STUDIES ON REAL-VALUED NEGATIVE SELECTION ALGORITHMS FOR SELF-NONSELF DISCRIMINATION

A Thesis

Presented to the Faculty of

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Shane Edward Dixon

February, 2010

COMMITTEE MEMBERSHIP

TITLE:	Studies on Real-Valued Negative Selection Algorithms for Self-Nonself Discrimination
AUTHOR:	Shane Edward Dixon
DATE SUBMITTED:	February 2010
COMMITTEE CHAIR:	Xiao-Hua (Helen) Yu, Associate Professor
COMMITTEE MEMBER:	Arthur MacCarley, Department Chair
COMMITTEE MEMBER:	Bryan Mealy, Assistant Professor

ABSTRACT

Studies on Real-Valued Negative Selection Algorithms for Self-Nonself Discrimination

Shane Edward Dixon

The artificial immune system (AIS) is an emerging research field of computational intelligence that is inspired by the principle of biological immune systems. With the adaptive learning ability and a self-organization and robustness nature, the immunology based AIS algorithms have successfully been applied to solve many engineering problems in recent years, such as computer network security analysis, fault detection, and data mining.

The real-valued negative selection algorithm (RNSA) is a computational model of the self/non-self discrimination process performed by the T-cells in natural immune systems. In this research, three different real-valued negative selection algorithms (i.e., the detectors with fixed radius, the V-detector with variable radius, and the proliferating detectors) are studied and their applications in data classification and bioinformatics are investigated. A comprehensive study on various parameters that are related with the performance of RNSA, such as the dimensionality of input vectors, the estimation of detector coverage, and most importantly the selection of an appropriate distance metric, is conducted and the figure of merit (FOM) of each algorithm is evaluated using real-world

datasets. As a comparison, a model based on artificial neural network is also included to further demonstrate the effectiveness and advantages of RNSA for specific applications.

ACKNOWLEDGMENTS

The completion of this thesis represents the greatest accomplishment of my educational career at Cal Poly. I owe my thanks to many individuals that contributed to my success in this substantial undertaking.

First and foremost, I would like to thank my thesis advisor, Dr. Helen Yu, for her recommendation of this topic. The research into a new branch of computational intelligence was a difficult endeavor; however, I was able to persevere with the continuous support and guidance provided by Dr. Yu.

I must take this opportunity to also thank Dr. Art MacCarley. Since working with him over a previous summer, he has become an inspirational role model and a valued friend. His unwavering trust and faith in my capabilities instilled a much needed sense of confidence.

Most importantly, I would like to thank my immediate family. I could not have accomplished this task without the love, sacrifice and encouragement from my wife, Kari Sweet. I would like to acknowledge my sister, Sheena Dixon, who looks upon me as a role model in her own schooling, and reminds me to set the example for self-perseverance. Finally, I would like to dedicate this study to my mother, Cheryl, whose lifelong selfless acts of love and positive influence have made me into the person I am today.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Many biological systems provide inspiration for developing new ideas in problem solving strategies and computing paradigms. Similar to neural networks and genetic algorithms, the mechanisms of learning, prediction, memory and adaptation in the immune system are important biological metaphors in the research of bio-inspired computation methods. Although relatively young, Artificial Immune System (AIS) models are emerging as an active and attractive field involving models and applications of great diversity. There are many immunologically inspired algorithms being explored in the field of computational intelligence; the most dominant of these are the immune network model, clonal selection, and negative selection algorithm. Each model can perform a variety of tasks, including pattern recognition, data classification, fault detection, network and computer security, data mining and numerous others.

An important aspect of the biological immune system is its ability to recognize and categorize all of the cells or molecules in the body as either self or non-self cells. Through an evolutionary learning process, the immune system is able to distinguish between foreign antigens (bacteria, viruses, etc.) and the body's own cells or molecules, which became the inspiration for the artificial negative selection algorithm. The artificial negative selection algorithm is a computational imitation of the self/non-self

immunological discrimination process. Since its conception, negative selection algorithms have attracted the attention of many computational intelligence researchers.

This thesis addresses the task of data classification, specifically using the self/non-self discrimination methods implemented in a real-valued negative selection algorithm. Since gaining popularity, the negative selection algorithm has already undergone several variations from its original implementation. Three specific variations of the real-valued negative selection algorithm are tested using three different real world datasets to determine the efficiency of each implementation. The central mechanism to a negative selection algorithm is the selection of an appropriate matching rule, or distance measure in the case of real-valued data. Therefore, five different distance metrics are tested for each variation of the negative selection algorithm to compare the advantages and disadvantages of each implementation. An artificial feedforward neural network model is tested as a comparison model to established adaptive learning algorithms. Finally, a figure of merit is proposed to measure each algorithm's overall effectiveness in performing correct data classification.

This study is separated into six distinct chapters. Chapter 2 introduces some background concepts on the biological immune system and how it inspired and relates to the AIS model. Various AIS models are reviewed, followed by an in-depth discussion about the negative selection algorithm. Chapter 3 begins with a complete description of each real-valued distance metric tested in this study. It also details the three unique variations of the real-valued negative selection algorithms implemented, including pseudo-code to aid in the understanding of each version. Chapter 4 includes a brief background on neural networks followed by a discussion on the architecture and

calculations performed by the artificial feedforward neural network algorithm implemented in this study. The last section of this chapter details the back-propagation algorithm used to train the neural network. Chapter 5 covers the datasets, testing methodology, and final results from this research. Finally, Chapter 6 presents the conclusion of the findings and potential for future studies. Appendix A provides additional data table not included in the body of this report and Appendix B includes samples of the actual MatLab source code written for each algorithm version.

CHAPTER 2

Background

2.1 Biological Immune System

The biological immune system is a complex adaptive system of cells, molecules, and organs that give an organism the ability to recognize foreign substances and neutralize or degrade them, with or without injury to the organism's own tissue. To accomplish this task, the immune system has evolved sophisticated pattern recognition and response mechanisms using its network of chemical messengers for communication. They recognize an almost limitless variety of infectious foreign cells and substances known as *nonself elements* and are distinguished from those native noninfectious cells, known as *self molecules*.

There are two major branches of the biological immune system. The innate immune system is present before birth and consists of the cells and mechanisms that defend the host from infection by other organisms, in a non-specific manner. One important component of the innate immune system is a class of blood proteins known as complement; this class has the ability to identify bacteria, activate cells and to promote clearance of dead cells or antibody complexes. Several other functions of the innate immune system include the recruiting of immune cells to sites of infection through the production of chemical factors, and the identification and removal of foreign substances present in organs, tissues, the blood and lymph, by specialized white blood cells.

The immune cells responsible for engulfing and destroying harmful pathogens and particles are known as phagocytes. Phagocytic cells, including macrophages, neutrophils and dendritic cells, function within the immune system by identifying and eliminating pathogens that might cause infection. Phagocytes generally patrol the body searching for pathogens, but are also able to react to a group of highly specialized molecular signals produced by other cells [14]. Phagocytes also play a role in regular tissue development and maintenance, and are an important part of the healing process following tissue injury.

The other important immune cells in the innate immune system are the white blood cells known as leukocytes. Leukocytes are different from other cells of the body in that they are not tightly associated with a particular organ or tissue; thus, they function similar to independent, single-celled organisms. Leukocytes are able to move freely and interact with and capture cellular debris and foreign particles, or invading microorganisms. Unlike many other cells in the body, most innate immune leukocytes cannot divide or reproduce on their own, but are the products of pluripotent hematopoietic stem cells present in the bone marrow [14].

The most important aspect of the innate immune system is the fact that it induces the expression of co-stimulatory signals in antigen presenting cells (APCs) that will lead to T-cell activation promoting the start of the adaptive immune response [7]. To clarify, the adaptive or "specific" immune system is activated by the “non-specific” and evolutionarily older innate immune system. The adaptive immune system is the main focus of interest here as learning, adaptability, and memory are important characteristics of adaptive immunity. The adaptive immune system is composed of highly specialized,

systemic cells and processes that eliminate or prevent pathogenic challenges. The adaptive immune response provides the vertebrate immune system with the ability to recognize and remember specific pathogens to generate immunity, and to mount stronger attacks each time the pathogen is encountered.

The adaptive immune system is highly adaptable because of somatic hypermutation (a process of accelerated somatic mutations), and V(D)J recombination (an irreversible genetic recombination of antigen receptor gene segments). This mechanism allows a small number of genes to generate a vast number of different antigen receptors which are then uniquely expressed on each individual lymphocyte. The adaptive immune system uses clonally distributed, somatically generated antigen receptors on two types of lymphocytes, memory B-cells and memory T-cells [7]. B-cells and T-cells are derived from the same pluripotential hematopoietic stem cells, and are indistinguishable from one another until after they are activated. B-cells play a large role in the humoral immune response; T-cells are intimately involved in cell-mediated immune responses [14].

The humoral branch of the immune system involves the interaction of B-cells with antigens and their subsequent proliferation and differentiation into antibody-secreting plasma cells. Upon activation, B-cells produce antibodies, each of which recognizes a unique antigen, and neutralize specific pathogens. An antigen is a substance that prompts the generation of antibodies and can cause an immune response [14]. "Self" antigens are usually tolerated by the immune system; "Non-self" antigens are identified as intruders and attacked by the immune system. Antibodies function as the effectors of the humoral response by binding to antigens and facilitating their elimination. When an

antigen is coated with an antibody, it can be eliminated in several ways, such as ingestion by phagocytes or activation of the complement system [1]. The main point is that long-lived antigen specific memory B-cells will remain after this process occurs; these cells can be called upon to respond quickly if the same pathogen re-infects the host.

Effector T-cells generated in response to antigens are responsible for cell-mediated immunity. Cytotoxic T-cells are a sub-group of T-cells which induce the death of cells that are infected with viruses or are otherwise damaged or dysfunctional. Helper T-cells are immune response mediators and play an important role in establishing and maximizing the capabilities of the adaptive immune response. These cells have no cytotoxic or phagocytic activity; they cannot kill infected cells or clear pathogens, but, in essence, "manage" the immune response by directing other cells to perform these tasks [14]. Figure 2.1 illustrates the basic structure of the biological immune system [6].

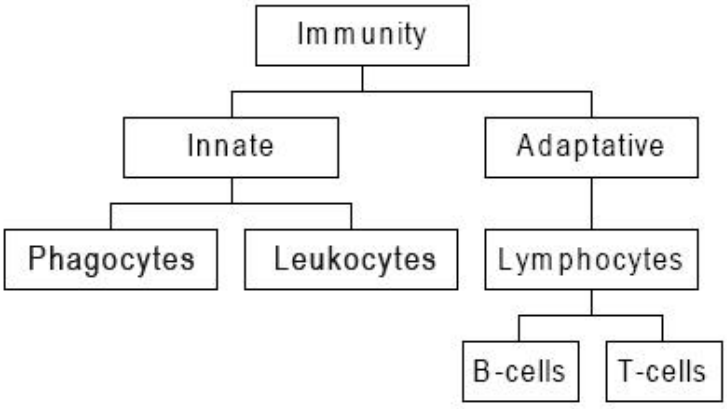


Figure 2.1: The Biological Immune System Structure

In terms of information processing, the biological immune system is a fascinating distributed adaptive system with partially decentralized control mechanisms. The system

utilizes feature extraction, signaling, learning, memory, pattern recognition, and associative retrieval to solve recognition and classification tasks. It has the ability to learn to recognize relevant patterns, remember patterns that have been seen previously, and use a combinatorics to construct pattern detectors efficiently. Remarkably, the overall behavior of the system is an emergent property of many local interactions within the immune system [4]. As with many other biologically inspired methods, the immune system provides several important aspects in the field of computational intelligence. In particular, idiotypic network theory, negative selection mechanisms, clonal selection and somatic hypermutation theories have emerged in Artificial Immune System models [1, 6, 7].

2.2 Artificial Immune Systems

In the 1990s a new branch of computational intelligence emerged, commonly referred to as an Artificial Immune System (AIS). Since its inclusion into the field of computational intelligence, a variety of models have been proposed which are inspired by the biological immune system. Researchers have explored a variety of applications, including pattern recognition, data classification, fault detection, network and computer security, data mining, and numerous others [8]. Despite the Artificial Immune System models gaining more attention recently, the underlining fundamental methodologies have not changed dramatically. The most discussed models to date are the immune network models, clonal selection, and negative selection. Figure 2.2 illustrates the placement of AIS models within the field of artificial intelligence.

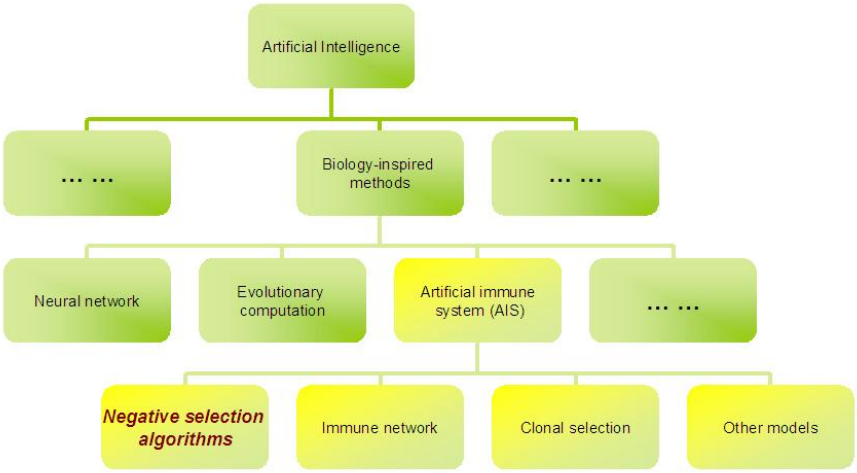


Figure 2.2: AIS as a branch of Computational Intelligence [4]

Proposed in the mid-seventies, the earliest form of immune network theory suggests that the immune system maintains an idiotypic network of interconnected B-cells for antigen recognition. This particular model is inspired by the biological adaptive immune system, specifically the humoral branch dealing with lymphocyte B-cells. These cells work together using stimulation and suppression to attain network stabilization. The basic principle is that any two B-cells will connect if the affinity they share reaches a specific threshold; the strength of this connection is directly proportional to the affinity in which they share [1, 4].

Consequently, in an artificial immune network (AIN) model, populations of B-cells are divided into two distinct categories: the initial population and the cloned population. The initial population set is derived from a subset of the raw training data to create a B-cell network. The remainders of the training data are used as antigen training items and are selected randomly and presented to areas of the B-cell network. If the antigen shares an affinity with a B-cell and binds successfully, the B-cell is cloned and mutated. The mutated B-cell represents a diverse set of antibodies, and an attempt is

made to integrate it into the existing B-cell network. If the new B-cell cannot integrate, it is removed from the network. If the antigen cannot bind with any B-cells in the existing network, a B-cell is generated using the antigen as a template, and is then incorporated into the network [4]. This model has become popular in network intrusion detection systems for computer security [1, 8, 23].

Similar to the artificial immune network, the clonal selection principle describes the basic features of an immune response to an antigenic stimulus [1, 4, 7]. Operating on both B-cells and T-cells, clonal selection establishes the foundation that only those cells that recognize an antigen proliferate, eliminating those which do not. The main features of clonal selection theory are that new cells are clones of their parent cells, and subject to high rates of mutation (somatic hypermutation). Proliferation and differentiation occur whenever mature cells come into contact with antigens. Any lymphocytes (B and T-cells) which include self-reactive receptors are eliminated [4, 7]. Figure 2.3 illustrates the concept of the clonal selection principal.

The clonal selection principles should seem obviously similar to other evolutionary algorithms, such as natural selection. The fittest candidates are the ones which best recognize an antigen, and therefore are the cells allowed to proliferate; only the clones which best perform are allowed to mature. The clonal selection algorithms which exist produce several remarkable features: 1) population sizes dynamically adjustable, 2) exploitation and exploration of the search space is achieved, 3) location of multiple optima, 4) capability of maintaining local optima solutions, and 5) defined stopping criteria [4,7]. Many of the algorithms proposed require minimal control parameters as each emphasizes self-organization.

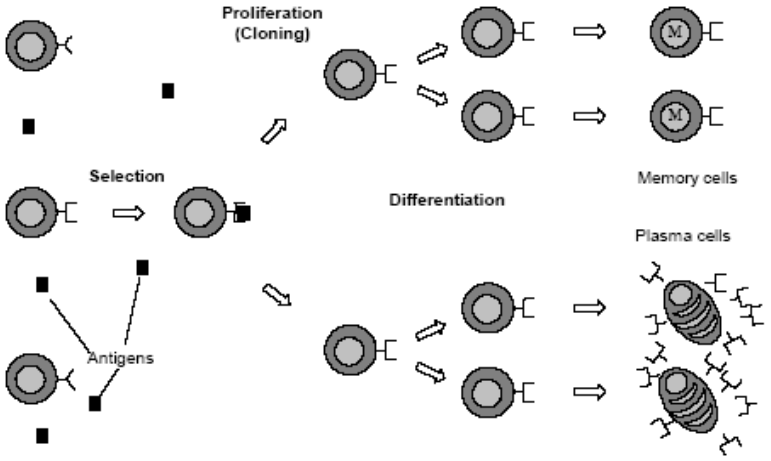


Figure 2.3: The Clonal Selection Principle [7]

There are many other immunologically inspired algorithms being explored in the field of computational intelligence. Other features of the immune system being considered include adaptation, immunological memory and protection against auto-immune attacks. Approaches have been made to combine the power of the neural network to immune system models, such as increasing the memory capacity and retrieval performance using a Hopfield network to aid an associative memory model based on the immune network [7]. A major branch of Artificial Immune Systems is negative selection, and is the topic of discussion in the next section. Before an explanation of negative selection is given, a new theory should be mentioned which may affect the future of negative selection algorithms. Danger theory is a new theory becoming popular among immunologists, which explores the discrimination that goes beyond the self/non-self distinction previously believed. For example, there is no immune response to foreign bacteria in some of the food we eat. Conversely, some auto-reactive processes are useful,

such as attacking self molecules produced by stress. The theory concludes that the immune system only discriminates “some self from non-self” [1].

2.3 Negative Selection

An important aspect of the biological immune system is its ability to recognize and categorize all of the cells or molecules in the body as either self or non-self cells. Through an evolutionary learning process, the immune system is able to distinguish between foreign antigens (bacteria, viruses, etc.) and the body’s own cells or molecules. The purpose of negative selection is to ensure that lymphocytic cells are trained to only eliminate harmful antigens, and to avoid reacting to self cells to avoid internal cellular damage.

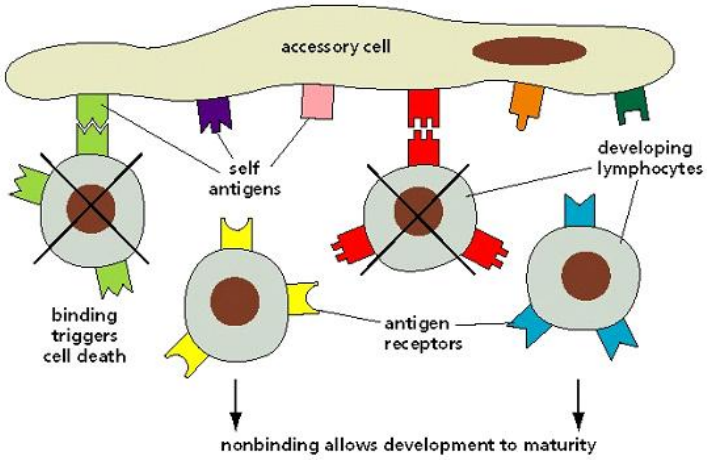


Figure 2.4: The Negative Selection Principle [18]

The negative selection process begins with the generation of T-cells, where the receptor sites attached to the lymphocytes are created through a pseudo-random genetic

rearrangement process. Within the thymus, they undergo a rigorous censoring process, where T-cells that react against self-proteins are destroyed. The cells that do not bind to self-proteins are allowed to leave the thymus. These matured T-cells are then allowed to circulate in the body and perform immunological functions to protect the body from harmful foreign pathogens [4]. It is the process of self-nonself discrimination censoring of the T-cells that is referred to as negative selection, which is illustrated in Figure 2.4.

The concept of a negative selection algorithm for computational intelligence was first conceived by Stephanie Forrest in 1994 [9]. Forrest compared the problem of protecting computer systems to that of learning to distinguish between self and non-self. It is one of the earliest Artificial Immune System algorithms that was applied to real-world applications. Since its conception, negative selection algorithms have attracted the attention of many computational intelligence researchers. While the process has evolved though various implementations, the fundamental characteristics remain intact.

Before a formal discussion of the negative selection algorithm can proceed, a new set of terminology must be defined. The lymphocytic cell receptors which discriminate between self and non-self cells are called "detectors." The body's immunological functions recognize and categorize antigens, while the negative selection algorithm operates to classify unknown data. The negative selection algorithm is not appropriate for general classification tasks because it is a one-class based classification algorithm, currently only utilized to discriminate between two classes of data. The terms "self" and "non-self" are artificial labels given to the classification of data instances. For example, in network security implementations, "self" would refer to standard incoming "safe" data, while "non-self" would represent data deemed malicious or intrusive to the network.

Either the full or partial "self" data set is typically employed for training the negative selection algorithm.

The negative selection algorithm consists of two phases: the generation stage and the detection stage. Beginning with the generation stage, detectors are generated by some random process and are eliminated if they match any self samples. The matching criteria are based on the data representation and is discussed later. After a sufficient number of detectors are generated, determined by certain stopping criteria, the generation phase is terminated. The collection of retained "mature" detectors (or detector set) is then implemented in the detection phase. Each unknown data instance is presented to the detector set and is classified as either self or non-self. If the unknown data instance matches any detector in the detector set, then it is classified as non-self or an anomaly. If the incoming data instance is not recognized by any detector, it is safely assumed to be a member of the self set. The generation and detection phases are shown below in Figure 2.5.

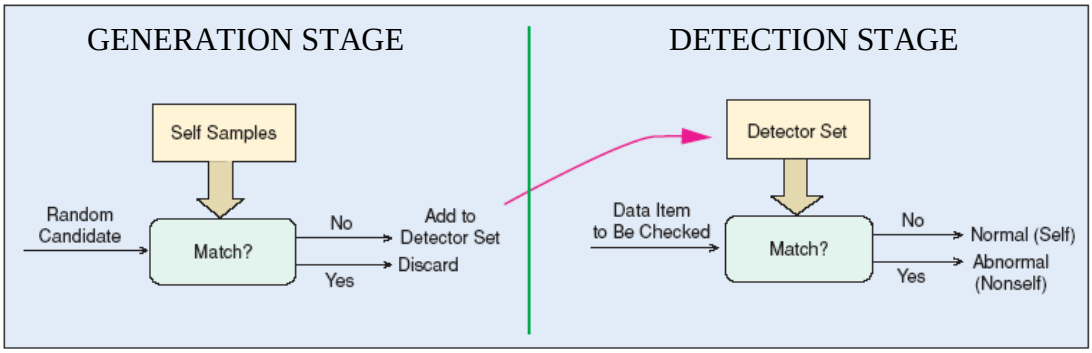


Figure 2.5: The Basic Concept of the Negative Selection Algorithm [4]

As in any other computational intelligence technique, different negative selection algorithms are characterized by particular data representation schemes, matching rules and detector generation processes. The fundamental purpose of a negative selection algorithm is to classify data; therefore, the algorithm is defined first and foremost by the data representation scheme. The first implementations of negative selection algorithms classified strictly binary data. Later on, it was extended to handle data in string (alphabetic) representation. The focus of this study concerns real-valued data representation, a more recent topic of research. Negative selection algorithms have also been modified to handle hybrid data, comprising both real-valued and string data representations [4].

