Intrusion and Fraud Detection using Multiple Machine Learning Algorithms

by

Chad Aaron Peters

A thesis submitted to
The Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements
of the degree of

Master of Science

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba, Canada
August 2013

© Copyright by Chad Aaron Peters, 2013

Thesis advisor Author

John Anderson

Jacky Baltes **Chad Aaron Peters**

Intrusion and Fraud Detection using Multiple Machine Learning Algorithms

Abstract

New methods of attacking networks are being invented at an alarming rate, and pure signature detection cannot keep up. The ability of intrusion detection systems to generalize to new attacks based on behavior is of increasing value. Machine Learning algorithms have been successfully applied to intrusion and fraud detection; however the time and accuracy tradeoffs between algorithms are not always considered when faced with such a broad range of choices. This thesis explores the time and accuracy metrics of a wide variety of machine learning algorithms, using a purpose-built supervised learning dataset. Topics covered include dataset dimensionality reduction through pre-processing techniques, training and testing times, classification accuracy, and performance tradeoffs. Further, ensemble learning and meta-classification are used to explore combinations of the algorithms and derived data sets, to examine the effects of homogeneous and heterogeneous aggregations. The results of this research are presented with observations and guidelines for choosing learning schemes in this domain.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to begin by thanking my advisors, for endless encouragement through the darkest of days; my committee, for sound judgement and insightful correction; my parents, for dismissing my resolution never to go back to school; my lovely wife, for kicking me into action when I thought I didn't have anything left; and my beautiful children, for letting daddy work at a boring machine on the sunniest of days. It's play time!

"A year spent in artificial intelligence
is enough to make one believe in God."
– Alan Perlis

Chapter 1

Introduction

1.1 Overview

The art of digital exploitation has proliferated in recent years. Since the advent of online business transactions, and with the ever-increasing amount of potentially sensitive personal and corporate data being transmitted over the Internet, computer hacking has never been more profitable [Leeson and Coyne, 2006]. To combat this, businesses, instituitions, and individuals are protecting intellectual property extensively, using different physical and digital layers.

Attempts to breach security always leave some indication, however sparse, and security researchers have devoted much time to documenting the patterns these indications take [Schultz and Shumway, 2001; Liu et al., 2010]. Armed with a knowledge of these patterns, one has the ability to audit system states, network traffic, and user logs to detect known patterns of attack. In order to detect these known attack patterns, modern intrusion detection methods rely on signature and threshold analysis (i.e., matching specific and behavioral patterns respectively) using historical traffic [Schultz and Shumway, 2001; Liu et al., 2010].

Signature and threshold analysis both look for patterns in datasets. Some of the biggest challenges in applying pattern recognition algorithms to any problem domain include finding an appropriate classification model for the domain, measuring classification accuracy, and performing feature selection when faced with a problem domain with a high dimension space [Guyon and Elisseeff, 2003; Patcha and Park, 2007]. Performance can be relative to the feature sets that are chosen, depending on the number of features in the feature space, and any possible correlations between the many feature combinations that are available for further examination. For example, if only the payload is inspected, then well-known compromised hosts are treated the same way as normal, uncompromised hosts. If only source address reputation is considered, then newly-compromised hosts are trusted until their reputation has been updated. In the domain of network intrusion detection, there is yet to be found a "golden egg" - a one-size-fits-all solution - that is optimal for every feature set [Patcha and Park, 2007]. As we will see, modern research has found a number of plausible methods to apply to the domain of behavior-based network intrusion detection, however each individual analysis methodology appears to have particular strengths and weaknesses to consider when attempting to learn network traffic behaviors.

My research explores this issue through an empirical study of various machine learning algorithms using a dataset produced with anomaly detection in mind. The testing methodology was designed so that a variety of machine learning algorithms, each based on statistical and mathematical methods known to have particular strengths and weaknesses, could be tested against the dataset in an unbiased fashion. These experiments are accomplished through a methodology encompassing data pre-processing, running algorithms individually, ensemble learning (using multiple instances of the same algorithm), and meta-classification (using a group of different algorithms). The resulting performance metrics are then systematically examined for patterns that may help determine which combinations of techniques are superior for intrusion detection, given the limitations of real-world scenarios of computing power and time. In the conclusion of this thesis, some guidelines are offered based on the observations made through evaluation, so that future researchers may have an idea of which schemes are best to start with before attempting to tune the individual algorithms for a specific security application.

1.2 Terminology

This thesis discusses the concept of applying machine learning algorithms to learn the difference between *good* and *bad* network data. Before exploring the rest of the topics in this work, some terminology behind the domain should be discussed.

1.2.1 Ethics

In the context of this thesis, *good* network data is the result of transactions between two computers over a network for the legitimate purpose of conducting work or research in sanctioned manner (a general concept used by others in the security domain) [Bishop, 2005]. The datasets used later on classify good data as *normal*, since the data is expected and is what administrators would not react to defensively.

Bad network data, on the other hand, is data produced by transactions that are fraudulent in nature, whether intentional or not Stolfo et al. [2000]. *Fraud* can be said to be any activity that is conducted with intentional deception. Bad transactions have no legitimate purpose, or worse, are designed to disrupt or illegitimately manipulate services offered by another host residing on the network. Computer network *intrusions* (a form of fraudulent activity) typically attempt to send data from one host to another in order to illicit behavior that is detrimental to the people, processes, or technology involved [Farid and Rahman, 2010; Bishop, 2005]. In order to detect fraud and system intrusion at the network layer, network flow data is examined in order to determine if it is suspected of being fraudulent in nature.

Although these intrusions are, unfortunately, expected, for the purpose of this research they are classified as *anomalies*, since they are not expected to occur as part of legitimate traffic, and therefore are warranted to cause a defensive reaction from network administrators.

1.2.2 Traffic Data

Network *flow data* is the sum of all packets that consists of one transaction between two hosts on a network, and can be described using a number of *features* (variables), that in combination consists of that network flow's overall *behavior* [Guyon and Elisseeff, 2003]. Considering we are interested in modeling network flow behavior, the machine learning algorithms tested in this domain will be using the network data flow features in order to come up with a way to predict whether or not flow data is normal or anomalous based on past observances.

1.2.3 Hypothesis Models

Machine learning algorithms operate on the concept of a *hypothesis model* [Mitchell, 1997]. A *hypothesis* is a statement made about reality based on currently known facts. A *model* is a particular framework (at a point in time), that describes the correlation between one or more input variables, and one or more output variables. A simple example would be using temperature and humidity to predict good and bad weather. The temperature and humidity would be considered input variables, and the classification of "good", or "bad" weather would be output variables. My hypothesis would be the statement I make about the weather based on observable facts, whereas my model is the framework (whether gut feeling or previous weather statistics input into a formula) that I use in order to predict my hypothesis.

1.2.4 Machine Learning

There are two broad types of machine learning paradigms, based on the amount of human interaction involved in the learning process; namely *supervised* and *unsupervised* learning [Mitchell, 1997]. Supervised learning algorithms use previously labeled training sets (both the input and output variables are defined) in order to (hopefully) find a correlation between the input and output variables, and produce accurate predictions based on future observations that are not labeled. Unsupervised learning algorithms only provide the input variables, with the goal of finding patterns in the data that may indicate some relationship between the input variables (such as patterns) that can be used to find interesting correlations specific to the domain in question.

1.2.5 Purpose

My thesis will use supervised machine learning algorithms to produce a prediction model capable of classifying normal and abnormal traffic. The observed network flow data will contain traffic features, as well as a classification of whether each network flow instance (each of which is comprised of multiple behavioral features) is normal, or an anomaly. Using the constructed hypothesis model, previously unobserved network flow data instances will be presented to the model for prediction. The model will attempt to predict whether or not the instance is normal or anomalous, and the result recorded. This will be repeated for all instances in the unobserved set to determine how closely the created model represents reality, and thus the overall usefulness for traffic classification.

1.3 Experimental Methods

The experiments used in this work are based on supervised learning techniques. The experimental method includes various phases of data pre-processing, algorithm training, and testing of the various algorithms available against labeled network flow data. In order to accomplish this, three components are required. First, the approach requires a dataset that is representative of a typical network (whether real or produced) including a variety of network flow samples that have a ratio of *normal* to *anomalous* labels on which to train and test. Second, the machine learning algorithms are required so that a hypothesis model can be established and future predications can be made (the nature of these algorithms are explained in Section 1.3.2). Finally, the experiments require a method by which to systematically read the network data, introduce it to the machine learning algorithms, create a learning model, and repeatedly run the model against unseen data in order to compare the algorithm's prediction abilities, as well as ability to generalize to unseen data in the hypothesis space.

1.3.1 Data Sets

Historically, the creation of a hypothesis model based on previously observed network data has been a difficult task for many reasons, most of which boil down to a lack of properly labeled data with which to test (explained further in Chapter 2). Correctly labeling data as good or bad is a time consuming effort that is not easily automated. Organizations that do take the time to label data for these purposes are often under non-disclosure agreements or industry and government data protections rules that do not allow sharing labeled data, so the use of such data to support research is not possible. For this reason there are very few data sources that are available for public consumption which come close to representing a "real" network that contains both benign and malicious traffic.

For this reason the US-based Defense Advanced Research Projects Agency (DARPA) Intrusion Detection Data Sets were created at the Massachusetts Institute of Technology (MIT) Lincoln Laboratory. These data sets, created in 1999, were designed with intrusion detection testing in mind, using a simulated computer network that closely resembled the Air Force Research Laboratory (AFRL/SNHS) network under whose sponsorship the project took place. A modified version of this data (correcting statistical inconsistencies found since creation) will be used for testing [Network

Security Laboratory, 2012].

1.3.2 Machine Learning Algorithms

The algorithms used for evaluation in Chapter 4 are publicly available and are already used in a multitude of industry applications, and have been highly refined and optimized. The re-creation of these algorithms would be redundant from the standpoint of this work, and thus a well-developed machine learning algorithm library framework was used instead. My experiments used varients of the Logistic Regression [Xu et al., 2005], Naive Bayes Classifier [Khor et al., 2009], Support Vector Machines [Mukkamala and Sung, 2003], K-nearest-neighbor [Faraoun and Boukelif, 2007; Chairunnisa et al., 2009], and Artificial Neural Networks (multi-level perceptrons) [Ali et al., 2010; Vatisekhovich, 2009; Zilberbrand, 2009], commonly used in Machine Learning.

1.3.3 Testing Framework

A number of robust machine learning frameworks were compared [Shogun, 2013; Hall et al., 2009; Collobert et al., 2002] to find a suitable candidate to test a wide range of well-known algorithms, with the ability to re-use models in a variety of combinations for ensemble and meta-classification. After comparing the available algorithms, and third-party programming support, I chose the Weka toolbox [Hall et al., 2009; Witten et al., 2011], for development, as the graphics user interface, open application programming interface, and documentation have been found to be robust, peer-reviewed, and amenable to custom scripting.

1.4 Research Questions

This thesis compares a variety of machine learning algorithms, alone and in combination, capable of making classification predictions on network flow data. My research attempts to answer the following questions:

1. Which algorithms have the fastest learning time? This is important to modify complex models on-the-fly.

2. Which algorithms have the fastest prediction time? This is important to classify network flows in high traffic environments.

3. Which algorithms have the highest accuracy using the dataset? How do we measure accuracy in this context?

4. What is the effect of pre-processing (filtering) on the training and testing data?

5. Which is the appropriate algorithm in terms of performance and accuracy, for the purposes of classifying network traffic?

6. Is the added complexity of more sophisticated single and ensemble learning schemes worth the trade-off of runtime performance vs. classification accuracy?

Through data gathered from experiments and observations made during experimentation, I will attempt to answer these research questions and provide insight for future research into the problem domain.

1.5 Summary

In this chapter I introduced the domain of fraud and intrusion detection, and the terminology used within this domain, as well as the motivation for studying the topic in depth. I also discussed the general methodology that will be used, and the dataset to be tested against in order to predict each algorithm's overall performance for the problem domain.

The remainder of this document is outlined as follows. Chapter 2 offers an introduction and brief history of auditing and intrusion detection, with related research in the field of intrusion and anomaly detection using established behavior profiles, and comparisons of the effectiveness of applying various machine learning techniques to this problem domain. Chapter 3 outlines my experimental methodology used for dataset pre-processing and algorithm evaluation. In Chapter 4, the algorithm experiments are described in further detail, and the results of each algorithm, alone and in a number of combinations, is systematically evaluated. Finally, Chapter 5 reviews the contributions of this work, and outlines future research to be conducted.

Chapter 2

Background

2.1 Overview

Network security has an interesting history, with some of the earliest auditing methodologies appearing in the 1980s [Anderson, 1980]. Since that time, many methods have been used to try to make this task easier. The ideas are fundamentally the same, but the techniques become more sophisticated over time. This chapter will discuss the background of ideas behind auditing and intrusion detection, as well as fundamental machine learning models that can be employed. Some examples of how these models have been applied will be discussed, as will the fundamental algorithms behind the construction of these models. Some common algorithm evaluation metrics will be introduced, and these metrics will be referenced as algorithms and combinations of algorithms are compared and contrasted in future chapters.

2.1.1 Anatomy of a Hack

According to McClure et al. [2005], sophisticated attacks will begin with a common set of reconnaissance techniques — namely, footprinting, scanning, and enumeration — before access or infection attempts. Footprinting involves a non-intrusive casing of the target using open source searches and public queries. Scanning attempts, such as ping sweeps and port scans, are used to elicit a response on listening services. Enumeration involves acquiring user accounts, file shares, and identifying specific software on the target with known exploits. Once all of these steps have been completed, an informed attempt can be made on the target to gain privileged access and work directly within the host. According to Ye [2008], at this point, the keys to moving around and maintaining constant and undetected access involve privilege escalation, architecture and information pilfering, rootkit installation, and log modifications.

2.1.2 Auditing

Efforts continue to be made to compromise secured systems, regardless of the host or network security measures in place. Anderson [1980] predicted that an audit of the system or data in question must be done by a trusted party using audit trails to detect the possibility of an attempted or successful exploit. The main components of modern audit systems include logging, analysis, and notification [Bishop, 2005]. Logging data involves capturing changes in host state, or the capture of packets in a distributed environment.

2.1.3 Analysis

After the data to be audited has been logged, an analyzer will look for patterns matching a defined rule set. Each instance of suspicious data that matches a known pattern will raise a *red flag* for the notifier. The notifier communicates the flagged instance in a number of potential methods for manual intervention and response. Audits are also usually done on individual host data or network traffic that crosses an aggregated point. The amount of time it takes to audit data increases as the complexity of software systems and the amount of transmitted network data increases. Intrusion Detection Systems attempt to remedy this workload by automating the audit process.

2.2 Intrusion Detection

2.2.1 Analysis Models

According to Bishop [2005], a good intrusion detection system detects a wide variety of intrusions, in a timely fashion, and presents analysis results as simply and accurately as possible, using any combination of anomaly detection, misuse modeling, or signature detection to identify threats. Anomaly detection works by assuming that attacks look out of the ordinary. Before we can find an anomaly, we need to map out what is normal, and using thresholds look for traffic that is out of these bounds. Misuse modeling looks for specific commands or actions that lead to a known misuse or abuse of otherwise appropriate system states. Signature detection involves recognizing patterns of known code states, that can put the system in an undesirable state when executed.

2.2.2 Traffic Scanning

One of the first steps in recognizing a remote attack is flagging suspicious activity during the scanning phase of a remote attack. Properly classifying a remote host as a scanner can then be done in two steps, namely, connection and host classification [Allman et al., 2007]. Connection classification involves recording specific details about attempted connections, including connection time and duration, host and destination addresses, data transferred, and matching destination ports to well-known services. Once the connection data has been used to classify the connection types, the suspected remote host can be then be classified based on the *fan out*, or variety and order by which local services are accessed.

Nychis et al. [2008] have found that through a series of synthetic anomalies such as distributed network port scans, distributed denial of service floods, and bandwidth floods, a more complicated approach to traffic analysis produces better results when facing more sophisticated attacks. The caveat of using a complicated scanning system is that there is a performance cost for such sophistication.

The following section reviews recent advancements in intrusion detection models, to better understand how advanced profile analysis methods can be accomplished without increasing resource overhead, and their original applications to ever-changing network environments.

2.3 Recent Research in Intrusion Detection and Behavior Classification Systems

Intrusion Detection Systems (IDSs)and behavior classification have come a long way since the inception of digital forensic auditing. Intrusion Detection Systems cannot detect all types of intrusions, as attack permutations are constantly being generated. Attack types can have many permutations, and static signatures do not always work. The alternative to signature detection, namely threshold establishment and monitoring, is used to detect the unknown.

A number of machine learning algorithms can be used to effect the classification of normal and malicious network traffic, enhancing an IDS to be able to generalize network traffic into "good" and "bad", thereby avoiding the necessity to use exact string matches [Chairunnisa et al., 2009]. These algorithms fall into two categories, based on their use of supervised and unsupervised learning. Supervised learning techniques require a human operator to classify a training set for the algorithm to learn from. A testing set with known classifications is then used to verify the accuracy of the learner. Unsupervised learning uses unlabeled data, and groups (clusters) training samples by distinguishing features [Mitchell, 1997].

Supervised (also known as *classification*) algorithms include maximum entropy, naive Bayes, support vector machines, K-nearest-neighbor, and neural networks [Mitchell, 1997]. These will be discussed below. Unsupervised (also known as *clustering*) algorithms include categorical mixture models, K-means clustering, and hierarchical clustering [Mitchell, 1997]. Since unsupervised algorithms create new classes based on feature similarity, and the focus of my research is on pre-defined classes, clustering algorithms are mostly out of scope of this research, and will be explained further where warranted.

Artificial Neural Networks can be applied with input nodes reading various network feature sets [Faraoun and Boukelif, 2007; Farid and Rahman, 2010]. Ali et al. [2010] accomplished this through an architectured approach consisting of network sensor, event manager, response manager, and applied learning model.

K-Means and K-Nearest Neighbour algorithms [Chairunnisa et al., 2009; Portnoy, 2000; Portnoy et al., 2001; Zanero and Savaresi, 2004] have been used to reduce Neural Network input selection, and to speed up overall learning time [Faraoun and Boukelif, 2007]. A centroid function can be used to choose the average and closest grouping of new instances in order to effectively group training examples together. Testing sets can then be classified based on which previously-established group they are closest to.

Support Vector Machines (SVMs) can be successfully applied to binary classification (i.e. classification into two categories) by dividing a p dimensional space with a $p-1$ dimensional hyperplane [Farid and Rahman, 2010; Gornitz et al., 2009]. Kloft et al. [2008] have shown that SVMs can be used to automate feature selection, which is important when dealing with multiple traffic header parameters, thus reducing overall computation and increasing success.

Network traffic modeling and analysis in a security context has been largely focused on static threshold-based anomaly detection, or known signature matching. Giroire et al. [2008] found that traditional network-based profile models were not flexible enough to accommodate user profiles in a modern, roaming environment. The way users behave at work, for example, differed from how they behave at home or in public access points, rendering traditional uniform threshold detection less useful in wireless environments.

Genetic algorithms (GAs), inspired by chromosomal duplication in biology, permute parent signatures to find viable combinations of pattern detection rules. Pillai et al. [2004] developed a core-plus-module framework called STAT (based on a state transition analysis technique) to allow for a well-defined modular design of IDSs tailored to specific environments and traffic-stream types. Farid and Rahman [2010] compared Genetic Algorithms to other approaches such as Naive Bayes Classifiers and K-Nearest Neighbors.

Yu et al. [2008] take a similar, but more flexible approach. Since attacks, hardware, software, and operators change over time, using a static dataset for training loses efficiency the longer the training data is use. They propose a system that tunes the detection model on-the-fly based on two factors. The first factor considers how many of the generated alarms the operator can handle, and throttles alarm output to match the operator's pace. The second factor tunes the alarm aggressiveness (sensitivity) based on the accuracy of previous predictions, also determined by operator feedback.

Reading traffic streams for anomalous behavior is usually accomplished by looking for statistical outliers – for example using entropy. Entropy is the measure of uncertainty associated with a random variable [Mitchell, 1997]. Traffic flow source and destination, as well as flow size distributions, can be measured for the level of deviation from the norm to detect outliers. Xu et al. [2007, 2005, 2008] have shown that general user profiles can be built by finding the traffic patterns unique to specific hosts using entropy-based approximation to single out signal noise, regardless of sample size or duration. Using efficient data structures and algorithms, their system efficiently deals with large amounts of information on high-speed links with minimal resource overhead, while establishing new profiles as the network users evolve their behaviors.

Other explored methods used for traffic classification include Decision Trees [Chairunnisa et al., 2009] and Plan Recognition Trees [Geib and Goldman, 2009]. Stolfo et al. [2006] developed a method of social hierarchy mapping using a combination of traffic data mining techniques and social psychology. Stolfo's Email Mining Toolkit method uses communication style and rate measurements to create social clique behavior profiles aimed specifically at email use monitoring and forensics investigation. This method, while powerful, is limited in the sense that it requires unhindered access to the communications logs used to generate the profiles.

2.4 Dataset Pre-Processing

Feature selection, also known as *dimensionality reduction*, is a pre-processing step that is applied to machine learning datasets before it is used in the training and testing of a model [Mukkamala and Sung, 2003; Kloft et al., 2008; Khor et al., 2009]. One of the primary motivations, when using extremely large datasets is to remove the features that may have little to no statistical correlation to each training instance classification value, and thus have fewer features to process. This reduction in the number of features to process intuitively will increase the performance of the training model in terms of total training run time, as there are fewer correlations to consider. Another motivation behind feature selection is in the level of generalization by a model [Refaeilzadeh et al., 2000]. When too many features are considered, there exists the possibility of over-fitting the training dataset [Mitchell, 1997]. When over-fitting occurs, the prediction accuracy will appear very high when testing against similar datasets, however if new observations are made in a test set that don't fit the training data well, prediction accuracy will decrease, and overall generalizability of the trained model is reduced.

Pre-processing generally falls into two categories: *filter*, and *wrapper*. A filter pre-processor uses intrinsic features of the data in order to find an optimal subset of features that better represent the dataset [Guyon and Elisseeff, 2003]. The variables are in turn assigned a score, ranked according to their score, and the variables with the highest scores are selected to represent the feature space. Guyon et al. point out that this method may have the tendency to select redundant features: even though the redundant features contribute to the ability to model the feature space, their presence limits the algorithm from selecting other non-redundant features that could make a better overall contribution.

A wrapper pre-processor uses a classifier on a limited amount of data in order to evaluate which of the features contribute the most to overall classification accuracy [Mukkamala and Sung, 2003; Kloft et al., 2008; Khor et al., 2009]. This approach is more expensive, and can be biased towards the classifier that is being used if the classifier is not treated as a *black box* [Guyon and Elisseeff, 2003]; a-priori knowledge of the internals of the classifier must not influence the data supplied to it. The positive trade off to this approach, however, is that it may produce better performance on the resulting dataset [Guyon and Elisseeff, 2003]. Guyon et al. point out, however, that attempting to find all combinations of sub-features is considered to an *NP-hard* problem and is not practical in real world use; it is computationally intractable. Efficient search strategies have been devised to significantly reduce computational overhead, while contributing a great deal to identifying the most relevant features [Bermejo et al., 2010].

2.5 Machine Learning Methods

The following section will describe various approaches to machine learning. The data sets previously described will sometimes have human involvement to decide whether or not the data falls within specific classifications, or the algorithms are trusted to decide how many classes there are, and which instances will belong to which class.

2.5.1 Supervised vs Unsupervised Learning

The most common machine learning problems fall into two categories; namely supervised, and unsupervised learning [Mitchell, 1997]. Supervised learning generally consists of a supplied data sets that includes the correct classification of each instance. Supplying the correct values to the learning function allows the trainer to supervise the algorithm, which learns a model that represents the complete set of observed instances and hopefully provides the ability to classify new, or previously unseen, instances correctly.

In contrast to supervised learning, unsupervised learning algorithms are supplied the training datasets with an unknown classification, typically because no known classification exists [Zanero and Savaresi, 2004]. Unsupervised algorithms, typically clustering algorithms (such as K-means), are designed to look for feature clusters, or groups and patterns of features that have some sort of correlation, or relationship. These relationships can then be used to determine where a future dataset may fit into the model developed from the previously supplied instances.

2.5.2 Supervised Learning vs Anomaly Detection

Anomaly detection is typically used when the features presented in the dataset fit nicely into a Gaussian distribution (i.e., a Normal probability distribution), and the features belonging to the positive class(es) are statistical outliers. The two main differences between anomaly detection and standard supervised learning are the ratio of positive to negative class examples, and the types of classes found in the data sets [Farid and Rahman, 2010].

For example, in anomaly detection there is a very low number of positive examples (suspicious traffic) found within thousands more negative examples (normal traffic). Conversely, standard supervised learning would typically be expected to have relatively even number of positive and negative examples (such as my training and testing data sets explained in Chapter 3). It is worth noting the number of anomalous cases could be artificially increased to balance the training sets for model creation.

In the context of fraud detection, normal behavior is considered to have a very high probability of occurring, whereas abnormal (or undesirable) behavior has a very low probability of occurring. If the features that describe the behavior in question can be normalized into a normal probability distribution, then it is possible to detect anomalous features that are unusual in nature. A realistic example of this would be polymorphic viruses, that one would hardly see occurring more than a small number of times in order to evade signature-matching detection algorithms that have already been trained to look for malicious patterns found in flow payloads [Bishop, 2005].

This presents a difficulty when training machine learning algorithms, as some attack types do not necessarily follow the same pattern from a payload perspective, yet the net behavior of the malware will generally perform the same series of actions. In this sense, traditional anomaly detection algorithms may not work as desired, since attack behaviors do not change over time; only the payload does. Attack behaviors on the other hand will stay relatively consistent, and enough positive examples should be present for the algorithm to get a sense of what a typical positive example will look like; thus a future positive example is likely to look similar to the training set, and the algorithm can be said to *generalize* fairly well to the problem domain.

2.6 Evaluation Metrics

In machine learning and statistics, there is a variety of methods used to evaluate the performance of a classifier. This section will introduce and discuss the most common evaluation metrics; the ideas of true and false positive and negative classifications, how they contribute to precision and recall, the F-Score, and finally the Receiver Operating Characteristic (ROC) curve. Training and testing paradigms such as holdout and cross-validation will also be examined for applicability to pre-processing, training, and testing.

2.6.1 Algorithm Validation

According to Refaeilzadeh et al. [2009] a number of validation techniques have been developed over time, such as Resubstitution Validation, Hold-Out Validation, K-Fold Cross-Validation, Leave-One-Out Cross-Validation, and Repeated K-Fold Cross-Validation. The most widely used form of cross validation for model parameter tuning is *k-fold* cross validation. Cross Validation is a method used to compare the performance of two or more machine learning algorithms, while relying on the same pool of labeled test set data for both training and testing, yet doing so in a statistically significant manner [Refaeilzadeh et al., 2009]. This is done such that an indicator of algorithm accuracy can be obtained without using unseen test set data, so that only the best hypothesis model configurations can be tested against the designated testing set. K-fold cross validation involves dividing up the test dataset into k partitions, with each partition serving as the test set on one training experiment, while the rest of the data is used as the training set to create the hypothesis model Mitchell [1997]. The classification performance across all test sets is then averaged to find the mean error. 10-fold cross validation remains the most widely-accepted method of comparison in the Machine Learning community [Refaeilzadeh et al., 2009].

The correct form of cross validation should yield a close approximation to the classification accuracy found on a new, unseen testing set, provided the sampling methodology provides a close approximation to the real world distribution. This does not hold true, however, if the unseen test set contains enough instances found outside of the previously trained hypothesis space.

The application of cross validation techniques to real world problems can be driven by multiple goals. Some of these goals include algorithm performance estimation, tuning model parameters, and ultimately model selection [Refaeilzadeh et al., 2009].

Hold-out validation [Refaeilzadeh et al., 2009] is an alternate method for algorithm testing, typically used on algorithms whose parameters have been previously fixed. The premise behind this testing is that training and preliminary testing will have already occurred (typically through a training and validation set) with a holdout dataset reserved for real world prediction estimation. The advantages of this method include a purely independent training and testing dataset – however this can be an issue when dealing with already small data sets, as well as a large variance in estimated performance.

As described in Chapter 3, I will be using a combination of Cross-Valation and Holdout-Validation techniques. Cross-Validation will be used to determine the most suitable features for data pre-processing, and Holdout-Validation (with novel attacks) will be used for testing each of the selected algorithms.

2.6.2 Confusion Matrices

Anomaly detection can be thought of as binary classification. The traffic is either normal or abnormal. If a single prediction has two possible outcomes, and each can be correct or incorrect, there are thus four possible combinations of predictions and actual outcomes. Based on this, one way to view the performance of a classification algorithm, is through a *confusion matrix*, also known as a *contingency table* [Powers, 2007]. Considering the anomaly dataset, there are two classes for any prediction: normal and anomaly. If we consider the classes and the correctness of each, we arrive at a 2x2 matrix, as shown in Table 2.1.

