The importance of time: Modelling network intrusions with long short-term memory recurrent neural networks

Ralf C. Staudemeyer
University of Applied Sciences Schmalkalden

The importance of time: Modelling network intrusions with long short-term memory recurrent neural networks

Ralf Colmar Staudemeyer

A thesis submitted in fulfilment of
the requirements for the degree of

Doctor Philosophiae

in the **Department of Computer Science**
UNIVERSITY OF THE WESTERN CAPE

May 2012
Supervisor: Prof. Dr. Christian W. Omlin

for Jasper

Keywords: network intrusion detection, machine learning, time series, feature extraction, decision trees, naïve Bayes, Bayesian networks, support vector machines, recurrent neural networks, long short-term memory

Abstract

The importance of time: Modelling network intrusions with long short-term memory recurrent neural networks

Ralf Colmar Staudemeyer
PhD thesis, Department of Computer Science
University of the Western Cape

We claim that modelling network traffic as a time series with a supervised learning approach, using known genuine and malicious behaviour, improves intrusion detection. To substantiate this, we trained long short-term memory (LSTM) recurrent neural networks with the training data provided by the DARPA / KDD Cup '99 challenge. After preprocessing all features to improve information gain, we applied a number of intuitive steps to extract salient features, which resulted in the creation of a number of minimal feature sets that could be used for detecting attack classes. The preprocessed KDD Cup '99 data was then used to test the performance of five very common and well-known classifiers: Decision trees, naïve Bayes, Bayesian networks, feed-forward neural networks, and support vector machines. Our results show a performance comparable to the winning entries of the KDD Cup '99 challenge. Finally, we applied the LSTM recurrent neural network classifier to the preprocessed data using the minimal feature sets. Our results show that the LSTM classifier provides superior performance in comparison to other strong static classifiers trained. This is due to the fact that LSTM learns to look back in time and correlate consecutive connection records. For the first time ever, we have demonstrated the usefulness of LSTM networks to intrusion detection.

Declaration

I declare that *The importance of time: Modelling network intrusions with long short-term memory recurrent neural networks* is my own work, that it has not been submitted for any degree or examination in any other university, and that all the sources I have used or quoted have been indicated and acknowledged by complete references.

Ralf Colmar Staudemeyer

May 2012

Contents

List of Tables

List of Figures

Chapter 1
Introduction

Contents

In modern society, increasingly powerful technologies have encouraged widespread dependency on *information and communication technology* (ICT), which, in turn, has created a strong requirement for dependable ICT functionalities. At the same time, however, there are increasingly sophisticated and diverse threats to modern ICT systems; this calls for novel security mechanisms. Intrusion detection aims at identifying various kinds of malicious activities, and is now a strategic task of the highest importance in safeguarding computer networks and systems.

While traditional approaches to *intrusion detection systems* (IDSs) ([Lunt 1988], [Lunt 1993], [Mukherjee *et al.* 1994], and [McHugh 2001]) have proven to be efficient at detecting intrusions based on well-known parameters, they are completely ineffective in cases involving novel intrusions ([Kumar 1995], [Bejtlich 2004], and [Scarfone & Mell 2007]).

For example, at the time of writing, one major threat against the ICT systems is the *stuxnet worm*. Stuxnet came to light in June 2010, and was analysed in detail by [Falliere *et al.* 2010]. Circumventing classic intrusion detection, using an arsenal of previously unidentified vulnerabilities, it is capable of sabotaging industrial control systems. The risks involved are quite plain to see, considering that some of these affected control systems are used to operate motors in systems responsible for the enrichment of uranium in atomic plants.

Current commercial products offering anomaly detection are solely threshold-based, or make use of statistical measures ([Garcia-Teodoro *et al.* 2009]). These methods can model only relatively simple patterns, expressed in counts or distributions. Similar to signature-based approaches, this still limits their application to the detection of well-known and precisely defined attacks.

One potential solution to this limitation could be self-learning systems ([Mitchell 1997] and [Han & Kamber 2006]), which are capable of detecting previously unknown threats. This is due to their ability to differentiate between 'normal' and 'anomalous' traffic, by learning from monitored-network and host data. More precisely, machine learning methods can learn complex system behaviour. And by learning whole classes of normal traffic and attacks, trained classifiers have the potential to detect irregularities and previously unseen attacks. In addition, machine learning methods promise to provide a solution that can detect possible attacks in real-time, so that countermeasures can be taken in a timely manner.

At this time, because the implementation of machine learning methods in intrusion detection is in the very early stages of development, its practical applications are still quite limited ([Paxson 1999], [Staniford *et al.* 2002], [Prelude 2011]). In addition, there are a number of significant issues that need to be resolved ([Liao & Vemuri 2006]).

For instance, trained classifiers still suffer from a high number of misclassifications because intrusive activity is too rare [Axelsson 2000a]). Furthermore, powerful classifiers require significant resources for training and optimisation ([Lee *et al.* 2002]), which is still unrealistic for commercial deployment. This is why feature selection is a key element in advancing the use of machine learning methods in intrusion detection ([Lee & Stolfo 2000]).

We also need to ensure that the analysis of network data is not becoming overwhelmingly complex, in order to prevent an IDS unreasonably restraining the exchange of data between parties. Nowadays, the amount of data being transmitted via computer networks is huge. Any attempt to perform real-time traffic analysis on continuous streams of data necessitates a careful selection of information to be extracted. Communication sessions between hosts can be characterised by so-called *connection records* ([Lee & Stolfo 2000]).

Every connection record contains a number of features uniquely identifying the connection. Some features, such as the duration of the connection, bytes transferred in each direction, and the TCP/UDP ports used for communication, can be easily extracted. More complex features can, using increased bandwidth, become too expensive resource-wise to obtain ([Lee *et al.* 2002]). These features require packet inspection in order to extract information from the application layer. Other features, such as the number of users logged in, require information only available locally on the communicating systems.

Now, the challenge is to build intrusion detection systems based on artificial intelligence; systems that require a minimal input extracted with reasonable complexity from network traffic data, and with the highest possible effectiveness with regard to detecting novel threats to computer systems.

In this work, we investigate the application to network intrusion detection of a number of *static* machine learning methods, as well as that of *long short-term memory* (LSTM), a powerful *dynamic* classifier introduced by [Hochreiter & Schmidhuber 1996] and [Hochreiter & Schmidhuber 1997], and enhanced by [Gers *et al.* 1999] and [Gers *et al.* 2002]. In investigating the possibilities of data mining, we were especially interested in the effects of a reduced feature set on the network intrusion detection performance, using strong machine learning classifiers.

As representatives of static machine learning methods, we applied decision trees as introduced by [Quinlan 1986] and [Quinlan 1993], Bayesian learning as described in [John & Langley 1995] and [Heckerman *et al.* 1995], the neural network backpropagation algorithm as invented by [Werbos 1990] and [Rumelhart *et al.* 1994]), and support vector machines as developed by [Boser *et al.* 1992] and [Cortes & Vapnik 1995]) to the publicly available DARPA / KDD Cup '99 dataset ([Hettich & Bay 1999] and [DARPA 2011]). The KDD Cup '99 dataset consists of connection records with 41 features whose relevance for intrusion detection are not clear. This work documents experiments with different subsets of these features.

1.1 Motivation

Computer network attackers leave faint traces of their presence in network traffic. By carefully analysing this traffic, it is possible to trace malicious behaviour and identify attacks. Analysis can be done by scanning for (1) signatures of known attacks, or (2) abnormal traffic trends ([Debar *et al.* 1999], [Debar *et al.* 2000], [Axelsson 2000b]). Today, most Intrusion Detection Systems (IDSs) are still signature-based but products using anomaly-detection are starting to be more common ([Garcia-Teodoro *et al.* 2009]). Since systems using signature detection are only able to model attacks that have been carried out before and require frequent updates of their signatures. Anomaly-based intrusion detection systems are able to detect unknown attacks, but still suffer from high false alarm rates. For this reason, they require continuous supervision by highly qualified experts in order to successfully distinguish between false alarms and real attacks ([Scarfone & Mell 2007]).

This thesis is aimed at the development of intrusion detection systems that can detect known, as well as new attacks, with a high detection rate. Ideally, these systems should have a false alarm rate of 0%. But unfortunately, continuous infrastructure development of faster networks and faster computers make it increasingly difficult to observe huge amounts of network data in real-time. Furthermore, the continuous invention of new attacks, the use of variants of old attacks, and the exploitation of security flaws in network protocols and software by hackers and security experts, make it necessary to scan for malicious traffic with unknown patterns. This can only be done by using anomaly-based intrusion detection systems.

1.2 Premises

We assume that usage patterns can be learned. This will enable us to detect malicious behaviour buried deep in legitimate traffic. New machine learning methods look promising for detecting attacks with a very low profile over long periods of time. In this thesis, we will apply and compare the detection and failure rates of a variety of traditional machine learning methods to those of LSTM, a recurrent neural network architecture. According to our hypothesis, LSTM recurrent neural networks will outperform other machine learning methods previously used with intrusion detection data. This is because LSTM is specialised in time series learning and can keep information over very long periods of time.

This project is based on the following three premises:

1. Intruders will try to masquerade as genuine users, but their behavioural patterns will differ in some important aspects due to their specific objective; i.e. unauthorised access to resources such as networks, computers, data, etc.

2. We assume that usage patterns can be learned; however, static-pattern recognition for intrusion detection has had very limited success so far.

3. Intrusion patterns are buried deep in legitimate traffic with a very low profile over long time periods.

1.3 Research Questions

The central research question is whether modelling observed network traffic as a time series with known genuine and malicious behaviour improves intrusion detection.

This question implies the following sub-questions:

1. What are the salient features that need to be extracted from the data for modelling?

2. Can we model traffic containing thousands of records?

3. Can we model time series for which comparatively few examples exist, i.e. when the overwhelming majority of user patterns are genuine?

4. Can the modelling technique be generalised to detect *signatures* of variants of intrusions, even though no such data was seen in the training set? What performance can we achieve?

5. Can information from higher layers significantly improve our intrusion detection approach?

1.4 Technical Objectives

In order to answer these questions, we need to achieve the following technical objectives:

1. Identify features that are useful for discrimination.
2. Construct a model that allows discrimination between genuine and malicious user patterns from examples.
3. Use information from higher layers with time series data to measure the possible improvement of the intrusion detection rate and analyse the trade-off between cost and benefit.

1.5 Research Methodology

We arranged our research into the following steps:

1. Review related literature and studies.
2. Extract important features from the KDD Cup '99 intrusion detection dataset.
3. Train static machine learning methods, such as decision trees, Bayesian learning and neural networks, on the training sets and measure their performance.
4. Implement and test LSTM.
   (a) Train LSTM runs on training sets and measure performance.
   (b) Compare performance of static methods.
5. Test LSTM runs on variants of attacks for which LSTM was not trained.

1.6 Thesis Contributions

The major contributions presented in this thesis are as follows:

1. We present a number of data preprocessing steps which significantly improve the performance of machine learning classifiers on the KDD Cup '99 dataset in comparison to those directly using the original 41 attributes. After preprocessing, the majority of features show approximately 20% improved information gain for all traffic classes (see Figure 4.3 on Page 103, Figure 4.5 on Page 107, and Figure 4.6 on Page 108). Using the preprocessed features, the C4.5 decision tree classifier shows a comparable performance to the C5 decision tree classifier, which was the winning entry at the KDD Cup '99 challenge.

2. We present a salient feature subset derivation technique. Our custom feature selection algorithm is based on C4.5 decision tree pruning, combined with a biased backward elimination and forward selection approach. This process is supported by using heuristic domain knowledge to favour features that can easily be extracted from network traffic. Avoiding an exhaustive search through the whole feature space, this technique proves very effective at finding minimal feature sets with as few as 4–8 features.

3. We present extracted minimal feature sets for detecting all attacks and four individual classes of attacks (*denial-of-service*, *network probe*, *remote-to-local* and *user-to-root*) in two-class and multi-class classification. We present minimal feature sets with 11, 8 and 4 features for detecting all attacks with one trained classifier and minimal feature sets with 4–6 features for detecting individual attack classes. The C4.5 classifier shows, on all sets, a similar performance to using all features. Our minimal set for detecting all attacks consists of only 4 features.

   For detecting individual attacks, we present a 6-feature set for the detection network probes with 98.27% accuracy and 0.004 false positive rate, a 5-feature set for the detection of denial-of-service attacks with 98.11% accuracy and 0.004 false positive rate, a 6-feature set for the detection of remove-to-local attacks with 79.45% accuracy and <0.001 false positive rate, and a 5-feature set for the detection of user-to-root attacks with 99.96% accuracy and <0.001 false positive rate.

These sets are, by far, the smallest feature sets extracted on the given KDD Cup '99 feature dataset found in literature. In 'X-1' histograms, we show that, by using the C4.5 classifier, any further removal of remaining features leads to a remarkable degradation in performance in terms of incorrectly classified instances (see Figure 4.7 on Page 110, Figure 4.8 on Page 111, Figure 4.9 on Page 113, and Figure 4.10 on Page 114).

4. We present a detailed performance evaluation of five static classifiers on our preprocessed KDD Cup '99 dataset. Specifically, we evaluate *decision trees*, *naïve Bayes*, *Bayesian networks*, *feed-forward neural networks* and *support vector machines*. We compare the performance of these classifiers on all features and on the extracted minimal feature sets (see Table 5.6 on Page 127 and Table 5.10 on Page 135). We observe classification performance on all attacks and on individual attack classes (see Table 5.11 on Page 138, Table 5.12 on Page 140, Table 5.13 on Page 142, and Table 5.14 on Page 144).

   The results achieved are comparable to the best-performing entries of the KDD Cup '99 challenge, with the exception of the naïve Bayes classifier. We show that all tested classifiers benefit from preprocessing, and, observing the static classifier performance using the minimal feature sets and training on all attacks, we show that the performance of the C4.5 decision tree and the feed-forward neural network classifier is nearly unaffected by excessive feature reduction in terms of costs and incorrectly classified instances. Again, in terms of accuracy and costs, using these minimal feature sets, the performance is comparable to using all the features. However, we note that using 4 minimal features, only the decision tree classifier excels in performance. We suggest optional 8 and 11-feature minimal sets where all static classifiers show comparable performance.

   The performance of the static classifiers trained on individual attack classes show the following results in terms of accuracy, cost and false positive rate. For network probes trained with 6 minimal features, all classifiers show good performance. The best-performing classifiers are the decision tree with 98.52% accuracy and 0.004 false positive rate, and the neural network classifier with 98.81% accuracy and 0.005 false positive rate, with all other classifiers closely following. For denial-of-service attacks trained on 5 minimal features, the best-performing classifiers are again the decision tree with 98.11% accuracy and 0.003 false positive rate, and the neural network with 98.12% accuracy and 0.004 false positive rate.

The results for remote-to-local attacks trained on the 6-feature minimal set show a noticeable loss of performance for most classifiers. Here, we suggest sets with 14 and 18 features, where all classifiers show a similar performance to the performance achieved using the full feature set. The best-performing classifier on the 14-feature set is the decision tree classifier with 80.62% accuracy and <0.001 false positive rate. On detecting user-to-root attacks, all classifiers show a similar performance or improve using the 5-feature set. The best-performing classifier here is the decision tree classifier with 99.96% accuracy and <0.001 false positive rate.

A more detailed analysis of the neural network classifier performance on individual attack classes via AUC (area under ROC-curve) reveals that for most attacks this classifier can fully maintain performance after extensive feature reduction. The AUC values for the classification of the four attack classes in the test data remain or improve after feature reduction from 0.997 to 0.997 using all and 6 features (network probes), from 0.998 to 0.979 (denial-of-service attacks using all and 5 features), from 0.601 to 0.618 (remote-to-local attacks using all and 18 features), and from 0.707 to 0.984 (user-to-root attacks using all and 5 features) (see Figure 5.4 on Page 137, Figure 5.5 on Page 139, Figure 5.6 on Page 141, and Figure 5.7 on Page 143).

5. We present a performance evaluation of the strong *dynamic* classifier *long short-term memory* recurrent neural networks on our preprocessed KDD Cup '99 dataset. Again, we compare the performance of the classifier on all features and the minimal feature sets, and we observe classification on all attacks and individual attack classes.

   LSTM shows a better performance than all tested static classifiers.

Using all features and training on all attacks, LSTM shows the best results achieved. These results clearly outperform the results of all static classifiers, including the winning results of the KDD Cup '99 challenge. After feature reduction using 4 minimal features, LSTM still shows exceptional results.

The classifier comparison of all tested static and dynamic classifiers on detecting individual attacks shows that LSTM again outperforms all static classifiers. For denial-of-service attacks using our 5-feature minimal set, LSTM shows the best performance achieved with 99.78% accuracy, 0.0044 costs and 0.004 false positive rate. This also holds for network probes using our 6-feature set with 99.35% accuracy, 0.0065 costs and 0.004 false positive rate. Observing remote-to-local attacks, the best results are shown using a the 14 feature subset with 80.41% accuracy, 0.5568 costs and 0.039 false positive rate, although LSTM still outperforms the other neural network-based classifiers (support vector machine and feed-forward neural network) using the 6-feature minimal set. Detecting user-to-root attacks, the performance of LSTM is still acceptable with 99.93% accuracy, 0.0026 costs and <0.001 false positive rate, and on the same level as the best-performing decision tree classifier.

Summaries of test results for training all classifiers with individual attack classes using the minimal features is shown for denial-of-service attacks in Table 6.11 on Page 172, for network probes in Table 6.12 on Page 173, for 'r2l' attacks in Table 6.13 on Page 173) and for 'u2r' attacks in Table 6.14 on Page 174.

The maximum AUC values achieved for the classification of the four attack classes in the test data using all features and using the minimal feature sets are from 0.997 to 0.999 (network probes using all and 5 features), from 0.997 to 0.998 (denial-of-service attacks all and 6 features), from 0.877 to 0.883 (remote-to-local attacks using all and 14 features), and from 0.974 to 0.992 (user-to-root attacks using all and 5 features).

The strength of the LSTM classifier shows through when it is trained on high-frequency attacks that generate high volumes of consecutive records, such as denial-of-service attacks and network probes. On low-frequency attacks, such as remote-to-local and user-to-root attacks, the benefit of LSTM vanishes. We have demonstrated, for the first time ever, the successful application of LSTM recurrent neural networks to intrusion detection systems.

The minor contributions are:

1. a detailed analysis of the KDD Cup '99 dataset,
2. an extraction of custom training and test sets for feature extraction with improved attack distributions for rare attacks, and
3. a custom implementation of the long short-term memory classifier with a focus on large datasets and parallel processing, which we used for our experiments.

1.7 Thesis Overview

In Chapter 2, the basic concepts of network intrusion detection are introduced. We explain the steps an attacker carries out to compromise a system and cover the different main types of threats. Then we present the two different kinds of intrusion detection: Host and network intrusion detection. We explain the applied techniques for detecting intrusions and how to deal with classification errors. Chapter 3 covers different static and dynamic data mining methods, namely, decision trees, Bayesian learning, neural networks, support vector machine classifiers, recurrent neural networks and, specifically, long short-term memory recurrent neural networks.

In Chapter 4, we analyse the KDD Cup '99 datasets and their potential for feature reduction. First, we provide some additional background knowledge by presenting different performance metrics and attribute search strategies we successfully applied. Then, we present the data preprocessing steps and the feature selection process we developed for preparing the KDD Cup '99 data. Here, we introduce distribution histograms, scatter plots, information gain and decision tree pruning as supportive feature reduction tools. Using our custom feature selection process, we show how we can significantly reduce the number features in the dataset to a few salient features. We conclude by presenting minimal sets with 4–8 salient features for two-class and multi-class categorisation for detecting intrusions, as well as for the detection of individual attack classes; the performance using a static classifier compares favourably to the performance using all features available.

We start Chapter 5 with a presentation of a literature overview, summarising the results of a variety of learning algorithms, which were successfully applied to these datasets. Then, we test the data with the five static classifiers introduced in Chapter 3 and compare the results to the winning entries of the KDD Cup '99 challenge. Next, we apply the classifiers using the minimal feature sets and compare the experimental results.

In Chapter 6, we present our results of applying the dynamic LSTM recurrent neural network classifier to the KDD Cup '99 data. Then, we present the results of experiments using all features and using our minimal feature sets, and compare these to the results using the static classifiers. We show that the LSTM classifier outperforms the static classifiers for most of the tested traffic classification tasks. Furthermore, preprocessing and our custom feature selection process represents a significantly improved classification performance of all classifiers.

We conclude, in Chapter 7, with a summary of our results and suggestions for future directions.

Chapter 2

Network Intrusion Detection

Contents

2.1 Introduction

Intrusion detection systems (IDSs) are concerned with the automatic detection of manual or machine attacks on a computing system ([Axelsson 2000b]). In this chapter, we cover the most important background information in this field of research relevant to this thesis. We open with a brief summary of earlier work.

The first generation of intrusion detection systems consisted of administrators sitting in front of their monitors, manually observing network and system activities, in an attempt to identify policy violations or other inappropriate use of the system. However, with the increasing amount of information collected, manual observation of audit logs became an unfeasible task ([Kemmerer & Vigna 2002]).

Efforts to automate this process reach back to concepts presented by [Anderson 1980] and [Denning 1987], and early implementations were developed during the late 1980s (see [Smaha 1988], [Jagannathan *et al.* 1993], and [Porras & Neumann 1997]). The history of intrusion detection is summarised in surveys at different points in time by [Lunt 1988], [Lunt 1993], [Mukherjee *et al.* 1994], and [McHugh 2001]. The related taxonomy of different approaches was devised by [Debar *et al.* 1999], [Debar *et al.* 2000] and [Axelsson 2000b].

Since intrusion detection proved to be a computationally intensive task, these early systems worked with a time delay so as to avoid interfering with user activities [Kemmerer & Vigna 2002]. As a result, these systems showed a significant delay between intrusion and triggered alarm. During the early 1990s, intrusion detection systems tried to address the issue of timely response and were developed with real-time analysis in mind. Well-known examples of successful open-source IDSs are Tripwire ([Kim & Spafford 1994], [Tripwire 2011]), Bro ([Paxson 1999], [Bro 2011]), Snort ([Roesch 1999], [Snort 2011]), Samhain ([Wotring *et al.* 2005], [Samhain 2011]), OSSEC ([Hay *et al.* 2008], [OSSEC 2011]) and Prelude ([Prelude 2011]).

Furthermore, excellent introductions to the fields of information assurance, internet security and intrusion detection are provided by [Shields 2006], [Cheswick *et al.* 2003] and [Stallings 2006]. [Northcutt & Novak 2003], [Bejtlich 2004], [Northcutt *et al.* 2005] and [Bejtlich 2006] provide more extensive information and hands-on tutorials in the field of network intrusion detection and traffic analysis.

In the remainder of this chapter, we first cover the different stages an attacker goes through when trying to compromise a system. Subsequently, we outline exploitable vulnerabilities of a typical computer system and classify the resulting possible threats against the system. Then, we describe different kinds of intrusion detection systems according to data sources and detection techniques. Next, we present a review of previous work applying machine learning methods to intrusion detection. Finally, we briefly touch on the basics of IDS response in terms of event correlation and report.

2.2 Phases of Compromise

Intrusions can be initiated by unauthorised users defined as 'attackers'. An attacker can try to access a machine remotely via the Internet or to render a service remotely unusable. A local attacker with access to a local machine might try to gain additional privileges by misusing their existing credentials.

If we want to detect intrusions, we need to learn about the procedure required in order to successfully attack a system. In most cases, attacks can be split into five phases: reconnaissance, exploitation, reinforcement, consolidation, and pillage. This outsider attack timeline is suggested by [Bejtlich 2004]. During the first three phases, there is a fair probability that the activities of the attacker will be detected. Once the attacker gains full control over the target system, however, it becomes increasingly difficult to differentiate between legitimate and illegitimate activity.

Most attacks start with the validation of connectivity. *Reconnaissance* involves the identification of reachable hosts and services, and the versions of operating systems and applications running. During this first phase, an attacker uses tools to collect information about potential targets. This includes the scanning of reachable IP addresses and open TCP/UDP ports, and the characterising of the target by other active and passive means (often referred to as 'fingerprinting'). From this information, the attacker can draw conclusions about the potential vulnerabilities of the target system. The list of potential vulnerabilities is then used to develop a structured plan for attacking the target in the most unobtrusive way possible. Two well-known tools for network and vulnerability scanning are nmap ([Lyon 2009], [nmap 2011]) and Nessus/OpenVAS ([Nessus 2011], [OpenVAS 2011]).

During the *exploitation* phase, an attacker tries to exploit a service in order to gain access to the target machine. This can be done by either abusing, subverting, or breaching the service. Abuse can involve 'dictionary' attacks or the use of stolen passwords, whereby the attacker gains access using a legitimate route. In subversion, the attacker causes the service to perform in a way not anticipated by the developers by using such techniques as SQL-injection ([Anley 2002]).

After a successful break-in, the attacker proceeds to take advantage of the privileges gained. During *reinforcement*, the typical steps of the attacker are to camouflage activity and to install additional tools and services. This might include removing logs, adding new user accounts, or disguising files and processes. It is common for attackers to install some kind of remote access service, which bypasses normal authentication, in order to facilitate access to the system. This is called a 'backdoor' and ensures that the attacker need not rely on the original exploit. In some cases, the attacker additionally fixes the original security leaks, which prevents other attackers from owning the system.

The escalation of privileges is also part of the reinforcement phase. The attacker works his way from the misused user account towards gaining full system access. To this end, the attacker might exploit applications accessible from the available user account. The Metasploit framework ([Maynor & Mookhey 2007], [Metasploit 2011]) is an example of a powerful tool for crafting and executing exploit code. It offers various options for post-compromise exploitation, such as the creation of backdoors and multiple-stage attacks.

During the *consolidation* phase, the attacker has complete control over the owned system and communicates via the installed backdoor. In theory, the backdoor could be some form of listening service to which the attacker connects. In most cases, the backdoor itself connects in an outbound direction to the attacker using a common service; for example, via an IRC channel. This prevents firewalls and intrusion detection systems from being easily triggered by the means of an attacker. It is very common for backdoors to use covert channels, where the backdoor communication then appears to be normal traffic.

The phase in which an attacker executes his final plan is called *pillage*. Possible malicious actions include theft of data and CPU time, and impersonation. By impersonation, the attacker can broaden the attack to include other internal or external hosts. Having gained root privileges, the

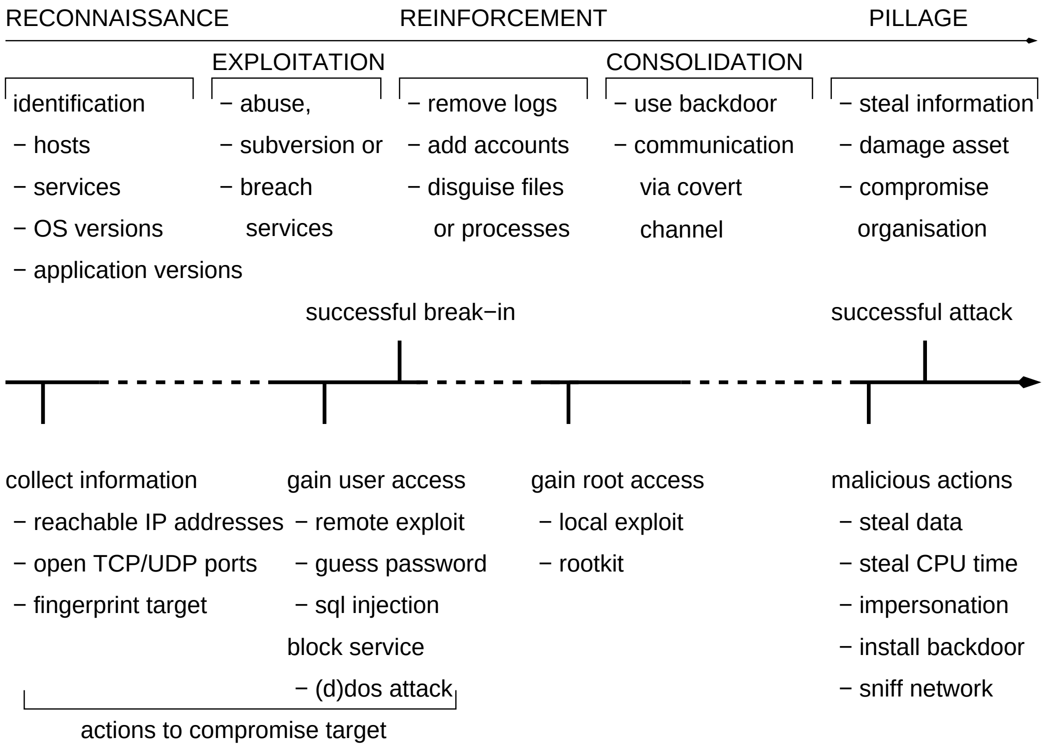


Figure 2.1: *The five phases of compromise—the outsider attack timeline. The attacker starts by collecting information about potential targets. Then, he tries to gain user access to the target machine. After a successful break-in, having gained full control over the system, he installs additional tools and services. Finally, the attacker can carry out malicious actions, such as compromising an organisation.*

attacker can easily 'sniff' and redirect network traffic on a local network. This can be achieved by exploiting well-known weaknesses in network protocols, such as ARP and DNS. The relevant weaknesses are discussed in detail by [Tripunitara & Dutta 1999] [Schuba & Spafford 1994]. The five phases of compromise are shown in Figure 2.1.

2.3 Vulnerabilities and Threats

Humans make mistakes; computers are built and programmed by humans. So, it is a fair assumption that computers also contain bugs in hardware and software, which might be related to errors in design and/or implementation. If this assumption holds, then the only questions remaining are how serious these bugs are, and whether or not they can cause (direct or indirect) damage to affected systems. A *vulnerability* is such a flaw, which can be used by an attacker to force interaction with a computer system or network ([Bace & Mell 2001]).

Computer systems have limited memory, disk space, processing power and network bandwidth. Thus, an attacker could render a system unusable by exhausting shared resources or forcing the system into undefined behaviour, such as that triggered by *division by zero* errors. This can happen when the system fails to validate the provided input, thereby enabling an attacker to input invalid data into the system.

Typical errors that can lead to vulnerabilities are associated with access validation, exception handling, and misconfiguration. An access validation error is caused by faulty access control mechanisms, whereas exception handling errors are related to the mishandling of an exceptional condition by the system. The vulnerability caused by misconfiguration is not due to system design, but rather how the user configured the system ([Bace & Mell 2001]).

The most common types of vulnerability are boundary condition errors. If the system does not check for input length, an attacker can overflow allocated memory. This kind of error occurs during a *buffer overflow* attack. Buffer overflows normally simply lead to erratic program behaviour, such as program termination or wrong results. But by carefully crafting the data written to the overflowed memory region, according to architecture, operating system and memory region, the attacker can force the system to execute injected code ([Cowan *et al.* 2000]).

The dictionary of *Common information security Vulnerabilities and Exposures* (CVE)[CVE 2012] defines a vulnerability as a state that allows an attacker to:

- execute commands as another user;
- access data that is contrary to the specified access restrictions for that data;
- pose as another entity; or
- initiate a denial-of-service.

It is clear that we can categorise threats, because computer systems can become corrupted in different ways. The three 'pillars' of information security are: confidentiality, data integrity and availability. We need to consider at least these three aspects when developing system security. Furthermore, there are numerous additional factors, such as authenticity and accountability, which are also important within specific environments ([Bishop 2004]).

Confidentiality mainly addresses passive attacks that are based on eavesdropping; a violation occurs whenever someone who is not authorised to do so is able to extract confidential information from a communication. IDS systems have limited capability to detect eavesdropping, which cannot be detected directly unless it involves alteration of data.

Attacks against the *integrity* of a system are generally active attacks. During this kind of violation, an attacker changes a system state or data without authorisation. This is especially problematic if altered messages cannot be easily distinguished from unchanged messages.

The integrity attacks addressed by IDS systems are system scanning activities (also called 'probing') and system penetration. These attacks can be launched either over a network or locally within a system.

Another type of attack is concerned with *availability* violation, which might prevent an authorised user or machine from accessing a particular resource in a legitimate manner. Denial-of-service attacks and distributed denial-of-service attacks try to render a computer resource unavailable. IDS systems usually also aim to detect these kinds of attack.

Summarising various approaches to classifying threats as proposed by [Anderson 1980], [Denning 1987] and [Smaha 1988], we can define an attack as a set of actions that potentially compromises the confidentiality, data integrity, availability, or any kind of security policy of a resource.

In this work we use the categorisation scheme as suggested by the *DARPA Intrusion Detection Evaluation* ([DARPA 2011]). Between 1998 and 2000, the MIT Lincoln Laboratory created the *DARPA Intrusion Detection Data Sets* ([Lippmann *et al.* 2000a] and [Lippmann *et al.* 2000b]). These datasets contain a wide variety of different attacks classified into four attack categories: denial-of-service ('dos'); network probe ('probe'); remote-to-local ('r2l'); and user-to-root ('u2r'). The two latter attack categories can be further aggregated to system penetration. The DARPA datasets are explained in more detail in Chapter 4.

2.3.1 Denial-of-Service

Denial-of-service (DOS) attacks can either flood the victim or exploit a flaw, rendering the system or the service unusable. By flooding, the attacker tries to exhaust the network or processing capabilities of the host. As a result of this, the target host is no longer able to provide a service to any valid user. If the attacker launches the attack from multiple hosts, it is termed a *distributed denial-of-service* (DDOS) attack. A well-known example of a successful DOS attack was that of the slammer worm, which hit in 2003 and successfully infected at least 75,000 hosts within the first 10 minutes of its release ([Moore *et al.* 2003]). A thorough introduction to denial-of-service attacks is provided by [Mirkovic *et al.* 2005].

2.3.2 System Scanning

By probing a computer system or an entire network infrastructure, a potential attacker can learn about its characteristics and vulnerabilities. There are two types of so-called 'scanners': Network scanners and vulnerability scanners. Among other things, network scanners search for IP addresses and TCP/UDP ports. An attacker may thus obtain information about the network topology, active hosts, operating systems and available services. One of the most favoured network scanners is nmap ([Lyon 2009], [nmap 2011]).

Vulnerability scanners search for vulnerabilities on active hosts. They output a list of active hosts in the scanned network that are vulnerable to attacks known to the scanner. OpenVAS ([OpenVAS 2011]) is an example of a well-established vulnerability scanner.

2.3.3 System Penetration

System penetration aims to gain unauthorised control over a system. This is achieved by exploiting bugs, glitches, or other vulnerabilities in order to cause unexpected behaviour that can be exploited for privilege escalation. These so-called *exploits* can be classified by the way in which they contact the vulnerable host. Remote exploits can directly attack the host over the network, whereas local exploits require the attacker to already have user privileges on the host.

The result of a successful remote exploit is local access to the attacked system, and such a successful local exploit inevitably increases the privileges of the attacker on the attacked host. The Metasploit framework ([Maynor & Mookhey 2007], [Metasploit 2011]) provides an impressive arsenal of applicable exploits. An open-source penetration testing framework focused on Web applications is w3af (Web Application Attack and Audit Framework) by [Riancho 2011].

2.4 Data Sources

Intrusion detection systems can be categorised according to 'what' or 'how' they analyse. The 'what' refers to what kind of data is analysed, and the 'how' refers to how the data is analysed.

Referring to what data is analysed, the data classified by an IDS can be either local host data or network data. An IDS that analyses the local data of computer systems, in order to detect attacks on that specific host, is called a *host intrusion detection system* (HIDS). An IDS that runs on specialised network equipment, collecting network data, is called a *network intrusion detection system* (NIDS). Network data is recorded and analysed in an attempt to identify attacks and potential threats buried deep within in network traffic. In this thesis, our focus is on network intrusion detection.

2.4.1 Host Intrusion Detection

Host intrusion detection systems monitor events that occur locally on a computer [Scarfone & Mell 2007]. Events are monitored and analysed by local sensors. Optionally, events can be forwarded to a central management system. Typical events analysed by host intrusion detection systems are those related to the system kernel, the file system, and users' account information. Information of interest, such as running processes, modified files and logged-in users, can be extracted from kernel audit trails, file integrity checks and registry data.

A major advantage of host intrusion detection systems is that they can detect attacks when the malicious action occurs locally within the monitored system. This is the case when the attacker has successfully gained user access to a monitored system and then tries to elevate his privileges to administrative level. Another advantage is that host intrusion detection is unaffected by the encryption of network traffic. Host-based intrusion detection systems are typically complex, because they require individual management and customisation for each monitored host ([Bace & Mell 2001]).

Early implementations of host-based intrusion detection systems are IDES/NIDES ([Lunt *et al.* 1992], [Anderson *et al.* 1995]) and Haystack ([Smaha 1988]), developed in the late 1980s. More recent HIDS projects worth noting are Tripwire ([Kim & Spafford 1994], [Tripwire 2011]), Samhain ([Wotring *et al.* 2005], [Samhain 2011]), and OSSEC ([Hay *et al.* 2008], [OSSEC 2011]).

2.4.2 Network Intrusion Detection

In network intrusion detection, attacks are detected by monitoring and analysing events associated with network traffic [Scarfone & Mell 2007]. Network traffic is mirrored by networking devices, such as switches, routers, and firewalls, or by network taps that are placed directly on the physical transmission media, such as copper wire or optical fibre [Bejtlich 2006]. The mirrored traffic is analysed and concentrated by sensors placed at various points in the network. It is common for the sensors to report potential attacks and forward connection records to a central management system. If sensors are well-placed, network intrusion detection systems can run completely passively, monitoring a large network infrastructure ([Bace & Mell 2001]). For a potential attacker, these systems are very difficult to detect. In most cases, the centralised nature of network intrusion detection systems makes the monitoring and analysis of all traffic difficult in large networks.

Early implementations of network-based network intrusion detection systems are DIDS ([Snapp *et al.* 1991]), NSM and NADIR (see both in

[Mukherjee *et al.* 1994]). More recent interesting NIDS projects are Bro ([Paxson 1999], [Bro 2011]) and Snort ([Roesch 1999], [Snort 2011]).

2.5 Detection Techniques

Investigating 'how' different IDS systems analyse collected data, we note that analysis is either static or heuristic. When an IDS uses filters and signatures to describe attack patterns, the analysis is static; this is called *signature detection* (or misuse detection). Signature detection is limited to the detection of known attack patterns.

For the detection of unknown attacks, heuristic methods must be used. Systems that use heuristic methods offer the possibility of detecting patterns that are not 'normal'; these detection methods are termed *anomaly detection*. A third very common approach for extending static detection methods is called *stateful protocol analysis*. A more detailed overview of IDS detection techniques is provided by [Kumar 1995], [Bejtlich 2004], and [Scarfone & Mell 2007].

2.5.1 Signature Detection

The majority of today's intrusion detection systems search for predefined patterns associated with a specific well-known attack. These signature-based intrusion detection systems provide a separate, searchable pattern for each attack. Signatures are hand-coded by human experts, based on detailed knowledge of the attack ([Han & Kamber 2006]). Some systems can model groups of attacks using state-based analysis techniques, but most systems fail to detect variants of attacks. Signature detection is the prevalent kind of intrusion detection in use today because it generates low false alarm rates in comparison to other systems. Signature-based intrusion detection systems depend upon frequent signature updates to keep up with new emerging threats ([Bace & Mell 2001] and [Scarfone & Mell 2007]).

There have been efforts to automate the task of signature generation. [Kim & Karp 2004], for example, propose a system that automatically generates worm signatures for novel Internet worms. [Song *et al.* 2007], on the other hand, reason that signature detection is an ‘arms race’ between attacker and defender which cannot be won by the defence. This is backed by [Spinellis 2003], who proved that finding a solution for the problem of reliably identifying a known, bounded-length, mutating virus is already very difficult.

A representative example of a signature-based intrusion detection system is the Snort NIDS ([Roesch 1999], [Snort 2011]).

2.5.2 Stateful Protocol Analysis

Stateful protocol analysis tracks the state of network, transport and application protocols. Each protocol has a profile that defines valid activities and the transitional states between them. Profiles are based on the protocol standards issued by official standards bodies and software vendors. Profiles should also take variations in standards and enhancements in different implementations into account [Scarfone & Mell 2007]. Protocol event sequences identified as having particular patterns of actions that deviate from the defined profile are reported as potentially malicious. State tracking is very resource intensive and relies heavily on carefully generated profiles. Attacks that do not violate any protocol profile cannot be detected through stateful protocol analysis.

One of the prominent examples of stateful protocol analysis is published by [Dreger *et al.* 2006]. They present an extension for the Bro IDS ([Paxson 1999], [Bro 2011]), which performs dynamic application-layer protocol analysis for a number of common and well-defined protocols (e.g. HTTP, IRC, FTP and SMTP).

2.5.3 Anomaly Detection

Anomaly detection is based on the assumption that ‘normal’ events can be distinguished from potentially malicious events. An anomaly detection system constructs a profile, or set of rules, from historical operational data. The historical data should be free of unacceptable events. The resulting profile then defines what the intrusion detection system will recognise as acceptable behaviour. Profiles can be related to the behaviour of users, or be completely user-independent. One method of profiling is threshold-based, defining the frequency of occurrence of various events. Other methods use various statistical or rule-based measures to map parametric 'known' distributions and non-parametric 'learned' distributions ([Scarfone & Mell 2007]). Finally, there is the option to use trained classifiers.

Intrusion detection systems based on anomaly detection have the potential to detect unknown attacks. The detection of unusual behaviour does not rely on knowledge of the details of the attack. A human expert is needed to classify the detected event and initiate appropriate action. New behaviour needs to be continuously added to the profiles in order to minimise the number of false positives. Unfortunately, the false positive rate in current anomaly-based intrusion detection systems is still unacceptably high, because these systems still have significant problems in modelling 'normal' user behaviour ([Bace & Mell 2001] and [Scarfone & Mell 2007]).

[Lakhina *et al.* 2005] show that anomalies naturally fall into distinct and meaningful clusters when they are treated as events that alter the distribution of traffic features. Focusing on application data, [Wang & Stolfo 2004] present a different approach based on the detection of anomalous payloads. [Gates & Taylor 2006] critically question the assumptions of the research community made in the field of anomaly detection, especially with regard to network intrusion detection. A more detailed classification scheme together with a summary and discussion of the main anomaly-based intrusion detection technologies are presented by [Garcia-Teodoro *et al.* 2009].

An anomaly sensor for detecting stealthy port scans, Spade, is presented by [Staniford *et al.* 2002]. Spade is available as a bleeding-edge extension for the Snort NIDS ([Roesch 1999], [Snort 2011]).

2.6 Machine Learning Techniques

Machine learning techniques have been used for network intrusion detection for some time, but the choice of the available training data is very limited. One of the few widely used datasets is from the DARPA datasets ([Lippmann *et al.* 2000a], [Lippmann *et al.* 2000b]), which also happens to be one of the most comprehensive. The tcpdump data provided by the 1998 DARPA Intrusion Detection Evaluation network was processed and used for the

1999 KDD Cup contest at the Fifth International Conference on Knowledge Discovery and Data Mining. The learning task of this competition was to classify the preprocessed connection records into either normal traffic, or one out of the four given attack categories ('dos', 'probe', 'r2l', 'u2r').

Preprocessing of the data for the KDD Cup '99 competition was done with the MADAMID framework described in [Lee 1999] and [Lee & Stolfo 2000]. Each connection record contains 41 input features grouped into *basic features* and *higher-level features*. The dataset provides the training and testing datasets in a full set, and also a '10%' subset version with modified class distributions. During the KDD Cup '99 competition, 24 entries were submitted. The first three places were occupied by entries that used variants of decision trees and showed only marginal differences in performance. In ninth place in the challenge, was the 1-nearest neighbour classifier. The first 17 submissions of the competition were all considered to perform well and are summarised by [Elkan 2000].

Observing feature reduction on the KDD Cup '99 datasets, the majority of published results are trained and tested on the '10%' training set only (see [Sung 2003], [Kayacik *et al.* 2005] and [Lee *et al.* 2006]). Some researchers used custom-built datasets, with 11,982 random records extracted from the '10%' KDD Cup '99 training set (see [Chavan *et al.* 2004], [Chebrolu *et al.* 2005] and [Chen *et al.* 2005]). [Sung 2003] applied single-feature deletion to the KDD Cup '99 datasets, using neural networks and support vector machines. With the SVM classifier, they extracted a 30-feature set and, using the neural network classifier, they extracted a 34-feature set. For the SVM classifier, they also reduced the number of features for the five individual traffic classes to 25 ('normal'), 7 ('probe'), 19 ('dos'), 8 ('u2r') and 6 ('r2l'). Important input features were also identified by [Chebrolu *et al.* 2005]. They investigated the performance of Bayesian networks and classification and regression trees, and suggest a hybrid model using both classifiers. The feature reduction using the Markov blanket model found a 17-feature set, whereas classification and regression trees resulted in a 12-feature set.

[Chavan *et al.* 2004] use a decision tree approach for feature ranking per class. For evaluation, they use artificial neural networks and fuzzy inference systems. The authors reduce the number of features to 13 ('normal'), 16 ('probe'), 14 ('dos'), 15 ('u2r') and 17 ('r2l'). [Kayacik *et al.* 2005] investigated the relevance of each feature provided in the KDD Cup '99 intrusion detection dataset in terms of information gain and present the most relevant feature for each individual attack. [Chen *et al.* 2005] reduce the number of input features using the flexible neural tree Model to 4 ('normal'), 12 ('probe'), 12 ('dos'), 8 ('u2r') and 10 ('r2l'). A genetic feature selection method was proposed by [Lee *et al.* 2006]. Performance was measured using a selective naïve Bayes classifier. Both methods extracted a total of 21 features, with 11 features in common.

A short time after the 1998 and 1999 DARPA intrusion detection system evaluations, [McHugh 2000] wrote a detailed critique, identifying shortcomings of the provided datasets. [Mahoney & Chan 2003] looked more closely at the content of the 1999 DARPA evaluation tcpdump data. [Sabhnani & Serpen 2004] investigated the reasons why classifiers fail to detect most of 'r2l' and 'u2r' attacks in the KDD Cup '99 datasets. [Brugger & Chow 2005] applied the tcpdump traffic data files provided with DARPA datasets to the Snort intrusion detection system.

The winning entries of the KDD Cup '99 contest, presented by [Pfahringer 2000], used a variant of the C5 decision tree algorithm. The second-placed decision tree solution, by [Levin 2000], build an optimal decision forest. Third place was awarded to a decision tree solution labelled as MP13, by [Vladimir *et al.* 2000]. [Agarwal & Joshi 2000] proposed a rule-based classifier model for multi-class classification called PNrule. The model consists of positive and negative rules that predict the presence or absence of a class respectively. Classes could be individual attacks or whole categories, such as 'r2l' and 'u2r'.

After the challenge, a number of new results using learning algorithms on the KDD Cup '99 data were published. In the following papers, the authors used the same training and testing data as requested in the challenge, and provided comparable results: [Sabhnani & Serpen 2003] evaluate a comprehensive set of machine learning algorithms and suggest a multi-classifier model with a multi-class topology. The different algorithms applied were a multilayer perceptron neural network, an incremental radial basis function neural network, a maximum likelihood Gaussian classifier, k-means clustering, a nearest cluster algorithm, a leader algorithm, a hypersphere algorithm, a fuzzy adaptive resonance theory mapping algorithm, and the C4.5 decision tree.

[Hu & Hu 2005] applied the classical Adaboost algorithm and a modified version of the same to the KDD Cup '99 datasets. Decision stumps were chosen as a weak classifier and given as input to Adaboost. [Song *et al.* 2005] demonstrate RSS–DSS, a genetic programming approach for large datasets comparing the results of using the first 8 basic features only with using all features of the dataset. A machine learning approach, based on unsupervised presentation of data, is applied by [Kayacik *et al.* 2007]. They use a multi-layer, self-organising feature-map hierarchy with customised datasets.

There are a number of papers with partially comparable results, where the authors used the DARPA or KDD Cup '99 training data but applied different test sets to their trained classifier. In an early paper, [Sinclair *et al.* 1999] suggest genetic algorithms and decision trees for automatic rule generation for an expert system that enhances the capability of an existing IDS. [Yeung & Chow 2002] observed a nonparametric density estimation approach, based on Parzen-window estimators with Gaussian kernels. [Mukkamala *et al.* 2004] compared the performance of a linear genetic programming approach to artificial neural networks and support vector machines.

[Abraham & Grosan 2006] investigate the results of linear genetic programming and multi-expression programming. Other hybrid approaches combine neural networks and support vector machines, published by [Mukkamala *et al.* 2003], artificial neural networks and a fuzzy inference system, by [Chavan *et al.* 2004], and decision trees and support vector machines, by [Peddabachigari *et al.* 2007].

There are also a number of interesting publications where the results are not directly comparable due to the use of different training and test datasets. [Debar *et al.* 1992] and [Cannady 1998] suggested the use of neural networks as components of intrusion detection systems. [Zhang *et al.* 2001] compared the performance of a selection of neural network architectures for statistical anomaly detection to datasets from four different scenarios. The use of hidden Markov models to detect complex multi-stage Internet attacks that occur over extended periods of time is described by [Ourston *et al.* 2003]. An event classification scheme based on Bayesian networks is proposed by [Kruegel *et al.* 2003].

A framework for unsupervised learning, with two feature maps mapping unlabelled data elements to a feature space, is suggested by [Eskin *et al.* 2002]. [Bivens *et al.* 2002] further illustrated that neural networks can be efficiently applied to network data in both a supervised and an unsupervised learning approach. [Laskov *et al.* 2005] demonstrate that supervised learning techniques applied to the KDD Cup '99 training data significantly outperform unsupervised methods. The best performance is achieved by non-linear methods.

2.7 Event Correlation and Report

IDSs automatically monitor and correlate events that are collected by IDS sensors, which are then analysed for potential intrusions. Monitored events can occur either locally on a computer system or within a network infrastructure. Once a potential attack or an anomalous pattern is detected, an IDS generates an alert.

The fact that IDS sensors report to a central IDS management system enables the intrusion detection system to observe the monitored infrastructure from multiple points of view. Alarms generated by individual IDS sensors and the IDS management system are documented in reports. These reports need to be analysed and evaluated by human experts, who decide on what action to take. Large IDS infrastructures can easily generate thousands or even millions of events per day; their evaluation would be an unrealistic task for any human being. For this reason, IDS systems need to be carefully tuned and maintained within a given environment.

Generally, IDS sensors and IDS management systems are limited to monitoring and analysis. Systems such as firewalls, which introduce countermeasures to prevent the success of attacks, are called *intrusion prevention systems* (IPSs). Intrusion prevention can defend a target system against an attacker without the administrator's direct involvement [Scarfone & Mell 2007]. In theory, these systems can block malicious traffic or system calls before they cause any harm. In this respect, an IPS can be seen as an enhancement of an existing firewall, using well-known IDS features with very low false alarm rates.

An example of an anomaly-based intrusion detection system focused on event correlation is Prelude ([Prelude 2011]).

2.8 Conclusions

In this chapter, we covered the essential concepts of intrusion detection. We learned that intrusions follow a timeline, which can be split into five phases: reconnaissance, exploitation, reinforcement, consolidation, and pillage. This process starts with the collection of information about potential targets and finishes with the successful compromise of the target system.

We defined attacks as sets of actions which attempt to compromise the confidentiality, data integrity and/or availability of a resource, and found that different types of attack can be classified into three main attack categories: denial-of-service, system scanning, and system penetration.

We learned that IDS systems can be classified by either data source or detection techniques. The former category can be subdivided into host intrusion detection systems and network intrusion detection systems, the latter into signature detection, stateful protocol analysis and anomaly detection. The majority of current IDSs use signature-based detection, but only systems based on anomaly detection have the potential to detect threats previously unknown to the particular detector.

We learned that machine learning techniques have been applied to intrusion detection for some time and that available training data is very scarce. We also learned that IDSs need to handle large numbers of events. These are produced by IDS sensors and correlated by a central IDS management system.

In the next chapter, we introduce a number of data mining methods that we have applied throughout this research work to handle these events of classification.

Chapter 3

Data Mining Methods

Contents

3.1 Introduction

In this chapter, we present a selection of machine learning methods. Five of these methods are traditional, static classifiers, and two are classifiers with dynamic capabilities. Later in this thesis, these classifiers are applied to intrusion detection data.

Machine learning endows a computer with the ability to learn from experience. In other words, it is concerned with the development of algorithms that automatically improve by practice. Ideally, the more the learning algorithm is run, the better the algorithm becomes. It is the task of the learning algorithm to create a classifier function from the training data presented. The performance of this built classifier is then measured by applying it to previously unseen data.

Based on the training data, the learning process can be categorised into supervised and unsupervised learning. In *supervised learning*, the training data consists of input and output object pairs. Every input object is labelled with a corresponding desired output. Seeing a number of valid input and target output pairs, the supervised classifier should learn to predict the target output for any given valid input object. *Unsupervised learning*, however, trains on data that consists only of input data without target output information. The unsupervised classifier learns from completely unlabelled data. It assumes that objects that belong to the same class tend to form a cluster in some metric space. In this thesis, we focus on the supervised learning approach by using a fully labelled dataset.

All classifiers presented are applicable to supervised learning tasks. The static classifiers can only provide a static mapping between input and output. They are not suitable for solving supervised learning tasks of a temporal nature, such as sequence classification problems where the input vectors are presented in sequences with separate and distinct time steps. To solve these problems, we require a classifier with dynamic capabilities.

The selected static classifiers covered in this chapter are decision trees (C4.5 and J4.8 [Quinlan 1986] and [Quinlan 1993]), naïve Bayes ([John & Langley 1995]), Bayesian networks ([Heckerman *et al.* 1995]), feed-forward neural networks (FFNN [Rumelhart *et al.* 1994]) and support vector machines (SVM [Boser *et al.* 1992] and [Cortes & Vapnik 1995]). Excellent overviews of these static classifiers presented are provided by [Mitchell 1997] and [Han & Kamber 2006].

Chosen classifiers with highly nonlinear dynamic capabilities are recurrent neural networks (RNN [Rumelhart *et al.* 1986], [Williams & Zipser 1989], [Williams & Zipser 1995], and [Hochreiter *et al.* 2001]) and, in particular, long short-term memory (LSTM [Hochreiter & Schmidhuber 1996], [Hochreiter & Schmidhuber 1997], [Gers *et al.* 1999] and [Gers *et al.* 2002]) recurrent neural networks. The section on recurrent neural networks is included in this chapter for reasons of clarity and completeness. A sound understanding of RNNs is essential for working with LSTM RNNs.

3.2 Decision Trees

Decision tree learning is one of the commonest machine learning methods and is very intuitive to understand. Learned functions are usually represented in the form of a tree-like structure, representing a set of decisions that can be translated into if-then rules. Depending on the algorithm used, the representation may be binary or multi-branched.

Each node in a decision tree represents one attribute of an instance. Branches descending from a node correspond to possible attribute values. Leaves represent possible values of the target variable, given the path starting at the root node and ending at the observed leaf.

To classify an item, the decision tree is followed from the root to a leaf. At every node, an attribute is tested and, based on the outcome, the corresponding branch is followed. This procedure continues until a leaf is reached. All instances that reach a certain leaf are classified by the value the leaf represents.

