A Framework for High Speed Lexical Classification of Malicious URLs

Submitted in fulfilment
of the requirements of the degree of

Master of Science

of Rhodes University

Shaun Peter Egan

Grahamstown, South Africa
April 2014

Abstract

Phishing attacks employ social engineering to target end-users, with the goal of stealing identifying or sensitive information. This information is used in activities such as identity theft or financial fraud. During a phishing campaign, attackers distribute URLs which; along with false information, point to fraudulent resources in an attempt to deceive users into requesting the resource. These URLs are made obscure through the use of several techniques which make automated detection difficult. Current methods used to detect malicious URLs face multiple problems which attackers use to their advantage. These problems include: the time required to react to new attacks; shifts in trends in URL obfuscation and usability problems caused by the latency incurred by the lookups required by these approaches. A new method of identifying malicious URLs using Artificial Neural Networks (ANNs) has been shown to be effective by several authors. The simple method of classification performed by ANNs result in very high classification speeds with little impact on usability. Samples used for the training, validation and testing of these ANNs are gathered from Phishtank and Open Directory. Words selected from the different sections of the samples are used to create a 'Bag-of-Words (BOW)' which is used as a binary input vector indicating the presence of a word for a given sample. Twenty additional features which measure lexical attributes of the sample are used to increase classification accuracy. A framework that is capable of generating these classifiers in an automated fashion is implemented. These classifiers are automatically stored on a remote update distribution service which has been built to supply updates to classifier implementations. An example browser plugin is created and uses ANNs provided by this service. It is both capable of classifying URLs requested by a user in real time and is able to block these requests. The framework is tested in terms of training time and classification accuracy. Classification speed and the effectiveness of compression algorithms on the data required to distribute updates is tested. It is concluded that it is possible to generate these ANNs in a frequent fashion, and in a method that is small enough to distribute easily. It is also shown that classifications are made at high-speed with high-accuracy, resulting in little impact on usability.

Acknowledgements

This thesis has taken two years to produce with the support of many people. I would like to thank all those people who have contributed and assisted me in any way to make this research a success.

I would like to thank a few people specifically; my family, friends and loved ones for all of the support, guidance and funding that they have provided during this time. This opportunity would not have been possible without them and for that I am especially grateful.

I would like to thank my supervisor, Prof. Barry Irwin for all of his hard work in obtaining data and his highly valued input, guidance and support throughout this research.

This work has been completed within the Centre of Excellence in Distributed Multimedia at Rhodes University. I acknowledge the financial and technical support of this project by Telkom SA, Comverse, Verso Technologies, Tellabs, StorTech, EastTel and THRIP. I would like to thank the Defence, Peace, Safety and Security pillar of the Council for Scientific and Industrial Research for the financial assistance that was provided, without which this thesis would not have been possible.

ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System (1998 version, valid through 2013) **D.3.3** [Language Constructs and Features] Frameworks
C.2.0 [General] Security and Protection
C.2.3 [Network Operations] Network Monitoring
I.2.6 [Learning] Connectionism and Neural Networks
I.2.7 [Natural Language Processing] Text Analysis
General-Terms: URL, Phishing, Artificial Neural Network, Perceptron

To my Emma, my Angel, you will never be forgotten. I love you.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AJAX Asynchronous JavaScript and XML

APWG Anti Phishing Working Group

AROW Adaptive Regularization of Weights

ANN Artificial Neural Network

API Application Program Interface

BOW Bag-of-Words

CDS Classifier Distribution Service

CGS Classifier Generation Service

CPU Central Processing Unit

CSV Comma Separated Values

CW Confidence Weighted

DLL Dynamic Link Library

DNS Domain Name System

DNSBL Domain Name System Blackhole List

FDR False Discovery Rate

FN False Negative

FP False Positive

FPR False Positive Rate

FTP File Transfer Protocol

GB Gigabytes

GC Garbage Collection

GPU Graphics Processing Unit

GUI Graphic User Interface

HTML HyperText Markup Language

HTTP Hyper Text Transfer Protocol

HTTPS Hyper Text Transfer Protocol Secure

IETF Internet Engineering Task Force

IO Input and Output

KB Kilobytes

LM Linear Model

LOOCV Leave One Out Cross-Validation

IP Internet Protocol

JSON JavaScript Object Notation

MB Megabytes

MCC Matthews Correlation Coefficient

MHz Mega Hertz

MRS Microsoft Reputation Services

MVC Model-View-Controller

NPV Negative Prediction Value

OP Online Perceptron

OS Operating System

PPV Positive Prediction Value

RAM Random Access Memory

RBL Real-time Blackhole List

REGEX Regular Expression

REST Representational State Transfer

RSS Really Simple Syndication

TCP Transmission Control Protocol

TN True Negative

TNR True Negative Rate

TP True Positive

TPR True Positive Rate

URI Uniform Resource Identifier

URL Uniform Resource Locator

VM Virtual Machine

XML Extensible Markup Language

CHAPTER

ONE

INTRODUCTION

Phishing attacks and email spam have become a common part of the internet and email usage. They are a form of social engineering; whereby, attackers try to mask malicious resources as legitimate ones; deceiving users into entering sensitive and identifying information (Hong, 2012). This information is often used in illegal activities; ranging from identity theft to financial fraud through stolen banking and credit card related details (Dhamija and Tygar, 2005). These attacks occur in several forms, the most notable being a wide attack which targets any users that are reachable through means such as email, to very specific targeted attacks, known as spear phishing, where individuals are singled out. Detection of phishing attacks is a non-trivial task as the attacks try to mimic legitimate resources. While it is possible to detect these attacks using Bayesian networks (Androutsopoulos, Koutsias, V. Chandrinos, Paliouras, and D. Spyropoulos, 2000) and host identification, these methods can be unreliable and often induce latency when a blacklist or another non-local resource is required. Another difficulty is that attackers will often try to actively obfuscate the URL in an effort to avoid detection and to mask the intent of a resource (Garera, Provos, Chew, and Rubin, 2007).

However, expert users are known to be able to 'see' phishing URLs through their lexical elements alone, as a direct result of obfuscation techniques used to avoid detection (Ma, Saul, Savage, and Voelker, 2009a). It has been shown by several authors that it is possible to detect phishing URLs through their lexical elements using ANNs (Ma *et al.*, 2009a; Le, Markopoulou, and Faloutsos, 2011). Additionally, these classifiers are able to predict

the nature of a resource to an accuracy that is as high as that of traditional classification techniques. Within this document it is shown that it is possible to generate these classifiers in a production environment to an accuracy that is reported by these authors, as well as how feasible they are to use as end-user solutions. A framework is built that allows for the generation of these ANNs as well as a distribution mechanism. Finally, an example client implementation is shown.

1.1 Motivation

As shown in Aaron and Rasmussen (2012) and Aaron and Rasmussen (2013), the number of unique phishing attacks to occur have increased during the period starting from the second half of 2010, to the first half of 2013. This number has increased from 67 677 reported incidents to 72 758. These reports indicate the large volume of phishing attacks that occur every month; with a total of 488 251 recorded attacks from the first half of 2011 to the same time in 2013. According to a survey conducted in Litan (2004); at least 1.78 million people had been victims of phishing attacks by 2004 in America alone. Phishing, in general; consists of several different categories, all of which entail an attacker sending a message containing a URL, often by email, to a website that is intended to mislead the user into thinking that it originates from a legitimate source (Dhamija, Tygar, and Hearst, 2006; Garera, Provos, Chew, and Rubin, 2007). The goal of this is to deceive the user into entering private data such as credit card details; which are then stolen (Hong, 2012). The complexity of these attacks can range from simple messages to high-fidelity websites; mimicking several functions of the targeted resource (Wu, Miller, and Garfinkel, 2006).

There are several methods of identifying phishing URLs currently in widespread use, including; black lists, spam filters, reputation services and content scanners. These methods have two main problems: firstly; since blacklists often require humans to check the intention of a resource; they cannot adapt quickly, often only being updated and useful after a phishing campaign has ended. Secondly; the end-user has to execute lookups from these services, which incur latency. This additional latency slows the end-user's experience, which impacts usability negatively (Le *et al.*, 2011). Spam filters which use heuristics and other detection methods, such as Bayesian networks; are often deceived through the use of various obfuscation methods employed by attackers executing phishing techniques. Reputation services, like blacklists, require lookups which cause the user to experience further latency when requesting resources. In-depth content scanners can be considered

as flawed in that they require a resource to be downloaded before they can be classified. This allows for the possibility of an exploit being executed against the browser before such a classification is made (Ma *et al.*, 2009a) as the browser has already been exposed to the content.

The drawbacks in the current methods of identifying malicious URLs can be summarized as follows: they cannot adapt fast enough when new phishing attacks occur; they incur overheads in the form of latency which is compounded when multiple resources are requested in typical scenarios and in the case of content scanners; are vulnerable to attack themselves. Additionally, obfuscation techniques employed by attacks today make phishing attacks difficult to identify by any method currently widely in use. The requirement for human input in the case of blacklists not only makes reaction time within these systems slow, but it also introduces the possibility of human error; especially when noting that phishing sites have been known to show different contents to users from known security organizations (Ma *et al.*, 2009a).

1.2 Research Objectives

The primary focus of this research is to develop a framework for producing ANNs as classifiers for use in the identification of phishing URLs through their lexical features. These should ideally be viable for use in real-world applications. While authors have shown that this approach is applicable to all categories of malicious resources available on the internet (Le *et al.*, 2011), the specific focus of this document and research is to identify phishing attacks. However, detecting other malicious URLs, such as malware downloads; is achievable by using an appropriate data source without changing the framework or training methods described in this document. Formally, the research objectives are as follows:

- Develop a framework which can generate ANNs in a fashion that is usable in client applications. These ANNs must be fast enough to incur little overhead in terms of latency and be as accurate as has been shown in previous research. The process of data gathering, ANN training and distribution must be automated to rule out human error.
- Determine the impact that increasing numbers of training samples has on training time. Additionally, examine the accuracy improvements made when using increasing numbers of training samples.

- Determine what data is required for clients to be able to download and install an ANN. Calculate the size of the resulting data set and determine what is the best compression algorithm to use for this type of data.
- Develop a method of storing and distributing these classifiers that is automated, reliable and feasible in terms of bandwidth and storage. Additionally, this service must be implementation-agnostic, so that other implementations of this research may use it for classifier updates without having to have ANNs generated on a per implementation basis.
- Create a sample client implementation to illustrate the capabilities of these classifiers, as well as demonstrate the end-to-end process of generating an ANN to the classification of a URL in a client's browser. Additionally, this must show what actions are available once a positive classification is made.

1.3 Approach

Research was done to determine state-of-the-art approaches when identifying malicious URLs using their lexical features alone. Data was collected from Phishtank[1] and Open Directory[2] to use in the training of sample classifiers. A prototype ANN inducer was developed to use this data to generate classifiers capable of performing this analysis. These classifiers were used to prove that the concept works and that the accuracy stated by other authors is possible.

Once this was known, a framework was built to facilitate the generation of ANNs for this purpose. Using the training data already mentioned, classifiers were trained using various parameters and then tested in terms of training time, accuracy and compressed sizes. When these values were known, the update distribution service was built and tested. Finally, the sample client implementation that uses this framework was developed and analyzed.

1.4 Document structure

The remaining five chapters of this document are as follows:

[1] http://www.phishtank.com/
[2] http://www.dmoz.org/

- Chapter 2 discusses background information regarding the structure of URLs and phishing attacks. Consideration is given as to why phishing attacks are difficult to detect and how prevalent they are. Also discussed within this chapter is what measures are currently used to detect and mitigate the threat from these phishing attacks, Artificial Neural Networks (ANNs) and the various technologies used within the execution of this research.

- Chapter 3 describes how the ANNs are built for this research in terms of the number of input nodes layers. The method used to format input data and the approach to training are also described. Finally, validation and testing techniques used to perform the tests described in Chapter 5 are addressed.

- Chapter 4 presents information on the constituent components within the framework. This includes information regarding how the framework was designed for the simple generation of ANNs and implemented. Also discussed are the update distribution service, as well as a sample client implementation capable of using one of these ANNs on the fly within the Google Chrome Browser.

- Chapter 5 discusses the tests performed on ANNs developed in Chapter 3 that were generated using the framework described in Chapter 4. It is determined what number of samples should be used when training these classifiers; at what speed they can be trained and the size of the resulting updates which clients need to download.

- Chapter 6 gives a conclusion based on the work presented in this research. Also discussed are topics regarding related future work.

Provided in the appendices is information regarding the research discussed in this document that is relevant, but not strictly necessary, within the main body. These are referred to in the text where relevant. Appendix A provides short descriptions of the components created as part of the network generation library. An example configuration for the Classifier Generation Service (CGS) is given in Appendix B. The Net Defence manifest file as well as the high-level detection algorithm are shown in Appendix C.

Appendix D lists publications by the authors that are related to this document. Finally, all of the code that was generated as part of this research is available online and is listed in Appendix E.

CHAPTER

TWO

BACKGROUND

This chapter addresses terms and concepts relevant to the research contained in the rest of this document. Firstly, the concept of Uniform Resource Locators (URLs) and Uniform Resource Identifiers (URIs), are covered in Section 2.1. Discussed in Section 2.2 are various topics regarding phishing and how the URL format is abused to aid in information stealing. Also covered are the different types of phishing, the prevalence of phishing as well as the methods used by attackers to avoid detection. Section 2.3 discusses what methods are currently used to protect users against phishing attacks, the effectiveness of each of these methods and what methods attackers employ to mitigate the effectiveness of these techniques. The basic structure of Artificial Neural Networks (ANNs) as well as the algorithms used to train them are covered in Section 2.4. Finally, the various technologies used in the implementation of the software output of this research are introduced and motivated in Section 2.5.

2.1 Uniform Resource Locator (URL)

A Uniform Resource Locator (URL) is a character string that acts as a method of finding and accessing specific resources on the internet (Berners-Lee, 1994) and was developed by Tim Berners-Lee and the Internet Engineering Task Force (IETF) in 1994. The term Uniform Resource Locator (URL) is often used interchangeably with the term Uniform

Resource Identifier (URI). Berners-Lee (2005) defines an *Identifier* as an object that "*embodies the information required to distinguish what is being identified from all other things within its scope of identification.*".

The RFC 3986 describes the generic format of a URI (Berners-Lee, 2005). This specification provides a method of identification of resources that is *uniform* through a set of rules that are extensible without placing restrictions on what the resource that is being identified may be. The generic structure of a URI with all optional sections present is shown below; with each part being defined in Table 2.1.

scheme://domain:port/path/file?query#fragment

In Berners-Lee (2005), a definition of what a *Resource* is, is said to be non-specific. It may be any electronic data such as a file, image or service and is not necessarily accessible via the internet. In the scope of this document however, a resource is defined as a resource that is intended to be accessible via the internet and is primarily concerned with Hyper Text Transfer Protocol (HTTP).

Table 2.1: The generic parts of a URL.

scheme	The protocol being used (HTTP, HTTPS, FTP)
domain	The address of the server (IP address or used by a DNS server)
port	The port that the resource is available on
path	The specific path that the resource is available on
file	The file that exposes the resource via whatever means
query	The specific query used by the file to access the resource
fragment	Which part of the resulting data to access

HyperText Markup Language (HTML) has a mechanism for displaying URLs on webpages known as anchor tags. These tags are clickable links which direct the browser to navigate to the URL defined in the href component of the tag. This mechanism enables users to access multiple resources quickly with the use of a mouse only, by clicking on anchor tags within rendered HTML documents. As a result, users do not need to type the URL directly, but simply follow links displayed to them. The HTML mechanism for displaying anchor tags allows for the masking of these URLs with text that is not required to have any specific bearing on what resource the URL accesses. This mechanism is often used as a way for words in a sentence to be clickable, allowing authors to say things like: 'click here for access to a particular resource', which a user may click on; without knowing the URL used to identify that resource.

When a user browses the internet, the URL for the current resource is often shown in the web browser's address bar, located at the top of the browser by convention. Many companies advertise the URLs that are used to access their websites and related resources. The fact that anchor tags do not have to display the URL they point to, or can use a different URL in the anchor text to the one used in the href component; leads to the opportunity for this system to be abused, as discussed in Section 2.2.

Once a user follows a URL, the browser will perform a Domain Name System (DNS) lookup which converts the domain into a address which can be used by the Internet Protocol (IP) and Transmission Control Protocol (TCP) to find the host which supplies the resource. The resource is then requested from the host by the user's browser, supplying the URL as a method of identifying the resource required as well as the desired protocol for accessing it.

In summary; while a URL or URI does not specifically apply to internet resources, they are used extensively as such. The method by which HTML makes URLs easier to use, also allows the URL location to be masked in an effort to hide technical details from non-expert users. URLs may also be emailed to users in HTML format, allowing the location of the resource to be masked in the same manner.

