACH ants in population:
        Make a list of possible moves
        Move to a random vertex in the proposed list of moves, according to attractiveness
        and pheromones
        IF ant has finished a solution
            Leave behind a pheromone trail along that path
        ENDIF
    END FOREACH
END WHILE
RETURN best solution found
```

Figure 1. Pseudo Code for the Ant Hill Algorithm

Genetic Algorithms

Genetic algorithms (GA) are a class of biologically-inspired algorithms which are used to solve search and optimization problems. Genetic algorithms are inspired by the process of natural selection and use the concepts of mutation, inheritance, selection, and crossover.

Like other biologically inspired algorithms, GAs use a population of proposed solutions, where each solution is like the genes of an organism in an environment. Like ant hill algorithms, GAs are iterative algorithms that run for a set number of "generations," or until a stopping condition is met. By applying the process of natural selection to the population of proposed solutions, a GA is able to create a set of solutions that is more and more fitted to its environment with each passing generation. The "fitness" of a proposed solution is defined by a fitness function, which assigns a fitness score to each proposed solution.

A genetic algorithm may use one of several genetic operators to improve the fitness score of the population of candidate solutions: mutation, crossover, and selection. Each of these operations is best understood in its biological sense, since they are direct copies from the biological domain. Mutation happens in an organism when its genes are randomly changed. Likewise, a GA

will randomly change one or more of the "genes" of a proposed solution. Crossover happens when two organisms combine their genes to produce offspring. In GAs, two solutions may combine their genes to produce a solution that shares their genes. Selection is implemented on the whole population at the end of each iteration. To do this, the portion of the population of solutions with the highest fitness scores will be selected to create the next generation of solutions, replacing the solutions with lower fitness scores. A simple pseudo code implementation of a genetic algorithm is shown in Figure 2.

As with other biologically inspired algorithms, it is not always easy to analyze the execution of a GA to find how it arrived at a solution. GAs have been used to solve scheduling and engineering problems. Generally speaking, they are most useful when the solution domain is very complex and contains many local optima.

```
Initialize population
WHILE !terminated
        Apply fitness function to all members of population
        Select a given portion of the population with the highest fitness function
        Select pairs to mate, add to population using offspring from mated pairs
        Apply mutation operator to a given portion of the population
END WHILE
RETURN solution with highest fitness score
```

Figure 2. Pseudo Code for the Genetic Algorithm

Particle Swarm Optimization

Particle swarm optimization (PSO) is a biologically inspired algorithm that is drawn from the behavior of swarms of insects as well as flocks of birds. PSO is used for optimization problems and it excels when the solution space is large and multidimensional. As with other algorithms discussed in this section, PSO works with a population of "particles," and it solves problems iteratively. Each particle represents a candidate solution, and the solution is encoded in the

coordinates of the particle in the solution space. Each particle also has an associated velocity. The velocity encodes the particle's direction and speed through the solutions space. The PSO algorithm also requires a function that measures the quality of a candidate solution, which is similar to a GA's fitness function.

During each iteration, the PSO algorithm will move its population of particles around the solution space according to mathematical formulas using each particle's position and velocity. Each particle's position and velocity at the end of each iteration is based not only on its original position and velocity, but also on the position and velocity of every other particle in the swarm. Over time, the swarm of particles will move towards better quality solutions. A pseudo code implementation of a simple PSO algorithm is shown in Figure 3.

PSO algorithms are particularly flexible in the types of problems that they can work with. PSO does not require that the function that calculates the quality of a solution be differentiable. PSO can also deal with problems that change over time. PSO cannot guarantee that the solution that is returned is the global optimum solution, however.

```
Initialize the population of particles with random positions and velocities
WHILE !terminated
    FOR EACH member of the population of particles
        Randomly move each particle's position w/ a bias towards the best position
        Apply the new velocity to update positions
        IF fitness of this particle is better than the best known global position
            Update the best known global position
        ENDIF
    END FOR EACH
END WHILE
RETURN solution with highest fitness
```

Figure 3. Pseudo Code for the Particle Swarm Optimization Algorithm

Artificial Immune System Algorithms

Artificial immune system algorithms are biologically inspired algorithms that are useful in classification and optimization problems. They are described in the following sections.

Natural Immune Systems

In the past 20 years, the mammalian immune system and other natural immune systems have inspired the development of computer algorithms that mimic their approach to accomplish classification and optimization in complex environments. These algorithms leverage the mammalian immune system's (MIS) memory and learning capabilities.

The immune system is made up of organs and cells, some of which move around the organism in the bloodstream. Some of the organs that make up the immune system are: the thymus, lymph nodes, and bone marrow. However, only some of these are useful when adapting the mammalian immune system paradigm to the field of computer science. The mammalian immune system has two main components: the innate immune system and the adaptive immune system. The difference between them is that the innate immune system is not able to adapt to the environment, the adaptive immune system does. Both sides of the MIS work together to protect the host from infections.

The innate immune system takes many forms, and is present in many part of the body. Some of the organs that implement this system are: the skin, gastrointestinal tract, respiratory airways, and eyes. Within the respiratory system, nasal hair, mucus, coughing and sneezing form obstacles against infections. In the digestive system, mucous membranes, stomach acids, and saliva also form physical barriers. Lastly, the pH level of the body as well as the temperature of the body also help to keep invaders at bay. These organs are the first line of defense against

pathogens, and provide a physical and chemical barrier against infection. However, the innate immune system is not useful to computer scientists, since it does not perform classification and does not adapt to the environment. The innate immune system is not only found in mammals, but also in plants, fungi, and bacteria.

Another type of cells used by the immune system are T-cells. They are involved in the regulation of other cell's actions as well as attacking pathogens that are attacking the host. There are few different types of T-cells, including T-helper cells, cytotoxic T-cells and suppressor T-cells. Some other organs active within the immune system are lymph nodes, which act as convergence sites of the lymphatic vessels, where each node stores immune cells, including B and T cells. Some specialized lymph nodes are: tonsils, adenoids, appendix, Peyer's patches. The spleen is also active in the destruction of organisms that have invaded the blood stream.

The adaptive immune system makes up the second part of the MIS, and is responsible for maintaining a memory of the pathogens that have previously attacked, as well as classifying previously-unseen pathogens. The adaptive immune system is able to distinguish between self and non-self cells in the body. The adaptive immune system is made up of several organs and several types of cells. Bone marrow produces many types of specialized cells, such as T-cells and B-cells. The thymus, located behind the sternum in humans, processes immature cells before they are activated, to prevent auto-immune diseases.

The basic action of recognition is accomplished through proteins found on the surfaces of all cells. The MIS produces molecules known as antibodies that lock onto these proteins, signaling that a pathogen is present to the rest of the immune system, which then takes action against the foreign cell. Within the bloodstream, cells that produce antibodies are called B-cells. The thymus

protects the host by removing immature NIS cells that produce antibodies that incorrectly recognize self tissues as an infection, before they can cause damage to the host. This process is known as negative selection.

The adaptive immune system is also remarkable because it leverages a small number of genes in the genotype of the host to produce a large number of different types of antibodies. This is notable because the space of possible proteins and possible antibodies is very large. This capability allows the body to search for and find the correct antibody to classify a previously-unseen infection, with very little energy investment.

The mammalian immune system can be viewed as a classification system functioning inside of bodies of mammals. The MIS performs binary classification, since it distinguishes between self (the organism's own cells and tissues), and non-self (foreign bacteria, viruses, etc.). It is able to protect the host against a large number of bacteria and viruses. It is able to do this mostly without prior knowledge of the pathogens that will attack, and is also able to learn new classification rules when new pathogens are encountered. It is also able to adapt to changes in the environment. Furthermore, the mammalian immune system is able to distinguish between useful bacteria that operate in the body and malignant bacteria, highlighting its discriminative power. Lastly, natural immune systems are self-organizing, lightweight, and energy efficient.

Algorithm Overview

The field of artificial immune system algorithms is based on the simulation of the elements of the natural immune systems found in animals, within computers. AIS algorithms use populations of elements to solve problems, much in the same way that genetic algorithms use

populations of candidate solutions. The goal of AIS algorithms is to take advantage of the adaptive capabilities of natural immune systems as well as their ability to memorize complex patterns. [1]

Feature Space

When working with data within the discipline of machine learning, it is often useful to see every vector as located within a "feature space," also known as "shape space." Each element in the vector corresponds to a feature in the data set. Often the values of the elements of the vector can be normalized to the range [0,1], which simplifies the implementation of the algorithms using the data. Each vector present in the data set is then located as a point in the feature space.

Negative Selection

The Negative Selection (NS) algorithm is inspired by the actions of the natural immune system. The first paper to describe the use of the Negative Selection algorithm in computer science is [8]. Negative selection is based on the action of the thymus, which selects the cells in the immune which detect self and removes them. By doing this, the thymus implements a simple strategy for detecting non-self tissues, and preventing auto immune diseases.

The negative selection algorithms trains a population of antibodies very simply, by using a set of samples for the patterns that need to be recognized. An individual antibody is not required to be of any specific type or structure, as long as it is able to implement a recognition rule, and can be a hyper sphere, a string matching rule, or a more complicated algorithm. An antibody is implemented as a simple classifier. The negative selection algorithm generates random antibodies, and tests them against all samples of "self" in the training set. If an antibody accepts a sample of the self class, then that antibody is rejected from the population of antibodies. Otherwise the

antibody is accepted into the population [8]. A pseudo code listing of the negative selection algorithm is found in Figure 4.

There have been many different implementations of antibodies tested in the literature. All of the implementations tested have one thing in common: they are all collections of requirements or filters that must be satisfied for the antibody to be "activated." All types of antibodies either recognize or do not recognize a sample. The "size" of the antibody is equal to the number of requirements. The most common implementations of antibodies are simple range antibodies, r-contiguous bit antibodies, and hyper-sphere antibodies.

```
input:
      S_self, set of seen self elements
output:
      population, the set of antibodies
functions:
      matches(): a function that returns true if an antibody recognizes an element
      generate_random_antibody(): a function to generate a random antibody

BEGIN
      population = {}
      WHILE !stopping_criteria
            new_antibody = generate_random_antibody()
            match = FALSE
            FOREACH { s | s ∈ S_self }
                  IF matches(s, new_antibody)
                        match = TRUE
                  ENDIF
            IF !match
                  population = population ∪ new_antibody
            ENDIF
      ENDWHLE
END
```

Figure 4. Negative Selection Pseudo Code

Simple range antibodies make comparisons between the antibody to the data. For every feature of the example, the antibody defines a range within which the antibody will accept the data. If an antibody accepts all of the features of an example, then the example is recognized by the antibody.

R-chunk and r-contiguous antibodies work with strings of symbols. The antibodies are themselves strings of symbols. With r-contiguous antibodies, the detectors are always of the same length as the strings to be matched, and the antibody matches a string if there is a substring of size r present in the example string that matches the antibody's string in the same positions.

Just like r-contiguous antibodies, r-chunk detectors are made up of a string of the same alphabet as the strings to be matched, and an integer r. An r-chunk detector will match a string if at any position in the given string substring of length r matches where all characters are identical to the detector's string. The only difference between r-contiguous and r-chunk detectors is that r-chunk detectors are not required to be the same length as the sample strings. [8]

Another detector type is hyper-sphere antibodies, which are defined by a center position in the feature space, and a radius. Some common distance measures are: Euclidian distance, Minkowski distance, and Chebyshev distance. There are also distance metrics that can be used with strings. For example: Hamming distance, binary distance, edit distance, and value difference metric. A complete description of these and other ways to calculate affinity can be found in [9]. Hyper-sphere antibodies match a data point when the data point lies within the radius of the antibody, as defined by the center point and radius of the antibody.

Positive Selection

Positive selection is found in the Natural Immune System and has been borrowed by AIS algorithms, although not as extensively as negative selection. Positive selection is modeled on a receptor filter found on the surfaces of immature T-cells. The receptor is a molecule called MHC and it allows the organism to filter out T-cells that have not matured yet. Positive selection works similarly to the Negative Selection algorithm, but instead of not adding antibodies to the population if they match self, they are added.

Clonal Selection

The clonal selection principle deals with the ways in which antibody-producing cells act to maximize the likelihood of finding an antibody to match a new type of pathogen, and the ways in which they remember previously encountered pathogens.

When a B-cell is exposed to a pathogen a second time, it undergoes the process of clonal selection, where it makes copies of itself in order to be able to deal with the threat. Clonal selection allows the immune system to respond a lot faster to pathogens that have been detected before, as well. The second time that a pathogen is detected, B-cells differentiate into plasma cells and long-lived B-cells, both of which are able to produce antibodies that match the pathogen that triggered the primary response.

Clonal selection in AIS is a way to ensure that B-cells that produce antibodies that are successful in identifying instances of non-self are able to get more antibodies in the antibody pool. This is a simple mechanism that allows the immune system to optimize itself to environment, and to marshal its resources effectively. However, the natural immune system does not clone B cells perfectly. Genetic mutations are inserted in the copied B-cells. This is implemented by having the

cloning process introduce small errors, in this way the cloned B-cells are able to recognize instances of non-self that are very close, to what the original B-cells recognized. The addition of the errors in cloning allows the population of B-cells to adapt itself to new threats [10]. Pseudo code for the Clonal Selection algorithm can be seen in Figure 5.

Classification

Although there are many complicated schemes to perform classification, the classification step of an AIS algorithm is very simple. The negative selection training creates a population of simple classifiers that are used to classify a test sample. If a singly antibody recognizes the test sample, then it is classified as "non-self" [8]. Pseudo code for the classification portion of the Negative Selection Algorithm can be seen in Figure 6.

Danger Theory

Danger theory is an aspect of natural immune systems that has inspired a type of artificial immune algorithm. It has been proposed by Matzinger [11] that negative selection is not the only mechanism that is at work when the immune system is classifying tissue as self on non-self. There are many ways in which non-self tissue is allowed to work in the body, for example: foreign bacteria in the gut. At the same time, there are examples of self tissues being attacked by the immune system as is the case with some tumors.

Because of these observations, Matzinger suggested Danger Theory as an alternative way in which the body recognizes pathogens. These extra distinctions are added: 'non-self but harmless' and of 'self but harmful', which makes four categories in total. The mechanism that danger theory uses to detect these new categories is the detection of damage being done to the

```
input: population_size: the size of the desired population
       selection_size: the size of the population to be kept at the end of each iteration
       random_size: the number of antibodies to generate at the end of an iteration
       clone_rate: the number of clones that are made from each selected antibody
       mutation_rate: the number of mutations applied to each antibody
output:
       population, the population of antibodies generated
functions:
       generate_random_antibody(), a function to generate a random antibody
       affinity(), a function to calculate the affinity of an antibody
       select(), a function to select the antibodies with the highest affinity
       clone(), a function to copy an antibody
       hypermutate(), a function to perform hyper mutation on an antibody
       replace(), a function to replace the lowest-affinity member of the population
BEGIN
       population = {}
       size = 0
       WHILE size < population_size
              Population = population ∪ generate_random_antibody()
       ENDWHILE
       WHILE !stopping_criteria
              FOREACH { p | p ∈ population }
                     p[affinity] = affinity(p)
              ENDFOREACH
              population_selected = select(population, selection_size)
              clones = {}
              FOREACH { p | p ∈ population_selected}
                     clone = clone(p, clone_rate)
                     clones = clones ∪ clone
              ENDFOREACH
              FOREACH { p | p ∈ clones }
                     hypermutate(p, mutation_rate)
                     p[affinity] = affinity(p)
              ENDFOREACH
              population = select(population, clones, population_size)
              size = 0
              random_population = {}
              WHILE size < random_size
                  random_population = random_population ∪   generate_random_antibody()
              ENDWHILE
              replace(population, random_population)
       ENDWHILE
END
```

Figure 5. Clonal Selection Pseudo Code

```
input:
        S_antibodies, set of antibodies created by training algorithm
        example, example to be classified
output:
        class of the example
functions:
        matches(): a function that returns true if an antibody recognizes an example