The test at each node usually compares an attribute value with a constant, but there are also trees that apply more complex operations, such as comparing two attributes with each other.

Attributes with nominal values are tested once, with one descending branch per possible value. Numeric attributes need to be converted into nominal attributes. Usually, the node determines if the observed value is greater or less than a predetermined constant, giving two descending branches. Nodes with three descending branches can also be produced by adding an equal-to comparison or by treating missing values as separate attribute values. Several other alternatives for handling numeric attributes also exist. Numeric attributes, in contrast to nominal attributes, can be tested several times in the tree, each time using a different predetermined constant.

Figure 3.1 illustrates a simplistic decision tree for the detection of an attack on a remotely accessible computer system. The leaf nodes of the tree return the result of the classification (attack? YES/NO). At the root node, the decision tree checks if a root shell is obtained on the observed system. In the absence of a running root shell, the left branch is processed. When the network communication protocol port is not whitelisted as an accepted destination, the amount of data sent to this port is tested against a given threshold, and connections with source data above the threshold are classified as an attack. The right branch is processed if the root shell is obtained on the observed system. Then the local host intrusion detection system is checked for related alerts; an alert classifies the connection as an attack. If the IDS does not give an alarm, the amount of data received is tested, and connections with destination data below a given threshold are classified as attacks. For an excellent introduction to decision trees, see the corresponding chapter in [Mitchell 1997]. For more information, see [Quinlan 1993].

Two popular decision tree learning algorithms are ID3 and its successor, C4.5, introduced by [Quinlan 1986] and [Quinlan 1993]. Like most decision tree learning algorithms, they pursue a top-down ‘greedy’ search through the space of possible decision trees. In the following, we outline ID3, which is a well-known decision tree algorithm.

ID3 starts with a single root node representing all training samples. At

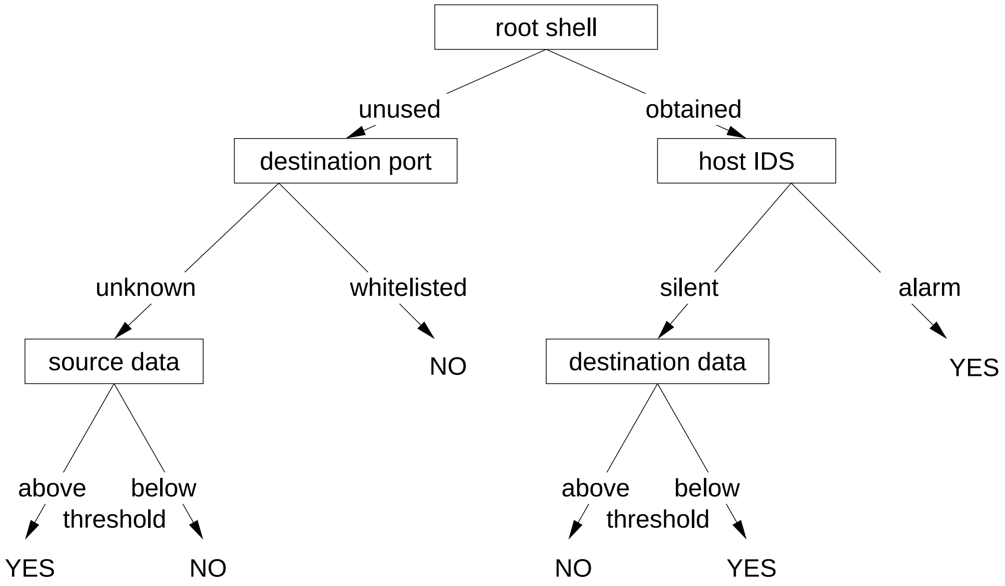


Figure 3.1: *A simplistic decision tree for the detection of a user-to-root attack on a computer system. An example is classified by sorting it through the tree. Classification starts at the root node and ends at one of the leaves, returning the classification associated with it (attack? YES/NO).*

each node, an attribute value test is conducted. The test returns the attribute that best separates the training examples according to the target classification, and the chosen attribute then becomes the so-called 'decision attribute' for the node. For every value of the decision attribute, a descending branch and a child node are created, and the samples are partitioned accordingly. Testing continues for every node representing a partition until all samples within every partition are of the same class. Finally, the node becomes a leaf and is labelled with the name of the target class.

Most decision tree algorithms use a statistical property called *information gain*. Information gain is an entropy-based measure that selects an attribute that best separates samples into individual classes. The attribute with the highest information gain becomes the decision attribute for the current node when building a tree.

Information gain is based on the concept of information entropy, which describes the amount of information in a signal or event. It is the expected reduction of entropy caused by the partitioning of examples, according to a considered attribute. It is defined as:

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

where S is the collection of training examples, A is the observed attribute, $Values(A)$ is the set of all possible values for A, and S_v is the subset of S where A has the value v.

Given that the target classification is Boolean, the information entropy is defined as:

$$Entropy(S) = -P_1 \log_2 P_1 - P_2 \log_2 P_2$$

with a collection of training examples S that can be classified, with respect to an attribute, into two example sets, P_1 and P_2. $0 log 0$ is defined as 0. The entropy is 1 when the collection contains an equal number of positive and negative examples. Otherwise, the entropy is between 0 and 1.

In cases where the target classification can take c different values, the information entropy is:

$$Entropy(S) = \sum_{i=1}^{c} -P_i \log_2 P_i$$

Decision tree learning can be applied to problems where the instances are described as a fixed set of attributes and values. The output of the target function has discrete values, and training data is noisy and may be missing attribute values. Both make decision tree learning well-suited to network intrusion detection.

3.3 Bayesian Classification

Bayesian classifiers are statistical classifiers that predict class membership probabilities. They are based on Bayes' theorem, proved by and named after the mathematician Thomas Bayes (1702 - 1761). It provides a way of calculating the posterior probability from the prior probability.

Bayes' theorem is:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

where $P(h)$ is the prior probability that the hypothesis h holds; it is 'prior' in the sense that it does not take into account any information about the training data, D. $P(h|D)$ is the posterior probability that h holds, given D. $P(D|h)$ is the posterior probability observing D, given a scenario in which h holds. $P(D)$ is the prior probability that the data D will be observed.

According to studies comparing classification algorithms referenced by [Han & Kamber 2006], the performance of the naïve Bayes classifier is comparable to decision trees and selected neural networks. The chapters on Bayesian learning by [Mitchell 1997] and [Han & Kamber 2006] provide an excellent introduction to this field.

3.3.1 Naïve Bayes

The *naïve Bayes* is a simple probabilistic classifier. It assumes that the effect of a variable value on a given class is independent of the values of other variables. This assumption is called 'class conditional independence' and reduces the number of parameters. In practice, this is not a serious problem, because the naïve Bayes models perform well, even if this assumption does not hold for the analysed data.

The algorithm stores the prior class probabilities and the posterior probability of each attribute assigned to that class. During the learning phase, it estimates these probabilities from examples by simply measuring the frequency of their occurrence. The prior probability is the portion of examples from each class. The posterior probability is the frequency at which attribute values occur in the given class.

During an observation, the algorithm operates under the assumption that attributes are conditionally independent. The algorithm uses Bayes' theorem to calculate the posterior probability of each class. It returns the class label with the highest probability.

Despite its simplicity and the assumptions made, given class conditional independence, this algorithm can often outperform more sophisticated classification methods in terms of both accuracy and speed, especially when applied to large datasets. More detailed information about the naïve Bayes classifier is provided by [John & Langley 1995].

3.3.2 Bayesian Networks

Bayesian networks, also called 'belief' or 'probabilistic' networks, are statistical classifiers that use graphical models. They are drawn as directed acyclic graphs of causal relationships. Every node represents an attribute, and the edges describe the relations between them.

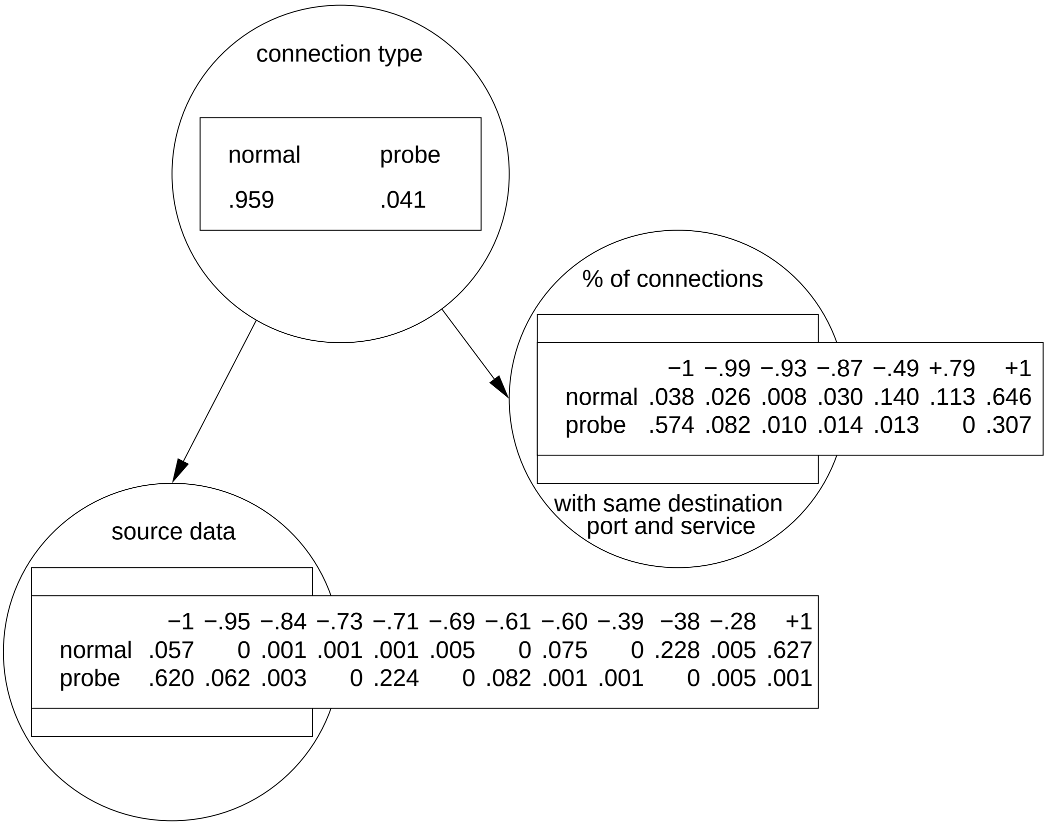
Every node contains a conditional probability table that defines the probability distribution, and this table is used to predict the class probabilities for every given instance. The probability of each feature value depends on the values of the attributes of the parent nodes, nodes without parents having an unconditional probability distribution.

Figure 3.2 shows a simplistic Bayesian belief network for the detection of network probes. There is a node for each of the two attributes 'source_data' and '%_of_connections_with_same_destination_port_and_service'. The third node for the class attribute 'connection_type' has edges to both other nodes. The nodes contain the tables with the probability distributions in order to predict the class probabilities for a given instance. In this example, table values are normalised to [-1,1].

To calculate the probability of each class value for a given instance, we need to assume that there are no missing attributes. The conditional probability table for every node in the network provides an attribute value based on the row, as determined by its parent's attribute values. The joint probability of each class value for any given instance $[a_1, a_2, ..., a_n]$ to the tuple of attributes $[A_1, ..., A_n]$ can be computed by multiplying the probabilities provided by each node:

$$Pr[a_1, a_2, ..., a_n] = \prod_{n=1}^{n} Pr[a_i | Parents(A_i)]$$

where $Parents(A_i)$ is the set of immediate predecessors of the attribute A_i. To obtain the conditional probabilities, we need to normalise the joint probabilities by dividing them by their sum.



normal	probe
.959	.041

	−1	−.99	−.93	−.87	−.49	+.79	+1
normal	.038	.026	.008	.030	.140	.113	.646
probe	.574	.082	.010	.014	.013	0	.307

	−1	−.95	−.84	−.73	−.71	−.69	−.61	−.60	−.39	−38	−.28	+1
normal	.057	0	.001	.001	.001	.005	0	.075	0	.228	.005	.627
probe	.620	.062	.003	0	.224	0	.082	.001	.001	0	.005	.001

Figure 3.2: *A simplistic Bayesian belief network for the detection of network probes. The class attribute 'connection_type' has edges to the two other attributes 'source_data' and '%_of_connections_with_same_destination_port_-and_service'. They contain the tables with the probability distributions to predict the class probabilities for a given instance.*

The applicable learning method for Bayesian networks depends on the availability of network structure and variables. The network structure may be either given in advance, be directly inferred from data, or be created by experts. The variables can be either fully observable, partially hidden, or entirely hidden.

If the network structure and variables are fully observable, the conditional probability table entries can be estimated. When the network structure and some variables are given, the gradient ascent algorithm can be applied in order to search through the space of all possible conditional probability table entries. Thus, a set of possible conditional probability table entries can be learned.

Bayesian learning is difficult if the network structure is unknown. In this case, a search through the space of all possible networks is required. A simple and very fast heuristic search algorithm called K2 can be applied if the data is fully observable.

K2 starts with a given attribute order where each attribute represents a node. The algorithm performs a greedy search, adding edges from previously processed nodes to the current node. For all edge combinations that can be added, K2 calculates the total network score. After processing the node, K2 keeps the edges that score highest. Then K2 continuous processing the next node until the final node is reached. To find a good Bayesian network, K2 needs to run several times with different random orders of attributes.

Besides being computationally intensive, the main advantage of the Bayesian networks classifier compared to the naïve Bayes is that it is less constraining. Bayesian networks can easily be interpreted by humans, and the estimates obtained can be ranked, which allows the cost to be minimised. For more information about Bayesian networks, see [Heckerman *et al.* 1995].

3.4 Backpropagation Neural Networks

Artificial neural networks are inspired by biological learning systems and loosely model their basic functions. They consist of a densely interconnected group of simple neuron-like threshold switching units. Each unit takes a number of real-valued inputs and produces a single real-valued output. Based on the connectivity between the threshold units and element parameters, these networks can model complex global behaviour. The corresponding chapters in [Mitchell 1997] and [Han & Kamber 2006] provide excellent introductions to neural networks. [Rumelhart *et al.* 1994] provide a survey of practical applications.

3.4.1 The Perceptron

The most basic type of artificial neuron is called a *perceptron*. Perceptrons consist of a number of external input links, a threshold, and a single output link. Additionally, perceptrons have an internal input, w_0, called *bias* that is always fixed at a value of '1'.

The perceptron takes a vector of real-valued input values, all of which are weighted by a multiplier. In a previous perceptron training phase, the perceptron learns these weights on the basis of training data. It sums all weighted input values and 'fires' if the resultant value is above a pre-defined threshold. The output of the perceptron is always Boolean, and it is considered to have fired if the output is '1'. The deactivated value of the perceptron is '-1', and the threshold value is, in most cases, '0'.

The perceptron output o, given the inputs $x_1, ..., x_n$ and trained weights $w_1, ..., w_n$, is computed as follows:

$$o = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_n x_n > 0; \\ -1 & \text{otherwise.} \end{cases}$$

The constant internal input x_0 of value '1' is associated with the weight w_0. For the perceptron to fire, the sum of the weighted inputs $w_1 x_1 + ... + w_n x_n$ must exceed the value $-w_0$ of the threshold.

Single perceptron units can already represent a number of useful functions. Examples are the Boolean functions AND, OR, NAND and NOR. Other functions are only representable using networks of neurons. Single perceptrons are limited to learning only functions that are linearly separable. In general, a problem is linear and the classes are linearly separable in an n-dimensional space if the decision surface is an $(n-1)$-dimensional hyperplane.

The general structure of a perceptron is shown in Figure 3.3.

3.4.2 Linear Separability

To understand linear separability, it is helpful to visualise the possible inputs of a perceptron on the axes of a two-dimensional graph. Figure 3.4 shows representations of the Boolean functions AND and XOR. The AND function is linearly separable, whereas the XOR function is not. In the figure, pluses are used for an input where the perceptron fires and minuses, where it does not. If the pluses and minuses can be completely separated by a single line, the problem is linearly separable. The weights of the trained perceptron should represent that line.

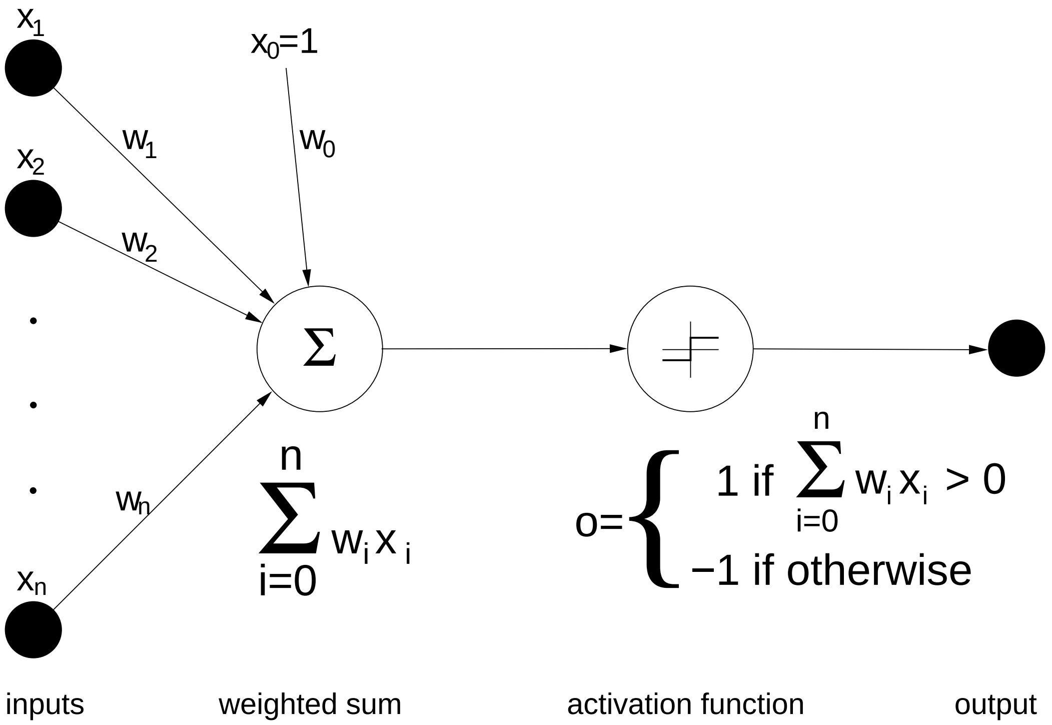


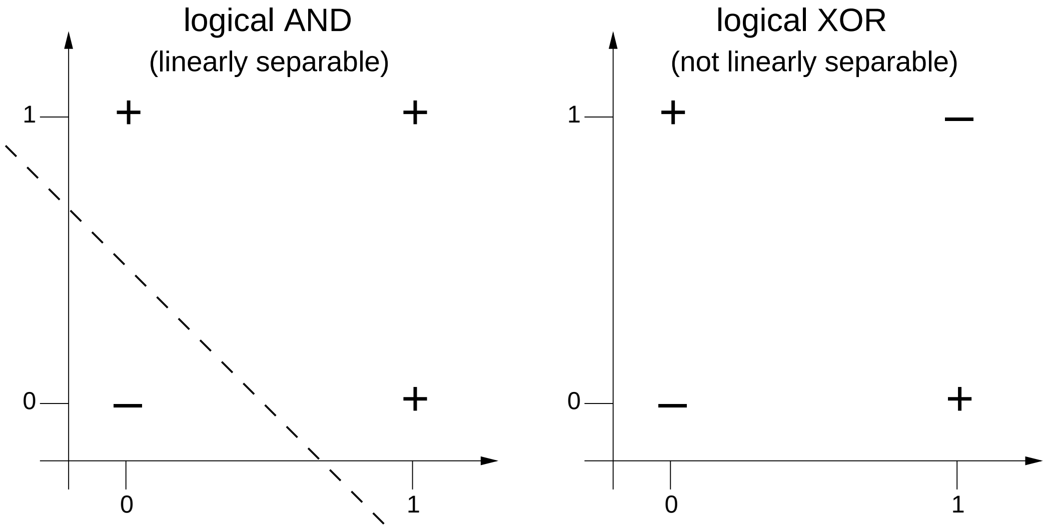
Figure 3.3: *The general structure of the most basic type of artificial neuron, called* a perceptron*. Single perceptrons are limited to learning linearly separable functions.*

3.4.3 The Perceptron and Delta Learning Rule

Perceptron training is learning by imitation, which is called 'guided learning'. During the training phase, the perceptron produces an output and compares it with a derived output value provided by the training data. In cases of misclassification, it then modifies the weights accordingly. [Minsky & Papert 1969] show that in a finite time, the perceptron will converge to reproduce the correct behaviour, provided that the training examples are linearly separable. Convergence is not assured if the training data is not linearly separable.

A variety of training algorithms for perceptrons exist, of which the most common are the perceptron learning rule and the delta learning rule. Both start with random weights and both guarantee convergence to an acceptable hypothesis.

Using the perceptron learning rule algorithm, each weight w_i associated



Input$_1$	Input$_2$	Output
1	1	1
1	0	1
0	1	1
0	0	0

Input$_1$	Input$_2$	Output
1	1	0
1	0	1
0	1	1
0	0	0

Figure 3.4: *Representations of the Boolean functions AND and XOR. The figures show that the AND function is linearly separable, whereas the XOR function is not.*

with input x_i is modified at every step using the rule:

$$w_{i+1} \leftarrow w_i + \Delta w_i$$

with

$$\Delta w_i = \eta(t - o)x_i$$

where t is the target output of the current example, o is the output generated by the perceptron, and η is the learning rate. The learning rate is a constant that controls the degree to which the weights are changed. The algorithm will only converge towards an optimum if the training data is linearly separable, and the learning rate is sufficiently small. The perceptron rule fails if the training examples are not linearly separable.

The delta learning rule was specifically designed to handle linearly separable and linearly non-separable training examples. It also calculates the errors between calculated output and output data from training samples, and modifies the weights accordingly. The modification of weights is achieved by using the gradient optimisation descent algorithm, which alters them in the direction that produces the steepest descent along the error surface towards the global minimum error. The delta learning rule is the basis of the error backpropagation algorithm, which we will discuss later in this section.

3.4.4 The Sigmoid Threshold Unit

The sigmoid threshold unit is a different kind of artificial neuron, very similar to the perceptron. It computes a linear combination of all weighted input *net* that can include an optional bias θ. Additionally, there is an optional positive constant k multiplied by the result, which determines the steepness of the bias. The output o is computed as:

$$o_j = \frac{1}{(1 - e^{-k \times net_j})}$$

with

$$net_j = \sum_i w_i x_i + \theta_j.$$

The major effect on the perceptron is that the output function of the sigmoid threshold unit is a sigmoid function. The threshold output is a continuous function of its input, which ranges between 0 and 1. It is often referred to as the ‘squashing’ function, because it maps a very large input domain onto a small range of outputs. For a low total input value, the output of the sigmoid function is close to zero, whereas it is close to one for a high total input value. The slope of the sigmoid function is adjusted by the threshold value.

The advantage of neural networks using sigmoid units is that they are capable of representing non-linear functions. Cascaded linear units, like the perceptron, are limited to representing linear functions. A sigmoid threshold unit is sketched in Figure 3.5.

3.4.5 Feed-Forward Networks and Backpropagation

In feed-forward neural networks (FFNNs), sets of neurons are organised in layers, where each neuron computes a weighted sum of its inputs. Input

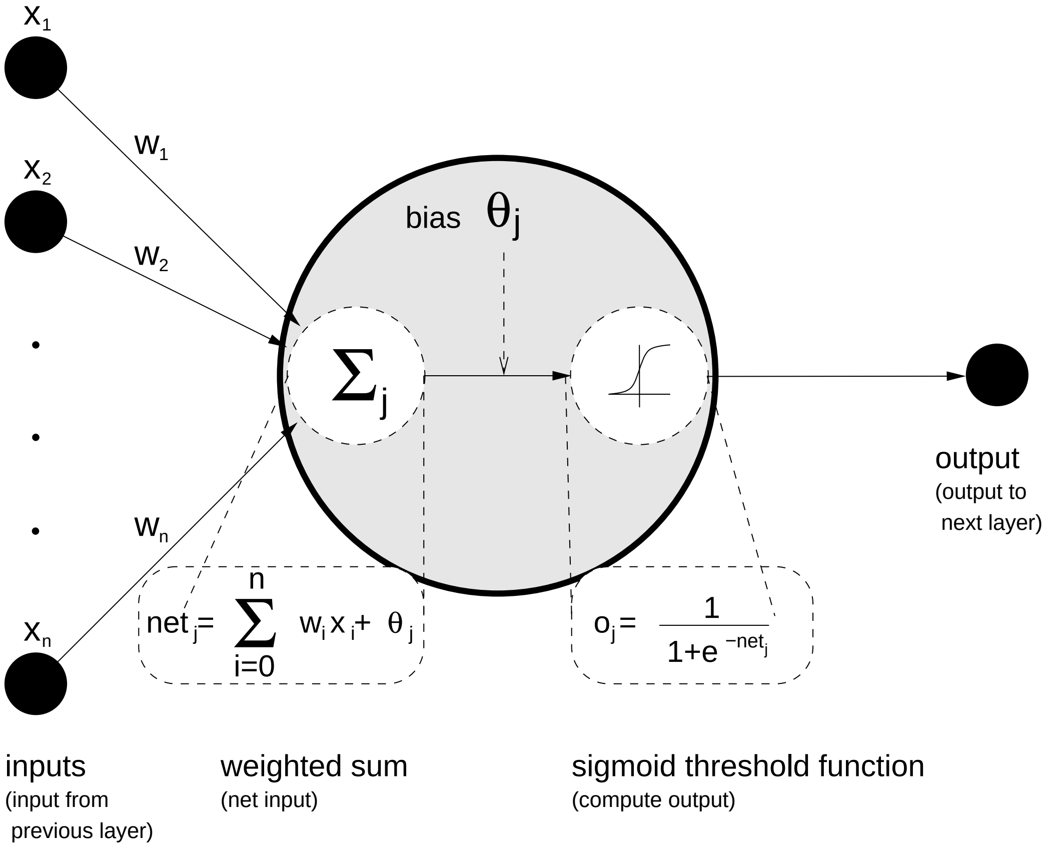


Figure 3.5: *The sigmoid threshold unit is capable of representing non-linear functions. Its output is a continuous function of its input, which ranges between 0 and 1.*

neurons take signals from the environment, and output neurons present signals to the environment. Neurons that are not directly connected to the environment, but which are connected to other neurons, are called *hidden* neurons.

Feed-forward networks are loop-free and fully connected. This means that each neuron provides the input to each neuron in the following layer, and that none of the weights give an input to a neuron in a previous layer.

The simplest type of neural feed-forward networks are single-layer perceptron networks. Single-layer neural networks consist of a set of input neurons, defined as the *input layer*, and a set of output neurons, defined as the *output layer*. The outputs of the input-layer neurons are directly connected to the neurons of the output layer. The weights are applied to the connections between the input and output layer.

In the single-layer perceptron network, every single perceptron calculates the sum of the products of the weights and the inputs. The perceptron fires '1' if the value is above the threshold value; otherwise, the perceptron takes the deactivated value, which is usually '-1'. The threshold value is typically zero.

Sets of neurons organised in several layers can form multilayer, forward-connected networks. The input and output layers are connected via at least one hidden layer, built from set(s) of hidden neurons. The multilayer feed-forward neural network sketched in Figure 3.6, with one input layer and three output layers (two hidden and one output), is classified as a 3-layer feed-forward neural network. For most problems, feed-forward neural networks with more than two layers offer no advantage.

Multilayer feed-forward networks using sigmoid threshold functions are able to express non-linear decision surfaces. Any function can be closely approximated by these networks, given enough hidden units.

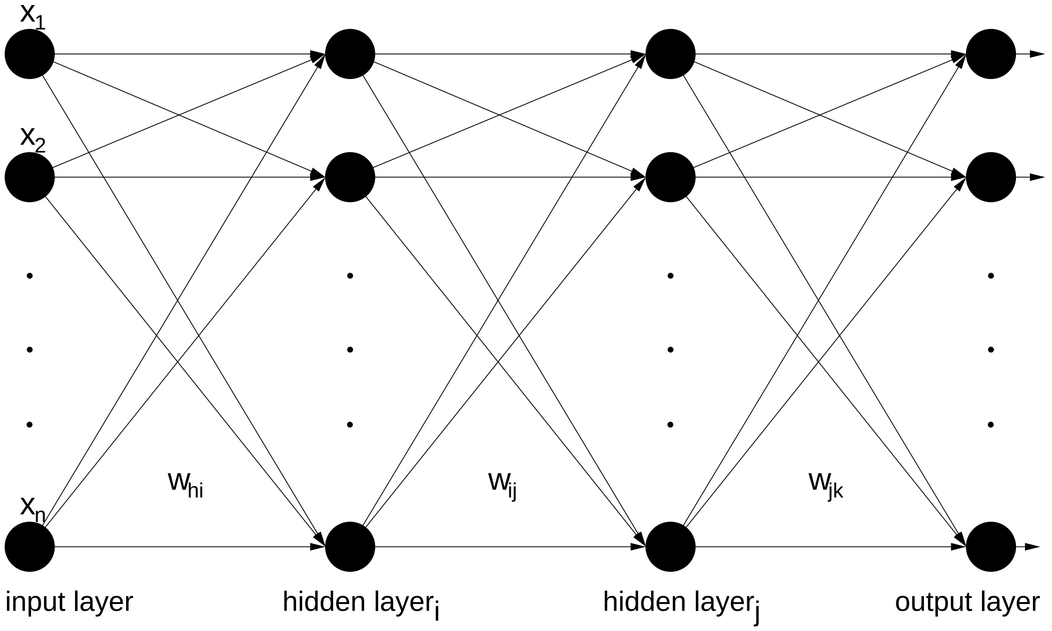


Figure 3.6: *A multilayer feed-forward neural network with one input layer and three output layers. Using neurons with sigmoid threshold functions, these neural networks are able to express non-linear decision surfaces.*

The most common neural network learning technique is the error backpropagation algorithm. It uses gradient descent to learn the weights in multilayer networks. It works in small iterative steps, starting backwards from the output layer towards the input layer. A requirement is that the activation function of the neuron is differentiable.

Usually, the weights of a feed-forward neural network are initialised to small, normalised random numbers using bias values. Then, error backpropagation applies all training samples to the neural network and computes the input and output of each unit for all (hidden and) output layers.

The inputs are propagated forwards through the network, starting from the input layer. Let j be some network unit, and let k be some non-input unit. If j is an input unit, the actual output value is equal to its input $o_j = net_j$. Given the network input net_j, of non-input unit k, the output o_k, using the sigmoid activation function, is computed as:

$$o_k = \frac{1}{1 + e^{-net_k}}$$

with

$$net_k = \sum_k w_{jk} o_j + \theta_k$$

where o_j are the outputs from the previous layer connected to unit k with their corresponding weights w_{jk}.

Next, the backpropagation learning algorithm propagates the error backwards, and the weights and biases are updated in order to reflect the error. Starting from the output layer, it compares the network output o_k with the corresponding desired target output t_k. It calculates the error e_k for each output neuron using some error function to be minimised. Using the mean squared error, the error e_k is computed as:

$$e_k = o_k(1 - o_k)(t_k - o_k)$$

Next, we calculate the error signal of each unit in preceding layers as follows:

$$e_j = o_j(1 - o_j) \sum_k e_k w_{jk}$$

Finally, we update the increments for each weight, $w_{ij} \leftarrow w_{ij} + e_j o_i$, and the biases, $\theta_j \leftarrow \theta_j + e_j$.

This process repeats itself until all network outputs are within an acceptable range, or some other terminating condition is reached.

3.5 Support Vector Machines

The basic principles for *support vector machines* (SVMs) were developed by [Boser *et al.* 1992] and [Cortes & Vapnik 1995]. Support vector machines can classify two-class problems with both linearly and non-linearly separable data. For linearly separable data, the support vector machine searches for an optimal linear hyperplane separating the tuples of the two classes. In the case of non-linear data, the support vector machine uses a non-linear mapping of the input vectors from the input space to classify the training data into a higher dimension. Such non-linear mapping is determined by a kernel function and always exists. Within this dimension, it finds the optimal linear separating hyperplane. The corresponding chapter on SVMs by [Han & Kamber 2006] provides an excellent introduction to this field. For more detailed information on the SVM classifier, see [Platt 1999] and [Keerthi *et al.* 2001].

3.5.1 The Maximum Marginal Hyperplane

Consider the simple case of a linearly separable problem with two classes, where the training data is given in the form (X, y), where X is the training tuple, and y is the associated class membership. The training tuple is presented repeatedly to the classifier in order for it to learn the decision function $D(x)$.

The decision function can be written as:

$$D(x) = \sum_{i=1}^{N} w_i \varphi_i(x) + b$$

where w_i is the weights, with n as the number of attributes, and φ_i is an identity function of x. For $D(x) > 0$, pattern x belongs to one class, and for $D(x) \leq 0$, pattern x belongs to the other class.

Once learned, the decision function can be used to classify previously unseen data.

In this context, $D(x)$ is referred to as the decision surface or separating hyperplane. All points x that lie on this decision surface satisfy the equation:

$$WX + b = 0$$

where W is the weight vector $W = w_1, w_2, ..., w_n$, with n attributes. X is the training tuple $X = (x_1, x_2)$, with x_1 and x_2 being the values of the attributes. b is the bias.

Ideally, the hyperplane will separate the two classes such that $WX + b > 0$ defines all points belonging to one class, and $WX + b \leq 0$ those belonging to the other class.

The input vectors can be represented as dots in a 2-dimensional space, where the hyperplane can be represented as a straight line. It splits the input vectors into the two classes they might belong to.

The SVM algorithm searches for the hyperplane with the maximum margin property. This special maximum-margin hyperplane maximises the smallest distance of the closest training samples of both classes to the hyperplane.

The separating hyperplane is constructed from selected data points that are close to the borders of the classes. They are called support vectors and reside on the boundary margin. The accuracy and speed of a support vector machine is determined by the number of chosen support vectors. A small number of required support vectors indicates that the data can be separated well.

Let us consider a linearly separable case with two classes identified by $y_i \in \{1, -1\}$. The shortest distance between the separating hyperplane and the closest positive (negative) pattern in the training set is $d_+(d_-)$. This value is defined as the ‘margin’, M, which the SVM algorithm tries to maximise during training.

Figure 3.7 shows two classes separated by a maximum margin hyperplane with the decision function $D(x)$. The distance from the hyperplane to any point on H_1 or H_2 is maximal. The support vectors are circled with a thicker border.

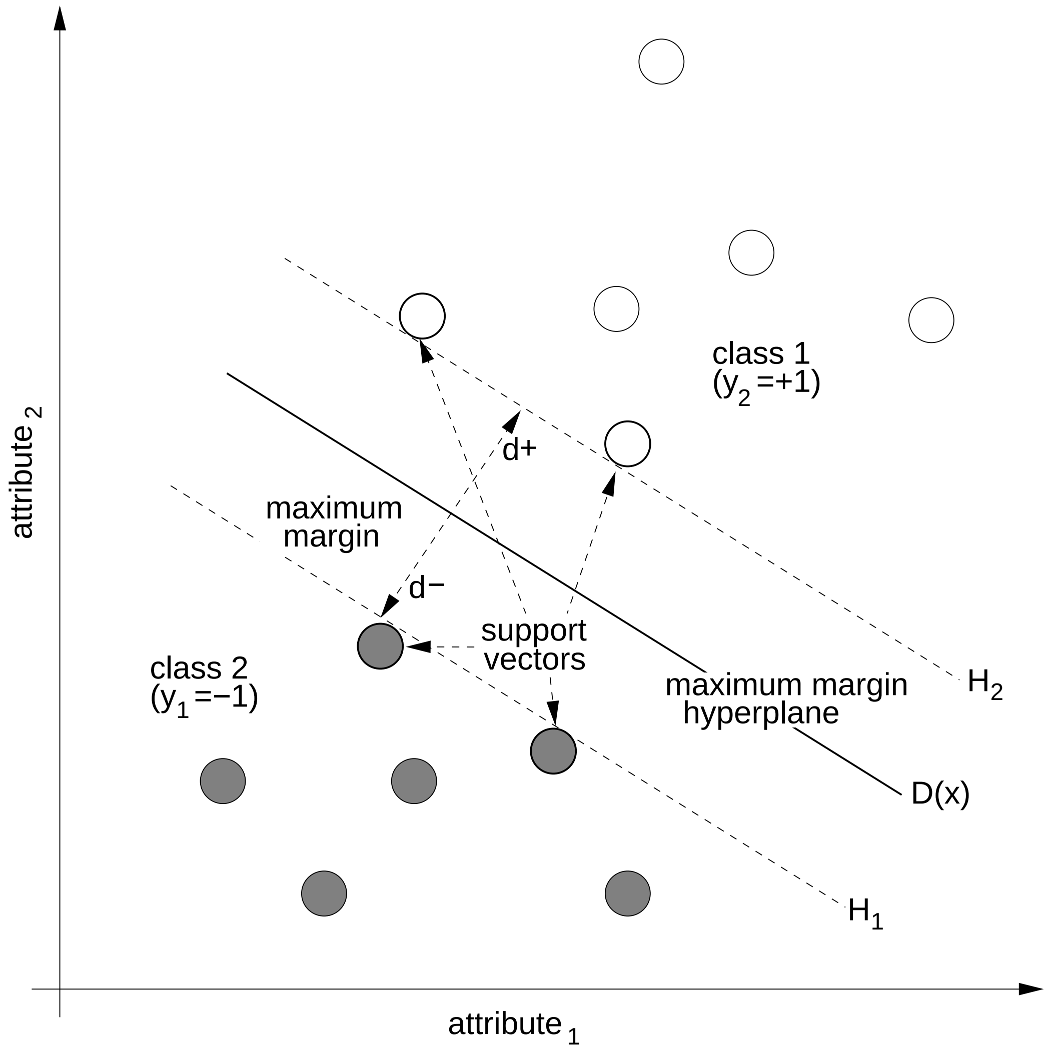


Figure 3.7: *The two classes are separated by the maximum margin hyperplane with the decision function $D(x)$. The figure shows the support vectors on H_1 and H_2 with a thicker border. The distance from the hyperplane to any point on the support vectors is maximal.*

In the linearly separable case, all training data will adhere to the following two constraints:

$$wx_i + b \geq +1 \text{ (for } y_i = +1)$$

$$wx_i + b \leq -1 \text{ (for } y_i = -1)$$

A vector w and scalar b therefore exist such that:

$$y_i(wx_i + b) - 1 \geq 0, i \in \{1, ..., l\}$$

We can define the hyperplanes $H_1 : wx_i + b = 1$ and $H_2 : wx_i + b = -1$, which are parallel to the separating hyperplane, but on opposite sides. The tuples that fall on these two hyperplanes are called support vectors. The perpendicular distance from H_1 and H_2 to the separating hyperplane is, therefore, $\frac{|1-b|}{||w||}$ and $\frac{|-1-b|}{||w||}$. It follows from this that $d+ = d- = \frac{1}{||w||}$, and margin $M = \frac{2}{||w||}$.

3.5.2 The Soft-Margin Method

Most real-world problems do not contain linearly separable data. The SVM algorithm can be extended with the so-called 'soft-margin' method in order to maximise the margin and, at the same time, minimise the upper limits of the error originating from misclassification. For this, we introduce the positive variables $\xi_i \geq 0, i = 1, ..., l$ called 'slack' variables, so that we have the following constraints:

$$y_i(wx_i + b) \geq 1 - \xi_i, i \in \{1, ..., l\}$$

where $\xi \geq 0, \forall i$.

Under the new conditions, ξ_i must exceed unity for an error to occur, and $\sum_i \xi_i$ represents an upper limit to the number of training errors.

The trade-off between margin and classification error is driven by an introduced constant, C. There is no known straightforward method for choosing this constant, although a number of supporting techniques do exist.

3.5.3 Kernel Functions

Another method exists that allows non-linear decision boundaries. For this, we need to transform the input vectors into a feature space of a higher dimension. The search for the linear separating hyperplane is done within that space. A linear hyperplane in the higher-dimension feature space corresponds to a non-linearly separating hypersurface in the original space.

To classify the data in a higher-dimension feature space, where it is linearly separable, the data needs to be transformed by a non-linear transformation, $\Phi : x \rightarrow x'$. This is illustrated in Figure 3.8.

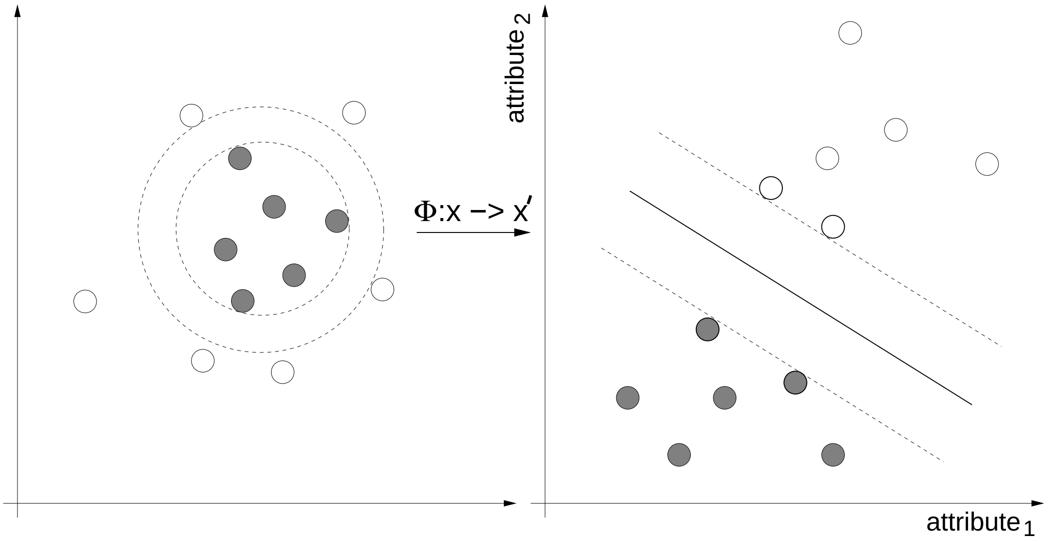


Figure 3.8: *Classification of non-linearly separable data in a higher-dimension feature space. The data needs to be transformed by a non-linear transformation $\Phi : x \rightarrow x'$. A linear hyperplane in the higher-dimension feature space corresponds to a non-linearly separating hypersurface in the original space.*

The transformation of the data points to a higher-dimension feature space comes at significant cost. A mathematical technique called the kernel 'trick' allows us to search for a linear hyperplane in new, higher-dimension feature spaces without transforming the data points.

3.5.4 The Kernel Trick

When searching for a linear SVM in a higher-dimension space, the training tuples $X = (x_1, x_2, ..., x_n)$ appear as dot products of two non-linear transformations $\Phi(X_i) \times \Phi(X_j)$. Fortunately, it is mathematically equivalent to applying a kernel function on the training tuple:

$$K(X_i, X_j) = \Phi(X_i) \times \Phi(X_j).$$

By using a kernel function, we can avoid the higher-dimension mapping and perform all calculations in the original feature space. Interestingly, this works even without knowledge of the correct mapping. Well-studied kernel functions, each resulting in a different non-linear classifier, include:

$$K(X_i, X_j) = \begin{cases} (X_i \times X_j + 1)^h \text{ (polynomial kernel of degree } h) \\ e^{-||X_i - X_j||^2/2\sigma^2} \text{ (Gaussian radial basis function kernel)} \\ \tan h(\kappa X_i \times X_j - \delta) \text{ (sigmoid kernel).} \end{cases}$$

There is no way to determine which kernel function will provide the best performance for a given dataset. In general, all kernel functions provide similar performance. An advantage of SVMs is that they always find a global solution. Neural networks, by contrast, can get stuck in a local minimum.

3.6 Recurrent Neural Networks

Recurrent neural networks (RNNs) are dynamic systems; they have an internal state at each time step of the classification. This is due to circular connections between higher- and lower-layer neurons and optional self-feedback connections. These feedback connections enable RNNs to propagate data from earlier events to current processing steps. Thus, RNNs build a memory of time series events.

3.6.1 Basic Architecture

RNNs range from partly to fully connected, and two simple RNNs are suggested by [Jordan 1986] and [Elman 1990]. The Elman network is similar to a three-layer neural network, but additionally, the outputs of the hidden layer are saved in so-called ‘context cells’. The output of a context cell is circularly

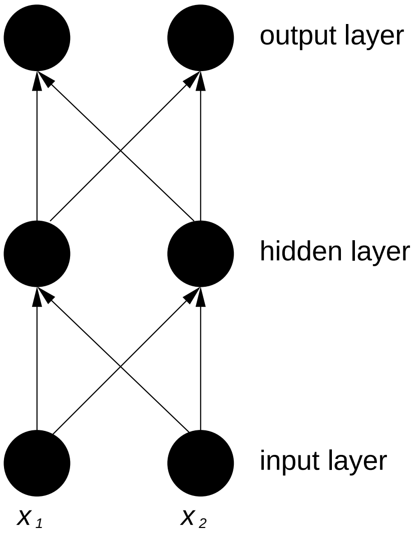


Figure 3.9: *This figure shows a feed-forward neural network.*

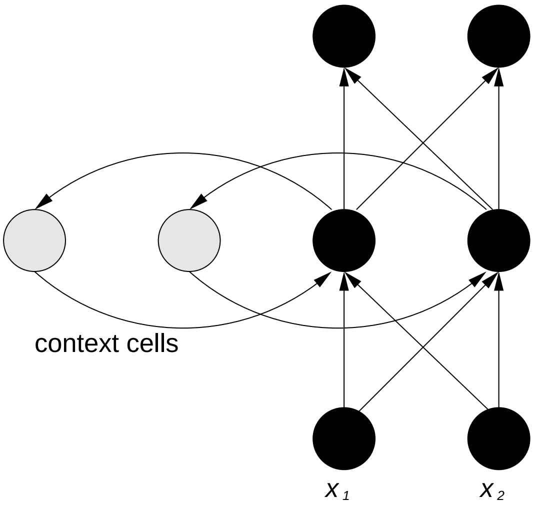


Figure 3.10: *This figure shows an Elman neural network.*

fed back to the hidden neuron along with the originating signal. Every hidden neuron has its own context cell and receives input both from the input layer and the context cells. Elman networks can be trained with standard error backpropagation, the output from the context cells being simply regarded as an additional input. Figures 3.9 and 3.10 show a standard feed-forward network in comparison with such an Elman network.

Jordan networks have a similar structure to Elman networks, but the context cells are instead fed by the output layer. A partial recurrent neural

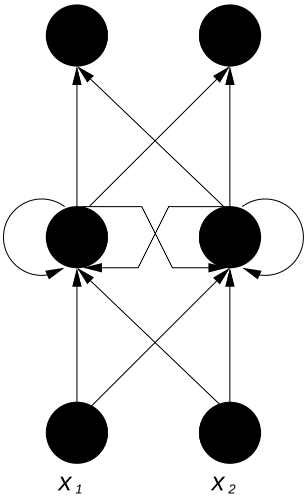


Figure 3.11: *This figure shows a partially recurrent neural network with self-feedback in the hidden layer.*

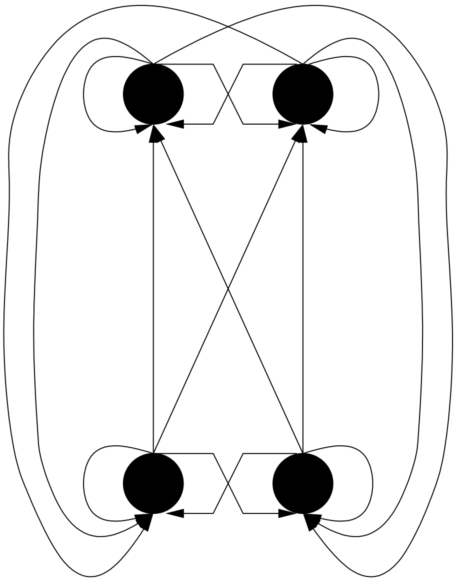


Figure 3.12: *This figure shows a fully recurrent neural network (RNN) with self-feedback connections.*

network with a fully connected recurrent hidden layer is shown in Figure 3.11. Figure 3.12 shows a fully connected RNN.

RNNs need to be trained differently to the feed-forward neural networks (FFNNs) described in Section 3.4.5. The most common and well-documented learning algorithms for training RNNs in temporal, supervised learning tasks are *backpropagation through time* (BPTT) and *real-time recurrent learning*

(RTRL). In BPTT, the network is unfolded in time to construct an FFNN. Then, the generalised delta rule is applied to update the weights. This is an offline learning algorithm in the sense that we first collect the data and then build the model from the system. In RTRL, the gradient information is forward propagated. Here, the data is collected online from the system and the model is learned during collection. Therefore, RTRL is an online learning algorithm.

3.6.2 Backpropagation Through Time

The BPTT algorithm makes use of the fact that, for a finite period of time, there is an FFNN with identical behaviour for every RNN. To obtain this FFNN, we need to unfold the RNN in time. Figure 3.13 shows a simple, fully recurrent neural network with a single two-neuron layer. The corresponding feed-forward neural network, shown in Figure 3.14, requires a separate layer for each time step with the same weights for all layers. If weights are identical to the RNN, both networks show the same behaviour.

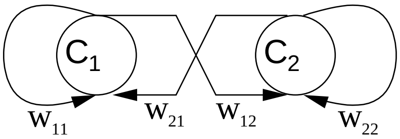


Figure 3.13: *A simple fully recurrent neural network with a two-neuron layer is shown in this figure.*

The unfolded network can be trained using the backpropagation algorithm described in Section 3.4.5. At the end of a training sequence, the network is unfolded in time. The error is calculated for the output units with existing target values using some chosen error measure. Then, the error is injected backwards into the network and the weight updates for all time steps calculated. The weights in the recurrent version of the network are updated with the sum of its deltas over all time steps.

We calculate the error signal for a unit for all time steps in a single pass, using the following iterative backpropagation algorithm. We consider discrete time steps $t = 1, 2, 3, ...$. The network starts at a point in time t_0 and runs until a final time t_1. For an intermediate time t we define $t_0 \leq t \leq t_1$. A single time

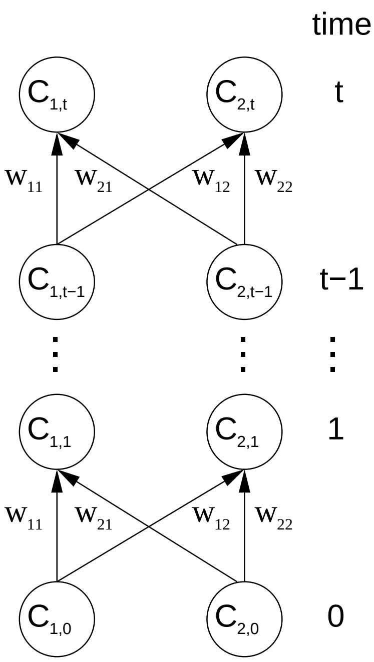


Figure 3.14: *Here, we have a feed-forward neural network with a separate layer for each time step.*

step involves the update of all units and the computation of all error signals. Let $x_k^{net}(t)$ be an m-tuple of input units $I = x_k(t), 0 < k < m$, and $y_k(t)$ be an n-tuple of non-input units $U = y_k(t), 0 < k < n$, in a fully connected recurrent neural network. Concatenating $x_k^{net}(t)$ and $y_k(t)$, all network units are an $m + n$-tuple defined as

$$x_k(t) = \begin{cases} x_k^{net}(t) & \text{if } k \in I \\ y_k(t) & \text{if } k \in U \end{cases} \tag{3.1}$$

Let f_i be the differentiable, non-linear squashing function of non-input unit $i \in U$, so that the output $y_i(t)$, at time step t, is given by

$$y_i(t) = f_i(net_i(t)) \tag{3.2}$$

with the weighted input

$$net_i(t) = \sum_j w_{ij} y_j(t-1) \tag{3.3}$$

where $y_j(t-1)$ are the outputs from the previous time step of units connected to i, with their corresponding weights w_{ij}.

We denote $T(t)$ as the set of indices $k \in U$ for which target values $d_k(t)$ matching the output $y_k(t)$ at time t should exist. Cost function is the summed error $E_{total}(0,t)$ from time t_0 to time t_1, which we want to minimise using a learning algorithm. It is defined as

$$E_{total}(t_0,t_1) = \sum_{\tau=t_0+1}^{t_1} E(\tau), \tag{3.4}$$

where the total error at time t, using the squared error as an objective function, is

$$E(t) = \frac{1}{2}\sum_{k\in U}(e_k(t))^2, \tag{3.5}$$

with the error of an arbitrary output unit k at time t

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{if } k \in T(t), \\ 0 & \text{otherwise.} \end{cases} \tag{3.6}$$

The error signal of a non-input unit i, at a time τ, is

$$\delta_i(\tau) = -\frac{\partial E_{total}(0,t)}{\partial net_i(\tau)}.$$

To get the backpropagated error $\delta_i(\tau)$ for all $i \in U$, $t_0 \leq \tau \leq t_1$ in a single pass, we can calculate

$$\delta_i(\tau) = \begin{cases} f_i'(net_i(\tau))e_i(\tau) & \text{if } \tau = t, \\ f_i'(net_i(\tau))\left(e_i(\tau) + \sum_{l\in U}^{n} w_{li}\delta_l(\tau+1)\right) & \text{if } t_0+1 \leq \tau < t \end{cases}. \tag{3.7}$$

After the backpropagation computation is performed down to time 1, we can now calculate the weight update Δw_{ij} in the recurrent version of the network. This is done by summing the corresponding weight updates for all time steps:

$$\Delta w_{ij}(t_1) = -\frac{\partial E_{total}(t_0, t_1)}{\partial w_{ij}} = -\sum_{\tau=t_0+1}^{t_1} \delta_i(\tau) x_j(\tau - 1).$$

BPTT is described in more detail in [Werbos 1990], [Rumelhart *et al.* 1986] and [Williams & Zipser 1995].

3.6.3 Real-Time Recurrent Learning

The RTRL algorithm does not require error propagation. All the information necessary to compute the activity gradient is collected as the input stream is presented to the network. This makes a dedicated training interval obsolete. The algorithm comes at significant computational cost per update cycle, and the stored information is non-local. But the memory required depends only on the size of the network.