The detector generation and elimination mechanisms implemented in a negative selection algorithm are a defining characteristic of the algorithm. For string data representation, both randomized algorithms (exhaustive algorithm) and deterministic algorithms (linear time and greedy algorithm) have been discussed [15, 17]. To date, only random-based generation schemes have been implemented for real-valued vector data representation. Numerous strategies are proposed for how the random generation of detectors are implemented. The classical approach is the random generation and elimination strategy, and is implemented in this study with different variations. Other approaches to detector generation include: 1) evolutionary approaches such as genetic algorithms, 2) one-shot randomized algorithms, 3) optimization with aftermath adjustment [12, 15, 17].

A significantly important factor in the performance of the negative selection algorithm, and focus of this study, is the choice of matching rules implemented for data

recognition. The choice of the matching rules or the threshold used in matching rules must be application specific and data representational dependent. The matching rule is a measure of distance, affinity or similarity that two data instances share. Regardless of representation, a matching rule *M* is symbolically defined as shown below [15].

dMx → affinity measure between detector "d" and data instance "x" **(2.1)**

Negative selection algorithms were first designed to detect changes in string data. Several matching rules have been proposed for measuring the affinity of string data. The Hamming distance or edit distance (equation 2.2) is an obvious choice for string data due to its simplicity. It is defined as the minimum number of point mutations required to transform one string data instance into another, where a point mutation is to change a letter or bit. There is also a variation of the Hamming distance, called the Roger and Tanimoto distance (R&T), shown in equation 2.3, where $\oplus$ is the exclusive-OR operator, and $0 \leq r \leq 1$ is the threshold value. Another popular matching rule is the *rcb* (r-contiguous bits) matching rule [15]. The matching requirement is defined as *r* contiguous matching symbols in corresponding positions in a string of arbitrary length broken up into shorter segments of predefined length. A variation of the *rcb* matching rule is the *r*-chunk matching rule, in which an *r*-chunk detector is a string of *r* bits together with a specific window. The detector *d* is said to match a string *x* if all bits of *d* are equal to the bits of *x* in the window specified by *d* [17]. Many other matching rules exist for string data representation including alternative variations to the Hamming distance, statistical correlation and Landscape-affinity matching [15].

Hamming Distance : $\sum_{i=1}^{N} \overline{(X_i \oplus Y_i)}$ **where *X, Y = binary n-dimensional vectors*** **(2.2)**

Roger and Tanimito Distance : $$\frac{\sum_i x_i \bar{\oplus} d_i}{\sum_i x_i \bar{\oplus} d_i + 2\sum_i x_i \oplus d_i} \geq r \qquad \textbf{(2.3)}$$

where *x, d = binary n-dimensional vectors*

For a real-valued vector data representation, the most common matching rule equates to a mathematical distance metric. The calculation of a mathematical distance metric outputs a real number to assign to the affinity, allowing simplistic comparison to an assigned matching threshold. The most common distance metric implemented is the Euclidean distance metric, but many others exist. The choice of distance metrics is central to the content of this thesis, and is discussed further in chapter 3.

Matching rules have also been formulated for hybrid (or mixed) data representations. One popular distance metric for handling mixed data is the Heterogeneous Euclidean-Overlap Metric (HEOM). Another useful metric for determining similarities in hybrid data is the Heterogeneous Value Difference Metric (HVDM) [17]. An explanation of each method is provided in equations 2.4 and 2.5. Alternative matching rules may exist for hybrid data, but these two represent the standards implemented currently in negative selection algorithms.

Heterogeneous Euclidean-Overlap : $$HEOM(x,y) = \sqrt{\sum_{g=1}^{G} heom(x_g - y_g)^2} \qquad \textbf{(2.4)}$$

where $$heom(x_x, y_g) = \begin{cases} overlap(x_g, y_g), & g \text{ is nominal} \\ \frac{|x_g - y_g|}{range_g}, & g \text{ is discrete or real} \end{cases}$$

Heterogeneous Value Difference : $$HVDM(x,y) = \sqrt{\sum_{g=1}^{G} hvdm(x_g - y_g)^2} \quad \textbf{(2.5)}$$

where

$$hvdm(x_x, y_g) = \begin{cases} \sqrt{vdm(x_g, y_g)}, & g \text{ is nominal} \\ \frac{|x_g - y_g|}{range_g}, & g \text{ is discrete or real} \end{cases}$$

$$vdm(x_g, y_g) = \sum_{c=1}^{C} \left(P(c|x_g) - P(c|y_g)\right)^2$$

While data representation, detector generation and matching rules define each negative selection algorithm, there are several other factors that affect the performance. The number of detectors affects the efficiency of generation and detection, and consequently the speed of the algorithm. Linked directly to the accuracy of detection, detector coverage is also an important factor to consider during detector generation. The stopping criteria and detector generation schemes are typical control parameters to determine an adequate number of detectors and coverage. Chapter 3 provides different implementations of each to optimize detector coverage and accuracy.

Since gaining recognition, the negative selection algorithm has already undergone several variations from the original implementation. The combination of negative selection with alternative classification techniques continues to grow. As mentioned previously, danger theory is one example of an extension to negative selection algorithms. Considering network security, danger theory would prove beneficial by elaborating on the self/non-self discrimination by identifying “non-self but harmless” and “self but harmful” [1]. Another new approach proposed is to allow the negative selection

algorithm to generate non-self samples and then apply a separate classification algorithm to generate the characteristic function of the self (or non-self). This characteristic function corresponds to an anomaly detection function, and is able to classify new samples as either self or non-self. From the proposed approaches published, two different classification algorithms were tested: 1) a multilayer neural network trained using back-propagation and, 2) an evolutionary algorithm to generate fuzzy classifier rules, using a genetic algorithm with a linear representation of tree structures in order to evolve complex fuzzy rule sets [10, 11].

The last variation of the negative selection algorithm of significance is a multilayer artificial immune system which employs both positive and negative selection. The alternative model of positive selection is suggested to reduce the number of false detections of self cells classified as non-self [20]. Detectors are generated in the same fashion for negative selection; but, in addition, a new subset of detectors is generated using positive selection to capture the knowledge of known self data. When an unknown data instance is applied to the system, the data instance is classified as non-self only if the negative selection detectors match and the positive selection detectors do not match [15,20].

CHAPTER 3

Real-Valued Negative Selection Algorithms

The real-valued negative selection algorithm was originally proposed in 2002 [11]. Several important factors determine the characterization and efficiency of a real-valued negative selection algorithm. By definition, the data and detectors are represented by real-valued data. The focuses of this study targets the implementation of different matching rules (distance metrics), detector generation and censoring schemes. The intention is to evaluate the performance of three different detector generation formats and to compare their results based on five selected distance metrics.

3.1 Real-Valued Distance Metrics

The selection of an appropriate distance measure is crucial to the overall performance of a real-valued negative selection algorithm. The entire process of a negative selection algorithm, or of any learning algorithm, is built on the concept of affinity or distance. First and foremost in a real-valued negative selection algorithm, the distance metric determines the shape of a detector in an n-dimensional space. While there are several control parameters that may be modified to affect the performance of the generation phase, the distance metric is the central mechanism for the functionality of the algorithm. The number of detectors generated and the estimation of detector coverage are both byproducts of the distance metric implemented. Most importantly, during the detection

phase, it is the decision rule implemented to classify the unknown incoming data instance as either self or non-self.

In Euclidean space R_n, the commonly used Euclidean distance, or 2-norm, can be generalized to the Minkowski distance of order *m*, or L_m distance, for any arbitrary *m*. For a point $(x_1, x_2, x_3, \ldots, x_n)$ and a point $(y_1, y_2, y_3, \ldots, y_n)$ in *n*-dimensional space the Minkowski distance, or *m*-norm distance, is defined as shown in equation 3.1 [15,16]. Four of the five distance metrics implemented in this study are simply variations of the Minkowski distance. The 1-norm distance is called the Manhattan distance metric (3.2), and is simply the absolute value of the difference between two points in *n*-dimensional space. The most common distance metric, and often the first to come to mind, is the Euclidean distance measure (equation 3.3), also referred to as the 2-norm. The next distance metric implemented has no special moniker, and is just simply stated as the 3-norm distance metric (equation 3.4). It is similar to the Euclidean distance, except the difference is cubed and the summation is cube-rooted. Unlike the Euclidean measure, the absolute value sign is critical here to avoid imaginary values. The final variation of the Minkowski distance is the infinity norm distance (equation 3.5). As shown, by taking the limit as *m* approaches infinite, it yields the maximum distance between two points in a single dimension. This distance metric is referred to in subsequent sections as simply the *Max* distance metric.

Minkowski Distance : $$\left(\sum_{i=1}^{n} |x_i - y_i|^m \right)^{\frac{1}{m}} \qquad \textbf{(3.1)}$$

Manhattan Distance : $$\sum_{i=1}^{n} |x_i - y_i| \quad \textbf{(3.2)}$$

Euclidean Distance : $$\sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \quad \textbf{(3.3)}$$

3-norm Distance : $$\left(\sum_{i=1}^{n} |x_i - y_i|^3 \right)^{\frac{1}{3}} \quad \textbf{(3.4)}$$

Infinity Norm Distance : $$\lim_{m \to \infty} \left(\sum_{i=1}^{n} |x_i - y_i|^m \right)^{\frac{1}{m}} \quad \textbf{(3.5)}$$

$$= \max(|x_i - y_i|, i = 1, 2, \cdots, n)$$

The final distance metric utilized in this study is fashioned after the *rcb* matching rule for string data, but is applied in real-valued data representation. This distance measure can be described as the partial Euclidean distance. The distance is defined over some of the elements of the vector, equivalent to the distance projected to a lower dimensional space degraded from the original space. In other words, the Euclidean distance is not calculated over all dimensions of a vector of data; only some of the dimensions are used instead to calculate the distance over a lower-dimensional space. In this manner, it is similar to partial matching in string representation that only uses some bits [16]. The measure can be chosen contiguously or randomly, but in either case the chosen positions need to match between the two points whose distance is calculated.

In the case of this study, the points are chosen contiguously using a mechanism referred to as a "sliding window." For example in a four dimensional space, two points are represented as (x_1, x_2, x_3, x_4) and (y_1, y_2, y_3, y_4). The partial Euclidean distance measure would perform the typical Euclidean distance calculation, but only for the points (x_1, x_2) and $(y_1, y_2,)$. Next, the window of observation will "slide" to the next two sets of data points, (x_2, x_3) and $(y_2, y_3,)$, and conclude with (x_3, x_4) and $(y_3, y_4,)$. Of the three separate distances calculated, only the least in size will be retained. Therefore, the partial Euclidean distance determines the smallest distance in two-dimensional space for n-dimensional points in space. For all implementations in this study, the window size is fixed to two, and this distance metric will often be referred to as simply the *Window* distance metric.

One unique feature of the distance metric chosen for a real-valued negative selection algorithm is the impact it has on the shape of the detectors. The detectors are assigned a real-valued threshold utilized in self/non-self discrimination, which can be envisioned as a radius of detection. If a calculated distance is less than this assigned threshold, the detector is said to "detect" that data instance; therefore, classifying it as non-self. This set threshold, or radius, combined with the desired distance measure yields a distinct shape for each detector implementation. Figure 3.1 illustrates the shape of each detector in two-dimensional space for a given distance metric with the same radius.

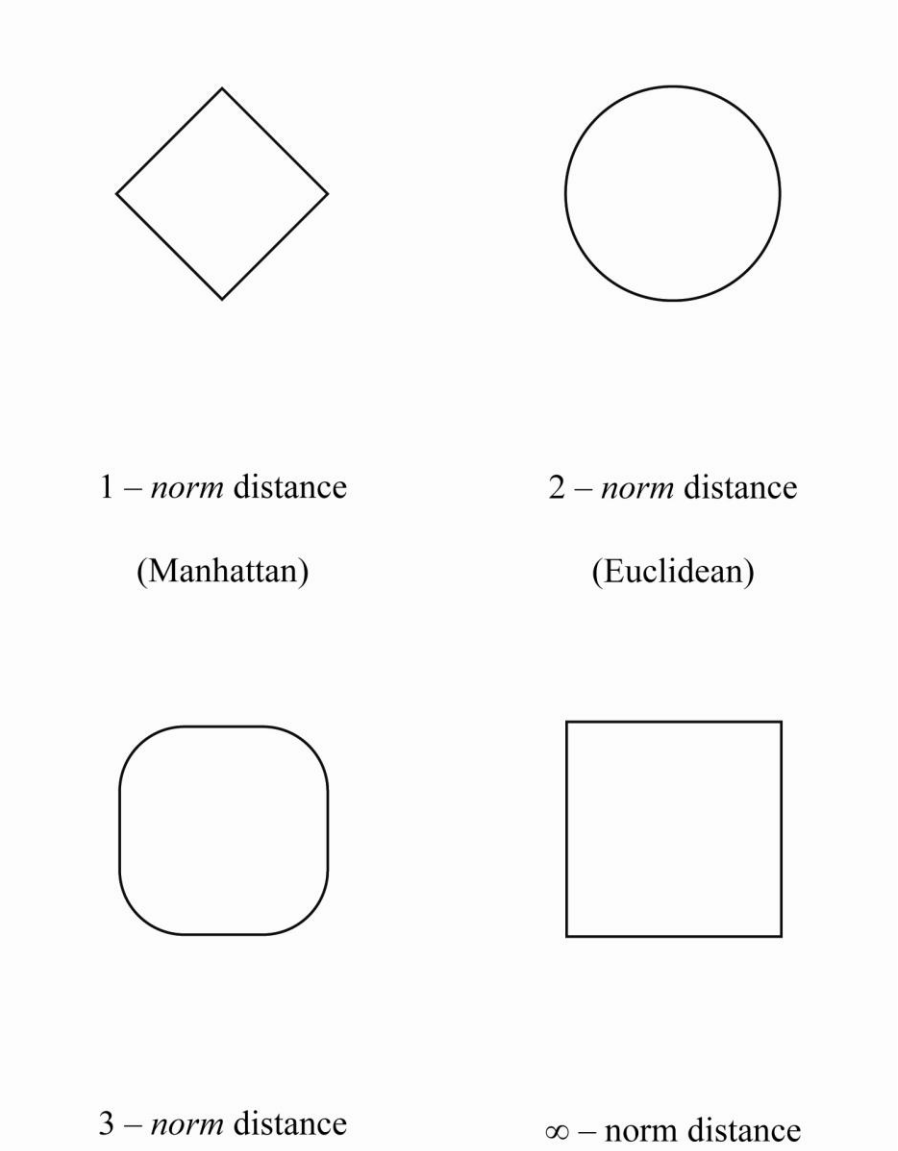
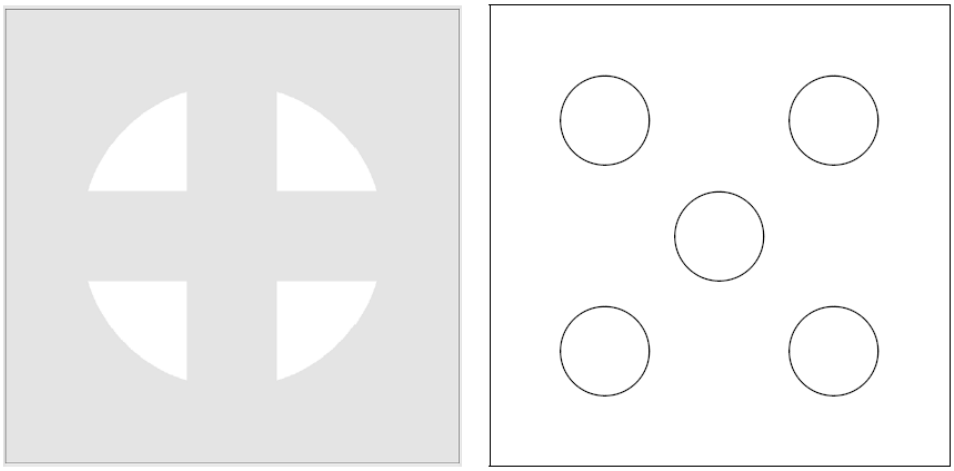


Figure 3.1: Various Geometric Shapes Associated with Different Distance Metrics [15]

In a previous study, the four distance metrics shown in Figure 3.1 were compared to estimate coverage. To test the algorithm, experiments were carried out using 2-dimensional synthetic data over the unit square $[0, 1]^2$. Two shapes were used as the 'real' self region in these experiments, the "intersection" and "five circles," as Figure 3.2 shows [15]. For the "intersection" shape, the Euclidean and Manhattan distance

measures performed the best. The “five circles” shape yielded nearly equal results for all four distance metrics with a tenth of a percent difference. However, for the “five circles,” the 3-norm out-performed the latter, with the Manhattan at a close second.



(a) Intersection (b) Five Circles

Figure 3.2: Synthetic Data Shapes of Self Regions [15]

The previous experiment further justifies the need for the content of this report. No research to date studies the effects of different real-valued negative selection algorithms and analyzes the effects of implementing various distance metrics. The previously mentioned study is the only study to evaluate the effects on different real-valued distance metrics, and it only used synthetic data in two dimensions that fit into symmetric shapes [15]. Because it was only in a two dimensional space, it did not take into account how each distance metric will perform in an n-dimensional space discriminating between real world data, or how it may compare to the partial Euclidean metric described previously.

3.2 Negative Selection Algorithm with a Fixed Radius

The first real-valued negative selection algorithm implemented and tested is based on the techniques proposed by Gonzalez and Dasgupta [11]. The approach uses real-valued data representation to characterize the self-nonself space and evolve a set of detectors that can cover the non-self complementary subspace. The inputs to the algorithm are the self samples represented by n-dimensional points (vectors). The algorithm then attempts to evolve another set of points (called detectors) to cover the non-self space. This is accomplished through an iterative process that updates the positions of the detectors driven by two fundamental goals. The detectors must remain a set distance (threshold) away from the self points and the detectors must remain separated from other detectors in order to maximize the non-self space covering. Figure 3.3 illustrates the iterative process of the detector generation phase, with a thorough discussion to follow [10].

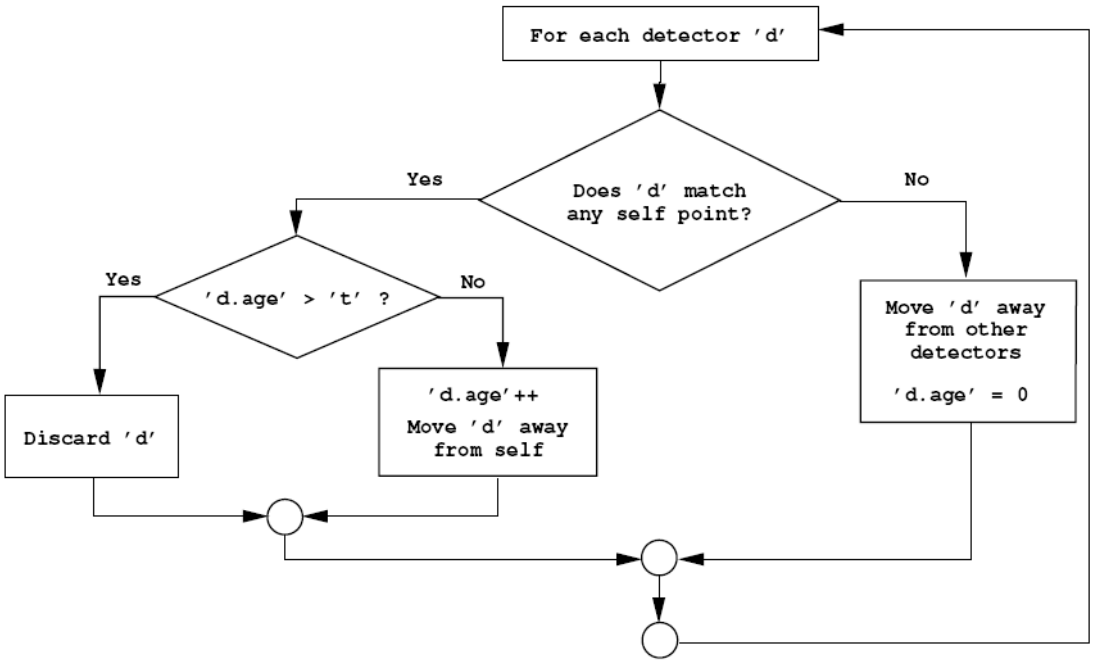


Figure 3.3: Iterative Process of the Detector Generation for Constant Sized Detectors

The generation phase of the real-valued negative selection algorithm implementing detectors with a fixed radius begins by assigning values to several control parameters. The total number of detectors generated is a predetermined control parameter. As mentioned previously, the threshold of a detector is a preset real-valued assignment to distinguish between self and non-self. The matching criteria in a real-valued negative selection algorithm are based on a distance metric; therefore, the threshold value logically takes the form of the detector's radius of detection. The detection threshold is often referred to as the detector's radius, or specified as simply *r*. Another important control parameter of the algorithm is the adaptation rate η_o, which controls the initial amount a detector is moved away from other self or detector points. An additional control parameter τ controls the decay rate of the step size implemented to move the detector for each iteration. The final control parameter t is a preset maximum age the detector must reach before being discarded. All of the control parameters become clearer as the algorithm is discussed in more detail.

The detector generation phase begins by randomly generating a preset number of *n*-dimensional points in space, distributed in a subset of R^n, specifically $[0,1]^n$, with a mean value of ½. The real-valued data utilized in testing is also normalized within the subset of $[0,1]^n$. The dimensionality of the subspace is determined by the dimensionality of the test data. Because the parameter *r* specifies the radius of detection for each detector, each detector can be envisioned as a hypersphere with a center and fixed radius in an *n*-dimensional space. The detectors are trained with only self samples; since it is undesirable for the detectors to match self points, the shortest allowable distance for a good detector to the self set is *r*.

The determination of the distance from any detector to a self point is computed using the distance metric. For this study, five different distance metrics are separately implemented. The algorithm begins by calculating the distance from a single detector to each self point individually, and the shortest distance from the detector to any self point is stored. If that distance is less than the threshold radius *r*, the detector is moved; otherwise, it is stored for the detection phase. Neglecting the first detector, each subsequent detector also computes the distance to all previously stored detectors, and again is moved or stored based upon the radius *r*.

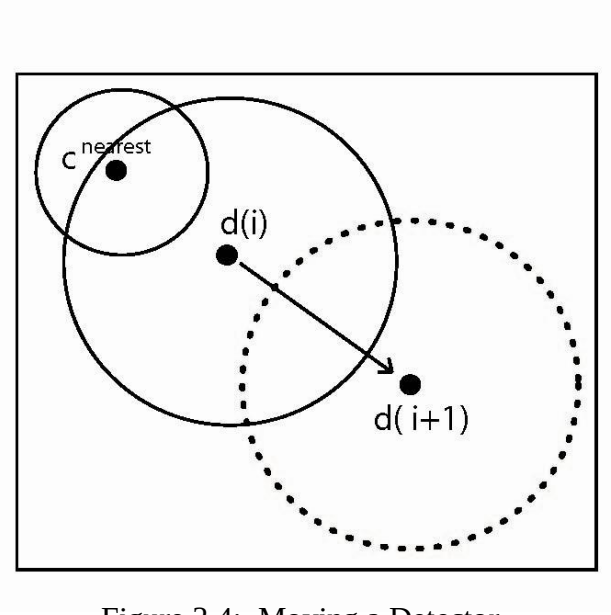


Figure 3.4: Moving a Detector

The preset adaptation rate parameter η_o represents the initial step size used to move the detectors. In order to guarantee that the algorithm converges to a stable state, it is necessary to decrease this parameter in each iteration in such a way that the $\lim_{i \to \infty} \eta_i = 0$. Equation 3.6 shows the updating rule for η_i, where η_o is the initial value of the adaptation rate, τ controls the decay rate, and i is the age of the detector. The movement of each detector is based on adaptation rate, the current position (center) of the detector,

and the direction in which to move the detector. The direction either takes the form of a positive or negative one, and is calculated based on the shortest calculated distance to any self point or detector. The nearest self point or detector center is stored along with the shortest distance computed to this point; the direction is found by equation 3.7, where *c* represents the nearest point. Finally, the new location of the detector is determined by the equation, $d(i+1) = d(i) + \eta_i * dir$, where $d(i)$ is the current position (center) of the detector, and $d(i+1)$ is the new position of the detector.

