ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

ADDIS ABABA INSTITUTE OF TECHNOLOGY

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

K-Means Clustering and Random Forest Based Hybrid Intrusion Detection Algorithm

A Thesis Submitted to the School of Graduate Studies of Addis Ababa University In Partial Fulfillment of the Requirements For the Degree of Masters of Science In Computer Engineering

By

Meseret Kebede

Advisor: Mr Yonas Yehualashet

December 2017

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

"K-Means Clustering and Random Forest Based Hybrid Intrusion Detection Algorithm"

By

Meseret Kebede

ADDIS ABABA INSTITUTE OF TECHNOLOGY

APPROVAL BY BOARD OF EXAMINERS

Chairman Department of Graduate Committee	Signature
Advisor	Signature
Examiner	Signature
External Examiner	Signature

Abstract

The rapid growth of computers transformed the way in which information and data was stored and transmitted. With this new paradigm of data access, comes the threat of this information being exposed to unauthorized and unintended users. Because of this the integrity, confidentiality, and availability of data in a network become the most challenging issue. Many systems have been developed which scrutinize the data for deviation from the normal behavior or search for a known signature within the data. These systems are termed as Intrusion Detection Systems (IDS). IDSs employ different techniques varying from statistical methods to machine learning algorithms.

This paper evaluates the performance of different intrusion detection algorithms using KDD'99 dataset and explores if certain algorithms perform better for certain attack classes and consequently, if a multi-expert classifier design can deliver desired performance measure. The algorithms detection performance is compared by using Detection Rate (DR) and False Alarm Rate (FAR) evaluation metrics.

The experiment performed shows that those algorithms did in fact have different detection performance for different attack types and no single algorithm exceeds in detecting all attack types. Based on this evaluation results, best algorithms for each attack category is chosen and an optimized hybrid algorithm called K-Means Clustering and Random Forest Based Hybrid Intrusion Detection Algorithm (KRHA) is proposed.

The proposed algorithm classifies DoS, Probe, U2R and R2L attacks with 99.12%, 99.06 %, 89.79% and 78.63% accuracy respectively. This is an improvement from Fuzzy Logic which has high detection rate for probe with 98.51% and Random Forest for U2R with 85.6% and K-means clustering algorithm for R2L with 72.04% detection rate.

Keywords:

Intrusion Detection System, Data Mining, Machine Learning, Anomaly, Misuse, Clustering, Classification, KDD'99 Dataset, Hybrid, Detection Rate, False Alarm Rate

Acknowledgments

Above all, thanks to the almighty God for his help in giving me courage to cope up with complicated situations I have faced for pursuing the study and for his help and encouragement during my whole study time.

I am heartily grateful to my advisor Yonas Yehualashet for accepting me as his advisee and for providing me the opportunity to work on this research. I would like to thank him for his encouragement, guidance and unfailing support throughout the completion of my thesis. Also for his invaluable and unreserved assistance during the whole work of this study by taking time out of his busy schedule and providing his valuable expertise and guidance from the very start of designing the research proposal up to thesis write up and implementation of the algorithm. I am very much indebted to him for the amount of work he put into this task, which made the study to be completed successfully.

I am also thankful to the faculty members who were present during my two seminar presentation for their invaluable feedback and suggestions that allowed me to improve the content and presentation of my thesis. I also want to express my gratitude to all of the teachers who have contributed to my education over the course of my masters' study. Grateful acknowledgements are extended to Addis Ababa University Technology Faculty for offering the opportunity to pursue this study.

My thanks also go to my friends for cooperation and excellent facilitation during study period. My warm thanks are extended to my loving family for their unfailing encouragement and unwavering support in the course of the entire work and also during all the study time. Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of this paper.

Contents

List of Tables

List of Figures

List of Abbreviations

ART - Adaptive Resonance Theory

BBCNIDS - Behavioral Based Cyber Network Intrusion Detection System

BP- Back Propagation

CART - Classification and Regression Trees

CBA- Classification Based on Association

CCA-S - Clustering and Classification Algorithm Supervised

CHAID - Chi-square Automatic Interaction Detector

CPU - Computer Processing Unit

DoS - Denial of Service

DR - Detection Rate

FAR - False Alarm Rate

FLS - Fuzzy Logic System

FN - False Negative

FP - False Positive

HIDE- Hierarchical Intrusion Detection

HIDS - Host-based Intrusion Detection Systems

HTTP- Hypertext Transfer Protocol

ICMP - Internet Control Message Protocol

ID - Intrusion Detection

IDS - Intrusion Detection System

IP- Internet Protocol

ISPs - Internet Service Providers

KDD - Knowledge Discovery and Data Mining

KRHA- K-Means Clustering and Random Forest Based Hybrid Intrusion Detection Algorithm

MLP - Multilayer Perceptron

NIDS - Network-based Intrusion Detection Systems

NSL_KDD- Non-redundant Knowledge Discovery Data Mining

OOB - Out of the Bag

OSI - Open Systems Interconnection

PBH- Perceptron-Back propagation-Hybrid

R2L - Remote to Local

RBF- Radial Basis Function

SMO - Sequential Minimal Optimization

SOM- Self-Organizing Map

SVM - Support Vector Machines

TCP - Transmission Control Protocol

TN - True Negative

TP - True Positive

U2R - User to Root

UDP- User Datagram Protocol

WEKA - Waikato Environment for Knowledge Analysis

Chapter 1

Introduction

In the era of information society, computer networks and their related applications are becoming more and more popular, so does the potential threat to the global information infrastructure to increase. Heavy reliance on the internet and worldwide connectivity has greatly increased the potential damage that can be inflicted by remote attacks launched over the internet. Today the world of business computing is faced with the ever-increasing likelihood of unplanned downtime due to various attacks and security breaches.

Network downtime results in financial losses and more harm to the credibility of commercial enterprises especially Internet Service Providers (ISPs). Minimizing or possibly eliminating the unplanned downtime of the system establishes the continuity of the computing services. This can be done by identifying, prioritizing and defending against misuse, attacks and vulnerabilities. The challenge is to reduce the likelihood of catastrophic incidents by [1]: a) using appropriate machine and statistical learning techniques to assess the relative danger of individual threats and b) autonomously providing effective and appropriate response to the relevant threats.

To defend against various cyber attacks and computer viruses, several computer security techniques have been intensively studied in the last decade, namely cryptography, firewalls, anomaly and intrusion detection. Among them, intrusion detection system (IDS) has been considered to be one of the most promising methods for defending complex and dynamic intrusion behaviors. Intrusion detection [2] is the process of monitoring the events occurring in a computer system or network and analyzing them for signs of intrusions.

Majority of the IDS currently in use are either rule-based or expert-system based [2]. Their strengths depend largely on the ability of the security personnel that develops them. The former can only detect known attack types and the latter is prone to generation of false positive alarms. This leads to the use of intelligence techniques known as data mining and machine learning technique as an alternative to expensive and strenuous human input. These techniques

automatically learn from data or extract useful pattern from data as a reference for normal/attack traffic behavior profile from existing data for subsequent classification of network traffic.

Intrusion detection techniques using data mining and machine learning have attracted more and more interests in recent years. As an important application area of data mining, they aim to improve the great burden of analyzing huge volumes of data and realizing performance optimization of detection rules. Researchers propose different algorithms in different categories which can learn from the training data and can generalize when exposed to new untrained data. Some of them are: neural networks, which learn relationship between given input and output vectors and generalize them to extract new relationship between input and output [31,41,43], a fuzzy logic that generalizes relationship between input and output vector based on degree of membership [31,34], a decision tree which learns knowledge from a fixed collection of properties or attributes in a top down strategy from root node to leaf node [31,17,32] and support vector machine that simply creates Maximum-margin hyper planes during training with samples from two classes [40,41,42].

Rough sets which produce a set of compact rules made up of relevant features are only suitable for misuse and anomalous detection [42,44,45,46]. Bayesian approaches which are powerful tools for decision and reasoning under uncertain conditions employing probabilistic concept representations [47, 48]. Prior to the use of machine learning algorithms raw network traffic must first be summarized into connection records containing a number of within-connection features such as service, duration, and so on. As a result of these, the detection efficiencies are becoming better and better than ever before.

In this research, a set of selected intrusion detection algorithms are evaluated on KDDCup'99 data set. Their performance are measured based on their detection rate and false positive rate. There are four major attack categories found on KDDCup'99 datasets: Probe (information gathering), DoS (denial of service), U2R (user to root) and R2L (remote to local). These four attacks have distinct unique execution dynamics and signatures, which is one of the motivations for this research to investigate if certain detection algorithms are likely to demonstrate superior performance for a given attack category and if a hybrid classifier model can provide a higher performance measure than traditional single classifier algorithms.

1.1 Statement of Problem

With recent advances in network based technology and increased dependability of our everyday life on this technology, assuring the reliability and safety of the network is very important. Traditional intrusion detection and prevention techniques, like firewalls, access control mechanisms, and encryptions, have several limitations in fully protecting networks and systems from increasingly sophisticated attacks like denial of service. Moreover, most systems built based on such techniques suffer from high false positive and negative detection rates, and the lack of continuously adapting to changing malicious behaviors. In the past decade, however, several data mining and machine learning techniques have been applied to the problem of intrusion detection with the hope of improving detection rates and adaptability [3].

As network attacks have increased in number and severity over the past few years, intrusion detection system (IDS) is increasingly becoming a critical component to secure the network. Due to large volumes of security audit data as well as complex and dynamic properties of intrusion behaviors, optimizing performance of IDS has become an important open problem. Numerous intrusion detection techniques such as SMO, MLP and BayesNet are available but the main issue is their performance [2].

Different types of data mining and machine learning algorithms have been proposed in the hopes of improving detection rate with minimum false alarms. However, most of those existing intrusion detection techniques still have some short comings based on the detection approach they use [11]. Techniques which use signature based approach can only detect known attacks and if they use anomaly detection approach they are at risk for high false alarm rate. They also significantly differ in detection rate based on attack type, which means some algorithms detection rate for a specific kind of attack is higher than others.

So the purpose of this research will be to develop an algorithm with higher capability in detecting different attacks and a potential in lowering rate of false positive by increasing the accuracy of detecting novel attack.

1.2 Thesis Objectives

The general objective of this research is to perform a deep analysis on different algorithms used for intrusion detection and other cyber attack mitigation solutions, and find a room for

optimizing existing algorithms for the betterment of intrusion detection systems that are efficient in detecting cyber attacks that occur in computer networks. It also aims to test the effectiveness of the new proposed hybrid algorithm using existing data and method.

Specific Objectives

- To study different types of intrusion detection approaches
- To study the working principle of diverse types of algorithms
- To perform an analysis on five different kinds of intrusion detection algorithms. Since there are so many algorithms few are selected for this particular research such as Fuzzy logic, MARS, Random Forest classifiers and K-means clustering
- To propose and set up an improved hybrid model that is suited for detecting attacks efficiently
- To perform performance comparison evaluation on the single and multi-classifier algorithms

1.3 Motivation and Thesis Contribution

There are many intrusion detection techniques that are proposed by different researchers, some of which are reviewed in the literature section, and most of those techniques are of different type. Based on their detection approach they can be classified as anomaly detection technique, which monitors the system for any deviation from the normal behavior or misuse detection which identifies suspicious data by comparing it with the stored signature of known attacks. They can also be classified based on their learning method as supervised and unsupervised learning which uses labeled and unlabeled data respectively.

The motivation for this research comes from the fact that the four attack categories that are found in KDDCup'99 datasets have distinct unique execution dynamics and signatures. This fact leads us to raise two important questions. The first one is the uncertainty to explore if in fact certain algorithms perform better for a specific kind of attack types. And the second is if a multi classifier intrusion detection algorithm can achieve an improved performance in discovering most attack types compared to the ones already employed single algorithms. So looking the answer for these two questions is the inspiration to do this research.

We believe that building a hybrid classifier model that uses a combination of different techniques has better performance than the individual techniques. Since drawbacks of one technique could be complemented by the other.

This thesis makes a number of contributions to the field of intrusion detection. Some of them are listed as follows:

1 An overview on working principle of five distinct algorithms which are selected from a wide variety of most popular intrusion detection algorithms. Those algorithms represent a broad variety of fields including clustering approaches, decision trees, rule based models and function studying.
2 An experiment which compares the detection performance of those selected algorithms and identifies which algorithm performs better for which attack types and which algorithm has the most false alarms.
3 A newly designed hybrid intrusion detection model that combines the advantages of two different algorithms. One is a clustering type which uses unlabeled data and the other is a classification decision forest which uses labeled data. As a result we are able to improve detection performances.
4 A simple and informative description of the mechanism and data structures of the implemented algorithms, which will make it easier for other researchers to quickly grasp how those algorithms operate and to understand their differences.

1.4 Paper Organization

Here, we provide a detailed outline of the entire thesis by giving a brief summary of each chapter. The introductory chapters include Chapter 1 and Chapter 2 while the main body of work is found in Chapter 3, Chapter 4 and Chapter 5. Specifically, the remainder of this paper is organized as follows:

Chapter 1 gives a brief introduction to this research giving statement of the problem, thesis objectives, motivation and the main contributions of this work. This is followed by Chapter 2 which consist an overview of Intrusion Detection Systems including definition of usually used IDS terms and a brief summary on different detection approaches and learning models. This background chapter also provides a quick and up-to-date literature survey on the various data

mining and machine learning algorithms in an attempt to give a general idea on previous IDS works. Chapter 3 deeply discusses working principles of the five selected intrusion detection algorithms which belong to different techniques such as clustering, classification, rule and function. Chapter 4 explores the study done including evaluation setup, criteria and performance analysis for each of the selected algorithms. It also provides an overview on the dataset used. Chapter 5 covers the various steps carried out in developing the proposed hybrid intrusion detection model, its design and implementation. It also discusses the performance evaluation done on the hybrid model and the performance comparisons between hybrid algorithm and existing selected algorithms. Finally, Chapter 6 contains concluding remarks on our study, present ideas for improvements and recommendations for future research are forwarded.

Chapter 2

Background

2.1 Overview of Intrusion Detection System

Network attacks are defined as a set of malicious activities to disrupt, deny, degrade or destroy information and service resident in computer networks. A network attack is executed through the data stream on networks and aims to compromise the integrity, confidentiality or availability of computer network systems. Network attacks can vary from annoying email directed at an individual to intrusion attacks on sensitive data, computer information systems and critical network infrastructure. Examples of computer attacks include viruses attached to emails, probing of a system to collect information, internet worms, unauthorized usage of a system, and denial-of-service by abusing a feature of a system, or exploiting a bug in software to modify system data. Some general approaches that attackers can use to gain access to a system or limit the availability of that system include Social Engineering and Masquerading.

Many attack recognition systems have been developed and are in use widely which inspect network data for any variation from the ordinary action of a system or user of the system. Some might also look for an already recognized behavior of an attack within the data. These systems are termed as Intrusion Detection Systems (IDS) and employ different techniques varying from statistical methods to machine learning algorithms.

Before we go any further with this paper, we will define important and usually used terms related with IDS from authors in [3,4,5,6]:

Intruder: it can be any person, system or program that tries to or is successful to break into the network and perform illegal actions.

The intruders may be an entity from outside or may be an inside user of the system trying to access unauthorized information. Based upon this observation intruders can be widely divided into two categories; *external intruders* and *internal intruders*. [4]

- ✓ *External intruders* are those who don't have an authorized access to the system they are dealing with.

- ✓ *Internal intruders* are those who have limited authorized access to the systems and they overstep their legitimate access rights.

Internal users can be further divided into two categories; *masqueraders* and *clandestine users*.

- ✓ *Masqueraders* are those who use the identification and authorization of other legitimate users.
- ✓ *Clandestine users* are those who successfully evade audit and monitoring measures.

Intrusion: In the terms of information, intrusion can be defined as when a user of information tries to access such information for which he/she is not authorized. The person is called intruder and the process is called intrusion. In short, an intrusion is defined as the unauthorized use, misuse, or abuse of computer systems by either authorized users or external perpetrators. An intrusion normally exploits a specific vulnerability and must be detected as quickly as possible.

Intrusion Detection (ID): is the process of identifying and (possibly) responding to malicious activities targeted at computing and network resources by the observation of the information available about the state of the system and monitoring the user activities. Detection of break-ins or attempts by intruders to gain unauthorized access of the system is *intrusion detection.* Any hardware or software automation that monitors detects or responds to events occurring in a network or on a host computer is considered relevant to the intrusion detection approach.

Intrusion Detection System (IDS): An *intrusion detection system* or *IDS* is any hardware, software or combination of both that monitors, identifies and responds to malicious activity targeted at computing and networking resources [5]. An ID is often compared with a burglar alarm system. Just like a burglar alarm system monitors for any intrusion or malicious activity in a building facility, IDS keeps an eye on intruders in a computer or network of computers [6]. Intrusion detection systems are notable components in network security infrastructure. They examine system or network activity to find possible intrusions or attacks and trigger security alerts for the malicious activities.

They are generally categorized as signature-based (Misuse) and anomaly-based (Anomaly) detection system based on the techniques they employ [6]. Misuse detection systems find known attack signatures in the monitored resources and anomaly detection systems find attacks by

detecting changes in the pattern of utilization or behavior of the system. Other categories are host-based based and network intrusion detection systems based on origin of the data. The former operates on information collected from within an individual computer system and the latter collect raw networks packets as the data source from the network and analyze for signs of intrusions. Different IDSs provide varying functionalities and benefits.

2.1.1 Detection Approaches

The basic principle of intrusion detection is based on the assumption that intrusive activities are noticeably different from normal ones and thus are detectable [7]. Many intrusion detection approaches have been suggested in the literature. Traditionally these approaches are classified into the following categories: misuse detection, anomaly detection and specification-based detection. Anomaly based intrusion detection approaches are dedicated to establishing a model of the data flow that is monitored under normal conditions without the presence of any intrusive procedures. In contrast, misuse detection approaches aim to encode knowledge about patterns in the data flow that are known to correspond to intrusive procedures in form of specific signatures. In specification based detection approaches, security experts predefined the allowed system behaviors and thus events that do not match the specifications are labeled as attacks. But most effective intrusion detection techniques generally fall into one of the two categories: misuse detection and anomaly detection.

Misuse Detection

In misuse detection approach intrusions are detected by matching actual behavior recorded in audit trails with known suspicious patterns. While misuse detection is fully effective in uncovering known attacks, it is useless when faced with unknown or novel forms of attacks for which the signatures are not yet available. Moreover, for known attacks, defining a signature that encompasses all possible variations of the attack is difficult. Any mistakes in the definition of these signatures will increase the false alarm rate and decrease the effectiveness of the detection technique.

Figure 2.1 illustrates a typical misuse detection model. The model consists of four components: namely, data collection, system profile, misuse detection and response [1]. Data are collected from one or many data sources including audit trails, network traffic, system call trace, etc.

Collected data are transferred into a format that is understandable by the other components of the system. The system profile is used to characterize normal and abnormal behaviors. The profiles characterize what a normal subject behavior should be and what operations the subjects typically would perform or do not perform on the objects. The profiles are matched with actual system activities and reported as intrusions in case of deviations. The system profile is prepared by a domain expert or expert system by using different classes of techniques which are commonly used to implement misuse detection, namely pattern matching, rule-based techniques, state-based techniques, and data mining.

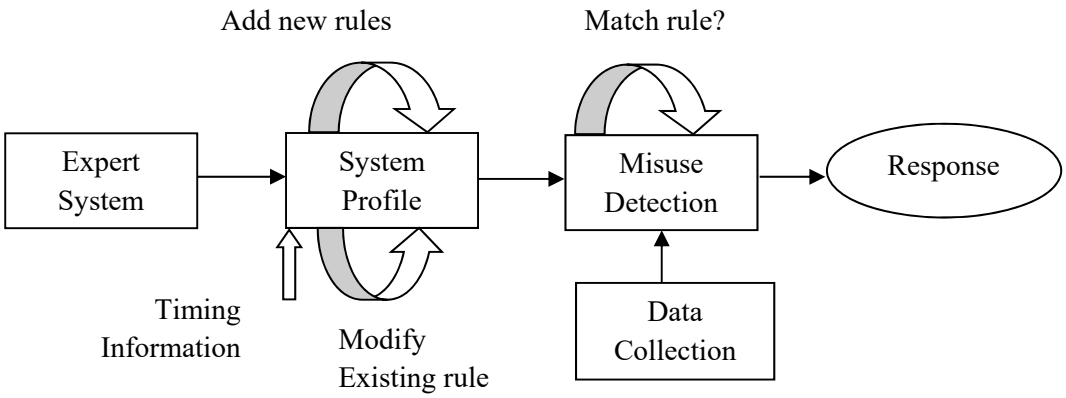

Figure 2.1: A typical misuse detection system

Anomaly Detection

Different from misuse detection, anomaly detection is dedicated to establishing normal activity profiles for the system. It is based on the assumption that all intrusive activities are necessarily anomalous. Anomaly detection studies start by forming an opinion on what the normal attributes for the observed objects are, and then decide what kinds of activities should be flagged as intrusions and how to make such particular decisions.

A typical anomaly detection model is illustrated in Figure 2.2. It consists of four components, namely data collection, normal system profile, anomaly detection and response. Normal user activities or traffic data are obtained and saved by the data collection component. The accepted system profiles is prepared by the specifications of the network administrators who manually provide specifications of the correct behavior or other modeling techniques including data mining, neural networks and model based approaches. The anomaly detection component

decides how far the current activities deviate from the normal system profiles and what percentage of these activities should be flagged as abnormal. Finally, the response component reports the intrusion and possibly corresponding timing information. The primary advantage of anomaly detection is its capability to find novel attacks; as such it addresses the biggest limitation of misuse detection.

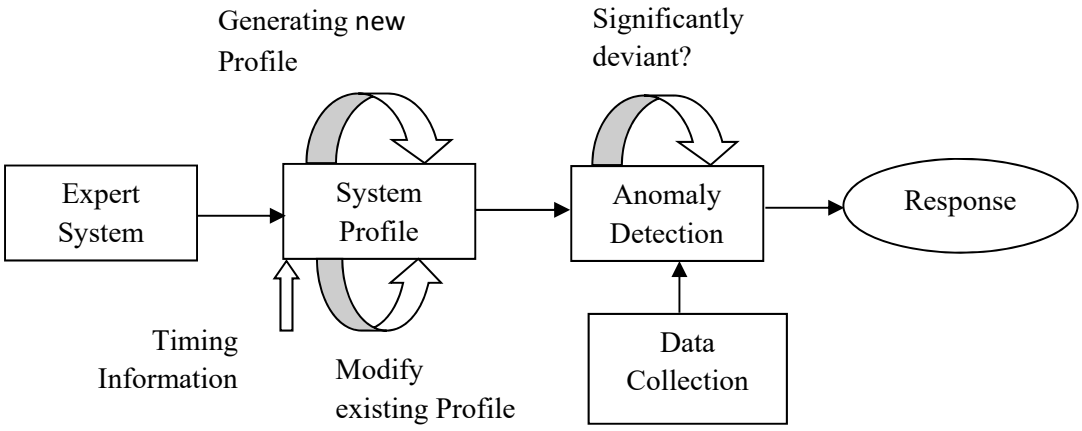

Figure 2.2: A typical anomaly detection system

However, due to the assumptions underlying anomaly detection mechanisms, their false alarm rates are in general very high. Specifically, the main reasons for this limitation include the following:

1. The user's normal behavior model is based on data collected over a period of normal operations; intrusive activities missed during this period are likely to be considered as normal behaviors.

2. Anomaly detection techniques can hardly detect stealthy attacks because these kinds of attacks are usually hidden in large number of instances of normal behaviors. Moreover, the types of parameters used as inputs of normal models are usually decided by security experts. Any mistake occurring during the process of defining these parameters will increase the false alarm rate and decrease the effectiveness of the anomaly detection system. As a result, the design of the detection methods and the selection of the system or network features to be monitored are two of the main open issues in anomaly detection.

2.1.2 Learning Models

Learning models incorporate learning capabilities in intrusion detection process, using artificial learning techniques. In recent years, learning techniques have been widely used in intrusion detection since the self-learning techniques can automatically form an opinion of what the subject's normal behavior is [1]. According to whether they are based on supervised or unsupervised learning techniques intrusion detection schemes are divided into two main categories: supervised (also known as predictive or directed) and unsupervised (also known as descriptive or undirected). Both categories encompass functions capable of finding different hidden patterns in large data sets. They differ in the types of training data available to the learner, the order and method by which training data is received and the test data used to evaluate the learning algorithm [4].

Supervised Learning

Supervised learning is the task of inferring a function from labeled training data. The training data consist of a set of *training examples*. The learner receives those sets of labeled examples as training data and makes predictions for all unseen points. In supervised learning, each example is a *pair* consisting of an input object (typically a vector) and a desired output value (also called the *supervisory signal*). A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples. An optimal scenario will allow for the algorithm to correctly determine the class labels for unseen instances. This requires the learning algorithm to generalize from the training data to unseen situations in a "reasonable" way.

Supervised learning problems are commonly associated with **"regression"** and **"classification"** problems. In a **regression** problem, we are trying to predict results within a **continuous** output, meaning that we are trying to map input variables to some **continuous** function. In a **classification** problem, we are instead trying to predict results in a **discrete** output. In other words, we are trying to map input variables into **discrete** categories.

Supervised learning is the most common technique for training neural networks and decision trees. Both of these techniques are highly dependent on the information given by the pre-determined classifications. In the case of neural networks, the classification is used to determine the error of the network and then adjust the network to minimize it, and in decision trees, the

classifications are used to determine what attributes provide the most information that can be used to solve the classification puzzle.