2.2 Phishing

Phishing is a form of social engineering where end-users are deceived into entering identifying or sensitive information into fraudulent resources (Berghel, 2006; Fette, Sadeh, and Tomasic, 2007; Dhamija and Tygar, 2005). This is usually achieved by providing a link to the fraudulent resource through email, links on a compromised website or other channels. These fake sites are often identical in facade and may have entire sets of functionality built in to deceive a user into thinking that it is a legitimate resource. The user is then prompted to enter information that is sensitive in nature; such as, log-in credentials, banking details, online gaming credentials or similar information (Dhamija *et al.*, 2006; Hong, 2012; Elser and Pekrul, 2009). An example of a real-world phishing attack (or phish) is shown in Figure 2.1. This phish targeted the PayPal online payment service and shows how high-fidelity phishing attacks may be.

The rest of this section will cover various topics regarding phishing. Section 2.2.1 discusses the four general types of phishing and their intended targets. Covered in Section 2.2.2

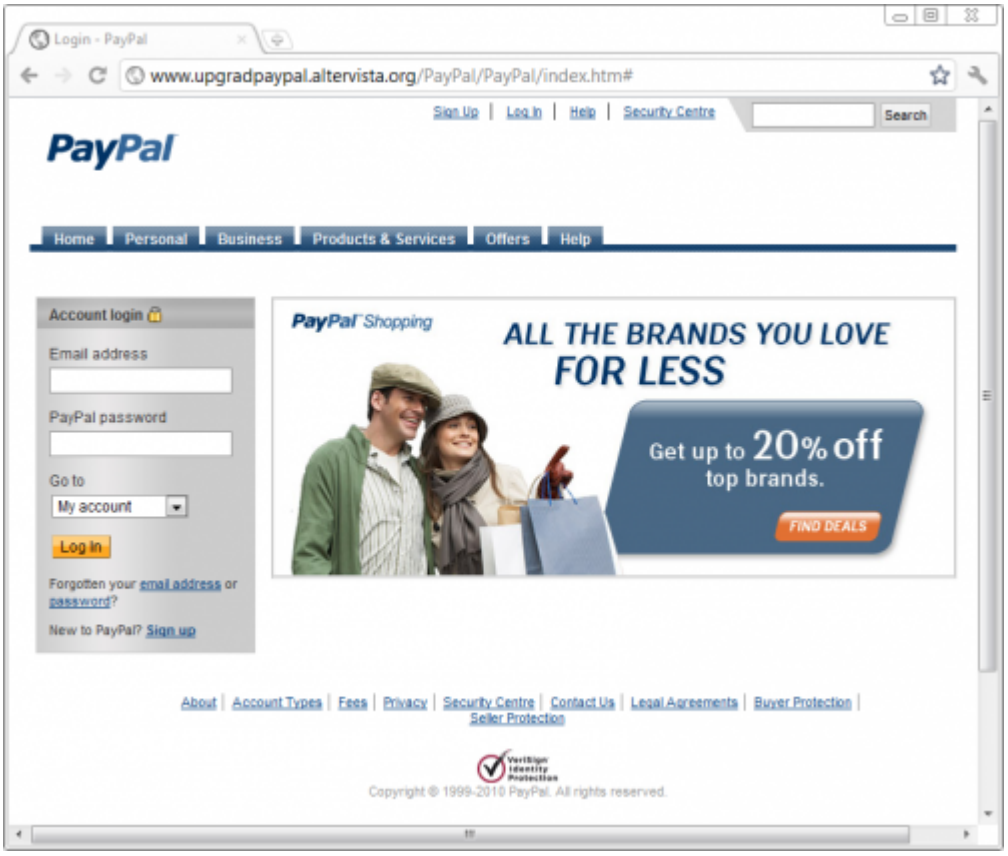


Figure 2.1: A phishing website targeting PayPal.

are the methods used by attackers to obfuscate phishing URLs. In Section 2.2.3, the prevalence and other statistics regarding phishing are discussed.

2.2.1 Types

There are four general categories of phishing which cover the act of stealing sensitive information by using fraudulent websites and other resources (Garera *et al.*, 2007). The first of which, simply called phishing, is a general term which covers an attempt by attackers to gather information from non-specific targets (Jagatic, Johnson, Jakobsson, and Menczer, 2007). These targets are often lured onto web sites masquerading as the legitimate service using emails, in an attempt to deceive them.

Spear phishing is a more targeted approach. In this scenario, attackers will target specific individuals, or organizations, by researching the target. In this manner, attackers seem to have 'inside' information, that makes them appear more legitimate than if they were to try the traditional phishing method (Brody, Mulig, and Kimball, 2007). Whaling is a form a spear phishing that targets high profile users, such as CEOs (Hong, 2012).

Within Clone Phishing, a previously sent email is copied and resent. This re-sent version will contain an URL or malware sample that replaces original content. This is done and is made to appear as an updated version of the email, with 'corrected' contents (Mohamed, 2013).

2.2.2 Detection-avoidance and obfuscation

As stated in Ma, Saul, Savage, and Voelker (2009b), every form of phishing hosted on the internet requires a URL to link to it. This URL has to reach the target, either by providing a link where the target audience will find it or, more often, in the form of an email. Often, attackers will exploit the fact that anchor tags do not have to display URL destinations, showing a different location in the anchor tag text to that in the href field of the tag (Chandrasekaran, Narayanan, and Upadhyaya, 2006b). Compounding this problem is that end-users do not necessarily understand the syntax of URLs and, as a result, the destination of the resource they point to (Dhamija *et al.*, 2006).

Another method used by attackers intended to mask the intent of a URL is to use obfuscation. In URL obfuscation, an attacker will create a URL that is either subtly different from the legitimate resource's URL, or will try to make the URL not understandable by non-expert users such as re-encoding the URL in hexadecimal format (Milletary, 2005). An example of this is shown in Listing 2.1 where the text shown to the user will be *http://www.safe-site.com* but clicking the link will direct the user to *http://www.phishing-site.com*.

Listing 2.1: Example of an anchor tag with mismatched href and text component

```
1 <a href="http://www.phishing-site.com">http://www.safe-site.
    com</a>
```

Examples of the first kind of obfuscation include duplication of letters in place, or addition of extra tokens to the domain section. An example of this kind of obfuscation where tokens are added to the domain is shown in Listing 2.2. There may be subtle misspelled words that are hard to notice without careful inspection of the link. Other techniques would include using an IP address instead of a domain in the URL, as is discussed in Kirda and Kruegel (2005); or appending the URL with a port number, which is also effective as some legitimate web services run on non-standard ports. For more information on these obfuscation methods, see Egan and Irwin (2011a,b) as well as Ma *et al.* (2009a).

Listing 2.2: Example of an obfuscated URL

```
1 http://update.paypal.com.user.id.39
    e3f0f901a9fe6dee45c57aa68ef100267afw161a2h1wcx1ac.the-
    deans.net/
```

The primary result of these slight adjustments to URLs of legitimate resources means that many non-expert users may not notice the incorrect tokens within the URL. This also makes it very difficult for automated detection methods to identify these URLs without latency-inducing measures which have a large impact on legitimate traffic. Compounding the problem is users who may ignore security warnings built into browsers or security toolbars (Wu *et al.*, 2006). Hiding URL destinations by using an IP address is more straightforward, and should be avoided by non-expert users as a general rule. It is these very subtle changes that users such as network security professionals may be able to spot at a glance. This may be because they are more aware of these attacks and are cautious when reading emails that contain links requesting information that should never be requested.

2.2.3 Prevalence

According to the Anti Phishing Working Group (APWG)[1]; the number of phishing attacks for the first half of 2013 was reported to be 72 758 instances, an increase of 5 081 instances during the same time in 2010 (7.5% increase). Within this same period there were 53 685 domain names registered for phishing purposes (26% increase from 2010) as well as 1 626 IP-based phishing attacks targetting 720 organisations. This represents a 22.7% increase in the number of organisations targeted for phishing attacks from 2010. For more information regarding these statistics, see Aaron and Rasmussen (2013).

According to Dhamija *et al.* (2006); up to 5% of users who encounter fraudulent resources will divulge sensitive information. Litan (2004) said that by May 2004, nearly 57 million American adults had had encounters from attackers pretending to be legitimate services via email. The authors go on to estimate that 19 percent of the victims will actually engage in the phishing scheme, and that 1.78 million of them handed over sensitive information. This resulted in a loss of $1.2 billion dollars in 2003 according to Litan (2004). By 2007, this loss was up to $3.2 billion dollars from 3.6 million American adults (McGrath and Gupta, 2008).

[1] http://www.apwg.org

Work discussed in Dhamija *et al.* (2006) shows that high-fidelity phishing resources may fool up to 90% of the users who encounter them. They go on to say that, within their study, 23% of the participants ignored mechanisms built into the browser to help avoid these phishing attacks. These mechanisms include the address bar, status bar and other indicators. Interestingly, users who are educated about phishing attacks have been shown to be no better at identifying phishing attacks than people who have not been educated. This was shown in Anandpara, Dingman, Jakobsson, Liu, and Roinestad (2007), which went on to say that educated individuals reported more attempts as phishing, but with no greater accuracy, indicating an increased fear rather than ability to identify phishing attacks.

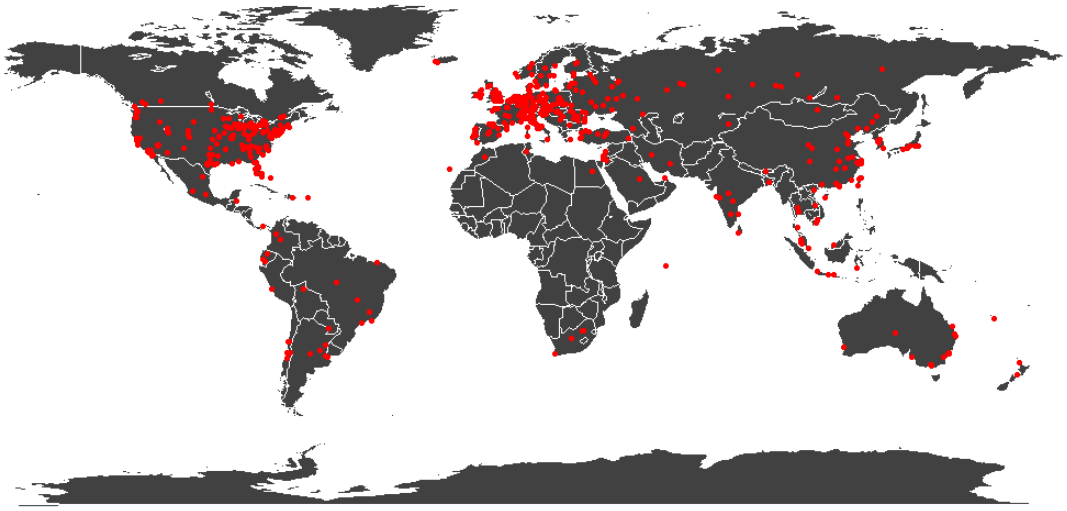


Figure 2.2: Unique locations of phishing hosts.

Using the MaxMind GeoLocation Library[2], 12 084 samples of phishing URLs were geolocated. This data set was collected in April 2012 from Phishtank. Phishtank is a website where users may report suspected phishing URLs. Moderators then manually confirm or deny these URLs by visiting them, and adding them to the Phishtank blacklist. This blacklist is available for download from their website[3]. Shown in Figure 2.2 are the unique locations of hosts used for phishing attacks. It is clear from this data that phishing hosts are found all over the planet, with the highest concentrations being in first-world countries.

[2]http://www.maxmind.com/app/api
[3]http://www.phishtank.com

Shown in Table 2.2 are the countries with the highest population densities from the April 2012 Phishtank data set in terms of hosting. Also shown within this table are the hosting densities from April 2013 according to EMC (2013). The data show that the United States hosted 44% of reported phishing sites during 2012, with that number rising to 47% by April 2013. Brazil, while hosting 10% of all reported phishing resources contained within the Phishtank 2012 data set, was not mentioned in the 2013 data set in EMC (2013). The same is true for Australia which hosted 4% of phishing sites in 2012. The Netherlands hosted 4% during 2012 and dropped to 3% by April 2013. Germany and the United Kingdom rose from 4% and 3% respectively in 2012, to 6% and 4% respectively in 2013.

Table 2.2: Population density of phishing hosts by country.

Country	2012	2013
United States	44%	47%
Brazil	10%	Not listed
Australia	4%	Not listed
Netherlands	4%	3%
Germany	4%	6%
United Kingdom	3%	4%

Also discussed within EMC (2013) are the number of phishing attacks and targets by country. In April 2013, there were 26 902 attacks identified, with the United States targeted 46% of the time. This was followed by the United Kingdom with 11% of attacks and South Africa by 9% of the attacks.

2.3 Phishing counter measures

Discussed within this section are the modern approaches used in security systems to mitigate the threat from phishing attacks. Each approach is described in terms of its basic operation, logical configuration and construction as well as the advantages and disadvantages of its approach.

2.3.1 Blacklists

A blacklist is a method of filtering access to resources by using a *default allow* approach. The traditional approach to creating blacklists is to maintain a list of resources that

should be denied access to (Ramachandran, Dagon, and Feamster, 2006). This method is generally used when there is a large number of resources that should be accessible which make it prohibitive to use a *default deny* (Whitelist) method which would require a list of all accessible resources to be maintained.

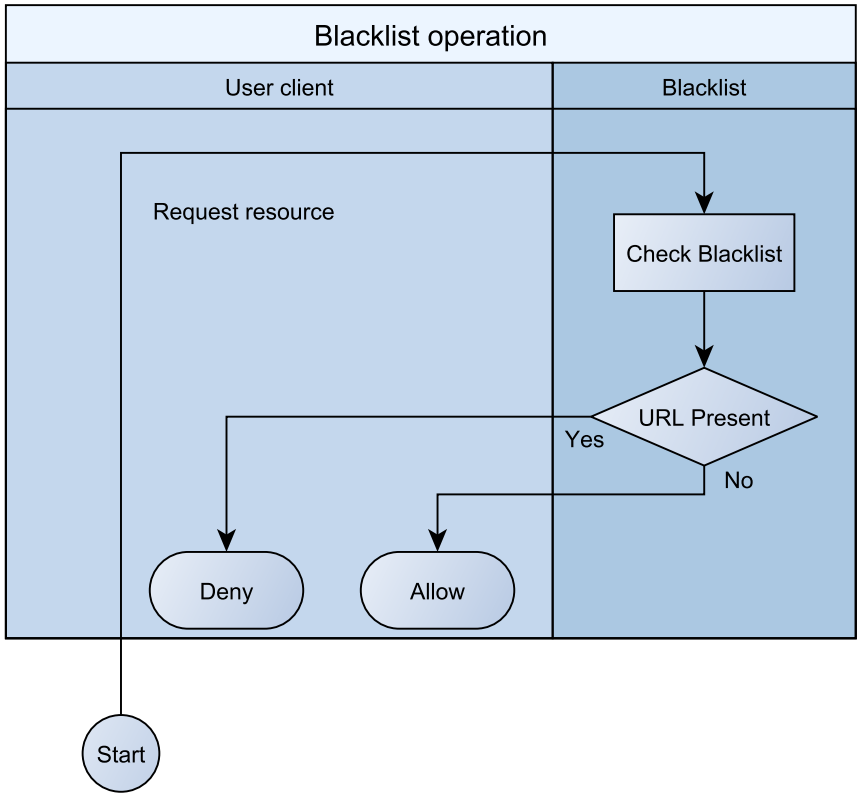


Figure 2.3: Blackhole list transaction.

In a typical transaction, a request is made which is compared against a blacklist and, if the resource does not appear within the blacklist; the request is permitted, if not; it is denied. This is depicted in Figure 2.3. This transaction is not necessarily enforced as it may be implemented as a service which a client may query, and may also be implemented as a gateway requirement on any request. Blacklists can also be implemented as a Domain Name System Blackhole List (DNSBL), also known as a Real-time Blackhole List (RBL), which may be used to mark hosts or networks that shouldn't be accessed (Ramachandran *et al.*, 2006).

This method of content filtering has been used extensively in web technologies due to the sheer number of resources available on the internet (Ramachandran *et al.*, 2006; West and Lee, 2011). Examples of commonly used blacklists include Google Safe Browsing (Google, 2013c) and Microsoft Smart Screen (Microsoft, 2013b). Blacklists have a high accuracy

as many of them are manually vetted. This means that humans confirm or deny that a particular resource is malicious in nature, thus keeping the accuracy of the blacklist very high. This, however, incurs its own problems. Blacklists may be slow to respond to new malicious resources for a number of reasons as shown in Sheng, Wardman, Warner, Cranor, Hong, and Zhang (2009). Firstly, the resource may not be discovered for some time after it becomes available and, as a result, the first requests made for the resource are not aware of its malicious nature. Another scenario is that the resource has not been flagged as it is waiting for a human technician to confirm the nature of its intent and add it to the blacklist (Sheng *et al.*, 2009; Zhang, Hong, and Cranor, 2007).

Another problem relating to blacklists is that of user-experience. Blacklists may exist as a local resource like in the case of Google (2013c) and anti-virus packages, or they may exist outside of the local network. In both cases, the blacklist needs to be polled before access can be granted or denied. In the case of a non-local blacklist, this incurs additional latency that may not be tolerable in certain applications or compound an existing problem, such as in South Africa; where network latency and low bandwidth is a common problem for end-users. This impact on user performance may cause users to not use blacklist services at all, causing a major security problem. A method of mitigating this issue is to use a local blacklist resource with periodic updates to avoid latency issues.