Adaptation Updating Rule : $$\eta_i = \eta_o e^{\frac{-i}{\tau}} \quad \textbf{(3.6)}$$

Direction Computation : $$\frac{\sum_{j=1}^{n}(d_i - c^{nearest})}{\left|\sum_{j=1}^{n}(d_i - c^{nearest})\right|} \quad \textbf{(3.7)}$$

Each detector is assigned an age which is incrementally increased after each iteration of detector movement, provided that its calculated distance is less than *r* for any self point or previously stored detector. Each time the detector is moved, the age increases by one until the detector reaches the maturity age *t*. If the detector reaches the maturity age *t* and has not been able to move out of the self subspace, it is eliminated and a new detector is randomly generated to replace it. If the detector is able to move out of the self subspace, the age is reset to zero and the detector is stored.

The maturity age is used to discard detectors which are not able to relocate a distance *r* from existing detectors and self points. There are two cases that require this necessity. Because the adaptation rate decays with each movement, it may never be moved far enough outside of the self subspace. The more likely case concerns self points and previously stored detectors, in which the detector is moved in the positive direction outside of the self subspace, but in turn relocates within the detection area of a previously stored detector. The next iteration of movement will cause the detector to be relocated in the negative direction, back into the self subspace. This pattern could repeat infinitely until the maturity age condition is met.

The stopping criterion for the real-valued negative selection algorithm using fixed sized detectors is based on a pre-specified number of detectors. This is not the best approach, and obviously provides no guarantee that the non-self space is completely covered. However, by selecting a large enough value for the number of detectors, the algorithm is expected to provide adequate results. Figure 3.5 provides pseudo-code for the generation phase of the algorithm.

After the generation phase has completed, the algorithm begins the detection phase. Once a predefined number of detectors are generated, each individual unknown data instance is presented to the detector set. The distance metric is applied for every detector in the detector set, and if the calculated distance is less than *r* for any detector, the detector is said to have detected that data instance. By definition of the negative selection algorithm, if a data instance is detected, it is classified as non-self. If no detectors are capable of detecting an unknown data instance, it is classified as self.

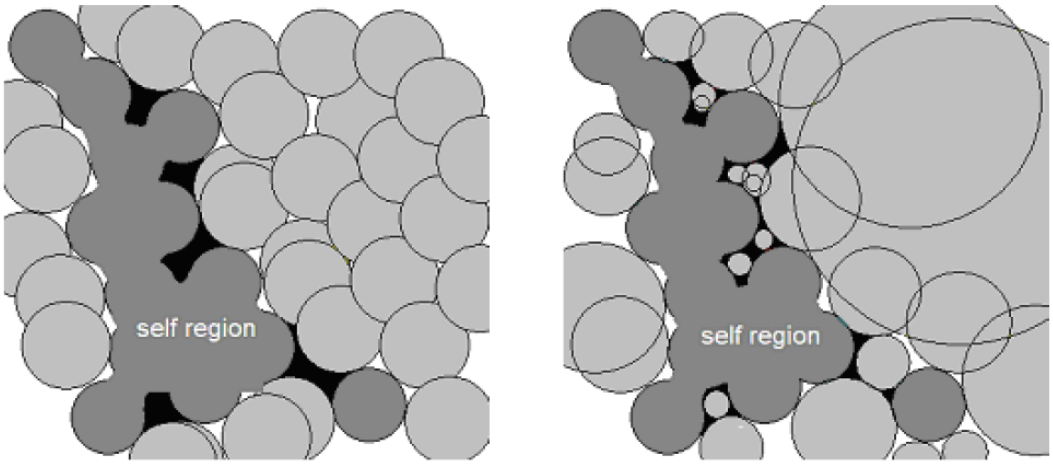
```
Real-Valued Negative Selection Algorithm with Fixed Detection Radius
Preset Control Parameters: r, η_o, t, τ,  # of Detectors
Generate a random population of Detectors based on # of Detectors
For each detector d_i,
        Calculate shortest distance to any self point, dist_min, and store nearest point c_i
        While (dist_min < r)
                If age > t
                        Generate new Detector d_i,
                Else
                        Calculate direction (dir) using c_i,
                        Calculate η_i,
                        Move detector by: d_(i+1) = d_i + η_i *dir
                        Increase age + 1,
                        Recalculate dist_min and c_i,
                End If
        End While
        If (Not the first detector),
                Calculate shortest distance to all previous detectors and self points, dist_min2,
                and store nearest point c_i,
                While (dist_min2 < r)
                        If age > t
                                Generate new Detector d_i,
                        Else
                                Calculate direction (dir) using c_i,
                                Calculate η_i,
                                Move detector by: d_(i+1) = d_i + η_i *dir
                                Increase age + 1,
                                Recalculate dist_min2, c_i
                        End If
                End While
                Store detector as d_i,
        Else
                Store detector,
End
```

Figure 3.5: Real-Valued Negative Selection Algorithm Pseudo-code

This concludes the explanation of the real-valued negative selection algorithm using a fixed-sized radius of detection. The next algorithm discussed is a more elegant approach to the negative selection algorithm which incorporates variable-sized detectors.

3.3 Negative Selection Algorithm with Variable-Sized Detectors

The first implementation of the real-valued negative selection algorithm generated detectors in which the distance threshold (or radius) was constant throughout the entire detector set. However, the detector features can reasonably be extended to overcome this limitation. Zhou and Dasgupta proposed a new scheme of detector generation and matching mechanisms for negative selection algorithms which introduced detectors with variable properties [17]. The proposed algorithm includes a new variable parameter, which is the radius of each detector. The threshold used by the distance matching rule defines the radius of the detectors; it is an obvious choice to make variable considering that the non-self regions covered by detectors are likely to be variable in size. The flexibility provided by the variable radius is illustrated in Figure 3.6[16].



a) Constant-Sized Detectors b) Variable-Sized Detectors

Figure 3.6: Comparison of Detector Coverage for Different Detector Schemes

Figure 3.6 actually illustrates several core advantages to the method of implementing variable-sized detectors. The first apparent advantage is that a larger area of non-self space is covered by fewer detectors. The issue of "holes" is a well-known problem with real-valued negative selection algorithms. Tiny spaces between detectors and self points cannot be filled by constant-sized detectors, as illustrated in black in the Figure 3.6 (a). However, by using variable-sized detectors as shown in Figure 3.6 (b), smaller detectors can be generated to cover small holes while larger detectors cover the wider non-self space.

Another advantage of the variable-sized detector method not shown in Figure 3.6 is that estimated coverage, instead of the number of detectors, can be utilized as a control parameter. As the detector set is generated, the algorithm can automatically evaluate the estimated coverage, providing a much more useful stopping criterion. This is discussed in greater detail later in this section.

The variable-sized detector negative selection algorithm, or *V-detector* algorithm, functions similarly to the fixed-sized radius algorithm discussed previously. First, a set of predefined control parameters must be initialized. The most influential of these parameters is the self threshold, or radius r_s. Because the detectors no longer share the same fixed radius, distinction must be made between the self radius r_s and the detector's variable radius r_d. The remaining two control parameters that determine the stopping criteria are the estimated coverage c_o and the maximum number of detectors D_{max}. Obviously, the eloquence and simplicity begins to become apparent as the control parameters (η_o, *t*, τ, *dir)* required to move each detector are eliminated, making the initialization of the *V-detector* algorithm much easier than the previous version.

The generation phase of the *V-detector* algorithm begins by randomly generating detector candidates; but instead of generating a full set of detectors determined by a fixed control parameter, it generates detector candidates one at a time. Each individual candidate is checked using the matching rule determined by the choice of distance metric. If the distance to the nearest self point is less than the threshold value (self radius r_s), the detector is eliminated and a new candidate is generated. If the minimum distance to any self point is greater than the self radius r_s, then the detector is stored temporarily (the reason the detector is only stored temporarily is discussed later) and the radius is recorded as r_d, based upon the minimum distance to the nearest self point. This is known as the aggressive approach to assign a detector's radius [16]. Detectors are iteratively generated and assigned a radius based on this simple mechanism until the stopping criteria is achieved.

A more conservative approach to detector radius assignment can also be implemented, whereas the detector radius r_d is assigned as the difference between the nearest self point c and the threshold radius r_s of the nearest self point [17]. Both implementations were initially tested, and the more aggressive strategy proved to produce more accurate results, and consequently was the method chosen for this study. Chapter 5 discusses how minor modifications to this aggressive strategy can produce optimized results. Figure 3.7 shows how the conservative detector radius is determined. Figure 3.8 (a & b) illustrates the differences between the conservative and aggressive approaches for variable radius detectors.

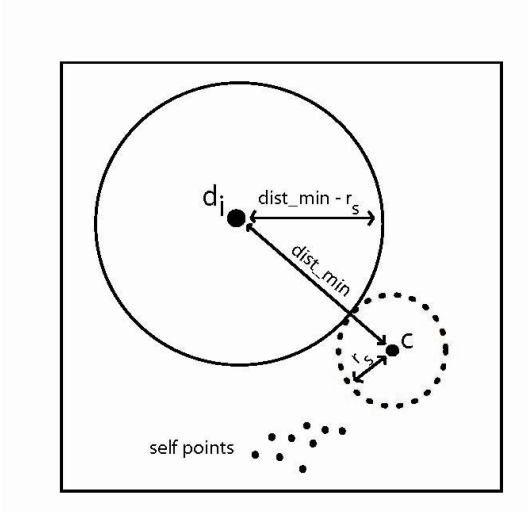
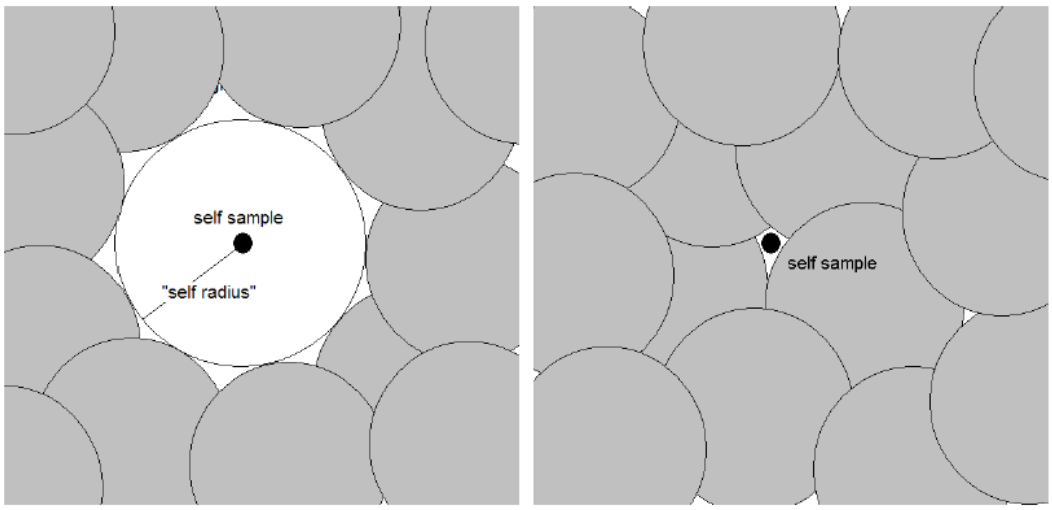


Figure 3.7: Calculating the Conservative Variable Detector Radius



a) Conservative Approach b) Aggressive Approach

Figure 3.8: Comparison of Detector Coverage Around a Self Sample

The control parameters of the *V-detector* algorithm consist of the self radius r_s, the estimated coverage c_o, and the maximum number of detectors D_{max}. The latter two are the central mechanisms for the stopping criteria; the maximum number of detectors is preset to allow the maximum allowable detectors in practice. Estimated coverage is a

by-product of the variable detector algorithm. When a detector candidate is generated and assigned a radius r_d based on the implementation described previously, it is not permanently stored for the detection phase. The detector candidate is then checked to determine if it can be detected by any previously stored detector. If the detector is detected, it is eliminated, and the attempt is recorded in a counter which will be used to estimate coverage. If the detector is not detected by any previously stored detectors, it is stored permanently for the detection phase and counter is reset to zero. If the counter of consecutive attempts that fall on covered points reaches a limit m_{max}, the generation stage finishes with enough confidence that the coverage is sufficient enough to cover the nonself space [15].

The limit of the counter m_{max} is decided by the estimated coverage, i.e., $m_{max} = 1 / (1-c_o)$. Assume "1" is for full coverage. If there is one uncovered point in a set of m samples, then the estimated uncovered region is $1/m$; i.e., the estimate of coverage is $c_o = 1- 1/m$ [15]. For example, for 99% estimated coverage, ($c_o = 0.99$), $m_{max}=100$.

The *V-detector* algorithm converges in one of two ways based on the stopping criteria. The first convergence scenario occurs when the estimated coverage is attained. This is the preferred method of convergence, as it displays the power of the *V-detector* algorithm to control the number of detectors generated. The alternative convergence scenario is when the limit of maximum detectors is reached. While not desirable, it still has the potential to cover more holes than the basic fixed-sized detector negative selection algorithm. Figure 3.9 provides pseudo-code for the generation phase of the V-detector algorithm.

```
Real-Valued Negative Selection Algorithm with Variable Detection Radius
Preset Control Parameters: r_s, m_max, D_max
While (m < m_max) || (i < D_max)
        Generate a random Detector candidate d_i,
        Calculate shortest distance to any self points, dist_min,
        If (dist_min < r_s)
                Return to top,
        Else
                If ( i = 1)
                        Store detector as d_i and dist_min = r_di,
                        Increment i + 1
                Else
                        Calculate shortest distance for each previous detector, dist_min2,
                        If (dist_min2 < r_d)
                                m = m + 1,
                        Else
                                Store detector as d_i and dist_min2 = r_di,,
                                Increment i + 1
                                m = 0,
                        End If
                End If
        End If
End While
End
```

Figure 3.9: Real-Valued Negative Selection V-Detector Algorithm Pseudo-code

The detection phase of the *V-detector* algorithm is almost exactly the same as the fixed-sized detector algorithm. The only exception is the detector threshold utilized for the unknown data detection is based on the variable radius r_d assigned to each detector. If an unknown data instance is detected (i.e. the minimum distance to any detector is less than r_d), it is classified as non-self, otherwise it is classified as self.

3.3 Negative Selection Algorithm with Proliferating Variable-Sized Detectors

One of the most recent advances in real-valued negative selection algorithms incorporates the implementation of proliferating variable-sized detectors [3]. This method, referred to as the *proliferating V-detector* algorithm, consists of three stages. It begins with a generation stage, (very similar to the standard *V-detector* algorithm), followed by a new proliferation stage, and finally the detection stage.

During the generation phase, the detector set is filled with an initial set of detectors in the same manner as the generation phase for the *V-detector* algorithm. The only difference is the assignment of the variable radius r_d. Recall two methods were described for the variable radius assignment, either the aggressive or conservative approach. The minimum distance *dist_min* is calculated from a single detector to the nearest self point, and the variable radius r_d is assigned accordingly: 1) aggressive method $r_d = dist\_min$; 2) conservative method $r_d = (dist\_min - r_s)$. The *proliferating V-detector* algorithm includes an additional threshold term θ which is also subtracted from the variable radius r_d. In relation to the two methods described above, the aggressive variable radius would yield $r_d = (dist\_min - \theta)$, and the conservative variable radius assignment would result in $r_d = (dist\_min - r_s - \theta)$. The implementation in this study is the aggressive approach.

After the generation phase concludes, the proliferation stage begins to proliferate (or clone) new detectors from the detector set initially created from the generation stage. These new detectors are referred to as offspring. At the beginning of the proliferation stage, the algorithm already has a set of detectors *D* from the previous generation stage.

In the *ith* iteration, it selects one of those detectors whose center and radius are x_i and r_i from the set *D*, and creates new offspring located at a distance r_i from x_i. In two dimensions, the original detector is regarded as a circle of radius r_i in the nonself region centered around x_i, and the offspring detectors will be located along the circle's circumference at a location $x_i + \hat{u}r_i$, where $\hat{u}$ is some unit direction vector [3]. The offspring's radius is set to be equal to the minimum distance from its center to the nearest self point, but modifications exist with the introduction of an additional threshold θ.

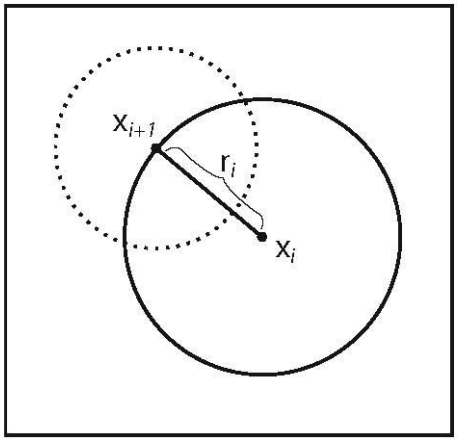


Figure 3.10: Proliferation of a Detector

Offspring coverage is controlled in the same manner as the detector generation phase of the *V-detector* algorithm. Since a new detector has additional coverage value only when another does not already cover the space, only those offspring detectors which are not covered are retained for the detection phase. The detectors in D are selected for proliferation in a sequential manner, and in this implementation the unit vectors $\hat{u}$ are kept to be either parallel (+1) or anti-parallel (-1) to each dimension. Hence, in a two dimensional input space, there are four possible values of $\hat{u}$: (1, 0), (-1, 0), (0, 1), and (0, -

1). In a three dimensional input space, there are six such vector, eight for four dimensions, ten for five dimensions, and so on.

The proliferation stage may involve more than one stage of proliferation. Several stages of proliferation, where the offspring from one stage is allowed to proliferate in the next stage, are often desirable. Maintaining the threshold θ initially high during the first the first generation stage, and lowering it towards zero in a stepwise manner during subsequent proliferation stages, can result in much better coverage of the non-self subspace. This is because decrementing the threshold θ at the end of each stage creates a gap between the self / non-self boundary. This gap can then be filled by the offspring detectors of the next proliferation stage. Steadily decreasing the gap by lowering θ results in increasingly smaller, but strategically placed offspring to proliferate around the self / non-self boundary region. To ensure full coverage of the non-self subspace, the threshold θ must be set to zero during the last stage of proliferation [3]. Figure 3.11 illustrates this concept, where r_d represents the radius of each detector.

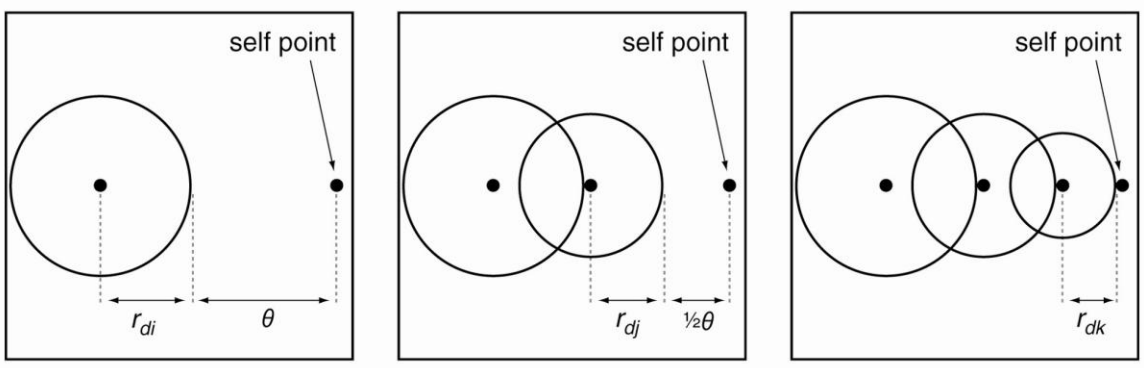


Figure 3.11: Examples of each Stage of Detector Proliferation

As mentioned previously, this study implements an aggressive approach for the assignment of the variable radius r_d. For the implementation of the *proliferating V-detector* algorithm, an additional threshold term θ is required. The proposed algorithm for this study takes advantage of the self threshold radius r_s, and assigns it to the value of the required threshold θ. Utilizing this method, the initial generation phase is no different than the conservative approach for variable detector generation and radius assignment. In subsequent proliferation stages, the threshold value r_s are reduced by 50%, 25%, and finally zero. Two implementations are carried out for this study, one involving three stages of proliferation, and one comprising only two stages. A more thorough discussion of these implementations is covered in Chapter 5. Pseudo-code for the implementation of the proliferation stage is presented at the end of this chapter. New code is not necessary for the generation phase, as it remains relatively unchanged from the *V-detector* generation algorithm.

The detection phase of the *proliferating V-detector* algorithm remains completely unchanged from the basic *V-detector* algorithm. A variable radius threshold is assigned to each detector, and a distance measure is calculated for each unknown data instance. Detection results in the classification of non-self; those not detected are classified as self.

Real-Valued Negative Selection Algorithm with Proliferating Variable Detectors
(Proliferation Stage only)
D_{old} includes all detectors generated in initial generation phase
Note ($\theta = r_s$) in generation phase
$i=1$,
$\theta = .5 * r_s$,
For each $d_i (x_i, r_i)$ in D_{old}
    For each unit vector $\hat{u}$ (determined by dimension n of training data)
        $x_j = x_i + \hat{u}\, r_i$,
        Calculate distance to nearest detectors, *dist_min*,
        If ($dist\_min < r_i$)
            $i = i + 1$,
            Return to top,
        Else
            x_j stored into D_{new},
            Calculate distance to nearest self point, *dist_min2*,
            $r_j = dist\_min2 - \theta$,
            $i = i + 1$,
            $j = j + 1$,
            Return to top,
        End If
End 1st Proliferation Stage
Begin 2nd Proliferation Stage
$j=1$,
$\theta = .25 * r_s$,
For each $d_j (x_j, r_j)$ in D_{new}
    For each unit vector $\hat{u}$ (determined by dimension n of training data)
        $x_k = x_j + \hat{u}\, r_j$,
        Calculate distance to nearest detectors, *dist_min*,
        If ($dist\_min < r_j$)
            $j = j + 1$,
            Return to top,
        Else
            X_k stored into D_{new2},
            Calculate distance to nearest self point, *dist_min2*,
            $r_k = dist\_min2 - \theta$,
            $j = j + 1$,
            $k = k + 1$,
            Return to top,
        End If
End 2nd Proliferation Stage
Repeat for each stage, decrementing θ for each subsequent stage until $\theta = 0$
End

Figure 3.12: Negative Selection Proliferating V-Detector Algorithm Pseudo-code

CHAPTER 4

Neural Networks

4.1 Background

The brain is a highly complex, nonlinear information processing system. It has the capability to organize its structural constituents, known as neurons, to perform certain computations many times faster than the fastest computer in existence today. Examples of the brain's computational functions include pattern recognition, perception, and motor control. Motivated by recognizing that the human brain computes in an entirely different way from conventional digital computers, researches have adopted this structure into a computational model known as artificial neural networks [13].

In its most general form, an artificial neural network is an information processing system that is designed to model the way in which the brain performs a particular task or function. The fundamental information processing unit in the human brain is the neuron, and likewise is the essential building blocks of any neural network. A neural network is a massively parallel distributed processor made up of simple processing units (neurons) that have a natural propensity for storing experiential knowledge and making it available for use. Like the brain, knowledge is acquired by the network from its environment (data) through a learning process; interneuron connection strengths, known as synaptic weights, are used to store the acquired knowledge [13].

The original neural network models date back to the 1940's, and was only able to solve simple linear problems based on simple binary decision units. The early implementations of neural networks only included an input and output layer, and were only capable of classifying linearly separable data patterns. Further investigation and development led to the inclusion of a hidden layer and more a complex architecture for each neuron. This allowed the neural network models to begin to solve more complex nonlinear problems. It was not until the invention of back propagation in the 1980's that neural networks finally began to realize their potential as an adaptive learning machine.