Unsupervised Learning

Unsupervised learning is the task of inferring a function to describe hidden structure from unlabeled data. The learner exclusively receives unlabeled training data, and makes predictions for all unseen points. Unsupervised learning allows us to approach problems with little or no idea what our results should look like. We can derive structure from data where we don't necessarily know the effect of the variables. We can derive this structure by clustering the data based on relationships among the variables in the data. With unsupervised learning there is no feedback based on the prediction results.

Clustering and dimensionality reduction are example of unsupervised learning problems. And among neural network models, the self-organizing map (SOM) and adaptive resonance theory (ART) are commonly used unsupervised learning algorithms.

The main drawback of supervised anomaly detection is the need of labeling the training data, which makes the process error-prone, costly and time consuming, and difficult to find new attacks. Unsupervised anomaly detection addresses these issues by allowing training based on unlabeled data sets and thus facilitating online learning and improving detection accuracy. Unsupervised anomaly detection is relatively new compared with supervised anomaly detection schemes [3].

2.1.3 Host based vs. Network based Intrusion Detection

Intrusion Detection Systems are security systems that collect information from various types of system and network sources, and analyze these data in an attempt to detect activity that may constitute an attack or intrusion on the system. Collection of data is one of the most important steps when designing an Intrusion Detection System (IDS) and it influences the whole design and implementation process. Usually, the attacks target not only one individual computer but also aim for a group of hosts. As a result, some intrusions might show an anomalous behavior at the network layer, while others could exhibit anomalous behaviors at the application layer. In order to cover various network intrusions we need to monitor each layer on networks. Although ideally it is possible to design and implement an IDS that can inspect a wide range of data

extracted from both network and application layer, it is infeasible in practical due to two main reasons: one is the diversity of the data, and the other one is the time and space resources that the system has to consume for collecting and interpreting the data [2].

Intrusion detection systems collect data from many different sources, such as system log files, network packets or flows, system calls and a running code itself. The place where the data are collected decides the location and scope of IDSs. Depending upon the origin of data source or location in a network, intrusion detection can also be classified into two categories: *host based and network based IDS.*

Network-based Intrusion Detection Systems (NIDS): Network Intrusion Detection Systems are placed at a strategic point or points within the network to monitor passing network traffic for signs of intrusion to and from all devices on the system. Ideally one would scan all inbound and outbound traffic. Relying on the location where the data is collected, a NIDS captures and analyzes the traffic within a subnet, a network, or between a network and the Internet. NIDS also analyzes network traffic at all layers of the Open Systems Interconnection (OSI) model and makes decisions about the purpose of the traffic by examining its contents and types.

Traditionally, NIDSs ignore the payload of the packets due to the computational time, the vast diversity of protocols, and sometimes privacy issues [1]. Most NIDSs are easy to deploy on a network and can often view traffic from many systems at once.

Some of the advantages of NIDS are:

- Running network intrusion detection monitors does not degrade the performance of other programs running over the network. This low performance cost is due to the fact that the monitors only read each packet as they come across its network segment.
- They are independent of the operating systems that they are installed on. Deployed network-based intrusion detection sensors will listen for all attacks, regardless of the destination operating system type.
- Network-based systems are portable. They can be inserted easily on part of a network and data can be collected with minimal work. This is beneficial in situations where network topology changes or where system resources have been moved.

However, network-based solutions have their share of problems also. One of the issues with network-based intrusion detection approaches is scalability. Network monitors must inspect every packet that is passed through the segment they are placed on. Due to this NIDSs sometimes have difficulty keeping up with systems especial systems with heavy traffic.

Encryption and switching represent two further limitations of network-based approaches. First, if network traffic is encrypted, an agent cannot scan the protocols or the content of these packets [7]. Second, the nature of switches makes network monitoring extremely difficult. In the case of switched networks, the network switch acts to isolate network connections between hosts so that a host can only see the traffic that is addressed to it. In these cases, a network-based monitor is essentially reduced to monitoring a single host, defeating much of the intent of the monitor. Some switches can now support a port for monitoring and scanning, which offers a partial solution to this problem [8].

Host-based Intrusion Detection Systems (HIDS): Host-based Intrusion Detection Systems run on individual hosts or devices on the network and look at user and process activity for signs of malicious behavior. A HIDS monitors the inbound and outbound packets from a device only and will alert the user or administrator if suspicious activity is detected. HIDS analyze network traffic and system-specific settings such as software calls, local security policy, local log audits, and more. A HIDS is usually software running on the protected host, and thus, the coverage of a HIDS is restricted to only one machine. As a result, to protect the entire enterprise network, a HIDS must be installed on each individual machine and requires configuration specific to that operating system. .

Some of the advantages of this type of IDS are:

- They can monitor all users' activities which is not possible in a network based system
- They are capable of identifying attacks that originate from inside the host.
- A host based system can analyze the decrypted traffic to find attack signature-thus giving them the ability to monitor encrypted traffic.
- They do not require any extra hardware since they can be installed in the existing host servers.

- Host-based sensors are also useful in that they can keep track of the behavior of individual users. This can help catch attacks while they are happening or possibly stop a potential attack before it affects the system.
- They are cost effective for a small scale network having a few hosts.

The main disadvantages of this system are they can be compromised as soon as the host server is compromised by an attack. In addition, they take up significant storage space as well as extra computing power from the host where they reside. They can be ineffective during the denial of service attacks.

Other drawback of host-based intrusion detection systems is its chronic problem of portability. The sensors are host-based, so they have to be compatible with the platform they are running over [6]. This lack of cross-platform support would represent a major obstacle for a corporation wishing to employ a host-based solution. Accordingly, in very large environments, a host-based approach could be economically infeasible.

2.2 Related Works

The literature survey reveals many results, the authors in [8], proposed a real-time intrusion detection system based on the Self-Organizing Map (SOM); an unsupervised learning technique that is appropriate for anomaly detection in wireless sensor networks. The proposed system was tested using KDD'99 intrusion detection evaluation dataset. The system groups similar connections together based on correlations between features. A connection may be classified as normal or attack. Attacks are classified again based on the type of attack. It took the system 0.5 seconds to decide whether a given input represents a normal behavior or an attack. In [9], Nong and Xiangyang developed a data mining algorithm called Clustering and Classification Algorithm Supervised (CCA-S) for intrusion detection in computer networks. The algorithm is used to learn signature patterns of both normal behaviors and attacks. Compared to anomaly detection techniques, signature recognition techniques always produce true alarms, but cannot detect unknown attacks. The algorithm's scalable and incremental learning results in a better performance than two other decision tree algorithms: CHAID and CART.

In [10], Jiong Zhang et.al addressed the main drawback of detecting intrusions by means of anomaly (outliers) detection, which is the high rate of false alarms when a behavior that has never been seen before is presented. In their work, they added a new feature to the unknown behaviors before they are considered as attacks, and they claim that the proposed system guarantees a very low ratio of false alarms, making unsupervised clustering for intrusion detection more effective, realistic and feasible. In [11], the authors propose machine learning algorithms such as Random Forest and AdaBoost, along with Naïve Bayes, to build an efficient intrusion detection model. They also report the experimental results over KDDCup'1999 datasets. When applied to the data set, their developed algorithms for learning classifiers were successful in detecting network attacks than standard data mining techniques based on neural networks. They also suggest that the choice of any data mining algorithm for intrusion detection purpose should be a compromise among the time taken to build the model, detection rate and low false alarm rate of the algorithm.

In [12], authors compared decision tree, naive bayes and the NBTree (hybrid between decision trees and naïve bayes) for classifying traffics to either normal or attack by using a standard dataset on open source tool. The hybrid algorithm (NBTree) had better predictive power with high accuracy and less error rate than using each algorithm alone but it needs more construction and processing time. In [13], K.Nageswara Rao et.al addressed the complexity of the intrusion detection datasets, as most of them are complex and contain large number of attributes. Some of these attributes may be redundant or do not have significant contribution for intrusion detection.

In [14], Alemtsehay proposed a Behavioral Based Cyber Intrusion Detection System (BBCNIDS). BBCNIDS used two data mining techniques; decision tree and association rule mining. The model first identifies attacks and normal dataset using decision tree algorithm, and then collects the so called normal data output. The normal output data was fed into association rule mining (apriori algorithm) as the testing data. Association rule mining identifies further classification into attack and normal. Decision tree algorithm almost classifies known attacks; whose signatures are already known and stored in database otherwise it considers the new attack as normal. On the other hand, association rule mining detects new attacks, those which have not yet stored signatures and considers some normal data as attack. The association rule mining works by learning on the current normal behavior and when there is a deviation from the normal behavior it considers as an attack. They used a reduced and enhanced non-redundant NSL_KDD dataset for training and testing. Evaluation results show that the proposed model provides improved detection rate and accuracy. In terms of detection rate, hybrid model was better than both decision tree and association rule mining. But in the case of accuracy, hybrid model is slightly less than decision tree and far better than association rule mining.

In [15], the authors introduced the framework of HIDE, a hierarchical anomaly network intrusion detection system using statistical preprocessing and neural network classification. They tested five different types of neural network classifiers: Perceptron, Back Propagation (BP), Perceptron-Back propagation-Hybrid (PBH), Fuzzy ARTMAP also known as Predictive ART, and Radial-based Function. The results showed that BP and PBH nets outperform Perceptron, Fuzzy ARTMAP and RBF. Thus, classification capabilities of BP and PBH are more desirable for statistical anomaly intrusion detection systems. They also stress-tested the entire system, which showed that HIDE can reliably detect UDP flooding attacks with attack intensity as low as

five to ten percent of background traffic. In [16], the author discusses statistical-based methods that are used for anomaly detection, where active users are compared to historical profiles. Users are identified as normal, if they closely matched to their historical profiles and are considered as intruder if they don't match. The author proposes using ART-2 net for clustering users through command profiles. The result shows that the proposed model largely improved the prediction rate.

In [17], Quinlan et.al discussed about C4.5, which is one of decision tree generators using top-down approach. It uses the information gain ratio as the splitting criterion for each internal node. Like other similar top-down approaches, C4.5 uses a greedy searching strategy with looking one step ahead to find the way of splitting instance space, so it often suffers from being trapped at local optimum and therefore performs poorly in dealing with some hard classification tasks, in which training data are described by high dimensional attribute vectors and the concept to be learned is complex. There have been several proposed non-greedy search approaches, which choose a splitting model automatically by observing the structure of data so that splitting models are different from nodes to nodes. Construction of optimal or near-optimal decision trees using a two stage approach has been attempted by many authors. In the first stage, a sufficient partitioning is induced using any reasonable greedy method. In the second stage, the tree is refined to be as close to optimal as possible. Another method which has been suggested by other researchers for generating non-greedy decision tree is to use genetic method.

Many current intrusion detection systems are constructed by manual encoding of an expert knowledge, changes to them are expensive and slow. In [18], the authors propose a data mining based new ensemble boosted decision tree approach for intrusion detection system. Experimental result shows better performance for detecting intrusions as compared to other existing methods. Author in [19] discusses the differences in host and network-based intrusion detection techniques to demonstrate how the two can work together to provide additionally effective intrusion detection and protection. He also proposes a hybrid system which combines both network and host based IDS, with anomaly and misuse detection mode. The model utilizes auditing programs to extract an extensive set of features that describe each network connection or host session, and applies data mining programs to learn rules that accurately capture the behavior of intrusions and

normal. The performance evaluation result illustrates the proposed hybrid IDS has a higher intrusion detection performance than a single host and network based IDS.

In [20], authors propose a novel hybrid model for misuse and anomaly detection. C4.5 based binary decision trees are used for misuse and CBA based classifier is used for anomaly detection. Firstly, the C4.5 based decision tree separates the network traffic into normal and attack categories. The normal traffic is sent to anomaly detector and parallel attacks are sent to a decision trees based classifier for labeling with specific attack type. The CBA based anomaly detection is a single level classifier where as the decision tree based misuse detector is a sequential multilevel classifier which labels one attack at a time in a step by step manner. Results show that the overall accuracy of the proposed model is 97.24%, which is a better performance compared to the single C4.5 and CBA based classifier with accuracy of 92.59% and 95.31% respectively. To overcome the deficiencies in KDD'99 dataset, a new improved dataset is also proposed. The overall accuracy of integrated model trained on new dataset is 97.495% compared to 97.24% of the old dataset.

The set of data mining and machine learning algorithms applied in the literature constitutes a very small subset of what is potentially applicable for the intrusion detection problem. Additionally, reported results suggest that much detection performance improvement is possible through combination of different techniques or by finding a way to optimize existing algorithms themselves. In light of the widely-held belief that attack execution dynamics and signatures show substantial variation from one attack category to another, identifying attack category specific detection algorithms offers a promising research direction for improved intrusion detection performance. But from the literature review we can see that there are not many researches done on identifying attack specific algorithms and on improving the detection performance by using multi classifier algorithms. Most of the previous works only try to improve the performance of a single algorithm. In this paper, a wide-ranging set of intrusion detection algorithms are evaluated on the KDD'99 data set, which is one of the few public domain data which utilizes TCP/IP level information and is embedded with domain-specific heuristics, to detect intrusions at the network level. The evaluation is done in order to identify which algorithms perform better for which specific attack category. And based on the evaluation result a hybrid intrusion detection model is proposed in the hopes for improving detection performance for multiple attack type.

Chapter 3

Intrusion Detection Algorithms

A significant problem of Intrusion Detection System is that how to efficiently identify the normal and abnormal behaviors from a huge number of raw information's attributes. And also how to effectively generate automatic intrusion rules following composed raw data of the network. In order to overcome such problems, different data mining (also known as Knowledge Discovery in Databases) and machine learning techniques have been studied such as: classification, clustering, association etc. in order to dissect the information. In this chapter, five most popular and widely used algorithms which belong to those techniques are discussed.

3.1 K-Means Clustering Algorithm

Clustering is the process of partitioning or grouping a given set of patterns into disjoint clusters. It is the most open-ended data-mining technique which finds and groups data points with natural similarities. This is done such that patterns in the same cluster are alike and patterns belonging to two different clusters are different. Clustering is used when there are no obvious natural groupings, in which case the data may be difficult to explore. Clustering the data can reveal groups and categories we were previously unaware of. These new groups may be fit for further data mining operations from which we may discover new correlations.

The K-means [21] algorithm is a distance-based clustering algorithm that partitions or groups the data into a predetermined number of clusters (provided there are enough distinct cases). Distance-based algorithms rely on a distance metric (function) to measure the similarity between data points based on attributes/features into K number of group. K is a positive integer number. The distance metric is either Euclidean, Cosine, or Fast Cosine distance. Data points are assigned to the nearest cluster according to the distance metric used. The K-means [22] algorithm can be interpreted as a mixture model where the mixture components are spherical multivariate normal distributions with the same variance for all components. Thus, the purpose of K-mean clustering is to classify the data.

The pseudo code of k-means clustering is written below [23]:

(1) Define the number of clusters K,

(2) Place randomly the initial K points into the space represented by the objects that are being clustered. These points represent initial group centroids.

(3) Assign each object to the group that has the closest centroid.

(4) When all objects have been assigned, recalculate the positions of the K centroids. (The values of attributes in the cluster / number of records in the cluster).

(5) Repeat Steps 2 and 3 until the centroids no longer move. This produces a separation of the objects into groups from which the metric to be minimized can be calculated.

Details of K-means Clustering Algorithm

Step 1: Determine the initial centroids: the first step in building clusters is setting up the initial group of centroids which represent the objects that are being clustered. Those centroids are instances found in the training data and are obtained using the random selection method. The numbers of centroids selected are same as the number of clusters.

Step2: Determine the distance of each object to the centroids: The next step taken was calculating the distance of each instance with the given initial centriods using a distance metric and assign them to the group that has the closest centroid according to this function. In the case of this hybrid model a Euclidean distance metric was used.

Step 3: Repeat Step 2 using new centroids until the centroids no longer move: After all instances have been assigned to a cluster based on the initial centriods, a new set of centroids were computed for each cluster using these latest instance memberships. Then the distance estimation was performed all over again until centroids could no longer move. This is known by comparing the grouping of last iteration with the previous iteration and if it reveals that no instance move from their group anymore, it means the computation of the k-mean clustering has reached its stability and no more iteration is needed. We get the final grouping as the results.

Data Preparation for K-means

Automatic Data Preparation performs outlier-sensitive normalization for K-means. When there are missing values in columns with simple data types (not nested), K-means [24] interprets them as missing at random. The algorithm replaces missing categorical values with the mode and

missing numerical values with the mean. When there are missing values in nested columns, K-means interprets them as sparse. The algorithm replaces sparse numerical data with zeros and sparse categorical data with zero vectors. One should keep in mind [24], while doing data preparation, those outliers with equi-width binning can prevent K-means from creating clusters that are different in content. The clusters may have very similar centroids, histograms, and rules.

A key limitation of K-means is its cluster model. It has problems when clusters are of differing sizes and densities. Since the K-means algorithm groups data points purely on their (Euclidean) *geometric closeness* to the cluster centroid, the clusters are expected to be of similar size so that the assignment to the nearest cluster center is the correct assignment. And because K-means implicitly assumes each cluster occupies the same volume in data space, it also does not take into account the different *densities* of each cluster. Thus, K-means is quite inflexible and degrades badly when the assumptions upon which it is based are violated by e.g. a tiny number of outliers. It works well on some data sets, and fails on others.

3.2 Decision Tree Algorithm

Decision Tree is a simple yet widely used classification technique. Classification problem can be solved by asking a series of carefully crafted questions about the attribute of the test record. Each time we receive an answer, a follow-up question is asked until we reach a conclusion about the class label of the record. The series of questions and their possible answers can be organized in the form of decision tree, which is a hierarchical structure consisting of nodes and directed edges. The tree has three types of nodes [29]:

- ✓ **A root node** has no incoming edges and zero or more outgoing edges.
- ✓ **Internal nodes**, each of which has exactly one incoming edge and two or more outgoing edges.
- ✓ **Leaf** or **terminal nodes**, each of which has exactly one incoming edge and no outgoing edges.

A decision tree is a tree in which each branch node represents a choice between a number of alternatives, and each leaf node represents a decision. So they are commonly used for gaining information for the purpose of decision making. Decision tree starts with a root node on which it is for users to take actions. From this node, users split each node recursively according to

decision tree learning algorithm. The final result is a decision tree in which each branch represents a possible scenario of decision and its outcome.

In decision tree, each leaf node is assigned to a class label. The **non-terminal** nodes, which include the root and other internal nodes, contain attribute test conditions to separate records that have different characteristics.

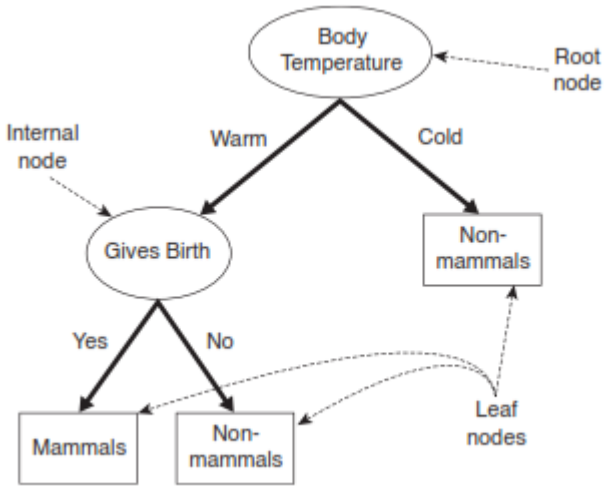

Figure 3.1: An example of a decision tree for the mammal classification problem

For example, the root node shown in Figure 3.1 uses the attribute Body Temperature to separate warm-blooded from cold-blooded vertebrates. Since all cold-blooded vertebrates are non-mammals, a leaf node labeled Non-mammals is created as the right child of the root node. If the vertebrate is warm-blooded, a subsequent attribute, Gives Birth is used to distinguish mammals from other warm-blooded creatures, which are mostly birds.

When we come to intrusion detection system, classifying a test record is also straightforward like the above example once a decision tree has been constructed. Starting from the root node, we apply the test condition to the record and follow the appropriate branch based on the outcome of the test. This will lead us either to another internal node, for which a new test condition is applied, or to a leaf node. The class label associated with the leaf node is then assigned to the record.

How to build a Decision Tree

In principle, there are exponentially many decision trees that can be constructed from a given set of attributes. While some of the trees are more accurate than others, finding the optimal tree is computationally infeasible because of the exponential size of the search space. Nevertheless, efficient algorithms have been developed to include a reasonably accurate, though suboptimal, decision tree in a reasonable amount of time. These algorithms usually employ a greedy strategy that grows a decision tree by making a series of locally optimum decisions about which attribute to use for partitioning the data. One such algorithm is **Hunt's algorithm** [29], which is the basis of many existing decision tree induction algorithms, including ID3, J48, C4.5, and CART.

Hunts Algorithm

In Hunt's algorithm, a decision tree is grown in a recursive fashion by partitioning the training records into successively purer subsets. Let D_t be the set of training records that are associated with node t and $y=\{y_1, y_2, \ldots\ldots y_c\}$ be class labels. The following is a recursive definition of Hunt's algorithm.

Step 1: If all the records in D_t belong to the same class y_t, then t is a leaf node labeled as y_t.

Step 2: If D_t contains records that belong to more than one class, an **attribute test condition** is selected to partition the records into smaller subsets. A child node is each outcome of the test condition and the records in D_t are distributed to the children based on the outcomes. The algorithm is recursively applied to each child node.

Decision Tree Algorithms

Let D_t be the set of training records that are associated with node t and c be the class labels. Create a root node for the tree

- If all the records in D_t belong to the same class c, return a new leaf node and label it with *c*
- If number of predicting attributes is empty, then return the single node tree Root, with label = most common value of the target attribute in the examples.
- Otherwise:
    - Let A be the attribute from attributes that best classifies training sets in D_t.
    - Assign t the decision attribute A.

- For each possible value "a" in A do:
  - Add a new tree branch below t, corresponding to the test A = "a".
  - Let D_t _ a be the subset of D that has value "a" for A.
  - If D_t _ a is empty:
    - Then add a leaf node with label of the most common value of target in D.
    - Else add the subtree ID3 (D_t _ a, Attributes n {A}, Target).
- Return t.

The main ideas behind the algorithm are:

- Each non-leaf node of a decision tree corresponds to an input attribute, and each arc (edge) to a possible value of that attribute. A leaf node corresponds to the expected value of the output attribute when the input attributes are described by the path from the root node to that leaf node.
- In a "good" decision tree, each non-leaf node should correspond to an input attribute which is the most informative about the output attribute when compared with other input attributes. Those other input attributes are attributes which are not yet considered in the path from the root node to that node. Using "best" split attribute at each node will helps us to predict the output attribute using the smallest possible number of questions on average.
- Entropy is used to determine how informative a particular input attribute is about the output attribute for a subset of the training data. Entropy is a measure of uncertainty in communication systems introduced by Shannon (1948). It is fundamental in modern information theory [29].

Measures for Selecting the Best Split

The main goal behind decision tree is to grow a simple tree with good attributes. A good attribute prefers attributes that split the data so that each successor node is as *pure* as possible i.e., the distribution of examples in each node is so that it mostly contains examples of a single class. In other words, we want a measure that prefers attributes that have a high degree of "order":

- ✓ Maximum order: All examples are of the same class
- ✓ Minimum order: All classes are equally likely

There are many measures that can be used to determine the best way to split the records. These measures are defined in terms of the class distribution of the records before and after splitting.

The measures developed for selecting the best split are often based on the degree of impurity of the child nodes. The smaller the degree of impurity, the more skewed the class distribution. For example, a node with class distribution (0, 1) has zero impurity, whereas a node with uniform class distribution (0.5, 0.5) has the highest impurity. One of the widely used impurity measures is called Entropy. **Entropy** is a measure for (un-)orderedness. It is the amount of information that is contained.

$$\text{Entropy (t)} = -\sum_{i=0}^{c-1} p(i/t) \log_2 p(i/t) \quad (1)$$

Where c is the number of classes and 0 $\log_2$ 0= 0 in entropy calculations and P(i/t) denote the fraction of records belonging to class *i* at a given node *t.*

To determine how well a test condition performs, we need to compare the degree of impurity of the parent node (before splitting) with the degree of impurity of the child nodes (after splitting). The larger the difference, the better the test condition will be. The gain,Δ, is a criterion that can be used to determine the goodness of a split. We define information gain as the expected reduction of entropy related to specified attribute when splitting a decision tree node [30].

$$\Delta = I \text{ (parent)} - \sum_{j=1}^{k} \frac{N(v_j)}{N} I(v_j), \quad (2)$$

Where $\boldsymbol{I}$ is the impurity measure of a given node, $\boldsymbol{N}$ is the total number of records at the parent node, $\boldsymbol{k}$ is the number of attribute values, and $\boldsymbol{N(v_j)}$ is the number of records associated with the child node $\boldsymbol{v_j}$. Decision tree induction algorithms often choose a test condition that maximizes the gain Δ. Since *I* (parent) is the same for all test conditions, maximizing the gain is equivalent to minimizing the weighted average impurity measures of the child nodes. Finally, when entropy is used as the impurity measure, the difference in entropy is known as the **information gain**, Δ_{info}.

We can use this notion of gain to rank attributes and to build decision trees where at each node is located the attribute with greatest gain among the attributes not yet considered in the path from the root.

The intention of this ordering is to create small decision tree so that records can be identified after only a few decision tree splitting and to minimize the process of decision making.