Privacy is another issue that blacklists are concerned with. In the standard method of maintaining a blacklist, a list of hosts or resources are kept. As a result, when a client makes a request for a lookup of a particular resource, the URL has to be transmitted in clear text to make the comparison. This makes it possible to determine what the user is requesting. A method of enforcing privacy when using blacklists is for the blacklist to keep a list of hashed URLs rather than the URLs themselves. The client makes requests for lookups by transmitting a hash of the URL rather than the URL itself, thus maintaining privacy.

2.3.2 Spam filters

Spam filters are used within email servers to separate email into *ham* and *spam* messages (wanted and unwanted) using several approaches (Cormack and Lynam, 2007). While these are relatively effective, without constant updating they quickly become irrelevant as obfuscation trends change. Another downfall of spam filters is that they generally apply heuristic rules to the content of emails, rather than identifying URLs specifically. Zhang, Zhu, and Yao (2004) show that statistical models trained to inspect both the message

body and the header improve the performance of these spam filters. Many spam filters will integrate with a RBL to improve performance.. As a result they tend to inherit the shortfalls of blacklists. However, they do not inherit the usability problems of blacklist implementations as they are able to apply filtering techniques when an email is received rather than when a user requests it. A method, discussed in Androutsopoulos *et al.* (2000) and Sahami, Dumais, Heckerman, and Horvitz (1998), is to use Naive Bayesian Networks as a spam filtering mechanism. Discussed in Li and Zhong (2006) is a method of making this approach more effective in large environments using approximate classification.

2.3.3 Reputation services

A third approach used today is collectively called Reputation Services, such as the Microsoft Reputation Services (MRS)[4]. These are offered by many security companies such as Avast with the *avast! Online Security Browser* plugin which is available for all major browsers. Another popular tool of this type is *Web of Trust*[5]. This works by having users rate websites and other resources as they use them, with the rating being reported back to a central reputation service that other users may query (Windley, Daley, Cutler, and Tew, 2007). When browsing, the plugin will create a visual cue near links to indicate the level of trust assigned to that resource by other users. While this approach is useful in theory, it will only work for large sites which have large numbers of users. Smaller, but still legitimate, sites will have no rating and users will be unable to tell whether they are benign or malicious. Any new phish site coming into operation has no rating, making the reputation service approach unusable for phishing avoidance. Like blacklists and spam filters; this service also incurs latency when clients have to make lookups requesting URLs they wish to visit.

2.3.4 Content scanners

In an approach known as Content Scanners, an application downloads the resource before attempting to identify it as illegitimate (Chandrasekaran, Chinchani, and Upadhyaya, 2006a). This is done by checking for obfuscation techniques such as those discussed in Section 2.2.2, among other detection methods. This approach differs slightly from the spam filters discussed in Section 2.3.2 in that they are used to identify malicious

[4]http://www.microsoft.com/mscorp/twc/endtoendtrust/vision/reputation.aspx
[5]http://www.mywot.com/

web pages rather than emails specifically. SpoofGuard for Microsoft Internet Explorer is a plugin which performs content scanning functions to validate authenticity of a page (Chou, Ledesma, Teraguchi, Boneh, and Mitchell, 2004, see). Another example of a content scanner is CANTINA which uses a heuristic based on the TF-IDF algorithm (Zhang *et al.*, 2007).

2.4 Artificial Neural Networks

A new approach to detecting malicious URLs is to use the statistical modelling approach implemented through Artificial Neural Networks (ANNs). These ANNs are a modelling mechanism which can be trained to classify samples. It is for this reason that they are often called *classifiers*, which is a term that is used interchangeably throughout this research. In Section 2.4.1 the structure of ANNs are discussed, followed by 3 methods of training them, shown in Section 2.4.2. Section 2.4.3 discusses techniques needed to create useful data sets used for training and validation of classifiers. Finally, in Section 2.4.4, reliable methods for determining how effective a trained classifier is at generalising to unseen data are shown.

2.4.1 Structure

Artificial Neural Networks (ANNs) are loosely modeled around how a subsection of a biological neural network works and, as a processor of information, is very different from the classical von Neuman model (Jain, Mao, and Mohiuddin, 1996). Many authors have cited them as being successful in applications of pattern recognition, image and speech recognition (Lippmann, 1988; Nadel and Stein, 1995; Rowley, Baluja, and Kanade, 1998). They consist of neurons which are interconnected by a layer of unidirectional edges, passing information between them (Gardner and Dorling, 1998). Neurons within an ANN are arranged in layers, every ANN consisting of at least two layers. These two layers are the *Input* and *Output* layers (Svozil, Kvasnicka, and Pospichal, 1997). Figure 2.4 shows the logical layout of a basic perceptron.

The neurons at the input layer represent a vector that contains values as parameters of a sample for the ANN to process. Each layer after the input layer is connected by a series of weighted synapses which act as multipliers on the input side of the synapse (Lippmann,

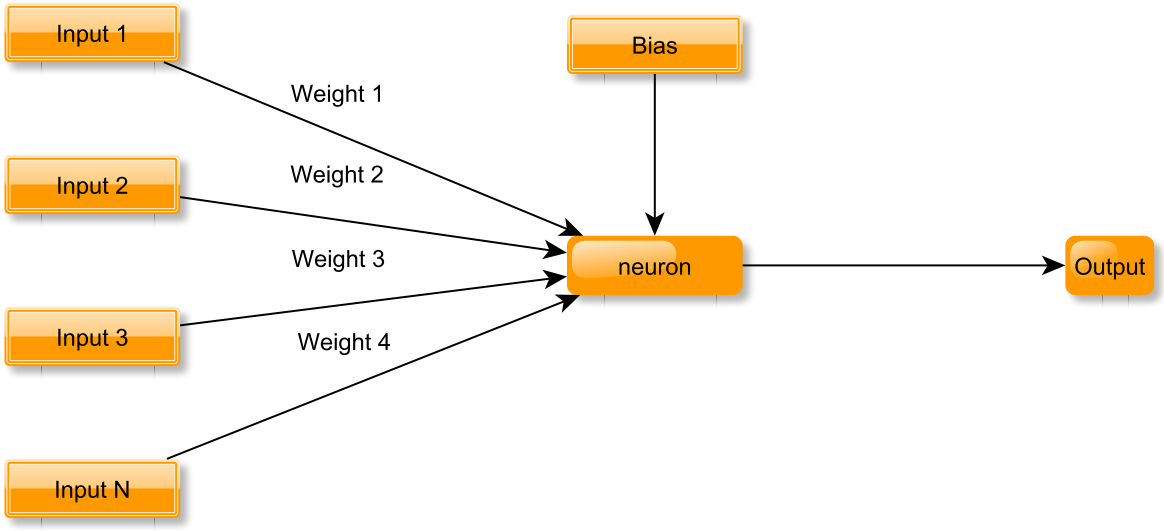


Figure 2.4: Basic layout of a perceptron.

1988). At the output side, the neuron uses a linear combination of all of the values of the synapses connected to it and is shown in Equation 2.1.

$$X = \sum_{i=1}^{n} x_i w_i \tag{2.1}$$

Within Equation 2.1, where X represents the resulting combined value of the equation; x represents the value of the input neuron and w is the weight of the associated synapse of that neuron. Once this result has been obtained, it is put through a hard limit which sets the final output of the ANN (Lippmann, 1988). Shown in Equation 2.2 is the method used in this research and is known as the *sign activation function* or *sign transfer function*. It works by adding a *bias* (or *threshold*) value to the linear combination value. The result, being either a 1 or a 0, is known as a binary classification (Lippmann, 1988).

$$f(x) = \left\{ \begin{array}{ll} 1 & \text{if } X + b > 0 \\ 0 & \text{if } X + b \leq 0 \end{array} \right\} \tag{2.2}$$

In this structure, the ANN is known as a *Perceptron* (Lippmann, 1988). The terms ANN, Perceptron and classifier are used interchangeably throughout this research. A more complex structure is to have one or more layers of neurons, each of which has varying sizes, connected between the input and output layer and is known as a *Multi Layer Perceptron* (Gardner and Dorling, 1998; Hornik, Stinchcombe, and White, 1989).

These layers are known as hidden layers (Svozil *et al.*, 1997). This structure is used when the problem space is not linearly separable and is outside of the scope of this research.

In summary, a ANN takes a vector of inputs, passes it through a series of weighted synapses, combines them and adds them to a threshold value. This value is then passed through an activation function which determines the final classification of the input vector.

2.4.2 Learning

The most simple form of Artificial Neural Network (ANN) is the Perceptron. It comprises of just two node layers: the input layer; which performs no calculations, and the output layer. The node in this layer is connected to all of the nodes in the input layer via a weight vector and makes classifications as discussed in Section 2.4.1. This structure and classification method were introduced in Rosenblatt (1958) and have been used widely.

Algorithm 1: Online Perceptron learning algorithm.

$$\gamma = Y_d - Y \tag{2.3}$$

$$\Delta w = \alpha \times x_i \times \gamma \tag{2.4}$$

$$w_{t+1} = w_t + \Delta w \tag{2.5}$$

Where:

- γ is the error of a classification.
- Y_d is the desired label of a sample.
- Y is the labeling result of a classifier for a sample.
- Δw is the amount of change required for a given input.
- α the learning rate scalar.
- x_i is the input for a particular node.
- w_t is the weight associated with the given node.

Rosenblatt's Online Perceptron (OP) method of training perceptrons is a common method of supervised learning (Freund and Schapire, 1999). The approach to learning taken by this algorithm is one which iterates repeatedly over a data set of labelled samples until a chosen accuracy metric is achieved (Riedmiller, 1994). For each sample, if the resulting

classification is correct, it moves onto the next sample. When an incorrect classification is made, the error is calculated.

This is calculated by subtracting the resulting classification from the desired prediction for the labelled sample and is shown in Equation 2.3. Once this error value has been calculated, a delta value is calculated by scaling the error with the value for the input node as well as a *learning rate* scalar as shown in Equation 2.4. Finally, a new weight value is calculate by adding this delta value to the existing weight value, shown in Equation 2.5. This process is executed for the entire weight vector for each incorrect classification made.

The primary shortfall for the OP method is that the learning rate scalar is static (Le *et al.*, 2011) and a hard limit is used. In this case, a correction (adjustment to the weight for a given node) is always of the same size. This has the effect of causing an oscillation around the optimal result when the training process approaches a converged solution, as the required correction may be smaller than the static learning rate scalar can provide. This problem is compounded by a training data set that is noisy. Within Le *et al.* (2011), two methods of training an ANN which try to mitigate these problems with the OP method are discussed.

Introduced in Crammer, Dredze, and Pereira (2008), is the Confidence Weighted (CW) training algorithm. The approach to classification within this algorithm is exactly the same as with the perceptron. It differs only in the training process, meaning that the induction of the ANN is modified. The CW algorithm does not have a static scaling vector, such as the learning rate within OP. Instead, it maintains a covariance matrix (Σ) which acts as a confidence for each input weight. This has the significant advantage of not changing weights that have a high confidence by very much, but also making large changes in weights of the inputs which have a low confidence weight associated with them. The general method used to determine the confidence is such that a weight that is not changed often is represented with high confidence within Σ while a weight that is updated frequently is represented by a low confidence. The exact methods for updating the covariance matrix and the input vector weight associations are shown in algorithm 2.

The final algorithm discussed in Le *et al.* (2011) is called Adaptive Regularization of Weights (AROW). AROW improves on CW by making it more tolerant of noisy training data through the use of an updated input weight adjustment method that is shown in Equation 2.8. However, like CW, it is able to train on correct classifications by increasing confidence in the weights of the features. This algorithm is a modified version of CW and

Algorithm 2: The Confidence Weighted (CW) training algorithm.

$$\mu_{t+1} = \arg\min_{\mu\Sigma} D_{KL}(\mathcal{N}(\mu, \Sigma)\|\mathcal{N}(\mu_t, \Sigma_t)) \tag{2.6}$$

$$\Sigma_{t+1} = \mathsf{s.t.}\mathsf{Pr}_{w\sim\mathcal{N}(\mu,\Sigma)}[y_t(w \cdot x_t)] \geq \eta \tag{2.7}$$

Where:

- μ is the average input weight vector.
- Σ is the covariance matrix indicating confidence of input weights.
- D_{KL} is the KL Divergence.

was introduced by Crammer in Crammer, Kulesza, and Dredze (2009) which is shown in Algorithm 3.

Algorithm 3: Adaptive Regularization of Weights (AROW) training algorithm.

$$\mu_{t+1} = \arg\min_{\mu\Sigma} D_{KL}(\mathcal{N}(\mu, \Sigma)\|\mathcal{N}(\mu_t, \Sigma_t)) + \lambda_1 l_{h^2}(y_t, \mu \cdot x_t) + \lambda x_t^T \Sigma x_t \tag{2.8}$$

$$\Sigma_{t+1} = \mathsf{s.t.}\mathsf{Pr}_{w\sim\mathcal{N}(\mu,\Sigma)}[y_t(w \cdot x_t)] \geq \eta \tag{2.9}$$

Where:

- μ is the average input weight vector.
- Σ is the covariance matrix indicating confidence of input weights.
- D_{KL} is the KL Divergence.

Le *et al.* (2011) show how both of these training algorithms show improvement in terms of cumulative error rate over the OP method.

2.4.3 Data balancing

A class of data points represents a group that has a unique quality when compared to data points of other classes. A balanced data set is a concept within modern statistics that identifies a sample of data that equally represents all classes within that overall sample. When, within a binary data set, one class has many more samples than the other, it is called an imbalanced data set (Estabrooks, Jo, and Japkowicz, 2004; Guo and Viktor, 2004). ANNs may have problems optimising class labeling when such conditions

are present and special consideration is not made to compensate for the class imbalance (Japkowicz and Stephen, 2002; Estabrooks *et al.*, 2004).

There are two general approaches to this problem. The first is to adjust the learning algorithm to deal with a data set that is not balanced while resampling the data set is the second. There are two approaches to the resampling option: Over and Under sampling. The first duplicates samples within a class until the data set is considered balanced, while under sampling removes samples from a class until it is balanced with the class that has a lower number of examples. There are several versions of each approach listed in Batista, Prati, and Monard (2004) where heuristic rules are used to select samples for duplication or removal. Non-heuristic approaches are also used where random samples are removed or duplicated in each approach. These are known as *Random over-sampling* and *Random under-sampling* (Batista *et al.*, 2004). A combination of these two approaches where the majority class is undersampled while the minority class is oversampled is used in Chawla, Bowyer, Hall, and Kegelmeyer (2002). Random under-sampling is used to balance the data set mentioned in Section 4.5.5 and it is an implementation of the *NetworkData-Balancer* interface (discussed in Appendix A.3 called the *DataRemovalNetworkBalancer* (discussed in Section A.10).

2.4.4 Classifier Validation

If a classifier is trained to match a sample population of training data, it is possible to train it beyond the point where it uses relevant features to make classifications and it starts to use features to identify the data set it is being trained on (Tetko, Livingstone, and Luik, 1995; Kohavi, 1995). This is known as *over-fitting* a data set and happens when a classifier is trained with few samples or when a network is trained for too long (Tetko *et al.*, 1995). Over-fitting is also known as a Type I error in this context (Peduzzi, Concato, Kemper, Holford, and Feinstein, 1996).

Classifier validation is a method by which a classifier's performance can be determined and used as a way of choosing a classifier during training (Rao and Wu, 2005). This is done by using an independent data set (a data set that has not been used in the training process). Known as Hold-out validation, the intention of this process is to estimate how accurately the classifier may perform in the real-world and is usually executed over several rounds to avoid variances in the data used (Arlot and Celisse, 2010).

The general method of performing validation is to extract a sample from a population of data and to use that sample only for validation purposes (Arlot and Celisse, 2010). The

rest of the data is then used for training. Typically, this process is reversed in that the data sets are then swapped in roles and training and validation are then executed again, with the validation score then calculated as an average of both validation runs. This has the advantage of all the data being used for both training and validation. When accuracy is calculated by this method of using an independent data set for validation, it is known as an *out-of-sample* estimate, while using the training data itself is known as an *in-sample* estimate as the accuracy is calculated using the training sample.

K fold cross-validation (Rotation Estimation (Kohavi, 1995)) is a cross-validation technique where the population data set is divided into K portions. One of these portions is then used for training, while the others are used for validation, with their results averaged. This process is then repeated, using a different portion of the data for training. This continues until all K portions are used as both a training and validation set. This has been shown to be computationally expensive in Bengio and Grandvalet (2004). A popular implementation used is ten fold validation; where a data set is divided into ten parts.

Another method of performing cross-validation is known as Leave One Out Cross-Validation (LOOCV). In this approach, a single sample is left out of the training sample and used as a validation sample (Kearns and Ron, 1999; Cawley and Talbot, 2003). The process is repeated, using a different sample as a validation sample each time, until each sample in the population has been used as a validation sample. This approach to cross-validation is considered to provide an unbiased estimate of how well a classifier will generalize (accuracy regarding unseen data) (Vapnik and Chapelle, 2000; Chapelle, Vapnik, Bousquet, and Mukherjee, 2002).