FOREACH { a | a ∈ S_antibodies }
        IF matches(a, example}
                return "non-self"
        ENDIF
ENFOREACH
RETURN "self"
```

Figure 6. Negative Selection Classification Pseudo Code

tissue of the organism. Very simply, if the organism's tissue is being damaged, then there is non-self tissue present or self tissue that is harmful.

Danger signals are sent out from tissue that is being damaged, usually because of cell death. A cell can die of natural causes, if this happens no danger signals are sent out. However, if a cell dies of unnatural causes chemical signals are sent to the immune system that bring T-cells to the affected area. The danger signal, in an AIS algorithm that uses it is grounded in data about system activities. Whereas non-self data in a Negative Selection algorithm is simply a set of feature vectors with no further information, the danger signal is an extra source of information when doing classification [12].

Dendritic Cell Algorithms

Dendritic cell algorithms are an area of AIS that is related to Danger Theory. Dendritic cells act as an interface between the innate and adaptive parts of the natural immune system.

Dendritic cells take in antigen material from the organism and make it available to other types of cells in the immune system. Dendritic cells also react to danger signals.

The dendritic cell algorithm works with a population of cells that recognize patterns in data. The cells may be in three states: mature, semi-mature, and immature. The population of cells is trained by exposing the immature cells to data, and transitioning immature cells to semi-mature, and then to mature status. Mature cells recognize patterns in data that are deemed "dangerous."

Applications of AIS

The natural immune system is a powerful decentralized classifier with some remarkable properties. For example, the immune system is able to learn the signatures of previously unseen pathogens keeping their signatures in memory for later use. It is also highly decentralized, having no central point of failure. Lastly, it eludes specialization, managing to learn a wide variety of pathogens. It is also capable of perceiving and combating dysfunction from the host's cells as well as foreign pathogens.

Because of its abilities, the natural immune system has been adapted to many different applications. Some examples are: fault detection, computer security, change detection, and network intrusion detection [13]. Artificial immune systems have also been used in robotics applications, multi-modal function optimization, and intelligent control systems [14].

In [15], the authors show how to implement an adaptive critic system for autonomous aircraft. An adaptive critic is an AI system which controls the actions of intelligent agents. It is used to predict the performance of the agent in the future and plan a correct path through the environment. In a similar application, the authors of [16] use an immune-inspired algorithm to implement a management system for a semiconductor production line.

Some other areas in which Artificial Immune Systems have been applied are: computer security [17], anomaly detection [18], fault diagnosis [19], data mining [20], adaptive control [22], robotics [23], and optimization [24]. These applications highlight the many places in which AIS and similar biologically inspired algorithms have been used.

AIS and Network Intrusion Detection

Network intrusion Detection (NID) systems is a type of software used to distinguish unusual and illegal behavior in networks. Because of the nature of the natural systems that inspired them, artificial immune system algorithms have been used to build network intrusion detection systems. NID systems have been developed for ad-hoc networks, wireless networks, for detecting botnets, and detecting binaries compromised by viruses. This area of research has been a major source of inspiration for the current research and an introduction is provided in this section.

NID systems can be classified into two ways: the way in which the NID system analyses the network traffic, and the placement of the agents that monitor network traffic. There are two ways to analyze network traffic: misuse detection and anomaly detection. Misuse detection examines network traffic for well-known attacks, comparing patterns found in the network to previously-seen and labeled patterns. Anomaly detection involves building a model of what normal network behavior looks like, and comparing current network traffic to the model, identifying attacks by their differences to normal behavior.

Misuse detection approaches usually have low false-positive rates, but also suffer from high false-negative rates. The models behind their classifications are updated after a novel attack is identified, and are therefore unable to identify new types of attacks or attacks that have been intentionally obfuscated. Anomaly detection NID systems are usually able to identify novel

attacks, since they do not fit their model of normal network behavior. However, anomaly-based NID systems suffer from a large false-positive error rates, but have a low false-negative error rates. They are also unable to deal with changing network traffic patterns since any behavior that deviates from the norm is seen as an attack on the network.

The placement of the sensors and agents that make up an NID system also characterizes it. Host-based NID systems monitor network traffic from the hosts that make up the network, while also monitoring log files and process behaviors on the host. Network-based NID systems run on the network hardware making up the infrastructure of the network. Host-based NID systems are able to detect a wider range of attacks than network-based systems, but are difficult to install and maintain and are unable to detect attacks involving more than one host on the network. Network-based NID systems are able to survey a large number of hosts at the same time, have low installation and maintenance costs, and are usually less complicated. However, they cannot detect encrypted attacks.

There has been much research in the application of artificial immune systems to building network intrusion detection systems. In most of the research AIS algorithms are used to detect anomalies in network traffic and host behavior, specifically because of their ability to map a class boundary using only samples from one class. This is very useful in anomaly-based NID systems, since they have easy access to normal network and system behavior metric, but not to samples of abnormal behavior. Additionally, almost all research presented in this section is of host-based NID systems. [25]

One of the first papers published on the application of AIS algorithms to NID systems is [26]. In it, a system for detecting abnormal network connections is built using negative selection.

The information used by the algorithm to classify network connections is limited to: the source IP address, destination IP address, and the port number of the connection. When classifying a network connection, the AIS paradigm is followed, meaning that "self" is considered to be safe and "non-self" is considered to be unsafe.

Within the algorithm used in [26] the information used to make a classification is encoded into a string, and r-contiguous string matching is used to perform negative selection. If the detectors survive the negative selection phase, they are used to detect network connections. If a detector is found to be useful in detecting a "non-self" network connection, then its "fitness" score is raised. If a detector has a high enough fitness score then it is promoted to a memory detector and is never deleted. Detectors that have made it through negative detection are subject to deletion if their fitness is not high enough. When an antibody matches a "non-self" (and therefore abnormal) connection in the network, a system administrator is emailed with the details. If the administrator confirms that the connection was abnormal and dangerous, the antibody then becomes a "mature" antibody. If two antibodies match the same connection, the one with the higher match gets its fitness increased.

Experiments were run with the algorithm described in [26] with network data gathered from 50 hosts, limiting the antibody population to 100 antibodies per host. The match length for the r-contiguous string matching was 12, and the experiments ran for 50 days observing 2.3 million connections. There were a total of 78 false-positives per day without human help, and 74 false-positives per day with human help. The AIS algorithm compared well with other NID algorithms of the day. This research does not use any statistical information to attacks to the network, and relies on simple network connection data.

In a series of papers, Kim and Bentley explore the use of negative selection and clonal selection on an AIS classifier, using both network data and several data sets from the UCI repository. When used with network data, their algorithm proved to be infeasible with real-world data, taking too long to train. It was estimated that for an 80% detection rate it would take 1,429 years to train a population of antibodies that is big enough. Additionally, to classify just 20 minutes worth of data, 6×108 detectors would be needed. Although this research was done more than a decade ago, and technology has improved, it would be safe to say that this approach would still be untenable today. [27-30]

Using both negative and positive selection Dasgupta and Gonzalez [31] built and tested an AIS algorithm to detect network intrusions. They use three features of network data: bytes per second, packets per second, ICMP packets per second. This approach is slightly different from [26], since they use calculated statistical features of the network data, and not information from the network data itself. To test their algorithm they use a small subset of the 1999 Lincoln Labs data set, made available for researches to investigate intrusion detection systems. In their results, they concentrate on only five attacks, detecting all five of them. The highest detection rates they found were, 95% accuracy for positive selection and 85% accuracy for negative selection.

Gonzalez and Dasgupta followed their previous publication with [32], which focused on using AIS, but relying on the use of positive samples only. They use hyper-spheres as detectors, being able to do so because all variables were real-valued. After a certain time, detectors die of old age and eventually a good set of detectors is found. They got detection rate of 95% with a very small false alarm rate.

Pioneering research by Kephart in [33] applied an AIS algorithm to the automatic detection of computer viruses and worms. Their algorithm monitors system binaries for changes, as well as monitoring the behavior of important daemons for changes. In related research [34], Forrest, Hofmeyr, Somayaji, and Longstaff develop an AIS algorithm is used to detect virus infections by detecting patterns in system call data. Several common sendmail attacks were detected, as well as some attempted attacks.

In [35], Kotov and Vasilyev propose an algorithm based on immune principles to do unsupervised learning with the goal of detecting network intrusions. The algorithm uses a gene library for detector generation, along with negative selection and clonal selection to train the population of antibodies. The algorithm uses the Hamming distance, along with a threshold value to define the matching rule between antibodies and data. This algorithm is considered to learn in an unsupervised manner because it does not need labeled data to learn normal network behavior. The performance of the algorithm was tested using the DARPA Intrusion Detection Dataset.

An IDS may also be implemented across many hosts which then work cooperatively to solve problems. For example, in [36] He, Yiwen, Tao, and Ting propose a collaborative model for AIS intrusion detection systems. More specifically, the algorithm is used to detect malware across a network through the use of Immune Collaborative Bodies (ICBs), which is a collaborative assembly of AIS algorithms working on different hosts which exchange information between them. In the paper an architecture is proposed to organize the different parts of an AIS algorithm: one element learns new patterns in the environment, another element select mature detectors (antibodies) from the population that have proven to be useful, and a third element exchanges

mature detectors between hosts. A protocol is also included for different hosts to join, collaborate with, and leaver ICBs. No experiments are performed with the new algorithm.

Yeom and Park proposed a collaborative AIS approach in [37]. The AIS algorithm is implemented as a set of mobile agents that work on a wireless ad-hoc network. Because of the ever-changing nature of an ad-hoc network, the features of the AIS approach to network security are very useful, for example: its distributed nature and its lack of a central point of failure. Additionally, autonomous agents are able to function autonomously and asynchronously, adapt dynamically, add robustness and fault tolerance to the system. In the proposed system, any host in the system is able to create agents, but only one host does so. Agents travel from host to host monitoring the system call data originated by only one process, looking for anomalous behavior, as defined by a database containing normal data about normal system behavior. Some experiments were carried out and described, but remained highly theoretical with no results reported.

In [38], Boukerche, et al. propose another mobile agent-based architecture for IDS that is inspired by AIS principles. In this system, the agents monitor the system logs of the host using Logcheck audit tool, and distribute the logs across the network using the syslogs-ng tool. There are four different types of agents: monitoring agents, delivery agents, reacting agents, and persistent agents. If abnormal activity is found, the IDS responds in two possible ways: by sending an email to a network administrator, or by disabling the offending service. The proposed system is extensively tested in real-world situations. In [39] and [40], Ou proposes a multi-agent AIS algorithm for detecting viruses. The algorithm works by gathering data from the system calls made by programs running on hosts. The agents work with a centralized proxy server to run a dendritic cell AIS algorithm that generates a danger signal.

Zeidanloo, Hosseinpour, and Borazjani [41] used an AIS algorithm as a part of an algorithm to detect botnets. The algorithm attempts to cluster hosts based on similar network activity based on two features: average number of bytes per second and average number of bytes per packet. The AIS algorithm is then used to detect two activities commonly carried out by botnets: port-scanning and spamming. The information from the clustering algorithm and the AIS algorithm is then combined by an analyzer to find the subset of hosts that are part of a botnet.

Zhang, et al. [42] apply AIS to accomplish NID on the smart grid, which is the growing field of applying algorithms to the power grid. The AIS algorithm is implemented at three locations: the Home Area Network (HAM), the Neighborhood Area Network (NAN), and the Wide Area Network (WAN). Data is exchanged between HANs at the NANs, and data is exchanged between NANs at the WANs. The AIS algorithms use clonal selection to detect attacks, and is also able to provide extra information about the attacks, such as what type of attack it is.

In [43], Rassam, and Maarof investigate the use of an immune-inspired clustering approach to perform unsupervised learning. The rough set method was used to perform feature selection on the dataset, then an algorithm known as aiNet was used to perform clustering on the DARPA KDD Cup ‘99 NID dataset. The clustering algorithm is based on artificial immune networks and was used to cluster the data into the following types of attacks: denial-of-service attack, probing, user to root attacks, and remote to user attacks. Feature selection is found to improve the performance of the aiNet clustering algorithm, and aiNet is found to give better detection accuracy than the k-Means algorithm.

Wang, He, Xue, and Dong, [44] developed a technique to detect DoS attacks in real time. Real-time detection is achieved by storing the features extracted from network flows in a tree data

structure, and training using negative selection. The algorithm is able to detect many types of DoS attacks without any previous knowledge. In [45], Srivastava and Mukhopadhyay apply the AIS paradigm to detecting anomalous behavior in a VoIP network using an immune-inspired algorithm called MILA. The algorithm gave a false alarm rate as low as 12.5% and a detection rate as high as 100%.

In [46] Le Boudec, and Sarafijanović present an algorithm to detect misbehavior in a mobile ad-hoc network. Mobile ad-hoc networks are especially susceptible to attackers, since the hosts in the network also forward and route traffic. Because of this, misbehavior detection is very important, so that compromised nodes can be discovered and dealt with. This paper deals with detecting misbehavior in networks using Dynamic Source Routing (DSR). The AIS algorithm described in this publication uses negative selection to train a population of antibodies to detect misbehaving nodes in the network. The antigens are string of network operations performed by a host, as detected by other hosts, which includes DSR route discovery and normal data transfer. The proposed algorithm is tested using a network simulator called Glomosim. The algorithm was able to detect misbehaving nodes 50%-60% of the time.

In [47], Drozda, Schaust, and Szczerbicka, apply the principles of AIS to detecting misbehavior in another type of network: wireless sensor networks (WSN). WSNs operate autonomously, have limited computing power, and are battery powered. The authors propose an AIS algorithm that uses r-contiguous string matching and negative selection, training on a dataset consisting of 5 features (called genes in this publication). The tests performed and statistics gathered were specifically modeled to determine whether an AIS algorithm works well in a WSN,

specifically using the random waypoint model. As with [46], Glomosim was used to perform the simulation.

In [48] and [49], Mohamed and Abdullah applied an immune-inspired technique to securing a mobile ad-hoc network (MANET). Their system is based around three types of agents: manager and monitor agents, which are static, and replicate and recover agents, which are mobile. The manager resides in one node and maintains a database of normal system behavior that is gathered by the monitor agents. The algorithm uses negative selection, clonal selection and danger theory to maintain a population of antibodies. No experiments were performed to test the efficacy of this approach.

Liu and Yu [50] work on NID in Wireless Sensor Networks (WSNs). The AIS algorithm proposed used negative selection and clonal selection with r-contiguous bit matching. Tests are performed on the algorithm, and it achieves 100% detection rates across five attack types: route looping, jamming, sinkhole attack, wormholes, and black holes.

Yang, Guo, and Deng, [51] use an AIS algorithm to detect illegal operations in RFID networks. They use a simple negative selection algorithm to train the population of antibodies to classify activity on the RFID network as “self” and “non-self.” Finally, an AIS clustering algorithm is used to cluster the “non-self” activity that is detected, and these clusters are used to build a danger signal. They were able to achieve a 98% recognition rate.

The research reviewed in this section shows that AIS algorithms are very promising as classifiers because they can evolve and deal with changing conditions. AIS algorithms have the potential to be a powerful approach to network intrusion detection. Their wide application to networking problems is useful for the current research. However, a common theme in the early

literature of NID system using AIS algorithms is the inability of AIS algorithms to scale to real-world problems. An exposition of research meant to address this shortcoming is shown in a future section. Being aware of these shortcomings, inspiration was drawn from the performance of AIS algorithms and their applications to network problems to use them in a new application area. However, the common negative selection and clonal selection algorithms were not to train the current classifier.

Network Flow Classification

Network flows are sets of packets which are defined by a common source IP address, destination IP address, source port number, destination port number, and protocol. A network flow can be thought of as a sequence of packets from a source to a destination on a network. A network flow is one of many ways to split up network traffic, and is useful in many situations. A network flow can be categorized in a few different ways, a few of which are explained in this section. All of the approaches featured in this section use the statistical features of the network traffic to perform classification.

One way to categorize network flows is by the application type that generated the flow. This section describes the work that has been done in applying Machine Learning algorithms to the classification of network flows by application.

Moore and Zuev tested the Naïve Bayes classifier in [52], applying it to a data set of network flows. The simplest Naïve Bayes classifier did not do very well on the full set of features of the dataset. However, the accuracy rose substantially when Fast Correlation-Based Filter (FCFB) feature reduction was performed on the data set, as well as when kernel density estimation

was completed. The methods were tested together and separately, and the highest accuracy was achieved when FCFB and kernel density estimation were used together, achieving 96.3% accuracy.

A review of the state of network traffic classification is presented in [53]. The paper includes an evaluation of the machine learning algorithms used, elephant and mice flows, and early classification of network flows. Information concerning the processing time, memory, and directional neutrality of all algorithms presented is also included in the paper.

A review of a few different Machine Learning Algorithms is presented in [54]. The algorithms tested were: Support Vector Machines, C4.5, RIPPER, and Naïve Bayes classifiers. The algorithms were tested on three publicly available data sets, and were also tested on encrypted network traffic. Additionally, the authors of [55] tested many of the same algorithms as [54]. The algorithms were: Multi-layer perceptrons, C4.5 trees, Bayes Net, Naïve Bayes, and Radial Basis Function Neural Networks. The best performing algorithm tested in [55] was the C4.5 algorithms. The authors of [56] also test learning tree classification algorithms on network flow classification, but focusing on the accurate classification of P2P traffic. They achieve a maximum accuracy of 97%.

The selection of the statistical features that are most useful in classifying flows is also an important area of research. The authors of [57] give a survey of the reasons why some algorithms perform well on the flow classification problem, and the features that are most important. They show that there are most useful are: ports, the sizes of the first one or two packets for UDP flows, and the sizes of the first four or five packets for TCP flows.

Early classification of network flows is done by limiting the number of packets that need to be examined before a flow is classified. The authors of [58] show that it is possible to achieve

good accuracy with information gleaned from only the first 7 packets in a flow, supporting the findings of [57]. They use a one-against-all SVM classifier. The last paper is [59], which contains a survey of many Machine Learning techniques, including the removal of outliers from the data set, dimensionality reduction, and data normalization. After this, decision trees, Naïve Bayes, and Bagging and Boosting classifiers are tested.

In [60], the authors use Extreme Learning Machines (ELM) to tackle the supervised learning network traffic classification problem. ELMs are like artificial neural networks, however, ELMs use randomized computational nodes in the hidden layer and generate their weights by solving linear equations. A similar approach is taken in [61], although the ELMs used are kernel based. In this study, over 95% accuracy was achieved using different activation functions.

In [62], the same problem is undertaken using an original approach that fuses Hidden Naïve Bayes and K* classifiers. Feature selection is done using Correlation Based feature selection and Minimum Description Length. In [63], the researchers build an anomaly detection system using machine learning techniques. The system is meant to detect anomalies within the traffic in a cellular network and it is built using Random Neural Networks. The approach is tested on synthetically generated data.

The research in [64] seeks to identify traffic flows generated by a mobile messaging app called WeChat. To achieve this, 50 features were extracted from every traffic flow within two data sets. Several different classification approached were used, including: SVM, C4.5, Bayes Net and Naive Byes are applied to classify the WeChat text messages traffic. Very high accuracy was achieved in both data sets. The research contained in [65] seeks to solve the traffic classification problem in the same way as other research presented in this section. They use SVM classifiers to

classify flows into two categories: Video and Other. The researchers were able to achieve accuracy of 90% and above.

CHAPTER IV

LITERATURE REVIEW

In this chapter, a literature review for three research topics will be presented. The first section of this chapter deals with previous work in the extension of AIS classifiers into the multi-class classification problem domain. The second section with previous research in positive selection AIS classifiers. The third section shows all previous work in the optimization of the Negative Selection algorithm.

Multi-Class Artificial Immune System Classifiers

Natural immune systems work as binary classifiers, categorizing every cell as either "self" or "non-self." Consequently, artificial immune systems classifiers are limited to binary classification as well. There has been some research to expand the ideas present in AIS classifiers into multi-class classification. This section presents all of the previous research found.

The earliest attempt is shown in [66], by Goodman, Boggess, and Watkins. They design and test an algorithm they call the Artificial Immune Recognition System (AIRS). They also test it against Kohonen's Learning Vector Quantization algorithm (LVQ). Their algorithm works similarly to the k Nearest Neighbors algorithm (kNN), in that it trains a population of antibodies to classify examples by matching to the nearest antibody, however, the training set is not directly used to perform classifications. However, the AIRS algorithm is more efficient, since it requires on average half of the comparisons to perform classification as kNN. It also does not require perfectly tuned parameters to perform well. Watkins expanded on the concept of resource-limited AIS algorithms in his master's thesis [69], and published two further papers on the subject [70-71] with Boggess. The algorithm described in [70] and [71] also works with multi-class classification.

The algorithm works in several stages to train a population of data points called artificial recognition balls or ARBs that are then used for classification.

Resource-limited algorithms seek to minimize the amount of time and memory taken to perform a task. Timmis and Neal take this into account and design a multi-class AIS classification algorithm and present it in [67] and [68]. A similar algorithm is developed by Cheng and Cheng in [72], in which apply their algorithm to the diagnosis of thyroid diseases. They implement an AIS algorithm which uses the clonal selection principle, forcing the population to compete for resources. They also implement multi-class classification by using an all-against-all classifier. Their classifier achieves 99.87% accuracy.

Greensmith and Cayzer also propose an algorithm similar to AIRS and show how to apply it to Internet document classification, although they do not show any results [73]. Carter, in [74], combines several lines of research into one algorithm, which he calls Immunos. The algorithm does not require antibodies to be of the same length, and performs well against other Machine Learning algorithms.

White and Garret propose a new way of using the clonal selection principle in [75]. They use it to recognize patterns, designing a new algorithm to do so, which they call CLONCLAS. They test their algorithm by recognizing binary character patterns, achieving good accuracy but taking a long time to train the classifier. In a similar approach, Brownlee presents a multi-class AIS classifier in [76] called CSCA.

Markowska-Kaczmar and Kordas use the Negative Selection algorithm in [77] and [78] with a one-against-all scheme to create a classifier that is able to do multi-class classification. Their algorithm works with each class in the data set individually, developing a sub-population of

antibodies for each class. An example is classified by exposing it to all of the antibodies, and the class is selected based on which sub population of antibodies matched the examples the least number of times.

The idea of performing multi-class classification with AIS algorithms is not new. However, all of the previously explored approaches are very complicated, and do not lend themselves to resource-constrained systems. This is because almost all of the approaches taken in previous research to do multi-class classification with AIS algorithms use an iterative optimization approach, using Clonal Selection. Clonal Selection is able to achieve high accuracy, however, they require a lot of time to achieve these results. As shown in previous sections, there is a large amount of research into the application of AIS algorithms to networking problems.

A problem that appears more than once in the literature is the inability of AIS algorithms to be useful in real-world applications. Specifically, in [27-30], it can be seen that generating the necessary number of antibodies will take a too long for the algorithm to be useful. One of the reasons behind this weakness is the fact that Negative Selection algorithms map the negative space of a class, the negative space being simply the feature space that is outside of the class boundaries of a particular class. Furthermore, the negative space is mapped using randomly generated detectors.

Because of the reasons detailed above, a simpler algorithm was developed by us for classifying network flows. The algorithm described in this work seeks to use the strengths of AIS algorithms, while making them usable in real world scenario. To this end, the negative selection algorithm is replaced with the positive selection algorithm. Also, the training algorithm longer generates detectors randomly, which takes a long time. Instead, it leverages samples taken directly

from the data set. These tradeoffs might decrease the accuracy of the algorithm, but they allow the algorithm to be fast enough to be applied to real-world problems. No other research that applies AIS algorithms to network flow classification by application has been found. This research marks the first time this has been done.

Positive Selection Artificial Immune System Classifiers

The use of AIS algorithms on multi-class problems has been explored in previous publications. The research made use of antibodies defined as hyper-spheres in the feature space, in the same way as the current research. However, there have been problems found in the use of hyper spheres in high-dimensional spaces [79]. Furthermore, hyper spheres have been used because of their simplicity of representation, and small memory requirements. There have been other representations proposed to replace hyper spheres [80-82]. Hyper spheres are a natural fit when the feature space contains data that is real-valued or ordinal. The authors of [83] provide a good review of the work that has been done up to 2007.

The authors of [84] use the positive selection algorithm to build an algorithm for change detection and also compare it to a negative selection algorithm. The algorithm is a simple reversal of the positive selection algorithm, where detectors are generated randomly, and selected to join the population of detectors if they match sample of the “self” set. The algorithm shows improved change detection in some situations, when compared with negative selection. The authors suggest that a hybrid of negative and positive selection may be useful.

The author of [85] demonstrates a hybrid algorithm, combining negative and positive selection approaches. The algorithm is used to detect anomalous behavior. The algorithm uses positive and negative selection in different stages to create a population of antibodies. The final

goal of the algorithm is to decrease the false positive rate in an anomaly detection algorithm, however the algorithm is not tested and no real-world performance measures are given.

The work found in [86] uses positive selection classification as well as hyper-sphere detectors and is based on the work in [84]. However, the population of antibodies is not trained using positive selection, but with the clonal selection algorithm. The authors use their classifier as a binary classifier and achieve multi-class classification using a one-against-all scheme. Their algorithm is tested on an anomaly detection task, being used to detect malware infection by examining API call traces and kernel mode callbacks. The algorithm was able to outperform all other algorithms that it was tested against on this task. The same authors further tested the same algorithm in [87], and found that it outperformed all other algorithms in the UCI Diabetes dataset, with 79.9% accuracy, also achieving 96.7% accuracy on the UCI Iris data set.

In [88], Yu and Dasgupta develop an AIS algorithm with a novel detector generation scheme and test its performance on a NID dataset. The detectors are hyperspheres with a variable radius, and the placement of any new detectors is only possible within spaces in the hyperspace that are not already within any detectors. This new detector generation scheme is near-deterministic and not completely random, unlike previous schemes. This new scheme is able to generate a detector population with an optimal distribution, and to make the algorithm much more efficient. The proposed algorithm is tested on the DARPA Intrusion Detection Dataset, and compared with a single-class SVM, showing promising results. Although this research is similar to the research presented in this work in its use of hyper-spheres as well as its optimizations, it is still differs substantially in many other respects.

In all of the research discussed in this section there has been no work in the optimization of positive selection algorithms. The way in which the training algorithm was developed in the previous research makes it very easy to optimize. Furthermore, the algorithm already does not require many parameters, making it an excellent choice for embedded computers. There are some optimizations for the Negative Selection algorithm as detailed in the next section, but they are not applicable to this work. The original contribution of this research will be to optimize the execution of the algorithm already developed, as well as maximizing the accuracy of the algorithm. Figure 7 shows the position of the current research in relation to other research described in this section.

Optimizations for the AIS Negative Selection Algorithm

Since the invention of the Negative Selection algorithm, its usefulness for real world applications has been questioned. Indeed, the algorithm is very slow when compared with other machine learning algorithms. Nonetheless, the algorithm has proven to be useful when training on data sets that only have examples of one class, essentially mapping the "positive space" of the missing class by mapping the "negative space" of the class that is present in the data set. For reasons of its usefulness in some contexts, there has been some research into optimizing the algorithm by speeding up the most common operation of the algorithm, the comparison between detectors and examples (strings).

The first research into optimizing the Negative Selection algorithm is by Elberfield and Textor in [89]. Their research only applies to NS algorithms that work on strings, and use r-contiguous and r-chunk string detectors. They show how compressing the population of antibodies can lead to faster training and classification performance. The compression scheme stores all common prefixes of antibodies in one place. The set of prefixes is used to make comparisons

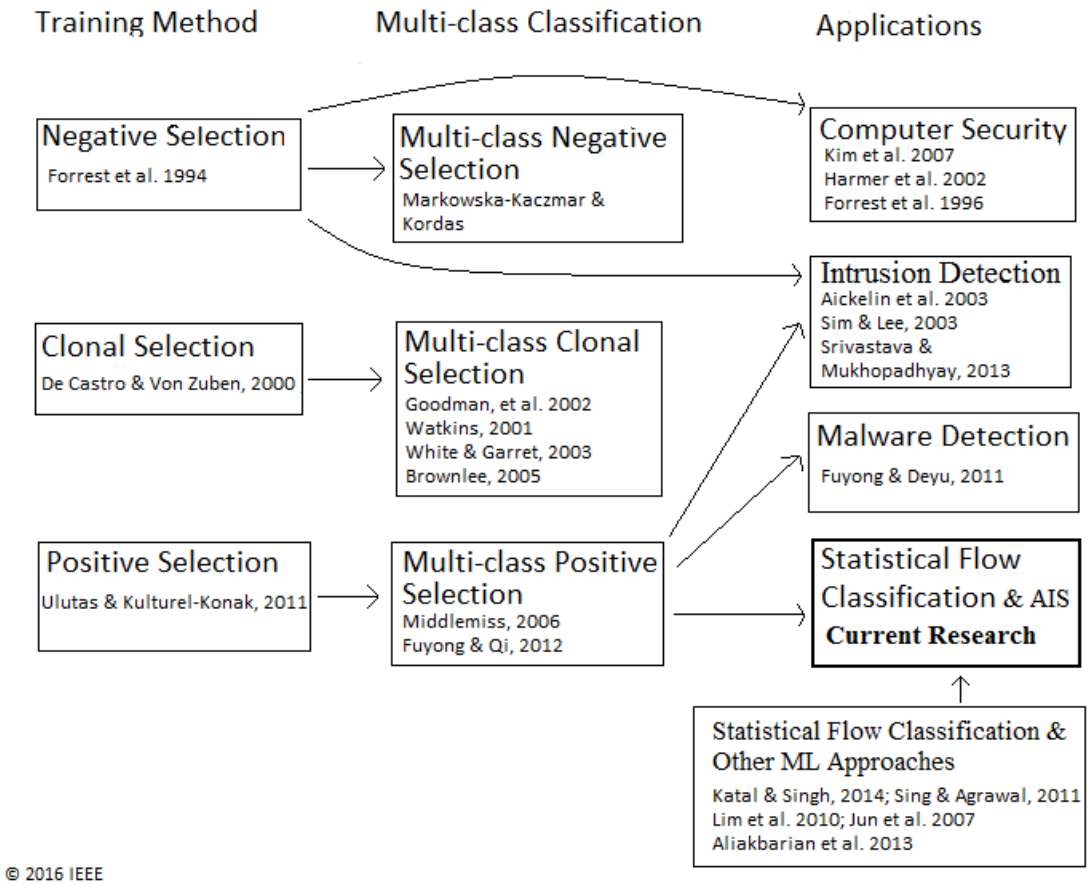

Figure 7. Current Research Position

between antibodies and strings, making both the training and classification much faster. The compression technique makes the complexity polynomial, where it was exponential before. This research demonstrates that it is not necessary to store antibodies in full, and it is not necessary to perform all comparisons when training an antibody.

Following the lead of [89], the same authors continue their research in [90]. However, the use a very different approach to the optimization of the Negative Selection algorithm. Whereas [89] showed how the antibody population could be compressed, [90] shows how the actions of the NS algorithm could be simulated without actually generating an antibody population at all. The

authors illustrate how this is possible using only r-contiguous and r-chunk detectors. The paper gives proofs of the feasibility of the proposed approach only, and does not implement or test it. As with [89], the complexity of the NS algorithm goes from exponential to polynomial with this approach.

The authors of [89] and [90] continue their research with [91]. In this paper, the authors prove that it is possible to train an automaton to simulate the NS algorithm, and to do so in polynomial time. The automaton is then used for classification. This paper uses prefix trees, as opposed to [89].

The last work by Textor on the subject of optimizing the execution of negative selection algorithms is [92], where one further variation from the previous approaches is explored. In the publication, detectors are generated by sampling from the set of S-consistent detectors. S-consistent detectors are defined to be the set of detectors that do not match any element in the S set, which is the set of self samples. Whereas the NSA algorithm normally samples the space of possible detectors uniformly, the author shows that by doing this differently, it is possible to speed up the execution of NSAs. This paper theoretically proves that it possible to speed up the execution of NSA by using probabilistic sampling techniques such as Markov Chain Monte Carlo methods.

In [93] and [94], Want, Yibo, and Dong also propose a method to do optimization for NS. Their work divides the feature space into “neighborhoods” for both antibodies and samples to be classified. By doing this the number of comparisons that need to be performed is reduced. They also introduce a method to improve the matching operation between detectors and samples that improves the performance, especially in high dimensions.

Both [95] and [96] show a similar approach to the optimization of NS as [93] and [94]. By separating the feature space into regions, called “grid cells,” the antibodies can be compared only with points within their own grid space. This approach to comparisons also eliminates exponential time complexity, speeding up the execution of the algorithm. The authors call the algorithm GF-RNSA, and also report experimental results.

Lastly, a novel antibody generation method is proposed in [97]. The method was developed by Ji and Dasgupta and is called V-detector. The strategy involves the statistical analysis of the data in order to improve the amount of non-self space that is covered while also minimizing the number of antibodies needed to do so. The antibody generation process also takes into account the boundary of the classes in the data set. The antibodies are allowed to be of variable size, as well. These techniques allow the algorithm to be very efficient. The scheme is also applicable across many different antibody types.

CHAPTER V

RESULTS

This section explains some of the work that is already completed. Some of the work described in this section is published and other work will be submitted at a later date.

Multi-Class AIS Applied to Classifying Network Flows

This section describes our first published work in this area, including a description of the data set used throughout our research.

The problem of traffic flow classification has become harder and harder in recent years as application designers take steps to hide the activities of their programs on the network. On the other hand, traffic classification has increased in importance as well because of the need of network administrators to stop the use of illegal applications on their networks. Because of all of the previous work in the use of AIS algorithms with networking problems, particularly to the classification of network connections, the problem of network traffic classification is particularly attractive to the application of an AIS classification algorithm. The material in this section shows an AIS-inspired classification algorithm applied to a network flow classification data set. The use of kernel functions with the algorithm is also tested. The algorithm is tested on the data set and several performance measures are displayed in figures. The work described in this section first appears in [98-99].

Description of the Data Set

The data set used to test the algorithm was first used in [52] in research by Moore and Zuev. The researchers have made the data set available to other researchers. The data set contains about

370,000 samples, with each sample having 249 features. Each sample represents one traffic flow on a real network, with the application type that generated the data carried over the flow. Since the current research aims to classify network flows according to the type of application that generated the data in the flow, this data set is ideal. A network flow is a sequence of packets, traveling over a network from one host to another. The flow is defined by a 5-tuple, which contains the source and destination IP addresses, the source and destination ports, and the transport protocol used.

We make use of the feature reduction process performed in [52], taking advantage of the previous research. The analysis is important to deal with the curse of dimensionality, a well-known problem in machine learning, because of which the processing time for a ML algorithm rises exponentially as the number of feature rises. Furthermore, not all features are useful in making predictions, and their inclusion may be detrimental to the accuracy of the classifier. The original 249 features are analyzed and 11 features are chosen to be used in the tests. The feature reduction process used in [52] is Fast Correlation-Based Filter. The names and descriptions of the 11 features used are listed in Table 1.

The data set assigns each sample to an application class, which may contain many individual applications within it. There are a total of 12 application classes in the data set, which are listed in Table 2. The FTP application class listed in the table is separated into three different class labels within the data set, encompassing control, passive and data FTP flows, respectively. Tables 1 and 2 originally appear in [52].

Table 1. Features and Descriptions

Feature	Description
Port, server	Port Number at server
Number of pushed data packets, server→client	# of packets with the PUSH bit set in the TCP header
Initial window bytes, client→server	# of bytes in the initial window
Initial window bytes, server→client	# of bytes in the initial window
Average segment size, server→client	The average segment size
IP data bytes median, client→server	Median of total bytes in IP packets
Actual data packets, client→server	# of packets with at least a byte of TCP data payload
Data bytes in the wire variance, server→client	Variance of # of bytes in Ethernet packet
Minimum segment size, client→server	The minimum segment size
RTT samples, client→server	The total number of Round Trip Time (RTT) samples.
Pushed data packets, client→server	# of packets with the PUSH bit set in the TCP header

Table 2. Class Labels and Applications

Class Label	Applications
FTP-CONTROL, FTP-PASV, FTP-DATA	FTP
DATABASE	Postgres, Sqlnet, Oracle
INTERACTIVE	SSH,rlogin, telnet
MAIL	IMAP, POP2/3, SMTP
SERVICES	X11, DNS, ident, LDAP, NTP
WWW	WWW
P2P	KaZaA, BitTorrent
ATTACK	Worm and virus attacks
GAMES	Half-Life
MULTIMEDIA	Windows Media Player

Description of the Algorithm

The algorithm described in this section is not the final version of the algorithm proposed, and has several weaknesses that were dealt with as the research progressed. It is the algorithm that is used to generate the results described in this section, so it is described briefly here.

As described in the previous section, the algorithm uses features that are calculated from the data flowing over the network. All of the features used in this research are real-valued and it is assumed that the features are calculated for each flow before the algorithm is executed. The set of classes found in the data set is given to the algorithm. The pseudo code for the first version of the training algorithm is in Figures 8 and 9.

As described in previous sections the algorithm is inspired by mechanisms found in the natural immune system. As such the algorithm is an ensemble classifier, with each base classifier being a hyper-sphere. Each hyper-sphere is defined in a space having the same number of dimensions as the data set, in this case 11 dimensions. Each hyper-sphere is also defined with a real-valued radius, as well as a class label which denotes to which class the antibody belongs to. Before the training algorithm is run the data set is normalized to the range [0, 1] using a simple formula:

$$X_i = \frac{X_i}{\max(X_i) - \min(X_i)} \tag{10}$$

where X_i is the value of the vector X in the i*th* dimension, and the data set is made up of real-valued vectors. The functions max() and min() are used to denote the maximum and minimum value in each dimension in the data set. This normalization is meant to simplify

```
input:
    training_set: set of training vectors, class label is the first element of the vectors
    classes: the set of class labels collected from the data set
    population_size: a parameter, the desired size of the antibody population
    step_size: a parameter given to the algorithm
output:
    population: the set of antibodies used to classify flows
functions:
    error_count: returns the number of misclassifications performed by an antibody

Initialization:
    training set = normalize(training_set)
    population = {}
    FOREACH { c | c ∈ classes }
        class_data = { i | i ∈ training_set, i["class"] = c }
        counter = 0
        WHILE counter < ⌈population_size / |classes|⌉
            new_antibody=["center"=random(class data), "radius"=0.0, "class"=c]
            population = population ∪ new_antibody
            counter = counter + 1
        ENDWHILE
    ENDFOREACH
Training:
    FOREACH {p | p ∈ population}
        changed = True
        WHILE changed
            IF error_count(p) > 0
                p["radius"] = p["radius"] - step_size
                changed = False
            ELSE
                p[radius] = p["radius"] + step_size
                changed = True
        ENDWHILE
    ENDFOREACH
```


Figure 8. Pseudo Code for Initialization and Training Algorithm

```
input:
    classes: the set of class labels collected from the data set
    pattern: the pattern to be classified
    population: the set of antibodies used to classify flows
output:
    class label: the class that example is predicted to be in