3.3 Random Forest Algorithm

Decision Tree algorithms are effective [25] in that they provide human-readable rules of classification. Beside this it has some drawbacks, one of which is the sorting of all numerical attributes when the tree decides to split a node. Such split on sorting all numerical attributes becomes costly i.e. efficiency or running time and memory size, especially if Decision Trees are set on data the size of which is large i.e. it has more number of instances. In 2001, Breiman [26] presented the idea of Random Forests which perform well as compared to other classifiers including Support Vector Machines, Neural Networks and Discriminant Analysis, and overcomes the over fitting problem.

Significant improvements in classification accuracy have resulted from growing an ensemble of trees and letting them vote for the most popular class. In order to grow these ensembles, often random vectors are generated that govern the growth of each tree in the ensemble. An early example is bagging (Breiman [1996]), where to grow each tree a random selection (without replacement), is made from the examples in the training set.

The essential idea in bagging is to average many noisy but approximately unbiased models, and hence reduce the variance of an estimated prediction function. Trees are ideal candidates for bagging, since they can capture complex interaction structures in the data, and if grown sufficiently deep, have relatively low bias. Given that trees are notoriously noisy, they benefit greatly from the averaging. Moreover, since each tree generated in bagging is identically distributed (i.d.), the expectation of an average of B such trees is the same as the expectation of any one of them. This means the bias of bagged trees is the same as that of the individual trees, and the only hope of improvement is through variance reduction.

Those methods such as Bagging or Random subspaces [27, 28] which are made from ensemble of various classifiers and those which use randomization for producing diversity have proven to be very efficient. In order to introduce diversity and to build classifiers different from each other, they use randomization in the induction process. Random Forest is a substantial modification of bagging that builds a large collection of de-correlated trees, and then averages them. As a

consequence, Random Forests have gained a substantial interest in different fields because of its efficient discriminative classification.

Random Forest as defined in [26] is a generic principle of classifier combination that uses L tree-structured base classifiers $\{h(X,\Theta_n), N=1,2,3,\ldots L\}$, where for the n^{th} tree, a random vector Θ_n which is a family of identical and dependent distributed random vector is generated, independent of the past random vectors $\Theta_1, \ldots, \Theta_{n-1}$ but with the same distribution; and a tree is grown using the training set and Θn , resulting in a classifier $h(X,\Theta n)$, where X is an input vector.

Every Decision Tree is made by randomly selecting the data from the available data. For example, a Random Forest for each Decision Tree (as in Random Subspaces) can be built by randomly sampling a feature subset, and/or by the random sampling of a training data subset for each Decision Tree (the concept of Bagging). After a large number of trees are generated, they vote for the most popular class. We call these procedures **Random Forests.**

Advantages

Random Forest not only keeps the benefits achieved by the Decision Trees, but most of the time it achieves better results. This is due to the use of bagging on samples, a random selection on subsets of variables and its voting scheme through which decision is made [33].

The Random Forest is appropriate for high dimensional data modeling because it can handle missing values and can handle continuous, categorical and binary data. The bootstrapping and ensemble scheme makes Random Forest strong enough to overcome the problems of over fitting and hence there is no need to prune the trees. Besides high prediction accuracy, Random Forest is efficient, interpretable and non-parametric for various types of datasets [25]. The model interpretability and prediction accuracy provided by Random Forest is very unique among popular classification methods. Accurate predictions and better generalizations are achieved due to utilization of ensemble strategies and random sampling.

In a Random Forest, the features are randomly selected in each decision split. The correlation between trees is reduced by randomly selecting the features which improves the prediction power and results in higher efficiency. As such some of the benefits of Random Forest are:

- ✓ High levels of predictive accuracy delivered automatically
- ✓ Only a few control parameters to experiment with
- ✓ Strong for both regression and classification
- ✓ Resistant to overtraining (over fitting) – generalizes well to new data
- ✓ Trains rapidly even with thousands of potential predictors
- ✓ No need for prior feature (variable) selection
- ✓ Diagnostics pinpoint multivariate outliers
- ✓ Offers a revolutionary new approach to clustering using tree-based between-record distance measures.

The Random Forest Algorithm

1. For b = 1 to B:
   (a) Draw a bootstrap sample Z* of size N from the training data.
   (b) Grow a random-forest tree T_b to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size n_{min} is reached.
      i. Select m variables at random from the *p* variables.
      ii. Pick the best variable/split-point among the *m*.
      iii. Split the node into two daughter nodes.

2. Output the ensemble of trees $\{Tb\}_1^B$.

To make a prediction at a new point x:

Regression: $f_{rf}^B(x) = \frac{1}{B}\sum_{b=1}^{B} Tb(x)$. (3)

Classification: Let $C_b(x)$ be the class prediction of the b_{th} random-forest tree.

Then $C_{rf}^B(x) = majority\ vote\{C_b(x)\}_1^B$. (4)

After growing all trees using random data, the classification process is done by obtaining a class vote from each tree and then prediction is made using majority vote.

Details of Random Forest Algorithm

The idea behind Random Forest algorithm is to improve the variance reduction of bagging by reducing the correlation between the trees, without increasing the variance too much. This is achieved in the tree-growing process through random selection of the input variables.

Random Forest Tree Growing

Since bagging is used to build a Random Forest, the datasets must be bagged before they are used to build the tree. This is accomplished by setting up a bagger. When we Bag a data we are creating a bootstrap sample of the given instances which is to mean creating a new set of dataset of the same size using random sampling with replacement according to the given weight vector.

Classifier is built by recursively generating each tree using these new In-Bag data as a training set. The number of trees created is specified at the beginning. Generating a single tree in random forest is similar to creating a decision tree but with some difference. In random forest, the features are randomly selected in each decision split. This is done to reduce the correlation between trees and improve the prediction power which results in higher efficiency.

- Randomness introduced in two simultaneous ways
    - By row: records selected for training at random with replacement (as in bootstrap re-sampling of the bagger)
    - By column: candidate predictors at any node are chosen at random and best splitter selected from the random subset
- Each tree is grown to the largest possible extent and no pruning is used since experiments convincingly show that pruning these trees hurt performance.
- Trees are deliberately over fit, becoming a form of nearest neighbor predictor
- Possibly over fit individual trees combine to yield properly fit ensembles

RF Split selection

- Before each split, for P number of input variables, the variable m is selected such that $m<<P$ is specified at each node, m variables are selected at random out of the P using random number generator and the best split on these m is used for splitting the node. During the forest growing, the value of m is held constant.

- For classification, the default value for m is $\lfloor\sqrt{P}\rfloor$ and the minimum node size is one.
- For regression, the default value for m is $\lfloor P/3\rfloor$ and the minimum node size is five.

- The best splitter from the eligible random subset is used to split the node in question. If the splitter is not very good we just end up with two children that are essentially alike
- Intuitively, reducing m will reduce the correlation between any pair of trees in the ensemble, and hence reduce the variance of the average. In practice the best values for these parameters will depend on the problem, and they should be treated as tuning parameters.

RF tree evolution

- Once a node is split on the best eligible splitter the process is repeated in its entirety on each child node
- A new list of eligible predictors is selected at random for each node
- With a large number of predictors, the eligible predictor set will be quite different from node to node
- Important variables will make it into the tree (eventually)
- Explains in part why the trees must be grown out to absolute maximum full size
- Aim for terminal nodes with one data record

RF prediction Mechanism

- Ultimately in modeling our goal is to produce a single score, prediction, forecast, or class assignment. The motivation for generating multiple models is the hope that by somehow combining models results will be better than if we relied on a single model.
- When used for classification, a random forest obtains a class vote from each tree, and then classifies using majority vote. When used for regression, the predictions from each tree at a target point x are simply averaged, as in

Random Forests and Uncorrelated Trees

- Combining trees via averaging or voting will only be beneficial if the trees are different from each other. The more similar the trees the less advantage to combining as the results of each model become near identical to one another.
- If you grow many trees each based on a different random sample of your data you expect some variation in the trees produced. So re-sampling of the bagger from the training data is intended to induce differences in trees. Bootstrap sample goes a bit further in ensuring that the new sample is of the same size as the original by allowing some records to be selected multiple times. Random Forests induces vastly more between-tree difference by forcing splits to be based on different predictors. In practice the different samples induce different trees.

The Out-Of-Bag (OOB) Error Estimate

Most of the time with Random Forests, bagging is used in tandem with random feature selection. Each new training set is drawn, with replacement, from the original training set. Then a tree is grown on the new training set using random feature selection. The trees grown are not pruned. There are two reasons for using bagging. The first is that the use of bagging seems to enhance accuracy when random features are used. The second is that bagging can be used to give ongoing estimates of the generalization error (PE*) of the combined ensemble of trees, as well as estimates for the strength and correlation. These estimates are done out-of-bag, which is explained as follows.

Assume a method for constructing a classifier from any training set. Given a specific training set T, form bootstrap training sets T_k, construct classifiers $h(\mathbf{x}, T_k)$ and let these vote to form the bagged predictor. For each y**,** **x** in the training set, where y and **x** are random vectors used to draw training sets T_k , aggregate the votes only over those classifiers for which T_k does not containing y**,** **x.** Call this the out-of-bag classifier. Then the out-of-bag estimate for the generalization error is the error rate of the out-of-bag classifier on the training set.

In each bootstrap training set, for every tree grown, about one-third of the instances are left out. These OOB cases are used as test data to evaluate the performance of the current tree. For any tree in RF, its own OOB sample is used: a true random sample. Therefore, the final out-of-bag estimates for the entire Random Forest are based on combining only about one third as many

classifiers as in the ongoing main combination. Since the error rate decreases as the number of combinations increases, the out-of-bag estimates will tend to overestimate the current error rate. To get unbiased out-of-bag estimates, it is necessary to run past the point where the test set error converges. But unlike cross-validation, where bias is present but its extent unknown, the out-of-bag estimates are unbiased. And once the OOB error stabilizes, the training can be terminated.

Variable Importance

In data mining applications the input predictor variables are seldom equally relevant. Often only a few of them have substantial influence on the response; the vast majorities are irrelevant and could just as well have not been included. It is often useful to learn the relative importance or contribution of each input variable in predicting the response. In Random Forest, at each split in each tree, the improvement in the split-criterion is the importance measure attributed to the splitting variable, and is accumulated over all the trees in the forest separately for each variable.

Random Forest uses the OOB samples to construct a different variable importance measure and determine the prediction strength of each variable. When the b_{th} tree is grown, the OOB samples are passed down the tree, and the prediction accuracy is recorded. Then the values for the j_{th} variable are randomly permuted in the OOB samples, and the accuracy is again computed. The decrease in accuracy as a result of this permuting is averaged over all trees, and is used as a measure of the importance of variable j in the random forest. This concept variable importance is used to rank attributes and the attribute with the highest importance measure is chosen as the test attribute for the current node in building the tree. This attribute minimizes the information needed to classify samples in the resulting partitions.

3.4 Fuzzy Logic Algorithm

Fuzzy logic was introduced by Dr. Lotfi Zadeh of UC/Berkeley in the 1960's as a means to model the uncertainty of natural language. There are two main reasons to introduce fuzzy logic for intrusion detection [35]. First, many quantitative features, both ordinal and categorical, are involved in intrusion detection and can potentially be viewed as fuzzy variables. For instance, the CPU usage time and the connection duration are two examples of ordinal measurements. An example of a linear categorical measurement is the number of different TCP/UDP services initiated by the same source host. The second reason to introduce fuzzy logic for intrusion detection is that security itself includes fuzziness. Given a quantitative measurement, a range

value or an interval can be used to denote a normal value. Then, any values falling outside the interval will be considered anomalous to the same degree regardless of their different distances to the interval. The same applies to values inside the interval, i.e., all will be viewed as normal to the same degree. Unfortunately, this causes an abrupt separation between normality and anomaly. The use of fuzziness in representing these quantitative features helps to smooth the abrupt separation and provides a measure of the degree of normality or abnormality of a particular measure.

Additional to those reasons other motivation for using fuzzy logic is that in intrusion detection when to raise an alarm is fuzzy. There would be too many alarms if we raise an alarm every time when we find an intrusion event. At what degree of intrusion we should raise an alarm is often depends on different situation. A fuzzy logic system (FLS) can be defined as the nonlinear mapping of an input data set to a scalar output data [36]. A FLS consists of four main parts: fuzzifier, rules, inference engine, and defuzzifier. These components and the general architecture of a FLS are shown in Figure 3.2.

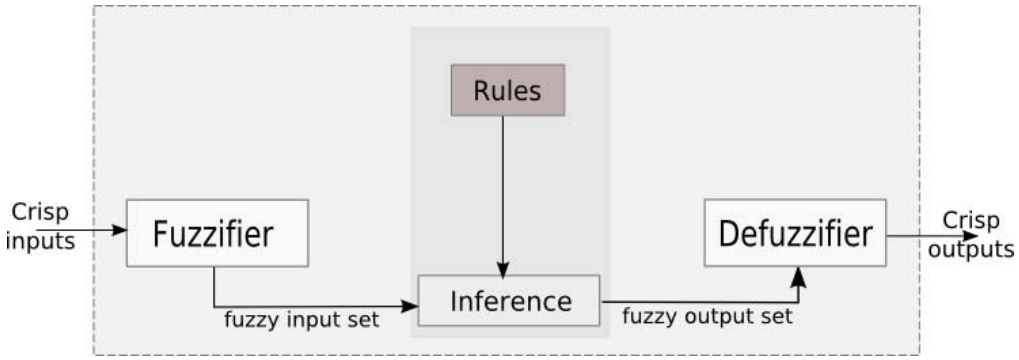

Figure 3.2: A Fuzzy Logic System.

The process of fuzzy logic is explained in Algorithm 1: Firstly, a crisp set of input data are gathered and converted to a fuzzy set using fuzzy linguistic variables, fuzzy linguistic terms and membership functions. This step is known as fuzzification. Afterwards, an inference is made based on a set of rules. Lastly, the resulting fuzzy output is mapped to a crisp output using the membership functions, in the defuzzification step.

Algorithm 1 Fuzzy logic algorithm

1. Define the linguistic variables and terms (initialization)
2. Construct the membership functions (initialization)
3. Construct the rule base (initialization)
4. Convert crisp input data to fuzzy values using the membership functions (fuzzification)
5. Evaluate the rules in the rule base (inference)
6. Combine the results of each rule (inference)
7. Convert the output data to non-fuzzy values (defuzzification)

Linguistic Variables

Linguistic variables are the input or output variables of the system whose values are words or sentences from a natural language, instead of numerical values. A linguistic variable is generally decomposed into a set of linguistic terms.

Example: The attributes of the KDD'99 dataset are the linguistic variable which represents the characteristics of either normal or an attack record. For instance, an attribute called Duration (D) is the linguistic variable which represents the length of the connection. To qualify the duration, terms such as "low" and "high" are used in real life. These are the linguistic values of the duration. Then, D(d) = {low, medium-low, medium, medium-high, high } can be the set of decompositions for the linguistic variable duration. Each member of this decomposition is called a linguistic term and can cover a portion of the overall values of the duration.

Membership Functions

Membership functions are used in the fuzzification and defuzzification steps of a FLS, to map the non-fuzzy input values to fuzzy linguistic terms and vice versa. A membership function is used to quantify a linguistic term. For instance, in Figure 3.3, membership functions for the linguistic terms of duration variable are plotted. Note that, an important characteristic of fuzzy logic is that a numerical value does not have to be fuzzified using only one membership function. In other words, a value can belong to multiple sets at the same time. For example, according to Figure 3.3, a duration value can be considered as "low" and "medium-low" at the same time, with different degree of memberships. The symbols L, ML, M, MH and H indicates linguistic terms low, medium-low, medium, medium-high, high respectively.

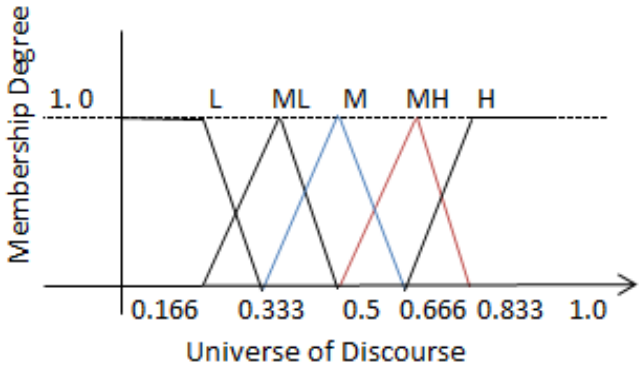

Figure 3.3: Membership Functions for D (duration) = {low, medium-low, medium, medium-high, high}

With fuzzy spaces, fuzzy logic allows an object to belong to different classes at the same time. This concept is helpful when the difference between classes is not well defined. This is the case in the intrusion detection task, where the differences between the normal and abnormal classes are not well defined.

There are different forms of membership functions such as triangular, trapezoidal, piecewise linear, Gaussian, or singleton (Figure 3.4). The most common types of membership functions are triangular, trapezoidal, and Gaussian shapes. The type of the membership function can be context dependent and it is generally chosen arbitrarily according to the user experience [36].

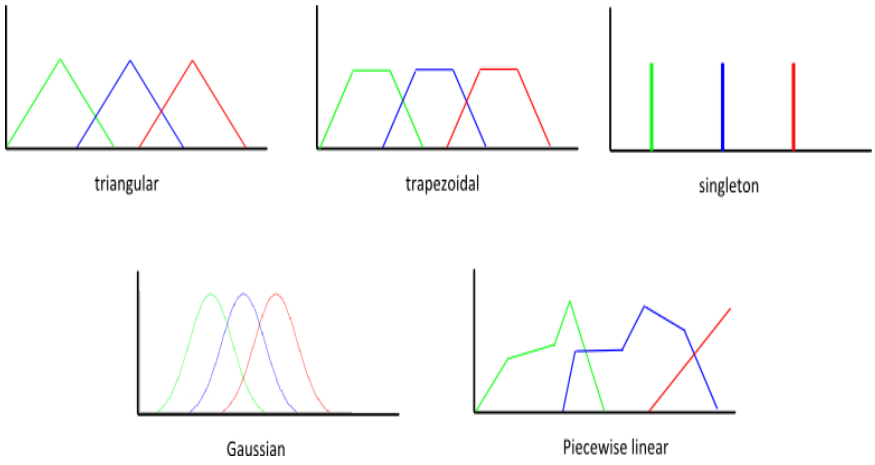

Figure 3.4: Different Types of Membership Functions.

Fuzzy Rules

In a FLS, a rule base is constructed to control the output variable. A fuzzy rule is a simple IF-THEN rule with a form

IF condition *THEN* conclusion

Where,

- *condition* is a complex fuzzy expression, i.e., a logic expression that uses fuzzy logic operators and atomic fuzzy expressions
- *conclusion* is an atomic expression

In general, fuzzy rules are defined within the fuzzy system manually by obtaining the rules from the domain expert or it can be generated automatically. Usually the fuzzy rules are generated from the definite rules, where the IF part of the rule is a numerical variable and THEN part is a class label related to attack name or normal in the intrusion detection system. But, the fuzzy rule should contain only the linguistic variable. So, in order to make the fuzzy rules from the definite rules, we should fuzzify the numerical variable of the *definite rules* and THEN part of the fuzzy rule is same as the consequent part of the *definite rules*.

For example, a rule such as *"IF attribute1 is greater than 5, THEN the data is attack"*, and *"IF attribute1 is less than 2, THEN the data is normal"*. Will be converted to *"IF attribute1 is H, THEN the data is attack* and *"IF attribute1 is VL, THEN the data is normal*" respectively.

The following are other examples of a fuzzy rule for intrusion detection:

- *IF x is HIGH and y is LOW THEN pattern is normal.*
- *If the COUNT of UNUSUAL SDPs on Port N is HIGH And the COUNT of DESTINATION HOSTS is HIGH And the COUNT of SERVICE Ports observed is MEDIUM-LOW Then Service Scan of Port N is HIGH*

Fuzzy Set Operations

Traditionally [37], a standard set like S = {a, b, c, d, e} represents the fact that every member totally belongs to the set S. However, there are many concepts that have to be expressed with

some vagueness. For instance, “tall” is fuzzy in the statement of “John’s height is tall” since there is no clear boundary between “tall” and not “tall” (Bridges et.al).

Fuzzy set theory established by Lotfi Zadeh is the basis of fuzzy logic [35]. A fuzzy set is a set to which its members belong with a degree between 0 and 1. For example, S’ = {(a 0), (b 0.3), (c 1), (d 0.5), (e 0)} is a fuzzy set in which a, b, c, d, and e have membership degrees in the set of S’ of 0, 0.3, 1, 0.5, and 0 respectively. So, it is absolutely true that a, and e do not belong to S’ and c does belong to S’, but b and e are only partial members in the fuzzy set S’. A fuzzy variable (also called a linguistic variable) can be used to represent these concepts associated with some vagueness. A fuzzy variable will then take a fuzzy set as a value, which is usually denoted by a fuzzy adjective. For example, “height” is a fuzzy variable and “tall” is one of its fuzzy adjectives, which can be represented by a fuzzy set.

A standard fuzzy logic system, FuzzyCLIPS provides several methods to represent a fuzzy set. These include singleton representation, standard function representation, and linguistic expression representation.

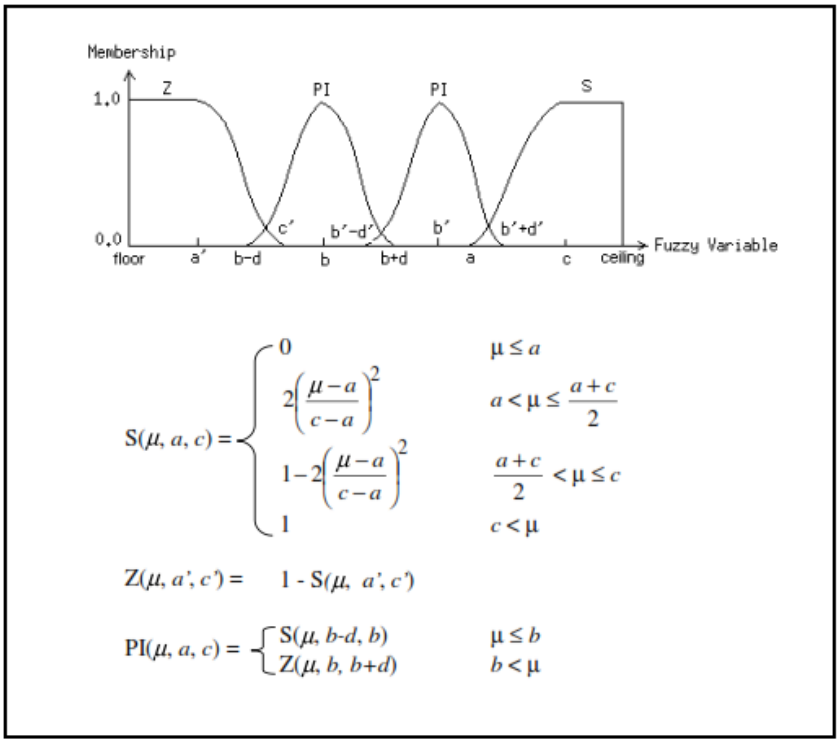

Figure 3.5: Standard Function Representation of Fuzzy Sets.

Where, *parameter* is an object, and *fuzzyset* is a fuzzy set that belongs to the defined fuzzy space for the parameter. The truth-value (TV) of an atomic expression is the degree of membership of the parameter to the fuzzy set. Because TVs are expressed by numbers between 0 and 1, (0 means entirely false, 1 means entirely true, and others values means partially true), the fuzzy expression evaluation process is reduced to arithmetic operations. Also, for each classical logic operator (*and*, *or*, *negation*), there is a common fuzzy logic arithmetic operator (shown in Table 3.1):

The evaluations of the fuzzy rules and the combination of the results of the individual rules are performed using fuzzy set operations. The operations on fuzzy sets are different than the operations on non-fuzzy sets. Let μ_A and μ_B are the membership functions for fuzzy sets A and B. Table 3.1 contains possible fuzzy operations for OR and AND operators on these sets, comparatively. The mostly used operations for OR and AND operators are max and min, respectively. For complement (NOT) operation, Eq. (5) is used for fuzzy sets.

$$\mu_{A^-}(x) = 1 - \mu_A(x) \qquad (5)$$

Table 3.1: Fuzzy set operations

OR(Union)		**AND(intersection)**	
MAX	$Max\{\mu_A(x), \mu_B(x)\}$	MIN	$Min\{\mu_A(x), \mu_B(x)\}$
ASUM	$\mu_A(x) + \mu_B(x) - \mu_A(x)\,\mu_B(x)$	PROD	$\mu_A(x)\,\mu_B(x)$
BSUM	$Min\{1, \mu_A(x) + \mu_B(x)\}$	BDIF	$Max\{0, \mu_A(x) + \mu_B(x) - 1\}$

After evaluating the result of each rule, these results should be combined to obtain a final result. This process is called **inference**. The results of individual rules can be combined in different ways. Table 3.2 contains possible accumulation methods that are used to combine the results of individual rules. The maximum algorithm is generally used for accumulation.

Table 3.2: Accumulation methods

Operation	**Formula**
Maximum	$\text{Max}\{\ \mu_A\ (x), \mu_B\ (x)\}$
Bounded sum	$\text{Min}\{1, \mu_A\ (x)\ + \mu_B\ (x)\}$
Normalized sum	$\frac{\mu_A(x) + \mu_B(x)}{Max\{1, Max\{\mu_A(x'), \mu_B(x')\ \}\}}$

Defuzzification

After the inference step, the overall result is a fuzzy value. This result should be defuzzified to obtain a final crisp output. This is the purpose of the defuzzifier component of a FLS. Defuzzification is performed according to the membership function of the output variable. For instance, assume that we have the result in Figure 3.6 at the end of the inference. In this figure, the shaded areas all belong to the fuzzy result. The purpose is to obtain a crisp value, represented with a dot in the figure, from this fuzzy result.