Hold-out validation is implemented as part of the ANN training process discussed in Section 4.5.5. This is done by implementing the *NetworkValidator* interface discussed in Appendix A.7. The implementation is known as the *SinglePassNetworkValidator* and is discussed in Appendix A.13.

2.5 Framework technologies

Part of this research is to deliver a framework by which ANNs can be trained, distributed and deployed. Discussed within this section are the relevant technologies involved in the implementation of this framework. Section 2.5.1 discusses the programming languages

used during this research, while Section 2.5.2 briefly covers Representational State Transfer (REST) as a suitable method of making updates available to clients. Finally, the Google Chrome Extension Application Program Interface (API) is discussed in Section 2.5.3 as it is used to develop a client for this service in Section 4.7.

2.5.1 Programming languages

Python is a scripting language that was introduced in 1989 (Chun, 2006). It is considered to be highly expressive and easy to learn (Chun, 2006). It was initially chosen for use within this framework for these reasons, as well as its support for object-oriented programming, its scalability and for the rapid development times associated with this high level scripting language. It was later dropped in favor of C# for reasons discussed in Section 4.5.

Unlike Python, C# is a language designed for use within Microsoft Windows that is compiled into a Microsoft Intermediate Language assembly (Liberty, 2001). Certain C# applications may be deployed on Linux using the Mono project[6]. The framework that is built as part of this research (Section 4.5) is built as a .NET library. Programs wishing to use some or all of the capability that this library provides, may import it as part of the capability of the .NET Common Language Runtime (Schult and Polze, 2002).

JavaScript is also used and is the basis for the Google Chrome Extension API (Section 2.5.3, see). It is a scripting language that is predominantly used within web technologies such as HTML (Flanagan, 1998). JavaScript is fully object-oriented and uses syntax similar to C++ (Flanagan, 1998).

2.5.2 Representation State Transfer (REST)

The decision was made to use a web service to distribute updates to global clients. A REST implementation is used for this purpose as it is the most common method of implementing web services used today (Battle and Benson, 2008) and is suitable for simple integration between client applications and the service provider (Pautasso, Zimmermann, and Leymann, 2008).

The concept of REST was first introduced in Fielding (2000) with the author being the principal developer of the idea. Within his PhD thesis; Fielding identifies several formal

[6]http://www.mono-project.com/

constraints on a REST implementation. He first identifies that a REST implementation is built in terms of the Client-Server model and that it should separate the user interface from the method by which data is stored. In this fashion, it improves the ability of client applications to consume the service as it has fewer requirements of them. The second constraint he imposes is that it is a stateless system as it reduces complexity in both client and server implementations. The final constraint covered here is that of a uniform interface. This standard method of communicating with the service decouples its functionality from the interface. For further information on Fielding's constraints and definitions of the REST architectural style, see Fielding (2000).

Within HTTP, REST is implemented using the HTTP verbs as methods of creating, reading, updating and destroying objects. HTTP naturally works with REST as it is stateless and has the use of URLs to direct clients towards resources. Using this stateless interaction between client and server, interaction with a REST API can be made with a single call, containing all of the data required. As a result, REST lends itself well to framework development.

2.5.3 Google Chrome Extensions

The Google Chrome browser is a popular browser used today which allows for functionality to be extended by allowing third-party developers to create extensions and plugins. Chrome allows these extensions to interact with user requests, web pages as well as other functionality of the browser through its API Google (2013a). Extensions may be built in three main forms: *Browser Actions*, *Page Actions* and *Packaged Applications*

Browser Actions are typically implemented as functionality which extends the capability of the browser. They take the form of a button to the right of the address bar which, when clicked, opens a popup which exposes some useful functionality. Capabilities that are required for some pages, but not all pages, are implemented as *Page Actions* and take the form of an icon inside the address bar. An example of this would be a Really Simple Syndication (RSS) feed reader which detects that there are RSS feeds available for subscription on the page. *Packed Applications* represent web applications that are able to leverage the functionality and portability of Chrome.

The *manifest.json* file is a file included in all Google Chrome extensions. Listing C.1 is the manifest file that is associated with the Net Defence extension which is developed in Section 4.7. It is of particular importance as it defines what security permissions the

extension will require to function as a least-privileged implementation. This security feature means that if an extension is compromised by an attacker, the attacker has to operate with those permissions already requested (Barth, Felt, Saxena, and Boodman, 2010). It is also responsible for listing resources that the extension should have access to. Permissions included in a *manifest* file are typically a list of possible websites that the extension needs to access, what API features are required as well as what events the extension should be able to intercept. The user is asked to accept these security requirements when installing the extension. The *manifest* is also used to identify icons used for the extension, image sets and other options such as which files to use as entry points for the *options* page for each extension. Like the main extension; *options* pages are built as *HTML* pages with JavaScript implementations of functionality. These pages typically store settings through *Local Storage*; cookies for storing data in the browser itself, or through online APIs.

2.6 Summary

Several topics have been covered within this chapter. URLs are used in all categories of web usage as a method of identifying resources. As such, they are used in all facets of internet usage. Phishing is highly prevalent in internet usage today, causing large financial losses every year to many users the world over. Attacks of this kind can use URLs as a method of deceiving users into navigating to fraudulent resources or may use URL obfuscation as a method of avoiding detection by current technologies employed to mitigate this threat. These conventional methods have been shown to have flaws in terms of latency-induced user-experience issues and the inability to adapt quickly to new phishing attacks or trends in URL obfuscation.

Artificial Neural Networks (ANNs) are shown not to suffer from these problems. They do not suffer from latency-induction due to their parallel processing nature and are able to predict well on unseen data if they have been trained correctly and validated using cross-validation techniques. If they are to be used in a production environment, various technologies will need to be employed to make them a feasible option to end-users. The technologies chosen and discussed in regards to this were C#, Representational State Transfer (REST) and browser extensions, with Google Chrome being the browser of focus within this research. Discussed in the Chapter 3 is how the ANNs are created for this purpose, with the framework implementation being discussed in chapter 4.

CHAPTER

THREE

ARTIFICIAL NEURAL NETWORK DESIGN

It has been shown in several papers, including both Ma *et al.* (2009a) and Le *et al.* (2011), that is is possible to identify malicious URLs using ANNs. Within this document, research is performed to attempt to prove that it is possible to generate the accuracy that these authors report, as well as determine if it is feasible to create these classifiers in a way that can be done quickly enough to be useful, distributable and usable by clients. As a result; their classifiers are implemented in this research using methods which they recommend. The method by which data is gathered and formatted for use is discussed in Section 3.1. Covered in Section 3.2 is the method by which the samples are labelled. Section 3.3 describes how the samples are divided into the various data sets required for the training and validation process. The method by which a sample is used to create an input vector is discussed in Section 3.4. Section 3.5 covers what the structure of the ANN implementation is and the various algorithmic options used. The training process is covered in Section 3.6 while the validation and testing methods used are discussed in Section 3.7 and Section 3.8 respectively.

3.1 Data Sources

Required for the training of ANNs are three sets of unique data. These sets are known as the *training*, *validation* and *testing* data sets. Within each set, there needs to be relatively

balanced numbers of both positive and negative samples, although this can be achieved through various balancing techniques, discussed in Section 2.4.3.

The primary sources of data used during this research were Phishtank[1] for phishing URLs while Open Directory[2] was used for benign samples. These sources were chosen as they are manually vetted. This means that individuals visit each URL by hand to verify the intent and content of each end point. As a result, the data has a high accuracy on the intention of the sample, with low noise. The Phishtank data source yielded 14 707 samples, while the Open Directory database contained 9 198 532 samples that were drawn upon to create training, validation and testing data sets. Both data sets were downloaded in September 2013.

3.1.1 Extraction

These data sets described are available for download from github (details in Appedix E). The first operation performed on the data is to extract it from its relevant container format, as implemented within the framework designed in Section 4.5. First in this process is to decompress the data and then extract it from the source's standard method of storage. In the case of Phishtank, this was a Comma Separated Values (CSV) format file, while Open Directory storage is in Extensible Markup Language (XML) format. Once all of the data are extracted, it is checked for duplicates which are then removed, and then it is stored. This ensures that each data set only contains samples which appear once within that data set, and in no other data set. The data is then transformed for standard input into the ANN.

3.2 Data labels

Data labels are required for testing, validation and training processes within the framework and testing environments. Classifier inducers require training data to be labelled so that missclassifications can be detected. This allows the inducer to propagate a correction through the ANN and continue with training. This is how the '*learning*' process is achieved. Labeled data is also required for validation and testing purposes which are discussed more in Section 3.3.

[1] A manually vetted blacklist of phishing URLs, http://www.phishtank.com
[2] A directory of benign URLs that have been manually vetted. http://www.dmoz.org

As mentioned in Section 3.1, each data source manually vets samples before they are added to the database. This guarantees the malicious or benign nature of each sample. As a result, each sample is labelled according to its data source. Samples originating from Phishtank are labelled as malicious and that are represented as positive classifications, indicated by a 1 within both the framework, and the testing environment. Samples which were obtained from Open Directory are considered negative classifications which is indicated by a 0.

3.3 Data set generation

The final procedure executed on the data before they are used for training and testing is that of separation into the different data sets required. As already mentioned, each set consists of samples that only appear in one data set, and that data set only contains a single occurrence of that sample. The First set generated is known as the *Training set* which is used to train the classifier. The next set that is generated is also used by the inducer while training a classifier. It is known as a validation set and is discussed in Section 2.4.4. This set is used to validate the accuracy of the classifier to avoid over fitting to the training data set. A secondary function of this set is to allow for early stopping of training when the over fitting is detected (also discussed in Section 2.4.4). The final set generated by this process is known as the testing set. The testing data is used as data that has had no involvement in the training process.

3.4 Vector extraction

During training, both positive and benign data sets are loaded. They are then broken down into the constituent sections which make up a valid URL. Each section has an associated list which is used to store the words which will make up the Bag-of-Words (BOW) used as input to the ANN. Each section of each URL is separated into the tokens which make it up. These words are then inserted into their associated lists, even if they are duplicates. Only once this process is complete are the lists checked for duplicates inside themselves. This means that a word will appear once within any given list, but may be present in multiple lists. This distinction is important as lists may have words which are duplicates of words from other lists. This allows the classifier to assign different

weights to a word appearing in different sections of a URL. The lists generated by this process are then used as a mapping to create a bag of words for any specific URL.

When a URL is being used to create an input vector for the ANN, the BOW is used. A vector with the same single dimension as the BOW is created which will represent a set of binary inputs. Each value in the vector is assigned a 0 at initialisation time, indicating that there are no words present within the sample. Then, each word within each section of the URL is checked against the list of the BOW which is associated with that section. If that word is found at a particular index within that list, that index within the input vector is set to 1. For further lists, an offset is used to find the index for the associated list within the input vector. In addition to these binary inputs, there are 20 'obfuscation resistant' features which are used, as described by the authors. These features are continuous values which represent various metrics regarding a URL. These values are described as follows:

- The URL as a whole has two associated metrics: the number of dots present as well as the total length.
- The domain name length is recorded, whether or not it represents an IP address or uses a port number. The number of tokens and hyphens that are used are recorded, as well as the length of the longest token that appears within the domain name.
- The directory section of the URL is analysed for length, the number of directories traversed. Additionally, the longest directory name, the highest number of dots that appear within a directory name as well as the highest number of URL delimiters used in a directory name are calculated.
- The length, number of dots and the number of delimiters used within the filename are calculated and stored within the input vector.
- Within the argument portion of the sample, the length and number of variables defined are stored. Finally, the length of the longest variable is calculated and stored, as well as the highest number of delimiters used within a variable value.

Finally, where relevant, there are flag features which accompany some of these continuous values. These flags will be discussed in Section 3.4.1.

Table 3.1: Example of vector data pre and post normalisation.

x	71	3	26	3	0	18	0	0	0	0	0	0
normalised	0.034	0.085	0.111	0.142	0	0	0	0	0	0	0	1

3.4.1 Normalisation

Once the input vector has been constructed, the continuous variables which describe various metrics regarding the URL are normalised. Examples of the continuous variables include the length of the sample or the number of tokens in the domain section of the URL. These variables differ to binary variables which only have two possible values: 0 and 1. Some of these metrics have naturally higher values than others, such as length versus the number of dots used in the path section. As a result, within this example, length would have a naturally higher bias associated with it at the start of training. While this would eventually be trained out; it adds training time through iterations required for the ANN to correct this natural error. As a result, these values are all normalised to a range of between 0 and 1, where 1 represents a value equivalent to the highest value encountered for that field within the training data set. It is important to note that 0 does not imply a 0 value, but a value that is the lowest in the encountered data set. For this reason, for fields that are relevant, a flag value is associated, as mentioned in Section 3.1.1. This is a binary value where a 1 indicates that a continuous value field with a 0 value represents an equivalence to the lowest number encountered within the training data set. A 0 value for this binary field indicates that the associated continuous field has no input.

$$x_n = \frac{x - l}{h - l} \tag{3.1}$$

The normalisation values for the input vectors are calculated before training begins. Firstly, a pair of vectors which represent maximum and minimum values for each field is initialised to a 0 value for each field. Then each extracted sample vector is iterated over, checking each field for a value that is higher than the highest value encountered thus far, or lower than the lowest value encountered. When an input vector is extracted from a sample URL, these maximum and minimum values are used to calculate the normalised value of the field. The method by which these values are used is shown in Equation 3.1 where x_n represents the normalised value of x, h represents the highest value encountered in the training data set and l represents the lowest value for that field encountered in the data set.

An example of this normalisation process is shown in Table 3.1. The first six fields shown

in this table are continuous value variables, while the remaining six are the associated zero-value flags for each field. The first four fields are normalised as values which are somewhere inside the range of value encountered. The fifth value stays at 0, with the eleventh flag also set at zero, indicating a 0 value pre-normalisation. The sixth field is normalised from 18 to 0, with its zero-value flag set to 1, indicating that 18 is equivalent to the smallest value encountered during normalisation for that field.

3.5 Structure and initialisation

There are two basic components that were built regarding the ANN. The first component, called an *Inducer*, uses training data to create a perceptron (Kohavi, 1995). The inducer uses the lists of vectors extracted as per the method described in Section 3.1 to create the classifiers which will be tested and used throughout this document.

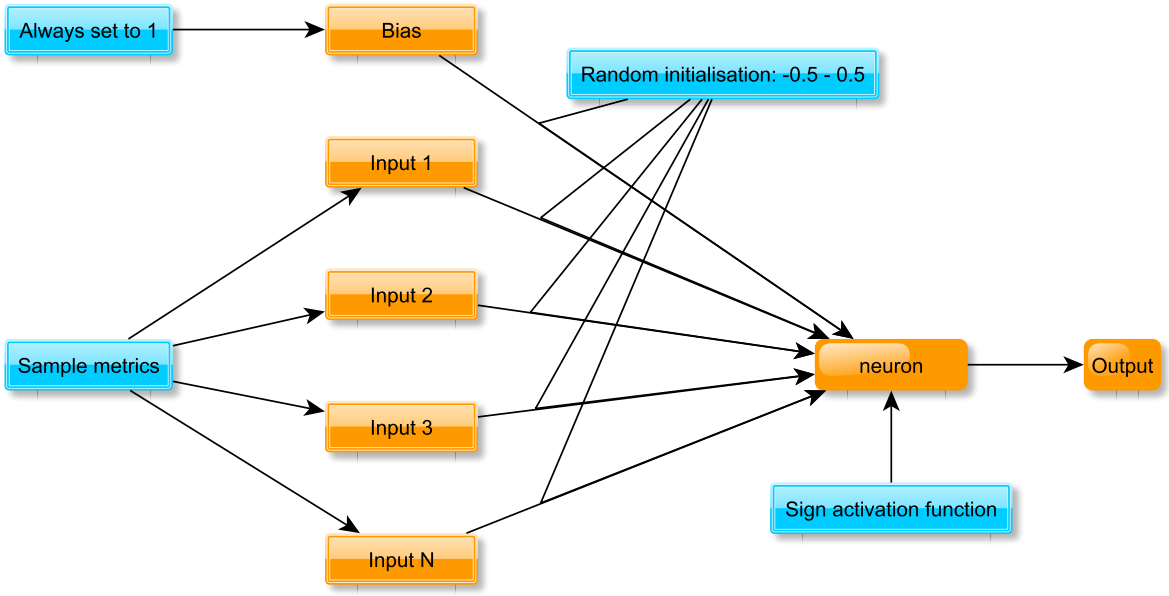


Figure 3.1: Structure and initialisation of the classifier.

It starts by creating an ANN with just two layers; the input and output layer, known as a perceptron. The size of the input vectors supplied to the inducer are used to determine the size of the input layer of this perceptron. The weighted connections from this input layer to the single neuron of the output layer are each randomly initialised to a value of between -0.5 to 0.5. The perceptron also uses a bias or threshold value as a hard limiter when combining the values from the input layer. Since there is no specific way to train

this bias, it is implemented as an extra input to the classifier which always uses the value 1 as its input. The bias value is then randomly initialised in the same manner as the other weighted connections. This has the result of being trainable in the same fashion as the other inputs, but still being able to shift the result in the same manner as the threshold is intended. Finally, when making classifications, the sign activation function is used as described in Section 2.4.1, resulting in a 0 for benign classifications and a 1 for malicious or positive classifications. Figure 3.1 shows this arrangement of the ANN.