Classification:
    distances = {}
    FOREACH { p | p ∈ population }
        d = distance(pattern, p)
        IF d <= p[radius]:
            return p[class]
        ELSE
            distances = distances ∪ {(p[class], d)}
    ENDFOREACH
    a = argmin distance(pattern, a) {a | a ∈ population}
    return a[class]
```


Figure 9. Pseudo Code for Classification Algorithm

the code, allowing the radius of each hyper-sphere to be defined once instead of for each dimension. In the pseudo code, p["center"] defines the center point, p["radius"] defines the radius, and p["class"] defines the class that the hypersphere belongs to, all attributes of hypersphere p. In the same way, i["center"] defines the center point, and i["class"] defines the class that the point belongs to, all attributes of point i.

The training algorithm takes two parameters: the size of the population of antibodies to be generated, and the step size. The size of the population is simply the number of hyper-spheres used to make the ensemble classifier. The step size parameter will be explained below. Since each hyper-sphere exists within a space with the same number of dimensions as the data set, the data set can be used to initialize the antibody centers. Indeed, to initialize the population of antibodies the training data set is randomly sampled with replacement, and each new hyper-sphere added is

centered on a sample. The radius of each new hyper-sphere is initialized to zero, with the new antibody having the same class label as the sample used to create it. Each class is allocated an equal portion of the antibody population.

Once the population has been initialized the training algorithm proceeds to set the radius of each hyper-sphere. To set the radius of one hyper-sphere, the radius is iteratively increased, increasing by the step size parameter for each iteration. After each increase in the radius, the antibody is checked against the whole training set. If hyper-sphere contains a data point that is not of its own class, then the radius is decreased by the step size. This guarantees that the radius of each antibody is a multiple of the step size parameter. Within the pseudo code in Figure 8, the antibody is checked against the whole training data set using the error_count function, which returns the number of misclassified data points for the antibody under examination.

The output of the training algorithm is the set of antibodies which is used by the classification algorithm. During the classification phase, the captured features of the network traffic flow that needs to be classified is matched against the population of antibodies. To perform classification, the set of antibodies is compared, one by one, to the point in feature space that represents the flow to be classified. The class of the first antibody that is found that contains the flow is returned as the class label of the flow. If no antibody is found that contains the flow, then k-NN is performed with the set of antibody centers, with k being equal to 1. If the k-NN algorithm is unable to classify a pattern because it is the same distance from more than one antibody, then one of the classes that the antibodies belong to is chosen at random. The pseudo code for the first version of the classification algorithm is in Figure 9.

The tests described in this section of the dissertation were all performed with the step size parameter set to 0.01, this value was found to be a good compromise between training time and accuracy. As mentioned, the k parameter is set to 1, and is not changeable in this version of the algorithm. In this section the algorithm is tested with the Euclidian distance function. Kernel functions are also implemented. Kernel functions are used in other classifiers to deal with data that is not linearly separable into classes. Kernel functions are able to deal with this problem by projecting the data into higher dimensional spaces, were it can be more easily separated. Kernel functions are used extensively with SVM classifiers.

Results

The tests performed to show the performance of the algorithm all follow standard Machine Learning practices. For every test in this section, the data set was split up into three sub data sets, the testing set, validation set, and training set; with each taking 10%/10%/80% of the data set, no matter the size. All of the tests performed make use of stratified cross validation. All tests were performed on an Intel Core i5 running at 1.8 GHz with 4 GB of memory.

The algorithm is tested using Euclidian distance, and this is used to provide a baseline for the performance of the algorithm. However, a technique borrowed from SVM classifiers is also tested here, in an attempt to improve the accuracy of the algorithm. Kernel functions are used to improve a classifier's accuracy by projecting the vectors that make up the data set into higher-dimensional spaces. In doing so, kernel functions make it easier to find and define the class boundaries.

To test the accuracy achieved by the algorithm on the network flow data a set of tests were performed. The results of the first test are graphed in Figure 10. The independent variable is the

number of antibodies used by the classification algorithm, with the data set size held constant at 1000 examples. Since the size of the antibody population is the independent variable, it is increased by quantities of 50, starting at 200 and ending at 1000.

The accuracy achieved by the algorithm increased from 81.8% with 200 antibodies, to 91% with 1000 antibodies. The best performance on this test was achieved with the linear kernel, with 92.3% accuracy. Figure 10 also includes data about the performance of the Naïve Bayes and SVM classification algorithms. The maximum accuracy of the Naïve Bayes algorithm is 82.2%, and the maximum accuracy of the SVM classifier is 44.1%. The Naïve Bayes and SVM algorithms are trained and tested with the same data sets as the AIS algorithm, ensuring a fair comparison, although they do not use an antibody population and are therefore not affected by the independent variable.

The second test that was performed to measure the accuracy achieved by the algorithm is graphed in Figure 11. The dependent variable is the data set size, and it is increased in increments of 50, from 2000 to 1000 elements. The antibody population is held constant at 1000. The independent variable is the accuracy achieved by the classifiers. The accuracy of the algorithm did not rise as dramatically as in Figure 10, but it did go up from 86% to 92.3%. The best performance was achieved by the polynomial kernel, with 93.6% accuracy. As with the previous test, the SVM and Naïve Bayes classifiers were also trained and tested with the same data set, achieving a maximum accuracy of 42.6%, and 83.5%, respectively. Figure 11 shows the AIS algorithm having a higher accuracy than the Naïve Bayes and SVM classifiers. This highlights the AIS algorithm's ability of generalizing from small data sets, outperforming the more well-known algorithms by a noticeable margin. Although the SVM algorithms have been proven to achieve very high

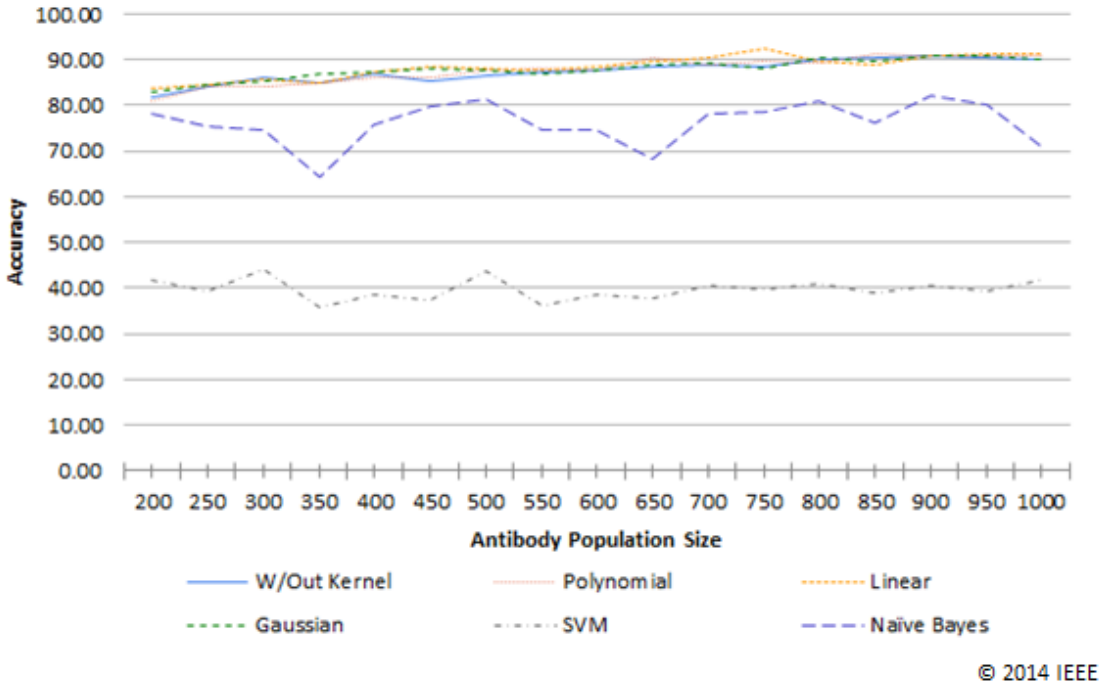

Figure 10. Classification Accuracy and Antibody Population

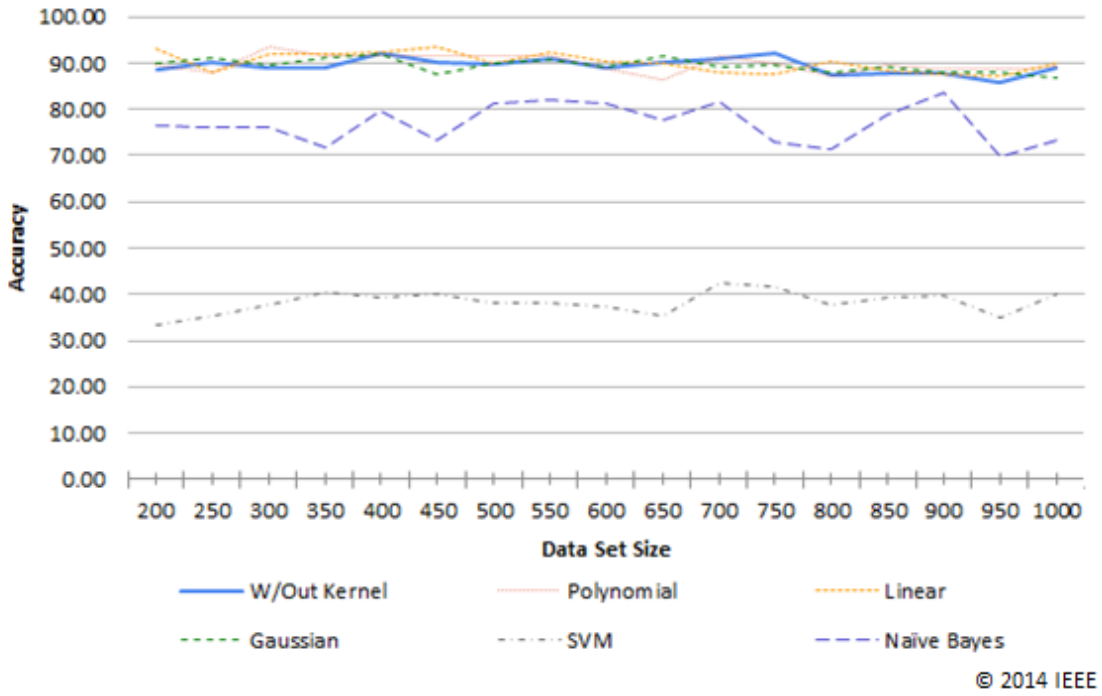

Figure 11. Classification Accuracy and Data Set Size

accuracies, they typically require more training data to do so.

Figs. 10 and 11 suggest the fact that using kernels does not improve the performance of the algorithm significantly. Although the maximum accuracy is achieved with kernel functions, it is not obvious that the AIS algorithm working with kernel functions outperforms the AIS algorithm without them. This observation is particularly important when utilizing the AIS algorithm in memory constrained embedded systems, as in the case of the Internet of Things (IoT). On such systems, it is not critical to employ kernel functions in conjunction with the immune system inspired classification algorithm to achieve high accuracy

The next set of experiments were designed to test the running time of the training and classification portions of the AIS algorithm. Figure 12 shows the amount of time required to train the population of antibodies. The dependent variable is the size of the data set, the independent variable is the training time is seconds, and the size of the population of antibodies is held constant at 1000. The size of the data set used to perform training is increased from 200 to 1000 in increments of 50, as mentioned previously the size of the training set is 80% of that. The relationship is roughly linear, meaning that the size of the data set is linearly related to the running time of the algorithm. As with previous tests, the SVM and Naïve Bayes algorithms were also tested with the same data sets. Their running times were much faster and can be seen along the bottom of the figure. The kernel functions used with the training algorithm do not affect the running time in a marked way. Although the relationship between the size of the antibody population and the training time is not tested, it is expected that the relationship will be roughly linear as well.

The running time of the classification algorithm is also tested. Figure 13 shows the amount of time required to classify 100 examples from the testing set. The amount of examples is 100

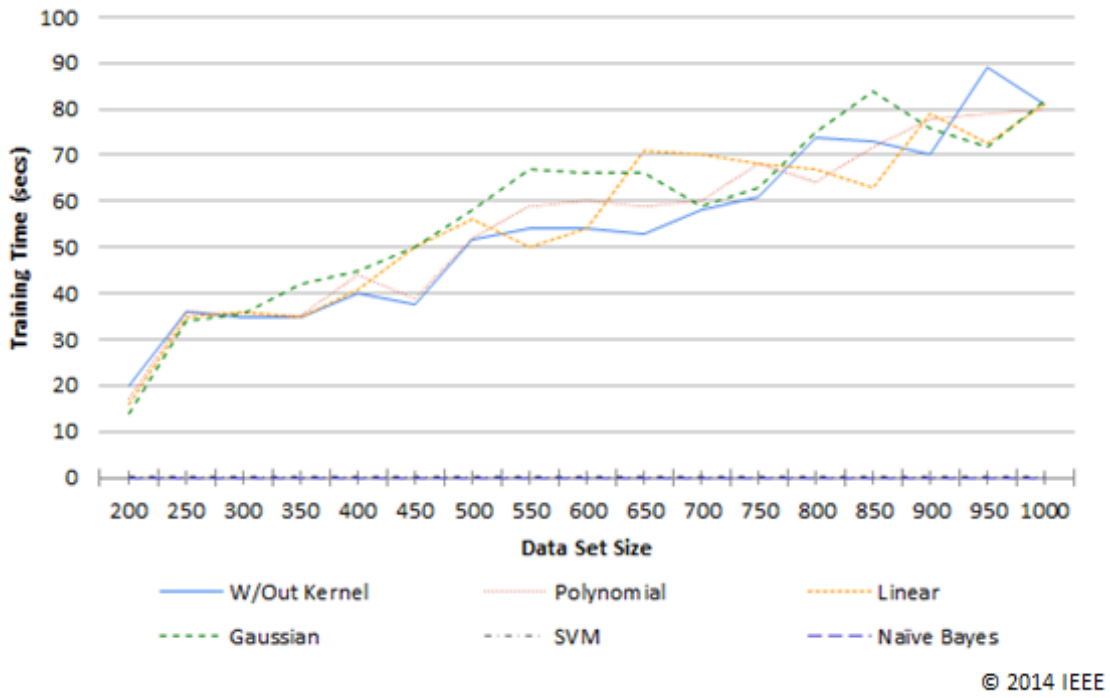

Figure 12. Training Time and Data Set Size

because the testing set is 10% of the data set, which is held at 1000 for this test. The dependent variable is the size of the population of antibodies, the independent variable is the classification time in seconds. The dependent variable is increased from 200 to 1000 in increments of 50. It can be seen the classification time is roughly linearly related to the size of the population of antibodies. The use of kernel functions does not affect the running time of the classification in a discernible way. The SVM and Naïve Bayes algorithms are tested on the same data set, their running times prove to be much faster, seen along the bottom of the figure. However, these classifiers do not use a population of antibodies to perform classification, and are unaffected by the dependent variable, but they are included for completeness.

When the accuracy of the AIS classifier is compared to results found in [6], an interesting contrast is seen: the AIS algorithm is able to achieve greater or equal accuracy than all other

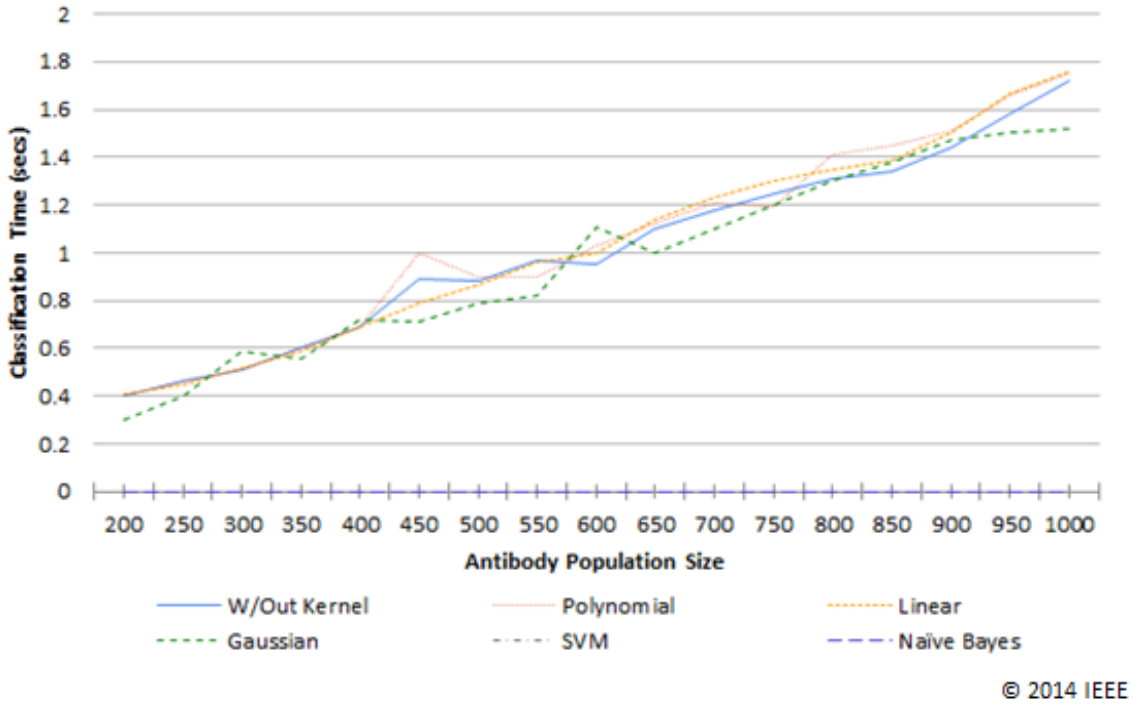

Figure 13. Classification Time and Antibody Population Size

classifiers with 1/3 of the training examples. However, this comparison is not stringent as different data sets are used. Although the current proposed algorithm classifier does not exceed the classification accuracy of the best classifiers, it is very good at generalizing from small training sets. This conclusion is supported by Figs. 10 and 11.

The maximum accuracy achieved by the AIS algorithm was 93.66%, achieved with the polynomial kernel. The AIS algorithm was able to achieve 91.75% accuracy without using kernels. The algorithm is not able to match the accuracy achieved by other algorithms in the literature. However, the AIS algorithm proved to be able to get high accuracy with small sample sizes. The simple logic of the algorithm and its insensitivity to the use of kernels functions make it useful for resource-constrained systems, where these features are important.

Comparison to Previous Work

The work found in [87] is most similar to the current research, thus their work is compared and contrasted to the current algorithm in this section. Both classifiers are designed to perform positive selection, with the antibody population mapping the positive self-space of each class. To achieve this, both algorithms attach a class label to each antibody in the population, while also using hyper-sphere antibodies. Both classifiers also use k-nearest neighbors with the hyper-sphere centers as a fall back when the hyper-spheres themselves fail to classify a novel example. Unlike the current algorithm, the classifier described in [87] does not allow hyper-spheres to overlap each other, incurring an extra processing cost to ensure this condition is met. The current AIS-inspired classifier does allow hyper-spheres to overlap, greatly simplifying the training process.

Since the SVM algorithm is one of the best classifiers for almost any task, it is also compared here to the current algorithm. Simple SVM classification relies on classes present in the data set to be linearly separable, meaning that a line, plane, or hyper plane can be drawn that separates the space into distinct regions. A classic example of this is the XOR function, which is shown in Figure 14. In the figure, class A and class B cannot be separated with a single line. This problem is traditionally solved with the use of kernel functions. A kernel function projects the vectors in feature space into a higher-dimensional kernel space, within which the classes can be linearly separated. There are a few different kernel functions, each of which takes one or more parameters, which are usually determined by using a grid search on the validation set.

The current algorithm is able to handle non-linearly separated classes within the data set easily, and without the use of kernel functions. Kernel functions are used in the tests discussed in this section, and it can be seen that their use did not clearly improve the performance of the AIS

algorithm. More importantly, since the current algorithm does not require kernel functions, the necessary parameters for them do not need to be determined, saving time during the training step.

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

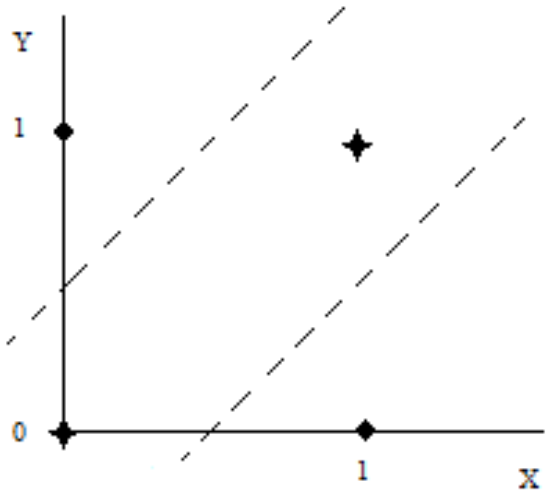

Figure 14. The XOR Function

The SVM algorithm, like other AIS algorithms, is a binary classification algorithm. To be able to perform multi-class classification, more than one SVM classifier must be trained. With the on-against-all technique trains n classifiers, where n is the number of classes in the data set, each classifier being trained to distinguish between one class and all other classes. The all-against-all technique requires (n*(n-1))/2 classifiers to be trained, where n is the number of classes present in the data set, and each pair of classes is distinguished by one classifier. For the same reasons as the training algorithm, classification also takes a lot of time, since each classifier must be run on the sample to be classified. The current AIS-inspired classifier is built from the start to handle multi-class classification.

Determining the optimal location of the hyper plane that defines an SVM classifier is done using Lagrange multipliers, and is a quadratic programming optimization problem. The

optimization problem can be solved using well-known quadratic programming techniques that are used to find the maximum or minimum of a quadratic function that is limited by one or more linear constraints. The training portion of the current classification algorithm does not require any optimization techniques. It also does not require many parameters to run. The algorithm is simple and does not require complicated mathematics to understand.

Kernel functions have been used with AIS algorithms in previous work [100]. However, this research does not implement a Negative or Positive Selection algorithm using kernel functions, but instead uses the aiNet algorithm.

In conclusion, the classifier is simpler and requires less processing time, and requires fewer parameters. Additionally, the tests performed in this section have shown that the AIS classifier does not require a lot of training data to generalize well.

Optimizing an Artificial Immune System Internet Flow Classification Algorithm

This section further develops the algorithm proposed in the previous section, seeking to make the algorithm viable for real-world application. To accomplish this, the training and classification portions of the algorithm are optimized, and different techniques for improving the accuracy of the algorithm are tested. The work described in this section first appears in [101].

Changes to the Algorithm

This section gives details on the optimizations designed for the original algorithm proposed in the previous section. The optimization done to the training algorithm is now described. To speed up the training process, k-d trees are used, making the search for nearest point much faster. K-d trees are a well-known data structure which organize a set of points in a binary tree, which then

allows the nearest point to be found in log n time, instead of linear time. The optimized training algorithm generates antibodies by sampling the training set, and creating a hyper-sphere center from the sample. The algorithm then set the radius of the hyper-sphere so as to not misclassify any points in the training set. The unoptimized training algorithm did this with a very inefficient loop, checking for misclassifications at every iteration.

The optimized training algorithm iterates over the set of classes in the training set, creating a population of antibodies for each class of the same size. For the creation of each population of antibodies, the training algorithm creates two subsets out of the training set: the set of examples belonging to the class of antibodies being created, and the set of training samples belonging to all other classes. A k-d tree data structure is then created using the second subset. Generating a single antibody for the current class of antibodies being created is simple: sample the set of samples with the same class label as the antibody for the antibody center, then query the k-d tree for the distance to the nearest non-self sample. The radius is then calculated to be just small enough to not misclassify the non-self training sample returned by the k-d tree. In this manner, the radius is set in one step instead of iteratively as before. The pseudo code for the new training algorithm is found in Figure 15. In the previous section the step_size parameter was not allowed to vary, this is changed in this section and differing values of step_size are tested, improving the accuracy of the algorithm. In the pseudo code, p[“center”] defines the center point, p[“radius”] defines the radius, and p[“class”] defines the class that the hypersphere belongs to, all attributes of hypersphere p. In the same way, i[“center”] defines the center point, and i[“class”] defines the class that the point belongs to, all attributes of point i.

An important feature of this optimization is the fact that the optimized training algorithm is functionally identical to the unoptimized training algorithm. Accounting for the randomness used for sampling, the training algorithm will generate identical sets of antibodies with the optimized and unoptimized versions.

The optimization done to the classification algorithm is now described. The classification algorithm has required an extensive redesign to be optimized. The unoptimized classification algorithm used a single loop, iterating over the set of hyper-spheres making up the antibody population. The distance between the hyper-sphere and the example to be classified was calculated, determining whether the example was within the hyper-sphere by comparing the distance calculated with the radius of the antibody. This required a lot of time and was very inefficient.
The optimization for the classification is achieved through a process of filtering. Although the distance function is still calculated on a se*t of hyper-spheres, it is a much smaller set, requiring less time. The filtering happens in two stages, primary and secondary filtering. During primary filtering, the example to be classified is compared to the set of antibodies feature by feature, selecting only the antibodies that contain the example to remain in the set. Once a feature is processed, the set of antibodies is smaller, making subsequent comparisons faster. During primary filtering, the distance function is not calculated at all, relying instead on the radius of each antibody.

Figure 16 shows primary filtering happening in two dimensions with two hyper-spheres. It can be seen that the example to be classified falls within the radius of hyper-sphere B. Filtering based on X feature, both hyper-spheres would be kept in the population, since the example falls within both in that dimension. Filtering based on the Y dimension would filter out the A antibody, since the example does not fall within the radius of the A antibody in that dimension.

```
input:
    training_set: a list of the training data points, each with an attached class label
    classes: the set of class labels present in the data
    population_size: the size of the desired population of antibodies
    step_size: a parameter (explained in text)
    tree: a k-d tree data structure for holding the training set
output:
    population: the set of antibodies used to classify flows
functions:
    random: a function for selecting a random element from a set
    distance: a function for calculating the distance between points
    normalize: a function to normalize the data to the range [0, 1]

Initialization:
    training set = normalize(training_set)
Training Algorithm:
    antibodies = {}
    FOREACH {c | c ∈ classes }
        class_data = {i | i ∈ training_set AND i["class"] = c}
        non_class_data = {i | i ∈ training_set AND i["class"] != c}

        tree = tree.construct(non_class_data)
        counter = 0
        WHILE counter < ⌈population size / |classes|⌉
            proposed_center = random(class_data)
            nearest = tree.query(proposed_center)
            distance = distance(nearest, proposed_center)
            IF distance <= step_size
                radius = 0.0
            ELSE
                radius = distance – (distance % step_size)
            ENDIF
            new_antibody=[center=proposed_center,    radius=radius,class=c]
            antibodies = antibodies ∪ new_antibody
            counter = counter + 1
        ENDWHILE
    ENDFOREACH
```


Figure 15. Pseudo Code for the Training Algorithm

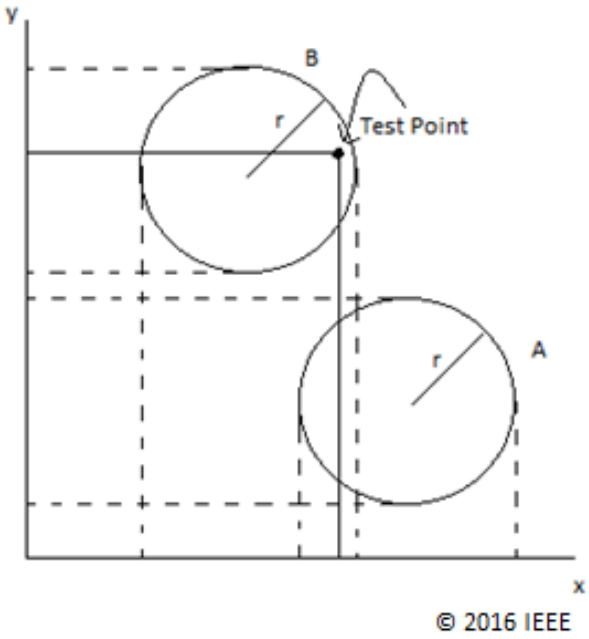

Figure 16. Explanation of Primary Filtering Scheme

Secondary filtering is necessary because primary filtering is too permissive, because primary filtering only filters out antibodies based on the hyper-cubes removes antibodies that are outside the hyper cube that contains the hyper sphere that is each individual antibody. This is because of the "roundness" of hyper-spheres. Secondary filtering simply calculates the distance function between the example to be classifier and every hyper-sphere center remaining in the set.

When the population of antibodies fails to classify an example, the optimized classification algorithm acts in the same way as before, by performing k-nearest neighbors classification with the antibody centers. However, the optimized algorithm does this in a more efficient manner with a k-d tree. Although, the k-d tree does cut down slightly in the classification time, it is not used in a large portion of the classifications, and its removal would not affect the classification time greatly. The pseudo code for the optimized classification algorithm is found in Figure 17. The conventions used in Fig 17 are the same as used in figure 16.

The optimized classification algorithm is not functionally the same as the unoptimized classification algorithm, unlike the training algorithm. The previous algorithm was naïve in one respect: it did not deal with the fact that hyper-spheres in the population are allowed to overlap. This is a problem because the first hyper-sphere that contained the example being tested was returned as the correct class label of the example. While being a more efficient implementation, there is a chance that the first antibody found would misclassify the test point. This is changed in this section by implementing majority voting with the antibodies that contain the example to be classified, each antibody having one vote and voting for its own class. If there is no majority found, the classification returns. The classification algorithm takes one parameter: k, which determines the number of antibodies used when performing k-NN classification. In the previous algorithm, this was not allowed to vary, being set to 1.

No other research has been found that takes this or a similar approach to the comparisons between an example and a set of hyper-spheres. However, there have been some attempts at optimizing the Negative Selection algorithm.

Model and Experimental Setup

The algorithm is tested using 10-fold cross validation. This means that, for each data set created for the tests, the data set was split into 10 sub data sets, each with an equal proportion of samples for each class in the data set. The tests were repeated 10 times, using each sub data set for testing, training, and validation; the results were averaged after the tests were done. As before, the split between training, testing, and validation sets is 80%/10%/10%. The results of the tests are graphed in Figs. 18 through 23. The data set used is the same as in the previous section. The used

are listed and described in Table 1. There are 12 classes in the data set, they are listed in Table 2.

```
input:
    antibodies: set of the antibodies generated by the training algorithm
    tree: a k-d tree data structure for the antibody population
    k: a parameter for classification
    x: the test point to be classified
output:
    class label: the predicted class label
functions:
    most_common: returns the element with the highest count in the given set, or a random
    element if there is no majority
    distance: a function for calculating the distance between points