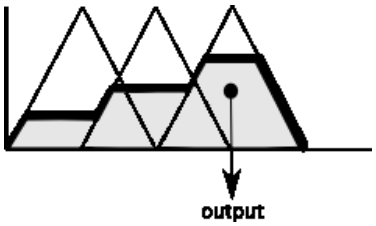

Figure 3.6: Defuzzification step of a FLS.

3.5 Support Vector Machine

Support Vector Machines were first suggested by Vapnik in the 1960s for classification and have recently become an area of intense research owing to developments in the techniques and theory coupled with extensions to regression and density estimation. The support vector machines (SVM) [17, 41] are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis and can handle multiple continuous and categorical variables.

Support Vector Machines are based on the concept of decision planes that define decision boundaries. A decision plane is one that separates between a set of objects having different class memberships. Given a set of training examples, each marked for belonging to one of two categories, an SVM training algorithm builds a model that assigns new examples into one category or the other, making it a non-probabilistic binary linear classifier. A support vector machine model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on. In addition to performing linear classification, SVMs can efficiently perform a non-linear classification using what is called the **kernel trick**, implicitly mapping their inputs into high-dimensional feature spaces.

Support Vector Machine - Classification (SVM)

A Support Vector Machine (SVM) is a discriminative classifier formally defined by a separating hyperplane. In other words, given labeled training data (*supervised learning*), the algorithm outputs an optimal hyperplane which categorizes new examples.

In which sense is the hyperplane obtained optimal? Let's consider the following simple problem: For a linearly separable set of 2D-points which belong to one of the two classes, find a separating straight line.

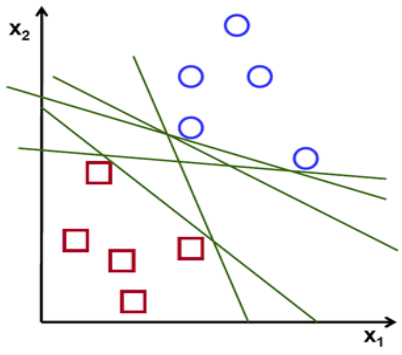

Figure 3.7: Example of separating hyperplanes

In Figure 3.7 you can see that there exist multiple lines that offer a solution to the problem. Are any of them better than the others? We can intuitively define a criterion to estimate the worth of

the lines: A line is bad if it passes too close to the points because it will be noise sensitive and it will not generalize correctly. Therefore, our goal should be to find the line passing as far as possible from all points. Then, the operation of the SVM algorithm is based on finding the hyperplane that gives the largest minimum distance to the training examples. Twice, this distance receives the important name of **margin** within SVM's theory. Therefore, the optimal separating hyperplane *maximizes* the margin of the training data as shown in Figure 3.8.

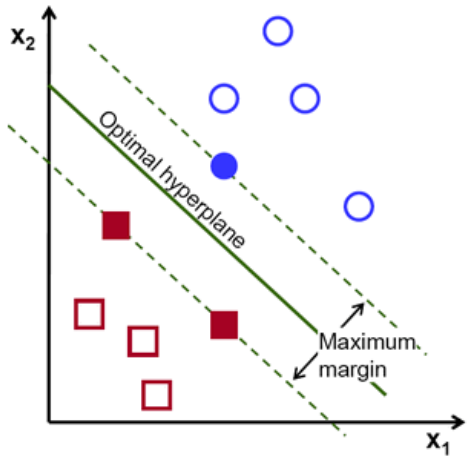

Figure 3.8: Linear separation of the data points into two classes

SVM Algorithm

Let D be a classification dataset with n points in a d-dimensional space $D = \{(x_i, y_i)\}$, with $i = 1, 2, \ldots, n$ and let there be only two class labels such that y_i is either +1 or -1. A hyperplane $h(x)$ gives a linear discriminant function in d dimensions and splits the original space into two half-spaces:

$$h(x) = wx + b = w_1x_1 + w_2x_2 + \cdots + w_dx_d + b \qquad (6)$$

, where w is a d-dimensional weight vector, b is a scalar bias and x symbolizes the training examples closest to the hyperplane. In general, the training examples that are closest to the hyperplane are called **support vectors**. This representation is known as the **canonical hyperplane**. Points on the hyperplane have $h(x) = 0$, i.e. the hyperplane is defined by all points for which $wx = -b$.

According to [38], if the dataset is linearly separable, a separating hyperplane can be found such that for all points with label -1, $h(x) < 0$ and for all points labeled +1, $h(x) > 0$. In this case, $h(x)$ serves as a linear classifier or linear discriminant that predicts the class for any point. Moreover, the weight vector w is orthogonal to the hyperplane, therefore giving the direction that is normal to it, whereas the bias b fixes the offset of the hyperplane in the d-dimensional space.

Constraints

- Separation with margin, i.e.

$$(w, x_i) + b \geq 1 \qquad if\ y_i = 1 \qquad (7)$$

$$(w, x_i) + b \leq -1 \qquad if\ y_i = -1 \qquad (8)$$

- Equivalent constraint

$$y_i(w, x_i) + b \geq 1 \qquad (9)$$

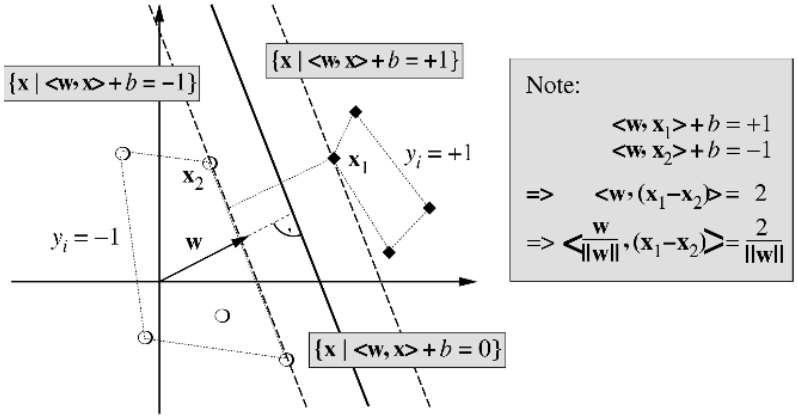

Figure 3.9: Optimal Separating Hyperplane

Given a separating hyperplane $h(x) = 0$, it is possible to calculate the distance between each point x_i and the hyperplane by:

$$\delta_i = \frac{y_i h(x_i)}{||w||} \qquad (10)$$

The margin of the linear classifier is defined as the minimum distance of all *n* points to the separating hyperplane.

$$\text{Width} = \delta^* = \min_{x_i} \{ \frac{y_i h(x_i)}{||w||} \} \quad (11)$$

All points (vectors x^*_i) that achieve this minimum distance are called the support vectors for the linear classifier. In other words, a support vector is a point that lies precisely on the margin of the classifying hyperplane. In a canonical representation of the hyperplane, for each support vector x^*_i with label y^*_i we have that $y_i^* h(x_i^*) = 1$. Similarly, for any point that is not a support vector, we have that $y_i h(x_i) > 1$, since, by definition, it must be farther from the hyperplane than a support vector. Therefore we have that $y_i h(x_i) \geq 1, \forall x_i \epsilon D$.

The optimal hyperplane can be represented in an infinite number of different ways by scaling of *w* and *b*. The fundamental idea behind SVMs is to choose the hyperplane with the maximum margin, i.e. the optimal canonical hyperplane. To do this, one need to find the weight vector *w* and the bias *b* that yields the maximum margin among all possible separating hyperplanes, that is, the hyperplane that maximizes $\frac{1}{||w||}$. The problem then becomes that of solving a convex minimization problem (notice that instead of maximizing the margin $\frac{1}{||w||}$, one can obtain an equivalent formulation of minimizing $||w||$) with linear constraints, as follows:

Objective Function

$$min \ \frac{||w||^2}{2} \quad (12)$$

Linear Constraints

$$y_i h(x_i) \geq 1, \forall x_i \epsilon D \quad (13)$$

This minimization problem can be solved using the *Lagrange multiplier* method, which introduces a Lagrange multiplier α for each constraint:

$$L(w, b, \alpha) = \text{PrimalObjective} + \sum_i \alpha i \, Ci \ , \ Ci = constraints \quad (14)$$

$$L(w, b, \alpha) = \frac{||w||^2}{2} + \sum_{i=1}^{m} \alpha_i (1 - y_i((w, x_i) + b)) \quad (15)$$

This method states that $\alpha_i = 0$ for all points that are at a distance larger than $\frac{1}{||w||}$ from the hyperplane, and only for those points that are exactly at the margin, i.e. the support vectors, $\alpha_i > 0$. The weight vector of the classifier is obtained as a linear combination of the support vectors, while the bias is the average of the biases obtained from each support vector [38].

Saddle Point Condition

Derivatives of L with respect to w and b must vanish.

$$\partial_w L(w, b, \alpha) = w - \sum_{i=1}^{m} \alpha_i y_i x_i = 0 \quad \leftrightarrow \quad w = \sum_{i=1}^{m} \alpha_i y_i x_i \quad (16)$$

$$\partial_b L(w, b, \alpha) = -\sum_{i=1}^{m} \alpha_i y_i x_i = 0 \quad \leftrightarrow \quad \sum_{i=1}^{m} \alpha_i y_i = 0 \quad (17)$$

To obtain the dual optimization problem we have to substitute the values of w and b into L. Note that the dual variables αi have the constraint $\alpha i \geq 0$.

Dual Optimization Problem

After substituting in terms for b, w the Lagrange function becomes

$$L(w, b, \alpha) = -\frac{1}{2} \sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j (x_i, x_j) + \sum_{i=1}^{m} \alpha_i \quad (18)$$

Subject to

$$\sum_{i=1}^{m} \alpha_i y_i = 0 \ \ and \ \alpha_i \geq 0 \text{ for all } 1 \leq i \leq m \quad (19)$$

Note: Only points at the decision boundary can contribute to the solution.

Support Vector Expansion (Decision Rule)

The decision rule for the classification becomes as follows

$$w = \sum_{i=1}^{m} \alpha_i y_i x_i \ \ and \quad (20)$$

$$hence \ h(x) = wx + b = \sum_{i=1}^{m} \alpha_i y_i (x_i, x) + b \quad (21)$$

- m is the number of support vector training records

- x is the new dataset which will be classified
- x_i is the support vectors
- w is given by a linear combination of training patterns x_i. Independent of the dimensionality of x.
- w depends on the Lagrange multipliers αi.

Support Vector Machines for Non-Linearly Separable Data

SVMs can handle linearly non-separable points, where the classes overlap to some extent so that a perfect separation is not possible, by introducing slack variables ε_i for each point x_i in D. If $0 \leq \varepsilon_i < 1$, the point is still correctly classified but if $\varepsilon_i > 1$, the point is misclassified. So the goal of the classification becomes that of finding the hyperplane (w and b) with the maximum margin that also minimizes the sum of slack variables. A methodology similar to that described above is necessary to find the weight vector w and the bias b.

The simplest way to separate two groups of data is with a straight line (1 dimension), flat plane (2 dimensions) or an N-dimensional hyperplane. However, there are situations where a non-linear decision boundary can separate the groups more efficiently. SVM handles this by using a kernel function (nonlinear) to map the data into a different space where a hyperplane (linear) cannot be used to do the separation. It means a non-linear function is learned by a linear learning machine in a high-dimensional feature space while the capacity of the system is controlled by a parameter that does not depend on the dimensionality of the space. This is called *kernel trick* which means the kernel function transform the data into a higher dimensional feature space to make it possible to perform the linear separation.

The main idea is to map the original d-dimensional space into a d'-dimensional space ($d' > d$), where the points can possibly be linearly separated. Given the original dataset $D = \{x_i, y_i\}$ with $i = 1,...,n$ and the transformation function Φ, a new dataset is obtained in the transformation space $D_\Phi = \{\Phi(x_i), y_i\}$ with $i = 1,...,n$. After the linear decision surface is found in the d'-dimensional space, it is mapped back to the non-linear surface in the original d-dimensional space [39]. To obtain w and b, $\Phi(x)$ needn't be computed in isolation. The only operation required in the transformed space is the inner product $\Phi(x_i)^T\Phi(x_j)$, which is defined with the kernel function (K) between x_i and x_j.

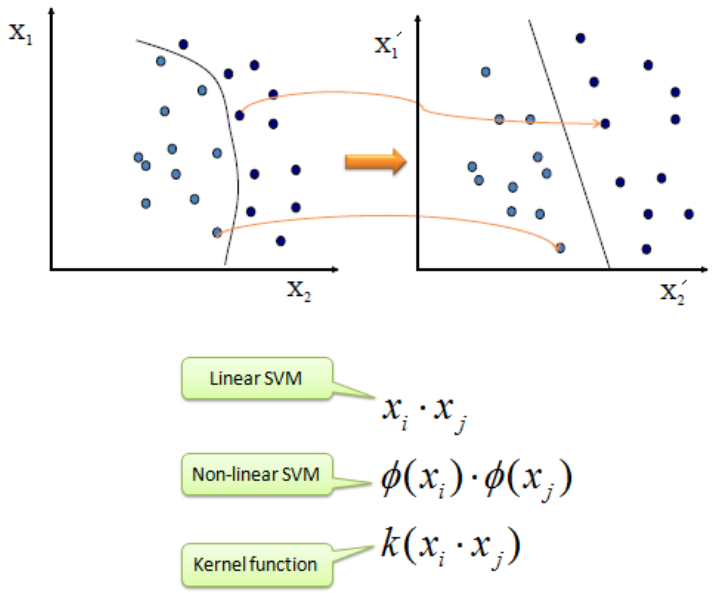

Figure 3.10: Mapping of Non-Linearly Separable Data using Kernel Function

Nonlinearity via Feature Maps

Replace x_i by $\Phi(x)$ in the optimization problem.

Equivalent optimization problem

$$minimize\ \frac{1}{2}\sum_{i,j=1}^{m} \alpha_i \alpha_j y_i y_j k(x_{i,} x_j) - \sum_{i=1}^{m} \alpha_i \quad (22)$$

Subject to

$$\sum_{i=1}^{m} \alpha_i y_i = 0 \ \ and\ \alpha_i \geq 0 \quad (23)$$

Decision Function

$$w = \sum_{i=1}^{m} \alpha_i y_i\, \phi(x_i) \ \ implies \quad (24)$$

$$h(x) = \left(\left(w, \phi(x)\right) + b = \sum_{i=1}^{m} a_i y_i\, k\left(x_{i,} x\right) + \ b\right. \quad (25)$$

Kernels commonly used with SVMs include:

- the polynomial kernel

$$K\left(x_i, x_j\right) = (x_i^T x_j)^q \text{ , where } q \text{ is the degree of the polynomial} \quad (26)$$

- the gaussian kernel:

$$K(x_i, x_j) = e^{-\frac{\left\|x_i - x_j\right\|^2}{2\sigma^2}}, where\ \sigma\ is\ the\ spread\ or\ standard\ deviation \quad (27)$$

- the gaussian radial basis function (RBF):

$$K(x_i, x_j) = e^{-y||x_i - x_j||^2}, y \geq 0 \quad (28)$$

- the Laplace Radial Basis Function (RBF) kernel:

$$K(x_i, x_j) = e^{-y\left||x_i - x_j|\right|}9, y \geq 0 \quad (29)$$

- the hyperbolic tangent kernel:

$$K(x_i, x_j) = \tanh(x_i^T x_j + of\ fset) \quad (30)$$

- the sigmoid kernel:

$$K(x_i, x_j) = \tanh(ax_i^T x_j + of\ fset) \quad (31)$$

- the Bessel function of the first kind kernel:

$$K(x_i, x_j) = \left(\frac{Bassel_{v+1}^{n}(\sigma||x_i - x_j||)}{(||x_i - x_j||)^{-n(v+1)}}\right) \quad (32)$$

- the ANOVA radial basis kernel:

$$K(x_i, x_j) = \left(\sum_{k=1}^{n} e^{-\sigma\left(x_i^k - x_j^k\right)^2}\right)^d \quad (33)$$

- the linear splines kernel in one dimension:

$$K(x_i, x_j) = 1 + x_i x_j min(x_i, x_j) - \frac{x_i + x_j}{2} min(x_i, x_j)^2 + \frac{min(x_i, x_j)^3}{3} \quad (34)$$

According to [40], the Gaussian and Laplace RBF and Bessel kernels are general-purpose kernels used when there is no prior knowledge about the data. The linear kernel is useful when dealing with large sparse data vectors as is usually the case in text categorization. The polynomial kernel is popular in image processing, and the sigmoid kernel is mainly used as a proxy for neural networks. The splines and ANOVA RBF kernels typically perform well in regression problems.

In this chapter we have seen the working principles of five selected algorithms which use different techniques to classify data. Some, like K-means clustering uses unsupervised data and groups objects in a predetermined number of clusters by only using similarities in attributes. Decision Tree and Random Forest are a tree based classification models which are trained on labeled data in order to classify unknown test data. The former introduces variable's entropy and the latter uses randomization to select training data as well as attribute variables. Others such as Fuzzy Logic and SVM use a rule base and a function to categorize a given data respectively.

All of those techniques provide advantages of their own. But they also have some weaknesses in detecting intrusions. Researchers have shown that those algorithms can be optimized or combined together for the purpose of improving performance of intrusion detection. In order to discover their strength and weakness especially in identifying specific type of attack, a performance evaluation is done. This experiment is explained in the next chapter.

Chapter 4

Evaluation study

In order to verify the effectiveness of those diverse algorithms, which are deeply explained in previous section, this research uses the KDD'99 dataset and construct an experiment step-by-step. Firstly, the experimental evaluation environment is built with major steps: environment/ evaluation setup, choosing the data mining software, data preprocessing. Secondly, a comprehensive set of intrusion detection algorithms are selected. Those five distinct widely used algorithms are K-means, Decision Tree, Random Forest, Fuzzy Logic and SVM. They represent a wide variety of fields such as clustering approaches, decision trees, rule based models and function studying. An overview of which specific parameters of these algorithms were used as well as their detection performance will be given. Finally, a performance comparison between those five selected algorithms is done.

4.1 Evaluation Setup

All evaluation experiments were performed in a two-year-old computer with the following hardware and software specifications: Intel(R) Core i5 CPU @2.40GHz, processor with 8 GB RAM, and the operating system platform is Microsoft Windows 7, 64 bit with HDD 500 GB. The software used is an open source machine learning package called Weka (the latest Windows version: Weka 3.6.4). Weka is a collection of machine learning algorithms for data mining tasks that contains tools for data preprocessing, classification, regression, clustering, association rules, and visualization. Any programming has been done using java programming language on NetBeans IDE 8.

WEKA

The name Weka stands for Waikato Environment for Knowledge Analysis and was developed at the University of Waikato in New Zealand [49]. The Weka workbench is a collection of state-of-the-art data mining and machine learning algorithms and data preprocessing tools. It is designed so that we can quickly try out existing methods on new datasets in flexible ways.

WEKA offers a uniform interface to many different learning algorithms and provides extensive support for the whole process of experimental data mining, including preparing the input data,

evaluating learning schemes such as clustering, classification, regression, visualization, and feature selection statistically, and visualizing the input data and the result of learning. All of WEKA's techniques are predicated on the assumption that the data is available as a single flat file or relation, where each data point is described by a fixed number of attributes (normally, numeric or nominal attributes, but some other attribute types are also supported).

This diverse and comprehensive toolkit is accessed through a common interface so that its users can compare different methods and identify those that are most appropriate for the problem at hand. All algorithms take their input in the form of a single relational table in the ARFF format. The easiest way to use WEKA is through a graphical user interface called Explorer. This gives access to all of its facilities using menu selection and form filling but essentially the same functionality can be accessed through the component-based Knowledge Flow interface and the command line. There is also the Experimenter, which allows the systematic comparison of the predictive performance of Weka's machine learning algorithms on a collection of datasets. The Explorer interface features several panels providing access to the main components of the workbench.

The WEKA system is written in Java and distributed under the terms of the GNU General Public License. It runs on almost any platform and has been tested under Linux, Windows, and Macintosh operating systems and even on a personal digital assistant. It also provides access to SQL databases using Java Database Connectivity and can process the result returned by a database query.

4.2 KDD CUP 99 Data Set

The KDD'99 intrusion detection datasets are based on the 1999 DARPA initiative which was IDS evaluation program [50]. It provides designers of intrusion detection systems (IDS) with a benchmark on which to evaluate different methodologies. This simulation is made of a factitious military network consisting of three "target" machines running various operating systems and services. Additional three machines are then used to spoof different IP addresses to generate traffic. Finally, there is a sniffer that records all network traffic using the TCP dump format. DARPA'99 is about 4 gigabytes of compressed raw (binary) tcpdump data of 7 weeks of network traffic, which can be processed into about 5 million training connection

records, each with about 100 bytes. The two weeks of test data have around 2 million connection records.

The learner can be trained using the training portion of the dataset and evaluated for its efficiency on the test portion. A single training dataset contains 41 features and is labeled with exact one specific type i.e., either normal or an attack which is a different type. A smaller version 10% training dataset is also provided for memory constrained machine learning methods. It consists of 10% of original dataset that is approximately 494,020 single connection vectors. The training dataset has

- ✓ 19.69% normal and
- ✓ 80.31% attack connections.

The simulated attack types found in the KDD CUP 99 falls in one of the following four broad categories [52].

1. Denial of Service Attack (DOS): This type of attack aims to interfere with the normal operation of network services by flooding, exhausting and overwhelming the resources of the targeted network or host. The resources may be network bandwidth, router's packet forwarding capacity, name servers, memory/computing power on servers, or operating systems' data structures. During DoS attacks, attackers usually generate large amounts of nonsense traffic such as incomplete TCP connections, malformed IP packets, bots-generated requests to web pages and some other carefully crafted methods, causing the service or program to cease functioning or prevent others from making use of the service or program. DoS contains the attacks: 'neptune', 'back', 'smurf', 'pod', 'land', and 'teardrop'.

2. Users to Root Attack (U2R): In this category the attacker starts out with access to a normal user account on the system and is able to exploit some vulnerability to obtain root access to the system. U2R contains the attacks: 'buffer_overflow', 'loadmodule', 'rootkit' and 'perl'.

3. Remote to Local Attack (R2L): occurs when an attacker who has the ability to send packets to machine over a network but who does not have an account on that machine and exploits some vulnerability to gain local access as a user of that machine. R2L contain the attacks: 'warezclient', ' multihop', ' ftp_write', 'imap', 'guess_passwd', 'warezmaster', 'spy' and 'phf'.

4. Probing Attack (PROBE): Network probes are usually attacks scanning computer networks to gather information or find known vulnerabilities, which are exploited for further or future attacks. The goal of this information gathering is to find out about computer and services that are present in a network as well as to detect the possibility of attack based on known vulnerabilities. PROBE contains the attacks: 'portsweep', 'satan', 'nmap' and 'ipsweep'.

Table 4.1: Distribution of connection types in KDD'99 10% training dataset

Class	Number of records	% of occurrence
Normal	97,277	19.69
DoS	391,458	79.24
Probe	4,107	0.83
U2R	52	0.01
R2L	1,126	0.23
Total	494,020	100.00

The first column in Table 4.1 is the name of the classes, including normal and list of attack types, found in the dataset , the second column mentions the number of records found in the dataset that are used for experimentation and the last column is the percentage of occurrence in the dataset. DoS attack type contains the maximum number of records and U2R contains the least.

Table 4.2: Classification of dataset based on attack category

Attack type	Sub attacks types
Normal	Normal
DOS	Smurf,teardrop,pod,back,land,apache2,udpstrom,mailbomb,processtable, neptune
Probe	Ipseewp,portsweep,nmap,satan,saint,mscan
R2L	Dictionary,ftp_write,guess_password,imap,named,sendmail,spy,xlock,xs noop,Snmpgetattack,httptunnel,worm,snmpguess,multihop,phf,wraezclie nt,wrazemaster
U2R	Perl,ps,xterm.loadmodule,eject,buffer_overflow,sqlattack

Table 4.2 shows classification of dataset based on attack category. The first column contains the major dataset categories found in the KDD'99 datasets, which are normal and the four attack types. The second column is the sub attack types that are found in each main attack category. The number of these sub attack types found in KDD'99 is presented in Table 4.3. The first column lists the name of the attacks. The second column shows how many of those attack types found in the training dataset. And the third column illustrates to which category those attack types belong to. From the table can see that smurf and neptune attack types, which both belong to DoS attack category, have the largest number among the others. Attack types with smallest number are perl which is a U2R attack and spy which is R2L attack type. This shows the dataset contains more DoS and PROBE attack types than U2R and R2L.