3.6 Training

While this research was compiled and tested using a data set which was already downloaded, the framework discussed in Chapter 4 has the capability to download, preformat and store data on a scheduled basis. Training is performed using 18 000 samples, 9 000 of which make up the positively labelled set and the remaining 9 000 the negatively labeled set. As a result, the data set is considered balanced. The number of samples yields the best classifiers, the reasons for which are discussed in Section 5.4. These sets are interleaved on a 1 to 1 basis, making up a flat distribution across the training data set. The training algorithm used from training is the Online Perceptron (OP) as described in Section 2.4.2 using the equations shown in Algorithm 1. This method was chosen as it consists of the most common steps used in ANN training, is easily implementable and is one of the methods shown to be successful in Le *et al.* (2011). The classifier is trained on a single sample at a time in an online fashion, and updates the weighted connections on error, as per the OP method. At the end of each epoch (a complete iteration through the training data), the accuracy of the classifier is calculated using cross-validation. The training stops when a desired accuracy is reached or the maximum epoch is reached.

3.7 Validation

Validation of the model is performed at every epoch. This allows for the network to be tested for how well it will generalise to unseen data. This validation is done using a *hold-out validation* approach, where the training data set is made up of the bulk of the data set. Validation using this approach is discussed in Section 2.4.4. When the inducer finishes a training epoch, the resulting classifier is used to classify 2 000 labelled samples, balanced with 1 000 samples of each label. The final hold-out sample set is used for further testing,

discussed in Section 3.8. The results of this are used to determine the cumulative error of the ANN. Once this is known, it is compared to the previous best performing ANN generated. If it is better than this previous ANN, or being the first iteration through the data, the new classifier is stored. Once one of the end conditions are met, discussed in Section 3.6, and the training process is completed, the best seen ANN at that point is returned as the classifier result of the induction process. Using the accuracy calculated by this validation process to stop the training process is known as early stopping. By setting end conditions that can cause the process to end helps set goals and can speed up the process of training acceptable ANNs. A graph of this validation process during training is shown in Figure 3.2. It is a representation of the validated cumulative error generated by a classifier during training. This is built into the training process of the algorithm described in Section 4.5.4. It is implemented as a *SinglePassNetworkValidator*, described in Appendix A.13.

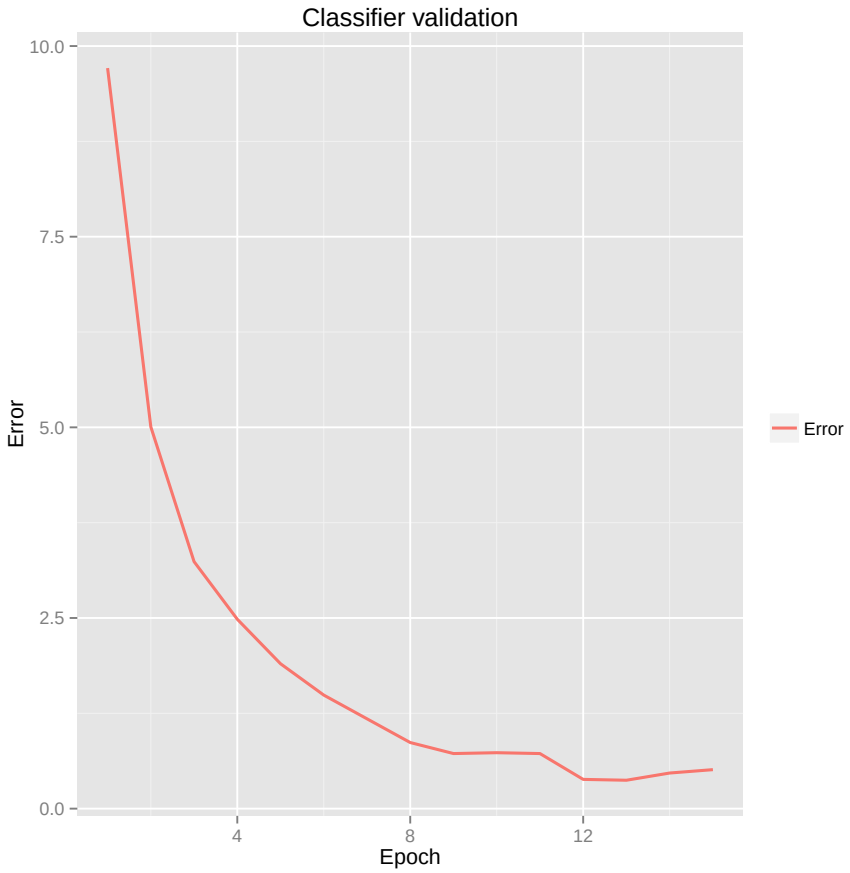


Figure 3.2: Validation graph of a classifier training run.

3.8 Testing

After the inducer has chosen the best performing ANN generated, it is then tested against another data set which has not been seen by the classifier. This serves as a second round of validation to test how well the classifier will generalise. The testing data set, like the validation data set, is made up of a balanced sample of 1 000 positive samples and 1 000 negative samples; to determine a unbiased view of the classifier's performance in terms of True Positive Rate (TPR) and True Negative Rate (TNR). These, along with other accuracy metrics, are calculated as per the methods shown in Algorithm 4 and are discussed in Section 5.2. The results obtained from this testing phase are averaged with the results from the validation phase. This average gives a good indication of how well the classifier will generalise and are returned with the associated ANN by the inducer.

3.9 Summary

Shown within this chapter has been how data is sourced from manually vetted sources and stored in a common format for the framework to use. These samples are loaded and a Bag-of-Words (BOW) is extracted from them. They are then analysed to find various lexical metrics which are normalised to a range of 0 to 1. The BOW and normalisation data are then used to create labelled input vectors which are then passed to an ANN inducer. This inducer uses this collection of vectors to train a classifier, with each training round being validated using unseen data. Finally, as a second round of testing, the classifiers are tested using a third slice of unseen data known as the testing set.

The logical structure and training mechanisms used to generate these ANNs discussed here, are implemented in a framework form. This framework forms the subject matter of the next chapter, Chapter 4. Within this chapter it will show how this is implemented in an automated fashion and how these classifiers may be transmitted, stored and used in client implementations.

CHAPTER

FOUR

SYSTEM IMPLEMENTATION

4.1 Introduction

The primary goals of this framework are to provide developers the tools to develop a method of generating classifiers trained to identify malicious URLs, provide these classifiers to the general public through a distribution service and to facilitate the ability to build URL screening mechanisms based on these classifiers. A requirement of these systems is that they should be accessible and usable by end-users, irrespective of whether the end-user is computer-security literate, but without introducing significant additional latency.

The framework described in this chapter has been designed as an implementation of these requirements, supplying a library which provides all the necessary interfaces required to describe this process on an ANN generation level. The framework also supplies mechanisms for storing and transmitting the generated classifiers in such a manner that client implementations may reconstruct them as well as the required input vector for each classifier. This library contains interfaces which describe the relationship between objects with independent goals within the ANN generation process and is based on the methods described in Chapter 3. For a description of these interfaces; see Appendix A. Within the project are implementations of all of these interfaces as well as implemented classes which describe these objects as data containers.

As previously mentioned in Chapter 3 and shown in Chapter 5; the advantages of using an ANN to make these classifications include the speed at which classifications can be done and the predictive nature of ANN. This results in a layer of protection that is not only adaptable, easily implemented and requires infrequent updates, but also has very simple and small targeted updates.

A valid ANN description for an OP includes a list of input weights, the number of inputs, as well as a bias value. The normalisation values include minimum and maximum values for all non-binary inputs; allowing them to be scaled to a value between 0 and 1; preventing inputs having unnatural importance resulting from their measurement scale. Finally, the list of possible inputs is encapsulated as a BOW which describes unique words found in each section of each URL that appears in the training data as binary inputs.

This is achieved using the number 1 to indicate the presence of a particular word within any part of the URL or a 0 to indicate a word's absence. This list of words is generated during the ANN training phase and is made up of five separate lists; each describing a collection of unique words found in that section. Those words are not necessarily unique to other parts of the URL, so a particular word could be contained in more than one BOW.

The short training period required to train classifiers (see Section 5.4.2) as well as the size and ease of distribution has resulted in a one to two week update cycle being suggested; which, while not a requirement, helps act as a rapid response mechanism to any shifts in trends in malicious URL creation and detection-avoidance techniques.

4.1.1 Chapter structure

In the following section; Section 4.2, the design of the high-level entities presented within this research and their interaction with each other are discussed. Section 4.3 discusses which platforms are specifically targeted by this research and the motivations for these decisions. The standards by which the code of each framework entity is developed are discussed in Section 4.4.

The first implementation as well as the final implementation of the *Network Generation Server* are discussed in Section 4.5. In the following section; Section 4.6, the service which stores ANN descriptors and updates is described in detail. Finally, an example implementation in the form of a Google Chrome browser extension called *Net Defence* is discussed in Section 4.7.

4.2 High-level design

The system design chosen to be implemented for this research is depicted in Figure 4.1. This design has three main entities; namely, the *Classifier Generation Service (CGS)*, the *Classifier Distribution Service (CDS)* and the *Net Defence* Google Chrome extension. The implementation is centered around the generation and use of classifiers trained to identify phishing URLs.

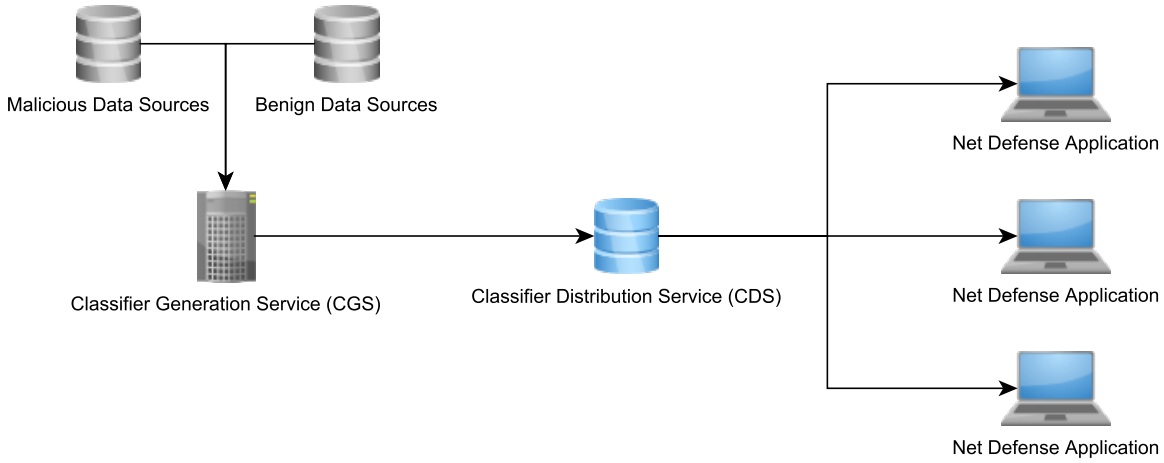


Figure 4.1: High-level infrastructure layout.

The *Network Generation Server* is where much of this research is focused as it is responsible for creating and training ANN as classifiers in a manner which facilitates rapid response to changing trends in malicious URLs with as little human input as possible. The library developed for facilitating this generation is used and executed on this entity. When the training of a new classifier has completed, the relevant data is sent to the *CDS*.

The CDS is the entity which acts as a mediator between ANN generation and client consumption of those classifiers and additionally as a separator of the concerns of classifier generation and classifier distribution. This results in a cleaner and more maintainable infrastructure.

This service has the ability to accept and securely store new data required to define an effective classifier as well the ability to distribute that data to different implementations of this research without needing to have implementation-specific protocols or communication methods. This service is only responsible for classification data updates and has no means of providing implementation (application) updates. As a result, there is only the need

for a single service to provide classifier updates to any application implementing this classification method.

The final entity depicted in Figure 4.1 is that of Net Defence. This entity represents an implementation of this research which uses classifiers sources from the CDS to identify malicious URLs within a browser. Specific implementation applications have been identified in Section 4.3, but because of the design methodology of CDS, unidentified applications wishing to consume updates may do so. The requirements of this implementation are discussed in Section 4.6.

4.3 Targeted classifier platforms

One targeted implementation for these URL classifiers is the area of browser extensions which have the ability to classify a URL when the user requests it, but before the URL is retrieved, thereby avoiding any possible damage caused by the malicious party.

Another implementation considered is that of an extension for a network proxy; for example, those common to businesses and educational institutions. The nature of these high speed classifiers make them ideal for use where a large number of URLs are requested, requiring a high throughput. This potentially helps as another layer of defence; protecting a corporate network from phishing attacks, and may also be used as a method of identifying these attacks.

The final use for these classifiers specifically considered in this research is that of email client extensions. These extensions would be able to classify URLs contained in emails and then remove malicious emails before the user has the opportunity to open them, or to warn the user when trying to access an email containing malicious URLs.

4.4 Code standards

The primary rule regarding code structure followed when designing this framework was that of separation of concerns between separate entities within the framework. As shown in Figure 4.1, each entity has a clear and specific role. This is followed from the code level of the framework. Each piece of software is layered so as to provide separation between

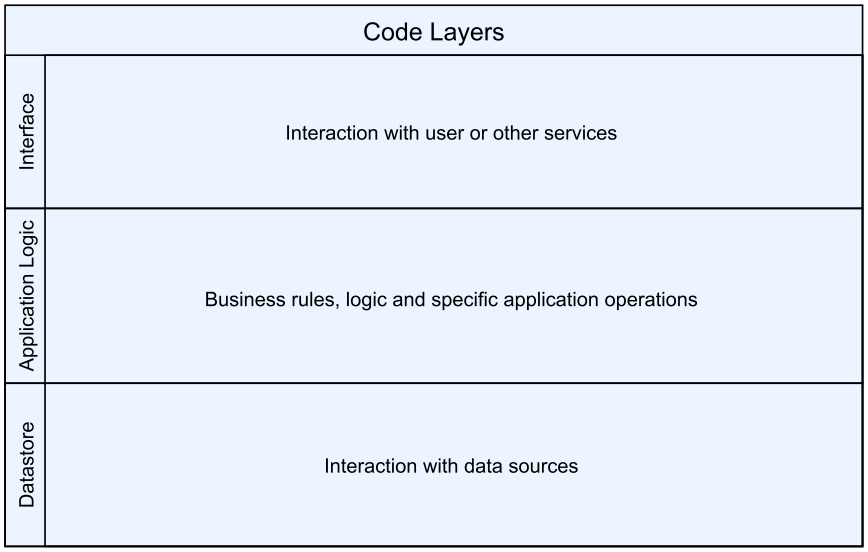


Figure 4.2: Application code layers.

functionality concerns, and is generally depicted by Figure 4.2. This is done to keep file sizes down and to make code cleaner, more readable and easier to understand.

The *Interface or Communication Layer* is concerned with the interaction of users or other programs. It handles the formatting and communication of data into and out of the application. In the case of a browser extension, this is the layer that a user interacts with to control the application. It formats data that it receives from the lower layers of the application in such a way as to make the information understandable and useful. It is also concerned with passing instructions from the user to the lower layers of the application and provides input validation of these instructions.

The second layer, the *Application Logic Layer*, is concerned with performing operations that make the application functional. These include instructions from the interface layer that affect data from the lower layer; the *Data Access Layer*. This layer is where the bulk of the functionality of the application is developed, and is where the majority of the code for this framework is situated as the generation of ANNs is considered a logical operation.

This decoupling of code is furthered by the use of the Dependency Injection design methodology. Where possible, interfaces are used as parameters instead of implemented classes. This approach is not always required as demonstrated in the case of JavaScript. JavaScript, being a loosely typed language, performs no type validation when a parameter is passed; it simply tries to access member variables and methods when they are needed. The name *Duck Typing* follows from this behavior (Python-Software, 2013), since, if an

object "looks like a duck and quacks like a duck, it must be a duck."

As a result of this behavior, JavaScript does not have interfaces defined as part of the language. Interfacing, or the ability to pass any object as a parameter as long as it has the required methods and members, follows naturally from *Duck Typing* and, as a result, JavaScript does not need to implement an interface explicitly.

Finally, the naming and formatting conventions used within the framework's code are shown and described in Table 4.1.

Table 4.1: Naming and formatting conventions.

Classes	UpperCamelCase is used. This is where words all start with a capital letter and are not separated. This is also known as Pascal case (Microsoft, 2013a).
Member variables	lowerCamelCase is used, which is the same as UpperCamelCase except the first word does not contain a capital letter. This is also known as Camel case (Microsoft, 2013a).
Methods	Like member variables, lowerCamelCase is used for methods.
Constants	Are capitalized with words separated by the use of an underscore.
Indentation styles	These are language dependent. In the C# sections, Kernighan and Ritchie (KR) style is used (Kernighan, 1988), while JavaScript code is written using the One True Brace Style (OTBS) variant.