An abundance of research has been conducted within the field of artificial neural networks. The procedure used to perform the learning process, called the learning algorithm, concerns the modification of the synaptic weights of the network in an orderly fashion to attain a desired learning objective. The modification of the synaptic weights has provided researchers with various implementations in the design of neural networks. The modification of the topology of neural networks has also caught the interest of many researchers motivated by the fact that neurons in the brain often die and new synaptic weights are allowed to grow in their place.

Neural network applications offer a wide variety of useful properties and beneficial capabilities. Neural networks have a built-in capability to adapt their synaptic weights to changes in their environment. This allows applications in input-output mapping and the solving of both linear and nonlinear problems. It can be applied to pattern recognition and data classification, where contextual information is dealt with naturally by the network. From a hardware perspective, neural networks have the potential to be inherently fault tolerant, or capable of robust computation due to the

distributed nature of information stored in the network. Due to the massively parallel nature of a neural network, it is well suited for the implementation very-large-scale-integrated (VLSI) technology [13]. The list of applications and benefits go on, but suffice it to say it makes for a perfect candidate for comparison to the artificial immune system negative selection algorithm.

4.2 Artificial Neural Network Model

Artificial neural networks are suitable for cases where the input-output classification of data is known, but no distinguishable pattern can be easily modeled to determine the distinction. The artificial neural network approach is a generic technique for mapping the relationship between inputs and outputs and requires less expertise and experimentation than traditional modeling of non-linear multivariate systems. The neural network learns the input-output mapping of a system through an iterative training and learning process. It contains the built-in ability to update its acquired knowledge on-line for each iteration of training. This automatic learning property makes a neural network based system inherently adaptive and ideal for data classification [24].

The artificial neural network model implemented in this study is a multilayer feedforward network trained with back propagation. The fundamental unit of this model is the neuron, known as a multilayer perceptron (MLP). Figure 4.1 illustrates the basic concept of a multilayer perceptron. The input signals x_i are multiplied with their respective weights w_i and then summed together along with the bias b_i of each node to form the intermediate value v_i. The weighted connections w_i can take on either a positive value (exciter) or negative value (inhibitor) to guide the output signal to the desired

value. The intermediate value v_i is subjected to an activation function f_i that transforms the net input of the perceptron depending on the desired range of the output. The final result of the perceptron is the output value y_i [21].

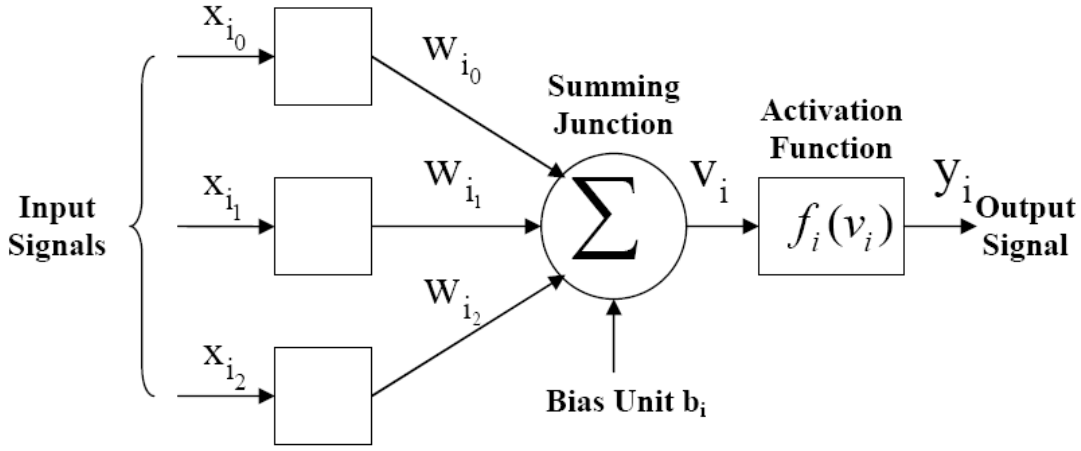


Figure 4.1: Basic Structure of a Multilayer Perceptron [21]

The general layout of a fully constructed feedforward network consists of an input layer, hidden layers, and an output layer. The input layer receives the first set of training data, such as an n-dimensional vector of data whose desired output is known. The hidden layers consist of an interconnected network of multilayer perceptrons to perform the learning process. The final layer of the neural network is the output layer, which produces a final output based on the classification criteria. The output layer could be as simple as producing a '1' for self or '0' for nonself, if related to the artificial immune system negative selection algorithm. Figure 4.2 shows the architecture of an artificial neural network model utilizing multilayer perceptrons.

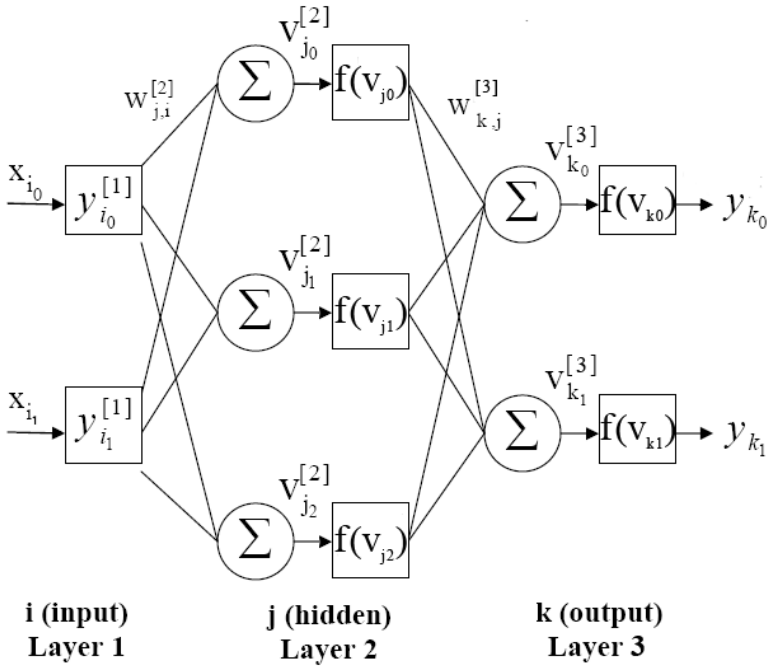


Figure 4.2: Architecture of an Artificial Neural Network

From Figure 4.2, the value specified by the superscript $w^{[\#]}$ represents the current layer of the variable shown. The subscript represents the node at which the variable is located, and the case of multiple subscripts such as $w_{j,i}$, the weight $w_{j,i}$ is stated as connecting node '*j*' in the current layer to node '*i*' from the previous layer. In terms of each multilayer perceptron, the intermediate value v_i for a node in a particular layer is calculated according to Equation 4.1, where *N* represents the total number of nodes in the previous layer. The output of the same multilayer perceptron is then calculated according to the activation function *f*, and is defined in Equation 4.2. The activation function can take on many forms designated by the desired output for data classification.

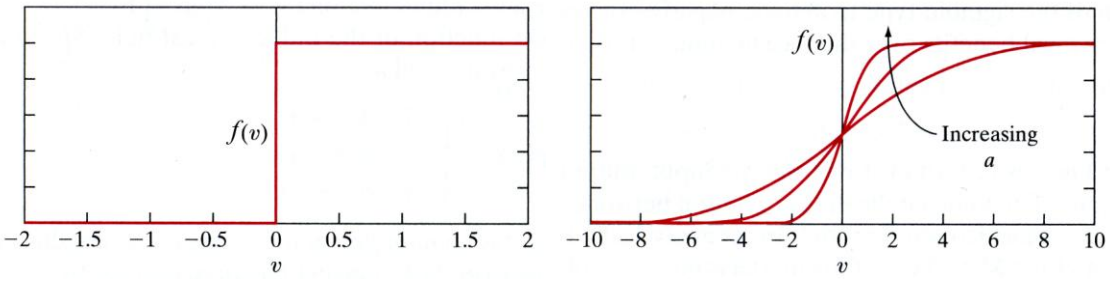
Intermediate Value : $$v_i = \sum_{i=1}^{N} w_{j,i}\ y_{j,i} \quad \textbf{(4.1)}$$

Output Value : $$y_i = f(v_i + b_i) \quad \textbf{(4.2)}$$

The activation function, denoted by $f(v_i)$, defines the output of a neuron in terms of the intermediate value v_i. The most basic activation function is the threshold function, where any positive value of v_i outputs a '1', and any negative value outputs a '0' (equation 4.3). This function is primarily implemented for data sets which require simple binary outputs. The next activation function, the logistic function, performs in a similar manner to the threshold function, except the output takes on a value between [0, 1]. Figure 4.3 illustrates the subtle differences between the threshold and logistic activation functions.

Threshold Function : $$f(v_i) = \begin{cases} 1, & v_i \geq 0 \\ 0, & v_i < 0 \end{cases} \quad \textbf{(4.3)}$$

Logistic Function : $$f(v_i) = \frac{1}{1 + e^{-\alpha v_i}} \quad \textbf{(4.4)}$$



a) Threshold Function b) Logistic Function

Figure 4.3: Plots of Different Activation Functions [13]

When the desired range of the output is [-1, 1], the logistic function is often replaced by the hyperbolic tangent function, expressed as $f(v_i) = tanh(\alpha v_i)$. Equation 4.5 shows a more practical implementation of the hyperbolic tangent function. It is worth noting that each of the previously mentioned activation functions accept inputs within the range $[-\infty,\infty]$. When the classification of data sets requires multiple outputs, and each output belongs to a different class, the softmax function is an ideal choice. The softmax function, presented in Equation 4.6, forces all of the outputs to sum up to one. Each output of the softmax function is interpreted as probabilities that the input is of a specific type [21].

Hyperbolic Tangent Function :

$$f(v_i) = \frac{e^{\alpha v_i} - e^{-\alpha v_i}}{e^{\alpha v_i} + e^{-\alpha v_i}} \tag{4.5}$$

Softmax Function :

$$f(v_i) = \frac{e^{v_i}}{\sum_{j=1}^{C} e^{v_j}} \tag{4.6}$$

4.3 Learning Process of an Artificial Neural Network

A properly trained neural network must configure its parameters so that the given inputs yield an output which matches the desired outputs. To correspond with the real-valued negative selection algorithms, the neural network model proposed in this study has only a single output node to discriminate between self and non-self data. To begin, first let a training sample be denoted by (x_k, d_k), where x_k is the stimulus applied to the input layer and d_k is the desired output for that specific input. Let y_k denote the actual output

produced by the input x_k at the output layer of the neural network. Correspondingly, the error signal produced at the output layer is defined as $e_k = d_k - y_k$. This is the instantaneous error for one output associated with one input pattern in the training set. From this metric, the general measure of a neural network's performance is defined as the mean squared error, (where P is the total number of input training patterns). The mean squared error (MSE) is the basis for the stopping criteria of the network (equation 4.7).

Mean Squared Error :

$$MSE = \frac{1}{P}\sum_{p=1}^{P}\frac{1}{2}\varepsilon_p^2 \qquad \textbf{(4.7)}$$

Instantaneous Mean Squared Error :

$$Instant\ MSE = \frac{1}{2}\varepsilon^2 \qquad \textbf{(4.8)}$$

The first decision to make when training a neural network is which type of supervised learning method to use. In this research, on-line learning is employed; that is, adjustments to the synaptic weights of each multilayer perceptron are performed on an example-by-example basis. The cost function to be minimized is the instantaneous mean squared error described above. The advantages of using on-line learning are its ability to track small changes in the training data, thereby providing effective solutions to difficult pattern-classification problems and ease of implementation [13].

There are a variety of options proposed and available to adjust the parameters of the network to achieve the desired input/output matching needed for proper data classification. One of the earliest and most popular of these options is the back propagation algorithm. The updated value of a synaptic weight is simply adjusted by the

addition of a correction term to the previous weight, $w_{j,inew} = w_{j,iold} + \Delta w_{j,i}$. The correction term is proportional to the partial derivative of the energy function with respect to the corresponding synaptic weight (equation 4.8). Neglecting the derivation, it is proven that this equation simplifies to the more elegant solution in equation 4.9 [13]. The learning rate η controls the changes to the synaptic weights in the network. The smaller the value of η, the slower the rate of learning; however, increasing the parameter too large may lead to the network become unstable (oscillatory).

Weight Correction Term : $$\Delta w_{j,i} = \eta \frac{\partial \varepsilon}{\partial w_{j,i}} \quad \textbf{(4.8)}$$

where the instantaneous error $\varepsilon = d_k - y_k$

Weight Correction Term (simplified) : $$\Delta w_{j,i} = \eta \delta_j y_i \quad \textbf{(4.9)}$$

The term δ_j, referred to as the local gradient, defines the required changes in the synaptic weights based on the activation function and instantaneous error signal. The local gradient is defined separately for the cases when the neuron is an output node or a hidden node. For an output node j, the local gradient δ_j is equal to the product of the corresponding error signal e_j for that neuron and the derivative $f_j'(v_j)$ of the associated activation function. The activation function implemented in this study is the logistic function, and the associated derivative simplifies to Equation 4.10 [13]. Therefore, in the case of an output neuron j, the local gradient δ_j is defined as Equation 4.11.

Derivative of Logistic Function : $$f_j'(v_j) = a\, y_j (1 - y_j) \quad \textbf{(4.10)}$$

Local Gradient for Output Neuron *j* : $$\delta_j = e_j a\, y_j (1 - y_j) \quad \textbf{(4.11)}$$

When neuron *j* is located in a hidden layer of the network, there is no specified desired response for that neuron. The error signal for a hidden neuron must be determined recursively, working backwards in terms of the error signals of all neurons to which that hidden neuron is directly connected. This is where the name back propagation originates. Equation 4.12 describes the back propagation formula for the local gradient of a hidden neuron *j* after simplifying the derivative of the logistic function [13]. The formula utilized to update the synaptic weights is now generalized to Equation 4.13.

Local Gradient for Hidden Neuron *j* : $$\delta_j = a\, y_j (1 - y_j) \sum_{k}^{K} \delta_k w_{k,j} \quad \textbf{(4.12)}$$

Synaptic Weight Update Formula : $$w_{j,i_{new}} = w_{j,i_{old}} + \eta \delta_j y_i \quad \textbf{(4.13)}$$

For the on-line learning approach utilized in this study, an input sample pattern is fed into the network and an error signal is produced. The error signal is then back propagated through the network to adjust the synaptic weights for each neuron. The iteration of forward and backward computations repeats until all input samples within the training set have been exhausted. The order of the training samples is then randomly rearranged and another training pass is conducted. This training continues to repeat until a preset number of iterations are reached. After the preset number of training iterations

completes, the weights are fixed and the neural network calculates the average *MSE* from all input-output pairs. If the *MSE* is less then some preset threshold, MSE_{th}, then the algorithm terminates and testing begins. If the *MSE* is greater than the threshold, it resumes training for another preset number of iterations.

The testing phase of the neural network is very similar to the detection phase of the negative selection algorithm. Each unknown data instance is presented to the algorithm, and the network produces an output corresponding to the class in which the data belongs. To remain consistent with the negative selection algorithm, the neural network algorithm produces an output value between [0, 1]. A decision threshold of 0.5 either classifies the data as '1' (self) if $y_{out} \geq 0.5$ or '0' (non-self) if $y_{out} < 0.5$. Figure 4.4 provides pseudo code for the neural network algorithm on the following page.

Multilayer Feedforward Neural Network with Back Propagation
(1 Hidden Layer)
Initialize parameters: a, η, $bias\ (b)$, MSE_{th}, $iter\_count_{max}$
Randomly assign weights with zero mean, std = 1.0
$iter\_count=0$
Begin Training Phase:
While $(iter\_count<iter\_count_{max})$
    Randomly rearrange Training Set P1,
    For p=1, P1 (where P1 is total # input-output pairs of training set),
        Assign the input of training sample to y_i,
        Calculate $v_j = sum(y_i*w_{j,i}) + b_i*w_{bi}$,
        Calculate logistic function output, $y_j = 1 / (1 + exp(-a*v_j))$,
        Calculate v_k for output neuron, $v_k = sum(y_j*w_{k,j}) + b_j*w_{bj}$,
        Calculate logistic function output, $y_k = 1 / (1 + exp(-a*v_k))$,
        Calculate error signal, $e_k = (d_k - y_k)$,
    (Begin Back Propagation)
        Calculate local gradient of output, $\delta_k = a*e_k *y_k*(1-y_k)$,
        Update weights of output layer, $w_{k,jnew} = w_{k,jold} + (\eta * \delta_k*y_j)$,
        Calculate local gradient of hidden layer, $\delta_j = a*y_j*(1-y_j)*sum(\delta_k*w_{k,j})$,
        Update weights of hidden layer, $w_{j,inew} = w_{j,iold} + (\eta * \delta_j*y_i)$,
    End For
    $iter\_count=iter\_count+1$,
End While
Stopping Criteria:
Calculate $MSE = 1/P1\ sum(e_k^2 / 2)$ for $P1$ training samples
If ($MSE < MSE_{th}$)
    End Training Phase, move down to Testing Phase,
Else
    $iter\_count=0$,
    Resume Training Phase,
End If
Begin Testing Phase:
For p=1, P2 (where P2 is total # input-output pairs for testing set)
    Assign the input of testing sample to y_i,
    Calculate $v_j = sum(y_i*w_{j,i}) + b_i*w_{bi}$,
    Calculate logistic function output, $y_j = 1 / (1 + exp(-a*v_j))$,
    Calculate v_k for output neuron, $v_k = sum(y_j*w_{k,j}) + b_j*w_{bj}$,
    Calculate logistic function output, $y_k = 1 / (1 + exp(-a*v_k))$,
    If $y_k \geq 0.5$
        Classify as self
    Else
        Classify as non-self
    End if
End For

Figure 4.4: Multilayer Feedforward Neural Network Algorithm Pseudo-code

CHAPTER 5

Testing and Results

The purpose of this study is to evaluate the affects of different distance metric on three distinct implementations of the real-valued negative selection algorithm. The implementation of a multilayer feedforward neural network with back propagation is employed as a comparison model, traditionally utilized in the field of computational intelligence for data classification. This chapter discusses the datasets utilized in testing in meticulous detail. The discussion includes the methodology behind the implementation of each algorithm, along with the experimental techniques to optimize each algorithm. The study includes balanced testing procedures and explanations of experimental decisions to handle distinctions between the neural network and negative selection algorithms. The chapter concludes with experimental results and final conclusions based on these results.

5.1 Datasets

Three distinct datasets are used in the experiments implemented in this study. The first dataset is the famous *Fisher's Iris Dataset* [2], which has been widely used in discrimination analysis. The dataset consists of 50 samples from each of three species of *Iris* flowers (*Iris setosa*, *Iris virginica* and *Iris versicolor*). Four distinct features were

measured from each sample; the length and the width of sepal and petal. Therefore, the data set includes 150 total datasets, each a vector of four dimensions.

To better understand the distribution of the Fisher Iris dataset, plots were generated to graphically illustrate the datasets characteristics in two dimensions. Figure 5.1 shows the plot of the first two dimensions, sepal length and width, while Figure 5.2 provides the third and fourth dimensions, petal length and width. Before the plots were produced, the datasets were first normalized to values between [0, 1].

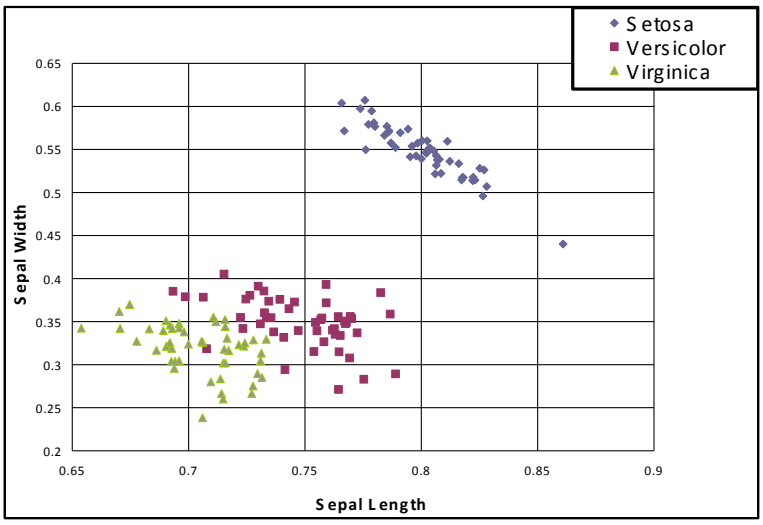


Figure 5.1: Distribution of 1st and 2nd Dimensions of Iris Dataset

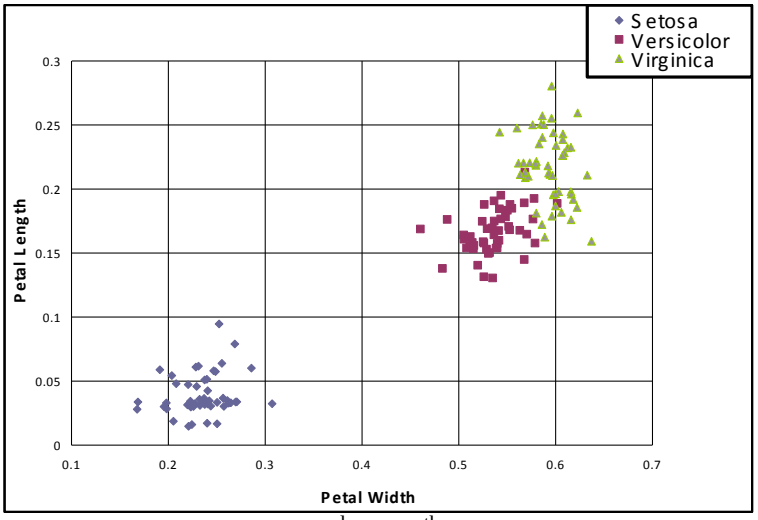


Figure 5.2: Distribution of 3rd and 4th Dimensions of Iris Dataset

The *Iris-Setosa* data shown in blue is clearly separated from the other two datasets, making classification easy. The remaining two datasets, *Iris-Versicolor* in violet and *Iris-Virginica* in green, are intermingled but centralized. While this makes data classification more difficult, the fact that each dataset is clustered close together makes discrimination less cumbersome than the next dataset to be discussed.

The second dataset, referred to as *Biomedical Data* [22], is from blood measurements of 194 patients, after removing those datasets which are missing data points. The dataset arose in a study to develop screening methods to identify carriers of a rare genetic disorder. Of the 194 datasets, 127 are classified as "normal" or free of the disorder, and the other 67 are identified as "carriers" of the disorder. Each patient had four different types of blood measurements, yielding a total of 194 data sets with four data points in each set.

Figures 5.3 provides perspective of the dataset's distribution for the first two dimensions, and Figure 5.4 displays the third and fourth dimensions. Clearly, the distribution of the *Biomedical Dataset* is much more complicated than the *Iris Dataset*. The normal dataset in blue is heavily intermingled within a cluster of carrier data points, and proves to be very difficult to discriminate precisely. The carrier dataset is slightly easier to classify because some outlier points are easily separable from the central cluster of data points.