Table 4.3: Number of Attacks in the dataset

Attack Name	**Attacks in training dataset**	**Category**
normal	97,277	Normal
neptune	101,201	DOS
portsweep	1,040	PROBE
satan	1,589	PROBE
warezclient	1,020	R2L
back	2,203	DOS
ipsweep	1,247	PROBE
buffer_overflow	30	U2R
multihop	7	R2L
loadmodule	9	U2R
teardrop	979	DOS
rootkit	10	U2R
ftp_write	8	R2L
imap	12	R2L
guess_passed	53	R2L
nmap	231	PROBE
smurf	280,790	DOS

warezmaster	20	R2L
perl	3	U2R
spy	2	R2L
pod	264	DOS
land	21	DOS
phf	4	R2L

The major objectives performed by detecting network intrusion are stated as recognizing rare attack types such as U2R and R2L, increasing the accuracy detection rate for suspicious activity, and improving the efficiency of real-time intrusion detection models.
The protocols that are considered in KDD'99 dataset [51] are TCP, UDP, and ICMP. And they are explained below:

TCP: TCP stands for "Transmission Control Protocol". TCP is an important protocol of the Internet Protocol Suite at the Transport Layer, which is the fourth layer of the OSI model. It is a reliable connection-oriented protocol which implies that data sent from one side is sure to reach the destination in the same order. TCP splits the data into labeled packets and sends them across the network. TCP is used for many protocols such as HTTP and Email Transfer.

UDP: UDP stands for "User Datagram Protocol". It is similar in behavior to TCP except that it is unreliable and connection-less protocol. As the data travels over unreliable media, the data may not reach in the same order, packets may be missing and duplication of packets is possible. This protocol is a transaction-oriented protocol which is useful in situations where delivery of data in certain time is more important than loosing few packets over the network. It is useful in situations where error checking and correction is possible in application level.

ICMP: ICMP stands for "Internet Control Message Protocol". ICMP is basically used for communication between two connected computers. The main purpose of ICMP is to send messages over networked computers. The ICMP redirect the messages and it is used by routers to provide the up-to-date routing information to hosts, which initially have minimal routing information. When a host receives an ICMP redirect message, it will modify its routing table according to the message. Type of attacks grouped by protocol is shown in Table 4.4. The first

column is the type of protocols that are found in the KDD'99 datasets. The second column contains the sub attack types that belong in each protocol.

Table 4.4: Type of attacks grouped by Protocol

Protocol Type	Attack Name
UDP	Normal, teardrop, satan, nmap,rootkit
TCP	Normal,neptune,guess_passed,land,portsweep, buffer_overflow,phf,warezmaster,ipsweep, multihop,warezclient,perl,back, ftp_write,loadmodule, satan,spy, imap, rootkit
ICMP	Normal,portsweet,ipsweep,smurf,satan,pod,nmap

In 1999, the original TCP dump files were preprocessed for utilization in the Intrusion Detection System benchmark of the International Knowledge Discovery and Data Mining Tools Competition. To do so, packet information in the original TCP dump file were summarized into connections. Specifically, "a connection is a sequence of TCP packets starting and ending at some well-defined times, between which data flows from a source IP address to a target IP address under some well-defined protocol". This process is completed using the Bro IDS [52], resulting in 41 features for each connection (and one final feature for classifying, of course),

KDD'99 features can be classified into three groups:
1) Basic Features: this category encapsulates all the attributes that can be extracted from a TCP/IP connection. Most of these features leading to an implicit delay in detection.

2) Traffic features: this category includes features that are computed with respect to a window interval and is divided into two groups:

a) "Same host" features: examine only the connections in the past 2 seconds that have the same destination host as the current connection, and calculate statistics related to protocol behavior, service, etc.
b) "Same service" features: examine only the connections in the past 2 seconds that have the same service as the current connection.

The two aforementioned types of "traffic" features are called time-based. However, there are several slow probing attacks that scan the hosts (or ports) using a much larger time interval than 2 seconds, for example, one in every minute. As a result, these attacks do not produce intrusion patterns with a time window of 2 seconds. To solve this problem, the "same host" and "same service" features are re-calculated but based on the connection window of 100 connections rather than a time window of 2 seconds. These features are called connection-based traffic features.

3) Content features: unlike most of the DoS and Probing attacks, the R2L and U2R attacks don't have any intrusion frequent sequential patterns. This is because the DoS and Probing attacks involve many connections to some host(s) in a very short period of time; however, the R2L and U2R attacks are embedded in the data portions of the packets, and normally involves only a single connection. To detect these kinds of attacks, we need some features to be able to look for suspicious behavior in the data portion, e.g., number of failed login attempts. These features are called content features.

Inherent Problems of KDD'99 Data Set

- One of the most important deficiencies in the KDD'99 data set is the huge number of redundant records [51], which causes the learning algorithms to be biased towards the frequent records, and thus prevent them from learning infrequent records which are usually more harmful to networks such as U2R and R2L attacks. In addition, the existence of these repeated records in the test set will cause the evaluation results to be biased by the methods which have better detection rates on the frequent records. In order to avoid this kind problem we have remove repeated records in the dataset and kept only one copy of each record.

The main reason this dataset is used is because, a relevant data that can easily be shared with other researchers allowing all kinds of techniques to be easily compared in the same baseline is needed, and a KDD'99 dataset provide that. The common practice in intrusion detection to claim good performance with "live data" makes it difficult to verify and improve pervious research results, as the traffic is never quantified or released for privacy concerns. The KDD'99 dataset might have been criticized for its potential problems [51], but the fact is that it is the most widespread dataset that is used by many researchers and it is among the few comprehensive datasets that can be shared in intrusion detection nowadays.

4.2.1 Data Preprocessing

Attributes in the KDD datasets had all forms continuous, discrete, and symbolic, with significantly varying resolution and ranges. Most pattern classification methods are not able to process data in such a format. Hence preprocessing was required before pattern classification models could be built. Preprocessing consisted of two steps: first step involved mapping symbolic-valued attributes to numeric-valued attributes and second step implemented scaling. Attack names (like buffer_overflow, guess_passwd, etc.) were first mapped to one of the five classes, 0 for Normal, 1 for Probe, 2 for DoS, 3 for U2R, and 4 for R2L, as described in [50]. Symbolic features like protocol_type (3 different symbols), service (70 different symbols), and flag (11 different symbols) were mapped to integer values ranging from 0 to N-1 where N is the number of symbols. Then each of these features was linearly scaled to the range [0.0, 1.0]. Features having smaller integer value ranges like duration [0, 58329], wrong_fragment [0, 3], urgent [0, 14], hot [0, 101], num_failed_logins [0, 5], num_compromised [0, 9], su_attempted [0, 2], num_root [0, 7468], num_file_creations [0, 100], num_shells [0, 5], num_access_files [0, 9], count [0, 511], srv_count [0, 511], dst_host_count [0, 255], and dst_host_srv_count [0, 255] were also scaled linearly to the range [0.0, 1.0]. Two features spanned over a very large integer range, namely src_bytes [0, 1.3 billion] and dst_bytes [0, 1.3 billion]. Logarithmic scaling (with base 10) was applied to these features to reduce the range to [0.0, 9.14]. All other features were either boolean, like logged_in, having values (0 or 1), or continuous, like diff_srv_rate, in the range [0.0, 1.0]. Hence scaling was not necessary for these attributes.

The KDD 1999 Cup dataset has a very large number of duplicate records. For the purpose of training different classifier models, these duplicates were removed from the datasets. The total number of records in the original labeled training dataset is 97,277 for Normal, 391,458 for DoS, 4,107 for Probe, 52 for U2R, and 1,126 for R2L attack classes. After filtering out the duplicate records, there were a total of 81,281 records for Normal, 247,267 for DoS, 3,860 for Probe, 52 for U2R, and 1026 for R2L attack classes. The following Figure shows the KDD'99 dataset we used for the experiment when it is loaded to WEKA.

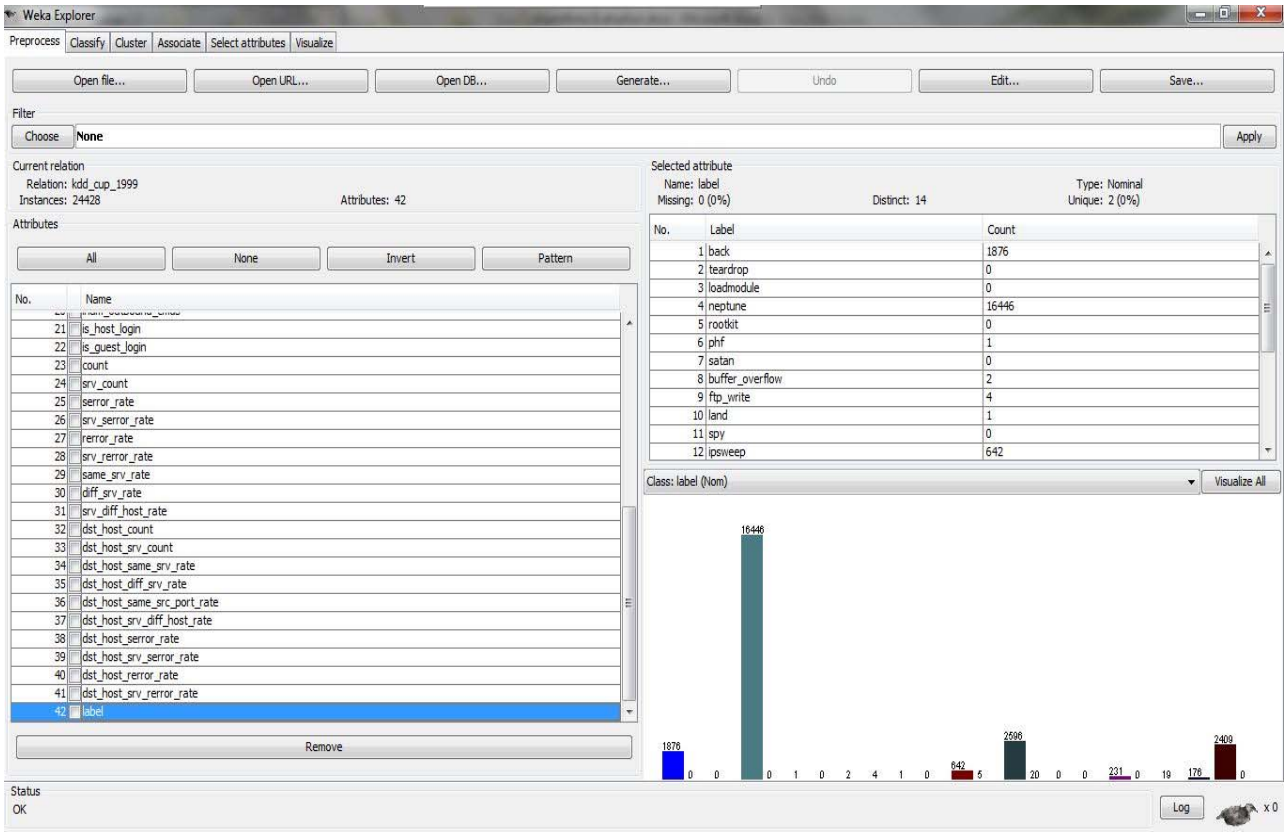

Figure 4.1: The KDD'99 dataset after its loaded on WEKA

4.3 Evaluation Criteria

Accuracy is a statement of how correct an IDS works, measuring the percentage of detection and failure as well as the number of false alarms that the system is producing [53, 54]. A system that has 80% accuracy is a system that correctly classifies 80 instances out of 100 in their actual class. While there is a big diversity of attacks in intrusion detection, the main focus remains if the system detects an attack or not. Thus, the number of target classes can be considered to be two (i.e. normal and abnormal/intrusion), even if intrusions can be of many types which belong to different categories such as Dos, Probe, U2R and R2L. From the real life experience, one can infer that the actual percentage of abnormal data is much smaller than the percentage of normal one in live network traffic [55, 56]. Consequently, the intrusions are harder to detect than the normal traffic, which results in having excessive false alarms as the biggest problem facing IDSs. In [56] Joshi et al refer to this issue as the detection of Rare Class problem, where the Rare Class represents all the detected and undetected intrusions that happen in a certain interval of time.

4.3.1 False Positive and Negative

In intrusion detection a positive data is considered to be an attack data, while a negative data is considered to be a normal data. Furthermore, when IDS tries to classify data, its decision can be either right or wrong. Assume that true and false stands for right and wrong, respectively. Therefore, due to the two-class nature of the detection we have four combinations of the previous defined variables as follows:

- ✓ True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN).

A TP occurs when an IDS correctly classifies an intrusion, whereas a FP occurs when a legitimate action is misclassified as being an intrusion. Likewise, a TN is produced whenever a normal data is correctly classified as a legitimate action, while a FN occurs when an attack is not detected by the IDS [55, 56, 57].

Table 4.5: IDS Classification (This table is adapted from [57].

	The class of actual data	The prediction of the IDS
True Positive (TP)	Attack	Attack
False Positive (FP)	Normal	Attack
True Negative (TN)	Normal	Normal
False Negative (FN)	Attack	Normal

Consequently, the aim of an IDS is to produce as many TP and TN as possible, while trying to reduce the number of both FP and FN

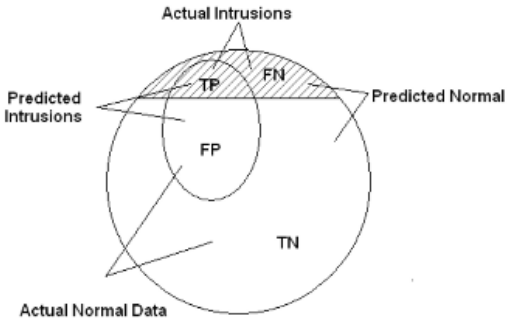

Figure 4.2: The whole data space in a typical Intrusion Detection two-class scenario

Joshi et al. [53] propose a graphical method to visualize the relation between the four variables. Figure 4.2 depicts the normal case of an intrusion detection problem, as follows:

- ✓ the big circle defines the space of the whole data (i.e., normal and intrusive data)
- ✓ The small ellipse defines the space of all predicted intrusions by the classifier. Thus, it will be shared by both TP and FP.
- ✓ The ratio between the real normal data and the intrusions is graphically represented by the use of a horizontal line.

(True Positive Rate) Detection Rate (DR): it is a true positive rate which is computed as the ratio between the numbers of correctly detected attacks and the total number of attacks [57].

(False Positive Rate) False Alarm Rate (FAR): it is false positive rate which is computed as the ratio between the numbers of normal connections that are incorrectly misclassified as attacks and the total number of normal connections [57].

The previous four defined variables (i.e. TP, FP, TN, and FN) encapsulate all the possible results of a two-class intrusion detection problem. Thus, the big majority of evaluation criteria metrics use these variables and the relations between them in order to model the accuracy of the IDSs.

The Following equation shows how we can attain the evaluation criteria metrics: detection rate and false alarm rate using TP, TN, FP and FN.

$$\textit{Detection Rate} = \frac{\textit{Number of attacks classified correctly}}{\textit{Total number of attack connection}} = \frac{TP}{FN+TP} \qquad (34)$$

$$\textit{False Alarm Rate} = \frac{\textit{False Positives}}{\textit{Total number of normal connection}} = \frac{FP}{FP+TN} \qquad (35)$$

Based on the above formula, we have evaluated the five selected algorithms (K-means clustering, Decision Tree, Random Forest, Fuzzy Logic and Support Vector Machine) independently and then the combined approach (our hybrid model). These evaluation criteria help us to compare the performance of each algorithm alone and/to the combined hybrid model.

4.4 Test Options

The result of applying the chosen classifier will be tested according to the options that are set by selecting in the Test options box in WEKA. There are four test modes [49]:

1. **Use training set:** The classifier is evaluated on how well it predicts the class of the instances it was trained on.
2. **Supplied test set:** The classifier is evaluated on how well it predicts the class of a set of instances loaded from a file.
3. **Cross-validation:** The classifier is evaluated by cross-validation, using the number of folds that are entered in the Folds text field.
4. **Percentage split**: The classifier is evaluated on how well it predicts a certain percentage of the data which is held out for testing. The amount of data held out depends on the value entered in the percentage (%) field.

From the four test modes this research chooses to use the Cross-validation. What this mode is and reason for why it is used is explained below.

Cross-Validation

Cross-validation, sometimes called **rotation estimation** [49] is a model validation technique for assessing how the results of a statistical analysis will generalize to an independent data set. It is mainly used in settings where the goal is prediction, and one wants to estimate how accurately a predictive model will perform in practice. In a prediction problem, a model is usually given a dataset of *known data* on which training is run (*training dataset*), and a dataset of *unknown data* (or *first seen* data) against which the model is tested (*testing dataset*) [58]. There are different types of cross-validation but the one that is used for this thesis purpose is 10-fold cross-validation.

***10*-fold cross-validation:** In *10*-fold cross-validation, the original sample is randomly partitioned into *10* equal sized subsamples. Of the *10* subsamples, a single subsample is retained as the validation data for testing the model, and the remaining 9 (*10* – 1) subsamples are used as training data. To reduce variability, the cross-validation process is then repeated *10* times (the *folds*) using different partitions, with each of the *10* subsamples used exactly once as the validation data. The *10* results from the folds can then be averaged over the rounds to produce a

single estimation. The advantage of this method over repeated random sub-sampling is that all observations are used for both training and validation, and each observation is used for validation exactly once.

One of the advantages of using cross- validation is that CV will not only produce 1 model like the others (e.g. percentage split), but 10 models. Which means we have 10 accuracies and since 10-fold CV calculates the average accuracy, it gives a better indication of the true accuracy (it also provides some distribution). And also WEKA's CV performs class- balanced split of data into 10 CV portions which each have the same class distribution.

In summary, cross-validation combines (averages) measures of fit (prediction error) to derive a more accurate estimate of model prediction performance.

4.5 Performance Comparison Measures

Five distinct intrusion detection algorithms are tested on the KDD'99 dataset. These algorithms were selected so that they represent a wide variety of fields: Clustering approaches, decision trees, rule based models and function studying. An overview of the parameters used for each algorithm and their intrusion detection performance on the KDD'99 data set is given. WEKA provides a default value for all parameters. But in the case of this research some of the default parameter values are changed. These modifications are made in order to increase the performance of the algorithm and are based on trying on different values.

4.5.1 K-means clustering (K-M)

K-means clustering algorithm positions K centers in the pattern space such that the total squared error distance between each training pattern and the nearest center is minimized [22].
K-means clustering parameters in WEKA and their values used are specified below.

- ✓ Distance function = Euclidean Distance (the distance function to be used for instances comparison)
- ✓ dontReplaceMissingValues = True (missing values will be replaced globally with mean/mode)
- ✓ MaxIterations = 50 (the maximum number of iterations for the algorithm)
- ✓ numClusters= 10 (the number of clusters)

- ✓ Seed = 10 (The random number seed to be used). It is WEKA's default value. Other parameters which uses default values are preseveInstancesOrder and whether or not to display the standard deviation of the attributes

4.5.2 Decision Tree (J48)

Perhaps C4.5 algorithm which was developed by Quinlan [17] is the most popular tree classifier. Weka classifier package has its own version of C4.5 known as J48. J48 is an optimized implementation of C4.5 rev. 8. J48 is used for this study with the value of parameters as follows:

- ✓ binarySplits = False (Whether to use binary splits on nominal attributes when building the trees)
- ✓ confidenceFactor = 0.25 (The confidence factor used for pruning and 0.25 is selected since smaller values incur more pruning)
- ✓ minNumObj = 4 (The minimum number of instances per leaf)
- ✓ reducedErrorPruning =True (reduced-error pruning is used instead of C.4.5 pruning)
- ✓ seed =1 (The seed used for randomizing the data when reduced-error pruning is used)
- ✓ unpruned = False (It specifies whether pruning is performed)
- ✓ The value of other parameters such as the amount of data used to reduce error pruning and additional factors used for visualization taken from WEKA's default value.

4.5.3 Random Forest

This method combines bagging and the random selection of features to construct a group of decision trees with controlled variation. The selection of a random subset of features is a method of random subspace method, which is a way to implement stochastic bias.

Parameters and their values

- ✓ maxDepth =0 (The maximum depth of the trees and the value 0 is for unlimited)
- ✓ numFeatures =10 (The number of attributes to be used in random selection)
- ✓ numTrees =50 (The number of trees to be generated)
- ✓ seed = 1 (The random number seed to be used)

4.5.4 Fuzzy Logic

Fuzzy logic (or fuzzy set theory) is based on the concept of the fuzzy phenomenon to occur frequently in real world. Fuzzy set theory considers the set membership values for reasoning and the values range between 0 and 1. That is, in fuzzy logic the degree of truth of a statement can range between 0 and 1 and it is not constrained to the two truth values (i.e. true, false). A fuzzy system comprises of a group of linguistic statements based on expert knowledge. This knowledge is usually in the form of if-then rules. A case or an object can be distinguished by applying a set of fuzzy logic rules based on the attributes' linguistic value. All the parameters used are default value of WEKA.

4.5.5 Support Vector Machine (SVM)

SMO [39] implements the sequential minimal optimization algorithm for training a support vector classifier. Data points are randomly generated and contain actual attacks and normal usage patterns. Training is done using the RBF (radial bias function) kernel option; an important point of the kernel function is that it defines the feature space in which the training set examples will be classified [40]. This implementation globally replaces all missing values and transforms nominal attributes into binary ones. It also normalizes all attributes by default. (In that case the coefficients in the output are based on the normalized data, not the original data, this is important for interpreting the classifier.)

- ✓ C=1.0 (The complexity parameter)
- ✓ epsilon =The epsilon for round-off error (shouldn't be changed). =1.0e-12
- ✓ filterType .=Normalize training data (It determines how/if the data will be transformed)
- ✓ kernel = RBF (radial bias function) (The kernel to be used)
- ✓ numFolds = -1, The number of folds for cross-validation used to generate training data for logistic models (-1 means use training data)
- ✓ randomSeed =1(Random number seed for the cross-validation)
- ✓ toleranceParameter = 0.001 (It is the tolerance parameter and shouldn't be changed).
- ✓ Other parameters such as debug, buildLogisticModels and checksTurnedOff take the default value available in WEKA.

4.6 Performance Evaluation Result

The selected five intrusion detection algorithms which were built using training dataset in the previous section were also evaluated on the testing dataset. To compare performance of the models parameters such as, True Positive Rate (TP) which is the detection rate and False Positive Rate (FP) also known as False Alarm Rate of each algorithm on a specific attack category were recorded. These evaluation parameters are the most important criteria for the algorithms to be considered as the best models for the given attack category. Experiment results are given in the Table 4.6.

The first column in the table is the algorithms used to detect the intrusion, the second column contains the criteria used to measure the performance of the algorithms and the last four columns are the attack types in which those algorithms are required to detect. The highest of the accuracies between all algorithms is highlighted in bold.

Table 4.6: Performance Comparison of the five algorithm- TP and FP

Classifier	Metric	Dos	Probe	U2R	R2L
K-Means	TP (DR)	98.02	93.1	49.3	**72.04**
	FP (FAR)	2.01	0.39	0.7	3.9
Decision Tree	TP (DR)	97.2	81.9	14.2	5
	FP (FAR)	0.65	0.55	0.1	0.51
Random Forest	TP (DR)	**99.12**	98.27	**85.6**	59.87
	FP (FAR)	**0.04**	**0.11**	0.02	0.39
SVM	TP (DR)	96.0	72.8	17.09	8.53
	FP (FAR)	1.81	0.2	**0.01**	**0.19**
Fuzzy Logic	TP (DR)	94.01	**98.51**	65.6	37.1
	FP (FAR)	5.5	1.8	8.9	10.7

The evaluation result, in Table 4.6, shows that for a given attack category, certain algorithms demonstrate superior detection performance compared to others. The algorithm Random Forest Classifier has the highest detection rate for attack types such as DOS and U2R with 99.12% and

85.6% respectively, while Fuzzy Logic algorithm outperforms the others with its TP rate at 98.51% in detecting Probe attack. And for the case of R2L attacks, which is one of the rare attack types, K- means is the best classifier with 72.04% detection rate.

The models' simulation result of the other performance measurement, which is the false alarm rate, also shows different value for different attack types. Some algorithms have very low alarm rate (FAR) for specific attack type whereas others have higher FP rate regardless of their detection performance. Meaning that despite the fact that a specific algorithm out performs the other in the detection rate it might not do well on lowering its false alarm rate. Such is the case for Fuzzy Logic algorithm. It might be one of the algorithms to have higher performance in detecting most attack types but it is also one of the algorithms with higher FAR in all attack types. Random Forest algorithm has the lowest FP rate (false alarm rate) for both DoS and Probe attack types with 0.04% and 0.11%. Although Support Vector Machine (SVM) has low detection rate for U2R and R2L, it is the algorithm with lowest FAR for those attack types with 0.01% and 0.19% respectively.

