AN EVALUATION OF MACHINE LEARNING TECHNIQUES IN INTRUSION DETECTION

By

Christina Lee

Thesis
Submitted to the Faculty of the
Graduate School of Vanderbilt University
in partial fulfillment of the requirements
for the degree of

Master of Science

in

Computer Science

May, 2007

Nashville, Tennessee

Approved:
Professor Gabor Karsai
Professor Douglas Fisher

ACKNOWLEDGEMENTS

I would like to thank Gabor Karsai, my advisor, for all of his help on this project. Our discussions on intrusion detection and machine learning techniques allowed me to recognize areas I had overlooked and pointed out interesting areas to explore. I would also like to thank Dr. Fisher, my second reader, for his input on the experiments and thesis background. I would like to thank Eric Hall for the the information he provided me on network topology. In addition, Bradley Malin gave me many useful suggestions on evaluation methods such as ROC curves and cost curves. I would also like to thank those that attended my pizza lecture who asked questions and gave comments. In addition, I would like to thank Sean Duncavage for his comments about neural networks. This research was supported by a Graduate Assistantship sponsored by NASA and an NSF grant.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

This paper will examine the use of machine learning algorithms in intrusion detection. Machine learning algorithms[1] use artificial intelligence and data-mining techniques to analyze and find patterns in data.

Intrusion detection is the art of detecting the break-ins of malicious attackers. Today, computer security has grown in importance with the widespread use of the Internet. Firewalls[2] are commonly used to prevent attacks from occurring. Antivirus and anti-spyware programs can help people to remove already existing automated attacks from their computer. Access control limits physical and networked use of a computer. However, an important component of setting up a secure system is to have some way to analyze the activity on the computer and determine whether an attack has been launched against the computer. Such a system is called an intrusion detection system. An intrusion detection system (IDS) can detect both the automated and manual attacks that a human intruder launches on a network. The system can then decide on a course of action: it may do anything from giving a simple alert message to taking action against the intrusion.

This thesis examines the differences between a Naive Bayes[1], a Decision Tree[1], and an Artificial Neural Net (ANN)[1] to determine the relative strengths and weaknesses of using these approaches. The purpose is to give an evaluation of the performance of these algorithms that will allow someone who wishes to use one of these approaches to understand how accurate the approach is and under what conditions it works well. In addition, a novel evaluation technique will be considered. Accuracy can be evaluated effectively by using Receiver Operating Characteristic (ROC) curves[3]. Cost curves[4][5] can indicate the conditions under which the algorithm works well. We will judge these approaches by their speed and accuracy. Also, the benefits of on-line versus off-line approaches will be examined. These three approaches will also be compared to other possible approaches. In addition, the responses of these three algorithms to different data sets will be analyzed.

Chapter 2 will discuss background information on the problem. A short description of the field of computer security is included. In addition, the literature on intrusion detection will be analyzed to determine the progression of the field and the state-of-the art in IDS's. Furthermore, the algorithms for decision trees, Bayesian networks, and neural networks will be presented.

Chapter 3 discusses the procedure of the experiments that the author has run. Some details about the naive Bayes and decision tree algorithms will be discussed. The programs that run the algorithms were either coded by the author or were from the Weka data mining software[6][7]. The use of these programs, and the steps taken to run the experiments and collect the data will be described. The computing environment used

will be explained. In addition, the datasets used will be explained, as well as any modifications made to these datasets.

Chapter 4 discusses the results of the work. Confusion matrices including the percentage of correctly analyzed data is included, along with the percentages of false positives and negatives. The speed of the systems are discussed, as well as possible aids in human comprehension that the systems afford.

Chapter 5 presents the conclusions drawn from the work. Future work is also presented.

CHAPTER II

BACKGROUND

In this section, concepts relevant to the application of machine learning algorithms in intrusion detection will be discussed. These concepts include computer security, machine learning, and the setup of local networks. When discussing computer security, I will present a taxonomy of computer attacks and categorize the defenses against such attacks. In machine learning, I will briefly present several different techniques, comparing and contrasting these techniques.

Computer Security

An important subject in computer security is the general problem of intrusion detection[8]. There are a large variety of methods in which the security of a system can be compromised. While physically compromising a computer is an important security threat, we will focus on the problem of detecting intruders across the network. Since all data must come through the network via packets, our goal is to examine the various parts of the packet data to determine if an attack is in progress. Intrusion detection is sometimes considered to describe only attacks from an external network and the term misuse is then assigned to describe attacks from the internal network [8].

In order to understand what kinds of data may be useful for detecting intrusions, it is important to consider the basic pathways that both legitimate and malicious users may traverse to use a system. The three basic methods of accessing a computer are: physical access to the host computer, using a physical network to access the computer, and using a wireless network. This thesis will focus on using a physical network, as described below.

Physical Network

In this type of network, data is transmitted through the physical network (as opposed to a wireless network) in the form of packets. Packets have three parts: a header, a payload, and a trailer[9]. The header indicates the beginning of the packet. The payload carries the data being transmitted by the packet. The trailer indicates the end of the packet. The Internet Protocol (IP) is used to communicate across the Internet via IP addresses. It is a higher-level protocol that is used to carry information about many lower-level protocols as described in 9 and [10].

Many large organizations, such as corporations and universities have their own local networks [11]. As an example, consider a network constructed with eight CISCO machines functioning as routers and switches.

These machines are dual connected in a ring with fiber-optic cables. They have two routers connected to the commodity Internet, and two routers connected to the high speed, low latency Internet 2. Each router/switch is connected to a closet containing 24 ports. The network uses firewalls and Virtual Private Networks (VPN). This type of network setup can support an organization about the size of Vanderbilt University, which has a similar setup[11].

Types of Attacks

There are a large variety of different attack types[12]. An attacker may attempt to guess a user's password. An attacker may also monitor the network to obtain the information they require to launch an attack. Sometimes attackers try to put unauthorized programs onto computers that they have access to. Sometimes they may steal information or corrupt information. They may also try to perform a denial of service attack.

In the section below, various attacks are explained. These attacks may be referred to in other parts of this document to describe the behavior of an IDS to some variants of the attack [12].

Buffer Overflow

A buffer overflow attack exploits code that does not check the bounds of a buffer. The attacker may overwrite sensitive data on the stack of an executing process. For example, the instruction pointer on the stack may be overwritten to point to an instruction that the attacker wishes to execute. A detailed description of this problem, as well as possible remedies, can be found in [13].

Teardrop

A teardrop attack is an attack that exploits incorrect handling of overlapping packets. Teardrop is a type of denial of service(DoS) attack because it crashes any machine that is vulnerable to this attack[12]. Some older Windows and Linux systems have been reported to be vulnerable to this attack.

Ping of Death (PoD)

The ping of death is a denial of service(DoS) attack that uses improperly formatted pings. These ping packets have a length greater than the 65535 octets that are allowed. Some Windows, Linux, and Macintosh systems are vulnerable to this attack[12].

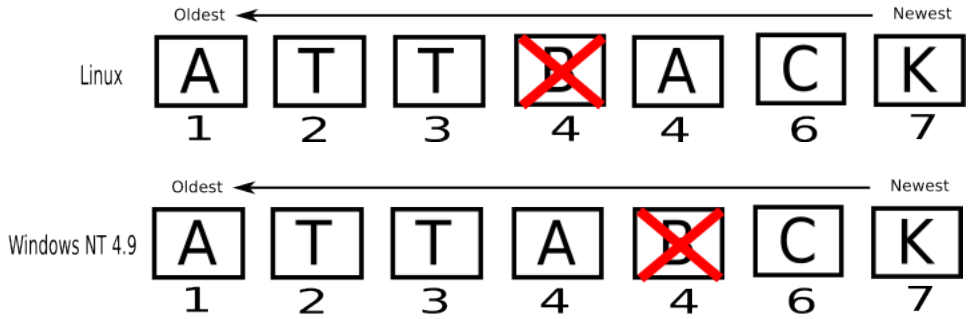

Figure 1: Overlapping Packets

Attack Variations across Operating Systems

Attacks will vary across operating systems. Some operating systems that are discussed here include Windows, Linux, and FreeBSD[14]. There are several factors that cause the variance in attacks across these operating systems. Each operating system has different default behavior for handling incoming packets on a network.

Packets do not always arrive in order. To prevent miscommunication, packets in the same connection are numbered by the sender and then reconstructed by the receiving computer. The problem is that different receivers construct packets differently.

For example, consider the problem of overlapping packets. If two packets have the same sequence number in a connection, they are overlapping packets. In this case, the system must decide how to handle the overlapping packets. The default behavior in Linux is to favor new packets, while the default behavior in Windows is to favor old packets.

In [14], they state that the overlap behavior and response to incorrect options are two factors that can differ between OSes. In [15], the concept of reassembly timeout and how it differs between operating systems is explained in detail. All of the factors listed in these two papers determine how the operating system handles packets. The defaults themselves do not cause the attacks, but rather allow an attacker to slip by an IDS that doesn't behave like the operating system it guards.

Overlap behavior is the behavior an operating system exhibits when responding to overlapping packets. Overlapping packets are packets that are assigned the same TCP reassembly number and have other identical fields but may have different payloads. Windows always favors older overlapping packets and discards the newer packet, a policy called 'First'.

Packet sniffers gather information about packets but do not determine whether those packets are part

of an attack or not. The packet information can help the user decide whether an attack is taking place, however. Wireshark (previously called Ethereal)[16] is one open source packet sniffer.

Intrusion Detection

Intrusion Detection is the art of identifying attacks on a computer. While there are a large variety of attacks, most of them fit into one of four categories:

1. Probe
2. Denial of Service (DoS)
3. User to Root
4. Remote to User

A probe attack is an attempt to learn specific setup information about a network or individual computer. This is not exactly an attack in itself, but rather an attempt to learn information that could facilitate an attack. A probe may be an indication of an impending attack. Many attacks are likely preceded by probes of some sort.

A DoS is an attack that overloads the resources of a system and prevents legitimate users from accessing the system. Such an attack is a concern for commercial applications or mission-critical network components.

A 'user to root' attack is an attack where someone with user permissions attempts to gain root permissions.

A 'remote to user' attack is an attempt to go from an unauthenticated state with a system to a state where the attacker has user permissions. This type of attack can then escalate into a 'user to root' attack.

IDS systems can be described by several characteristics. One of these characteristics is the location they watch for attacks. Thus, IDS's can generally be divided into two categories (as stated in [17]):

1. Network Based IDS use raw packets as the data source
2. Host Based IDS of which there are three types:
   (a) examines logs of host looking for attack patterns
   (b) examines patterns in network traffic (but not in promiscuous mode, which would allow it to view network traffic that is not addressed to the particular host the IDS resides on[18])
   (c) executes both log-based and stack-based IDS

3 types of detection techniques(as stated in [17]) are pattern, frequency, and anomaly. Pattern detection, also known as signature detection, uses known information about attacks to detect them. Anomaly detection requires the collection of normal activity, and flags any deviation from normal activity as abnormal (although not necessarily an attack). Frequency detection checks for behavior that crosses a certain threshold [19].

Common Intrusion Detection Components

There are several common intrusion detection components. In the Common Intrusion Detection Framework (CIDF) model [14] outlined in 1998, these are called:

1. event generators ("E-boxes") – the E-box gathers data and organizes it into events. Events can be simply packets read off the wire or higher-level information.
2. analysis engines ("A-boxes")–Analyze input to detect attacks.
3. storage mechanisms ("D-boxes")–Store information from E-box and A-box. For example, the information may be stored in a database for later retrieval [20].
4. countermeasures("C-boxes")–React to analysis information.

Problems with CIDF systems

Any intrusions detection system in the CIDF model is highly vulnerable to the following types of attacks:

1. Insertion: Insertion is the creation of packets that will be accepted only by the IDS. Usually, these packets are malformed in some way so that they will be dropped by the target system. The target system will reconstruct the message without the inserted packets, and this message can be an attack.
2. Evasion: Some IDS systems try to combat insertion techniques by rejecting bad packets. These can be susceptible to evasion techniques. Evasion is the creation of packets that will be rejected by the IDS but accepted by the target system.
3. Denial of Service: Denial of Service attacks on the IDS can affect its availability and thus leave the target system open to attacks.

As stated by [15], Snort, an open source rule-based IDS[20], has since addressed many of the issues that plagued the CIDF model.

Snort utilizes a preprocessor that can be customized to fit the default behavior of a variety of operating systems. This preprocessor is customized through text-based configuration files. While it requires network

administrators to have detailed knowledge of the operating systems used and their default behavior, this knowledge will allow them to configure Snort to more accurately monitor the machines on their network.

However, many challenges still exist in the field of intrusion detection.

Previous Work

A paper [21] titled "The Base-Rate Fallacy and its Implications for the Difficulty of Intrusion Detection" explains why trying to obtain low false alarm rates are one of the major challenges in ID. The base-rate fallacy is the error of not taking the rate of occurence of an event into consideration when calculating the probability that a given classification of the system is correct. The paper points out that because of the base-rate fallacy, people may often view the percentage of false negatives as within an acceptable range despite the fact that it is unacceptably high. The author demonstrates, through the use of Bayes theorem, that the false negatives in actual intrusion detection systems are very high. They conclude that their results present a challenge for the present models of intrusion detection, but that there is no conclusive evidence for whether or not the types of modern-day systems can, with modifications, meet these standards. The paper also points out the lack of data available in this area of study, as well as the neglect of most papers to discuss several fundamental issues to intrusion detection. These are listed by the authors as:

- Effectiveness
- Efficiency
- Ease of Use
- Security
- Inter-Operability
- Transparency

This paper addresses the issue of effectiveness in terms of false alarms. The paper also states that even as the false alarm rate is lowered, a trend exists that the false negatives increase.

Snort [22] is open source intrusion detection software. This software uses rules created by security experts to detect intrusions. A Snort rule resembles the rules found in firewalls. One example of such a rule (similar to one in [23]) is:

```
alert tcp any any -> any 21 (msg: "FTP ROOT"; content:"user root";)
```

is a Snort rule that will alert the user when someone tries to log in as root over FTP (port 21). The software is primarily composed of four components(as described in [20]):

1. packet sniffer and decoder using libpcap, an application primarily used for packet capture[24].

2. preprocessor examines for malformations, anomalies, and noncompliance.

3. compares normalized stream re-assembled data against rule base.

4. output and alerting module writes and logs packets in various ways according to the configuration.

Snort also has the following modes:

1. Packet Sniffer mode

2. Intrusion Detection Mode

In addition, Snort is a command line tool uses a rule-based method to detect attacks over a network. To perform this task, Snort has a NIDS (network intrusion detection system) mode. The software uses rules to detect attacks on the network. These rules must be written by people with experience with network attacks. The rules used in Snort are similar to firewall rules. Snort may be used by other Intrusion Detection systems to collect data for training an Intrusion Detection system.

Snort can be configured to work with various software. Some of this software includes:

1. Barnyard: Barnyard is a program used to interface between Snort and a MySQL database. Though Snort can perform the task of placing entries in the database itself, Barnyard is specialized for this purpose.

2. MySQL: A MySQL database can be used to store all events captured by Snort, either through a Snort plugin or through Barnyard

3. Custom intrusion detection systems

The paper [25] discusses a prototype design for a multi-agent system. This design of this system involves two multi-agent systems. The first system, called Adaptive Hierarchical Agent-based Intrusion Detection System(AHA!IDS), is intended to provide the ability to improve detection capabilities, whereas the Adaptive Agent-Based Intrusion Response system (AAIRS), is intended to provide the ability to improve responses.

Some potential issues with this architecture include:

- No results are cited by the authors–the prototype has not been implemented

- The plan for implementation is somewhat vague.

[26] discusses some of the work that has been done in the Columbia IDS project. This paper states that:

- Much work has been done in the area of rule-based detection

- Research has been done in data mining algorithms that propose useful feature sets for modeling attacks. e.g Madam/ID
- Attacks on IDS's themselves are a concern

The following systems were used in the Columbia IDS project:

- RIPPER: AT&T software using a rule-based system of machine learning
- JUDGE: entire IDS system that was developed at Columbia University

The authors identified several problems that need to be addressed. One issue is that intrusion detection commonly only uses rules. For this they propose using a hybrid system that uses both signature-based and anomaly-based techniques. Signature based techniques use known information about the structure of an attack to alert the IDS user, while anomoly based techniques use known information about normal behavior to alert the IDS user when behavior is out of the ordinary. This author hypothesizes that the approach they descrbe could possibly increase the accuracy of the system, but requires more work to collect information than one technique or the other. Another issue is that data-mining techniques tend to require non-noisy training data. They have therefore designed a system that uses probabilities to handle noise in training data. They also discuss the use of data-mining as an anti-virus tool.

In [27] the STAT framework, a modular approach to attack modeling languages, is described. This approach utilizes abstraction-based intrusion detection in distributed environments. It also utilizes a hierarchical model for event abstraction, signatures, and system views.