4.5 Classifier Generation Service (CGS)

The original framework is written in Python due to the ability to rapidly develop applications and the ease of use. It was developed with the goal of being deployed on a Linux server, as discussed in Egan and Irwin (2012b,a), for ANN generation and is shown as a logical representation in Figure 4.3.

Python uses a Random Access Memory (RAM) management mechanism known as Garbage Collection (GC), which cleans dereferenced objects out of RAM on a scheduled basis. While this approach has both positive and negative aspects; it is not suitable for this application as the RAM requirements from object creation are extreme when creating thousands of objects, each with thousands of member variables.

The code is optimised greatly, making sure that URLs are only extracted to a descriptor at the point at which they were needed as input for the ANN. This tradeoff is considered necessary as the calculation can be performed faster than that of the Input and Output (IO) operations required by the operating system to perform swap file swapping on an already extracted data set. However, this does not have the desired result, as Python's GC system is not able to clean the dereferenced objects from RAM in a way that results in more acceptable RAM usage. While in development, a training data set of 11 235 URLs (a subset of the samples gathered and described in Section 3.1), each of which extracted into a BOW containing 18 124 items for each URL descriptor, the training process quickly used all of the host machine's eight GB of RAM, slowing training to an unacceptable rate.

For this reason, the decision was made to rebuild the project using a strongly typed, compiled platform. It was decided that the new ANN training framework would be built in Microsoft's C# language, running on a Windows .NET server. C# is different from Python in many ways, but most importantly it is a strongly typed, partially compiled language. The compiler is able to optimise code for efficient execution, resulting in the inducer being able to train a classifier in much more acceptable times. Using the C# application, the full data set of 22 000 samples was used without exceeding the test bed's memory capacity, never using more than 120 MB of RAM.

4.5.1 Python implementation

Shown in Figure 4.3 is the class structure of the Python version of the classifier generator (inducer) as described in Egan and Irwin (2012b,a). As discussed earlier, tests early in the development cycle showed that this implementation is not viable. As a result of this, this version of the code base is deprecated.

Within the framework; *Scheduler* is the entry point and serves the purpose of timing training events and coordinating the efforts of several objects instantiated to execute the training algorithm. It runs as a daemonized process on a Linux server and is controlled by administrators through the use of system signals to perform functions such as reloading configuration files and forcefully exiting.

The framework and application have been built following the *Observer* design pattern[1]. This pattern fires events to which *observers* respond. Each main element within the framework is an implementation of the base *Observer* class and executes on specific events,

[1] http://msdn.microsoft.com/en-us/library/ff649896.aspx

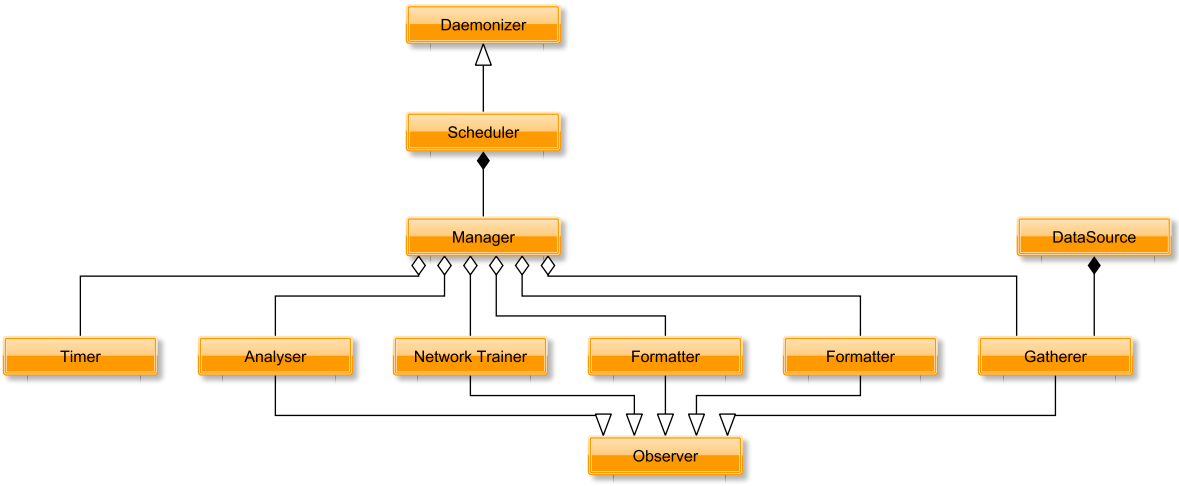


Figure 4.3: Overview of Python framework implementation.

such as the completion of the *Gatherer* - triggering the *Formatter* instance to execute. Another benefit of this design is that independent entities may be executed concurrently. In the case of this application, the *Analyser* may be run concurrently with all other processes after the training data has been fetched. The eventual outcome of this design was to implement a scheduling configuration mechanism through a file or web Graphic User Interface (GUI), but this was abandoned after the project was rewritten.

The first step in the algorithm is to instantiate the manager which is then responsible for instantiating the observers. The manager is also responsible for notifying each observer when events occur. The *Gatherer* observer runs twice a day, when the time is twelve o'clock, and fetches new data to be used for training the ANNs. This process is extendable through the *TimerObserver* which can fire new events based on the time.

The data fetched by the *Gatherer* is primarily sourced from Phishtank[2] and Open Directory[3] for malicious and benign data respectively, but can also be fetched from a list of alternative sources. The *DataSource* interface was written to define a usage contract for the framework, defining a single *fetchData()* method. Each entry in the list of sources for training data must have an associated implementation of this interface which is responsible for fetching data from that resource and formatting it into a usable format (only URLs should be present, separated by a carriage return and line feed). This allows the framework to fetch data from various sources, but be independent of the method by which these resources are fetched. Every implementation must be present within the *sources* directory and is executed by the *Gatherer* entity. Once the data are fetched, they are stored

[2]http://www.phishtank.com
[3]http://www.dmoz.org

within the database, implemented in MySQL, within the *preformatted_urls* table.

Each *DataSource* implementation must format the data in such a way that the *Formatter* object is able to interpret them. The formatting performed by the *DataSource* is implementation-specific and is responsible for extracting the relevent data from the data provider. This can be an action such as extracting data from a compressed archive followed by field extraction from the particular format in which the data are stored. There are two-phases of formatting within this implementation of the framework. The first is the extraction that is performed by the *DataSource*, followed by the formatting that is performed by the *Formatter* object. This final phase of formatting serves to convert the data into a format that is usable by the *Analyser* and the *Network Trainer* observers. This two-phase formatting allows the requirements of the framework for the data format to change without having to re-implement each *DataSource*. Conversely, a data provider may change the method by which it exposes or deploys its data, leading to a refactoring of the associated *DataSource* but no other objects further down the observer pipeline due to this decoupling.

The next observer which executes is the *Network Trainer* which performs the primary task of the framework. Firstly, it selects a range of input data from the database. This selection is controlled through the configuration of the framework, allowing the operator to adjust the selection size, time span or random selection methods used to select the training data. The BOW is then extracted and the data are ready for use. The *Network Trainer* then instantiates a *Neural Network* instance as well as a *Validator* instance. Once the data are extracted and the required objects have been instantiated and initialised, the *Network Trainer* then begins training the ANN and validating it at each epoch until the ANN reaches some exiting criteria, such as reaching the desired accuracy or reaching the epoch count limit. When an acceptable network has been trained, it is serialised along with its input data and stored in the database. Within the *Normandy Framework*, the *Network Trainer* is the observer which is meant to be subclassed and replaced in order to apply new or improved training methods, further increasing the flexibility of the framework.

While the *Network Trainer* is executing, a second observer, called the *Analyser*, is running concurrently. This observer is meant for research purposes, rather than directly aiding in the training and distribution of these classifiers. Its primary purpose when originally developed was to resolve domain names to IP addresses, and then, using unique IP addresses, geolocate these addresses. It was found that further analysis of the training data was required and this observer fulfils the role perfectly. As a result; the *Analyser*

observer acts as an analysis manager, which includes files developed by researchers that are present within a directory specified in the configuration. These files must contain a class which subclasses the *AnalyserTask* class. This allows the *Analyser* to import and instantiate each analysis task in an array and execute them. Since some of these tasks could potentially take a substantial amount of time to complete, the *AnalyserTask* class implements a Python thread and is run concurrently with the other tasks which have been loaded. An advantage to this dynamic loading of tasks is that it allows researchers to develop, add and remove research activities to the system without requiring the system to be taken offline. This is done through the use of a system signal (SIGHUP) which invokes *Normandy* to reload its configuration files.

The final Observer, at the time this research was compiled, is the publisher. It has the job of taking the JavaScript Object Notation (JSON) ANN descriptor and the BOW and compressing them into a zip format. It then pushes these to the database of a web service which may be used by client side implementations. These clients simply query the web service and download a new version of the ANN once a week. They may then decompress the zip file and replace their current ANN description and BOW used to break up URLs that they encounter. This is a useful approach as it is platform and implementation independent as it only contains a list of words and a list of input weights.

However, during the testing of the *Normandy Framework*, as previously mentioned in Section 4.5, it became apparent that the Python platform is not able to support the requirements of the training process due to the RAM requirements and the way in which Python's Garbage Collection (GC) mechanism works. It may have been possible to make the framework work, but this would have been an exercise in Python, rather than research into a framework for delivering ANNs capable of detecting malicious URLs. For this reason, the code base was deprecated and started ab initio, as described in Section 4.5.2.

4.5.2 Neural network generation library

As a result of the limits identified in the Python implementation, the framework was rewritten in Microsofts' C# language. During testing this framework proved to be far more reliable and able to cope with the requirements of the ANN generation process. This is discussed in Section 4.5. As a result of this, this code was developed to a stage that is acceptable for the purposes of this research and is more mature and complex than the Python implementation. The class hierarchy of the final implementation of the *Network Generation Server*, as used in Chapter 5, is shown in Figure 4.4.

The use of interfaces instead of solid implementations is in accordance with the *Dependency Injection*[4] design model which allows implementations to be swapped out without affecting other components of the application. For this reason, the *NetworkTrainingConfiguration* class was designed and reads a configuration XML file which dictates which implementations of interfaces to load. These implementations are then loaded as publicly accessible member instances of the *NetworkTrainingConfiguration* instance. An example of this configuration file is shown in Appendix B.

This configuration object can then be passed to any object within the application and may be used to access the specific implementation that has been instantiated as determined by the configuration XML file. This not only allows a project to be configured with different behaviours without the need to recompile, but it also allows a single parameter to be passed to all objects requiring access to other objects, without the need to debug and validate those options. This is as a result of a single configuration instance being loaded, which allows for a single point of validation for all configuration. This separating of the concern of configuration management from ANN generation leads to a cleaner and more maintainable code base.

4.5.3 Classifier Generation Service code structure

As mentioned in Section 4.4, the *Network Generation Server* is designed with a specific layering within the code. This layering makes maintainability and bug tracking a simple task as there are clear delineations of responsibilities between components. Shown in Figure 4.5 is the general layering of the application. Seperation is not only achieved on code level, but at a file level as well. The application exists as an *exe* that calls methods and libraries available from the *NeuralNetwork* and *Lib* Dynamic Link Library (DLL).

The *Network Generation Server* is implemented as a demonstration implementation of the library available from the *Lib* DLL. This DLL provides several interfaces which define the interaction contracts of several elements identified as requirements in the process of generating classifiers for use as malicious URL classifiers. Also contained within this library are implementations of all of the interfaces provided except for that of the *Network Inducer*, which are discussed in detail in Appendix A.

Within the *NeuralNetwork* DLL is an example application that implements the *Network Inducer* interface and manages all of the associated tasks, such as data collection and ANN

[4]http://msdn.microsoft.com/en-us/library/ff921152.aspx

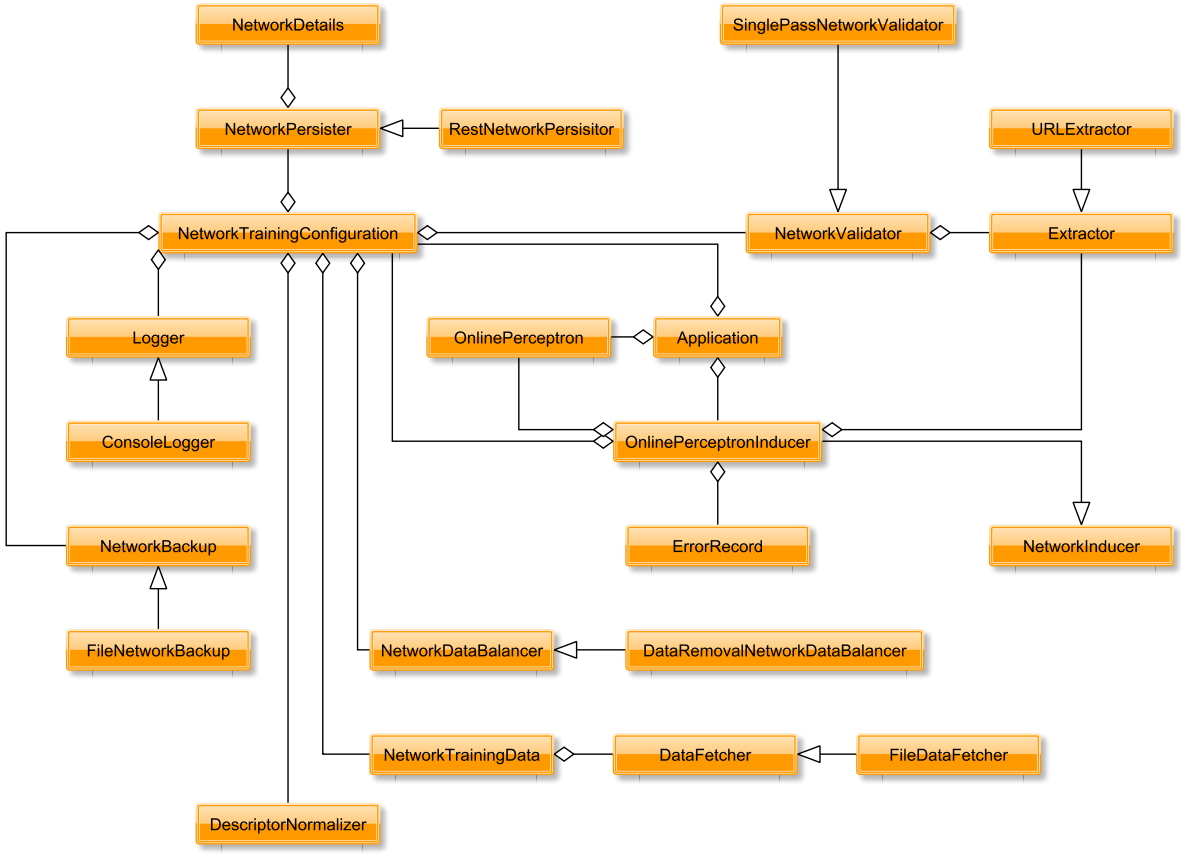


Figure 4.4: Overview of the Classifier Generation Service (CGS).

training. This DLL acts as the application logic layer of the code structure while the *Lib* layer provides mechanisms through the *DataFetcher* interface, the *FileDataFetcher* class as well as the *NetworkPersister* interface and its associated implementation which act as the data layer of the application.

4.5.4 Network generation algorithm

Th Application class acts as the entry point to the program. This class is responsible for bootstrapping the training process by loading the required data and passing it to an inducer which generates and persists the newly trained ANN in the form of an update.

The first step is to instantiate a *NetworkTrainingConfiguration* object. This object is found within the *Lib* library and fetches settings from a configuration file called "configuration.xml". An example of this configuration file can be found in the Appendix B.1. This configuration file is at the core of the *Dependency Injection* design methodology

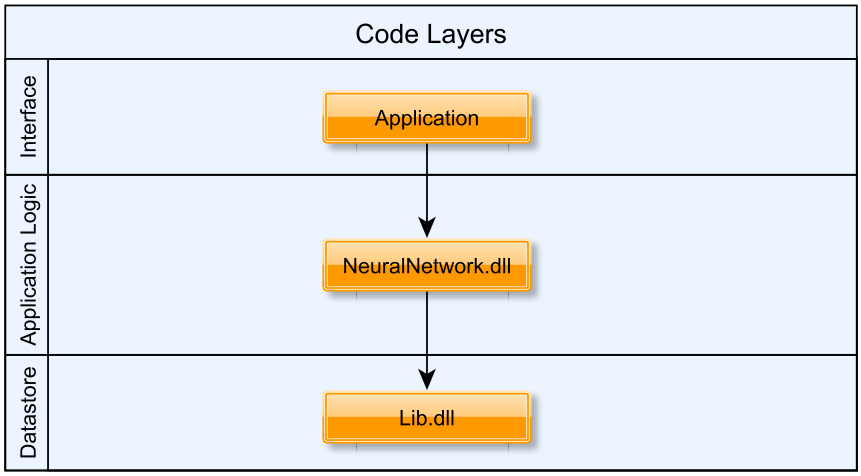


Figure 4.5: Layout of code structure withing the Classifier Generation Service (CGS).

implementation of this project. It specifies which implementations of the various interfaces to instantiate and to pass to the inducer. It is by using this file that a developer or researcher may implement alternative methods of performing various operations of the framework by simply implementing the interfaces provided in the library and specifying those instances within the configuration. This configuration also specifies parameters regarding ANN training; such as the maximum number of epochs allowed, the desired error rate and the learning rate of the training algorithm. Also identified here are paths for training and validation data. The other fields that are found within the configuration file will be identified when their associated interfaces are mentioned within this chapter.