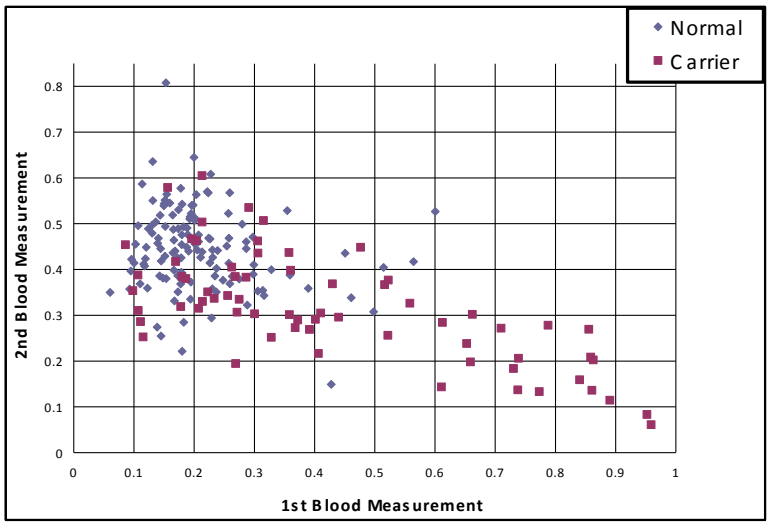


Figure 5.3: Distribution of 1st and 2nd Dimensions of Biomedical Dataset

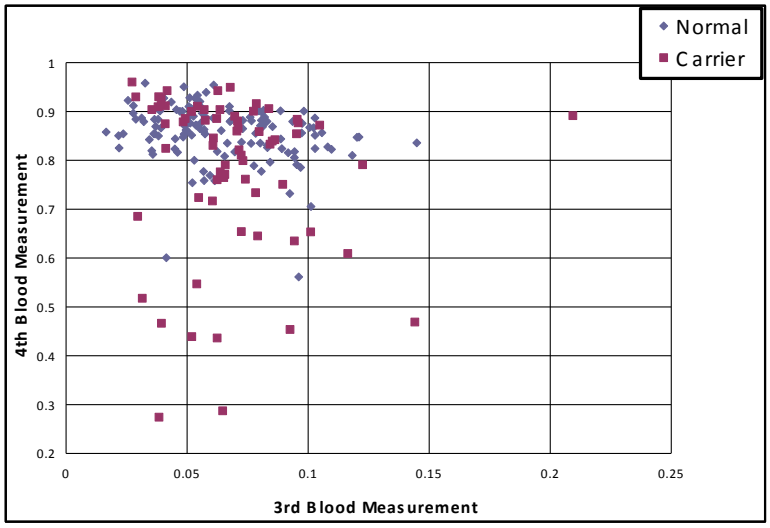


Figure 5.4: Distribution of 3rd and 4th Dimensions of Biomedical Dataset

The final dataset tested in this study is the *BUPA Liver Disorder* [2]. Performed by the BUPA Medical Research Ltd, the first 5 variables are all blood tests which are thought to be sensitive to liver disorders that might arise from excessive alcohol consumption; the last variable represents the number of alcoholic beverages consumed daily. The dataset comprises measurements of 345 patients, 200 of which

were designated "clean" from the disorder; the remaining 145 are labeled as "disorder". Figures 5.5 – 5.7 illustrate the distribution of the data for the 1st-2nd, 3rd-4th, and 5th-6th dimensions respectively. The complex distribution and increase in dimensionality and sample size over the *Biomedical Dataset* made this an ideal choice for the final dataset.

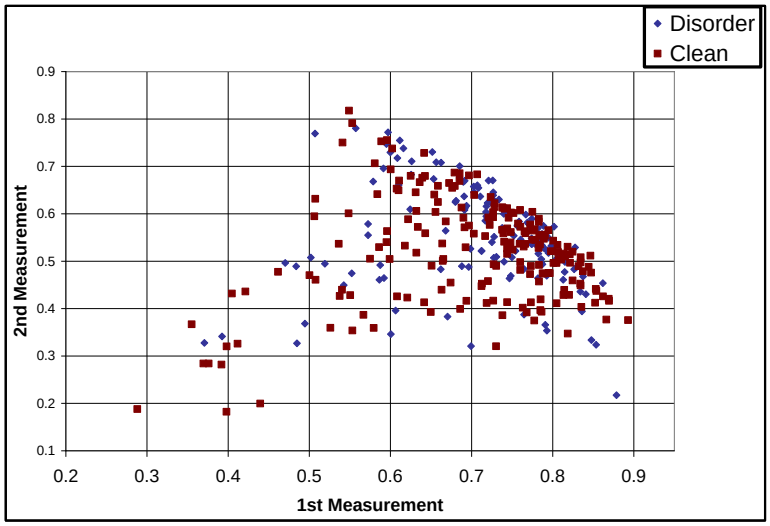


Figure 5.5: Distribution of 1st and 2nd Dimensions of BUPA Dataset

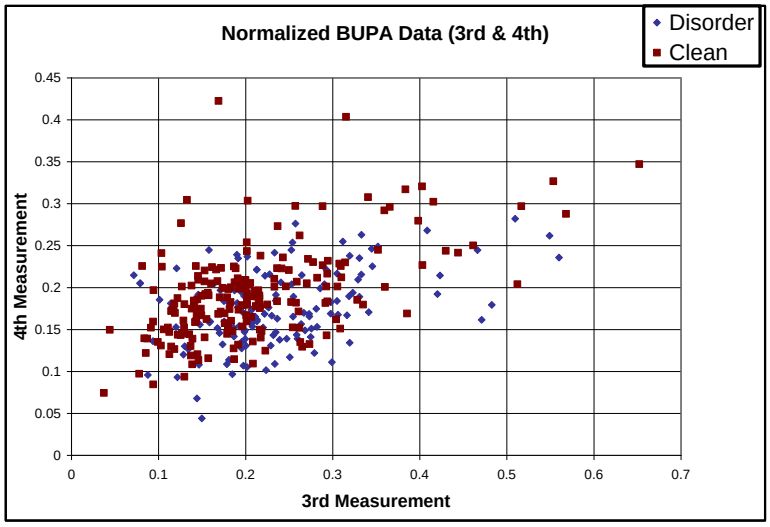


Figure 5.6: Distribution of 3rd and 4th Dimensions of BUPA Dataset

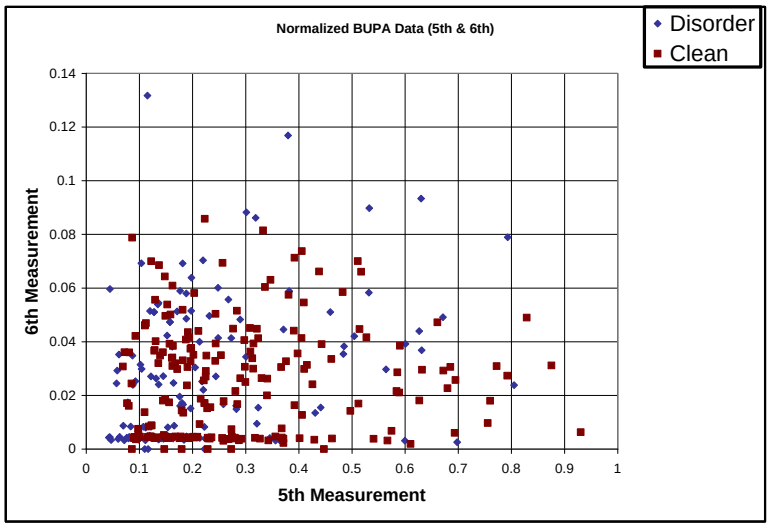


Figure 5.7: Distribution of 5th and 6th Dimensions of BUPA Dataset

5.2 Testing Methodology and Algorithm Optimization

This section describes the various testing methodologies and optimization techniques to produce the best possible results for each algorithm. It revisits several references to the algorithms proposed in the previous chapters and how minor adjustments can achieve optimal implementations. The section concludes by covering general similarities that each implementation shares and formally discussing the distinctions of each algorithm separately.

The general purpose of this study is to test an algorithm's ability to classify real-valued data. For the generation (or training) phase of each negative selection algorithm, the input data consists of only self data. In this study, self data is assigned separately to each class of data. In regards to the *Iris dataset*, one type of flower is designated as self, while the other two are assumed non-self. Therefore, three separate tests are conducted for the *Iris dataset*, one for each class of flower assigned as self. Since the remaining two datasets only have two classes, only two separate tests are conducted for each dataset.

The self data assignment to a dataset is further separated into different test cases. The methodology implemented in this study analyzes two cases, one in which the negative selection algorithm is trained with 100% of the self data class, and the other is trained with only 50% of the self data class. This results in fourteen separate tests for a single negative selection algorithm with a specific distance metric. Each test is trained with the following self data classes: *100% Setosa, 50% Setosa, 100% Versicolor, 50% Versicolor, 100% Virginica, 50% Virginica, 100% Normal, 50% Normal, 100% Carrier, 50% Carrier, 100% Clean, 50% Clean, 100% Disorder and 50% Disorder*.

There is a major distinction between the negative selection and neural network algorithm. While a negative selection algorithm, by design, requires training of only one class of data, the neural network algorithm must be trained with samples from both classes of data. The results section of this chapter provides evidence to support this claim, and led to modifications to the training data to address this issue. The final portion of this section will address these changes along with a formal discussion of the implementation of the neural network model.

Originally introduced by the first implementation of a real-valued negative selection algorithm, two performance metrics are utilized to evaluate their effectiveness, the *detection rate* and *false alarm rate* [10]. The detection rate (DR) is defined as the number of correctly identified non-self points divided by the total number of non-self data points multiplied by 100%. This yields a percentage of correctly identified non-self points, signifying how well the algorithm detected anomalies. Conversely, the false alarm rate (FA) is calculated as the number of self points classified incorrectly divided by the total number of self data points. This produces a percentage of self points classified

incorrectly, signifying how poorly the algorithm misclassified self data as an anomaly. A figure of merit (FOM) is formulated for the need to determine an overall final score for the performance of the algorithm, which is defined as the false alarm rate subtracted from the detection rate (DR-FA). The figure of merit is a method of comparing how well the algorithm detects anomalies while simultaneously penalizing it for self misclassifications.

The real-valued negative selection algorithm with a fixed-sized radius is the first model implemented in this study. The initialization of the control parameters vary for each dataset to achieve the best performance. For the *Iris Dataset*, the adaptation rate $\eta_o = 0.005$, the decay rate $\tau = 15$, the maximum age $t = 15$, and the total number of detectors is *1000*. For the *Biomedical Dataset*, the adaptation rate $\eta_o = 0.0025$, the decay rate $\tau = 10$, the maximum age $t = 15$, and the total number of detectors is *1000*. For the *BUPA Dataset*, the adaptation rate $\eta_o = 0.0025$, the decay rate $\tau = 10$, the maximum age $t = 15$, and the total number of detectors is *5000*. The major difference for the *BUPA dataset* implementation is the total number of detectors generated, which was required to produce adequate coverage of the non-self space.

Several experimental tests are performed to decide the ideal values of control parameters. The most crucial control parameter i.e., detector radius *r*, requires extensive analysis to determine the optimal value. The worst case scenario defined as the most difficult dataset implementation to correctly classify is identified for each dataset: *1) Iris Dataset = 50% Virginica, 2) Biomedical Dataset = 50% Normal, and 3) BUPA Dataset = 50% Clean.* Five seed detectors sets are randomly generated for implementation of various detector radii. The FOM proposed earlier is the basis for measuring the efficiency of each test, and is averaged over the five seed detector results to yield an

overall percentage of accuracy. Understanding that each distance metric will attain optimal results for different detector radii, each distance metric is tested for each dataset. The optimization results are plotted with the radius along the x-axis and the average FOM along the y-axis, and are displayed in Figures 5.8 - 5.10.

Similar testing strategies are employed to determine an optimal number of detectors for the *BUPA Dataset*. The initial attempts to optimize the *BUPA* data are highly unsuccessful with only 1000 detectors. The five seed detectors are again utilized using only the Euclidean distance measure to determine an adequate number of detectors to produce sufficient results. While even at 5000 detectors the Euclidean FOM scores seemed low, by performing optimization techniques for the remaining distance metrics it is concluded that 5000 detectors is sufficient. Increasing beyond 5000 detectors required extensive time consumption (48-72 hours), and often resulted in algorithm failure due to the impossibility to 'fit' more detectors into the non-self subspace. Figure 5.11 shows a plot of the effects of increasing detector counts corresponding to a change in radius and FOM score.

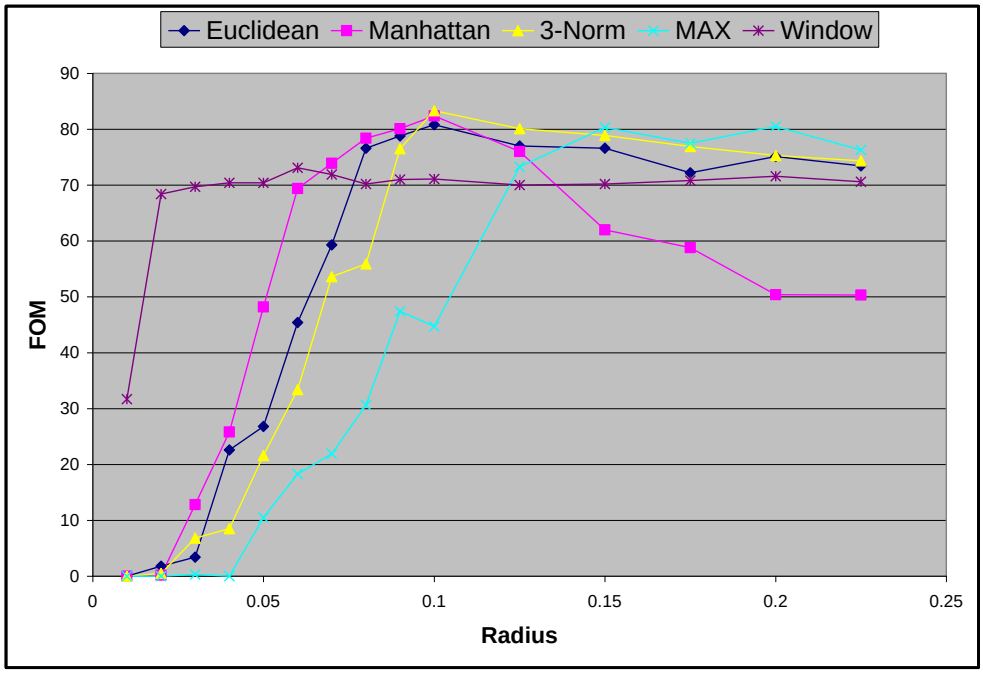


Figure 5.8: Iris Data Radius Optimization Plot for Various Distance Metrics

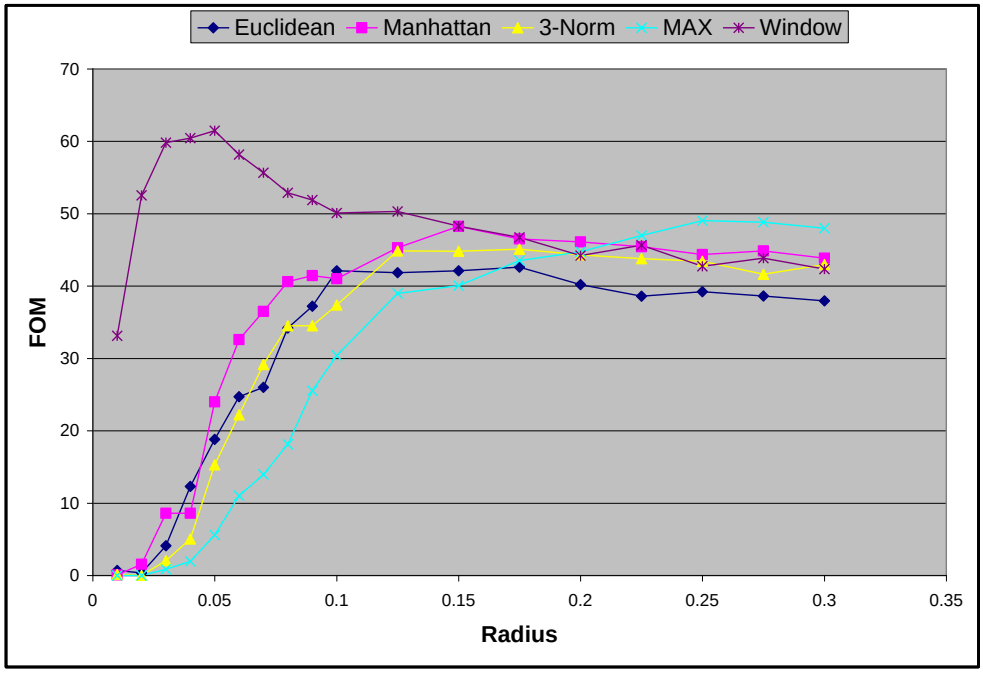


Figure 5.9: Biomedical Data Radius Optimization Plot for Various Distance Metrics

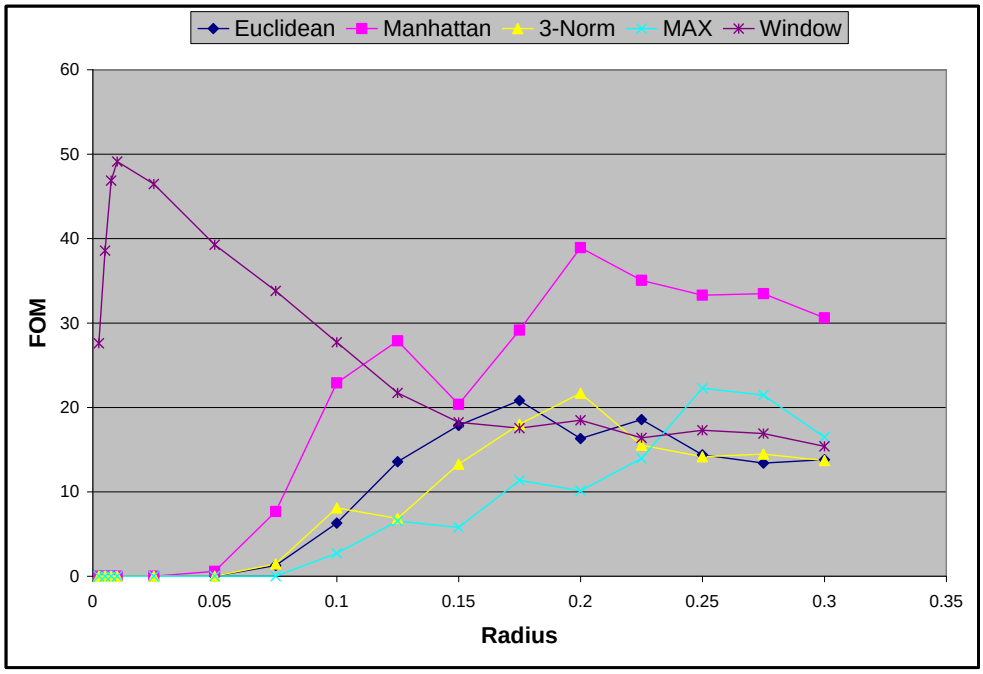


Figure 5.10: BUPA Data Radius Optimization Plot for Various Distance Metrics

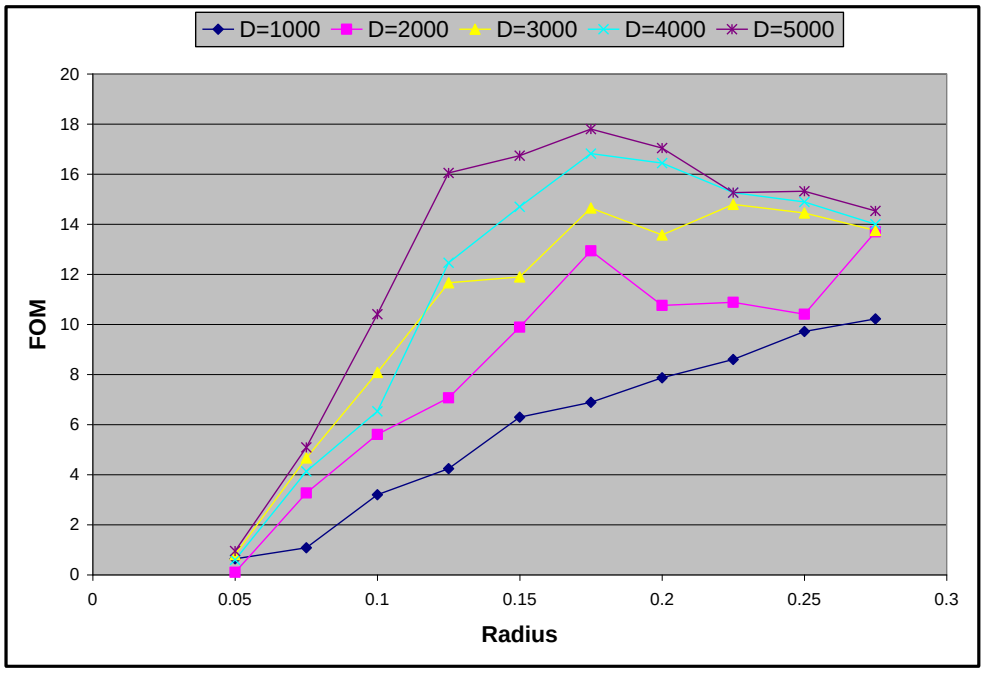


Figure 5.11: Detector Count Optimization Plot for Euclidean Distance Metric

For the real-valued negative selection algorithm using a fixed-sized radius, making a slight modification assists in the detector placement. Previously, a detector is only stored if the minimum calculated distance to the nearest self point or nearest detector center is greater than the detector threshold radius *r*. As the number of detectors stored increases, it is difficult to allow space for more incoming detectors to find placement. A modification to the placement criterion allows detector overlap and results in multiple benefits. The detector is still required to remain a fixed distance *r* from the nearest self point, but is now allowed to be within *0.25r* to the nearest detector. This amount of detector overlap allows the possibility of a greater number of detectors to be placed, and also increases the amount of non-self subspace coverage. By allowing overlap, the 'holes' produced by detectors spaced a distance *r* away from each other are now filled, since the radius of each detector still remains fixed at *r*.

Formal presentation of individual radius assignments and analysis of the final results for the fixed-sized detector algorithm are covered in the next section of this chapter. The next topic of discussion is the *V-detector* algorithm, a new sophisticated and intelligent approach to the negative selection algorithm.

Two different implementations of the real-valued negative selection algorithm with variable detectors (*V-detector*) are tested. The first implementation is exactly the same as the proposed algorithm in Chapter 3. The two control parameters, estimated coverage c_o and maximum number of detectors D_{max}, are predetermined for each data set as: 1) *Iris* $\rightarrow c_o = 99.9\%, D_{max} = 250$, 2) *Biomedical* $\rightarrow c_o = 99.99\%, D_{max} = 250$, and 3) *BUPA* $\rightarrow c_o = 99.98\%, D_{max} = 1000$. The self radius, r_s, is still the same as the detector

radius implemented in the fixed sized detector algorithm. This first implementation did not yield satisfactory results, and required several modifications to achieve optimal results.

The optimized second implementation of the *V-detector* algorithm produces superior results over the original method. The estimated coverage c_o and maximum number of detectors D_{max} are not changed, but the self radius threshold is modified. Similarly as before, several tests are performed to determine the optimal value of the self radius. For both the *Iris* and *Biomedical* datasets, a unique method is employed which sets the self radius as the average standard deviation of the training data samples. This allows the self radius threshold to vary proportionally to the distribution of the self data. The *BUPA* dataset did not allow this methodology, because the distribution of the data across six dimensions varies so much that the standard deviation was too large to adequately represent the self radius. For the *BUPA* dataset, individual self radius optimization tests are required for each distance metric to produce optimal results. Similar to the fixed sized radius algorithm, detector overlap is also implemented in the modified *V-detector* algorithm. This allows the possibility of the placement of a greater number of detectors before the estimated coverage is reached, and simultaneously removes 'holes' and improves non-self space coverage.

Additional modifications are devised for the second implementation of the *V-detector* algorithm. In Chapter 3, two methods are discussed regarding the assignment to the variable radius r_d. It is specified that this study implements the aggressive approach, where r_d is set equal to the minimum distance to the nearest self point. This value is actually modified to allow a small amount of variability in the self data. Instead of

assigning r_d = *dist_min*, the modified variable radius is r_d = *(dist_min*(1-*r_s*))*. By performing this modification, if r_s*=0.01*, then r_d= *dist_min*.99*, or 99% of *dist_min*. While this may seem counter intuitive to achieving better non-self coverage, it actually decreases false alarm rates greatly while minimally lowering detection rates, therefore improving FOM scores.