In "Testing Intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory"[28], the authors state that there is little prior work evaluating Intrusion Detection. They also point out the following problems with the evaluations:

- They were developed from attackers point of view, not the attacked
- Synthesized attack data is problematic

The paper titled "Mimicry attacks on host-based intrusion detection systems", [29] describes what a mimicry attack is. The authors state that it allows the attacker to conceal their attack from the IDS. This article demonstrates the vulnerability of a certain IDS to mimicry attacks.

The MORPHEUS system [30] was created as a tool to use on an unlabeled training set to ensure that the set consists of only normal activity. It can be used with anomaly detection tools in order to increase the chances of feeding them a data set with normal activity. They use motifs for the purpose of classifying

normal data, which is described also as "clean" data. Motifs are defined as a sequence of system call sequences (SCS's). They are a subsequence of a total set of SCS's. Two motif extraction processes are described: auto-match and cross-match. Auto match looks for matching subsequences in a single sequence. Cross-match looks for matching subsequences across several sequences. Using the motif data, outliers corresponding to anomalous activity can be calculated with LOF (local outlier factor). MORPEUS was tested in conjunction with two anomaly detection systems, STIDE and LERAD. MORPHEUS improved attack detection for the ftpd and ps applications in both STIDE and LERAD but not for other applications since the other applications had attacks that were different in character than the ones in the training set, and which were therefore detected by the anomaly detection systems anyway.

In [31] , the authors examine the possibility of using sequences of system calls for classifying intrusions.They use a method called anomaly dictionaries to perform this function.

In [32] describe a system called DOME. This host-based ID checks that system calls are being made from executables running on the system.

The authors of [33] describe a way to build a model of suspicion. This model states that a system should check any behavior that is a little unusual by performing a more thorough examination. However, it should not assume that all abnormal behavior is an attack. The authors state that such a model could link disparate indications of an attack together.

A paper by [34] titled "EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances" discusses an architecture that is suitable for intrusion detection in large networks, employs distributed real-time analysis, and can detect coordinated attacks. The authors state that this solution does not replace but complements host-based approaches. Emerald uses light-weight monitors to accomplish the many tasks it is to perform. These monitors can be coded and tested separately and then integrated into a single tool.

[35] focus on drawing conclusions about the implementation of EMERALD, which at this point is underway. No results are cited, but the authors state that they expect the system to perform well.

Work with EMERALD continued in [36], where the eXpert-BSM monitor is described. The system is stated to have been successfully deployed in more than one system and some results of the monitor usage are presented.

Additional monitors are described in [37], [38], and [39]. Use of P-BEST in EMERALD is described in [40]. The Production-Based Expert System Toolset (PBEST) uses rules, which the system can construct from reasoning about facts that it is given.

Machine Learning Intrusion Detection Algorithms

A paper [41] called "ADAM: A Testbed for Exploring the Use of Data Mining in Intrusion Detection" discusses a testbed for data-mining approaches in intrusion detection.

ADAM (Audit Data Analysis and Mining) is an intrusion detector built to detect intrusions using data mining techniques. It first absorbs training data known to be free of attacks. Then it uses an algorithm to group attacks, unknown behavior, and false alarms. It uses association rules to store knowledge about the data. In the future, they wish to decrease the necessity of training data, which is difficult to obtain. They also want to combine results from a sensor network.

ADAM has several useful capabilities. The authors describe ADAM as capable of:

* Classifying an item as a known attack.

* Classifying an item as a normal event.

* Classifying an item as an unknown attack.

* Match audit trail data to the rules it gives rise to.

ADAM operates by first storing a collection of normal behavior by mining data known to be free of attacks. Then it uses an on-line method to compare incoming connections to the normal dataset. Those that do not match are further examined. Then ADAM uses a classifier to place the suspicious data into one of three categories: false alarm, unknown attack, or known attack.

ADAM, when tested against six other techniques, was found to be the second best[41] in total number of correct evaluations. It was found to be the third best[41] when calculating the percent of attacks it detects. This seems to indicate that ADAM has a higher false negative rate than false positive rate. While this may cause it to miss attacks, it decreases the number of false positive alerts issued.

In [42], the authors propose a method of intrusion detection using an evolving fuzzy neural network. This type of learning algorithm combines artificial neural networks (ANNs) and Fuzzy Inference Systems (FIS's), as well as evolutionary algorithms. They create an algorithm that uses fuzzy rules and allow new neurons to be created in order to accomplish this. They use Snort to gather data for training the algorithm. They compared their technique to an unaugmented neural network. The classification accuracy of the two algorithms was similar and both had a very high accuracy for the types of attacks compared. They indicate that they also had better results by using 40% fewer input variables.

ADMIT [43] is an anomaly-based intrusion detection system that works by clustering its data to distinguish between normal and anomalous computer use. It is host-based, as opposed to network-based. The security issues this IDS handles therefore involve those such as detecting anomalous use after the intruder

has already gained access to the system, perhaps by using an unattended terminal, guessing a password, or being an authorized user.

Some of the advantages of ADMIT are that the system only presents the centers of clusters to the administrator, with the hope that this will reduce the data that the administrator must cope with. In addition, the authors state that it requires less training time than other methods. In addition, their system had more correct categorization of behavior than another system using the same dataset. Lastly, ADMIT can perform real-time intrusion detection, as opposed to needing to be run offline.

The paper explains the algorithms used in detail. It also discusses the dataset used, which was taken from a thesis by T. Lane from Perdue University. In addition, the paper cites that the detection rate was as high as 80.3%, the false positives being as low as 15.3%. The paper also describes how the authors varied parameters in their algorithm and gives a summary of how these changes affected the results.

In "Artificial Intelligence Techniques Applied to Intrusion Detection", [44] propose a technique for intrusion detection using neural networks and fuzzy logic. Data capture is accomplished using Snort. They propose an architecture for an IIDS (Intelligent Intrusion Detection System). Their work is in a preliminary stage and no experimental results are given in the paper.

In "Intrusion Detection in Wireless Networks using Clustering Techniques with Expert Analysis" [45] present an unsupervised learning approach for intrusion detection.

In "An Approach to Implement a Network Intrusion Detection System using Genetic Algorithms", [46] discusses a genetic Algorithm as an Alternative Tool For Security Audit Trails Analysis (GASSATA) The paper describes an approach whereby the GA evaluates rules and discards bad rules, while generating more rules to reduce the false alarm rate and increase the intrusion detection rate. No results of the detection rate are available yet.

In "A Data Mining Approach for Database Intrusion Detection" [47] describe a system that uses data mining on data dependencies in a database to identify transactions corresponding to an attack.

In "Data Mining Aided Signature Discovery in Network-based Intrusion Detection System" [48] implement a system called SigSniffer. This method combines data mining and rule-based approaches. The authors claim this represents an improved approach to data mining for signatures. The data mining methods are used to create a signature discovery system (SDS).

In "The Utilization of Artificial Intelligence in a Hybrid Intrusion Detection System" [49] , intrusion detection using trend analysis, fuzzy logic and neural networks is discussed.

In "Application of a Distributed Data Mining Approach to Network Intrusion Detection" [50], the authors state that collecting and analyzing data from several hosts may not be possible. Therefore, instead of attempting this, they create a network profile with distributed data analysis. The network profile records

information about the network instead of information about the individual intrusions. In the system described, they transfer only the location of data on a computer (not the data itself) to be compiled into a decision tree. They can also compress the transfer of indices to the location of such data on the network.

Both [44] and [42] have examined the possibility of applying a combination of fuzzy logic and neural networks to intrusion detection. The latter examines the difference between a typical ANN and their own fuzzy neural net hybrid. These differences are somewhat small, which may indicate that a typical neural net performs just as well if the statistical significance of the results is taken into account. However, the fuzzy neural net runs faster.

In the table below, the columns categorize the type of Intrusion detection as anomaly detection, rule-based, machine learning, or hybrid. A system can be in more than one category. Hybrid systems employ more than one technique (such as using both rule-based and machine learning approaches).

Table 1: Categories of Intrusion Detection Systems

Anomaly Detection	Rule-Based	Machine Learning	Hybrid
ADMIT	SNORT	EFuNN	ADAM
Decision Tree	MORPHEUS	Decision Tree	EMERALD
Naive Bayes		Naive Bayes	
		RIPPER	

Experimental Results of Machine Learning Systems

In [42], the authors compared an Evolving Fuzzy Neural Network (EFuNN) and an Artificial Neural Network (ANN). They used a modified version of the DARPA Intrusion Detection Evaluation data provided by MIT Labs. The original dataset contained 4,940,000 connection records with 41 attributes each. They processed this into 11,982 records. The training dataset was selected from this set to have 5092 records. The testing dataset was selected to have 6890 records. They also reduced the number of attributes to 13,14, 15,17, and 16 for Normal, DOS, U2L, U2R and Probe, respectively.

Table 2: Comparison between EFuNN and ANN

	Percentage Correctly Identified	
	EFuNN	ANN
Normal	99.56%	99.57%
Probe	99.88%	94.62%
DOS	98.99%	98.97%
U2R	65.00%	59.00%
R2L	97.26%	97.02%

From the table, it can be seen that the U2R class has the lowest percentage of correctly identified connections. The authors state that the EFuNN took a few seconds in the training phase while the ANN took a few minutes. They conclude that the rapid training time was a result of reducing the number of attributes. In addition, this author believes that the small number of training examples also contributed significantly to the fast training time. However, the paper does not present the tradeoffs of using a different numbers of attributes.

In another paper[51], the authors compared the results of clustering using a swarm algorithm with the results of some other unsupervised training algorithms. They recorded the results the system obtained at five different values of the swarm similarity coefficient, β. Swarm similarity calculates the integration of the similarity of an example with another example. When used to aid clustering, it can be calculated as:

$f(o_i) = \sum_{o_j \varepsilon Neigh(r)} |1 - \frac{d(o_i,o_j)}{\beta}|$

Where Neigh(r) is the neighborhood, usually a circular region of radius r. The value $d(o_i, o_j)$is the distance between examples o_i and o_j. This distance can be either Euclidean or Manhattan distance. β, the swarm similarity coefficient, determines how similar clusters are and how fast the algorithm converges.

When run on the KDD training dataset[52], the detection rate varied between 25.34% and 99.97% as β varied, and the false positive rate varied between .67% and 20.1%. In the KDD testing dataset, the detection rate varied between 21.23% and 93.07%, and the false positive rate varied between .11% and 8.24%.

The results were then compared to the unsupervised approaches tested in [53] over a ROC curve. These approaches included Clustering, K-NN, and SVM. The highest detection rates were 93%, 91%, and 98% respectively. The highest false positive rates were all 10%.

The authors state that their approach is better when the detection rate is high. This is due to the lower false positive rate in the testing dataset.

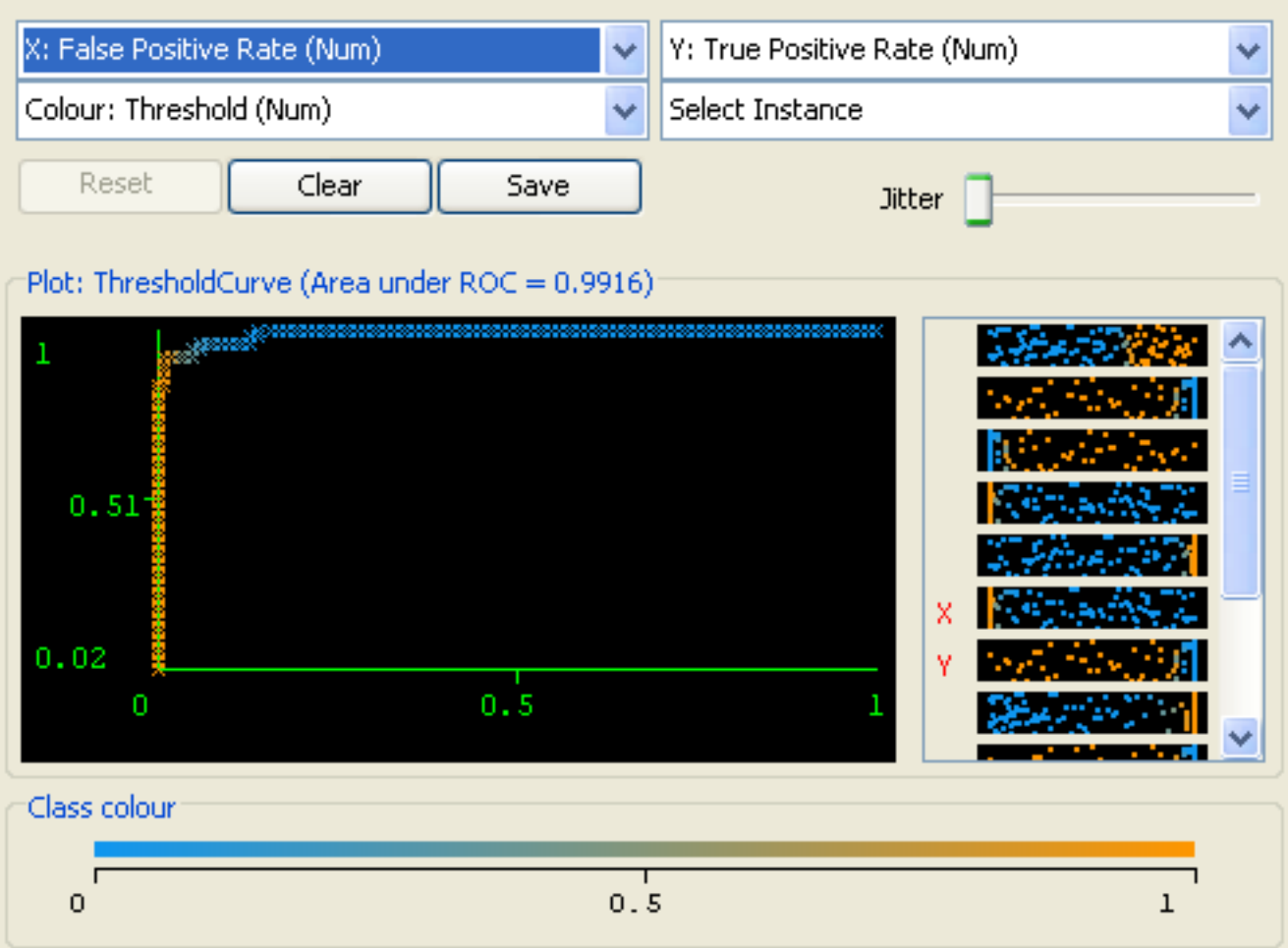

Figure 2: ROC Curve obtained by Weka

Evaluation Methods

Receiver Operating Characteristic (ROC) curves, identify how the true positives vary with the false positives[54]. An example of a ROC curve, obtained from the open source software Weka[6][7], is shown in Figure 2.

The area under these curves signifies how well the test used can distinguish between the examples. The more the example classes overlap relative to the test, the less the area under the ROC curve will be. ROC curves show how well a test does at distinguishing between classes without taking the relative frequency of the classes into account. The author has observed that in the ROC curves created by the Decision Tree in Weka, there are few points. This is most likely because many of the decision tree branches are discrete. Whether an edge is followed is therefore a binary decision (e.g. Something either is or isn't an ftp connection). On continuous valued attributes, the numbers can be changed to affect the number of examples classified as normal and as an attack)

Cost curves complement the use of ROC curves in that they allow a person to quickly compare how two algorithms perform over differences in class frequency[54][5][4]. the expected cost of two algorithms given the probability cost function (PCF). The PCF can be expressed as the following:

$PFC(pos) = \frac{p(pos)c(neg|pos)}{p(pos)c(neg|pos)+p(neg)c(pos|neg)}$

where p(pos) is the probability of a positive classification, p(neg) is the probability of a negative classification, c(neg|pos) is the cost of misclassifying a positive example as negative, and c(pos|neg) is the cost of misclassifying a negative example as positive. The PCF for positive examples increases as the probability of a positive example increases and when the cost of misclassifying a positive example as negative increases.

An additional interesting question is to ask how a supervised learning algorithm changes when the training data changes. Supervised learning is learning with labeled training data[1]. This question has particular relevance to the tests done in this thesis because these tests use simulated data (from [52]). This data contains a large number of attacks compared to the amount of normal data. Therefore, it would be useful to know how well this simulated data can be used to train intrusion detection systems in discovering real attacks. Measuring how the performance of the learning algorithm changes when the training data is altered (e.g. by removing entries, changing the relative percentages of classes, and grouping classes together in various ways) could be useful.

One problem in training supervised learning algorithms in intrusion detection is that test data is difficult to obtain. In addition, the data used is often not real data. Privacy concerns make the use of real data challenging. Perhaps obtaining real data using 'honeypots'[55] could help to eliminate privacy concerns. While this might make the network unrealistic, the attacks performed on it would be real attacks.

Datasets

This section explains what datasets the various systems described above were run on.

Many of the tests conducted used the 1999 KDD Cup data [52] in their tests, although they used it in different ways. Some of them used the larger dataset and others used the dataset containing 10 percent of the larger dataset.