Classification Algorithm:
    dimension = 0
    WHILE dimension < | antibodies[0][“center”] |
        antibodies = {a | a ∈ antibodies AND
        x[“point”][dimension] > (a[“point”][dimension] - a[“radius”]) AND
        x[“point”][dimension] < (a[“point”][dimension] + a[“radius”])]
        dimension = dimension + 1
    ENDWHILE
    selected_antibodies = {}
    FOREACH {a | a ∈ antibodies}
        d = distance(x[“center”], a[“center”])
        IF d <= a[“radius”]
            selected_antibodies = selected_antibodies ∪ a
        ENDIF
    END FOREACH
    IF |selected_antibodies| > 0
        return most_common([[“class”] in selected_antibodies])
    ELSE
        return tree.query(x, t=1)[”class”]
    ENDIF
```


Figure 17. Pseudo Code for Optimized Classification Algorithm

As in the previous section, one subset is set aside to determine the best parameters for the algorithm, before being tested with the training set. Toward this goal, grid search is performed on the validation set, used for finding the parameters that give the highest accuracy. Grid search is performed on each test in this section, used for finding the best parameters for the k and step_size

parameters, with the step_size parameter used in the training algorithm, and k used in the classification algorithm. The k parameter is varied between 1 and 12, and the step_size is varied between 0.01 and 0.99 in 0.01 increments. This process is different from the approach taken in the previous section, were these values were fixed.

In general, four sets of experiments were performed on the algorithm, testing different aspects of the algorithm. The four sets of experiments are related to three claims about the performance of the algorithm:

1. The new training algorithm is faster than the old training algorithm.
2. The new classification algorithm is faster than the old classification algorithm.
3. The classification performance of the old algorithm might be improved with different distance measures.

The first two claims are easy to understand and follow from the previous sections. The third test is related to the use of distance functions other than the Euclidian distance, which is a standard in AIS research. To maximize the accuracy of the algorithm, several different distance functions are tested, including: Manhattan distance, dot product, cosine distance, Chebyshev distance, and for the purposes of comparison, Euclidian distance.

The tests performed to compare the optimized and unoptimized version of the algorithm are designed to be as fair as possible. To this end, data sets are created of the required size by sampling the original data set with replacement. The same data sets are then used to test both versions of the algorithm. The random nature of the sampling used by the training algorithm is not accounted for, however. For the same reason, when testing the optimized and unoptimized versions

of the classification algorithm, a population of antibodies of the required size is created, and the same population is used to test both versions of the algorithm. This was done so that both versions of the algorithm could be compared without worrying about randomness affecting the results.

The tests were performed on the same computer as the ones in the previous section. A computer with an Intel i5 processor running at 1.8 Ghz is used with 4 GB of memory. The operating system is 64-bit Windows 8.1.

Results

This section explains the results of the tests performed on the unoptimized and optimized algorithms. The algorithm is also tested with different distance functions.

The results of the tests used to compare the performance of the optimized and unoptimized training algorithms are shown in Figs. 18 and 19. The relationship between the size of the data set and the time required by the training algorithm to create a 1000-member population of antibodies is shown in Figure 18. The dependent variable is the size of the data set, the independent is the time required by the training algorithm to finish, in seconds. The dependent variable increases from 200 to 1000 in increments of 50. It can be seen that while the training algorithm remains linear, the optimized version is still much faster.

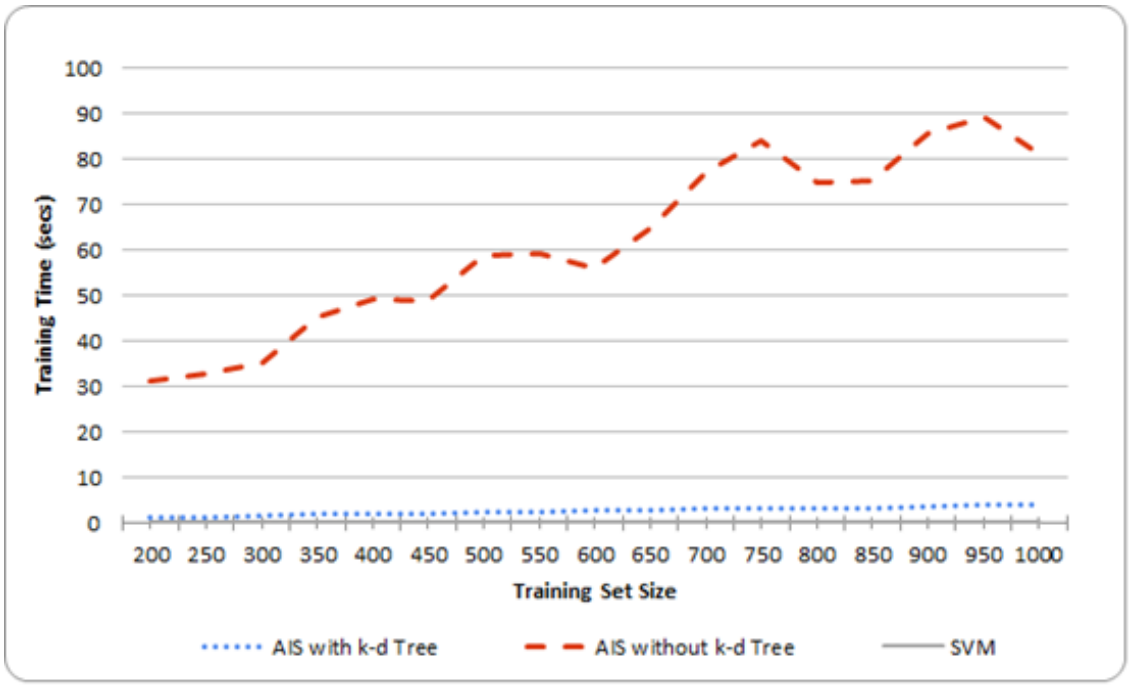

Figure 18. Training Set Size and Training Time

The relationship between the size of the antibody population and the time taken by the training algorithm is shown in Figure 19. The size of the data set used is 1000. The dependent variable is the antibody population, which varies from 200 to 1000 in increments of 50, the independent variable is the time taken for the training algorithm to complete. The optimized training algorithm is still, however, it is much faster than the old training algorithm. The SVM algorithm is tested on the same data sets as the optimized and unoptimized AIS algorithms in Figs. 18 and 19. The SVM algorithm is still faster than the AIS algorithm, and its line can be seen along the bottom of the graphs.

The relationship between the number of dimensions present in the data set and the time taken by the training algorithm is shown in Figure 20. The size of the data set used is 1000 samples, and the number of antibodies in the population is 100. The dependent variable is the number of

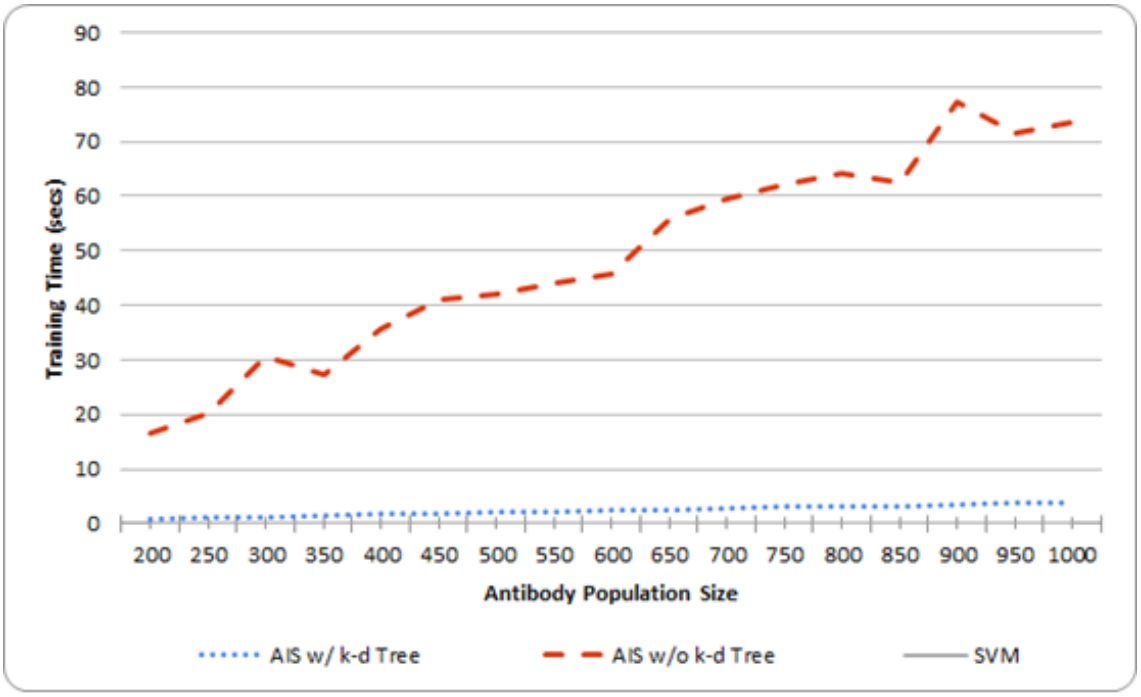

Figure 19. Antibody Population Size and Training Time

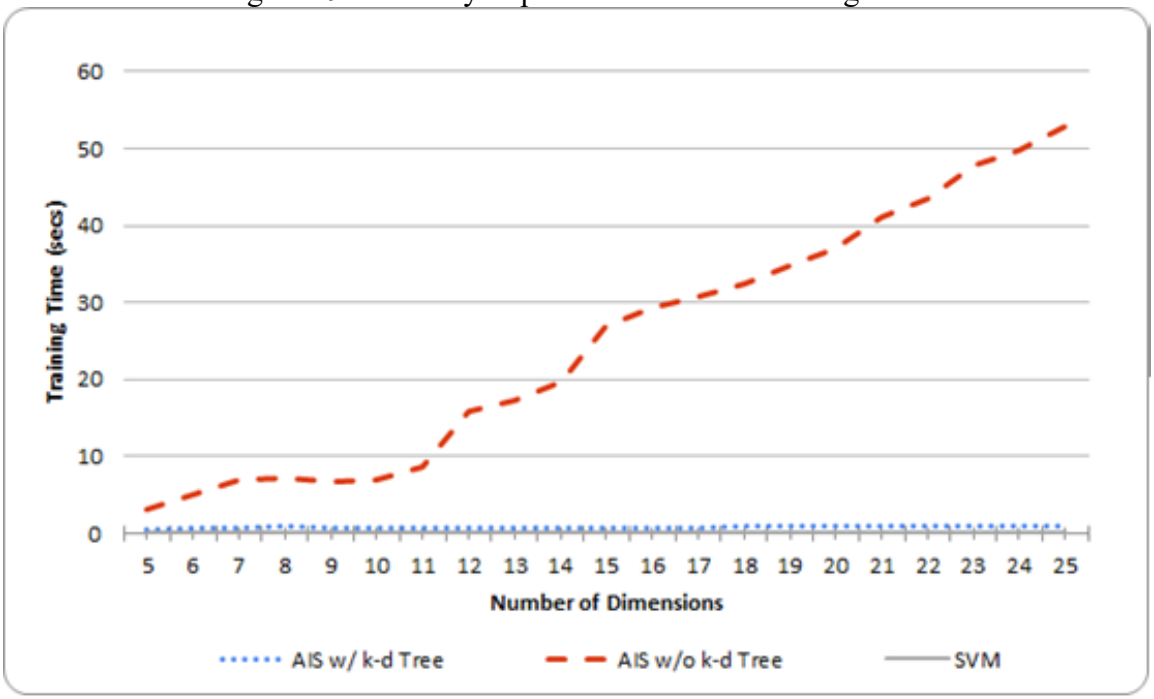

Figure 20. Number of Dimensions in Data Set and Training Time

dimensions in the data set, which varies from 5 to 25, the independent variable is the time taken for the training algorithm to complete. The SVM algorithm is tested on the same data sets as the optimized and unoptimized AIS algorithms in Figs. 18 and 19. The SVM algorithm is still faster than the AIS algorithm in all three tests, and the line depicting its performance can be seen along the bottom of the graphs.

The next set of tests is used to measure the running time of the classification algorithm, which works with the population of antibodies created by the training algorithm. Since there is no relationship between the size of the data set used in the training algorithm to train the antibody population and the running time of the classification algorithm, this variable does not appear in these tests or in the figures.

The tests performed to compare the time required by the optimized and unoptimized classification algorithms are especially complex. The aim of these tests is to show the effect of the new majority voting classifier as compared to the previous implementation, as well as the effect of secondary filtering on the execution time. The four lines displayed in the figures are: the unoptimized classification algorithm without majority voting, the unoptimized classification algorithm with majority voting, and the optimized classification algorithm with and without secondary filtering. These tests were used in order to show the performance of the algorithm in many different configurations.

A graph showing the relationship between the antibody population size and the time it takes to make a prediction is shown in Figure 21. There are several versions of the prediction algorithm compared in the figure. The dependent variable is the size of the antibody population, and the independent variable is the number of seconds it takes for the prediction portion of the algorithm

to complete. The dependent variable increases from 200 to 1000, in increments of 50. As can be seen in the figure, the prediction algorithm is still linearly related to the size of the population of antibodies, however, it is much faster. Majority voting proves to slow down the unoptimized algorithm considerably. When secondary filtering is not used by the optimized algorithm, the execution becomes much faster, however this does not give very good accuracy, as will be seen in another figure.

The relationship between the number of dimensions present in the antibody population and the time taken by the prediction algorithm is shown in Figure 22. The number of antibodies in the population is 100. The dependent variable is the number of dimensions in the antibody population, which varies from 5 to 25, the independent variable is the time taken for the prediction algorithm to complete.

Figs. 21 and 22 show that secondary filtering in the optimized algorithm is very important, since without it the accuracy of the predictions drops. However, secondary filtering also requires time to complete. Figure 21 also shows that primary filtering provides most of the speedup gained by the optimized algorithm. The predictions performed to make Figs. 21 and 22 are run 100 times, size because of the split between the training, validation, and testing data sets, which are 80%/10%/10% respectively. Because of this, the prediction time displayed is actually the time taken to make 100 predictions, this is needed because predictions can be made very quickly and are harder to measure individually.

The optimized prediction function proves to be faster than the unoptimized prediction function in Figs. 21 and 22. However, it is important to prove that the optimization does not degrade the results provided by the algorithm. In this context, it is of

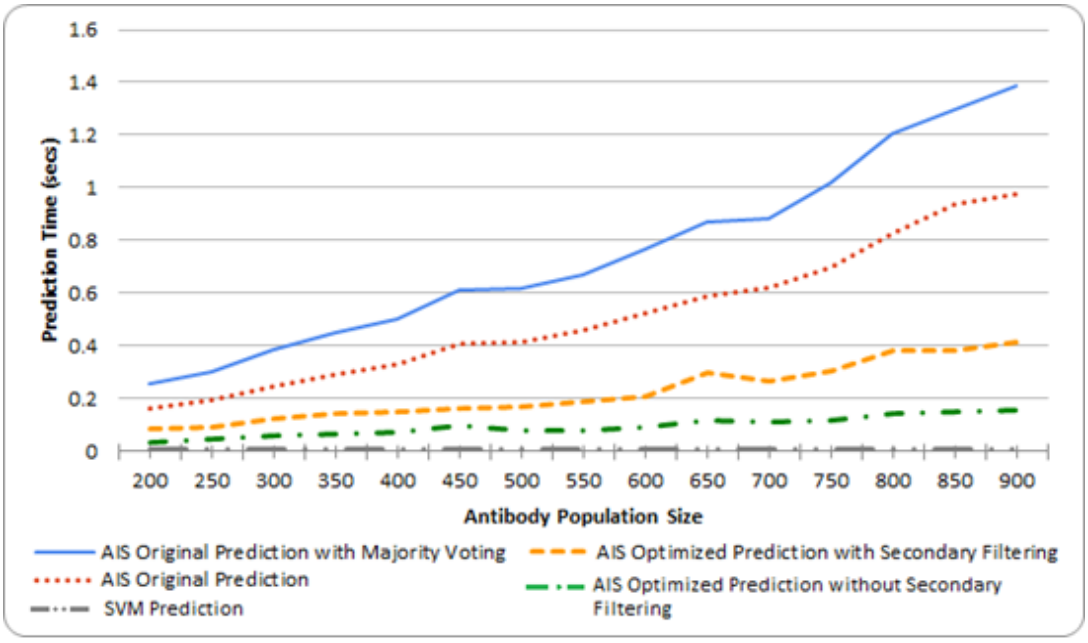

Figure 21. Antibody Population Size and Prediction Time

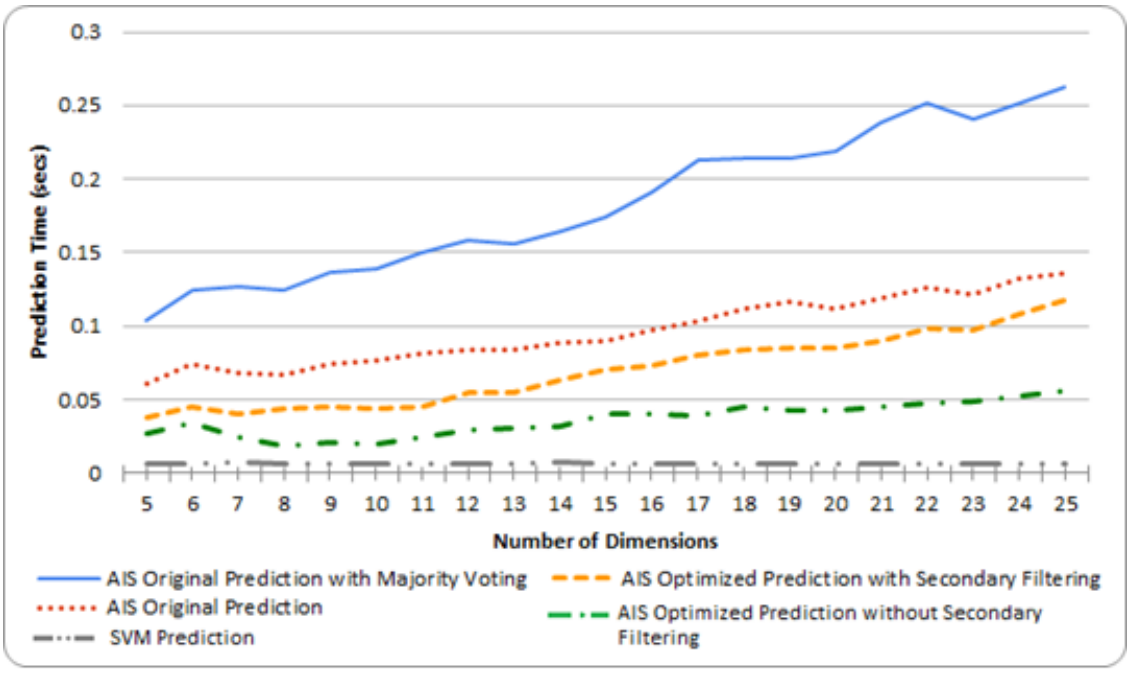

Figure 22. Number of Dimensions of Antibodies and Classification Time

interest to maintain the accuracy of the predictions. To prove that the accuracy of the optimized algorithm did not decrease as a result of the changes implemented, the accuracy of the four prediction functions is graphed in Figure 23. The depended variable is the size of the population of antibodies, the independent variable is the accuracy of the prediction algorithm. The dependent variable increases from 200 to 1000, with the size of the data set fixed at 1000. The prediction functions worked with the same population of antibodies, trained with the same data set. The accuracy remains the same, except for the version of the algorithm without secondary filtering, which decreased the accuracy significantly. The accuracy was expected to decrease, and this graph proves that the secondary filtering is absolutely essential for the algorithm to provide accurate predictions.

As in the previous set of tests, the SVM is also tested against the AIS-inspired algorithm. The SVM algorithm is still faster than the AIS algorithm in Figs. 21 and 22, and its line can be

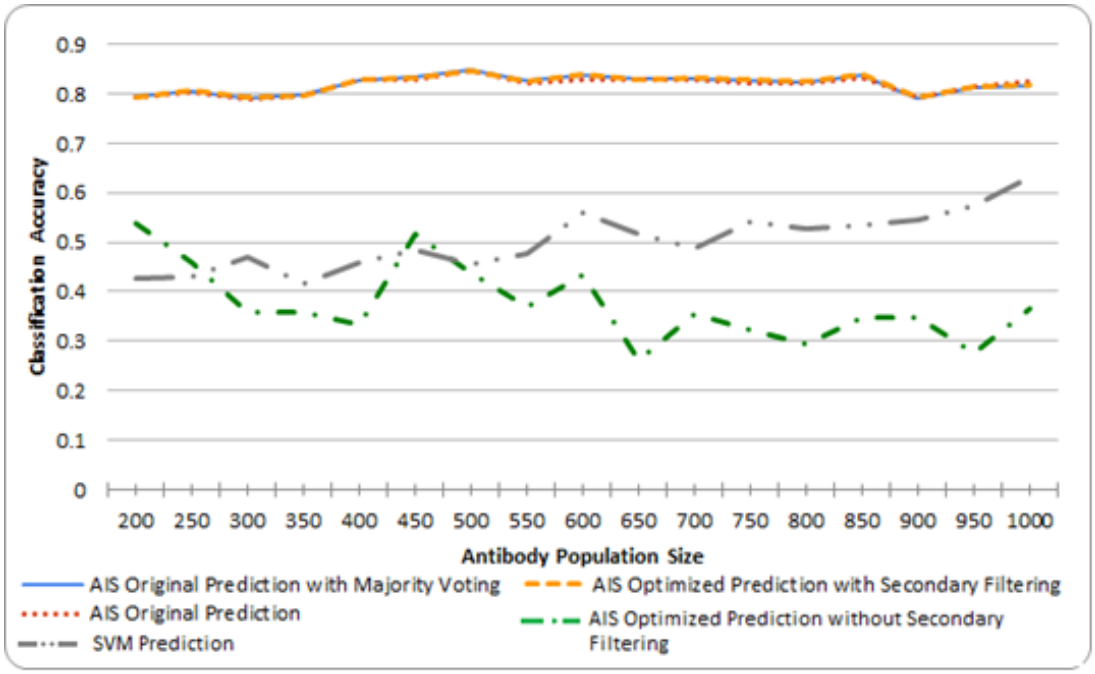

Figure 23. Population Size and Accuracy

seen along the bottom of the graphs. However, Figure 23 shows that the AIS algorithms have higher accuracy, for the given data set size.

For the tests that are displayed in Figs. 18, 19, 20, 21, 22, and 23, the algorithm was run using the Euclidian distance measure. This is a standard distance measure used in AIS research. However, in order to try to maximize the accuracy of the algorithm, some tests were performed using alternative distance measures. The distance measures tested are: Euclidian distance, and Chebyshev distance, Manhattan, and Cosine distance. Figs. 24, 25, and 26 show the results of these tests. Although the dot product as a distance function was implemented and tested, the resulting accuracy was very low and did not merit inclusion in any of the figures.

Figure 24 shows the relationship between the antibody population size used and the accuracy. The antibody population size is the depended variable, and the accuracy is the

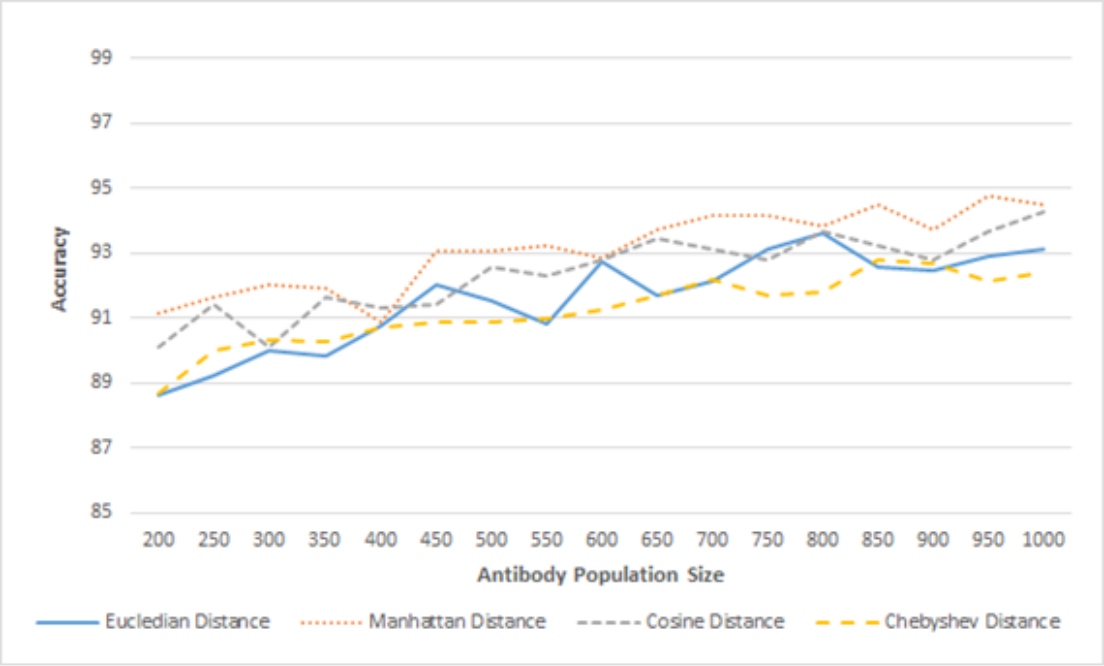

Figure 24. Antibody Population and Accuracy

independent variable. The antibody population size varies between 200 and 1000, in increments of 50. The size of the data set is fixed at 1000 for this test. Although it is not always the most accurate distance measure, it can be seen that the Manhattan distance is the most accurate throughout most of the range. The maximum accuracy achieved is 94.77% by the Manhattan distance measure.

A confidence interval was calculated using data from Figure 24 for the difference in the accuracy between the algorithm using the Euclidian distance and the algorithm using the Manhattan distance functions. The difference was calculated by subtracting the accuracy of the Manhattan distance and the accuracy of the Euclidian distance, this made the difference positive. With these values a confidence interval was calculated at the 95% confidence level. The average difference in the accuracy was calculated to be between 1.9 and 1.16 percentage points. This shows that using the Manhattan distance gives higher prediction accuracy, and that it is not due to chance but is statistically significant.

Figure 25 shows the relationship between the data set size used and the accuracy for different distance functions. The data set size is the depended variable, and the accuracy is the independent variable. The data set size varies between 200 and 1000, in increments of 50. The size of the antibody population is fixed at 1000 for this test. This chart also shows the Manhattan distance giving the highest accuracy, with a maximum of 95.99% accuracy.

A confidence interval was calculated using data from Figure 25 for the difference in the accuracy between the Euclidian distance and the Manhattan distance functions. The difference was calculated by subtracting the accuracy of the algorithm using the Manhattan distance and the accuracy of the algorithm using the Euclidian distance, this made the difference positive. With

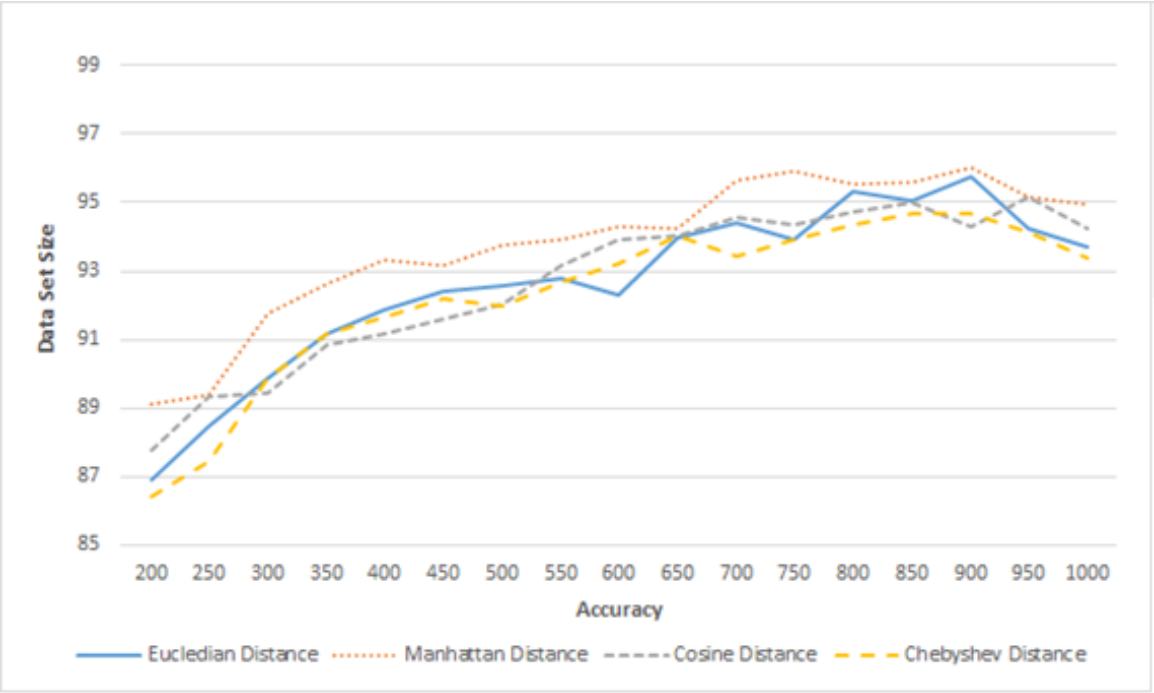

Figure 25. Data Set Size and Accuracy for Different Distance Functions

these values a confidence interval was calculated at the 95% confidence level. The average difference in the accuracy was calculated to be between 1.45 and 0.86 percentage points. This shows that using the Manhattan distance gives higher prediction accuracy, and that it is not due to chance but is statistically significant.

Figure 26 shows the relationship between the data set size and the F-measure achieved by the classifier. The F-measure is a measurement of the performance of a classifier, but is calculated on a class-by-class basis, as opposed to accuracy and error rate, which are calculated for all classes. F-measure is the weighted average of precision and recall, and has a range of [0, 1], with 1 being the best performance. The dependent variable is the size of the data set used to train the classifier, the independent variable is the F-measure. The size of the population of antibodies is fixed at 1000. The Chebyshev and

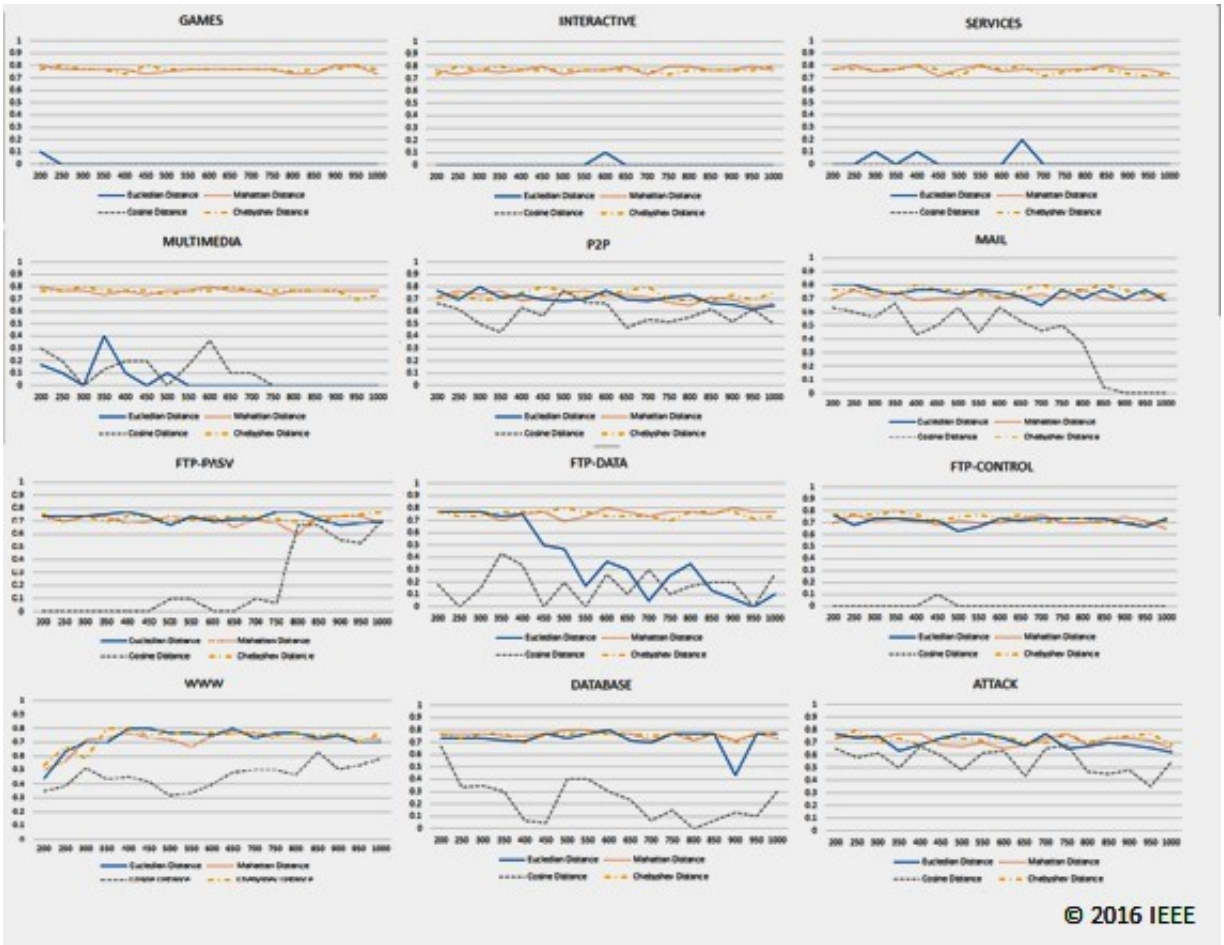

Figure 26. Data Set Size and F-Measure

Manhattan distances are the highest performing distance measures tested across all classes of flows in the data set.

The modifications to the algorithm tested in this section have shown that is indeed possible to bring the performance of the AIS-inspired algorithm closer to that of the SVM algorithm. The optimized algorithm is also functionally identical to the original, unoptimized algorithm. The performance of the optimized algorithm is especially good as the number of features in the data set rises and as the number of dimensions in the population of antibodies rises, seemingly dealing with the curse of dimensionality easily. Furthermore, the results show that the Euclidian distance

measure does not give the highest accuracy for this application. It has been shown that the Manhattan distance measure gives better classification performance.

In this chapter it is shown how the changes and optimizations applied to the original algorithm do not functionally change the original algorithm, while making its execution 50-60% faster. We also show that the classification accuracy of the Euclidian distance is superseded by the Manhattan distance for this application, giving 1-2% higher accuracy, making the accuracy of the algorithm comparable to that of a Naïve Bayes classifier in previous research that uses the same data set.

Comparison to Previous Work

This work has shown that is indeed possible to optimize the performance of an AIS-inspired algorithm. It is also shown that the algorithm is functionally identical to the previous unoptimized version. Also, the most common distance measure used, the Euclidian distance measure, does not give the best performance for this problem. Lastly, it can be seen that the Manhattan distance measure gives better classification performance, however, I believe that this is problem dependent and there is no good way to predict which distance measure will maximize the accuracy achieved by the algorithm. All distance measures must be tested to find the best one.

Although there has been previous work in the application of non-Euclidian distance measures to AIS algorithms [102], it is only applicable to the AIRS algorithm. The present research tests several distance measures with a positive selection algorithm.

Support Vector Machine classifiers are usually trained using quadratic optimization techniques. There has been much research done into ways to do this on large data sets. For example, sequential minimal optimization and feasible direction decomposition, are algorithms

designed to train SVM classifiers on large data sets, in the order of 10^5 and 10^6 examples [103]. For even bigger data sets, the Hadoop framework and Map-Reduce algorithms have been adapted to solve the optimization problem. Lastly, stochastic gradient descent has also been used to train SVMs on very large data sets. These approaches do not redefine the problem and still frame it as an optimization problem.

Our proposed approach uses fewer parameters than other natural computing algorithms and does not incur the training costs associated with discovering such parameters. For example, the performance of the genetic algorithms depends highly on the mutation and cross-over operations and parameters. Similarly, the performance of artificial neural network, deep learning and extreme machine learning based approaches depends highly on the number of hidden layers, the number of neurons in each layer and the employed activation function. Also, the performance of SVM is highly dependent of the Kernel function used and its parameters. In this paper, we propose an optimized AIS algorithm that needs few parameters and produces results comparable to these produced by the optimal parameters of the aforementioned methods. Therefore, the proposed approach eliminates all the overhead and subjectivity involved in the selection of the parameters in other biologically inspired approaches.

Furthermore, Artificial Immune System algorithms are able to operate in highly distributed systems and can be easily adapted to run on networked computers. AIS algorithms are capable of learning new patterns, remember previously learned patterns, and do pattern recognition in networked systems. At the same time, their performance degrades gracefully, in the same way as Artificial Neural Networks. In past research, AIS algorithms have been used to detect malicious activity in computer networks [8]. Because of this research and the capabilities of AIS classifiers

we are encouraged to explore their performance on the task of network flow classification. Research has also shown that positive selection AIS algorithms can perform very well in certain problems compared to negative selection in problems that require a comprehensive data set of negative examples. Positive selection is also being simpler to code and faster to train. For this reason, the algorithm presented in this paper is a positive selection algorithm.

The original algorithm described here is designed to be simple and fast so that it will work well in resource-constrained systems. Because of our previous findings, we have been motivated to develop optimizations for the algorithm, to make it competitive with other Machine Learning approaches while depending on lesser configurable parameters.

When testing the optimizations made to the algorithm, a speedup of about 10x-30x was achieved in the training algorithm. A speedup of around 2x was observed in the classification portion of the algorithm. No significant differences where observed in the accuracy of the optimized and unoptimized algorithms. When testing different distance functions, it was observed that Manhattan distance was 1% to 2% more accurate for the data set used.

The training algorithm used for the classifier does not frame the problem as an optimization problem at all, choosing instead to make certain assumptions about the data. The first assumption is that the training set contains points that are good approximations for the best placement of the hyper-sphere centers. The second assumption is that by setting the radius of each hyper-sphere so that it will not misclassify any points in the training set, a good voting ensemble classifier can be built. During the literature review, no other classifiers have been found that follow this path. However, there have been a few that have some of these elements. These assumptions have allowed the classifier to be easily optimized, and for it to have certain qualities that the SVM classifier

lacks. Furthermore, there has not been much research into the optimization of the running times of ensemble classifiers.

Optimizing the AIS Negative Selection Algorithm

The Negative Selection algorithm is the first algorithm to be derived from the study of the natural immune system [8]. The Negative Selection algorithm has been studied widely since it was first proposed and has been applied to many different problems. It is often applied to intrusion detection tasks and security-related tasks. The ability of the Negative Selection algorithm to model a class boundary using only samples from one class can be especially useful in certain types of problems. A description of the algorithm is given in the second section of chapter 3 and a pseudo code listing for it is given in Figure 5. The work described in this section first appears in [104].

In an early application of Negative Selection, Kim and Bentley [27-30] used the algorithm to detect malicious network connections. They found that the algorithm needed an excessive amount of time to create a set of antibodies that was big enough to achieve the necessary accuracy. Specifically, the algorithm would need to run for 1429 years to achieve 80% accuracy on a sample of network data that was only 20 minutes long. To achieve that accuracy 6×108 detectors would be needed.

The research presented aims to create an optimized version of the Negative Selection Algorithm, both in the training and classification phases. The optimization described here is applicable on a wide variety of data types and applications. Furthermore the optimized algorithm is functionally the same as the unoptimized version of the Negative Selection Algorithm. This is proven here by showing that the accuracy of the optimized algorithm is not higher or lower than

the unoptimized algorithm. The algorithms are compared using three claims and using four experiments.

As described in previous sections, the Negative Selection algorithm relies on a population of simple classifiers to perform classification. The classifiers are known as “antibodies” and can be implemented in many different ways. However, all antibodies must return a Boolean value, they either “match” or “don’t match” a sample to which they are compared.

Using the concept of affinity, hyper-sphere antibodies can be defined. These antibodies are made of two components: a center point defined by a set of coordinates, and a radius that is defined by a scalar value. A hyper-sphere antibody “matches” a sample if the sample falls within the radius defined by the antibody. The radius can be defined by a distance function used to describe the affinity of the antibody. This type of detector works only in feature spaces that contain continuous (real-valued) features. The most common distance measure used in Euclidian distance. A more thorough description of antibody types can be found in the second section of chapter 3

Description of the Optimization

This section gives a general description of the optimization scheme along with implementation details.

In the pseudo code found in Figure 4, the function matches() is used to compare a new detector with every sample in the data set. This function is used heavily to find detectors that do not match any samples in the subset of samples in a data set not in the “self” class. This is because the detector is applied one-by-one to each sample in the training set.

Instead of comparing a new detector with the set of self samples individually, in the optimized training algorithm the comparison proceeds feature-by-feature. This process is best

explained as a “filtering” process that is applied on each feature in the data set individually. The filtering process discards the self samples that do not match the new detector in the current feature being processed. The set of self samples becomes smaller and smaller as the filtering proceeds, speeding up the comparison as the features are processed. If there are any self samples remaining in the set after all of the features are processed, the proposed detector matches one or more self samples, and is therefore discarded (as with normal Negative Selection). If there are no self samples left in the set at any point in the processing, the proposed detector does not match any self samples, and can therefore be added to the set of detectors.

As in Figure 4, in the pseudo code found in Figure 6 the function matches() is used to compare the set of detectors with the sample to be classified. This function is used to find detectors that match the sample to be classified. It works by comparing each detector in the set of detectors to the sample to be classified individually.

In the same way as the training algorithm, when classifying a sample into self or non-self, the comparison between the sample and the set of detectors proceeds feature-by-feature. The set of detectors that could match the self-sample becomes smaller and smaller as more and more detectors are “filtered” from the set. If the set of detectors is emptied during this filtering process, then the sample does not belong to the self class. If there are detectors remaining in the set after all of the features are processed, then the sample belongs to the non-self class.

Figure 16 shows primary filtering happening in two dimensions with two hyper-sphere detectors. It can be seen that the point to be classified falls within the radius of hyper-sphere B. Filtering based on the feature X, both hyper-spheres would be kept in the set, since the example

falls within both in that dimension. Filtering based on the Y dimension would filter out the A detector, since the example does not fall within the radius of the A detector in that dimension.

The Negative Selection algorithm implemented to test the optimization proposed in this research is implemented using hyper-sphere detectors. The detector radius is defined using Euclidian distance. Each detector contains a center point, defined as a set of coordinates in Euclidian space, as well as a radius. A detector matches a sample only if the sample falls within the radius of the detector. Each detector has a fixed radius, given to the training algorithm as a parameter. The only other parameter needed by the training algorithm is the size of the detector set to be generated. The range of the values in each feature of the data set was normalized to the range [0,1]. This is done to simplify the code, but is not necessary and the optimization can be implemented without this step.

The training algorithm uses a two-stage filtering process to speed up the comparison between the new detector and the set of samples containing the self class. The first stage compares each feature of the sample to the allowed range of the detector in that feature. The allowed range is calculated by adding and subtracting the radius from the coordinate of the center point in that dimension. If the sample does not fall within the range calculated, it is removed from the set. In this manner, the set of samples that could be contained by a detector is iteratively reduced in size. The secondary filtering step is necessary in this case because of the “roundness” of the hyper-sphere detectors. The primary filtering process could leave some samples in the set if they fall within the hyper-cube that contains the hyper-sphere that is the detector. Secondary filtering then proceeds as normally done by the Negative Selection algorithm, by iteratively comparing the remaining sample set with the detector. After primary and secondary filtering are completed, the

samples remaining in the set are the ones that fall within the detector radius. If the set is empty, then the detector can be added to the set of detectors, since it does not match any samples in the self set. If at any point in the primary filtering process the set of samples becomes empty, then the algorithm is able to add the detector immediately, since it is known that the detector does not match any sample in the self set without having to perform secondary filtering. The pseudo code for the optimized Negative Selection training algorithm can be found in Figure 27. In this pseudo code listing the primary filter section contains while loop that is filtering out all detectors that do not meet the criteria. This filter applies the logic described in the previous sections.

In the pseudo code, p[“center”] defines the center point, p[“radius”] defines the radius, and p[“class”] defines the class that the hypersphere belongs to, all attributes of hypersphere p. In the same way, i[“data”] defines the center point, and i[“class”] defines the class that the point belongs to, all attributes of point i.

When using a detector in the pseudo code, d[“center”] references the vector that contains the center point of the detector d, and d[“center”][0] references the first dimension of that vector. Similarly, d[“radius”] references the scalar value that defines the radius of the detector. When using a data point in the training set in the pseudo code, s[“class”] references the category that the sample s belongs to. Also, the vector that defines the sample s is stored in s[“data”], with s[“data”][0] referencing the first dimension of the data vector of s.

The filtering process is very similar in the classification algorithm as it is in the training algorithm, only it is done in reverse. Instead of comparing the set of self samples with one detector, the filtering process compares a set of detectors with one sample. The set of detectors is iteratively reduced, and the remaining detectors are subjected to secondary filtering. If a detector remains in

Definitions:
training_set: a list of the training data points, each with an attached class label
detectors: the set of detectors to be created
population_size: the size of the desired population of detectors
self_class_label: the label of the class designated as "self"
normalize(): a function to normalize the data set
generate_random_antibody(): a function to generate a random antibody
distance(): a function for calculating the Euclidian distance between points
nd: total number of dimensions in the data set

Initialization:
```
training set = normalize(training_set)
detectors = {}
```
Training Algorithm:
```
WHILE | detectors | < population_size:
    self_class = { s | s ∈ training_set AND s["class"] = self_class_label }
    na = generate_random_antibody()