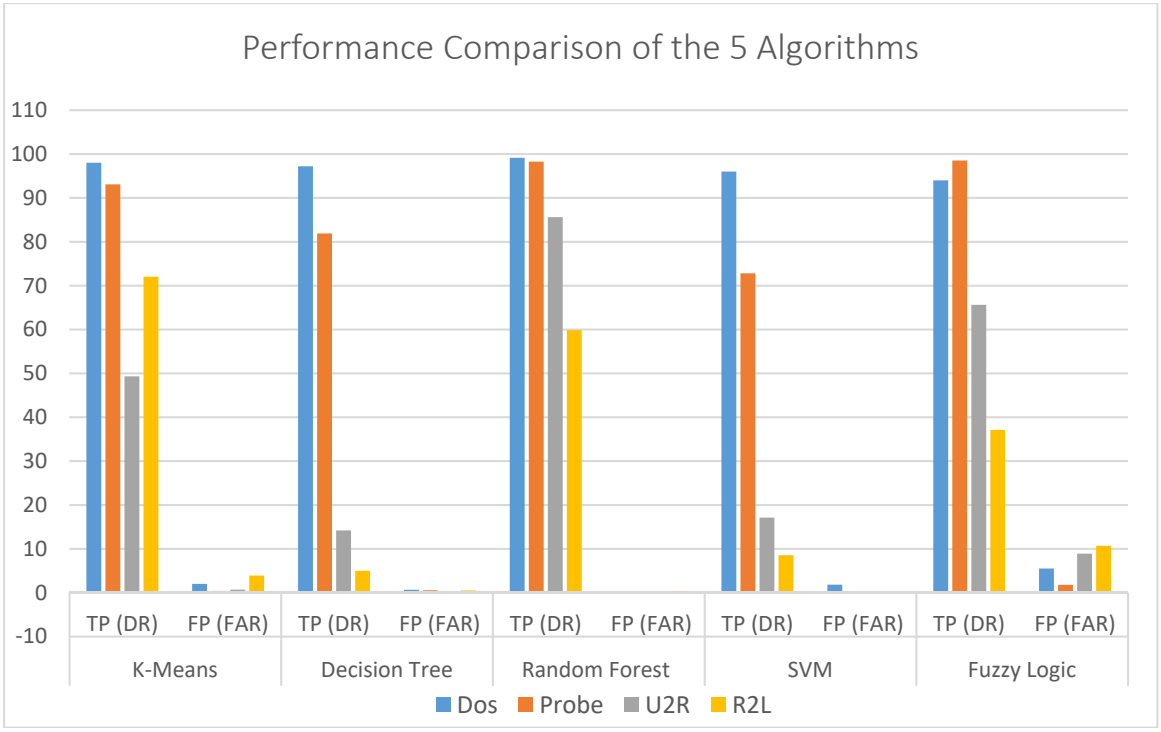

Figure 4.3: Performance comparison chart of selected algorithms

From the experiment result we can see that all algorithms were able to detect DoS attack types with high detection performance, the highest being Random Forest with 99.12% detection rate. Fuzzy Logic surpasses the others with 98.51% while detecting Probe, followed by Random Forest and K-means with 98.27% and 93.1% respectively. Rare attack types like U2R and R2L were only detected with higher detection rate by Random Forest and K-mean correspondingly. Fuzzy Logic algorithm has the second highest detection rate for U2R attack types but it has the highest false alarm rate compared with the others. Random Forest and K-means algorithms performed well in detecting most attack types with relatively low FAR including novel attacks found in U2R and R2L attack category. But most importantly the experiment illustrates no single algorithm could detect all attack categories with a high probability of detection and a low false alarm rate. This observation strengthens the thinking that the combination of different algorithms should be used to deal with different types of network attacks.

Chapter 5

Proposed Hybrid Algorithm

Our aim is to design and develop a Hybrid Intrusion Detection Algorithm (HIDA) that would be more accurate, low in false alarms, not easily deceived by small variations, capable of sniffing and detecting unknown attacks. Hence from above performance comparison evaluation set up we can conclude that a single algorithm cannot detect all intrusion types with exceptional performance. Some might exceed in detecting a certain attack type but lacks on the other. This reason leads for the proposal of a hybrid system. Since our main objective from the start is to be able to detect unknown attack types like U2R and R2L with higher detection capability and lower false rate, we selected two algorithms which show this potential.

The proposed hybrid algorithm, which uses a different approach for generating a forest and detecting an intrusion, is based on K-means clustering and a Random forest classifier and is called the K-means Clustering and Random Forest based hybrid Intrusion Detection Algorithm. Unlike other Random Forest generation methods, KRHA uses clustering as a pre-processing procedure and the random forest is generated according to the result of this clustering.

The models which made up the proposed hybrid model are two very different algorithms. One is a clustering method that uses unsupervised data and the other is a random tree creation method which is trained using supervised data. This difference can actually helps in improving the overall detection rate as the drawbacks of one technique might be solved by the other one. Besides having completely different approach what is common for both techniques is that they can process huge amount of data without affecting their performance and are both capable of finding out ignored and hidden information.

To explain the proposed hybrid intrusion detection algorithm in detail, we first define the two algorithms; K-means and Random Forest. The detailed working principles of both K-means and Random forest algorithms are already given. So we will just give a summary of both.

K-means Clustering: is a data exploration technique that attempts to find groups of data based on similar characteristics to separate data objects into meaningful and reasonable groups [21]. In order to measure these similarities between data objects, distance metric plays an important role. While many distance measures exist, Euclidean distance is one of the most commonly used and easy metric, which is why it was also used here.

The objects in a cluster are more similar (minimum distance) to each other than to objects in other clusters. Partitioning methods usually start with a random partition and refines partitions iteratively to constructs different groups by maximizing the homogeneity within the cluster by minimizing the square error.

Random Forests: is an ensemble of un-pruned classification or regression trees. It is unsurpassable in accuracy among the current data mining algorithms, especially for large datasets with many features [25]. Random Forest algorithm generates many classification trees. Each tree is constructed by a different bootstrap sample from the original data using a tree classification algorithm.

After the forest is formed, a new object that needs to be classified is put down each of the tree in the forest for classification. Each tree gives a vote that indicates the tree's decision about the class of the object. The forest chooses the class with the most votes for the object.

5.1 Hybrid Model Scheme

In this section, we describe the proposed framework of the hybrid model.

Figure 5.1 shows the proposed K-means Clustering and Random Forest Based Hybrid Intrusion Algorithm scheme. It uses two different intrusion detection models and also has two phases which are described below.

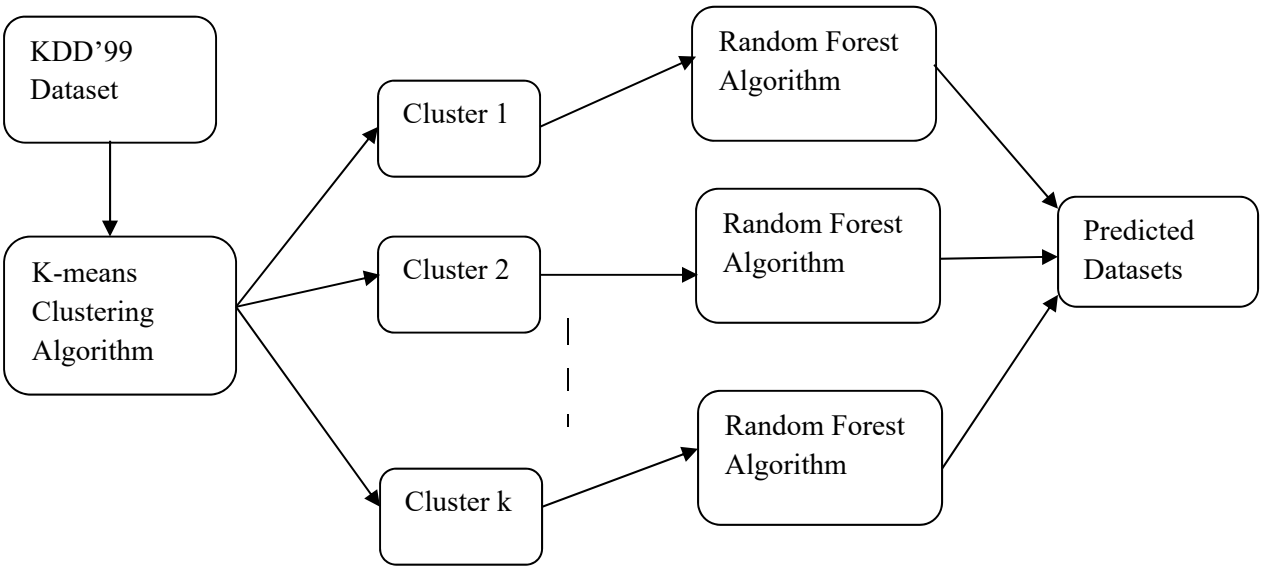

Figure 5.1: Model Architecture

As shown in Figure 5.1, the KDD'99 data set is given as an input to the K-means clustering algorithm. Using this algorithm the data are grouped into different clusters such that data within a cluster are more similar to each other than data found in another cluster. Dataset partitioning is done repetitively until the model is stable. The data in each cluster are then taken as training sets upon which the Random Forest is built using random sampling. After the hybrid model is built, a given KDD'99 test data will be classified using both classifiers one after the other. The Random Forest used for final class prediction of the testing data depends on the cluster the data fails into.

5.1.1 Training Phase Scheme

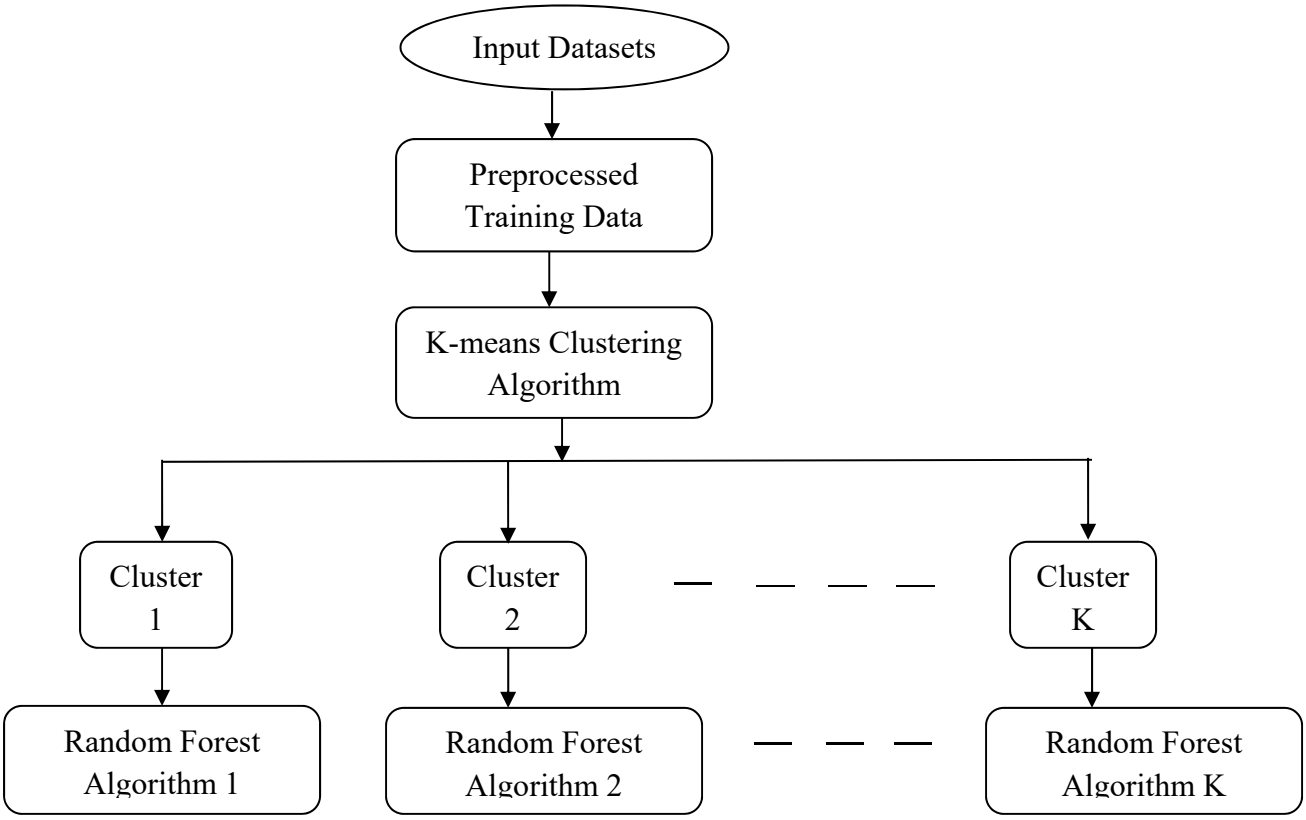

Figure 5.2: Training Phase Scheme

In training phase, as Figure 5.2 shows, the first step is to perform the preprocessing on the input data (it is done using WEKA), which includes removing duplicates, replacing missing values and processing all the data formats available on the dataset such that the classification models will be able to handle the data. The first algorithm to be trained is the K-means clustering algorithm using unsupervised data by removing the label attribute from each instance in the dataset. After the training, the K-means model is built. This model creates K number of clusters and groups the training data using Euclidian distance metric.

The next step is to build the Random Forest algorithm model by using training data found in each clusters created while building the K-means. Unlike the K-means algorithm the Random Forest is trained on supervised data. This is accomplished by restoring the removed label attribute. This is a pattern builder phase, which is going to be used in the intrusion (testing) phase for intrusion detection.

The choice of K-means clustering as a first classifier for the hybrid model is because it uses unsupervised learning to cluster instances. Given a collection of unlabeled data it discovers structures or patterns of data behaviors and finds their deviations to the problem without any prior knowledge. Consequently, the algorithm is able to classify instances without being biased by its class label which makes it ideal first classifier. But since the behavior of an attack is dynamic and always changing, a clusterer may sometimes misclassify a data and group it to the wrong cluster. That is when the Random Forest model comes in. It is used as a second classifier to further insure data are classified correctly. Unlike K-means, Random Forest is trained using supervised learning.

5.1.2 Testing Phase Scheme

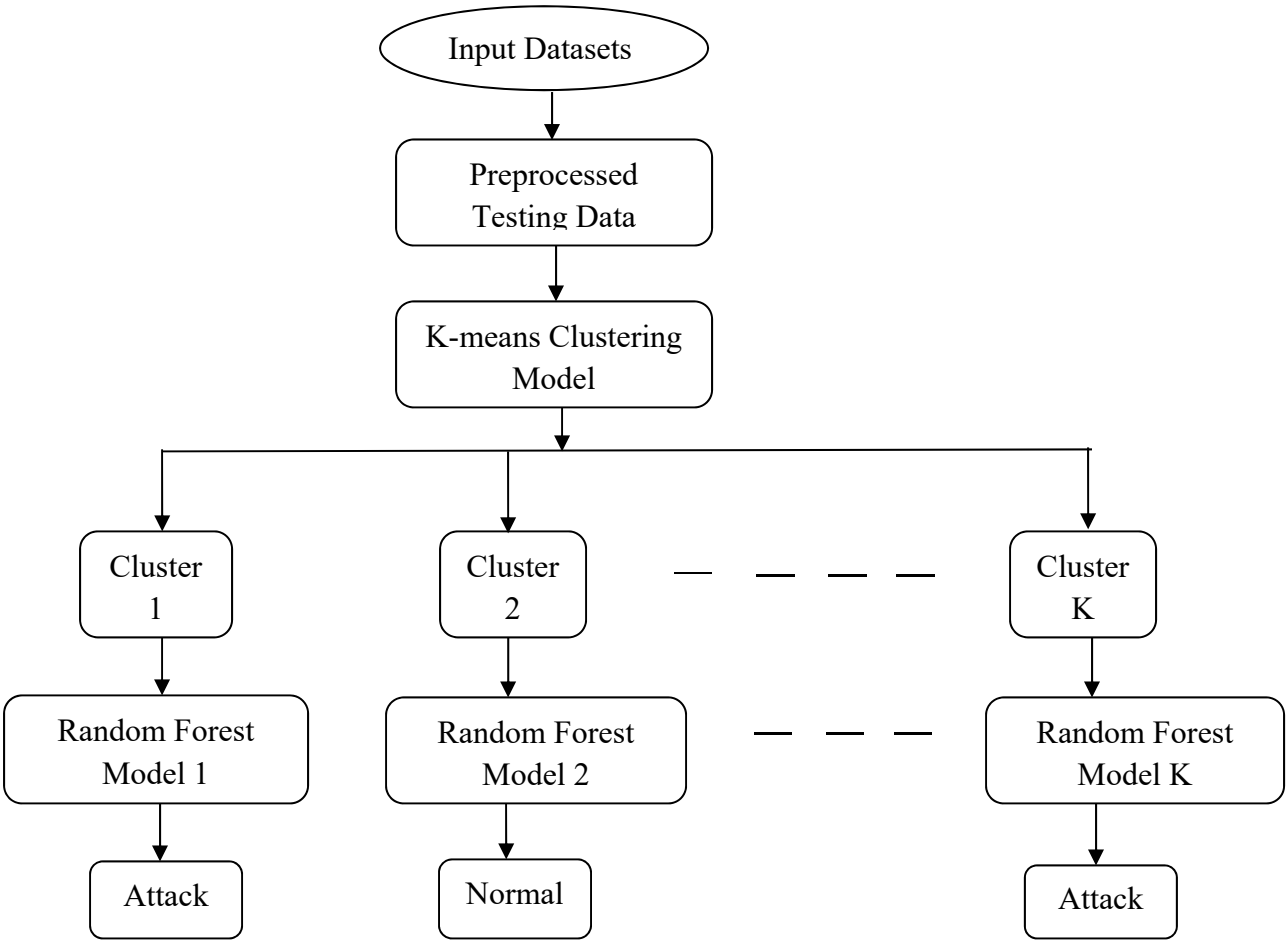

Figure 5.3: Testing Phase Scheme

Figure 5.3, illustrates intrusion detection phase, which is also called testing phase. When the test data go down through K-means for classification, the clustering model first evaluates each incoming test instance and groups them to their closest cluster by calculating the level of similarity it has with instances that belong in each cluster using a distance metric. For each record the model calculates the clustering statistics and stores the cluster assignment using an array. The cluster assignment array carries the number which points out to the cluster a specific instance belongs to. This computation is done to help us identify to which Random Forest that instance will be sent out to, since we generated several forests based on the created clusters.

After classifying a test instance to a cluster using K-means, the instance also went another classification process using the Random Forest which was built by training records that belong to a particular cluster the give test instance also now belongs.

To classify this received test data record, the Random Forest Classifier first allows all the decision trees in the forest to classify the record by using tree recursion and testing the relevant variable at each node. Each tree then returns the class membership probabilities for the given test instances. If the instance is classified as intrusion, each tree will specify type of the attack. Majority vote is used in order to obtain the predicted class as the final answer.

At the end, we process the classifier's prediction for an instance and then update all the statistics about a classifiers performance for the current test instance and additional information already gathered.

5.2 Hybrid Model Implementation

We have built and implemented the model in WEKA using java programming language on NetBeans tool. We have used a standard dataset, KDD'99 intrusion detection system dataset from [DARPA initiative program], which is explained in the dataset section. We have separated the implementation into two stages, training and testing. During the training phase, the classification models are constructed. And during the testing stage, identification of intrusions (attacks) and normal data are taken place. To implement this proposed hybrid model we have trained and tested the models using the 10-fold cross validation testing option. The propose hybrid algorithm is built by implementing the following pseudo code. The pseudo code of the hybrid model stated below is for both the training and testing phase.

Pseudo code of Proposed Model

For the Training Phase

Algorithm: Build KRHA K-Means Clustering

Input: K- the number of clusters and R the records of the dataset.

Output: A KRHA K-Means clusters

Method:

(1) Identify the number of clusters

(2) Randomly choose K objects and make them the K cluster centroids

(3) Do

(4) For each record in R

(5) Calculate distance between each cluster centroid and the record using Euclidean distance.

(6) Assign the record to the cluster that has the minimum distance.

(7) End for loop

(8) Recalculate the positions of the K centroids

(9) While records assignment to clusters do not change (while the centroids no longer move or change)

(10) End function

For each cluster:

Algorithm: Build KRHA Random Forest

Input: The training data D from the cluster this Forest belong to, the attributes_available p, no of trees in the forest B, no of attributes to be selected m_{sze}

Output: A KRHA Random Forest (An ensemble of trees $\{Tb\}_1^B$)

Method:

(1) For each tree T_b (b = 1 to B)

(2) Draw a bootstrap sample Z* of size N from the training data.

//Grow a Random-Forest Tree Tb to the bootstrapped data

(3) Create a node N.

(4) If data in D has only one record

(5) return N as a leaf node with record class.

(6) If all records in D have same target class

(7) return N as a leaf node with target class.

(8) If attributes_available is empty

(9) return N as leaf node with maximum target class for the records.

(10) Select m attributes at random from the available p attributes.

(11) Get best_attribute (D, attributes_selected (m)). // Pick the best attribute/split-point among the selected (m) by calculating importance of variable

(12) Split the records based on best_attribute (best_attribute, D)
//for each split, grown a subtree by calling the //Grow Random Forest function

(13) For each split D_i of D on best_attribute attach a new node returned by growRandomForest_Tree (split records D_i , attributes_available)

(14) end for

(15) end function

For the Testing Phase:

(1) For each testing instance

(2) Calculate the distance between the testing instance and each cluster centroid.

(3) Identify the nearest cluster to the testing instance

(4) Apply the Random Forest for the testing instance based on the nearest cluster Random Forest.

(5) Let Cb(x) be the class prediction of tree T_b // T_b is the b_{th} tree in the Random Forest

(6) Then $C_{rf}^{B}(x)$ =majority vote$\{C_b(x)\}_1^B$.

(7) end for

5.3 Results and Discussion

To implement and prove the proposed model, we used Weka software as our support and implementation tool for IDS. Any research work should be verified with some form of experiment using data. Specifically, in the field of Intrusion Detection, testing plays a vital role. To fulfill the above requirements and also to obtain proof of our concept, we tested the proposed algorithm using the same dataset we used to evaluate the previous five algorithms, which is the KDD'99 datasets.

In order to compare the result of the newly designed hybrid algorithm with those five algorithms, which are previously evaluated, the same benchmarks were used for the experimental set up. So, both evaluation of the five algorithms and implementation and testing of the hybrid algorithm, were carried out in the same platform.

After building the k-means algorithm, we have collected data from each cluster then feed each one of them into the Random Forest algorithm to build the classifier model. Since the k-means algorithm uses unsupervised data, it does not consider the data type (whether it is normal or attack data type) to cluster the training datasets, so data are clustered based on only other characteristics. This is advantageous since the training of the algorithm will not be affected by class labels. When the building of both models is finished we used the validating datasets which are the testing datasets for classification. A single data will go down through the k-means cluster model and be grouped to its closest cluster and then it will go down through several random trees which were built under this cluster for classification. The final decision is obtained using majority vote of the forest.

Behavior of an attack is not constant and always changing. Sometimes a clusterer might group a single data in wrong cluster by considering it as something else. Therefore, by combining both techniques, the drawbacks of each other will be complemented.

The table showing the classification accuracy for various algorithms including the proposed hybrid algorithm is given in Table 5.1. The performance table shows that the detection rate of the hybrid model is comparatively higher than all other systems for most attacks.

Table 5.1: Detection result of selected algorithm and hybrid model

Classifier	Metric	Dos	Probe	U2R	R2L
Hybrid	TP (DR)	99.04	**99.06**	**89.79**	**78.63**
	FP (FAR)	**0.02**	**0.07**	0.54	3.95
K-Means	TP (DR)	98.02	93.1	49.3	72.04
	FP (FAR)	2.01	0.39	0.7	3.9
Decision Tree	TP (DR)	97.2	81.9	14.2	5
	FP (FAR)	0.65	0.55	0.1	0.51
Random Forest	TP (DR)	**99.12**	98.27	85.6	59.87
	FP (FAR)	0.04	0.11	0.02	0.39
SVM	TP (DR)	96.0	72.8	17.09	8.53
	FP (FAR)	1.81	0.2	**0.01**	**0.19**
Fuzzy Logic	TP (DR)	94.01	98.51	65.6	37.1
	FP (FAR)	5.5	1.8	8.9	10.7

With the exception of DoS which is classified better by Random Forest with an accuracy of 99.12%., all the other attack types are classified with higher detection rate by the proposed hybrid algorithm. KRHA algorithm classifies Probe, U2R and R2L datasets with 99.06 %, 89.79% and 78.63% accuracy respectively. This is an improvement from Fuzzy Logic which has high detection rate for probe with 98.51% and Random Forest for U2R with 85.6% and K-means clustering algorithm for R2L with 72.04% detection rate.

When we evaluate the algorithms based on their false alarm rate, KRHA has the lowest rate for DoS and Probe with 0.02 and 0.07 respectively. But while the hybrid model manages to identify rare attack types such as U2R and R2L with higher detection rate, it is not the algorithm with the lowest false positive rate when compared to the other single algorithms. Its FP rate is 0.54% for U2R and 3.95% for R2L attacks. These results illustrate that for U2R attack type, the proposed hybrid model able to improve the FP rate from the K-means and Fuzzy Logic which have 0.7% and 8.9% FP rate correspondingly but it is still behind Random Forest (0.02% FP rate) and SVM (0.01% FP rate).

And regarding R2L attack type the proposed model's false alarm rate may be only slightly higher than K-means algorithm but when it is compared to other algorithms such as Random Forest, SVM and Decision tree the model also exhibits a much higher total false alarm rate. It is only lower than Fuzzy Logic algorithm, and Fuzzy logic is an algorithm that has high false positive rate for most of the attack types. In the future, we will try to bring down the false positives rate for both U2R and R2L.

The result shows that the R2L attack performance is satisfactory because the hybrid was able to detect most of the attacks with higher detection rate. Which means the model was able to detect new (rare) attacks. But it was also interesting to note that during the experimental stage U2R and R2L attacks performance was relatively less for most algorithms, one of the reason for this was because in the dataset used for the training contains more "smurf" and "neptune" attack types than other attack. This was why most algorithms were able to detect DoS with higher detection rate.