The 1999 KDD Cup data is actually a version of the 1998 dataset from the MIT Lincoln Laboratory DARPA Intrusion Detection Evaluation [56].

The KDD Cup 10% testing data contains 311, 029 entries in total. The testing data contains 60,593 normal entries (19.48% of the total). The testing data also contains 250,436 attack entries (80.52% of the total entries).

The KDD Cup 10% training data contains a total of 494,020 entries. It contains 97,277 normal entries (19.69 %) and 396,743 attack entries (80.31% of the total).

A paper discussing the ADMIT IDS [43] uses a dataset that was taken from a thesis by T. Lane from Perdue University.

MIT Lincoln Laboratory has several datasets that are part of the DARPA Intrusion Detection Evaluation. These are divided into 1998, 1999, and 2000 datasets. The 1998 dataset contains seven weeks of training data and two weeks of testing data. The 1999 DARPA Intrusion data contains three weeks of training data and two weeks of testing data. The 2000 DARPA Intrusion data contains datasets for two scenarios where a novice attacker runs an attack against a naive defender.

Machine Learning

Two categories of machine learning are supervised learning and unsupervised learning. Supervised learning uses labeled training data, whereas unsupervised learning uses unlabled training data. Supervised algorithms classify examples into known classes, whereas clustering algorithms first discover the classe and then categorize them[1]. This section will focus on supervised learning with the machine learning techniques of Decision Trees, Naive Bayes, and Artificial Neural Networks (ANN).

Decision Trees

A Decision Tree is a representation of how to make a decision according to a particular attribute set[1]. Any given Decision Tree is completely deterministic, although some algorithms can alter their trees given additional knowledge (e.g. [57]). Each node of a decision tree is some attribute, with the branches representing alternative values of that attribute. Leaves represent the class to place an example in. To make a decision using a decision tree, select a set of values for an attribute and start at the root of the tree. Using the value of the root attribute, make a choice of which branch to take. Then continue making such choices at each node until a leaf node is reached. The decision in this leaf node is (hopefully) the best decision to make given the information available. Building Decision Trees is a form of supervised learning. A decision tree must use a heuristic to determine which attribute is the most informative attribute, since this attribute will be placed at the root of the tree. The 'most informative attribute' heuristic is taken from information theory, which is a method to calculate the information in a piece of data independent of its meaning[58]. It seeks to calculate the information content of any piece of data by thinking of the answer as a series of bits. Therefore, a one bit answer encodes one bit of yes/no information.

Decision trees are dependent on the order of the data, and it is possible for the optimality of the tree to change when changing the order of the data. However, the heuristic used is designed to minimize this difference.

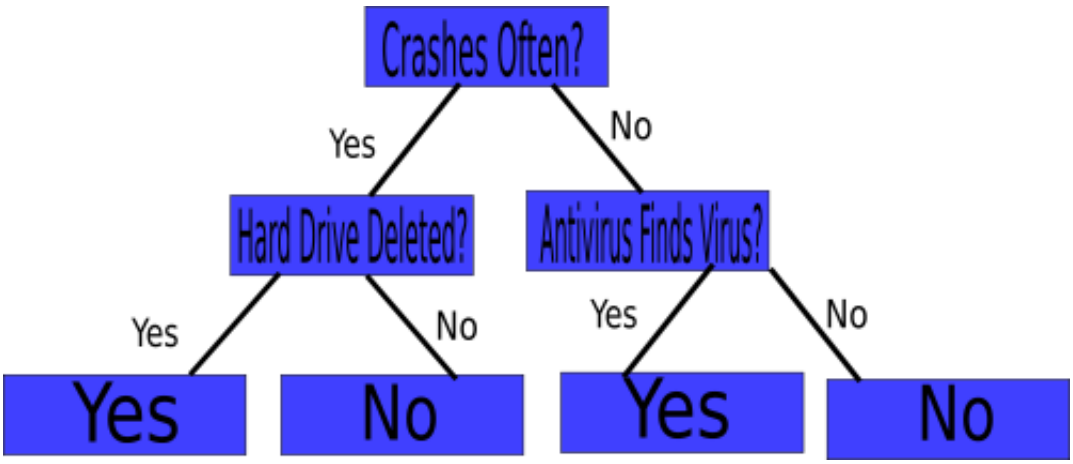

Figure 3: Decision Tree

Information Theory

Information theory proposes a method to measure the amount of information in a given piece of data. Information is defined in terms of a set of possible messages[58]. The more information is in the data, the greater the number of possible messages out of which this particular message must be chosen. For example, if the value of the message m can be {heads, tails} then m can communicate 1 bit of information. That is, it can take on two possible values. The message n chosen from the set {0, 1} communicates the same amount of information. In addition, a message p chosen from the set {a,b,c} contains more information than the messages m and n. Information under this definition is independent from the concept of the semantic value of the message. Let v_i be the value i of an attribute. Let n be the number of possible values that the attribute can take. Then the information in bits needed to describe the attribute is:

$I_{entropy} = -\sum_{i=1}^{n} P(v_i) \log P(v_i)$

For example, the information in bits needed for a fair coin is:

$-.5 \log_2(.5) - .5 \log_2(.5) = -.5(-1) - .5(-1) = .5 + .5 = 1$

The amount of information in bits needed for a weighted coin where heads comes up only 25% of the time:

$-.25 \log_2(.25) - .75 \log_2(.75) = .25(-4) + .75(.41) = 1.31 = 1.31$

For example, if the event always occurs, then logiP(vi) is 0 where P(vi) is 1, and undefined everywhere else. So the information needed to characterize the attribute is zero. The answer is already known in advance. If the event never occurs, then P(vi) is zero and logiP(vi) is undefined. This means that because the event has never happened, it is not possible to characterize anything about the event.

Let Y be the number of 'yes' answers and N be the number of 'no' answers. The information needed to answer a yes/no question is then the information that one would need before knowing anything about the attribute values. This can be summarized as the information:

$I_{neededwithoutattribute}(\frac{Y}{Y+N}, \frac{N}{Y+N}) = -\frac{Y}{Y+N} \log_2(\frac{Y}{Y+N}) - \frac{N}{Y+N} \log_2(\frac{N}{Y+N})$

$$I_{neededwithattribute} = \sum_{i=1}^{V} \frac{Y_i + N_i}{Y + N} I_{withoutattributevalue}(\frac{Y_i}{Y_i+N_i}, \frac{N_i}{Y_i+N_i})$$
$$I_{gain} = I_{neededwithoutattribute} - I_{neededwithattribute}$$

Information gain calculates the difference in the amount of information required to obtain the answer before and after a given value of a variable is found. The larger this difference is, the higher the information gain is. The smaller this difference is, the more information is required to obtain the answer. Another way to think of this is that the larger the information gain, the more useful the variable is to determining the answer. The smaller the information gain is, the less useful the variable value is to obtaining the answer.

Naive Bayes

The naive Bayes model is a heavily simplified Bayesian probability model[1]. In this model, consider the probability of an end result given several related evidence variables. The probability of the end result is encoded in the model along with the probability of the evidence variables occurring given that the end result occurs. The probability of an evidence variable given that the end result occurs is assumed to be independent of the probability of other evidence variables given that the end result occurs.

Now we will consider an example. Suppose that a hypothetical car alarm that responds correctly 99% of the time. The other 1% is divided into two categories, false positives, and false negatives. False positives make up all the situations in which the car alarm goes off, but where there is no criminal activity occurring. Assume that 1% of the time that the alarm rings, that this is the case. False negatives make up all of the situations in which the car alarm does not go off, but there is an attempted theft. Assume that this event also makes up 1% of all cases in which the alarm does not go off. Now, assume that the probability of criminal activity occurring with this particular car to be 1% in any given hour. Over a period of 1 hour, the car is left unsupervised. The alarm goes off once in this time–what is the probability that a theft occurred when the alarm went off? What is the probability that a theft did not occur when the alarm went off?

One way to approach this problem is to use the concept of natural frequencies[59]. Natural frequencies translate the probability into concrete whole numbers before transferring them back into probabilities. For example, a probability that a fair coin gives heads can be thought of as the idea that out of 1000 cases, 500 will be heads.

Examining the car burglary case, we know that the probability that a theft occurred is 1% in any hour. Therefore, in considering the natural frequency, we can assume that over a period of 10,000 hours, 100 hours will have thefts (since there is a 1% probability for theft in any hour). This period of 10,000 hours can therefore be divided into two categories: those that have thefts, and those that do not. The number of hours having thefts, as stated earlier, is 100. the number of hours not having thefts is 9,900. Therefore, in the

number of hours having thefts, 100, the car alarm will, on average, go off 99 times. the other 1 time it will not go off. In the 9,900 hours in which no thefts occur, the alarm will go off 99 times. 9,801 times, it will not go off. Therefore, the total number of hours with alarms is 198. The total number of hours without alarms is 9,802. So the probability that a theft occurred when the alarm went off is 99/198, or 50%. The probability that a theft did not occur given that the alarm went off is 99/198, also 50%. Note that despite the fact that the false negatives only occur 1% of the time, the alarm is nonetheless incorrect 50% of the time that it goes off due to the fact that thefts occur much less commonly than non-thefts.

The above problem can also be expressed as follows:

Let $P(correct) = .98$

Let $P(alarm|\overline{event}) = P(falsepos) = .01$

Let $P(\overline{alarm}|event) = P(falseneg) = .01$

Let $P(event) = .01$

Therefore,

$P(correct|event) = P(\overline{falseneg}) = .99$

$P(alarm) = P(correct|event) \times P(event) + P(falsepos) \times P(\overline{event}) = .99 \times .01 + .01 \times .99 = .0099 + .0099 = .0198$

$P(alarm|event) = P(\overline{falsepos}) = P(\overline{alarm|\overline{event}}) = .99$

$P(event|alarm) = \frac{P(alarm|event) \times P(event)}{P(alarm)} = \frac{.99 \times .01}{.0198} = \frac{.0099}{.0198} = .5$ (using Bayes's rule)

$P(\overline{event}|alarm) = \frac{P(alarm|\overline{event}) \times P(\overline{event})}{P(alarm)} = \frac{.01 \times .99}{.0198} = .5$ (using Bayes's rule)

As can be seen from the above example, the number of false positives must be reduced to significantly to prevent the alarm from becoming more annoying than helpful.

Now, we will consider the alarm example using a naive Bayes classifier. Assume that we have a set of examples that monitor some attributes such as whether it is raining, whether an earthquake has occurred, where the car is parked, etc. Lets assume that we also know, using this monitor, about the behavior of the alarm under these conditions. In addition, having knowledge of these attributes, we record whether or not a theft actually occurred. We will consider the category of whether a theft occurred or not as the class for the naive Bayes classifier. This is the knowledge that we are interested in. The other attributes will be considered as knowledge that may give us evidence that the theft has occurred (the actual usefulness of this knowledge as evidence will be discussed later).

The naive Bayes classifier operates on a strong independence assumption[1]. This means that the probability of one attribute does not affect the probability of another. For example, we assume that the probability of an earthquake does not affect the probability that the alarm goes off. So for two events X and Y, the probability of X occurring given that Y occurs is simply the probability that X occurs. In other words,

$P(X|Y) = P(X)$

The strong independence assumption is unrealistic in most situations. Given a series of n attributes, the naive Bayes classifier makes 2n! independence assumptions. Nevertheless, the results of the naive Bayes classifier are often correct. [60] examines the circumstances under which the naive Bayes classifier performs well and why. They state that that the error is a result of three factors: training data noise, bias, and variance. Training data noise can only be minimized by choosing good training data. The training data must be divided into various groups by the machine learning algorithm. Bias is the error due to groupings in the training data being too large. Variance is the error due to these groupings being too small. The error due to bias in zero-one loss is stated to be generally much lower than the error from variance.

In the training phase, the naive Bayes algorithm calculates the probabilities of a theft given a particular attribute and then stores this probability. This is repeated for each attribute. The time taken on this activity is proportional to n, the number of attributes, and the amount of time taken to calculate the relevant probabilities for each attribute. The amount of time taken to calculate the probability of theft for an attribute is proportional to e, the number of examples given to calculate the probability from, and v_i, the number of different values that attribute a_i can take. Therefore, the amount of time taken to finish the training phase is $O(\sum_{i=1}^{n} a_i v_i e)$. In practice, $\sum_{i=1}^{n} a_i v_i \ll e$, so this value can be considered a constant and thus the time taken is $O(e)$. Therefore, the naive Bayes training phase is linear with the number of examples given.

In the testing phase, the amount of time taken to calculate the probability of the given class for each example in the worst case is proportional to n, the number of attributes. Assuming that the index v_i, the value of the attribute, can be accessed in constant time, the time taken to find the relevant probabilities and multiply them together is $O(n)$ for each example. Given e examples, the time taken is proportional to $O(ne)$. Because it is generally true that $e \gg n$, the time taken in the worst case is therefore $O(e)$. Thus the testing phase takes the same amount of time in the worst case as the training phase. For each example in the testing phase, the following multiplication is performed to calculate the predicted class of the example:

$P(Class, v_1, v_2, ..., v_n) = P(Class) \prod_{i=1}^{n} P(v_i|Class)$

Then, we calculate:

$P(\overline{Class}, v_1, v_2, ..., v_n) = P(\overline{Class}) \prod_{i=1}^{n} P(v_i|\overline{Class})$

The probability that is greater is the one that our naive Bayes model assumes the answer to be. It should be noted that both examples above should be divided by $\prod_{i=1}^{n} P(v_i)$ over the entire set of examples. This is necessary to calculate $P(Class|v_1, v_2, ..., v_n)$ or $P(\overline{Class}|v_1, v_2, ..., v_n)$ according to Bayes's rule. However,

as this is the same for both equations, it is unnecessary and therefore not included in the final calculation to save a constant factor of time.

Artificial Neural Networks

An artificial neural network (ANN), as described by [1], is a network in which the most important component is called a 'neuron'. The neuron has connections to other neurons through which it receives and transmits data. The neuron performs the following computation: the values of the connections into the neuron are multiplied by the respective weights of those connections:

Let $y_k$be the value sent over the kth connection

Let $w_k$be the weight sent over the kth connection

$a_j = \sum_{k=1}^{p} w_k y_k$

Then, a non-linear activation function f is applied to to this value:

$y_j = f(a_j)$

A common choice for f is the sigmoid function, $f(x) = \frac{1}{1+e^{-x}}$. However, other choices for the activation function include the tanh function.

The neurons are arranged into two or more layers. The data goes through the layers in a linear fashion, going first through the input layer, passing through each hidden layer in turn, until it is finally output by the final layer. Hidden layers are the layers between the input and output layers in a feedforward neural network[1].

The process of back-propagation is one method used by neural networks in order to learn the data set[1]. The neural net first goes through forward propagation and then compares its output with the actual classes of the examples. It then adjusts the weights of each layer to better learn the results. Each phase of forward-propagation and backpropagation is called an epoch. A neural net continues running epochs until a certain threshold of accuracy is reached or a certain amount of epochs have been run, whichever comes first.

Backpropagation updates the weights of the hidden and output layer. This method is discussed in [1] and [61].

CHAPTER III

EXPERIMENTS

Algorithms Used

The algorithms used were Naive Bayes, Decision Tree, and an Artificial Neural Network (ANN) using feedforward propagation and backpropagation for learning.

Programs Used

The author programmed two Python programs which run the Decision Tree and Naive Bayes algorithms. These programs accept training data and testing data and will then implement the training and testing phases of the algorithms. The programs are supervised learning algorithms and require the labels that identify the training and the testing data to appear in the data sets. The programs output the decision tree and naive Bayes data structures after the training phase and confusion matrices of correctly and incorrectly classified results after the testing phase.

The open source software Weka[6][7] was also used to collect data.

Procedure to collect data

Python program data collection

The Python algorithms for the decision tree and the naive Bayes algorithm were programmed by the author. The original datasets for the 10% testing and 10% training sets from the KDD Cup website are used in their original format for the Python algorithms. These datasets are found in the corrected.gz and kddcup.data_10_percent.gz in [52]. They were derived by a detailed analysis of useful attributes from the data in the MIT Lincoln Laboratory experiment[56]. The extraction of the attributes from this data is described in [62]. However, the authors do not appear to say how the 10 percent data is related to the original data. The original data is not used in the experiments in this thesis, however. Only the 10 percent data, and subsets of this data extracted using Weka[6][7], was used. These subsets preserve the relative percentages of the attack and normal classes. The decision tree algorithm was based closely on the algorithm for creating a decision tree found in [1]. The naive Bayes is also based closely on the description of the algorithm found in [1]. These algorithms do not have any options for the user to choose besides those relating to the training set and testing set to be used. They output a model after training is performed and a confusion matrix after testing is performed. They also time the training and testing phases.

Weka program data collection

For each machine learning algorithm, the algorithm was always trained with the training data, and testing was performed with either the testing data or the training data. In addition, the option to output detailed statistics was selected and to output the model. The model consist of all the information necessary to reproduce the trained machine learning data structure (e.g. the decision tree, naive bayes, or neural network trained on the 10% dataset).