Following the notation from the previous section, we will now define for the network units $k \in U$, $i \in U$ and $j \in U \cup I$, and the time steps $t_0 \leq t \leq t_1$. The training objective is to minimise the overall network error, which is given by Equations 3.4, 3.5 and 3.6. The overall network error at time step t is

$$E(t) = \frac{1}{2} \sum_{k \in U} (d_k(t) - y_k(t))^2.$$

We conclude from Equation 3.4 that the gradient of the total error is also the sum of the gradient for all previous time steps and the current time step:

$$\nabla_w E_{total}(t_0, t+1) = \nabla_w E_{total}(t_0, t) + \nabla_w E(t+1).$$

During presentation of the time series to the network, we need to accumulate the values of the gradient at each time step. Thus, we can also keep track of the weight changes $\Delta w_{ij}(t)$. After presentation, the overall weight change for w_{ij} is then given by

$$\Delta w_{ij} = \sum_{t=t_0+1}^{t_1} \Delta w_{ij}(t). \tag{3.8}$$

To get the weight changes we need to calculate

$$\Delta w_{ij}(t) = -\frac{\partial E(t)}{\partial w_{ij}}$$

for each time step t. After expanding this equation via gradient descent and by applying Equation 3.5, we find that

$$\begin{aligned} \Delta w_{ij}(t) &= -\sum_{k \in U} \frac{\partial E(t)}{\partial y_k(t)} \frac{\partial y_k(t)}{\partial w_{ij}} \\ &= \sum_{k \in U} (d_k(t) - y_k(t)) \left(\frac{\partial y_k(t)}{\partial w_{ij}} \right). \end{aligned} \tag{3.9}$$

Since the error $e_k(t) = d_k(t) - y_k(t)$ is always known, we need to find a way to calculate the second factor only. We define the quantity

$$p_{ij}^k(t) = \frac{\partial y_k(t)}{\partial w_{ij}}, \tag{3.10}$$

which measures the sensitivity of the output of unit k at time t to a small change in the weight w_{ij}, in due consideration of the effect of such a change in the weight over the entire network trajectory from time t_0 to t. The weight w_{ij} does not have to be connected to unit k, which makes the algorithm non-local. Local changes in the network can have an effect anywhere in the network.

In RTRL, the gradient information is forward-propagated. Using Equation 3.1, and analogous to Equations 3.2 and 3.3, the output $y_i(t+1)$ at time step $t+1$ is given by

$$y_i(t+1) = f_i(net_i(t)) \tag{3.11}$$

with the weighted input

$$net_i(t) = \sum_j w_{ij} x_j(t). \tag{3.12}$$

By differentiating Equations 3.10, 3.11 and 3.12, we can calculate results for all time steps $\geq t+1$ with

$$\begin{aligned} p_{ij}^{k}(t+1) = \frac{\partial y_k(t+1)}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}} \left[f_k \left(\sum_{l \in U} w_{kl} x_l(t) \right) \right] \\ &= f_k'(y_k(t)) \left[\sum_{l \in U} w_{kl} \frac{\partial x_l(t)}{\partial w_{ij}} + \sum_{l \in U} \frac{\partial w_{kl}}{\partial w_{ij}} x_l(t) \right] \end{aligned}$$

and using Equation 3.10 and that after labelling the Kronecker delta with

$$\delta_{ik} = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{if } otherwise, \end{cases}$$

we finally get

$$p_{ij}^{k}(t+1) = f_k'(y_k(t)) \left[\sum_{l \in U} w_{kl} p_{ij}^{l}(t) + \delta_{ik} x_j(t) \right]. \tag{3.13}$$

Assuming that the initial state of the network has no functional dependency on the weights, the derivative for the first time step is

$$p_{ij}^{k}(t_0) = \frac{\partial y_k(t_0)}{\partial w_{ij}} = 0. \tag{3.14}$$

Knowing the value for time t_0 from Equation 3.14, we can now calculate the quantities p_{ij}^{k} for the first and all subsequent time steps using Equation 3.13. Combining these values with the error vector $e(t)$ for that time step, using Equation 3.9, we can finally calculate the negative error gradient $\bigtriangledown wE(t)$. The final weight change for w_{ij} can be calculated using Equations 3.9 and 3.8.

A more detailed description of the RTRL algorithm is given in [Williams & Zipser 1989] and [Williams & Zipser 1995].

3.6.4 The Vanishing Error Problem

Standard RNN cannot bridge more than 5–10 time steps. Error signals tend to either blow-up or vanish. Blown-up error signals lead straight to oscillating weights, whereas with a vanishing error, learning takes an unacceptable amount of time, or does not work at all.

Given a fully recurrent neural network with an n-tuple of non-input units U, the local error signal that occurs at any chosen output-layer neuron $k \in U$, at time-step t, is propagated back through time for $t-s$ time-steps, with $s < t$ to an arbitrary neuron v. This causes the error to be scaled by the following factor:

$$\frac{\partial \delta_v(s)}{\partial \delta_k(t)} = \begin{cases} f'_v(net_v(t-1))w_{kv} & \text{if t-s=1,} \\ f'_v(net_v(s))\left(\sum_{l=1}^{n} \frac{\partial \delta_l(s+1)}{\partial \delta_k(t)} w_{lv}\right) & \text{if t-s>1} \end{cases}$$

To solve the above equation, we unroll it over time. For $s < \tau < t$, let l_τ be the index of a non-input-layer neuron in one of the replicas in the unrolled network at time τ. Now, by setting $l_s = v$ and $l_t = k$, we can solve the equation through proof by induction:

$$\frac{\partial \delta_v(s)}{\partial \delta_k(t)} = \sum_{l_{t-1}=1}^{n} \dots \sum_{l_{s-1}=1}^{n} \left(w_{l_t l_{t-1}} \left(\prod_{\tau=t-1}^{s+1} f'_{l_r}(net_{l_r}(\tau) w_{l_r l_{r-1}} \right) f'_{l_s}(net_{l_s}(s)) \right) \tag{3.15}$$

By using the next equation, we can show that if the local error vanishes, the global error also vanishes:

$$\sum_{k \in O} \frac{\partial \delta_v(s)}{\partial \delta_k(t)}$$

where O is the set of output units.

Observing Equation 3.15, it follows that if

$$|f'_{l_\tau}(net_{l_\tau}(\tau)) w_{t_\tau l_{\tau-1}}| > 1 \tag{3.16}$$

for all τ, then the product will grow exponentially with $t-s-1$. The error blows-up, and conflicting error signals arriving at neuron v can lead to oscillating weights and unstable learning. If now

$$|f'_{l_\tau}(net_{l_\tau}(\tau)) w_{l_\tau l_{\tau-1}}| < 1 \tag{3.17}$$

for all τ, then the product decreases exponentially with $t-s-1$. In this case, the error vanishes and nothing can be learned within an acceptable time period.

A more detailed theoretical analysis of the problem with long-term dependencies is presented in [Hochreiter *et al.* 2001]. The paper also briefly outlines several proposals on how to address this problem.

3.7 LSTM Recurrent Neural Networks

One solution that addresses the vanishing error problem is a gradient-based method called *long short-term memory* (LSTM) published by [Hochreiter & Schmidhuber 1996], [Hochreiter & Schmidhuber 1997], [Gers *et al.* 1999] and [Gers *et al.* 2002]. LSTM can learn how to bridge minimal time lags of more than 1,000 discrete time steps. The solution uses *constant error carousels* (CECs), which enforce a constant error flow within special cells. Access to the cells is handled by multiplicative gate units, which learn when to grant access.

3.7.1 Constant Error Carousel

The local error back flow of a single cell j at a single time-step t follows from Equation 3.7 and is given by

$$\delta_j(t) = f_j'(net_j(t))w_{jj}\delta_j(t+1)$$

with the weight w_{jj}, which connects the cell to itself. It follows from Equations 3.16 and 3.17 that, in order to ensure a constant error flow through cell j, we need to have

$$|f_j'(net_j)(t))w_{jj}| = 1.0.$$

and by integration we get

$$f_j(net_j(t)) = \frac{net_j(t)}{w_{jj}}.$$

From this, we learn that f_j must be linear, and that j's activation must remain constant:

$$y_j(t+1) = f_j(net_j(t+1))f_j(w_{jj}y_j(t)) = y_j(t).$$

This is ensured by using the identity function $f_j : f_j(x) = x, \forall x$, and by setting $w_{jj} = 1.0$. CECs are the central feature of LSTM where short-term memory storage is achieved for extended periods of time.

3.7.2 Memory Cells

In the absence of new inputs to the cell, we now know that the CEC's backflow remains constant. However, as part of a neural network, the CEC is not only connected to itself, but also to other units in the neural network. We need to take these additional weighted inputs and outputs into account. Incoming connections to neuron j can have conflicting weight update signals, because the same weight is used for storing and ignoring inputs. For weighted output connections from neuron j, the same weights can be used to both retrieve j's contents and prevent j's output flow to other neurons in the network.

To address the problem of conflicting weight updates, LSTM extends the CEC with *input and output gates* connected to the network input layer and to other memory cells. This results in a more complex LSTM unit, called a *memory cell*; its standard architecture is shown in Figure 3.15.

The input gates, which are simple sigmoid threshold units, with an activation function range of $[0, 1]$, control the signals from the network to the memory cell by scaling them appropriately; when the gate is closed, activation is close to zero. Additionally, these can learn to protect the contents stored in j from disturbance by irrelevant signals. The activation of a CEC by the input gate is defined as the *cell state*. The output gates can learn how to control access to the memory cell contents, which protects other memory cells from disturbances originating from j. So we can see that the basic function of multiplicative gate units is to either allow or deny access to constant error flow through the CEC.

3.7.3 Memory Blocks

In an LSTM network, all units in the hidden layer are placed within new central units—so-called *memory blocks*. Each memory block contains at least one memory cell containing regulating gates to control incoming and outgoing information flow. Each memory cell has its own CEC to ensure a constant

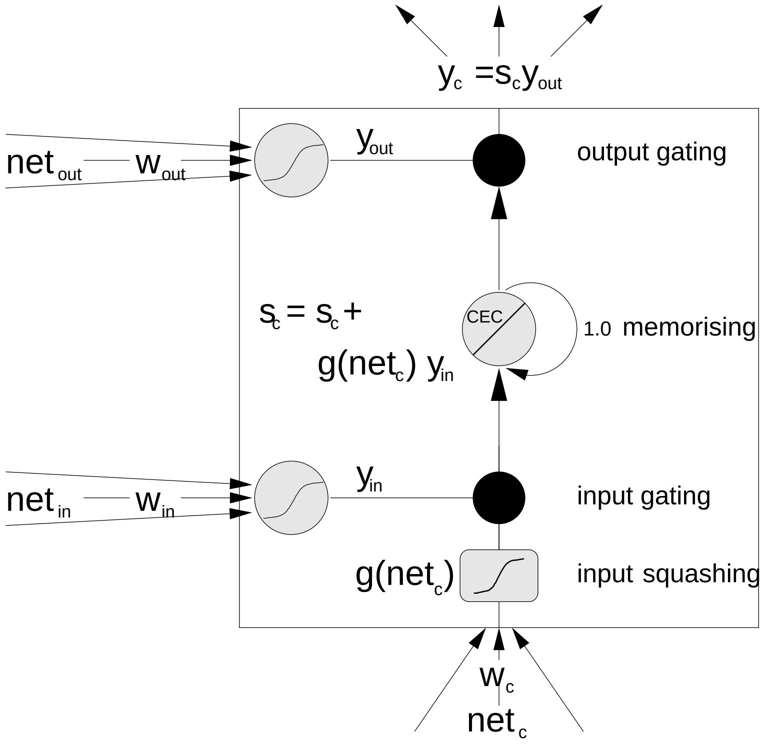


Figure 3.15: *A standard LSTM memory cell with a recurrent self-connection (CEC) and weight of '1'. The state of the cell is denoted as s_c. Read and write access is regulated by the input gate, y_{in}, and the output gate, y_{out}. The internal cell state is calculated by multiplying the result of the squashed input, g, by the result of the input gate, y_{in}, and then adding the state of the last time step, $s_c(t-1)$. Finally, the cell output is calculated by multiplying the cell state, s_c, by the activation of the output gate, y_{out}.*

error flow through the cell, and all memory cells within a memory block share the same input and output gate. LSTM recurrent neural networks can be equipped with several such memory blocks.

3.7.4 The Forward Pass

We define discrete time steps in the form $t = 1, 2, 3, ...$ Each time step has a forward pass and a backward pass. In the forward pass, the update of all activations and states are calculated, whereas in the backward pass, the calculation of the error signals for all weights is performed. The index m refers to the source units. Let c_j^v be the v-th memory cell in the j-th memory block, and w_{lm} be a weight connecting unit m to unit l.

The internal state of a standard LSTM memory cell is updated according to its current state s_c and three sources of external input: Input gate net_{in}; output gate net_{out}; and the input of the standard memory cell $net_{c_j^v}$.

The activation of the input gate y_{in} is computed as

$$y_{in_j}(t) = f_{in_j}(net_{in_j}(t)) \tag{3.18}$$

with the input gate input

$$net_{in_j}(t) = \sum_m w_{in_j m} y_m(t-1) \tag{3.19}$$

and the activation of the output gate y_{out} respectively

$$y_{out_j}(t) = f_{out_j}(net_{out_j}(t)) \tag{3.20}$$

with the output gate input

$$net_{out_j}(t) = \sum_m w_{out_j m} y_m(t-1) \tag{3.21}$$

where the index m refers to the feeding units.

The results of the gates are scaled, using a non-linear squashing function,

$$f(x) = \frac{1}{1+e^{-x}} \tag{3.22}$$

so that they are within the range $[0,1]$. Thus, the input for the memory cell will only be able to pass if the signal at the gate is sufficiently close to ‘1’.

The cell input net_c, passed at time t from the previous network layer, is computed as

$$net_{c_j^v}(t) = \sum_m w_{c_j^v m} y_m(t-1). \tag{3.23}$$

To calculate the internal state $s_c(t)$ of the memory cell, the squashed input is multiplied by the result of the input gate, and the state of the last time step $s_c(t-1)$ is then added. The corresponding equation is

$$s_{c_j^v}(t) = \begin{cases} 0 & \text{if } t = 0 \\ s_{c_j^v}(t-1) + y_{in_j}(t) g(net_{c_j^v}(t)) & \text{if } t > 0 \end{cases} \tag{3.24}$$

with the optional non-linear squashing function for the cell input

$$g(x) = \frac{4}{1 + e^{-x}} - 2 \tag{3.25}$$

which, in this case, scales the result to the range $[-2, 2]$.

The cell output y_c is now calculated by multiplying the cell state s_c by the activation of the output gate y_{out}:

$$y_{c_j^v}(t) = y_{out_j}(t) s_{c_j^v}(t). \tag{3.26}$$

Assuming a layered, recurrent neural network with standard input, standard output and hidden layer consisting of memory blocks, the activations for the output units k are finally computed as

$$y_k(t) = f_k(net_k(t)) \tag{3.27}$$

with

$$net_k(t) = \sum_m w_{km} y_m(t-1). \tag{3.28}$$

where we can again use the logistic sigmoid in Equation 3.22 as a squashing function f_k.

3.7.5 Forget Gates

The self-connection in a standard LSTM network has a fixed weight set to ‘1’ in order to preserve the cell state over time. Unfortunately, the cell states s_c tend to grow linearly during the progression of a time series presented in a continuous input stream. The main negative effect is that the entire memory cell loses its memorising capability, and begins to function like an ordinary RNN network neuron.

By manually resetting the state of the cell at the beginning of each sequence, the cell state growth can be limited. But this is not practical for continuous input, where there is no distinguishable end, or subdivision is very complex and error prone.

To address this problem, [Gers *et al.* 1999] suggested that an adaptive forget gate could be attached to the self-connection. Forget gates can learn to reset the internal state of the memory cell when the stored information is no longer needed. To this end, we replace the weight '1.0' of the self-connection from the CEC with a multiplicative, forget gate activation y_φ, which is computed using a similar method as for the other gates:

$$y_{\varphi_j}(t) = f_{\varphi_j}(net_{\varphi_j}(t)) \tag{3.29}$$

with

$$net_{\varphi_j}(t) = \sum_m w_{\varphi_j m} y_m(t-1) \tag{3.30}$$

using the squashing function in Equation 3.22, with a range $[-1, 1]$.

The updated equation for calculating the internal cell state s_c for $t > 0$ and $s_{c_j^v}(0) = 0$ is

$$s_{c_j^v}(t) = s_{c_j^v}(t-1) \underbrace{y_{\varphi_j}(t)}_{\substack{=1 \text{ without} \\ \text{forget gate}}} + y_{in_j}(t) g(net_{c_j^v}(t)) \tag{3.31}$$

using the squashing function in Equation 3.25, with a range $[-2, 2]$. The extended forward pass is given simply by exchanging Equation 3.24 for Equation 3.31.

The bias weights of input and output gates is initialised with negative values, and the weights of the forget gate are initialised with positive values. From this, it follows that at the beginning of training, the forget gate activation will be close to '1.0'. The memory cell will behave like a standard LSTM memory cell without a forget gate. This prevents the LSTM memory cell from forgetting, before it has actually learned anything.

3.7.6 Backward Pass

LSTM learning uses a hybrid approach, combining customised versions of backpropagation through time with real-time recurrent learning. The standard backpropagation through time is used for output units. The output gates use a truncated and slightly modified version of backpropagation through time. The input gate, the optional forget gate, and the weights of memory cells use a truncated version of real-time recurrent learning. In this context, truncation cuts off the error once it has leaked out of the cell or gate, although it still serves to change the incoming weights. Thus, the continuous error flow is limited to the constant error carousel.

Following the notation used in previous sections, and using Equations 3.4, 3.5 and 3.6, the overall network error at time step t is

$$E(t) = \frac{1}{2} \sum_{k \in U} (\underbrace{d_k(t) - y_k(t)}_{e_k(t)})^2,$$

using the squared error as an objective function and where $e_k(t)$ is the externally injected error.

Following and expanding Equation 3.9 via gradient descent, we can calculate the weight changes $\Delta w_{lm}(t)$ connecting unit m to unit l, using a learning rate α as

$$\begin{aligned}\Delta w_{lm}(t) &= -\alpha \frac{\partial E(t)}{\partial w_{lm}} = \alpha \sum_{k \in U} e_k(t) \left(\frac{\partial y_k(t)}{\partial w_{lm}} \right) \\ &= \alpha \sum_{k} \sum_{l'} e_k(t) \frac{\partial y_k(t)}{\partial y_{l'}(t)} \frac{\partial y_{l'}(t)}{\partial net_{l'}(t)} \frac{\partial net_{l'}(t)}{\partial w_{lm}}.\end{aligned}$$

Using the Kronicker delta we then get

$$\Delta w_{lm}(t) = \alpha \sum_{k} \sum_{l'} e_k(t) \frac{\partial y_k(t)}{\partial y_{l'}(t)} \frac{\partial y_{l'}(t)}{\partial net_{l'}(t)} \left(\delta_{l'l} y_m(t-1) + \frac{\partial net_{l'}(t)}{\partial y_m(t-1)} \right). \tag{3.32}$$

When leaving the memory block, the errors are truncated by setting the derivatives $\frac{\partial net_{l'}(t)}{\partial y_m(t-1)} \overset{\text{tr}}{=} 0$ for $l' \in \{\varphi, in, c_j^v\}$. Therefore the sum for unit l'

vanishes. After error truncation, Equation 3.32 is reduced to

$$\begin{aligned}\Delta w_{lm}(t) &\overset{\text{tr}}{=} \alpha \sum_{k \in U} e_k(t) \frac{\partial y_k(t)}{\partial y_l(t)} \frac{\partial y_l(t)}{\partial net_l(t)} y_m(t-1) \\ &= \alpha \underbrace{\frac{\partial y_l(t)}{\partial net_l(t)} \left(\sum_k \frac{\partial y_k(t)}{\partial y_l(t)} e_k(t) \right)}_{=:\delta_l(t)} y_m(t-1)\end{aligned} \tag{3.33}$$

where $\overset{\text{tr}}{=}$ indicates that the error is truncated. Truncation is used to calculate the derivatives of the output gates δ_{out_j} and the output units δ_k.

For output unit weights and output gate weights, we use BPTT to calculate the weight changes. For an arbitrary output weight $w_{km}(t)$, connecting unit m to output unit k, the sum in Equation 3.33 reduces to $\sum_k \frac{\partial y_k(t)}{\partial y_k(t)} e_k(t) = e_k(t)$. Then, the weight change is given by

$$\Delta w_{km}(t) = \alpha \delta_k(t) y_m(t-1). \tag{3.34}$$

with

$$\begin{aligned}\delta_k(t) &\overset{\text{tr}}{=} \frac{\partial y_k(t)}{\partial net_k(t)} e_k(t) \\ &= f'_k(net_k(t)) e_k(t)\end{aligned} \tag{3.35}$$

where the derivatives are given by differentiating Equation 3.27.

For the output gate, the weight changes of a connection from an output gate of an arbitrary memory block j to a unit m is given by

$$\Delta w_{out_j m}(t) = \alpha \delta_{out_j}(t) y_m(t). \tag{3.36}$$

with

$$\begin{aligned}\delta_{out_j}(t) &\overset{\text{tr}}{=} \frac{\partial y_{out_j}(t)}{\partial net_{out_j}(t)} \left(\sum_k \frac{\partial y_k(t)}{\partial y_{out_j}(t)} e_k(t) \right) \\ &= f'_{out_j}(net_{out_j}(t)) \left(\sum_{v=1}^{S_j} s_{c_j^v}(t) \sum_k w_{kc_j^v} \delta_k(t) \right)\end{aligned} \tag{3.37}$$

where v indexes the memory cells within a block j with S_j cells. The derivatives are given by differentiating Equations 3.20, 3.26 and 3.27.

To calculate the weight changes for connections to the memory cell, input gate and forget gate, we use a different approach. Instead of truncated BPTT, we use a variant of RTRL. Analogous to Equations 3.9 and 3.32, but splitting the gradient in a different way, we get the following equations

$$\begin{aligned}\Delta w_{lm}(t) = -\alpha \frac{\partial E(t)}{\partial w_{lm}} &\overset{\text{tr}}{=} -\alpha \underbrace{\frac{\partial E(t)}{\partial s_{c_j^v}(t)}}_{=:-e_{s_{c_j^v}}(t)} \frac{\partial s_{c_j^v}(t)}{\partial w_{lm}} \\ &= \alpha e_{s_{c_j^v}}(t) \left(\frac{\partial s_{c_j^v}}{\partial w_{lm}} \right)\end{aligned} \tag{3.38}$$

with the internal state error $e_{s_{c_j^v}}$ of each memory cell

$$\begin{aligned} e_{s_{c_j^v}} &\overset{\text{tr}}{:=} \frac{\partial E(t)}{\partial s_{c_j^v}(t)} = \frac{\partial E(t)}{\partial y_k(t)} \frac{\partial y_k(t)}{\partial y_{c_j^v}} \frac{\partial y_{c_j^v}}{\partial s_{c_j^v}} \\ &= \underbrace{\frac{\partial y_{c_j^v}}{\partial s_{c_j^v}}}_{=y_{out_j}(t) s_{c_j^v}(t)} \sum_k \underbrace{\frac{\partial y_k(t)}{\partial y_{c_j^v} e_k(t)}}_{=w_{kc_j^v}\delta_k(t)} \\ &= y_{out_j}(t) s_{c_j^v}(t) \left(\sum_k w_{kc_j^v} \delta_k(t) \right) . \end{aligned}$$

To calculate the remaining partial derivative $\frac{\partial s_{c_j^v}}{\partial w_{lm}}$ from Equation 3.38, we differentiate the internal state update in Equation 3.31 and obtain a sum of the following four terms:

$$\begin{aligned}\frac{\partial s_{c_j^v}}{\partial w_{lm}} = & \underbrace{\frac{\partial s_{c_j^v}(t-1)}{\partial w_{lm}}}_{\neq 0 \text{ for all } l \in \{\varphi, \text{in}, c_j^v\}} + \underbrace{y_{in_j}(t) \frac{\partial g(net_{c_j^v}(t)}{\partial w_{lm}}}_{\neq 0 \text{ for } l = c_j^v \text{ (cell)}} + \\ & \underbrace{g(net_{c_j^v}(t) \frac{\partial y_{in_j}(t)}{\partial w_{lm}}}_{\neq 0 \text{ for } l = \text{in (input gate)}} + \underbrace{s_{c_j^v}(t-1) \frac{\partial y_{\varphi_j}(t)}{\partial w_{lm}}}_{\neq 0 \text{ for } l = \varphi \text{ (forget gate)}} .\end{aligned} \tag{3.39}$$

After differentiation, using the forward pass Equations 3.24, 3.18 and 3.29 for g, y_{in} and y_φ, and substituting unresolved partials, we can split Equation 3.39 into three separate equations.

For the cell we get:

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{c_j^v m}} \overset{\text{tr}}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{c_j^v m}} y_{\varphi_j}(t) + g'(net_{c_j^v}(t)) y_{in_j}(t) y_m(t-1), \tag{3.40}$$

for the input gate we get:

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{in_j m}} \overset{\text{tr}}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{in_j m}} y_{\varphi_j}(t) + g(net_{c_j^v}(t)) f'_{in_j}(net_{in_j}(t)) y_m(t-1), \tag{3.41}$$

and for the forget gate respectively, we get:

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j m}} \overset{\text{tr}}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\varphi_j m}} y_{\varphi_j}(t) + s_{c_j^v}(t-1) f'_{\varphi_j}(net_{\varphi_j}(t)) y_m(t-1). \tag{3.42}$$

Since the initial state of the network is independent of the weights, we also need to set the first time step to zero:

$$\frac{\partial s_{c_j^v}(t=0)}{\partial w_{lm}} = 0 \text{ for } l \in \{\varphi, \text{in}, c_j^v\}. \tag{3.43}$$

We can now insert the partials of Equations 3.40, 3.41, 3.42 and 3.43 into Equation 3.32 in order to obtain the full equations for the corresponding weight updates at each time step. For the weight deltas of the cell input weights, we need to calculate

$$\Delta w_{c_j^v m}(t) = \alpha e_{s_{c_j^v}}(t) \left(\frac{\partial s_{c_j^v}}{\partial w_{c_j^v m}} \right) \tag{3.44}$$

and for the input and forget gate weights, summed over all contributions from all cells in each block, we need to calculate

$$\Delta w_{lm}(t) = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t) \left(\frac{\partial s_{c_j^v}}{\partial w_{lm}} \right) \text{ for } l \in \{\varphi, in\}. \tag{3.45}$$

A more detailed version of the LSTM backward pass with forget gates is described in [Gers *et al.* 1999].

3.7.7 Peephole Connections

In standard LSTM, gates do not have direct access to the internal cell state. They are connected only to the input units and the cell outputs. As long as the output gate is closed, the output of the cell is close to zero. Until the output gate starts to open, none of the gates have any information about the state of the CEC they should control.

To address this problem, we can add weighted connections from the internal cell state to all gates within a memory block. These connections are labelled as *peephole connections* which allow gates to learn to protect the internal cell state from unwanted inputs during the forward pass, whereas in the backward pass, they learn to protect the internal cell state from unwanted error signals.

The gates and the internal cell state need to be updated in two separate phases. In the first phase, we need to update input gate y_{in}, forget gate y_{φ}, and cell state s_c, in this specific order. Then, in the second phase, we finally update the output gate y_{out}.

To enable peephole connections, in the forward pass we change the Equation 3.19 for input gate activation as follows:

$$net_{in_j}(t) = \sum_m w_{in_j m} y_m(t-1) + \underbrace{\sum_{v=1}^{S_j} w_{in_j c_j^v} s_{c_j^v}(t-1)}_{=0 \text{ without peepholes}}, \tag{3.46}$$

the Equation 3.30 for forget gate activation as follows:

$$net_{\varphi_j}(t) = \sum_m w_{\varphi_j m} y_m(t-1) + \underbrace{\sum_{v=1}^{S_j} w_{\varphi_j c_j^v} s_{c_j^v}(t-1)}_{=0 \text{ without peepholes}}, \tag{3.47}$$

and the Equation 3.21 for output gate activation as follows:

$$net_{out_j}(t) = \sum_m w_{out_j m} y_m(t-1) + \underbrace{\sum_{v=1}^{S_j} w_{out_j c_j^v} s_{c_j^v}(t)}_{=0 \text{ without peepholes}}. \tag{3.48}$$

We also require modifications in the LSTM backward pass in order to use peephole connections. Analogous to Equations 3.41 and 3.42, the partial derivatives for peephole connections to input and forget gate are computed using the following equations:

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{in_j c_j^v}} \overset{\text{tr}}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{in_j c_j^v}} y_{\varphi_j}(t) + g(net_{c_j^v}(t)) f'_{in_j}(net_{in_j}(t)) s_{c_j^v}(t-1) \tag{3.49}$$

$$\frac{\partial s_{c_j^v}(t)}{\partial w_{\varphi_j c_j^v}} \overset{\text{tr}}{=} \frac{\partial s_{c_j^v}(t-1)}{\partial w_{\varphi_j c_j^v}} y_{\varphi_j}(t) + s_{c_j^v}(t-1) f'_{\varphi_j}(net_{\varphi_j}(t)) s_{c_j^v}(t-1). \tag{3.50}$$

As in Equations 3.36 and 3.45, the changes to the peephole connection weights are calculated as follows:

$$\Delta w_{out_j c_j^v}(t) = \alpha \delta_{out_j}(t) s_{c_j^v} \tag{3.51}$$

$$\Delta w_{l_j c_j^v}(t) = \alpha \sum_{v=1}^{S_j} e_{s_{c_j^v}}(t) \left(\frac{\partial s_{c_j^v}}{\partial w_{l_j c_j^v}} \right) \text{ for } l \in \{\varphi, in\}. \tag{3.52}$$

In the updated backward pass, the error signals are not propagated back from the gates through the peephole connections to the internal cell.

A standard LSTM memory cell, extended by a forget gate and peephole connections, is shown in Figure 3.16.

Figure A.9 on Page 192 shows an LSTM network with two memory blocks, each of which contain two cells. The network input layer is fully connected to the hidden layer and the output layer. The connections to the output layer are so-called *shortcuts*. The output of the hidden layer is fully connected to the output layer and back to the inputs of the hidden layer. The inputs of the hidden layer are the cell's inputs and the three gate types *input*, *forget* and *output*. The memory cells are likewise extended with *peephole connections*.

In our experiments using peephole connections, we did not experience outstanding performance improvements. Nevertheless, we decided to use them. The updated forward and backward pass for peephole LSTM is described in more detail in [Gers *et al.* 2002].

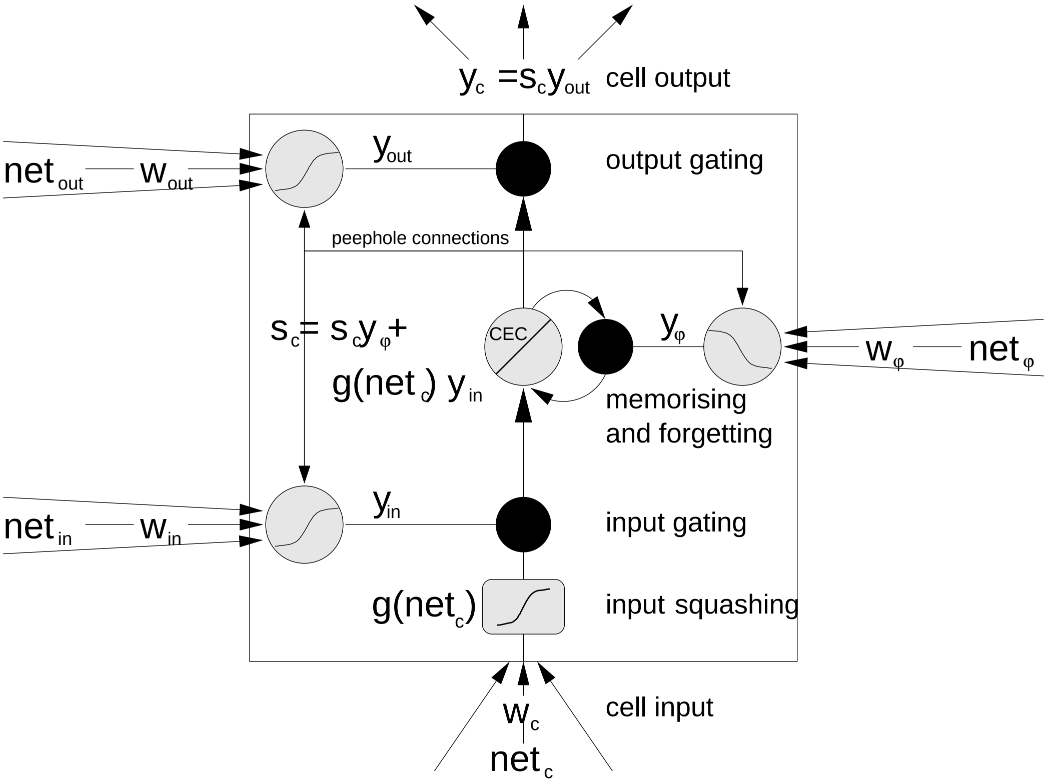


Figure 3.16: *A standard LSTM memory cell with forget gate and peephole connections. The state of the cell is denoted as s_c. Read and write access is regulated by the input gate, y_{in}, and the output gate, y_{out}. The gates and the internal cell state need to be updated in a specific order. Firstly, we need to update input gate, y_{in}, forget gate, y_φ, and cell state, s_c. Then, we finally update the output gate, y_{out}. For the forward pass, we need the equations for input gate activation 3.19, forget gate activation 3.30, and output gate activation 3.21.*

3.8 Conclusions

In this chapter, we learned about five static and two dynamic classifiers. All static classifiers presented have strong classification capabilities ([Mitchell 1997] and [Han & Kamber 2006]), but are not well-suited to time series prediction. To address temporal learning, we require dynamic classifiers, such as recurrent neural networks.

The static classifiers covered were decision trees ([Quinlan 1986] and [Quinlan 1993]), naïve Bayes ([John & Langley 1995]), Bayesian networks ([Heckerman *et al.* 1995]), feed-forward neural networks ([Rumelhart *et al.* 1994]) and support vector machines ([Boser *et al.* 1992] and [Cortes

& Vapnik 1995]).

The dynamic classifiers presented were recurrent neural networks ([Rumelhart *et al.* 1986], [Williams & Zipser 1995], [Williams & Zipser 1989], [Williams & Zipser 1995], and [Hochreiter *et al.* 2001]) and, in particular, long short-term memory (LSTM [Hochreiter & Schmidhuber 1996], [Hochreiter & Schmidhuber 1997], [Gers *et al.* 1999] and [Gers *et al.* 2002]) recurrent neural networks .

In this chapter, we covered the derivation of LSTM in detail, summarising the most relevant literature. Specifically, we highlighted the vanishing error problem, which is a serious shortcoming of RNNs. LSTM provides a possible solution to this problem by introducing a constant error flow through the internal states of special memory cells. In this way, LSTM is able to tackle long time-lag problems, bridging time intervals in excess of 1,000 time steps. Finally, we introduced two LSTM extensions that enable LSTM to learn self-resets and precise timing. With self-resets, LSTM is able to free memory of irrelevant information.

In the next chapter, we will delve into the KDD Cup '99 benchmark dataset, which will be applied to all classifiers presented and extract small subsets of salient features from the full feature set.

Chapter 4

Extracting Salient Features for IDS

Contents

4.1 Introduction

In this chapter, we cover the extraction and preprocessing of important features. These are essential tasks for data mining and intrusion detection. From the perspective of data mining, dimension reduction aims to find the set of minimal features that best classifies the training data. Some attributes may contain redundant information, while others may contain information suggesting false correlations; either type can hinder correct classification. Additionally, unnecessary features add to computation time. To our knowledge, no general theory exists that captures the relationship between different attacks and provided features.

[Amaldi & Kann 1998] show that the time complexity of the feature selection problem (also referred to as dimension reduction) is NP-hard. From this, we conclude that if the number of variables becomes too large, the search quickly becomes computationally intractable. A review of alternative search strategies is presented by [Kohavi & John 1997], [Dash & Liu 1997] and [Chen *et al.* 2006].

From the perspective of network intrusion detection systems, there are strong reasons to reduce the number of collected features and choose features that can easily be extracted from a high-speed data stream. Connections in today's local area networks forward packets at tens of gigabits per second (using the minimal frame size of 64 bytes, we can transfer up to 14.8 million frames per second in 10-Gigabit Ethernet networks); simply put, even the monitoring of data at 10 Gbps is a major challenge.

In order to perform in-depth packet analysis, it is essential to perform massive data reduction in order to obtain an amount that can actually be processed further. This is extremely important if the objective is real-time detection.

Traffic information reduction can be accomplished in various ways. Prior to network data collection, for example, we can apply filters that ignore certain types of traffic; but although this may leave only traffic considered potentially interesting, filtering might also remove important data.

Observed traffic can also be compressed into connection records that summarise the essential information about individual sessions between two parties; each connection record contains preprocessed features; this kind of compression was performed for the KDD Cup '99 data set.

The so-called basic (or base) features of the KDD Cup '99 dataset require only the header information from IP packets and TCP/UDP/ICMP segments, and the total size of IP packets. Extraction of header information is much less complex than the extraction of content features.

In-depth packet data analysis requires the computationally demanding and memory-intensive reassembling of data streams. Furthermore, the data analysis frequently requires domain knowledge, which needs to be provided by a human expert. Extraction and analysis of content features is unlikely to be cost-effective on a large scale in high-speed network environments.

At this point, we note that the connection records in the KDD Cup '99 dataset are outdated, and are neither representative of current network traffic nor contain attacks currently relevant in today's computer networks. But it is still the only fully labelled intrusion detection dataset available of a decent size; and we think it is still relevant in evaluating the performance of machine learning algorithms in the field of intrusion detection.

In the rest of this chapter, we first discuss relevant performance metrics applied to the selected classifiers, then we discuss the feature set reduction techniques applied to the dataset. Finally, we introduce and dive into the details of the DARPA and KDD Cup '99 datasets.

After a short literature review of related work published on feature reduction of the KDD Cup '99 datasets, we present our contributions. Using a custom network feature preprocessing framework, and custom-built training sets derived from the KDD Cup '99 datasets, we present a framework that supports a data mining and network security expert in minimal feature set extraction. This process is supported by detailed visualisation and examination of class distributions.

The feature reduction process applied is based on decision tree pruning. Finally, we conclude by presenting a number of minimal feature sets for detecting all attacks and individual attacks, using one-classifier training with very few features.

4.2 Performance Metrics

For a meaningful comparison of the performance of different network intrusion systems, it is necessary to at least agree on training data, testing data and what performance metrics are applied.

A model built by a classifier from given examples is only an approximation of the true model if we can assume the data used is valid. To evaluate how closely the built model complies with the true model, we need to divide the available examples into training data and testing data. Firstly, the training data is used to build the model, and then the model is evaluated using the test data. Based on the fact that the labels of the test data are known, we can apply a number of performance metrics.

An overview of common performance metrics for machine learning applied to computer security is provided by [Maloof 2006]. For a comprehensive introduction to performance metrics for machine learning, see [Han & Kamber 2006].

4.2.1 Measuring IDS Performance

Intrusion detection systems generate two types of misclassification errors: False positives and false negatives. A false positive wrongly reports an attack, and a false negative fails to report an occurring attack. The corresponding performance metrics are the false positive rate and the false negative rate.

For the evaluation of IDS systems, and for the purpose of comparison between different systems, the crossover error rate can additionally be applied as a third performance metric. The crossover error rate is the portion of misclassification errors when the sensitivity of the system is adjusted in such a way that the false positive rate and the false negative rate are equal (see Figure 4.1).

In the field, most commercial IDS products have a strong focus on reducing the number of false positives. This is due to the fact that alerts need to be manually analysed and filtered by a security expert. The number of alerts from an IDS system can easily reach thousands of events per day when monitoring large infrastructures. Assuming that only 10 per cent of all alerts are false

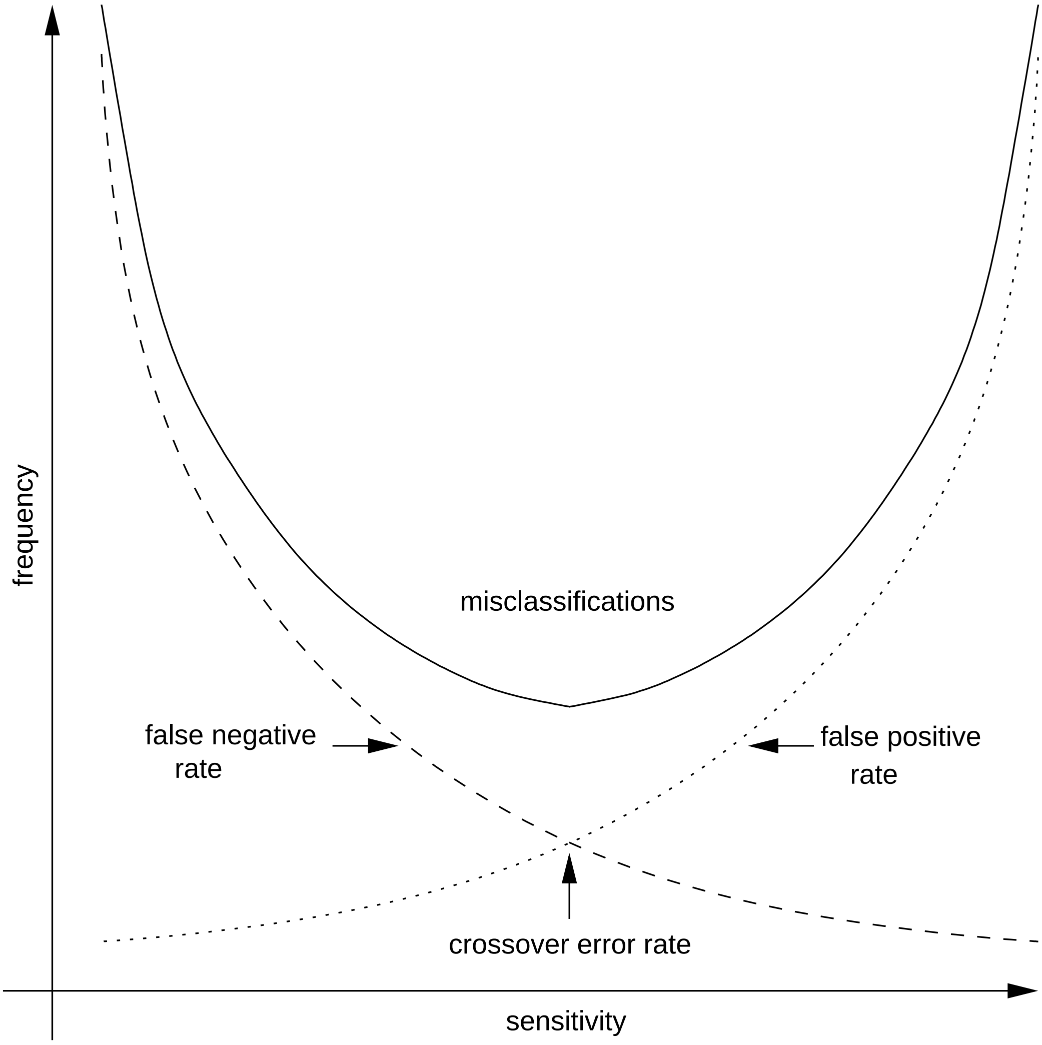


Figure 4.1: *The crossover error rate can be used as a performance metric for the evaluation of intrusion detection systems. It requires the IDS to be adjusted in such a way that the false positive rate and the false negative rate are equal.*

positives, such a system is already rendered unusable if maintained by a single security manager.

IDS security managers are forced to tune the sensitivity of the system's detection and reporting algorithms to keep the number of generated alerts by the IDS manageable. But the downside of reducing the number of events produced is that reliability of the IDS system is lowered. It is the job of the security manager to tune the IDS such that the number of generated alerts and reliability are appropriate for the monitored environment.

In highly sensitive environments, such as military applications, a large number of false positives might be acceptable if this ensures that more detectable intrusions are logged. In an academic environment with very limited human resources, a low percentage of false positives might be essential in order to guarantee that all events can be manually analysed.

4.2.2 Simple Performance Measures

For two-class problems, the result of a classification can be either predicted correctly or incorrectly. This yields four different conditions:

- (a) true positive—model correctly predicts positive
- (b) false negative (type II error)—model incorrectly predicts negative
- (c) false positive (type I error)—model incorrectly predicts positive
- (d) true negative—model correctly predicts negative

A confusion matrix shows the predicted and actual classifications. The size of a confusion matrix is $n \times n$, where n is the number of different classes. The two types of errors, false positive and false negative, can have different costs. If the costs of misclassification are known with a cost matrix, we can calculate the average cost per test example for a given model. The size of the cost matrix is equal to the size of the confusion matrix; it specifies the costs associated with the different error conditions. All elements of the confusion matrix are multiplied by the corresponding value of the cost matrix, and the total cost of a model is the sum of all these products. The average cost is the total cost divided by the total number of classifications. The confusion and cost matrices shown in Tables 4.1 and 4.2 are for a two-class problem.

Table 4.1: *Confusion matrix*

		predicted	
		positive	negative
actual	positive	a	b
	negative	c	d

Table 4.2: *Cost matrix*

		predicted	
		positive	negative
actual	positive	0	1
	negative	4	0

From the counts of the four conditions, we can calculate the following simple performance metrics:

- sensitivity, *detect(ion) rate* (DR) or *true positive rate* (TPR): $\frac{a}{a+b}$
- *false negative rate* (FNR): $\frac{b}{a+b}$
- *false alarm rate* (FAR) or *false positive rate* (FPR): $\frac{c}{c+d}$
- specificity or *true negative rate* (TNR): $\frac{d}{c+d}$
- precision $\frac{a}{a+c}$

Sensitivity is the portion of positive tuples that are correctly predicted as positive. The false negative rate is the portion of positive tuples that are wrongly predicted as negative. The false positive rate is the portion of negative tuples wrongly predicted as positive. Specificity is the portion of negative tuples that are correctly predicted as negative. Additionally, precision is the probability that an instance is correctly classified.

From sensitivity and specificity we can derive *accuracy* (ACC):

- ACC: $\frac{a+d}{a+b+c+d}$

Accuracy is the portion of test results that the model predicts correctly. A classifier with high accuracy will ideally have most of the tuples presented along the diagonal of the confusion matrix.

4.2.3 The Mean Squared Error

For numeric prediction tasks, the *mean squared error* (MSE) is another simple method that can be used. The mean squared error quantifies the amount by which an estimator differs from the targeted value.

The squared error between the value y_i and the predicted value y'_i is $(y_i - y'_i)^2$. The MSE of a dataset is the average of the sum of all squared errors of each pattern. Given n patterns from the dataset, for the ith example, let p_i be the predicted value, and a_i the actual value.

The mean squared error of a given dataset is

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(a_i - p_i)^2$$

4.2.4 ROC Analysis

The *receiver operating characteristic* (ROC) analysis evaluates an algorithm over a range of possible operational scenarios. An excellent introduction into ROC analysis is provided by [Fawcett 2006].

The ROC graph is a two-dimensional plot of the false positive rate (x-axis) of a model against its true positive rate (y-axis). A true positive rate of unity and false positive rate of zero are indicators of perfect performance. The lower-left point (0,0) on the graph represents a model with neither false positive errors nor true positives. This model would always classify negative, and never positive. The opposite point on the upper-right (1,1) represents a model that always classifies positive, and never negative. The point on the upper-left (0,1) of the ROC graph represents a model that always classifies correctly.

4.2.4.1 Curve Generation

Discrete classifiers, such as decision trees, generate a single-class decision for every instance of a test set. For a given decision threshold, all classified instances yield one confusion matrix. In two-type classification, each matrix has exactly one true positive rate and one false positive rate. These two produce a single point on the ROC graph. To generate an ROC curve, the threshold is varied in n-steps from $-\infty$ to $+\infty$ producing n-points on the ROC graph.

For probabilistic classifiers, there is a more efficient method of generating the ROC curve. In contrast to discrete classifiers, the results of probabilistic classifiers produce numeric values. Neural networks are probabilistic classifiers with target values in the range of $[0, 1]$. The target value represents the probability that the observed instance is a member of a specific class, where a higher value indicates a higher probability. Typically, a decision threshold of

0.5 is applied to the classifier result to produce the decision. If the resulting target value is above the threshold, the instance belongs to a specific class. A value under the threshold is classified as noise. Every threshold applied to the target values produces its own confusion matrix and a different point on the ROC graph.

The resulting ROC curve of a successfully learned classifier should look like an inverted 'L', with the corner pushing forward the upper-left of the graph. Results similar to random guessing yield a diagonal line between (0,0) and (1,1).

One important property of ROC curves is that they are insensitive to changes in class distribution. Different test sets, with changing proportions of positive and negative instances, will generate the same graphs.

4.2.4.2 Area Under Curve

The *area under curve* (AUC) summarises the curve in a single value as a measure of expected performance. The AUC value of a classifier is a numerical value in the range $[0, 1]$. It is equal to the probability that a randomly chosen positive instance will be ranked higher than a randomly chosen negative instance.

Various methods of estimating the AUC value exist; the most precise way is to use a non-parametric method. The so-called trapezoidal rule involves dividing the area under the curve into a number of trapezoids of equal width. The area of each trapezium is estimated by replacing the upper-end of each trapezium with a straight line. The AUC value is the sum of these approximations.

No meaningful classifier should have an AUC value below 0.5, which is equivalent to random guessing. A model with perfect accuracy will have an AUC value of 1.0. Depending on the shape of the ROC curves, it is possible for a high AUC value of one classifier to perform worse in a specific region of the curve than a low AUC value of another classifier. But in practice, the AUC value performs very well.

4.2.4.3 Multi-Class Problems

ROC analysis is much more complex when handling decision problems with more than two classes. For handling n-classes, it is most common to break the problem down into n two-class problems and produce a graph for every class.

To calculate *Multi-class AUC* (MAUC) values, we need to sum the AUC values weighted according to class prevalence of the n-classes:

$$AUC_{total} = \sum_{c_i \in C} AUC(c_i) \times p(c_i)$$

with $AUC(c_i)$ being the AUC value and $p(c_i)$ being the prevalence of class c_i.

Unfortunately, the resulting MAUC value is now sensitive to class prevalence in the test set. Different class distributions in different test sets will generate different MAUC values. This makes results difficult to interpret and compare.

4.2.5 Comparison of Methods

Accuracy and mean squared error are the most common performance metrics, but simple measures are problematic. Accuracy does not provide information about the performance per class—missing a positive or missing a negative is treated as the same thing. If the majority of examples in a dataset are negative, then a high accuracy might be due only to the exceptional performance on these negative examples, and observing the true positive rate may indicate that the performance on the positive examples is very poor. [Provost *et al.* 1998] provide a more detailed discussion of why performance evaluations using accuracy are problematic.

We need to satisfy certain conditions in order to apply simple performance metrics. In the case of accuracy, this includes having an equal number of examples in each class. For highly skewed data, where one class is much larger than the other, these metrics are not very meaningful.

By contrast, it is a powerful strength of ROC analysis that it is independent of class distribution in two-class problems. The curve remains the same despite changes in the proportions of positive and negative examples.

Of the three metrics—accuracy, MSE and AUC—AUC is the most suitable measure of performance. Like accuracy and MSE, AUC provides a single number as a summary of performance, but it is also independent of class distribution. The drawbacks are that the computing of the AUC value requires significant resources, and that the result is not always intuitively understood. The fact that summarising the ROC curve into a single number comes at the cost of significant information loss should also be taken into consideration.

4.3 Attribute Search Strategies

Feature selection is a method of obtaining a reduced presentation of the data set. It can be broken down into the four phases of subset generation, subset evaluation, stopping criteria and result validation, as suggested by [Dash & Liu 1997].

During *subset generation*, we generate the subsets that are observed and examined during the subset evaluation phase. Feature subset generation is essentially a search through the space of suitable subsets. It has a specified starting point and an initially decided search strategy. It can start with no features, all features, or with a random subset of features available. Depending on the chosen starting point, features are iteratively added or removed. The search strategy can be anything from an exhaustive search through all possible subsets to a random selection of a limited number of candidates.

After the generation of each subset, the results are evaluated during the *subset evaluation* phase and compared to the previous best result. Subsets found to have superior results replace the previous best subset. The evaluation criteria can be dependent or independent of the learning algorithm to which the data is finally applied.

The generation–evaluation loop continues until a matching *stopping criterion* is reached, at which point the subset search is terminated. Suitable stopping criteria can depend on the generation procedure or on the evaluation function. Generation-procedure-dependent stopping criteria could be, for example, a specified number of features or a limited number of iterations reached. A common evaluation-function-dependent stopping criterion is that the addition or deletion of any feature does not create a better subset.

In the *result validation* phase, we finally confirm the validity of the created subset, and we compare the results to previous results. The whole feature selection process is shown in Figure 4.2.

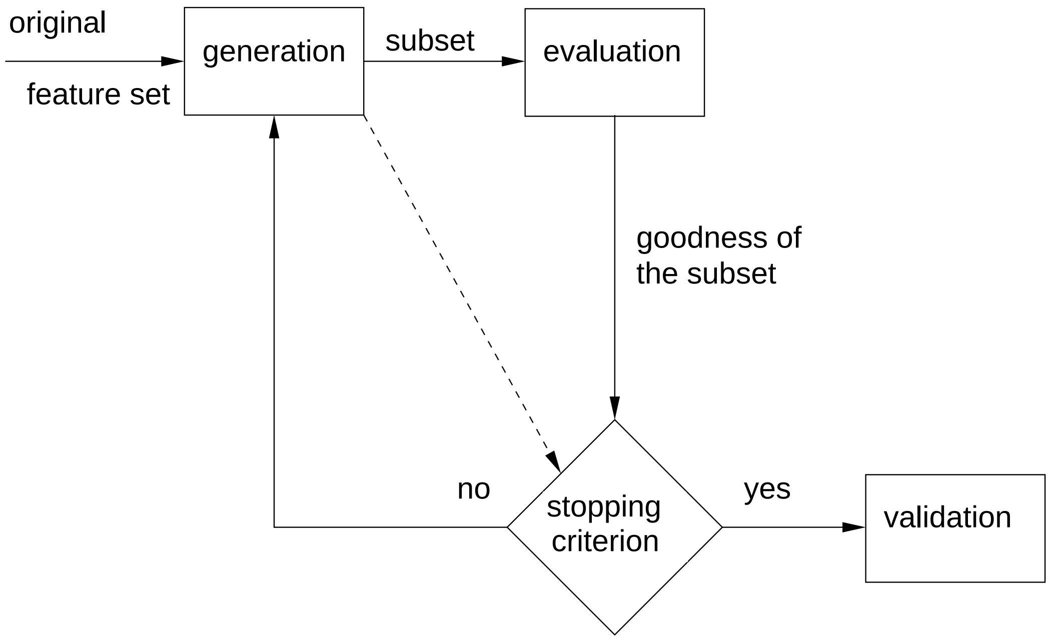


Figure 4.2: *The feature selection process as suggested by [Dash & Liu 1997]. During* subset generation*, we generate the subsets that are observed and examined in the* subset evaluation *phase. This process continues until a matching* stopping criterion *is reached, at which point the subset search is terminated. Finally, the validity of the result is confirmed during* result validation.

Neural networks prefer to process numeric values in a previously specified range (i.e. $[-1, 1]$). Most methods for the transformation of nominal features having a great number of discrete values also increase the total number of required input neurons, which significantly increases computation time. This makes it almost essential for these resource-intensive machine learning methods to perform feature selection prior to training.