    #primary filtering
    d = 0
    WHILE d < nd
        self_class = { s | s ∈ self_class
        AND na["center"][d] > (s["data"][d] - na["radius"] )
        AND s["data"][d] < (na["center"][d] + na["radius"] )}
        d = d + 1
    ENDWHILE

    #early decision
     IF | self_class | = 0
        detectors = detectors ∪ na
    #secondary filtering
    ELSE
        flagged = FALSE
        FOREACH { s | s ∈ self_class }
            IF distance(na["center"], s["data"]) < na["radius"]
                flagged = TRUE
            ENDIF
        ENDFOREACH
        IF flagged = FALSE:
            detectors = detectors ∪ na
        ENDIF
    ENDIF
ENDWHILE
```


Figure 27. Pseudo Code for the Optimized Negative Selection Training Algorithm

the set after both primary and secondary filtering are complete, then the sample is classified as non-self, since it is "matched" by one or more detectors, otherwise it is classified as self. The pseudo code for the optimized Negative Selection classification algorithm can be found in Figure 28. In this pseudo code listing the primary filter section contains while loop that is filtering out all detectors that do not meet the criteria. This filter applies the logic described in the previous sections.

```
detectors: set of the detectors generated by the training algorithm
x: the sample to be classified
self_class_label: label of the class designated as "self"
non_self_class_label: label of the class designated as "non-self"
distance(): a function for calculating the Euclidian distance between points
nd: total number of dimensions in the data set

Classification Algorithm:
#primary filtering
d = 0
WHILE d < nd
detectors = { a | a ∈ detectors
AND x["data"] [d] > (a["point"][d] - a["radius"])
AND x["data"] [d] < (a["point"][d] + a["radius"]) }
    d = d + 1
ENDWHILE