Table 2.1: Example Confusion Matrix.

a	b	classified as
TP	FP	a = anomaly
FN	TN	b = normal

In this case, the first row represents all of the actual normal classified instances, the second row represents the actual anomalous test instances, and each column represents how those instances were classified during prediction. Reading into this table, all correctly classified instances are in the top-left and bottom-right corners, whereas the incorrectly classified instances are in the bottom-left or top-right corners, and the total of all entries represents the cardinality of the entire dataset. In the case of intrusion detection, as described above, a traffic anomaly is a *positive* sample (the sample we are interested in), whereas normal traffic is *negative*.

Considering the confusion matrix in Table 2.1, if the actual class is normal, and the predicted value is normal, then we have a *true negative*. In practice, we desire this value to be as high as possible, as it signals everything is as it should be.

If the actual class is normal, and the predicted class is anomaly, we have a *false positive*. In practice, we want this value to be as low as possible, as it represents traffic that is normal, but was mistakenly classified as anomalous; these are the bane of network administrators as they eat up valuable resources investigating traffic only to realize there is nothing wrong.

If the actual class is anomaly, and the predicted value is normal, then we have a *false negative*. Just like false positives, we want this value to be as low as possible, as it represents traffic that is not normal, but was mistakenly classified as normal; possibly attacks that went “under the radar” so-to-speak.

Finally, if the actual class is anomaly, and the predicted value is anomaly, then we have a *true positive*. These represent anomalies (possibly malicious traffic) that were correctly flagged as such, and warrant either further investigation or outright blocking, as they do no conform to some preset behavioral policy. In practice, although we do not wish for very many of these, fraud and intrusion attacks are a reality and hence we wish to catch as many as we can.

Based on the above description, we can define the *False Positive Rate* as

$$FPRate = \frac{FP}{Nn}$$

where FP is the number of false positives observed, and Nn is the total number of observed instances in the negative class.

As well, we can define the *True Positive Rate* as

$$TPRate = \frac{TP}{Np}$$

where TP is the number of true positives observed, and Np is the total number of observed instances in the positive class.

These values are interesting in and of themselves; however there are more inclusive methods of calculating the relationship between these rates, as described in the next section.

2.6.3 Recall

Recall (or *Sensitivity* in other domains) is a measurement method found in data mining used to demonstrate a model's ability to pick up relevant positive instances in a dataset [Powers, 2007; Fawcett, 2004]. The Recall ability of a model can be calculated as

$$Recall = \frac{TP}{TP + FN}$$

where FN is the number of false negatives observed. Since this value can be used to demonstrate how good the model is at picking out positives, if this was the only measurement being considered, then the model could cheat by always predicting the positive class. In the case of the Zero-R algorithm, this would happen if the majority of training instances seen were positive.

2.6.4 Specificity

Specificity is a measurement method found in data mining used to demonstrate a model's ability to pick up relavent negative instances in a dataset [Powers, 2007; Fawcett, 2004]. The Specificity of a model can be calculated as

$$Specificity = \frac{TN}{TN + FP}$$

Just as with Recall, if this was the only measurement being considered, then the model could cheat by always predicting the negative class. In the case of the Zero-R algorithm, this would happen if the majority of training instances seen were negative.

2.6.5 Precision

Precision (or *Confidence* in other domains) is a measurement method found in data mining used to demonstrate the tradeoff in a model between the sensitivity of picking up true positives, while balancing the false positives. [Powers, 2007; Fawcett, 2004]. The Precision ability of a model can be calculated as

$$Precision = \frac{TP}{TP + FP}$$

This test is different from the other two as it is dependent on the proportion of positive examples seen in the test set.

2.6.6 F-Score

The relationship between Precision and Recall, as previously observed, is not mutually exclusive. There is a tradeoff between one and the other. The *F-Score* is a method used in machine learning to compute the harmonized mean between the previously defined Precision and Recall values of a model [Powers, 2007; Fawcett, 2004]. This score can be calculated as

$$F - Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

and is typically used to determine the average between the two results. This metric can be valuable in calculating the overall accuracy of a model, however it has been shown that this method of calculation does not take the true negative rate into account [Powers, 2007].

2.6.7 Receiver Operating Characteristic

The *Receiver Operating Characteristic* (ROC) curve is a method of measuring the accuracy of a binary classifier by using the Recall (sensitivity) and the False Positive rate ($1-Specificity$) as defined above [Fawcett, 2004]. This form of measurement has been shown to be a more complete method than other traditional scoring algorithms, at it takes all of the true/false positive/negative combinations into account [Fawcett, 2004; Walter, 2005]. The ROC curve, when considering a binary classifier, can be visualized as a two dimensional graph, depicting the ratio between the true positive rate and the false positive rate.

Considering point B in Figure 2.1 [Wikimedia, 2009], any model with a threshold value of 0.5 would be considered on par with random guessing. A model, such as that depicted in point C (lower-right), would have a higher false positive rate than true positive rate, and would be considered worse than random guessing. Alternately, points A and C' (*C-prime* in the top-left being a mirror of *C* in the bottom-right) successively approach what would be considered a *perfect* classifier, with consideration to all positive and negative classes.

The *Area under the ROC curve* (AUC) can be calculated in a step-wise fashion, by considering all of the instances seen, and plotting their TPR-to-FPR ratio as further instances are observed during the testing phase. These points can then be connected, and the area under this curve taken to form the AUC value. In practice, all of the classifiers that perform better than random guessing will have an AUC between 0.5 and 1.0.

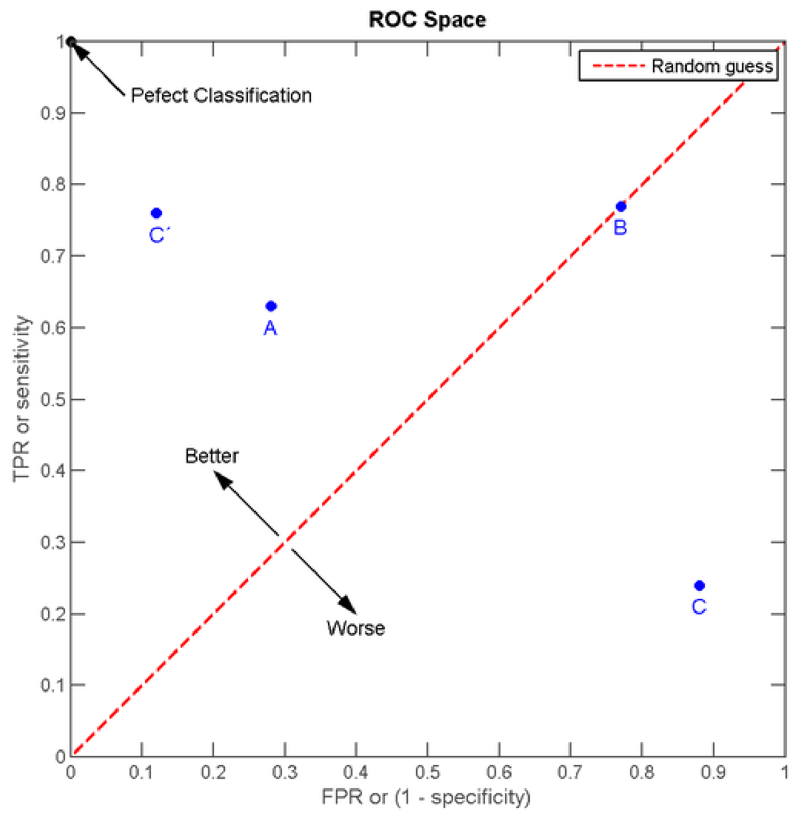


Figure 2.1: Receiver Operating Characteristic

2.7 Machine Learning Algorithms

Having examined the methods by which we evaluate the outcomes of classification, we can now turn to examining machine learning algorithms themselves. The machine learning algorithms described below will form the basis for the experiments performed in my thesis. The formulas in consideration will provide a basis for the specific implementations used, with some consideration to how they are applied in the context of my work.

2.7.1 ZeroR

One of the simplest classifiers is the *ZeroR* [Mitchell, 1997] , or *0-R*, which is typically used as a baseline classifier against which other, more complicated, classifiers are measured. The ZeroR classifier is unique in that it does not actually take into account the features or attributes of the training data used to build the model. In the case of supervised learning, ZeroR calculates the average value of the supplied class (when numeric) or the mode of the supplied class (when nominal). This value is then used every time a new test instance is given to the model.

Intuitively, the model prediction will have an accuracy roughly the same as the most common class found in the training data, provided the training, holdout, and cross validation random sampling contains the same distribution of classes. In the my experimental test set, the attack types have been simply labeled as *normal* and *anomaly.*

2.7.2 Naive Bayes

The Naive Bayes [Mitchell, 1997] algorithm uses *Bayes' Theorem* to predict the *posterior probability* of a class, given the presence of one or more features. Bayes' Theorem can be shown as

$$P(c|x) = \frac{P(x|c)P(c)}{P(x)} \text{ [1]}$$

[Mitchell, 1997]

where $P(x|c)$ is the likelihood of feature x given class c, P(c) is the probability of class c, and $P(x)$ is the probability of feature x occurring. This algorithm is said to be *naive* because of the strong assumption that features are independent from one another in a probabilistic sense. Despite this limitation, compared to other more sophisticated algorithms, some researchers [iri, 2001] have found that it is fairly competitive in predictive performance in general.

2.7.2.1 Learning Algorithm

When considering multiple features, calculating the posterior probability can be stated as follows: given predictors $x_1, x_2, ..., x_n$ calculate the probability of the instance belonging to each class, and then select the class with the highest probability. Let c be a specific class, and X be the set of the features observed in the current instance. This can then be shown as

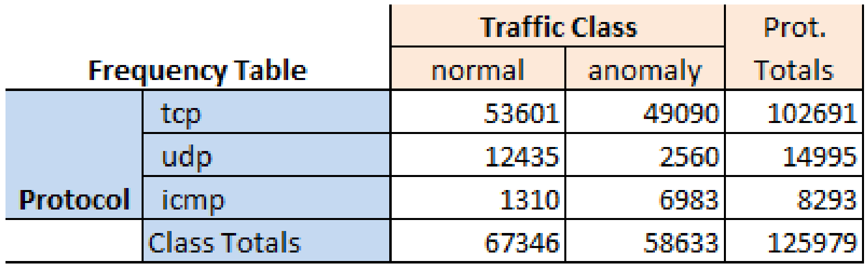
$$P(c|X) = P(x_1|c) \times P(x_2|c) \times ... \times P(x_n|c) \times P(c) \text{ [2]}$$

[Mitchell, 1997]

This assumes that the effect of a feature x on any class c is independent of the effect of other features on class c. In order to translate the trained feature space to testing class prediction, the data must be transformed into a frequency table for each feature, and then to a likelihood table.

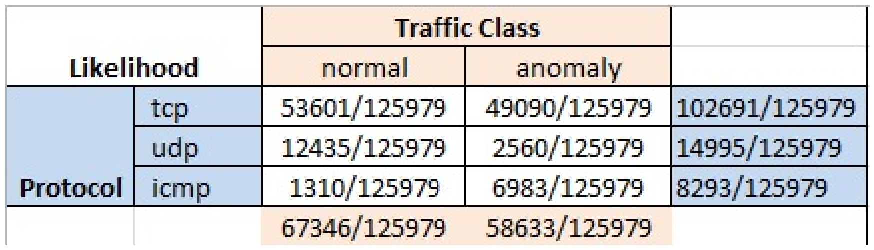
Table 2.2 illustrates a sample breakdown of traffic classes by netflow protocol found in all the instances used to train the model. Using this data, we can construct the total frequencies for each feature against the classes. These totals can then be used to create the likelihood table, Table 2.3:

Based on Table 2.3, we can calculate the probably of the traffic being normal



Frequency Table		Traffic Class		Prot.
		normal	anomaly	Totals
Protocol	tcp	53601	49090	102691
	udp	12435	2560	14995
	icmp	1310	6983	8293
	Class Totals	67346	58633	125979

Table 2.2: Traffic Frequency Table



Likelihood		Traffic Class		
		normal	anomaly	
Protocol	tcp	53601/125979	49090/125979	102691/125979
	udp	12435/125979	2560/125979	14995/125979
	icmp	1310/125979	6983/125979	8293/125979
		67346/125979	58633/125979	

Table 2.3: Traffic Likelihood Table

if the netflow protocol is tcp. The probability of class normal, $P(c) = P(normal)$, is approximately $67346/125979 = 0.53$. The probably of seeing the tcp protocol, $P(x) = P(tcp)$, is approximately $102691/125979 = 0.86$. And finally, the probability of class normal given the presence of protocol feature tcp, or $P(c|x = P(normal|tcp)$ is approximately $53601/125979 = 0.43$. These can now be combined, as seen above in formula 1, to calculate the *posterior probability*, as $P(c|x) = P(normal|tcp) = \frac{0.43 \times 0.53}{0.86} = 0.27$.

2.7.3 J48 Decision Tree

Decision trees are used to classify data by sorting the attributes and their values in a tree fashion, with the leaves of the tree being the classification. The most basic example of decision tree algorithms are the ID3 and C4.5 [Quinlan, 1993] which uses entropy and information gain (see Section 3.3.1.1) to recursively split the available attributes into decisions that will decrease the entropy across all values associated with that attribute. The result is such that the highest information gain split is at the root node, all the way down to the values with the lowest near to the leaves.

2.7.3.1 Learning Algorithm

The C4.5 decision tree algorithm is based on the assumption that the features in question are not uniformly distributed across all classifications. When the observed value of a feature is uniformly distributed in a probability table, it is said to have a low level of entropy (randomness) [Kotsiantis, 2007]. Considering Figure 2.2, if the probability that variable X is 0.5, then that trial is said to have the highest degree of uncertainty (e.g., flipping a fair coin), and therefore the $H(X)$, or level of entropy, approaches 1.

Consider a training dataset $S = s_1, s_2, ...$ of classified instances. Each of the instances contains a list of features in a p-dimensional vector $x_{1,i}, x_{2,i}, ..., x_{p,i}$, such that each x_j is the list of the attributes of that instances, as well the class s_i belongs to [Kotsiantis, 2007]. At each node of the tree, all of the attributes are tested for their individual *information gain* (contribution to classification), and the attributes are then split into their respective subsets depending on which classification they belong to [Kotsiantis, 2007]. This is then recursively run on each successive list derived from the original split until a base case is met [Kotsiantis, 2007]. The base cases in consideration are [Kotsiantis, 2007]

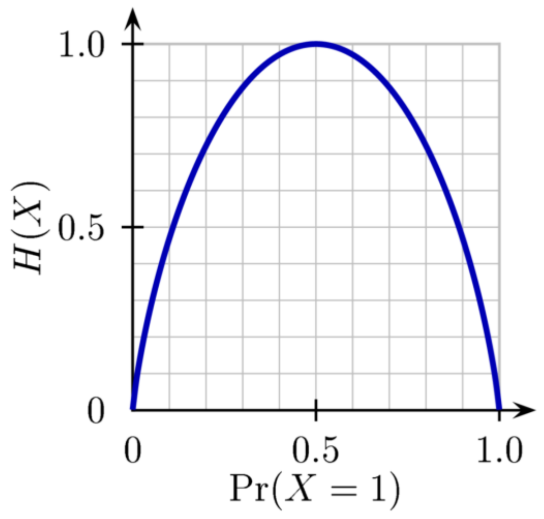


Figure 2.2: Binary Entropy Plot [Wikimedia, 2007a]

1. All of the instances in the list belong to the same class. A leaf node is then created to choose that class.

2. All of the features in the sample have the same entropy. Create a decision node higher up using the value of that class.

3. An instance of a previously unseen class is encountered. Create a decision node higher up using the value of that class.

The pseudo code for creating a decision tree, taken from [Kotsiantis, 2007], can be shown as:

Algorithm 1 J48 Decision Tree

```
$best\_attribute \leftarrow 0$
Check for a base case
for each attribute $a$ in the set of attributes do
    Find the normalized information gain from splitting the sublist by that attribute.
    if $a >$ $best\_attribute$ then
        $best\_attribute \leftarrow a$
    else
        get the next attribute $a$
    end if
end for
Create a decision tree node that splits the set on $best\_attribute$
Recursively create the tree by calling J48 on the sublists for each branch
```

Once the decision tree is created, the list of p-dimensional attributes are then compared to each node in the decision tree, until a leaf is hit, thus classifying that instance [Kotsiantis, 2007].

2.7.4 Linear and Logistic Regression

Linear regression, as the name specifies, tries to find a function that intersects the average of all supplied data points in order to predict a real-valued output (as opposed to discrete valued output) [Mitchell, 1997]. The algorithm can be supervised, as the training data in this case also supplies the known class (correct answer). Consider the following variables:

1. m : the number of training examples
2. x : the input variables, or features of a training set
3. y : the output variables, or target classification value

In the case of univariate linear regression, a single training instance could be denoted (x, y), where x is the input value, and y is the associated output value. In the case of multiple training examples, the i^{th} instance would be $(x,^{(i)}, y^{(i)})$ [Mitchell, 1997].

The hypothesis function, traditionally denoted $h(x)$, represents our model of the input and output variables [Mitchell, 1997]. The intention of the algorithm is to train the model such that for any one or more input values $x_0, x_1, +...+, x_n$, the accurate output value y will be given [Mitchell, 1997].

Each training set is given to the learning algorithm, that is responsible for generating a hypothesis function h [Mitchell, 1997]. When new instances of x are given to function h, the function will hopefully output a prediction value y that is as close possible to the correct outcome, should such a relationship exist between input value x and output value y [Mitchell, 1997].

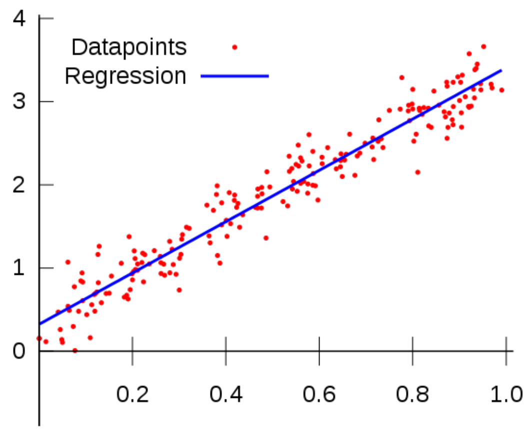


Figure 2.3: Linear Regression [Wikimedia, 2007c]

As seen in Figure 2.3 the line intersects as many of the instances as possible such that given a new value x, the proper prediction of y can be generated.

2.7.4.1 Cost Function

Considering the θ parameters of the hypothesis function, how do we choose θ_0 and θ_1 such that $h_\theta(x)$ is as close to y for all (x, y)?

In the case of a binary classifier, the segmentation would ideally separate the two classes we are interested in. The main question is, how do we choose the parameter values for θ_0 and θ_1?. This is done with a formula called the *cost function*[Mitchell, 1997]. The idea is to choose the values for θ_0 and θ_1 so that the hypothesis function is close to y for all of our training examples *(x,y)*. Such that we minimize [Ng, 2012]

$$\Theta_0, \Theta_1 = \sum_{i=1}^{m} (h_0(x)^i - y(i))^2$$

The cost function is calculated based on the Squared Error between the expected value y given input x, and the real value [Mitchell, 1997]. This error function has been known to work well with most regression problems, and is found in many real-world applications [Friedman et al., 1998]. It is also considered to be optimal in the Bayesian-probabilistic sense Mitchell [1997].

By this we denote the cost function [Ng, 2012]

$$J(\theta_0, \theta_1) = 1/(2m) \sum_{i=1}^{m} (h_\theta(x) - y)^2 (1)$$

This equation is known as the *squared error cost function* [Mitchell, 1997], and is mostly used for regression problems over one or more variables [Ng, 2012]. As the input values are changed, and the value of $J(\theta_1)$ goes as close to 0 as possible, the model is said to be accurate with respect to the training data. This method is also known as *Gradient Descent* when trying to find a global optimum (minimal error) [Mitchell, 1997].

2.7.4.2 Gradient Descent

Gradient Descent is the method by which the learning algorithm attempts to find a global minimum to the cost function [Mitchell, 1997]. This is performed over the error gradient of the possible input space. If we consider the following equation [Ng, 2012]

$$\theta_j := \theta_j - \alpha(\delta/(\delta\theta_j))J(\theta_0, \theta_1)$$

we can see the value being assigned to θ is the current value of θ minus the partial derivative of the adjusted cost function, at the rate of α. The α value is known as the *learning rate*; a factor determining the amount of change during each iteration of the gradient descent algorithm. Intuitively, if the value of α is too high, then the value may oscillate across the desired global minimum (or optimal value of θ). Conversely, if the value of α is too small, the algorithm will have undesirable performance as the iterative steps are very small [Mitchell, 1997].

Substituting $J(\theta_0, \theta_1)$ from the cost function gives us [Ng, 2012]

$$\theta_j := \theta_j - \alpha(\delta/(\delta\theta_j))1/(2m)\sum_{i=1}^{m}(h_\theta(x) - y)^2(1)$$

This is also known as a *convex function*, as gradient descent is always a "bow" shape, and converges to a global optimum [Mitchell, 1997].

In the case of multiple variables, we want to update $\theta_0, \theta_1, ..., \theta_n$ *simultaneously*, otherwise each computation of the error rate will use the partially changed variables [Ng, 2012]

$$\theta_j := \theta_j - \alpha(\delta/(\delta\theta_j))J(\theta_0, \theta_1, ..., \theta_n)$$

Linear regression is a simple algorithm that can be used to derive the numeric value along multiple feature spaces, however in most forms it is limited to dealing with regression (real valued) predictions, and is not well suited for classification problems which we are interested in [Ng, 2012]. As we will now see, a slight modification to the training function can create a fairly powerful classification learner called *logistic regression* [Mitchell, 1997].

2.7.4.3 Logistic Regression Cost Function

Logistic regression is a fairly popular algorithm used in classification, and can handle both binary and multiple classes [Mitchell, 1997]. In this case we want to predict $y\epsilon\{0, 1\}$ for the binary class, based on the input x, using a threshold classifier:

if $h_0(x) \geq 0.5$, then $y = 1$.

To accomplish this, logistic regression will apply the sigmoid function. The sigmoid function used in place of the hypothesis function [Ng, 2012]

$$h_0(x) = 1/(1 + e^{-\Theta^T x})$$

The term x is controlled for each input feature. Considering the number of input features for our dataset is relatively low (less than ten thousand), the Normal equation can be used, which solves for each co-efficient in one step by using matrix algebra, as opposed to the traditional method of trying to find a global minimum through gradient descent, and testing multiple learning rates to avoid oscillation [Mitchell, 1997; Ng, 2012].

As shown in Figure 2.4, the logistic regression cost function has a sigmoid curve.

In the case of applying Logistic Regression to multiple classes (for example, two classes), we can consider the predicted value y being an element of either 0, or 1, where 0 is the negative class, and 1 is the positive class [Mitchell, 1997; Ng, 2012].

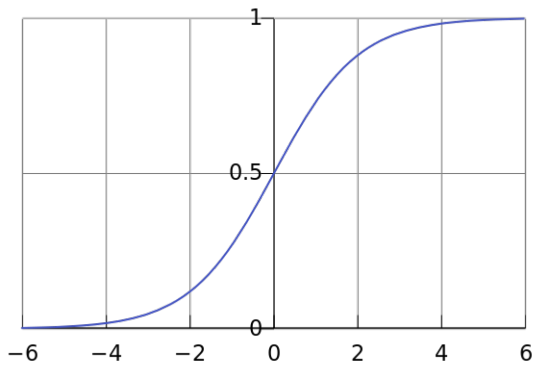


Figure 2.4: Logistic Curve [Wikimedia, 2008a]

For this we use a threshold classifier, such that our hypothesis function $h_\sigma(x) >= 0.5$ if $y = 1$, and $h_\sigma(x) <= 0.5$ if $y = 0$ [Mitchell, 1997; Ng, 2012]. In other words, $0 <= h_\sigma(x) <= 1$ [Mitchell, 1997; Ng, 2012]. We can compute this function using the *logistic function*, also known as the sigmoid function, in the context of the hypothesis function [Mitchell, 1997; Ng, 2012].

The logistic regression function $h(x)$ is estimating the probability that $y = 1$ based on the input features [Mitchell, 1997; Ng, 2012]. Non-linear predictions can be obtained by using polynomial or higher order values for each feature [Mitchell, 1997; Ng, 2012]. This can be accomplished in a number of ways, including higher-order approximation on the same feature, and combining features together to create a new feature altogether [Mitchell, 1997; Ng, 2012].

The addition of higher order or new features will have the tendency to make the model more accurate [Ng, 2012], however this may also have the potential danger of causing over-fitting of the training data, should too many higher-order features be added to the model [Mitchell, 1997; Ng, 2012].

The cost function for logistic regression is used to automatically fit the parameters θ for each input variable [Mitchell, 1997; Ng, 2012]. It can be shown that the Square Mean Error, when applied in a logistic fashion, is then [Mitchell, 1997; Ng, 2012]

1. $Cost(h_0(x), y) = -log(h_0(x)\ if\ y = 1$
2. $Cost(h_0(x), y) = -log(1 - h_0(x)\ if\ y = 0$

This can also be written as [Ng, 2012]

$$Cost(h_0(x), y) = -y \cdot log(h_0(x) - (1 - y) \cdot log(1 - h_0(x))$$

2.7.4.4 Logistic Regularization

Both linear and logistic regression can suffer from *under-fitting* (also known as a *high bias*), as it assumes the relationships are usually linear, and will not consider outliers very well [Mitchell, 1997; Ng, 2012]. On the other hand, using higher-order polynomials (or quadratics), the model can cause *over-fitting* or *high variance* and fit the training set too well [Mitchell, 1997; Ng, 2012]. Over-fitting the training set then limits the model's ability to generalize to new test instances [Mitchell, 1997; Ng, 2012]. There are two general approaches to solving this. The first approach is to reduce the overall number of features (manually, or automatically using feature selection algorithms). The second approach is to keep all of the features, but use a regularization value to reduce the weight of each one, so each one contributes a little bit to predicting the class [Mitchell, 1997; Ng, 2012].

2.7.5 K-Nearest Neighbour

The K-nearest-neighbor algorithm is an instance-based learning algorithm that relies on the distance of each new sample to one or more previously seen instances in a feature space for classification [Aha et al., 1991].

2.7.5.1 Algorithm

There are a variety of methods to calculate the distance between the new instances, and the closest neighbours by proximity, including Euclidean, Manhattan, and Minkowski [Aha et al., 1991]. The proximity distance calculation used in this experiment is the Euclidean, given [Aha et al., 1991]

$$\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

where n is the number of dimensions in the feature space for each point, and x and y are the two points for which the distance is being calculated.

As seen in Figure 2.5, the new instance (green) must be classified. Based on immediate proximity, the red class is closer than the blue class. However, based on a larger radial diameter, there are more blue instances than red. Depending on the k value, or number of nearest neighbours, the class of the newest instance may vary.

The issue when creating a feature space is whether or not the features can be put in the same scale [Aha et al., 1991; Chairunnisa et al., 2009]. This is a bigger issue for nominal attributes that numeric. Nominal attributes can first be put into a numeric continuum, accomplished through encoding them by intervals [Aha et al., 1991; Chairunnisa et al., 2009]. All numeric attributes can then normalized to fit the

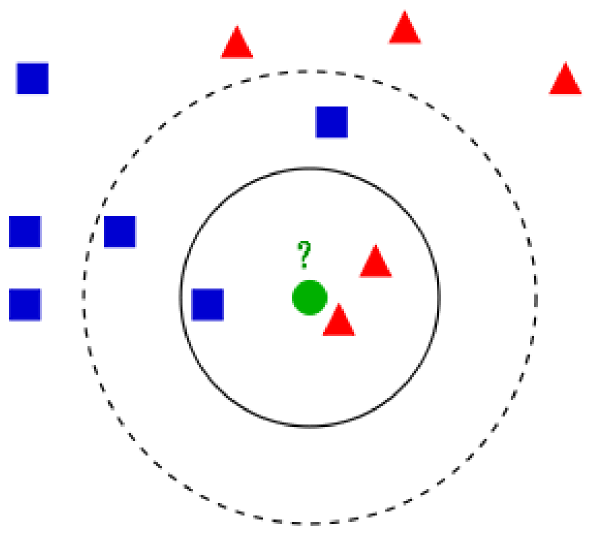


Figure 2.5: K-Nearest Neighbour Classification [Wikimedia, 2007b]

same scale.

2.7.6 Support Vector Machines

Support Vector Machines involve finding a plane of separation between two groups of graphed instances such that the plane has a maximum margin between the groups. This is done by finding an $n-1$ dimensional hyperplane in an $n-dimensional$ space to separate the two classes [Fradkin and Muchnik, 2000].

2.7.6.1 Algorithm

Considering two similar groups of instances in a two-dimensional feature space space, Figure 2.6 illustrates three example planes of possible separation.