The option -Xmx1024m was used to increase the memory available to the JRE to 1024 MB. Naive Bayes algorithm outputs the results of training and testing, as well as the model for the naive Bayes.

Besides the options mentioned above, the Weka naive Bayes and J48 decision tree algorithms were run using the defaults, and no other options were selected for them.

For the multilayer perceptron algorithm, several parameters could be chosen by the user. The leaning rate is .3, the momentum is .2, the nominal-to-binary filter option is True, the normalize attribute option is true, the normalize numeric class option is true, the number of hidden layers is 1, the training time (number of epochs) is 50, and the validation threshold is 20. These values are mostly the default values. However, the number of hidden layers and epochs has been decreased from the default in order to make the neural network run faster (since it is the slowest algorithm).

Later, to collect data on the testing times and to ensure that all testing times were from the same computer, the previously collected models from the 1%, 5%, 10%, 20%, 30%, 40%, 50% and 100% of the 10% 1999 UCI KDD training dataset were run on the full 10% 1999 UCI KDD testing dataset using a Windows batch file that also recorded timestamps before and after each testing phase was run so that the time taken to run each testing phase could be calculated.

Dataset Details

KDD

The datafiles used are from the University of California, Irvine Knowledge Discovery and Data Mining (UCI KDD) website[52]. The data files give the necessary information to create and train the algorithms.

The kddcup.names file lists the class types, including 'normal.', which signifies that no attack is in progress. The attack types are back, buffer_overflow, ftp_write, guess_passwd, imap, ipsweep, land, loadmodule, multihop, neptune, nmap, normal, perl, phf, pod, portsweep, rootkit, satan, smurf, spy, teardrop, warezclient, and warezmaster. The .names file also list the attribute names. Each attribute name states whether it is a continuous or symbolic variable. A symbolic variable has a finite number of possible values and can be completely enumerated. A continuous variable cannot be enumerated.

Each example uses forty-one attributes, and the testing data contains 23 different classes. The attributes can be divided into three types:

Table 3: Basic Features of individual TCP connections

feature name	description	type
duration	length (number of seconds) of the connection	continuous
protocol type	type of protocol, e.g. tcp, udp, etc.	symbolic
service	network service on the destination, e.g., http, telnet, etc.	symbolic
flag	normal or error status of the connection	symbolic
src_bytes	number of data bytes from source to destination	continuous
dst_bytes	number of data bytes from destination to source	continuous
land	1 if connection is from/to the same host/port; 0 otherwise	symbolic
wrong_fragment	number of "wrong" fragments	continuous
urgent	number of urgent packets	continuous

Table 4: Content features suggested by domain knowledge

feature name	description	type
hot	number of "hot" indicators	continuous
num_failed_logins	number of failed login attempts	continuous
logged_in	1 if successfully loggin in, 0 otherwise	discrete
num_compromised	number of "compromised" conditions	continuous
root_shell	1 if root shell obtained, 0 otherwise	symbolic
su_attempted	1 if "su root" command attempted, 0 otherwise	symbolic
num_root	number of "root" accesses	continuous
num_file_creations	number file creation operations	continuous
num_shells	number of shell prompts	continuous
num_access_files	number of operations on access control files	continuous
num_outbound_cmds	number of outbound commands in an ftp session	continuous
is_hot_login	1 if the login belongs to the "hot" list, 0 otherwise	symbolic
is_guest login	1 if the login is a "guest" login, 0 otherwise	symbolic

Table 5: A two-second window where various traffic features were computed

feature name	description	type
count	# of connections to the same host as this one in the past two seconds	continuous
	Note: the following features refer to these same-host connections	
serror_rate	% of connections that have "SYN" errors	continuous
rerror_rate	% of connections that have "REJ" errors	continuous
same_srv_rate	% of connections to the same service	continuous
diff_srv_rate	% connections to different services	continuous
srv_count	# of connections to the same service as this one in the past two seconds	continuous
	Note: The following features refer to these same-service connections.	
srv_serror_rate	% of connections that have "SYN" errors	continuous
srv_rerror_rate	% of connections that have "REJ" errors	continuous
srv_diff_host_rate	% of connections to different hosts	continuous

The kddcup.data file lists the value of the class and the value of the attributes. It should be noted that neither [52] nor [62], the main references in this thesis for this dataset, mention what a "hot" indicator is.

The testing data for the 10 percent data set contains 311,029 examples. These examples contain 60,593 normal items and 250,436 attacks. Therefore, this data is most likely atypical because it contains more attacks than normal data. The attack connections make up 80.52% of the dataset.

The training dataset contains 494,020 items. There are 97,277 normal connections and 39,6743 attack connections. The attacks make up 80.31% of the dataset.

For the Weka algorithms, the dataset was converted to arff format, which is a standard data mining format used by Weka. This was accomplished by first converting the file to csv format using Cream, a version of Vim[63]. The CSV format is a standard comma-separated format[7][6]. The version of the csv format read by WEKA has a row of entries at the top that lists the name of each attribute[7]. In this step, a line with the feature names as given above was added, along with the classification name. Then the lines were processed using Cream to contain only the classes 'normal' and 'attack' (i.e. any class that wasn't normal was changed to read 'attack'). Next, the Weka CSVLoader was used to convert both the training and the testing dataset (now in csv format) to arff format. The two arff headers were manually compared

and merged to form a single arff header with all the possible attribute values from both the testing and the training phase (Weka requires that the arff headers for the testing and training data match).

Subsets of the 10% dataset

Using the Weka resampling filter, several subsets of the 10% dataset were derived. These subsets make up 1%, 5%, 10%, 20%, 30%, 40%, and 50% of size of the 10% dataset found on the KDD Cup website. They were constructed in such a way that they preserve the relative percentages of attack and normal data from the original 10% dataset.

CHAPTER IV

RESULTS

In this section, a novel method for evaluating the results is introduced. This method takes into account the relative sizes of the classes to each other in the dataset. This allows the user of the IDS to evaluate how well it will predict the classes given the distribution of the dataset. In Figure 4 and Table 6 below, $N_K$represents the set of those examples that are known to be normal, A_K, represents those that are known to be attacks, N_C represents those that are classified as normal, and A_C represents those that are classified as attacks. In the Venn diagram below, the four possible situations are shown. Normal connections may be classified as an attack or may be classified as normal and attacks may be classified as an attack or as normal data. Usually when the rates in a confusion matrix are calculated, they are calculated out of the total of the class by definition, as shown in the confusion matrix on the left below. In the method presented in this paper, they are calculated out of the total classified as normal or classified as attacks. This makes it easy to see the percentages from the IDS user's point of view, who will not know the actual nature of the data ahead of time, but will know the classification given by the IDS.

Table 6: Two different perspectives

		classified as				classified as	
		normal	attack			normal(N_C)	attack(A_C)
known as	normal (N_K)	$\frac{N_K \Lambda N_C}{N_K}$	$\frac{N_K \Lambda A_C}{N_K}$	known as	normal	$\frac{N_K \Lambda N_C}{N_C}$	$\frac{N_K \Lambda A_C}{A_C}$
	attack (A_K)	$\frac{A_K \Lambda N_C}{A_K}$	$\frac{A_K \Lambda A_C}{A_K}$		attack	$\frac{A_K \Lambda N_C}{N_C}$	$\frac{A_K \Lambda A_C}{A_C}$

Running Time Performance

The running time of the algorithms was obtained by running the programs on an eMachines computer with a 2.93 GHz Pentium 4 processor and with 512 Mb DDR SDRAM.

Performance of the Python Algorithms

The testing data for the 10 percent data set contains 311,029 examples. These examples contain 60,593 normal items and 250,436 attacks. Therefore, this data is most likely atypical because it contains more attacks than normal data. The training dataset contains 494020 items. There are 97277 normal connections

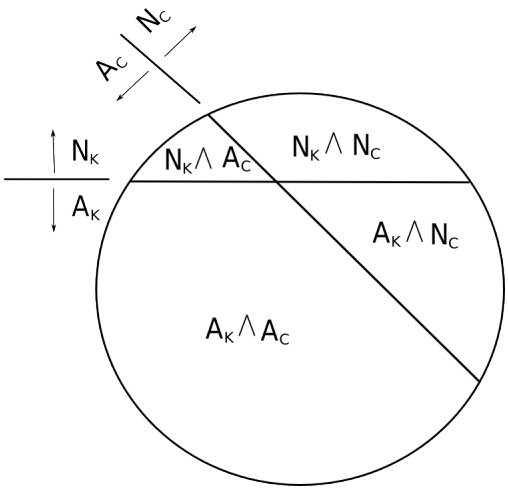

Figure 4: Venn Diagram of A_K,N_K,A_C, and N_C

and 396743 attack connections. For the 10 percent kddcup dataset, the naive Bayes ran faster than the decision tree. The total running time on this data of the training phase for the naive Bayes algorithm was 8.27 hours. The total time taken to create the naive Bayes data structure was 5.59 hours. The total running time for the training phase of the decision tree was 10.8 hours. The total time taken to create the decision tree data structure was 8.19 hours. These tests were run on an eMachines computer with a 2.93 GHz Pentium 4 processor and with 512 Mb DDR SDRAM.

The testing phase took less time to run. When the algorithm was run on the testing data for the 10% 1999 UCI KDD dataset, this phase took 1.73 hours for the naive Bayes and .77 hours for the decision tree.

Weka Algorithms

For the Weka algorithms, the training times for the three algorithms are shown in Figure 5. The left graph shows all of the points graphed, whereas the graph on the right shows a zoomed in portion of the bottom of the graph to better illustrate the differences between the Naive Bayes and J48 Decision Tree algorithm.

As the graph demonstrates, the training time for the multilayer perceptron grows very quickly with the size of the data compared to the other two algorithms. In addition, the Weka algorithms for the naive bayes and decision tree are much faster than the Python algorithms used. While the reasons for this are presently unknown, it may be due to algorithmic differences in the way the datasets are processed. Of the Weka algorithms, the multilayer perceptron algorithm takes the longest time to train among all of the dataset sizes tested, taking 11.68 hours to complete for the 50% dataset. The naive Bayes is the fastest of the three datasets on all of the data sizes, and the J48 is the second fastest on all data sizes. The naive Bayes completes

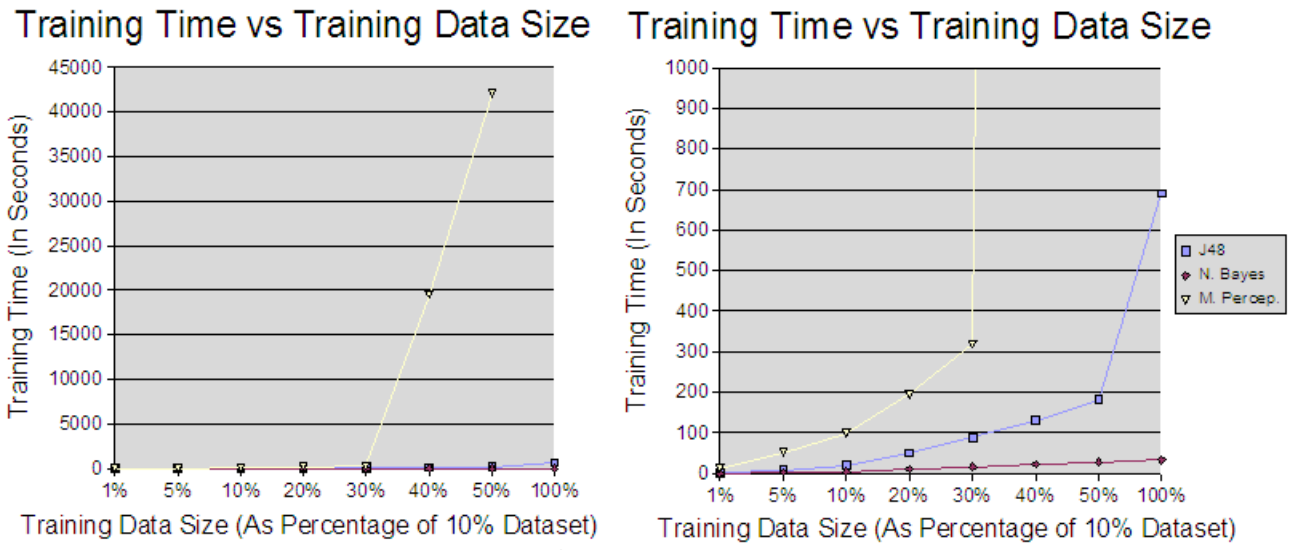

Figure 5: Training Time vs Training Data Size (left) and magnification of bottom of left figure (right)

the training phase in at most 32.2 seconds, while the J48 decision tree completes the training phase in at most 11 minutes and 30.74 seconds.

The testing time for the algorithms was considerably shorter than the training time. The testing time is the time for the full 10% dataset. This also changed depending on the training set, though the size of the testing data was kept constant. This is due to the training set affecting the model used. The change in testing time is especially large for the multilayer perceptron algorithm, taking over 14 minutes and 11 seconds to complete with the 50% dataset. While the exact causes of this result are uncertain, it has been observed that the file size of the multilayer perceptron model (which is in binary format) grows as the size of the training data grows. It is possible that it keeps some information that would be proportional to the number of examples in the training data, such as information about backpropagation weight changes for each item. In [61], the author mentions that weight changes might be collected and then added up later. The testing time for the J48 Decision Tree and the Naive Bayes algorithm was nearly constant, however. Between the 5% and 100% of the 10% training dataset, the J48 decision tree does not vary by more than .25 seconds, while the Naive Bayes does not vary by more than .52 seconds. Both of these algorithms take the longest to complete the testing phase when trained on 1% of the 10% 1999 UCI KDD dataset. The naive Bayes takes at most 52.97 seconds to complete the testing phase while the J48 algorithm takes at most 17.53 seconds. The Naive Bayes goes down to 39.35 seconds at the 5% of the 10% dataset while the J48 Decision Tree goes down to 10.79 seconds at the 5% of the 10% dataset.

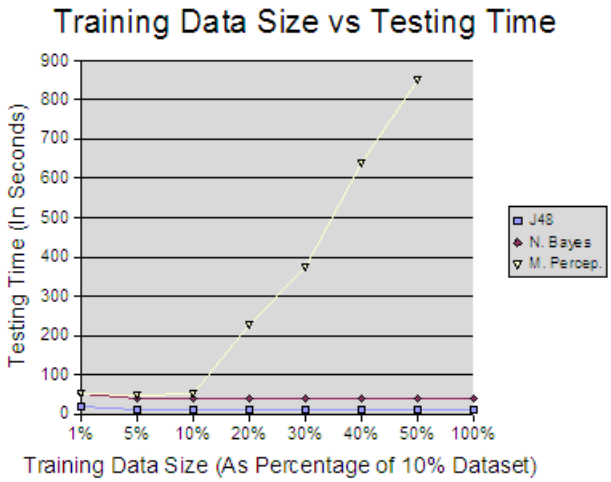

Figure 6: Training Data Size vs Testing Time

Classification Evaluation

The experiment results in this section show the differences in false positives and false negatives in the data. A false positive occurs when the the IDS flags a connection as an attack when it is not. A false negative occurs when the IDS flags a connection as normal when it is actually an attack. The accuracy rates for each algorithm are also recorded below. The accuracy rate is the percentage of the connections that are correctly classified over the total number of connections.

The decision tree and naive Bayes were trained on the KDD 10 percent training data and tested on the KDD 10 percent testing data of the KDD data set and results were obtained. The results are listed in the confusion matrices below (rows are actual value and columns are predicted value). The confusion matrices are listed as follows First, the rates for true negatives, false negatives, false positives, and true negatives are given out of the total number of normal connections by definition or the total number of attacks by definition. Lastly, the rates for true negatives, false negatives, true positives, and true negatives are given out of the total number of connections classified as normal or the total number of connections classified as attack connections. This is done for both the training and testing phases. The probabilities p(c|k) and p(k|c) are defined as the probability that the connection will be classified as the given classification by the IDS given the known classification (p(c|k)) and the probability that the known classification is a given class given that the classification by the IDS is known (p(k|c)).

These results are similar to the results found in [64]. The Percent Correct Classification (PCC) values (i.e. the accuracy) are slightly better than the values given in [64], but the difference in the percentage values is less than 1%.

When the Decision Tree and Naive Bayes were both trained and tested on the 10 percent training data, the results were as listed in Table 7.

Table 7: Python Decision Tree vs Python Naive Bayes

		N. Bayes		D. Tree	
Eval on Testing Set		p(c|k)		p(c|k)	
		Normal (84374)	Attack(226655)	Normal(83603)	Attack(227426)
	Normal (60593)	98.2	1.8	99.35	0.65
	Attack (250436)	9.93	90.07	9.34	90.66
		p(k|c)		p(k|c)	
		Normal(84374)	Attack(226655)	Normal(83603)	Attack(227426)
	Normal (60593)	70.52	0.48	72.01	0.17
	Attack (250436)	29.48	99.52	27.99	99.83
Eval on Training Set		p(c|k)		p(c|k)	
		Normal(101883)	Attack(392137)	Normal(100879)	Attack(393141)
	Normal(97277)	99.31	0.69	99.15	0.85
	Attack(396743)	1.33	98.67	1.12	98.88
		p(k|c)		p(k|c)	
		Normal(101883)	Attack(392137)	Normal(100879)	Attack(393141)
	Normal(97277)	94.82	0.17	95.61	0.21
	Attack(396743)	5.18	99.83	4.39	99.79

The accuracy of the algorithms when trained on the 10% KDD Cup training data is given in Figure 7.