Starting from n attributes, there are 2^n possible subsets. With an increasing number of attributes, the exhaustive search over all attributes gets very expensive. It is very common to use heuristic methods that only search a reduced search space.

4.3.1 Forward Selection and Backward Elimination

Examples of very straightforward search strategies are stepwise forward selection, stepwise backward elimination, or a combination of both. In the first, we start with an empty attribute set as the reduced set and add the best attribute from the original attribute set. Then, at each subsequent iteration, we add the best of the remaining attributes to the reduced set.

In stepwise backward elimination, we start with all attributes and stepwise remove the worst attributes from the set. In a combined approach of stepwise forward selection and stepwise backward elimination, we start with a reduced set having a fixed number of promising attributes. In each round, we now select the best attribute from the remaining attributes and remove the worst from the reduced set.

One way to find either the best or worst attribute is to perform experiments with a selected classifier. The classifier's performance is then compared to the previous best performances. A decrease in performance is an indication of the lack of an important feature. If the performance remains unchanged, or increases after adding a feature, it is an indication that the observed feature is unimportant or irrelevant.

4.3.2 Information Gain and Decision Trees

We can also rank features according to their importance. This helps to determine irrelevant or less significant attributes, which can then be deleted first. But it needs to be considered that ranking assumes attribute independence and, therefore, neglects possible interactions between features.

A typical evaluation measure suitable for ranking is information gain. Information gain is the underlying statistical property of feature evaluation used by decision trees. An observed feature with the highest information gain is considered to be the most effective for classifying presented data for a given class.

Decision tree classifiers, such as C4.5, are themselves also well-suited to attribute selection. In the tree-like structure constructed by the learning algorithm, every node represents a test of an attribute. Attributes that the algorithm assumes are irrelevant are not part of the tree. In most cases, the attributes used already represent a reduced subset.

Additionally, the selected attributes are in a hierarchical order. The attribute tested by the node at the root of the tree is considered to be that which best partitions the data into classes. The attributes tested by the nodes in the last layer prior to the leaves are considered to be the worst attributes.

The decision tree classifier and the information measure are described in more detail in Section 3.2.

4.3.3 Domain Knowledge

Furthermore, it is advisable to use domain knowledge for feature selection. Domain knowledge can be provided by an expert in order to remove unimportant features. An investigation of the provided data can reveal non-changing values of features or that noise and outliers devalue the quality of certain features. Human experts can also exclude features that are known to be irrelevant or that are prone to false correlations.

4.4 DARPA and KDD Cup '99 Datasets

The choice of training data available for machine learning in the field of network intrusion detection systems is very limited. One of the few widely used datasets are the DARPA datasets ([Lippmann *et al.* 2000a], [Lippmann *et al.* 2000b]), which also happen to be one of the most comprehensive. They are freely available from the *Information Systems Technology Group* (IST) Web site, which is part of the MIT Lincoln Laboratory ([DARPA 2011]). These datasets are called DARPA datasets because their generation was sponsored by the *Defense Advanced Research Projects Agency* (DARPA ITO) and the *Air Force Research Laboratory* (AFRL/SNHS).

Between 1998 and 2000, the MIT Lincoln Laboratory conducted the DARPA Intrusion Detection Evaluation, which resulted in three scenario-specific datasets. The two main datasets, collected in 1998 and 1999, provided offline evaluation data based on network traffic data and audit logs collected from a simulation network.

The 1998 DARPA Intrusion Detection Evaluation network simulated an air force base local area network. Seven weeks of training data and two weeks of testing data were collected. The total collected data contains more than 200 instances of 39 mostly network-based attack types embedded in background traffic similar to that of an air force base local area network.

All traffic is either classified as ('normal') or as one of various attack types. The attack types are grouped into the four attack categories: *denial-of-service* ('dos'), *network probe* ('probe'), *remote-to-local* ('r2l') and *user-to-root* ('u2r') attacks. In addition, the data contains anomalous user behaviour, such as a normal user acting like a privileged user. All attacks and traffic types used during the evaluation are shown in Table 4.3.

Table 4.3: *Attacks and traffic types which ran during the evaluation. All traffic is either classified as 'normal' or as one of various attack types. The attack types are grouped into the four categories:* denial-of-service *('dos'),* network probes *('probe'),* remote-to-local *('r2l') and* user-to-root *('u2r') attacks.*

<table>
<tr><th>attacks</th><th>Solaris Server</th><th>SunOS internal</th><th>Linux internal</th><th>Cisco Router</th></tr>
<tr><td>probe</td><td colspan="4">ipsweep, nmap, portsweep, satan, *mscan, *saint</td></tr>
<tr><td>dos</td><td colspan="3">neptune, pingofdeath, smurf, syslogd, back, land, teardrop, *apache2, *mailbomb, *processtable, *udpstorm</td><td></td></tr>
<tr><td rowspan="2">r2l</td><td colspan="3">#spy, guess_passwd, ftp_write, multihop, phf, *httptunnel, *xlock, *worm, *xsnoop</td><td rowspan="2">*snmpguess,
*snmpgetattack</td></tr>
<tr><td></td><td>imap,
#warezclient
warezmaster
*named,
*sendmail</td><td></td></tr>
<tr><td rowspan="3">u2r</td><td colspan="3">*sqlattack</td><td rowspan="3"></td></tr>
<tr><td colspan="2">*ps</td><td rowspan="2">perl,
rootkit</td></tr>
<tr><td>buffer_overflow</td><td>loadmodule,
*xterm</td></tr>
</table>

* = *attack appears in test set only*
# = *attack appears in training set only*

The aim of 'dos' attacks is to prevent users accessing a service. 'TCP syn floods' are an example of this type of attack. 'Probe' attacks, such as 'port scans' and 'ip sweeps' are used to collect information about potential targets.

Attackers on a remote machine using ‘r2l’ attacks try to gain user access on a machine they do not have access to. This can be achieved by, for example, dictionary attacks based on password guessing. A ‘u2r’ attack occurs when an attacker who has already achieved user access on a system tries to gain privileged access. Various buffer overflow attacks against network services fall in this category.

Attackers often use combinations of the attack types classified above. In the majority of cases, attackers follow a ‘probe’ → ‘r2l’ → ‘u2r’ pattern of behaviour. The different attack types and the different phases of compromise are described in more detail in Section 2.3 and Section 2.2.

The 1998 DARPA Intrusion Detection Evaluation training data contains the following information for every day of the evaluation:

- tcpdump data collected by the tcpdump packet sniffer
- Sun *Basic Security Modules* (BSM) audit data from one UNIX Solaris host in the inside network for some network sessions
- ps-monitor file containing process status information of a machine running BSM
- UNIX filesystem snapshots of a machine running BSM
- *listfiles* for tcpdump and BSM audit data with additional information for selected network sessions

The tcpdump data contains information about every packet transmitted between devices on the inside and the outside networks. BSM audit data contains audit information describing system calls made to the Solaris kernel. The ps-monitor file contains the output of the UNIX command *ps* running periodically. The additional information provided by the listfiles basically adds session start time, session duration, source and destination port, source and destination IP, and attack name for the tcpdump files and the BSM audit data. The training set contains a total of 22 different attacks types.

The test data used to evaluate a trained intrusion detection system provides the same sensor data except the listfiles with the labelled sessions containing the attacks. The test set contains approximately 114 instances of

37 different attacks. Of the attacks, 17 are new and not part of the training set. Two attacks appear only in the training data.

The tcpdump data provided by the 1998 DARPA Intrusion Detection Evaluation network was further processed and used for the 1999 KDD Cup contest at the fifth International Conference on Knowledge Discovery and Data Mining[1]. The learning task of this competition was to classify the preprocessed connection records into either normal traffic or one out of the four given attack categories (‘dos’, ‘probe’, ‘r2l’, ‘u2r’).

The seven weeks of network traffic collected from the DARPA training data were preprocessed into five million labelled and categorised connection records of approximately 100 bytes each; and the two weeks of training data were processed into two million unlabelled connection records. Preprocessing of the DARPA data for the 1999 KDD Cup contest was done with the MADAMID framework described in [Lee 1999] and [Lee & Stolfo 2000]. The KDD Cup ’99 datasets are available from the UCI KDD Archive as the 1999 KDD Cup Dataset [Hettich & Bay 1999].

A connection record summarises the packets of a communication session between a connection initiator with a specified source IP address and a destination IP address over a pair of TCP/UDP ports. The labelled connection records in the training set are either categorised as ‘normal’ or indicate one of 22 types of attack. As far as we know, the KDD Cup ’99 dataset is, as of today, still the only publicly available dataset with fully labelled connection records spanning several weeks of network traffic and a large number of different attacks.

Each connection record contains 41 input features—34 continuous- and 7 discrete-valued—grouped into *basic features* and *higher-level features*. The *basic features* are directly extracted or derived from the header information of IP packets and TCP/UDP segments in the tcpdump files of each session (basic features 1–9 in Table A.1 on Page 182). This was done by using a modified version of the freely available *Bro Intrusion Detection System*[2] presented in [Paxson 1999]. Each connection record was produced when either

[1] The KDD Cup is an annual *Knowledge Discovery and Data Mining competition* organised by the ACM Special Interest Group on Knowledge Discovery and Data Mining.

[2] http://bro-ids.org/

the connection was terminated or Bro was closed. The listfiles for tcpdump from the DARPA training data where used to label the connection records.

The so-called *content-based higher-level features* use domain knowledge to look specifically for attacks in the actual data of the segments recorded in the tcpdump files. These address 'r2l' and 'u2r' attacks, which sometimes either require only a single connection or are without any prominent sequential patterns. Typical features include the number of failed login attempts and whether or not root access was obtained during the session (features 10–22 in Table A.1 on Page 182).

Furthermore, there are *time-based and connection-based derived features* to address 'dos' and 'probe' attacks. *Time-based features* examine connections within a time window of two seconds and provide statistics about these. To provide statistical information about attacks exceeding a two-second time-window, such as slow probing attacks, *connection-based* features use a connection window of 100 connections. Both are further split into *same-host features*, which provide statistics about connections with the same destination host, and *same-service features*, which examine only connections with the same service (features 23–41 in Table A.1 on Page 182).

The KDD Cup '99 competition provides the training and testing datasets in a full set, and also a so-called '10%' subset version. The '10%' subset was created due to the huge amount of connection records present in the full set; some 'dos' attacks have millions of records. For this reason, not all of these connection records were selected. Furthermore, only connections within a time-window of five minutes before and after the entire duration of an attack were added into the '10%' datasets. To achieve approximately the same distribution of intrusions and normal traffic as the original DARPA dataset, a selected set of sequences with 'normal' connections were also left in the '10%' dataset. Training and test sets have different probability distributions.

The full training dataset contains nearly five million records. The full training dataset and the corresponding '10%' both contain 22 different attack types in the order that they were used during the 1998 DARPA experiments.

The full test set, with nearly three million records, is only available unlabelled; but a '10%' subset is provided both as unlabelled and labelled test data. It is specified as the 'corrected' subset with a different distribution and additional attacks not part of the training set. For the KDD Cup '99 competition, the '10%' subset was intended for training. The 'corrected' subset can be used for performance testing; it has over 300,000 records containing 37 different attacks.

The distributions of the four different attack classes in the full training set, '10%' training set, and the 'corrected' test set is shown in Table 4.4. It is to be noticed that the sample distribution of 'probe', 'r2l' and 'u2r' attacks varies strongly between the training sets and the test set.

Table 4.4: *The varying distributions of the five traffic classes in the KDD Cup '99 datasets. The distributions of remote-to-local ('r2l') and user-to-root ('u2r') attacks vary strongly between the training set and the test set.*

traffic class	full train		10% train		10% test	
normal	972781	19.8590%	97278	19.6911%	60593	19.4815%
dos	3883366	79.2778%	391458	79.2391%	229853	73.9008%
probe	41102	0.8391%	4107	0.8313%	4166	1.3394%
r2l	1126	0.0230%	1126	0.2279%	16347	5.2558%
u2r	52	0.0011%	52	0.0105%	70	0.0225%
$\sum$ attacks	3925646	80.1409%	396743	80.3089%	250436	80.5185%
$\sum$ records	4898427	100%	494021	100%	311029	100%

4.5 Extracting Salient Features

The majority of published results observing feature reduction on the KDD Cup '99 datasets are trained and tested on the '10%' training set only (see [Sung 2003], [Kayacik *et al.* 2005] and [Lee *et al.* 2006]). Some researchers used custom-built datasets with over 10,000 random records extracted from the '10%' KDD Cup '99 training set (see [Chavan *et al.* 2004], [Chebrolu *et al.* 2005] and [Chen *et al.* 2005]). These sets were split into training and test records. Due to the fact that 'r2l' and 'u2r' attacks very seldom occur in the training data, and produce very few connection records per attack, the results for these attacks cannot be very meaningful, even if corresponding records are manually added to the training data. Furthermore, these results using only the KDD Cup '99 training data cannot be directly compared to results using the original test set. One strong reason for this is the widely differing traffic types and distributions of these two sets.

[Sung 2003] applied single-feature deletion to the KDD Cup '99 datasets using neural networks and support vector machines. Using the SVM classifier, they extracted a 30-feature set with improved training time and, in terms of accuracy, comparable performance. With the neural network classifier using 34 important features, they improved training time and false negative rate, but with a significant loss of accuracy. For the SVM classifier, they also reduced the number of features for the five individual traffic classes to 25 ('normal'), 7 ('probe'), 19 ('dos'), 8 ('u2r') and 6 ('r2l').

Important input features with a focus on building computationally efficient intrusion detection systems were identified by [Chebrolu *et al.* 2005]. They investigated the performance of Bayesian networks, and classification and regression trees. Both classifiers already provide methods for significant feature selection; Bayesian networks use the Markov blanket model, whereas classification and regression trees use the Gini impurity measure. The feature reduction using the Markov blanket model found 17 important features. Using classification and regression trees, only primary splitters were considered, resulting in a set of 12 important features. The authors conclude by suggesting a hybrid model using both classifiers.

[Chavan *et al.* 2004] use a decision tree approach for feature ranking per class. For evaluation, they use artificial neural networks and fuzzy inference systems. The authors reduce the number of features to 13 ('normal'), 16 ('probe'), 14 ('dos'), 15 ('u2r') and 17 ('r2l').

[Kayacik *et al.* 2005] investigated the relevance of each feature provided in the KDD Cup '99 intrusion detection dataset in terms of information gain. The paper presents the most relevant feature for each individual attack that occurs in the training set. An important result is that 9 features make no contribution whatsoever to intrusion detection.

[Chen *et al.* 2005] reduce the number of input features using the flexible neural tree Model to 4 ('normal'), 12 ('probe'), 12 ('dos'), 8 ('u2r') and 10 ('r2l').

A genetic feature selection method, based on feature weighting, was proposed by [Lee *et al.* 2006]. The proposed genetic algorithm wrapper approach is compared to a non-linear filter. Performance was measured using a selective naïve Bayes classifier. Both methods extracted a total of 21 important features, with 11 features in common. The overall performance of the genetic feature selection method shows a slight improvement in terms of accuracy. The proposed approach was especially effective in detecting unknown attacks.

From these experiments, we conclude that the potential for feature reduction is significant. At least a quarter of the features provided by the KDD Cup '99 datasets seem to be unimportant for classifying the observed attacks.

4.5.1 Custom Data Preparation and Preprocessing

The initial preprocessing of the network data collected at the 1998 DARPA Intrusion Detection Evaluation network for the 1999 KDD Cup contest was done with the MADAMID framework described in [Lee 1999] and [Lee & Stolfo 2000]. The connection records of the KDD Cup '99 dataset contains continuous and nominal (discrete) features preprocessed in very different ways.

The continuous features are in various ranges and some have very large values (up-to 700M). The number of discrete values of the nominal features range from three ('protocol_type') to 71 ('services').

We estimated MLP neural networks and SVMs as the strongest candidates for classification of the network datasets. Neural networks require floating point numbers for the input neurons, preferably in the range $[-1, 1]$, and floating point numbers in the range $[0, 1]$ for the target neurons. All features were preprocessed to fulfil this requirement. Scaling the input values to $[0, 1]$ is possible as well, but $[-1, 1]$ works better, since any scaling that sets the mean closer to zero improves the value of the feature.

Preprocessing was done using our own customised network feature preprocessing scripts. For nominal features with three distinct values, we used effects coding mapping from one or two input features, such as protocol type: UDP $= [0, 1]$, protocol type: ICMP $= [1, 0]$, and protocol type: TCP $= [-1, -1]$. For nominal features with a large number of distinct values, we first mapped to ordered numbers using a least-first ranking score. Then we scaled the numbers to the range $[-1, 1]$, for example flag: S3 (50 occurrences) $= -1$ and flag: SF (3744328 occurrences) $= 1$. We chose the ranking order according to the number of occurrences in the test set.

The nominal target value 'connection type', containing the specific traffic label, is first mapped to one of the five connection classes ('normal', 'dos', 'probe', 'r2l', 'u2r'), according to the categorisation script by W.Lee used in the KDD Cup '99 contest scoring.[3] Then each class is represented by its own output feature having a binary value, such as connection type: 'normal' $= [1, 0, 0, 0, 0]$ and connection type: 'r2l' $= [0, 0, 0, 1, 0]$. We removed features with non-changing values from the training data, such as 'num_outbound_-cmds' and 'is_host_login'.

For the numeric features 'duration', 'src_bytes', and 'dst_bytes', we started with the removal of outliers before scaling the values to the range $[-1, 1]$. This was done by reducing the maximum value of each feature to a manually defined threshold; for example, duration: Maximum 30,000 sec. We estimated the threshold values using expert knowledge on expected maximum values to be considered for 'normal' connections. Before normalisation, we finally applied the natural logarithm to the continuous features with strongly biased distributions having a lower bound but no upper bound. For all operations, we used a precision of 10^{-6}.

After preprocessing, our datasets consisted of 39 input features and one output feature, where the input features were mapped to 40 inputs, and the output feature was mapped to the 5 outputs.

4.5.2 Visualisation of Class Distributions

For feature reduction, we used a custom-built training set with 10,422 instances and the original '10%' KDD Cup '99 training set. For testing, we applied 10-fold cross-validation or used the original KDD Cup '99 test set.

The custom training set was extracted from the full KDD Cup '99 dataset to optimise training performance. One aim was to improve the attack distribution in favour of rare attack traffic patterns. The dataset contains 10,422 connection records, including all 41 features. It was sampled and

[3] http://www-cse.ucsd.edu/users/elkan/tabulate.html (2009-05-08)

randomised from up to 1,000 samples out of the 23 traffic types contained in the full dataset. We preprocessed all features as described in Section 4.5.1 using our custom preprocessing scripts.

The traffic types and their occurrences in the three training sets used (full, '10%' and the generated 10,422 connection set), as well as the '10% corrected' test set, are shown in Table A.2 on Page 183.

The WEKA data mining suite was applied for data visualisation and classification. WEKA provides a large number of different machine learning algorithms[4][Witten & Frank 2005]. For classification, we applied the C4.5 decision tree algorithm (in WEKA specified as J4.8), naïve Bayes, Bayesian networks, standard backpropagation with a multilayer perceptron feed-forward neural network (MLP) and support vector machines (SVM, in WEKA specified as SMO) to the DARPA/KDD Cup '99 training data. All classifiers were run with WEKA's default parameters, unless explicitly stated otherwise.

We visualised the distributions of the five different traffic classes in the training data using distribution histograms and scatter plots.

4.5.2.1 Distribution Histograms

First and foremost, we investigated the distribution of features in the data by visualisation using histograms. In a distribution histogram, the value of the feature is plotted against how often the value exists in the data. In the histograms, outliers and skewed distributions are easily detected. This gives valuable advice on necessary preprocessing steps, such as data cleaning and necessary data transformations. After preprocessing the data, the histograms can reveal obvious correlations between features and target classes. The distribution histograms have been proven very valuable for deciding on and optimising the necessary preprocessing steps.

The investigation of the original training data using distribution histograms, shown in Figure A.1 on Page 184, revealed that the features 'num_outbound_cmds' and 'is_host_login' have no variance at all in the training data. They always have a zero value, and so do not provide any information. We removed them from all datasets. The features 'duration',

[4] http://www.cs.waikato.ac.nz/ml/weka/ (2009-05-08)

'src_bytes' and 'dst_bytes' have strongly biased distributions. Furthermore, these features contain, in comparison to their average values, some huge outliers. We decided to threshold connections longer than 30,000 sec. (8 h 20 m) and larger than three megabytes to this maximum value. For these three features, we also applied the natural logarithm to all values. Feature preprocessing is described in more detail in Section 4.5.1.

Figure A.2 on Page 185 shows the distributions of the features in the '10%' training set after preprocessing. Now, we note some obvious correlations between individual features and the 'dos' attack target class. The features 'protocol_type', 'service', 'flag', 'src_bytes', 'dst_bytes', 'land', 'wrong_fragment', 'count', 'srv_count', 'serror_rate', 'srv_serror_rate', 'same_srv_rate', 'diff_srv_rate', 'srv_diff_host_rate', 'dst_host_srv_count', 'dst_host_same_srv_rate', 'dst_host_serror_rate', 'dst_host_srv_serror_rate', 'dst_host_rerror_rate', and 'dst_host_srv_rerror_rate' strongly correlate with 'dos' attacks.

Unfortunately, the occurrences of network probes, 'r2l' and 'u2r' attacks are much too rare to be plotted in sufficient numbers in the histograms shown in Figures A.1 on Page 184 and A.2 on Page 185 to visually attract attention. We addressed this by using our custom-built dataset with 10,422 instances described in Section 4.5.2. This dataset holds a far more favourable distribution for visually investigating these rare attack types. The corresponding histogram is shown in Figure A.3 on Page 186. It reveals that:

'dos' attacks and network probes correlate with the features 'protocol_type', 'flag', 'count', 'srv_count', ''serror_rate', 'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate ', 'same_srv_rate', 'dst_host_srv_count', 'dst_host_diff_srv_rate', 'dst_host_serror_rate', 'dst_host_srv_serror_rate', 'dst_host_serror_rate' and 'dst_host_srv_serror_rate';

network probes correlate with the features 'service', 'flag', 'src_bytes', 'serror_rate', 'rerror_rate', 'same_srv_rate', 'diff_srv_rate', 'srv_diff_host_rate', 'dst_host_srv_count', 'dst_host_same_srv_rate', 'dst_host_diff_srv_rate', 'dst_host_same_src_port_rate', 'dst_host_srv_diff_host_rate', 'dst_host_serror_rate' and 'dst_host_rerror_rate';

the features 'duration', 'service', 'src_bytes', 'urgent', 'hot', 'num_failed_logins', 'su_attempted', 'num_root', 'num_file_creations', 'num_-

access_files', 'is_guest_logged_in', 'dst_host_srv_diff_host_rate' correlate with 'r2l' attacks;

and the features 'root_shell', 'urgent', 'num_failed_logins', 'root_shell', 'num_root', 'num_file_creations', 'num_shells', 'num_access_files' correlate with 'u2r' attacks.

4.5.2.2 Scatter Plots

Another applied data visualisation method that reveals relationships between investigated features are scatter plots. A scatter plot is a plot of two variables against each other. A scatter plot matrix shows all pairwise scatter plots on a single page. Relationships between variables can be identified by a non-random distribution of the points in the plot. Furthermore, scatter plots show the presence of outliers.

Each scatter plot provides information about the strength, shape and direction of the relationship between two features. The more points are clustered along a line, the stronger is the relationship between the observed variables. The relationship is positive if the line goes from lower-left to upper-right, and negative when contrariwise. The shape of the line can be linear or curved, a curve usually being quadratic or exponential.

We build scatter plot matrices from the remaining features after removal of features with low information gain and decision tree pruning. We considered feature pairs showing strong correlations in the scatter plots as candidates for further removal.

4.5.3 Feature Extraction using Decision Tree Pruning

Prior to applying feature selection using decision trees, we compared information gain for the original '10%' KDD Cup '99 training data and the training data after preprocessing. The results, shown in Figure 4.3, confirm that the majority of features actually benefit from the preprocessing steps. An overview of all features is presented in Table A.1 on Page 182. All applied preprocessing steps are explained in detail in Section 4.5.1.

The first six basic features all improve as a result of feature preprocessing. The strongest beneficiaries are the features 'src_bytes' and 'dst_bytes', where we removed the outliers and improved the distribution by applying the natural logarithm. Applying effects coding to the 'protocol_type' feature also had a strong positive impact. The other three basic features, 'land', 'wrong_-fragment' and 'urgent', and most of the following content features, are of low significance for the majority of the connection records. Exceptions are the features 'hot', 'num_failed_logins', 'logged_in', and 'is_guest_login'.

The time-based and connection-based features all show information gain significance for the classification of the data. The features 'count', 'srv_count', 'rerror_rate', 'srv_rerror_rate', 'same_srv_rate', 'diff_srv_rate', 'srv_-diff_host_rate', 'dst_host_count', 'dst_host_same_srv_rate', 'dst_host_-diff_srv_rate', 'dst_host_same_src_port_rate', 'dst_host_rerror_rate', and 'dst_host_srv_rerror_rate' even show improved significance after normalisation.

In terms of information gain, the features 'hot', 'logged_in', 'num_root', 'num_access_files', 'serror_rate' , 'srv_serror_rate', 'dst_host_count', 'dst_host_srv_diff_host_rate', 'dst_host_serror_rate', and 'dst_host_-srv_serror_rate' suffered slightly from normalisation. But since this transformation was lossless, we do not expect any significant negative impact on classification.

Feature selection was done by building and examining post-pruned decision trees. The applied J4.8 decision tree algorithm implements subtree raising as a pruning operation. In subtree raising, the decision tree algorithm moves nodes up towards the root of the tree and discards other nodes on the way.

After the first build from the training set using all features, we removed features from the dataset that were not part of the tree. Then we continued with a leave-one-out reduction until the removal of any feature led to significant performance loss in any of the five applied classifiers. We used true positive rate, false positive rate, precision, accuracy and costs as performance metrics in each traffic class. We also frequently estimated the ROC curve and calculated the *area under curve* (AUC) value using the Mann Whitney statistic. All values, except costs, were provided by WEKA. Costs were manually calculated using the suggested values provided by the KDD Cup '99.

To limit the number of iterations, our leave-one-out approach was biased.

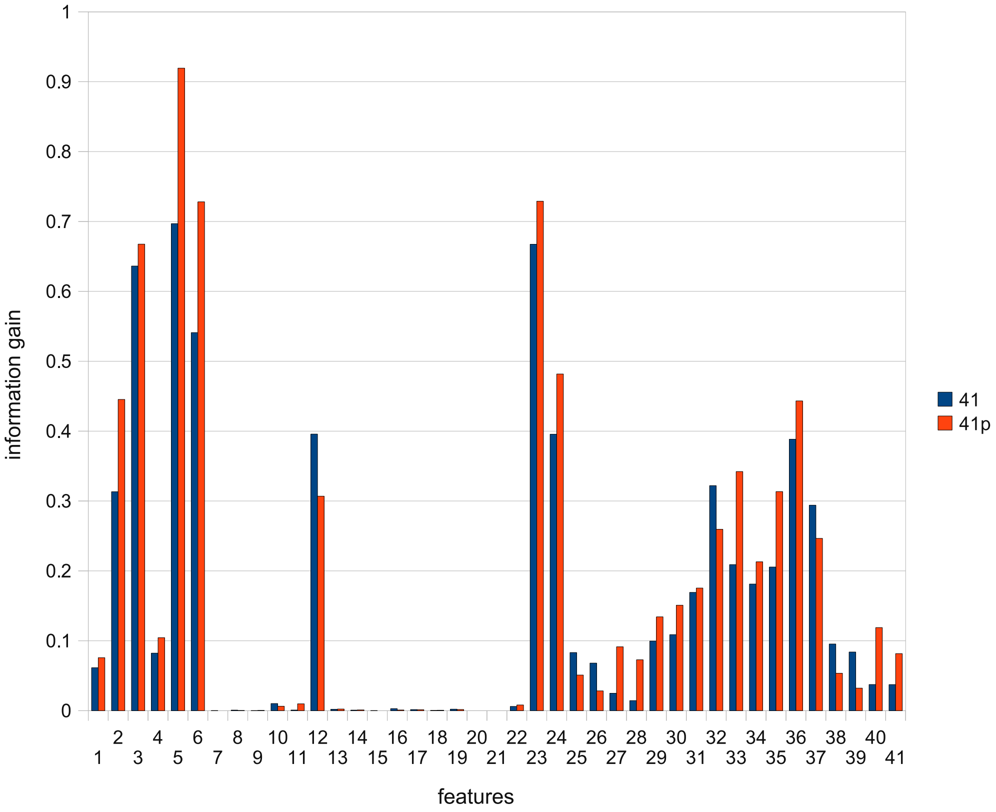


Figure 4.3: *Comparison of information gain of all features in the original and the preprocessed KDD Cup '99 '10%' training data. An overview of the features is shown in Table A.1 on Page 182. The strongest beneficiaries are the features 'src_bytes' and 'dst_bytes'. Some features also suffered slightly from normalisation.*

By default, we kept features close to the root of the tree, and one-by-one, removed features close to or at leaves. We preferred the removal of features that require domain knowledge or detailed traffic data analysis to features easily extracted from network data. We also frequently observed the classification and run-time performance of the five applied classifiers.

From the observed subsets, in every run with improved or comparable performance, we picked the best-performing attribute set. We declared the absent attribute of the best-performing subset as an unimportant attribute. We tested the performance of the final minimal feature set against the KDD Cup '99 test set.

The applied feature selection algorithm can be summarised as follows:

1. Construct a decision tree from all given training data using the full m attributes set.

2. If not all attributes are used to construct the tree,

   (a) mark the unused attributes as irrelevant attributes a_i, and

   (b) construct a new set with $m = m - a_i$ attributes.

3. Build trees for all possible subsets with $m - 1$ attributes.

4. If at least one subset is found with improved or comparable performance,

   (a) mark the removed attribute of the best-performing subset as unimportant attribute a_u, and

   (b) construct a new set with $m = m - a_u$ attributes.

5. Until all subset trees with $m{-}1$ attributes have a significant performance loss,
   continue with 3.

6. Test found minimal feature set against training set and test set.

4.6 Minimal Sets for All Attacks

We followed two different approaches in order to find minimal sets for detecting all attacks with one trained classifier. The aim of the first approach was to extract a reduced feature set with few, if any, content features. All tested classifiers should at least maintain their performance on the reduced dataset in comparison to using all features.

In our second approach to feature reduction, we searched for essential minimal features, which could still compete with the well-performing classifiers of the KDD Cup '99 challenge. This time, we considered only the results of the best-performing classifier.

4.6.1 The 11 Feature Minimal Set

The first approach resulted in a set of 11 selected features, which consisted of 7 basic features and 4 higher-level features [Staudemeyer & Omlin 2009]. The selected minimal features were 'duration', 'protocol_type', 'service', 'source_-bytes', 'dst_bytes', and 'wrong_fragment'. Chosen higher-level features were 'serror_rate', 'dst_host_srv_count', 'dst_host_diff_srv_rate', 'dst_host_-same_src_port_rate', and 'dst_host_rerror_rate'.

Figure A.4 on Page 187 shows a scatter plot matrix of the 11 features using the custom training set with 10,422 instances. The scatter plots show that there are strong correlations between the 5 selected higher-level features. This is due to the fact that not all of these 5 features are essential for all five traffic classes. The strong correlations relate to 'dos' and 'probe' attacks, which both generate large numbers of connection records per attack.

The correlations are shown by the diagonal clustering of data points along a line in the scatter plots between these features. At least for some attacks within these two classes, not all selected higher-level features are essential. For the remaining traffic types, there are no strong relationships affecting a noticeable number of connection records between the selected features.

4.6.2 The 8 and 4 Feature Minimal Sets

The exhaustive, feature-by-feature reduction of our second approach led to 8 important features, in which we identified the 4 most important minimal features. The 4 features are 'service', 'src_bytes', 'dst_host_diff_srv_rate', and 'dst_host_rerror_rate'. Additional important features are 'dst_bytes', 'hot', 'num_failed_logins', and 'dst_host_srv_count'.

The '4-1' histogram in Figure 4.4 shows that, in terms of misclassifications, any further feature removal leads to a significant degradation of performance on the non-statistical classifiers. For training, we used our custom training set, and for testing, we applied 10-fold cross-validation.

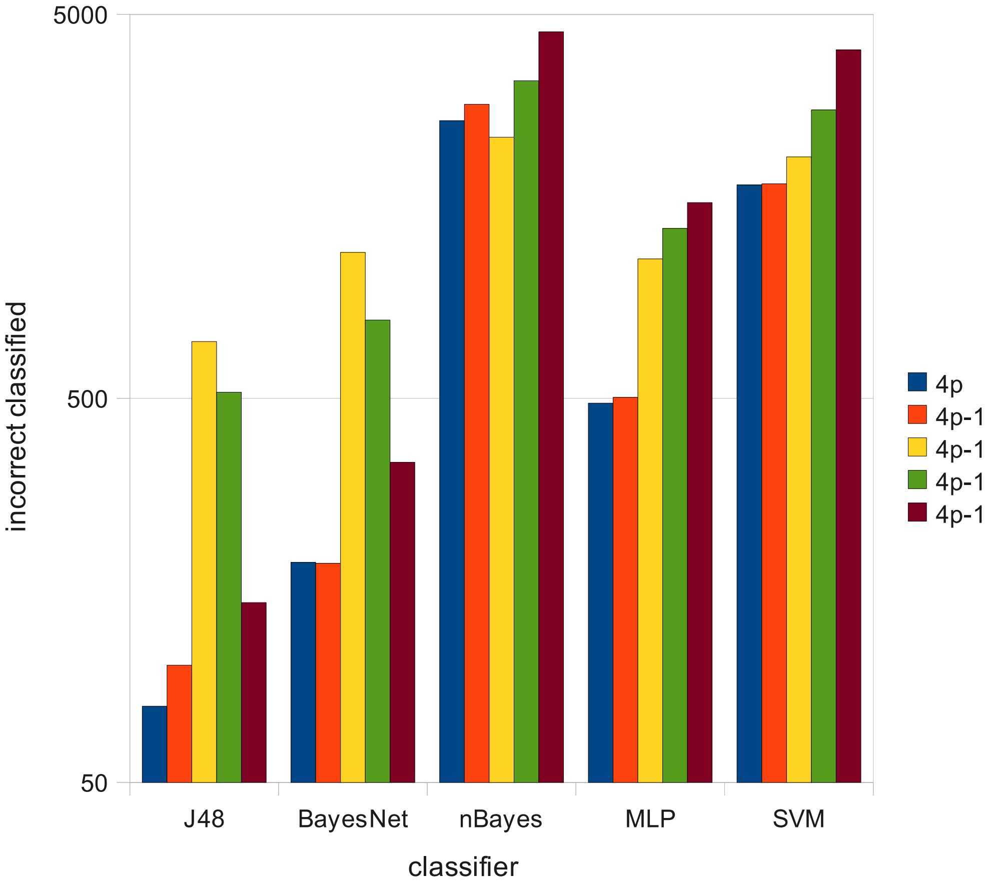


Figure 4.4: *Performance degradation of the 4 minimal feature dataset, removing any of the features. The minimal features are 'service', 'src_bytes', 'dst_host_-diff_srv_rate', and 'dst_host_rerror_rate'.*

4.7 Minimal Sets for Individual Attacks

We split the preprocessed '10%' training dataset and the testing dataset into four sets, each containing all normal traffic, but only one out of the four attack traffic types ('probe', 'dos' , 'r2l', 'u2r'). For each attack, we built pruned decision trees containing only the most relevant features.

Figures 4.5 and 4.6 show the information gain of the four different attacks types for all features in the preprocessed KDD Cup '99 '10%' training data. This allows us a more detailed individual analysis of the preprocessed features for each attack class.

In terms of information gain, the figures show that many attributes are

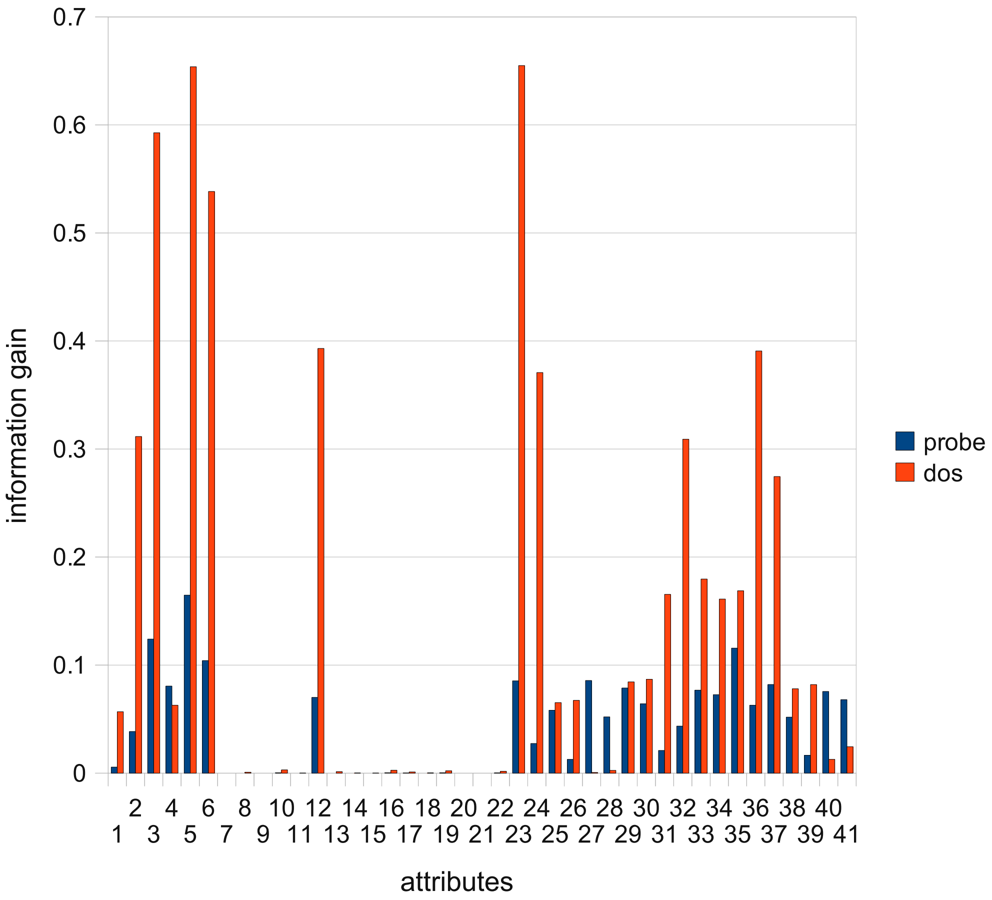


Figure 4.5: *Information gain of network probes and 'dos' attacks, observing all features in the preprocessed KDD Cup '99 '10%' training data.*

not suitable for classifying all four different attack types. We notice that, for network probes and 'dos' attacks, the first 6 basic features and most of the time- and host-based features are relevant. With the exception of the 'logged_in' feature, content features for these attacks are of very low relevance.

For 'r2l' and 'u2r' attacks, it is noticeable that, due to the nature of these attacks, some of the relevant features are content features. The first 6 basic features and all host-based features are significant for 'r2l' attacks. Additionally, the 4 content features 'hot', 'num_failed_logins', 'logged_in', and 'is_guest_logged_in'; and the 3 time-based features 'count', 'srv_count', and 'srv_diff_host_rate', are also significant.

For 'u2r' attacks all features are, if any, of very low relevance. The

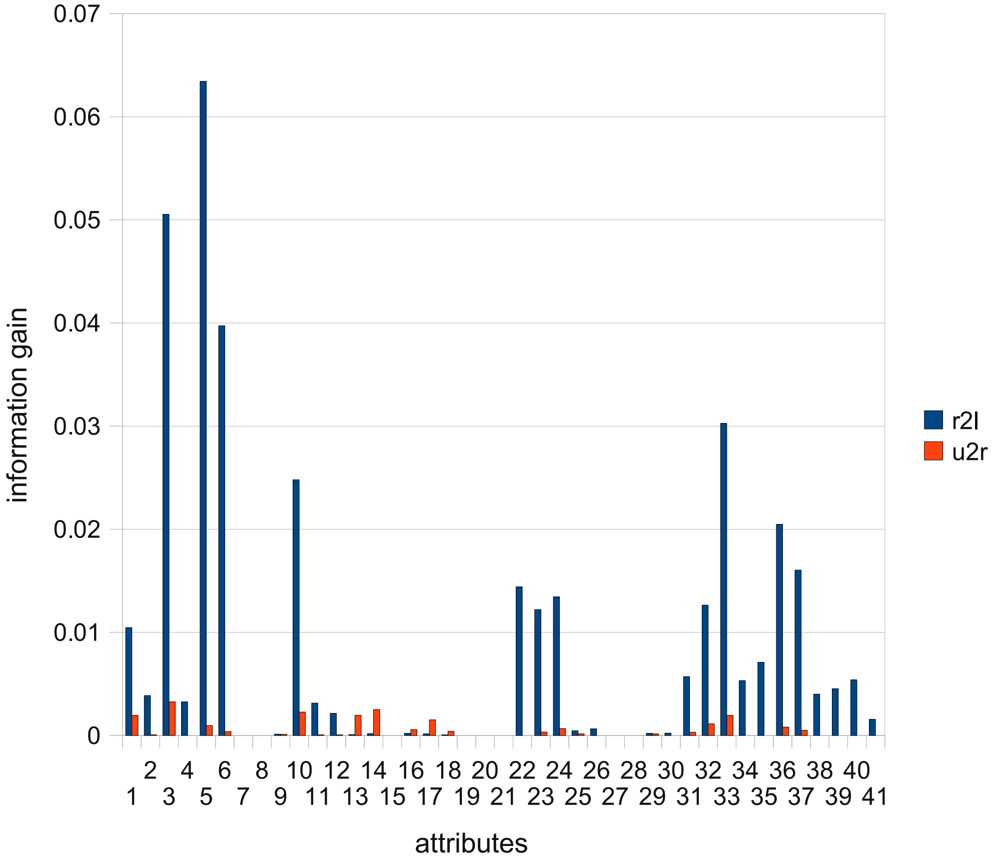


Figure 4.6: *Information gain of 'r2l' and 'u2r' attacks observing all features in the preprocessed KDD Cup '99 '10%' training data.*

10 features with the highest information gain are 'service', 'root_shell', 'hot', 'num_compromised', 'duration', dst_host_srv_count', 'num_file_-creations', 'dst_host_count', 'src_bytes', and 'dst_host_same_src_port_-rate'.

First, we removed all features with no or very little information gain. Examples are the 'land' and the 'su_attempted' features, which are of very low relevance for any attack. Then, we built pruned decision trees and discarded all features that were not part of the pruned tree.

From this set of remaining features, we generated a scatter plot matrix to visualise the remaining relationships between the features and continued with our decision tree feature reduction algorithm as described in Section 4.5.3.

We applied this process to the datasets of each attack type, in order to further reduce the redundancies, until we reached a minimal set with approximately 4–6 features, where further feature removal leads to significant performance loss.

4.7.1 Detecting Network Probes

Observing information gain for features in the network ‘probe’ dataset, we removed the features ‘land’, ‘wrong_fragment’, ‘urgent’, ‘num_failed_logins’, ‘num_compromised’, ‘root_shell’, ‘su_attempted’, and ‘num_shells’, because they do not contribute to the classification of network probes.

After building the pruned decision tree, the remaining ‘14’ features are ‘protocol_type’, ‘service’, ‘flag’, ‘src_bytes, dst_bytes’, ‘logged_in’, ‘same_srv_rate’, ‘dst_host_count’, ‘dst_host_srv_count’, ‘dst_host_same_srv_rate’, ‘dst_host_diff_srv_rate’, ‘dst_host_same_src_port_rate’, ‘dst_host_srv_diff_host_rate’, and ‘dst_host_rerror_rate’.

These ‘14’ features are visualised in the scatter plot matrix shown in Figure A.5 on Page 188. We note that some of the higher-level features still show strong correlations, which confirms the potential for further feature reduction.

After feature reduction, the remaining 6 minimal features are ‘protocol_type’, ‘src_bytes’, ‘same_srv_rate’, ‘dst_host_srv_count’, ‘dst_host_same_srv_rate’, and ‘dst_host_diff_srv_rate’. Observing the scatter plots between these minimal features, we note very few correlations affecting only a small number of connection records.

Furthermore, the ‘6-1’ histogram in Figure 4.7 shows that any further removal of features leads to performance degradation. We also note that the feature ‘src_bytes’ is the most important for successful classification of this traffic class, and its removal causes the most significant increase of misclassification errors. For the histogram, we used the ‘10%’ training set with 10-fold cross-valuation.

Another well-performing set for probe attacks containing only 2 features is that of ‘src_bytes’ and ‘dst_host_same_srv_rate’.

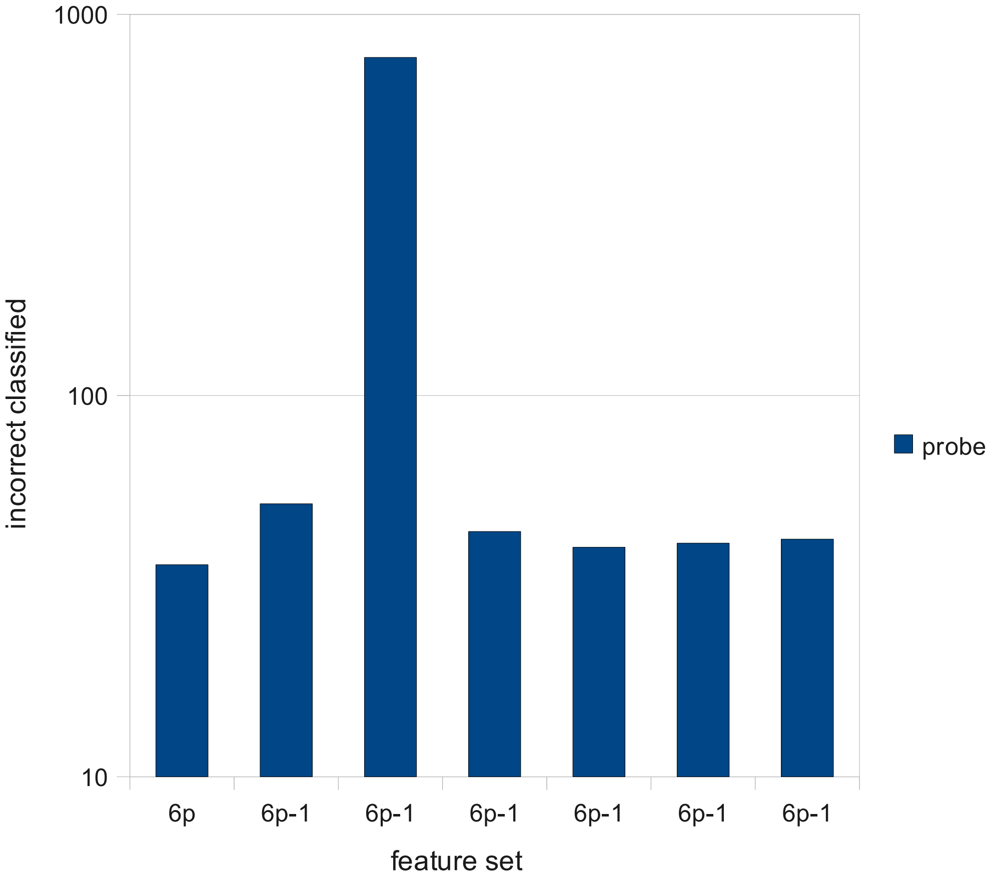


Figure 4.7: *The '6-1' histogram of minimal features related to network probes. The 6 tested minimal features are, from left to right: 'protocol_type', 'src_bytes', 'same_srv_rate', 'dst_host_srv_count', 'dst_host_same_srv_rate', and 'dst_-host_diff_srv_rate', where 'src_bytes' is the most important.*

4.7.2 Detecting 'dos' Attacks

For 'dos' attacks, we removed the features 'land' and 'urgent' due to their lack of information gain. After tree pruning, the remaining 11 features are 'service', 'flag', 'src_bytes', 'dst_bytes', 'wrong_fragment', 'count', 'same_srv_rate', 'dst_host_same_src_port_rate', 'dst_host_serror_rate', 'dst_host_srv_-serror_rate', and 'dst_host_rerror_rate'.

The scatter plot matrix of the important features is shown in Figure A.6 on Page 189. We note that the scatter plots between some higher-level features still show strong correlations.

We extracted a number of different well-performing subsets. One outstand-

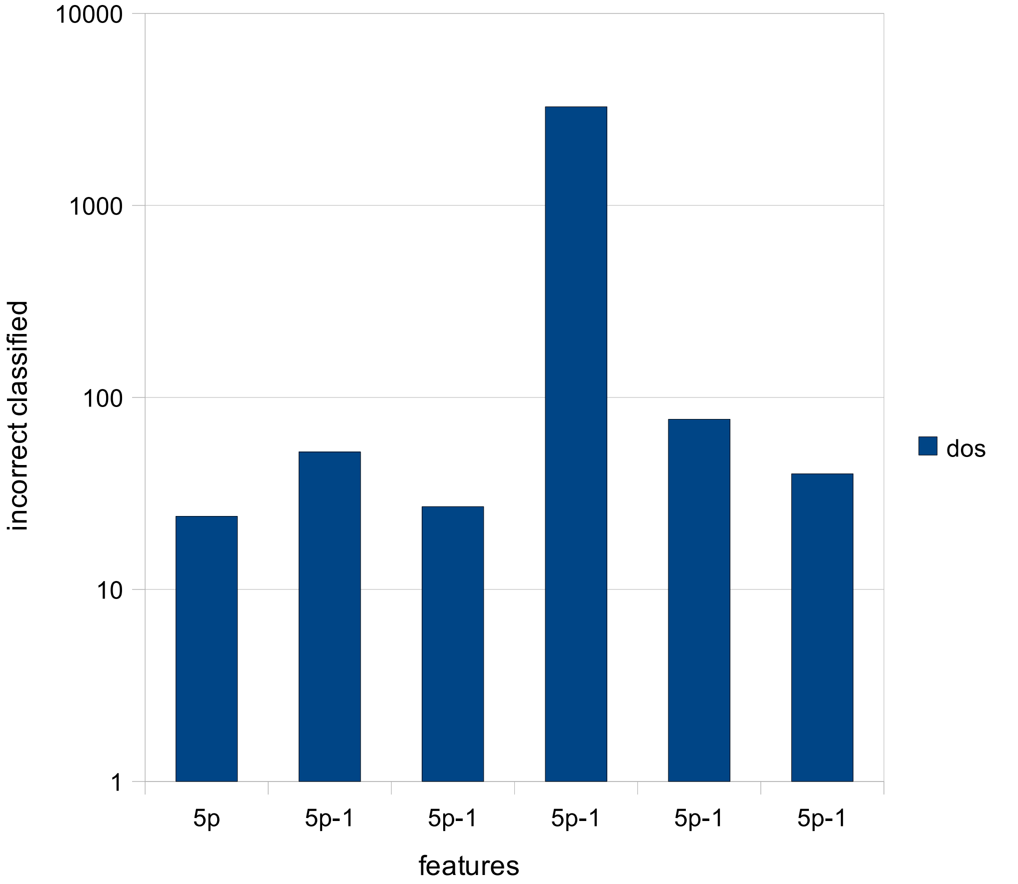


Figure 4.8: *The '5-1' histogram of minimal features for 'dos' attacks using the '10%' training set with 10-fold cross-validation. The 5 tested minimal features are, from left to right: 'service', 'flag', 'src_bytes', 'same_srv_rate', and 'dst_host_srv_serror_rate'. The feature 'src_bytes' proves to be most important.*

ing, well-performing minimal set we found has the 5 minimal features 'service', 'flag', 'src_bytes', 'same_srv_rate', and 'dst_host_srv_serror_rate'. The scatter plots between the minimal features do not show correlations.

The '5-1' histogram shown in Figure 4.8, using the '10%' training set with 10-fold cross-validation, shows that the removal of any feature leads to performance loss. The histogram also shows that, just as for network probes, the feature 'src_bytes' is the most important feature for the classification of 'dos' attacks.

Another well-performing subset contains the features 'service', 'src_bytes', 'same_srv_rate', 'dst_host_same_src_port_rate', 'dst_host_serror_rate', and ‘dst_host_srv_serror_rate’. A 4 feature subset, with an only slightly poorer performance, contains the features ‘service’, ‘flag’, ‘src_bytes’, and ‘count’.

4.7.3 Detecting ‘r2l’ Attacks

For ‘r2l’ attacks, the features ‘land’, ‘wrong_fragment’, ‘su_attempted’, ‘num_access_files’, ‘rerror_rate’, and ‘srv_rerror_rate’ have no information gain and can be discarded. The 18 features remaining after pruning are: ‘duration’, ‘protocol_type’, ‘service’, ‘src_bytes’, ‘dst_bytes’, ‘hot’, ‘logged_in’, ‘root_shell’, ‘num_root’, ‘srv_count’, ‘dst_host_count’, ‘dst_host_srv_count’, ‘dst_host_diff_srv_rate’, ‘dst_host_same_src_port_rate’, ‘dst_host_srv_diff_host_rate’, ‘dst_host_serror_rate’, ‘dst_host_srv_serror_rate’, and ‘dst_host_srv_rerror_rate’.

We did not have enough computing resources for an exhaustive search of this rather large set of 18 feature candidates. For this reason, we removed the features ‘protocol_type’, ‘logged_in’, ‘root_shell’, and ‘num_root’, which have very low information gain for ‘r2l’ attacks, and which we, therefore, considered as least relevant.

The scatter plot matrix of the remaining ‘14’ features is shown in Figure A.7 on Page 190. Again, some of the scatter plots between higher-level features show strong correlations.

After feature reduction, the 6 remaining minimal features are: ‘duration’, ‘service’, ‘src_bytes’, ‘hot’, ‘srv_count’, and ‘dst_host_srv_count’. The scatter plots between these 6 features do not show strong correlations.

The ‘6-1’ histogram is shown in Figure 4.9. This shows once again that ‘src_bytes’ is the most important feature. Another well-performing minimal set of 6 features is: ‘service’, ‘src_bytes’, ‘hot’, ‘srv_count’, ‘dst_host_srv_count’, and ‘dst_host_serror_rate’.