#secondary filtering
FOREACH {a | a ∈ detectors}
    d = distance(x["center"], a["center"])
    IF d <= a["radius"]
        return non_self_class_label
    ENDIF
ENDFOREACH
return self_class_label
```


Figure 28. Pseudo Code for the Optimized Negative Selection Classification Algorithm

In both of the optimized training and classification algorithms an “early decision” can be made. This happens when the set of self samples is emptied, in the training algorithm, or when the set of detectors is emptied, in the classification algorithm.

Model and Experimental Setup

This section contains details about the way in which the optimization was implemented, along with the data set used and the claims being tested.

To test the optimization, the Breast Cancer Wisconsin (Diagnostic) Dataset was chosen from the UCI repository [19]. This data set was chosen because it is limited to two classes, which fits with the AIS paradigm. To test the algorithm, some preprocessing was done to the data set. The class label of each sample was placed in the first column of the data set. All samples with missing values were removed from the data set, making the data set smaller but simplifying the algorithm. All duplicate rows were removed from the data set as well. After this was done, the data set contained 683 labeled samples, each with 9 real-valued features. The class labels found in this data set are “malignant” with 239 samples found in the data set, and “benign” with 444 samples found in the data set.

The model was validated using 10-fold cross validation. To do this, the dataset used was split evenly into 10 subsets. From these 10 subsets, training, validation, and testing sets are created. The training set created used 80% of the samples, the validation 10%, and the testing set 10% of the data. Stratification was also used, which is a technique used to make sure that each of the 10 subsets is created so that it contains the same proportion of each class in the data set. Through this process, it is possible to create 10 unique testing sets, 10 unique validation sets, and 10 unique

training sets. By cycling through these, the experiments are performed 10 times and the results are averaged.

Since the algorithm requires one parameter, one subset is set aside in every test run to determine the best values for these parameters. To accomplish this, a grid search is performed on the validation set, with the objective of finding the value for the parameter which maximizes the accuracy of the algorithm. The parameter is the radius of the hyper-spheres. The radius is varied from 0.01 to 0.99 in 0.01 increments

Four experiments were performed with the original Negative Selection algorithm and the optimized version of the Negative Selection algorithm. The results of the experiments are detailed in the next section. The experiments are designed to demonstrate three claims that are made about the optimized Negative Selection algorithm. The claims deal with the execution time, classification time, and classification performance of the algorithm. Our claims about the algorithm are these:

1. The optimized training algorithm is faster than the unoptimized training algorithm.
2. The optimized classification algorithm is faster than the unoptimized classification algorithm.
3. The optimization does not affect the accuracy of the algorithm, being functionally the same.

To make the comparisons between the optimized and unoptimized algorithms as unbiased as possible, two methods were used: when testing the training algorithm, both versions of the algorithm were given the same parameters and the exact same data set, with the same set of sub data sets (due to the 10-fold cross validation). When testing the classification algorithm, the exact same set of detectors is provided to both versions of the algorithm. This was done so that both

versions of the algorithm could be compared without worrying about randomness affecting the results.

When comparing the accuracy of the optimized and unoptimized algorithms, the accuracy is calculated as follows:

$$\text{Accuracy} = (TP + TN) / (TP + TN + FP + FN)) \quad (1)$$

were TP is the number of true positive predictions, TN is the number of true negative predictions, FP is the number of false positive predictions, and FN is the number of false negative predictions.

All experiments were performed on an Intel i5-based computer running at 1.80 GHz. The computer has 4 GB of memory, and the operating system used is 64-bit Windows 8.1. Both the optimized and unoptimized algorithms were coded in Python 3.4.

Results

This section shows the results of the experiments and demonstrates the validity of the claims made in the previous section. To simplify the tests, the detector radius was set to 0.5 for the experiments graphed in Figures 29, 30, and 31. This radius was found using a grid search, which was used to find the detector radius that maximized the accuracy of the algorithm. The grid search was performed using the validation set.

The relationship between the training time and the data set size is shown in Figure 29. The detector set size is held constant at 1000, and the data set size was increased from 100 to 500. It can be seen that the optimized training algorithm remains linear on the number of samples in the data set. The time is measured in seconds.

A confidence interval was calculated using data from Figure 29 for the difference in the average time taken to finish by both algorithms. To do this, 10 data points were taken from the last test graphed in the figure. For this test, the detector set size was 1000, and the data set size was 500. With these values a confidence interval was calculated at the 95% confidence level. The difference in the average time taken to complete training was calculated to be between 1.3 and 1.01 seconds, with the optimized algorithm being faster. This helps to show that claim 1 is true.

The relationship between the training time and the detector set size is shown in Figure 30. The data set size is held constant at 500, and the detector set size was increased from 100 to 1000. The optimized training algorithm is also linear with the number of detectors in the set. It is faster than the unoptimized training algorithm.

A confidence interval was calculated using data from Figure 30 for the difference in the average time taken to finish by both algorithms. To do this, 10 data points were taken from the last test graphed in the figure. For this test, the detector set size was 1000, and the data set size was 500. With these values a confidence interval was calculated at the 95% confidence level. The difference in the average time taken to complete was calculated to be between 1.07 and 1.04 seconds, with the optimized algorithm being faster. This also helps to demonstrate the validity of claim 1.

The relationship between the size of the set of detectors and the classification time is shown in Figure 31. The classification time is the time taken to classify one sample. The size of the detector set was increased from 100 to 1000.

The confidence interval was calculated using data from Figure 31 for the difference in the average time taken to finish by both algorithms. To do this, 10 data points were taken from the last

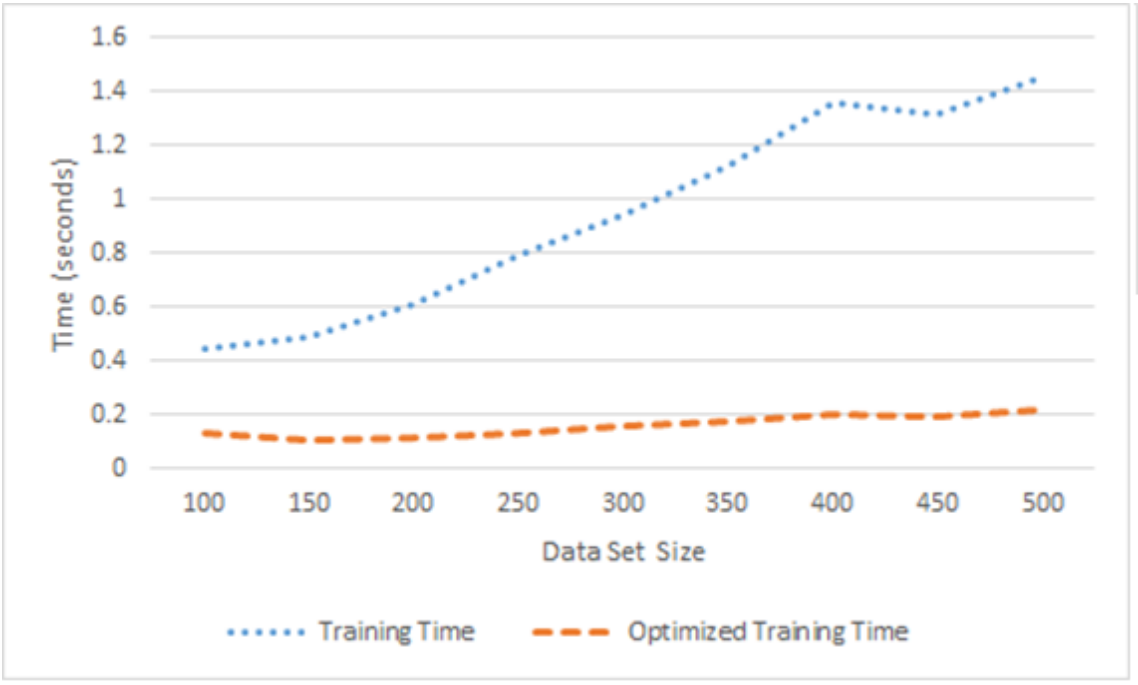

Figure 29. Data Set Size and Training Time for Optimized Negative Selection

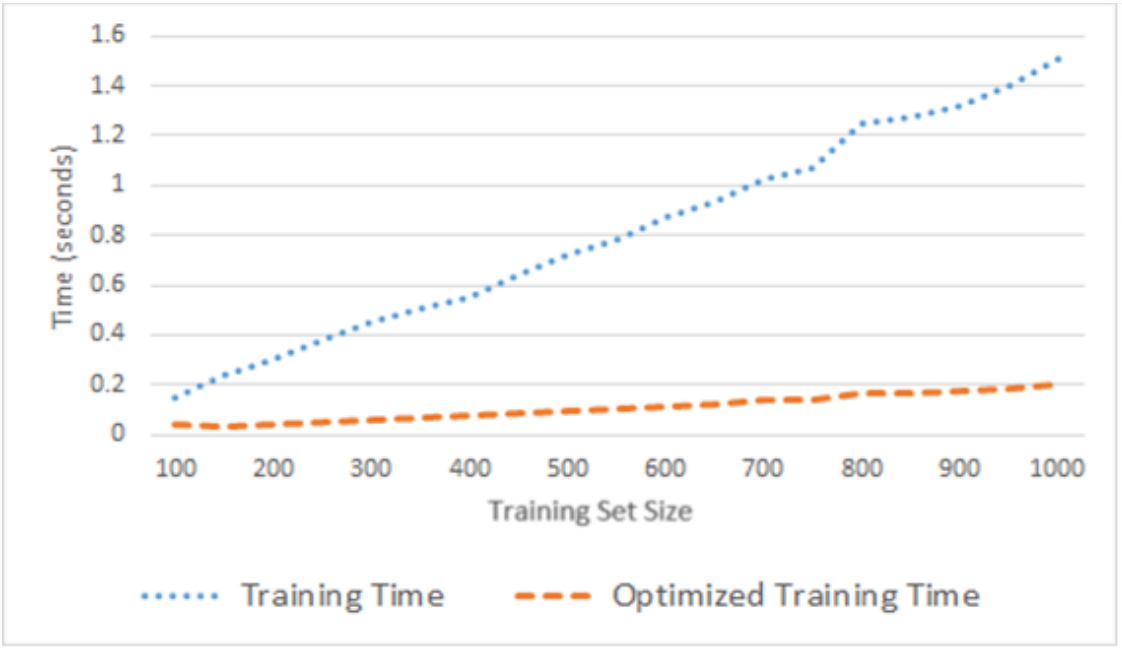

Figure 30. Antibody Population Size and Training Time for Optimized Negative Selection

test graphed in the figure. For this test, the detector set size was 1000.

With these values a confidence interval was calculated at the 95% confidence level. The difference in the average time taken to complete was calculated to be between 0.0033 and 0.00033, with the optimized algorithm being faster. The results of this experiment demonstrate that claim 2 is valid.

The fourth experiment is done on the training and classification algorithms in tandem, proving that the combination of the optimized training and classification algorithms does not negatively affect the accuracy of the algorithm. Figure 32 shows the relationship between the data set size and the accuracy achieved by the algorithm. As mentioned, the optimized and unoptimized versions of the algorithm use the exact same data set to create the set of detectors. The size of the set of detectors generated is held constant at 1000. Although it is not easily seen, the accuracy achieved by the optimized algorithm does not match the accuracy of the unoptimized algorithm exactly. This is due to the fact that the Negative Selection algorithm uses randomness in the training process.

A t-test was performed to compare average accuracy achieved by both the optimized and unoptimized algorithms. The samples were paired according to the data set size used, using the same data that is graphed in Figure 9. The confidence level used was 95%. The null hypothesis could not be rejected, meaning that the analysis did not provide evidence against claim 3. Additionally, the Pearson correlation between the paired accuracies was calculated to be 0.996, a value that shows that the accuracies of the unoptimized and optimized versions of the algorithm are very closely related.

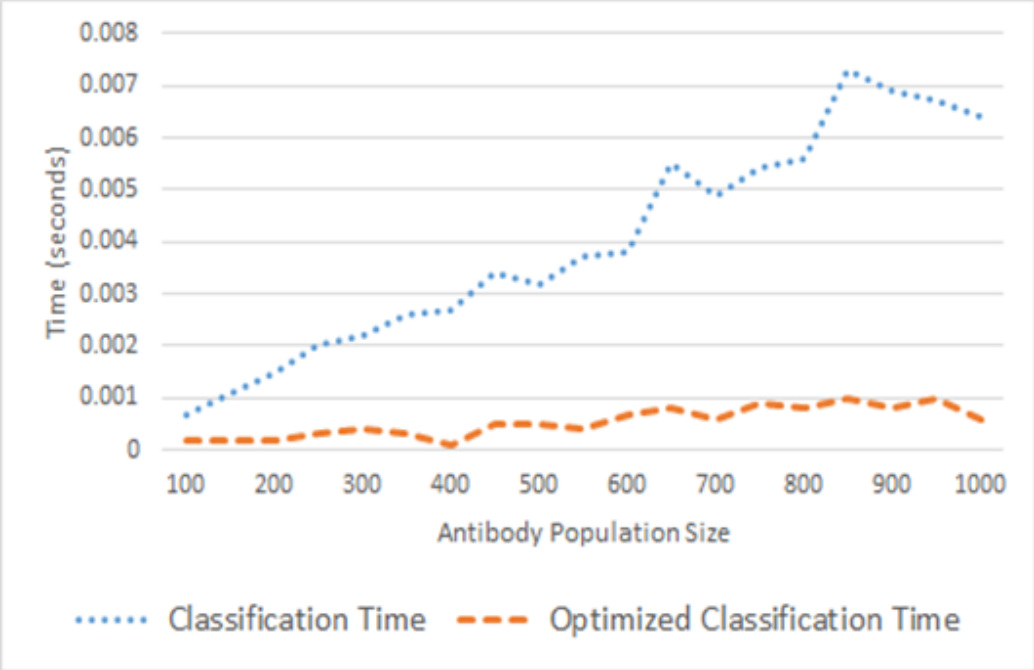

Figure 31. Population Size and Classification Time for Optimized Negative Selection

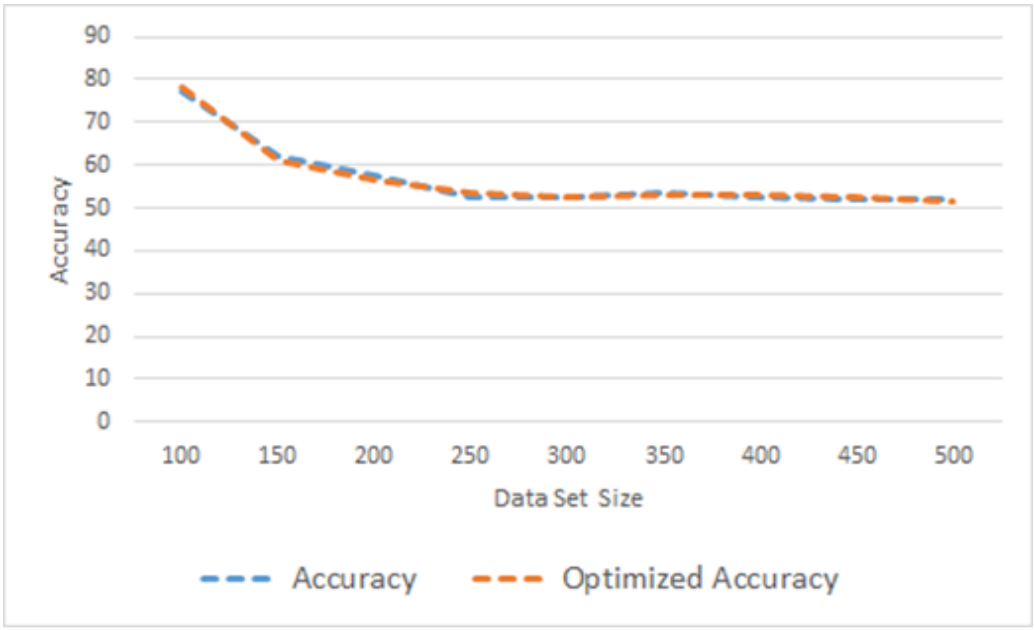

Figure 32. Data Set Size and Accuracy for Optimized Negative Selection

Comparison to Previous Work

The optimization described in this section is very similar to the optimization of the classification algorithm in the previous section. However, the optimization described here applies to the class of algorithms known as Negative Selection. The optimization described in the previous section was specifically designed for that algorithm and only worked in the classification portion of the algorithm. This optimization works in both the classification and training portions of the Negative Selection algorithms.

In the literature review performed for this research, no other approach similar to this one was found. A lot of the work in the area of optimizing the Negative Selection algorithm was only applicable to r-contiguous and r-chunk detectors.

CHAPTER VI

THEORETICAL ANALYSIS OF THE ALGORITHM

This chapters looks at the algorithm from a theoretical perspective. It includes both an analysis of the VC-dimensionality achieved by the classification algorithm as well as the big O complexity of the classification and training algorithms. The analysis described in this section only applied to the algorithm described in the first two sections of chapter 5 and does not include the Negative Selection algorithm used in the third section of chapter 5.

VC-Theory

Vapnik-Chervonenkis theory was developed by Vladimir Vapnik and Alexey Chervonenkis between 1960 and 1990. The theory aims to explain learning from a rigorous mathematical and statistical point of view. Through this theory, many important proofs can be constructed concerning the learnability of a concept, the descriptive power of a classifier, and the generalization ability of a learning process. [106]

Through the use of Vapnik-Chervonenkis theory it is possible to quantify the descriptive power of a model. This is done by measuring the VC-dimensionality of the functions that define the model with the concept of a shattering set. Through the use of VC-dimensionality, it is also possible to provide a bound on the generalization error of an algorithm. It is applied to the current problem to explain the ability of the algorithm developed in this work to describe class boundaries, as well as to make it easier to derive bounds.

VC-Dimensionality

VC-dimensionality measures the descriptive "capacity" of a hypothesis space, where the hypothesis space is defined by the functions that make up a classifier. When applied to machine learning, the VC-dimensionality of a classifier shows how expressive the classifier can be. For example, a high-degree polynomial has a high capacity, because it is capable of separating a more complicated set of points into two regions of space. A line has a low capacity because it is unable to separate a complicated set of points into two distinct regions. This can be seen in the graph of the XOR function in Figure 14, since it is impossible to find a line to separate both classes. The VC-dimension of a classifier is defined to be the cardinality of the largest set of points that the classifier can "shatter."

The notion of "shattering" is now defined more formally. Given a set D with the structure:

$$D = \{(x_1, y_1), (x_2, y_2), \ldots (x_N, y_N)\} \quad (11)$$

the size of D being:

$$|D| = N \quad (12)$$

With each vector x_i being defined in the real numbers with d dimensions:

$$X = \mathbb{R}^d \quad (13)$$

and each y_i being a member of the set Y:

$$y_i \in Y \quad (14)$$

Which is equal to:

$$Y = \{1, -1\} \quad (15)$$

From the above definitions it can be seen that:

$$(x_i, y_i) \in X \times \{-1, 1\} \quad (16)$$

The elements in Y are called the "label" of the vector x_i. The number of possible class assignments for the vectors is:

$$|Y|^{|D|} = 2^N \tag{17}$$

Therefore, it can be seen that with a binary classification problem and a data set of size N, there are a possible 2N learning problems, because there are 2N possible ways to assign the class labels in set Y to the vectors in D.

A hypothesis space is denoted in this way:

$$H \tag{18}$$

A hypothesis space is useful in the context of this research because it helps us define the capacity of a classifier. This is because a classifier can be thought of as a function h, defined as:

$$y = h(x) \tag{19}$$

where, $y \in Y$ and $x \in \mathbb{R}^d$, and h: $X \rightarrow Y$. The function h maps every member of the vector space X to the class labels in set Y.

A data set is said to be "shattered" by a hypothesis space H if and only if for every subset of the data set, there is a hypothesis $h \in H$ that separates the subset without any errors into regions containing only points from one of the labels. The VC-dimension of a hypothesis space H is denoted:

$$VC_{dim}(H) \tag{20}$$

where VC_{dim} may be equal to any natural number, up to infinity.

More formally: the VC dimension of a hypothesis space H defined over instance space D is the size of the largest finite subset of D shattered by H. If arbitrarily large finite sets of D can be shattered by H, then VC dimensionality of H is infinite.

Stated in a different way, the VC-dimensionality of a hypothesis space is defined to be the cardinality of the largest data set that can be shattered by the hypothesis space, including every subset of the data set. If the classifier is able to express hypotheses that are able to classify all possible assignments of classes to samples, then the VC-dimensionality of the classifier is equal to infinity. When calculating the VC-dimensionality of a class of functions, it is sufficient that one set of points that can be shattered is found; it is not necessary to prove that the class of functions has the capacity to shatter every possible set of points of a certain size.

As mentioned before, the VC-dimension is a property of a set of functions, which denote the behavior of a classifier. At the same time the functions also define a hypothesis space, which define the hypotheses that the classifier is capable of implementing. The VC-dimension of a classifier is a measurement of the expressive capacity of the classifier that implements the hypothesis space, which is itself a set of functions. The VC-dimension of a hypothesis space is not related to the size of the hypothesis space, but is related to the number of distinct samples of a data set that can be correctly classified by a hypothesis in the hypothesis space implemented by the classifier.

As an example, a data set of three points in two dimensions, with two possible classes can have 8 possible class assignments:

$$|X| = 3 \tag{21}$$

$$|Y| = 2 \tag{22}$$

the number of possible class assignments is:

$$2^3 = 8 \tag{23}$$

If a linear classifier is used, such as an SVM, then a classifier can be found that will separate the data set perfectly. In terms of VC theory, it is said that the class of linear classifiers "shatters" the data set, because for all 8 possible class assignments to the 3 points in the data set, classifier can be found that will separate them correctly. It is important to understand that the set of hyper-plane classifiers is able to shatter a set of size three because there is at least one arrangement of the points that can be shattered. It is not necessary for the class of functions to shatter all possible arrangements of points of a certain size for the VC dimension to be equal to the size of the set in question.

The VC-Dimension of Hyper-Sphere Classifiers

To lay down a foundation that more complicated proofs can be built on, a proof for the VC-dimension of a single hyper-sphere classifier is given here. A hyper-sphere is defined like this:

$$S^n = \{ x \in \mathbb{R}^n, c \in \mathbb{R}^n \mid d(c,x) = r \} \quad (24)$$

where n is the number of dimensions that the hyper-sphere is defined in, c is the center of the hyper-sphere and d is the distance function being used. Given that d is a distance metric, it can only be a positive real number. The distance function need not be the Euclidian distance function.

The VC-Dimensionality of One Hyper-Sphere Classifiers

To accomplish classification using a hyper-sphere, a function must be defined that maps members of X to members of Y, as defined above. Therefore, a formalization of a hyper-sphere classifier is given by defining a function that can be used as a classifier:

$$f(x) = \begin{cases} 1 & if\ d(c,x) \leq r \\ -1 & otherwise \end{cases} \quad (25)$$

A single hyper-sphere classifier is a simple classifier that defines every point within its radius to be of class 1, and everything outside of it to be of class -1. A class of functions is defined:

$oh_i \in OHSC$ (for "one hyper-sphere classifiers") (26)

where each function oh_i within OHSC is defined by two parameters:

$oh_i = [c_i, r_i]$ (27)

The center of the hyper-sphere is the vector c, and the radius r is a scalar. Both defined to be:

$c = \mathbb{R}^d$ (28)

$r = \mathbb{R}$ (29)

and the center vector c_i has d dimensions.

For the first proof, it is useful to remind ourselves of Radon's theorem. The theorem states that any set of d + 2 points in $\mathbb{R}$ d can be divided into two disjoint sets whose convex hulls intersect. In other words, there always exists a way to partition a set of points so that the convex hulls of the subsets have at least one point in common. The convex hull of a set of points can be visualized as the set of points that correspond to a string stretched around the points (in two dimensions).

The proof for Theorem 1 shows that sets of size *d*+2 cannot be shattered by hyperspheres by proving that the VC dimensionality of hyper-spheres is the same as the VC dimensionality of hyper-planes in the same number of dimensions. The proof can be better understood by visualizing a hyper-sphere with a radius equal to infinity. Such a hyper-sphere would behave in the same way as a hyper-plane. Theorem 1 is important because every other proof within this chapter depends on the VC-dimensionality of a single hyper-sphere. The second part of the proof relies on the fact

that half-spaces are proven to have a VC-dimensionality of d+1. The proof for Theorem 1 first appears in [107] which is an unpublished manuscript. To the best of our knowledge, there is no other publication describing the VC-dimensionality of hyper-spheres.

Theorem 1: *VC dimension of hyper-spheres in* $\mathbb{R}^d$ *is equal to d+1, where d is the number of dimensions of the hyper-sphere.* Stated in another way, $VC_{dim}(OHSC) = d+1$.

Theorem 1 establishes for the VC-dimensionality of the class of functions OHSC. The proof is done in two parts. First, it is proven that a set of points of size d+1 can be shattered by the class OHSC, then it is proven that no set of points of size d+2 can be shattered by the class OHSC. This is enough to prove that the VC-dimensionality of the class of functions is equal to d+1, according to the definition of VC-dimensionality. Lemmas 1 and 2 below contain the two parts of the proof.

Proof:

Lemma 1: *A set of d+1 points consisting of the unit vectors and the origin can be shattered by hyper-spheres of OHSC.* Suppose A is a subset of the d+1 points. The center a_0 of a hyper-sphere will be the sum of the vectors in A. For every unit vector in set A, its distance to the center a_0 will be $\sqrt{|A| - 1}$ and for every unit vector outside A, its distance to the center a_0 will be $\sqrt{|A| + 1}$. The distance of the origin to the center is $\sqrt{|A|}$. Thus it is easy to see that the radius can be chosen so that precisely the points in A are in the hyper-sphere.

Lemma 2: *No set of points of size d+2 can be shattered by class OHSC.* Proceeding by contradiction. Suppose that there exists a set S with d + 2 points in it that can be shattered by a

hyper-sphere of *d* dimensions which is a member of OHSC. Applying Radon's theorem, for any partition of set S into subsets A_1 and A_2, there exist hyper-spheres B_1 and B_2 such that $S \cap B_1 = A_1$ and $S \cap B_2 = A_2$. B_1 and B_2 may intersect, but assume without loss of generality that there is no point belonging to S in their intersection. Because no points in S are allowed to exist in the intersection of B_1 and B_2, it is easy to see then that there is a hyper plane with all of A_1 on one side and all of A_2 on the other. This implies that half-spaces are able to shatter the set S which is of size d + 2, which is a contradiction since all half spaces are proved to have VC_{dim} of *d*+1. Therefore, no set of d+2 points can be shattered by a hyper-sphere.

Corollary 1: *All classifiers that implement a function of the class OHSC will have a VC-dimensionality of d+1.* This follows from the fact that the VC dimensionality of a single hyper-sphere is equal to d+1.

Corollary 1 shows that all one hyper-sphere classifiers will have a VC-dimensionality of *d*+1, no matter how they are parametrized. These results will be used for all of the remaining proofs in this chapter.

The VC-Dimension of a Multi Hyper-Sphere Classifier

A definition is now given for a classifier made up of multiple hyper-spheres. As before, each hyper-sphere is parametrized by:

$$h_i = [c_i, r_i] \tag{30}$$

and the set of hyper-spheres that makes up the classifier is then defined to be:

$$HS = \{ h_1, h_2, h_3, \dots , h_M\} \tag{31}$$

where the size of the set is:

$$M = |HS| \tag{32}$$

the classification function then becomes:

$$f(x) = \begin{cases} 1 & if\ \exists h\,\{\, h \mid h\ \in\ HS,\ d(h[c], x) \leq h[r]\,\} \\ -1 & otherwise \end{cases} \tag{33}$$

Using the definition above, a class of functions, denoted as OHSC, where:

$$mh_i \in MHSC \qquad \text{(for "multi hyper-sphere classifiers")} \tag{34}$$

where each function mh_i within MHSC is defined by many pairs of parameters, each pair representing one hyper-sphere:

$$mh_i = \{\, [c_1, r_1]_1, [c_2, r_2]_2, [c_3, r_3]_3, \ldots, [c_M, r_M]_M \,\} \tag{35}$$

where each center c_i is a vector, and each radius r_i is a scalar, defined to be in:

$$c_i = \mathbb{R}^d \tag{36}$$

$$r_i = \mathbb{R} \tag{37}$$

and the center vector c_i has d dimensions.

The lower bound for the VC-dimensionality of the class of functions MHSC is now given through two corollaries. These corollaries are possible because it can easily be seen that the class of functions MHSC contains all possible one hyper-sphere classifiers, which are described in the class of functions OHSC.

Corollary 2: *The VC-dimensionality of the class of functions MHSC when M=1 is d+1, where d is the number of dimensions.* Stated in another way, $VC_{dim}(MHSC) = d+1$ when M=1.

This corollary is easily seen from Theorem 1, since when M=1 the function in MHSC would be equivalent to a function in OHSC.

Corollary 3: *The lowest possible value of the VC-dimensionality of the class of functions MHSC when M≥2 is d+1, where d is the number of dimensions.* Stated in another way, $VC_{dim}(MHSC) \geq d+1$ when $M \geq 2$.

Corollary 3 can easily be seen when visualizing the class of functions MHSC. The lowest possible VC dimensionality of the class is achieved when it equals the VC dimensionality of a single hyper-sphere (a classifier that is a member of OHSC). This happens when all hyper-spheres in the set HS have the same radius and the same center. The hyper-spheres behave as one and the classifier behaves as a member of OHSC even though it belongs to MHSC. It can also be seen that the VC dimensionality of the classifier can only increase if the radii and centers of the hyper-spheres do not match, since a single hyper-sphere is the simplest class boundary representable by MHSC.

Corollary 4: *The upper bound of the VC-dimensionality of the class of function MHSC is M(d+1), where d is the number of dimensions and M is the number of hyper-spheres in the set.* Stated in another way, $VC_{dim}(MHSC) \leq M(d+1)$ if $M \in \{1,2,3, \ldots, N\}$. This applies when M is bounded by an integer N.

As an example, suppose that there exists a set of points of size $(M(d+1))+1$ that can be shattered by a function in the class MHSC. This is not possible, since the maximum number of points that an individual hyper-sphere can shatter is $d+1$, meaning that the maximum number of points a set of hyper-spheres can shatter is $M(d+1)$.

The maximum VC dimensionality of the class MHSC is achieved when each hyper-sphere in the set HS shatters d+1 points in the set. In aggregation, then, the set of hyper-spheres would shatter M(d+1) points. Since it is enough to show that there is an arrangement of points in a set of a certain size to achieve a certain VC dimensionality, it is easy to imagine a set of points on which it is possible to place M hyper-spheres, each hyper-sphere shattering a subset of points of size d+1, and achieve a VC dimensionality of M(d+1). Furthermore it is easy to see that it would be possible to place an additional point in the set so that the set of hyper-spheres would not be able to shatter the set, making a new set of size (M(d+1))+1. Therefore, the VC-dimensionality of the class of functions MHSC can never be more than M(d+1).

Theorem 2: *The VC dimensionality of MHSC is infinite if the number of hyper-spheres in the set is unbounded.* More formally: $VC_{dim}(MHSC) = \infty$ if $M \in \{1,2,3, \ldots\}$. This theorem applies when M is unbounded.

Proof: Proceeding by contradiction, assume that there does not exist a function in class MHSC that can shatter a set of points D. But a function can be easily built by centering a hyper-sphere on every element of D1, setting the radius to be equal to 0. Then proceed to set the radius of every hyper-sphere in the classifier so that each hyper-sphere does not contain any element in D-1. This is a contradiction, since such a function would be able to shatter the set D and is in MHSC. Therefore, the VC dimension of the class of functions MHSC is infinite because it is able to shatter a set of points of any size.

Corollary 5: T*he VC-dimensionality of all multi hyper-sphere classifiers is equal to the VC-dimensionality of the class of functions MHSC*. It can also be seen that the maximum descriptive

power of a classifier that uses a function from the class MHSC grows along with the number of hyper-spheres. This follows from the fact that all multi hyper-sphere classifiers implement a function in the class MHSC.

Theorem 2 is important because it shows the ability of the classifier to distinguish non-linearly separable data sets, given enough hyper-spheres. This also means that the training set error will be 0%, if enough hyper-spheres are used in the classifier. However, the only way to guarantee that this will hold true is if the number of hyper-spheres used in the classifier is equal to the number of elements in the training set with class label equal to 1. In contrast, SVMs cannot separate non-linearly separable data sets without giving a training error of more than 0%, when used without kernel functions.

Bound for the Generalization Error of the Classifier

Because the set of functions in MHSC has a VC-dimensionality equal to infinity, the bound on the generalization error of the classifier is based on the concept of the margin of the ensemble classifier. The case of function classes with infinite VC-dimensionality is specifically excluded from the work on generalization error bound by Vapnik [106], which is better known than the approach taken here. It is possible, however, to use the work of Breiman and Shapire et. al. [108] since they define the generalization error bound based only on the VC-dimensionality of the base classifiers used in an ensemble classifier. Specifically, it can be shown that their bound applies directly to the classifier described in this dissertation, because it is a voting ensemble classifier.

Introduction to Margins

In this section an analysis is given for the expected classification performance of our algorithm. To accomplish this, the margin of an ensemble classifier and margin distributions are described and linked to the current work. The key insight in this section is the fact that the positive selection AIS algorithm that is being here examined is, in fact, an ensemble classifier, using many hyper-spheres to classify a test example. The hyper-spheres are in fact base classifiers.

The way that the current classifier being analyzed is built has some similarity to bagging, which was first described in [109] by Breiman. Bagging is used to randomly create many data sets out of the original data set, and train one "base" classifier with each created data set. The outputs of the base classifiers are then combined to make a prediction. Bagging is used to diminish the variability of the data set and can often give good results. Subagging is a variation of bagging in which the randomly created datasets are smaller than the original data set.

The work by Breiman in [110] is also useful to us, since it shows an analysis of the generalization error bound for random forests, which are a type of bagging classifier. The research links the generalization error of the ensemble classifier to the strength of individual trees and the correlation between them, two concepts that are defined thoroughly in the publication. This analysis is valuable because of the similarity of both AIS and random forest approaches, both being ensemble classifiers. There has also been a lot of research into the generalization error bound for the Adaboost algorithm, which uses the concept of boosting, itself is first described in [111]. The classification error bound of AdaBoost is first described in [108], where the concept of the margin is first introduced as an explanation of the success of ensemble classifiers.

The problem is couched in the following definitions, some of which are repeated from previous sections but are included for clarity. The definitions specific to this section are taken from Cai, Chang, and Peng's work in [112].

Let X be the feature space, and let Y be the set of class labels. The data set D is made up of (x, y) pairs, with each x and y being members of X and Y respectively. Therefore, it can be seen that the set D is a subset of X x Y, and follows an unknown underlying distribution.

A classifier is the product of a learning process, and is implemented as a mapping function from X → Y, which seeks to minimize the generalization error between the predictions of the classifier and the true underlying distribution, but accomplishes this through the minimization of the error on the training set. The classifier is written as:

$$h(x;\ \theta): X\ \rightarrow\ Y \tag{38}$$

where x stands for the input vector which is a member of X, θ is a vector of parameters, and the output is a member of Y. The classifiers can be thought of as existing within a classifier space which is defined to be Θ, which contains all possible classifiers, with $\theta \in \Theta$.

The voting margin is a concept related to majority voting ensemble classifiers, such as the one described in this dissertation. The voting margin is defined as follows:

$$mg(x, y;\ \theta_1, \theta_2, \ldots\ \theta_k) = 1/k\ (\sum_{i=1}^{k} I(h(x;\ \theta_i) = y)\ - \max_{j \neq y;\ j \in Y} \sum_{i=1}^{k} I(h(x, \theta i) =\ j)\ ) \tag{39}$$

where there are k base classifiers in the ensemble classifier, each defined by a set of parameters θ, y is the correct class label of x, and I is the indicator function. The margin function mg has a range of [-1, 1]. The sign of the output of mg signifies the accuracy and confidence of the classification.

If the sign is negative the classification is incorrect, if it is positive it is correct. The absolute value of the margin function indicates the confidence that the classifier has about the prediction.

According to Shepiro et. al. [113], with high probability, the bound on the generalization error of a voting classifier is:

$$R_v \leq \widehat{\Pr}[mg(x,y;\ \theta_1,\theta_2,\dots\ \theta_k)] + \tilde{O}\left(\sqrt{\frac{d}{m\theta^2}}\right) \tag{40}$$

where R_v refers to the risk of ensemble voting classifiers, which is the same as the generalization error of the classifiers. Here, $\widehat{\Pr}[\text{mg(x,y; } \theta_1, \theta_2, \dots\ \theta_k)]$ is the probability distribution of the margin function, d is the VC- dimensionality of the base classifiers, and m is the size of the data set. This bound holds for any $\theta > 0$. The proof for the above bound is found in [114].

By the margin explanation of ensemble methods, the best classifiers have a large margin. In the previous version of the algorithm, the margin was related to the value of the step_size parameter, which was the same for all base classifiers. In the new classifier, the margin is determined for each individual base classifiers, and is also not given as a parameter.

Since the algorithm used in this work also uses the k-Nearest Neighbors classification algorithm, the generalization error bound is shown here for it as well. The bound for the generalization error of the k-Nearest Neighbor is first found in [115]. A stronger bound for the k-NN algorithm is found in [116] and is reproduced here:

$$R_k \leq R^*\left(1 + \frac{\gamma}{\sqrt{k}}(1 + O(k^{-1/6}))\right) \tag{41}$$

where R_k is the risk of the k-NN classifier, which is the same as the generalization error of the classifier. Here, γ is a constant equal to 0.33994241 and the O notation refers to the limit as $k \to \infty$. R* is the risk obtained by an optimal decision strategy, known as the Bayes error.

Applying the Bound to the Classifier

Since the algorithm proposed in this dissertation is also a voting classifier, the bound given in the previous section also applies to it. However, not every base classifier in the ensemble classifier is allowed to cast a vote. As seen in previous sections, only hyper-spheres that contain the example are used to make a prediction. Because of these differences, the margin function is similar, but not the same as in other ensemble classifiers. My work is an extension of [113].

The margin of the classifier is calculated for multi-class classification, but is otherwise the same as in previous research. Each hyper-sphere that contains the test point votes for its own class. Therefore, the margin function becomes:

$$mg(x, y;\ \theta_1, \theta_2, \ldots\ \theta_k) = 1/k\ (\sum_{i=1}^{k} I(hs(x;\ \theta_i) = y)\ - \max_{j \neq y;\ j \in Y} \sum_{i=1}^{k} I(hs(x, \theta i) =\ j)\ ) \tag{42}$$

where hs is a hyper-sphere in the set HS which defines the classification function, and each hyper-sphere hs_i is parametrized by a set of parameters θ_i. As previously mentioned, the number of hyper-spheres that cast a vote (k, above) is not necessarily equal to the size of the set HS, since only hyper-spheres that contain the test point are allowed to cast a vote.

Described more informally, the margin function for the classifier under study is calculated by subtracting the number of incorrect votes made by the hyper-spheres from the number of correct votes made by the hyper-spheres. This is multiplied by one divided by the total number of votes cast, which causes the margin to be between [0, 1].

Applying the margin function described above to the margin drawn by Shapiro is simple, now that the margin function is defined for it. The VC-dimension of the base classifier is (d+1), as

proven in previous sections. The data set size is easily determined, and the margin function is defined above. Therefore, the following bound on the generalization error of voting classifiers also holds for the current algorithm under study:

$$\text{Generalization Error} \leq \begin{cases} if \exists h\{h|h \in HS, d(h[c], x) \leq h[r]\} & \hat{\Pr}[mg(x, y; \theta_1, \theta_2, \ldots \theta_k)] + \tilde{O}(\sqrt{\frac{d}{m\theta^2}}) \\ otherwise & R^*(1 + \frac{\gamma}{\sqrt{k}}(1 + O(k^{-1/6}))) \end{cases} \tag{43}$$

The bound is a combination of the *k*-NN and voting classifier bounds defined in the previous section. When the point to be classified is found within one of the hyper-spheres that makes up the classifier, then the voting classifier bound is applied. Otherwise, the *k*-NN bound is applied.

The generalization error bound is graphed in Figure 33. The figure is created assuming that the margin distribution is maximized, meaning that the best margin possible margin of the classification surface between the classes in the data set has been found. This is not necessarily possible in real-world data sets, but it is assumed here to make a clear graph. The three lines in the figure correspond to three VC-dimensionalities, being set to 2, 4, and 6 respectively. The theta value is set to 1 for all three lines. The data set size is increased to show how the bound decreases, following the law of large numbers.

Empirical Risk Minimization

The principle of empirical risk minimization is a component of statistical learning theory, and it is used to give bounds on the accuracy of classifiers. The principle is used to elucidate the relationship between the training set error and generalization error of a classifier. It is covered here because it can be used to explain some of the qualities of the classifier. Although the definitions

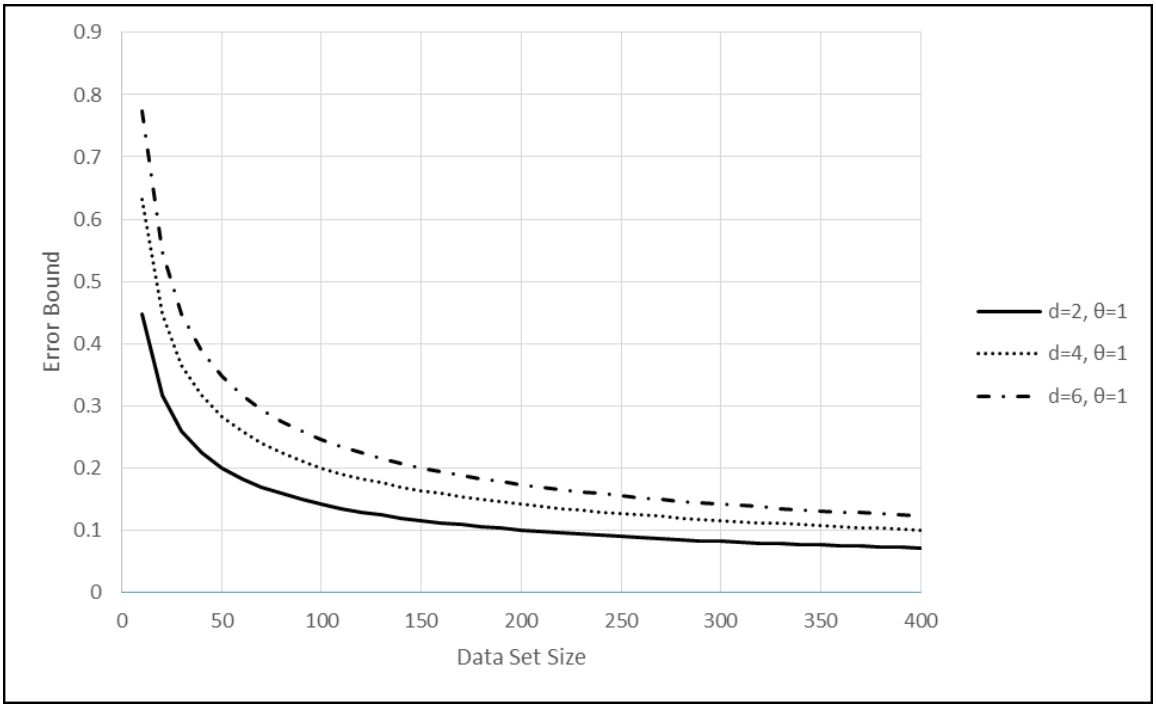

Figure 33. Value of the Generalization Error Bound When Voting Classifier is Used

used in this section are the same as used in previous sections, they will be restated for clarity.

Given a set of labeled vectors of the form:

$$D = \{(x_1, y_1), (x_2, y_2, \ldots (x_N, y_N)\} \quad (44)$$

where $y_i \in Y$ and $x_i \in \mathbb{R}^d$. The goal of a classifier is to learn a classification function of the form:

$$y = h(x) \quad (45)$$

where, again, $y \in Y$ and $x \in \mathbb{R}^d$, and h: $X \rightarrow Y$. The function h is a member of a hypothesis space H, which contains all possible hypotheses. The data in the set D is assumed to be sampled independently and identically from the underlying joint probability distribution:

$$P(x, y) \quad (46)$$

over X and Y. The classifier should be built to correctly classify samples that where not seen in the data set. When a classifier performs classification on unseen examples, it is called generalization.

Within the hypothesis space of classifiers H, the best classifier is chosen by minimizing a function. The minimization is done using a loss function:

$$L(y, y') \tag{47}$$

which measures the difference between two members of the set of class labels Y. The loss function most frequently used in the 0-1 loss function:

$$L(y, y') = I(y \neq y') \tag{48}$$

where I is the indicator function which is equal to one if the statement is true and 0 otherwise. The loss function is assumed to give a positive real number.

The risk of a classifier f is essentially the probability that it will misclassify an unseen sample drawn from the joint probability distribution P(x,y). It is stated as the expected value of the loss function on the distribution P:

$$R(f) = E[L(f(x), y)] \tag{49}$$

and is calculated by the integral:

$$R(f) = \int L(h(x), y)\, dP(x, y) \tag{50}$$

The empirical risk minimization principle states the best classifier f is one that minimizes the average risk on the training set. This is due to the law of large numbers, which states that the empirical risk will converge to the expected risk of the probability distribution as the number of samples in the data set goes to infinity [106].

More formally, the best classifier f is found by solving this minimization problem:

$$f^* = \underset{f \in F}{\operatorname{argmin}}\, R(f) \tag{51}$$

However, the risk of a classifier cannot be calculated because the probability distribution P(x,y) is unknown, therefore the generalization error cannot be minimized. This shortcoming is dealt with by approximating the generalization error by using the empirical error:

$$R_{emp}(f) = 1/N \sum_{i=1}^{N} L(f(x_i), y_i) \quad (52)$$

where x_i and y_i are members of X and Y, respectively. The minimization problem then becomes:

$$f^* = \underset{f \in F}{\operatorname{argmin}} R_{emp}(f) \quad (53)$$