As shown in the figure, we see two distinctive classes. The *H1* hyperplane does not separate the two groups. The *H2* hyperplane does separate the groups, but just

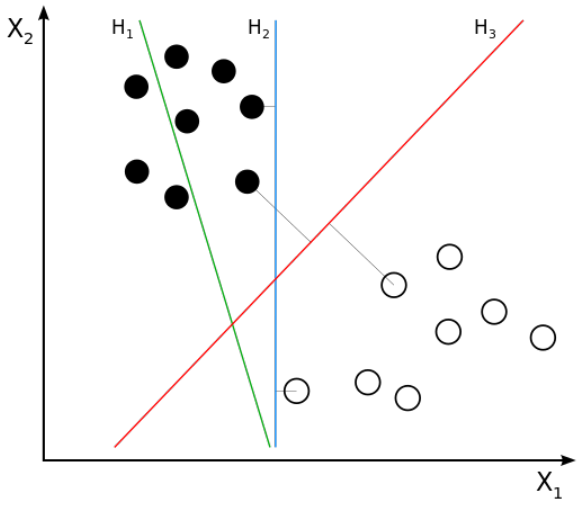


Figure 2.6: SVM Seperating Hyperplanes [Wikimedia, 2012]

barely. The *H3* hyper plan does separate the groups with the largest margin on each side, such that new instances placed within the vicinity of their respective classes would have the greatest chance of being classified correctly.

As shown in Figure 2.7, the selected hyperplane maximizes the margin between the two disparate clusters of instances, and thus provides an optimal separation for future classification of like-instances. This algorithm does have the ability to be more efficient and powerful than both logistic regression and artificial neural networks [Ng, 2012].

For a binary classification support vector machine, let y be the target class of the seen instance, $h_\theta(x)$ be the hypothesis function of instance x. Then

1. if $y = 1$, we want $h_\theta(x)$ to be ≈ 1, and $\theta_x^T >> 0$ [Ng, 2012]

2. if $y = 0$, we want $h_\theta(x)$ to be ≈ 0, and $\theta_x^T << 0$ [Ng, 2012]

Considering the cost function [Ng, 2012]

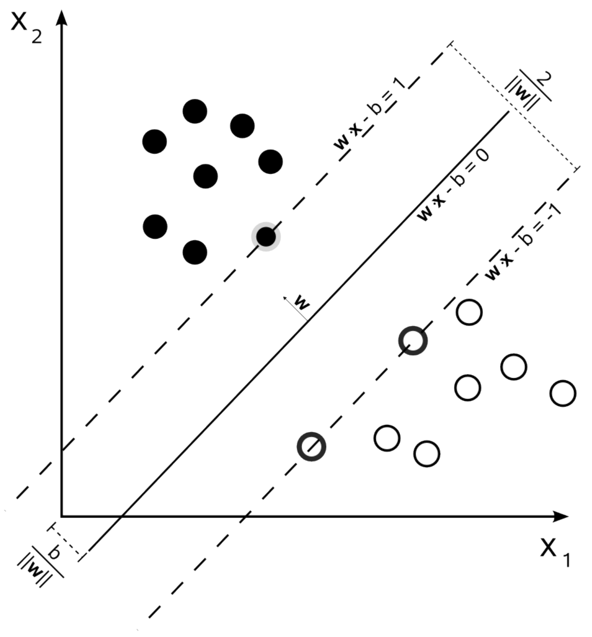


Figure 2.7: SVM Large Margins [Wikimedia, 2008b]

$$J(\theta) = -y \cdot log \frac{1}{1+e^{-\Theta^T x}} - (1-y) \cdot log(1 - \frac{1}{1+e^{-\Theta^T x}})$$

Therefore if $y = 1$ (we want $\theta_x^T >> 0$), the cost function $J(\theta) \approx -log \cdot \frac{1}{1+e^{-\Theta^T x}}$ [Ng, 2012].

Otherwise if $y = 0$ (and we want $\theta_x^T >> 0$), then $J(\theta) \approx -log \cdot (1 + \frac{1}{1+e^{-\Theta^T x}})$ [Ng, 2012].

which gives us the hypothesis [Ng, 2012]

$$h_\theta(x) = 1 \; if \; \theta_x^T >> 0, \; otherwise \; h_\theta(x) = 1$$

Let the cost function $\theta_x^T = z$. This means if the instances in question belongs to class 1, the cost function z should be $>> 1$ (preferably much higher), while the instances belonging to class 0 should produce a cost function $z << 0$ (preferably in the low negatives) [Mitchell, 1997; Ng, 2012; Fradkin and Muchnik, 2000]. Considering the function is logistic, the cost function is roughly $-2 << z << 2$. This is termed the *Large Margin* feature of the support vector, as it attempts to distance itself equally from all instances, and in this example, have the largest Euclidean distance from every point on the Cartesian plane simultaneously [Mitchell, 1997; Ng, 2012; Fradkin and Muchnik, 2000].

2.7.6.2 Kernel Functions

Support Vectors Machines with a large margin work well if all of the features are cleanly separated [Mitchell, 1997; Ng, 2012; Fradkin and Muchnik, 2000]. What about cases in which they overlap, or form strange patterns? This can be solved using a *kernel function* [Mitchell, 1997; Ng, 2012; Fradkin and Muchnik, 2000]. The function can be exacted by two main methods. One is to encompass the feature clusters through a polynomial cost function expression similar to which was seen in logistic regression [Mitchell, 1997; Ng, 2012; Fradkin and Muchnik, 2000]. The other method is to transform the feature space into a higher dimension such that linear separation of the classes is much easier. The kernel function used to perform this dimensional transformation in my experiments, as explained below, is the *radial basis function* assuming a Gaussian distribution when viewing each feature point in a $n-1$ sub-plane to the $n-dimensional$ hyperplane [Mitchell, 1997; Ng, 2012; Fradkin and Muchnik, 2000].

2.7.6.3 Pseudocode

The Support Vector Machine algorithm is typically executed as follows [Mitchell, 1997; Ng, 2012; Fradkin and Muchnik, 2000]:

Algorithm 2 Support Vector Placement

```
Define the maximum hyperplane
Margin Placement: extend the distance to non-linear classifications with a penalty value
Produce the kernel function: map to highest dimension to classify with linear decision surface (kernel function)
```

As described above, the SVM will attempt to find the maximum hyperplane by calculating the distance between classes such that every instance in the plane has the maximum distance from the hyperplane. The feature space is then mapped through a kernel function such that the hyperplane can easily bisect the classes [Mitchell, 1997; Ng, 2012]. Having reviewed the concept of support vector machines, I will now introduce the last of the single algorithms, the artificial neural network.

2.7.7 Artificial Neural Networks

2.7.7.1 Introduction

Modern neural networks are based on the structure of interconnected networks of nodes, or neuron-like structures [Mitchell, 1997; Ali et al., 2010]. The neurons in an artificial neural network are also referred to as *units*, or *nodes* [Mitchell, 1997; Ali et al., 2010]. The simple neuron contains input signals, each signal representing a variable under consideration [Mitchell, 1997; Ali et al., 2010] to be passed through the neuron's function, ultimately producing an output signal [Mitchell, 1997; Ali et al., 2010].

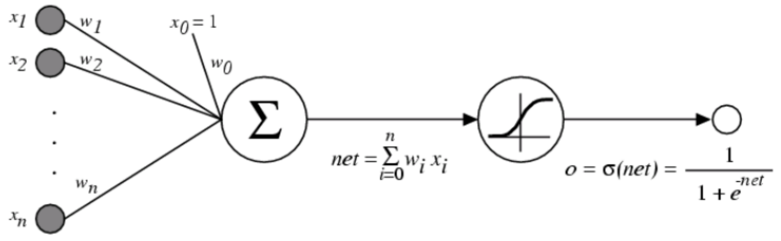

x_1
w_1
x_2
w_2
$x_0 = 1$
w_0
w_n
x_n
Σ
$net = \sum_{i=0}^{n} w_i x_i$
$o = \sigma(net) = \frac{1}{1 + e^{-net}}$

Figure 2.8: Perceptron with Sigmoid Function [Mitchell, 1997]

As demonstrated in figure 2.8, a network which has only one layer is known as a perceptron, and is no different than a single-variable logistic function [Mitchell, 1997; Ali et al., 2010]. The cumulative strength of the neuron's inputs determines what the activation level of the neuron is [Mitchell, 1997]. The node's threshold function $f(net)$ determines what the neuron's output is, depending on how high or low the activation level is above or below the predefined threshold [Mitchell, 1997]. The output of a simple neuron is usually 1 or -1, meaning *yes* or *no* [Mitchell, 1997]. This is a simple model as the cumulative inputs' sign determines the sign of the activation functions result [Mitchell, 1997]. Some neurons have a bias input associated with it, which always has an input of +1 or -1, regardless of what the other inputs are [Mitchell, 1997]. This in turn shifts where the neuron would be activated with regards to the threshold function [Mitchell, 1997].

2.7.7.2 Feed Forward Signal Propagation

Networks characteristics can be defined by the network topology, the learning algorithm used, as well as the encoding scheme to normalize input variables suitable for the neuron function [Mitchell, 1997; Ali et al., 2010]. The network topology is the structured connection between all the neurons in the network [Mitchell, 1997; Ali et al., 2010].

As shown in Figure 2.9, neurons can be combined together to form a network of neurons, with the output values from the input layer (and subsequent layers) serving as the input values for successive layers [Mitchell, 1997; Ali et al., 2010].

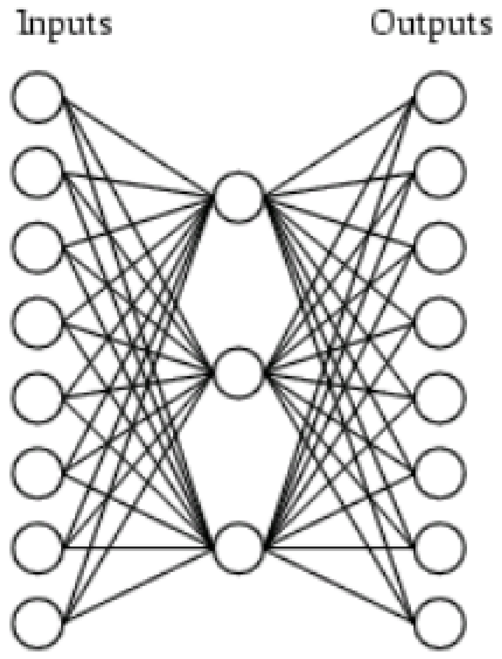


Figure 2.9: Neural Network with Layers [Mitchell, 1997]

As there can be multiple levels in the network, the input values will be calculated by each neuron's function, then signals are successively passed on to a second or even third layers [Mitchell, 1997; Ali et al., 2010]. In most problem domains, including network data classification, the features (or input node values) may not all have equal contribution to the desired output node value (the classification) [Mitchell, 1997].

Each node in a neural network will typically have a weight assigned to it, such that once the network has been trained, the features with the highest information gain will have the highest weights assigned to them [Mitchell, 1997; Ali et al., 2010]. Thus, for each input x_i a weight $w_i, (i = 1...n)$ is assigned which describes the connection strength of each input [Mitchell, 1997; Ali et al., 2010]. When the variables are being calculated for each layer (*feed forward propagation*), the function calculations will eventually reach the output node(s) [Mitchell, 1997; Ali et al., 2010].

As described above, feed forward propagation will calculate an output classification based on the input values supplied. However, in order to modify the model, the connections between the neurons need to be adjusted [Mitchell, 1997], as described in the next section.

2.7.7.3 Back Propagation Learning Algorithm

The *back propagation* learning algorithm used for neural networks determines how the weights are adjusted after each iteration of training [Mitchell, 1997]. If the network produces a desired output, the weights need not change. However, if the output is not correct, the weights of each node must be modified depending on multiple factors [Mitchell, 1997]. These connections are strengthened through the process of back propagation [Mitchell, 1997].

Let c be a constant whose size determines the *learning rate* and d be the desired output value. The adjustment for the weight on the *ith* component of the input vector, Δw_i, is given by [Mitchell, 1997]:

$$\Delta w_i = c \cdot (d - \sum x_i \cdot w_i) \cdot x_i$$

The change in weight is affected by the neurons last known output value $\sum x_i \cdot w_i$, which is the signal produced by applying the sum of all inputs i (from $1..n$) multiplied by their respective weights to the threshold function [Mitchell, 1997]. This value is then subtracted from the desired output d, and multiplied by the adjustment constant c, and the last input value x_i [Mitchell, 1997]. In simpler terms, the pseudocode to change for the weight of input i(from [Mitchell, 1997]) is:

$$c * (desired\ output\ - last\ output\ of\ neuron) * (last\ input\ of\ i))$$

This change in weight is then repeated for every layer in the neural network, up to and including the input layer [Mitchell, 1997]. One round of this back propagation training algorithm is known as an *epoch*, and during the training of the network, will typically be repeated until the gradient descent cost function (as described in Logistic Regression) converges on a local (and hopefully global) minimum [Mitchell, 1997].

2.7.8 Summary

This section has described and considered the single learning algorithms that were used in my research. The next section will consider ensemble learning, and meta-classification using both homogeneous and heterogeneous models, and how this can be applied to classifying data for network security.

2.8 Meta-Heuristic Learning Algorithms

The single learning algorithms in the previous chapter were all designed to search a hypotheses space for a good hypothesis that fit the problem domain, and would allow for hopefully better-than-average prediction. The term *ensemble* [Zhou, 2009] in learning generally means using more than one prediction model in order to combine multiple hypothesis spaces in the problem domain to increase the overall prediction power.

Boosting and Bagging algorithms work by training multiple simple classifiers to focus on different aspects of the dataset, with the intention of improving the overall accuracy [Schapire, 1990; Witten et al., 2011].

Voting algorithms work in a similar manner by using multiple disparate classifiers to overcome individual classifier weaknesses, however this method differs in that classifiers of a different nature can be used together [Schapire, 1990; Witten et al., 2011].

2.9 Boosting

Boosting algorithms are described by Schapire [1990] as the combination of multiple weak learners to successively increase the overall prediction power of the ensemble, thus producing a stronger learner. It is used to reduce the bias of the overall learner. After some initial research in leveraging systems, [Freund and Schapire, 1999] proposed the AdaBoost algorithm (described below), that successfully applies new learners to previously misclassified samples, thereby increasing the strength of the ensemble they are added to once trained.

2.9.1 AdaBoost M1

The AdaBoost algorithm (short for Adaptive Boosting) proposed by Freund and Schapire [1996], is considered to be statistically provable to significantly increase the learning accuracy of weak learners – those that are only marginally improved over chance. Freund and Schapire [1996] found it to contribute only a slight improvement for more sophisticated algorithms such as C4.5 Freund and Schapire [1996]. They also have two versions of the AdaBoost algorithm that do not make any difference for binary classifiers, but will have a performance different for multiple class learners. The general AdaBoost algorithm can be described as follows [Raul Rojas, 2013]:

Considering we have an attack instance x_i, and a classifier k_j, then every classifier can submit an opinion on the class of the instance $k_j(x_i)$, such that each opinion is $\in -1, 1$. Note the opinions are either *yes* (1) or *no* (-1), not a result in a continuum. The opinions can be combined as a group K of experts with $sign(C(x_i))$, such that the sum is weighted depending on the previously observed accuracy of that classifier [Raul Rojas, 2013]. Then the group of weighted opinions can be shown as:

$$C(x_i) = \alpha_1 k_1(x_i) + \alpha_1 k_1(x_i) + ... + \alpha_n k_n(x_i)$$

where n is the total number of classifiers present [Raul Rojas, 2013]. The constants $\alpha_1, \alpha_2, ..., \alpha_n$ are the weights assigned to each classifier. This can also be described as a linear decision by each classifier, which is then combined into a non-linear (weighted) sum for the final decision [Raul Rojas, 2013].

$$C(x_i) = \sum_{j=1}^{n} \alpha_j k_j(x_i)$$

In order to assign the weights to each classifier-instance pair, there is a penalty cost e^{β} when a classifier is wrong on the training instance, and a reward $e^{-\beta}$ when it is correct [Raul Rojas, 2013]. In this algorithm, β must be > 0 so that prediction failures have a greater cost than prediction success [Raul Rojas, 2013]. The training algorithm is run once for each classifier in the pool. Given training set T of instances x_i and labels y_i, we can assign initial weights $w_i = 1$ to all instances x_i [Raul Rojas, 2013]. If we have M classifiers ($M = n$ iterations), we can describe it as follows [Raul Rojas, 2013]:

Let W be the sum of weights of all the instances the classifier correctly classified, and W_e the sum of all weights of the instances the classifier was incorrect. Then

For $m = 1$ *to* M

1. Select the classifier that minimizes the total weights of misclassified instances:

$$W_e = \sum_{y_i \neq k_m(x_i)} w_i^m$$

2. Set the weight α_m of the classifier to

$$\alpha_m = 1/2ln(\frac{1 - e_m}{e_m})$$

where $e_m = \frac{W_e}{W}$

3. Update the weights of the data points for the next iteration $(m + 1)$. If $k_m(x_i)$ is a misclassification, we set

$$w_i^{(m+1)} = w_i^{(m)} e^{(\alpha_m)} = w_i^{(m)} \sqrt{\frac{1 - e_m}{e_m}}$$

otherwise

$$w_i^{(m+1)} = w_i^{(m)} e^{(-\alpha_m)} = w_i^{(m)} \sqrt{\frac{e_m}{1 - e_m}}$$

This is then repeated for each $m = 1, 2, ..., n$ [Raul Rojas, 2013]. After each successive round, the classifier is added to the ensemble, and the remaining misclassified instance weights are increased such that successive learners will put more emphasis on learning those previously misclassified instances [Raul Rojas, 2013]. The end result is an ensemble of classifiers that are algorithmically homogeneous, but weighted according to their individual contribution to a specific type of previously seen instance [Raul Rojas, 2013].

2.10 Bagging

Bagging is an ensemble learning method used to reduce the variance found in some learner/dataset combinations, that in turn can enhance the overall predictive performance of that classifier [Breiman, 1996]. Breiman et al. proposed this system as a way to generate a series of similar classifiers on training data derived from the original set, called a *boostrap sample* (akin to stratified instance selection), and then averaging the combined decision of those classifiers when tested on a new instance.

2.10.1 Learning Algorithm

The Bagging algorithm is different from Boosting in that instead of serializing the learners, with the performance of each learner affecting how subsequent learners are built, all of the learners are trained in parallel using a unique bootstrap sample [Breiman, 1996].The bootstrap samples are generated by *sub-sampling* the original training data *with replacement* so the resulting dataset is the same size as the original dataset [Breiman, 1996]. The bagging algorithm used in my experiments is derived from work by Breiman [1996], and can be described using pseudo-code [Zhou, 2009]:

Consider the dataset D containing instances $(x_1, y_1), (x_2, y_2), ..., (x_m, y_m)$, the base learning algorithm L, and the number of learning rounds T. Algorithm 3 describes the process as

Algorithm 3 Calculate $y = x^n$

```
for t = 1, ... , T do
    $D_t = Bootstrap(D);$ % Generate a bootstrap sample from D
    $h_t = L(D_t)$% Train a base learner $h_t$ from the bootstrap sample
end for
```

The resulting ensemble of learners is then consulted whenever a new instance is seen, and their output values are averaged as [Breiman, 1996]:

$$H(x) = argmax_{y \in Y} \sum_{t=1}^{T} 1(y = h_t(x))$$

For the case of a binary classifier we see that the result will be *true* if the majority of classifiers return *true*. But what about the case when there is an even number of classifiers and there is a draw? Bagging includes a random initialization seed that is consulted as a tie breaker in the unlikely event that this happens [Breiman, 1996].

2.11 Voting

Ensembles differ slightly from other *meta-classifiers* in that traditional ensembles use the same algorithm, whereas *meta* classification uses a logical meta-layer to combine the results from different learners [Kittler et al., 1998; Kuncheva, 2004]. In this section we consider how to combine *multiple disparate* models in order to combine the strengths (and unfortunately potential weaknesses) of a disparate learning algorithms through a technique called *Voting*.

Voting ensembles [Kittler et al., 1998; Kuncheva, 2004] are similar to Bagging because they run multiple learners in parallel to one another. However, the learners in Voting may be heterogeneous [Kittler et al., 1998; Kuncheva, 2004]. The Voting algorithm used an *Average of Probabilities* combination rule, such that every included classifier prediction had equal weighting [Kittler et al., 1998; Kuncheva, 2004].

2.12 Summary

There are many machine learning algorithms to choose from when considering the problem of traffic classification for intrusion detection. As mentioned in this chapter, each algorithm has various applications, as well as strengths and weaknesses. The next chapter outlines the experimental environment that was used for testing various machine learning algorithms.

Chapter 3

Experimental Methodology

3.1 Overview

The fundamental purpose of my research was to explore the performance trade-offs of a variety of machine learning schemes in the network intrusion detection traffic domain. The resulting analysis should answer the questions proposed in Chapter 1, so that current and future researchers in the network security domain can more productively apply machine learning while narrowing the large variety of algorithm and dataset scheme options available. In order to accomplish this goal, a suitable experimental methodology was designed with the following properties in mind:

1. Algorithms should be selected from a variety of statistical models in the machine learning domain, so a proper representation of the fundamental options are tested

2. The dataset against which the algorithms are tested should be a realistic rep-

resentation of both normal and abnormal traffic, including zero-day attack instances

3. Algorithm evaluation should be performed in an unbiased fashion, using an experimental framework that support consistency and reliability

My research focus is on supervised learning algorithms, as I am interested in using labeled instances to train each representative model, and testing the model against an established test set to evaluate the potential predictive power of each model against attacks that represent both previously seen and novel network behaviors. Both single and ensemble learning algorithm schemes were evaluated to see which approach to learning provides the best solution to the difficult task of network traffic classification for security purposes.

Finding an appropriate dataset against which to test supervised learning algorithms is a difficult task. A modern Network Security Laboratory [2012] derivation of DARPA's military network attack simulation dataset from the KDD Challenge Cup 99 [1999] was used as a representation of network flows found in a realistic network environment. This dataset, despite having origins dating back a decade, is still being used today as it contains a variety of well-known attack type behavior representative of similar attacks seen today, and through many years of analysis has been fine tuned for more accurate machine learning evaluations such as the one presented in this thesis.

The testing framework I developed started out as a design around a generic training and testing API that would be interfaced using publicly available algorithms released in Java or C++. As mentioned in Chapter 1, my research questions revolved around the comparison of existing algorithms, and not the creation of novel machine learning methods, and hence some time was spent in discovery of robust algorithms that are adequate representative samples of what is currently used by the ML community. This exploration naturally led to several existing frameworks that already supplied a fair number of robust algorithms. After spending some time in each of these frameworks, I came to the conclusion that the Weka [Hall et al., 2009] framework (developed at Waikato University) not only included well-developed and widely vetted libraries, but also included a fairly robust data exploration and experimentation framework that would satisfy the consistency and reliability attributes I was after.

My experimental methodology follows five main phases to ensure a comprehensive evaluation of the NSL-KDD dataset against the proposed algorithms. These steps are:

1. pre-processing the dataset to take advantage of dimensionality reduction through information gain;
2. training and testing single algorithms against each generated dataset;
3. bagging and boosting each of the algorithm-to-dataset combinations to evaluate homogeneous ensemble learning;
4. testing meta-classification learning through a voting committee based on multiple heterogeneous hypothesis models already generated;
5. systematic evaluation of each training and testing round against standard error evaluation metrics;

The dataset generation, model parameter selection, and selected evaluation metrics will be further explained below. The algorithm evaluations then follow in Chapter 4.

3.2 Evaluation Dataset

3.2.1 DARPA Intrusion Detection Evaluation Dataset

In 1998-1999 the Defense Advanced Research Projects Agency and Airforce Research Laboratory provided funding to the Massachusetts Institute of Technology Lincoln Lab MIT [2012] to create a body of network traffic flows that could be used to evaluate the performance of network intrusion detection tools. These data sets were revolutionary at the time due to the rarity of open source, labeled, and sizable data samples for this very purpose. Lincoln Labs set up a network which closely resembled the network used by the sponsoring United States Air Force, and operated the network as if it were real; however the network was also attacked by a variety of attack types.

3.2.2 KDD CUP '99

This dataset was later used by the ACM Special Interest Group for Knowledge and Data Discovery in a challenge to find the best known machine learning algorithm for fraud and intrusion detection. Four Gigabytes of captured network packet headers, representing almost two months worth of traffic, were compressed into network flows. Stolfo et al. [2000] derived additional, high-level, flow features from this dataset that may help indicate normal and attack type data. According to the KDD-Cup 99 Dataset description, attacks largely fell into four main categories [KDD Cup, 2012]):

- DOS: denial-of-service — attacks designed to disrupt network services; e.g. syn flood;
- R2L: remote-to-local — unauthorized access from a remote machine, e.g. guessing password;
- U2R: user-to-root — unauthorized access to local superuser (root) privileges, e.g., various "buffer overflow" attacks;
- Probing: surveillance and other probing, e.g., port scanning.

In order to further simulate what may be encountered in the real world, additional attack types were added to the test data that were not found in the training data, in order to help determine which challenging intrusion detection systems were able to generalize to new attacks that may not have the same signature or behavior as previously seen training categories.

In order to assist challenging participants, the data flow features were also offered in the standard ARFF format. The following traffic flow features are provided for reference: A list of basic TCP flow features can be found in Appendix A.1 A list of content features that may indicate domain-specific behavior can be found in Appendix A.2 And a list of features calculated based on a two-second time window can be found in Appendix A.3 The attack type allocation for the training and novel types can be found in Appendix A.4.

3.2.3 NSL-KDD Data Set Corrections

After a number of years of evaluation, researchers [McHugh, 2000; Tavallaee et al., 2009b] found the data contained some artifacts due to the artificial attack data injected into the TCP flows, and presented some issues of determining an appropriate unit of analysis and problems associated with the use of the ROC method of analysis [McHugh, 2000]. Some statistical flaws were found in the original dataset that cause bias in training and testing algorithms [McHugh, 2000], causing a skewed distribution of simulated attack types to bias the results. Many of these issues have been demonstratively corrected – the NSL-KDD dataset [Tavallaee et al., 2009a] being one example.

Due to these issues in properly identifying false-positive rates, a new dataset was created by the Network Security Laboratory at the University of New Brunswick Information Security Centre of Excellence. This dataset was deemed the *NSL-KDD* Data Set [Network Security Laboratory, 2012], and was created for the purpose of removing undesirable statistical artifacts. The dataset is reported to still have some of the same issues as the original KDD dataset [Tavallaee et al., 2009b]. However, due to the lack of alternative real-world datasets available for testing intrusion detection systems, it is reasonable for this research experiment in order to compare machine learning algorithms in the information security domain. Some of the improvements the NSL-KDD dataset has over the original KDD dataset are [Network Security Laboratory, 2012]:

- It does not include redundant records in the training set, so the classifiers will not be biased towards more frequent records.

- There are no duplicate records in the proposed test sets; therefore, the performance of the learners is not biased by the methods which have better detection rates on the frequent records.
- The number of selected records from each difficulty level group is inversely proportional to the percentage of records in the original KDD dataset. As a result, the classification rates of distinct machine learning methods vary in a wider range, which makes it more efficient to have an accurate evaluation of different learning techniques.
- The number of records in the training and testing sets is reasonable, which makes it affordable to run the experiments on the complete set without the need to randomly select a small portion. Consequently, evaluation results of different research works will be consistent and comparable.

The resulting dataset removed redundant records from both the training and test set in order to remove the bias towards frequent and repetitive net flows typical in an artificial network environment, allowing machine learning algorithms to observe traffic more typically found in a spontaneous and ever-changing real world networking environment, and hence improve the utility of the learned models, and the evaluation of potential commercial products.

3.3 Feature Selection and Dimensionality Reduction

Classifier accuracy largely depends on the quality of the information present in the attributes that form a correlation to the target classes. Feature selection in Machine Learning assumes that large feature sets may include redundant features. Removing these features will increase the overall learning and prediction efficiency of the chosen model. In order to select the best attributes in the feature space, we need to be able to measure the quality, or contribution, each attribute contributes. The quality of the information in each attribute can be quantified, using measurements such as information gain, or entropy, and gain ratio. These measurements, described below, are used to create the datasets used for further algorithmic testing.

Many of the machine learning libraries found in the Weka toolkit support the use of filters and wrappers, and their effect to the algorithms will be considered in terms of the performance metrics outlined below. Two examples of how the above approach works involves applying a feature filter [Khor et al., 2009; Mukkamala and Sung, 2003] by systematically removing one feature at a time to determine the overall contribution (Information Gain) of each feature, and using only the highest ranking features. Another example could apply a feature wrapper [Choubey et al., 1996; Guyon and Elisseeff, 2003] by using the individual classifier to choose optimal feature subsets and measuring the accuracy gain or loss.

The NSL-KDD train and test dataset has a relatively high number of classes; feature spaces greater than 10 are generally considered large, and subject to the curse of dimensionality. Fortunately, as the traffic behavior descriptors created by Stolfo et al. [2000] are universal to all traffic flow samples, there are no missing values for any of the instances.

3.3.1 Pre-processing

Due to the relatively large number of attributes in this dataset, pre-processing was performed in order to reduce the dimensionality of the dataset, and attempt to narrow down the features to those which have the highest information gain, or overall contribution to the classification models. Train and test set were put through *weka.filters.supervised.attribute.remove* [Witten et al., 2011], to get rid of the last attribute *difficulty*, which was artificially added by Network Security Laboratory [2012] to allow future researchers to filter out attack types by difficulty, and improve overall learning accuracy. Since the NSL-KDD study results will not be used as a factor to limit instances to the easiest sets, my research includes attacks of every difficulty (which lower test accuracy considerably) for the greatest realism possible.