The bar graph shows that the accuracy was noticeably higher when the algorithm was tested on the training data than when it was tested on the testing data.

False alarm rates for KDD 10 percent training data used for training and KDD 10 percent testing used for testing

The false alarm rates for naive Bayes and the decision tree are 1.8% and .65%, respectively.

These values may seem fairly low, but there are several factors to consider that may make expending effort to lower these quantities desirable. Firstly, it is important to look at the ratio of the number of false alarms to the total number of alarms (both false and real). The importance of this ratio is that it indicates

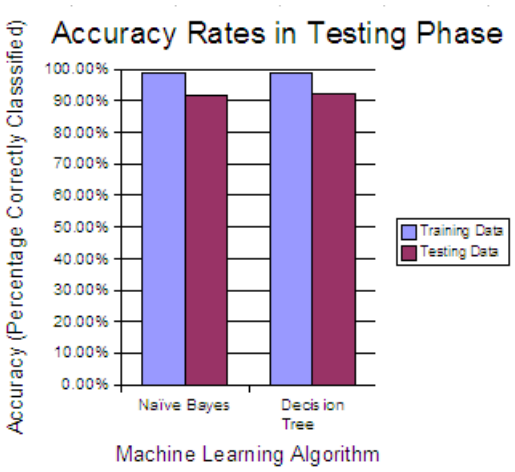

Figure 7: Accuracy rates on training and testing data

whether or not the IDS is "crying wolf" all the time. In the naive Bayes, this ratio is .48%. In the decision tree, this ratio is .17%. These are even smaller, but not as small as the .001% false positive rate, which is the amount that [21] recommends to have a system with a tolerable number of false alarms. Also, the data in this data set consists largely of attacks. Therefore, the math discussed in [21] works in favor of the IDS in these circumstances. That is, since there are a lot of attacks compared to normal activity (in fact there are more attacks than normal activity), and most of these are detected, the number of false alarms compared to the number of actual attacks is likely to be small anyway. In a real system, the number of attacks packets is likely to be low compared to the number of normal packets, making even a small percentage of falsely classified normal packets large compared to the number of actual attacks.

However, the nature of the machine learning algorithm may work in our favor if it can be given a method to incorporate new data that it sees in real-time. This is because the more recent data that the algorithm is able to process (assuming it can be correctly identified), the more it becomes able to identify new instances of normal behavior and new attacks (in general). Because it would likely be impossible to know with 100% certainty whether the data identified in real-time has been identified correctly, a certain amount of misclassification must be tolerated. However, the algorithm must be able to calculate the probability that it has identified an attack correctly in order make this misclassification manageable.

Accuracy and False Alarms

As seen above, the percentage of false positives compared to the total number of alarms is very low. However, the number of attacks not detected is very high. The percentage of data flagged as normal that is actually an attack (out of all the data flagged as normal) is 29% by the naive Bayes and 28% for the decision tree algorithm. Clearly, this number is very high. The accuracy in this instance is high (which is the same

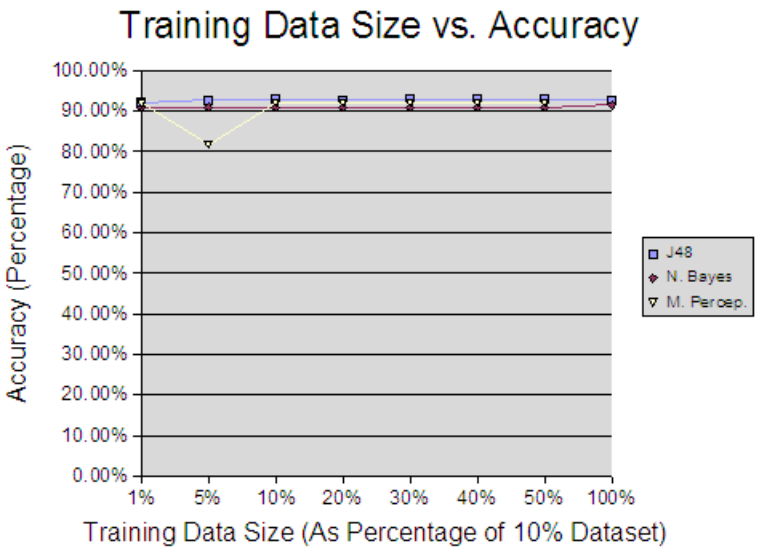

Figure 8: Training Data Size vs. Accuracy

as the PCC values above). However, the percentage of the non-alarm connections that are false negatives would be fairly high. These results indicate that the algorithm may have been overtrained.

Experiments using Weka

Experiments were also run using the Weka software[7][6], version 3.4.8. Three of the standard classification algorithms included in the Weka software package were used. These algorithms were the Naive Bayes, Decision Tree, and Multilayer Perceptron. The Multilayer Perceptron has A+1 input neurons (where A is the number of attributes, and the extra neuron is a threshold neuron), C output neurons (where C is the number of classes), and a user-specified number of hidden neurons (In the experiments in this section, this number was one). The testing phase of these experiments was run using the 10% UCI KDD testing dataset found in [52]. Training was done using 1%, 5%, 10%, 20%, 30%, 40%, 50%, and 100% subsets of the 10% training data subset. The results are shown in the graph and two tables below. The graph presents a comparison of the accuracy of each algorithm when trained on the different datasets. The tables present the confusion matrices for the three Weka algorithms on each of the datasets. For each confusion matrix, the columns represent whether the connections were classified as normal or attack and the rows represent the actual class of the data.

As the graph above demonstrates, the accuracy of the three algorithms changes very little over the datasets tested, with the exception of the noticeable decrease in performance of the multilayer perceptron algorithm when the 5% of 10% dataset is used. It is unknown why this decrease in performance occurs.

Possibly some quality of the dataset has affected the algorithm (while all datasets preserve the qualities of the class values-that is, relative percentages of attack and normal data in the full 10% dataset-they do not necessarily preserve the qualities of the attribute values). However, if this is the case, then this difference does not seem to affect the other algorithms, indicating that they can handle the difference in the dataset better.

Table IV represents the classification abilities of the Naive Bayes, Decision Tree, and Multilayer Perceptron when they are trained on 1%, 5%,10%, 20%,30%, 40%, and 50% of the 10% 1999 UCI KDD CUP training data. All trained algorithms below were tested on the full 10% 1999 UCI KDD CUP testing data. The numbers in the 'Normal' and 'Attack' columns are the total number of connections classified as normal and attack, respectively. The total number of known normal items is 60,593 and the total number of known attacks is 250,436 in the 10% 1999 UCI KDD CUP training data.

Table 8: Attacker's perspective of WEKA classifications

	Naïve Bayes		Decision Tree		Multilayer Perceptron	
	Trained on 1% of 10%		Trained on 1% of 10%		Trained on 1% of 10%	
	Normal (85120)	Attack (225909)	Normal (81711)	Attack (229318)	Normal (83327)	Attack (227702)
Normal	96.95	3.05	97.07	2.93	97.9	2.1
Attack	10.53	89.47	9.14	90.86	9.59	90.41
	Trained on 5% of 10%		Trained on 5% of 10%		Trained on 5% of 10%	
	Normal (85884)	Attack (225145)	Normal (82726)	Attack (228303)	Normal (115783)	Attack (195246)
Normal	97.53	2.47	99.4	0.6	98.58	1.42
Attack	10.7	89.3	8.98	91.02	22.38	77.62
	Trained on 10% of 10%		Trained on 10% of 10%		Trained on 10% of 10%	
	Normal (85812)	Attack (225217)	Normal (81834)	Attack (229195)	Normal (83502)	Attack (227527)
Normal	97.53	2.47	99.45	0.55	98.3	1.7
Attack	10.67	89.33	8.62	91.38	9.56	90.44
	Trained on 20% of 10%		Trained on 20% of 10%		Trained on 20% of 10%	
	Normal (85933)	Attack (225096)	Normal (82663)	Attack (228366)	Normal (82906)	Attack (228123)

Normal	97.61	2.39	99.51	0.49	97.87	2.13
Attack	10.7	89.3	8.93	91.07	9.42	90.58
	Trained on 30% of 10%		Trained on 30% of 10%		Trained on 30% of 10%	
	Normal (85990)	Attack (225039)	Normal (82136)	Attack (228893)	Normal (83688)	Attack (227341)
Normal	97.64	2.36	99.54	0.46	98.46	1.54
Attack	10.71	89.29	8.71	91.29	9.59	90.41
	Trained on 40% of 10%		Trained on 40% of 10%		Trained on 40% of 10%	
	Normal (85992)	Attack (225037)	Normal (82065)	Attack (228964)	Normal (83833)	Attack (227196)
Normal	97.63	2.37	99.47	0.53	98.46	1.54
Attack	10.71	89.29	8.7	91.3	9.65	90.35
	Trained in 50% of 10%		Trained on 50% of 10%		Trained on 50% of 10%	
	Normal (85992)	Attack (225037)	Normal (82132)	Attack (228896)	Normal (83507)	Attack (227196)
Normal	97.64	2.36	99.48	0.52	98.41	1.59
Attack	10.71	89.29	8.73	91.27	9.53	90.47
	Trained on 100% of 10%		Trained on 100% of 10%		Trained on 100% of 10%	
	Normal (84250)	Attack (226779)	Normal (82825)	Attack (228204)	Normal (–)	Attack (–)
Normal	97.62	2.38	99.44	0.56	–	–
Attack	10.02	89.98	9.01	90.99	–	–

In the table above, we can make several observations:

1. The performance of the machine learning algorithms does not change much between 1% and 100% of the 10% dataset. One exception to this is the performance of the multilayer perceptron on the 5% of 10% dataset. The performance using this algorithm decreases by 12.79 percentage points before going back up to around 90%.

2. The three machine learning algorithms have very similar performance overall when compared to each other.

3. The performance of the three machine learning algorithms in correctly classifying the normal and attack data is very high (except for the previously mentioned deviation from this trend). The classification of the normal data is approximately between 97% and 99% for the normal data and between 89% and 91% for the attack data. The performance of the algorithms is significantly better at the classification normal data than at the classification of attack data. It is stated in [52] that some attacks in the testing data are not found in the training data. This may be the cause of the difference in performance.

In the next table, we will examine the data from a different perspective. In this table, we will examine the items in the confusion matrices (TN, FP, FN, TP) as divided by the total items in their classification. This perspective can help the IDS user discover the percentage of alarms that are false positives and the percentage of non-alarms that are false negatives. This data is important to the IDS user. An IDS does not have the same perspective as the attacker. The IDS user is concerned about the number of false alarms out of the total alarms, whereas the attacker is concerned about the total number of false alarms out of the total number of attacks. As can be seen by comparing the Table and Table 9 below, these two totals may differ.

Table 9: User's Perspective of WEKA classifications

	Naïve Bayes		Decision Tree		Multilayer Perceptron	
	Trained on 1% of 10%		Trained on 1% of 10%		Trained on 1% of 10%	
	Normal (85120)	Attack (225909)	Normal (81711)	Attack (229318)	Normal (83327)	Attack (227702)
Normal	69.01	0.82	71.98	0.78	71.19	0.56
Attack	30.99	99.18	28.02	99.22	28.81	99.44
	Trained on 5% of 10%		Trained on 5% of 10%		Trained on 5% of 10%	
	Normal (85884)	Attack (225145)	Normal (82726)	Attack (228303)	Normal (115783)	Attack (195246)
Normal	68.81	0.66	72.81	0.16	51.59	0.44
Attack	31.19	99.34	27.19	99.84	48.41	99.56
	Trained on 10% of 10%		Trained on 10% of 10%		Trained on 10% of 10%	
	Normal (85812)	Attack (225217)	Normal (81834)	Attack (229195)	Normal (83502)	Attack (227527)

Normal	68.87	0.66	73.63	0.15	71.33	0.45
Attack	31.13	99.34	26.37	99.85	28.67	99.55
	Trained on 20% of 10%		Trained on 20% of 10%		Trained on 20% of 10%	
	Normal (85933)	Attack (225096)	Normal (82663)	Attack (228366)	Normal (82906)	Attack (228123)
Normal	68.83	0.64	72.94	0.13	71.53	0.57
Attack	31.17	99.36	27.06	99.87	28.47	99.43
	Trained on 30% of 10%		Trained on 30% of 10%		Trained on 30% of 10%	
	Normal (85990)	Attack (225039)	Normal (82136)	Attack (228893)	Normal (83688)	Attack (227341)
Normal	68.8	0.64	73.43	0.12	71.29	0.41
Attack	31.2	99.36	26.57	99.88	28.71	99.59
	Trained on 40% of 10%		Trained on 40% of 10%		Trained on 40% of 10%	
	Normal (85992)	Attack (225037)	Normal (82065)	Attack (228964)	Normal (83833)	Attack (227196)
Normal	68.8	0.64	73.45	0.14	71.16	0.41
Attack	31.2	99.36	26.55	99.86	28.84	99.59
	Trained in 50% of 10%		Trained on 50% of 10%		Trained on 50% of 10%	
	Normal (85992)	Attack (225037)	Normal (82132)	Attack (228896)	Normal (83507)	Attack (227196)
Normal	68.8	0.64	73.39	0.14	71.4	0.42
Attack	31.2	99.36	26.61	99.86	28.6	99.58
	Trained on 100% of 10%		Trained on 100% of 10%		Trained on 100% of 10%	
	Normal (84250)	Attack (226779)	Normal (82825)	Attack (228204)	Normal (–)	Attack (–)
Normal	70.21	0.64	72.75	0.15	–	–
Attack	29.79	99.36	27.25	99.85	–	–

From the Table 9, we get a different picture of the classification. Though the differences between the algorithms are still similar when compared to each other and when compared over the different training data,

the data classified as normal shows an unacceptably large percentage of false negatives. This new method of classification indicates how the distribution of data used affects the classification abilities of the IDS from the perspective of the IDS user.

Additional Weka Experiments

In the previous section, Weka experiments using the training and testing set available from the UCI KDD website were used. These can be easily compared to previous experiments. However, they are only from one randomized distribution of the dataset. In this section, a different dataset, as well as subsets of this new dataset, will be explored. The new datasets for testing and training were derived by first combining the original 10% training and 10% testing datasets from the UCI KDD files into a single large file. Then the datasets used in this section were derived using the Weka StratifiedRemoveFolds filter. The StratifiedRemoveFolds filter divides the input file into n equally sized partitions, called folds, which each have some arrangement of examples that preserve the class distribution in the original file but are otherwise randomly arranged[7][6]. The options for this filter include the random seed, the number of folds, and the particular fold extracted. The random seed was set to 0-9. The number of folds was set to 2 and the particular fold extracted was set to 1 or 2.

In the next step, subsets were extracted from each of these 10 new training datasets using the Weka resampling filter that was used in the previous experiments. These subsets were 1%,5%,10%,20%,30%,40%, and 50% of the randomized training datasets. The new training datasets have 402,524 examples in total. The dataset has 78,935 normal connections and 323,589 attack connections. This means that the data contains a total of 80.39% attack connections.

In Tables 10, 11, 12, and 13 in this section and the next section, the means and standard deviations for each training set was calculated for the three algorithms: Naive Bayes, J48 Decision Tree, and Multilayer Perceptron. The equation used for the standard deviation was

$s_{N-1} = \sqrt{(\frac{1}{N-1})\Sigma_{i=1}^{N}(x^i - \overline{x})^2}$.

This equation is also known as the square root of the bias-corrected variance[65].

In Table 10, the data is from the perspective of the attacker.

Table 10: Attacker's perspective on new dataset classifications

	Naïve Bayes	Decision Tree	Multilayer Perceptron
	Trained on 1% of rand. train.	Trained on 1% of rand. train.	Trained on 1% of of rand. train.