The final real-valued negative selection algorithm implementation is the proliferating *V-detector*. Like the *V-detector* algorithm, there are tests for two separate implementations of this algorithm. The first implementation utilizes the same radius assignment from the fixed-sized algorithm for the self radius r_s. The proliferation consists of three stages, where the additional threshold $\theta = r_s$ for the initial generation stage. For each subsequent proliferation stage, θ takes on the following values: 1st stage = *(0.5** r_s*)*, 2nd stage = *(0.25** r_s*)*, and 3rd stage = *(*θ*=0)*. The estimated coverage and maximum number of detectors are the same for each dataset, c_o = *99.98%* and D_{max} = *250*. Due to the poor choice of r_s and three stages of proliferation, this algorithm produces poor results with the longest runtime (72+ hrs).

The modified proliferating *V-detector* algorithm makes several improvements over the initial implementation. First, the self radius is optimized for each particular dataset, as performed for the various algorithms previously. The standard deviation did not provide adequate results for this algorithm, so optimized values were chosen by the iterative testing process of comparing FOM scores for each radius assignment. The maximum number of detectors is raised to D_{max} = *500*, and estimated coverage is increased to c_o = *99.99%*. Because the proliferating *V-detector* implementation produces

overlapping offspring detectors which fills 'holes' adequately by design, no additional detector overlap was needed.

The modified proliferating *V-detector* algorithm only implements two stages of detector proliferation. Experimental tests proved that three-stage proliferation increased the total number of detectors generated with little to no change in the overall figure of merit score. The only factor which increased dramatically was the amount of time each algorithm required to run a single trial. The final modification is similar to the variable radius assignment implemented in the modified *V-detector* algorithm. For the modified proliferating *V-detector* algorithm, the final stage of proliferation does not assign the threshold $\theta=0$, but rather allows small percentage of the threshold to remain. In the final stage of proliferation, the variable radius is $r_d = (dist\_min - (0.1 * \theta))$. Again, this is performed to decrease false alarm rates while minimally affecting detection rates, producing improved figure of merit scores.

The final algorithm in this discussion is the multilayer feedforward neural network model. The neural network model consists of one hidden layer with fifteen hidden neurons. The control parameters were preset identically for each dataset, with the learning rate $\eta = 0.2$, $a = 1$ and all bias values $b_i = 1$. To achieve optimal results, the stopping criteria threshold MSE_{th} was decreased for each experimental test until the algorithm was no longer capable of converging. The minimal values of MSE_{th} yielding optimal results are *0.01* for the *Iris Dataset*, *0 .07* for *Bio*, and *0.08* for *BUPA*.

A major distinction between the neural network and negative selection algorithm concerns the choice of training data. For a negative selection algorithm, the input to the system consists of only self data, either 100% or 50%. The neural network model, by

design, cannot be trained with only self data. If the training data all share the same desired value, for example *self = 1*, then the dynamics of the back propagation algorithm fail to train the algorithm properly to identify any new incoming data instance as anything besides *1*. Future testing procedures in the next section will prove this hypothesis. Because the neural network cannot be trained with only self data, a new methodology is required to implement a fair training comparison.

Similarly to the negative selection algorithm training with 100% and 50% of the self data, for the neural network the datasets are split into two training sets, 50% and 25%. The 50% training set consists of 50% self data and 50% non-self data. Likewise, the 25% training set consists of 25% self data and 25% non-self data. Table 5.1 shows the training data distribution. Note for the *Iris Dataset* there are three classes of data, and therefore three versions of each training dataset were formulated, in which the flower of interest is designated as self. For the Iris non-self column in Table 5.1, the addition equation represents the number of datasets from each flower designated as non-self.

Training Set	Self	Non-self	Total Data Sets
50% Iris	25	25 + 25 = 50	75
25% Iris	13	13 + 13 = 26	39
50% Bio	64 Normal	33 Carrier	97
25% Bio	32 Normal	17 Carrier	49
50% BUPA	72 Clean	100 Disorder	172
25% BUPA	36 Clean	50 Disorder	86

Table 5.1: Training Data Distribution for Neural Network Implementation

5.3 Experimental Testing and Results

The implementation of each algorithm depends on a certain degree of randomness, from the detector generation placement of the negative selection algorithm to the initial weight assignments of the neural network model. Due to highly random nature of each algorithm, 50 trials are conducted for each experimental test performed. Consider for each negative selection algorithm implemented using a different distance metric, 14 distinct datasets are tested. Five total negative selection algorithm versions are tested; the fixed sized detector, two versions of the *V-detector*, and two versions of the proliferating *V-detector* algorithm. Each version is tested for five different distance metrics. The neural network model required ten different dataset configurations, which combined with the 350 unique negative selection tests; means a total of 360 experimental tests are performed. Because each test was averaged over 50 trials, the total number of experimental trials conducted is 18,000. This does not include the several hundreds of tests performed to achieve optimal results before each final test is implemented.

The experimental testing for each real-valued negative selection algorithm yields four important performance metrics. The detection rate (DR) yields a percentage of correctly identified non-self points, while the false alarm rate (FA) produces a percentage of self points classified incorrectly. The figure of merit (FOM) is a method of comparing how well the algorithm detects anomalies while simultaneously penalizing it for self misclassifications, and is a byproduct of detection rate and false alarm rate, calculated as (DR-FA). The fourth performance metric is the average total number of detectors implemented for each test. While not an actual measure of the algorithm's efficiency, it

is discussed later as an additional method of comparison to determine the best candidate when implementing a negative selection algorithm.

The first real-valued negative selection algorithm tested was for the case of fixed sized detectors. The results for each distance metric for the Iris dataset are provided in Tables 5.2-5.6. This is only a sample of the results tabulated to illustrate content and formatting for each experimental trial. There are over 75 tables of results produced for this study, and the inclusion of an appendix of results is neglected to reduce the number of pages for this report. Appendix A provides a brief comprehension of the intermediate results for detection rate and false alarm rate. A complete catalogue of data tables and specific Matlab code implementations is in the accompanying CD-ROM included with this report.

FINAL RESULTS	Detection Rate (%)		False Alarm (%)		F.O.M.	Detector Count	
Datasets	Mean	Std. Dev.	Mean	Std. Dev.	(DR%-FA%)	Mean	Std. Dev.
Setosa 100%	100	0	0	0	100.00	1000	0
Setosa 50%	100	0	10.18	1.623	89.82	1000	0
Versicolor 100%	91.36	4.758	0	0	91.36	1000	0
Versicolor 50%	95.02	3.491	12.64	4.052	82.38	1000	0
Virginica 100%	95.34	5.113	0	0	95.34	1000	0
Virginica 50%	97.16	1.687	16.04	5.085	81.12	1000	0

Table 5.2: Final Results for Fixed Sized Radius using Manhattan Distance Metric

FINAL RESULTS	Detection Rate (%)		False Alarm (%)		F.O.M.	Detector Count	
Datasets	Mean	Std. Dev.	Mean	Std. Dev.	(DR%-FA%)	Mean	Std. Dev.
Setosa 100%	100	0	0	0	100.00	1000	0
Setosa 50%	100	0	7.56	3.199	92.44	1000	0
Versicolor 100%	83.7	10.07	0	0	83.70	1000	0
Versicolor 50%	89.04	7.473	8.32	3.33	80.72	1000	0
Virginica 100%	93.38	8.166	0	0	93.38	1000	0
Virginica 50%	92.3	10.809	13.12	4.734	79.18	1000	0

Table 5.3: Final Results for Fixed Sized Radius using Euclidean Distance Metric

FINAL RESULTS	Detection Rate (%)		False Alarm (%)		F.O.M.	Detector Count	
Datasets	Mean	Std. Dev.	Mean	Std. Dev.	(DR%-FA%)	Mean	Std. Dev.
Setosa 100%	99.88	0.5206	0	0	99.88	1000	0
Setosa 50%	99.92	0.338	6.96	3	92.96	1000	0
Versicolor 100%	79.78	11.07	0	0	79.78	1000	0
Versicolor 50%	86.18	9.652	8.84	3.504	77.34	1000	0
Virginica 100%	87.44	11.362	0	0	87.44	1000	0
Virginica 50%	92.42	8.379	11.44	5.267	80.98	1000	0

Table 5.4: Final Results for Fixed Sized Radius using 3-Norm Distance Metric

FINAL RESULTS	Detection Rate (%)		False Alarm (%)		F.O.M.	Detector Count	
Datasets	Mean	Std. Dev.	Mean	Std. Dev.	(DR%-FA%)	Mean	Std. Dev.
Setosa 100%	100	0	0	0	100	1000	0
Setosa 50%	100	0	12.4	4.37	87.6	1000	0
Versicolor 100%	93.76	3.1	0	0	93.76	1000	0
Versicolor 50%	97.54	1.89	16.08	3.49	81.46	1000	0
Virginica 100%	96.4	2.55	0	0	96.4	1000	0
Virginica 50%	97.54	1.71	21.8	4.77	75.74	1000	0

Table 5.5: Final Results for Fixed Sized Radius using ∞-Norm Distance Metric

FINAL RESULTS	Detection Rate (%)		False Alarm (%)		F.O.M.	Detector Count	
Datasets	Mean	Std. Dev.	Mean	Std. Dev.	(DR%-FA%)	Mean	Std. Dev.
Setosa 100%	100	0	0	0	100	1000	0
Setosa 50%	100	0	13.52	1.42	86.48	1000	0
Versicolor 100%	92.62	1.028	0	0	92.62	1000	0
Versicolor 50%	97.86	0.869	16.08	1.744	81.78	1000	0
Virginica 100%	98.98	0.141	0	0	98.98	1000	0
Virginica 50%	99	0	24.6	2.16	74.4	1000	0

Table 5.6: Final Results for Fixed Sized Radius using Partial Euclidean Distance Metric

The FOM performance metric is tabulated in the previous result tables for each designated training dataset. The computation of the average FOM score for each algorithm implementation uses two separate methods. The total FOM score represents the average FOM of all data training sets, simply computed by averaging all data in the

FOM column. The 50% FOM score is the average FOM score only for the cases when 50% of the self data is utilized for training. This score is indicative of the case when not all 'self' data is available for training, and provides better insight into the efficiency of each algorithm. Table 5.7 is a condensed version of the final results for the negative selection algorithm using fixed sized detectors for each dataset, which only includes the designation of the detector radius and two FOM performance metrics.

Radius	Distance Metric	Total FOM	50% FOM	D #
Iris Dataset				
Constant R=0.1	**Euclidean**	**88.24**	**84.11**	***1000***
Constant R=0.1	**Manhattan**	**90.00**	**84.44**	***1000***
Constant R=0.06	**Partial Euclidean (Window)**	**89.04**	**80.89**	***1000***
Constant R=0.1	**3-Norm**	**86.40**	**83.76**	***1000***
Constant R=0.2	**Infinity Norm (MAX)**	**89.16**	**81.60**	***1000***
Biomedical Dataset				
Constant R=0.15	**Euclidean**	**26.56**	**27.93**	***1000***
Constant R=0.15	**Manhattan**	**32.20**	**30.95**	***1000***
Constant R=0.05	**Partial Euclidean (Window)**	**59.01**	**53.32**	***1000***
Constant R=0.15	**3-Norm**	**26.89**	**28.39**	***1000***
Constant R=0.25	**Infinity Norm (MAX)**	**27.65**	**26.64**	***1000***
BUPA Dataset				
Constant R=0.175	**Euclidean**	**23.49**	**21.86**	***5000***
Constant R=0.2	**Manhattan**	**36.43**	**32.76**	***5000***
Constant R=0.01	**Partial Euclidean (Window)**	**74.16**	**50.38**	***5000***
Constant R=0.2	**3-Norm**	**23.22**	**21.42**	***5000***
Constant R=0.25	**Infinity Norm (MAX)**	**24.63**	**21.96**	***5000***

Table 5.7: FOM Final Results for Fixed Sized Radius

The next real-valued negative selection algorithm tested is the variable radius technique. Two versions of the *V-detector* algorithm is tested. The first method employs the same strategies proposed in Chapter 3 for the *V-detector* algorithm, and retains the same value for r_s designated in the previous implementation for the fixed sized radius.

The second method is a modified version of the *V-detector* algorithm, where several aspects of the algorithm are improved to achieve optimal results. The modified *V-detector* algorithm includes the additional benefit of assigning optimal values for r_s based upon several preliminary testing results. Tables 5.9 and 5.10 provides the final results for each implementation.

An important distinction between the fixed radius and *V-detector* algorithms is the assignment of the detector radius and stopping criteria. The *V-detector* implementation does not rely on the generation of a fixed number of detectors, but instead relies heavily on the estimated coverage stopping criteria. Therefore, the number of detectors generated for each implementation of the *V-detector* algorithm is an important performance metric worth mentioning. Table 5.8 is an example of the results tabulated for a single modified *V-detector* algorithm trained with *Biomedical Data*. Notice the average number of detectors generated and standard deviation of detector generation are now included in the data results. The column (*D #)* in Tables 5.8-5.10 represents the total average of detectors generated for every training instance.

FINAL RESULTS	Detection Rate (%)		False Alarm (%)		FOM	Detector Count	
Datasets	Mean	Std. Dev.	Mean	Std. Dev.	(DR%-FA%)	Mean	Std. Dev.
Normal 100%	77.55	2.61	0	0	77.55	362.52	17.57
Carriers 100%	34.65	63.4	0	0	34.65	239.16	11.01
Normal 50%	83.22	3.2	25.8	2.05	57.42	276.76	16.92
Carriers 50%	56.44	6.84	32.84	3.5	23.6	213.16	12.24
					D # =	272.9	

Table 5.8: Final Results for Modified *V-Detector* using Euclidean Distance Metric

Radius	Distance Metric	Total FOM	50% FOM	D #
Iris Dataset				
Rs=0.1	Euclidean	91.28	86.12	*13.07*
Rs=0.1	Manhattan	89.85	84.51	*11.61*
Rs=0.06	Partial Euclidean (Window)	87.89	84.69	*8.65*
Rs=0.1	3-Norm	91.45	86.63	*14.67*
Rs=0.2	Infinity Norm (MAX)	89.83	85.49	*15.74*
Biomedical Dataset				
Rs=0.15	Euclidean	25.07	28.56	*15.32*
Rs=0.15	Manhattan	24.50	26.29	*13.56*
Rs=0.05	Partial Euclidean (Window)	53.56	45.02	*42.3*
Rs=0.15	3-Norm	25.28	28.64	*17.45*
Rs=0.25	Infinity Norm (MAX)	22.19	22.64	*18.02*
BUPA Dataset				
Rs=0.175	Euclidean	12.75	11.04	*30.69*
Rs=0.2	Manhattan	10.27	10.00	*18.88*
Rs=0.01	Partial Euclidean (Window)	48.86	39.34	*116.13*
Rs=0.2	3-Norm	13.04	11.46	*34.86*
Rs=0.25	Infinity Norm (MAX)	16.31	14.67	*53.35*

Table 5.9: FOM Final Results for Original *V-Detector* Implementation

Radius	Distance Metric	Total FOM	50% FOM	D #
Iris Dataset				
Rs=std(T)	Euclidean	89.66	87.97	*15.53*
Rs=std(T)	Manhattan	87.30	85.63	*13.19*
Rs=std(T)	Partial Euclidean (Window)	88.93	85.02	*11.18*
Rs=std(T)	3-Norm	89.05	86.74	*17.18*
Rs=std(T)	Infinity Norm (MAX)	87.94	86.41	*19.72*
Biomedical Dataset				
Rs=std(T)/2	Euclidean	48.30	40.51	*272.9*
Rs=std(T)/2	Manhattan	48.84	40.48	*235.84*
Rs=std(T)/4	Partial Euclidean (Window)	51.12	50.43	*123.74*
Rs=std(T)/2	3-Norm	54.20	49.42	*328.06*
Rs=std(T)/2	Infinity Norm (MAX)	69.08	51.62	*506.45*
BUPA Dataset				
Rs=0.025	Euclidean	63.77	50.75	*831.95*
Rs=0.025	Manhattan	65.32	50.54	*829.14*
Rs=0.001	Partial Euclidean (Window)	72.05	49.53	*503.66*
Rs=0.025	3-Norm	64.48	50.91	*864.13*
Rs=0.05	Infinity Norm (MAX)	66.68	50.47	*927.99*

Table 5.10: FOM Final Results for Modified *V-Detector* Implementation

The results for each implementation of the *V-detector* algorithm clearly illustrates that the modified version outperforms over the original implementation. This comes as no surprise, considering the modified implementation is an improved design over the original version. The interesting aspects of the modified *V-detector* algorithm results begin to surface when compared to the fixed sized radius results. An overall improvement in figure of merit scores is displayed by the modified *V-detector* algorithm approach. Even more astounding, the improvement in FOM scores results from a decrease in the average number of detectors generated. Later in this report, a more concise table presents results from which formal conclusions are derived.

The real-valued negative selection algorithm with proliferating variable detectors is the final version tested. Similarly to the *V-detector* algorithm, there are tests for two separate implementations of the proliferation algorithm. The first method is the original implementation with three stages of proliferation, and the second version is a modified and condensed two stage implementation. Tables 5.11 and 5.12 show the final figure of merit scores for each proliferating algorithm implementation. Again, it is no surprise that the modified version attains better overall efficiency when compared to the original implementation.

The last test implemented in this study was a feedforward neural network model trained with back propagation. It was mentioned previously that the comparison between the negative selection model and neural network is not ideal. The distinction between the two models arises in the choice of training data. A negative selection algorithm requires only self data for training, whereas the neural network requires samples from both self and non-self. Experimental test results provide the proof to this assumption.

Radius	Distance Metric	Total FOM	50% FOM	D #
Iris Dataset				
Rs=0.1	Euclidean	89.34	83.69	*120.51*
Rs=0.1	Manhattan	89.83	83.73	*68.68*
Rs=0.06	Partial Euclidean (Window)	84.90	82.95	*37.82*
Rs=0.1	3-Norm	87.72	82.05	*166.54*
Rs=0.2	Infinity Norm (MAX)	88.69	85.45	*179.5*
Biomedical Dataset				
Rs=0.15	Euclidean	39.33	35.03	*457.75*
Rs=0.15	Manhattan	56.68	47.40	*617.1*
Rs=0.05	Partial Euclidean (Window)	36.85	35.96	*174.84*
Rs=0.15	3-Norm	37.27	33.60	*445.21*
Rs=0.25	Infinity Norm (MAX)	29.53	27.42	*364.95*
BUPA Dataset				
Rs=0.175	Euclidean	29.22	26.22	*623.12*
Rs=0.2	Manhattan	45.66	36.55	*762.61*
Rs=0.01	Partial Euclidean (Window)	44.79	36.94	*366.15*
Rs=0.2	3-Norm	19.57	18.12	*522.93*
Rs=0.25	Infinity Norm (MAX)	11.64	10.27	*428.32*

Table 5.11: FOM Final Results for Original Proliferating Implementation

Radius	Distance Metric	Total FOM	50% FOM	D #
Iris Dataset				
Rs=0.1	Euclidean	89.30	86.96	*287.34*
Rs=0.1	Manhattan	90.86	87.77	*110.18*
Rs=0.05	Partial Euclidean (Window)	86.06	85.19	*41.79*
Rs=0.1	3-Norm	87.82	86.60	*284.74*
Rs=0.1	Infinity Norm (MAX)	85.95	86.06	*271.2*
Biomedical Dataset				
Rs=0.05	Euclidean	59.56	49.32	*607.91*
Rs=0.05	Manhattan	63.08	48.93	*648.01*
Rs=0.02	Partial Euclidean (Window)	57.47	50.47	*373.86*
Rs=0.05	3-Norm	56.65	48.23	*593.31*
Rs=0.075	Infinity Norm (MAX)	51.26	44.95	*537.67*
BUPA Dataset				
Rs=0.05	Euclidean	62.19	47.33	*1264.5*
Rs=0.05	Manhattan	62.95	46.04	*1249.06*
Rs=.0005	Partial Euclidean (Window)	65.82	45.27	*514.12*
Rs=0.05	3-Norm	59.40	46.15	*1442*
Rs=0.1	Infinity Norm (MAX)	54.30	45.30	*1389.7*

Table 5.12: FOM Final Results for Modified Proliferating Implementation

The neural network model is tested for two cases. The first is the case in which the algorithm is trained with the same data as the negative selection algorithm, while in the latter case the network is trained with the modified training data presented in Table 5.1. Table 5.13 shows the results from training with only self data using the Iris dataset, which illustrates how the neural network will fail for this case. Since the network is only trained with self data, the desired output for all training data is always the same (e.g. '1'). Therefore, the network is basically trained to only output a '1', and any new unknown data instance will always be classified as a '1'. This is why the detection rate is constantly zero, because all non-self data is consistently classified as self.

FINAL RESULTS	Detect Rate (%)		False Alarm (%)		F.O.M.
Datasets	Mean	Std. Dev.	Mean	Std. Dev.	(%NS+%S)
Setosa 100%	0	0	0	0	0.00
Versicolor 100%	0	0	0	0	0.00
Virginica 100%	0	0	0	0	0.00
Setosa 50%	0	0	0	0	0.00
Versicolor 50%	0	0	0	0	0.00
Virginica 50%	0	0	0	0	0.00

Table 5.13: Neural Network Failure Results

Table 5.14 displays the final results derived from the experimental testing of the neural network algorithm. The total FOM score represents the average FOM score of all tests performed for a single dataset. The 50% FOM score is the average of only the tests performed using the 25% training data, which correspond to training the negative selection algorithm with only 50% of the self data. The FOM scores utilize the same nomenclature to aid in the comparison analysis despite differences in the training data monikers.

Dataset	Total FOM	50% FOM
Iris Dataset	94.57	93.61
Biomedical Dataset	46.99	46.35
BUPA Dataset	38.51	36.64
Average Results	60.02	58.87

Table 5.14: Final FOM Results for Neural Network Model

The final experimental results for each algorithm implementation are present, but two more tables are necessary before a the procession of a formal analysis. A complete summary of the FOM scores for each implementation are consolidated into two distinct formats. Table 5.15 presents the total FOM scores for each negative selection and neural network algorithm determined individually by dataset. The average total FOM score is calculated for all three datasets, as well as a total average score for each algorithm's performance. Table 5.16 maintains the same format, but provides the results for only the 50% FOM scores.

It is now possible to present a formal evaluation of the experimental results. The overall performance of each algorithm implementation has an assigned score to determine efficiency. The performance of each distance metric is also associated with a particular score for each algorithm implementation.