As described in Chapter 2, the two primary methods of supervised attribute (feature) selection are filtering, and wrapping.

3.3.1.1 Information Gain Ratio

Information gain can be said to be the amount of predictive value that a specific attribute has with respect to the target class [Mitchell, 1997]. If we understand the conditional and independent probability with which each class occurs, both with and without the presence of an attribute value, then we can state how much information is gained through that feature.

When considering Decision Trees 4.4, these attributes would be the ones placed nearest to the top, as their selection has the greatest impact on what the final classification would be.

This can be stated through the following equation [Mitchell, 1997]:

$$InfoGain(Class, Attribute) = H(Class) - H(Class|Attribute)$$

Stated simply, the information gained through the correlation between a class and an attribute is the difference between probability of the class occurring, and the probability of the class occurring given the presence of that attribute [Mitchell, 1997].

The issue with this method of calculating information gain is present in the bias towards features that have a relatively high number of values compared to those that only have a few [Mitchell, 1997]. This is overcome by modifying how the implicit decision tree construction is partitioned by penalizing those features that have a greater range of possible values. This is done through *split information*, a method of changing the calculated information gain based on how broadly and uniformly the data is split within the attribute [Mitchell, 1997]. Given S, the collection of attribute and class examples, and A, the set of all possible attributes, then split information gain can be shown as [Mitchell, 1997]:

$$SplitInformation(S, A) = -\sum_{i=1}^{c} \frac{|Si|}{|S|} log_2 \frac{|Si|}{|S|}$$

where S_c are the c subsets of examples by partitioning S by the c-valued attribute A [Mitchell, 1997]. The *GainRatio* can then be shown using the previous value of Information Gain by

$$GainRatio(S, A) = \frac{InfoGain(S, A)}{SplitInformation(S, A)}$$

which is used for the purpose of deriving the filtered attribute set *KDDTrain GainRatio* used the rest of these experiments. The specific search algorithm used is the Best First search [Witten and Frank, 2005], used with the *weka.filters.supervised.attribute.GainRatio* algorithm [Witten et al., 2011].

The top 10 features in Table 3.1 were selected based on their modified Information Gain (Ratio), using 10-fold cross validation for evaluation.

Table 3.1: Information Gain Ratio Feature Selection.

Gain	Index	Feature
0.418	12	logged_in
0.3739	26	srv_serror_rate
0.3399	4	flag
0.3327	25	serror_rate
0.332	39	dst_host_srv_serror_rate
0.2673	30	diff_srv_rate
0.2648	38	dst_host_serror_rate
0.2585	6	dst_bytes
0.2313	5	src_bytes
0.2243	29	same_srv_rate

3.3.1.2 Subset Evaluation

The second method of feature selection, known as wrapping, can be thought of as a search technique to find the subset of features with the smallest error rate.

According to Hall [1998], this technique can be used to evaluate how a subset of attributes affects the predictive ability of each feature. This supervised attribute selection algorithm was run against the full KDDTrain dataset in order to derive the features that have the highest correlation with each other and the target class, with lowest inter-correlation with features not contained in the subset. The *weka.attributeSelection.CfsSubsetEval* [Witten et al., 2011] algorithm was used, with a *BestFirst* search strategy.

SubsetEval was run against KDDTrain, using the training dataset as the test set.

The algorithm was run again using 10-fold Cross Validation, and it selected the same attributes for all 10 folds, with the addition of 29 (same_srv_rate), 30 (diff_srv_rate), and 37 (dst_hold_srv_diff_host_rate). Considering there may be a partial correlation with the last few attributes, the 10-Fold Cross Validation results have been used to generate a new dataset.

I ran it against the KDDTrainAttack NSL dataset, that contains all attack class types instead of categorizing them as either anomaly/normal. Both the binary class dataset and multiple class datasets using 10-fold Cross Validation generated the same list of attributes with 100 percent confidence, validating the subsets with the highest contribution. The Best First algorithm was run in forward mode, with a stale search after 5 node expansions. 398 subsets were evaluated, with the merit of the best subset found at 0.568.

Table 3.2: Subset Evaluation Feature Selection.

Gain	Index	Feature
10(100 %)	4	flag
10(100 %)	5	src_bytes
10(100 %)	6	dst_bytes
10(100 %)	12	logged_in
10(100 %)	26	srv_serror_rate
1(10 %)	29	same_srv_rate
9(90 %)	30	diff_srv_rate
4(40 %)	37	dst_host_srv_diff_host_rate

Based on the above results, I generated a new anomaly training set based on the features shown in Table 3.2, that had a merit rating found in a majority of the cross validation averaged over 10 folders.

3.4 Evaluation Methodology

The evaluation of machine learning algorithms on the DARPA attack data can be done a number of ways. Considering the algorithms introduced in Chapter 2, the following subsections explain how they were configured for use with each of the datasets I generated, both single and in combination, as well as the framework and environment in which they were trained and tested.

3.4.1 Test Platform

The experiments described below were performed on a computer with hardware and software specifications shown in Table 3.3:

Table 3.3: Test Platform

Feature	Value
Processor	Intel i5-2540M @ 2.6GHz
Memory	4 Gigabytes
Operating System	Windows 7 Pro. SP1
Weka	v3.7.9 (Developer)

For the purpose of the Weka environment, the Java Virtual Machine was allocated a 2 Gigabyte heap size to deal with the more sophisticated (and memory hungry) models.

3.4.2 Algorithm Configuration

There are a variety of machine learning and data mining-based classification algorithms that have been used in this problem domain. In order to cover a broad range of learning types, I have sampled algorithms from Logistic Regression [Xu et al., 2005], Naive Bayes Classifier [Khor et al., 2009], Support Vector Machines [Mukkamala and Sung, 2003], K-nearest-neighbor Algorithm [Faraoun and Boukelif, 2007; Chairunnisa et al., 2009], and Artificial Neural Networks [Ali et al., 2010; Vatisekhovich, 2009; Zilberbrand, 2009]. Additional configuration for the number of iterations and instances were made to Boosting [Freund and Schapire, 1996; Schapire, 1990] and Bagging [Breiman, 1996] respectively.

The following sections describe the parameters used in each algorithm:

3.4.2.1 Naive Bayes

The *weka.supervised.bayes.NaiveBayes* [Witten et al., 2011] algorithm was run against each dataset using the default settings such that it compares the frequency across all attributes to the class without assuming any conditional probabilities between classes, in other words complete independence.

3.4.2.2 J48 Decision Tree

The *weka.supervised.trees.J48* [Witten et al., 2011] Decision Tree implemented in Weka is an open source version of C4.5, and was run with the confidence factor (used for later pruning the tree) set to 0.25, and the minimum number of instances per leaf set to 2.

3.4.2.3 Logistic Regression

Evaluation of logistic regression was done using the *weka.classifiers.rule.Logistic* library [Witten et al., 2011], which is a standard implementation of the sigmoid function training through gradient descent. This library was run with the log-likelihood ridge value of $1.0E - 8$.

3.4.2.4 K-Nearest Neighbour

Evaluation of the instance-based learning algorithm (or K-Nearest Neighbour as it is commonly called) was performed using the *weka.classifiers.lazy.iBk* [Witten et al., 2011] library, which is a standard implementation of the K-NN training through calculating nearest points in an n-dimensional euclidean plane. For the purpose of this experiment, the independent variable k, being the number of grouped points involved in the voting process, was initialized to 5 for initial training and testing, with no distance weighting for penalization. This means that for every new instance seen in the training and testing sets, the average class of the neighbouring 5 points with the smallest euclidean distance was considered as the classification of the newly seen instance.

3.4.2.5 Support Vector Machine

The LibSVM wrapper WLSVM [EL-Manzalawy and Honavar, 2005] , based on the popular Sequential Minimal Optimization (SMO) algorithm in Weka, was used for the support vector machine experiments. The WLSVM wrapper allows many useful statistical performance information to be generated from the classifier. Testing involved training one binary classifier with each set of traffic features in full and reduced through a filter and wrapper pre-processor. Considering the hyperplane used to classify each parameter may be non-linear, I employed a radial basis function kernel machine, as proposed by Aizerman et al. [1964], to improve the classification boundary.

3.4.2.6 Artificial Neural Network

Evaluation of the multi-layer perceptron algorithm was done using the *weka.classifiers.functions.MultilayerPerceptron* library [Witten et al., 2011], which is a standard implementation of the sigmoid perceptron using Least Mean Squared cost calculation, and gradient descent back-propagation.

The number of input nodes corresponds to the number of features in each training set. Only one output node is required for anomaly detection. The number of hidden layer nodes is generated by the following forumula:

$$\frac{attributes + classes}{2}$$

The learning rate used was 0.3, and the learning momentum was 0.2. Backpropagation was cycled for 500 epochs per learning example.

3.4.2.7 Boosting Ensemble

As described in Chapter 2, the Boosting Freund and Schapire [1996]; Schapire [1990] ensemble learner allows multiple homogeneous models to be combined, such that successive iterations of the algorithm can learn on instances that were previously misclassified. For the purpose of this thesis, the *weka.supervised.meta.AdaBoostM1* [Witten et al., 2011] classifier was run against each algorithm and dataset combination. The number of maximum iterations was set to 10, using re-weighting on the instances (as opposed to re-sampling). The weight threshold was set to 100.

3.4.2.8 Bagging Ensemble

As described in Chapter 2, the Bagging [Breiman, 1996] ensemble learner allows multiple homogeneous models to be combined, such that multiple instances of the algorithm can run in parallel on sub-sampled datasets, while averaging the classification results amongst them.. For the purpose of this thesis, the *weka.supervised.meta.Bagging* [Witten et al., 2011] classifier was run against each algorithm and dataset combination. The number of model instances was set to 10, using sub-sampling on the dataset.

3.4.2.9 Voting Meta-Classifier

As described in Chapter 2, the Voting meta-learner allows multiple heterogeneous models to be combined, such that the strengths (and weaknesses) of each may conribute towards a final prediction value. For the sake of this thesis, the *weka.supervised.meta.Voting* [Witten et al., 2011] classifier was tested against both the best-in-class and worst-in-class learners from each dataset and algorithm combination.

3.4.3 Model Validation

The Gain Ratio and Subset Evaluation pre-processing used 10-Fold Cross Validation in order find a high degree of confidence for the contribution of each attribute to classification.

The Hold-out Validation method using the NSL-KDD designed training and test sets (with novel attacks) was used for all algorithm and dataset pairs for final evaluation. Although 10-Fold Cross Validation would have been desirable in order to explore the variance measurement for each algorithm-dataset learning scheme, this was deemed unfeasible due to the overall number of schemes tested, and the amount of time each scheme took to train. As cross validation in practice is typically used for model tuning, this was also felt to be unnecessary since the algorithms tested were using industry standard settings (as defined by [Witten et al., 2011]), and hyper-parameter optimization was not a factor in the test methodology.

In the next section, I will explain the metrics that will be used in order to conduct error analysis on each of the algorithms, and the method by which they can be appropriately assessed for performance through visualization.

3.5 Evaluation Metrics

Considering each classifier, the experiments measure these dependent variables through the use of receiver operative curve (ROC), and precision and recall measurements, in order to determine the overall *effectiveness* of each algorithm. The evaluation types will be outlined below

3.5.1 Error Analysis

The experiment observation is described in terms of classification performance, or accuracy. Classification performance in this context is measured in terms of true-positive, false-positive, true-negative, and false-negative. These dependent measurements are typically performed in terms of percentage of data sets identified over all possible sets to be identified correctly, A measurement applicable to classification is the concept of precision and recall Makhoul et al. [1999], which is used in the ROC calculation as well as additional analysis.

3.5.2 Receiver Operating Characteristic

The relative operating curve (ROC) Farid and Rahman [2010] has been used to compare the relationship between detection and false positive rates. The deciding metric used to measure overall accuracy was the mean Area under the ROC. As explained in Chapter 2, the Precision, Recall, and Specificity of each algorithm was dependent on which of the classes was considered for the positive and negative label. The Mean AUC, as used in my performance comparison, considers both scenarios, and takes the average of the two values. The observed AUC deviation between the classes were minimal or non-existent in most cases, and hence the AUC mean was chosen as an all-encompassing measurement such that the positive and negative class designations were not an issue.

3.5.3 Training and Testing Time

Each of the algorithms was tested for overall training and test time. The absolute times are highly dependent on the implementation platform, however the relative times compared in Chapter 4 are a strong indicator of relative algorithm performance. These metrics were analyzed according to their contribution for each algorithm. In practice, the training time is not as important as the testing time. Model creation in this case would be performed offline using already labeled data, whereas testing would be on-line and produces an events-per-second metric that is a major consideration in the intrusion detection industry.

3.6 Summary

In this chapter we have discussed the history and relevant research of intrusion detection, and introduced the NSL-KDD dataset that will be evaluated. The pre-processing methods for dataset dimensionality reduction using filter and wrapper algorithms has been described, as well the various hold-out, and cross-validation techniques. Algorithm evaluation will be performed using mean Area under Receiver Operating Characteristic curve for general comparison.

In the next chapter I will discuss the algorithm evaluation for all supervised learning algorithms with the NSL-KDD dataset, in order to establish benchmark measurements for each major type of algorithm found in contemporary Machine Learning.

Chapter 4

Algorithm Evaluation

4.1 Overview

General supervised machine learning algorithms provide a method for building a prediction model using historical data comprised of attribute-classification sets. This chapter will focus on the application of systematic model construction using the machine learning experimental methodology outlined in Chapter 3 to determine performance for the network intrusion domain, both individually and in combination. The evaluation of the various algorithms investigates the performance of each individual learning algorithm, when applied to the original and pre-processed training and testing data sets. For each dataset created, homogeneous ensemble learning sets are tested for comparing the performance results to the previously evaluated individual algorithms.

Each learning *scheme* (algorithm and database combination) applies a number of independent, controlled, and dependent variables specific to the operation of that algorithm as defined in Chapter 2, according to an established testing methodology as defined in Chapter 3, and describes the measured the outcome of each algorithm according to the metrics previously defined.

The single classifiers that are evaluated in the first group are ZeroR [Witten et al., 2011], Logistic Regression [Xu et al., 2005], Naive Bayes Classifier [Khor et al., 2009], Support Vector Machines [Mukkamala and Sung, 2003], K-nearest-neighbor Algorithm [Faraoun and Boukelif, 2007; Chairunnisa et al., 2009], and Artificial Neural Networks (multi-level perceptrons) [Ali et al., 2010; Vatisekhovich, 2009; Zilberbrand, 2009].

Each individual classifier type will be described in terms of the mathematical model used to enable supervised learning, the model parameters used in the experiments, and a final evaluation of each model against the test sets.

My research explores the performance indicators established to demonstrate the overall fitness of both individual classifier accuracy and weighing additional classifiers, to demonstrate the feasibility of application to the network intrusion detection domain. The boosting algorithms tested in this phase, to explore the possibility of improvements over single-algorithm classification, include AdaBoost, Bagging, and Voting.

Detailed performance metrics can be found in the following Appendices: Appendix B contains single classification results; Appendix C contains Boosting classification results; Appendix D contains Bagging classification results; and Appendix E contains Voting classification results.

4.2 Baseline Prediction with Zero-R

The ZeroR function in *weka.classifiers.rules.ZeroR* [Witten et al., 2011] was trained against the full NSL-KDD training set. Since 50% of the training instances are classified as normal (that is, the mode of the nominal classes) Zero-R classifies every test instance as normal. Averaging out over all test folds reveals exactly what one would expect – the classifier correctly identifies the same ratio of normal-to-anomaly as the training set, and in this case is correct roughly more than half the time; a truly sub-optimal performing classifier.

This prediction model was trained in 1 second, and tested against the test set in 1 second, using the platform in Section 3.4.1.

4.2.1 Evaluation Configuration

As described in Chapter 2, ZeroR can be evaluated through a confusion matrix. Considering the anomaly dataset, there are two classes – normal and anomaly. Considering all classes in a row by column matrix, this will produce a 2x2 matrix. The confusion matrix produced is shown in Table 4.1:

Table 4.1: ZeroR Standard Confusion Matrix.

a	b	classified as
9711	0	a = normal
12833	0	b = anomaly

In this case, the first row represents all of the normal classified test instances, and each column represents how those instances were classified. In the case of Zero-R, because the algorithm defaults all instance classifications to the class that is seen the most often in training instances, there are only true positive, and no mis-classifications as the “b” class. The training set, as seen earlier, had a larger number of normal class instances, and thus is the class Zero-R will always select. Since the test set has more anomalies than normal, the classifier performs worse than guessing (50%), and is obviously a poor choice.

4.2.2 Feature Reduction Evaluation

The Zero-R algorithm was run against the Gain Ratio and Subset Evaluation datasets, and in all three cases the mean AUC is consistently 0.5 across all datasets, which is to be expected given the simple prediction of the class with highest occurrence being the normal class. Although feature reduction would make a difference in accuracy for any other more complex classifier, feature reduction makes no difference for ZeroR, as the features are not even considered as part of the training or testing process.

4.2.3 Ensemble Evaluation

The ZeroR algorithm was run against the Boosting and Bagging meta-classifiers, and in both cases, the results were identical to the single classifier. This intuitively makes sense, since the classifier will make the same prediction, regardless of the features or number of instances used, and hence serializing or parallelizing a homogeneous ensemble will not change the accuracy.

4.2.4 Summary

The ZeroR algorithm, as simple as it is, does provide us with a worst-case baseline; using the testing set provided demonstrates that blindly selecting the class based on the majority of seen classes in the training set can be worse than guessing, for example when the majority of training classes becomes the minority in the test dataset. As the results are identical regardless of pre-processing or ensemble setup, they are not discussed further.

4.3 Naive Bayes

The *weka.supervised.bayes.NaiveBayes* [Witten et al., 2011] algorithm was run against each dataset using the default settings required such that it compares the frequency across all attributes to the class, without assuming any conditional probabilities between classes - in other words, complete independence. An overview of runtime performance is explained below.

4.3.1 Single Performance

4.3.1.1 Standard Dataset

The single Naive Bayes algorithm was run against the Standard dataset, and took 3 seconds to build, and less than a second to run, using the platform described in Section 3.4.1.

Table 4.2 illustrates the results of the model accuracy using 10-fold Cross Validation against the training set. The following performance metrics were produced using

Table 4.2: Naive Bayes Single Standard Training Confusion Matrix.

a	b	classified as
63106	4237	a = normal
7834	50796	b = anomaly

the standard dataset: the algorithm using the standard dataset produced a 0.904 average TP rate, and 0.101 FP rate. The average Precision and Recall measurements were 0.905 and 0.904 respectively. The mean AUC is 0.966 (out of a possible 1.000).

Table 4.3: Naive Bayes Single Standard Testing Confusion Matrix.

a	b	classified as
9041	670	a = normal
4714	8119	b = anomaly

Considering the confusion matrix in Table 4.3, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.761 average TP rate, and 0.198 FP rate. The average Precision and Recall measurements were 0.809 and 0.71 respectively. The mean AUC is much higher than the previously seen ZeroR implementation, sitting at 0.908 (out of a possible 1.000), however it is lower than the training set evaluation, since novel attacks were introduced that were previously unseen (as expected). This trend is found with the majority of the following algorithm and dataset combinations.

4.3.1.2 Gain Ratio Dataset

The single Naive Bayes algorithm was run against the Gain Ratio dataset, and took less than a second to build and test, using the platform described in Section 3.4.1.

Considering the confusion matrix in Table 4.4, the following performance metrics

Table 4.4: Naive Bayes Single Gain Ratio Confusion Matrix.

a	b	classified as
9537	174	a = normal
5893	6940	b = anomaly

were produced: the algorithm using the Gain Ratio dataset produced a 0.731 average TP rate, and 0.208 FP rate. The average Precision and Recall measurements were 0.822 and 0.731 respectively. The mean AUC is slightly lower than the standard dataset at 0.879.

4.3.1.3 Subset Evaluation Dataset

The single Naive Bayes algorithm was run against the Subset Evaluation dataset, and took less than a second to build and test, using the platform described in Section 3.4.1.

Table 4.5: Naive Bayes Single Subset Confusion Matrix.

a	b	classified as
9467	244	a = normal
5675	7158	b = anomaly

Considering the confusion matrix in Table 4.5, the following performance metrics were produced: the algorithm produced a 0.737 average TP rate, and 0.205 FP rate. The average Precision and Recall measurements were 0.82 and 0.737 respectively. The mean AUC is in between the other two standard datasets with a value of 0.892.

Using a single classifier, the Standard dataset performed the best overall, with a similar on-line test speed to the other two data sets.

4.3.2 Boosting Performance

4.3.2.1 Standard Dataset

The Boosting Naive Bayes algorithm was run against the Standard dataset, performed over 6 iterations. The model took 25 seconds to build, and two seconds to test, using the platform described in Section 3.4.1. The build time was much higher than the single classifiers offline training, and effectively halving the online detection rate.

Table 4.6: Naive Bayes Boosting Standard Confusion Matrix.

a	b	classified as
9113	598	a = normal
5321	7512	b = anomaly

Considering the confusion matrix in Table 4.6, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.737 average TP rate, and 0.214 FP rate. The average Precision and Recall measurements were 0.799 and 0.737 respectively. The mean AUC is much lower than the previously seen Single classifier, sitting at 0.822. This trend is found with the other two data sets.

4.3.2.2 Gain Ratio Dataset

The Boosting Naive Bayes algorithm was run against the Gain Ratio dataset, performed over 4 iterations. The model took 5 seconds to build, and less than a second to test, using the platform described in Section 3.4.1. The build time was roughly the same as the single classifiers offline training.

Considering the confusion matrix in Table 4.7, the following performance metrics

Table 4.7: Naive Bayes Boosting Gain Ratio Confusion Matrix.

a	b	classified as
9041	670	a = normal
4714	8119	b = anomaly

were produced: the algorithm using the standard dataset produced a 0.731 average TP rate, and 0.208 FP rate. The average Precision and Recall measurements were 0.822 and 0.731 respectively. The mean AUC is much lower than the previously seen Single and Boosting classifiers, sitting at 0.764.

4.3.2.3 Subset Evaluation Dataset

The Boosting Naive Bayes algorithm was run against the Subset Evaluation dataset, performed over 3 iterations. The model took 3 seconds to build, and less than a second to test, using the platform described in Section 3.4.1.

Table 4.8: Naive Bayes Boosting Subset Confusion Matrix.

a	b	classified as
9467	244	a = normal
5675	7158	b = anomaly

Considering the confusion matrix in Table 4.8, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.737 average TP rate, and 0.205 FP rate. The average Precision and Recall measurements were 0.82 and 0.737 respectively; this is roughly equivalent with the Single Classifier performance. The mean AUC is lower than the previously seen Standard dataset, but slightly higher than the Gain Ratio, sitting at 0.768.

Using a Boosting ensemble, the Naive Bayes classifier performed much worse than the Single version on all three datasets, while the train and test time was roughly the same or higher depending on the dataset.

4.3.3 Bagging Performance

4.3.3.1 Standard Dataset

The Bagging Naive Bayes algorithm was run against the Standard dataset, with 10 parallel instances. The model took 9 seconds to build, and 3 seconds to test, using the platform described in Section 3.4.1. The build time was in between the single and boosting algorithms.

Table 4.9: Naive Bayes Bagging Standard Confusion Matrix.

a	b	classified as
9041	670	a = normal
4674	8159	b = anomaly

Considering the confusion matrix in Table 4.9, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.763 average TP rate, and 0.196 FP rate. The average Precision and Recall measurements were 0.81 and 0.763 respectively. The mean AUC is much higher than the previously seen Boosting classifiers, and only slightly higher than the Single algorithm, sitting at 0.91. This trend is found with the other two data sets.

4.3.3.2 Gain Ratio Dataset

The Bagging Naive Bayes algorithm was run against the Gain Ratio dataset, with 10 parallel instances. The model took 3 seconds to build, and less than a second to test, using the platform described in Section 3.4.1. The build time was slightly higher than the single classifiers offline training.

Table 4.10: Naive Bayes Bagging Gain Ratio Confusion Matrix.

a	b	classified as
9536	175	a = normal
5817	7016	b = anomaly

Considering the confusion matrix in Table 4.10, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.734 average TP rate, and 0.206 FP rate. The average Precision and Recall measurements were 0.823 and 0.734 respectively. The mean AUC is at 0.878.

4.3.3.3 Subset Evaluation Dataset

The Bagging Naive Bayes algorithm was run against the Subset Evaluation dataset, with 10 parallel instances. The model took 2 seconds to build, and less than a second to test, using the platform described in Section 3.4.1.

Table 4.11: Naive Bayes Bagging Subset Confusion Matrix.

a	b	classified as
9465	246	a = normal
5656	7177	b = anomaly

Considering the confusion matrix in Table 4.11, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.738 average True Positive rate, and 0.204 False Positive rate. The average Precision and Recall measurements were 0.82 and 0.738 respectively; this is roughly equivalent with the Single Classifier performance. The mean AUC is slightly lower than the previously seen Standard dataset, but slightly higher than the Gain Ratio, sitting at 0.891.

Using a Boosting ensemble, the Naive Bayes classifier performed much worse than the Single version on all three datasets, while the train and test time was roughly the same or higher depending on the dataset.

4.3.4 Summary

The mean area under the ROC for all Naive Bayes dataset and configuration is shown in Figure 4.1.

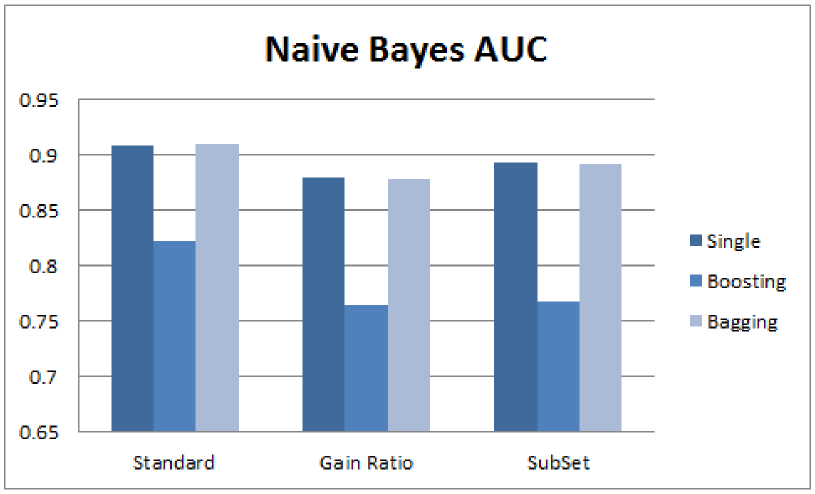


Figure 4.1: Naive Bayes Area Under ROC

It is obvious that boosting does not fare nearly as well as the standard and bagging implementations. Given all training and testing times, and the accuracy metrics, the Bagging ensemble is superior in both time and accuracy to the Boosting algorithm alternative, and roughly the same as the single algorithm, across all three data sets.

It is interesting to note that similar experiments performed by Caruana and Niculescu-mizil [2006] have shown this algorithm may not fare as well as its boosting equivalent. However, against the NSL-KDD dataset, clearly the Single and Bagging variants have a higher overall mean AUC.

4.4 J48 Decision Tree

The *weka.supervised.trees.J48* [Witten et al., 2011] tree implemented in Weka is an open source version of C4.5, and was run with the confidence factor (used for later pruning the tree) set to 0.25, and the minimum number of instances per leaf set to 2. The J48 decision tree evaluation, using the parameters described above, resulted in the following performance metrics for each of the Raw, Gain Ratio, and SubsetEval training and testing sets. The J48 algorithm performed the best with the Subset Evaluation wrapper dataset, despite taking 30 seconds less than the raw dataset to create the learning model,, using the platform described in Section 3.4.1.

4.4.1 Single Performance

4.4.1.1 Standard Dataset

The single J48 algorithm was run against the Standard dataset, took 35 seconds to build, and less than a second to run, using the platform described in Section 3.4.1.

Table 4.12: J48 Single Standard Confusion Matrix.

a	b	classified as
9448	263	a = normal
3900	8933	b = anomaly

Considering the confusion matrix in Table 4.12, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.815 average

True Positive rate, and 0.146 average False Positive rate. The average Precision and Recall measurements were 0.858 and 0.815 respectively. The mean AUC is 0.84. This value is much lower than the Naive Bayes counterpart, only surpassing the Boosting variant thereof.

4.4.1.2 Gain Ratio Dataset

The single J48 algorithm was run against the Gain Ratio dataset, took 6 seconds to build, and less than a second to test, using the platform described in Section 3.4.1. It is much faster than the standard dataset.

Table 4.13: J48 Single Gain Ratio Confusion Matrix.

a	b	classified as
9383	328	a = normal
5090	7743	b = anomaly

Considering the confusion matrix in Table 4.13, the following performance metrics were produced: the algorithm using the Gain Ratio dataset produced a 0.76 average True Positive rate, and 0.19 average False Positive rate, worse than the standard dataset. The average Precision and Recall measurements were 0.825 and 0.76 respectively, confirming the above results. The mean AUC is slightly lower than the standard dataset at 0.813.

4.4.1.3 Subset Evaluation Dataset

The single J48 algorithm was run against the Subset Evaluation dataset, took 5 seconds to build, and less than a second to test, using the platform described in Section 3.4.1, similar in time performance with the Gain Ratio filter.