4.7.4 Detecting ‘u2r’ Attacks

In the remaining ‘u2r’ attack class, the features with no information gain are: ‘flag’, ‘land’, ‘wrong_fragment’, ‘su_attempted’, ‘num_access_files’,

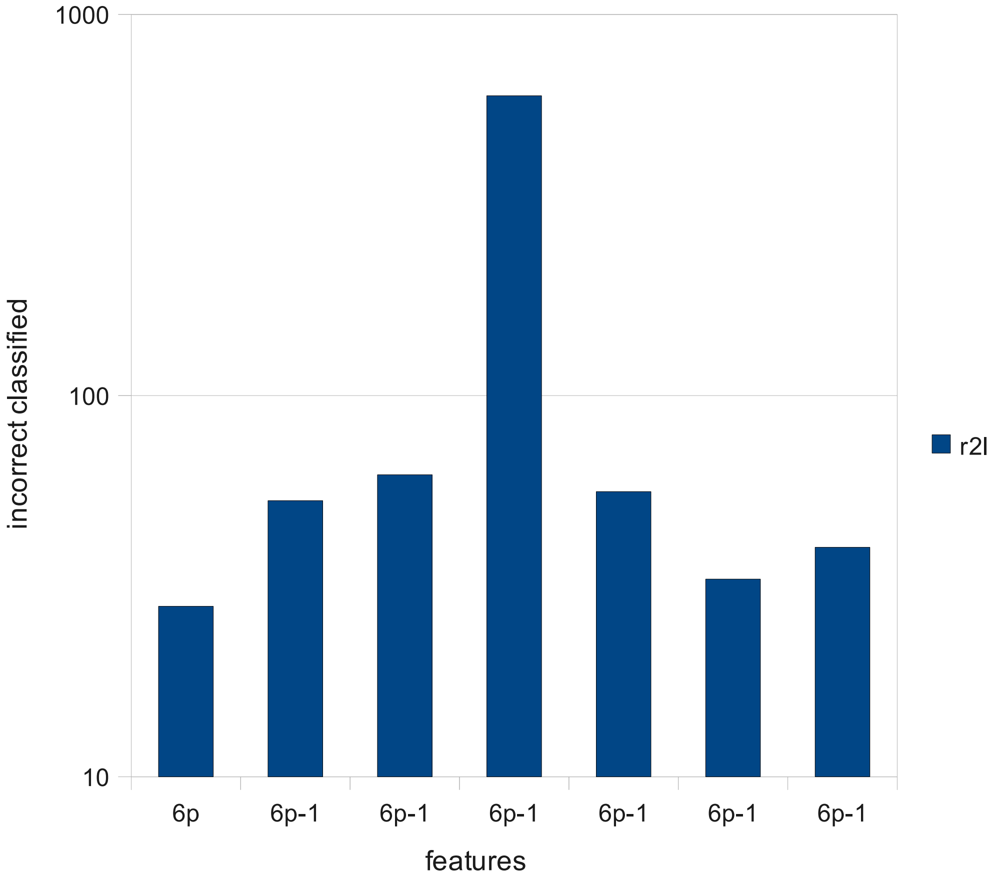


Figure 4.9: *'6-1' histogram of the minimal features for 'r2l' attacks using the '10%' training set with 10-fold cross-validation. The 6 minimal features are: 'duration', 'service', 'src_bytes', 'hot', 'srv_count', and 'dst_host_srv_count', with 'src_bytes' being the most important feature.*

'is_guest_login', 'srv_serror_rate', 'rerror_rate', 'srv_rerror_rate', 'diff_srv_rate', 'dst_host_same_srv_rate', 'dst_host_diff_srv_rate', 'dst_host_serror_rate', 'dst_host_srv_serror_rate', 'dst_host_rerror_rate', and 'dst_host_srv_rerror_rate'.

Pruning reduces the features to 8, namely: 'service', 'srv_bytes', 'dst_bytes', 'hot', 'root_shell', 'num_file_creations', 'dst_host_count', and 'dst_host_srv_count'. Figure A.8 on Page 191 shows the scatter plot matrix of the important features. We note no salient correlations, which might be related to the fact that few examples are available.

After feature reduction, the remaining minimal set containing the 5 min-

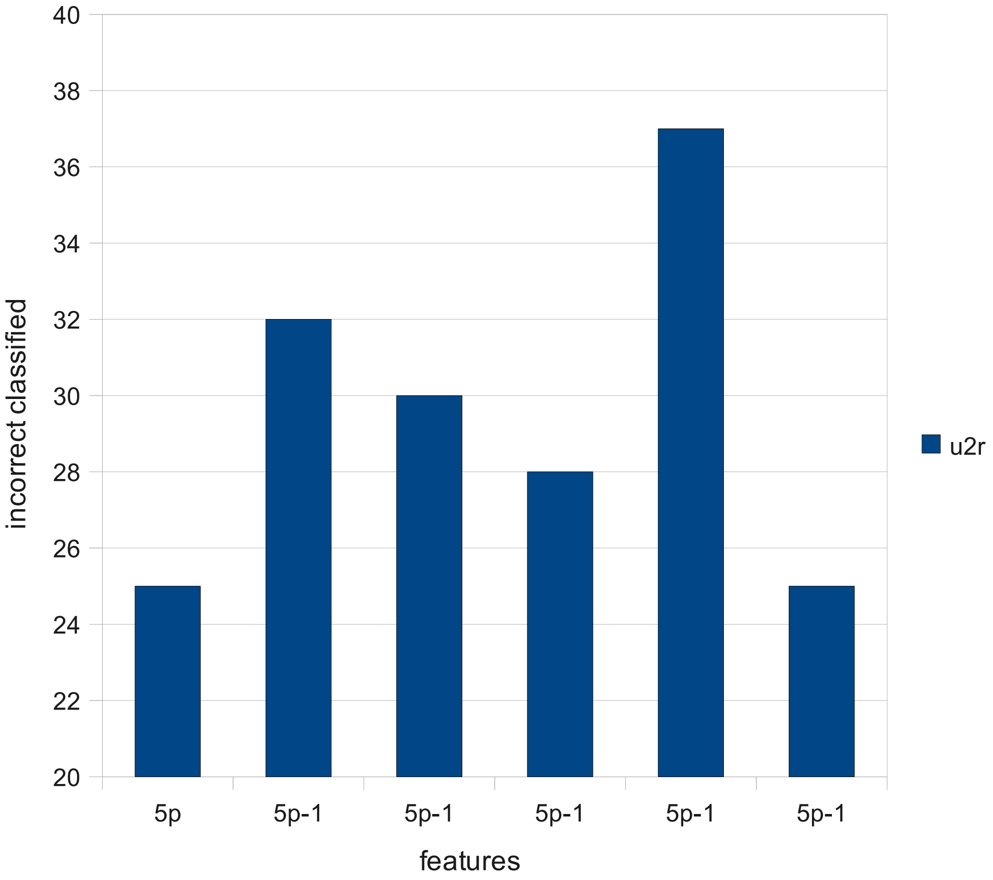


Figure 4.10: *'5-1' histogram of the minimal features for 'u2r' attacks using the '10%' training set with 10-fold cross-validation. The 5 minimal features are: 'src_-bytes', 'dst_bytes', 'hot', 'num_file_creations', and 'dst_host_srv_count', where the first 4 are the most important.*

imal features for most detectable 'u2r' attacks are: 'src_bytes', 'dst_bytes', 'hot', 'num_file_creations', and 'dst_host_srv_count'. Figure 4.10 shows the '5-1' histogram; it shows that the feature 'num_file_creations' is the most important.

4.8 Conclusions

In this chapter, we investigated the potential to reduce the number of features used for classifying the five traffic classes 'normal', 'probe', 'dos', 'r2l', and 'u2r'. Our approach started with preprocessing all features using the custom feature preprocessing framework presented.

We removed features with non-changing values in the training data from all datasets. For continuous features, we removed outliers, applied normalisation, and, if necessary, performed logarithmic scaling. For nominal features, we applied least-first ranking and effects coding. We finally mapped the target features to one of the five traffic classes, as suggested in the KDD Cup '99 challenge.

Then, we visualised the distributions of all features, with the calculation of information gain for each feature. This has already given us a supportive overview of important feature candidates. The distribution histograms and information gain of most features showed significant improvements after preprocessing. Using our custom training set of 10,422 preprocessed instances also helped to visualise the distributions for the three traffic classes 'probe', 'r2l', and 'u2r', with only a few instances in the original data.

We successfully reduced the features for all five traffic classes with a feature selection approach based on decision tree pruning and domain knowledge. We presented minimal sets for the multi-class categorisation of all five classes with 11, 8 and 4 features. For the individual attack classes for extracted minimal sets with 6, 5, 6 and 5 features for 'probe', 'dos', 'r2l' and 'u2r' attacks respectively.

In the 'X-1' histograms, we showed that any further feature removal leads to significant performance degradation of at least one classifier. We visualised the feature relationships of all selected features with scatter plot matrices, using our custom 10,422 instances training set. The plots reveal that very few correlations remain between the selected features.

Our results show that a large number of features are, in fact, redundant or, at least, unimportant. We were able to drastically reduce the number of features from the initial 41 down to 4–8 minimal features for each attack class. An important side-effect is that this extensive feature reduction significantly decreases the computational resources required for training the classifier.

In the next chapter, we dive further into the details of the KDD Cup '99 dataset, summarise criticisms, and compare results previously published. Then, we present our results applying the static classifiers covered in Chapter 3 to the dataset using all features and our minimal feature sets.

Chapter 5

Evaluating Static Classifiers for IDS

Contents

5.1 Introduction

In previous chapters, we looked at different machine learning methods, we compared applicable performance metrics, and we analysed the only publicly available and labelled intrusion detection dataset. Furthermore, we optimised the KDD Cup '99 dataset by preprocessing and evaluated a number of well-known feature reduction methods on the dataset. Then, we presented a method for the extraction of salient features from the KDD Cup '99 data and important features for all attack classes.

In this chapter, we model network traffic using the static classifiers presented in Chapter 3. We start with a look into criticisms and at results previously published on the KDD Cup '99 dataset. We then present our various contributions and evaluate our experimental results with these five classifiers. We use the preprocessed KDD Cup '99 datasets and the reduced feature sets presented in Chapter 4. In this process, we choose a number of suitable performance metrics and we show how our preprocessing and feature reduction of the datasets has a large positive impact on the performance of the classifiers.

We present two experiment series in this chapter. For the first series, we present the results using our 11-feature set. The results of these early experiments were used to optimise our series of operations for preprocessing and feature selection. For the second series, we present a detailed performance comparison of our final experimental results, using all the selected classifiers. For multi-class categorisation and for the detection of individual attack classes, we investigate the performance of the corresponding minimal feature sets in the target range of 4–8 features.

5.2 Criticism of the DARPA Datasets

A short time after the 1998 and 1999 DARPA intrusion detection system evaluations, [McHugh 2000] wrote a detailed critique, identifying shortcomings of the provided datasets. The primary criticism of the paper was that the evaluation failed to verify that the network realistically simulated a real-world network. [Mahoney & Chan 2003] looked more closely at the content of the 1999 DARPA evaluation tcpdump data and discovered that the simulated traffic contains problematic irregularities. The authors state that many of the network attributes, which have a large range in real-world traffic, have a small and fixed range in the simulation. Since the 1998 evaluation data was generated by the same framework, it can be assumed that it suffers from similar problems.

[Sabhnani & Serpen 2004] investigated the reasons why classifiers fail to detect most of ‘r2l’ and ‘u2r’ attacks in the KDD Cup ’99 datasets. They conclude that it is not possible for any classifier to accomplish an acceptable detection rate for these two attack classes. The authors concede that this might be not be the case when the KDD Cup ’99 datasets are used in an anomaly detection context.

[Brugger & Chow 2005] applied the tcpdump traffic data files provided with DARPA datasets to the Snort intrusion detection system. The performance of this mainly signature-based intrusion detection system was rather poor. The authors reason that this is due to the fact that it is difficult to detect ‘dos’ and ‘probe’ attacks with a fixed signature. So, the detection of the ‘r2l’ and ‘u2r’ attacks is, in contrast, much better. The paper emphasises the need to build a more realistic intrusion detection dataset, with a focus on false positive evaluation and more recent attacks. For a detailed description of the Snort IDS, see [Roesch 1999].

5.3 Results of the KDD Cup ’99 Competition

During the KDD Cup ’99 competition, 24 entries were submitted. The first three places used variants of decision trees and showed only marginal differences in performance. The first 17 submissions of the competition were all considered to perform well. A summary of all results is provided by [Elkan 2000].

The winning entry, presented by [Pfahringer 2000], used a variant of the C5 decision tree algorithm, with cost-sensitive bagged boosting. For training, 50 samples were drawn from the full training data, enforcing a custom distribution and removing duplicates. Then an ensemble of 10 C5 decision trees were generated from each sample, both C5’s error-cost and boosting options. The authors calculate the final predictions, on top of the 50 single predictions of the sub-ensembles, by minimising the conditional risk.

The second-placed decision tree solution, by [Levin 2000], used the data-mining tools ‘Kernel-Miner’ to build an optimal decision forest. Therefore, the provided ‘10%’ training data was divided into a set of partitions. A dedicated decision tree was constructed for each partition.

Third place was awarded to a solution labelled as *MP13*, by

Table 5.1: *Results of applying a variant of the C5 decision tree classifier to the KDD Cup '99 datasets. This winning entry of the challenge was presented by [Pfahringer 2000].*

		prediction					
		normal	probe	dos	u2r	r2l	TPR (DR)
actual	normal	60262	243	78	4	6	0.995
	probe	511	3471	184	0	0	0.833
	dos	5299	1328	223226	0	0	0.971
	u2r	168	20	0	30	10	0.132
	r2l	14527	294	0	8	1360	0.084
PRECISION:		0.746	0.648	0.999	0.714	0.988	COST: 0.2331
FPR (FAR):		0.082	0.006	0.003	0.000	0.000	ACC: 92.71%

Table 5.2: *Classifier results after building an optimal decision forest using the KDD Cup '99 datasets. This solution was presented by [Levin 2000] and ranked in second place at the challenge.*

		prediction					
		normal	probe	dos	u2r	r2l	TPR (DR)
actual	normal	60244	239	85	9	16	0.994
	probe	458	3521	187	0	0	0.845
	dos	5595	227	224029	2	0	0.975
	u2r	177	18	4	27	2	0.118
	r2l	14994	4	0	6	1185	0.073
PRECISION:		0.739	0.878	0.999	0.614	0.985	COST: 0.2356
FPR (FAR):		0.085	0.002	0.003	0.000	0.000	ACC: 92.92%

[Vladimir *et al.* 2000]. The authors summarised the method as 'recognition based on voting decision trees using pipes in potential space'. For training data, a custom '10%' training data subset was built, involving some data reduction.

[Agarwal & Joshi 2000] proposed a rule-based classifier model for multi-class classification called PNrule. The model consists of positive and negative rules that predict the presence or absence of a class respectively. Classes could be individual attacks or whole categories, such as 'r2l' and 'u2r'.

In ninth place in the challenge was the 1-nearest neighbour classifier, which showed that simple methods also achieved good results.

The results of the classifiers ranked in first, second and ninth places in the KDD Cup '99 challenge, and the results of the PNrule-classifier are shown in Tables 5.1, 5.2, 5.3 and 5.4.

Table 5.3: *Results of applying a rule-based classifier model for multiclass classification called PNrule to the KDD Cup '99 dataset as suggested by [Agarwal & Joshi 2000].*

		prediction					
		normal	probe	dos	u2r	r2l	TPR (DR)
actual	normal	60316	175	75	13	14	0.995
	probe	889	3042	26	3	206	0.730
	dos	6815	57	222874	106	1	0.970
	u2r	195	3	0	15	15	0.066
	r2l	14440	12	1	6	1730	0.107
PRECISION:		0.730	0.925	1.000	0.105	0.880	COST: 0.2381
FPR (FAR):		0.089	0.001	0.001	0.000	0.001	ACC: 92.59%

Table 5.4: *The 1-nearest neighbour classifier ranked in ninth place at the KDD Cup '99 challenge. This shows that a simple classifier is as well able to achieve good results on the data.*

		prediction					
		normal	probe	dos	u2r	r2l	TPR (DR)
actual	normal	60322	212	57	1	1	0.996
	probe	697	3125	342	0	2	0.750
	dos	6144	76	223633	0	0	0.973
	u2r	209	5	1	8	5	0.035
	r2l	15785	308	1	0	95	0.006
PRECISION:		0.725	0.839	0.998	0.889	0.922	COST: 0.2523
FPR (far):		0.091	0.002	0.005	0.000	0.000	ACC: 92.33%

5.4 Other Results

After the challenge, a number of other results of learning algorithms successfully applied to the KDD Cup '99 data were published. In the following papers, the authors used the same training and testing data as requested in the challenge, and provided comparable results.

[Sabhnani & Serpen 2003] evaluate a comprehensive set of machine learning algorithms and suggest a multi-classifier model. The different algorithms applied were a multilayer perceptron neural network, incremental radial basis function neural network, maximum likelihood Gaussian classifier, k-means clustering, a nearest cluster algorithm, a leader algorithm, a hypersphere algorithm, a fuzzy adaptive resonance theory mapping algorithm, and the C4.5 decision tree. The results showed that certain algorithms gave a higher probability of detection in specific attack categories than others, but that no single algorithm showed superior results in all five attack categories. The authors suggest a multi-class classifier with a multi-class topology, as shown in Figure 5.1.

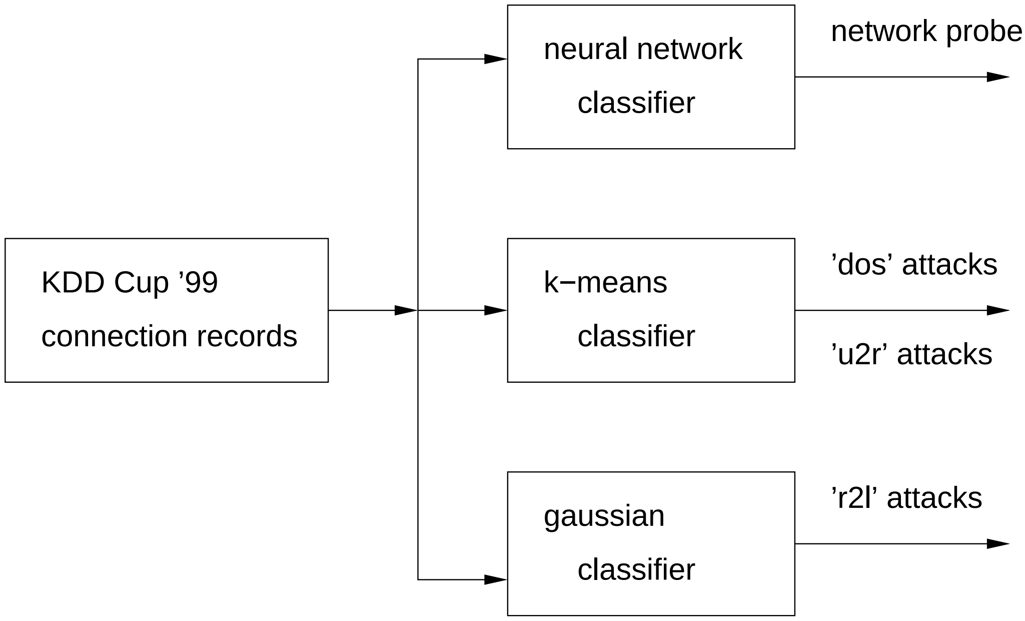


Figure 5.1: *The multi-classifier model suggested by [Sabhnani & Serpen 2003]. Certain algorithms show a higher probability of detection in specific attack categories than others.*

[Hu & Hu 2005] applied the classical Adaboost algorithm and a modified version of the same to the KDD Cup ’99 datasets. The training data was preprocessed within a constructed intrusion detection framework. Decision stumps were chosen as a weak classifier and given as input to Adaboost. The results are comparable to the well-performing KDD Cup ’99 submissions. They show a fair detection rate, with a low false positive rate. This approach offers a noticeable advantage in computational complexity compared to other successful approaches.

[Song *et al.* 2005] demonstrate RSS–DSS, a genetic programming approach for large datasets. The proposed framework divides the data into partitions of 50 patterns, using random subset selection (RSS). In a second processing layer, hierarchical dynamic subset selection (DSS) is applied so that evolution may take place. The authors compared the results of using the first 8 basic features of the dataset to the results using all features. The detection rates of the three larger categories are similar to the better KDD Cup ’99 submissions, but with a much higher false positive rate. The two rare ‘u2r’ and ‘r2l’ attacks suffer from the feature reduction carried out. They are the worst performing categories in terms of large error rates in comparison to the KDD Cup ’99 submissions.

A machine learning approach, based on unsupervised presentation of data, is taken by [Kayacik *et al.* 2007]. They use a multi-layer, self-organising feature-map hierarchy. The datasets were preprocessed and customised for the suggested architecture. Experiments were conducted with the first 6 features using a three-layer hierarchy, and all features using a two-layer hierarchy. Detection rates on the KDD Cup ’99 ‘corrected’ test set are similar to the winning entry of the competition but, not unexpectedly for an unsupervised approach, the false positive rates are three times higher. The authors conclude that the principle reason for this is the lack of suitable boosting algorithms in the field of unsupervised learning.

Machine learning techniques have been applied to network intrusion detection for some time. There are a number of papers with partially comparable results, where the authors used the DARPA or KDD Cup ’99 training data but applied different test sets to their learned classifier.

In an early paper, [Sinclair *et al.* 1999] suggest genetic algorithms and decision trees for automatic rule generation for an expert system that enhances the capability of an existing IDS. [Yeung & Chow 2002] observed a nonparametric density estimation approach, based on Parzen-window estimators with Gaussian kernels.

[Mukkamala *et al.* 2004] compared the performance of a linear genetic programming approach to artificial neural networks and support vector machines. [Abraham & Grosan 2006] investigate the results of linear genetic programming and multi-expression programming, which both outperformed support vector machines and decision trees.

Other hybrid approaches combine neural networks and support vector machines, published by [Mukkamala *et al.* 2003], artificial neural networks and a fuzzy inference system, by [Chavan *et al.* 2004], and decision trees and support vector machines, by [Peddabachigari *et al.* 2007]. The results show that the suggested hybrid approaches provide superior detection. It is also noted that support vector machines outperform neural networks, and that decision trees perform slightly better than support vector machines if the dataset is small.

There are also a number of interesting publications where the results are not comparable due to the use of different training and test datasets. [Debar *et al.* 1992] and [Cannady 1998] suggested the use of neural networks as components of intrusion detection systems.

[Zhang *et al.* 2001] compared the performance of a selection of neural network architectures for statistical anomaly detection to datasets from four different scenarios. In their experiments, backpropagation and perceptron-backpropagation-hybrid neural networks outperformed the other methods.

The use of hidden Markov models to detect complex multi-stage Internet attacks that occur over extended periods of time is described by [Ourston *et al.* 2003]. They determined that hidden Markov models performed slightly better than decision trees and neural networks. An event classification scheme based on Bayesian networks is proposed by [Kruegel *et al.* 2003]. The scheme significantly reduces the number of false alarms in comparison to threshold-based systems like naïve Bayes.

A framework for unsupervised learning, with two feature maps mapping unlabelled data elements to a feature space, is suggested by [Eskin *et al.* 2002]. [Bivens *et al.* 2002] further illustrated that neural networks can be efficiently applied to network data in both a supervised and an unsupervised learning approach. The authors used classifying, self-organising maps for data clustering, and multilayer perceptron neural networks for classification. They trained their system to detect denial-of-service attacks, distributed denial-of-service attacks, and port scans out of packet captures.

[Laskov *et al.* 2005] demonstrate that supervised learning techniques applied to the KDD Cup '99 training data significantly outperform unsupervised methods. The best performance is achieved by non-linear methods, such as support vector machines, multilayer perceptron neural networks, and rule-based methods. Support vector machines proved to be most robust in the presence of unknown attacks.

5.5 Classifier Performance Metrics

To get a baseline, we trained five different classifiers with the KDD Cup '99 training set, using all data features, and tested them against the provided 'corrected' test set. The classifiers are J4.8 decision trees, naïve Bayes, Bayesian networks, multilayer perception neural network (MLP), and support vector machines (SVM). The J4.8 decision tree is an implementation of the C4.5 algorithm introduced by [Quinlan 1993]. All classifiers are part of the WEKA data mining suite. We used the default settings described in the publications underlying the particular classifier.

The performance of the classifiers on the KDD Cup '99 test set were evaluated in terms of true positive rate, false positive rate, precision and accuracy. We estimated these measures as follows:

- true positive rate (TPR) $= \frac{\text{true positive classifications}}{\text{total number of positives}}$
- false positive rate (FPR) $= \frac{\text{false positive classifications}}{\text{total number of negatives}}$
- precision $= \frac{\text{true positive classifications}}{\text{total number of positive classifications}}$
- accuracy (ACC) $= \frac{\text{correct classifications}}{\text{total number of classifications}}$

Additionally, we estimated the costs. To calculate the costs, we used the cost matrix provided by the KDD Cup '99 challenge as presented in Table 5.5. To ensure comparability, we increased the cost of misclassified connection records of the class 'u2r' from 3 to 4. This was necessary, since the original classification script from the KDD Cup '99 challenge allowed some attacks to be classified as either 'u2r' or 'r2l' class. We did not allow dual-classifications, but decided to adjust the costs to minimise the error. Due to the small number of 'u2r' attacks, only a few records are affected by this modification. The average cost of the first 17 entries of the KDD Cup '99 challenge had a range of [0.2331–0.2684]. According to [Elkan 2000], results with cost values within this range are considered to perform well.

Table 5.5: *The cost matrix provided by the KDD Cup ’99 challenge with a minor modification. We increased the cost for classifying ‘u2r’ attacks as ‘normal’ traffic from 3 to 4.*

		prediction				
		normal	probe	dos	u2r	r2l
actual	normal	0	1	2	2	2
	probe	1	0	2	2	2
	dos	2	1	0	2	2
	u2r	4*	2	2	0	2
	r2l	4	2	2	2	0

*was 3 in original KDD Cup ’99 matrix

5.6 Performance Analysis Using All Features

Observing the performance of the investigated classifiers on the original KDD Cup ’99 test set in terms of cost and accuracy, we note a few facts: The naïve Bayes classifier shows a poor performance; not being a very strong candidate in comparison to the other classifiers, which is not unexpected. The poor performance of the SVM is simply as a result of its not being directly applicable to multi-class problems. Here, we expect strengths for the detection of individual attacks. The decision tree, the Bayesian network, and the MLP neural network classifier all show acceptable performance.

We note that after preprocessing, the poor performing naïve Bayes classifier improves slightly in performance, while the Bayesian network and the MLP neural network classifiers maintain their performance. The SVM classifier benefits strongly from preprocessing and achieves very good results; close to the best-performing entries of the competition. And we finally note that the decision tree classifier now shows a better performance than the KDD Cup ’99 winning entry. This is particularly interesting, since the applied cost-sensitive bagged boosting C5 decision tree algorithm is a further development of the C4.5-based decision tree algorithm and is expected to perform better. This indicates that we significantly improved the quality of the KDD Cup ’99 data by conducting data preparation.

Table 5.6 summarises the KDD Cup ’99 test dataset performance of the three available results from the challenge, and our results using the five classifiers tested by us. In the table, the original 41 feature dataset is labelled as ‘41’, and our preprocessed feature dataset is labelled as ‘39p’.

Table 5.6: *Performance comparison of the winning results of the KDD Cup ‘99 challenge to our evaluated classifiers trained with the original 41-feature set and the preprocessed 39-feature set (39p). The SVM classifier strongly benefits from preprocessing and achieves very good results. The J4.8 classifier even beats the KDD Cup ‘99 winning entry.*

10%	classifier	normal		probe		dos		u2r		r2l			
set		TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	ACC	COST
1st	c5	.995	.082	.833	.006	.971	.003	.132	.000	.084	.000	92.71%	.2336
2nd	dforest	.994	.085	.845	.002	.975	.003	.118	.000	.073	.000	92.92%	.2362
9th	1-nn	.996	.091	.750	.002	.973	.005	.035	.000	.006	.000	92.33%	.2530
41	J4.8	.995	.089	.747	.002	.973	.003	.086	.000	.058	.000	92.58%	.2421
	BayesNet	.990	.084	.836	.014	.950	.002	.629	.005	.101	.001	91.19%	.2539
	nBayes	.944	.085	.895	.136	.792	.018	.700	.011	.006	.001	78.18%	.3958
	MLP	.984	.090	.725	.001	.973	.011	.086	.000	.056	.000	92.37%	.2484
	SVM	.983	.110	.748	.003	.951	.004	.371	.000	.056	.000	90.71%	.2795
39p	J4.8	.994	.077	.778	.008	.974	.004	.071	.000	.091	.000	92.89%	.2213
	BayesNet	.991	.084	.832	.015	.950	.000	.629	.005	.102	.001	91.16%	.2539
	nBayes	.959	.079	.881	.135	.792	.017	.814	.003	.115	.006	79.00%	.3768
	MLP	.995	.091	.791	.002	.972	.004	.343	.000	.012	.000	92.34%	.2529
	SVM	.990	.087	.764	.002	.973	.003	.314	.000	.092	.000	92.72%	.2364

5.7 Comparison of Feature Sets

Classifiers initially investigated were decision trees (C4.5, in WEKA specified as J4.8), naïve Bayes, Bayesian networks, and multilayer perceptron feed-forward neural networks (MLP). We ran experiments using all 41 features, using only the basic features, a combination of basic and traffic features, and the 17 and 12-feature sets. After feature reduction, we added experiments with our own minimal feature set using 11 features. For training, we used our custom-built 10,422-instance training set outlined in Section 4.5.2. For testing, we used the 'corrected' test set supplied by the KDD Cup '99 challenge.

The feature sets with 12 and 17 important features are suggested in [Chebrolu *et al.* 2005]. Prior to running experiments with our custom dataset, we confirmed the published results using the '10%' training dataset and the original test set.

5.7.1 Two-Class Categorisation

We started with experiments using feature sets with all, 17 and 12 features. The results of these experiments with the custom-built training set lead us to the assumption that the basic features already contain most of the information required for successful classification for the majority of the connection records. To confirm this, we added experiments with the basic features and a combination of basic and traffic features. With the aim of extracting a reduced feature set, with only a few, if any, content features and taking previous results into account, we built a feature set with only 11 features. This new minimal set is described in detail in Section 4.6.1.

We investigated the classifier performance on the test set in terms of accuracy. The best-performing classifier is the Bayesian network, which shows good performance on all tested feature sets. Using only the basic features, we note that this classifier already performs very well, with 91.21% accuracy. It shows almost no performance loss in comparison to using the full feature set. This classifier also provides the best result, with 91.82% accuracy on the test set, using the 17-feature set.

The neural network shows similar performance to the Bayesian network classifier but fails when using only the basic features. The decision tree and the naïve Bayes classifier both show a rather poor performance on all feature sets. We note that the feature sets with 17 and 12 selected features show a comparable test set performance for all classifiers.

Using our 11 features minimal set, all strong classifiers perform very well. Even the simple naïve Bayes classifier improves significantly in performance in comparison to previous results.

In comparison to the full feature set, the results show that our 11-feature set trained with the customised training set boosts accuracy on the test set. The performance increase is 9.5% for the decision tree algorithm and 7.5% for the naïve Bayes. The Bayesian network and the MLP neural network were nearly able to hold their performance.

The performance results of applying the four selected classifiers on the 10,422 datasets with 41, basic only, basic and traffic, 17, 12 and 11 selected features, using traffic label classification, are presented in Table 5.7.

Table 5.7: *Performance comparison using the 10,422 records training set with traffic label classification. The results show that our 11-feature set trained on the customised training set shows a good performance for all classifiers on the test set in terms of accuracy.*

	accuracy			
classifier → feature set ↓	J4.8 test set	nBayes test set	bayesNet test set	MLP test set
41	81.88%	70.83%	91.21%	91.71%
basic only	79.09%	75.62%	91.20%	61.98%
basic & traffic	81.81%	67.73%	90.90%	91.12%
17	82.07%	71.56%	91.82%	88.77%
12	80.59%	72.76%	90.81%	90.37%
11	91.23%	78.16%	91.19%	90.98%

5.7.2 Multi-Class Categorisation

We proceeded with further investigation of the performance of our 11-feature minimal set in comparison to the full feature set. We modified the datasets by mapping the 40 different traffic types to five traffic classes as shown in Table 4.3. We used the custom 10,422 training set and the test set provided by the KDD Cup '99 competition with traffic type classification into the five classes 'normal', 'dos', 'probe', 'r2l' and 'u2r'. From now onwards, we always trained the classifiers on traffic classes.

Next, we investigated the possibility of optimising the preprocessing process for the selected features to increase performance further. This lead us to the preprocessing steps described in Section 4.5.1. We marked the experiment where we used the preprocessed datasets with a following 'p' ('11p').

After feature reduction, the results show a performance increase of 0.6% in accuracy for the decision tree. The three other classifiers lose performance. The Bayesian network, the neural network, and the naïve Bayes classifier all lose accuracies 1%, 0.5% and 10% respectively.

For the 11-feature minimal set using the preprocessed datasets, we find that in terms of accuracy of decision trees, Bayesian networks and neural networks are able to hold their performance after preprocessing and feature reduction. The decision tree classifier improves even further in performance after preprocessing. Naïve Bayes is the only classifier that loses performance noticeably.

Observing the classifier performance on the test set using the preprocessed 11 features, the results show an impressive 93.20% accuracy for decision trees. In comparison to the 11-feature set prior to preprocessing, the Bayesian network and the neural network hold their performance, and the naïve Bayes classifier improves by 5% in accuracy.

An investigation into the true positive rate and the false positive rate per attack traffic class reveals more interesting details: For the decision tree, the detection of network probes decreases, but the false alarm rate remains stable. The detection of 'u2r' attacks increases. Bayesian networks slightly decrease on the detection of 'probe' attacks, but this comes with an improved false alarm rate. Here, the detection of network probes, 'r2l' and 'u2r' attacks all decrease. The false alarm rate for 'u2r' attacks is reduced. The MLP neural network improves the false alarm rate on the detection of 'dos' attacks. The detection of 'probe' attacks improves with a decrease of the false alarm rate. Detection of 'r2l' attacks is decreased, but detection of 'u2r' attacks is increased. We note that due to the few examples of 'u2r' attacks, the very

Table 5.8: *The results of experiments using the 41, 11 and 11p-feature sets with the 10,422 records training set and the 'corrected' KDD Cup '99 test set. The results show that in terms of accuracy the J4.8 decision tree classifier performs very well using the preprocessed 11-feature set (11p).*

10%	classifier	normal		dos		probe		r2l		u2r		
set		TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	ACC
41	J4.8	.964	.079	.934	.017	.907	.014	.106	.002	.257	.000	91.62%
11	J4.8	.974	.077	.936	.006	.898	.012	.199	.001	.543	.001	92.26%
11p	J4.8	.968	.061	.937	.016	.793	.011	.617	.004	.414	.001	93.20%
41	nBayes	.930	.760	.766	.024	.913	.136	.083	.001	.686	.011	78.09%
11	nBayes	.273	.028	.800	.659	.846	.136	.060	.000	.686	.010	67.75%
11p	nBayes	.919	.062	.697	.072	.969	.198	.159	.007	.571	.002	72.90%
41	BayesNet	.980	.081	.924	.007	.825	.016	.201	.002	.743	.002	91.33%
11	BayesNet	.973	.082	.912	.010	.792	.025	.094	.004	.629	.003	89.97%
11p	BayesNet	.971	.081	.912	.007	.850	.025	.139	.004	.657	.002	90.12%
41	MLP	.977	.083	.942	.004	.815	.003	.169	.002	.600	.002	92.55%
11	MLP	.944	.079	.941	.023	.701	.006	.217	.006	.414	.001	91.79%
11p	MLP	.957	.077	.932	.004	.751	.015	.242	.006	.114	.001	91.52%

low false alarm rates are not very meaningful and difficult to compare.

The results of the experiments with the 41, 11 and 11p features using the KDD Cup '99 test set are summarised in Table 5.8.

5.8 Performance Analysis with Minimal Feature Sets

In following experiments, we additionally investigated the J4.8 decision trees, naïve Bayes, Bayesian networks, and MLP feed-forward neural networks, and also the support vector machines (SVM, in WEKA specified as SMO) classifier. We used the reduced feature sets as described in Sections 4.6.1 and 4.6.2. For these and for all following experiments, we trained using the preprocessed '10%' training set and tested on the KDD Cup '99 test set.

5.8.1 Multi-Class Categorisation

For the preprocessed 11 feature dataset, the best-performing classifier was the SVM. Table 5.9 shows the resulting confusion matrix. In terms of costs, all classifiers except naïve Bayes showed performance comparable to the winning entries of the KDD Cup '99 competition.

Table 5.9: *The resulting confusion matrix of the SVM classifier trained with 11 features on the preprocessed '10%' dataset and tested on the KDD Cup '99 test set. The performance is in terms of cost comparable to the winning entries of the KDD Cup '99 challenge.*

		prediction					
		normal	probe	dos	u2r	r2l	TPR (DR)
actual	normal	60017	235	78	6	6	0.990
	probe	1143	2761	262	0	0	0.663
	dos	6020	137	223696	0	0	0.973
	u2r	57	0	3	0	10	0.000
	r2l	15407	3	17	0	920	0.056
PRECISION:		0.726	0.878	0.998	-	0.930	COST: 0.2463
FPR (FAR):		0.090	0.001	0.007	0.000	0.000	ACC: 92.40%

Using the 8-feature set, the decision tree and the MLP neural network classifier maintain their performance. Bayesian networks and the SVM

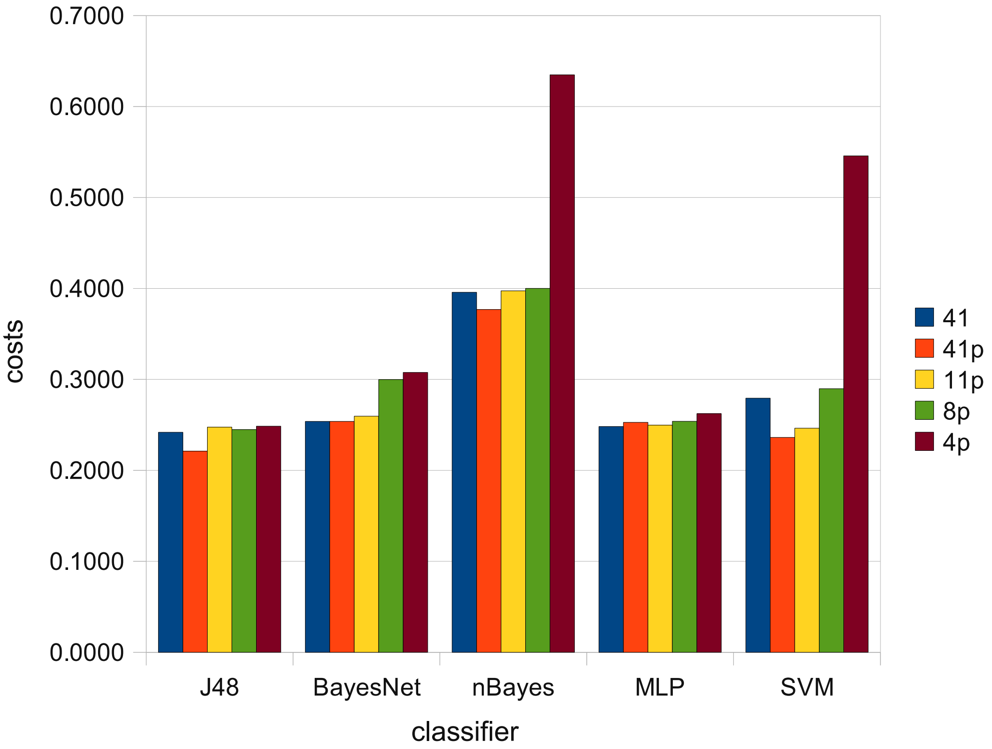


Figure 5.2: *Classifier performance comparison in terms of costs, using the cost matrix provided by the KDD Cup '99, using the four dataset variants with 41, 41p, 11, 8 and 4 features. Only the decision tree classifier can still keep up its performance after feature reduction to 4 minimal features. The MLP neural network classifier shows only a minimal decrease in performance. Analysing the results, we note that the trade-off is a noticeable decrease in performance for the detection of rare 'r2l' and 'u2r' attacks.*

classifier show a decrease in performance. After reducing the feature set to the 4 minimal features only, the decision tree classifier can still keep up its performance, although there is a noticeable decrease in performance for the detection of rare 'r2l' and 'u2r' attacks. In terms of accuracy and costs, however, the results are still on the same level as the KDD Cup '99 winning entry. Figure 5.2 shows a classifier performance comparison in terms of costs, using the cost matrix provided by the KDD Cup '99.

Additionally, Figure 5.3 shows the different numbers of misclassifications by the tested machine learning algorithms applied to six variants of the dataset (41/11 features original/preprocessed and 8/4 features preprocessed). The results show that in terms of misclassifications, the decision tree and the neural

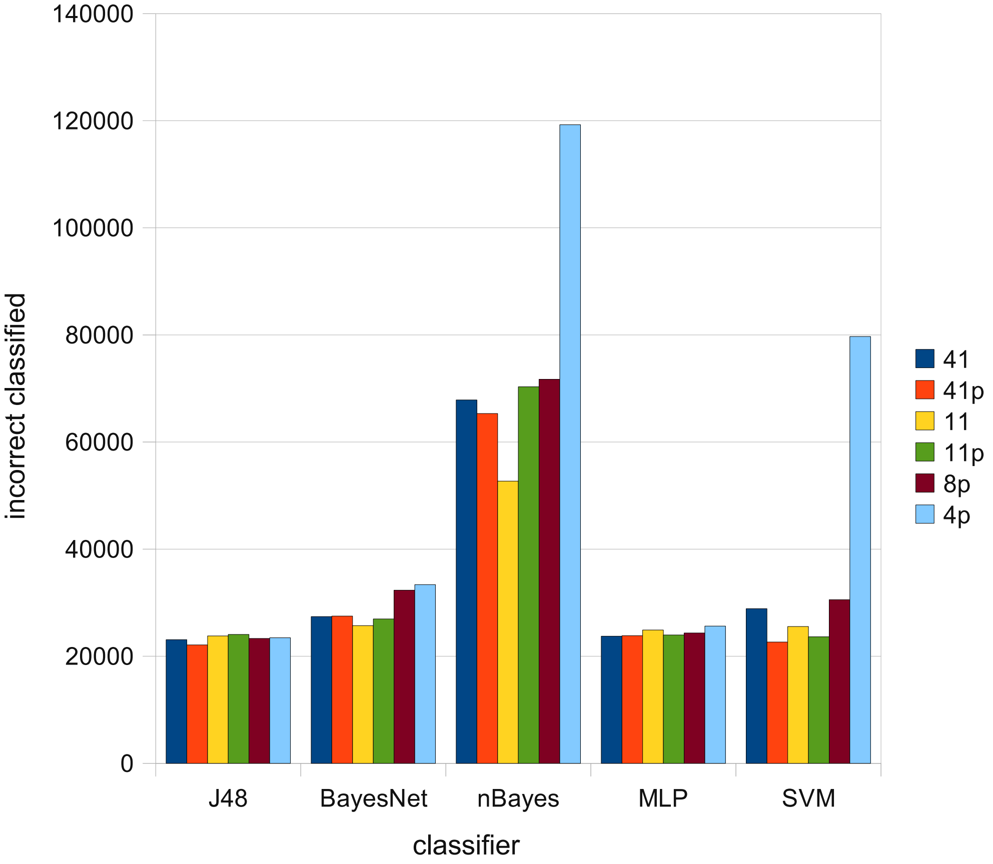


Figure 5.3: *Classifier performance comparison in terms of the total number of incorrectly classified instances (false positives + false negatives), using the four dataset variants with the original 41, 11, 8 and 4 features and preprocessed respectively. This also shows that the performances of the decision tree and the neural network classifier are almost unaffected by excessive feature reduction.*

network classifier show a stable performance almost unaffected by excessive feature reduction.

Table 5.10 summarises the performance of all results trained with the '10%' training set and tested on the original test set. The tables include the KDD Cup '99 challenge winning entry results for convenient comparison with the five classifiers trained with 41 and 11, original and preprocessed respectively, and 8 and 4 features preprocessed. The results for the experiments using all features are summarised in Table 5.6.

We also tested the performance of all classifiers using the full five-million-record training set with preprocessed 41 and 11 features. No observed

Table 5.10: *Performance comparison of the winning results of the KDD Cup '99 challenge with our evaluated classifiers trained with the preprocessed 11, 8 and 4-feature sets. Using the 4-feature set only, the J4.8 decision tree classifier can keep up its performance. In terms of accuracy and costs, the results are on the same level as the KDD Cup '99 winning entry.*

10%	classifier	normal		probe		dos		u2r		r2l			
set		TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	TPR	FPR	ACC	COST
1st	c5	.995	.082	.833	.006	.971	.003	.132	.000	.084	.000	92.71%	.2336
2nd	dforest	.994	.085	.845	.002	.975	.003	.118	.000	.073	.000	92.92%	.2362
9th	1-nn	.996	.091	.750	.002	.973	.005	.035	.000	.006	.000	92.33%	.2530
	pnrule	.995	.089	.730	.001	.970	.001	.066	.000	.107	.001	92.59%	.2387
11p	J4.8	.995	.092	.665	.002	.970	.003	.143	.000	.057	.000	92.26%	.2477
	BayesNet	.988	.089	.804	.010	.957	.003	.471	.002	.053	.002	91.33%	.2596
	nBayes	.895	.076	.720	.133	.792	.116	.100	.001	.085	.003	77.39%	.3974
	MLP	.994	.091	.753	.003	.971	.004	.114	.000	.034	.000	92.29%	.2499
	SVM	.990	.090	.663	.001	.973	.007	.000	.000	.056	.000	92.40%	.2465
8p	J4.8	.995	.088	.776	.002	.971	.007	.257	.000	.055	.000	92.50%	.2450
	BayesNet	.990	.116	.834	.005	.933	.002	.514	.004	.043	.002	89.61%	.2999
	nBayes	.894	.074	.742	.138	.788	.122	.471	.001	.059	.002	76.95%	.4001
	MLP	.995	.092	.648	.001	.973	.012	.000	.000	.000	.000	92.18%	.2541
	SVM	.908	.089	.654	.004	.968	.085	.000	.000	.015	.000	90.17%	.2900
4p	J4.8	.993	.088	.766	.002	.973	.006	.043	.000	.037	.001	92.46%	.2488
	BayesNet	.994	.119	.826	.006	.930	.004	.129	.003	.014	.001	89.27%	.3079
	nBayes	.013	.005	.787	.135	.816	.942	.000	.000	.000	.000	61.66%	.6349
	MLP	.987	.090	.654	.005	.970	.019	.000	.000	.000	.000	91.76%	.2625
	SVM	.129	.065	.623	.001	.961	.775	.000	.000	.000	.000	74.38%	.5460

classifiers improve in either feature sets in terms of accuracy or cost. The total performance is comparable to training with the '10%' dataset.

5.8.2 Individual Attack Classes

In these experiments, we trained the five classifiers with the minimal feature sets for individual attack classes, as described in Section 4.7. We used the preprocessed KDD Cup '99 datasets ('10%' training set and 'corrected' test set).

The best-performing classifier for network probes on the 6 features dataset set, in terms of detection rate, accuracy and cost, is the neural network. The three strong classifiers—decision trees, neural networks and the support vector machines—all maintain or improve in performance with the reduced feature sets. The two statistical classifiers, naïve Bayes and Bayesian networks, both perform better using all 41 features.

We plotted the ROC curves of the neural network classifier of all three tested feature sets. The curves are very similar over wide areas. Calculating the AUC values confirms a comparable performance. The ROC curves of the preprocessed 41, 14 and 6-feature sets are shown in Figure 5.4.

In terms of true positive rate, true negative rate, accuracy and cost, the results for all applied classifiers and all three feature sets are shown in Table 5.11.

For 'dos' attacks, the neural network and the decision tree classifier both perform well with the 5 feature dataset. Both maintain a high classification performance with the 11-feature set and with the minimal feature set in comparison to the full feature set. The SVM classifier significantly decreases in performance with the 11-feature set, but only loses slightly on performance with the 5-feature set. Both statistical classifiers slightly improve in performance with the 11-feature set, but the reduction to 5 minimal features has a strong negative impact.

The ROC curves, with the corresponding AUC values of the neural network classifier, are shown in Figure 5.5. The AUC value reveals a noticeable performance loss in the detection of 'dos' attacks due to the feature reduction.

Table 5.12 shows the results of the full preprocessed 41, 11 and 5-feature

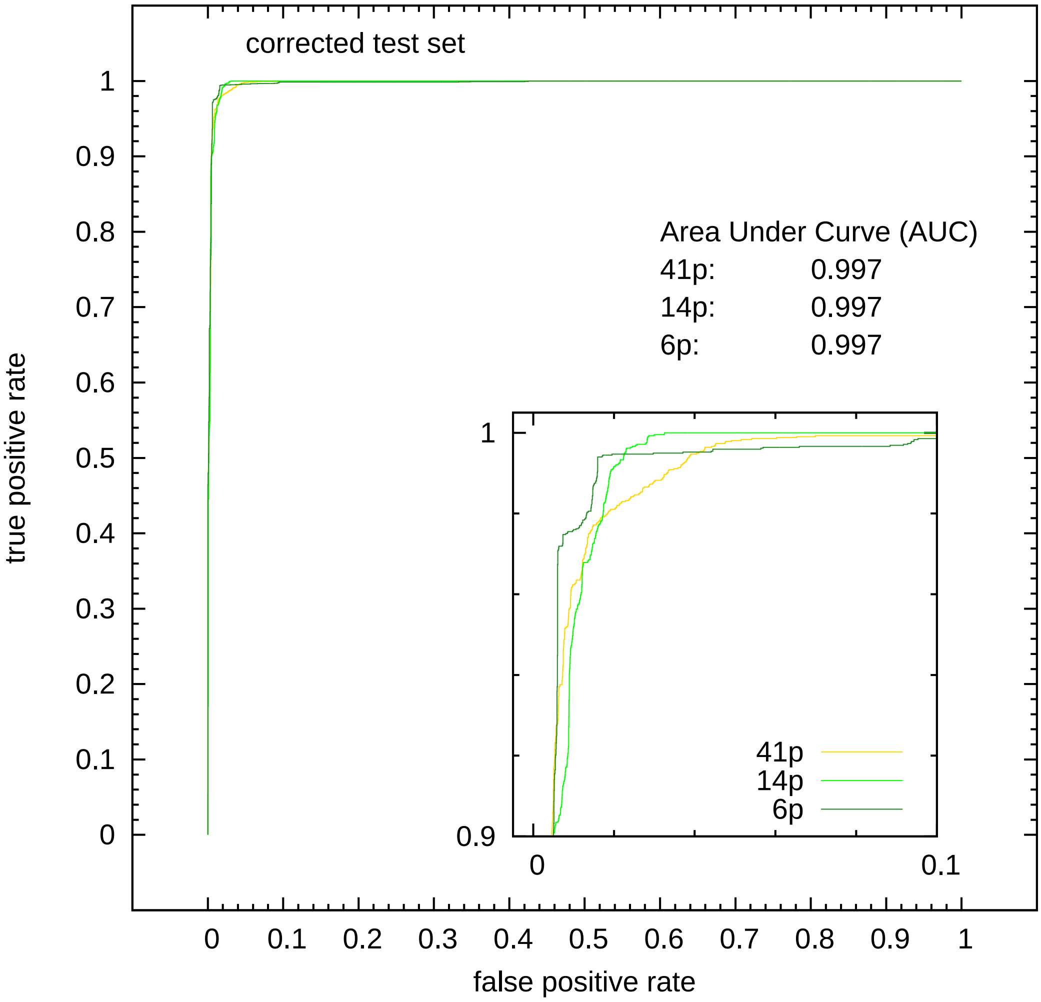


Figure 5.4: *This figure shows the corresponding ROC curves of well-performing neural networks for classifying network probes using the MLP neural network classifier. The similar ROC curves and the equal AUC values show that the classifier can keep its performance after feature reduction.*

sets.

We had great difficulty finding a minimal feature set for the ‘r2l’ attacks class. The performance with the test data suffered for all classifiers on the observed minimal feature sets in the target range of 4–8 features. The performance of the decision tree classifier slightly improves when using the 18 and 14-feature sets. All other classifiers continuously lose performance with the reduction of features. The neural network and the support vector machine classifiers do not detect any attacks on the minimal set.

The ROC curves of the neural network classifier are presented in Figure 5.6.

Table 5.11: *Results for detecting network probes using the preprocessed 39 (39p), 14 (14p) and 6 (6p) features with the '10%' training set and KDD Cup '99 test set. The best-performing classifier in terms of detection rate, accuracy and cost is the MLP neural network trained on the 6 preprocessed features.*

10%	classifier	probe		classifier	
set		TPR	FPR	ACC	COST
39p	J4.8	.779	.005	98.13%	.0187
	BayesNet	.841	.005	98.49%	.0151
	nBayes	.923	.007	98.82%	.0118
	MLP	.844	.004	98.62%	.0138
	SVM	.772	.014	97.23%	.0277
14p	J4.8	.779	.004	98.18%	.0182
	BayesNet	.813	.006	98.22%	.0178
	nBayes	.882	.007	98.63%	.0137
	MLP	.832	.004	98.56%	.0144
	SVM	.743	.004	97.97%	.0203
6p	J4.8	.831	.004	98.52%	.0148
	BayesNet	.722	.004	97.82%	.0218
	nBayes	.860	.009	98.23%	.0177
	MLP	.886	.005	98.81%	.0119
	SVM	.815	.006	98.27%	.0173

They show that the performance in terms of AUC is similar for the 41 and 18-feature set; but significantly drops on the 14-feature set.

The results of our experiments using the preprocessed 41, 18, 14 and 6 features are shown in Table 5.13.

For the very rare 'u2r' attacks, the best-performing classifier is decision trees after feature reduction to the 5-feature set. All classifiers, except the SVM, improve or maintain their performance with the minimal feature datasets. The support vector machine classifier does not detect any attacks using the minimal feature set.

The observation of the ROC curves, using the neural network classifier shown in Figure 5.7, reveals a continuous performance improvement in terms of AUC.

The results of all observed classifiers and feature sets in detecting 'u2r' attacks are summarised in Table 5.14.