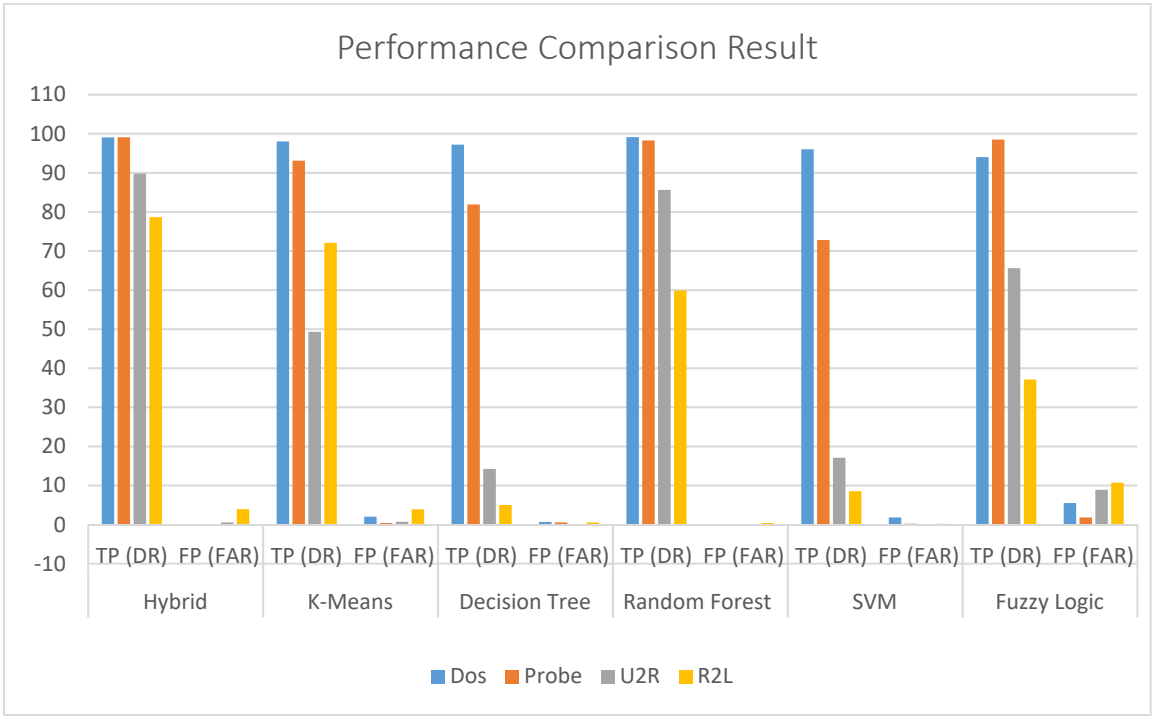

Figure 5.4: Performance comparison chart of all algorithms

Moreover, the experiment shows that the hybrid model detection performance for Dos was slightly lower than Random Forest which has the highest rate among the others. This is due to the presence of the k-means algorithm, which is unsupervised systems, in the hybrid model. Some attacks (*e.g.,* DoS) produce a large number of connections, which may undermine an unsupervised detection system. This is another subject that will be considered for future work by also focusing on optimization of the k-means cluster.

In general, when comparing the hybrid of K-means clustering and Random Forest with traditional single algorithms applied previously on the same dataset, integrating k-means clustering with Random Forest actually enhance the accuracy of intrusion detection and also was able to detect novel intrusions found in KDD'99 Dataset even if it still needs improving in its false alarm rate. The proposed hybrid algorithm (KRHA) combines the advantages of both supervised and unsupervised detection approaches, which operates blindly (i.e. not taking the class into account) over the data, and manages to improve the accuracy of the system significantly, when compared to the basic single algorithm systems.

Chapter 6

Conclusion and Future Work

Computers have become an essential component of our daily lives. The World Wide Web has transformed the world into a global village. Everyday there are millions of transactions on the Internet. As long with the rapid growth of the computers that are interconnected, the crime rate against those computers has also increased with an exponential pace. Those crimes are cyber attacks as a result of crackers exploiting flaws in internet protocols, operating system and application software. Cyber crimes like compromised server, phishing and sabotage of privacy information has also increased in the recent past. It need not be a massive intrusion; instead a single intrusion can result in loss of highly privileged and important data. Consequently the necessity for protecting those networked systems has also increased.

Several protective measures such as firewall have been put in place to check the activities of intruders, but those measures could not guarantee the full protection of the system. Hence, there is a need for a more dynamic mechanism, like intrusion detection system (IDS) as a second line of defense.

Literature survey indicates that, for intrusion detection, most researchers employed a single algorithm to detect multiple attack categories with dismal performance in some cases. And the uncertainty to explore if other intrusion detection algorithms can demonstrate better performance compared to the ones already employed constitutes the motivation for the study reported herein. So this paper evaluates performance of a comprehensive set of intrusion detection algorithms such as k –means clustering, Decision Tree, Random Forest, Fuzzy logic and SVM on four attack categories as found in the KDD'99 Cup dataset and proposes a new hybrid model.

Results of experimental study implemented to that effect indicated that certain classification algorithms perform better for certain attack categories: a specific algorithm is specialized for a given attack category. Consequently, a hybrid-classifier model, where selected algorithms have a promising detection performance with low FAR for rare attack types, was built. The hybrid model designed is called K-means and Random Forest based hybrid ID. This hybrid model is a composed of K-Means Clustering and Random Forest classification.

Empirical results obtained indicate that noticeable performance improvement was achieved for probing, user-to-root attacks, and root-to-local attacks by integrating K-means with Random Forest. This was due to the fact that the drawbacks of one technique can be solved by the other one. And also the use of data mining techniques helps to process huge amount of data and it was more useful to find out the ignored and hidden information.

The proposed model produces detection rates of 99.04%, 99.06% and 89.79% for DoS, Probe and U2R respectively. And 78.63% of detection rate for R2L attack type. Its false alarm rate recorded while detection those attacks are 0.02%, 0.07%, 0.54% and 3.95% for DoS, Probe, U2R and R2L correspondingly.

So over all when compared to the other five single algorithms, which only demonstrate higher detection ability on a single attack type, the hybrid model shows a better detection performance capability. But its false alarm rate still needs work to further decrease its value.

For future work, we will try to evaluate the model using real time traffic data in real environment including time elapsed to detect an attack. More attention will also be paid into decreasing the false alarm rate of the hybrid implementation even further by identifying and optimizing different issues that affect its performance. One of the biggest issues in any problem solving model is computational complexity. Complexity of an algorithm is a rough estimate of the number of steps that will be required on an instance and algorithm's inherent ability to solve a problem and provide a solution. Since the proposed hybrid model is a combination of two algorithms, its computational cost is higher and more complex. So we will also work on reducing its complexity by focusing on number of trees and clusters used, cluster structure sensitivity as well as clustering quality.

REFERENCE

[1] Ghorbani, Ali A., Wei Lu, and Mahbod Tavallaee. *Network intrusion detection and prevention: concepts and techniques*. Vol. 47. Springer Science & Business Media, 2009.

[2] Bruneau, Guy. "The history and evolution of Intrusion Detection." *SANS Institute* (2001).

[3] Siddiqui, Muazzam Ahmed. "High performance data mining techniques for intrusion detection." (2004).

[4] Jones, Anita K., and Robert S. Sielken. "Computer system intrusion detection: A survey." *Computer Science Technical Report* (2000): 1-25.

[5] Koziol, Jack. *Intrusion detection with Snort*. Sams Publishing, 2003.

[6] Bace, Rebecca Gurley. *Intrusion detection*. Sams Publishing, 2000.

[7] Kaushik, Sapna S., and P. R. Deshmukh. "Detection of attacks in an intrusion detection system." *International Journal of Computer Science and Information Technologies (IJCSIT)* 2.3 (2011): 982-986.

[8] Ioannis, Krontiris, Tassos Dimitriou, and Felix C. Freiling. "Towards intrusion detection in wireless sensor networks." *Proc. of the 13th European wireless conference*. 2007.

[9] Ye, Nong, and Xiangyang Li. "A scalable clustering technique for intrusion signature recognition." *Proceedings of 2001 IEEE workshop on information assurance and security*. 2001.

[10] Zhang, Jiong, and Mohammad Zulkernine. "Anomaly based network intrusion detection with unsupervised outlier detection." *Communications, 2006. ICC'06. IEEE International Conference on*. Vol. 5. IEEE, 2006.

[11] Patra, Mrutyunjaya Panda1and Manas Ranjan. "Evaluating machine learning algorithms for detecting network intrusions." (2009).

[12] Mahmood, Deeman Yousif, and Mohammed Abdullah Hussein. "Analyzing NB, DT and NBtree intrusion detection algorithms." *Journal of Zankoy Sulaimani-Part A (JZS-A)* 16.1 (2014): 1.

[13] Nageswararao, K., D. Rajya Lakshmi, and T. Venkateswararao. "Robust statistical outlier based feature selection technique for network intrusion detection." *International Journal of Soft Computing and Engineering* (2012): 2231-2307.

[14] Alemtsehay Adhanom, “Behavioral Based Cyber Network Intrusion Detection.” (2016).

[15] Zhang, Zheng, et al. "HIDE: a hierarchical network intrusion detection system using statistical preprocessing and neural network classification." *Proc. IEEE Workshop on Information Assurance and Security*. 2001.

[16] Li, T. *Behavioral clustering and statistical intrusion detection*. Diss. M. Sc. Thesis, Florida State University, 1997.

[17] Quinlan, J.L. “*C4.5 Program for Machine Learning*”, Morgan Kaufmam Publishers, Inc. 1993

[18] Gudadhe, Mrudula, Prakash Prasad, and Lecturer Kapil Wankhade. "A new data mining based network intrusion detection model." *Computer and Communication Technology (ICCCT), 2010 International Conference on*. IEEE, 2010.

[19] Duanyang Zhao,”Analysis of an intrusion detection system.” *Second International Conference on Security and Management,* 2012

[20] Goel, Radhika, Anjali Sardana, and Ramesh C. Joshi. "Parallel Misuse and Anomaly Detection Model." *IJ Network Security* 14.4 (2012): 211-222.

[21] Madhulatha, T. Soni. "An overview on clustering methods." *arXiv preprint arXiv:1205.1117* (2012).

[22] Alsabti, Khaled, Sanjay Ranka, and Vineet Singh. "An efficient k-means clustering algorithm." (1997).

[23] Teknomo, Kardi. "K-means clustering tutorial." *Medicine* 100.4 (2006): 3.

[24] Siddiqui, Mohammad Khubeb, and Shams Naahid. "Analysis of KDD CUP 99 dataset using clustering based data mining." *International Journal of Database Theory and Application* 6.5 (2013): 23-34.

[25] Ali, Jehad, et al. "Random forests and decision trees." *IJCSI International Journal of Computer Science Issues* 9.5 (2012): 272-278.

[26] Breiman, Leo. "Random forests." *Machine learning* 45.1 (2001): 5-32.

[27] Breiman, Leo. "Bagging predictors." *Machine learning* 24.2 (1996): 123-140.

[28] Ho, Tin Kam. "The random subspace method for constructing decision forests." *IEEE transactions on pattern analysis and machine intelligence* 20.8 (1998): 832-844.

[29] Tan, Pang-Ning, Michael Steinbach, and Vipin Kumar. "Classification: basic concepts, decision trees, and model evaluation." *Introduction to data mining* 1 (2006): 145-205.

[30] Peng, Wei, Juhua Chen, and Haiping Zhou. "An implementation of ID3-decision tree learning algorithm." *From web. arch. usyd. edu. au/wpeng/DecisionTree2. pdf Retrieved date: May* 13 (2009).

[31] Abraham, Ajith, et al. "D-SCIDS: Distributed soft computing intrusion detection system." *Journal of Network and Computer Applications* 30.1 (2007): 81-98.

[32] Laskov, Pavel, et al. "Learning intrusion detection: supervised or unsupervised?." *Image Analysis and Processing–ICIAP 2005* (2005): 50-57.

[33] Qi, Yanjun. "Random forest for bioinformatics." *Ensemble machine learning*. Springer US, 2012. 307-323.

[34] Bridges, Susan M., and Rayford B. Vaughn. "Intrusion detection via fuzzy data mining." *12th Annual Canadian Information Technology Security Symposium*. 2000.

[35] Luo, Jianxiong, and Susan M. Bridges. "Mining fuzzy association rules and fuzzy frequency episodes for intrusion detection." *International Journal of Intelligent Systems* 15.8 (2000): 687-703.

[36] Mendel, Jerry M. "Fuzzy logic systems for engineering: a tutorial." *Proceedings of the IEEE* 83.3 (1995): 345-377.

[37] Luo, Jianxiong. *Integrating fuzzy logic with data mining methods for intrusion detection*. MS thesis. Mississippi State University. Department of Computer Science., 1999.

[38] Karatzoglou, Alexandros, David Meyer, and Kurt Hornik. "Support vector machines in R." (2005).

[39] Cristianini, Nello, and John Shawe-Taylor. "An introduction to support vector machines." (2000).

[40] Osuna, Edgar, Robert Freund, and Federico Girosi. "Support vector machines: Training and applications." (1997).

[41] Mukkamala, Srinivas, Guadalupe Janoski, and Andrew Sung. "Intrusion detection using neural networks and support vector machines." *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*. Vol. 2. IEEE, 2002.

[42] Zhang, Lian-hua, et al. "Intrusion detection using rough set classification." *Journal of Zhejiang University-Science A* 5.9 (2004): 1076-1086.

[44] Adetunmbi, A. O., et al. "A Rough Set Approach for Detecting Known and Novel Network Intrusions." *Second International Conference on Application of Information and Communication Technologies to Teaching, Research and Administrations (AICTTRA, 2007) eds. Kehinde, LO, Adagunodo, ER and Aderounmu, GA, OAU, Ife*. 2007.

[45] Adetunmbi, A. O., et al. "A Data Mining Approach to Network intrusion Detection." (2007).

[46] Adetunmbi, Adebayo O., et al. "Network intrusion detection based on rough set and k-nearest neighbour." *International Journal of Computing and ICT Research* 2.1 (2008): 60-66.

[47] Axelsson, Stefan. "The base-rate fallacy and its implications for the difficulty of intrusion detection." *Proceedings of the 6th ACM Conference on Computer and Communications Security*. ACM, 1999.

[48] Amor, Nahla Ben, Salem Benferhat, and Zied Elouedi. "Naive bayes vs decision trees in intrusion detection systems." *Proceedings of the 2004 ACM symposium on Applied computing*. ACM, 2004.

[49] Bouckaert, Remco R., et al. "WEKA manual for version 3-6-13." *University of Waikato, New Zealand* (2010): 588-595.

[50] Lippmann, Richard, et al. "Analysis and results of the 1999 DARPA off-line intrusion detection evaluation." *Recent Advances in Intrusion Detection*. Springer Berlin/Heidelberg, 2000.

[51] Tavallaee, Mahbod, et al. "A detailed analysis of the KDD CUP 99 data set." *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*. IEEE, 2009.

[52] Lahre, Mr Kamlesh, et al. "Analyze different approaches for IDS using KDD 99 data set." *International Journal on Recent and Innovation Trends in Computing and Communication* 1.8 (2013): 645-651.

[53] Kruegel, Christopher, et al. "Stateful intrusion detection for high-speed network's." *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*. IEEE, 2002.

[54] Lee, Wenke, et al. "Toward cost-sensitive modeling for intrusion detection and response." *Journal of computer security* 10.1-2 (2002): 5-22.

[55] Dokas, Paul, et al. "Data mining for network intrusion detection." *Proc. NSF Workshop on Next Generation Data Mining*. 2002.

[56] Joshi, Mahesh V., Ramesh C. Agarwal, and Vipin Kumar. "Predicting rare classes: Can boosting make any weak learner strong?." *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2002.

[57] Weiss, Sholom M., and Tong Zhang. "Performance analysis and evaluation." *The handbook of data mining* (2003): 426-440.

[58] Mohri, Mehryar, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2012.

[59] MIT Lincoln Labs, 1998 DARPA Intrusion Detection Evaluation. Available on: http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/index.html, February 2008.

[60] KDD Cup 1999. Available on: http://kdd.ics.uci.edu/databases/kddcup_99/kddcup99.html, Ocotber 2007.

Appendix A

Description and Rule Structure of KDD 99 Intrusion Detection Dataset

Table A.1: List of KDD'99 features and description [60], Type C is continuous, while D is discrete

#	**Feature name**	**Description**	**Type**
1	duration	Length (# of seconds) of the connection	C
2	protocol type	Type of the protocol, e.g. tcp, udp, etc.	D
3	3 service	Network service on the destination, e.g., http, telnet, etc.	D
4	flag	Normal or error status of the connection	D
5	src_bytes	# of data bytes from source to destination	C
6	6 dst_bytes	# of data bytes from destination to source	C
7	land	1 if connection is from/to the same host/port; 0 otherwise	D
8	wrong_fragment	# of "wrong" fragments	C
9	urgent	# of urgent packets	C
10	10 hot	# of "hot" indicators	C
11	num_failed_logins	# of failed login attempts	C
12	logged in	1 if successfully logged in; 0 otherwise	D
13	num_compromised	# of compromised conditions	C
14	root_shell	1 if root shell is obtained; 0 otherwise	D
15	su_attempted	1 if "su root" command attempted; 0 otherwise	D
16	16 num_root	# of "root" accesses	C
17	num_file_creations	# of file creation operations	C
18	18 num_shells	# of shell prompts	C
19	num_access_files	# of operations on access control files	C
20	num_outbound_cmds	# of outbound commands in an ftp session	C

21	is_host_login	1 if the login belongs to the “hot” list; 0 otherwise	D
22	is_guest_login	1 if the login is a “guest’ login; 0 otherwise	D
23	count	# connections to the same host as the current one during past two seconds	C
24	srv_count	# of connections to the same service as the current connection in the past two seconds	C
25	serror_rate	% of connections that have “SYN” errors	C
26	srv_serror_rate	% of connections that have “SYN” errors	C
27	rerror_rate	% of connections that have “REJ” errors	C
28	srv_rerror_rate	% of connections that have “REJ” errors	C
29	same_srv_rate	% of connections to the same service	C
30	30 diff_srv_rate	% of connections to different services	C
31	srv_diff_host_rate	% of connections to different hosts	C
32	dst_host_count	Count of connection	C
33	dst_host_srv_count	Count of connections	C
34	dst_host_same_srv_rate	% of connections having the same destination host and using the same service	C
35	dst_host_diff_srv_rate	% of different services on the current host	C
36	dst_host_same_src_port_rate	% of connections to the current host having the same src port	C
37	dst_host_srv_diff_host_rate	% of connections to the same service coming from different host	C
38	dst_host_serror_rate	% of connections to the current host that have an SO error	C
39	dst_host_srv_serror_rate	% of connections to the current host and specified service that have SO error	C
40	dst_host_rerror_rate	% of connections to the current host that have an RST error	C
41	dst_host_srv_rerror_rate	% of connection to the current host specified service that have an RST error	C

Table A.2: Rule Structure for KDD Cup 99 Dataset

Rule No	Attack Description	Attack type
1	protocol =ICMP, service =ecr_i, src_byte =1032, flag =SF, host_count =255 Smurf	Smurf
2	protocol =tcp, service =private or ctf, flag =SO or SF, serror_rate = 1, srv_serror_rate =1	Neptune
3	protocol =ICMP, service =SF or SH, src_byte =8, same_srv_rate =1, srv_diff_host_rate = 1	Nmap
4	protocol =tcp, service =http, flag =SF or RSTFR, src_byte = 54540, dst_byte =7300 or 8314, same_srv_rate = 1, srv_count >=5	Back
5	protocol =UDP, service =private, flag = SF, src_byte = 1, dst_host_count = 255, dst_host_same_src_port_rate =1	Satan
6	protocol =UDP, service =SF, src_byte =28, wrong fragment =3, dst_host_count = 255	teardrop
7	protocol =icmp, service = eco_i, flag = SF, src_byte = 18, count =1, dst_host_count =1	ipsweep
8	protocol =TCP, service = Private or remote_ic, dst_host_count = 255, dst_host_srv_count =1	portsweep
9	duration =26 or 134, protocol =tcp, service =FTP or login, flag = SF, logged_in =1	ftp_write
10	protocol =tcp, service =telnet, flag =RSTO, src_byte = 125 or 126, dst_byte =179, hot =1, num_failed_login =1	guess_passwd
11	service = imap4, count <=4, dst_host_same_srv_rate =1, dst_host_srv_count < =1	Imap
12	service = tcp, flag =telnet or ftp_data, flag = SF, dst_host_srv_count <=3, dst_host_same_src_port_rate = 1	Multihop
13	duration =377or 299, service =tcp, flag = telnet, dst_host_count =255, dst_host_diff_srv_rate =0.01	Spy
14	protocol =tcp, service =ftp, flag = SF, src_byte >980, dst_byte >=1202, hot P3, dst_host_count = 255	warezclient
15	protocol =tcp, service =telnet or ftp_data, flag = SF, loggin_in	Buffer_overflow

	=1,dst_host_same_srv_rate = 1	
16	duration >=2, protocol =tcp, service =ftp or ftp_data, flag =SF, dst_host_count > 2, dst_host_srv_count >=1	warezmaster
17	protocol =tcp, service =telnet, flag =SF, dst_host_count = 1, dst_host_same_src_port_rate = 1	load module
18	duration >=25, protocol =tcp, service =telnet, flag =SF, logged_in =1, dst_host_srv_count <=2, dst_host_diff_srv_rate 6 0.07	Perl
19	protocol =tcp, service =telnet or ftp, flag =SF, dst_host_count = 255, dst_host_diff_srv_rate =0.02	root kit
20	protocol =tcp, service =finger, flag =SO, land = 1, srv_count = 2, dst_host_srv_serror_rate >=0.17	Land
21	protocol =ICMP, service =ecr_i, flag =SF, src_byte = 1480, wrong_fragment =1, dst_host_count = 255, dst_host_diff_srv_rate = 0.02	Pod
22	protocol =tcp, service =telnet, flag =SF, dst_host_count = 255, dst_host_serror rate =0.02	Phf

Appendix B

Sample java codes for KRHA Model Building

Sample java codes to build K-Means Model

```
 // remove the class attribute (if set) and build the clusterer
private Instances removeClass(Instances inst) {
   Remove af = new Remove();
   Instances retI = null;
   try {
    if (inst.classIndex() < 0) {
          retI = inst;
    } else {
          af.setAttributeIndices(""+(inst.classIndex()+1));
          af.setInvertSelection(false);
          af.setInputFormat(inst);
          retI = Filter.useFilter(inst, af);
    }
   } catch (Exception e) {
    e.printStackTrace();
   }
   return retI;
 }
```

==========

```
 // build the clusterer
 / * Generates a clusterer. Has to initialize all fields of the clusterer  that are not being set via options.
  * @param data set of instances serving as training data  // data without class
  * @exception Exception if the clusterer has not been  generated successfully
  */
```

```
public void buildClusterer(Instances data) throws Exception {
 m_Iterations = 0;
 if (data.checkForStringAttributes()) {
  throw  new Exception("Can't handle string attributes!");
 }
 m_ReplaceMissingFilter = new ReplaceMissingValues();
 Instances instances = new Instances(data);
 instances.setClassIndex(-1);
 m_ReplaceMissingFilter.setInputFormat(instances);
 instances = Filter.useFilter(instances, m_ReplaceMissingFilter);
 m_Min = new double [instances.numAttributes()];
 m_Max = new double [instances.numAttributes()];
 for (int i = 0; i < instances.numAttributes(); i++)
 {
  m_Min[i] = m_Max[i] = Double.NaN;
 }
 m_ClusterCentroids = new Instances(instances, m_NumClusters);
 int[] clusterAssignments = new int [instances.numInstances()];
 for (int i = 0; i < instances.numInstances(); i++)
 {
  updateMinMax(instances.instance(i));
 }
 Random RandomO = new Random(m_Seed);
 int instIndex;
 HashMap initC = new HashMap();
 DecisionTable.hashKey hk = null;
 // To find the centroids (M)
 for (int j = instances.numInstances() - 1; j >= 0; j--)
```

```
{
  instIndex = RandomO.nextInt(j+1);
  hk = new DecisionTable.hashKey(instances.instance(instIndex), instances.numAttributes(), true);
  if (!initC.containsKey(hk))
  {
        m_ClusterCentroids.add(instances.instance(instIndex));
        initC.put(hk, null);
  }
  instances.swap(j, instIndex);
  if (m_ClusterCentroids.numInstances() == m_NumClusters)
  {
        break;
  }
}
m_NumClusters = m_ClusterCentroids.numInstances();
int i;
boolean converged = false;
int emptyClusterCount;
Instances [] tempI = new Instances[m_NumClusters];
m_squaredErrors = new double [m_NumClusters];
m_ClusterNominalCounts = new int [m_NumClusters][instances.numAttributes()][0];
while (!converged)
{
  emptyClusterCount = 0;
  m_Iterations++;
  converged = true;
  for (i = 0; i < instances.numInstances(); i++)
  {
```

```
        Instance toCluster = instances.instance(i);
        int newC = clusterProcessedInstance(toCluster, true);
        if (newC != clusterAssignments[i]) {
          converged = false;
        }
        clusterAssignments[i] = newC;
    }
// update centroids
    m_ClusterCentroids = new Instances(instances, m_NumClusters);
    for (i = 0; i < m_NumClusters; i++)
    {
        tempI[i] = new Instances(instances, 0);
    }
    for (i = 0; i < instances.numInstances(); i++)
    {
        tempI[clusterAssignments[i]].add(instances.instance(i));
    }
    for (i = 0; i < m_NumClusters; i++)
    {
        double [] vals = new double[instances.numAttributes()];
        if (tempI[i].numInstances() == 0) {
          // empty cluster
          emptyClusterCount++;
        }
      else
      {
          for (int j = 0; j < instances.numAttributes(); j++) {
            vals[j] = tempI[i].meanOrMode(j);
```

```
          m_ClusterNominalCounts[i][j] =
            tempI[i].attributeStats(j).nominalCounts;
         }
         m_ClusterCentroids.add(new Instance(1.0, vals));
        }
}
     if (emptyClusterCount > 0)
{
        m_NumClusters -= emptyClusterCount;
  if (converged)
  {
    Instances[] t = new Instances[m_NumClusters];
    int index = 0;
    for (int k = 0; k < tempI.length; k++)
    {
      if (tempI[k].numInstances() > 0) {
        t[index++] = tempI[k];
      }
    }
    tempI = t;
  }
  else {
    tempI = new Instances[m_NumClusters];
  }
}
if (!converged) {
        m_squaredErrors = new double [m_NumClusters];
        m_ClusterNominalCounts = new int [m_NumClusters][instances.numAttributes()][0];
```

```
    }
   }
   m_ClusterStdDevs = new Instances(instances, m_NumClusters);
   m_ClusterSizes = new int [m_NumClusters];
   for (i = 0; i < m_NumClusters; i++)
   {
    double [] vals2 = new double[instances.numAttributes()];
    for (int j = 0; j < instances.numAttributes(); j++)
    {
          if (instances.attribute(j).isNumeric()) {
           vals2[j] = Math.sqrt(tempI[i].variance(j));
          } else {
           vals2[j] = Instance.missingValue();
          }
    }
    m_ClusterStdDevs.add(new Instance(1.0, vals2));
    m_ClusterSizes[i] = tempI[i].numInstances();
   }
Instance inst;
     for (i = 0; i < m_NumClusters; i++)
    {
        for (int j=0; j< instances.numInstances(); j++)
       {
         if (clusterAssignments[j]== i)
         {
         inst=tempI[clusterAssignments[j]].instance(j);
         //build Random Forest with cluster instance
         Classifier RFclassifier= new Classifier ();
```

```
        RFclassifier.buildClassifier(inst);
        }
      }
    }
 }
```