	Normal (92345.3)	Attack (310178.7)	Normal (81063.3)	Attack (321460.7)	Normal (88586.8)	Attack (313937.2)
Normal	mean: 97.66 std dev:0.89	mean: 2.34 std dev:0.89	mean:97.1 std dev:1.26	mean:2.9 std dev:1.26	mean:99.09 std dev:0.41	mean:0.91 std dev:0.41
Attack	mean: 4.72 std dev:0.09	mean:95.28 std dev:0.09	mean:1.37 std dev:0.37	mean:98.63 std dev:0.37	mean:3.21 std dev:0.59	mean:96.79 std dev:0.59

	Trained on 5% of rand. train.		Trained on 5% of rand. train.		Trained on 5% of rand. train.	
	Normal (92483.6)	Attack (310040)	Normal (79425.5)	Attack (323098.5)	Normal (87737.3)	Attack (314786.7)
Normal	mean: 98.2 std dev:0.13	mean: 1.8 std dev:0.13	mean:97.52 std dev:0.61	mean:2.48 std dev:0.61	mean:99.23 std dev:0.56	mean:0.77 std dev:0.56
Attack	mean: 4.63 std dev:0.19	mean: 95.37 std dev:0.19	mean:0.76 std dev:0.15	mean:99.24 std dev:0.15	mean:2.91 std dev:0.42	mean:97.09 std dev:0.42

	Trained on 10% of rand. train.		Trained on 10% of rand. train.		Trained on 10% of rand. train.	
	Normal (92840.7)	Attack (309683.3)	Normal (79760.1)	Attack (322763.9)	Normal (88226.5)	Attack (314297.5)
Normal	mean: 98.32 std dev:0.09	mean:1.68 std dev:0.09	mean:98 std dev:0.53	mean:2 std dev:0.53	mean:99.36 std dev:0.29	mean:0.64 std dev:0.29
Attack	mean:4.71 std dev:0.15	mean:95.27 std dev:0.15	mean:0.74 std dev:0.15	mean:99.26 std dev:0.15	mean:3.03 std dev:0.52	mean:96.97 std dev:0.52

	Trained on 20% of rand. train.		Trained on 20% of rand. train.		Trained on 20% of rand. train.	
	Normal (92955.7)	Attack (309568.3)	Normal (78941.5)	Attack (323582.5)	Normal (88484.6)	Attack (314039.4)
Normal	mean:98.38 std dev:0.09	mean:1.62 std dev:0.09	mean: 97.65 std dev:0.41	mean:2.35 std dev:0.41	mean:99.49 std dev:0.26	mean:0.51 std dev:0.26
Attack	mean:4.73 std dev:0.04	mean:95.27 std dev:0.04	mean:0.58 std dev:0.09	mean:99.42 std dev:0.09	mean:3.08 std dev:0.49	mean:96.92 std dev:0.49

	Trained on 30% of rand. train.		Trained on 30% of rand. train.		Trained on 30% of rand. train.	
	Normal (93066.2)	Attack (309457.8)	Normal (78951.4)	Attack (323572.6)	Normal (87860.5)	Attack (314663.5)

Normal	mean:98.42 std dev:0.08	mean:1.58 std dev:0.08	mean:97.7 std dev:0.46	mean:2.3 std dev:0.46	mean:99.35 std dev:0.39	mean:0.65 std dev:0.39
Attack	mean:4.75 std dev:0.05	mean:95.25 std dev:0.05	mean:0.57 std dev:0.11	mean:99.43 std dev:0.11	mean:2.92 std dev:0.47	mean:97.08 std dev:0.47
	Trained on 40% of rand. train.		Trained on 40% of rand. train.		Trained on 40% of rand. train.	
	Normal (93173.8)	Attack (309350.2)	Normal (78964.2)	Attack (323559.8)	Normal (87599.4)	Attack (314924.6)
Normal	mean:98.45 std dev:0.08	mean:1.55 std dev:0.08	mean:97.74 std dev:0.27	mean:2.26 std dev:0.27	mean:99.25 std dev:0.46	mean:0.75 std dev:0.46
Attack	mean:4.78 std dev:0.04	mean:95.22 std dev:0.04	mean:0.56 std dev:0.07	mean:99.44 std dev:0.07	mean:2.86 std dev:0.41	mean:97.14 std dev:0.41
	Trained in 50% of rand. train.		Trained on 50% of rand. train.		Trained on 50% of rand. train.	
	Normal (93215.7)	Attack (309308.3)	Normal (78803.8)	Attack (323720.2)	Normal (86894.9)	Attack (315629.1)
Normal	mean:98.48 std dev:0.07	mean:1.52 std dev:0.07	mean:97.69 std dev:0.1	mean:2.31 std dev:0.1	mean:99.33 std dev:0.19	mean:0.67 std dev:0.19
Attack	mean:4.79 std dev:0.05	mean:95.21 std dev:0.05	mean:0.52 std dev:0.02	mean:99.48 std dev:0.02	mean:2.62 std dev:0.33	mean:97.38 std dev:0.33
	Trained on 100% of rand. train.		Trained on 100% of rand. train.		Trained on 100% of rand. train.	
	Normal (93292.7)	Attack (309231.3)	Normal (78835.4)	Attack (323688.6)	Normal (–)	Attack (–)
Normal	mean: 98.51 std dev:0.03	mean:1.49 std dev:0.03	mean:97.8 std dev:0.06	mean:2.2 std dev:0.06	mean:– std dev:–	mean:– std dev:–
Attack	mean:4.8 std dev:0.04	mean:95.2 std dev:0.04	mean:0.51 std dev:0.02	mean:99.49 std dev:0.02	mean:– std dev:–	mean:– std dev:–

In Table 11, the data is from the perspective of the IDS user. There is an overall downward trend for the standard deviation for all of the machine learning algorithms to decrease as the percentage of the data used increases. The numbers suggest that these algorithms will exhibit more consistent performance on a larger dataset. The standard deviation for the Multilayer Perceptron normal connections are the highest

for all the datasets tested. Since the ZeroR (which simply classifies every instance as the majority class, which is in this case 'attack') algorithm was correct on an average of 80% of the attack cases and 0% of the normal cases, the algorithms below show an improvement from this base rate performance of 20%, 19%, and 20% for the attack cases and 83%, 98%, and 90% with normal cases in the 50% randomized training data. This indicates that these algorithms significantly outperform the base rate performance. In addition, this data indicates that the decision tree clearly has the best classification rate of the three algorithms on the datasets used. Table 11 shows that the J48 decision tree outperforms the other two algorithms on the 1%, 5%, 10%,20%, 30%, 40%, and 50% datasets and outperforms the naive bayes on the 100% datasets (the multilayer perceptron was not tested on the 100% datasets). In addition, the standard deviations are smaller than the differences in performance of the three algorithms, so this difference is statistically significant.

Table 11: User's perspective on new dataset classifications

	Naïve Bayes		Decision Tree		Multilayer Perceptron	
	Trained on 1% of rand. train.		Trained on 1% of rand. train.		Trained on 1% of of rand. train.	
	Normal (85120)	Attack (225909)	Normal (81711)	Attack (229318)	Normal (83327)	Attack (227702)
Normal	mean:83.47 std dev:0.21	mean:0.6 std dev:0.22	mean:94.55 std dev:1.34	mean:0.71 std dev:0.31	mean:88.29 std dev:1.82	mean:0.23 std dev:0.1
Attack	mean:16.53 std dev:0.21	mean:99.4 std dev:0.22	mean:5.45 std dev:1.34	mean:99.29 std dev:0.31	mean:11.71 std dev:1.82	mean:99.77 std dev:0.1
	Trained on 5% of rand. train.		Trained on 5% of rand. train.		Trained on 5% of of rand. train.	
	Normal (92483.6)	Attack (310040)	Normal (79425.5)	Attack (323098.5)	Normal (87737.3)	Attack (314786.7)
Normal	mean:83.82 std dev:0.57	mean:0.46 std dev:0.03	mean:96.92 std dev:0.59	mean:0.61 std dev:0.15	mean:89.28 std dev:1.33	mean:0.19 std dev:0.14
Attack	mean:16.18 std dev:0.57	mean:99.54 std dev:0.03	mean:3.08 std dev:0.59	mean:99.39 std dev:0.15	mean:10.72 std dev:1.33	mean:99.81 std dev:0.14
	Trained on 10% of rand. train.		Trained on 10% of rand. train.		Trained on 10% of of rand. train.	
	Normal (92840.7)	Attack (309683.3)	Normal (79760.1)	Attack (322763.9)	Normal (88226.5)	Attack (314297.5)
Normal	mean:83.59 std dev:0.45	mean:0.43 std dev:0.02	mean:96.99 std dev:0.58	mean:0.49 std dev:0.13	mean:88.9 std dev:1.66	mean:0.16 std dev:0.07

Attack	mean:16.41 std dev:0.45	mean:99.57 std dev:0.02	mean:3.01 std dev:0.58	mean:99.51 std dev:0.13	mean:11.1 std dev:1.66	mean:99.84 std dev:0.07
	Trained on 20% of rand. train.		Trained on 20% of rand. train.		Trained on 20% of of rand. train.	
	Normal (92955.7)	Attack (309568.3)	Normal (78941.5)	Attack (323582.5)	Normal (88484.6)	Attack (314039.4)
Normal	mean:83.54 std dev:0.12	mean:0.41 std dev:0.02	mean:97.64 std dev:0.37	mean:0.57 std dev:0.1	mean:88.75 std dev:1.57	mean:0.13 std dev:0.06
Attack	mean:16.46 std dev:0.12	mean:99.59 std dev:0.02	mean:2.36 std dev:0.37	mean:99.43 std dev:0.1	mean:11.25 std dev:1.57	mean:99.87 std dev:0.06
	Trained on 30% of rand. train.		Trained on 30% of rand. train.		Trained on 30% of of rand. train.	
	Normal (93066.2)	Attack (309457.8)	Normal (78951.4)	Attack (323572.6)	Normal (87860.5)	Attack (314663.5)
Normal	mean:83.48 std dev:0.14	mean:0.4 std dev:0.02	mean:97.68 std dev:0.42	mean:0.56 std dev:0.11	mean:89.26 std dev:1.49	mean:0.16 std dev:0.1
Attack	mean:16.52 std dev:0.14	mean:99.6 std dev:0.02	mean:2.32 std dev:0.42	mean:99.44 std dev:0.11	mean:10.74 std dev:1.49	mean:99.84 std dev:0.1
	Trained on 40% of rand. train.		Trained on 40% of rand. train.		Trained on 40% of of rand. train.	
	Normal (93173.8)	Attack (309350.2)	Normal (78964.2)	Attack (323559.8)	Normal (87599.4)	Attack (314924.6)
Normal	mean:83.4 std dev:0.12	mean:0.4 std dev:0.02	mean:97.71 std dev:0.27	mean:0.55 std dev:0.06	mean:89.44 std dev:1.32	mean:0.19 std dev:0.11
Attack	mean:16.6 std dev:0.12	mean:99.6 std dev:0.02	mean:2.29 std dev:0.27	mean:99.45 std dev:0.06	mean:10.56 std dev:1.32	mean:99.81 std dev:0.11
	Trained on 50% of rand. train.		Trained on 50% of rand. train.		Trained on 50% of of rand. train.	
	Normal (93215.7)	Attack (309308.3)	Normal (78803.8)	Attack (323720.2)	Normal (86894.9)	Attack (315629.1)
Normal	mean:83.39 std dev:0.13	mean:0.39 std dev:0.02	mean:97.86 std dev:0.07	mean:0.56 std dev:0.02	mean:90.23 std dev:1.09	mean:0.17 std dev:0.05
Attack	mean:16.61 std dev:0.13	mean:99.61 std dev:0.02	mean:2.14 std dev:0.07	mean:99.44 std dev:0.02	mean:9.77 std dev:1.09	mean:99.82 std dev:0.05
	Trained on 100% of rand. train.		Trained on 100% of rand. train.		Trained on 100% of of rand. train.	

	Normal (93292.7)	Attack (309231.3)	Normal (78835.4)	Attack (323688.6)	Normal (–)	Attack (–)
Normal	mean:83.35 std dev:0.12	mean:0.38 std dev:0.01	mean:97.92 std dev:0.08	mean:0.54 std dev:0.02	mean:– std dev:–	mean:– std dev:–
Attack	mean:16.65 std dev:0.12	mean:99.62 std dev:0.01	mean:2.08 std dev:0.08	mean:99.46 std dev:0.02	mean:– std dev:–	mean:– std dev:–

Neural Net Experiments With Hidden Layer Nodes

Experiments were done to determine how the number of hidden layers might affect the performance of the neural network. In these experiments, the 10 training files obtained for each of the 1%,5%,10%,20%,30%, and 40% datasets by using StratifiedRemoveFolds on the file combining the 10% training and testing set (as was used on the previous 2 tables above) were used. These experiments were done using one hidden layer with the following numbers of neurons: 2,3,4,5, and 6. The data was calculated from the attackers perspective and the user's perspective. Table 12 shows the data from the attacker's perspective.

Table 12: Increasing the number of neurons in the hidden layer of the multilayer perceptron (attacker's perspective)

	Hidden Layer Neurons: 2		Hidden Layer Neurons: 3		Hidden Layer Neurons: 4		Hidden Layer Neurons: 5		Hidden Layer Neurons: 6	
	Trained on 1% of rand. train.		Trained on 1% of rand. train.		Trained on 1% of of rand. train.		Trained on 1% of of rand. train.		Trained on 1% of of rand. train.	
	Normal (163652)	Attack (314705.17)	Normal (163162.89)	Attack (316137.83)	Normal (163125.56)	Attack (316140.17)	Normal (163008)	Attack (316776.33)	Normal (162867.33)	Attack (316322.33)
Normal	mean: 99.2 std dev: 0.47	mean: 0.8 std dev: 0.47	mean: 99.25 std dev:0.33	mean:0.75 std dev:0.33	mean:99.12 std dev: 0.4	mean:0.88 std dev: 0.4	mean: 99.1 std dev: 0.29	mean:0.9 std dev: 0.29	mean: 99.14 std dev: 0.39	mean: 0.86 std dev: 0.39
Attack	mean: 2.78 std dev:0.57	mean: 97.22 std dev: 0.57	mean:2.42 std dev:0.43	mean:97.58 std dev:0.43	mean:2.38 std dev: 0.42	mean: 97.62 std dev: 0.42	mean:2.33 std dev: 0.48	mean: 97.67 std dev: 0.48	mean: 2.31 std dev: 0.49	mean: 97.69 std dev: 0.49
	Trained on 5% of rand. train.		Trained on 5% of rand. train.		Trained on 5% of of rand. train.		Trained on 5% of of rand. train.		Trained on 5% of of rand. train.	
	Normal (160383.78)	Attack (321751.5)	Normal (160326.11)	Attack (322416.5)	Normal (162199.67)	Attack (317586.83)	Normal (160751.67)	Attack (322583.5)	Normal (161895.22)	Attack (318795)
Normal	mean:96.39 std dev:5.56	mean:3.61 std dev: 5.56	mean:96.84 std dev: 5.05	mean:3.16 std dev: 5.05	mean:99 std dev: 0.53	mean:1 std dev: 0.53	mean: 96.51 std dev: 5.48	mean: 3.49 std dev: 5.48	mean: 98.62 std dev: 1.99	mean: 1.38 std dev: 1.99
Attack	mean: 2.46 std dev: 1.18	mean: 97.54 std dev: 1.18	mean:1.87 std dev: 0.82	mean: 98.13 std dev: 0.82	mean:2.26 std dev: 0.26	mean:97.74 std dev: 0.26	mean: 2.09 std dev: 1.06	mean:97.91 std dev: 1.06	mean: 2.2 std dev: 0.7	mean: 97.8 std dev: 0.7

	Trained on 10% of rand. train.		Trained on 10% of rand. train.		Trained on 10% of of rand. train.		Trained on 10% of of rand. train.		Trained on 10% of of rand. train.	
	Normal (161095.78)	Attack (317872.33)	Normal (161384.89)	Attack (318883.67)	Normal (162227.44)	Attack (316982.5)	Normal (162734.89)	Attack (316381)	Normal (162509)	Attack (316730.83)
Normal	mean: 98.16 std dev: 4.28	mean: 1.84 std dev: 4.28	mean:97.88 std dev: 4.12	mean: 2.12 std dev: 4.12	mean:99.33 std dev: 0.3	mean:0.67 std dev: 0.3	mean:99.27 std dev: 0.38	mean:0.73 std dev: 0.38	mean: 99.07 std dev: 1.15	mean: 0.93 std dev: 1.15
Attack	mean: 2.64 std dev: 0.84	mean: 97.36 std dev: 0.84	mean:2.22 std dev: 0.6	mean:97.78 std dev: 0.6	mean:2.31 std dev: 0.36	mean: 97.69 std dev: 0.36	mean: 2.42 std dev: 0.39	mean:97.58 std dev: 0.39	mean: 2.4 std dev: 0.6	mean: 97.6 std dev: 0.6

	Trained on 20% of rand. train.		Trained on 20% of rand. train.		Trained on 20% of of rand. train.		Trained on 20% of of rand. train.		Trained on 20% of of rand. train.	
	Normal (162457.22)	Attack (319126)	Normal (163258.67)	Attack (317380.5)	Normal (163087.67)	Attack (316393.17)	Normal (162760.22)	Attack (316886.33)	Normal (162887.33)	Attack (316557.83)
Normal	mean: 96.74 std dev: 5.39	mean: 3.26 std dev: 5.39	mean:99.33 std dev: 0.42	mean:0.67 std dev: 0.42	mean:99.49 std dev: 0.21	mean: 0.51 std dev: 0.21	mean: 99.32 std dev: 0.21	mean: 0.68 std dev: 0.21	mean: 99.51 std dev: 0.26	mean: 0.49 std dev: 0.26
Attack	mean: 2.19 std dev: 0.94	mean: 97.81 std dev: 0.94	mean:2.47 std dev: 0.36	mean:97.53 std dev: 0.36	mean: 2.32 std dev: 0.25	mean: 97.68 std dev: 0.25	mean: 2.24 std dev: 0.09	mean: 97.76 std dev: 0.09	mean: 2.29 std dev: 0.08	mean: 97.71 std dev: 0.08

	Trained on 30% of rand. train.	Trained on 30% of rand. train.	Trained on 30% of of rand. train.	Trained on 30% of of rand. train.	Trained on 30% of of rand. train.