More simply stated, a classifier can approximate the best classifier for the unknown probability distribution by finding the classifier that gives the lowest error rate on the training set.

According to work by Feldman et. al [117], finding the best classifier for a given sample distribution in an NP-Hard problem. This result only holds when the learning algorithm is agnostic, meaning that it makes no assumptions about the probability distribution P(x,y).

Because of the research referenced above, it is known that training a classifier to implement the function that minimizes the error on the training set is an NP-Hard problem. Therefore, approximation algorithms are used. In this dissertation, an ensemble classifier is presented, and an approximation algorithm is used for training it. The approximation algorithm specifically minimizes the error on the training set through techniques described in previous sections.

Structural Risk Minimization

Structural risk minimization is used in Machine Learning to deal with the problem of overfitting. Overfitting happens when a classifier trained on a data set makes prediction based on noise present in the data, instead of on the underlying relationship between X and Y. The main cause of overfitting is using a classifier that is too complex, in other words, having a classifier with

a too-high VC-dimension. A classifier that overfits the data set will have low empirical risk on the training data, but will have very poor generalization. This happens when the classifier starts to learn the characteristics of the training data. The structural risk minimization principle was first stated in [118].

Another way to understand SRM is to understand that a class of functions with a high VC-dimension does not necessarily help the classifier that implements a function of the class to have high generalization performance, and may harm it by allowing the training process to fit the classifier to the nuances of the training data. However, a class of functions with a VC-dimensionality that is too small will also harm the generalization ability of the classifier, since it may not allow the classifier to find a hypothesis that is descriptive enough to fit the probability distribution P(x,y).

The SRM principle can be seen at work in Figure 34. The VC-dimensionality of the function is graphed along the horizontal axis of the figure, and the error rate along the vertical axis of the figure. As can be seen, the training error (empirical risk) can be made arbitrarily low by having a very descriptive class of functions. However, the generalization error (test error) does not follow the same relationship, only achieving a minimum and the rising again. It can be seen that there is an optimal VC-dimension at which a classifier is able to give the lowest generalization error.

A related concept is Occam's razor, a principle which was first stated by William of Occam in the Middle Ages. Occam's razor purports that among competing explanations for phenomena, the one with the least number of assumptions is best and should be chosen. In this context, the classifier that should be chosen is the one with the lowest VC-dimensionality that is able to explain

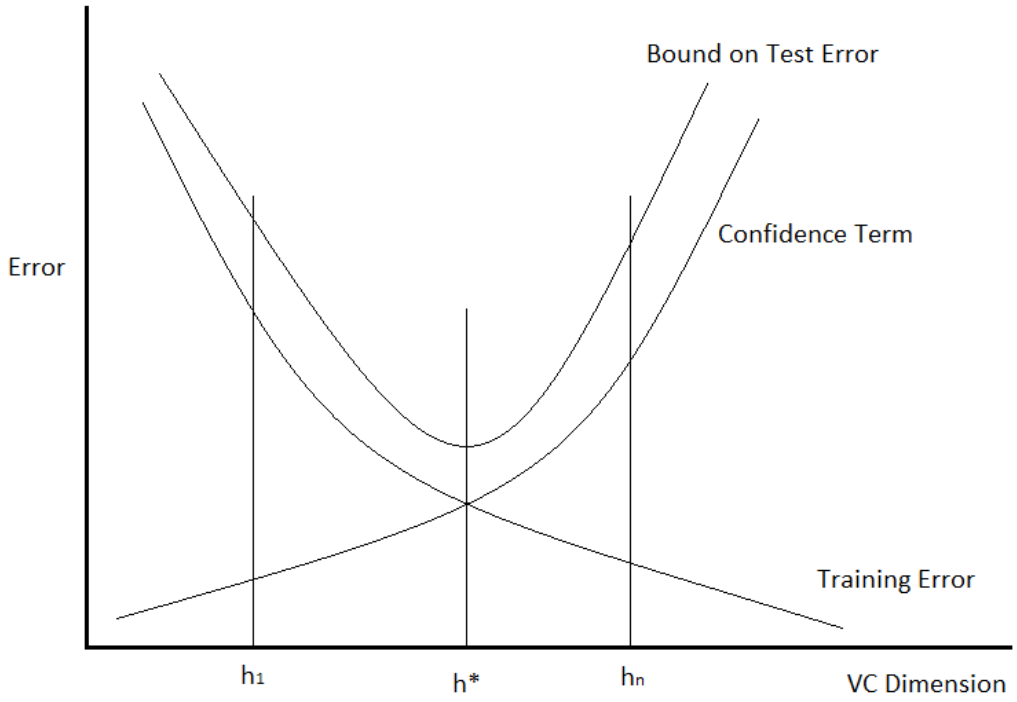

Figure 34. Structural Risk Minimization

the training data. As with other parameters, the optimal VC-dimensionality is often chosen through cross-validation.

Because of the SRM principle, it is important to find a model with the right descriptive power for the data and application. As proven in the previous sections, the VC-dimensionality of the classifier described in this work is proportional to the number of hyper-spheres used in the classifier. The upper and lower bounds are given in a previous section. Because of this, the VC-dimensionality of the multi hyper-sphere classifier can be scaled up and down by setting a parameter given to the training algorithm. This is useful in many situations, and is a simple way to only use as much descriptive power as necessary to solve a classification problem. Minimizing the

descriptive power of a classifier also means that unneeded processing is not performed, and the generalization error is minimized.

Big O of the Algorithms

This section deals with the analysis of the complexity of the optimized training and classification algorithms. The big O complexity of the optimized algorithm is given, as well as the expected classification performance of the algorithm.

Analysis of the Training Algorithm

The training algorithm is a one-shot algorithm that is dominated by the cost of creating and querying *k*-d trees. Like the original, unoptimized algorithm, the optimized algorithm is still a linear algorithm, although the constants are much smaller than the constants in the original algorithm.

For the creation of the *k*-d trees, the data set must be divided into subsets, this is linear on the size of the training set:

$$N \tag{54}$$

where N is the size of the training set.

The algorithm must first initialize one *k*-d tree for each class in the training set, therefore:

$$O(|Y| * e\, n \log n) \tag{55}$$

where Y is the set of classes present in the training set, *e* is the number of dimensions in the dataset, and *n* is the total number of samples in the training set that are not in the current class.

Once the *k*-d tree is built for each class, the algorithm queries one *k*-d tree one time for each antibody created in order to determine the correct radius for each antibody. The complexity is now dominated by the number of antibodies created:

$$O(p \log n) \tag{56}$$

where p is the number of antibodies required, and n is the number of samples in the training set not in the class of the antibody being generated. A query from one of the *k*-d trees created in the previous step requires $O(\log n)$ time.

Lastly, since the classification falls back to *k*-NN classification if there is no antibody that contains the test point, a *k*-d tree must be created that contains all of the antibody centers. This takes:

$$O(e p \log p) \tag{57}$$

where *e* is the number of dimensions in the dataset, and *p* is the total number of antibodies created.

Putting these terms together, the training algorithm has a big O complexity of:

$$O(N + [|Y| * (e n \log n)] + [p \log n] + [e p \log p]) \tag{58}$$

which accounts for both the cost of creating the *k*-d trees and querying them, as well as creating the *k*-d tree for fallback classification.

The new training algorithm works faster and more efficiently than the previous version of the algorithm. The previous training algorithm handled the comparison between a proposed antibody and all training samples naively, performing a comparison between each sample and the proposed antibody. This comparison required the calculation of the distance function, and was performed even when the sample was guaranteed to not fall into the proposed antibody.

Big O of the Optimized Classification Algorithm

The classification algorithm is linear on the number of antibodies present in the population. The analysis is split up into two parts. The primary filtering process proceeds dimension by dimension, therefore:

$$O(d * |A_{primary}|) \quad (59)$$

where d is the number of dimensions of the antibody centers, and $A_{primary}$ is the set of antibodies present in the population before primary filtering begins. Furthermore, the size of the antibody population decreases with each dimension processed, meaning that the primary filtering takes less and less time as it executes. This is one of the reasons why it is much faster than the original algorithm, even though the filtering process is still linear, like the original algorithm.

The secondary filtering process is linear on the number of antibodies remaining in the population after the primary filtering is done:

$$O(e * |A_{secondary}|) \quad (60)$$

where e is the number of dimensions of the dataset, and $A_{secondary}$ is the set of antibodies present in the population before secondary filtering begins and after primary filtering is done.

Lastly, the assignment of the class to the test point is done in two ways. If the point is within one or more antibodies, then majority voting is performed, otherwise a query to a *k*-d tree is done. The time taken for final classification is described by the function h, parametrized by the set of antibodies A_{final}, and the size of the original set of antibodies N. Final classification takes:

$$h(A_{final}, M) = \begin{cases} |A_{final}| & if\ \exists a\ \{a | a \in A_{final}, d(a[center], x) \leq a[radius]\} \\ \log M & otherwise \end{cases} \quad (61)$$

where A_{final} is the set of antibodies remaining in the set after secondary filtering is completed, and M is the size of the original antibody population. The distance function used in the classification is shown as d(), the sample to be classified is shown as x. The majority voting step of the algorithm is linear on the number of antibodies left in the population after the filtering steps are finished.

Combining the three costs, the big O complexity of the optimized classification algorithm becomes:

$$O([d * |A_{primary}|] + e * |A_{secondary}| + h(A_{final}, M)) \quad (62)$$

As expected, the algorithm is still linear on the number of antibodies in the population, and any speedup present is due to the efficient implementation of the filtering, which is usually faster than the previous implementation of the classification algorithm. However, the new classification algorithm will be faster than the previous classification algorithm in almost all cases.

Memory Requirements of the Optimized Algorithm

The memory requirements of the algorithm can be broken down into four categories: the memory required to store the data set, the memory required to store the antibody population, the memory required by the training algorithm to perform its calculations, and the memory required by the classification algorithm to make a prediction. In this section, the variable i is the number of bytes required to store an integer, k is the number of dimensions present in the data set, and f is the number of bytes required to store a floating point number

The memory required to store the data set used during the training algorithm can be calculated as:

$$|S| * [i + (e * f)] + (|S| * 2 * p) \quad (63)$$

where S is the data set. This amount of memory is required when the class label is encoded as an integer, which simplifies the calculation. Lastly, the amount of memory used for pointers is calculated, where each node in the tree contains two pointers and the amount of memory required to store a pointer is p.

The antibody population memory requirement is very simple to calculate and depends on the size of the population and the number of dimensions of the data set:

$$(|A| * [i + (e * f) + f]) \tag{64}$$

where A is the set of antibodies. A floating point number is used to store the radius of the hyper-sphere. Again, this amount of memory is required only when the class label is encoded as an integer.

The initialization step of the algorithm involves a simple normalization step, which can be done in-place, requiring no more memory than it takes to store the highest and lowest values of each variable in the data set:

$$e * 2f \tag{65}$$

where e is the number of dimensions of the data set.

The training algorithm, which generates the population of antibodies requires no memory to run, since the memory it uses is already accounted for in the above analysis of the antibody population.

The classification algorithm requires memory to perform the filtering steps. However, by encoding the structure of the k-d tree used for fallback classification on the array storing the population of antibodies, it is possible to avoid storing the antibody population twice. The amount of memory required to perform primary filtering never exceeds:

$$|A_{primary}| * [\ i + (\ e * f\ ) + f\ ] \quad (66)$$

where $A_{primary}$ is the set of antibodies present in the population before primary filtering starts. This is due to the fact that the antibody population is copied every time primary filtering is performed on one dimension. However, the filtering seeks to remove portions of the population at each step, so the amount of memory required is always guaranteed to be equal or smaller than the original antibody population.

Secondary filtering can be done in place, and does not require any memory, the copied antibody population is guaranteed to never exceed:

$$|A_{secondary}| * [\ i + (\ e * f\ ) + f\ ] \quad (67)$$

where $A_{secondary}$ is the set of antibodies present in the population before secondary filtering starts, but after primary filtering is done. The majority voting step only requires a set of counter variables, which keep track of the number of times each class appears in the population that remains after primary and secondary filtering is done:

$$|Y| * i \quad (68)$$

where Y is the set of classes present in the data set.

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

This work has sought to improve Artificial Immune System algorithms in several ways. First, a new technique for doing multi-class classification with Artificial Immune Systems was developed and tested. Second, an optimization for this technique was developed and tested. Third, a similar optimization to the one applied to the multi-class classification algorithm was developed and tested for the binary-classification negative selection algorithm. Lastly, the second algorithm was analyzed and a bound for the accuracy that the algorithm is able to achieve was found.

When developing the multi-class AIS algorithms it was attempted to improve the classification accuracy of the algorithm with kernel functions, and although the highest accuracy was achieved with a kernel function, it is believed that kernel functions do not significantly improve the algorithm. The algorithm is also directly compared with SVM and Naive Bayes classifiers. The algorithm's accuracy is similar to the accuracy of other algorithms tested. Even though the algorithm is useful in any situation where classification is performed, certain features of the algorithm make it especially useful in resource-limited systems such as IoT applications, specifically, the algorithm's ability to generalize well from small training sets, as well as its insensitivity to kernel functions. In short, it is possible to improve on the accuracy of Naïve Bayes and SVM classifiers when used with small training sets, but the training and classification steps of our AIS algorithm are slower than these algorithms. Also, the algorithm could be easily modified to work in parallel processors such as GPUs, greatly increasing its performance.

The optimized positive-selection AIS algorithm has shown that it is indeed possible to optimize the performance of an AIS algorithm. Also, the algorithm is functionally identical to the previous unoptimized version. Moreover, a bound was drawn on the generalization error of the algorithm, basing it on previous work in the field of ensemble classifiers. No other research has been found pertaining to the application of bounds to artificial immune system classification algorithms. In this work, it has been shown that the most common distance measure used, the Euclidian distance measure, does not give the best performance for this problem.

We have found that the algorithm performed as well as other algorithms, and was insensitive to input parameters. It is also very simple to implement, and generalizes well from small training data sets. The asymptotic complexity of the optimized algorithm and a bound on the generalization error of the algorithm are given. In this paper it has been shown how the changes and optimizations applied to the original algorithm do not functionally change the original algorithm, while making its execution much faster.

In the analysis chapter of this work, it has been shown how the quality of the margin of each individual classifier affects the final performance of an ensemble classifier. Our classifier is very naïve in its approach to selecting the placement of it base classifiers in the feature space, using a very simple strategy. This is done this way to make the training faster, and good classification performance has been achieved in spite of this simplicity. Future research can be done on different ways to optimize the margin of each base classifier during the training portion of the algorithm by optimizing the placement of its center as well as it radius. Furthermore, the scheme used to set the radius of each antibody is also very simple, since it is an approximation of the class boundary and does not take into account the generalization ability of the algorithm. It might be interesting to try

other ways of finding an optimal antibody radius which allow an antibody to misclassify examples, but increase the generalization ability of the classifier.

Since beginning work on this AIS-inspired algorithm is has been found to be particularly insensitive to the choice in parameters. Unlike other classification algorithms, this algorithm does not seem need to have its parameters set up perfectly to give good performance. An open question for us is: how insensitive is the algorithm to the values of the parameters given to it? Another interesting research direction would be to apply a similar bound to the negative selection algorithm, using hyper-spheres as base classifiers. Furthermore, a bound could be drawn on this and other artificial immune system algorithms that use base classifiers that are not hyper-spheres.

The optimized negative selection algorithm could be improved in several ways. First, the complexity of the algorithm remains in the same class, although the constants are decreased significantly. Second, the optimization is only applicable to detectors in which each dimension can be evaluated individually, and which allow the set of data points or detectors to be filtered. That is, the optimization works on detectors types that allow a data point or detector to be taken out of the set if it does not match in one individual dimension, this is not always possible. Third, the density of the points in the data set can have a significant effect on the performance of the optimized training algorithm. In future research, this optimization scheme could be applied to other data sets to highlight the effect that the density of the data set has on the performance of the optimized algorithm. Lastly, although the optimization has been demonstrated experimentally to not affect the accuracy of the algorithm, this has not been proven formally. Future research could be completed to provide a formal proof of the optimized Negative Selection algorithm's equivalency with the unoptimized Negative Selection algorithm. Future work can also be done in

the application of the optimization to more complex data sets and exploring the performance of the optimized algorithm.

REFERENCES

[1] L. N. De Castro and J. Timmis, *Artificial immune systems: a new computational intelligence approach*. Springer Science & Business Media, New York, New York, 2002.

[2] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks,* vol. 54, no. 15, 2010, pp. 2787-2805.

[3] S. Marsland, *Machine Learning: An Algorithmic Perspective.* Chapman & Hall/CRC, London, UK, 2009.

[4] D. Michie, D. J. Spiegelhalter and C. C. Taylor, *Machine learning, neural and statistical classification.* Overseas Press, New York, New York, 2009.

[5] M. Aly, "Survey on multiclass classification methods," in *Neural Networks,* 2005, pp.1-9.

[6] H. Kim, K. C. Claffy, M. Fomenkov, D. Barman, M. Faloutsos and K. Lee, "Internet traffic classification demystified: myths, caveats, and the best practices," in *Proc. of the 2008 ACM CoNEXT Conf.*, Madrid, Spain, 2008, pp. 11.

[7] A. W. Moore and K. Papagiannaki, "Toward the accurate identification of network applications," in *6th International Workshop on Passive and Active Network Measurement*, Boston, MA, USA, 2005, pp. 41-54.

[8] S. Forrest, A. Perelson, L. Allen, and R. Cherukuri, "Self-nonself discrimination in a computer," in *Proc. of the 1994 IEEE Computer Society Symp. on Research in Security and Privacy*, Oakland, CA, 1994, pp. 202-212.

[9] D. Dasgupta and F. Nino, *Immunological computation: theory and applications*. CRC Press, London, UK, 2008.

[10] L. N. De Castro and F. J. Von Zuben, "Learning and optimization using the clonal selection principle," *IEEE Transactions on Evolutionary Computation,* vol. 6, num. 3, 2002, pp. 239-251.

[11] P. Matzinger, "Tolerance, danger, and the extended family," in *Annu. Review of Immunology,* vol. 12, no. 1, 1994, pp. 991-1045.

[12] U. Aickelin and S. Cayzer, "The Danger Theory and Its Application to Artificial Immune Systems," in *Proc. of the 1st Int. Conf. on Artificial Immune Systems*, Canterbury, UK, 2002, pp. 141-148.

[13] J. Kim, P. J. Bentley, U. Aickelin, J. Greensmith, G. Tedesco and J. Twycross, "Immune system approaches to intrusion detection–a review," *Natural Computing*, 2007, pp. 413-466.

[14] D. Dasgupta, “Artificial immune systems and their applications,” vol. 1. Berlin: Springer, 1999.

[15] K. K. Kalmanje, and J. Neidhoefer, "Immunized adaptive critic for an autonomous aircraft control application," *Artificial Immune Systems and Their Applications*, 1999, pp. 221-241.

[16] T. Fukuda, K. Mori, and M. Tsukiyama, "Immunity-Based Management System for a Semiconductor Production Line*," *Artificial Immune Systems and Their Applications*, 1999, pp. 278-288.

[17] P. K. Harmer, P. D. Williams, G. H. Gunsch, and G. B. Lamont, "An artificial immune system architecture for computer security applications," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 3, 2002, pp. 252-280.

[18] F. A. González, and D. Dasgupta, "Anomaly detection using real-valued negative selection," *Genetic Programming and Evolvable Machines,* vol. 4, no. 4, 2003, pp. 383-403.

[19] Y. Ishida, "An immune network model and its applications to process diagnosis," *Systems and Computers in Japan,* vol. 24, no. 6, 1993, pp. 38-46.

[20] J. Timmis, M. Neal and J. Hunt, "Data analysis using artificial immune systems, cluster analysis and Kohonen networks: some comparisons," in *IEEE Int. Conf. on Systems, Man, and Cybernetics,* Tokyo, Japan, 1999, pp. 922-927.

[22] H. Bersini, "Immune network and adaptive control," in *Proceedings of the first European Conference on Artificial Life,* Bradford Books, MIT Press, Cambridge, MA, pp. 217-226.

[23] A. Ishiguro, T. Kondo, Y. Watanabe, Y. Shirai and Y. Uchikawa, "Emergent construction of artificial immune networks for autonomous mobile robots," in *1997 IEEE Int. Conf. on Computational Cybernetics and Simulation*, Orlando, FL, 1997, pp. 1222-1228.

[24] X. Wang, X. Zhi Gao and S. J. Ovaska. "Artificial immune optimization methods and applications-a survey," in *2004 IEEE Int. Conf. on Systems, Man and Cybernetics,* The Hague, Netherlands, 2004, pp. 3415-3420.

[25] U. Aickelin, J. Greensmith, and J. Twycross. "Immune system approaches to intrusion detection–a review," *Natural Computing*, vol. 6, num. 4, 2007, pp. 413-466.

[26] S. A. Hofmeyer, and S. Forrest, "Immunity by design: An artificial immune system," in *Proc. of the Genetic and Evolutionary Computation Conf.*, Orlando, FL, 1999, pp. 1289-1296.

[27] J. Kim, and P. Bentley, “Negative selection and niching by an artificial immune system for network intrusion detection,” in *Late Breaking Papers at the 1999 Genetic and Evolutionary Computation Conf.*, Orlando, FL, 1999, pp. 149-158.

[28] J. Kim, and P. Bentley, “Negative Selection within an Artificial Immune System for Network Intrusion Detection,” in *14th Annu. Fall Symp. of the Korean Information Processing Society*, Seoul, Korea, 2000.

[29] J. Kim, and P. Bentley, “Evaluating Negative Selection in an Artificial Immune System for Network Intrusion Detection,” in *Genetic and Evolutionary Computation Conf. (GECCO-2001)*, San Francisco, 2001, pp.1330-1337.

[30] J. Kim, and P. Bentley, “The Artificial Immune System for Network Intrusion Detection: An Investigation of Clonal Selection with a Negative Selection Operator,” in the *Congr. on Evolutionary Computation*, Seoul, S. Korea, 2001, pp.1244-1252.

[31] D. Dasgupta, and F. González, "An immunity-based technique to characterize intrusions in computer networks," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 3, 2002, pp. 281-291.

[32] F. González, and D. Dasgupta, “Anomaly detection using real-valued negative selection,“ *Genetic Programming and Evolvable Machines*, vol. 4, no. 4, 2003, pp. 383-403.

[33] J. Kephart, “A biologically inspired immune system for computers,“ in *Artificial Life IV: Proc. of the 4th Int. workshop on the synthesis and simulation of living systems*, MIT Press, Cambridge, MA, 1994, pp. 130-139.

[34] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proc. of the IEEE Symp. on Security and Privacy,* Oakland, CA, 1996, pp. 120-128.

[35] V. D. Kotov, and V. Vasilyev, “Immune model based approach for network intrusion detection,” in *Proc. of the 3rd Int. Conf. on Security of information and networks,* Taganrog, Russia, 2010, pp. 233-237.

[36] Y. He, L. Yiwen, L. Tao, and Z. Ting, “A model of collaborative artificial immune system,” in *2nd Int. Asia Conf. on Informatics in Control, Automation and Robotics,* Funchal, Madeira, Portugal, 2010, pp. 101-104.

[37] K. W. Yeom and J. H. Park, “An immune system inspired approach of collaborative intrusion detection system using mobile agents in wireless ad hoc networks,” *Computational Intelligence and Security,* 2005, pp. 204-211.

[38] A. Boukerche, R. B. Machado, K. R. Jucá, J. B. M. Sobral and M. S. Notare, “An agent based and biological inspired real-time intrusion detection and security model for computer network operations,” *Computer Communications*, vol. 30, no. 13, 2007, pp. 2649-2660.

[39] C. M. Oil, Y. T. Wang and C. R. Ou, "Intrusion detection systems adapted from agent-based artificial immune systems," in *2011 IEEE Int. Conf. on Fuzzy Systems,* Taipei, Taiwan, 2011, pp. 115-122.

[40] C. M. Ou, "Host-based intrusion detection systems adapted from agent-based artificial immune systems," *Neurocomputing*, vol. 88, 2012, pp. 78-86.

[41] H. R. Zeidanloo, F. Hosseinpour and P. N. Borazjani, "Botnet detection based on common network behaviors by utilizing artificial immune system (AIS)," in *2nd Int. Conf. on Software Technology and Engineering*, San Juan, Puerto Rico, USA, 2010, pp. 1-21.

[42] Y. Zhang, L. Wang, W. Sun, R. C. Green and M. Alam, "Artificial immune system based intrusion detection in a distributed hierarchical network architecture of smart grid," in *2011 IEEE Power and Energy Society General Meeting,* Detroit, MI, 2011, pp. 1-8.

[43] M. A. Rassam and M. Maarof, "Artificial Immune Network Clustering approach for Anomaly Intrusion Detection," *Journal of Advances in Information Technology*, vol. 3, no. 3, 2012.

[44] D. Wang, L. He, Y. Xue and Y. Dong, "Exploiting Artificial Immune systems to detect unknown DoS attacks in real-time," in *2012 IEEE 2nd Int. Conf. on Cloud Computing and Intelligent Systems,* Hangzhou, China, 2012, pp. 646-650.

[45] S. Srivastava and A. Mukhopadhyay, "AIS for Intrusion Detection in VoIP over LAN," in *International Conference on Intelligent Infrastructure*, Hong Kong, China, 2013, pp 1-6.

[46] J. Y. Le Boudec and S. Sarafijanović, "An artificial immune system approach to misbehavior detection in mobile ad hoc networks," *Biologically Inspired Approaches to Advanced Information Technology,* 2004, pp. 396-411.

[47] M. Drozda, S. Schaust and H. Szczerbicka, "AIS for misbehavior detection in wireless sensor networks: Performance and design principles," in *IEEE Congr. on Evolutionary Computation,* Singapore, 2007, pp. 3719-3726.

[48] Y. Mohamed and A. Abdullah, "Security Mechanism for MANETs," *Journal of Engineering and Science Technology*, vol. 4, no. 2, 2009, pp. 231-242.

[49] Y. Mohamed and A. Abdullah, "Immune Inspired Approach for Securing Wireless Ad hoc Networks," *Int. Journal of Computer Science and Security*, vol. 9, no. 7, 2009, pp. 206-212.

[50] Y. Liu and F. Yu, "Immunity-based intrusion detection for wireless sensor networks," in *IEEE International Joint Conference on Neural Networks,* Hong Kong, China, 2008, pp. 439-444.

[51] H. Yang, J. Guo F. and Deng, "Collaborative RFID intrusion detection with an artificial immune system," *Journal of Intelligent Information Systems*, vol. 36, no. 1, 2011, pp. 1-26.

[52] A. W. Moore and D. Zuev, “Internet traffic classification using bayesian analysis techniques,” in *ACM SIGMETRICS Performance Evaluation Review*, New York, New York, 2005, pp. 50-60.

[53] T. T. Nguyen, and G. Armitage, “A survey of techniques for internet traffic classification using machine learning,” *Communications Surveys & Tutorials*, 10(4), 2008, pp. 56-76.

[54] R. Alshammari and A. N. Zincir-Heywood, "Machine learning based encrypted traffic classification: Identifying ssh and skype," in *IEEE Symp. on Computational Intelligence for Security and Defense Applications*, Ottawa, ON, Canada, 2009, pp. 1-8.

[55] K. Singh and S. Agrawal, “Comparative analysis of five machine learning algorithms for IP traffic classification,” in *2011 Int. Conf. on Emerging Trends in Networks and Computer Communications,* Udai Udaipur, Rajasthan, India, 2011, pp. 33-38.

[56] L. Jun, Z. Shunyi, L. Shidong and X. Ye, "P2P traffic identification technique," in *2007 Int. Conf. on Computational Intelligence and Security*, Harbin, China, 2007, pp. 37-41.

[57] Y. S. Lim, H. C. Kim, J. Jeong, C. K. Kim, T. T. Kwon and Y. Choi, “Internet traffic classification demystified: on the sources of the discriminative power,” in *Proc. of the 7th International Conference on emerging Networking Experiments and Technologies*, Tokyo, Japan, 2011, pp. 9-20.

[58] T. S. Tabatabaei, F. Karray, and M. Kamel, "Early internet traffic recognition based on machine learning methods," in *25th IEEE Canadian Conf. on Electrical & Computer Engineering*, Montreal, QC, Canada, 2012, pp. 1-5.

[59] M. S. Aliakbarian, A. Fanian, F. S. Saleh, and T. A. Gulliver, "Optimal supervised feature extraction in internet traffic classification," in *2013 IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing*, Victoria, BC, Canada, 2013, pp. 102-107.

[60] F. Ertam, and E. Avci “Classification with Intelligent Systems for Internet Traffic in Enterprise Networks,” in *International Journal of Computing, Communications & Instrumentation Engineering* (IJCCIE), Vol. 3, pp. 2349-1469, 2013.

[61] F. Ertam, and E. Avcı, “Network Traffic Classification via Kernel Based Extreme Learning Machine”, in *International Journal of Intelligent Systems and Applications in Engineering* (IJISAE), Vol 4 (Special Issue), pp. 109–113, 2016.

[62] R. Raveendran, and R. R. Menon, “A novel aggregated statistical feature based accurate classification for internet traffic,” in *International Conference on Data Mining and Advanced Computing* (SAPIENCE), pp. 225-232, IEEE, 2016.

[63] P. Casas, A. D'Alconzo, P. Fiadino, and C. Callegari, “Detecting and diagnosing anomalies in

cellular networks using Random Neural Networks," in *International Wireless Communications and Mobile Computing Conference* (IWCMC), pp. 351-356, IEEE, 2016.

[64] M. Shafiq, X. Yu, and A. A. Laghari, "WeChat Text Messages Service Flow Traffic Classification Using Machine Learning Technique", in *6th International Conference on IT Convergence and Security* (ICITCS), pp. 1-5, IEEE, 2016.

[65] S. V. Westlinder, "Video Traffic Classification: A Machine Learning approach with Packet Based Features using Support Vector Machine", Master's Thesis, Karlstad University, Faculty of Health, Science and Technology, Department of Mathematics and Computer Science, 2016.

[66] D. E. Goodman, L. Boggess and A. Watkins, "Artificial immune system classification of multiple-class problems," in *Proc. of the Artificial Neural Networks in Engineering Conference*, St. Louis, MO, 2002, pp. 179-183.

[67] J. Timmis and M. Neal, "Investigating the evolution and stability of a resource limited artificial immune system," in *Proc. of the Genetic and Evolutionary Computation Conf.*, Las Vegas, NV, 2000, pp. 40-41.

[68] J. Timmis and M. Neal, "A resource limited artificial immune system for data analysis," *Knowledge-Based Systems*, vol. 14, no. 3, 2001, pp. 121-130.

[69] A. Watkins, "AIRS: A Resource Limited Artificial Immune Classifier," M.S. thesis, Department of Computer Science. Mississippi State University, MS, 2001.

[70] A. Watkins and L. Boggess, "A New Classifier Based on Resource Limited Artificial Immune Systems," in *Proceedings of Congress on Evolutionary Computation,* Honolulu, HI, 2002, pp. 1546-1551.

[71] A. Watkins and L. Boggess, "A Resource Limited Artificial Immune Classifier," in *Proc. of the 2002 Congr. on Evolutionary Computation*, Honolulu, HI, 2002, pp. 926-931.

[72] H. P. Cheng and C. S. Cheng, "A hybrid multiclass classifier based on artificial immune algorithm and support vector machine," in *3rd Int. Conf. on Data Mining and Intelligent Information Technology Applications*, Macau, China, 2011, pp. 46-50.

[73] J. Greensmith and S. Cayzer, "An artificial immune system approach to semantic document classification," *Artificial Immune Systems*, 2003, pp. 136-146.

[74] J. H. Carter, "The immune system as a model for pattern recognition and classification," *Journal of the American Medical Informatics Association*, vol. 7, no. 1, 2000, pp. 28-41.

[75] J. A. White and S. M. Garrett, "Improved pattern recognition with artificial clonal selection?," *Artificial Immune Systems*, 2003, pp. 181-193.

[76] J. Brownlee, “Clonal selection theory and CLONALG - the clonal selection classification algorithm (CSCA),” Centre for Intelligent Systems and Complex Processes, Faculty of Information and Communication Technologies, Swinburne
University of Technology, Victoria, Australia, Tech. Report 2-01, 2005.

[77] U. Markowska-Kaczmar and B. Kordas, “Multi-class iteratively refined negative selection classifier,” *Applied Soft Computing,* vol. 8, no. 2, 2008, pp. 972-984.

[78] U. Markowska-Kaczmar and B. Kordas, “Negative Selection based method for Multi-Class problem classification,” in *6th Int. Conf. on Intelligent Systems Design and Applications*, Jian, China, 2006, pp. 1165-1170.

[79] T. Stibor, J. Timmis and C. Eckert, “On the use of hyperspheres in artificial immune systems as antibody recognition regions,” *Artificial Immune Systems,* 2006, pp. 215-228.

[80] J. M. Shapiro, G. B. Lamont, and G. L. Peterson, “An evolutionary algorithm to generate hyper-ellipsoid detectors for negative selection,” in *Proceedings of GECCO*, Washington, D.C., USA, 2005, pp. 337-344.

[81] H.E. Shen, L. Wen-Jian, W. Xu-Fa, “A Negative Selection Algorithm with the Variable Length Detector,” *Journal of Software*, vol. 18, no. 16, 2007, pp.1361-1368.

[82] A. Chmielewski and S. T. Wierzchoń, “On the distance norms for detecting anomalies in multidimensional datasets,” in *Zeszyty Naukowe Politechniki Białostockiej. Informatyka*, 2007, pp. 39-49.

[83] J. Kim, P. J. Bentley, U. Aickelin, J. Greensmith, G. Tedesco and J. Twycross, “Immune system approaches to intrusion detection–a review,” in *Natural Computing*, vol. *6*, no. 4, 2007, pp. 413-466.

[84] K. B. Sim, and D. W. Lee, "Modeling of positive selection for the development of a computer immune system and a self-recognition algorithm," *Int. Journal of Control, Automation, and Systems*, vol. 1, no. 4, 2003, pp. 453-458.

[85] M. Middlemiss, “Positive and negative selection in a multilayer artificial immune system,” Information Science Discussion Papers Series, Department of Computer Science, University of Otago, New Zealand, Report No. 2006/03.

[86] Z. Fuyong and Q. Deyu, "Run-time malware detection based on positive selection," *Journal in Computer Virology*, vol. 7, no. 4, 2011, pp. 267-277.

[87] Z. Fuyong, and D. Qi, "A Positive Selection Algorithm for Classification," *Journal of Computational Information Systems*, vol. 8, no. 1, 2012, pp. 207-215.

[88] S. Yu, and D. Dasgupta, “An effective network-based intrusion detection using conserved self pattern recognition algorithm augmented with near-deterministic detector generation,” in *2011 IEEE Symp. on Computational Intelligence in Cyber Security,* Paris, France, 2011, pp. 17-24.

[89] M. Elberfeld and J. Textor. "Efficient algorithms for string-based negative selection," in *Artificial Immune Systems*, 2009, pp. 109-121.

[90] M. Liskiewicz and J. Textor, “Negative Selection Algorithms without Generating Detectors,” in *Proc. of the 12th Annu. Conf. on Genetic and Evolutionary Computation*, Portland, OR, 2010, pp. 1047-1054.

[91] M. Elberfeld and J. Textor, “Negative Selection Algorithms on Strings with Efficient Training and Linear-Time Classification,” *Theoretical Computer Science*, vol. 412, no. 6, 2011, pp. 534-542.

[92] J. Textor, "Efficient negative selection algorithms by sampling and approximate counting," *Parallel Problem Solving from Nature*, 2012, pp. 32-41.

[93] D. Wang, Y. Xue and D. Yingfei, "Anomaly Detection Using Neighborhood Negative Selection," *Intelligent Automation & Soft Computing,* vol. 17, no. 5, 2011, pp. 595-605.

[94] D. Wang, Y. Xue and D. Yingfei, "NNS: A Novel Neighborhood Negative Selection algorithm," in *World Automation Congr.,* 2012, Puerto Vallarta, Mexico, pp. 453-457.

[95] T. Yang, H. L. Deng, W. Chen and Z. Wang, "GF-NSA: A Negative Selection Algorithm Based on Self Grid File," *Applied Mechanics and Materials,* vol. 44, 2011, pp. 3200-3203.

[96] C. Wen, D. Ming, L. Tao and Y. Tao, "Negative selection algorithm based on grid file of the feature space," *Knowledge-Based Systems,* vol. 56, 2014, pp. 26-35.
[97] Z. Ji and D. Dasgupta. "V-detector: An efficient negative selection algorithm with “probably adequate” detector coverage," *Information sciences,* vol. 179, no. 10, 2009, pp. 1390-1406.

[98] B. Schmidt, D. Kountanis, A. Al-Fuqaha, “A Biologically-Inspired Approach to Network Traffic Classification for Resource-Constrained Systems,” in *Int. Symp. on Big Data Computing*, London, UK, 2014. © 2014 IEEE.

[99] B. Schmidt, D. Kountanis, A. Al-Fuqaha, “Artificial Immune System Inspired Algorithm for Flow-Based Internet Traffic Classification,” in *IEEE CloudCom 2014*, Big Data Track, Singapore, 2014. © 2014 IEEE.

[100] T. S. Guzella, T. A. Mota-Santos and W. M. Caminhas, “Artificial immune systems and kernel methods,” *Artificial Immune Systems*, 2008, pp. 303-315.

[101] Schmidt, B., Al-Fuqaha, A., Gupta, A., & Kountanis, D, “Optimizing an Artificial Immune System Algorithm in Support of Flow-Based Internet Traffic Classification.,” *Applied Soft Computing*, © IEEE, 2017

[102] J. S. Hamaker and L. Boggess, “Non-euclidean distance measures in AIRS, an artificial immune classification system,” In *Congr. on Evolutionary Computation*, Portland, OR, 2004, pp. 1067-1073.

[103] P. Laskov, "Feasible direction decomposition algorithms for training support vector machines," *Machine Learning,* vol. 46, no. 1-3, 2002, pp. 315-349.

[104] Schmidt, B., and Al-Fuqaha, A. “A new approach to optimized negative selection,” In *IEEE Congress on Evolutionary Computation*, 2016, pp. 1793-1799, © IEEE, 2016

[105] K. Bache and M. Lichman, UCI Machine Learning Repository [http://archive.ics.uci.edu/ml], Irvine, CA: University of California, School of Information and Computer Science. Accessed December 3, 2014.

[106] V. Vapnik, “*The nature of statistical learning theory,”* Springer, 2000.

[107] From the lecture notes of CS4850, Section 10, Spring 2010, “Mathematical Foundations for the Information Age,” Cornell University, http://www.cs.cornell.edu/Courses/cs4850/2010sp/Course %20Notes/Chap%206%20Learning-march_9_2010.pdf

[108] R. Schapire and Y. Freund. “Boosting the margin: a new explanation for the effectiveness of voting methods,” Annals of Statistics, no. 26, pp. 322-330, 1998.

[109] L. Breiman, "Bagging predictors." in *Machine learning*, vol. 24, no. 2, pp. 123-140, 1996

[110] L. Breiman, "Random forests," in *Machine learning*, no. 45, vol. 1, pp. 5-32, 2001

[111] R. Schapire, “The strength of weak learnability,” in *Machine Learning*, no. 5, vol. 2, pp. 197–227, 1990.

[112] Q. Cai, Z. Changshui, and P. Chunyi, "Analysis of classification margin for classification accuracy with applications," Neurocomputing, vol. 72, no. 7, pp. 1960-1968, 2000

[113] Y. Freund, R. Schapire, and N. Abe. "A short introduction to boosting," in Journal-Japanese Society For Artificial Intelligence no. 14, pp. 771-780, 1999

[114] R. Schapire, Y. Freund, P. Bartlett, and W.S. Lee, “Boosting the Margin: A New Explanation for the Effectiveness of Voting Methods,” in Annals of Statistics, vol. 26, no. 5, pp. 1651-1686, 1998.

[115] T. Cover, and P. Hart. "Nearest neighbor pattern classification," in IEEE Transactions on Information Theory, vol. 13, no. 1, pp. 21-27, 1967

[116] L. Devroye, "On the almost everywhere convergence of nonparametric regression function estimates," in The Annals of Statistics, vol. 9, no. 6, pp. 1310-1319, 1981

[117] V. Feldman, V. Guruswami, P. Raghavendra, and Y. Wu, "Agnostic learning of monomials by halfspaces is hard," in *SIAM Journal on Computing*, vol. 41, no. 6, pp. 1558-1590, 2012

[118] V. Vapnik, and A. Chervonenkis. "Theory of pattern recognition." (in Russian). Nauka, Moscow, 1974

APPENDIX A

PYTHON CODE FOR MULTI-CLASS AIS ALGORITHM