Total FOM SCORES	IRIS	BIO	BUPA	Avg.
Euclidean	88.24	26.56	23.49	46.10
Manhattan	90.00	32.20	36.43	52.88
Partial Euclidean (Window)	89.04	59.01	74.16	74.07
3-Norm	86.40	26.89	23.22	45.50
Infinity Norm (MAX)	89.16	27.65	24.63	47.15
Constant Radius AVG results	88.57	34.46	36.39	53.14
V-Detector Euclidean	91.28	25.07	12.75	43.03
V-Detector Manhattan	89.85	24.50	10.27	41.54
V-Detector Window	87.89	53.56	48.86	63.44
V-Detector 3-Norm	91.45	25.28	13.04	43.26
V-Detector MAX	89.83	22.19	16.31	42.78
V-Detector AVG results	90.06	30.12	20.25	46.81
Modified V-Detector Euclidean	89.66	48.30	63.77	67.24
Modified V-Detector Manhattan	87.30	48.84	65.32	67.15
Modified V-Detector Window	88.93	51.12	72.05	70.70
Modified V-Detector 3-Norm	89.05	54.20	64.48	69.24
Modified V-Detector MAX	87.94	69.08	66.68	74.57
Modified V-Detector AVG results	88.58	54.31	66.46	69.78
Prolif V-Detector Euclidean	89.34	39.33	29.22	52.63
Prolif V-Detector Manhattan	89.83	56.68	45.66	64.06
Prolif V-Detector Window	84.90	36.85	44.79	55.51
Prolif V-Detector 3-Norm	87.72	37.27	19.57	48.19
Prolif V-Detector MAX	88.69	29.53	11.64	43.29
Prolif V-Detector AVG results	88.10	39.93	30.18	52.73
Modified Prolif V-Detector Euclidean	89.30	59.56	62.19	70.35
Modified Prolif V-Detector Manhattan	90.86	63.08	62.95	72.30
Modified Prolif V-Detector Window	86.06	57.47	65.82	69.78
Modified Prolif V-Detector 3-Norm	87.82	56.65	59.40	67.96
Modified Prolif V-Detector MAX	85.95	51.26	54.30	63.84
Modified Prolif AVG results	88.00	57.60	60.93	68.84
Neural Network	94.57	46.99	38.51	59.89

Table 5.15: Final Total FOM Experimental Results

Total FOM SCORES	IRIS	BIO	BUPA	Avg.
Euclidean	84.11	27.93	21.86	44.63
Manhattan	84.44	30.95	32.76	49.38
Partial Euclidean (Window)	80.89	53.32	50.38	61.53
3-Norm	83.76	28.39	21.42	44.52
Infinity Norm (MAX)	81.60	26.64	21.96	43.40
Constant Radius AVG results	82.96	33.45	29.68	48.69
V-Detector Euclidean	86.12	28.56	11.04	41.91
V-Detector Manhattan	84.51	26.29	10.00	40.27
V-Detector Window	84.69	45.02	39.34	56.35
V-Detector 3-Norm	86.63	28.64	11.46	42.24
V-Detector MAX	85.49	22.64	14.67	40.93
V-Detector AVG results	85.49	30.23	17.30	44.34
Modified V-Detector Euclidean	87.97	40.51	50.75	59.74
Modified V-Detector Manhattan	85.63	40.48	50.54	58.88
Modified V-Detector Window	85.02	50.43	49.53	61.66
Modified V-Detector 3-Norm	86.74	49.42	50.91	62.36
Modified V-Detector MAX	86.41	51.62	50.47	62.83
Modified V-Detector AVG results	86.35	46.49	50.44	61.10
Prolif V-Detector Euclidean	83.69	35.03	26.22	48.31
Prolif V-Detector Manhattan	83.73	47.40	36.55	55.89
Prolif V-Detector Window	82.95	35.96	36.94	51.95
Prolif V-Detector 3-Norm	82.05	33.60	18.12	44.59
Prolif V-Detector MAX	85.45	27.42	10.27	41.05
Prolif AVG results	83.57	35.88	25.62	48.36
Modified Prolif V-Detector Euclidean	86.96	49.32	47.33	61.20
Modified Prolif V-Detector Manhattan	87.77	48.93	46.04	60.91
Modified Prolif V-Detector Window	85.19	50.47	45.27	60.39
Modified Prolif V-Detector 3-Norm	86.60	48.23	46.15	60.33
Modified Prolif V-Detector MAX	86.06	44.95	45.30	58.77
Modified Prolif AVG results	86.52	48.38	46.02	60.30
Neural Network	93.61	46.35	36.64	58.87

Table 5.16: Final 50% FOM Experimental Results

The experimental data shows that the modified *V-detector* algorithm is the best method for self/nonself discrimination. The total and 50% FOM scores from Tables 5.15 and 5.16 support this claim, but careful review of the results show only marginal improvements over the modified proliferating *V-detector* algorithm. The anticipated argument of these results is with more stages of proliferation and more experimental testing, the proliferating *V-detector* algorithm could eventually outperform the standard *V-detector* implementation. Despite this argument, other factors contribute to the success of the *V-detector* algorithm as the preferred method of implementing a negative selection algorithm.

Directing attention to the results provided in Tables 5.10 and 5.12, the FOM scores are accompanied with the average number of detectors generated for each algorithm implementation. This is where the *V-detector* algorithm improves upon the proliferating *V-detector* method. In all cases, the *V-detector* generates far less detectors than the proliferation version, and still manages to yield higher FOM scores. The modified proliferating V-*detector* algorithm only includes two stages, with the intent to reduce the number of detectors and maintain optimal results. Despite all experimental efforts, the efficiency of the *V-detector* algorithm could not be matched by the two stage proliferating implementation.

Time complexity of each algorithm is another important attribute for measuring performance. For the *BUPA* dataset, the modified *V-detector* algorithms required 5-10 hours of run time to complete 50 trials, while the modified proliferating *V-detector* implementation took between 24-48 hours. The extended run time is a direct result of the greater number of detectors generated for each implementation. The proliferation stages

also contribute to this runtime, as each previous detector is given multiple opportunities to produce offspring, and then each offspring is given the same opportunity in each successive proliferation stage. If time constraints are not a concern, the proliferating *V-detector* algorithm with multiple stages of proliferation may prove to be the better choice of implementation. For reference, runtimes are based on using a PC with a 2.4MHz Intel Celeron processor and 512Mb of RAM running Windows XP.

As expected, the negative selection algorithm using fixed-sized detectors was the least efficient model of the three distinct negative selection algorithms tested. The FOM scores and required number of detectors combine to prove this algorithm should not be considered for real-valued negative selection algorithm implementation. The neural network model outperforms the simple fixed sized detector method, but fails to match the efficiency of the *V-detector* and proliferating implementations. To reiterate, the neural network model is not a perfect comparison model since modifications to the training data is required. However, the efforts put forth in this study did provide sufficient comparison conditions, as evident by the neural network's overall performance.

A major focus of this study was the determination of an appropriate distance metric in the application of a specific real-valued negative selection implementation. The initial hypothesis was that the partial Euclidean distance metric would produce the best results. The partial Euclidean distance metric proved to be the most efficient implementation when using the fixed sized detector algorithm, as it greatly exceeded the other distance metrics in both total and 50% FOM scores. The explanation of these results is straightforward; each distance metric implementation required the same fixed number of total detector, but the partial Euclidean distance metric had a smaller non-self

space to cover. Since the partial Euclidean distance metric only calculates distance in two dimensions, the overall self/non-self space is much smaller than in four or six dimensions. For this reason, it is expected to outperform alternative distance metric for every implementation.

The difference between the fixed sized detector and *V-detector* algorithms is that detector count is determined by estimated coverage. This distinction is the reason why the partial Euclidean failed to remain the most efficient implementation. Since the radius is variable and detector count is flexible, each distance metric can adapt to its given self/non-self space, removing the previously stated advantage held by the partial Euclidean metric. The partial Euclidean distance metric may not be the most efficient, but it does produce comparable results while producing far less detectors for both the *V-detector* and proliferation algorithms. Despite producing fewer detectors, the partial Euclidean algorithm maintained the disadvantage of having the longest runtime. This arose from the fact that each distance calculation required several calculations in a lower dimensional space. For a single distance calculation, 3-5 distances were calculated for a single self point to a single detector. Multiplied over many self points and detectors, and compounded with detector to detector distance calculations, resulted in the partial Euclidean calculation time complexity to increase dramatically (3-5 times longer) over the single distance calculation requirement of the other distance metrics.

For the modified *V-detector* algorithm, the prevailing distance metric with the highest overall total and 50% FOM scores was the infinite-norm (or MAX) distance metric. Following closely behind, the 3-norm distance metric had the second highest 50% FOM score, while the partial Euclidean had the second highest total FOM score.

The 3-norm is third place in total FOM scores. The conclusion from experimental testing is that the infinite-norm should be considered as the optimal choice when implementing a real-valued negative selection *V-detector* algorithm. The ease of distance calculation made this the fastest implementation, and combined with the best overall FOM score, makes this the perfect choice for future *V-detector* implementations.

The proliferating *V-detector* algorithm results did not clearly indicate a preferred distance implementation. The 50% FOM scores designated the standard Euclidean distance metric as the most efficient, but only minimally over the Manhattan distance metric. Conversely, the Manhattan distance metric outperformed the Euclidean for the total FOM results. With the exception of the infinite-norm, all FOM scores were very close for the proliferating *V-detector* algorithm. The only formal conclusions which can be derived from these results is that either the Euclidean or Manhattan distance metric should be implemented for the proliferation algorithm, and the infinite-norm should be avoided.

It is interesting to note that while the infinite-norm is the preferred choice for the *V-detector* algorithm, it is the least acceptable choice for the proliferating implementation. For the *V-detector* algorithm, detector generation is the only mechanism for non-self space coverage and the infinite-norm distance metric produces adequate non-self coverage. The proliferating *V-detector* algorithm's strength in non-self coverage derives from its proliferation stages, not detector generation. The infinite-norm fails to perform adequately when proliferation stages occur. The proliferation of detector offspring using the infinite-norm is not as productive as other distance implementations. This may be a result of the offspring generation scheme.

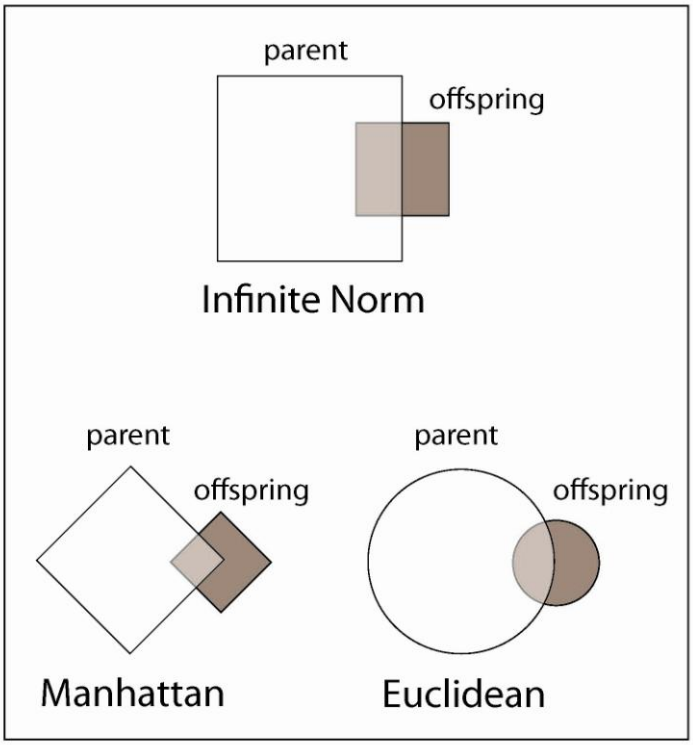


Figure 5.12: Offspring Detector Coverage

Detectors only generate offspring parallel and anti-parallel to the detector centers. Recall the shape of the detector in Figure 3.1. The circular and diamond shape (in two dimensions) of the Euclidean and Manhattan distance seem to have an advantage over the more square-shaped infinite and 3-norm distance metrics. Recall, the 3-norm is actually the second worst implementation for the proliferation algorithm. While these shapes provide benefits in only detector generation stages, they apparently become a hindrance during proliferation stages. Figure 5.12 illustrates offspring detector coverage for different distance metrics. The amount of area not already covered by the parent detector is greatest for the Euclidean and Manhattan offspring detectors.

CHAPTER 6

Conclusions

A formal evaluation of implementing different distance metrics for various real-valued negative selection algorithms is the purpose of this research. This research focuses on three existing variations of the real-valued negative selection algorithm, and evaluates each implementation using five different distance metrics. Distance metrics have been proven to affect the quality of a negative selection algorithm's performance, yet no formal study to date has incorporated real world data and various implementations to determine which distance metric provides maximum effectiveness based on a figure of merit.

Experimental findings suggest the *V-detector* algorithm utilizing the infinite-norm distance metric is the best performing implementation. It not only results in shorter execution runtimes, but also produces superior FOM results. If runtimes are not a concern, the proliferating *V-detector* algorithm using either Euclidean or Manhattan distance metrics is also a good alternative option. The negative selection algorithm using fixed-sized detectors should be avoided, and if implemented; the partial Euclidean distance metric is the definitive choice for optimal performance.

A multilayer feedforward neural network algorithm implementation is a basis of comparison to alternative computational intelligence models. The major discrepancy between negative selection algorithms and alternative approaches is the method of

training. The negative selection algorithm has the applicable advantage of data discrimination when only large amounts of 'self' (normal) samples are available. Most alternative learning algorithms require training of both normal and abnormal data to adequately discriminate between the two.

This study leads to many future research opportunities. More sophisticated negative selection algorithms are being proposed currently, leading to new prospects in evaluating distance metric performance. One new method employs both negative and positive selection mechanisms to improve the correct classification of data by lowering false alarm rates [20]. The most recent advancement is danger theory, which incorporates fuzzy rules to further disseminate the classification of self/non-self [1]. Expanding the research to include more datasets is another possibility, extending into higher dimensional data or more applicable scenarios where most of the data is normal. A final proposition is testing more distance measures. The concept of partial Euclidean distance can be expanded to partial Manhattan or partial 3-norm, or the window size can be extended to include more than two dimensions. This study represents the beginning of a whole new area of negative selection research.

REFERENCES

[1] Aickelin, U., Dasgupta, D., “Artificial Immune Systems Tutorial”, In: Introductory Tutorials in Optimization, Decision Support and Search Methodology, E. Burke and G. Kendall Ed. New York: Springer, 2005, pp375-399

[2] Blake, E. K., Merz, C., “UCI Repository of Machine Learning Databases”, Irvine, CA. University of California, Department of Information and Computer Science, 1998. (Last accessed Jun 7, 2009)

[3] Das, S., Gui, M., Pahwa, A., “Artificial Immune Systems for Self-Nonself Discrimination: Application to Anomaly Detection”, Studies in Computational Intelligence (SCI) 116, 2008. pp229–246

[4] Dasgupta, D., “Advances in Artificial Immune Systems”, IEEE Computational Intelligence Magazine. November, 2006.

[5] Dasgupta, D., Krishna Kumar, K., Wong, D., Berry, M., “Negative Selection Algorithm for Aircraft Fault Detection”, 3rd International Conference on Artificial Immune Systems. Catania, Italy. September, 2004.

[6] De Castro, L., " An Introduction to the Artificial Immune Systems ", presented at International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA), Prague, CZ, 2001.

[7] De Castro, L., Von Zuben, F., "Artificial Immune Systems: Part I – Basic Theory and Applications", Tech. Rep. TR-DCA, January, 1999.

[8] De Castro, L., Von Zuben, F., "Artificial Immune Systems: Part II – A Survey of Applications", Tech. Rep. TR-DCA, February, 2000.

[9] Forrest, S., Perelson, A., Allen, L., R., “Self-nonself discrimination in a computer”, In Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy, IEEE Computer Society Press, Los Alamitos, CA. 1994. pp 202–212

[10] Gonzalez, F., Dasgupta, D., “Anomaly Detection Using Real-Valued Negative Selection”, Journal of Genetic Programming and Evolvable Machines. Volume 4, Issue 4. December, 2003. pp383-403.

[11] Gonzalez, F., Dasgupta, D., Kozma, R., “Combining Negative Selection and Classification Techniques for Anomaly Detection”, Volume 1, Congress on Evolutionary Computation. Honolulu, Hawaii. May, 2002. pp705-710.

[12] Gonzalez, F., Dasgupta, D., Nino, L. F., "A Randomized Real-Valued Negative Selection Algorithm", Second International Conference on Artificial Immune Systems. United Kingdom. September, 2003.

[13] Haykin, S., "Neural Networks and Learning Machines", 3rd ed. New Jersey: Pearson Education, 2009.

[14] Janeway, C., Travers, P., Walport, M., "Immunobiology", Fifth Ed. New York and London: Garland Science, 2001.

[15] Ji, Z., "Negative Selection Algorithms: from the Thymus to V-detector", Ph. D. dissertation, University of Memphis, Memphis, TN, USA, August 2006.

[16] Ji, Z., Dasgupta, D., "Applicability Issues of the Real-Valued Negative Selection Algorithms", Genetic and Evolutionary Computation Conference (GECCO). Seattle, Washington. July, 2006.

[17] Ji, Z., Dasgupta, D., "Revisiting Negative Selection Algorithms", Issue 15.2. Evolutionary Computation Journal. July, 2007.

[18] Ji, Z., Dasgupta, D., "Real-Valued Negative Selection Using Variable-Sized Detectors", Genetic and Evolutionary Computation Conference (GECCO). Seattle, Washington. June, 2004.

[19] Ji, Z., Dasgupta, D., " V-Detector: A Negative Selection Algorithm", presented at Computer Science Research Day, University of Memphis, TN, USA, 2005.

[20] Middlemiss, M., "Positive and Negative Selection in a Multilayer Artificial Immune System", Discussion Paper 2006/03. Department of Information Science, University of Otago, Dunedin, New Zealand, 2006.

[21] Sabo, D., "A Modified Iterative Pruning Algorithm for Neural Network Dimension Analysis", M.S. thesis, California Polytechnic State University, San Luis Obispo, CA, USA, 2007.

[22] "Statlib datasets archive", World Wide Web, http://lib.stat.cmu/datasets. (Last accessed May 12, 2009)

[23] Yang, X.R., Shen, J, Wang R., "Artificial Immune Theory Based Network Intrusion Detection System and the Algorithms Design", First International Conference on Machine Learning and Cybernetics, Beijing, China, November 4-5, 2002

APPENDIX A

Additional Data Tables

IRIS DATASET	Total Averages		50% Averages	
Algorithm Implementation	DR (%)	FA (%)	DR(%)	FA (%)
Euclidean	93.07	4.83	93.78	9.67
Manhattan	96.48	6.48	97.39	12.95
Partial Euclidean (Window)	98.07	9.03	98.95	18.07
3-Norm	97.54	8.38	98.36	16.76
Infinity Norm (MAX)	90.94	4.54	92.84	9.08
Constant Radius AVG results	95.22	6.65	96.26	13.31
V-Detector Euclidean	97.18	5.90	97.92	11.80
V-Detector Manhattan	95.80	5.96	96.43	11.91
V-Detector Window	93.86	4.93	94.87	9.85
V-Detector 3-Norm	97.37	5.91	98.45	11.83
V-Detector MAX	95.85	6.02	97.54	12.04
V-Detector AVG results	96.01	5.74	97.04	11.49
Modified V-Detector Euclidean	90.52	3.23	92.08	6.45
Modified V-Detector Manhattan	90.52	3.23	92.08	6.45
Modified V-Detector Window	92.25	4.36	93.41	8.72
Modified V-Detector 3-Norm	92.55	3.50	93.74	7.00
Modified V-Detector MAX	92.14	4.20	94.81	8.40
Modified V-Detector AVG results	91.60	3.70	93.22	7.40
Prolif V-Detector Euclidean	95.92	6.58	96.85	13.16
Prolif V-Detector Manhattan	96.42	6.59	96.91	13.19
Prolif V-Detector Window	88.47	3.57	90.09	7.15
Prolif V-Detector 3-Norm	95.06	5.85	96.72	11.71
Prolif V-Detector MAX	93.00	4.31	94.07	8.63
Prolif AVG results	93.77	5.38	94.93	10.77
Modified Prolif V-Detector Euclidean	93.28	3.98	94.92	7.96
Modified Prolif V-Detector Manhattan	93.96	3.10	93.97	6.20
Modified Prolif V-Detector Window	83.04	1.73	84.37	3.46
Modified Prolif V-Detector 3-Norm	91.56	3.74	94.08	7.48
Modified Prolif V-Detector MAX	88.60	2.65	91.37	5.31
Modified Prolif AVG results	90.09	3.04	91.74	6.08

Table A.1: Iris Averages for Detection & False Alarm Rates

BIOMEDICAL DATASET	Total Averages		50% Averages	
Algorithm Implementation	DR (%)	FA (%)	DR(%)	FA (%)
Euclidean	32.30	5.74	39.41	11.48
Manhattan	39.63	7.42	45.80	14.85
Partial Euclidean (Window)	74.54	15.53	84.38	31.06
3-Norm	32.88	6.04	40.38	12.07
Infinity Norm (MAX)	34.51	6.86	40.35	13.71
Constant Radius AVG results	42.77	8.32	50.06	16.63
V-Detector Euclidean	30.06	5.00	38.56	10.00
V-Detector Manhattan	28.96	4.46	35.21	8.92
V-Detector Window	66.71	13.15	71.33	26.31
V-Detector 3-Norm	30.68	5.40	39.43	10.80
V-Detector MAX	28.86	6.70	36.03	13.40
V-Detector AVG results	37.05	6.94	44.11	13.89
Modified V-Detector Euclidean	62.97	14.66	69.83	29.32
Modified V-Detector Manhattan	64.22	15.38	71.24	30.76
Modified V-Detector Window	66.76	15.63	81.70	31.27
Modified V-Detector 3-Norm	71.95	17.74	84.91	35.48
Modified V-Detector MAX	91.10	22.01	95.65	44.02
Modified V-Detector AVG results	71.40	17.08	80.67	34.17
Prolif V-Detector Euclidean	50.55	11.22	57.48	22.44
Prolif V-Detector Manhattan	77.00	17.32	82.04	34.64
Prolif V-Detector Window	46.28	9.43	54.82	18.86
Prolif V-Detector 3-Norm	47.17	9.91	53.42	19.81
Prolif V-Detector MAX	37.23	7.71	42.83	15.41
Prolif AVG results	51.65	11.12	58.12	22.23
Modified Prolif V-Detector Euclidean	75.80	16.24	81.80	32.48
Modified Prolif V-Detector Manhattan	82.52	19.44	87.80	38.88
Modified Prolif V-Detector Window	73.46	16.00	82.45	31.98
Modified Prolif V-Detector 3-Norm	72.03	15.39	79.01	30.78
Modified Prolif V-Detector MAX	65.43	14.17	73.28	28.33
Modified Prolif AVG results	73.85	16.25	80.87	32.49

Table A.2: Biomedical Averages for Detection & False Alarm Rates

BUPA DATASET	Total Averages		50% Averages	
Algorithm Implementation	DR (%)	FA (%)	DR(%)	FA (%)
Euclidean	31.12	7.63	37.12	15.26
Manhattan	47.07	10.65	54.05	21.29
Partial Euclidean (Window)	98.77	24.61	99.59	49.21
3-Norm	31.13	7.91	37.24	15.83
Infinity Norm (MAX)	33.34	8.72	39.39	17.43
Constant Radius AVG results	48.29	11.90	53.48	23.80
V-Detector Euclidean	17.00	4.24	19.53	8.48
V-Detector Manhattan	13.50	3.22	16.43	6.44
V-Detector Window	64.90	16.04	71.43	32.09
V-Detector 3-Norm	17.55	4.51	20.49	9.03
V-Detector MAX	21.27	4.96	24.61	9.93
V-Detector AVG results	26.84	6.59	30.50	13.19
Modified V-Detector Euclidean	82.46	18.70	88.14	37.39
Modified V-Detector Manhattan	85.20	19.89	90.03	39.77
Modified V-Detector Window	95.23	23.17	95.87	46.35
Modified V-Detector 3-Norm	83.61	19.12	89.15	38.25
Modified V-Detector MAX	88.28	21.60	93.66	43.20
Modified V-Detector AVG results	86.96	20.50	91.37	40.99
Prolif V-Detector Euclidean	37.93	8.71	43.63	17.41
Prolif V-Detector Manhattan	60.67	15.00	66.57	30.02
Prolif V-Detector Window	59.62	14.83	66.60	29.67
Prolif V-Detector 3-Norm	24.23	4.66	27.44	9.32
Prolif V-Detector MAX	14.75	3.11	16.49	6.22
Prolif AVG results	39.44	9.26	44.15	18.53
Modified Prolif V-Detector Euclidean	81.22	19.03	85.39	38.06
Modified Prolif V-Detector Manhattan	82.30	19.36	84.75	38.71
Modified Prolif V-Detector Window	86.06	20.24	85.75	40.48
Modified Prolif V-Detector 3-Norm	78.35	18.95	84.05	37.90
Modified Prolif V-Detector MAX	71.55	17.25	79.80	34.50
Modified Prolif AVG results	79.90	18.97	83.95	37.93