Table 4.14: J48 Single Subset Confusion Matrix.

a	b	classified as
9383	328	a = normal
5206	7627	b = anomaly

Considering the confusion matrix in Table 4.14, the following performance metrics were produced: the algorithm produced a 0.755 average True Positive rate, and 0.194 average False Positive rate. The average Precision and Recall measurements were 0.823 and 0.755 respectively. The mean AUC is highest among the other two datasets with a value of 0.867.

4.4.1.4 Single Algorithm Summary

Using a single classifier, the Subset Evaluation wrapper dataset performed the best overall, with a similar on-line test speed to the other two data sets.

4.4.2 Booting Performance

4.4.2.1 Standard Dataset

The Boosting J48 algorithm was run against the Standard dataset, took 4 minutes 59 seconds to build, and less than a second to run, using the platform described in Section 3.4.1.

Table 4.15: J48 Boosting Standard Confusion Matrix.

a	b	classified as
9433	278	a = normal
4715	8118	b = anomaly

Considering the confusion matrix in Table 4.15, the following performance metrics

were produced: the algorithm using the standard dataset produced a 0.779 average True Positive rate, and 0.175 average False Positive rate. The average Precision and Recall measurements were 0.838 and 0.779 respectively. The mean AUC is 0.936, a vast improvement over all of the single J48 classifiers.

4.4.2.2 Gain Ratio Dataset

The Boosting J48 algorithm was run against the Gain Ratio dataset, took 1 minute 9 seconds to build, and less than a second to test, using the platform described in Section 3.4.1. It is much faster than the standard dataset.

Table 4.16: J48 Boosting Gain Ratio Confusion Matrix.

a	b	classified as
9335	376	a = normal
4626	8207	b = anomaly

Considering the confusion matrix in Table 4.16, the following performance metrics were produced: the algorithm using the Gain Ratio dataset produced a 0.778 average True Positive rate, and 0.177 average False Positive rate, approximately the same as the standard dataset. The average Precision and Recall measurements were 0.832 and 0.778 respectively, confirming the above results. The mean AUC is slightly lower than the standard dataset at 0.904, not performing as well as the standard dataset classifier.

4.4.2.3 Subset Evaluation Dataset

The single J48 algorithm was run against the Subset Evaluation dataset, took 45 seconds to build, and less than a second to test, using the platform described in

Section 3.4.1. The build time was less than the Gain Ratio filter, with similar testing time performance.

Table 4.17: J48 Boosting Subset Confusion Matrix.

a	b	classified as
9375	336	a = normal
4815	8018	b = anomaly

Considering the confusion matrix in Table 4.17, the following performance metrics were produced: the algorithm produced a 0.772 average True Positive rate, and 0.181 average False Positive rate. The average Precision and Recall measurements were 0.831 and 0.772 respectively. The mean AUC is only slightly higher than the Gain Ratio dataset, with a value of 0.907.

4.4.2.4 Boosting Summary

Using a Boosting classifier, the standard dataset produced the best mean AUC overall against the test set. The training build time was far greater than the other two datasets, with a slight increase for on-line test speed.

4.4.3 Bagging Performance

4.4.3.1 Standard Dataset

The Bagging J48 algorithm was run against the Standard dataset, took 4 minutes 11 seconds to build, and less than a second to run, using the platform described in Section 3.4.1.

Considering the confusion matrix in Table 4.18, the following performance metrics

Table 4.18: J48 Bagging Standard Confusion Matrix.

a	b	classified as
9437	274	a = normal
3395	9438	b = anomaly

were produced: the algorithm using the standard dataset produced a 0.837 average True Positive rate, and 0.13 average False Positive rate. The average Precision and Recall measurements were 0.87 and 0.837 respectively. The mean AUC is 0.907, also a vast improvement over all of the single J48 classifiers.

4.4.3.2 Gain Ratio Dataset

The Bagging J48 algorithm was run against the Gain Ratio dataset, took 51 seconds to build, and less than a second to test, using the platform described in Section 3.4.1. It is much faster than the standard dataset in both training and testing time.

Table 4.19: J48 Bagging Gain Ratio Confusion Matrix.

a	b	classified as
9391	320	a = normal
5111	7722	b = anomaly

Considering the confusion matrix in Table 4.19, the following performance metrics were produced: the algorithm using the Gain Ratio dataset performed with a 0.759 average True Positive rate, and 0.19 average False Positive rate, slightly worse than the standard dataset. The average Precision and Recall measurements were 0.826 and 0.759 respectively, confirming the above results. The mean AUC is slightly lower than the standard dataset at 0.847, not performing as well.

4.4.3.3 Subset Evaluation Dataset

The Bagging J48 algorithm was run against the Subset Evaluation dataset, took 41 seconds to build, and less than a second to test, using the platform described in Section 3.4.1. The build time was less than the Gain Ratio filter, with faster testing time performance.

Table 4.20: J48 Bagging Subset Confusion Matrix.

a	b	classified as
9386	325	a = normal
5105	7728	b = anomaly

Considering the confusion matrix in Table 4.20, the following performance metrics were produced: the algorithm produced a 0.759 average True Positive rate, and 0.19 average False Positive rate, exactly the same as the Gain Ratio filter dataset. The average Precision and Recall measurements were 0.825 and 0.759 respectively, confirming these results. The mean AUC is only significantly higher than the Gain Ratio dataset, with a value of 0.912.

4.4.3.4 Bagging Summary

Using a Bagging classifier, the Subset Evaluation dataset produced the best mean AUC overall against the test set. The training build time was the fastest compared to the other two datasets, with the fastest on-line test speed at 0.11 seconds.

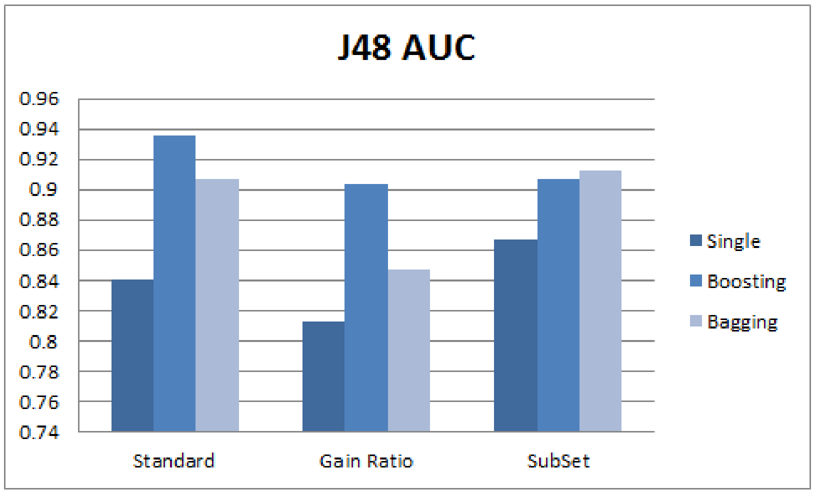


Figure 4.2: J48 Decision Tree Area Under ROC

4.4.4 Summary

The mean AUC for all J48 classifiers is shown in Figure 4.2. The boosted J48 classifiers have the highest overall accuracy, with the lowest accuracy almost matching the best accuracy found in the Bagging ensembles. Given all training and testing times, and the accuracy metrics, the Boosting ensemble is superior in both time and accuracy to the Single and Bagging algorithm alternatives using most of the datasets.

4.5 Linear and Logistic Regression

Evaluation of the logistic regression algorithm was done using the *weka.classifiers.rule.Logistic* [Witten et al., 2011] library, that is a standard implementation of the sigmoid function training through gradient descent. This library was run with the log-likelihood ridge value of $1.0E - 8$.

4.5.1 Single Performance

4.5.1.1 Standard Dataset

The single Logistic Regression algorithm was run against the Standard dataset, and took 23 seconds to build, and 0.67 seconds to test, using the platform described in Section 3.4.1.

Table 4.21: Logistic Regression Single Standard Confusion Matrix.

a	b	classified as
8882	829	a = normal
4542	8291	b = anomaly

Considering the confusion matrix in Table 4.21, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.762 average True Positive rate, and 0.201 average False Positive rate. The average Precision and Recall measurements were 0.803 and 0.762 respectively. The mean AUC is 0.749, and is the lowest performing classifier of the Logistic Regression series.

4.5.1.2 Gain Ratio Dataset

The single Logistic Regression algorithm was run against the Gain Ratio dataset, that took 5.78 seconds to build, and 0.78 seconds to test, using the platform described in Section 3.4.1. It is much faster than the standard dataset.

Table 4.22: Logistic Regression Single Gain Ratio Confusion Matrix.

a	b	classified as
9493	218	a = normal
5111	7722	b = anomaly

Considering the confusion matrix in Table 4.22, the following performance metrics were produced: The single algorithm using the Gain Ratio dataset produced a 0.764 average True Positive rate, and 0.184 average False Positive rate, slightly better than the standard dataset. The average Precision and Recall measurements were 0.834 and 0.764 respectively. The increased precision confirming the above results as the average False Positive rate is where the performance gain took place, with the True Positive rate staying approximately the same. The mean AUC is higher than the standard dataset at 0.853, and clearly outperforms it in terms of overall accuracy, build time, and test time.

4.5.1.3 Subset Evaluation Dataset

The single Logistic Regression algorithm was run against the Subset Evaluation dataset, that took 4.79 seconds to build, and 0.19 seconds to test, using the platform described in Section 3.4.1, similar in build time performance with the Gain Ratio filter, but far faster than both of the others.

Table 4.23: Logistic Regression Single Subset Confusion Matrix.

a	b	classified as
9439	272	a = normal
5417	7416	b = anomaly

Considering the confusion matrix in Table 4.23, the following performance metrics were produced: the algorithm produced a 0.748 average True Positive rate, and 0.198 average False Positive rate, slightly worse than the Gain Ratio dataset. The average Precision and Recall measurements were 0.823 and 0.748 respectively, confirming these results. The mean AUC is between the other two datasets with a value of

0.803.

4.5.1.4 Single Algorithm Summary

Using a single classifier, the Gain Ratio wrapper dataset performed the best overall, with a similar on-line test speed to the other two data sets. Considering the 5% drop in accuracy when using Subset Evaluation, it may not be worth the test time performance increase.

4.5.2 Booting Performance

4.5.2.1 Standard Dataset

The Boosting Logistic Regression algorithm was run against the Standard dataset, took 4 minutes 49 seconds to build, and 0.6 seconds to test, using the platform described in Section 3.4.1.

Table 4.24: Logistic Regression Boosting Standard Confusion Matrix.

a	b	classified as
8918	793	a = normal
4982	7851	b = anomaly

Considering the confusion matrix in Table 4.24, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.744 average True Positive rate, and 0.214 average False Positive rate. The average Precision and Recall measurements were 0.793 and 0.744 respectively. The mean AUC is 0.931, a vast improvement over all of the single Logistic Regression classifiers.

4.5.2.2 Gain Ratio Dataset

The Boosting Logistic Regression algorithm was run against the Gain Ratio dataset, took 37.63 seconds to build, and 0.35 seconds to test, using the platform described in Section 3.4.1. It is much faster than the standard dataset.

Table 4.25: Logistic Regression Boosting Gain Ratio Confusion Matrix.

a	b	classified as
9496	215	a = normal
5357	7476	b = anomaly

Considering the confusion matrix in Table 4.25, the following performance metrics were produced: the algorithm using the Gain Ratio dataset produced a 0.753 average True Positive rate, and 0.192 average False Positive rate, approximately the same as the standard dataset. The average Precision and Recall measurements were 0.829 and 0.753 respectively, confirming the above results. The mean AUC is lower than the standard dataset at 0.864, and not performing as well as the standard dataset classifier despite being faster.

4.5.2.3 Subset Evaluation Dataset

The Boosting Logistic Regression algorithm was run against the Subset Evaluation dataset, took 25.43 seconds to build, and 0.2 seconds to test, using the platform described in Section 3.4.1. The build time was slightly less than the Gain Ratio filter, with less testing time performance.

Considering the confusion matrix in Table 4.26, the following performance metrics were produced: the algorithm produced a 0.748 average True Positive rate, and 0.198 average False Positive rate. The average Precision and Recall measurements were

Table 4.26: Logistic Regression Boosting Subset Confusion Matrix.

a	b	classified as
9439	272	a = normal
5417	7416	b = anomaly

0.823 and 0.748 respectively. The mean AUC is only slightly lower than the Gain Ratio dataset, with a value of 0.855.

4.5.2.4 Boosting Summary

Using a Boosting classifier, the standard dataset produced the best mean AUC overall against the test set. The training build time was far greater than the other two datasets, with a slight increase for on-line test speed. The Subset wrapper proved to be the worst boosting classifier.

4.5.3 Bagging Performance

4.5.3.1 Standard Dataset

The Bagging Logistic Regression algorithm was run against the Standard dataset, took 2 minutes 53 seconds to build, and 0.74 seconds to test, using the platform described in Section 3.4.1.

Table 4.27: Logistic Regression Bagging Standard Confusion Matrix.

a	b	classified as
8889	822	a = normal
4670	8163	b = anomaly

Considering the confusion matrix in Table 4.27, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.756 average

True Positive rate, and 0.205 average False Positive rate. The average Precision and Recall measurements were 0.8 and 0.756 respectively. The mean AUC is 0.816, also slightly less performance over all of the Boosting classifiers.

4.5.3.2 Gain Ratio Dataset

The Bagging Logistic algorithm was run against the Gain Ratio dataset, took 49.16 seconds to build, and 0.28 seconds to test, using the platform described in Section 3.4.1. It is much faster than the standard dataset in both training and testing time.

Table 4.28: Logistic Regression Bagging Gain Ratio Confusion Matrix.

a	b	classified as
9491	220	a = normal
5289	7544	b = anomaly

Considering the confusion matrix in Table 4.28, the following performance metrics were produced: the algorithm using the Gain Ratio dataset produced a 0.756 average True Positive rate, and 0.19 average False Positive rate, approximately the same as the standard dataset. The average Precision and Recall measurements were 0.83 and 0.756 respectively, confirming the above results. The mean AUC is slightly higher than the standard dataset at 0.844.

4.5.3.3 Subset Evaluation Dataset

The Bagging Logistic algorithm was run against the Subset Evaluation dataset, took 39.14 seconds to build, and 0.26 seconds to test, using the platform described in

Section 3.4.1. The build time was less than the Gain Ratio filter, with only marginally faster testing time performance.

Table 4.29: Logistic Regression Bagging Subset Confusion Matrix.

a	b	classified as
9397	314	a = normal
5323	7510	b = anomaly

Considering the confusion matrix in Table 4.29, the following performance metrics were produced: the algorithm produced a 0.75 average True Positive rate, and 0.197 average False Positive rate, approximately the same as the Gain Ratio filter dataset. The average Precision and Recall measurements were 0.821 and 0.745 respectively, confirming these results. The mean AUC is only slightly lower than the Gain Ratio dataset, with a value of 0.833.

4.5.3.4 Bagging Summary

Using a Bagging classifier, the Gain Ratio dataset produced the best mean AUC overall against the test set. The training build time and testing time was only a little bit slower compared to the other two datasets, and with a marginally higher AUC score.

4.5.4 Logistic Summary

The mean AUC for all Logistic Regression classifiers can be seen in Figure 4.3. As with the J48 classifiers, the boosted Logistic classifiers have the highest overall accuracy, with the lowest accuracy almost matching the best accuracy found in the Single classifier and Bagging ensembles. Boosting ensemble is superior in accuracy to

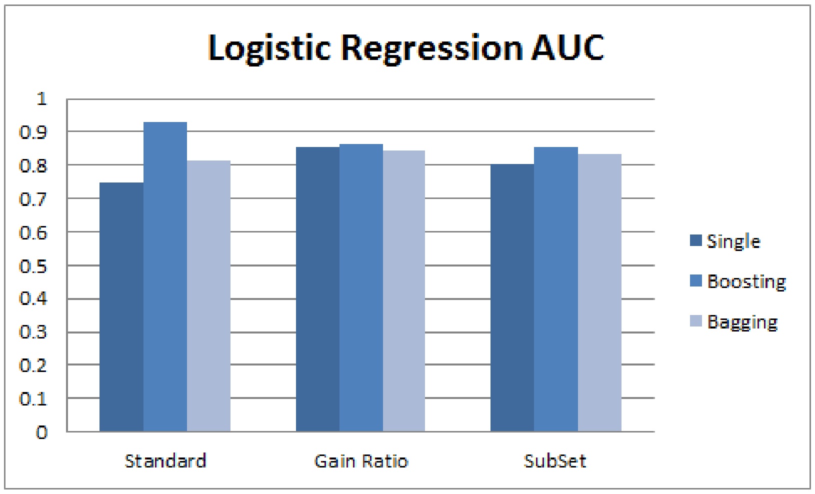


Figure 4.3: Logistic Regression Area Under ROC

the Single and Bagging algorithm alternatives. However, this only applies using the standard unmodified dataset. The testing time is faster than the single and bagging ensembles on the same dataset, and is therefor the clear winner if only by virtue of the average AUC ROC.

4.6 K-Nearest Neighbour

Evaluation of the instance-based learning algorithm (or K-Nearest Neighbour as it is commonly called) was performed using the *weka.classifiers.lazy.iBk* [Witten et al., 2011] library, which is a standard implementation of the K-NN training through calculating nearest points in an n-dimensional Euclidean plane. For the purpose of this experiment, the independent variable k, being the number of grouped points involved in the voting process, was initialized to 5 for initial training and testing, with no distance weighting for penalization. This means that for every new instance seen in the training and testing sets, the average class of the neighbouring 5 points with the smallest Euclidean distance was considered as the classification of the newly seen instance.

The algorithm was run on the Raw, GainRatio, and SubsetEval sets as explained in the following subsections.

4.6.1 Single Performance

4.6.1.1 Standard Dataset

The single iBk algorithm was run against the Standard dataset, and took 0.03 seconds to build, and 6 minutes, 1 second to run, using the platform described in Section 3.4.1.

Table 4.30: K-Nearest Neighbour Single Standard Confusion Matrix.

a	b	classified as
9351	360	a = normal
4534	8299	b = anomaly

Considering the confusion matrix in Table 4.30, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.783 average True Positive rate, and 0.173 average False Positive rate. The average Precision and Recall measurements were 0.836 and 0.783 respectively. The mean AUC was 0.846.

4.6.1.2 Gain Ratio Dataset

The single iBk algorithm was run against the Gain Ratio dataset, that took 0.02 seconds to build, and 2 minutes, 29 seconds to test, using the platform described in

Section 3.4.1t. The build time was comparable to the standard dataset, however the test time comparatively faster, considering the dimensionality reduction.

Table 4.31: K-Nearest Neighbour Single Gain Ratio Confusion Matrix.

a	b	classified as
9403	308	a = normal
4136	8697	b = anomaly

Considering the confusion matrix in Table 4.31, the following performance metrics were produced: The single algorithm using the Gain Ratio dataset produced a 0.803 average True Positive rate, and 0.157 average False Positive rate, only slightly better than the standard dataset based on weighted averages alone. The average Precision and Recall measurements were 0.849 and 0.803 respectively. The mean AUC is slightly lower than the standard dataset at 0.838, and only outperforms it in terms of testing time.

4.6.1.3 Subset Evaluation Dataset

The single iBk algorithm was run against the Subset Evaluation dataset, that took 0.03 seconds to build, and 2 minutes, 17 seconds to test, using the platform described in Section 3.4.1, similar in build time performance with the Gain Ratio filter, but far faster than both of the others.

Table 4.32: K-Nearest Neighbour Single Subset Confusion Matrix.

a	b	classified as
9463	248	a = normal
4731	8102	b = anomaly

Considering the confusion matrix in Table 4.32, the following performance metrics

were produced: the algorithm produced a 0.779 average True Positive rate, and 0.173 average False Positive rate. The average Precision and Recall measurements were 0.84 and 0.779 respectively, confirming these results. The mean AUC is nearly identical with the Gain Ratio filter, having a value of 0.838.

4.6.1.4 Single Algorithm Summary

Using a single classifier, the Standard dataset performed the best overall, with a similar training speeds, but took twice as long for on-line test speeds compared to the other two data sets.

4.6.2 Booting Performance

4.6.2.1 Standard Dataset

The Boosting iBk algorithm was run against the Standard dataset, took 4 hours, 38 minutes, and 52 seconds to build over 4 iterations of the dataset, using the platform described in Section 3.4.1. It took 30 minutes to test the model. This is obviously a significantly increased amount of time.

Table 4.33: K-Nearest Neighbour Boosting Standard Confusion Matrix.

a	b	classified as
9351	360	a = normal
4534	8299	b = anomaly

Considering the confusion matrix in Table 4.33, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.783 average True Positive rate, and 0.173 average False Positive rate. The average Precision and

Recall measurements were 0.836 and 0.783 respectively. The mean AUC is 0.837, performing worse than all of the single iBk classifiers.

4.6.2.2 Gain Ratio Dataset

The Boosting iBk algorithm was run against the Gain Ratio dataset, took 3 hours, 39 minutes, and 45 seconds to build over 6 iterations of the dataset, using the platform described in Section 3.4.1. It took 15 minutes to test the model. Building was roughly the same amount of time as the standard dataset, with the testing performing twice as fast.

Table 4.34: K-Nearest Neighbour Boosting Gain Ratio Confusion Matrix.

a	b	classified as
9333	378	a = normal
3927	8906	b = anomaly

Considering the confusion matrix in Table 4.34, the following performance metrics were produced: the algorithm using the Gain Ratio dataset produced a 0.809 average True Positive rate, and 0.154 average False Positive rate, approximately the same as the standard dataset. The average Precision and Recall measurements were 0.849 and 0.809 respectively, confirming the above results. The mean AUC is slightly higher than the standard dataset at 0.859, while still predicting twice as fast.

4.6.2.3 Subset Evaluation Dataset

The Boosting iBk algorithm was run against the Subset Evaluation dataset, took 4 hours, 11 minutes, and 31 seconds to build over 7 iterations of the dataset, using the platform described in Section 3.4.1. It took 17 minutes to test the model. Training time was roughly between the other two data sets, with the testing performing two minutes slower than Gain Ratio.

Table 4.35: K-Nearest Neighbour Boosting Subset Confusion Matrix.

a	b	classified as
9411	300	a = normal
3917	8916	b = anomaly

Considering the confusion matrix in Table 4.35, the following performance metrics were produced: the algorithm produced a 0.813 average True Positive rate, and 0.149 average False Positive rate. The average Precision and Recall measurements were 0.855 and 0.813 respectively. The mean AUC is slightly higher than the Gain Ratio and Standard datasets, with a value of 0.867.

4.6.2.4 Boosting Summary

Using a Boosting classifier, the Subset Evaluation dataset produced the best mean AUC overall against the test set. The training build time was between the other two datasets, with a slight increase for on-line test speed. The standard dataset proved to be the worst for boosting the iBk classifier in all evaluation categories.

4.6.3 Bagging Performance

4.6.3.1 Standard Dataset

The Bagging iBk algorithm was run against the Standard dataset, took 0.77 seconds to build, and 1 hour, 49 minutes, and 53 seconds to test, using the platform described in Section 3.4.1.

Table 4.36: K-Nearest Neighbour Bagging Standard Confusion Matrix.

a	b	classified as
9351	360	a = normal
4535	8298	b = anomaly

Considering the confusion matrix in Table 4.36, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.783 average True Positive rate, and 0.173 average False Positive rate. The average Precision and Recall measurements were 0.836 and 0.783 respectively. The mean AUC is 0.858, with slightly more accurate performance over the Single classifiers, and similar to the Boosting classifiers.

4.6.3.2 Gain Ratio Dataset

The Bagging iBk algorithm was run against the Gain Ratio dataset, took 0.69 seconds to build, and 1 hour, 1 minutes, and 54 seconds to test, using the platform described in Section 3.4.1.

Table 4.37: K-Nearest Neighbour Bagging Gain Ratio Confusion Matrix.

a	b	classified as
9416	295	a = normal
4080	8753	b = anomaly

Considering the confusion matrix in Table 4.37, the following performance metrics were produced: the algorithm using the Gain Ratio dataset produced a 0.806 average True Positive rate, and 0.154 average False Positive rate. The average Precision and Recall measurements were 0.851 and 0.806 respectively, a slight increase compared to the standard dataset. The mean AUC is slightly higher than the standard dataset at

0.863.

4.6.3.3 Subset Evaluation Dataset

The Bagging iBk algorithm was run against the Subset Evaluation dataset, and took 0.90 seconds to build and 1 hour, 26 seconds to test, using the platform described in Section 3.4.1. The build time was greater than both the standard and the Gain Ratio filter, with only marginally faster testing time performance.

Table 4.38: K-Nearest Neighbour Bagging Subset Confusion Matrix.

a	b	classified as
9455	256	a = normal
4671	8162	b = anomaly

Considering the confusion matrix in Table 4.38, the following performance metrics were produced: the algorithm produced a 0.781 average True Positive rate, and 0.172 average False Positive rate. The average Precision and Recall measurements were 0.84 and 0.781 respectively, confirming these results. The mean AUC is slightly greater than the other two datasets, with a value of 0.871.

4.6.3.4 Bagging Summary

Using a Bagging classifier, the Subset Evaluation dataset produced the best mean AUC overall against the test set. The training build time was slightly higher than the other datasets. Both filter and wrapper testing times were significantly faster than the standard dataset, both having a marginally higher AUC score, demonstrating an overall gain when performing dimensionality reduction.

4.6.4 iBk Summary

The iBk training and testing times were slightly different than the previously seen classifiers, in that single and Bagging training times were fairly quick. Boosting training time was quite a bit longer (hours vs. minutes), and all of the ensemble testing times were an order of magnitude longer. The AUC accuracy for all iBk combinations are shown in Figure 4.4.

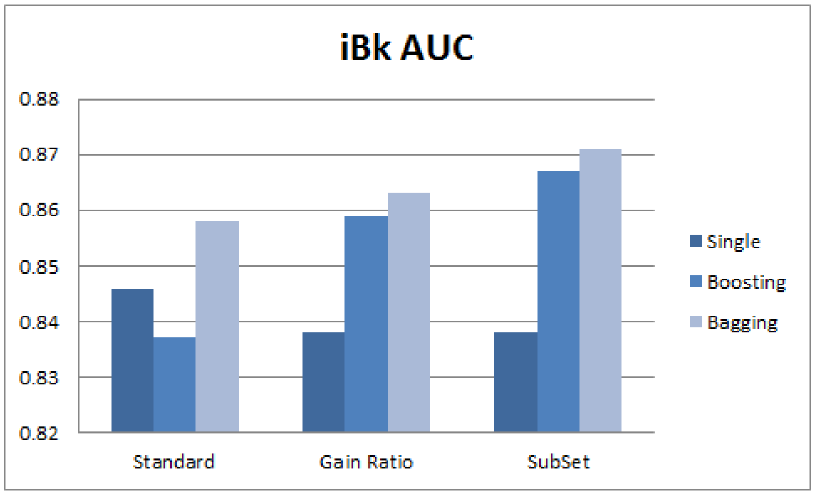


Figure 4.4: K-Nearest Neighbour Area Under ROC

Both the Gain Ratio filter and Subset Evaluation wrapper improved overall classification performance for ensembles. The opposite appears to hold true for the single iBk instance, where the Standard dataset provides the highest overall accuracy. That being said, the Subset Evaluation wrapper provided the best overall accuracy. The boosting algorithm, given the serialized training of 4-to-7 weighted models, took the longest for off-line model creation, and yet was significantly faster at on-line prediction given the reduced number of models polled compared with the standard 10 parallel models used in Bagging.

4.7 Support Vector Machines

The LibSVM wrapper WLSVM [EL-Manzalawy and Honavar, 2005], based on the popular Sequential Minimal Optimization (SMO) algorithm in Weka, was used for the support vector machine experiments. The WLSVM wrapper allows many useful statistical performance information to be generated from the classifier. Testing involved training one binary classifier with each set of traffic features in full and reduced through a filter and wrapper pre-processor. Considering the hyperplane used to classify each parameter may be non-linear, I employed a radial basis function kernel machine, as proposed by Aizerman et al. [1964], to improve the classification boundary. The results of running LibSVM against the raw, Gain Ratio, and Subset Evaluation data sets for each single and ensemble combination are described in the following subsections.

4.7.1 Single Performance

4.7.1.1 Standard Dataset

The single LibSVM algorithm was run against the Standard dataset, and took 2 hours, 8 minutes, and 14 seconds to build the model, and 4 minutes, 38 seconds to test the model, using the platform described in Section 3.4.1.

Table 4.39: Support Vector Machine Single Standard Confusion Matrix.

a	b	classified as
9515	196	a = normal
6507	6326	b = anomaly

Considering the confusion matrix in Table 4.39, the following performance metrics

were produced: the algorithm, using the standard dataset, produced a 0.703 average True Positive rate, and 0.23 average False Positive rate. The average Precision and Recall measurements were 0.808 and 0.703 respectively. The mean AUC was 0.736.

4.7.1.2 Gain Ratio Dataset

The single LibSVM algorithm was run against the Gain Ratio dataset, that took 29 minutes 32 seconds to build, and 39 seconds to test, using the platform described in Section 3.4.1. Both the training and testing time was much faster than the standard dataset.