==========

```
/**
  * Updates the minimum and maximum values for all the attributes based on a new instance.
  * @param instance the new instance
  */
 private void updateMinMax(Instance instance) {
  for (int j = 0;j < m_ClusterCentroids.numAttributes(); j++)
  {
    if (!instance.isMissing(j))
    {
          if (Double.isNaN(m_Min[j]))
     {
           m_Min[j] = instance.value(j);
           m_Max[j] = instance.value(j);
         }
      else
      {
            if (instance.value(j) < m_Min[j]) {
              m_Min[j] = instance.value(j);
            }
        else
        {
             if (instance.value(j) > m_Max[j])
```

```
          {
                m_Max[j] = instance.value(j);
               }
           }
          }
    }
   }
 }
```

==========

```
/**
   * clusters an instance that has been through the filters
   * @param instance the instance to assign a cluster to
   * @param updateErrors if true, update the within clusters sum of errors
   * @return a cluster number
   */
  private int clusterProcessedInstance(Instance instance, boolean updateErrors) {
    double minDist = Integer.MAX_VALUE;
    int bestCluster = 0;
    for (int i = 0; i < m_NumClusters; i++) {
      double dist = distance(instance, m_ClusterCentroids.instance(i));
      if (dist < minDist) {
            minDist = dist;
            bestCluster = i;
      }
    }
    if (updateErrors) {
      m_squaredErrors[bestCluster] += minDist;
    }
```

```
   return bestCluster;
  }
==========================================================================
/**
   * Calculates the distance between two instances
   * @param test the first instance
   * @param train the second instance
   * @return the distance between the two given instances, between 0 and 1
   */
  private double distance(Instance first, Instance second) {
    double distance = 0;
    int firstI, secondI;
    for (int p1 = 0, p2 = 0;
            p1 < first.numValues() || p2 < second.numValues();) {
      if (p1 >= first.numValues()) {
            firstI = m_ClusterCentroids.numAttributes();
      } else {
            firstI = first.index(p1);
      }
      if (p2 >= second.numValues()) {
            secondI = m_ClusterCentroids.numAttributes();
      } else {
            secondI = second.index(p2);
      }
      double diff;
      if (firstI == secondI)
      {
            diff = difference(firstI, first.valueSparse(p1), second.valueSparse(p2));
```

```
            p1++; p2++;
        }
        else if (firstI > secondI)
        {
            diff = difference(secondI, 0, second.valueSparse(p2));
            p2++;
        }
        else
        {
            diff = difference(firstI, first.valueSparse(p1), 0);
            p1++;
        }
        distance += diff * diff;
      }
      return distance;
    }
```

```
================================================================================
```

```
/**
   * Computes the difference between two given attribute values.
   */
  private double difference(int index, double val1, double val2) {
    switch (m_ClusterCentroids.attribute(index).type()) {
    case Attribute.NOMINAL:
      // If attribute is nominal
      if (Instance.isMissingValue(val1) || Instance.isMissingValue(val2) || ((int)val1 != (int)val2)) {
            return 1;
      } else {
            return 0;
```

```
    }
   case Attribute.NUMERIC:
// If attribute is numeric
    if (Instance.isMissingValue(val1) ||
          Instance.isMissingValue(val2)) {
         if (Instance.isMissingValue(val1) &&
           Instance.isMissingValue(val2)) {
          return 1;
         } else {
          double diff;
          if (Instance.isMissingValue(val2)) {
           diff = norm(val1, index);
          } else {
           diff = norm(val2, index);
          }
          if (diff < 0.5) {
           diff = 1.0 - diff;
          }
          return diff;
         }
    } else {
         return norm(val1, index) - norm(val2, index);
    }
   default:
    return 0;
   }
 }
*******************************************************************************
```

Sample java codes to build Random Forest model

```
 /**
  * Builds a classifier for a set of instances.
  * @param instances the instances to train the classifier with and it is obtained from the cluster this
  * forest belongs to
  * @exception Exception if something goes wrong
  */
 public void buildClassifier(Instances data) throws Exception {
   m_bagger = new Bagging();
   RandomTree rTree = new RandomTree();
   // set up the random tree options
   m_KValue = m_numFeatures;
   if (m_KValue < 1) m_KValue = (int) Utils.log2(data.numAttributes())+1;
   rTree.setKValue(m_KValue);
   // set up the bagger and build the forest
   m_bagger.setClassifier(rTree);
   m_bagger.setSeed(m_randomSeed);
   m_bagger.setNumIterations(m_numTrees);
   m_bagger.setCalcOutOfBag(true);
   m_bagger.buildClassifier(data);
 }
```

==========

```
 /**
  * Bagging method.
  * @param data the training data to be used for generating thebagged classifier.
  * @exception Exception if the classifier could not be built successfully
  */
 public void buildClassifier(Instances data) throws Exception {
```

```
super.buildClassifier(data);
if (m_CalcOutOfBag && (m_BagSizePercent != 100)) {
  throw new IllegalArgumentException("Bag size needs to be 100% if " +
                                     "out-of-bag error is to be calculated!");
}
int bagSize = data.numInstances() * m_BagSizePercent / 100;
Random random = new Random(m_Seed);
boolean[][] inBag = null;
if (m_CalcOutOfBag)
  inBag = new boolean[m_Classifiers.length][];
for (int j = 0; j < m_Classifiers.length; j++)
{
  Instances bagData = null;
  // create the in-bag dataset
  if (m_CalcOutOfBag) {
        inBag[j] = new boolean[data.numInstances()];
        bagData = resampleWithWeights(data, random, inBag[j]);
  } else {
        bagData = data.resampleWithWeights(random);
        if (bagSize < data.numInstances()) {
          bagData.randomize(random);
          Instances newBagData = new Instances(bagData, 0, bagSize);
          bagData = newBagData;
        }
  }
  if (m_Classifier instanceof Randomizable) {
        ((Randomizable) m_Classifiers[j]).setSeed(random.nextInt());
  }
```

```
  // build the classifier
  m_Classifiers[j].buildClassifier(bagData);
}
// calc OOB error?
if (getCalcOutOfBag()) {
  double outOfBagCount = 0.0;
  double errorSum = 0.0;
  boolean numeric = data.classAttribute().isNumeric();
  for (int i = 0; i < data.numInstances(); i++)
  {
        double vote;
        double[] votes;
        if (numeric)
          votes = new double[1];
        else
          votes = new double[data.numClasses()];
        // determine predictions for instance
        int voteCount = 0;
        for (int j = 0; j < m_Classifiers.length; j++)
   {
          if (inBag[j][i])
            continue;
          voteCount++;
          if (numeric)
    {
            votes[0] = m_Classifiers[j].classifyInstance(data.instance(i));
          }
    else
```

```
 {
        double[] newProbs = m_Classifiers[j].distributionForInstance(data.instance(i));
   // average the probability estimates
   for (int k = 0; k < newProbs.length; k++) {
     votes[k] += newProbs[k];
   }
       }
     }
     // "vote"
     if (numeric)
{
       vote = votes[0];
  if (voteCount > 0) {
   vote  /= voteCount;    // average
  }
}
else
{
  if (Utils.eq(Utils.sum(votes), 0)) {
  } else {
   Utils.normalize(votes);
  }
       vote = Utils.maxIndex(votes);   // predicted class
}
     // error for instance
     outOfBagCount += data.instance(i).weight();
     if (numeric) {
       errorSum += StrictMath.abs(vote - data.instance(i).classValue()) * data.instance(i).weight();
```

```
          }
          else {
            if (vote != data.instance(i).classValue())
              errorSum += data.instance(i).weight();
          }
    }
    m_OutOfBagError = errorSum / outOfBagCount;
  }
  else {
    m_OutOfBagError = 0;
  }
 }
```

==========

```
/**
  * Builds classifier.
  */
 public void buildClassifier(Instances data) throws Exception {
  // Make sure K value is in range
  if (m_KValue > data.numAttributes()-1) m_KValue = data.numAttributes()-1;
  if (m_KValue < 1) m_KValue = (int) Utils.log2(data.numAttributes())+1;
  // Check for non-nominal classes
  if (!data.classAttribute().isNominal()) {
    throw new UnsupportedClassTypeException("RandomTree: Nominal class, please.");
  }
  // Delete instances with missing class
  data = new Instances(data);
  data.deleteWithMissingClass();
  if (data.numInstances() == 0) {
```

```
    throw new IllegalArgumentException("RandomTree: zero training instances or all " +
                                       "instances have missing class!");
  }
  if (data.numAttributes() == 1) {

throw new IllegalArgumentException("RandomTree: Attribute missing. Need at least " +
                                       "one attribute other than class attribute!");
  }
  if (data.checkForStringAttributes()) {
    throw new UnsupportedAttributeTypeException("Cannot handle string attributes!");
  }
  Instances train = data;
  // Create the attribute indices window
  int[] attIndicesWindow = new int[data.numAttributes() - 1];
  int j = 0;
  for (int i = 0; i < attIndicesWindow.length; i++) {
    if (j == data.classIndex())
      j++; // do not include the class
    attIndicesWindow[i] = j++;
  }
  // Compute initial class counts
  double[] classProbs = new double[train.numClasses()];
  for (int i = 0; i < train.numInstances(); i++) {
    Instance inst = train.instance(i);
    classProbs[(int) inst.classValue()] += inst.weight();
  }
  // Build tree
  Random rand = data.getRandomNumberGenerator(m_randomSeed);
```

```
  buildTree(train, classProbs, new Instances(data, 0), m_MinNum, m_Debug,
          attIndicesWindow, rand);
}
=====================================================
/**
 * Recursively generates a tree.
 */
protected void buildTree(Instances data, double[] classProbs, Instances header,
                    double minNum, boolean debug, int[] attIndicesWindow,
                    Random random) throws Exception {
  // Store structure of dataset, set minimum number of instances
  m_Info = header;
  m_Debug = debug;
  m_MinNum = minNum;
  // Make leaf if there are no training instances
  if (data.numInstances() == 0) {
    m_Attribute = -1;
    m_ClassDistribution = null;
    m_Prop = null;
    return;
  }
  // Check if node doesn't contain enough instances or is pure
  m_ClassDistribution = (double[]) classProbs.clone();
  if (Utils.sum(m_ClassDistribution) < 2 * m_MinNum
      || Utils.eq(m_ClassDistribution[Utils.maxIndex(m_ClassDistribution)], Utils
              .sum(m_ClassDistribution))) {
    // Make leaf
    m_Attribute = -1;
```

```
  m_Prop = null;
  return;
}
// Compute class distributions and value of splitting criterion for each attribute
double[] vals = new double[data.numAttributes()];
double[][][] dists = new double[data.numAttributes()][0][0];
double[][] props = new double[data.numAttributes()][0];
double[] splits = new double[data.numAttributes()];
// Investigate K random attributes
int attIndex = 0;
int windowSize = attIndicesWindow.length;
int k = m_KValue;
boolean gainFound = false;
while ((windowSize > 0) && (k-- > 0 || !gainFound)) {
  int chosenIndex = random.nextInt(windowSize);
  attIndex = attIndicesWindow[chosenIndex];
  // shift chosen attIndex out of window
  attIndicesWindow[chosenIndex] = attIndicesWindow[windowSize - 1];
  attIndicesWindow[windowSize - 1] = attIndex;
  windowSize--;
  splits[attIndex] = distribution(props, dists, attIndex, data);
  vals[attIndex] = gain(dists[attIndex], priorVal(dists[attIndex]));
  if (Utils.gr(vals[attIndex], 0))
    gainFound = true;
}
// Find best attribute
m_Attribute = Utils.maxIndex(vals);
double[][] distribution = dists[m_Attribute];
```

```
  // Any useful split found?
  if (Utils.gr(vals[m_Attribute], 0)) {
    // Build subtrees
    m_SplitPoint = splits[m_Attribute];
    m_Prop = props[m_Attribute];
    Instances[] subsets = splitData(data);
    m_Successors = new RandomTree[distribution.length];
    for (int i = 0; i < distribution.length; i++) {
      m_Successors[i] = new RandomTree();
      m_Successors[i].setKValue(m_KValue);
      m_Successors[i].buildTree(subsets[i], distribution[i], header, m_MinNum, m_Debug,
                                attIndicesWindow, random);
    }
    // If all successors are non-empty, we don't need to store the class distribution
    boolean emptySuccessor = false;
    for (int i = 0; i < subsets.length; i++) {
      if (m_Successors[i].m_ClassDistribution == null) {
        emptySuccessor = true;
        break;
      }
    }
    if (!emptySuccessor) {
      m_ClassDistribution = null;
    }
  } else { // Make leaf
    m_Attribute = -1;
  }
}
```

Appendix C

Sample java codes for KRHA Model Evaluation

```
for (int jj=0;jj<userTestT.numInstances();jj++)
      {
            int x=eval.evaluateClusterer(userTestT); // it returns best cluster
      for (i= 0; i<numCluster; i++ )
        {
              if (i=x)
          RFclassifier[i].processClassifierPrediction(inst.instance(jj), classifier, eval, predictions,
predInstances, plotShape,plotSize);
        }
                    if (outputPredictionsText) {
                      outBuff.append(predictionText(classifier, inst.instance(jj), jj+1));
                    }
               }
```

Sample java codes for evaluating K-Means Model

```
/**
  * Evaluate the clusterer on a set of instances.
  * Calculates clustering statistics and stores cluster assigments for the instances in
  * m_clusterAssignments
  * @param test the set of instances to cluster
  * @exception Exception if something goes wrong
  */
public int evaluateClusterer(Instances test) throws Exception {
   int i = 0;
   int cnum;
   double loglk = 0.0;
   double[] dist;
```

```
double temp;
int cc = m_Clusterer.numberOfClusters();
m_numClusters = cc;
int numInstFieldWidth = (int)((Math.log(test.numInstances())/Math.log(10))+1);
double[] instanceStats = new double[cc];
m_clusterAssignments = new double [test.numInstances()];
Instances testCopy = test;
boolean hasClass = (testCopy.classIndex() >= 0);
int unclusteredInstances = 0;
// If class is set then do class based evaluation as well
if (hasClass) {
  if (testCopy.classAttribute().isNumeric()) {
        throw new Exception("ClusterEvaluation: Class must be nominal!");
  }
  Remove removeClass = new Remove();
  removeClass.setAttributeIndices(""+(testCopy.classIndex()+1));
  removeClass.setInvertSelection(false);
  removeClass.setInputFormat(testCopy);
  testCopy = Filter.useFilter(testCopy, removeClass);
}
for (i=0;i<testCopy.numInstances();i++) {
  cnum = -1;
  try {
        if (m_Clusterer instanceof DensityBasedClusterer) {
          loglk += ((DensityBasedClusterer)m_Clusterer).
            logDensityForInstance(testCopy.instance(i));
          cnum = m_Clusterer.clusterInstance(testCopy.instance(i));
          m_clusterAssignments[i] = (double)cnum;
```

```
            } else {
              cnum = m_Clusterer.clusterInstance(testCopy.instance(i));
              m_clusterAssignments[i] = (double)cnum;
            }
        }
        catch (Exception e) {
            unclusteredInstances++;
        }
        if (cnum != -1) {
            instanceStats[cnum]++;
        }
      }
return m_classToCluster[i];
}
```

==========

*** Replaces all missing values for nominal and numeric attributes in a**

*** dataset with the modes and means from the training data.**

```
  public int clusterInstance(Instance instance) throws Exception {
    m_ReplaceMissingFilter.input(instance);
    m_ReplaceMissingFilter.batchFinished();
    Instance inst = m_ReplaceMissingFilter.output();
    return clusterProcessedInstance(inst, false);
  }
```

==========

```
/**
```

*** clusters an instance that has been through the filters**

```
   * @param instance the instance to be assigned to a cluster
   * @param updateErrors if true, update the within clusters sum of errors
```

```
  * @return the number of the assigned cluster as an interger
  * if the class is enumerated, otherwise the predicted value
  * @exception Exception if instance could not be classified successfully
  */
 private int clusterProcessedInstance(Instance instance, boolean updateErrors) {
  double minDist = Integer.MAX_VALUE;
  int bestCluster = 0;
  for (int i = 0; i < m_NumClusters; i++) {
    double dist = distance(instance, m_ClusterCentroids.instance(i));
    if (dist < minDist) {
          minDist = dist;
          bestCluster = i;
    }
  }
  if (updateErrors) {
    m_squaredErrors[bestCluster] += minDist;
  }
  return bestCluster;
 }
```

==========

Sample java codes for evaluating Random Forest

```
 /**
  * Process a classifier's prediction for an instance and update a
  * set of plotting instances and additional plotting info. plotInfo
  * for nominal class datasets holds shape types (actual data points have
  * automatic shape type assignment; classifier error data points have box shape type)
  * For numeric class datasets, the actual data points
  * are stored in plotInstances and plotInfo stores the error (which is
```

```
 * later converted to shape size values)
 * @param toPredict the actual data point
 * @param classifier the classifier
 * @param eval the evaluation object to use for evaluating the classifier on the instance to predict
 * @param predictions a fastvector to add the prediction to
 * @param plotInstances a set of plottable instances
 * @param plotShape additional plotting information (shape)
 * @param plotSize additional plotting information (size)
 */
public static void processClassifierPrediction(Instance toPredict,
                                                Classifier classifier,
                                                Evaluation eval,
                                                FastVector predictions,
                                                Instances plotInstances,
                                                FastVector plotShape,
                                                FastVector plotSize) {
 try {
   double pred;
   // classifier is a distribution classifier and class is nominal
   if (predictions != null) {
         Instance classMissing = (Instance)toPredict.copy();
         classMissing.setDataset(toPredict.dataset());
         classMissing.setClassMissing();
         Classifier dc = classifier;
         double [] dist =
           dc.distributionForInstance(classMissing);
         pred = eval.evaluateModelOnce(dist, toPredict);
         predictions.addElement(new
```

```
            NominalPrediction(toPredict.classValue(), dist, toPredict.weight()));
    } else {
          pred = eval.evaluateModelOnce(classifier,
                                        toPredict);
    }
    double [] values = new double[plotInstances.numAttributes()];
    for (int i = 0; i < plotInstances.numAttributes(); i++) {
          if (i < toPredict.classIndex()) {
            values[i] = toPredict.value(i);
          } else if (i == toPredict.classIndex()) {
            values[i] = pred;
            values[i+1] = toPredict.value(i);
            /* // if the class value of the instances to predict is missing then
            // set it to the predicted value
            if (toPredict.isMissing(i)) {
              values[i+1] = pred;
              } */
            i++;
          } else {
            values[i] = toPredict.value(i-1);
          }
    }
```

==========

```
/**
  * Evaluates the classifier on a single instance.
  * @param classifier machine learning classifier
  * @param instance the test instance to be classified
  * @return the prediction made by the clasifier
```

```
   * @throws Exception if model could not be evaluated
   * successfully or the data contains string attributes
   */
  public double evaluateModelOnce(Classifier classifier, Instance instance) throws Exception {
    Instance classMissing = (Instance)instance.copy();
    double pred = 0;
    classMissing.setDataset(instance.dataset());
    classMissing.setClassMissing();
    if (m_ClassIsNominal) {
          double [] dist = classifier.distributionForInstance(classMissing);
          pred = Utils.maxIndex(dist);
          if (dist[(int)pred] <= 0) {
            pred = Instance.missingValue();
          }
          updateStatsForClassifier(dist, instance);
    } else {
      pred = classifier.classifyInstance(classMissing);
      updateStatsForPredictor(pred, instance);
    }
    return pred;
  }
=================================================================================
/**
   * Calculates the class membership probabilities for the given test instance.
   * @param instance the instance to be classified
   * @return preedicted class probability distribution
   * @exception Exception if distribution can't be computed successfully
   */
```

```
  public double[] distributionForInstance(Instance instance) throws Exception {
    double [] sums = new double [instance.numClasses()], newProbs;
    for (int i = 0; i < m_NumIterations; i++) {
      if (instance.classAttribute().isNumeric() == true) {
            sums[0] += m_Classifiers[i].classifyInstance(instance);
      }
else {
            newProbs = m_Classifiers[i].distributionForInstance(instance);
            for (int j = 0; j < newProbs.length; j++)
              sums[j] += newProbs[j];
      }
    }
    if (instance.classAttribute().isNumeric() == true) {
      sums[0] /= (double)m_NumIterations;
      return sums;
    } else if (Utils.eq(Utils.sum(sums), 0)) {
      return sums;
    } else {
      Utils.normalize(sums);
      return sums;
    }
  }
=============================================================================
/**
   * Computes class distribution of an instance using the decision tree.
   */
  public double[] distributionForInstance(Instance instance) throws Exception {
    double[] returnedDist = null;
```

```
if (m_Attribute > -1) {
  // Node is not a leaf
  if (instance.isMissing(m_Attribute)) {
    // Value is missing
    returnedDist = new double[m_Info.numClasses()];
    // Split instance up
    for (int i = 0; i < m_Successors.length; i++) {
      double[] help = m_Successors[i].distributionForInstance(instance);
      if (help != null) {
        for (int j = 0; j < help.length; j++) {
          returnedDist[j] += m_Prop[i] * help[j];
        }
      }
    }
  } else if (m_Info.attribute(m_Attribute).isNominal()) {
    // For nominal attributes
    returnedDist = m_Successors[(int) instance.value(m_Attribute)].distributionForInstance(instance);
  } else {
    // For numeric attributes
    if (instance.value(m_Attribute) < m_SplitPoint) {
      returnedDist = m_Successors[0].distributionForInstance(instance);
    } else {
      returnedDist = m_Successors[1].distributionForInstance(instance);
    }
  }
}
// Node is a leaf or successor is empty?
if ((m_Attribute == -1) || (returnedDist == null)) {
```

```
    // Is node empty?
    if (m_ClassDistribution == null) {
      return null;
    }
    // Else return normalized distribution
    double[] normalizedDistribution = (double[]) m_ClassDistribution.clone();
    Utils.normalize(normalizedDistribution);
    return normalizedDistribution;
   } else {
    return returnedDist;
   }
  }
```

==========

```
  /**
   * Updates all the statistics about a predictors performance for the current test instance.
   * @param predictedValue the numeric value the classifier predicts
   * @param instance the instance to be classified
   * @throws Exception if the class of the instance is not set
   */
  protected void updateStatsForPredictor(double predictedValue,Instance instance)
      throws Exception {
   if (!instance.classIsMissing()){
// Update stats
    m_WithClass += instance.weight();
    if (Instance.isMissingValue(predictedValue)) {
          m_Unclassified += instance.weight();
          return;
    }
```

```
    m_SumClass += instance.weight() * instance.classValue();
    m_SumSqrClass += instance.weight() * instance.classValue() *instance.classValue();
    m_SumClassPredicted += instance.weight() * instance.classValue() * predictedValue;
    m_SumPredicted += instance.weight() * predictedValue;
    m_SumSqrPredicted += instance.weight() * predictedValue * predictedValue;
 if (m_ErrorEstimator == null) {
          setNumericPriorsFromBuffer();
    }
    double predictedProb = Math.max(m_ErrorEstimator.getProbability(
                                        predictedValue   - instance.classValue()), MIN_SF_PROB);
    double priorProb = Math.max(m_PriorErrorEstimator.getProbability(
                            instance.classValue()), MIN_SF_PROB);
    m_SumSchemeEntropy -= Utils.log2(predictedProb) * instance.weight();
    m_SumPriorEntropy -= Utils.log2(priorProb) * instance.weight();
    m_ErrorEstimator.addValue(predictedValue - instance.classValue(),instance.weight());
    updateNumericScores(makeDistribution(predictedValue),
makeDistribution(instance.classValue()), instance.weight());
   } else
    m_MissingClass += instance.weight();
 }
```