Table A.3: BUPA Averages for Detection & False Alarm Rates

Appendix B
Samples of Matlab Source Code

```
%Constant-sized Detector Algorithm for Negative Selection Artificial Immune System using Euclidean Distance
Metric
%Currently training with half of self data (25 data sets)
%Data Structure for Full_Data (1-50=Setosa,51-100=Versicolor,101-150=Virginica)
%You only need to change T=(flower name) to test for each flower,
T=Virginica_Half;                                   %Initialize Training Set (Self Set)
for i=1:25
    T_norm(i,1:4)=T(i,1:4)/norm(T(i,1:4));          %Normalize Training Set
end
T=T_norm;                                           %Set T equal to normalized values
self=zeros(1,150);                                  %Initialize self=0, used to count successful Self detections
D=rand(1000,4);                                     %Generates 1000 random detectors from (0,1)
X=Full_Data;                                        %Initialize Full data set for testing (includes training set)
for i=1:150
    X_norm(i,1:4)=X(i,1:4)/norm(X(i,1:4));          %Normalize Full Data Set
end
X=X_norm;                                           %Set X equal to normalized Full Data Set
r=.1;                                               %Detector radius (Optimal value determined as 0.1)
for i=1:1000
    d_min=inf;                                      %Initialize minimum distance to infinite
    for j=1:25
        dist=((sum((D(i,1:4)-T(j,1:4)).^2))/4).^.5; %finds distance from detector 'i' to each self
                                                      %set(training set), Euclidean distance used
        if dist<=d_min          %used to determine minimum distance from detector 'i' to any self set
            d_min=dist;         %Compares each distance calculated to minimum distance, saves minimum value
            C(1,1:4)=T(j,1:4);  %determines nearest self set, saves as 'C'
        end
    end
    age=0;                                  %Initialize age to zero
    while(d_min<r)                          %While minimum distance<radius, move detector or generate new one
        if age>=15              %Max age set to 15, (number of detectors moves before generate new detector)
            D(i,1:4)=rand(size(D(1,1:4)));    %if max age is reached, generate new detector
            age=0;                            %Reset age to zero after new detector generation
```

```
        for j=1:25
            dist=((sum((D(i,1:4)-T(j,1:4)).^2))/4).^.5;    %Establish new minimum distance from NEW
                                                           %detector 'i' to each self set(training set)
            if dist<=d_min    %used to determine minimum distance from NEW detector 'i' to any self set
                d_min=dist;                                                  %same as before
                C(1,1:4)=T(j,1:4);                    %same as before
            end
        end
    else
        dir=(sum(D(i,1:4)-C(1,1:4)))/(abs(sum(D(i,1:4)-C(1,1:4))));  %Determines direction to move
                                                           %detector from nearest self set(either 1 or -1)
        age=age+1;                                         %Increment age by one
        n=.005*exp(-age/15);     %Calculates amount to move detector (n=n*exp(-age/t) (n=.005, t=15)
        D(i,1:4)=D(i,1:4)+(n*dir);           %move detector  (d=d+n*dir)
        d_min=inf;                           %Initialize minimum distance to infinite
        for j=1:25
            dist=((sum((D(i,1:4)-T(j,1:4)).^2))/4).^.5;  %Recalculate distance from detector 'i' to
                                                              %each self set
            if dist<=d_min     %used to determine minimum distance from detector 'i' to any self set
                d_min=dist;                       %same as before
                C(1,1:4)=T(j,1:4);                %same as before
            end
        end
    end
end
if i>1                    %test detector (excluding 1st) against previous detectors
    d_min=inf;            %Initialize minimum distance to infinite
    age=0;                %Initialize age to zero
    for j=1:i-1
        dist=((sum((D(i,1:4)-D(j,1:4)).^2))/4).^.5;  %finds distance from detector i to each previous
                                                           % detector
        if dist<=d_min     %used to determine minimum distance from detector i to previous detector set
            d_min=dist;
            C1(1,1:4)=D(j,1:4);          %Stores nearest Detector Set to 'C'
        end
    end
    while(d_min<(.25*r))  %check if detector 'i' is too close to nearest detector (allows 75% overlap)
        dir=(sum(D(i,1:4)-C1(1,1:4)))/(abs(sum(D(i,1:4)-C1(1,1:4))));   %determine direction to move
```

```
                                                          %detector from nearest Detector (either 1 or -1)
            n=.005*exp(-age/15);   %Calculates amount to move detector (n=n*exp(-age/t) (n=.005, t=15)
            D(i,1:4)=D(i,1:4)+(n*dir);              %move detector  (d=d+n*dir)
            age=age+1;                              %Increment age by one
            d_min=inf;                              %Initialize minimum distance to infinite
            for j=1:i-1
            dist=((sum((D(i,1:4)-D(j,1:4)).^2))/4).^.5;   %finds distance from detector 'i' to each
                                                            %previous detector
                if dist<=d_min   %used to determine minimum distance from detector 'i' to previous detector
                    d_min=dist;                 %saves minimum value
                    C1(1,1:4)=D(j,1:4);         %Stores nearest Detector Set to 'C'
                end
            end
         end
    else
        D(i,1:4)=D(i,1:4);            %First detector remains same, no previous detectors to compare
    end
end                                   %END detector generation
for i=1:150                           %Begin testing Full_data for classification
    d_min=inf;                        %Initialize minimum distance to infinite
    for j=1:1000
        dist=((sum((X(i,1:4)-D(j,1:4)).^2))/4).^.5;    %finds distance from detector 'i' to each
                                                        %test data sample
        if dist<=d_min       %used to determine minimum distance from detector 'i' to any test data sample
            d_min=dist;             %Sets d_min to minimum distance to any test data
        end
    end
    if d_min<r                            %Check if minimum distance is less than radius (0.1)
        self(1,i)=0;                      %If not, self remains zero (Nonself)
    else
        self(1,i)=1;                      %If yes, self set to one (Self)
    end
end
self=self;                                   %Outputs array of classifications (1=self, 0=Nonself)
Total_Self_Incorrect=50-sum(self(1,101:150));  %Outputs count of total self cells identified Incorrect
Total_Nonself=100-sum(self(1,1:100));          %Outputs count of total Nonself cells identified Correctly
Detection_Rate=Total_Nonself*100               %Outputs Detection Rate as a percentage
False_Alarm=(Total_Self_Incorrect*2)*100       %Outputs False Alarm rate as a percentage
```

```
%Modified V-Detector Algorithm for Negative Selection Artificial Immune System
%Testing BUPA Data: 345 Data sets, 145 Disorder, 200 Clean
%Currently training with 100% Data as Clean
T=0;
T=Clean;                                        %Initialize Training Set (Self Set)
for i=1:200
    T_norm(i,1:6)=T(i,1:6)/norm(T(i,1:6));      %Normalize Training Data
end
T=T_norm;                                       %Set T=Normalized Training Data
self=zeros(1,345);                              %Initialize self=0, used to count successful Self detections
F=Full_Data;                                    %Full data set for testing (includes training set)
for i=1:345
    F_norm(i,1:6)=F(i,1:6)/norm(F(i,1:6));      %Normalize Full Data Set
end
F=F_norm;                                       %Set F=Normalized Testing Data
r=.025;                                         %Set Self radius
m=0;   %Initialize m=0, estimates total coverage of nonself space m=1/(1-c) c=%coverage (99.98% currently)
k=1;                                %Initialize k=1, determines detector count/placement
while (m<5000)&&(k<1000)            %generate detector until either threshold is achieved
    x=rand(1,6);                    %generate random detector candidate
    d_min=inf;                      %Initialize dmin to infinite
    for i=1:200
        dist=(sum(abs(T(i,1:6)-x(1,1:6))))/6;  %calculates min distance to self from detector
                                                %(Manhattan metric)
        if dist<=d_min
            d_min=dist;                         %Store minimum distance to dmin
        end
    end
    if d_min<r                      %if min distance less than self threshold (r), generate new detector
        continue
    else
        if (k==1)                   %if 1st detector, store detector
            D(k,1:6)=x;
            rd(k,1)=(d_min*.99);    %store radius of detector as dmin*.99
            k=k+1;                  %increment detector counter
        else
            p=0;                    %initialize p=0, used later
            for j=1:k-1
```

```
                dist=(sum(abs(D(j,1:6)-x(1,1:6))))/6;  %calculate min distance to previous stored detectors
                                                       %(Euclidean metric)
                if dist<(rd(j,1)*.75)  %if new detector distance less than previous detector radius,
                                       %generate new detector, allows overlap of 25%
                    break
                else
                    p=p+1;             %counts if new detector not detected by previous detectors
                end
            end
            if (p==(k-1))              %if new detector not detected by all previous detectors
                D(k,1:6)=x;            %store new detector
                rd(k,1)=(d_min*.99);   %store new detector radius as dmin*.99
                k=k+1;                 %increment detector count
                m=0;                   %Reset detector coverage counter
            else
                m=m+1;                 %If new detector is already covered, increment coverage counter
            end
        end
    end
end                                    %End Detector Generation phase
for i=1:345                            %Begin testing Full_data for classification
    for j=1:k-1
        dist=(sum(abs(F(i,1:6)-D(j,1:6))))/6;   %finds distance from detector 'i' to each test data sample
        if dist<(rd(j,1))                       %Check if distance is less than detector radius
            self(1,i)=0;                        %If yes, set self=0 (Nonself)
            break
        else
            self(1,i)=1;                        %If no, set self to one (Self)
        end
    end
end
self=self;                                      %Outputs array of classifications (1=self, 0=Nonself)
Total_Self_Incorrect=200-sum(self(1,146:345));  %Outputs count of total self cells identified Incorrect
Total_Nonself=145-sum(self(1,1:145));           %Outputs count of total Nonself cells identified Correctly
Detection_Rate=(Total_Nonself/145)*100          %Calculate Detection Rate percentage
False_Alarm=(Total_Self_Incorrect/200)*100      %Calculates False Alarm Rate percentage
k=k                                             %Outputs Detector count
%Proliferating V-Detector Algorithm for Negative Selection Artificial
```

```
%Immune System (2 Stages)
%Currently training with all self data (50 data sets)
%Data Structure for Full_Data (1-50=Setosa,51-100=Versicolor,101-150=Virginica)
%You only need to change T=(flower name) to test for each flower,
T=Setosa;                                       %Initialize Training Set (Self Set)
for i=1:50
    T_norm(i,1:4)=T(i,1:4)/norm(T(i,1:4));      %Normalize Training Data
end
T=T_norm;                                       %Set T=Normalized Training Data
self=zeros(1,150);                              %Initialize self=0, used to count successful Self detections
F=Full_Data;                                    %Full data set for testing (includes training set)
for i=1:150
    F_norm(i,1:4)=F(i,1:4)/norm(F(i,1:4));      %Normalize Full Data Set
end
F=F_norm;                                       %Set F=Normalized Testing Data
r=0.1;                                          %Self radius (Optimal value determined as 0.1)
m=0;    %Initialize m=0, estimates total coverage of nonself space m=1/(1-c) c=%coverage (99.98% currently)
k=1;                                            %Initialize k=1, determines detector count/placement
D=0;                                            %Initialize Detector value to zero
rd=0;                                           %Initialize Detector radius to zero
while (m<5000)&&(k<200)             %generate detector until either threshold is achieved
    x=rand(1,4);                                %generate random detector candidate
    d_min=inf;                                  %Initialize dmin to infinite
    for i=1:50
        dist=(sum((T(i,1:4)-x(1,1:4)).^2)).^.5;  %calculates min distance to self from detector
                                                 %(Euclidean metric)
        if dist<=d_min
            d_min=dist;                          %Store minimum distance to dmin
        end
    end
    if d_min<r                      %if min distance less than self threshold (r), generate new detector
        continue
    else
        if (k==1)                                %if 1st detector, store detector
            D(k,1:4)=x;
            rd(k,1)=(d_min-(r));                 %store radius of detector as (dmin-r)
            k=k+1;                               %increment detector counter
        else
```

```
            p=0;                                    %initialize p=0, used later
            for j=1:k-1
                dist=(sum((D(j,1:4)-x(1,1:4)).^2)).^.5;  %calculate min distance to previous stored
                                                         %detectors (Euclidean metric)
                if dist<rd(j,1)     %if new detector distance less than previous detector radius,
                                    %generate new detector
                    break
                else
                    p=p+1;          %counts if new detector not detected by previous detectors
                end
            end
            if (p==(k-1))           %if new detector not detected by all previous detectors
                D(k,1:4)=x;               %store new detector
                rd(k,1)=(d_min-(r));      %store new detector radius as (dmin-r)
                k=k+1;                    %increment detector count
                m=0;                      %Reset detector coverage counter
            else
                m=m+1;                    %If new detector is already covered, increment coverage counter
            end
        end
    end
end                                       %End Detector Generation phase
off=0;                              %Initialize offspring counter (used for informational purposes only)
k2=k;       %Initialize offspring iteration count (prevents offspring from proliferating until next stage)
for i=1:k-1                         %Begin 1st Proliferation Stage
    p=0;
    x=D(i,1:4)+U1*rd(i,1);          %Generate offspring detector along circumference of prev. stored
                                    %detector, U1=[0,0,0,1]
    for j=1:k2-1
        dist=(sum((D(j,1:4)-x(1,1:4)).^2)).^.5;   %calculate min distance from offspring to previous stored
                                                  %detectors (Euclidean metric)
        if dist<rd(j,1)     %if offspring detector distance less than previous detector radius,
                            %generate new detector
            break
        else
            p=p+1;          %counts if offspring detector not detected by previous detectors
        end
    end
```

```
    if (p==(k2-1))        %if offspring detector not detected by all previous detectors
        d_min=inf;
        for l=1:50
        dist=(sum((T(l,1:4)-x(1,1:4)).^2)).^.5;     %calculates min distance from offspring detector to self
                                                    %(Euclidean metric)
            if dist<=d_min
                d_min=dist;                         %Store minimum distance to dmin
            end
        end
        D(k2,1:4)=x;                                %store offspring detector
        rd(k2,1)=(d_min-(.5*r));                    %store offspring detector radius as (dmin-.5*r)
        k2=k2+1;                                    %increment detector count
        off=off+1;                                  %increment offspring counter
    end
end
for i=1:k-1
    p=0;
    x=D(i,1:4)+U2*rd(i,1);      %Generate offspring detector along circumference of prev. stored detector,
                                %U2=[0,0,0,-1]
    for j=1:k2-1
        dist=(sum((D(j,1:4)-x(1,1:4)).^2)).^.5;  %calculate min distance from offspring to previous stored
                                                 %detectors (Euclidean metric)
        if dist<rd(j,1)                          %if offspring detector distance less than previous
                                                 %detector radius, generate new detector
            break
        else
            p=p+1;                          %counts if offspring detector not detected by previous detectors
        end
    end
    if (p==(k2-1))                      %if offspring detector not detected by all previous detectors
        d_min=inf;
        for l=1:50
        dist=(sum((T(l,1:4)-x(1,1:4)).^2)).^.5;  %calculates min distance from offspring detector to self
                                                 %(Euclidean metric)
            if dist<=d_min
                d_min=dist;                      %Store minimum distance to dmin
            end
        end
```

```
        D(k2,1:4)=x;                                %store offspring detector
        rd(k2,1)=(d_min-(.5*r));                    %store offspring detector radius as (dmin-.5*r)
        k2=k2+1;                                    %increment detector count
        off=off+1;                          %increment offspring counter (only for informational purposes)
    end
end
for i=1:k-1
    p=0;
    x=D(i,1:4)+U3*rd(i,1);    %Generate offspring detector along circumference of prev. stored detector,
                               %U3=[0,0,1,0]
```

(Algorithm repeats as such for U4=[0,0,-1,0], U5=[0,1,0,0], U6=[0,-1,0,0], U7=[1,0,0,0], U8=[-1,0,0,0])

```
%End 1st Proliferation Stage
off2=0;                      %Initialize 2nd offspring counter (informational purpose)
k3=k2;     %Initialize offspring iteration count (prevents offspring from proliferating until next stage)
for i=k:k2-1                 %Begin 2nd Proliferation Stage
    p=0;
    x=D(i,1:4)+U1*rd(i,1);    %Generate offspring detector along circumference of prev. stored detector,
                               %U1=[0,0,0,1]
    for j=1:k3-1
        dist=(sum((D(j,1:4)-x(1,1:4)).^2)).^.5;    %calculate min distance from offspring to previous stored
                                                   %detectors (Euclidean metric)
        if dist<rd(j,1)      %if offspring detector distance less than previous detector radius, generate
                             %new detector
            break
        else
            p=p+1;           %counts if offspring detector not detected by previous detectors
        end
    end
    if (p==(k3-1))  %if offspring detector not detected by all previous detectors (determined by counter p)
        d_min=inf;
        for l=1:50
        dist=(sum((T(l,1:4)-x(1,1:4)).^2)).^.5;  %calculates min distance from offspring detector to self
                                                 %(Euclidean metric)
            if dist<=d_min
```

```
                d_min=dist;         %Store minimum distance to dmin
            end
        end
        D(k3,1:4)=x;                %store offspring detector
        rd(k3,1)=(d_min-(.1*r));    %store offspring detector radius as (dmin-(.1*r))
        k3=k3+1;                    %increment detector count
        off2=off2+1;                %increment 2nd offspring counter (only for informational purposes)
    end
end
for i=k:k2-1
    p=0;
    x=D(i,1:4)+U2*rd(i,1);  %Generate offspring detector along circumference of prev. stored detector,
                            %U2=[0,0,0,-1]
```

(Repeats again for each variation of U, U2, U3, U4, U5, U6, U7, U8)

```
%End 2nd Proliferation Stage

for i=1:150                     %Begin testing Full_data for classification
    for j=1:k3-1
        dist=(sum((F(i,1:4)-D(j,1:4)).^2)).^.5;  %finds distance from detector 'i' to each test data sample
        if dist<(rd(j,1))                        %Check if distance is less than detector radius
            self(1,i)=0;                         %If yes, set self=0 (Nonself)
            break
        else
            self(1,i)=1;                         %If no, set self to one (Self)
        end
    end
end
self=self;                                       %Outputs array of classifications (1=self, 0=Nonself)(
Total_Self_Incorrect=50-sum(self(1,1:50));       %Outputs count of total self cells identified Incorrect
Total_Nonself=100-sum(self(1,51:150));           %Outputs count of total Nonself cells identified Correctly
k3=k3;                                           %Outputs Detector count



%Neural Network Model for Iris Data
```

```
%One layer of 15 hidden neurons
%Self defined as Versicolor (desired output=1)
%Nonself defined as other flowerr (desired output=0)
MSE=100;                    %Initialize MSE to 100
b=1;                        %Assign bias to one
a=1;                        %Initialize constant of logistic function to 1
learn=.2;                   %Initialize learning rate to 0.2
w1=randn(15,4);             %Randomly assign initial weights with zero mean, std=1
w2=randn(1,15);             %Randomly assign initial weights with zero mean, std=1
wb1=randn(15,1);            %Randomly assign initial weights to bias 'b' with zero mean, std=1
wb2=randn(1,1);             %Randomly assign initial weights to bias 'b' with zero mean, std=1
T=Versicolor;               %Assign T=Training Data
F=Iris_Full_Data;           %Assign F=Full Iris Data (All Data Points)
while(MSE>.01)              %Stopping Criteria (Various metrics tested)
    for trial=1:10
        T=T';                   %Transpose T
        T=T(randperm(75),:);    %Randomly arrange columns of T
        T=T';                   %Transpose T
        for i=1:75
            for j=1:15
                yi=T(1:4,i);                %Assign yi=input data from T
                v(1,j)=(sum(w1(j,1:4)*yi(1:4,1)))+b*wb1(j,1);   %Calculate vj=sum(w1(j,i)*yi)+bias*wb1
                                                                  %for each neuron
                yj(j,1)=1/(1+exp(-a*v(1,j)));                   %Calculate output of logistic
                                                                  %function=yj (output of neuron)
            end
            for k=1:15
                vk_temp=sum(w2(1,1:k)*yj(1:k,1));               %Calculate vk from output of each neuron
                                                                   %vk=sum(w2(k,j)*yj)
            end
            vk=vk_temp+(b*wb2);                                 %Calculate final value of vk to include
                                                                   %bias weight
            yk=1/(1+exp(-a*vk));                                %Calulate final output, yk using
                                                                  %logistic function
            ek(1,i)=T(5,i)-yk;                                  %Calculate error =
                                                                  %desired output-actual output

            dk=a*ek(1,i)*yk*(1-yk);                             %Begin Back propogation,
```

```
                                                                %Calculate local gradient (delta k)
        for k=1:15
            w2_new(1,k)=w2(1,k)+(learn*dk*yj(k,1));          %Update weights from output neuron to
                                                               %hidden neuron w2(k,j)
            wb2_new=wb2+(learn*dk*b);                 %Update bias weight of output neuron
        end
        for j=1:15
            dj(1,j)=a*yj(j,1)*(1-yj(j,1))*((sum(dk*w2(1,1:j)))+(dk*wb2));   %Calculate local gradient
                                                                  %for each hidden neuron (delta j)
        end
        for j=1:15
            for k=1:4
                w1_new(j,k)=w1(j,k)+(learn*dj(1,j)*yi(k,1));   %Update weights from input to
                                                                    %hidden neurons w1(j,i)
            end
            wb1_new(j,1)=wb1(j,1)+(learn*dj(1,j)*b);      %Update bias weight of hidden neurons
        end
        w2=w2_new;                              %Assign new weights to w2
        w1=w1_new;                              %Assign new weights to w1
        wb1=wb1_new;                            %Assign new weights to wb1
        wb2=wb2_new;                            %Assign new weights to wb2
    end
end
T=T';                       %Transpose T
T=T(randperm(75),:);        %Randomly arrange columns of T
T=T';                       %Transpose T
for i=1:75
    for j=1:15
        yi=T(1:4,i);                                        %Assign yi=input data from T
        v(1,j)=(sum(w1(j,1:4)*yi(1:4,1)))+b*wb1(j,1);   %Calculate vj=sum(w1(j,i)*yi)+bias*wb1
                                                              %for each neuron
        yj(j,1)=1/(1+exp(-a*v(1,j)));          %Calculate output of logistic
                                               %function=yj (output of neuron)
    end
    for k=1:15
        vk_temp=sum(w2(1,1:k)*yj(1:k,1));     %Calculate vk from output of each neuron
                                                   %vk=sum(w2(k,j)*yj)
    end
```

```
        vk=vk_temp+(b*wb2);              %Calculate final value of vk to include bias weight
        yk=1/(1+exp(-a*vk));             %Calculate final output, yk using logistic function
        ek(1,i)=T(5,i)-yk;               %Calculate error = desired output-actual output
    end
    for k=1:75
        MSE=(sum((ek(1,1:k).^2)/2))/75;    %Calculate mean squared error = sum(ek(N)^2/2)/N
                                                %(N=number of data sets)
    end
end                      %End Training phase (once stopping criteria achieved)
for i=1:150                                              %Begin Testing Full Data
    for j=1:15
        yi=F(1:4,i);                                     %Assign yi to input data T
        v(1,j)=(sum(w1(j,1:4)*yi(1:4,1)))+b*wb1(j,1);    %Calculate vj as before
        yj(j,1)=1/(1+exp(-a*v(1,j)));                    %Calculate yj as before
    end
    for k=1:15
        vk_temp=sum(w2(1,1:k)*yj(1:k,1));                %Calculate vk as before
    end
    vk=vk_temp+(b*wb2);                        %Update vk to include bias weight
    yk=1/(1+exp(-a*vk));                       %Calculate final output yk
    if yk>.5                                   %If output is greater than 1/2
        self(1,i)=1;                           %Data is classified as self
    else
        self(1,i)=0;                           %Otherwise, Data is classified as nonself
    end
end
Detection_Rate=100-sum(self(1,1:50))-sum(self(1,101:150))    %Calculate Detection Rate as defined by AIS
False_Alarm=(50-sum(self(1,51:100)))*2                       %Calculate False Alarm Rate as defined by AIS
```