Table 4.40: Support Vector Machine Single Gain Ratio Confusion Matrix.

a	b	classified as
9390	321	a = normal
5307	7526	b = anomaly

Considering the confusion matrix in Table 4.40, the following performance metrics were produced: The single algorithm using the Gain Ratio dataset produced a 0.75 average True Positive rate, and 0.197 average False Positive rate; this is only slightly better than the standard dataset based on weighted averages alone. The average Precision and Recall measurements were 0.821 and 0.75 respectively. The mean AUC is higher than the standard dataset at 0.777, and only outperforms it in both prediction accuracy and time metrics.

4.7.1.3 Subset Evaluation Dataset

The single LibSVM algorithm was run against the Subset Evaluation dataset, that took 30 minutes 56 seconds to train, and 39 seconds to test, using the platform described in Section 3.4.1. This scheme was only slightly longer for build time, and near identify test time compared with the Gain Ratio filter.

Table 4.41: Support Vector Machine Single Subset Confusion Matrix.

a	b	classified as
9390	321	a = normal
5271	7562	b = anomaly

Considering the confusion matrix in Table 4.41, the following performance metrics were produced: the algorithm produced a 0.752 average True Positive rate, and 0.196 average False Positive rate, almost identical with the Gain Ratio filter The average Precision and Recall measurements were 0.822 and 0.752 respectively, confirming these results. The mean AUC is thus nearly identical with the Gain Ratio filter, having a value of 0.778 and only detectable at 3 significant digits of measurement.

4.7.1.4 Single Algorithm Summary

Using a single classifier, the Standard dataset performed the worst overall. The filter and wrapper datasets performed the best overall with nearly identical testing times and overall prediction accuracy. The performance is so close, selecting a *best* dataset would be difficult without further analysis into the effects of feature set selection, optimization, or variance measurements over multiple folds.

4.7.2 Booting Performance

4.7.2.1 Standard Dataset

The Boosting LibSVM algorithm was run against the Standard dataset, and took 16 hours, 28 minutes, and 51 seconds to build over 10 iterations of the dataset, using the platform described in Section 3.4.1. It took 29 minutes 40 seconds to test the model. Just as described earlier, for K-Nearest Neighbour, Boosting adds a significantly increased amount of time.

Table 4.42: Support Vector Machine Boosting Standard Confusion Matrix.

a	b	classified as
9519	192	a = normal
6469	6364	b = anomaly

Considering the confusion matrix in Table 4.42, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.705 average True Positive rate, and 0.228 average False Positive rate. The average Precision and Recall measurements were 0.809 and 0.705 respectively. The mean AUC is 0.763, having similar accuracy to the single SVM classifiers.

4.7.2.2 Gain Ratio Dataset

The Boosting LibSVM algorithm was run against the Gain Ratio dataset, and took 9 hours, 13 minutes, and 7 seconds to build over 10 iterations of the dataset, using the platform described in Section 3.4.1. It took 17 minutes 54 seconds to test the model. Both training and testing were faster than the standard dataset.

Considering the confusion matrix in Table 4.43, the following performance metrics

Table 4.43: Support Vector Machine Boosting Gain Ratio Confusion Matrix.

a	b	classified as
9370	341	a = normal
5344	7489	b = anomaly

were produced: the algorithm using the Gain Ratio dataset produced a 0.748 average True Positive rate, and 0.199 average False Positive rate, a little better than the standard dataset. The average Precision and Recall measurements were 0.819 and 0.748 respectively, confirming the above results. The mean AUC is slightly higher than the standard dataset at 0.79, while still predicting nearly twice as fast.

4.7.2.3 Subset Evaluation Dataset

The Boosting LibSVM algorithm was run against the Subset Evaluation dataset, and took 11 hours, 38 minutes, and 5 seconds to build over 10 iterations of the dataset, using the platform described in Section 3.4.1. It took 21 minutes 13 seconds to test the model. Training and test time was a few minutes longer than the Gain Ratio dataset.

Table 4.44: Support Vector Machine Boosting Subset Confusion Matrix.

a	b	classified as
9369	342	a = normal
5322	7511	b = anomaly

Considering the confusion matrix in Table 4.44, the following performance metrics were produced: the algorithm produced a 0.749 average True Positive rate, and 0.199 average False Positive rate. The average Precision and Recall measurements were 0.819 and 0.749 respectively. The mean AUC is identical with the Gain Ratio datasets, tied at value of 0.79. Just as was found with the single algorithm, additional testing against the filter and wrapper methods would be required to distinguish any sort of advantage of one over the other.

4.7.2.4 Boosting Summary

Just as was observed using a single classifier, the Standard dataset performed the worst overall. The filter and wrapper datasets performed the best overall with nearly identical testing times and overall prediction accuracy. All of the Boosting datasets outperformed their single algorithm counterparts, while using the filter and wrapper pre-processors performed best overall.

4.7.3 Bagging Performance

4.7.3.1 Standard Dataset

The Bagging LibSVM algorithm was run against the Standard dataset, and took 19 hours, 1 minutes, and 33 seconds to simultaneously build 10 instances of the model on the dataset. It took 26 minutes 45 seconds to test the ensemble.

Table 4.45: Support Vector Machine Bagging Standard Confusion Matrix.

a	b	classified as
9517	194	a = normal
6517	6316	b = anomaly

Considering the confusion matrix in Table 4.45, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.702 average True Positive rate, and 0.23 average False Positive rate. The average Precision and

Recall measurements were 0.808 and 0.702 respectively. The mean AUC is 0.738, with far less overall accuracy than the rest of the classifiers seen thus far, save a very slight difference with the single algorithm run against the standard dataset.

4.7.3.2 Gain Ratio Dataset

The Bagging LibSVM algorithm was run against the Gain Ratio dataset,and took 2 hours, 56 minutes, and 10 seconds to simultaneously build 10 instances of the model on the dataset, using the platform described in Section 3.4.1. It took 5 minutes 24 seconds to test the ensemble.

Table 4.46: Support Vector Machine Bagging Gain Ratio Confusion Matrix.

a	b	classified as
9383	328	a = normal
5323	7510	b = anomaly

Considering the confusion matrix in Table 4.46, the following performance metrics were produced: the algorithm using the Gain Ratio dataset produced a 0.749 average True Positive rate, and 0.198 average False Positive rate. The average Precision and Recall measurements were 0.82 and 0.749 respectively, with similar results to the single algorithm. The mean AUC is higher than the standard dataset at 0.777, also very similar to the single algorithm results.

4.7.3.3 Subset Evaluation Dataset

The Bagging LibSVM algorithm was run against the Subset Evaluation dataset and took 3 hours, 15 minutes, and 29 seconds to simultaneously build 10 instances of the model on the dataset, using the platform described in Section 3.4.1. It took 5 minutes 55 seconds to test the ensemble.

Table 4.47: Support Vector Machine Bagging Subset Confusion Matrix.

a	b	classified as
9384	327	a = normal
5289	7544	b = anomaly

Considering the confusion matrix in Table 4.47, the following performance metrics were produced: the algorithm produced a 0.751 average True Positive rate, and 0.197 average False Positive rate. The average Precision and Recall measurements were 0.821 and 0.751 respectively, confirming these results. The mean AUC is slightly greater than the other two datasets, with a value of 0.778.

4.7.3.4 Bagging Summary

The bagging classifier produced almost identical average AUC compared to the single algorithms, against all test datasets. Bagging works best using the filter and wrapper datasets, and is outperformed in this regard by the Boosting equivalents. Training against the standard set was the highest of all categories, however the pre-processed datasets were less than their Boosting equivalents. Both filter and wrapper testing times were significantly faster than the boosting equivalents, but much slower than the single classifiers.

4.7.4 SVM Summary

The mean AUC values can be seen in Figure 4.5. It is fairly obvious that the Lib-SVM classifier performs well with pre-processed data. Single, Boosting, and Bagging

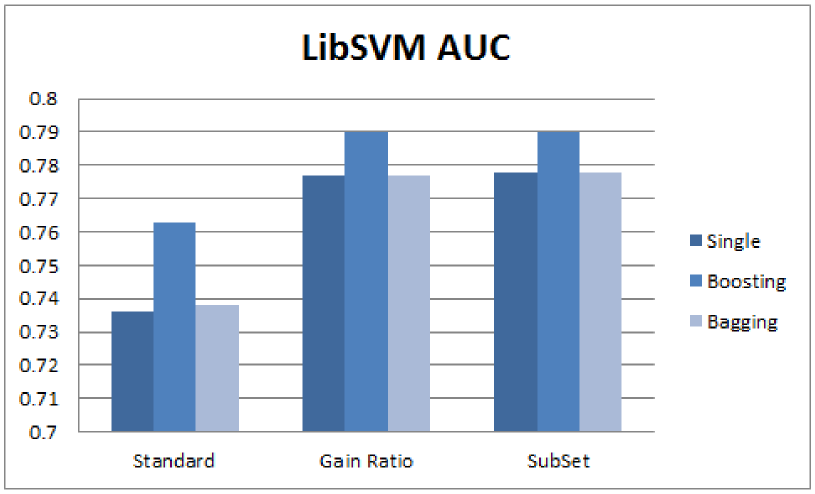


Figure 4.5: Support Vector Machines Area Under ROC

ensembles all performed better with the Gain Ratio and Subset Evaluation datasets than the Standard unprocessed dataset. All Boosting algorithms performed better than Single and Bagging training models under all datasets. The clear winner here is the use of Boosting with pre-processed data. However, additional testing would be required in order to establish any major tradeoffs between filter and wrapper methodologies.

4.8 Artificial Neural Networks

Evaluation of the multi-layer perceptron algorithm was done using the *weka.classifiers.functions.MultilayerPerceptron* [Witten et al., 2011] library, which is a standard implementation of the sigmoid perceptron using Least Mean Squared cost calculation, and gradient descent back-propagation.

The number of input nodes corresponds to the number of features in each training set. Only one output node is required for anomaly detection. The number of hidden layer nodes is generated by the following formula:

$$\frac{attributes + classes}{2}$$

The learning rate used was 0.3, and the learning momentum was 0.2. Back-propagation was cycled for 500 epochs per learning example. The performance of the neural network using the raw, Gain Ratio, and Subset Evaluation datasets is described in the following subsections.

4.8.1 Single Performance

4.8.1.1 Standard Dataset

The single Multilayer Perceptron algorithm was run against the Standard dataset, and took 2 hours, 11 minutes, and 49 seconds to build the model, and 9 seconds to test the model, using the platform described in Section 3.4.1.

Table 4.48: Artificial Neural Network Single Standard Confusion Matrix.

a	b	classified as
9015	696	a = normal
4429	8404	b = anomaly

Considering the confusion matrix in Table 4.48, the following performance metrics were produced: The single multilayer perceptron algorithm, using the standard dataset, produced a 0.733 average True Positive rate, and 0.189 average False Positive rate. The average Precision and Recall measurements were 0.815 and 0.773 respectively. The mean AUC was 0.788.

4.8.1.2 Gain Ratio Dataset

The single multilayer perceptron algorithm was run against the Gain Ratio dataset, and took 5 minutes 37 seconds to build, and 0.97 seconds to test, using the platform described in Section 3.4.1. Both the training and testing time was much faster than the standard dataset.

Table 4.49: Artificial Neural Network Single Gain Ratio Confusion Matrix.

a	b	classified as
9470	241	a = normal
5332	7501	b = anomaly

Considering the confusion matrix in Table 4.52, the following performance metrics were produced: The single multilayer perceptron algorithm using the Gain Ratio dataset produced a 0.753 average True Positive rate, and 0.193 average False Positive rate; only slightly better than the standard dataset based on weighted averages alone. The average Precision and Recall measurements were 0.827 and 0.753 respectively. The mean AUC is higher than the standard dataset at 0.838, and outperforms the single dataset in both prediction accuracy and time metrics.

4.8.1.3 Subset Evaluation Dataset

The single multilayer perceptron algorithm was run against the Subset Evaluation dataset, that took 4 minutes 41 seconds to train, and 0.56 seconds to test, using the platform described in Section 3.4.1. It performed the fastest in build time and test time compared with the Standard data and and Gain Ratio filter.

Considering the confusion matrix in Table 4.56, the following performance metrics were produced: the algorithm produced a 0.751 average True Positive rate, and 0.193

Table 4.50: Artificial Neural Network Single Subset Confusion Matrix.

a	b	classified as
9527	184	a = normal
5440	7393	b = anomaly

average False Positive rate, almost identical with the Gain Ratio filter The average Precision and Recall measurements were 0.83 and 0.751 respectively, confirming these results. The mean AUC is the highest of the datasets, having a value of 0.878.

4.8.1.4 Single Algorithm Summary

Using a single classifier, the Standard dataset performed the worst overall. The filter and wrapper datasets performed the best overall with the Subset Evaluation dataset providing the highest accuracy, build time, and test time of the single algorithm experiments.

4.8.2 Boosting Performance

4.8.2.1 Standard Dataset

The Boosting Multilayer Perceptron algorithm was run against the Standard dataset, and took 6 hours, 45 minutes, and 14 seconds to build over 2 iterations of the dataset, using the platform described in Section 3.4.1. It took 9 seconds to test the model. Just as was observed for LibSVM algorithm, Boosting adds a significantly increased amount of time to process the instances.

Considering the confusion matrix in Table 4.51, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.773 average

Table 4.51: Artificial Neural Network Boosting Standard Confusion Matrix.

a	b	classified as
9015	696	a = normal
4429	8404	b = anomaly

True Positive rate, and 0.189 average False Positive rate. The average Precision and Recall measurements were 0.815 and 0.773 respectively. The mean AUC is 0.792, having a slightly more accurate AUC than the lowest of the single classifiers, the standard dataset.

4.8.2.2 Gain Ratio Dataset

The Boosting Multilayer Perceptron algorithm was run against the Gain Ratio dataset, and took 53 minutes, and 3 seconds to build over 8 iterations of the dataset, using the platform described in Section 3.4.1. It took 0.78 seconds to test the model. Both training and testing were faster than the standard dataset.

Table 4.52: Artificial Neural Network Boosting Gain Ratio Confusion Matrix.

a	b	classified as
9470	241	a = normal
5331	7502	b = anomaly

Considering the confusion matrix in Table 4.52, the following performance metrics were produced: the algorithm using the Gain Ratio dataset produced a 0.753 average True Positive rate, and 0.193 average False Positive rate. The average Precision and Recall measurements were 0.827 and 0.753 respectively, confirming the above results. The mean AUC is moderately greater than the standard dataset at 0.847, while still predicting nearly four times as fast. It is interesting to note the Precision of the

Gain Ratio filter is greater than the standard dataset (fewer anomalies classified as normal), while the Recall is lower (fewer anomalies classified as anomaly), but still resulting in a greater AUC, considering the tradeoff.

4.8.2.3 Subset Evaluation Dataset

The Boosting Multilayer Perceptron algorithm was run against the Subset Evaluation dataset, and took 45 minutes, and 58 seconds to build over 8 iterations of the dataset, using the platform described in Section 3.4.1. It took just 0.75 seconds to test the model. Training and test time were both faster than the Gain Ratio dataset.

Table 4.53: Artificial Neural Network Boosting Subset Confusion Matrix.

a	b	classified as
9526	185	a = normal
5366	7467	b = anomaly

Considering the confusion matrix in Table 4.53, the following performance metrics were produced: the algorithm produced a 0.754 average True Positive rate, and 0.191 average False Positive rate. The average Precision and Recall measurements were 0.831 and 0.754 respectively. The mean AUC is almost identical with the Gain Ratio datasets, at 0.749. Additional testing against the filter and wrapper methods would be required to distinguish any sort of advantage of one over the other.

4.8.2.4 Boosting Summary

Just as was observed using a single classifier, the Standard dataset performed the worst overall. The filter and wrapper datasets performed the best overall with nearly identical testing times and overall prediction accuracy. However, neither was as good as the Subset Evaluation wrapper on the single algorithm test. All of the Boosting datasets outperformed their single algorithm counterparts.

4.8.3 Bagging Performance

4.8.3.1 Standard Dataset

The Bagging Multilayer Perceptron algorithm was run against the Standard dataset, and took 22 hours, 34 minutes, and 13 seconds to simultaneously build 10 instances of the model on the dataset. It took 23 seconds to test the ensemble. The increased build time may seem counterintuitive, as the standard dataset feature space are the same, however the Boosting ensemble classified the majority of the instances with only 2 passes of the dataset, whereas the bagging still required construction of 10 models.

Table 4.54: Artificial Neural Network Bagging Standard Confusion Matrix.

a	b	classified as
9015	696	a = normal
4641	8192	b = anomaly

Considering the confusion matrix in Table 4.54, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.763 average True Positive rate, and 0.197 average False Positive rate. The average Precision and Recall measurements were 0.809 and 0.763 respectively. The mean AUC is 0.9, with far greater accuracy than the rest of the classifiers seen thus far, save for a very slight difference with the single algorithm run against the Subset Evaluation dataset.

4.8.3.2 Gain Ratio Dataset

The Bagging Multilayer Perceptron algorithm was run against the Gain Ratio dataset,and took 59 minutes, and 53 seconds to simultaneously build 10 instances of the model on the dataset, using the platform described in Section 3.4.1. It took 0.95 seconds to test the ensemble.

Table 4.55: Artificial Neural Network Bagging Gain Ratio Confusion Matrix.

a	b	classified as
9528	183	a = normal
5708	7125	b = anomaly

Considering the confusion matrix in Table 4.55, the following performance metrics were produced: the algorithm using the Gain Ratio dataset produced a 0.739 average True Positive rate, and 0.202 average False Positive rate. The average Precision and Recall measurements were 0.824 and 0.739 respectively, with results similar to the single algorithm. The mean AUC is higher than the standard dataset at 0.863, slightly lower than the standard dataset due to a lower recall.

4.8.3.3 Subset Evaluation Dataset

The Bagging Multilayer Perceptron algorithm was run against the Subset Evaluation dataset and took 50 minutes, and 10 seconds to simultaneously build 10 instances of the model on the dataset, using the platform described in Section 3.4.1. It took 0.89 seconds to test the ensemble.

Considering the confusion matrix in Table 4.56, the following performance metrics were produced: the algorithm produced a 0.749 average True Positive rate, and 0.194 average False Positive rate. The average Precision and Recall measurements were

Table 4.56: Artificial Neural Network Bagging Subset Confusion Matrix.

a	b	classified as
9519	192	a = normal
5457	7376	b = anomaly

0.829 and 0.749 respectively, a slight improvement over the filter dataset. The mean AUC is slightly lower than the standard dataset, with a value of 0.892.

4.8.3.4 Bagging Summary

Unlike the Single and Boosting algorithm runs, the Bagging ensemble performed the best against the Standard dataset, and outperforms the Single and Boosting variants across all data sets. All of the wrapper datasets were slower than the Single and Boosting equivalents, due to the large difference in number of models being used.

4.8.4 Artificial Neural Network Summary

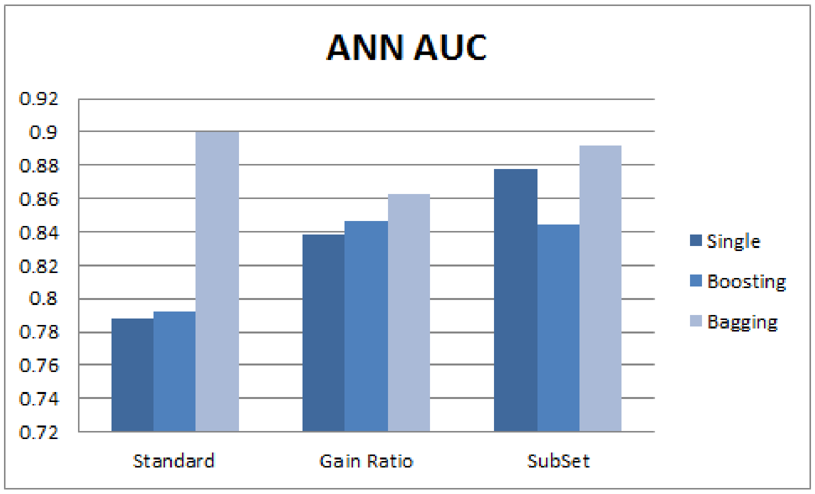


Figure 4.6: Artificial Neural Network Area Under ROC

The mean AUC values are shown in Figure 4.6. It can be seen that the Multilayer Perceptron network has a significant degradation in training time when using ensemble algorithms, however, when dealing with high-dimensionality this tradeoff is largely compensated for with an increase in precision and recall performance. Pre-processing does not see as much of an improvement when using ensemble learning.

4.9 Committee Voting with Multiple Models

As described in Chapter 3, the Voting meta-learner allows multiple heterogeneous models to be combined, such that the strengths (and weaknesses) of each may contribute towards a final prediction value. For the purpose of my research, the *weka.supervised.meta.Voting* [Witten et al., 2011] classifier was tested against both the best-in-class and worst-in-class learners from each dataset and algorithm combination.

4.9.1 Best Set Evaluation

The best-in-class algorithms were selected from the Standard, Boosting, and Bagging ROC AUC value for each algorithm type. Let this set be called the *Best Set*. Table 4.57 lists the algorithms selected to create the Best Set:

The Best Set models, previously created, were combined using an *Average of Probabilities*, such that every classifier contributed towards the average. The final result was then used to label the instance. As table 4.57 shows, a different set was created for each of the Standard, Gain Ratio, and Subset Evaluation.

Table 4.57: Classifiers for Best-Case Voting

Raw	GainRatio	Subset
NB (bag)	NB	NB
J48 (bos)	J48 (bos)	J48 (bag)
Log (bos)	Log (bos)	Log (bos)
iBk (bag)	iBk (bag)	iBk (bag)
SVM (bos)	SVM (bos)	SVM (bos)
ANN (bag)	ANN (bag)	ANN (bag)

4.9.1.1 Standard Dataset

The Best Class Voting committee was run against the Standard dataset, and took 20 seconds to assemble the model on the dataset, using the platform described in Section 3.4.1. This combination utilized the previously built and serialized models created in each trial. The ensemble took 1 hour, 58 minutes and 12 seconds to test all instances.

Table 4.58: Voting Committee Best-Case Standard Confusion Matrix.

a	b	classified as
9407	304	a = normal
5136	7697	b = anomaly

Considering the confusion matrix in Table 4.58, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.759 average True Positive rate, and 0.19 average False Positive rate. The average Precision and Recall measurements were 0.826 and 0.759 respectively. The mean AUC was 0.945 — the greatest accuracy seen during the entire experiment at this point.

4.9.1.2 Gain Ratio Dataset

The Best Class Voting committee was run against the Gain Ratio dataset, and took 3 seconds to assemble the model on the dataset, using the platform described in Section 3.4.1. This combination utilized the previously built and serialized models created in each trial. The ensemble took 1 hour, 10 minutes and 41 seconds to test all instances.

Table 4.59: Voting Committee Best-Case Gain Ratio Confusion Matrix.

a	b	classified as
9416	295	a = normal
4839	7994	b = anomaly

Considering the confusion matrix in Table 4.59, the following performance metrics were produced: the algorithm using the Gain Ratio dataset produced a 0.772 average True Positive rate, and 0.18 average False Positive rate. The average Precision and Recall measurements were 0.834 and 0.772 respectively. The mean AUC was slighly lower than the standard dataset at 0.937.

4.9.1.3 Subset Evaluation Dataset

The Best Class Voting committee was run against the Subset Evaluation dataset, and took 7 seconds to assemble the model on the dataset, using the platform described in Section 3.4.1. This combination utilized the previously built and serialized models created in each trial. The ensemble took 1 hour, 14 minutes and 7 seconds to test all instances.

Considering the confusion matrix in Table 4.60, the following performance metrics were produced: the algorithm produced a 0.764 average True Positive rate, and 0.185

Table 4.60: Voting Committee Best-Case Subset Confusion Matrix.

a	b	classified as
9462	249	a = normal
5079	7754	b = anomaly

average False Positive rate. The average Precision and Recall measurements were 0.832 and 0.764 respectively, very similar to the filter dataset As with the Precision and Recall, the mean AUC, with a value of 0.936, was almost identical with the Gain Ratio dataset.

4.9.1.4 Best Set Summary

The Best Set Voting committee ultimately outperformed every homogeneous single and ensemble learning algorithm system previously tested. This observation holds true for all of the Standard, Gain Ratio, and Subset Evaluation data sets.

4.9.2 Worst Set Evaluation

The worst-in-class algorithms were selected from the Standard, Boosting, and Bagging ROC AUC value for each algorithm type. Let this set be called the *Worst Set*. Table 4.61 illustrates the algorithms selected to create the Worst Set:

Table 4.61: Classifiers for Worst-Case Voting

Raw	GainRatio	Subset
NB (bos)	NB (bos)	NB (bos)
J48	J48	J48
Log	Log (bag)	Log
iBk	iBk	iBk
SVM	SVM (bag)	SVM (bag)
ANN	ANN	ANN (bos)

The Worst Set models, previously created, were combined using an *Average of Probabilities*, such that every classifier contributed towards the average. The final result was then used to label the instance. As illustrated in table 4.61, a different set was created for each of the Standard, Gain Ratio, and Subset Evaluation.

4.9.2.1 Standard Dataset

The Worst Class Voting committee was run against the Standard dataset, and took 4.32 seconds to assemble the model on the dataset, using the platform described in Section 3.4.1. This combination utilized the previously built and serialized models created in each trial. The ensemble took 8 minutes and 56 seconds to test all instances. This discrepancy in testing times between Best and Worst class can be intuitively attributed to the far few number of ensemble models that have been included in the voting committee.

Table 4.62: Voting Committee Worst-Case Standard Confusion Matrix.

a	b	classified as
9361	350	a = normal
4791	8042	b = anomaly

Considering the confusion matrix in Table 4.62, the following performance metrics were produced: the algorithm using the standard dataset produced a 0.722 average True Positive rate, and 0.181 average False Positive rate. The average Precision and Recall measurements were 0.83 and 0.772 respectively. The mean AUC was 0.882; not as good as any of the Best Class committees, but definitely not the worst measurement seen previously.

4.9.2.2 Gain Ratio Dataset

The Worst Class Voting committee was run against the Gain Ratio dataset, and took 2.19 seconds to assemble the model on the dataset, using the platform described in Section 3.4.1. This combination utilized the previously built and serialized models created in each trial. The ensemble took 6 minutes and 53 seconds to test all instances.

Table 4.63: Voting Committee Worst-Case Gain Ratio Confusion Matrix.

a	b	classified as
9417	294	a = normal
5178	7655	b = anomaly

Considering the confusion matrix in Table 4.63, the following performance metrics were produced: the algorithm using the Gain Ratio dataset produced a 0.757 average True Positive rate, and 0.191 average False Positive rate. The average Precision and Recall measurements were 0.826 and 0.757 respectively. The mean AUC was slightly higher than the standard dataset at 0.919.

4.9.2.3 Subset Evaluation Dataset

The Worst Class Voting committee was run against the Subset Evaluation dataset, and took 2.05 seconds to assemble the model on the dataset, using the platform described in Section 3.4.1. This combination utilized the previously built and serialized models created in each trial. The ensemble took 7 minutes and 5 seconds to test all instances; a little longer than the filter dataset.

Considering the confusion matrix in Table 4.64, the following performance metrics were produced: the algorithm produced a 0.757 average True Positive rate, and 0.19 average False Positive rate, nearly indistinguishable from the Gain Ratio filter set.

Table 4.64: Voting Committee Worst-Case Subset Confusion Matrix.

a	b	classified as
9467	244	a = normal
5228	7605	b = anomaly

The average Precision and Recall measurements were 0.829 and 0.757 respectively, with a slightly higher Precision, not not quite as high as the Standard set. The mean AUC, with a value of 0.892, was not as high as the Gain Ratio Dataset, but higher than the Standard set.

4.9.2.4 Worst Set Summary

The Worst Set Voting committee favoured the Gain Ratio dataset. However, as anticipated it was still outperformed by the Best Set voting committees in every dataset. Despite the drastic reduction in training and test time, the overall ROC AUC was only a few percentage lower than the Best Set voting committees.

4.9.3 Committee Voting Summary

As seen in figure 4.7, the Standard dataset performed the best overall, followed closely by the Gain Ratio and Subset Evaluation, which had roughly the same performance relative with one another. This demonstrates the power behind capitalizing on the strengths of each single and ensemble classification systems, and appears to be a systematic advantage despite the possible weaknesses contributed by the same algorithms.