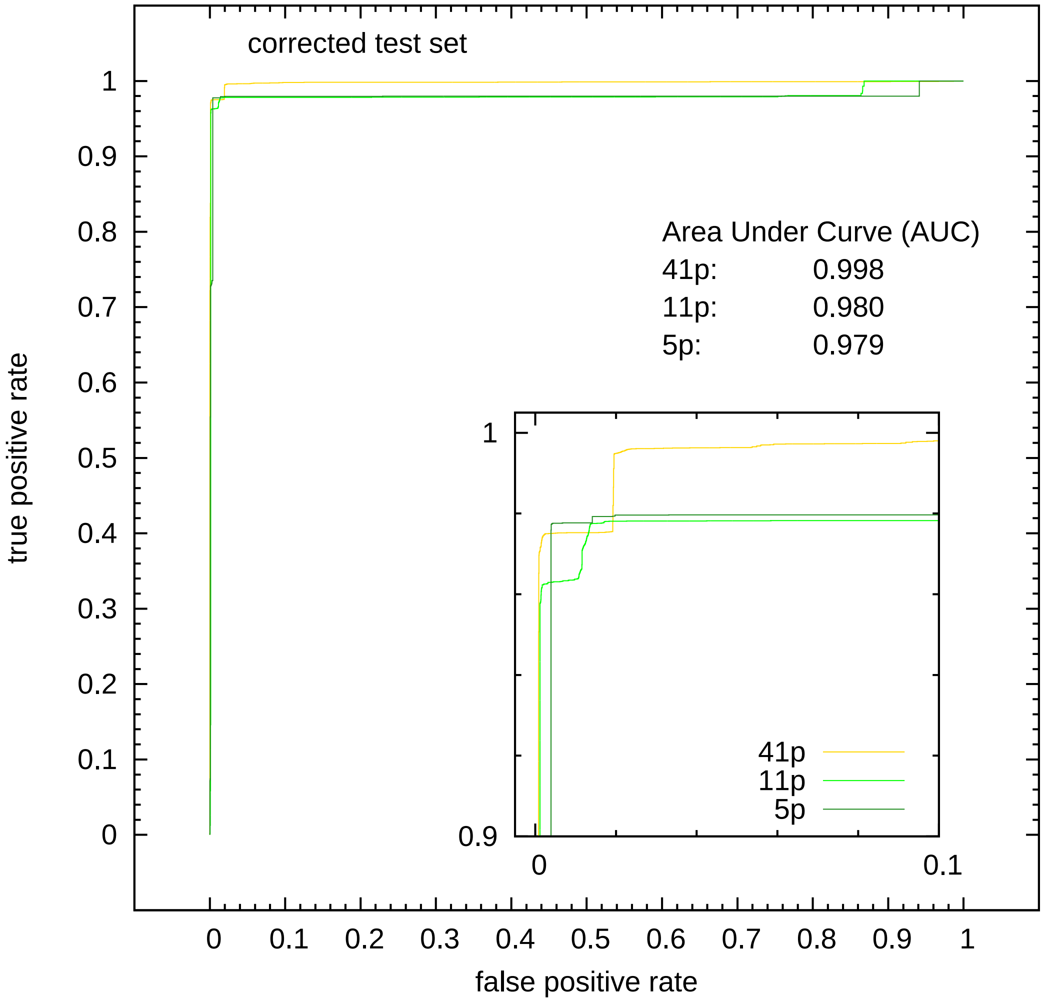


Figure 5.5: *This figure shows the ROC curves of well-performing neural networks for classifying 'dos' attacks using the MLP neural network classifier. Here, the decreasing AUC value reveals that the classifier suffers from feature reduction.*

5.9 Discussion

We applied a variety of machine learning algorithms to the KDD Cup '99 dataset for network intrusion detection. We tested the performance of J4.8 decision trees, naïve Bayes, Bayesian networks, MLP feed forward neural networks, and support vector machines (SVM). One important outcome is that in previous work, not enough effort was made to prepare the data for classification. The KDD Cup '99 datasets contain features with non-changing values and heavily skewed distributions. We added a number of preprocessing steps that significantly improved the classifier performance.

Table 5.12: *Results for detecting 'dos' attacks using the preprocessed 39 (39p), 11 (11p) and 5 (5p) features with '10%' training set and KDD Cup '99 test set. The J4.8 decision tree classifier and the MLP neural network classifier perform very well using the minimal feature set with 5 preprocessed features.*

10%	classifier	dos		classifier	
set		TPR	FPR	ACC	COST
39p	J4.8	.974	.002	97.88%	.0424
	BayesNet	.968	.001	97.43%	.0515
	nBayes	.970	.024	97.15%	.0571
	MLP	.974	.002	97.91%	.0419
	SVM	.973	.002	97.84%	.0433
11p	J4.8	.974	.002	97.88%	.0424
	BayesNet	.970	.002	97.59%	.0482
	nBayes	.971	.022	97.22%	.0556
	MLP	.975	.013	97.73%	.0454
	SVM	.842	.015	87.18%	.2564
5p	J4.8	.977	.003	98.11%	.0377
	BayesNet	.942	.002	95.36%	.0928
	nBayes	.260	.015	41.12%	1.1776
	MLP	.977	.004	98.12%	.0375
	SVM	.971	.011	97.48%	.0503

As expected, the naïve Bayes classifier is not well-suited to this learning task. It shows poor performance for all traffic types. We learned that Bayesian networks show strengths in the classification of network probes but suffer from high false alarm rates in general.

J4.8 decision trees, MLP neural networks and the SVM classifier show acceptable performance for this type of dataset. The decision tree classifier shows strengths in the detection of rare 'r2l' and 'u2r' attacks. The slight decrease in detection of 'dos' attacks and network probes, which in some cases comes with feature reduction, is not significant. Due to the number of connections initiated in series by these attacks, a detection rate of 80% is still acceptable.

In the 11 feature dataset for multi-class categorisation, the first 6 features are *basic* (or base) features that can be easily extracted from network traffic with very little overhead. The remaining features are *connection-based* (*time-based* and *host-based*) traffic features. We were able to dismiss all *content-based* features that are much more complex to extract. This is also true

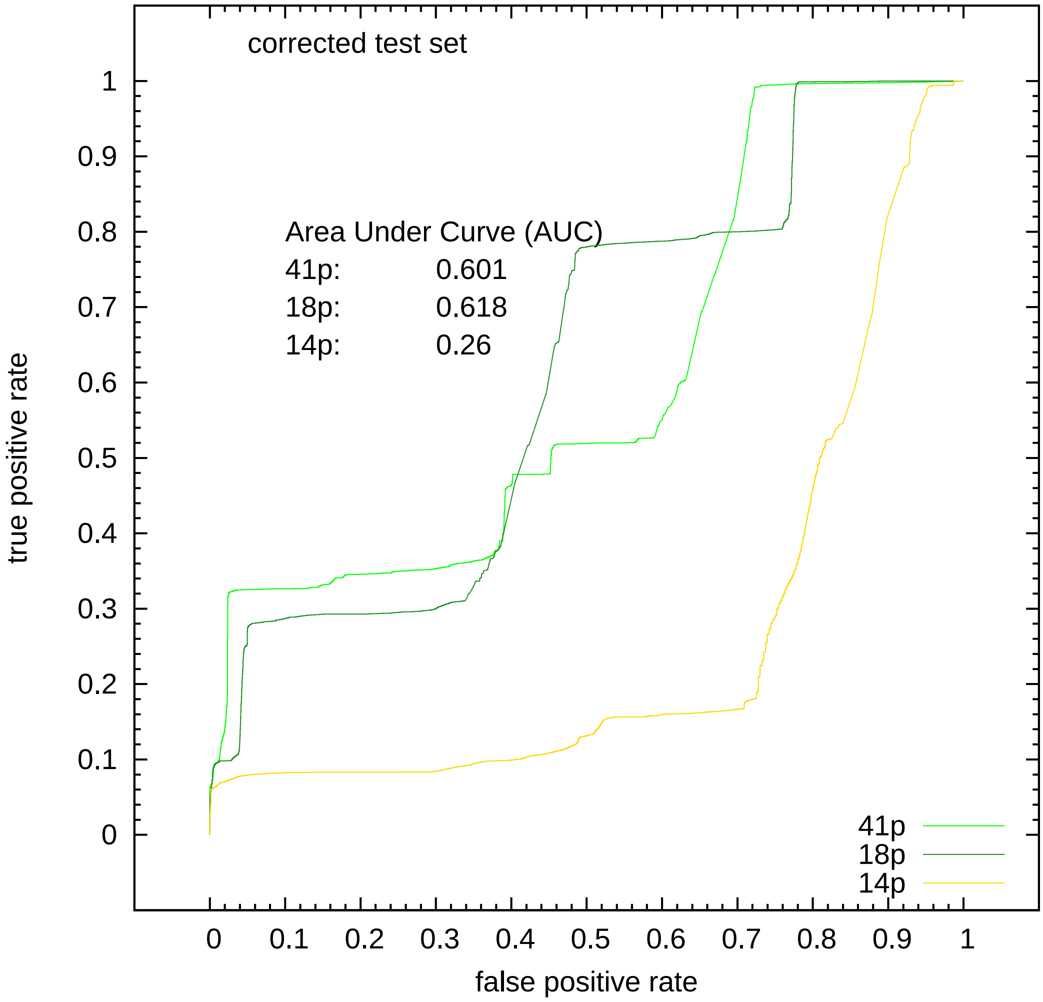


Figure 5.6: *This figure shows the ROC curves for classifying 'r2l' attacks with well-performing networks using the MLP neural network classifier. AUC performance using the 41 and 18-feature set is similar, but drops using the 14-feature set.*

for the 4-feature set. For the 8-feature set, we picked the two *content-based* features 'hot' and 'num_failed_logins' that proved to be valuable for the detection of the rare attack classes using the decision tree classifier. In terms of misclassifications, the decision tree and the neural network classifiers were able to keep their performance, though at the expense of the reduced detection of rare attacks.

With the individual attack classes using two-class categorisation, the observation of information gain per class revealed some interesting facts. For the three network-related attacks 'probe', 'dos' and 'r2l', the most relevant

Table 5.13: *Results for detecting 'r2l' attacks using the preprocessed 39 (39p), 18, 14 and 6 features with '10%' training set and KDD Cup '99 test set. No classifier benefits from using the minimal feature sets in the aimed range of 4–8 features. Only the performance of the decision tree classifier slightly improves when using the 18 and 14-feature sets.*

10%	classifier	r2l		classifier	
set		TPR	FPR	ACC	COST
39p	J4.8	.088	.000	80.61%	.5815
	BayesNet	.104	.004	80.70%	.5764
	nBayes	.127	.010	80.69%	.5716
	MLP	.046	.000	79.72%	.6084
	SVM	.092	.000	80.66%	.5797
18p	J4.8	.088	.000	80.62%	.5813
	BayesNet	.063	.001	80.00%	.5991
	nBayes	.072	.007	79.72%	.6027
	MLP	.037	.000	79.53%	.6141
	SVM	.056	.001	79.90%	.6026
14p	J4.8	.088	.000	80.62%	.5813
	BayesNet	.065	.001	80.03%	.5980
	nBayes	.054	.007	79.35%	.6140
	MLP	.029	.000	79.34%	.6196
	SVM	.034	.001	79.42%	.6168
6p	J4.8	.033	.000	79.45%	.6164
	BayesNet	.055	.002	79.76%	.6057
	nBayes	.081	.008	79.87%	.5979
	MLP	.000	.000	78.74%	.6377
	SVM	.000	.000	78.72%	.6380

features were basic and connection-based features, with the content features 'logged_in' and 'hot' being the only exceptions. The correlations for 'r2l' attacks are mostly weak. The 'u2r' attack class is the only attack traffic class that very weakly correlates with some content-based features. This shows that for the detection of local attacks, the dataset is not suitable.

For network probes and 'dos' attacks, we achieved a comparable performance in terms of accuracy and costs after feature reduction, with the decision tree, neural network and support vector machine classifier. Observing the ROC curves for the neural network classifier reveals a noticeable, but not significant, performance loss.

Feature reduction with the two rare attack classes 'r2l' and 'u2r' caused great difficulty. Overall, we were not able to significantly improve the

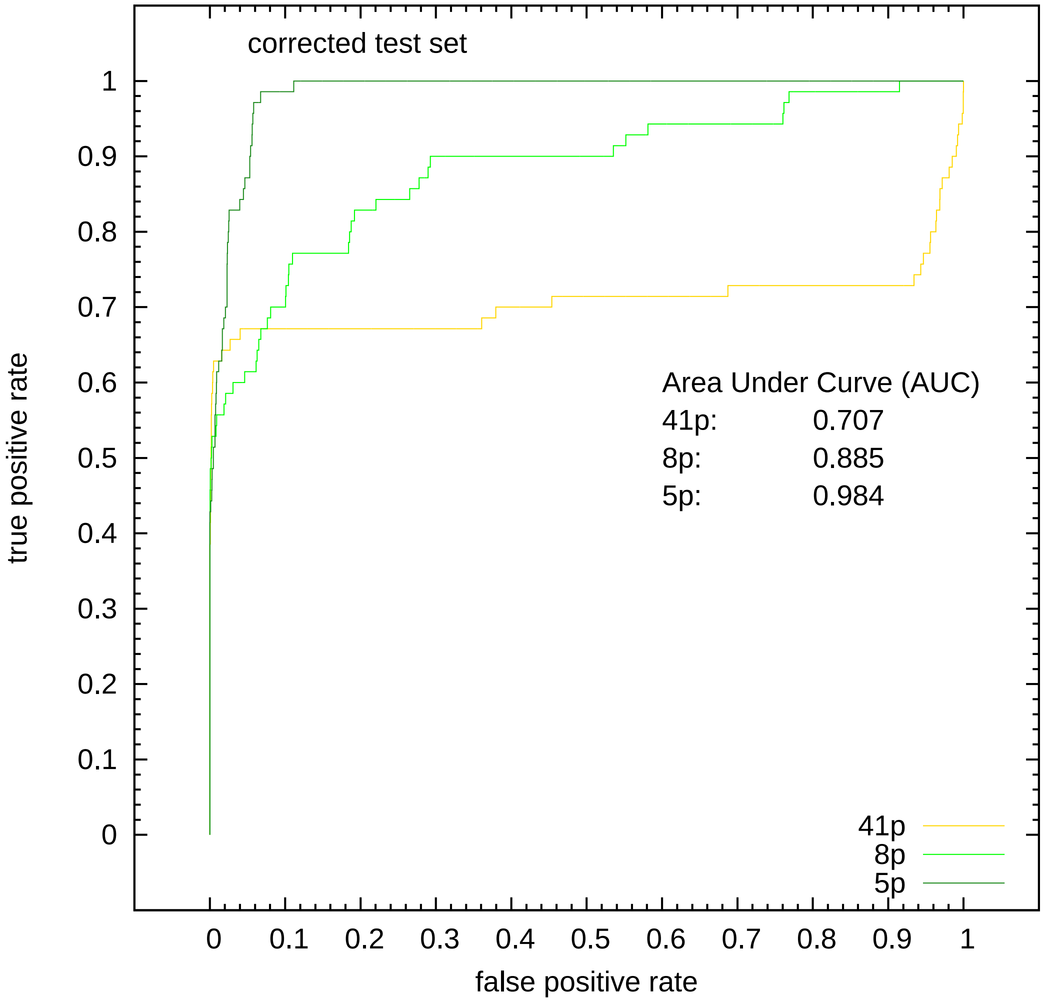


Figure 5.7: *This figure shows the ROC curves of well-performing neural networks for classifying 'u2r' attacks, using the MLP neural network classifier. In terms of AUC performance, there is a significant performance improvement after feature reduction.*

detection of these two attack classes, but at least we could maintain the performance for most classifiers. For the 'r2l' attacks, only the decision tree classifier shows an acceptable performance with the corresponding minimal feature set. The neural network and the support vector machine requires at least our proposed 14-feature set to classify any attacks correctly. For 'u2r' attacks, the decision tree classifier shows at least a noticeable performance improvement in terms of accuracy and cost with the corresponding minimal feature set. Unfortunately, the SVM classifier is not able to classify any attacks on the reduced feature sets.

Table 5.14: *Results for detecting 'u2r' attacks using the preprocessed 39 (39p), 8 (8p) and 5 (5p) features with '10%' training set and KDD Cup '99 test set. Nearly all classifiers improve or maintain their performance on the reduced feature datasets (exception: SVM).*

10%	classifier	u2r		classifier	
set		TPR	FPR	ACC	COST
39p	J4.8	.457	.000	99.92%	.0028
	BayesNet	.671	.001	99.86%	.0035
	nBayes	.843	.013	98.67%	.0270
	MLP	.343	.000	99.91%	.0033
	SVM	.357	.000	99.91%	.0032
8p	J4.8	.371	.000	99.91%	.0032
	BayesNet	.586	.000	99.92%	.0026
	nBayes	.600	.004	99.58%	.0092
	MLP	.371	.000	99.92%	.0030
	SVM	.371	.000	99.91%	.0032
5p	J4.8	.643	.000	99.96%	.0017
	BayesNet	.371	.000	99.92%	.0030
	nBayes	.543	.001	99.81%	.0049
	MLP	.314	.000	99.92%	.0032
	SVM	.000	.000	99.88%	.0046

Further research might reveal that some of these remaining traffic features are also dismissible using machine learning algorithms, which are able to extract time series information.

5.10 Conclusions

In this chapter, we outlined the shortcomings of the DARPA / KDD Cup '99 datasets. We noted previously published results, and presented a performance comparison of five selected static classifiers, using all available features. The investigated classifiers are J4.8 decision trees, naïve Bayes, Bayesian networks, multilayer perceptron artificial neural network (MLP), and support vector machines (SVM).

The results show significant performance improvements after preprocessing and data preparation for most of the investigated classifiers. The only exception is the Bayesian network, which shows no change in performance. The gain in performance of the decision tree classifier is impressive. Our results supersede the results of the winning entries of the KDD Cup '99.

Experiments observing feature sets using our custom-built, 10,422-instance training set showed that the basic features and the traffic features are of salient importance in terms of accuracy. Our 11-feature set further enhances the performance of the decision tree classifier.

In the experiments, training with the ‘10%’ set, we compared our preprocessed minimal feature sets with 4, 8 and 11 features to the winning entries of the KDD Cup ’99 challenge. The results show a massive reduction to a small number of salient features where most classifiers perform very well. Observing our small sets with preprocessed 4–8 minimal features for individual attacks reveals similar results.

In the next chapter, we further improve the classification performance by using a dynamic classifier that is able to extract time series information. We apply long short-term memory recurrent neural networks (LSTM) to our preprocessed KDD Cup ’99 data and compare it to the performance of the static classifiers.

Chapter 6

Modelling IDS as a Time Series

Contents

6.1 Introduction

In Chapter 5, we evaluated the performance of static classifiers. In this chapter, we model intrusion detection as a time series. Here, we present the results of our experiments on the KDD Cup '99 dataset, using the LSTM recurrent neural network classifier.

We start by outlining the experimental set-up and some notes on LSTM code parallelisation. Then we give an overview of all experiments performed.

In the following sections, we analyse the results of all our experiments using the LSTM classifier. We compare LSTM performance using all features and the minimal feature sets. We analyse the performance of LSTM when training one network for all traffic classes, and when training individual networks for each traffic class. We conclude this chapter with a performance comparison of all classifiers we evaluated with the KDD Cup '99 datasets.

6.2 Experiment Design

We experimented with different parameters and structures of an LSTM recurrent neural network, such as the number of memory blocks and the cells per memory block, the learning rate, and the number of passes through the data. We also ran experiments with a layer of hidden neurons, with peephole connections, with forget gates, and with LSTM shortcut connections.

6.2.1 Experimental Parameters

As a starting point for finding suitable network parameters and structure, we used a basic LSTM network using 43 input neurons and 5 target neurons. The hidden layer consisted of two LSTM memory blocks, with two cells each and peephole connections. The input neurons were fully connected to the hidden layer with the two memory blocks. We did not use shortcut connections.

We applied our customised and preprocessed version of the KDD Cup '99 datasets, using all input features. The '10%' training dataset was used for training, and the '10% corrected' dataset was used for testing. The number of iterations for training was fixed at 50 epochs for these experiments.

Each of the five target neurons corresponds to one of the five network traffic types. We tested the target values in the order 'normal', 'dos', 'probe', 'u2r' and 'r2l'. The traffic at this time was simply classified according to the first value larger than the decision threshold. The decision threshold was fixed to 0.5. We classified cases where no output was larger than the decision threshold per default as 'normal'.

We evaluated the performance of the learned networks by manual observation of the confusion matrix and by calculating the accuracy.

In our first experiments, we focussed on optimising the learning rate and evaluating the impact of adding a layer with hidden neurons to the network. We also experimented with LSTM-specific features.

We started by finding a suitable learning rate for our datasets. The learning rate was varied in the interval [0.01–0.5], and we did not use any weight decay.

The experiments with lower learning rates showed a slightly better classification performance. Naturally, for a low learning rate (0.01), the expected number of required iterations for low frequency attacks was very large (>10,000 epochs). As a trade-off between training time and classification performance, we decided to set the learning rate to not lower than 0.1 for following experiments. The best detection rate result for each experiment is shown in Figure 6.1. We can conclude from these results that 50 epochs are not sufficient to train 'r2l' and 'u2r' attacks using any learning rate; but for the traffic classes 'normal', 'dos' and 'probe', a learning rate of around 0.1 already provides good results.

In a second step, we experimented with different 'pure' feed-forward networks and hybrid networks, including hidden neurons and LSTM memory blocks. The number of hidden layers was fixed to one in all experiments. We did not expect improvements with more than one hidden layer. For both types of networks, we ran experiments with 5, 10, 15, 20, 32, 43 and 86 hidden neurons. Each experiment consisted of eight trials.

All LSTM hybrid networks showed good performance in terms of accuracy. Learning of the hybrid networks was also faster. Best results were achieved using one feed-forward layer with 20 hidden neurons. All of these trained networks achieved 'good' results with an accuracy value of >90%. Unfortunately, 'best' results are slightly less accurate than well-performing results using standard LSTM. The generalisation performance of the hybrid network seems to be weakened in comparison to an LSTM network without a hidden layer. From this we conclude that adding hidden neurons makes LSTM more prone to over-fitting. Furthermore, we noted that the detection rate on rare and 'difficult-to-learn' 'r2l' and 'u2r' attacks decreased.

Finally, we consecutively added peephole connections and shortcuts to the basic LSTM network to assess their impact on learning performance.

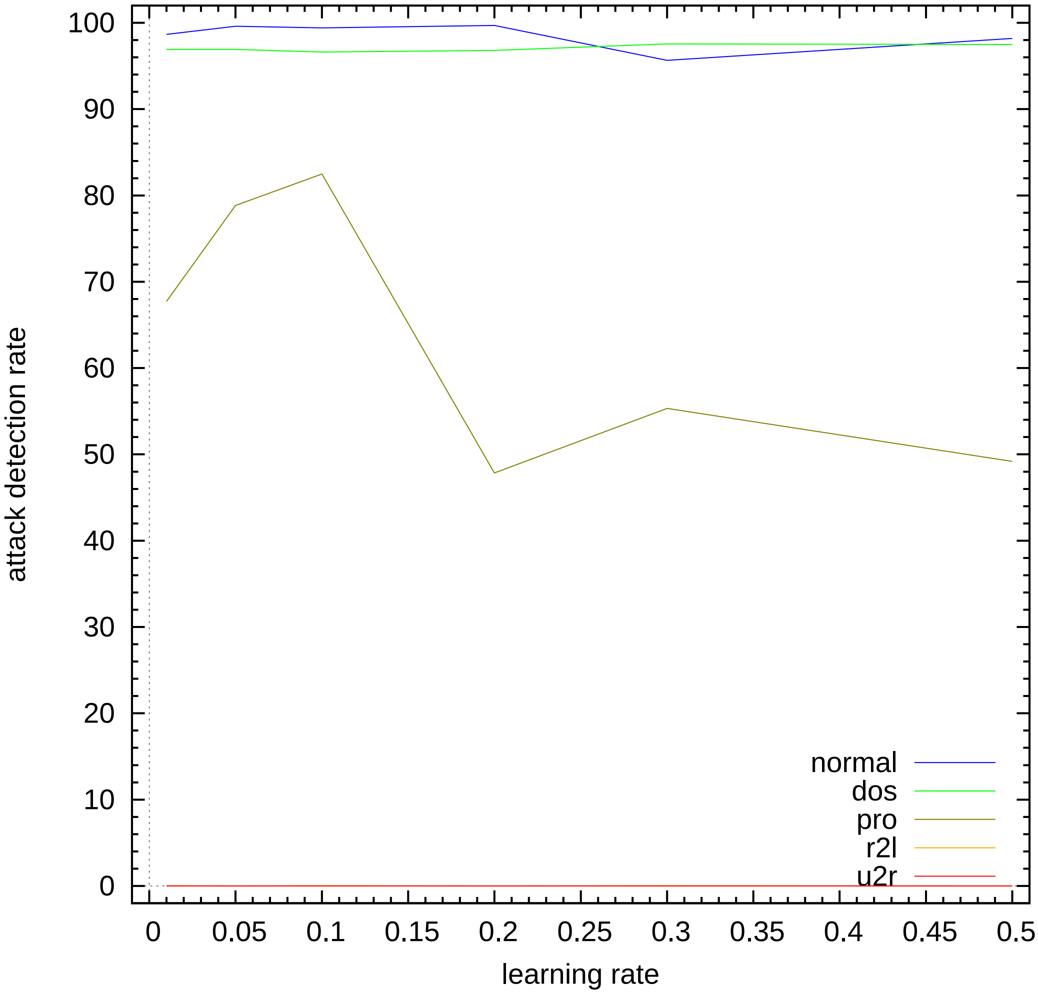


Figure 6.1: *The detection rate results for all traffic types using well-performing LSTM recurrent neural networks, containing two memory blocks with two cells each. We trained the networks for 50 epochs, with different learning rates, in the interval between [0.01–0.5]. From the results, we can conclude that we can already get good results classifying the traffic classes 'normal', 'dos' and 'probe' when using a learning rate of around 0.1.*

Adding peephole and shortcut connections did not yield any significant performance improvements either, but using shortcuts slightly improved the average classification performance. We activated shortcuts for all further experiments. We used peephole connections only in some experiments on the datasets using all features.

6.2.2 Network Topology

To find a suitable network structure for training the KDD Cup '99 data, we experimented with LSTM networks using four different topologies: Two memory blocks with two cells each; four memory blocks with two cells each; four memory blocks with four cells each; and eight memory blocks with four cells each.

All networks used forget gates, peephole connections and shortcuts. The learning rate was fixed at 0.1, and the decision threshold was set to 0.5. Traffic classification was according to the first value larger than the threshold in the order 'normal', 'dos', 'probe', 'r2l' and 'u2r'. Default classification was 'normal'. We used the preprocessed '10% training' and the '10% corrected' test set datasets with all features.

To find the minimum number of required iterations, we presented the training data for 5–1000 epochs to each of the four observed network structures. We ran eight trials of each network setup.

Results with good accuracy for attack detection (attack/normal two-class categorisation), at reasonable cost in terms of run-time, was reached at 60–150 epochs. The lowest standard error was reached after little more than 500 epochs for all four network topologies. More complex LSTM networks needed more iterations to get acceptable results, but finally also attained a higher accuracy.

For each of the five attack classes, the LSTM network requires a different number of 'optimal' iterations. After 25–90 epochs, most networks learned 'dos' attacks. The detection rate peaks initially at 125 epochs, and then again at about 500 epochs. Network probes are mostly learned after 50–125 epochs and also peak at about 500 epochs. The rare attack categories, 'r2l' and 'u2r', need many more presentations of the training data. Attacks of the class 'r2l' need 200–1,000 epochs, and ‘u2r’ attacks need 125–1,000 epochs before they are learned as well as possible. We observed that some networks still improve very slowly on rare attack types after 1,000 epochs. Some of the rare attacks require more than 1,000 presentations. For us, it remains questionable if they can be learned at all using the available training data. After 500–600 epochs, approximately 50% of the trials achieve results with a low error rate. The total performance of all networks decreases only after further training.

In comparison to standard neural networks with a hidden layer, LSTM is much more prone to over-fitting. After learning a specific traffic type, the network improves in memorising the learned traffic types in the training data, and the generalisation performance for these continuously decreases. Naturally, the detection performance of trained networks on the test set quickly degrades after reaching peak performance for the most frequent traffic type, which are, in this case, ‘dos’ attacks.

Out of the eight trials we ran on the four different network structures, we picked three well-performing networks for further observation. Accuracy for normal/attack two-class categorisation was used as performance measurement method.

With increasing size of the LSTM network, learning requires more presentations of the training data. Rare ‘r2l’ attacks require more than 1,000 epochs on the two larger networks. On the other hand, the small LSTM network with two memory blocks has problems learning the very rare and difficult-to-learn ‘u2r’ attacks.

We think that networks with four memory blocks containing two cells each offer a good compromise between calculation cost and detection performance. We used this type of network structure for all our experiments. Figure 6.2 shows a performance comparison of the best results of the four different network types evaluated.

6.2.3 Parallelisation

Due to the huge amount of expected processing time for the LSTM experiments, we decided to improve our LSTM implementation by parallelising parts of the code. Neural networks offer great potential for

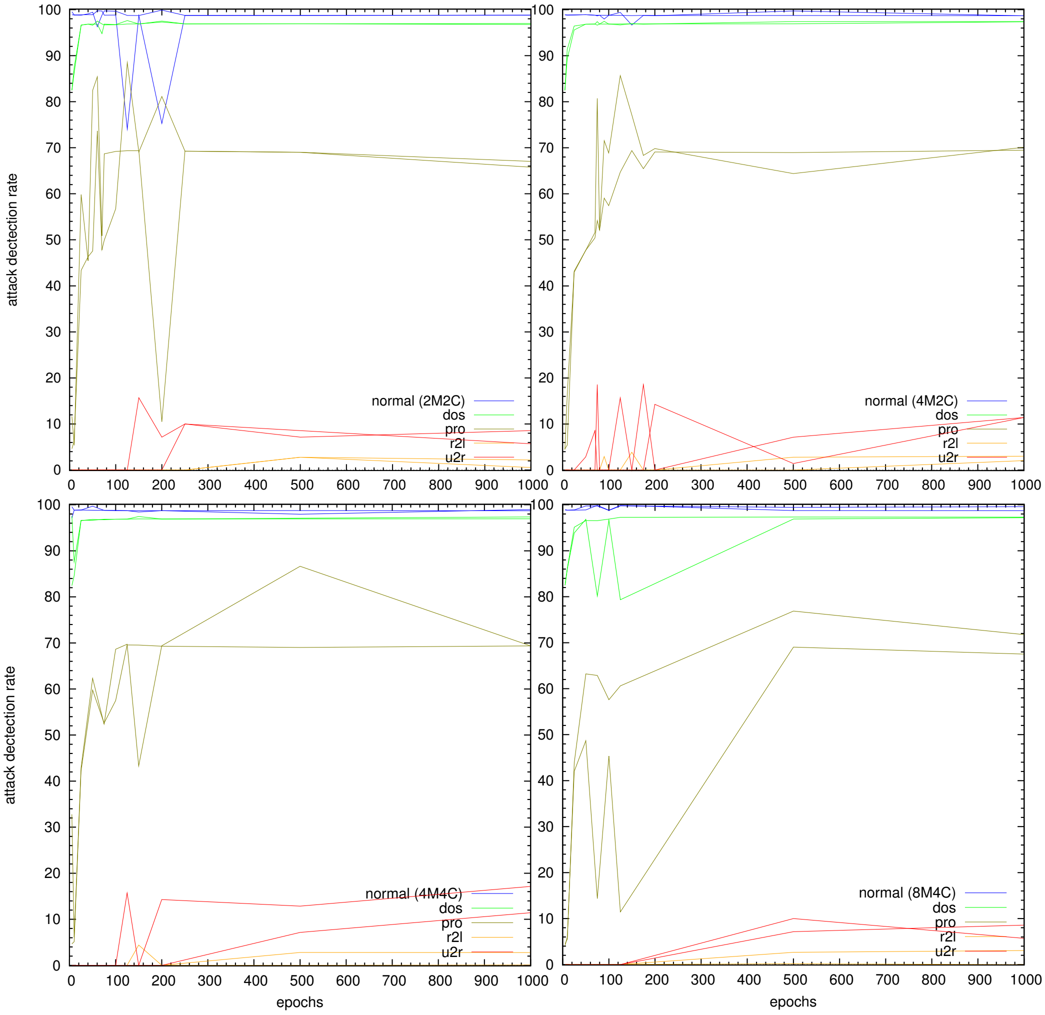


Figure 6.2: *This figure shows a performance comparison of the four different LSTM network structures, with 2, 4 and 8 memory blocks, containing 2 and 4 memory cells respectively (2M2C, 4M2C, 4M4C and 8M4C). Detection rate performance was measured for each traffic class according to the number of epochs trained. Neural networks with four memory blocks containing two cells per block show a good compromise between calculation cost and detection performance.*

parallelisation. To keep things simple, we decided to make use of the fact that neural networks require multiple runs. Each run starts with different random start values, followed by a manual selection of the 'best' result.

6.2.3.1 Thread Parallelisation Using OpenMP

Most of today's workstation computers contain at least one processor with two or more CPU cores. All CPU cores share main memory. A program that is processed in sequential order can make use of only one processing core. Most programs contain sections that could be executed in parallel. OpenMP provides an intuitive way of manually marking these parallel sections for the compiler to generate parallel executable code for these sections.

A program marked with OpenMP directives contains one 'master' thread that requires consecutive execution. This master thread can call a number of 'slave' threads when it comes to a parallel section of the program. 'Slave' threads can run concurrently on different processing cores. When the execution of the parallel code section finishes, the 'slave' threads merge back into the one 'master' thread.

In our implementation, every trial runs in its own thread, having its own private copies of global variables and also private pointers to the records of the datasets. Running the trials as separate programs would require separate copies of the datasets for each trial. We achieved major performance improvements using OpenMP parallelisation.

6.2.3.2 Compiler Optimisation

Today's C++ compiler suites offer a variety of powerful optimisation features. Compiler options strongly depend on the architecture where optimisation is executed and also on the code to be optimised. We conducted a number of experiments on two machines:

- HP Prolient 365DL, with an AMD Opteron 2222@3 GHz, 2 Processors (4 Cores), 32 GB Memory, 64-bit Debian GNU/Linux

- Custom-built Workstation, with Intel Core2 6600@2.40 GHz, 1 Processor (2 Cores), 3 GB Memory, 32-bit Debian GNU/Linux

We used the GNU Compiler Collection (gcc 4.2.3) and Intel's C++ Compiler (icc 10.1 20080112).

We built an LSTM network with 43 input neurons, 4 memory blocks, 4 memory cells per block, 4 hidden neurons, and 5 output neurons, running 4 trials with 10 epochs each.

We used the KDD Cup '99 '10%' dataset containing 494,021 connection records as a training dataset. The '10% corrected' dataset was used as a test set. These datasets contained all features available. Attacks were mapped to the five traffic types ('normal', 'probe', 'dos', 'r2l', 'u2r'). All features were preprocessed, as described in Section 4.5.1.

To measure the performance, we used the GNU/Linux *time* command. We noted the total number of CPU-seconds that the process spent on a single CPU in user mode, and calculated to maximum speed-up using all processor cores.

The ICC compiler already optimises execution performance by default, without any options given; GCC only optimises if requested.

We got the best CPU performance per core on the Intel Core2 CPU using the Intel ICC compiler with the options *-O3 -us -funroll-loops -ip -ipo -xT -parallel*. Using the GCC compiler, we got the best performance on the AMD Opteron CPU using the options *-O3 -mtune -opteron -march -opteron -m64 -lm -finline-functions -ffast-math -profile-use -foptimize-register-move -fprefetch-loop-arrays -funroll-loops -static -ftree-vectorize*. In both set-ups, after parameter-tuning the performance, the Intel compiler exceeded the performance of the GCC compiler by approximately 20%.

Surprisingly, the thread distribution of OpenMP does not have a negative impact on the single core performance. The run-time performance improvement of LSTM code compiled with the ICC compiler was approximately 50% in comparison with code compiled with the GCC compiler. We finally used the ICC compiler on the AMD Opteron architecture, due to the larger number of available CPU cores and memory on the system.

Results of the run-time performance comparison are shown in Table 6.1. It shows that in comparison to the default compiler settings, we achieved a 4 to 8 times performance improvement using OpenMP directives, profiling and other compiler-specific optimisations.

Table 6.1: *Run-time performance comparison using automatically (-O1-3) and manually selected compiler options on AMD Opteron architecture using 64-bit GNU/Linux. In comparison to the default compiler settings, we achieved a 4 to 8 times performance improvement using OpenMP directives, profiling and other compiler specific optimisations.*

Dual-Core AMD Opteron 3 GHz, 2 CPU, 32 GB Mem. 64-bit Debian GNU/Linux					
compiler	options	OpenMP/ profiling	user-mode CPU in s (single-core)	max. cores	user-mode CPU in s (multi-core)
gcc-4.2.3	none	O/0	895.03	1	895.03
	-O1	O/O	720.57	1	720.57
	-O2	O/O	704.01	1	704.01
	-O3	O/O	722.71	1	722.71
	-O3	X/O	718.81	4	179.70
	custom*	X/X	473.98	4	118.50
icc 10.1	none	O/O	387.47	1	387.47
	-O1	O/O	572.32	1	572.32
	-O2	O/O	386.89	1	386.89
	-O3	O/O	387.22	1	387.22
	-O3	X/O	406.41	4	101.60
	custom**	X/X	381.12	4	95.28

*-O3 -mtune=opteron -mcpu=opteron -m64 -lm -finline-functions -ffast-math
-profile-use -foptimize-register-move -fprefetch-loop-arrays
-funroll-loops -static -ftree-vectorize
**-O3 -xW -ipo -ip -static -us -p -funroll-loops -parallel -prof_use

6.3 Experiments

In all our experiments, we built LSTM networks with four memory blocks, containing two memory cells each. We used forget gates and shortcut connections, and we also experimented with peephole connections. The applied learning rates were 0.5 or 0.1. In some experiments, we applied an exponential learning decay of 0.99 or 0.999. We limited learning to a maximum of 1,000 epochs and 30 trials. For training, we used the preprocessed ‘10%’ training dataset. We tested the performance on the training set as well as on the ‘10% corrected’ test set. The preprocessed datasets are described in Section 4.5.1.

We presented the training data to the LSTM networks in a continuous input stream. For every presented traffic type, we ran a forward pass, a backward pass and updated all the weights. We reset the network state at the beginning of each presentation of the training dataset. It was the task of the network to learn to predict the correct traffic class. We used the target output with the highest numerical value for traffic classification.

For every performance test on a dataset, we generated a 5×5 confusion matrix and calculated the accuracy and the mean squared error. By evaluating the confusion matrix, we calculated precision and detection rate for each target neuron representing one of the five traffic classes. Additionally, we calculated the AUC value for each target neuron. We generated the ROC curves directly from the parametric points without using any non-parametric estimation. We got the parametric points by varying a threshold over the output of the target neuron.

For the ROC calculations, we used the library version of the *proproc.2.8.0* software [Pesce & Metz 2007] and [Metz *et al.* 2009]. The *proproc* software and excellent support was kindly provided to us by the developers.

We trained LSTM networks and ran performance tests for (1) all features for all attacks in a single network, (2) all features for all attacks in different networks, (3) minimal features for all attacks in a single network, and (4) minimal features for individual attacks in individual networks.

An overview of all experiments is summarised in Table 6.2.

Table 6.2: *Overview of all performed experiments. We ran 30 trials of every experiment. All LSTM networks were trained for up to 1,000 Epochs.*

features	attacks	learning rate	decay
all	all	0.1	-
all	all	0.1	0.999
11	all	0.5	0.990
8	all	0.1	0.999
4	all	0.1	0.999
5	dos	0.5	0.990
6	probe	0.5	0.990
5	u2r	0.5	0.990
8	u2r	0.5	0.990
6	r2l	0.5	0.990
14	r2l	0.5	0.990
14	r2l	0.1	0.999

6.4 Performance Analysis Using All Features

We ran two experiments, using all features, for training LSTM recurrent neural networks with all attack classes. The aim of the first experiment was twofold:

On the one hand, we plotted ROC curves to confirm that LSTM is actually able to correctly classify all five traffic classes in the training data. On the other hand, we estimated the minimum required training epochs for each traffic class.

We trained networks for 25, 50, 75, 90, 100, 125, 150, 175, 200, 250, 300, 400, 500, 600, 750 and 1,000 epochs, with 30 trials each, adding up to a total of 30×16=480 experiments. We built LSTM networks with peephole connections and a fixed learning rate of 0.1.

We tested the performance of the trained neural network at the end of each trial. For every test, we calculated the AUC values for every target neuron and saved the sequence of empirical operating points to plot the ROC curves.

We used all features of the preprocessed datasets, except two features with non-changing values in the training data. This left us with 39 input features, which we mapped to an input layer of size 40. We mapped the target feature, categorising one of the five traffic classes, to five target neurons.

In the second experiment, we focused on training well-performing networks. Here, we directly calculated the AUC values, without saving the parametric points for the ROC curves. This dramatically reduced the amount of collected data. We kept the previously selected network architecture and parameters, but removed the peephole connections because they showed no benefit.

Furthermore, we added an exponential decay to the learning rate. We initialised the learning rate to either 0.5 or 0.1, and set an exponential decay of either 0.99 or 0.999. We multiplied the decay by the learning rate after every presentation of the training data.

Generally, training was stopped after 1,000 epochs, or after 300 epochs if this proved to be sufficient.

6.4.1 Multi-Class Categorisation

We first analysed the results of the first experiment to find the best-performing LSTM network trained using all features and attack types. We observed all the trained LSTM networks in terms of ROC performance. Since multi-class ROC graphs are not able to be plotted, we examined the five target neurons of every trained LSTM network separately. For each neuron, we generated a separate ROC graph and calculated the corresponding AUC value, representing one of the five traffic classes: 'normal', 'dos', 'probe', 'r2l' and 'u2r'.

For the majority of trained networks, we found that the target neurons representing normal traffic and network probes showed an excellent ROC performance. The 10 highest AUC values achieved for normal traffic results were in the range between [0.9665–0.9716], and for network probes, in the range between [0.9679–0.9826]. The corresponding ROC curves are shown in Figures 6.3 and 6.4.

The target neuron representing 'dos' attacks achieves a close to perfect discrimination between 'dos' attacks and other traffic classes (perfect = AUC value equal to 1.0) in well-performing networks. Here, the 10 highest AUC values are in the range between [0.9950–0.9971]. The corresponding ROC curves are shown in Figure 6.5.

For 'u2r' attacks, we still achieved a good ROC performance. The 10 networks with the highest AUC values have a range of [0.8766–0.8909]. 'r2l' attacks proved to be the most difficult to classify in the test set. The 10 highest AUC values, in the range between [0.5380–0.5627], all show a poor performance. We also note that only 60% of all trained networks showed any classification performance better than random guessing. The Figures 6.6 and 6.7 show the corresponding ROC plots.

A derivation of the curves from a straight diagonal between the coordinates [0,0] and [1,1] to a curve bowing into the corner [1,0] of the graph is recognisable in all graphs. This shows that the trained networks were actually able to learn at least parts of all five traffic classes.

The partially bumpy ROC curves are expected, since we present whole classes of traffic where every class contains various subclasses of traffic. During the learning process, distinguishable subclasses do appear as bumps in the

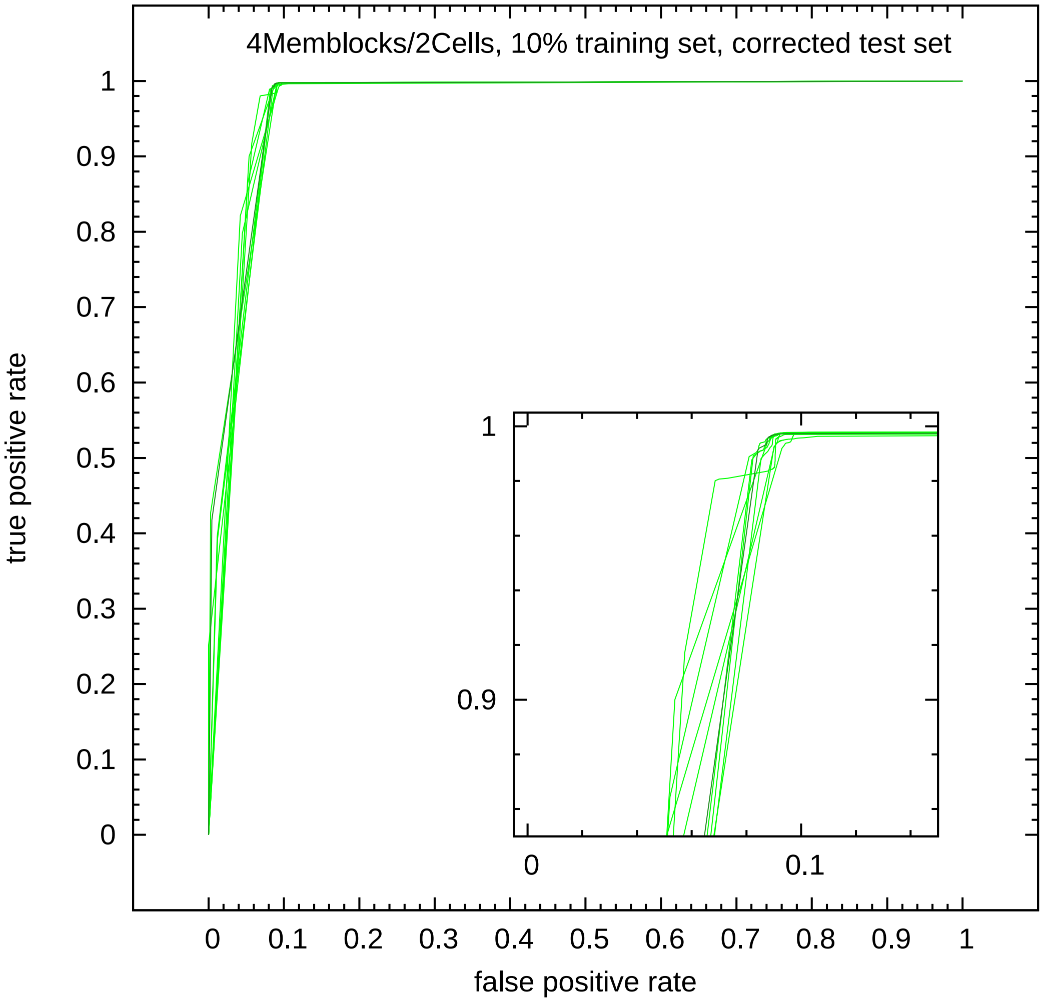


Figure 6.3: *ROC curves of 10 well-performing networks for the target neuron representing the traffic class containing 'normal' traffic. The corresponding AUC values are in the range between [0.9665–0.9716]. This shows a very good classification performance in terms of AUC.*

ROC graph. The curves for network probes and 'dos' attacks show that the classifier separates at least two subclasses.

Further analysing the results of the first experiment, we noted that we have to address LSTM's strong tendency for over-fitting. We did this by significantly increasing the number of performance tests within each trial. In this second, extended experiment on the datasets, we changed our testing procedure by freezing the weights after each epoch and testing the performance on the training and the test set, which led to much better results.

From these experiments onwards, we also added mean squared error, to

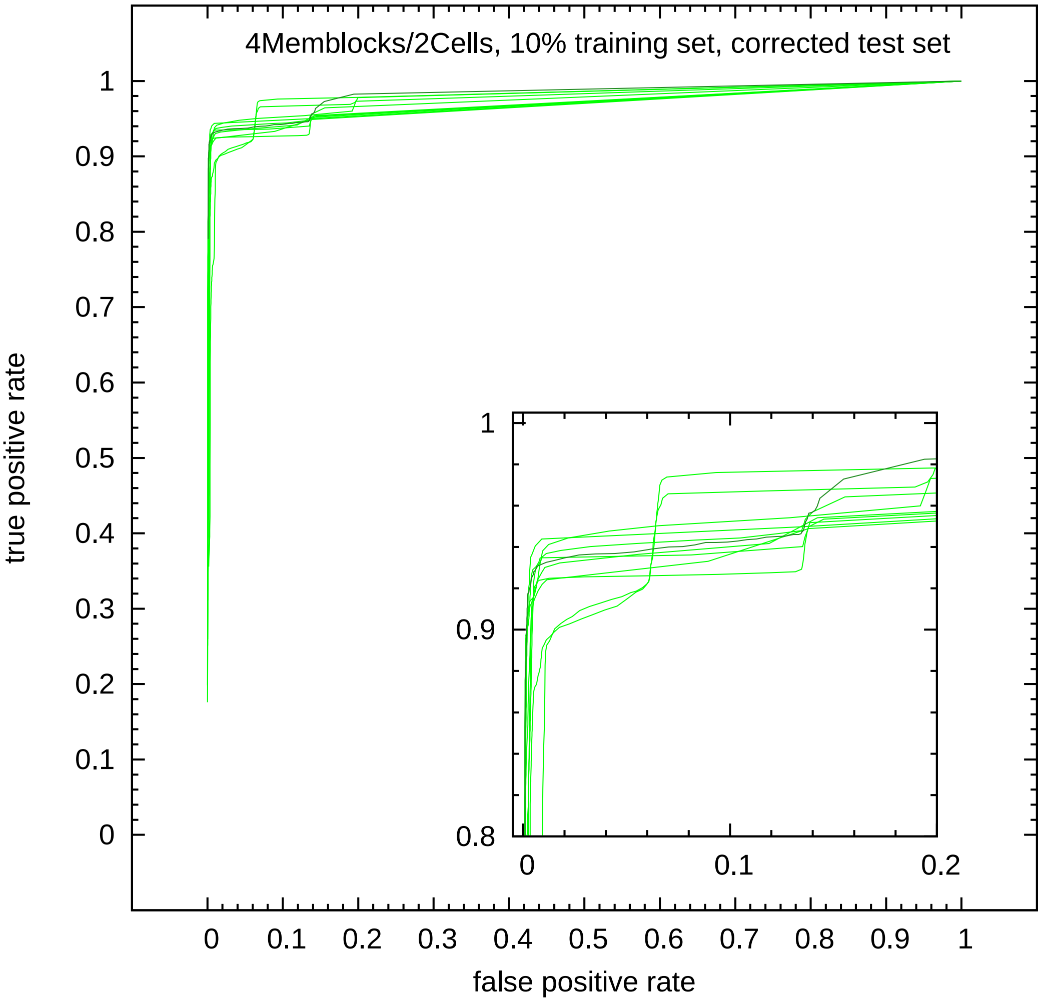


Figure 6.4: *ROC curves of 10 well-performing networks for the target neuron representing the traffic class containing network probes. The corresponding AUC values are in the range [0.9679–0.9826]. In terms of AUC, this shows a very good classification performance. The bumpy curves are an indication that this class contains distinguishable subclasses of traffic.*

accuracy and AUC, as standard performance measures; but due to the large amounts of resulting data, we discarded the empirical operating points for the ROC curves after calculating the AUC values.

To train LSTM networks with all features and all attack classes, we used the same datasets as in the first experiment. We ran 30 trials, for 1,000 epochs each, with a learning rate of 0.1 and a decay of 0.999. We also experimented with a learning rate of 0.5 and 0.1, with a decay of 0.99 each, but the results did not show any performance improvements and are, therefore, not further

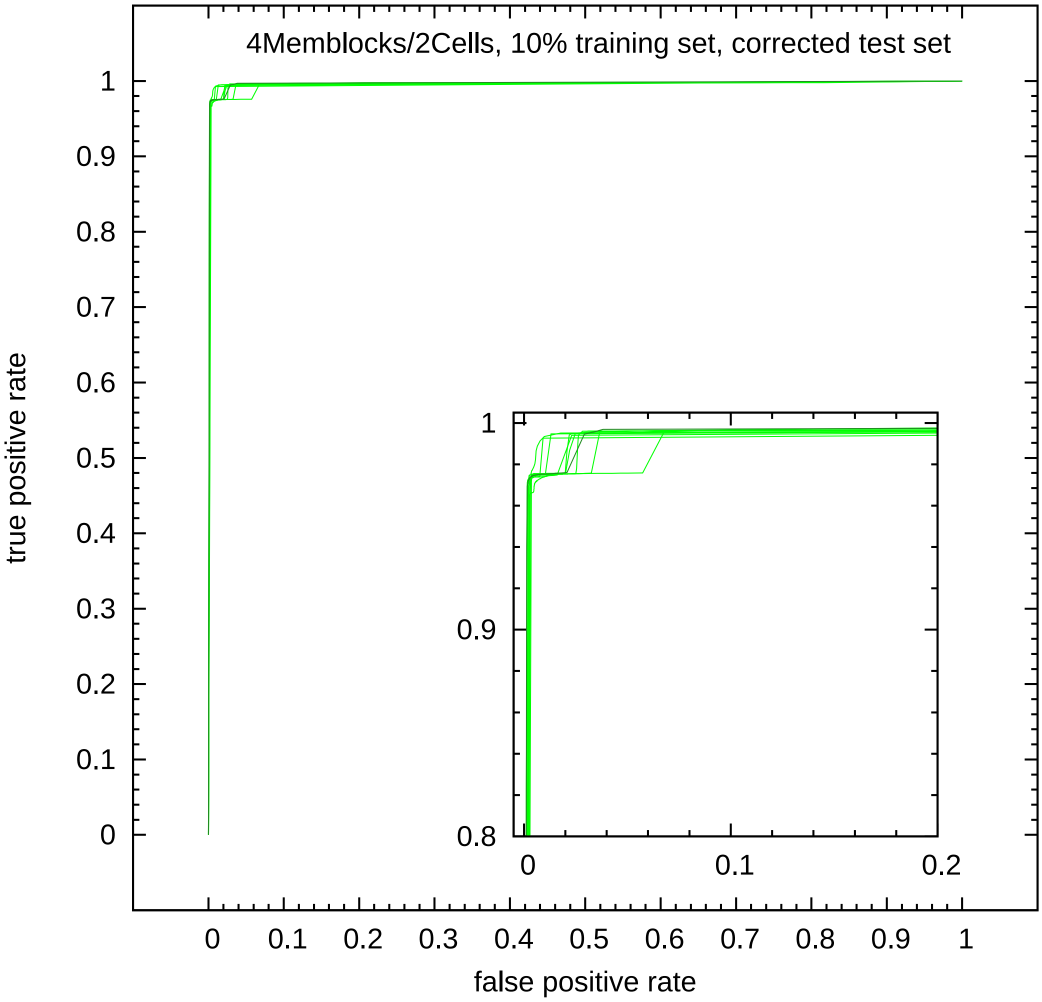


Figure 6.5: *ROC curves of 10 well-performing networks for the target neuron representing the traffic class containing 'dos' attacks. The corresponding AUC values are in the range [0.9950–0.9971], which shows a close to perfect discrimination between 'dos' attacks and other traffic classes.*

considered.

The investigated experimental configuration for training LSTM networks with all features and all attacks showed good results in terms of MSE and accuracy. The 10 lowest MSEs achieved on the test data in 30 trials are in the range [0.0259–0.0293]. The average MSE, picking the result with the lowest MSE of each trial, is 0.0308.