	Normal (159507.22)	Attack (321280.33)	Normal (162262.22)	Attack (317380.5)	Normal (162673.22)	Attack (316854.17)	Normal (163582.56)	Attack (317485.17)	Normal (164253.67)	Attack (316980.5)
Normal	mean: 96.72 std dev: 5.55	mean: 3.28 std dev: 5.55	mean:99.23 std dev: 0.49	mean:0.77 std dev: 0.49	mean:99.36 std dev: 0.28	mean: 0.64 std dev: 0.28	mean: 98.56 std dev: 2.59	mean: 1.44 std dev: 2.59	mean: 98.07 std dev: 4.19	mean: 1.93 std dev: 4.19
Attack	mean: 2.15 std dev: 0.91	mean: 97.85 std dev: 0.91	mean:2.25 std dev: 0.23	mean:97.75 std dev: 0.23	mean: 2.28 std dev: 0.15	mean: 97.72 std dev: 0.15	mean: 1.97 std dev: 0.51	mean: 98.03 std dev: 0.51	mean: 2.05 std dev: 0.54	mean: 97.95 std dev: 0.54

	Trained on 40% of rand. train.		Trained on 40% of rand. train.		Trained on 40% of of rand. train.		Trained on 40% of of rand. train.		Trained on 40% of of rand. train.	
	Normal (164109.56)	Attack (313863.33)	Normal (162670.78)	Attack (316263.33)	Normal (162801.44)	Attack (316554.5)	Normal (163293.11)	Attack (316001.5)	Normal (162188.89)	Attack (317164.5)
Normal	mean: 98.46 std dev: 3.83	mean: 1.54 std dev: 3.83	mean:99.26 std dev: 0.58	mean:0.74 std dev: 0.58	mean: 99.41 std dev: 0.27	mean: 0.59 std dev: 0.27	mean: 98.24 std dev: 3.99	mean: 1.76 std dev: 3.99	mean: 99.46 std dev: 0.13	mean: 0.54 std dev: 0.13
Attack	mean: 2.73 std dev: 0.87	mean: 97.27 std dev: 0.87	mean:2.39 std dev: 0.35	mean: 97.61 std dev: 0.35	mean: 2.24 std dev: 0.28	mean: 97.76 std dev: 0.28	mean: 2.23 std dev: 0.64	mean: 97.77 std dev: 0.64	mean: 2.25 std dev: 0.37	mean: 97.75 std dev: 0.37

The information in Table 12 can be evaluated by observing the changes in the means as the number of neuron in the hidden layer increases. From this data, it can be seen that the best classifications of normal data are at 5,3,3,2,5 and 5 neurons at 1%,5%,10%,20%, 30%, and 40% respectively. The best overall performance is 98.13% and is obtained using 3 neurons with the 5% training data. This data suggests that only a small percentage of the training set is necessary for good performance. However, the standard deviations for the best and worst classifications are larger than the difference between the best and worst classifications. Therefore, the data cannot conclusively suggest an optimum number of neurons. In addition, the results obtained using 1 neuron are similar to the results obtained using 2,3,4,5, or 6 neurons, and are often within one standard deviation of each other.

In Table 13 the data is given from the user's perspective. This table shows that the user recieves the best classifications of normal data at 3,3,3,2,5, and 5 neurons at 1%,5%,10%,20%,30%, and 40% of the training data respectively. This information suggests that 3 neurons is the best number of neurons to use for 30% of the training data or less. The data also suggests that 5 neurons gives the best classifications for datasets of 10% of the training data or less. In addition, the best overall performance for normal data classification (a correct classification of 92.87%) is obtained using 3 neurons at 1% of the training data. This information suggests that using a small percentage of the training data with three neurons gives the best performance for the multilayer perceptron. However, the standard deviations for the best and worst values are again larger than the difference between the best and worst classifications. This perspective also shows little difference between using 1 neuron versus 2-6 neurons, with results often close to or within one standard deviation of each other.

Table 13: Increasing the number of neurons in the hidden layer of the multilayer perceptron (user's perspective

	Hidden Layer Neurons: 2		Hidden Layer Neurons: 3		Hidden Layer Neurons: 4		Hidden Layer Neurons: 5		Hidden Layer Neurons: 6	
	Trained on 1% of rand. train.		Trained on 1% of rand. train.		Trained on 1% of of rand. train.		Trained on 1% of of rand. train.		Trained on 1% of of rand. train.	
	Normal (163652)	Attack (314705.17)	Normal (163162.89)	Attack (316137.83)	Normal (163125.56)	Attack (316140.17)	Normal (163008)	Attack (316776.33)	Normal (162867.33)	Attack (316322.33)
Normal	mean: 89.73 std dev: 1.84	mean: 0.2 std dev: 0.12	mean: 92.87 std dev: 1.42	mean: 0.76 std dev: 0.08	mean: 91.05 std dev:1.4	mean: 0.22 std dev: 0.1	mean: 91.25 std dev: 1.63	mean: 0.23 std dev: 0.07	mean: 91.31 std dev: 1.62	mean: 0.21 std dev: 0.1
Attack	mean: 10.27 std dev: 1.84	mean: 99.8 std dev: 0.12	mean: 7.13 std dev: 1.42	mean: 99.24 std dev: 0.08	mean: 8.95 std dev: 1.4	mean: 99.78 std dev: 0.1	mean: 8.75 std dev: 1.63	mean: 99.77 std dev: 0.07	mean: 8.69 std dev: 1.62	mean: 99.79 std dev: 0.1
	Trained on 5% of rand. train.		Trained on 5% of rand. train.		Trained on 5% of of rand. train.		Trained on 5% of of rand. train.		Trained on 5% of of rand. train.	
	Normal (160383.78)	Attack (321751.5)	Normal (160326.11)	Attack (322416.5)	Normal (162199.67)	Attack (317586.83)	Normal (160751.67)	Attack (322583.5)	Normal (161895.22)	Attack (318795)
Normal	mean: 90.88 std dev: 4.09	mean: 0.87 std dev: 1.31	mean: 92.87 std dev: 2.94	mean: 0.76 std dev: 1.19	mean: 91.47 std dev: 0.91	mean: 0.25 std dev: 0.13	mean: 92.13 std dev: 3.65	mean: 0.83 std dev: 1.29	mean: 91.72 std dev: 2.35	mean: 0.34 std dev: 0.48
Attack	mean: 9.12 std dev: 4.09	mean: 99.13 std dev: 1.31	mean: 7.13 std dev: 2.94	mean: 99.24 std dev: 1.19	mean: 8.53 std dev: 0.91	mean: 99.75 std dev: 0.13	mean: 7.87 std dev: 3.65	mean: 99.17 std dev: 1.29	mean: 8.28 std dev: 2.35	mean: 99.66 std dev: 0.48

	Trained on 10% of rand. train.		Trained on 10% of rand. train.		Trained on 10% of of rand. train.		Trained on 10% of of rand. train.		Trained on 10% of of rand. train.	
	Normal (161095.78)	Attack (317872.33)	Normal (161384.89)	Attack (318883.67)	Normal (162227.44)	Attack (316982.5)	Normal (162734.89)	Attack (316381)	Normal (162509)	Attack (316730.83)
Normal	mean: 90.24 std dev: 2.85	mean: 0.44 std dev: 1.01	mean: 91.62 std dev: 2.04	mean: 0.51 std dev: 0.97	mean: 91.31 std dev:1.23	mean: 0.17 std dev: 0.07	mean: 90.92 std dev: 1.3	mean: 0.18 std dev: 0.09	mean: 91.02 std dev: 1.93	mean: 0.23 std dev: 0.28
Attack	mean: 9.76 std dev: 2.85	mean: 99.56 std dev: 1.01	mean: 8.38 std dev: 2.04	mean: 99.49 std dev: 0.97	mean: 8.69 std dev: 1.23	mean: 99.83 std dev: 0.07	mean: 7.87 std dev: 1.3	mean: 99.17 std dev: 0.09	mean: 8.98 std dev: 1.93	mean: 99.77 std dev: 0.28

	Trained on 20% of rand. train.		Trained on 20% of rand. train.		Trained on 20% of of rand. train.		Trained on 20% of of rand. train.		Trained on 20% of of rand. train.	
	Normal (162457.22)	Attack (319126)	Normal (163258.67)	Attack (317380.5)	Normal (163087.67)	Attack (316393.17)	Normal (162760.22)	Attack (316886.33)	Normal (162887.33)	Attack (316557.83)
Normal	mean: 91.77 std dev: 3.3	mean: 0.78 std dev: 1.28	mean: 90.76 std dev: 1.19	mean: 0.17 std dev: 0.1	mean: 91.28 std dev: 0.84	mean: 0.13 std dev: 0.05	mean: 91.54 std dev: 0.3	mean: 0.17 std dev: 0.05	mean: 91.4 std dev: 0.25	mean: 0.12 std dev: 0.06
Attack	mean: 8.23 std dev: 3.3	mean: 99.22 std dev: 1.28	mean: 9.24 std dev: 1.19	mean: 99.83 std dev: 0.1	mean: 8.72 std dev: 0.84	mean: 99.87 std dev: 0.05	mean: 8.46 std dev: 0.3	mean: 99.83 std dev: 0.05	mean: 8.6 std dev: 0.25	mean: 99.88 std dev: 0.06

	Trained on 30% of rand. train.	Trained on 30% of rand. train.	Trained on 30% of of rand. train.	Trained on 30% of of rand. train.	Trained on 30% of of rand. train.

	Normal (159507.22)	Attack (321280.33)	Normal (162262.22)	Attack (317380.5)	Normal (162673.22)	Attack (316854.17)	Normal (163582.56)	Attack (317485.17)	Normal (164253.67)	Attack (316980.5)
Normal	mean: 91.9 std dev: 3.21	mean: 0.78 std dev: 1.31	mean: 91.51 std dev: 0.77	mean: 0.19 std dev: 0.12	mean: 91.4 std dev: 0.5	mean: 0.16 std dev: 0.07	mean: 92.48 std dev: 1.8	mean: 0.35 std dev: 0.62	mean: 92.22 std dev: 1.85	mean: 0.46 std dev: 0.99
Attack	mean: 8.23 std dev: 3.21	mean: 99.22 std dev: 1.31	mean: 8.49 std dev: 0.77	mean: 99.81 std dev: 0.12	mean: 8.6 std dev: 0.5	mean: 99.84 std dev: 0.07	mean: 7.52 std dev: 1.8	mean: 99.65 std dev: 0.62	mean: 7.78 std dev: 1.85	mean: 99.54 std dev: 0.99

	Trained on 40% of rand. train.		Trained on 40% of rand. train.		Trained on 40% of of rand. train.		Trained on 40% of of rand. train.		Trained on 40% of of rand. train.	
	Normal (164109.56)	Attack (313863.33)	Normal (162670.78)	Attack (316263.33)	Normal (162801.44)	Attack (316554.5)	Normal (163293.11)	Attack (316001.5)	Normal (162188.89)	Attack (317164.5)
Normal	mean: 89.94 std dev: 2.9	mean: 0.37 std dev: 0.91	mean: 91.02 std dev: 1.15	mean: 0.18 std dev: 0.14	mean: 91.55 std dev: 0.95	mean: 0.15 std dev: 0.07	mean: 91.62 std dev: 2.2	mean: 0.42 std dev: 0.95	mean: 91.53 std dev: 1.25	mean: 0.13 std dev: 0.03
Attack	mean: 10.06 std dev: 2.9	mean: 99.63 std dev: 0.91	mean: 8.98 std dev: 1.15	mean: 99.82 std dev: 0.14	mean: 8.45 std dev: 0.95	mean: 99.85 std dev: 0.07	mean: 8.38 std dev: 2.2	mean: 99.58 std dev: 0.95	mean: 8.47 std dev: 1.25	mean: 99.87 std dev: 0.03

CHAPTER V

CONCLUSION AND FUTURE WORK

Conclusion

An IDS user has different concerns from an attacker. The IDS user is concerned about the number of false alarms out of the total alarms, whereas the attacker is concerned about the total number of false alarms out of the total number of attacks. As shown in the four equations below, these two totals may differ.

Using an ordinary confusion matrix, the percentages that are of interest to the attacker can be calculated by dividing each entry by the horizontal sum of its row. The percentages that are relevant to the IDS user can be calculated by dividing each entry by the vertical sum of its column.

From the information shown above, we can see that when considering the data from the attacker's point of view, the percentage of false negatives can be calculated by dividing the attacks not found by the IDS by the total attacks and the false positives can be found by dividing the normal data that causes an alarm by the total normal data. On the other hand, the IDS user wants to know the percentage of total alarms that the false positives make up. In addition, they would like to know the percentage of non-alarm data made up by the false negatives (data that is actually an attack). It is noticed that the percentages given in [64] are only from the attacker's point of view. The four equations below state the method used to calculate the 2 different points of view, where A_K and N_K are those examples known to be attacks and normal, respectively, and where A_C and N_C are those classified as attack and normal connections, respectively.

Attacker's False Negatives: $\frac{A_K \wedge N_C}{A_K}$

Attacker's False Positives: $\frac{N_K \wedge A_C}{N_K}$

IDS User False Negatives: $\frac{A_K \wedge N_C}{N_C}$

IDS User False Positives: $\frac{N_K \wedge A_C}{A_C}$

It is important to note that the distribution of false alarms and false negatives from the user's perspective is affected by the relative sizes of the classes, whereas the distribution from the attacker's perspective is not.

From these experiments it can be concluded that the machine learning algorithms used would misclassify an undesirably large amount of data when the number of attack connections is much larger than the number of normal connections. In a real network, there may be times when this type of activity occurs. For example, during a DoS attack, attack data may exceed normal data. Such attacks may target certain networks more frequently than others. For example, the network for a popular website may be the target of a DoS attack. In other circumstances, normal connections may greatly exceed attack connections.

The Weka Multilayer Perceptron runs much more slowly than the Weka Naive Bayes and J48 decision tree. This is likely a result of the fact that the size of the model for the multilayer perceptron grows considerably each time the training data is increased. This may be a result of the way that the Weka Multilayer Perceptron algorithm stores backpropagation information, although due to the apparent lack of documentation on this issue, some other factor may be the actual cause.

Future Research

The algorithms tested use supervised learning algorithms. These algorithms require an offline training phase, but the testing phase requires much less time and future work could investigate how well it can be adapted to performing online. The main difficulties in adapting these techniques for practical use are the difficulties involved in acquiring labeled training data and in investigating how the training on this dataset can be useful in classifying real datasets.

Another possible application is to use the machine learning algorithm models to create new rules for rule-based IDS's such as Snort. Once they have been trained and tested, rules could be extracted from examining the data structures of the machine learning algorithms. These rules might then be used in Snort or another online intrusion detection system. In order to accomplish this, however, the algorithms might have to be run on a new dataset to obtain rules that would be useful to Snort or another rule-based IDS. This dataset would use only attributes examined by such an IDS.

In addition, a classifier which uses the dataset and the results of several classifiers on that dataset may be of interest. Such a technique can be found in [66] and [67]. This technique can result in improved performance and the identification of which portions of the dataset that each classifier identifies well. Such a classifier might be run on the dataset used in this thesis along with the classifications of that dataset given by the Naive Bayes, Decision Tree and Multilayer Perceptron. The performance of this classifier could then be compared to the use of these three classifiers by themselves to determine if an increase in performance is obtained.

It is possible that honeypots[55] simulating various types of networks may be able to obtain a more realistic view of what actual attacks are like than the 1999 UCI KDD dataset used in this thesis. However, honeypots are designed to collect attack data only. The advantage of the honeypot is that it has no authorized activity, and so any activity found on it is therefore unauthorized. However, it might be possible to somehow combine this activity with simulated normal activity, perhaps by combining it with a dataset that contains simulated normal activity. The simulated normal activity should resemble actual normal activity as much as possible. Another possibility would be to compare it with a dataset that contains both simulated normal

and attack connections (such as from the UCI KDD dataset). In this way it may be possible to obtain more accurate data about intrusions without using data from real networks.