```
import sys as sys
import random as random
import math as math
from collections import defaultdict
from operator import itemgetter
import csv as csv
from random import choice
from os import listdir
from os.path import isfile, join
from random import shuffle
import copy as copy
import time as time
import datetime as datetime

#this function is for importing data
def getdata(file_name):
    with open(file_name, 'r') as f:
        rowdata = []
        reader = csv.reader(f)
        for row in reader:
            rowdata.append(row)
    return rowdata

def proportion_per_class(data):
    prop = {}
    for d in data:
        if d[0] not in prop.keys():
            prop[d[0]] = 1
        else:
            prop[d[0]] = prop[d[0]] + 1
    for k in prop:
        prop[k] = float(prop[k]) / float(len(data))
    return prop

def get_class_labels(data):
    classes = []
    for i in data:
        if i[0] not in classes:
            classes.append(i[0])
    return classes

def normalize(data):
    #cycling through each feature, but not the class label
    for i in range(1, len(data[0])):
        lowest = 100000000000000000
        highest = -100000000000000000
        for j in data:
            if float(j[i]) < lowest:
                lowest = float(j[i])
```

```
            if float(j[i]) > highest:
                highest = float(j[i])
        #now that we have the highest and lowest values, we can calculate the
normalized value
        for j in data:
            if highest == lowest:
                j[i] = 0.5
            else:
                j[i] = (float(j[i]) - lowest)/(highest-lowest)
    return data

#this function is to create a stratified folded data set from a normal datase
def stratify(data, folds):
    #building a dictionary to hold all data by class which is in data[0][0]
    classes = {}
    #splitting data into classes
    for d in data:
        if d[0] not in classes:
            classes[d[0]] = []
            classes[d[0]].append(d)
        else:
            classes[d[0]].append(d)

    # n-fold stratified samples
    data = []
    for r in range(folds):
        data.append([])
    #spreading the classes evenly into all data sets
    for key,items in classes.items():
        for i in range(len(items)):
            data[i%folds].append(items[i])
    return data


def distance(x1, x2, parameters):
    if len(x1) != len(x2):
      return False
    else:
        distance = 0
        for i in range(1,len(x1)):
            distance = distance + ((float(x1[i]) - float(x2[i]))**2)
        distance = math.sqrt(distance)
        return distance


#these functions are used for doing prediction with the population of
antibodies
def distance(x1, x2, parameters):
    if len(x1) != len(x2):
      return False
    else:
        distance = 0
        for i in range(1,len(x1)):
            distance = distance + ((float(x1[i]) - float(x2[i]))**2)
```

```
        distance = math.sqrt(distance)
        return distance

#testing the prediction performance
def test_fmeasure(antibodies, test_data,class_label, parameters):
    TP, TN, FP, FN = 0, 0, 0, 0
    for x in test_data:
        if x[0] == class_label:
            yhat = predict(antibodies, x, parameters)
            if x[0] == yhat:
                TP += 1
            else:
                FN += 1
        else:
            yhat = predict(antibodies, x, parameters)
            if yhat == class_label:
                FP += 1
            else:
                TN += 1

    #print TP, TN, FP, FN
    if float(TP+FP) != 0:
        precision = float(TP) / float(TP+FP)
    else:
        precision = 0.0
    if float(TP+FN) != 0:
        recall = float(TP) / float(TP+FN)
    else:
        recall = 0.0
    if (precision + recall) != 0:
        fmeasure = 2*((precision*recall)/(precision + recall))
    else:
        fmeasure = 0.0

    #return "TP: " + str(TP)+ " TN: "+ str(TN)+ " FP: "+ str(FP)+ " FN: "+
str(FN)
    return [precision, recall, fmeasure]

#testing the prediction performance
def test_accuracy(antibodies, test_data, parameters):
    error_count = 0
    correct_count = 0
    for x in test_data:
        yhat = predict(antibodies, x, parameters)
        if x[0] != yhat:
            error_count = error_count + 1
        else:
            correct_count = correct_count + 1
        #print "predicted: ", yhat, " actual: ", x[0]
    return float(correct_count) / float(len(test_data))

#vote for the best classification
def predict(antibodies, x, parameters):
    distances = []
```

```
    for a in antibodies:
        d = distance(x, a[0], parameters)
        if d <= a[1]:
            return a[0][0]
        else:
            distances.append([a[0][0], d])
    distances.sort(key=itemgetter(1))
    return distances[0][0]

def error_count(antibody, training_set, parameters):
    error_count = 0
    class_data = [i for i in training_set if i[0] != antibody[0][0]]
    for t in class_data:
        if distance(t, antibody[0], parameters) <= antibody[1]:
            error_count = error_count + 1
    return error_count

def generate_population(training_set, classes, size, parameters):
    antibodies = []
    #select random antibodies from the self class, and add with a radius of 0
    for c in classes:
        class_data = [i for i in training_set if i[0] == c]
        num_of_antibodies = int(float(size) / float(len(classes)))
        for i in range(num_of_antibodies):
            proposed_antibody = [choice(class_data), 0.0]
            antibodies.append(proposed_antibody)

    #expand the antibodies by a step size until it misclassify a non-self
point
    for a in antibodies:
        changed = True
        while changed:
            if error_count(a, training_set, parameters) > 0:
                a[1] = a[1] - parameters["step_size"]
                changed = False
            else:
                a[1] = a[1] + parameters["step_size"]
                changed = True

    return antibodies

#structure of antibody: [ [class, x1, x2, x3,... ], radius]
files = [ f for f in listdir("C:/Users/Brian/Documents/IPython
Notebooks/network_data/") ]
original_data = []
for f in files:
    original_data = original_data + getdata(
"C:/Users/Brian/Documents/IPython Notebooks/network_data/" + f )

classes = get_class_labels(original_data)

set_size = 1000

parameters = {}
```

```
parameters["step_size"] = 0.01
parameters["c"] = 0.08
parameters["d"] = 2

print ("Fig 2. Classification accuracy with data set size held at 1000
samples")
print("set_size \t popsize \t accuracy")
for pop_size in range(100, 1100, 50):
    average_accuracy = 0.0

    #building a balanced data set
    data = []
    for c in classes:
        class_data = [d for d in original_data if d[0] == c]
        shuffle(class_data)
        data = data + class_data[:int( float(1000) / float(len(classes)))]

    data = normalize(data)
    data = stratify(data, 10)

    for st in range(len(data)):
        test_set = data[st%len(data)] + data[(st+1)%len(data)] +
data[(st+2)%len(data)]
        training_set = []
        for tsp in range(len(data)-1):
            training_set = training_set + data[(st+3+tsp)%len(data)]

        antibodies = generate_population( training_set, classes, pop_size,
parameters)
        accuracy = test_accuracy(antibodies, test_set, parameters)
        average_accuracy = average_accuracy + accuracy

    print( "1000 \t ", pop_size, " \t ", average_accuracy/10.0 )
print("")
print("")

print("Fig 3.  Classification accuracy and data set size 200-1000")
# average accuracy over sample size, three training methods, x=# of flows,
y=average accuracy, three training methods, three lines
print("set_size \t popsize \t accuracy")
for sample_size in range(100, 1100, 50):
    average_accuracy = 0.0

    #building a balanced data set
    data = []
    for c in classes:
        class_data = [d for d in original_data if d[0] == c]
        shuffle(class_data)
        data = data + class_data[:int( float(sample_size) /
float(len(classes)))]

    data = normalize(data)
    data = stratify(data, 10)
```

```
    for st in range(len(data)):
        test_set = data[st%len(data)] + data[(st+1)%len(data)] +
data[(st+2)%len(data)]
        training_set = []
        for tsp in range(len(data)-1):
            training_set = training_set + data[(st+3+tsp)%len(data)]

        antibodies = generate_population( training_set, classes, 1000,
parameters)
        accuracy = test_accuracy(antibodies, test_set, parameters)
        average_accuracy = average_accuracy + accuracy

    print( sample_size, " \t 1000 \t ", average_accuracy/10.0 )
print("")
print("")

print ("Fig 4. Classification time with data set size at 1000")
print("set_size \t popsize \t classification time")
for pop_size in range(100, 1100, 50):

    average_time = 0.0

    #building a balanced data set
    data = []
    for c in classes:
        class_data = [d for d in original_data if d[0] == c]
        shuffle(class_data)
        data = data + class_data[:int( float(1000) / float(len(classes)))]

    data = normalize(data)
    data = stratify(data, 10)

    for st in range(10):
        test_set = data[st%len(data)] + data[(st+1)%len(data)] +
data[(st+2)%len(data)]
        training_set = []
        for tsp in range(len(data)-1):
            training_set = training_set + data[(st+3+tsp)%len(data)]

        antibodies = generate_population( training_set, classes, pop_size,
parameters)
        t1 = datetime.datetime.now()
        accuracy = test_accuracy(antibodies, test_set, parameters)
        t2 = datetime.datetime.now()
        average_time = float(average_time) +
float(datetime.timedelta.total_seconds(t2-t1))

    print( " 1000 \t ", pop_size, average_time/10.0 )
print("")
print("")

print("Fig 5. Training time and training dataset size 200-1000")
print("set_size \t popsize \t training time")
for set_size in range(100, 1100, 50):
```

```
    average_time = 0.0

    #building a balanced data set
    data = []
    for c in classes:
        class_data = [d for d in original_data if d[0] == c]
        shuffle(class_data)
        data = data + class_data[:int( float(set_size) /
float(len(classes)))]

    data = normalize(data)
    data = stratify(data, 10)

    for st in range(10):
        test_set = data[st%len(data)] + data[(st+1)%len(data)] +
data[(st+2)%len(data)]
        training_set = []
        for tsp in range(len(data)-1):
            training_set = training_set + data[(st+3+tsp)%len(data)]

        t1 = datetime.datetime.now()
        antibodies = generate_population( training_set, classes, 1000,
parameters)
        t2 = datetime.datetime.now()

        average_time = float(average_time) +
float(datetime.timedelta.total_seconds(t2-t1))

    print( set_size, " \t 1000 \t ", average_time/10.0 )
print("")
```

APPENDIX B

PYTHON CODE FOR OPTIMIZED AIS INTERNET FLOW CLASSIFICATION ALGORITHM

```
import sys as sys
import random as random
import math as math
from collections import defaultdict
from operator import itemgetter
import csv as csv
from random import choice
from os import listdir
from os.path import isfile, join
from random import shuffle
import copy as copy
import time as time
import datetime as datetime
from sklearn import svm

def separate(data):
    labels = []
    dataa = []
    for i in data:
        labels.append(i[0])
        dataa.append(i[1:])
    return [labels, dataa]

def square_distance(pointA, pointB):
    # squared euclidean distance
    distance = 0
    dimensions = len(pointA) # assumes both points have the same dimensions
    for dimension in range(dimensions):
        distance += (pointA[dimension] - pointB[dimension])**2
    distance = math.sqrt(distance)
    return distance

class KDTreeNode():
    def __init__(self, point, left, right):
        self.point = point
        self.left = left
        self.right = right

    def is_leaf(self):
        return (self.left == None and self.right == None)

class KDTreeneighbors():
    def __init__(self, query_point, t):
        self.query_point = query_point
        self.t = t # neighbors wanted
        self.largest_distance = 0 # squared
        self.current_best = []
```

```
    def calculate_largest(self):
        if self.t >= len(self.current_best):
            self.largest_distance = self.current_best[-1][1]
        else:
            self.largest_distance = self.current_best[self.t-1][1]

    def add(self, point):
        sd = square_distance(point[1], self.query_point[1])
        # run through current_best, try to find appropriate place
        for i, e in enumerate(self.current_best):
            if i == self.t:
                return # enough neighbors, this one is farther, let's forget
it
            if e[1] > sd:
                self.current_best.insert(i, [point, sd])
                self.calculate_largest()
                return
        # append it to the end otherwise
        self.current_best.append([point, sd])
        self.calculate_largest()

    def get_best(self):
        return [element[0] for element in self.current_best[:self.t]]

class KDTree():
    def __init__(self, data):
        def build_kdtree(point_list, depth):
            #code based on wikipedia article:
http://en.wikipedia.org/wiki/Kd-tree
            if not point_list:
                return None

            # select axis based on depth so that axis cycles through all
valid values
            axis = depth % len(point_list[0][1]) # assumes all points have
the same dimension

            # sort point list and choose median as pivot point,
            # TODO: better selection method, linear-time selection,
distribution
            point_list.sort(key=lambda x: x[1][axis])
            median = int(len(point_list)/2) # choose median

            # create node and recursively construct subtrees
            node = KDTreeNode(point=point_list[median],
                              left=build_kdtree(point_list[0:median],
depth+1),
                              right=build_kdtree(point_list[median+1:],
depth+1))
            return node

        self.root_node = build_kdtree(data, depth=0)

    @staticmethod
```

```
    def construct_from_data(data):
        tree = KDTree(data)
        return tree

    def query(self, query_point, t=1):
        statistics = {'nodes_visited': 0, 'far_search': 0, 'leafs_reached':
0}

        def nn_search(node, query_point, t, depth, best_neighbors):
            if node == None:
                return

            #statistics['nodes_visited'] += 1

            # if we have reached a leaf, let's add to current best neighbors,
            # (if it's better than the worst one or if there is not enough
neighbors)
            if node.is_leaf():
                #statistics['leafs_reached'] += 1
                best_neighbors.add(node.point)
                return

            # this node is no leaf

            # select dimension for comparison (based on current depth)
            axis = depth % len(query_point[1])

            # figure out which subtree to search
            near_subtree = None # near subtree
            far_subtree = None # far subtree (perhaps we'll have to traverse
it as well)

            # compare query_point and point of current node in selected
dimension and figure out which subtree is farther than the other
            if query_point[1][axis] < node.point[1][axis]:
                near_subtree = node.left
                far_subtree = node.right
            else:
                near_subtree = node.right
                far_subtree = node.left

            # recursively search through the tree until a leaf is found
            nn_search(near_subtree, query_point, t, depth+1, best_neighbors)

            # while unwinding the recursion, check if the current node
            # is closer to query point than the current best,
            # also, until t points have been found, search radius is infinity
            best_neighbors.add(node.point)

            # check whether there could be any points on the other side of
the
            # splitting plane that are closer to the query point than the
current best
```

```
            if (node.point[1][axis] - query_point[1][axis])**2 <
best_neighbors.largest_distance:
                #statistics['far_search'] += 1
                nn_search(far_subtree, query_point, t, depth+1,
best_neighbors)

            return

        # if there's no tree, there's no neighbors
        if self.root_node != None:
            neighbors = KDTreeneighbors(query_point, t)
            nn_search(self.root_node, query_point, t, depth=0,
best_neighbors=neighbors)
            result = neighbors.get_best()
        else:
            result = []

        #print (statistics)
        return result



#this function is for importing data
def getdata(file_name):
    with open(file_name, newline='') as f:
        rowdata = []
        reader = csv.reader(f)
        for row in reader:
            for i in range(1, len(row)):
                row[i] = float(row[i])
            rowdata.append([row[0], row[1:]])
    return rowdata

def proportion_per_class(data):
    prop = {}
    for d in data:
        if d[0] not in prop.keys():
            prop[d[0]] = 1
        else:
            prop[d[0]] = prop[d[0]] + 1
    for k in prop:
        prop[k] = float(prop[k]) / float(len(data))
    return prop


def get_class_labels(data):
    classes = []
    for i in data:
        if i[0] not in classes:
            classes.append(i[0])
    return classes

def normalize(data):
```

```
    #cycling through each feature, but not the class label
    for i in range(len(data[0][1])):
        lowest = 100000000000000000
        highest = -100000000000000000
        for j in data:
            if float(j[1][i]) < lowest:
                lowest = j[1][i]
            if float(j[1][i]) > highest:
                highest = j[1][i]
        #now that we have the highest and lowest values, we can calculate the
normalized value
        for j in data:
                if highest == lowest:
                    j[1][i] = 0.5
                else:
                    j[1][i] = (j[1][i] - lowest)/(highest-lowest)
    return data

#this function is to create a stratified folded data set from a normal datase
def stratify(data, folds):
    #building a dictionary to hold all data by class which is in data[0][0]
    classes = {}
    #splitting data into classes
    for d in data:
        if d[0] not in classes:
            classes[d[0]] = []
            classes[d[0]].append(d)
        else:
            classes[d[0]].append(d)

    # n-fold stratified samples
    data = []
    for r in range(folds):
        data.append([])
    #spreading the classes evenly into all data sets
    for key,items in classes.items():
        for i in range(len(items)):
            data[i%folds].append(items[i])
    return data

def distance(x1, x2, parameters):
    if len(x1) != len(x2):
      return False
    else:
        distance = 0
        for i in range(len(x1)):
            distance = distance + ((float(x1[i]) - float(x2[i]))**2)
        distance = math.sqrt(distance)
        return distance

#testing the prediction performance
def test_fmeasure(antibodies, test_data, class_label, parameters):
    TP, TN, FP, FN = 0, 0, 0, 0
    for x in test_data:
```

```
        if x[0] == class_label:
            yhat = predict(antibodies, x, parameters)
            if x[0] == yhat:
                TP += 1
            else:
                FN += 1
        else:
            yhat = predict(antibodies, x, parameters)
            if yhat == class_label:
                FP += 1
            else:
                TN += 1

    #print (TP, TN, FP, FN)
    if float(TP+FP) != 0:
        precision = float(TP) / float(TP+FP)
    else:
        precision = 0.0
    if float(TP+FN) != 0:
        recall = float(TP) / float(TP+FN)
    else:
        recall = 0.0
    if (precision + recall) != 0:
        fmeasure = 2*((precision*recall)/(precision + recall))
    else:
        fmeasure = 0.0

    return "TP: " + str(TP)+ " TN: "+ str(TN)+ " FP: "+ str(FP)+ " FN: "+
str(FN)
    #return [precision, recall, fmeasure]

#testing the prediction performance
def test_accuracy(antibodies, test_data, parameters):
    error_count = 0
    correct_count = 0
    for x in test_data:
        yhat = predict(antibodies, x, parameters)
        if x[0] != yhat:
            error_count = error_count + 1
            #print ("predicted: ", yhat, " actual: ", x[0], "\t\t#")
        else:
            correct_count = correct_count + 1
            #print ("predicted: ", yhat, " actual: ", x[0])
    return float(correct_count) / float(len(test_data))


#vote for the best classification
def predict(antibodies, x, parameters):
    distances = []
    for a in antibodies:
        d = distance(x[1], a[1], parameters)
        if d <= a[2]:
            return a[0]
        else:
```

```
            distances.append([a[0], d])
    distances.sort(key=itemgetter(1))
    return distances[0][0]


def generate_population(training_set, classes, size, parameters):
    antibodies = []
    num_of_antibodies = int(float(size) / float(len(classes)))

    for c in classes:
        class_data = [i for i in training_set if i[0] == c]
        non_class_data = [i for i in training_set if i[0] != c]

        tree = KDTree.construct_from_data(non_class_data)

        for i in range(num_of_antibodies):
            proposed_center = choice(class_data)
            nearest = tree.query(proposed_center, t=1)[0]
            dist = distance(nearest[1], proposed_center[1], parameters)

            if dist <= parameters["step_size"]:
                radius = 0.0
            else:
                radius = dist - (dist%parameters["step_size"])

            proposed_antibody = [proposed_center[0], proposed_center[1],
radius]
            antibodies.append(proposed_antibody)
    return antibodies

def error_count(antibody, training_set, parameters):
    error_count = 0
    class_data = [i for i in training_set if i[0] != antibody[0]]
    for t in class_data:
        if distance(t[1], antibody[1], parameters) <= antibody[2]:
            error_count = error_count + 1
    return error_count

def original_generate_population(training_set, classes, size, parameters):
    antibodies = []
    #select random antibodies from the self class, and add with a radius of 0
    for c in classes:
        class_data = [i for i in training_set if i[0] == c]

        num_of_antibodies = int(float(size) / float(len(classes)))
        for i in range(num_of_antibodies):
            proposed_center = choice(class_data)
            proposed_antibody = [proposed_center[0], proposed_center[1], 0.0]
            antibodies.append(proposed_antibody)

    #expand the antibodies by a step size until it misclassify a non-self
point
    for a in antibodies:
        changed = True
```

```
        while changed:
            if error_count(a, training_set, parameters) > 0:
                a[2] = a[2] - parameters["step_size"]
                changed = False
            else:
                a[2] = a[2] + parameters["step_size"]
                changed = True

    return antibodies


#new structure of antibody: [ class, [x1, x2, x3,... ], radius]
files = [ f for f in listdir("C:/Users/Brian/Documents/IPython
Notebooks/network_data/") ]
original_data = []
for f in files:
    original_data = original_data + getdata(
"C:/Users/Brian/Documents/IPython Notebooks/network_data/" + f )

classes = get_class_labels(original_data)
proportions = proportion_per_class(original_data)

parameters = {}
parameters["step_size"] = 0.01

#varying the training set size
# learning time, seconds of time over training set size,
print ("set_size \t time_with_kd \t time_without_kd")
for set_size in range(200, 1050, 50):

    average_time = 0.0

    #building a balanced data set
    data = []
    for c in classes:
        class_data = [d for d in original_data if d[0] == c]
        shuffle(class_data)
        data = data + class_data[:int( float(set_size) /
float(len(classes)))]

    data = normalize(data)
    data = stratify(data, 10)

    time_with_kd = 0.0
    time_without_kd = 0.0
    SVM_time = 0.0

    for st in range(10):
        test_set = data[st%len(data)]
        validation_set = data[(st+1)%len(data)]
        training_set = []
        for tsp in range(len(data)-2):
            training_set = training_set + data[(st+2+tsp)%len(data)]
```

```
        start = time.time()
        antibodies = generate_population( training_set, classes, 1000,
parameters)
        end = time.time()

        time_with_kd = time_with_kd + (end - start)


        start = time.time()
        antibodies = original_generate_population(training_set, classes,
1000, parameters)
        end = time.time()

        time_without_kd = time_without_kd + (end-start)

        #preparing data for SVM
        datta = separate(training_set)
        training_set_labels = datta[0]
        training_set_data = datta[1]

        start = time.time()
        lin_clf = svm.LinearSVC()
        lin_clf.fit(training_set_data, training_set_labels)
        end = time.time()

        SVM_time = SVM_time + ( end - start )

    print (set_size, " \t ", time_with_kd/10.0, " \t ", time_without_kd/10.0,
" \t ", SVM_time/10.0)
print ("")


#varying the antibody population time
print ("pop_size \t time_with_kd \t time_without_kd")
for pop_size in range(200, 1050, 50):

    #building a balanced data set
    data = []
    for c in classes:
        class_data = [d for d in original_data if d[0] == c]
        shuffle(class_data)
        data = data + class_data[:int( float(1000) / float(len(classes)))]

    data = normalize(data)
    data = stratify(data, 10)

    time_with_kd = 0.0
    time_without_kd = 0.0
    SVM_time = 0.0

    for st in range(10):
        test_set = data[st%len(data)]
        validation_set = data[(st+1)%len(data)]
        training_set = []
```

```
        for tsp in range(len(data)-2):
            training_set = training_set + data[(st+2+tsp)%len(data)]

        start = time.clock()
        antibodies = generate_population( training_set, classes, pop_size,
parameters)
        end = time.clock()

        time_with_kd = time_with_kd + (float(end)-float(start))

        start = time.clock()
        antibodies = original_generate_population(training_set, classes,
pop_size, parameters)
        end = time.clock()

        time_without_kd = time_without_kd + (float(end)-float(start))

        #preparing data for SVM
        datta = separate(training_set)
        training_set_labels = datta[0]
        training_set_data = datta[1]

        start = time.time()
        lin_clf = svm.LinearSVC()
        lin_clf.fit(training_set_data, training_set_labels)
        end = time.time()

        SVM_time = SVM_time + (end-start)

    print (set_size, " \t ", time_with_kd/10.0, " \t ", time_without_kd/10.0,
" \t ", SVM_time/10.0)
print ("")
```

APPENDIX C

PYTHON CODE FOR OPTIMIZED AIS NEGATIVE SELECTION ALGORITHM

```
import csv as csv
import sys as sys
import random as random
from random import shuffle
import math as math
from collections import defaultdict
from operator import itemgetter
import copy as copy
import time as time
import datetime as datetime
from math import sqrt

#this function is for importing data
def getdata(file_name):
    with open(file_name, newline='') as f:
        rowdata = []
        reader = csv.reader(f)
        for row in reader:
            for i in range(1, len(row)):
                row[i] = float(row[i])
            rowdata.append(row)
    return rowdata

def distance(x1, x2):
    distance = sqrt(sum( (x1 - x2)**2 for x1, x2 in zip(x1, x2)))
    return distance

def normalize(data):
    for j in range(len(data)):
        for i in range(1, len(data[j])):
            data[j][i] = (data[j][i] - 1.0)/(10.0-1.0)
    return data

#this function is to create a stratified folded data set from a normal datase
def stratify(data, folds):
    #building a dictionary to hold all data by class which is in data[0][0]
    classes = {}
    #splitting data into classes
    for d in data:
        if d[0] not in classes:
            classes[d[0]] = []
            classes[d[0]].append(d)
        else:
            classes[d[0]].append(d)

    # n-fold stratified samples
    data = []
    for r in range(folds):
        data.append([])
```

```
    #spreading the classes evenly into all data sets
    for key,items in classes.items():
        for i in range(len(items)):
            data[i%folds].append(items[i])
    return data

#testing the prediction performance
def get_accuracy(antibodies, test_data, self_class, non_self_class):
    correct = 0.0
    incorrect = 0.0
    for x in test_data:
        yhat = predict(antibodies, x, self_class, non_self_class )
        if x[0] == yhat:
            correct += 1
            #print("correct")
        else:
            incorrect += 1
            #print("incorrect")
    accuracy = correct / float(len(test_data))
    return accuracy

def generate_random_antibody(data, parameters):
    #format: [[center], radius]
    radius = parameters["radius"]
    center = []
    for i in range(1,len(data[0])):
        center.append(random.uniform(0,1))
    return [center, radius]

def train_population(training_set, population_size, parameters, self_class,
non_self_class):
    antibodies = []
    self_class = [x for x in training_set if x[0] == self_class]
    while len(antibodies) < population_size:
        proposed_antibody = generate_random_antibody(data, parameters)
        flagged = False
        for x in self_class:
            if distance(proposed_antibody[0], x[1:]) < proposed_antibody[1]:
                flagged = True
        if flagged == False:
            antibodies.append(proposed_antibody)
    return antibodies

def predict(antibodies, x, self_class, non_self_class):
    for a in antibodies:
        if distance(a[0], x[1:]) < a[1]:
            return non_self_class
    return self_class

#testing the prediction performance
def optimized_get_accuracy(antibodies, test_data, self_class,
non_self_class):
    correct = 0.0
    incorrect = 0.0
```

```
    for x in test_data:
        yhat = optimized_predict(antibodies, x, self_class, non_self_class )
        if x[0] == yhat:
            correct += 1
            #print("correct")
        else:
            incorrect += 1
            #print("incorrect")
    accuracy = correct / float(len(test_data))
    return accuracy

def optimized_train_population(training_set, population_size, parameters,
self_class, non_self_class):
    antibodies = []
    original_self_class = [x for x in training_set if x[0] == self_class]
    while len(antibodies) < population_size:
        self_class = original_self_class   #this allows the selection above
to happen only once
        proposed_antibody = generate_random_antibody(training_set,
parameters)

        #select the self class points in each dimension that could be
contained in by the proposed antibody
        for i in range(1,len(self_class[0])):
            self_class = [s for s in self_class if s[i] >
(proposed_antibody[0][i-1]-proposed_antibody[1]) and s[i] <
(proposed_antibody[0][i-1]+proposed_antibody[1])]

        #if the self_class list is empty then add the antibody, since there
are no points in the self class contained by the hyper-cube containing the
hyper-sphere
        if len(self_class) == 0:
            antibodies.append(proposed_antibody)
        #check whether the self points selected are actually contained by the
hypersphere and not only the hyper cube
        else:
            flagged = False
            for s in self_class:
                if distance(proposed_antibody[0], s[1:]) <
proposed_antibody[1]:
                    flagged = True
            if flagged == False:    #if there are no points that are within
the hyper-sphere then add the antibody to the population
                antibodies.append(proposed_antibody)
    return antibodies

def optimized_predict(antibodies, x, self_class, non_self_class):
    #select the antibodies that could contain the point
    #for every dimension in the antibody center:

    for i in range(len(antibodies[0][0])):
        antibodies = [a for a in antibodies if x[i+1] > (a[0][i]-a[1]) and
x[i+1] < (a[0][i]+a[1])]
    #further filter the set of antibodies
```

```
    for a in antibodies:
        if distance(a[0], x[1:]) < a[1]:
            return non_self_class
    return self_class

    #if the set of antibodies is filtered down to zero, then we know that the
points is outside of the non-self class, there for it is self
    if len(antibodies) == 0:
        return self_class

#note: this script wiill be coded to only use the Breast Cancer Wisconsin
Data Set
original_data = getdata( "C:/Users/Brian/Documents/IPython
Notebooks/datasets/cancer.csv")

parameters = {}
parameters["radius"] = 0.93

print ("population size \t accuracy")
for population_size in range(100, 1050, 50):
    #building a balanced data set
    data = []
    for c in ["benign", "malignant"]:
        class_data = [d for d in original_data if d[0] == c]
        data = data + class_data[:int( float(500) / 2.0 ) ]

    data = normalize(data)
    data = stratify(data, 10)

    accuracy = 0.0

    for st in range(1):
        test_set = data[st%len(data)]
        validation_set =  data[(st+1)%len(data)]
        training_set = []
        for tsp in range(len(data)-2):
            training_set = training_set + data[(st+2+tsp)%len(data)]

        best_r = 0
        max_accuracy = 0.0
        #find the optimal value for the radius of the antibodies
        for r in range(1,100, 10):
            print(r/100.0)
            parameters = {}
            parameters["radius"] = float(r)/100.0
            antibodies = train_population(training_set, 1000, parameters,
"benign", "malignant")
            accuracy = get_accuracy(antibodies, validation_set, "benign",
"malignant")
            if accuracy > max_accuracy:
                max_accuracy = accuracy
                best_r = float(r)/100.0

        parameters = {}
```

```
        parameters["radius"] = best_r

        antibodies = train_population(training_set, population_size,
parameters, "benign", "malignant")

        accuracy = accuracy + get_accuracy(antibodies, test_set, "benign",
"malignant")

    print (population_size, " \t ", accuracy/1.0)
print ("")
```