The average AUC values for the five traffic classes are 0.961 ('normal'), 0.991 ('dos'), 0.969 ('probe'), 0.321 ('r2l') and 0.842 ('u2r'). This shows an excellent performance of the trained LSTM networks, which successfully

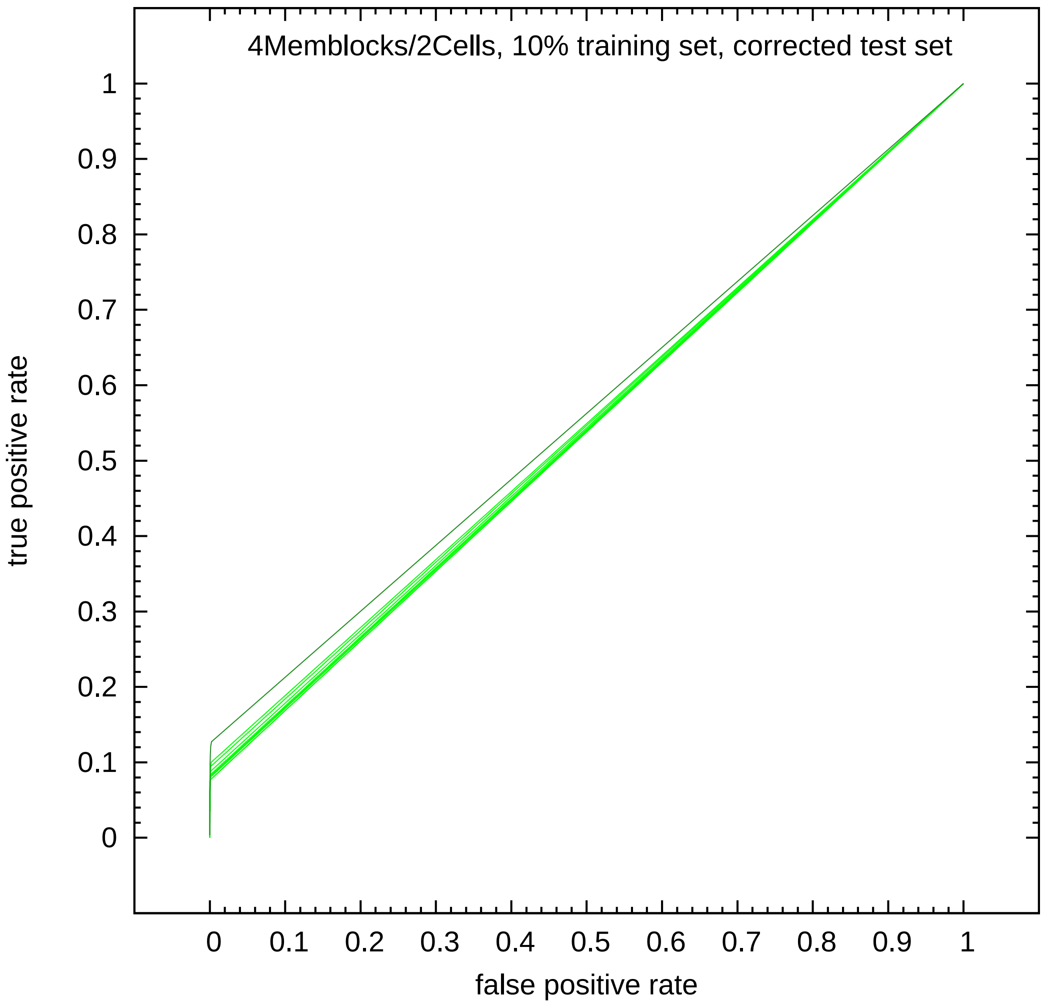


Figure 6.6: *ROC curves of the 10 LSTM networks with the highest AUC values for the target neuron representing the traffic class containing 'r2l' attacks. The corresponding AUC values are in the range between [0.5380–0.5627]. This shows a rather poor ROC performance.*

classified 'dos' attacks.

The performance in detecting 'normal' traffic, network probes and 'u2r' attacks is likewise very good. The AUC performance in classifying 'r2l' attacks is poor. Only a few networks learn to classify 'r2l' attacks with a performance better than guessing.

Table 6.3 shows the top five best-performing values in terms of MAUC. The table also includes the corresponding AUC and MSE values, and the minimum, average and 95% confidence interval over 30 trials of the test data.

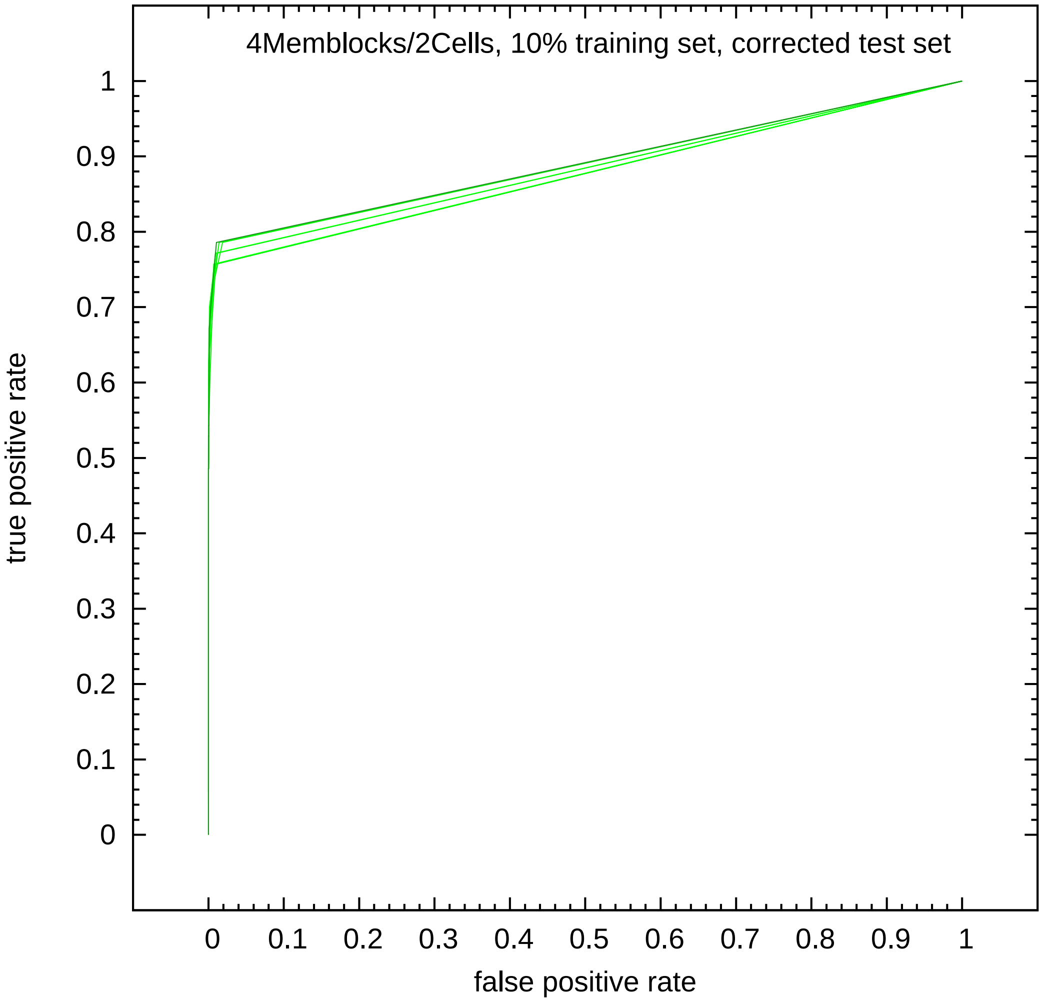


Figure 6.7: *ROC curves of 10 well-performing networks for the target neuron representing the traffic class containing 'u2r' attacks. The corresponding AUC values are in the range between [0.8766–0.8909], which shows an acceptable ROC performance.*

Picking a network with a very low MSE of 0.0259, the five AUC values are 0.959 ('normal'), 0.997 ('dos'), 0.955 ('probe'), 0.241 ('r2l') and 0.821 ('u2r'). Here, we see an exceptional performance of the LSTM network on 'dos' attacks and an above-average performance on normal traffic. This is expected since most connection records in training and test data are related to either 'dos' attacks or normal traffic. All other traffic classes show an AUC performance below average. This also reflects in a MAUC value of 0.949, which is slightly below average.

Nevertheless, in terms of costs and accuracy, our trained network still

Table 6.3: *Summary of test results for training LSTM networks with all features for all traffic classes in one network. The classification performance in terms of AUC for 'normal' traffic, network probes and 'r2l' attacks is very good. The performance for classifying 'dos' attacks is exceptional. Only a few networks learn to classify 'r2l' traffic with a performance beyond guessing.*

#	MSE	AUC					MAUC
		normal	dos	probe	r2l	u2r	
1	.030	.982	.994	.944	.617	.842	.971
2	.030	.954	.995	.990	.564	.796	.964
3	.031	.956	.989	.992	.604	.927	.962
4	.029	.976	.994	.965	.430	.833	.960
5	.031	.966	.990	.913	.473	.897	.957
min.	.029	.937	.982	.908	.186	.584	.935
avg.	.031	.961	.991	.969	.321	.842	.950
95% conf. interval		.956	.989	.960	.276	.812	.946
		.965	.992	.979	.366	.872	.953

outperforms the high-scoring entries of the KDD Cup '99 challenge. With 93.82% accuracy and 22.13 cost, our network is far ahead in first place in comparison to the winning entries of the KDD Cup '99 challenge. LSTM correctly classified 291,811 out of 311,029 instances from the test set. Our result outperforms the KDD Cup '99 winning entry by impressive 3,462 instances.

Observing the results in the confusion matrix shown in Table 6.4, we note the very good true positive rates and precision of 'dos' attacks and 'normal' traffic. The true positive rate and precision for network probes and 'u2r' attacks are also acceptable. Unfortunately, the result for 'r2l' attacks does not excel at all in this network.

6.4.2 Individual Attack Classes

Next, we looked for LSTM networks trained with all features, for all attacks, but performing well on the individual detection of one out of four attack classes. We sorted all trained networks in descending order, according to the AUC value of the target neuron representing the observed attack class. For each attack class, we picked the one network having the highest AUC value for one of the five target neurons. We evaluated all trained networks stored after every training epoch.

Table 6.4: *Confusion matrix of LSTM network trained with all features and all attacks having the lowest MSE on test data. True positive rates and precision of 'dos' attacks and 'normal' traffic are exceptional. True positive rate and precision for network probes and 'u2r' attacks are also acceptable. The result for 'r2l' attacks does not excel in this network.*

		prediction						
		normal	probe	dos	u2r	r2l	TPR (DR)	AUC
actual	normal	60272	233	81	6	1	0.995	0.959
	probe	898	3156	111	1	0	0.758	0.955
	dos	1262	233	228358	0	0	0.993	0.997
	u2r	58	0	0	12	0	0.171	0.821
	r2l	15828	106	399	1	13	0.001	0.241
	PRECISION	0.770	0.847	0.997	0.600	0.929	COST:	0.2213
	FPR (FAR)	0.072	0.002	0.007	0.000	0.000	ACC:	93.82%

The maximum AUC values achieved for the classification of the five traffic classes in the test data are 0.982 ('normal'), 0.997 ('dos'), 0.997 ('probe'), 0.877 ('r2l') and 0.974 ('u2r'). These results show that LSTM achieves excellent AUC values for 'normal' traffic, 'dos' attacks and network probes, and good AUC values for the remaining two traffic types. From this, we conclude that a significant number of the corresponding connection records are correctly identified.

That LSTM successfully learns all traffic classes is confirmed by observing the training data. As expected, we achieved remarkably high AUC values for the training data for all five traffic classes: >0.999 ('normal'), >0.999 ('dos'), >0.995 ('probe'), 0.987 ('r2l') and 0.987 ('u2r'). But when comparing these results to the AUC performance for the test data, we also note a significant degradation of classification performance on 'r2l' attacks. This shows that for these attacks, LSTM has major difficulties generalising from the training data to the test data.

Table 6.5 shows the best five results in terms of AUC, and the minimum, average and the 95% confidence interval over 30 trials. The table also shows the corresponding MSE values.

Table 6.5: *Summary of test results for LSTM training with all features for all traffic classes in different networks. The AUC values achieved for the classification of the five traffic classes in the test data are remarkably high. But when comparing these results to the AUC performance on the test data, we note that for 'r2l' attacks, LSTM has major difficulties generalising from the training data to the test data.*

#	MSE	AUC	MSE	AUC	MSE	AUC	MSE	AUC	MSE	AUC
	normal		dos		probe		r2l		u2r	
1	.033	.982	.031	.997	.072	.997	.034	.877	.031	.974
2	.031	.982	.031	.997	.063	.997	.031	.765	.030	.974
3	.031	.981	.032	.996	.031	.994	.032	.722	.041	.974
4	.032	.981	.029	.996	.062	.993	.056	.562	.029	.972
5	.041	.980	.031	.996	.032	.993	.071	.515	.061	.968
min.	.029	.960	.029	.984	.029	.972	.029	.272	.029	.916
avg.	.047	.973	.039	.994	.053	.987	.053	.435	.053	.952
95% conf. interval		.971		.993		.985		.384		.946
		.976		.995		.989		.486		.959

6.5 Performance Analysis with Minimal Feature Sets

For the detection of all attack classes using minimal feature sets, we used previously extracted minimal sets. The minimal sets used for training all attacks in a single LSTM network consisting of 11, 8 and 4 features. The extraction of these features is described in detail in Section 4.6. In the 11-feature set, the input features were mapped to an input layer of size 12. The 8 and 4-feature sets had input layers with a size equal to the number of used features.

We ran 30 trials to train LSTM networks with the 11-feature set. Applying a learning rate of 0.5 and a decay of 0.99, we ran 30 trials for 1,000 epochs. On the 8 and 4-feature sets, we also ran 30 trials for 1,000 epochs each. They applied a learning rate of 0.1 and a decay of 0.999.

The detection of individual attacks in individual networks was carried out with the minimal sets for individual attacks, as described in Section 4.7. In these experiments, all input layers were equal-sized to the number of used features.

We ran 30 trials per traffic class and trained feature set. Every trial ran for 1,000 epochs, with a learning rate of 0.5 and a decay of 0.99.

6.5.1 Multi-Class Categorisation

Using the 11 and 8-feature sets, all trials show good results in terms of accuracy and MSE. With the 4-feature set, we still get good results for approximately half of the trials (14/30). The lowest MSE found for 11, 8 and 4 features are 0.029, 0.025 and 0.024 respectively, with an average MSE of 0.031, 0.029 and 0.070. Here, we see the tendency of a performance improvement towards the smaller feature sets. But with the 4-feature set, there also comes the disadvantage of a decreasing yield of well-performing networks. This shows that LSTM has increasing difficulty learning to classify the data by using only 4 features.

In terms of MAUC, the LSTM networks trained with 8 features show the best results. In terms of AUC, the results show that LSTM learns to classify the traffic classes ‘normal’, ‘probe’ and ‘u2r’ very well. Once again, the results for classifying ‘dos’ attacks are exceptional. Few networks learn to classify few ‘r2l’ attacks. The top five networks are shown in Table 6.6.

Table 6.6: *Summary of test results for LSTM training with 8 features for all traffic classes in one network. In terms of AUC LSTM learns very well to classify the traffic classes ‘normal’, ‘probe’ and ‘u2r’. The results for classifying ‘dos’ attacks are exceptional. Only some networks learn to classify few ‘r2l’ attacks.*

Rank	MSE	AUC					MAUC
		normal	dos	probe	r2l	u2r	
1	.025	.985	.999	.875	.811	.939	.985
2	.030	.981	.998	.928	.788	.772	.983
3	.029	.973	.995	.952	.720	.804	.976
4	.026	.969	.995	.947	.517	.954	.964
5	.031	.984	.991	.949	.423	.809	.959
min.	.025	.962	.970	.714	.128	.678	.932
avg.	.029	.977	.990	.920	.338	.819	.952
95% conf. interval		.974	.986	.902	.268	.790	.947
		.979	.993	.939	.408	.849	.957

We note that the LSTM networks trained with the 4 and 8-feature sets for all attacks in one network are superior to the results of all other non-LSTM classifiers and the winning entries of the KDD Cup ’99 challenge in terms of accuracy and costs. The results are 93.69% accuracy and 22.29 costs for the 8-feature set, and 93.72% accuracy and 22.24 costs for the 4-feature set.

Compared to the LSTM network trained with all features, we note that these values almost match its performance. The network trained with the 11-feature set offers 92.50% accuracy and 24.45 costs, a competitive performance with an improved result in comparison to the other classifiers trained with 11 features.

The confusion matrices for the networks trained with 4 and 8 features show that the outstanding total performance is mainly due to an excellent detection of 'dos' attacks. The true positive rate for network probes is acceptable, taken into account that some probes are wrongly categorised as 'dos' attacks. The detection of 'r2l' attacks is poor. For this type of attack, the 11-feature set provides a slightly better performance. The 'u2r' attacks are not detected by LSTM using any of the reduced feature sets for all attacks in one network.

The results of the best-performing LSTM networks trained with the 11, 8 and 4-feature sets for all attacks in one network are presented in the form of confusion matrices in Tables 6.7, 6.8 and 6.9.

Table 6.7: *Confusion matrix of LSTM network trained with 11 features and all attacks having the lowest MSE on test data. This network shows a slightly better performance for classifying 'r2l' attacks than other trained networks using reduced feature sets.*

		prediction					TPR	AUC
		normal	probe	dos	u2r	r2l	(DR)	
actual	normal	59630	350	396	2	215	0.984	0.956
	probe	831	3052	281	0	2	0.733	0.948
	dos	5749	57	224047	0	0	0.975	0.995
	u2r	59	0	0	0	11	0.000	0.840
	r2l	15270	19	75	0	983	0.060	0.491
PRECISION		0.731	0.878	0.997	0.000	0.812	COST:	0.2445
FPR (FAR)		0.087	0.001	0.009	0.000	0.001	ACC:	92.50%

6.5.2 Individual Attack Classes

Using the minimal sets for each traffic class, we tested to see if LSTM performs even better when classifying individual attacks. The minimum MSE values found for two-class classification ('normal' / attack class) are 0.0009 (5 features 'dos'), 0.0023 (6 features 'probe'), 0.0810 (6 features 'r2l'), 0.0727 (14 features 'r2l'), 0.0003 (5 features 'u2r'), and 0.0003 (8 features 'u2r'). The maximum

Table 6.8: *Confusion matrix of LSTM network trained with 8 features and all attacks with the lowest MSE on test data. This network shows an outstanding overall performance due to an excellent detection of 'dos' attacks and an acceptable classification performance of network probes. The performance of detecting 'r2l' and 'u2r' attacks is poor; but the high AUC values suggest there is still potential to learn them.*

		prediction					TPR	AUC
		normal	probe	dos	u2r	r2l	(DR)	
actual	normal	60161	223	181	0	28	0.993	0.985
	probe	1001	2546	619	0	0	0.611	0.875
	dos	1217	226	228408	2	0	0.994	0.999
	u2r	70	0	0	0	0	0.000	0.939
	r2l	15687	169	199	4	288	0.018	0.811
PRECISION		0.770	0.805	0.996	0.000	0.911	COST:	0.2229
FPR (FAR)		0.072	0.002	0.012	0.000	0.000	ACC:	93.69%

Table 6.9: *Confusion matrix of LSTM network trained with 4 features and all attacks with the lowest MSE on test data. This network shows an outstanding overall performance due to an excellent detection of 'dos' attacks and a still acceptable performance for classifying network probes.*

		prediction					TPR	AUC
		normal	probe	dos	u2r	r2l	(DR)	
actual	normal	60182	154	221	0	36	0.993	0.967
	probe	889	2348	928	0	1	0.564	0.870
	dos	723	195	228935	0	0	0.996	0.993
	u2r	68	0	2	0	0	0.000	0.751
	r2l	16229	9	81	0	28	0.002	0.304
PRECISION		0.771	0.868	0.995	-	0.431	COST:	0.2264
FPR (FAR)		0.072	0.001	0.015	0.000	0.000	ACC:	93.72%

AUC values achieved are 0.9977 (5 features 'dos'), 0.9990 (6 features 'probe'), 0.8724 (6 features 'r2l'), 0.8826 (14 features 'r2l'), 0.9922 (5 features 'u2r'), and 0.9909 (8 features 'u2r').

Feature reduction did not have a negative effect on the classification of 'dos' attacks. Here, the MSE and AUC performance remains excellent. The classification performance of the three other attack classes improves after feature reduction. In terms of AUC, the improvement for 'r2l' and 'u2r' attacks is remarkable. These results show that feature reduction and the training of individual networks do have a strong positive effect.

The best five networks in terms of AUC for LSTM networks trained with the minimal feature sets for individual attack classes in individual networks are shown in Table 6.10.

Table 6.10: *Summary of test results for LSTM training with minimal features for individual attack classes in individual networks. Feature reduction did not have a negative effect on the classification of 'dos' attacks. The classification performance of the three other attack classes improves after feature reduction. These results show that feature reduction and the training of individual networks do have a strong positive effect.*

#	MSE	AUC	MSE	AUC	MSE	AUC	MSE	AUC
	dos (5 feat.)		probe (6 feat.)		r2l (14 feat.)		u2r (5 feat.)	
1	.005	.998	.002	.999	.085	.883	.000	.992
2	.002	.997	.002	.999	.085	.857	.000	.992
3	.007	.997	.003	.999	.085	.856	.000	.992
4	.001	.997	.003	.999	.085	.850	.000	.992
5	.011	.997	.003	.999	.085	.845	.000	.992
min.	.001	.989	.002	.998	.082	.605	.000	.991
avg.	.082	.995	.004	.998	.084	.788	.000	.992
95% conf.		.994		.998		.761		.991
interval		.996		.999		.815		.992

6.6 Classifier Performance Comparison

To compare the performance of the LSTM classifier to the classifiers evaluated in Chapter 5, we picked trained networks with high AUC values and competitively high true positive rates for each traffic type. This reveals more interesting details. In terms of true positive rate, false positive rate, precision, accuracy and cost, the performance of the LSTM classifier is superior at detecting 'dos' attacks and network probes. The results for detecting 'r2l' and 'u2r' attacks are indeed very competitive, but do not match the performance of the very well-performing decision tree classifier.

For network probes, 'dos' attacks and 'r2l' attacks, LSTM sacrifices precision for an improved true positive rate, when training with the minimal sets. Only for 'u2r' attacks does LSTM improve noticeably with a reduced feature set. Compared to the performance of other trained classifiers on the minimal set, we note that for 'dos' attacks and network probes, LSTM is superior in terms of true positive rate, false positive rate, precision, accuracy and cost. For ‘r2l’ attacks trained with 6 and 14 features, LSTM clearly outperforms the two other neural network-based classifiers (SVM, MLP).

The per-class performance results of LSTM networks trained with all features and minimal feature sets, in comparison with all other trained classifiers, are presented in Tables 6.11 (‘dos’), 6.12 (‘probe’), 6.13 (‘r2l’) and 6.14 (‘u2r’).

Table 6.11: *Performance comparison of ‘dos’ attack detection using the preprocessed full feature set and the 5-feature minimal set (5p). LSTM outperforms all other classifiers by far. Peak performance is shown by using the minimal set.*

dataset	classifier	dos		ACC	COST
10% set		TPR	FPR		
39p	J4.8	0.974	0.002	97.88%	0.0424
	BayesNet	0.968	0.001	97.43%	0.0515
	nBayes	0.970	0.024	97.15%	0.0571
	MLP	0.974	0.002	97.91%	0.0419
	SVM	0.973	0.002	97.84%	0.0433
	LSTM	0.993	0.001	99.46%	0.0109
5p	J4.8	0.977	0.003	98.11%	0.0377
	BayesNet	0.942	0.002	95.36%	0.0928
	nBayes	0.260	0.015	41.12%	1.1776
	MLP	0.977	0.004	98.12%	0.0375
	SVM	0.971	0.011	97.48%	0.0503
	LSTM	0.998	0.004	99.78%	0.0044

6.7 Conclusions

In this chapter, we evaluated the performance of the LSTM recurrent neural network classifier applied to the preprocessed KDD Cup ’99 datasets. We discussed the selected experimental parameters and the chosen LSTM network topology. Furthermore, we made some notes on improving LSTM performance.

We ran experiments, training all attacks in a single LSTM network, and training individual networks for each attack class. We used both the full feature and minimal feature sets. Finally, we compared the results achieved to the performance of the classifiers tested in Chapter 5.

Table 6.12: *Performance comparison of all tested classifiers for network probe detection using the preprocessed full feature set and the 6-feature minimal set (6p). LSTM shows a superior performance. Peak performance is shown using the minimal set.*

dataset	classifier	probe		ACC	COST
10% set		TPR	FPR		
39p	J4.8	0.779	0.005	98.13%	0.0187
	BayesNet	0.841	0.005	98.49%	0.0151
	nBayes	0.923	0.007	98.82%	0.0118
	MLP	0.844	0.004	98.62%	0.0138
	SVM	0.772	0.014	97.23%	0.0277
	LSTM	0.937	0.004	99.22%	0.0078
6p	J4.8	0.831	0.004	98.52%	0.0148
	BayesNet	0.722	0.004	97.82%	0.0218
	nBayes	0.860	0.009	98.23%	0.0177
	MLP	0.886	0.005	98.81%	0.0119
	SVM	0.815	0.006	98.27%	0.0173
	LSTM	0.964	0.004	99.35%	0.0065

Table 6.13: *Performance comparison of ‘r2l’ attack detection using the preprocessed full feature set, the 14-feature set (14p) and the 6-feature minimal set (6p). LSTM is the best-performing classifier, although all classifiers show a poor performance. Best results are achieved using the 14-feature set.*

dataset	classifier	r2l		ACC	COST
10% set		TPR	FPR		
39p	J4.8	0.088	0.000	80.61%	0.5815
	BayesNet	0.104	0.004	80.70%	0.5764
	nBayes	0.127	0.010	80.69%	0.5716
	MLP	0.046	0.000	79.72%	0.6084
	SVM	0.092	0.000	80.66%	0.5797
	LSTM	0.055	0.000	79.92%	0.6022
14p	J4.8	0.088	0.000	80.62%	0.5813
	BayesNet	0.065	0.001	80.03%	0.5980
	nBayes	0.054	0.007	79.35%	0.6140
	MLP	0.029	0.000	79.34%	0.6196
	SVM	0.034	0.001	79.42%	0.6168
	LSTM	0.223	0.039	80.41%	0.5568
6p	J4.8	0.033	0.000	79.45%	0.6164
	BayesNet	0.055	0.002	79.76%	0.6057
	nBayes	0.081	0.008	79.87%	0.5979
	MLP	0.000	0.000	78.74%	0.6377
	SVM	0.000	0.000	78.72%	0.6380
	LSTM	0.047	0.001	79.64%	0.6097

Table 6.14: *Performance comparison of 'u2r' attack detection using the preprocessed full feature set and the 5-feature minimal set (5p). Best-performing classifier is the J4.8 decision tree classifier. The LSTM performance is acceptable. Peak performance is achieved using the minimal set.*

dataset	classifier	u2r		ACC	COST
10% set		TPR	FPR		
39p	J4.8	0.457	0.000	99.92%	0.0028
	BayesNet	0.671	0.001	99.86%	0.0035
	nBayes	0.843	0.013	98.67%	0.0270
	MLP	0.343	0.000	99.91%	0.0033
	SVM	0.357	0.000	99.91%	0.0032
	LSTM	0.357	0.000	99.90%	0.0035
5p	J4.8	0.643	0.000	99.96%	0.0017
	BayesNet	0.371	0.000	99.92%	0.0030
	nBayes	0.543	0.001	99.81%	0.0049
	MLP	0.314	0.000	99.92%	0.0032
	SVM	0.000	0.000	99.88%	0.0046
	LSTM	0.486	0.000	99.93%	0.0026

The LSTM classifier shows its strengths when training 'dos' and 'probe' attacks. These attack classes tend to generate a high volume of consecutive connection records. Here, LSTM can strongly benefit from the fact that it can look back in time and learn to correlate these connections. The two exploit-based attack classes, 'r2l' and 'u2r', in most cases, generate only one connection record. If there is any time series information related to these attacks hidden between other connection records, it seems to be very difficult to extract them. We conclude that LSTM is very suitable for classifying high-frequency attacks. For very low-frequency attacks, the benefit of using LSTM vanishes. Although we should stress that LSTM is still able to compete with the other classifiers tested.

This chapter is the first reported demonstration of the successful application of LSTM networks to intrusion detection.

Chapter 7
Conclusions

In this thesis, we have assumed that usage patterns can be used to distinguish 'normal' computer traffic from network 'attacks' and that these patterns can be learned by a machine. 'Static' machine learning methods have been applied to intrusion detection for quite some time, but so far with very limited success to previously unseen attack patterns. This holds true especially for intrusions with a very low profile over long time periods.

We set out to show what can be gained by applying a classifier that is able to model the time series inherent in the network data. We assumed that the LSTM recurrent neural network classifier outperforms *static* machine learning methods for modelling network intrusions. To support this claim, we compared five *static* classifiers that are known to perform well on the training data used.

The results presented in Chapter 5 and Chapter 6 confirm our hypothesis. LSTM proved to be superior to all other tested classifiers. It shows its strengths when classifying attacks that generate a large number of consecutive connection records, such as network probes and 'dos' attacks. Here, our classifier benefits from the fact that LSTM can look back in time and correlate these records. LSTM requires only a few presentations of the training data to learn these attacks (less than 100 epochs). The well-trained LSTM networks are able to successfully distinguish these attacks from 'normal' traffic using test data with previously unseen variants of intrusions.

Some LSTM networks do also learn attacks with only a few examples in the training data, such as 'r2l' and 'u2r' attacks. Most of these attacks generate only one record, and here, LSTM requires many more presentations of the training data. We trained our LSTM networks for up to 1,000 epochs. But thereafter, most of the learned models suffer from overfitting of the training data. If there is any time series information related to these attacks hidden in the training data, it must be very difficult to extract. For these rare attacks, the benefit of LSTM in comparison to the other tested classifiers vanishes.

Our contributions are manifold:

- Data preprocessing: To prepare the KDD Cup '99 IDS data, we conducted a detailed analysis of the dataset and presented a preprocessing framework. After preprocessing, the majority of features show improved information gain of approximately 20%. Furthermore, all classifiers show an improved performance on the preprocessed datasets. The presented data preparation and preprocessing steps can be applied to any IDS dataset that may arise in future.

- Salient feature extraction: To reduce the amount of data processed by the classifiers, we then presented a number of intuitive steps to detect and remove unnecessary features. This process starts by ranking all features using information gain. Then we used a combined approach of feature reduction based on decision tree pruning, biased backward elimination and forward selection, and heuristic domain knowledge.

  This process was supported by the visualisation of feature distributions and relationships between features using distribution histograms and scatter plot matrices. Visualisation proved to be of high value in identifying features that were candidates for removal. The suggested technique proved to be very effective for finding small sets of salient features with 4–8 features.

- Minimal feature sets: The feature reduction process resulted in minimal feature sets for detecting all attacks and individual attack classes in two-class and multi-class classification. For detecting all attacks, we presented feature sets with 11, 8 and 4 features, with the majority of selected features being basic features, easily extracted from a network stream. For individual attack classes, we present feature sets with 4–6 features. For the two rare attack classes, 'r2l' and 'u2r' attacks, we additionally presented feature sets with 14 and 8 features respectively. We show in 'X-1' histograms that any further removal of remaining features leads to a remarkable degradation in performance.

- Model network traffic using static classifiers: We tested the performance of five very common and well-known classifiers on the KDD Cup '99 data, four strong, and one simple probabilistic machine learning method. As a simple machine learning method, we applied the well-known *naïve Bayes* classifier. For strong machine learning methods, we chose *J4.8 decision trees*, *Bayesian networks*, *MLP feed-forward neural networks*, and *support vector machines* (SVM). With all classifiers using our preprocessed KDD Cup '99 datasets and our minimal feature sets, we were able to produce at least competitive results, with a performance comparable to the winning entries of the KDD Cup '99 challenge.

- Model time series traffic using the LSTM classifier: Finally, we applied our own implementation of the *LSTM recurrent neural network* classifier to the KDD Cup '99 data. The results show that the LSTM classifier provides a superior performance in comparison to all other tested static classifiers. The strengths are in the detection of 'dos' attacks and network probes, which both produce a distinctive time series of events. The performance on the attack classes that produce only a few events is comparable to the results of the other tested classifiers.

In future research projects, we suggest that the following problems be addressed:

- Time series of connection records: To benefit fully from the capabilities of LSTM network intrusion detection, datasets are required that are more suited to time series prediction. For this, we need to address at least the following issues:
  - In the DARPA/KDD Cup '99 datasets, the connection records are ordered according to the timestamp when the connection is closed, which makes time series analysis unnecessarily difficult for the learning algorithm. Therefore, the connection records should be ordered according to the time at which the first packet was sent.
  - We used the number of correctly classified connection records as a performance measure. The results are misleading, because attacks can generate between one and millions of records. When a specific

attack is detected, the number of correctly identified connection records that are part of the attack needs to be counted for that attack, and not individually.

  - We will not benefit from LSTM applied to attacks with only a few related connection records. To detect these attacks, we need to prepare the network traffic data in a different way. It is conceivable to break a communication session into multiple records. For this, we could use a hybrid approach, taking into account a time-window and a transferred data threshold.

- Live data: The nature of computer attacks is that they are very dynamic. Most of the attacks generated during the DARPA IDS evaluation were already well-known and outdated in the year 1998 when the evaluation was conducted. Until today, approaches taken for detecting and addressing attacks have changed several times. Today's attacks are very different from the attacks of five years ago; and in five years time, they will be very different from today. And it is safe to assume that they will all require different features to be learned.

  Intrusion detection systems need to deal with live data, and we need to understand how to learn using live data. This implies that we should address the following issues:

  - We need to learn how to capture, aggregate and store live traffic data on highly utilised, high-performance links.
  - Once the collected traffic data is available offline, we need to learn how to process the data efficiently into connection records.
  - We need to work on the development of a framework that supports an expert to easily enrich the generated connection records with additional information. This is for the intuitive building of new features considered to be relevant for the detection of novel attacks. This could be processed log information, network flows and alarms provided by hosts, syslog servers, switches, routers, firewalls, intrusion detection systems, and penetration testing tools.

  - Next, we need to build a framework that supports an expert to label connection records in an efficient way. Using current tools, this task is so time consuming that until all traffic necessary to train a supervised machine learner is labelled, the traffic is already far outdated.

- Feature selection: There are other promising feature selection algorithms. We think that principal component analysis (PCA) is a worthy candidate. In PCA, we transform a large set of interrelated variables into a smaller set of new and uncorrelated variables called the *principal components* (PCs). They are ordered in such a way that the first few PCs retain most of the variation presented in all of the original variables.

- Unsupervised learning approach: We experimented in this thesis with a supervised learning approach. When dealing with live traffic, another and possibly more promising option, is to try an unsupervised learning approach for training LSTM. This also addresses the serious issue of labelling connection records, since an unsupervised learner does not depend on labelled training data. However, to evaluate the performance of the learner, we also need the support of an expert.

- Training 'normal' traffic: The applied test data contains at least two attacks that cannot be derived from attacks in the training data. The 'smnpguess' and 'snmpgetattack', for example, target network infrastructure devices. These two attacks differ fundamentally from the other 'r2l' attacks presented in the training data. But at the same time they generate 50% of all 'r2l' connections in the test data. If we want to detect novel attacks that cannot be derived from any attacks presented in the training data, we need to train the classifier to build a model of 'normal' traffic. This model will then be able extract traffic which differs from that learned behaviour.

- Build more complex LSTM networks: We expect LSTM to improve when we increase the number of memory blocks and memory cells, and lower the learning rate.

Appendix A
Tables and Figures

Table A.1: *The 41 features provided by the KDD Cup '99 datasets.*

Nr	**features**	
	name	description
1	duration	duration of connection in seconds
2	protocol_type	connection protocol (tcp, udp, icmp)
3	service	dst port mapped to service (e.g., http, ftp, etc.)
4	flag	normal or error status flag of connection
5	src_bytes	number of data bytes from src to dst
6	dst_bytes	bytes from dst to src
7	land	1 if connection is from/to the same host/port; else 0
8	wrong_fragment	number of 'wrong' fragments (values 0,1,3)
9	urgent	number of urgent packets
10	hot	number of 'hot' indicators (bro-ids feature)
11	num_failed_logins	number of failed login attempts
12	logged_in	1 if successfully logged in; else 0
13	num_compromised	number of 'compromised' conditions
14	root_shell	1 if root shell is obtained; else 0
15	su_attempted	1 if 'su root' command attempted; else 0
16	num_root	number of 'root' accesses
17	num_file_creations	number of file creation operations
18	num_shells	number of shell prompts
19	num_access_files	number of operations on access control files
20	num_outbound_cmds	number of outbound commands in an ftp session
21	is_hot_login	1 if login belongs to 'hot' list (e.g., root, adm); else 0
22	is_guest_login	1 if login is 'guest' login (e.g., guest, anonymous); else 0
23	count	number of connections to same host as current connection in past two seconds
24	srv_count	number of connections to same service as current connection in past two seconds
25	serror_rate	% of connections that have 'SYN' errors
26	srv_serror_rate	% of connections that have 'SYN' errors
27	rerror_rate	% of connections that have 'REJ' errors
28	srv_rerror_rate	% of connections that have 'REJ' errors
29	same_srv_rate	% of connections to the same service
30	diff_srv_rate	% of connections to different services
31	srv_diff_host_rate	% of connections to different hosts
32	dst_host_count	count of connections having same dst host
33	dst_host_srv_count	count of connections having same dst host and using same service
34	dst_host_same_srv_rate	% of connections having same dst port and using same service
35	dst_host_diff_srv_rate	% of different services on current host
36	dst_host_same_src_port_rate	% of connections to current host having same src port
37	dst_host_srv_diff_host_rate	% of connections to same service coming from diff. hosts
38	dst_host_serror_rate	% of connections to current host that have an S0 error
39	dst_host_srv_serror_rate	% of connections to current host and specified service that have an S0 error
40	dst_host_rerror_rate	% of connections to current host that have an RST error
41	dst_host_srv_rerror_rate	% of connections to the current host and specified service that have an RST error

Table A.2: *Traffic types and their occurrences in all labelled KDD Cup '99 datasets.*

		training			**test**
traffic label	**type/class**	**full**	**10%**	**10422**	**10%**
apache2	dos	0	0	0	794
back	dos	2203	2203	1000	1098
buffer_overflow	u2r	30	30	30	22
ftp_write	r2l	8	8	8	3
guess_passwd	r2l	53	53	53	4367
httptunnel	r2l(u2r)*	0	0	0	158
imap	r2l	12	12	12	1
ipsweep	probe	12481	1247	1000	306
land	dos	21	21	21	9
loadmodule	u2r	9	9	9	2
mailbomb	dos	0	0	0	5000
mscan	probe	0	0	0	1053
multihop	r2l(u2r)*	7	7	7	18
named	r2l	0	0	0	17
neptune	dos	1072017	107201	1000	58001
nmap	probe	2316	231	1000	84
normal	normal	972781	97278	1000	60593
perl	u2r	3	3	3	2
phf	r2l	4	4	4	2
pod	dos	264	264	264	87
portsweep	probe	10413	1040	1000	354
processtable	dos	0	0	0	759
ps	u2r	0	0	0	16
rootkit	u2r	10	10	10	13
saint	probe	0	0	0	736
satan	probe	15892	1589	1000	1633
sendmail	r2l	0	0	0	17
smurf	dos	2807886	280790	1000	164091
snmpgetattack	r2l	0	0	0	7741
snmpguess	r2l	0	0	0	2406
spy	r2l	2	2	2	0
sqlattack	u2r	0	0	0	2
teardrop	dos	979	979	979	12
udpstorm	dos	0	0	0	2
warezclient	r2l	1020	1020	1000	0
warezmaster	r2l(dos)*	20	20	20	1602
worm	r2l	0	0	0	2
xlock	r2l	0	0	0	9
xsnoop	r2l	0	0	0	4
xterm	u2r	0	0	0	13
$\sum$		4898431	494021	10422	311029
$\sum$	normal	972781	97278	1000	60593
	dos	3883370	391458	4264	229853
	probe	41102	4107	4000	4166
	r2l	1126	1126	1106	16347
	u2r	52	52	52	70

* = *multiple categorisation possible*

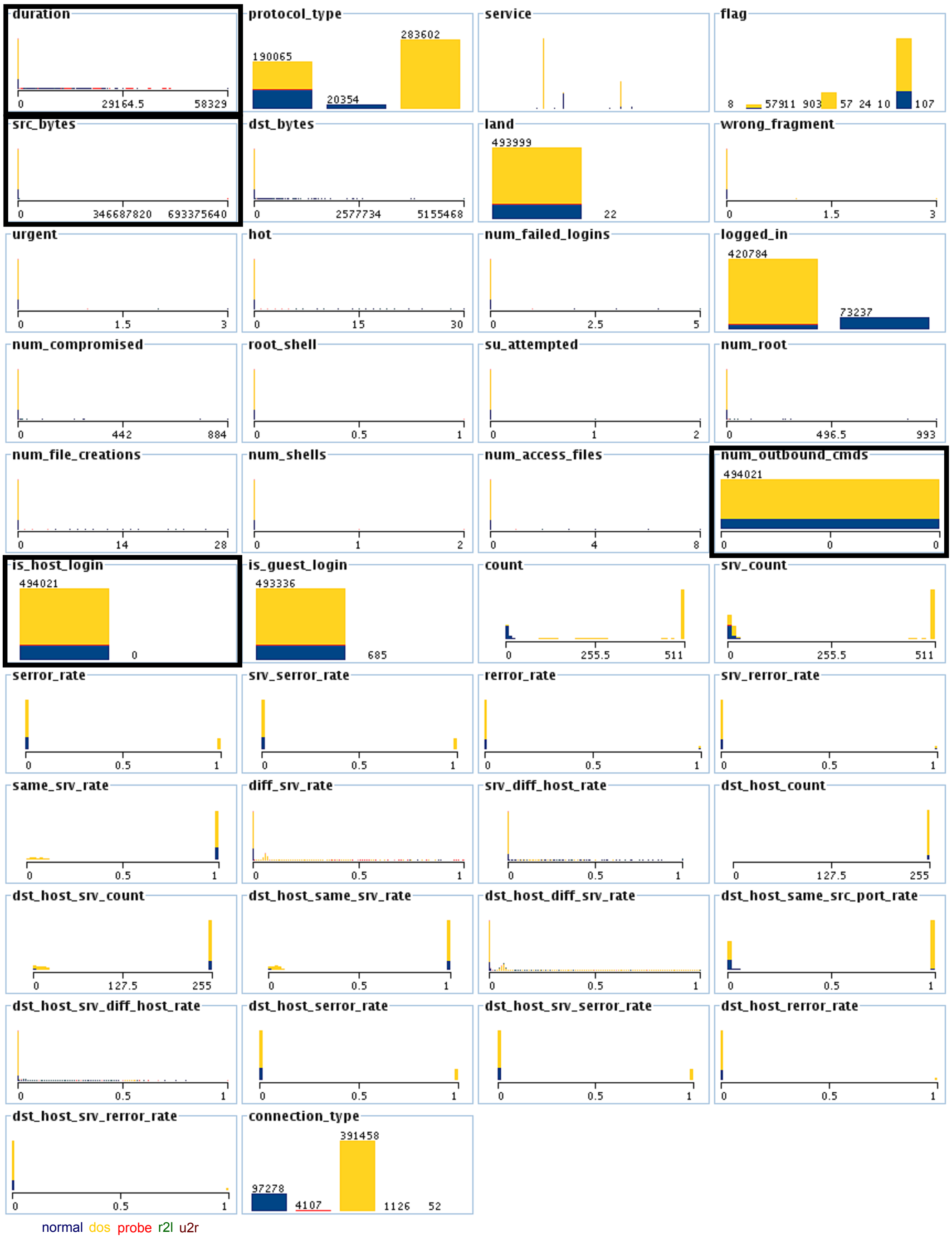


Figure A.1: *Distribution histograms of all features in the original KDD Cup '99 '10%' training data. The x-axis shows the value of the feature and the y-axis shows how often the value exists in the training data. The highlighted features 'num_outbound_cmds' and 'is_host_login' show no variance. The highlighted features 'duration', 'src_bytes' and 'dst_bytes' have strongly biased distributions with some huge outliers.*

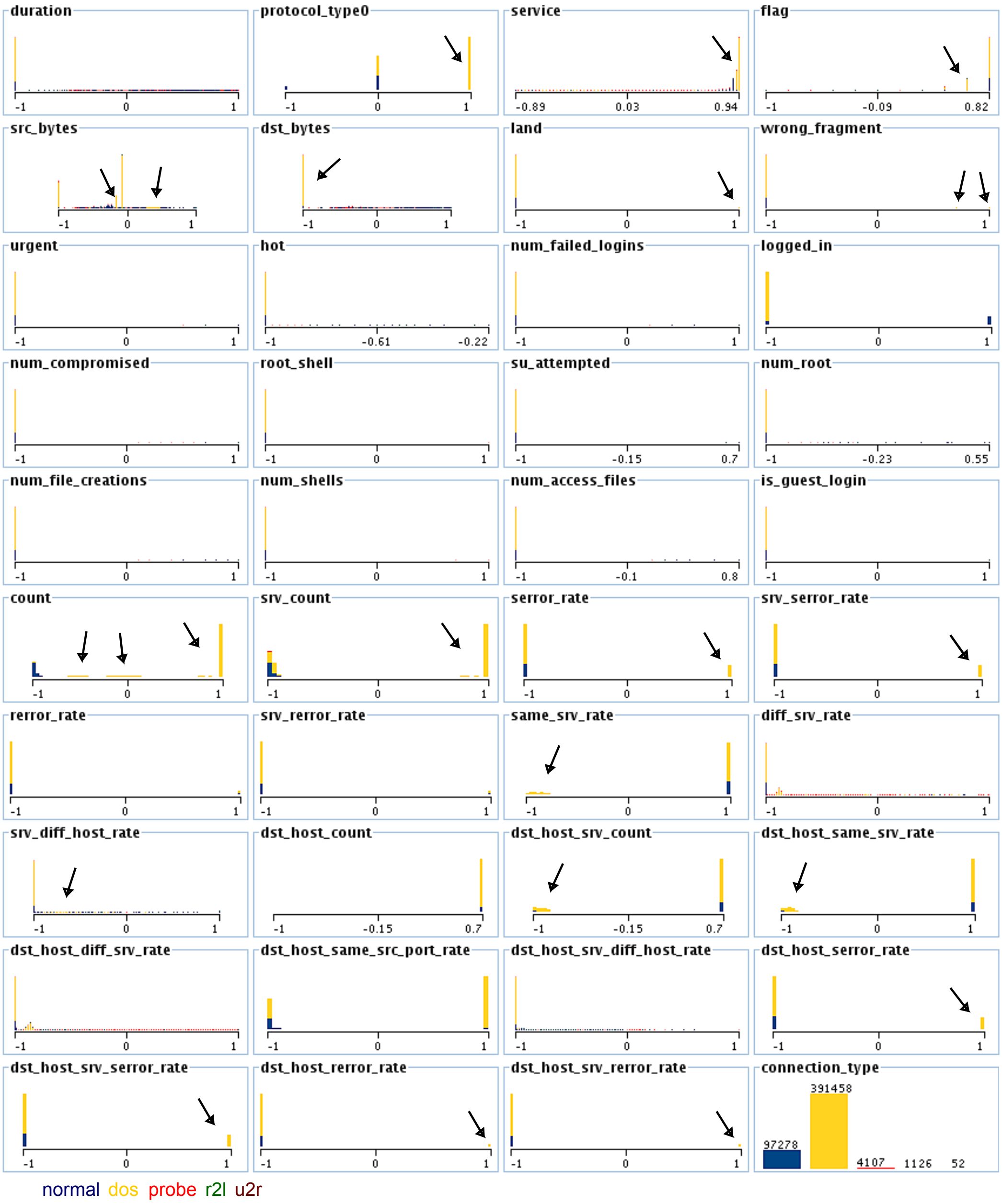


Figure A.2: *Distributions histograms of all features in the preprocessed KDD Cup '99 '10%' training data. The x-axis shows the value of the feature and the y-axis shows how often the value exists in the training data. The highlighted features strongly correlate with the 'dos' attack class.*

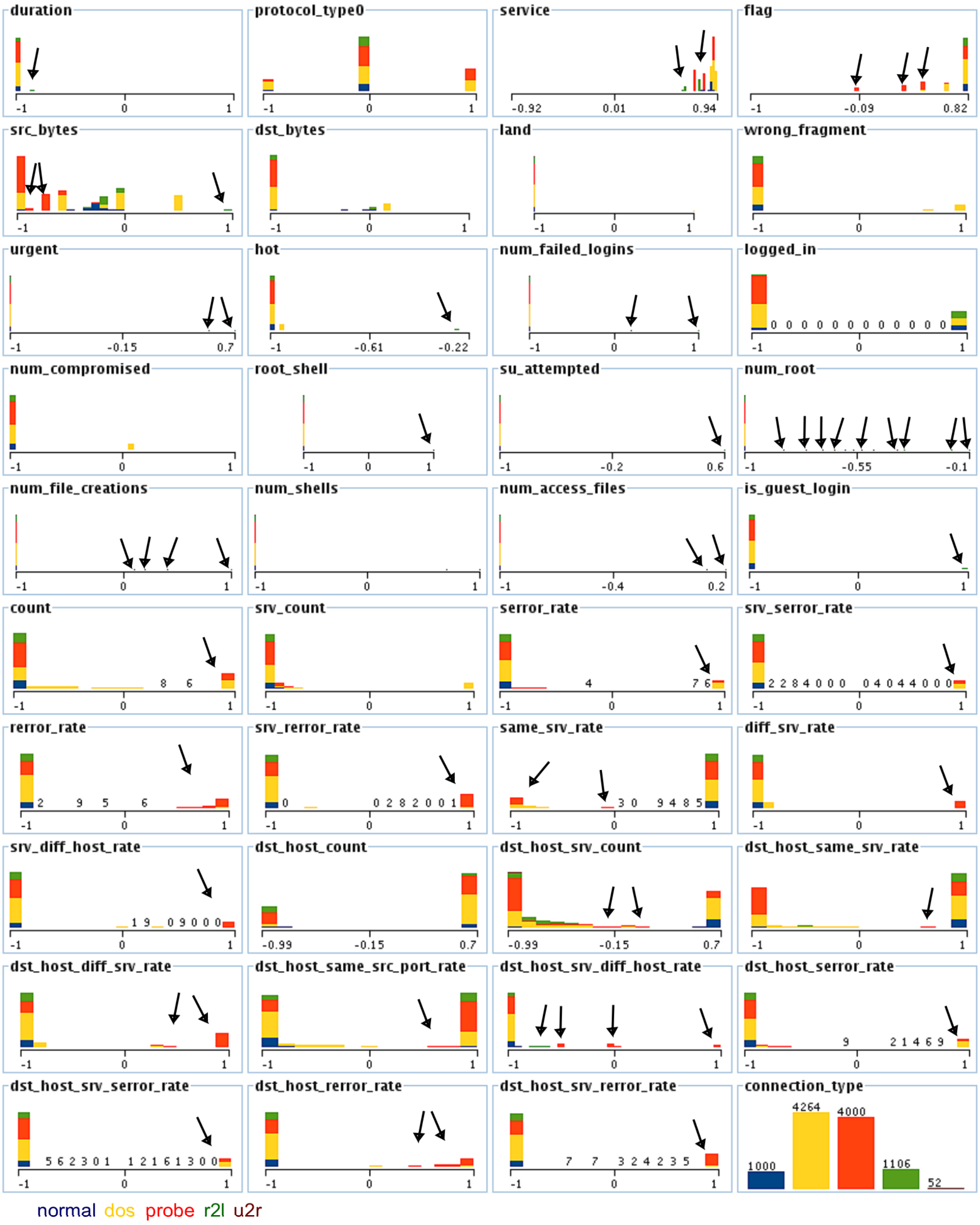


Figure A.3: *Distributions of all features in the custom dataset with 10,422 training instances extracted from the '10%' training dataset. The x-axis shows the value of the feature and the y-axis shows how often the value exists in the training data. A selection of interesting features with visible correlations to the attack classes are highlighted.*

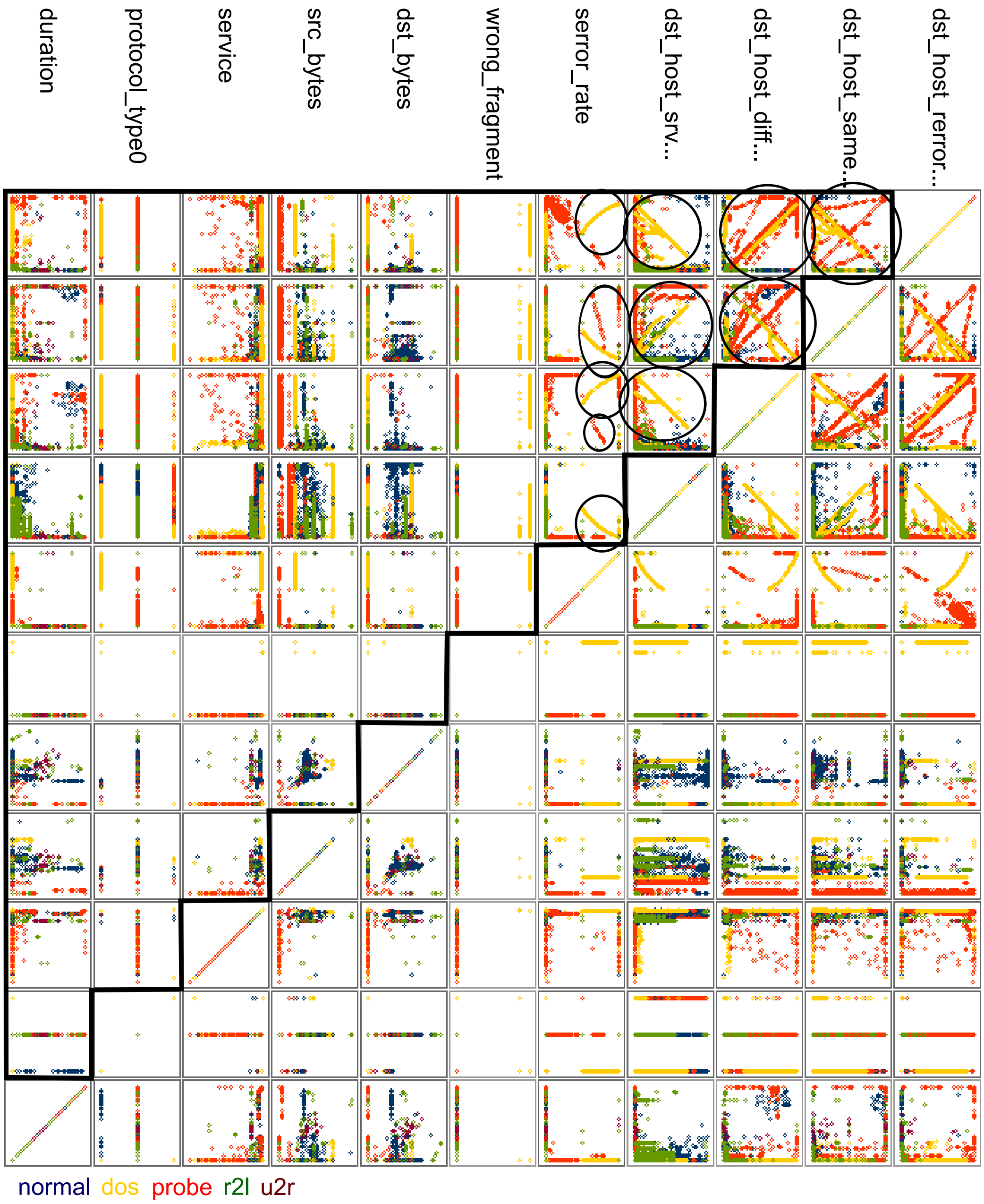


Figure A.4: *This shows the scatter plot matrix of the 11 features in the custom training set with 10,422 instances. The features in the order from left to right and from bottom to top are 'duration', 'protocol_type', 'service', 'source_bytes', 'dst_bytes', 'wrong_fragment', 'serror_rate', 'dst_host_srv_count', 'dst_host_diff_srv_rate', 'dst_host_same_src_port_rate' and 'dst_host_rerror_rate'. The group of relevant scatter plots are framed with a black line starting from the upper left corner. Interesting is the clustering of data points along a line in the scatter plots of the five higher level features. The areas are highlighted with a circle. The clustering indicates strong correlations for 'dos' and 'probe' attacks between these features.*