APPENDIX

The Backpropagation Learning Algorithm

In an artificial neural network (ANN), backpropagation can be used to update the weights in the hidden layers. In order to perform backpropagation, the ANN must first calculate $\frac{\delta E}{\delta w_{i,j}}$, the derivative of the error, E, with respect to the weight, w, of the link from neuron j to neuron i (this error is calculated by traveling the links backward, from i to j). There is only one error, E, which is the sum of the errors of the output neurons. The value is calculated as follows:

Let v be the total number of examples.

Let z be the total number of neurons in the output layer.

Let e be the example for which the error is being calculated.

Let j be the output neuron for which the error is being calculated (the hidden nodes use this error to compute their δ's since the error of those nodes cannot be directly calculated).

$predicted = f(\sum_{j=1}^{z} w_k y_k) = f(a_j)$

$E = \frac{1}{2} \sum_{e=0}^{v} \sum_{j=0}^{z} (actual_{e,n} - predicted_{e,n})^2$

$= \frac{1}{2} \sum_{e=0}^{v} \sum_{j=0}^{z} (actual_{e,n} - f(\sum_{i=1}^{p} w_k v_k))^2$

$= \frac{1}{2} \sum_{e=0}^{v} \sum_{j=0}^{z} (actual_{e,n} - f(a_j))^2$

$= \frac{1}{2} \sum_{e=0}^{v} \sum_{j=0}^{z} (actual_{e,n} - y_j)^2$

(where p is the number of neurons in the layer previous to the output layer)

$E = \sum_{e=0}^{z} E_e$

$E_e = \frac{1}{2} \sum_{j=0}^{z} (actual_{e,n} - y_j)^2$

$$\frac{\delta E}{\delta w_{j,i}} = \sum_{e=0}^{v} \frac{\delta E_e}{\delta w_{j,i}}$$

Using the chain rule it can be determined that

$$\frac{\delta E_e}{\delta w_{j,i}} = \sum \frac{\delta E_e}{\delta a_j} \frac{\delta a_j}{\delta w_{j,i}}.$$

$$\delta_j = \frac{\delta E_e}{\delta a_j} = \frac{\delta E_e}{\delta y_j} \frac{\delta y_j}{\delta a_j}$$

Let $u = actual_{e,n} - y_j$

Then $E_e = \frac{1}{2} \sum_{n=0}^{z} (u)^2 = \sum_{n=0}^{z} \frac{1}{2} (u)^2$

$$\frac{\delta E_e}{\delta y_j} = \frac{\delta E_e}{\delta u} \frac{\delta u}{\delta y_j} = (\sum_{n=0}^{z} u) u' = (\sum_{j=0}^{z} actual - y_j)(0 - 1) = -(\sum_{j=0}^{z} actual - y_j)$$

$\frac{\delta y_j}{\delta a_j} = f'(a_j) = f(a_j)(1 - f(a_j)) = y_j(1 - y_j)$ (in the case of the sigmoid function)

$$\delta_j = \frac{\delta E_e}{\delta a_j} = \frac{\delta E_e}{\delta y_j} \frac{\delta y_j}{\delta a_j} = -\sum_{e=0}^{z} (actual_{e,j} - y_{e,j}) f(a_{e,j})(1 - f(a_{e,j})) = -\sum_{e=0}^{z} (actual_{e,j} - y_{e,j}) y_{e,j}(1 - y_{e,j})$$ (for output nodes when using the sigmoid function)

where $Err = \sum_{e=0}^{z}(actual_{e,j} - y_{e,j})$.

To calculate δ_k for the hidden layers, a different chain rule derivation can be used.

$\delta_k = \frac{\delta E_e}{\delta a_k} = \frac{\delta E_e}{\delta a_j}\frac{\delta a_j}{\delta a_k}$

Where j is now the index to the neurons in the layer to which the output a_k is sent. This gives

$\delta_k = \frac{\delta E_e}{\delta a_k} = \delta_j \frac{\delta a_j}{\delta a_k}$

The second factor can then be calculated. Let z now be the total number of neurons in layer k

$a_j = \sum_{j=0}^{z} w_{j,k} f(a_k) = \sum_{j=0}^{z} w_{j,k} y_k$,

we can see that

$\frac{\delta a_j}{\delta a_k} = \frac{\delta a_j}{\delta f(a_k)}\frac{\delta f(a_k)}{\delta a_k}$.

Therefore, let $a_j = w_{k,j} f(a_k) = w_{k,j} y_k$ and use the chain rule to obtain the first part as

$\frac{\delta a_j}{\delta f(a_k)} = \sum_{j=0}^{z} w_{k,j}$.

The value is therefore $w_{k,j}$ for all nodes j connected to k. If the neural net is not fully-connected, then some of these values will be zero.

The second part can be calculated as

$\frac{\delta f(a_k)}{\delta a_k} = f'(a_k)$.

Therefore,

$\frac{\delta a_j}{\delta a_k} = (\sum_{j=0}^{z} w_{k,j}) f'(a_k)$.

This means that

$\delta_k = \frac{\delta E_e}{\delta a_k} = \delta_j \frac{\delta a_j}{\delta a_k} = (\sum_{j=0}^{z} \delta_j w_{k,j}) f'(a_k)$.

For the sigmoid function, the equation is

$\delta_k = \frac{\delta E_e}{\delta a_k} = \delta_j \frac{\delta a_j}{\delta a_k} = (\sum_{j=0}^{z} \delta_j w_{k,j}) f(a_k)(1 - f(a_k)) = (\sum_{j=0}^{z} \delta_j w_{k,j}) y_k(1 - y_k)$.

Now, recall that

$\frac{\delta E_e}{\delta w_{k,j}} = \sum \delta_j \frac{\delta a_k}{\delta w_{k,j}}$.

Because $a_k = w_{k,j} y_l$,

$\frac{\delta a_k}{\delta w_{k,j}} = y_l$, where y_l is the input to node k from node l.

BIBLIOGRAPHY

[1] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach (International Edition)*. Pearson US Imports & PHIPEs, November 2002.

[2] C. P. Pleeger, *Security in Computing, 2nd Edition*. Prentice Hall, 1997.

[3] L. K. Westin, "Receiver operating characteristic (roc) analysis: Evaluating discriminance effects among decision support systems," 2001.

[4] C. Drummond and R. Holte, "What roc curves can't do (and cost curves can)."

[5] C. Drummond and R. C. Holte, "Explicitly representing expected cost: an alternative to ROC representation," in *Knowledge Discovery and Data Mining*, pp. 198–207, 2000.

[6] "Weka 3: Data mining software in java. http://www.cs.waikato.ac.nz/ml/weka/."

[7] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2005.

[8] SANS, "Intrusion detection faq. http://www.sans.org/resources/idfaq/." Website, 2003.

[9] "Packet. wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/packet." Website, October 2006.

[10] I. S. Institute, "Rfc 791 (rfc791). http://www.faqs.org/rfcs/rfc791.html." Website, September 1981.

[11] E. Hall, "Personal communicatuion," 2006.

[12] Fyodor, "Insecure.org: Exploit world. http://insecure.org/sploits.html." Website, 2000.

[13] M. E. Donaldson, "Inside the buffer overflow attack: Mechanism, method, and prevention. http://www.sans.org/reading_room/whitepapers/securecode/386.php." Website, 2002.

[14] T. H. Ptacek and T. N. Newsham, "Insertion, evasion, and denial of service: Eluding network intrusion detection," tech. rep., Secure Networks, Inc., Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6, 1998.

[15] S. Siddharth, "Evading nids, revisited. http://www.securityfocus.com/infocus/1852." Website, 2005.

[16] G. Combs, "Wireshark." Website, 2006.

[17] B. Laing and J. Alderson, "How to guide: Intrusion detection systems. http://www. snort.org/docs/iss-placement.pdf." Website, 2000.

[18] "Promiscuous mode. http://www.linktionary.com/p/promiscuous.html."

[19] M. Wood, "Selecting a site for the software sentry. http://www.securitymanagement.com/library/000763.html.."

[20] A. Firman, "11 step guide to build a debian based intrusion detection sensor (ids) with snort 2.4.5 or snort 2.6. http://www.snort.org/docs/setup_guides/deb-snort-howto.pdf." Website, 2006.

[21] S. Axelsson, "The base-rate fallacy and its implications for the difficulty of intrusion detection," in *ACM Conference on Computer and Communications Security*, pp. 1–7, 1999.

[22] M. Roesch, "snort.org. http://www.snort.org/." Website, 2006.

[23] "Snort user's manual. http://www.snort.org/docs/snort|_manual/2.6.1/snort|_manual.pdf."

[24] "http://sourceforge.net/projects/libpcap/." Website.

[25] D. Ragsdale, C. Carver, J. Humphries, and U. Pooch, "Adaptation techniques for intrusion detection and intrusion response system," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics at Nashville, Tennessee*, pp. 2344–2349, October 2000.

[26] S. J. Stolfo, W. Lee, P. K. Chan, W. Fan, and E. Eskin, "Data mining-based intrusion detectors: an overview of the columbia ids project," *SIGMOD Rec.*, vol. 30, no. 4, pp. 5–14, 2001.

[27] G. Vigna, F. Valeur, and R. A. Kemmerer, "Designing and implementing a family of intrusion detection systems," in *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, (New York, NY, USA), pp. 88–97, ACM Press, 2003.

[28] J. McHugh, "Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory.," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 4, pp. 262–294, 2000.

[29] D. Wagner and P. Soto, "Mimicry attacks on host based intrusion detection systems. citeseer.ist.psu.edu/wagner02mimicry.html," 2002.

[30] G. Tandon, P. Chan, and D. Mitra, "Morpheus: motif oriented representations to purge hostile events from unlabeled sequences," in *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, (New York, NY, USA), pp. 16–25, ACM Press, 2004.

[31] J. B. D. Cabrera, L. Lewis, and R. K. Mehra, "Detection and classification of intrusions and faults using sequences of system calls," *SIGMOD Rec.*, vol. 30, no. 4, pp. 25–34, 2001.

[32] J. C. Rabek, R. I. Khazan, S. M. Lewandowski, and R. K. Cunningham, "Detection of injected, dynamically generated, and obfuscated malicious code," in *WORM '03: Proceedings of the 2003 ACM workshop on Rapid malcode*, (New York, NY, USA), pp. 76–82, ACM Press, 2003.

[33] T. Hollebeek and R. Waltzman, "The role of suspicion in model-based intrusion detection," in *NSPW '04: Proceedings of the 2004 workshop on New security paradigms*, (New York, NY, USA), pp. 87–94, ACM Press, 2004.

[34] P. A. Porras and P. G. Neumann, "EMERALD: Event monitoring enabling responses to anomalous live disturbances," in *Proc. 20th NIST-NCSC National Information Systems Security Conference*, pp. 353–365, 1997.

[35] P. G. Neumann and P. A. Porras, "Experience with EMERALD to date," pp. 73–80, 1999.

[36] U. Lindqvist and P. A. Porras, "expert-bsm: A host-based intrusion detection solution for sun solaris," pp. 240–251, December 10-14 2001.

[37] M. Almgren and U. Lindqvist, "Application-integrated data collection for security monitoring," in *Recent Advances in Intrusion Detection (RAID 2001)*, LNCS, (Davis, California), pp. 22–36, Springer, October 2001.

[38] A. Valdes and K. Skinner, "Probabilistic alert correlation," in *Recent Advances in Intrusion Detection (RAID 2001)*, no. 2212 in Lecture Notes in Computer Science, Springer-Verlag, 2001.

[39] A. Valdes and K. Skinner, "Adaptive, model-based monitoring for cyber attack detection," in *Recent Advances in Intrusion Detection (RAID 2000)* (H. Debar, L. Me, and F. Wu, eds.), no. 1907 in Lecture Notes in Computer Science, (Toulouse, France), pp. 80–92, Springer-Verlag, October 2000.

[40] U. Lindqvist and P. A. Porras, "Detecting computer and network misuse through the production-based expert system toolset (p-best)," in *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, (Oakland, California), pp. 146–161, IEEE Computer Society Press, Los Alamitos, California, may 1999.

[41] D. Barbará;, J. Couto, S. Jajodia, and N. Wu, "Adam: a testbed for exploring the use of data mining in intrusion detection," *SIGMOD Rec.*, vol. 30, no. 4, pp. 15–24, 2001.

[42] S. Chavan, K. Shah, N. Dave, S. Mukherjee, A. Abraham, and S. Sanyal, "Adaptive neuro-fuzzy intrusion detection systems," *itcc*, vol. 01, p. 70, 2004.

[43] K. Sequeira and M. Zaki, "Admit: anomaly-based data mining for intrusions," in *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, (New York, NY, USA), pp. 386–395, ACM Press, 2002.

[44] S. B. Idris, N.B., "Artificial intelligence techniques applied to intrusion detection," in *INDICON, 2005 Annual IEEE*, pp. 52–55–11–13, December 2005.

[45] T. M. Khoshgoftaar, S. V. Nath, and S. Zhong, "Intrusion detection in wireless networks using clustering techniques with expert analysis," in *Proceedings of the Fourth International Conference on Machine Learning and Applications (ICMLA'05)*, IEEE, 2005.

[46] M. M. Pillai, J. H. P. Eloff, and H. S. Venter, "An approach to implement a network intrusion detection system using genetic algorithms," in *SAICSIT '04: Proceedings of the 2004 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, (, Republic of South Africa), pp. 221–221, South African Institute for Computer Scientists and Information Technologists, 2004.

[47] Y. Hu and B. Panda, "A data mining approach for database intrusion detection," in *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, (New York, NY, USA), pp. 711–716, ACM Press, 2004.

[48] H. Han, X. L. Lu, J. Lu, C. Bo, and R. L. Yong, "Data mining aided signature discovery in network-based intrusion detection system," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 4, pp. 7–13, 2002.

[49] M. Botha, R. von Solms, K. Perry, E. Loubser, and G. Yamoyany, "The utilization of artificial intelligence in a hybrid intrusion detection system," in *SAICSIT '02: Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, (, Republic of South Africa), pp. 149–155, South African Institute for Computer Scientists and Information Technologists, 2002.

[50] J. Bala, S. Baik, A. Hadjarian, B. K. Gogia, and C. Manthorne, "Application of a distributed data mining approach to network intrusion detection," in *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, (New York, NY, USA), pp. 1419–1420, ACM Press, 2002.

[51] Y. Feng, Z.-F. Wu, K.-G. Wu, Z.-Y. Xiong, and Y. Zhou, "An unsupervised anomaly intrusion detection algorithm based on swarm intelligence," in *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*, pp. 3965– 3969, 2005.

[52] S. Hettich and S. D. Bay, "Kdd cup 1999 data. http://kdd.ics.uci.edu//databases/kddcup99/kddcup99.html.irvine, ca: University of california, department of information and computer science." Website, 1999.

[53] E. Eskin, A. Arnold, M. Prerau, L. Portnoy, and S. Stolfo, "A geometric framework for unsupervised anomaly detection: Detecting intrusions in unlabeled data," 2002.

[54] B. Malin, "Personal communicatuion," 2006.

[55] J. Tunnissen, "Honeypots.net. http://www.honeypots.net/," 2002-2007.

[56] S. Hettich and S. D. Bay, "Mit lincoln laboratory–darpa intrusion detection evaluation sets. http://www.ll.mit.edu/ist/ideval/data/data|_index.html." Website, 1998.

[57] D. Fisher, L. Xu, J. Carnes, Y. Reich, S. Fenves, J. Chen, R. Shiavi, G. Biswas, and J. Weinberg, "Applying ai clustering to engineering tasks," pp. 51–60, Dec 1993.

[58] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, pp. 379–423 and 623–656, 1948.

[59] G. Gigerenzer, *Calculated Risks: How to Know When the Numbers Deceive You*. Simon |& Schuster, 2002.

[60] P. Domingos and M. J. Pazzani, "On the optimality of the simple bayesian classifier under zero-one loss," *Machine Learning*, vol. 29, no. 2-3, pp. 103–130, 1997.

[61] R. D. Reed and J. Marks II, Robert, *Neural Smithing*. Massachusetts Institute of Technology, 1999.

[62] P. A. C. P. Stolfo J, Fan W. Lee W, "Cost-based modeling for fraud and intrusion detection: results from the jam project," in *DARPA Information Survivability Conference and Exposition*, vol. 2 of *DISCEX*, pp. 130–144, 2000.

[63] "Cream. http://cream.sourceforge.net/."

[64] N. B. Amor, S. Benferhat, and Z. Elouedi, "Naive bayes vs decision trees in intrusion detection systems," in *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, (New York, NY, USA), pp. 420–424, ACM Press, 2004.

[65] "Standard deviation. wolfram mathworld. http://mathworld.wolfram.com/standarddeviation.html."

[66] J. Ortega, M. Koppel, and S. Argamon, "Arbitrating among competing classifiers using learned referees," *Knowledge and Information Systems*, vol. 3, no. 4, pp. 470–490, 2001.

[67] J. Louis Anthony Cox, "Identifying chemical carcinogens and assessing potential risk in short-term bioassays using transgenic mouse models," *Environmental Health Perspectives*, vol. 103, no. 11, 1995.