Once the configuration has been loaded and all of the required objects instantiated, the application creates an instance of *NetworkInducer* class. This interface is instantiated as a *OnlinePerceptronInducer* from the *NeuralNetwork* DLL and is then initialised by passing it the *NetworkTrainingConfiguration*. The inducer then has all of the instances and data required to train an instance of *OnlinePerceptron*, which is the process that is then executed through the *generateNetwork* method and is discussed below.

4.5.5 Perceptron generation using the NerualNetwork DLL

The *NeuralNetwork* DLL, as discussed in Section 4.5.3, acts as a demonstration implementation that uses the *Lib* DLL. The only requirement of this library is that a *PerceptronInducer* interface is implemented. The reason for this is that every implementation and set of requirements are different. Within this DLL this implementation is called the

OnlinePerceptronInducer and is intended to load training data from files, generate an ANN and then to store this ANN in the form of an update exposed via a web service that is accessible by client consumers.

The *OnlinePerceptronInducer* is instantiated by the program and initialised with an instance of the *NetworkTrainingConfiguration* class. It is from this class that intended implementations of the library's interfaces are meant to be accessed. Once this class has been initialised, the *generateNetwork* method is called and follows the following high-level algorithm:

1. Extract the BOW and lexical features of URLs contained in the data files and save them as a list of input vectors for the classifier.
2. Normalise this list of inputs to a range of zero to one for each continuous variable within the lexical features portion of each input vector.
3. Train the classifier using these normalised lists and a second set of normalised lists used for validation. These lists are generated by splitting the first list.
4. Persist the classifier. This is done by sending it to the update service described in Section 4.6.

Extraction of data is done through the use of the *NetworkDataFetcher* interface which is available through the *ntd* member variable of the configuration instance. This interface provides a single method called *fetchDataArray* which loads the data and returns it as a single array element per data item. Within this implementation, each data set is loaded through its own instance of this interface implementation and loads the data from files specified as paths in the configuration file. Once the training and validation data have been loaded and balanced using random under-sampling (as discussed in Section 2.4.3), they are used to create instances of *URLExtractor* which will be used as an input to the classifier during training and validation. Each of the BOWs are then generated using the collection these extractors and are then used to insert a map of existing words into each of the extractors as part of the input data.

After extraction is completed, normalisation of the training data begins. Normalisation is handled by an object called the *DescriptorNormaliser* which is implemented in the library. Like all other objects within this application, a reference to this object is found in the configuration instance and is initialised with all the descriptors (extractors) already

loaded. Once the normaliser is initialised, it is used to normalise both the malicious and benign extractor lists to a range of 0 to 1 for each continuous field.

At this point, the inducer which is an object which has the specific purpose of generating (inducing) an ANN, has all the data that is required and is in the correct format. As already mentioned, this implementation uses the Online Perceptron model to create an ANN capable of classifying malicious URLs. This training method is discussed in Chapter 3. The only addition to this algorithm is that the *Logger* interface is used to create user interface outputs as an indication of progress and accuracy during the training process. After each iteration through the training data set, the *NetworkValidator* instance is used to validate the ANNs accuracy using independent data known as the validation data set. After validation, a copy of the classifier and its performance are stored. Once the inducer exhausts the number of allowed epochs or reaches the ANN's goal accuracy, training ceases and the validator's best performing ANN is chosen as the final trained ANN.

The final step after training is to store the ANN and make it available to end-users to use within applications implementing this classification method. A *NetworkDetails* object is created and stores several statistical metrics regarding the classifier which are obtained from the ANN validator. An instance of *NetworkPersistor* is fetched from the configuration and is passed the configuration, network details object, the classifier, the BOW, weights and normalisation data through the *persist* method. Like the other objects within this library, this instance is implementation-specific and can be implemented through any method. Within this research it is implemented as a series of HTTP POST (an HTTP verb used to add data to a service[5]) requests to the REST service which adds persistence to the framework. Client implementations can access this REST service to request updates. Each of these data sets is transmitted in this fashion as they are required by clients to rebuild the classifier exactly as it was trained; with the same weighting as well as the method by which to build input vectors for the classifier from requested URLs.

4.6 Classifier Distribution Service (CDS)

A classifier implementation was developed for use within the Google Chrome browser as an extension, the purpose of which is to autonomously check URLs as they are requested by the user when following links. Chrome extensions are normally updated through the Google Chrome Application Store. This allows Google a degree of control over the content

[5]http://tools.ietf.org/html/rfc2616

distributed through updates; providing consumers a level of security through Google's security mechanisms.

One of the goals of this research was to design a distribution service whereby updates to the classifier could be easily distributed without requiring implementation-specific update services and procedures. This decoupling of classifier updates from implementation specific updates allows for a single online service; which is able to provide new classification data, for all implementations of this research while being implementation-agnostic. As a result of this, this service could provide updates to implementations such as a fully featured classifier, an email scanner, browser plugin and proxy server plugin simultaneously, without needing reconfiguration. Secondly, a single online service for all classifier updates means that malicious parties cannot seed the internet with false services and thereby circumvent this research by claiming higher performance classifiers.

This has the added benefit of allowing developers to maintain their specific implementations without needing to be trained in the generation of, as well as the distribution of ANNs such as in Chapter 3. This guarantees universal performance of these classifiers as one body is in control of their generation and distribution. As a result of this requirement, updates only provide specific information to client classifiers. This information is summarized in Table 4.2.

Table 4.2: Resources available via classifier update service.

bias	The classifier bias shifter.
weights	The weight of each input within the classifier.
normalisation	Maximum and minimum values encountered during training.
BOW	The Bag-of-Words used to build the input vector.

While the bias and weights resources directly describe the ANN, the normalisation and BOW resources are equally as important. Normalisation is required by the ANN, as mentioned previously, to scale all inputs to the classifier to a range of zero to one. This prevents one input from having a naturally higher weight as a result of its measurement scale.

Within Table 4.2, the BOW resource is a general reference to a collection of resources which make up the completed BOW. A more specific description refers to multiple BOWs which contain unique words describing the top level domain, directory, file, file extension and arguments sections of an input URL.

The following sections describe the approach adopted to designing and implementing

the update distribution service, the data storage method and the intended method of consuming this service by client implementations.

4.6.1 Design

The primary goal of this service is to provide client implementations with frequent classifier updates while being implementation-agnostic, allowing these implementations to effectively classify malicious URLs. As a result of this, as well as the objective that states that an implementation should be effective when users are not necessarily computer or network security literate; implies that these updates should be freely available and through a transparent process; not requiring any user input.

To be able to supply these updates to client classifiers as well as to streamline the process from generation to detection; this service is also required to be able to programmatically receive these updates from CGS. As a result, a RESTful style web service, often referred to as a RESTful API; was chosen to provide these classifier updates as it allows for verbs to be used to implement programmatic pushing and fetching of data from the service. REST web services are discussed in Section 2.5.2. Typically, REST APIs communicate using JSON to represent resource objects and are accessed using the HTTP verbs: POST, PUT, GET and DELETE.

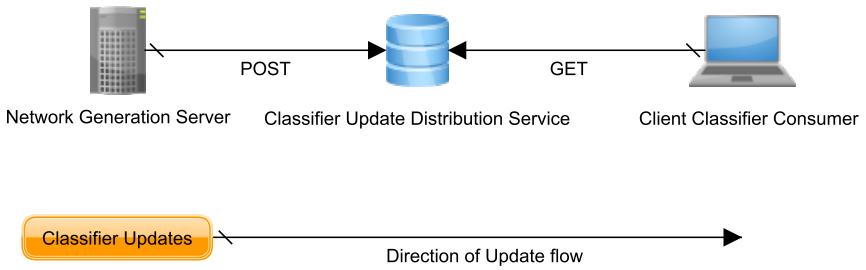


Figure 4.6: Flow of updates via Classifier Distribution Service (CDS).

Figure 4.6 shows the intended method of operation for the *CDS*. The *Network Generation Server* generates a new classifier and all the describing information as object instances. These objects are then JSON encoded and sent to the relevant resource on the *CDS* via the HTTP POST verb. When receiving a valid POST request, the service stores the resource for retrieval when an application requests an updated classifier definition. This is done by the client executing a series of HTTP GET requests on the relevant resources, causing the service to respond with the relevant JSON encoded data.

This method of exposing updates to client applications results in an update service that only needs a single implementation as it does not need to know anything about how to communicate with specific clients. This implementation makes updates available through URLs, each pointing to a resource that can be acquired by clients through the use of the HTTP GET verb, and updated by the CGS through the HTTP POST verb. The only requirements of this service are that clients need to be able to make HTTP requests and be able to handle JSON data.

4.6.2 RESTful API logical implementation

The RESTful web service is implemented using the *web.py* framework[6] (from here on, written as webpy) available for the Python programming language. This framework was chosen because, when coupled with this implementation and being written in Python; it is a cross-platform implementation, meaning that the update server may run on any operating system that supports Python. Secondly, webpy makes the task of developing REST services trivial due to the way that individual HTTP verbs are programmed.

All of the resources made available through this API are grouped by concerns. As a result, two distinct groups are defined; the *networks* group and *bows* group. The ANN's bias, input weight vector, normalisation values and validation details are considered part of the *networks* group, while all five distinct BOWs vectors constitute the *bows* group.

Shown in Figure 4.7 is the complete logical layout of resources available within the *CDS*. As mentioned above, the intended usage of this API is for the classifier generator to POST updated resources to the service, causing it to store these updates, and for client consumers to GET updated classifier data.

4.6.3 API URL structure

URL Structure is an important consideration when designing an API. This structure defines the way in which clients consume the service and as a result, defines how developers have to interact with the API. A well designed API structure makes it easier to understand the logical layout of the API. This assists developers in learning how to use it correctly and making code more maintainable as URL transactions are easier to read. Within the *CDS*, the following URL structure is defined:

[6] http://webpy.org

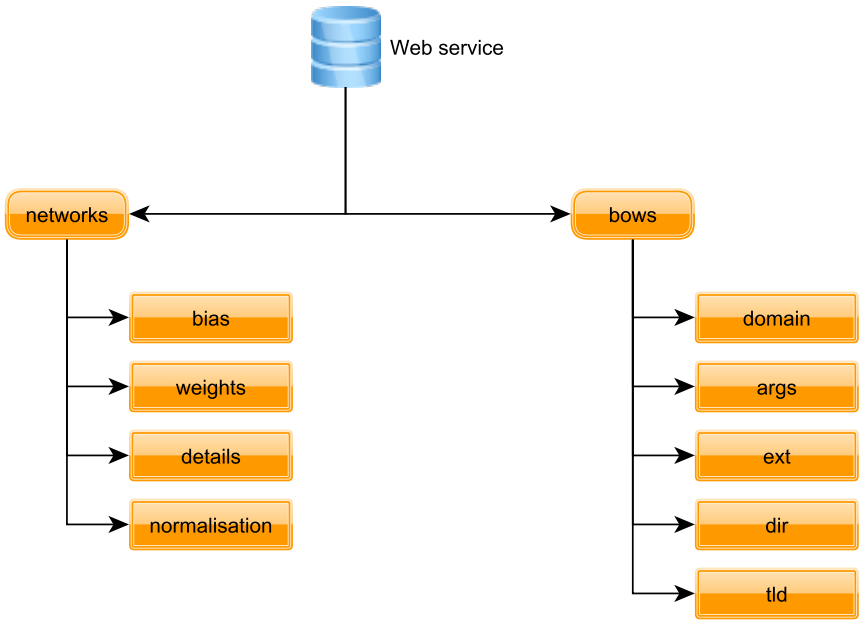


Figure 4.7: Logical resources available in Classifier Distribution Service (CDS).

http://domain:port/resource-group/resource-id

This structure allows a client to easily choose between related sets of data, resulting in a cleaner, more understandable access method than a simple collection of all resources available. The *resource-group* has two options: *networks* or *bows*. The *resource-id* allows the consumer to specify which resource to either GET or POST to. An example of these actions is that the CGS may POST a new set of *bows* and networks. A client implementation then uses the GET verb with the same URL request which causes the service to respond by sending the data to the client. Another benefit of grouping resources in this fashion is that it makes the API more maintainable and expandable. Having a *statistics* resource group that holds resources concerned with network usage, or a *user-logs* group where clients can report false positives and negatives to the service for incorporation in the training data would be trivial to add without requiring the structure of the URL or any naming of resources to change is trivial to implement.

Another important aspect to the APIs URL structure is that it is used by webpy to map resources to particular URLs. When a user requests a URL, webpy checks the defined mappings and passes control of the request and response logic over to that resource. Resources are represented as a single class per resource within webpy. Each class must have the appropriate methods named after the HTTP verbs which should be called when a request is made for the resource that the class represents.

4.6.4 API code structure

Like all the code written within this project, the API structure follows the code standards defined in Section 4.4 and is built in three separate layers which constitute the interface, application logic and data layers. The way in which this is implemented within the *CDS* is shown in Figure 4.8.

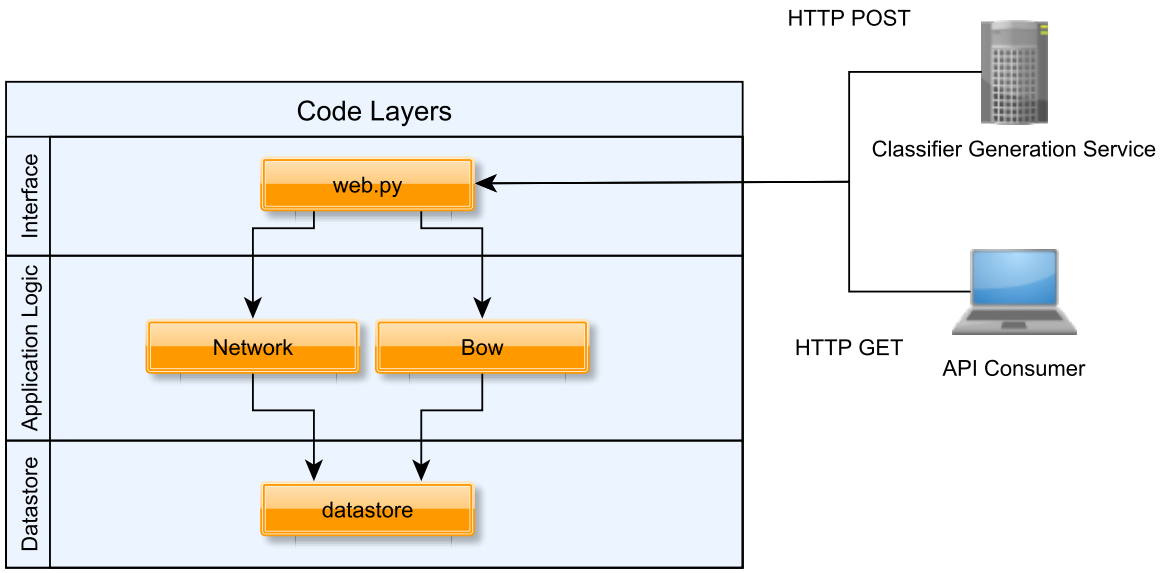


Figure 4.8: Layout of code structure within the Classifier Distribution Service (CDS).

A simple file called *index.py* serves as the entry point for the program. This file also contains an array of URL mappings as discussed in Section 4.6.3. These array entries map URLs such as 'networks/bias' to the *Network* class. This is considered the interface layer as it defines how the clients interact with the lower layers of the application.

The second layer, the application layer, comprises of two main classes. These are the Bow and Network classes and are responsible for saving and fetching resources requested by clients via the datastore layer. Both of these classes have *POST* and *GET* methods which are invoked by webpy when the appropriate resource-group is requested. Together, these two classes are responsible for the resources defined in this research. They both have statically assigned lists of resources which are valid to them as resource-groups. This way, no incorrectly named resource can be requested or saved. Secondly, this implementation does not allow for versioning and leads to a simpler API interface. The framework would be simple to extend to include versioning through the URL structure defined.

There are two other classes implemented for use within this layer that are not shown in

Figure 4.8. These classes are the *BowList* and *NetworkList* classes and are responsible for listing available resources within each resource group. Further, there are several exception classes implemented for use within the API that belong to the application logic layer and extend Python's web *HTTPError*[7] class.

4.6.5 Data storage

Within this implementation, each resource is stored in a file with the *.json* extension. This is a one-to-one representation of what data are sent from the *Network Generation Server*. This approach was chosen due to its simple implementation method and ease with which it can be debugged. This would ideally be updated to be implemented with the use of a database. This update would be done by replacing the data layer with one that uses a database storage. The new implementation is required to match the interface contracts defined for the data layer. The framework would then instantiate the new implementation and pass it via the dependency injection methodology. A database approach was not used in the testing environment as the additional complexity was deemed unnecessary in an academic environment.