More interesting still, the Worst Set committees performed better than, or tied, every previously observed homogeneous algorithm set tested on the Gain Ratio and

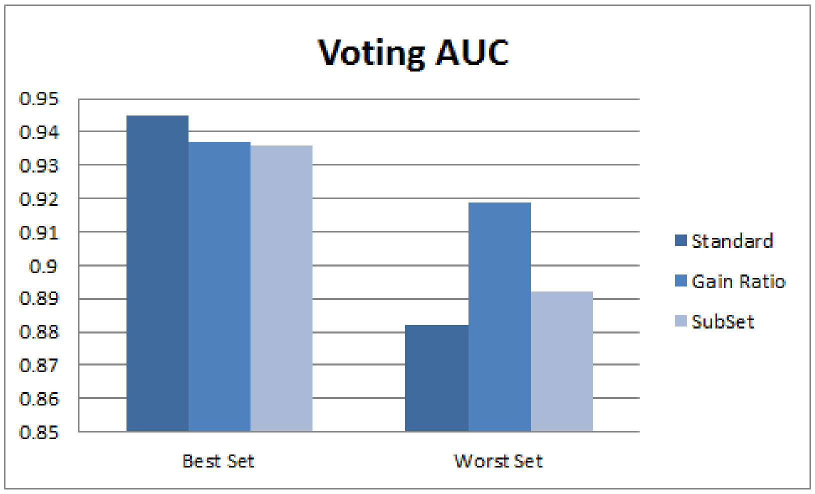


Figure 4.7: Voting Area Under ROC

Subset Evaluation data sets. The Standard dataset showed a slightly different result, as the Worst Set committee was outperformed by one of the homogeneous algorithm sets in each class, with the exception of the iBk and SVM sets. That being said, the Worst Set voting committee still outperformed the worst model found in every set by a fairly large margin.

As was seen with the Best Set voting committee, this unique combination - despite being composed of the worst-in-class models previously created - demonstrates the power behind capitalizing on the strengths of each single and ensemble classification systems. In this case, it appears the strengths help to overcome the obvious classification weaknesses inherent in the worst-in-class models.

4.10 Comparison of Results

The previous sections in this chapter have discussed the performance of each algorithm and dataset combination from the perspective of each algorithm. This section will take a more holistic view, comparing the performance of the algorithms to one another, in terms of overall testing and training times as well as accuracy.

4.10.1 Training Time Comparison

The training times for each of the datasets used are illustrated in the following Figures: Figure 4.8 illustrates all algorithms for the unmodified dataset; Figure 4.9 illustrates all algorithms for the Gain Ratio dataset; and Figure 4.10 illustrates all algorithms for the Subset Evaluation dataset. All of the figures are sorted by seconds in order from greatest to smallest, on a logarithmic scale for easier visualization.

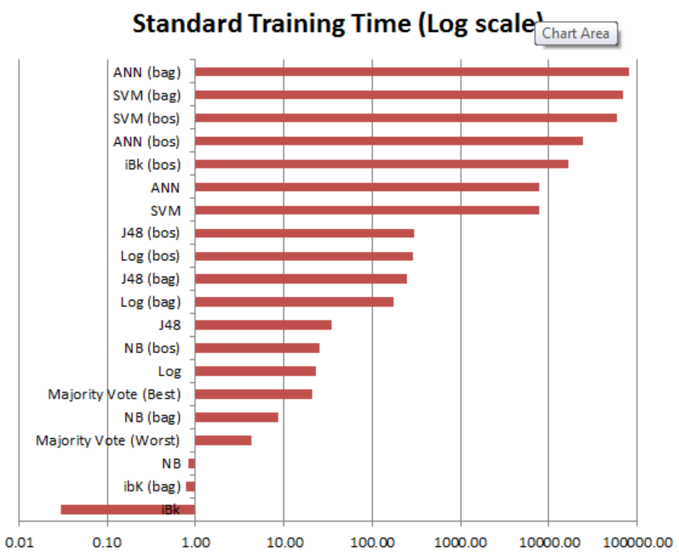


Figure 4.8: Standard Dataset Training Time in seconds

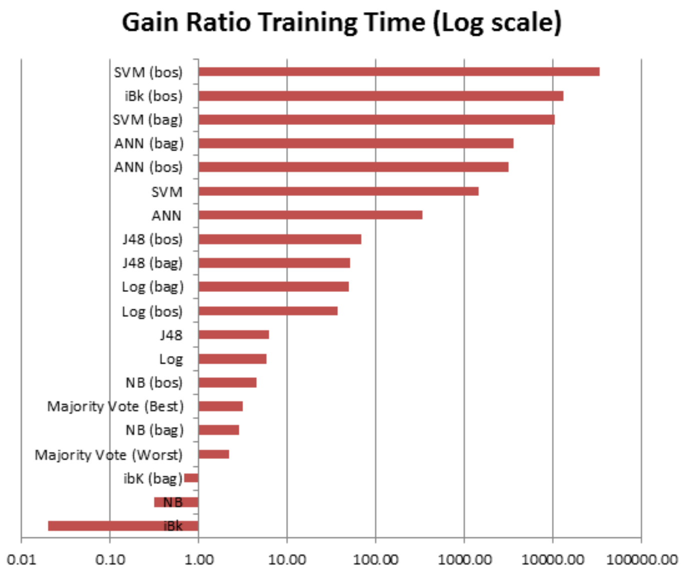


Figure 4.9: GainRatio Dataset Training Time in seconds

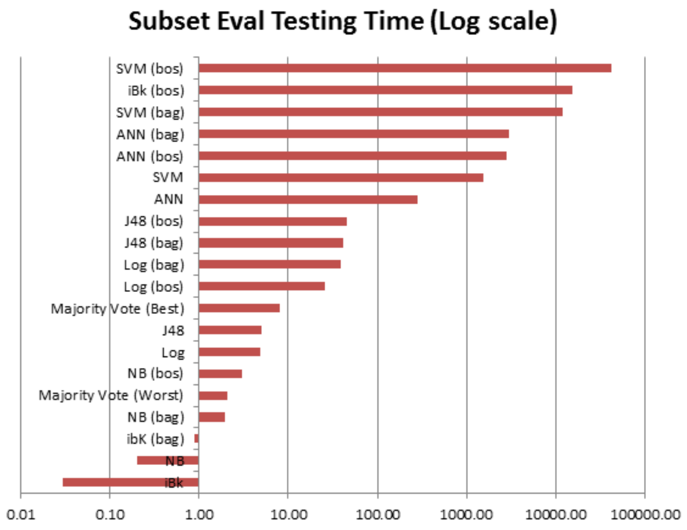


Figure 4.10: Subset Evaluation Dataset Training Time in seconds

The algorithm training times are widely distributed across every dataset. As expected, the more complex algorithms such as artificial neural networks and support vector machines require more time to train than less complex algorithms such as decision trees or logistic regression. This also applies to the ensembles for each algorithm type, as expected. Also of note is that the Voting meta classifiers take far less time than the rest, due to the utilization of models that have already been generated. Performance in this case is limited by the meta-logic required to calculate the committee results.

4.10.2 Testing Time Comparison

The testing times for each of the datasets used are illustrated in the following Figures: Figure 4.11 illustrates all algorithms for the unmodified dataset; Figure 4.12 illustrates all algorithms for the Gain Ratio dataset; and Figure 4.13 illustrates all algorithms for the Subset Evaluation dataset. All of the figures are sorted by seconds in order from greatest to smallest, on a logarithmic scale for easier visualization.

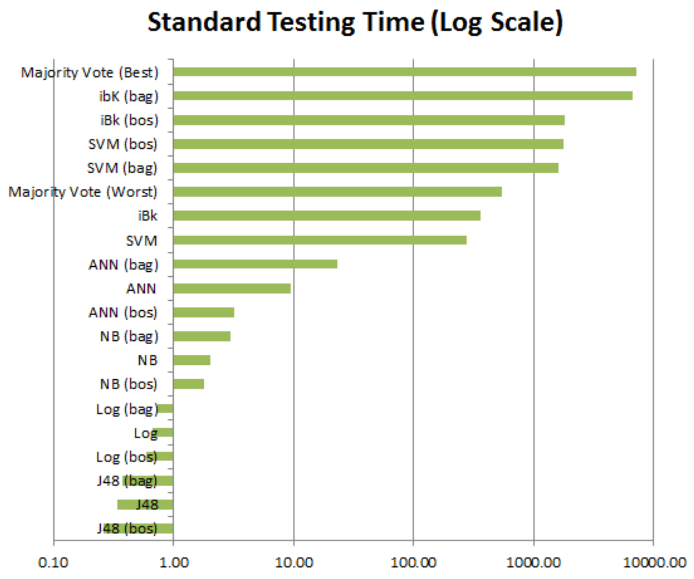


Figure 4.11: Standard Dataset Testing Time in seconds

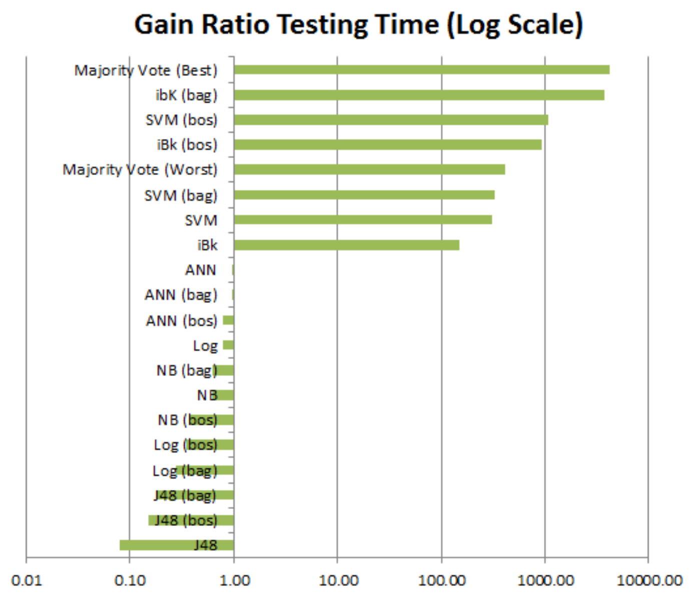


Figure 4.12: GainRatio Dataset Testing Time in seconds

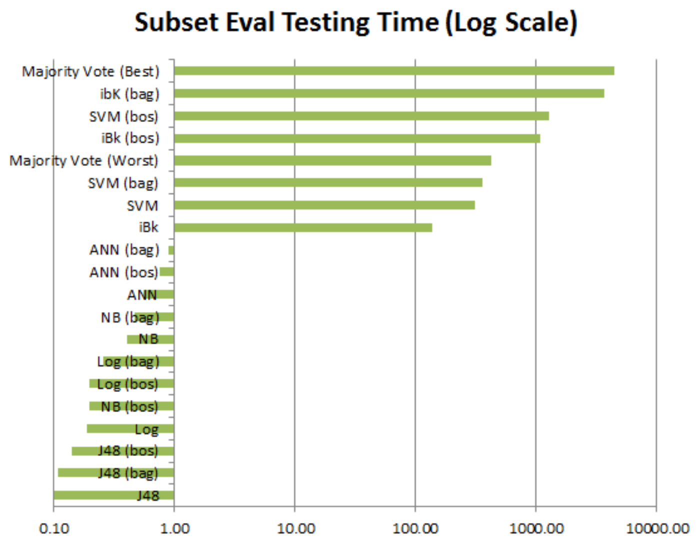


Figure 4.13: Subset Evaluation Dataset Testing Time in seconds

As with the training times, the algorithm test times are also widely distributed. It is interesting to note that the voting ensembles take the longest, as there are multiple algorithms to consult. The previously more expensive learning algorithms, due to their quick access (decision tree) and propagation (neural network), are somewhat faster. The simplest of algorithms (naive Bayes) remains orders of magnitude faster. The difference in testing time is much more apparent with pre-processed data, due to drastic reduction in dimensionality.

4.10.3 AUC Comparison

The ROC AUC for each of the datasets used are illustrated in the following Figures: Figure 4.14 illustrates all algorithms for the unmodified dataset; Figure 4.15 illustrates all algorithms for the Gain Ratio dataset; and Figure 4.16 illustrates all algorithms for the Subset Evaluation dataset. All of the figures are sorted by accuracy in order from highest to lowest.

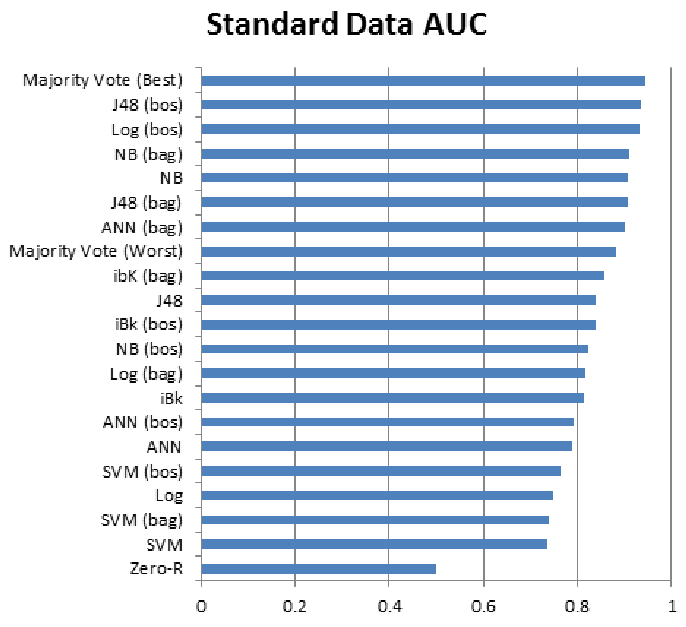


Figure 4.14: Standard Dataset AUC

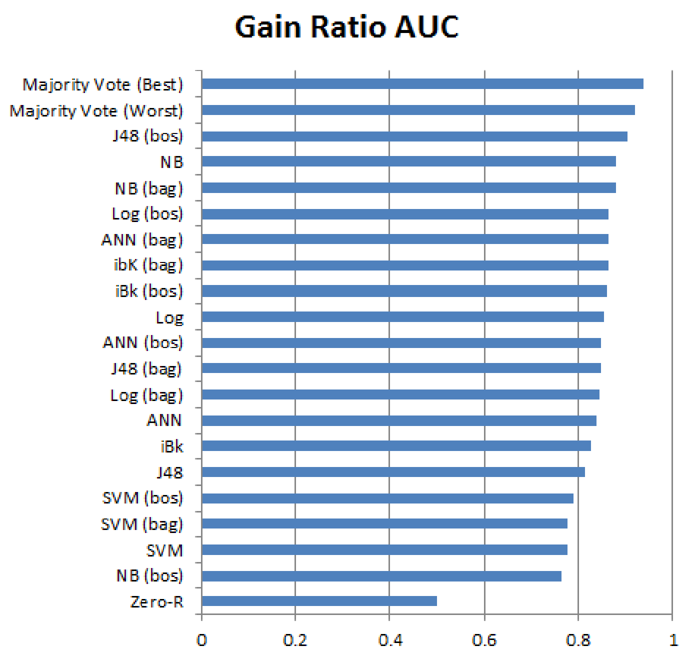


Figure 4.15: GainRatio Dataset AUC

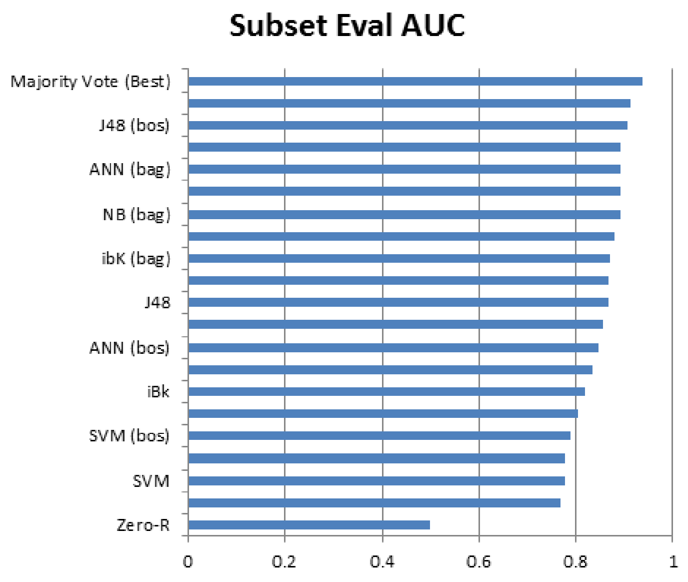


Figure 4.16: Subset Evaluation Dataset AUC

The difference in accuracy between algorithms is much less apparent than the differences previously seen in the training and testing times. As discussed earlier, the Majority Voting classifier does have the highest accuracy, but also at a much greater cost. ZeroR, as expected, is the worst-case lower bound against which all others are compared. The accuracy ordering of algorithms is not as distinguishable simply based the use of ensembles, however they do fall into a similar ordering by accuracy across datasets.

4.10.4 Training Time to AUC Tradeoff

As identified in the previous sections, the difference between testing time over the algorithm and dataset combinations is orders of magnitude greater than the difference in overall accuracy.

The ROC AUC to testing time comparison for each of the datasets used are illustrated in the following figures: Figure 4.17 illustrates all algorithms for the unmodified dataset; Figure 4.18 illustrates all algorithms for the Gain Ratio dataset; and Figure 4.19 illustrates all algorithms for the Subset Evaluation dataset.

Similar to the ROC curve, the figures illustrate the tradeoff between time and accuracy, with the most desirable algorithm combinations approaching the top-left of the graph (high accuracy, low testing performance time). As observed with standard dataset, the testing time distribution is more gradual over all algorithms, as opposed to the pre-processed data with a more obvious clustering of expensive algorithms. This demonstrates the relatively little difference in overall accuracy compared to the extremely large difference in network flow processing performance.

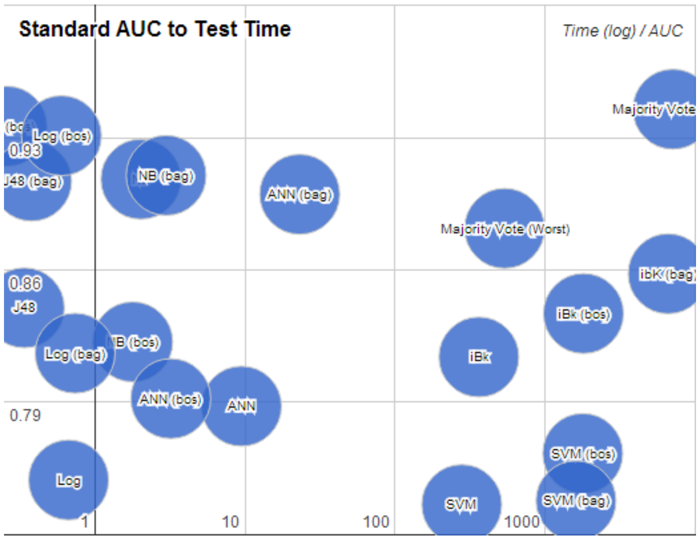


Figure 4.17: Standard AUC to Testing Time

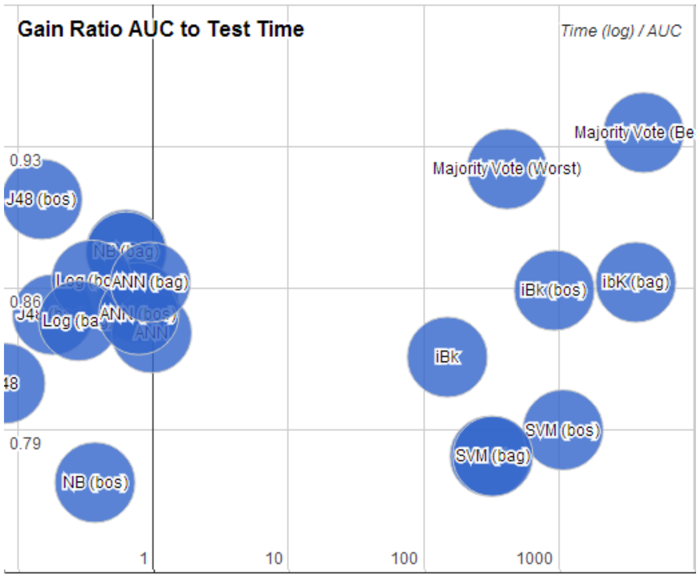


Figure 4.18: Gain Ratio AUC to Testing Time

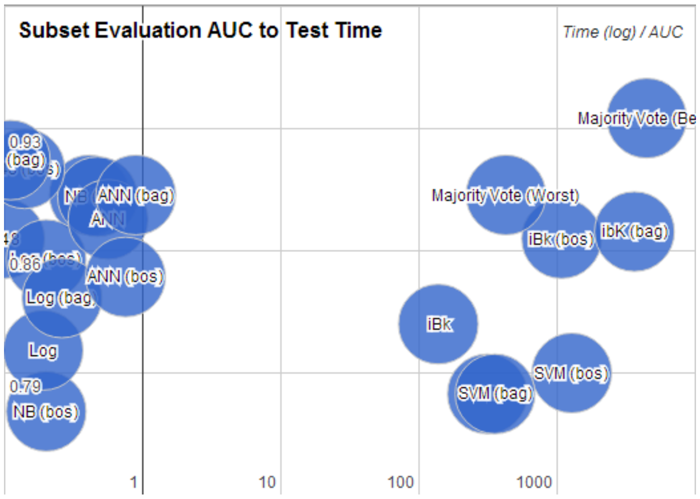


Figure 4.19: Subset Evaluation AUC to Testng Time

4.11 Summary

This chapter reviewed a standard sample of machine learning algorithms, from the simplistic Zero-R and Naive Bayes, to the more sophisticated and expensive Support Vector Machine and Artificial Neural Network. I reviewed the effects of dimensionality reduction on the overall training and testing time, as well as the effects of over-fitting and generalization granted by selecting the proper dataset size for each algorithm.

These experiments were performed utilizing a large number of combinations in order to demonstrate a number of factors. The basic confusion matrix was used to derive more intelligent measuring tools such as the True Positive and False Positive rate, the Precision and Recall, all of which can be combined into one summarized score through the Receiver Operating Curve.

The evaluation considered the effects of ensemble learning techniques such as serialized Boosting, and parallel Bagging, on each of the algorithm types. The pre-processed data sets — Standard, Gain Ratio, and Subset Evaluation — were tested against all of the algorithm types, single and in ensemble combination, to measure the overall accuracy as well as testing and training time taken for each combination, allowing us to compare and contrast the effects of these combinations.

Finally, the Voting meta-classifier was used to combine homogeneous learning models into a committee-based averaging system. The best and worst algorithms produced by the homogeneous combinations were assembled into a heterogeneous system, and using the lower and upper bounds of previously observed testing set performance, demonstrated the power of using meta-classifiers to increase overall accuracy.

These results produced a macro-view of machine learning learning experiments utilizing the NSL-KDD training and testing datasets. The next chapter will revisit the research questions introduced in Chapter 1, and attempt to answer each of these given the observations just presented to the reader.

Chapter 5

Contributions and Future Research

5.1 Overview

The previous chapter explored the myriad choices available to the machine learning community, from simple and complex algorithms, data pre-processing options, as well as more advanced techniques of model combination through ensembles and meta-classifications. In this chapter, the research questions proposed in Chapter 1 will be revisited, and answered based on observed trends in performance. The learning scheme results will be presented here so that the original questions can be answered, and some simple rules will be derived to guide future experiments in this problem domain.

5.2 Observations

The original research questions, as well as some additional observations, are here restated and explored in light of the experimentation findings from Chapter 4.

1. Which individual algorithms are the most accurate?

   Based on the results of my experiments, the Naive Bayes learner had the highest AUC (0.908) when used as a single algorithm against the standard high dimensionality dataset. This learning scheme does not consider any form of dataset pre-processing, or ensemble learning.

2. Which are the fastest algorithms by learning time?

   Based on the results of my experiments, the iBk (K-Nearest Neighbour) learner had the fastest learner build time (0.03 seconds) when used as a single algorithm against the standard high dimensionality dataset.

3. Which are the fastest algorithms by testing time?

   Based on the results of my experiments, the Logistic Regression learner had the fastest learner test time (0.67 seconds) when used as a single algorithm against the standard high dimensionality dataset.

4. What are the effects of pre-processing on the dataset, and overall time and accuracy measurements?

   As we discovered in Chapter 4, pre-processing the data based on information gain, or subset evaluation, reduces redundant features that do not appear to individually contribute to learning accuracy through information gain, or dependent features that do not contribute to any subset of features. As seen in the majority of the experiments, applying pre-processing to the datasets does increases the AUC by the following metrics.

Single algorithms using pre-processed data increased in accuracy in the majority of the cases, with Subset Evaluation having the highest mean AUC of 83.93% across all algorithms. Only Naive Bayes did not increase in AUC, and performed the best on an unaltered dataset.

When using Boosting, only half of the algorithms favoured pre-processed data (iBk, SVM, ANN), while the rest preferred unaltered data. The boosted single dataset had the highest mean AUC of 84.68%

When using Bagging, the results are similar to the single algorithms: only Naive Bayes preferred the original dataset. 66% of the algorithms preferred Subset Evaluation, with a mean AUC of 86.28%.

From these results we can see that pre-processing has the greatest effect on single and parallel learning models, where serialized resampling is not affected. The greatest mean AUC gain was found in Subset Evaluation.

Every single algorithm scheme, without exception, produced a decrease in overall training and testing time when using a pre-processed dataset. This is, intuitively, due to a reduction in overall computation costs when dealing with a fewer number of dimensions.

5. What are the effects of ensembles on the standard and derived datasets?

As the experiments in Chapter 4 demonstrated, resampling-based ensemble learning is preferable compared to single algorithms in the majority of the cases. This could be due to the reduction in over-fitting bias found in some of the more sophisticated algorithms that use conditional probabilities. By training them over different regions of the sample space this can be reduced.

For the standard dataset, every algorithm produced a higher AUC when using boosting or bagging.

The Gain Ratio and Subset evaluation data sets also produced a higher AUC when using boosting or bagging, with the exception of Naive Bayes which preferred the single algorithm over ensemble. 17 out of 18 algorithm-to-dataset schemes tested (94%) preferred ensemble learning.

Of all the combinations tested, Boosting was preferred by 8 out of 18 (roughly 44%), Bagging preferred by 8 out of 18 (roughly 44%), with the single algorithm on preferred by 2 out of 18 (roughly 12%) of the testing combinations.

Every tested ensemble scheme, with few exceptions, produced an increase in overall training and testing time, due to the number of iterations over the dataset when multiple models were produced serially (when boosting), and in parallel (when bagging).

6. What are the effects of meta-classification on overall performance?

   The experiments in Chapter 4 demonstrated the effect of voting through an average of probabilities to demonstrate the effect of combining multiple heterogeneous algorithms.

The Best Set Voting committe ultimately outperformed every homogeneous single and ensemble learning algorithm system previously tested. This observation holds true for all of the Standard, Gain Ratio, and Subset Evaluation data sets.

The Worst Set Voting committee favoured the Gain Ratio dataset. However, as anticipated it was still outperformed by the Best Set voting committees in every dataset. Despite the drastic reduction in training and test time, the overall ROC AUC was only a few percentage lower than the Best Set voting committees.

More interesting still, the Worst Set committees performed better than, or tied, every previously observed homogeneous algorithm set tested on the Gain Ratio and Subset Evaluation data sets. The Standard dataset showed a slightly different result, as the Worst Set committee was outperformed by one of the homogeneous algorithm sets in each class, with the exception of the iBk and SVM sets. That being said, the Worst Set voting committee still outperformed the worst model found in every set by a fairly large margin.

As was seen with the Best Set voting committee, this unique combination, despite being composed of the worst-in-class models previously created, demonstrates the power behind capitalizing on the strengths of each single and ensemble classification systems. In this case, it appears the strengths help to overcome the obvious classification weaknesses inherent in the worst-in-class models.

7. How do we appropriately determine what the "best" algorithm is?

   As we explored the concept of what matters when classifying network traffic, multiple factors were a consideration. Some of these factors include the value of tradeoff between true positive, false positive, true negative, and false negative, and how precision and recall can reflect the attribute choices through over-fitting and bias. The overall variance of an algorithm can determine the confidence we have in that particular scheme. However this can only be found through multiple iterations of cross validation which is computationally expensive, or utilizing a large number of unique datasets which is difficult to obtain. Training and testing time can be extremely low with simple yet efficient algorithms, or skyrocket when combining sophisticated algorithms. When processing and classifying large streams of flow data, sometimes the most accurate learning system isn't even a consideration.

In domains where any false negatives can mean disaster, such as when dealing with zero day attacks in network security, and potential for compromise, having the highest traffic classification accuracy can be of paramount importance, with compensatory controls such as traffic multiplexing to deal with high volumes and keep up with instance classification requirements. In this case, a scheme such as Logistic Regression resampling, or meta-classification through Voting, produces even higher classification accuracy, with a sacrifice to the number of instances processed per second.

When tasked with such a large hyper-parameter space within schemes, and the multiple combinations of pre-processing and learning algorithm combinations, it is a difficult task to definitively state that one solution is the best in every scenario — the proverbial "Holy Grail" of machine learning in the network intrusion detection domain. If all the aforementioned factors are of consideration, then it may be helpful to consider the tradeoff of accuracy to time, and choose the solution that encompasses the performance factors that are most important to the problem domain.

5.3 Suggested Guidelines

Based on the above answers, it may be possible to suggest a number of simple guidelines when choosing a machine learning scheme for network traffic classification based on flow data behavior. The results discussed in Section 5.2 demonstrate that the primary factors of classification scheme, as well as the pre-processing algorithm chosen, have the biggest impact on expected classification performance. Considering what has been discussed so far, I present a series of simple guidelines that may help choose the best learning scheme to begin with, depending on the circumstances.

1. When there is no intention to run pre-processing or ensemble schemes, choose Naive Bayes for classification (from Observation 1).

2. When learning time (model creation or updating) is the most important factor, use Naive Bayes or iBk algorithms (from Observation 2).