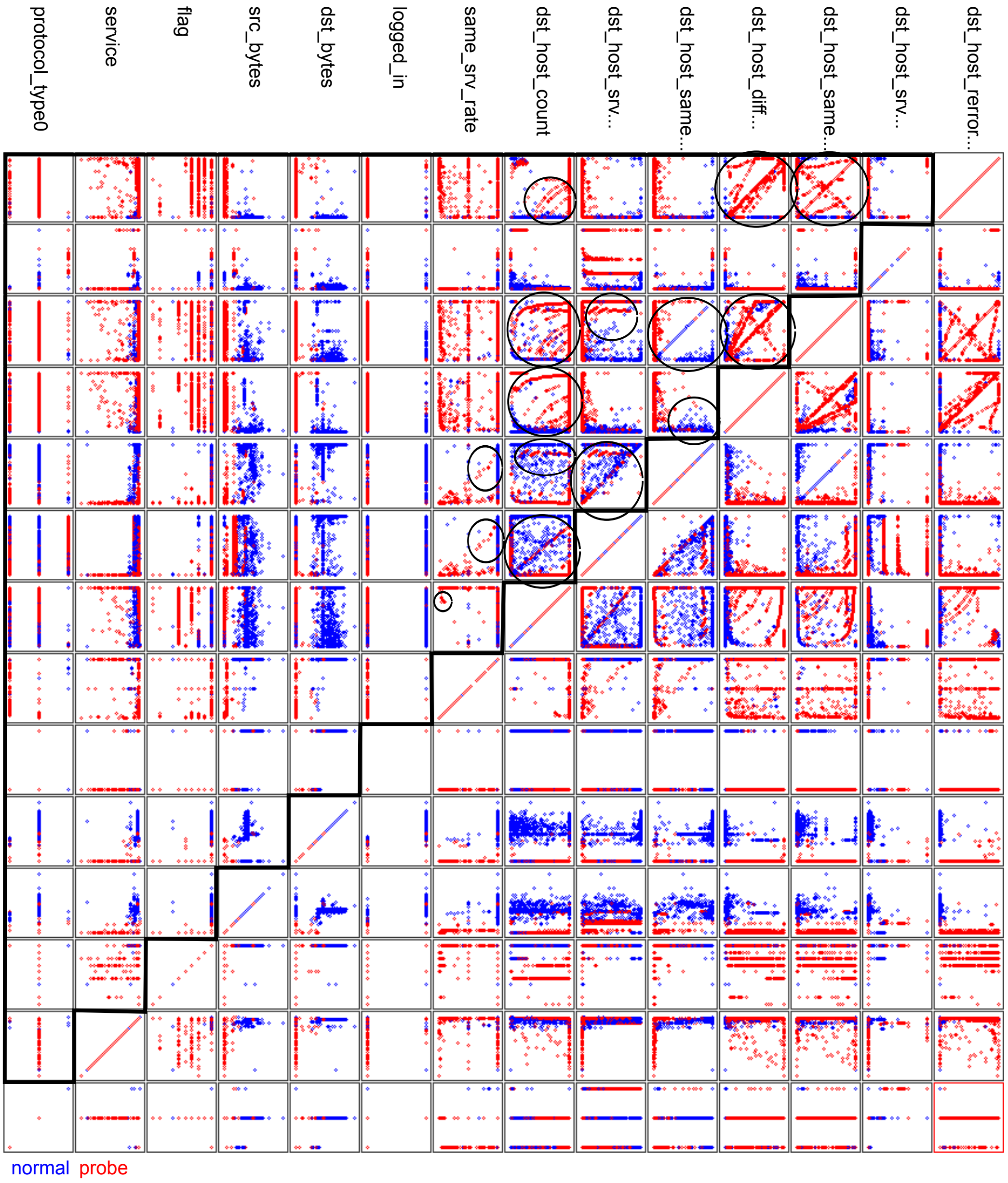


Figure A.5: *The scatter plot matrix of the '14' important features related to network probes in the custom training set with 10,422 instances. The features in the order from left to right and from bottom to top are 'protocol_type', 'service', 'flag', 'src_bytes, dst_bytes', 'logged_in', 'same_srv_rate', 'dst_host_count', 'dst_host_srv_count', 'dst_host_same_srv_rate', 'dst_host_diff_srv_rate', 'dst_host_same_src_port_rate', 'dst_host_srv_diff_host_rate' and 'dst_host_rerror_rate'. The scatter plots between some of the higher layer features shown in the matrix have strong correlations. The interesting areas are highlighted with circle.*

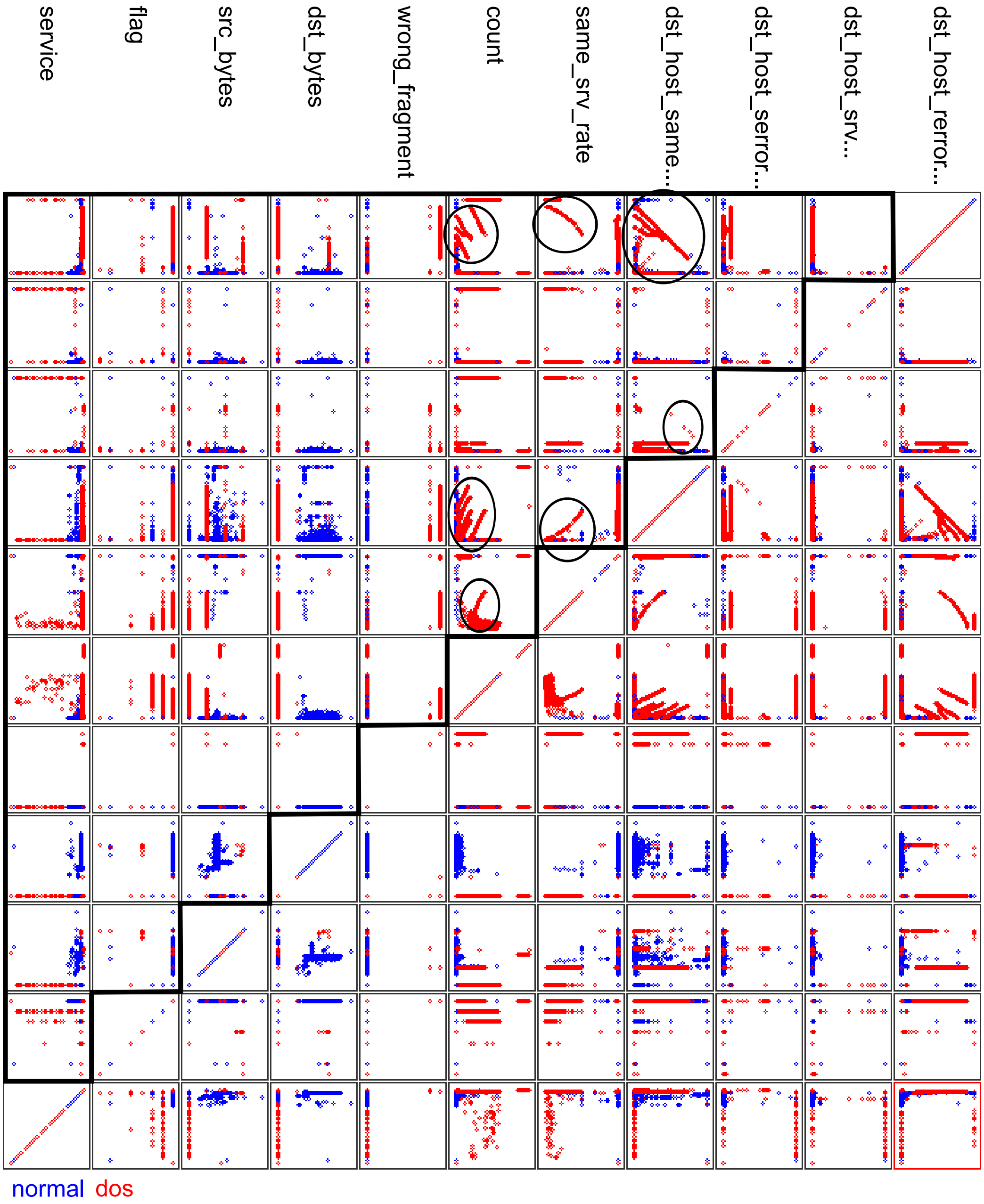


Figure A.6: *Scatter plot matrix of the '11' important features related to 'dos' attacks in the custom training set with 10,422 instances. The features in the order from left to right and from bottom to top are 'service', 'flag', 'src_bytes', 'dst_bytes', 'wrong_fragment', 'count', 'same_srv_rate', 'dst_host_same_src_port_rate', 'dst_host_serror_rate', 'dst_host_srv_serror_rate' and 'dst_host_rerror_rate'. The scatter plots between some of the higher layer features highlighted in the matrix show strong correlations.*

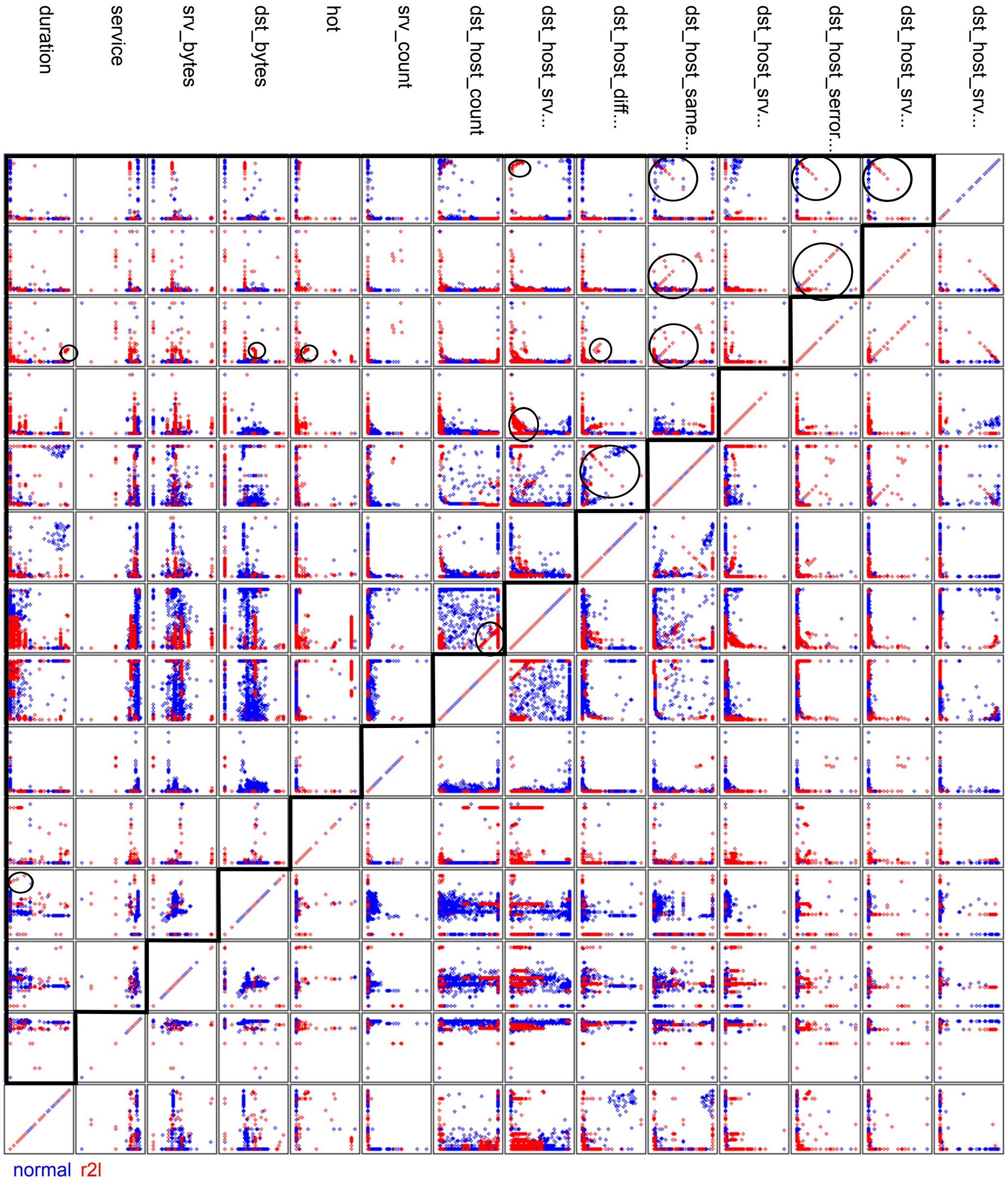


Figure A.7: *Scatter plot matrix of the '14' important features related to 'r2l' attacks in the custom training set with 10,422 instances. The features in the order from left to right and from bottom to top are 'duration', 'service', 'src_bytes', 'dst_bytes', 'hot', 'srv_count', 'dst_host_count', 'dst_host_srv_count', 'dst_host_diff_srv_rate', 'dst_host_same_src_port_rate', 'dst_host_srv_diff_host_rate', 'dst_host_serror_rate', 'dst_host_srv_serror_rate' and 'dst_host_srv_rerror_rate'. A number of feature pairs highlighted in the matrix show weak correlations.*

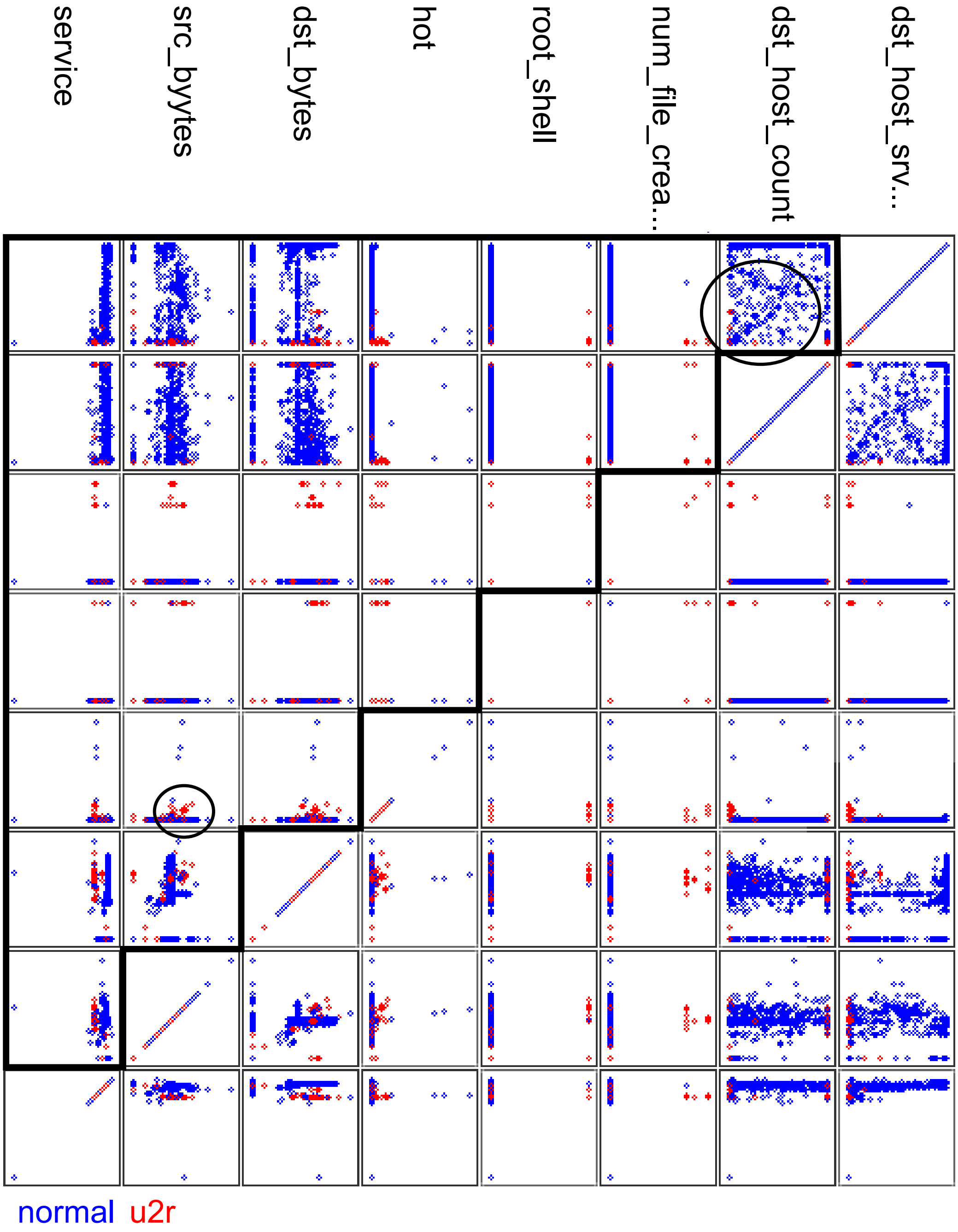


Figure A.8: *Scatter plot matrix of the '8' important features related to 'u2r' attacks in the custom training set with 10,422 instances. The features in the order from left to right and from bottom to top are 'service', 'srv_bytes', 'dst_bytes', 'hot', 'root_shell', 'num_file_creations', 'dst_host_count' and 'dst_host_srv_-count'. There are no salient correlations. The correlations in the highlighted areas are either related to the normal traffic class or are not very strong.*

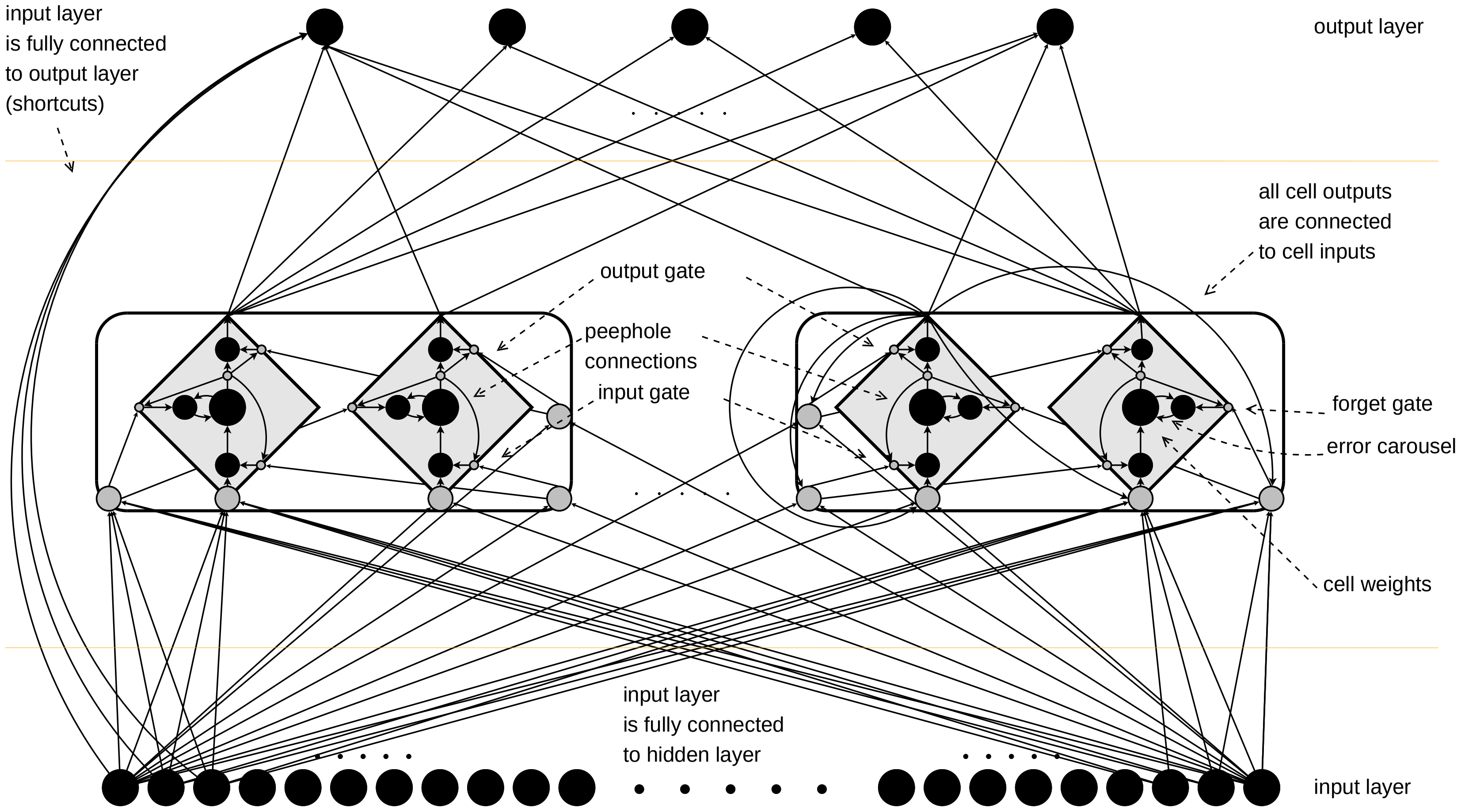


Figure A.9: *LSTM neural network with two memory blocks containing two cells each. The input layer is fully connected to the hidden and output layers. This network has* peephole connections *and* shortcuts*. For reasons of clarity, not all connections are shown.*

Bibliography

[Abraham & Grosan 2006] A. Abraham and C. Grosan. *Evolving intrusion detection systems.* In Genetic Systems Programming, volume 13 of *Studies in Computational Intelligence*, pages 57–79. Springer Berlin / Heidelberg, 2006. p. 28, 123

[Agarwal & Joshi 2000] R. Agarwal and M.V. Joshi. *PNrule: A new framework for learning classier models in data mining.* technical report 00-015, Department of Computer Science, University of Minnesota, 2000. p. 27, 120, 121

[Amaldi & Kann 1998] E. Amaldi and V. Kann. *On the approximability of minimizing nonzero variables or unsatisfied relations in linear systems.* Theoretical Computer Science, vol. 209, no. 1-2, pages 237–260, 1998. p. 78

[Anderson *et al.* 1995] D. Anderson, T. Frivold and A. Valdes. *Next-generation intrusion detection expert system (NIDES): A summary.* technical report, SRI International, 1995. p. 22

[Anderson 1980] J.P. Anderson. *Computer security threat monitoring and surveillance.* technical report, James P. Anderson Company, Fort Washington, Pennsylvania, 1980. p. 14, 19

[Anley 2002] C. Anley. *Advanced SQL injection in SQL server applications.* technical report, Next Generation Security Software Ltd, 2002. p. 16

[Axelsson 2000a] S. Axelsson. *The base-rate fallacy and the difficulty of intrusion detection.* ACM Transactions on Information and System Security (TISSEC), vol. 3, no. 3, pages 186–205, 2000. p. 2

[Axelsson 2000b] S. Axelsson. *Intrusion detection systems: A survey and taxonomy.* technical report, Department of Computer Engineering, Chalmers University of Technology, 2000. p. 4, 13, 14

[Bace & Mell 2001] R. Bace and P. Mell. *NIST special publication on intrusion detection systems.* technical report, DTIC Document, 2001. p. 18, 22, 23, 25

[Bejtlich 2004] R. Bejtlich. The tao of network security monitoring: Beyond intrusion detection. Addison-Wesley Professional, 2004. p. 1, 14, 15, 23

[Bejtlich 2006] R. Bejtlich. Extrusion detection: Security monitoring for internal intrusions. Addison-Wesley Professional, 2006. p. 14, 22

[Bishop 2004] M. Bishop. Introduction to computer security. Addison-Wesley Professional, 2004. p. 19

[Bivens *et al.* 2002] A. Bivens, C. Palagiri, R. Smith, B. Szymanski and M. Embrechts. *Network-based intrusion detection using neural networks.* In Proceedings of the Artificial Neural Networks in Engineering Conference (ANNIE), volume 12, pages 579–584. Citeseer, 2002. p. 29, 124

[Boser *et al.* 1992] B.E. Boser, I.M. Guyon and V.N. Vapnik. *A training algorithm for optimal margin classifiers.* In Proceedings of the fifth annual workshop on Computational learning theory, COLT '92, pages 144–152. ACM, 1992. p. 3, 33, 48, 75

[Bro 2011] Bro. *BroIDS.* World Wide Web electronic publication, 2011. http://www.bro-ids.org/. p. 14, 23, 24

[Brugger & Chow 2005] S.T. Brugger and J. Chow. *An assessment of the DARPA IDS evaluation dataset using snort.* technical report CSE-2007-1, Department of Computer Science, University of California, Davis (UCDAVIS), 2005. p. 27, 119

[Cannady 1998] J. Cannady. *Artificial neural networks for misuse detection.* In Proceedings of the 1998 National Information Systems Security Conference (NISSC), pages 443–456. Citeseer, 1998. p. 28, 124

[Chavan *et al.* 2004] S. Chavan, K. Shah, N. Dave, S. Mukherjee, A. Abraham and S. Sanyal. *Adaptive neuro-fuzzy intrusion detection systems.* In Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC), volume 1, pages 70–74. IEEE Computer Society, 2004. p. 26, 28, 95, 96, 123

[Chebrolu *et al.* 2005] S. Chebrolu, A. Abraham and J.P. Thomas. *Feature deduction and ensemble design of intrusion detection systems.* Computers & Security, vol. 24, no. 4, pages 295–307, 2005. p. 26, 95, 96, 128

[Chen *et al.* 2005] Y. Chen, A. Abraham and J. Yang. *Feature selection and intrusion detection using hybrid flexible neural tree.* In Advances in Neural Networks (ISNN), volume 3498 of *Lecture Notes in Computer Science*, pages 439–444. Springer Berlin / Heidelberg, 2005. p. 26, 27, 95, 96

[Chen *et al.* 2006] Y. Chen, Y. Li, X. Cheng and L. Guo. *Survey and taxonomy of feature selection algorithms in intrusion detection system.* In Information Security and Cryptology, volume 4318 of *Lecture Notes in Computer Science*, pages 153–167. Springer Berlin / Heidelberg, 2006. p. 78

[Cheswick *et al.* 2003] W.R. Cheswick, S.M. Bellovin and A.D. Rubin. Firewalls and internet security: Repelling the wily hacker. Addison-Wesley Professional, second edition, 2003. p. 14

[Cortes & Vapnik 1995] C. Cortes and V. Vapnik. *Support-vector networks.* Machine Learning, vol. 20, no. 3, pages 273–297, 1995. p. 3, 33, 48, 76

[Cowan *et al.* 2000] C. Cowan, F. Wagle, C. Pu, S. Beattie and J. Walpole. *Buffer overflows: attacks and defenses for the vulnerability of the decade.* In Proc. DARPA Information Survivability Conf. and Exposition DISCEX'00, volume 2, pages 119–129. IEEE Computer Society, 2000. p. 18

[CVE 2012] CVE. *CVE - Common Vulnerabilities and Exposures.* World Wide Web electronic publication, 2012. https://cve.mitre.org/. p. 18

[DARPA 2011] DARPA. *DARPA Intrusion Detection Evaluation.* World Wide Web electronic publication, 2011. http://www.ll.mit.edu/-mission/communications/ist/corpora/ideval/data/index.html. p. 3, 19, 90

[Dash & Liu 1997] M. Dash and H. Liu. *Feature selection for classification.* Intelligent Data Analysis, vol. 1, no. 1-4, pages 131–156, 1997. p. 78, 87, 88

[Debar *et al.* 1992] H. Debar, M. Becker and D. Siboni. *A neural network component for an intrusion detection system.* In Proceedings of the IEEE Computer Society Symposium on Research in Security and Privacy, pages 240–250. IEEE Computer Society, 1992. p. 28, 124

[Debar *et al.* 1999] H. Debar, M. Dacier and A. Wespi. *Towards a taxonomy of intrusion-detection systems.* Computer Networks, vol. 31, no. 8, pages 805–822, 1999. p. 4, 14

[Debar *et al.* 2000] H. Debar, M. Dacier and A. Wespi. *A revised taxonomy for intrusion-detection systems.* Annals of Telecommunications, vol. 55, no. 7, pages 361–378, 2000. p. 4, 14

[Denning 1987] D.E. Denning. *An Intrusion-Detection Model.* Software Engineering, IEEE Transactions on, no. 2, pages 222–232, 1987. p. 14, 19

[Dreger *et al.* 2006] H. Dreger, A. Feldmann, M. Mai, V. Paxson and R. Sommer. *Dynamic application-layer protocol analysis for network intrusion detection.* In USENIX Security Symposium, 2006. p. 24

[Elkan 2000] C. Elkan. *Results of the KDD'99 classifier learning.* SIGKDD Explorations Newsletter, vol. 1, pages 63–64, 2000. p. 26, 119, 125

[Elman 1990] J.L. Elman. *Finding structure in time.* cognitive science, vol. 14, pages 179–211, 1990. p. 53

[Eskin *et al.* 2002] E. Eskin, A. Arnold, M. Prerau, L. Portnoy and S. Stolfo. Applications of data mining in computer security, chapter A geometric framework for unsupervised anomaly detection, pages 77–101. Kluwer Academic Pub, 2002. p. 29, 124

[Falliere *et al.* 2010] N. Falliere, L. O Murchu and E. Chien. *W32.Stuxnet dossier V1.3.* technical report, Symantec Security Response, 11 2010. p. 1

[Fawcett 2006] T. Fawcett. *An introduction to ROC analysis.* Pattern Recognition Letters, vol. 27, no. 8, pages 861–874, 2006. p. 84

[Garcia-Teodoro *et al.* 2009] P. Garcia-Teodoro, J. Diaz-Verdejo and E. Macia-Fernandez G. andVazquez. *Anomaly-based network intrusion detection: Techniques, systems and challenges.* Computers & Security, vol. 28, no. 1-2, pages 18–28, 2009. p. 2, 4, 25

[Gates & Taylor 2006] C. Gates and C. Taylor. *Challenging the anomaly detection paradigm: a provocative discussion.* In Proceedings of the 2006 workshop on New security paradigms, pages 21–29. ACM, 2006. p. 25

[Gers *et al.* 1999] F.A. Gers, J. Schmidhuber and F. Cummins. *Learning to forget: Continual prediction with LSTM.* technical report IDSIA-01-99, IDSIA, Lugano, Lugano, CH, 1999. p. 3, 33, 63, 68, 72, 76

[Gers *et al.* 2002] F.A. Gers, N. Schraudolph and J. Schmidhuber. *Learning precise timing with LSTM recurrent networks.* Journal of Machine Learning Research, vol. 3, pages 115–143, 2002. p. 3, 33, 63, 74, 76

[Han & Kamber 2006] J. Han and M. Kamber. Data mining: Concepts and techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, second edition, 2006. p. 2, 23, 33, 37, 40, 48, 75, 80

[Hay *et al.* 2008] A. Hay, D. Cid and R. Bray. Ossec host-based intrusion detection guide. Syngress, 2008. p. 14, 22

[Heckerman *et al.* 1995] D. Heckerman, D. Geiger and D.M. Chickering. *Learning bayesian networks: The combination of knowledge and statisticaldata.* In Machine Learning, pages 20–197. Kluwer Academic Publishers, 1995. p. 3, 33, 40, 75

[Hettich & Bay 1999] S. Hettich and S.D. Bay. *KDD Cup 1999 Data, The UCI KDD Archive, Information and Computer Science, University of California, Irvine.* World Wide Web electronic publication, October 1999. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html. p. 3, 93

[Hochreiter & Schmidhuber 1996] S. Hochreiter and J. Schmidhuber. *Long Short-Term Memory.* technical report FKI-207-95, Technische Universität Muenchen, 1996. Version 3.0. p. 3, 33, 63, 76

[Hochreiter & Schmidhuber 1997] S. Hochreiter and J. Schmidhuber. *Long Short-Term Memory.* Neural Computation, vol. 9, no. 8, pages 1735–1780, 1997. p. 3, 33, 63, 76

[Hochreiter *et al.* 2001] S. Hochreiter, Y. Bengio, P. Frasconi and J. Schmidhuber. *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.* In A Field Guide to Dynamical Recurrent Neural Networks. IEEE Press, 2001. p. 33, 63, 76

[Hu & Hu 2005] W. Hu and W. Hu. *Network-based intrusion detection using Adaboost algorithm.* In Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence, WI '05, pages 712–717. IEEE Computer Society, 2005. p. 28, 122

[Jagannathan *et al.* 1993] R. Jagannathan, T. Lunt, D. Anderson, C. Dodd, F. Gilham, C. Jalali, H. Javitz, P. Neumann, A. Tamaru and A. Valdes. *System design document: Next-generation intrusion detection expert system (NIDES).* technical report, SRI International, 1993. p. 14

[John & Langley 1995] G. John and P. Langley. *Estimating continuous distributions in Bayesian classifiers.* In Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence, pages 338–345. Morgan Kaufmann, 1995. p. 3, 33, 38, 75

[Jordan 1986] M.I. Jordan. *Attractor dynamics and parallelism in a connectionist sequentialmachine.* In Proceedings of the Eigth Annual Conference of the Cognitive Science Society, pages 531–546, 1986. p. 53

[Kayacik *et al.* 2005] H.G. Kayacik, A.N. Zincir-Heywood and M.I. Heywood. *Selecting features for intrusion detection: A feature relevance analysison KDD 99 intrusion detection datasets.* In Proceedings of the Third Annual Conference on Privacy, Security and Trust (PST), 2005. p. 26, 27, 95, 96

[Kayacik *et al.* 2007] H.G. Kayacik, A.N. Zincir-Heywood and M.I. Heywood. *A hierarchical SOM-based intrusion detection system.* Engineering Applications of Artificial Intelligence, vol. 20, no. 4, pages 439–451, 2007. p. 28, 123

[Keerthi *et al.* 2001] S.S. Keerthi, S.K. Shevade, C. Bhattacharyya and K.R.K. Murthy. *Improvements to Platt's SMO algorithm for SVM classifier design.* Neural Computation, vol. 13, no. 3, pages 637–649, 2001. p. 48

[Kemmerer & Vigna 2002] R.A. Kemmerer and G. Vigna. *Intrusion detection: a brief history and overview.* IEEE Computer, vol. 35, no. 4, pages 27–30, 2002. p. 14

[Kim & Karp 2004] H. Kim and B. Karp. *Autograph: Toward Automated, Distributed Worm Signature Detection.* In USENIX Security Symposium, pages 271–286. USENIX, 2004. p. 23

[Kim & Spafford 1994] G.H. Kim and E.H. Spafford. *The design and implementation of tripwire: A file system integrity checker.* In Proceedings of the 2nd ACM Conference on Computer and CommunicationsSecurity, pages 18–29. ACM, 1994. p. 14, 22

[Kohavi & John 1997] R. Kohavi and G.H. John. *Wrappers for feature subset selection.* Artificial Intelligence, vol. 97, no. 1-2, pages 273–324, 1997. p. 78

[Kruegel *et al.* 2003] C. Kruegel, D. Mutz, W. Robertson and F. Valeur. *Bayesian event classification for intrusion detection.* In Proceedings of the 19th Annual Computer Security Applications Conference(ACSAC), pages 14–23, 2003. p. 29, 124

[Kumar 1995] S. Kumar. *Classification and detection of computer intrusions.* PhD thesis, Purdue University, 1995. p. 1, 23

[Lakhina *et al.* 2005] A. Lakhina, M. Crovella and C. Diot. *Mining anomalies using traffic feature distributions.* In Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications, pages 217–228. ACM, 2005. p. 25

[Laskov *et al.* 2005] P. Laskov, P. Dussel, C. Schafer and K. Rieck. *Learning intrusion detection: Supervised or unsupervised?* In Image Analysis and Processing (ICIAP), volume 3617 of *Lecture Notes in Computer Science*, pages 50–57. Springer Berlin / Heidelberg, 2005. p. 29, 124

[Lee & Stolfo 2000] W. Lee and S.J. Stolfo. *A framework for constructing features and models for intrusion detectionsystems.* Transactions on Information and System Security (TISSEC), vol. 3, no. 4, pages 227–261, 2000. p. 2, 3, 26, 93, 97

[Lee *et al.* 2002] W. Lee, W. Fan, W.M. Miller, S.J. Stolfo and E. Zadok. *Toward cost-sensitive modeling for intrusion detection and response.* Journal of Computer Security, vol. 10, no. 1/2, pages 5–22, 2002. p. 2, 3

[Lee *et al.* 2006] C.H. Lee, S.W. Shin and J.W. Chung. *Network intrusion detection through genetic feature selection.* In Seventh ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD), pages 109–114. IEEE Computer Society, 2006. p. 26, 27, 95, 96

[Lee 1999] W. Lee. *A data mining framework for constructing features and models for intrusion detection systems.* PhD thesis, USA: Columbia University, 1999. p. 26, 93, 97

[Levin 2000] I. Levin. *KDD-99 classifier learning contest LLSoft's results overview.* SIGKDD Explorations Newsletter, vol. 1, pages 67–75, 2000. p. 27, 119, 120

[Liao & Vemuri 2006] Y Liao and V.R. Vemuri. Enhancing computer security with smart technology, chapter Machine learning in intrusion detection, pages 93–124. Auerbach Publications, 2006. p. 2

[Lippmann *et al.* 2000a] R. Lippmann, J.W. Haines, D.J. Fried, J. Korba and K. Das. *The 1999 DARPA off-line intrusion detection evaluation.* Computer Networks, vol. 34, no. 4, pages 579–595, 2000. p. 19, 25, 90

[Lippmann *et al.* 2000b] R.P. Lippmann, D.J. Fried, I. Graf, J.W. Haines, K.R. Kendall, D. McClung, D. Weber, S.E. Webster, D. Wyschogrod, R.K. Cunningham*et al.* *Evaluating intrusion detection systems: The*

1998 DARPA off-line intrusion detection evaluation. In DARPA Information Survivability Conference and Exposition, 2000.DISCEX'00. Proceedings, volume 2, pages 12–26. IEEE, 2000. p. 19, 25, 90

[Lunt *et al.* 1992] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalali, H.S. Javitz, A. Valdes, P.G. Neumann and T.D. Garvey. *A real-time intrusion-detection expert system (IDES).* technical report, SRI International, 1992. p. 22

[Lunt 1988] Teresa F. Lunt. *Automated Audit Trail Analysis and Intrusion Detection: A Survey.* In Proceedings of the 11th National Computer Security Conference, pages 65–73, 1988. p. 1, 14

[Lunt 1993] T.F. Lunt. *A survey of intrusion detection techniques.* Computers & Security, vol. 12, no. 4, pages 405–418, 1993. p. 1, 14

[Lyon 2009] Gordon Fyodor Lyon. Nmap network scanning: The official nmap project guide to network discovery and security scanning. Insecure, USA, 2009. p. 15, 20

[Mahoney & Chan 2003] M.V. Mahoney and P.K. Chan. *An analysis of the 1999 DARPA/Lincoln Laboratory evaluation data for network anomaly detection.* In Recent Advances in Intrusion Detection, volume 2820 of *Lecture Notes in Computer Science*, pages 220–237. Springer Berlin / Heidelberg, 2003. p. 27, 118

[Maloof 2006] M.A. Maloof. *Some basic concepts of machine learning and data mining.* In Machine Learning and Data Mining for Computer Security, Advanced Information and Knowledge Processing, pages 23–43. Springer London, 2006. p. 80

[Maynor & Mookhey 2007] D. Maynor and K.K. Mookhey. Metasploit toolkit for penetration testing, exploit development, and vulnerability research. Syngress Media Inc, 2007. p. 16, 21

[McHugh 2000] J. McHugh. *Testing intrusion detection systems: A critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory.* ACM Transactions on Information and System Security, vol. 3, no. 4, pages 262–294, 2000. p. 27, 118

[McHugh 2001] J. McHugh. *Intrusion and intrusion detection.* International Journal of Information Security, vol. 1, no. 1, pages 14–35, 2001. p. 1, 14

[Metasploit 2011] Metasploit. *Metasploit.* World Wide Web electronic publication, 2011. http://www.metasploit.com/. p. 16, 21

[Metz *et al.* 2009] C.E. Metz, Y. Jiang, H. MacMahon, R.M. Nishikawa and X. Ran. *ROC software*. World Wide Web electronic publication, March 2009. http://www-radiology.uchicago.edu/krl/. p. 157

[Minsky & Papert 1969] M. Minsky and S. Papert. *Perceptrons*. MIT Press, Cambridge, 1969. p. 42

[Mirkovic *et al.* 2005] J. Mirkovic, S. Dietrich and P. Reiher. Internet denial of service: attack and defense mechanisms. Prentice Hall, 2005. p. 20

[Mitchell 1997] T. Mitchell. Machine learning. McGraw Hill, 1997. p. 2, 33, 34, 37, 40, 75

[Moore *et al.* 2003] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford and N. Weaver. *Inside the slammer worm*. Security & Privacy, IEEE, vol. 1, no. 4, pages 33–39, 2003. p. 20

[Mukherjee *et al.* 1994] B. Mukherjee, L.T. Heberlein and K.N. Levitt. *Network intrusion detection*. Network, IEEE, vol. 8, no. 3, pages 26–41, 1994. p. 1, 14, 23

[Mukkamala *et al.* 2003] S. Mukkamala, A.H. Sung and A. Abraham. Intelligent systems design and applications, chapter Intrusion Detection Using Ensemble of Soft Computing Paradigms, pages 239–248. Springer New York, 2003. p. 28, 123

[Mukkamala *et al.* 2004] S. Mukkamala, A.H. Sung and A. Abraham. *Modeling intrusion detection systems using linear genetic programming approach*. In Innovations in Applied Artificial Intelligence, volume 3029 of *Lecture Notes in Computer Science*, pages 633–642. Springer Berlin / Heidelberg, 2004. p. 28, 123

[Nessus 2011] Nessus. *Nessus*. World Wide Web electronic publication, 2011. http://nessus.org/. p. 15

[nmap 2011] nmap. *nmap*. World Wide Web electronic publication, 2011. http://http://nmap.org/. p. 15, 20

[Northcutt & Novak 2003] S. Northcutt and J. Novak. Network intrusion detection. New Riders Publishing Thousand Oaks, third edition, 2003. p. 14

[Northcutt *et al.* 2005] S. Northcutt, L. Zeltser, S. Winters, K. Kent and R.W. Ritchey. Inside network perimeter security. Sams Indianapolis, IN, USA, second edition, 2005. p. 14

[OpenVAS 2011] OpenVAS. *OpenVAS.* World Wide Web electronic publication, 2011. http://www.openvas.org/. p. 15, 20

[OSSEC 2011] OSSEC. *OSSEC.* World Wide Web electronic publication, 2011. http://www.ossec.net/. p. 14, 22

[Ourston *et al.* 2003] D. Ourston, S. Matzner, W. Stump and B. Hopkins. *Applications of hidden markov models to detecting multi-stage network attacks.* In Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS), pages 10–15, 2003. p. 28, 124

[Paxson 1999] V. Paxson. *Bro: A system for detecting network intruders in real-time.* Computer Networks, vol. 31, no. 23, pages 2435–2463, 1999. p. 2, 14, 23, 24, 93

[Peddabachigari *et al.* 2007] S. Peddabachigari, A. Abraham, C. Grosan and J. Thomas. *Modeling intrusion detection system using hybrid intelligent systems.* Journal of network and computer applications, vol. 30, no. 1, pages 114–132, 2007. p. 28, 123

[Pesce & Metz 2007] L.L. Pesce and C.E. Metz. *Reliable and computationally efficient maximum-likelihood estimation of "proper" binormal ROC curves.* Academic radiology, vol. 14, no. 7, pages 814–829, 2007. p. 157

[Pfahringer 2000] B. Pfahringer. *Winning the KDD99 classification cup: Bagged boosting.* SIGKDD Explorations Newsletter, vol. 1, pages 65–66, 2000. p. 27, 119, 120

[Platt 1999] J.C. Platt. Fast training of support vector machines using sequential minimal optimization. MIT Press, Cambridge, MA, USA, 1999. p. 48

[Porras & Neumann 1997] P.A. Porras and P.G. Neumann. *EMERALD: Event monitoring enabling responses to anomalous live disturbances.* In Proceedings of the 20th National Information Systems Security Conference, pages 353–365. Citeseer, 1997. p. 14

[Prelude 2011] Prelude. *PreludeIDS.* World Wide Web electronic publication, 2011. http://prelude-ids.org/. p. 2, 14, 30

[Provost *et al.* 1998] F. Provost, T. Fawcett and R. Kohavi. *The case against accuracy estimation for comparing induction algorithms.* In Proceedings of the Fifteenth International Conference on Machine Learning, pages 445–453, San Francisco, CA, 1998. Morgan Kaufmann. p. 86

[Quinlan 1986] J.R. Quinlan. *Induction of decision trees.* Machine Learning, vol. 1, no. 1, pages 81–106, 1986. p. 3, 33, 34, 75

[Quinlan 1993] J.R. Quinlan. C4.5: programs for machine learning. Morgan Kaufmann Publishers Inc., 1993. p. 3, 33, 34, 75, 125, 207

[Riancho 2011] A. Riancho. *w3af - Web Application Attack and Audit Framework.* World Wide Web electronic publication, 2011. http://www.ll.mit.edu/mission/communications/ist/corpora/-ideval/data/index.html. p. 21

[Roesch 1999] M. Roesch. *Snort–lightweight intrusion detection for networks.* In Proceedings of the 13th USENIX conference on System administration, pages 229–238, Seattle, Washington, 1999. p. 14, 23, 24, 25, 119

[Rumelhart *et al.* 1986] D.E. Rumelhart, G.E. Hinton and Williams R.J. *Learning internal representations by error propagation.* In J.L. McClelland and D.E. Rumelhart, editeurs, Parallel distributed processing: explorations in the microstructure of cognition, volume 1, pages 318–362. MIT Press, 1986. p. 33, 59, 76

[Rumelhart *et al.* 1994] D.E. Rumelhart, B. Widrow and M. Lehr. *The basic ideas in neural networks.* Communications of the ACM, vol. 37, no. 3, pages 87–92, 1994. p. 3, 33, 40, 75

[Sabhnani & Serpen 2003] M. Sabhnani and G. Serpen. *Application of machine learning algorithms to KDD intrusion detection dataset within misuse detection context.* In International Conference on Machine Learning, Models, Technologies and Applications (MLMTA), pages 209–215. CSREA Press, 2003. p. 27, 121, 122

[Sabhnani & Serpen 2004] M. Sabhnani and G. Serpen. *Why machine learning algorithms fail in misuse detection on KDD intrusion detection data set.* Intelligent Data Analysis, vol. 8, no. 4, pages 403–415, 2004. p. 27, 118

[Samhain 2011] Samhain. *Samhain.* World Wide Web electronic publication, 2011. http://www.la-samhna.de/. p. 14, 22

[Scarfone & Mell 2007] K. Scarfone and P. Mell. *Guide to intrusion detection and prevention systems (IDPS).* technical report 2007, National Institute of Standards and Technology (NIST), 2007. p. 1, 4, 21, 22, 23, 24, 25, 29

[Schuba & Spafford 1994] C.L. Schuba and E.H. Spafford. *Countering abuse of name-based authentication.* technical report, Purdue University, 1994. p. 17

[Shields 2006] C. Shields. *An Introduction to Information Assurance.* In Machine Learning and Data Mining for Computer Security, Advanced Information and Knowledge Processing, pages 7–21. Springer London, 2006. p. 14

[Sinclair *et al.* 1999] C. Sinclair, L. Pierce and S. Matzner. *An application of machine learning to network intrusion detection.* In Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC), pages 371–377. IEEE Computer Society, 1999. p. 28, 123

[Smaha 1988] S.E. Smaha. *Haystack: An intrusion detection system.* In Aerospace Computer Security Applications Conference, 1988., Fourth, pages 37–44. IEEE, 1988. p. 14, 19, 22

[Snapp *et al.* 1991] S.R. Snapp, J. Brentano, G.V. Dias, T.L. Goan, L.T. Heberlein, C.L. Ho, K.N. Levitt, B. Mukherjee, S.E. Smaha, T. Granceet *al.* *DIDS (distributed intrusion detection system)-motivation, architecture, and an early prototype.* In Proceedings of the 14th National Computer Security Conference. Citeseer, 1991. p. 22

[Snort 2011] Snort. *Snort.* World Wide Web electronic publication, 2011. http://www.snort.org/. p. 14, 23, 24, 25

[Song *et al.* 2005] D. Song, M.I. Heywood and A.N. Zincir-Heywood. *Training genetic programming on half a million patterns: An example from anomaly detection.* IEEE Transactions on Evolutionary Computation, vol. 9, no. 3, pages 225–239, 2005. p. 28, 122

[Song *et al.* 2007] Y. Song, M.E. Locasto, A. Stavrou, A.D. Keromytis and S.J. Stolfo. *On the infeasibility of modeling polymorphic shellcode.* In Proceedings of the 14th ACM conference on Computer and communications security, pages 541–551. ACM, 2007. p. 23

[Spinellis 2003] D. Spinellis. *Reliable identification of bounded-length viruses is NP-complete.* Information Theory, IEEE Transactions on, vol. 49, no. 1, pages 280–284, 2003. p. 24

[Stallings 2006] W. Stallings. Cryptography and network security: Principles and practice, chapter Intruders: Intrusion Detection. Prentice Hall, fourth edition, 2006. p. 14

[Staniford *et al.* 2002] S. Staniford, J.A. Hoagland and J.M. McAlerney. *Practical automated detection of stealthy portscans.* Journal of Computer Security, vol. 10, no. 1/2, pages 105–136, 2002. p. 2, 25

[Staudemeyer & Omlin 2009] R. Staudemeyer and C.W. Omlin. *Feature set reduction for automatic network intrusion detection with machine learning algorithms.* In Proceedings of the Southern African Telecommunication Networks and Applications Conference (SATNAC), 2009. p. 105

[Sung 2003] S. Sung A.H. Mukkamala. *Identifying important features for intrusion detection using support vector machines and neural networks.* In Proceedings of the Symposium on Applications and the Internet (SAINT), pages 209–216. IEEE Computer Society, 2003. p. 26, 95, 96

[Tripunitara & Dutta 1999] M.V. Tripunitara and P. Dutta. *A middleware approach to asynchronous and backward compatible detection and prevention of ARP cache poisoning.* In Proc. 15th Annual Computer Security Applications Conf. (ACSAC '99), pages 303–309. IEEE Computer Society, 1999. p. 17

[Tripwire 2011] Tripwire. *TripwireIDS.* World Wide Web electronic publication, 2011. http://tripwire.sf.net/. p. 14, 22

[Vladimir *et al.* 2000] M. Vladimir, V. Alexei and S. Ivan. *The MP13 approach to the KDD'99 classifier learning contest.* SIGKDD Explorations Newsletter, vol. 1, pages 76–77, 2000. p. 27, 120

[Wang & Stolfo 2004] K. Wang and S.J. Stolfo. *Anomalous payload-based network intrusion detection.* In Recent Advances in Intrusion Detection, pages 203–222. Springer, 2004. p. 25

[Werbos 1990] P.J. Werbos. *Backpropagation through time: what it does and how to do it.* Proceedings of the IEEE, vol. 78, no. 10, pages 1550–1560, 1990. p. 3, 59

[Williams & Zipser 1989] R.J. Williams and D. Zipser. *A learning algorithm for continually running fully recurrent neural networks.* Neural Computation, vol. 1, pages 270–280, 1989. p. 33, 61, 76

[Williams & Zipser 1995] R.J. Williams and D. Zipser. Backpropagation: Theory, architectures, and applications, chapter Gradient-Based Learning Algorithms for Recurrent Networks and Their Computational Complexity, pages 433–486. Lawrence Erlbaum, 1995. p. 33, 59, 61, 76

[Witten & Frank 2005] I.H. Witten and E. Frank. Weka–machine learning algorithms in java, chapter The Weka Machine Learning Workbench, pages 363–484. Morgan Kaufmann, 2005. p. 99

[Wotring *et al.* 2005] B. Wotring, B. Potter and M.J. Ranum. Host integrity monitoring using osiris and samhain. Syngress Media Inc, 2005. p. 14, 22

[Yeung & Chow 2002] D. Yeung and C. Chow. *Parzen-window network intrusion detectors*. In Proceedings of the 16th International Conference on International Conference on Pattern Recognition, volume 4, pages 385–388, 2002. p. 28, 123

[Zhang *et al.* 2001] Z. Zhang, J. Lee, C. Manikopoulos, J. Jorgenson and J. Ucles. *Neural networks in statistical anomaly intrusion detection*. Neural Network World, vol. 11, no. 3, pages 305–316, 2001. p. 28, 124

List of Abbreviations

ACC	accuracy. p. 83
ARP	address resolution protocol. p. 17
AUC	area under curve. p. 85
BayesNet	Bayesian network. p. 38
BPTT	backpropagation through time. p. 55
C4.5	decision tree learning algorithm introduced by [Quinlan 1993]. p. 34
CEC	constant error carousel. p. 63
CPU	central processing unit. p. 16
CVE	common information security vulnerabilities and exposures. p. 18
DARPA	Defense Advanced Research Projects Agency. p. 90
DDOS	distributed denial-of-service. p. 20
DNS	domain name system. p. 17
DOS	denial-of-service. p. 20
dos	denial-of-service attack category. p. 91
DR	detect(ion) rate. p. 83
FAR	false alarm rate. p. 83
FFNN	feed-forward neural network. p. 44
FNR	false negative rate. p. 83
FPR	false positive rate. p. 83
HIDS	host intrusion detection system. p. 21
ICT	information and communication technology. p. 1
IDS	intrusion detection system. p. 13
IP	internet protocol. p. 15

IPS intrusion prevention system. p. 29

IRC internet relay chat. p. 16

J4.8 WEKA's implementation of C4.5 algorithm. p. 125

KDD Cup Annual Knowledge Discovery and Data Mining competition. p. 93

LSTM long short-term memory. p. 63

MADAMID mining audit data for automated models for intrusion detection. p. 26

MAUC multi-class AUC. p. 86

MLP WEKA's implementation of a FFNN. p. 99

MSE mean squared error. p. 83

nBayes naïve Bayes. p. 37

NIDS network intrusion detection system. p. 21

probe network probe attack category. p. 91

r2l remote-to-local attack category. p. 91

RNN recurrent neural networks. p. 53

ROC receiver operating characteristic. p. 84

RTRL real-time recurrent learning. p. 55

SQL structured query language. p. 16

SVM support vector machine. p. 48

TCP transmission control protocol. p. 15

TNR true negative rate. p. 83

TPR true positive rate. p. 83

u2r user-to-root attack category. p. 91

UDP user datagram protocol. p. 15

WEKA Waikato Environment for Knowledge Analysis. p. 125