3. When classification time (events per second) is the most important factor, use Naive Bayes, Logistic Regression, or Multilayer Perceptrons (from Observation 3).

4. When the information gain of a dataset is unknown, choose Subset Evaluation for pre-processing (from Observation 4).

5. When time is not a factor, use ensemble learning to reduce overfitting, and increase the accuracy of homogeneous algorithms (from Observation 5).

6. When overall AUC is the biggest factor with no concern about training or testing time, use Voting meta classification with a committee of heterogeneous learning models (from Observation 6).

7. When all of accuracy, training time, and testing time are factors of concern, a simpler algorithm (Naive Bayes, J48 Decision Tree, or Logistic Regression) will generally give the best overall tradeoff of accuracy to time. When accuracy is the most important, a more sophisticated scheme like Voting is the most appropriate (from Observation 7).

5.4 Contributions

My research is focused on the application of multiple machine learning algorithms to network intrusion detection. The results of this application have shown an understanding of classification accuracy for each major supervised learning algorithm type, as well as the effects of applying traditionally simple and sophisticated classifier models to a raw, filtered, and wrappered feature set. My research has demonstrated the effects of ensemble learning over all classifiers and pre-processed datasets, as well as the cumulative effect of meta-heuristic voting systems on overall classification performance in time and accuracy.

The major contributions of this work are as follows:

1. A training and testing methodology that has been used to test both individual

and combined machine learning algorithms, covering areas such as the effects of pre-processing, ensembles, and meta-heuristics. This methodology has covered the staple components of machine learning in any domain, from the effects of dimensionality reduction, algorithm parameter selection, and algorithm combinations in a variety of ways.

2. A demonstrated correlation between features and overall classifier accuracy in the NSL-KDD dataset. This demonstration has proved the efficacy of dataset pre-processing.

3. An in-depth analysis of individual classifier performance metrics, such as precision, recall, bias, variance, and runtime. This analysis has shown the overall performance improvement gains from the methodology above.

4. A novel analysis of boosting, bagging, and voting algorithms using a combination of weaker single classifiers, and stronger more sophisticated learning schemes. This analysis has demonstrated which of the single classifier models improve when combined for homogeneous serial learning subsets of mis-classified feature sets, combinations into homogeneous parallel learning schemes, as well as heterogeneous council voting for best-of-breed outcomes.

5. A demonstration of machine learning algorithm performance when faced with novel attack types never seen in prior training sets.

The classification framework and algorithms used in my thesis can be adapted to real-world flow data using similar flow data transformations, in combination with offline training and online testing. Real-world applications in high-performance computing environments would likely include using classifier libraries that run natively in the target architecture, along with optimizations to multi-threading where applicable. However, assuming the observed traffic features are similar in nature to the NSL-KDD sets used in my thesis, the accuracy and time tradeoffs would also remain similar. Some of these issues will be discussed in the next section, along with possibilities for future analysis in the problem domain.

5.5 Future Research

This research involved the application of a large number of supervised learning algorithms for the purpose of anomaly detection. My thesis covered a broad survey of common tasks found in optimizing overall performance and tuning to the dataset in question, as well as more sophisticated application of combining multiple machine learning algorithms together.

The time and space requirements of this research have been limited by the appropriate scope of research for a Master's Thesis, as machine learning in itself is a massive subject area with many smaller areas of specialization. While performing this research, I have identified a number of possible future topics that would be interesting to explore in continuance of my journey in network traffic classification for the purpose of intrusion and fraud detection. Some of these topics are discussed in the following subsections.

5.5.1 Sub-Selective Ensembles

The voting ensemble learning experiments included the worst-of-breed and best-of-breed classifiers from each algorithm and pre-processed dataset combination. There were a total of six classifiers included in the voting structure. Considering the number of classifiers used, all sub-sample voting combinations (approximately 720) would be infeasible given the time frame available for experimentation. An experiment into which voting combinations produced the best accuracy would be interesting given the individual strengths and weakness in bias, variance, accuracy, and speed, of each classifier type.

5.5.2 Hyper-Parameter Optimization

My research used the default (i.e., best practice) settings for most of the algorithms tested, with a few deviations. Every algorithm is unique, and will perform differently depending on the dataset in question. Some of the algorithms tested, such as Naive Bayes, or J48, had very few options to consider when implementing them. Some of the more sophisticated algorithms like SVM, or ANNs, have more parameters that are tunable (such as learning rate, number of epochs, etc.). As discussed in Section 3.4.3, individual algorithm tuning is typically done using 10-fold cross validation before testing against the test set. In this thesis, most of the tuning was done on the macro-scale; combining pre-processed data sets with ensemble combinations. Considering the massive parameter space available to the more sophisticated algorithms, future research could be performed into optimal methods of finding the right parameters for each dataset/algorithm combination, in order to further increase the generalizability of the algorithms without sacrificing the bias or variance.

5.5.3 Unsupervised Behavior Profiling

This research focused on supervised learning, partially out of the convenience of having a labeled dataset. Many real world environments do not have a labeled dataset, in part due to the difficulty of identifying all malicious behavior (we wouldn't want to baseline our Intrusion Detection System to think attacks are normal), and partly because widely available sanitized datasets are few and far between. Unsupervised learning may hold the key to clustering behavior attributes together such that new behaviors can be classified once existing profiles have been identified with a higher degree of certainty.

5.5.4 Distributed Learning Architecture

The Weka Explorer environment was primarily used in order to perform the experiments, with a single machine as the platform. Many of the models created during my research took an excess of 20 hours to finish, and required a 2GB heap allocation. The serialized Voting models required an excess of 250MB each, given they are built on multiple, equally complex ensembles. Some of the previously identified experimental models, such as *Stacking* [Wolpert, 1992] (a variant of serialized Boosting using heterogeneous classifiers) would have been interesting to test, but were plainly unfeasible given the computation power available against the higher dimensionality of the raw datasets. Future research could utilize the distributed and multi-threaded architecture made available through Weka and other test platforms in order to finish building models of this size and complexity.

5.5.5 Multiple-Class Attack Categorization

Anomaly detection in this domain only requires a binary classifier to flag traffic as suspicious. As discussed in Section 3.2.2, the sample traffic was also available by attack type class. Creating learners around all attack types for our purposes does not make sense, given novel attacks types would not have a class assigned. However, all attacks can be largely categorized into their macro behavior: for example, Denial of Service, Reconnaissance, Local to Remote, and User to Root. Future research could involve a similar testing methodology, but with an increased number of classes to choose from when categorizing traffic. This would be useful in determining the behavior type behind the anomaly, not just the presence of suspected fraud.

5.5.6 Real-Time Online Learning

My experiments involved processing training samples in large batches, which is a behavior typically performed when a large amount of historical data is available. Many real-world problems do not afford the convenience of a large historical dataset for training purposes. Future research into real-time learning - for example a network tap - would prove useful for environments that require a fresh deployment and learning on-the-fly when network topologies suddenly change.

5.5.7 Incremental Learning Rate Analysis

Most of the algorithms compatible with the Weka framework did not support the *Updatable* interface, which would allow per-instance accuracy metrics to be observed during training. This can, however, be dealt with by applying non-randomized *instance-based filters* to train each algorithm, and observe the gain by incremental size. Future research could involve developing Updatable interface support for a larger proportion of the ML libraries in use.

5.6 Conclusion

In this thesis, I examined the application of a number of machine learning algorithms in a supervised learning environment to a number of dataset combinations. My experiments explored the effects of filter and wrapper preprocessing on the dataset, and the effects on algorithm classification accuracy and time performance. My research also explored the effects of running boosting and bagging ensemble learning, as well as meta-classification through voting. Through these experiments, I have demonstrated that it is possible to increase the overall performance of learning algorithms in the network security domain. I have also demonstrated that various algorithm combinations can have a detrimental effect on training and testing time, and that researchers should be wary of the tradeoffs from choosing pre-processor, algorithm, and learning scheme configuration. My research contributions have shown that there is no "Holy Grail" for intrusion detection using current machine learning techniques, but with foreknowledge an appropriate solution can be selected given known practical requirements.

Bibliography

An empirical study of the naive Bayes classifier. In *International Joint Conference on Artificial Intelligence*, pages 41–46. American Association for Artificial Intelligence, 2001.

K. C. 99. Darpa challenge, Accessed November 1999. `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`.

D. W. Aha, D. Kibler, and M. K. Albert. Instance-based learning algorithms. *Mach. Learn.*, 6(1):37–66, Jan. 1991. ISSN 0885-6125.

M. A. Aizerman, E. M. Braverman, and L. I. Rozonoer. The probability problem of pattern recognition learning and the method of potential functions. *Automation and Remote Control*, 25:1175–1190, 1964.

A. Ali, A. Saleh, and T. Ramdan. Multilayer perceptrons networks for an intelligent adaptive intrusion detection system. *International Journal of Computer Science and Network Security*, 10(2), feb 2010.

M. Allman, V. Paxson, and J. Terrell. A brief history of scanning. In *Proceedings of* *the 2007 Internet Measurement Conference*, pages 145–150, San Diego, California, 2007. ACM.

J. Anderson. Computer security threat monitoring and surveillance. National Institute of Standards and Technology, 1980.

P. Bermejo, J. A. Gámez, and J. M. Puerta. Improving incremental wrapper-based feature subset selection by using re-ranking. In *Proceedings of the 23rd international conference on Industrial engineering and other applications of applied intelligent systems - Volume Part I*, IEA/AIE'10, pages 580–589, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13021-6, 978-3-642-13021-2.

M. Bishop. *Introduction to Computer Security.* Addison Wesley, Reading, MA, 2005.

L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

R. Caruana and A. Niculescu-mizil. An empirical comparison of supervised learning algorithms. In *In Proc. 23 rd Intl. Conf. Machine learning (ICML06*, pages 161–168, 2006.

I. Chairunnisa, Lukas, and H. D. Widiputra. Clustering base intrusion detection for network profiling using k-means, ecm and k-nearest neighbor algorithms. In *Konferensi Nasional Sistem dan Informatika 2009*, nov 2009.

S. K. Choubey, J. S. Deogun, V. V. Raghavan, and H. Sever. A comparison of feature selection algorithms in the context of rough classifiers, 1996.

R. Collobert, S. Bengio, and J. Mariéthoz. Torch: a modular machine learning software library. Technical report, IDIAP, 0 2002.

Y. EL-Manzalawy and V. Honavar. *WLSVM: Integrating LibSVM into Weka Environment*, 2005. Software available at `http://www.cs.iastate.edu/~yasser/wlsvm`.

K. M. Faraoun and A. Boukelif. Neural networks learning improvement using the k-means clustering algorithm to detect network intrusions. *International Journal of Computational Intelligence*, 3(2), 2007.

D. M. Farid and M. Z. Rahman. Anomaly network intrusion detection based on improved self adaptive bayesian algorithm. *JCP*, 5(1):23–31, 2010.

T. Fawcett. Roc graphs: Notes and practical considerations for researchers. Technical report, 2004.

D. Fradkin and I. Muchnik. Dimacs series in discrete mathematics and theoretical computer science support vector machines for classification, 2000.

Y. Freund and R. Schapire. A short introduction to boosting. In *Journal of Japanese Society for Artificial Intelligence*, pages 771–780, 1999.

Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm, 1996.

J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: a statistical view of boosting. *Annals of Statistics*, 28:2000, 1998.

C. W. Geib and R. P. Goldman. A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence*, 173(11):1101–1132, 2009. ISSN 0004-3702.

F. Giroire, J. Chandrashekar, G. Iannaccone, K. Papagiannaki, E. M. Schooler, and N. Taft. The cubicle vs. the coffee shop: Behavioral modes in enterprise end-users. In *Proceedings of the 2008 Passive and Active Measurement Conference*, pages 202–211, Berlin, 2008. Springer.

N. Gornitz, M. Kloft, K. Rieck, and U. Brefeld. Active learning for network intrusion detection. In *2nd ACM workshop on security and artificial intelligence*, pages 47–54, 2009.

I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, Mar. 2003. ISSN 1532-4435.

M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11:10–18, November 2009. ISSN 1931-0145.

M. A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.

A. KDD Cup. Kdd cup 99 task description, Oct. 2012. URL `http://kdd.ics.uci.edu/databases/kddcup99/task.html`.

K.-C. Khor, C.-Y. Ting, and S.-P. Amnuaisuk. From feature selection to building of bayesian classifiers: A network intrusion detection perspective. *American Journal of Applied Sciences*, 6(11):1949–1960, 2009.

J. Kittler, M. Hatef, R. P. W. Duin, and J. Matas. On combining classifiers. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(3):226–239, Mar. 1998. ISSN 0162-8828.

M. Kloft, U. Brefeld, P. Dussel, C. Gehl, and P. Laskov. Automatic feature selection for anomaly detection. In *AISEC 2008*, pages 71–76, oct 2008.

S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. informatica 31:249268, 2007.

L. I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. John Wiley and Sons, Inc., 2004.

P. T. Leeson and C. J. Coyne. The economics of computer hacking. *Journal of Law, Economics and Policy*, 2006.

Q. Liu, J.-p. Yin, Z.-p. Cai, and M. Zhu. A novel threat assessment method for ddos early warning using network vulnerability analysis. In *Proceedings of the 2010 Fourth International Conference on Network and System Security*, NSS '10, pages 70–74, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4159-4.

J. Makhoul, F. Kubala, R. Schwartz, and R. Weischedel. Performance measures for information extraction. In *In Proceedings of DARPA Broadcast News Workshop*, pages 249–252, 1999.

S. McClure, J. Scambray, and G. Kurtz. *Hacking Exposed*. McGraw Hill, Emeryville, CA, 2005.

J. McHugh. Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. 3 (4):262–294, 2000.

MIT. Darpa intrusion detection evaluation, Oct. 2012. URL `http://www.ll.mit.edu/mission/communications/CST/darpa.html`.

T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

S. Mukkamala and A. Sung. Feature ranking and selection for intrusion detection systems using support vector machines. Technical report, Institute of Minera y Tecnologa, 2003.

U. o. N. B. Network Security Laboratory. The nsl-kdd data set, Oct. 2012. URL `http://nsl.cs.unb.ca/NSL-KDD/`.

A. Ng. Machine learning, Oct. 2012. URL `https://class.coursera.org/ml`.

G. Nychis, V. Sekar, D. G. Andersen, H. Kim, and H. Zhang. An empirical evaluation of entropy-based traffic anomaly detection. In *Proceedings of the 2008 Internet Measurement Conference*, pages 145–150, Vouliagmeni, Greece, 2008. ACM.

A. Patcha and J.-M. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448 – 3470, 2007. ISSN 1389-1286.

M. Pillai, J. Eloff, and H. Venter. An approach to implement a network intrusion detection system using genetic algorithms. In *Proceedings of South African Institude of Computer Scientists and Information Technologists*, pages 221–228, Western Cape, South Africa, 2004. ACM.

L. Portnoy. Intrusion detection with unlabeled data using clustering, 2000.

L. Portnoy, E. Eskin, and S. Stolfo. Intrusion detection with unlabeled data using clustering. In *ACM CCS Workshop on Data Mining Applied to Security*. ACM Press, nov 2001.

D. M. W. Powers. Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation. Technical Report SIE-07-001, School of Informatics and Engineering, Flinders University, Adelaide, Australia, 2007.

J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1-55860-238-0.

F. U. B. Raul Rojas. Adaboost and the super bowl of classifiers: A tutorial introduction to adaptive boosting, 2013. URL `http://www.inf.fu-berlin.de/inst/ag-ki/adaboost4.pdf`.

P. Refaeilzadeh, L. Tan, and H. Liu. On comparison of feature selection algorithms. *DARPA Information Survivability Conference and Exposition,*, 2:1130, 2000.

P. Refaeilzadeh, L. Tang, and H. Liu. Cross-validation. In *Encyclopedia of Database Systems*. Springer US, 2009.

R. E. Schapire. The strength of weak learnability, 1990.

E. Schultz and R. Shumway. *Incident Response: A Strategic Guide to Handling System and Network Security Breaches*. Sams, Indianapolis, IN, 2001.

Shogun. The shogun toolbox, Jan. 2013. URL `http://www.shogun-toolbox.org/`.

S. J. Stolfo, W. Fan, W. Lee, A. Prodromidis, and P. K. Chan. Cost-based modeling for fraud and intrusion detection: Results from the jam project. *DARPA Information Survivability Conference and Exposition,*, 2:1130, 2000.

S. J. Stolfo, S. Hershkop, C.-W. Hu, W.-J. Li, O. Nimeskern, and K. Wang. Behavior-based modeling and its application to email analysis. volume 6, pages 187–221, New York, NY, USA, 2006. ACM.

M. Tavallaee, E. Bagheri, W. Lu, and A. Ghorbani. Detailed analysis of the kdd cup 99 data set. In *Proceedings of 2009 IEEE Symposium on Computational Intelligence in Security and Defense Applications*, Barcelona, Spain, 2009a. CISDA 2009.

M. Tavallaee, E. Bagheri, W. Lu, and A. Ghorbani. A detailed analysis of the kdd cup 99 data set. In *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*, pages 1 –6, july 2009b.

L. Vatisekhovich. Intrusion detection in tcp/ip networks using immune systems paradigm and neural network detectors. In *XI International PhD Workshop*, oct 2009.

S. D. Walter. The partial area under the summary roc curve. *Statistics in Medicine*, 24(13):2025–2040, 2005.

C. Wikimedia. Binary entropy plot, Apr. 2007a. URL `http://commons.wikimedia.org/wiki/File:Binary_entropy_plot.svg`.

C. Wikimedia. K-nearest neighbour, May 2007b. URL `http://commons.wikimedia.org/wiki/File:KnnClassification.svg`.

C. Wikimedia. Logistic curve, June 2008a. URL http://commons.wikimedia.org/wiki/File:Logistic-curve.svg.

C. Wikimedia. Linear regression, June 2007c. URL http://commons.wikimedia.org/wiki/File:LinearRegression.svg.

C. Wikimedia. Roc space, Nov. 2009. URL http://commons.wikimedia.org/wiki/File:ROC_space-2.png.

C. Wikimedia. Support vector machines hyperplanes, Nov. 2012. URL http://commons.wikimedia.org/wiki/File:Svm_separating_hyperplanes_(SVG).svg.

C. Wikimedia. Support vector machines hyperplanes with margins, Feb. 2008b. URL http://commons.wikimedia.org/wiki/File:Svm_max_sep_hyperplane_with_margin.png.

I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. ISBN 0120884070.

I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, Burlington, MA, 3 edition, 2011.

D. H. Wolpert. Stacked generalization. *Neural Networks*, 5:241–259, 1992.

K. Xu, Z.-L. Zhang, and S. Bhattacharyya. Profiling internet backbone traffic: behavior models and applications. In *SIGCOMM '05: Proceedings of the 2005 conference* *on Applications, technologies, architectures, and protocols for computer communications*, pages 169–180, New York, NY, USA, 2005. ACM. ISBN 1-59593-009-4.

K. Xu, F. Wang, S. Bhattacharyya, and Z.-L. Zhang. A real-time network traffic profiling system. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 595–605, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2855-4.

K. Xu, Z.-L. Zhang, and S. Bhattacharyya. Internet traffic behavior profiling for network security monitoring. *IEEE/ACM Trans. Netw.*, 16(6):1241–1252, 2008. ISSN 1063-6692.

N. Ye. *Secure Computer and Network Systems — Modeling, Analysis and Design.* Wiley, Hoboken, NJ, 2008.

Z. Yu, J. Tsai, and T. Weigert. An adaptive automatically tuning intrusion detection system. *ACM Transactions on Autonomous and Adaptive Systems*, 3(3):1–25, 2008.

S. Zanero and S. M. Savaresi. Unsupervised learning techniques for an intrusion detection system. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 412–419, New York, NY, USA, 2004. ACM. ISBN 1-58113-812-1.

Z.-H. Zhou. When semi-supervised learning meets ensemble learning. In *Proceedings of the 8th International Workshop on Multiple Classifier Systems*, MCS '09, pages 529–538, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02325-5.

D. A. Zilberbrand. *Efficient Hybrid Algorithms for Plan Recognition and Detection of Suspicious and Anomalous Behavior.* PhD thesis, Bar-Ilan University, mar 2009.

Appendix A

KDD Traffic Details

A.1 Basic TCP Connection Details

Table A.1: Basic TCP Connection Details

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

A.2 Content Features with a Connection (Domain Knowledge)

Table A.2: Content Features with a Connection (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 logged in; 0 otherwise	discrete
num_compromised	number of “compromised” conditions	continuous
root_shell	1 if root shell is obtained; 0 otherwise	discrete
su_attempted	1 if “su root” command attempted; 0 otherwise	discrete
num_root	number of “root” accesses	continuous
num_file_creations	number of 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	discrete
is_guest_login	1 if the login is a “guest”login; 0 otherwise	discrete

A.3 Traffic Features within a Two Second Time Window

Table A.3: Traffic Features within a Two Second Time Window

feature name	description	type
count	number of connections to the same host as the current connection in the past two seconds *Note: The following features refer to these same-host connections.*	continuous
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	% of connections to different services	continuous
srv_count	number of connections to the same service as the current connection in the past two seconds *Note: The following features refer to these same-service connections.*	continuous
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

A.4 Attacks present in the DARPA 1999 Data Set

Table A.4: Attacks present in the DARPA 1999 Data Set

Attack Class	Attacks in Training Set	Additional Attacks in Test Set
Probe	satan, portsweep, ipsweep, nmap.	mscan, saint
DoS	back, neptune, smurf, teardrop, land, pod	apache2, mailbomb, processtable
R2L	warezmaster, warezclient, ftp-write, guesspassword, imap, multihop, phf, spy	sendmail, named, snmpgetattack, snmpguess, xlock, xsnoop, worm
U2R	rootkit, bufferoverflow, loadmodule, perl	httptunnel, ps, sqlattack, xterm

Appendix B

Single Classification Results

Table B.1: Naive Bayes Results

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.761	0.198	0.809	0.761	0.759	0.572	0.908	0.882	0.83	2.03
Gain Ratio	0.731	0.208	0.822	0.731	0.723	0.557	0.879	0.844	0.32	0.63
Subset	0.737	0.205	0.82	0.737	0.731	0.562	0.892	0.872	0.20	0.41

Table B.2: J48 Decision Tree

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.815	0.146	0.858	0.815	0.815	0.674	0.84	0.828	34.24	0.34
Gain Ratio	0.76	0.19	0.825	0.76	0.756	0.588	0.813	0.802	6.28	0.08
Subset	0.755	0.194	0.823	0.755	0.75	0.581	0.867	0.84	4.94	0.10

Table B.3: Logistic Regression

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.762	0.201	0.803	0.762	0.761	0.566	0.749	0.723	23.33	0.67
Gain Ratio	0.764	0.184	0.834	0.764	0.76	0.601	0.853	0.807	5.87	0.78
Subset	0.748	0.198	0.823	0.748	0.742	0.574	0.803	0.768	4.79	0.19

Table B.4: K-Nearest Neighbour

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.794	0.165	0.841	0.794	0.792	0.636	0.814	0.763	0.03	361.81
Gain Ratio	0.811	0.152	0.851	0.811	0.811	0.664	0.826	0.797	0.02	149.69
Subset	0.815	0.147	0.857	0.815	0.814	0.673	0.818	0.799	0.03	137.89

Table B.5: Support Vector Machines

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.703	0.23	0.808	0.703	0.691	0.516	0.736	0.691	7694.00	278.26
Gain Ratio	0.75	0.197	0.821	0.75	0.746	0.575	0.777	0.726	1461.97	309.99
Subset	0.752	0.196	0.822	0.752	0.748	0.578	0.778	0.728	1544.53	312.14

Table B.6: Artificial Neural Networks

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.733	0.189	0.815	0.773	0.772	0.589	0.788	0.755	7909.46	9.49
Gain Ratio	0.753	0.193	0.827	0.753	0.748	0.584	0.838	0.803	337.77	0.97
Subset	0.751	0.193	0.83	0.751	0.745	0.584	0.878	0.857	281.82	0.56

Appendix C

Boosting Classification Results

Table C.1: Naive Bayes Results

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.737	0.214	0.799	0.737	0.734	0.54	0.822	0.803	24.96	1.79
Gain Ratio	0.731	0.208	0.822	0.731	0.723	0.557	0.764	0.719	4.54	0.37
Subset	0.737	0.205	0.82	0.737	0.731	0.562	0.768	0.721	3.02	0.20

Table C.2: J48 Decision Tree

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.779	0.175	0.838	0.779	0.776	0.619	0.936	0.914	298.09	0.26
Gain Ratio	0.778	0.177	0.832	0.778	0.776	0.613	0.904	0.87	68.95	0.15
Subset	0.772	0.181	0.831	0.772	0.769	0.605	0.907	0.874	45.07	0.14

Table C.3: Logistic Regression

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.744	0.214	0.793	0.744	0.742	0.54	0.931	0.925	289.92	0.60
Gain Ratio	0.753	0.192	0.829	0.753	0.748	0.585	0.864	0.827	37.63	0.35
Subset	0.748	0.198	0.823	0.748	0.742	0.574	0.855	0.818	24.43	0.20

Table C.4: K-Nearest Neighbour

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.783	0.173	0.836	0.783	0.781	0.621	0.837	0.787	16732	1800.00
Gain Ratio	0.809	0.154	0.849	0.809	0.808	0.659	0.859	0.813	13185	926.00
Subset	0.813	0.149	0.855	0.813	0.812	0.669	0.867	0.823	15091	1076.00

Table C.5: Support Vector Machines

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.705	0.228	0.809	0.705	0.693	0.519	0.763	0.716	59331	1780
Gain Ratio	0.748	0.199	0.819	0.748	0.743	0.57	0.79	0.743	33187	1074
Subset	0.749	0.199	0.819	0.749	0.744	0.572	0.79	0.74	41885	1274

Table C.6: Artificial Neural Networks

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.773	0.189	0.815	0.773	0.772	0.589	0.792	0.738	24314	3.21
Gain Ratio	0.753	0.193	0.827	0.753	0.748	0.584	0.847	0.811	3183	0.78
Subset	0.754	0.191	0.831	0.754	0.749	0.589	0.845	0.808	2758	0.75

Appendix D

Bagging Classification Results

Table D.1: Naive Bayes Results

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.763	0.196	0.81	0.763	0.761	0.575	0.91	0.887	8.61	2.99
Gain Ratio	0.734	0.206	0.823	0.734	0.727	0.562	0.878	0.843	2.89	0.63
Subset	0.738	0.204	0.82	0.738	0.732	0.563	0.891	0.871	1.97	0.47

Table D.2: J48 Decision Tree

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.837	0.13	0.87	0.837	0.837	0.707	0.907	0.9	250.77	0.38
Gain Ratio	0.759	0.19	0.826	0.759	0.755	0.588	0.847	0.827	51.01	0.18
Subset	0.759	0.19	0.825	0.759	0.755	0.588	0.912	0.888	41.42	0.11

Table D.3: Logistic Regression

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.756	0.205	0.8	0.756	0.755	0.558	0.816	0.722	173.26	0.74
Gain Ratio	0.756	0.19	0.83	0.756	0.751	0.589	0.844	0.805	49.16	0.28
Subset	0.75	0.197	0.821	0.75	0.745	0.575	0.833	0.796	39.14	0.26

Table D.4: K-Nearest Neighbour

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.783	0.173	0.836	0.783	0.781	0.621	0.858	0.837	0.77	6593.00
Gain Ratio	0.806	0.154	0.851	0.806	0.805	0.658	0.863	0.84	0.69	3714.00
Subset	0.781	0.172	0.84	0.781	0.779	0.624	0.871	0.848	0.90	3626.00

Table D.5: Support Vector Machines

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.702	0.23	0.808	0.702	0.69	0.516	0.738	0.692	68493	1605
Gain Ratio	0.749	0.198	0.82	0.749	0.745	0.573	0.777	0.727	10570	325
Subset	0.751	0.197	0.821	0.751	0.746	0.576	0.778	0.728	11729	355

Table D.6: Artificial Neural Networks

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.763	0.197	0.809	0.763	0.762	0.574	0.9	0.882	81253	23.19
Gain Ratio	0.739	0.202	0.824	0.739	0.732	0.567	0.863	0.833	3594	0.95
Subset	0.749	0.194	0.829	0.749	0.744	0.582	0.892	0.886	3011	0.89

Appendix E

Voting Classification Results

Table E.1: Base-Case Voting Results

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.759	0.19	0.826	0.759	0.755	0.588	0.945	0.944	20.99	7092.00
GainRatio	0.772	0.18	0.834	0.772	0.769	0.609	0.937	0.923	3.16	4241.44
SubSet	0.764	0.185	0.832	0.764	0.76	0.599	0.936	0.938	7.94	4447.93

Table E.2: Worst-Case Voting Results

Dataset	TP	FP	Prec	Recall	F-Score	MCC	ROC Area	PRC Area	Build Time	Test Time
Standard	0.722	0.181	0.83	0.772	0.769	0.605	0.882	0.841	4.32	536.38
GainRatio	0.757	0.191	0.826	0.757	0.753	0.587	0.919	0.887	2.19	413.84
SubSet	0.757	0.19	0.829	0.757	0.753	0.59	0.892	0.861	2.05	425.53