4.6.6 Consumer access

As mentioned in Section 4.6.1, a client wishing to consume this service to obtain classifier updates is only required to be able to make HTTP requests, which is inherent in any code wishing to classify URLs, as well as be able to interpret JSON data. To use this service, a client makes a series of HTTP GET requests as shown in Figure 4.9, and receives JSON data with the relevant response.

Typically this response would be instantiated in the form of an object instance, representing the resource in a usable way within a particular language. This object is then persisted in local storage by the client for use when URL classifications are to be made. This data should be refreshed on a scheduled interval such as on browser restarts or on a temporal basis. Persisting is important as the data should not be fetched for every use, as this would incur significant latency, therefore negating the benefits of a local classification solution. Requesting the *networks* resource-group returns the JSON data shown in Listing 4.1 which contains four elements (bias, weights, details and normalisation) intended

[7] http://webpy.org/docs/0.3/api

```
127.0.0.1:15593 - - [08/Jun/2013 11:43:45] "HTTP/1.1 GET /networks/bias" - 200 OK
127.0.0.1:15596 - - [08/Jun/2013 11:43:45] "HTTP/1.1 GET /networks/details" - 200 OK
127.0.0.1:15595 - - [08/Jun/2013 11:43:45] "HTTP/1.1 GET /networks/normalization" - 200 OK
127.0.0.1:15597 - - [08/Jun/2013 11:43:45] "HTTP/1.1 GET /bows/args" - 200 OK
127.0.0.1:15596 - - [08/Jun/2013 11:43:45] "HTTP/1.1 GET /bows/ext" - 200 OK
127.0.0.1:15595 - - [08/Jun/2013 11:43:45] "HTTP/1.1 GET /bows/tld" - 200 OK
127.0.0.1:15598 - - [08/Jun/2013 11:43:45] "HTTP/1.1 GET /bows/dir" - 200 OK
127.0.0.1:15593 - - [08/Jun/2013 11:43:45] "HTTP/1.1 GET /bows/domain" - 200 OK
127.0.0.1:15594 - - [08/Jun/2013 11:43:46] "HTTP/1.1 GET /networks/weights" - 200 OK
```

Figure 4.9: Classifier Distribution Service (CDS) logging a request from a client for updates.

as navigation options for the API and do not represent data used in the construction of a classifier. This is similar to requesting the *bows* resource-group without a resource-id as it also returns a list intended for navigation purposes.

Listing 4.1: Server JSON response when requesting the networks resource group.

```
{
  "bias" : {
    "href" : "http://cds-server:8664/networks/bias"
  },
  "weights" : {
    "href" : "http://cds-server:8664/networks/weights"
  },
  "details" : {
    "href" : "http://cds-server:8664/networks/details"
  },
  "normalisation" : {
    "href" : "http://cds-server:8664/networks/normalisation"
  }
}
```

Listing 4.1 contains resources that are grouped as they represent a valid classifier description. Each element is returned in an unexpanded format, represented by a URL for each resource. This allows each resource to be fetched in parallel, as well as allowing updated resources to be uploaded simultaneously.

As a standard, every resource available through this API has a *href* field which represents the direct URL to that resource, even if that resource is already in an expanded state. This reduces application complexity for client implementations as well as simplifying debugging. This rule is also followed when POSTing a resource to the API; a copy of the resource in expanded state is returned to the sender, including an *href* which links to that resource. Again, this reduces client implementation complexity and increases ease of debugging the service.

Each resource returned in the request that Listing 4.1 represents a defining characteristic about the classifier. The resources defined are the network bias, the network weights (which imply network input size) and the normalisation values required to normalise an input vector for the classifier. The *details* resource is a resource that is available to clients as an extra information container describing the parameters used as well as the statistical results of the classifier training and validation and is shown in Listing 4.2. While the details and normalisation resources are not defining characteristics, they are still grouped within this resource-group as they describe information about the classifier. It is felt that it is not necessary to create a meta resource group specifically for this purpose.

Listing 4.2: Server JSON response when requesting the networks/details resource.

```
{
    "href" : "http://cds-server:8664/networks/details",
    "data" : {
        "fp" : 4.674,
        "fn" : 9.794,
        "features" : 16537,
        "benign" : 5365,
        "datasetsize" : 10599,
        "malcious" : 5234,
        "epoch" : 82,
        "specificity" : 95.326,
        "date" : "04/26/2013 19:03:59 PM (UTC+02:00)",
        "accuracy" : 92.791
    },
    "network" : "details"
}
```

The *fp* and *fn* fields represent the *False Positive Rate* and *False Negative Rate* of the classifier validation process as measured by the CGS with unseen data after training is completed. The *features* field indicates how many features, including both BOWs and lexical features, make up the total number of inputs to the classifier. The *benign* and *malicious* fields indicate the number of entries were used in the generation of the ANN and together make up the *datasetsize* field. The ANN training validation indicates which of the iterations of the classifier through the training dataset is the most effective classifier, and the *epoch* field indicates this iteration number. The fields *specificity*, *accuracy* and *date* are all obvious in what they represent.

The resources returned when requesting the *bows* resource group are *domain*, *args*, *ext*, *dir* and *tld*. Each resource in this list represents a BOW. These are considered separate and distinct from the resources which constitute the networks resource group as they do not directly describe the classifier. The BOWs are concerned with how an input vector is constructed instead of how it is processed.

While this is a simple implementation, it allows for clients to completely rebuild classifiers generated previously, as well as enabling clients to generate input vectors that are normalised to the values used in the training and validation of these classifiers. In an

effective use case, a client application would request all of the resource URLs shown in table 4.3 from the *CDS*.

Table 4.3: Resources required to rebuild a classifier and its input vectors.

Classifier bias	http://hostname:port/networks/bias
Input weight vector	http://hostname:port/networks/weights
Input normalisation vectors	http://hostname:port/networks/normalisation
Domain section BOW	http://hostname:port/bows/domain
Arguments section BOW	http://hostname:port/bows/args
File extension section BOW	http://hostname:port/bows/ext
Directory section BOW	http://hostname:port/bows/dir
Top level domain section BOW	http://hostname:port/bows/tld

In a production environment, the hostname would not be localhost, but the hostname of wherever the *CDS* is hosted. The port number was also chosen for testing purposes, has no special meaning and may be changed if required within a production environment.

4.7 A classifier client implementation: Net Defence

The final piece of software developed as part of this research is a *Client Classifier Consumer*. This software is meant to protect end-users from malicious URLs by employing ANNs to identify these URLs and is named the *Net Defence*. This software has been developed in the form of a Google Chrome extension, as it is the most visible method of demonstrating and testing the capabilities of such classifiers, is likely to be used in every day life and is not hidden behind IT infrastructure.

To summarize the goals of this project; it needs to be as transparent to the user as possible during daily usage, but must be able to stop, or at least warn, the user when requesting a URL classified as malicious. This must be achieved without incurring significant additional latency through lookups and must be able to query the *Classifier Update Distribution Service* periodically for updates as well as be able to interpret JSON data. Google's Chrome browser has been chosen as the target platform due to the ease at which extensions can be built and the developer tools available for debugging extensions. Finally, extensions are available that extend the developer tools within Chrome specifically for use with the AngularJS framework.

4.7.1 Design

Google's Chrome internet browser allows developers to create and add functionality to the browser via different approaches and are discussed in detail on the overview page at Google (2013a). These extensions are written as web pages and consist of *HTML5* (and any associated web programming files such as *CSS*), *JavaScript* and a *manifest* file that describes the structure as well as other meta data regarding the extension.

With the design requirements of this research being that the implementation must be transparent to the user when no malicious URLs are being requested; the extension is designed to run as a *background* script. This is a script that is run when the browser is opened and does not require user interaction to start or to be effective. This script, however, is required to be able to fetch data from the REST service discussed in Section 4.6 and as a result needs to be able to store JSON information regarding the classifiers as well as false positives and false negatives that the user may identify.

Users must have the ability, through the client interface, to be able to flag when a page has been classified as benign incorrectly (False Negative (FN)), or to identify when a URL has been incorrectly classified as malicious (False Positive (FP)). Additionally, the user must be able to modify the behavior of the extension when it identifies a threat, as well as be able to generate rules for sets of misclassified URLs or URLs to be ignored. As a result of these requirements; the Net Defence extension has several GUIs that the user may interact with and are discussed in Section 4.7.5. Each of these GUIs is designed as an HTML template with a JavaScript file that implements the functionality implied by each GUI.

Since Chrome extensions are built using JavaScript as the engine which provides logic, Net Defence extension is built using the AngularJS[8] framework. The AngularJS code is used in the main *options* GUI which affects all aspects of the extension's behavior. AngularJS allows for the code to be separated in a Model-View-Controller (MVC) fashion,[9] using HTML files for the view, *Controller* objects as controllers and resources and services to be used as abstractions of the model concept. AngularJS provides several useful mechanisms used by the extension, the first being the use of *Resource* entities to communication with REST web services. *AngularJS Resources* are extensively used throughout the extension to implement the ability to update classifier information from the *Classifier Update Distribution Service*.

[8] http://angularjs.org/
[9] http://docs.angularjs.org/tutorial/step_02

Another mechanism used extensively within the extension is that of AngularJS' two way data-binding which refers to AngularJS' ability to map data directly to the GUI through the use of the *$scope* object. This allows for the user to interact with entities on the GUI and directly modify values within the storage of the extension with little code.

4.7.2 Code structure

Within the extension, files are grouped by their extension and stored in directories named for those extension names. JavaScript files are further separated into files that are involved with the GUI, those that are required to implement and instantiate the classifier and those that serve as the core files implementing the algorithm described in Section 4.7.4.

As with all previous code discussed within this document; *Net Defence* has been designed with three main layers as described in Section 4.4 and is shown in Figure 4.10. AngularJS provides most of this layering as it is built as a MVC framework.

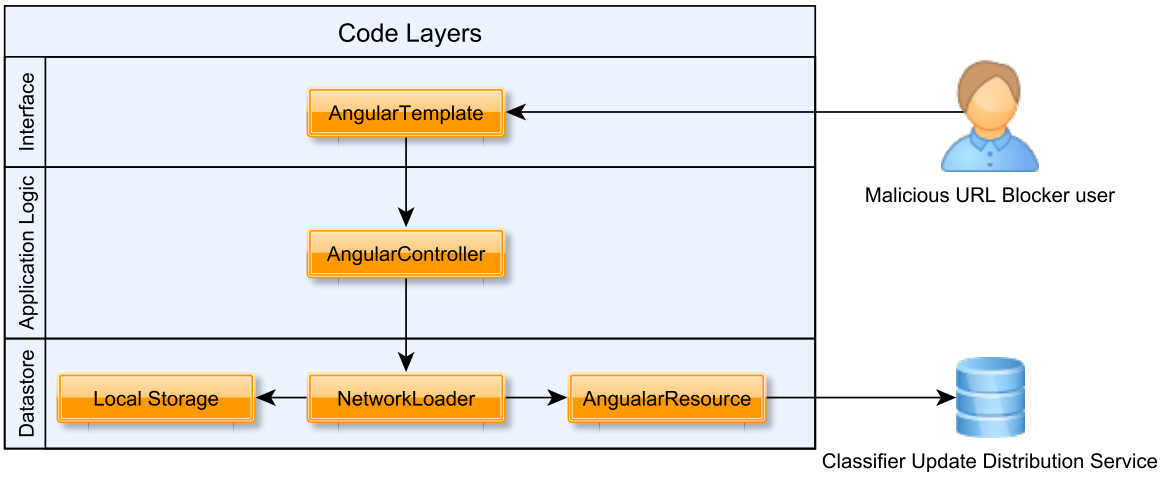


Figure 4.10: Layout of code structure within Net Defence.

Views are implemented as HTML files stored in the *HTML* directory. Data contained within the scope of the GUI is displayed using the synax {{*variable.index*}}. The *variable* may be a single scalar that contains a value used directly, in which case the syntax is {{*variable*}}. Alternatively, the *variable* may be an associative array which may be accessed in the GUI (or template) layer using the complete syntax. The template is populated by the AngularJS databinding mechanisms.

These data-binding mechanisms access the *$scope* object as mentioned in Section 4.7.1. This object holds data that is meant to be bound to the GUI. When this data is modified within the GUI, or within the logic code that supplies it, it is changed in both places. This logic code is known as the *Controller* within the MVC design pattern around which AngularJS is designed.

Each template file within this project has an associated controller which supplies the application (or business) logic of that GUI, as well as any data that should be displayed on the *view* via the *$scope* object. These AngularJS Controller instances also interact with the third and final layer of the application: the *Model*. This layer is responsible for accessing persistent data storage.

In the case of *Net Defence*, this takes the form of two seperate entities; the *LocalStorage* API and the AngularJS Resource API. *LocalStorage* is responsible for storing settings; previously fetched ANNs, exceptions and rules. The AngularJS Resource instances communicate with the REST service which supplies updates and is discussed in Section 4.6.

The interaction between the *Controller* and *Model* layers, as well as the process of classifying a URL is discussed in more detail in Section 4.7.4.

4.7.3 The Net Defence manifest

The manifest file for the *Net Defence* Chrome extensions is shown in Appendix C and serves to define the extension on a code level. The first few lines define the name, version, manifest format, description of the extension and the path to the different resolution icons. These are used by Chrome in the extensions settings page (chrome://extensions). The next section defines the permissions required by *Net Defence* and are described below.

- WebRequest — This permission allows the extension to access URLs requested by the user and other entities that relate to addresses that the extension has access to. Having this permission means that a request may be intercepted before it is made, and checked and blocked if required. WebRequest is used by *Net Defence* to block access to URLs that are classified malicious or set in the blacklist or custom rule sets.
- WebRequestBlocking — When WebRequest is required to operate in a blocking fashion, this permission is used. This means that when the extension intercepts a

request for a URL, that URL may not be fetched while the extension is currently processing the URL. When the extension has completed it's logical operations; the request may then be continued or cancelled, at the extension's discretion.

- <all_urls>— *Net Defence* is required to check all URLs to be an effective method of mitigating threats faced by the user; this permission is requested for this reason as it allows the extension to intercept all URL requests instead of a chosen few. This is where this approach to malicious URL detection starts to differ from black and white lists as it is able to make *informed* decisions about the intension of a URL without the use of a predefined list.

- storage — Requesting the storage permission gives Chrome extensions access to the *LocalStorage* API. This is implemented by use of cookies and allows extensions to persist data such as settings. The *Net Defence* uses this to store black listed and white listed URLs, as well as rules that the user creates. Additionally, the extension also uses this storage to store all information required to initialise and instantiate the ANN used when classifying URLs.

- notifications — The notifications permission allows Chrome extensions to make desktop notifications through Chrome. These notifications take the form of a small popup window in the bottom right of the screen and contain contextual information that may be dismissed. This approach is useful when the user's attention may not be on the entity which generates the notification and requires handling. This is also useful when operations are happening in the background, which the user should be made aware of.

- tabs — Finally, the tabs permission is used to allow the extension to use Chrome's tab API. This is used to allow the extension to open new tabs when the user flags a blocked request as a false positive. This is done so that the user does not have to request the URL a second time, making it a more user friendly experience.

The next section within the *manifest.json* file white lists files that are used by the extension to build the extensions GUI or functionality. This is needed as extensions, while built in the same fashion as websites, do not execute in the same environment, a web server. This in combination with Chrome's sandboxing means that relative URLs are not considered secure unless they have been specifically identified. Through the use of this section in the manifest, and the use of the *getURL* method Google (2013b), the extension is able to access an image for use within the GUI through the use of URL structured as:

chrome-extension://[PACKAGE ID]/[PATH]

Within this URL, the *PACKAGE ID* can be obtained using the *getURL* method mentioned previously and is an identifier generated by Google Chrome at the time of installation. The *PATH* is the path of the resource within the packaged extension. Using this URL, extensions may link to resources within its own HTML using web-like URLs.

The *background* section of the manifest file is important as it references all files which are required by the extension and run without the use of the GUI. This section acts to *include* all the files necessary by the last file included, esssentially instantiating the libraries used by the extension. This last file should have an execution entry point which is executed by Chrome when the browser opens. The files included by this section for the Net Defence extension will be discussed in detail in Section 4.7.2 and Section 4.7.4.

Finally, the *browser_action* and *options_page* sections both list the files required to build the GUI that the user will employ to interact with the extension. The browser action within this extension is a small popup which the user uses to identify sites as false negatives (sites which should not have been allowed to display and are, therefor, misclassifications). The options page is the part of the extension that the user has the most interaction with in the case of the *Malicious URL classifier*. This page allows the user to configure the behavior of the extension as well as giving the user the ability to manually force updates of the classifier data.

4.7.4 Algorithm

Within *Net Defence* there are several distinct pieces of code that perform discrete operations. The main algorithm, however, is responsible for making the actual classifications each time the user requests a URL and is depicted in the flow diagram shown in Figure 4.11.

When Google Chome starts up and loads *Net Defence* extension, it executes the background JavaScript file. Within this extension, this file is called *core.js* and is responsible for registering an event callback (a method to be executed when the event fires) for the *onBeforeRequest* event listener. A high-level code overview, which is a code sample from *core.js* is shown in Listing C.2.

This listener is triggered every time a URL is requested, including URLs requested by extensions and JavaScript scripts executing on web pages. The benefits of this include

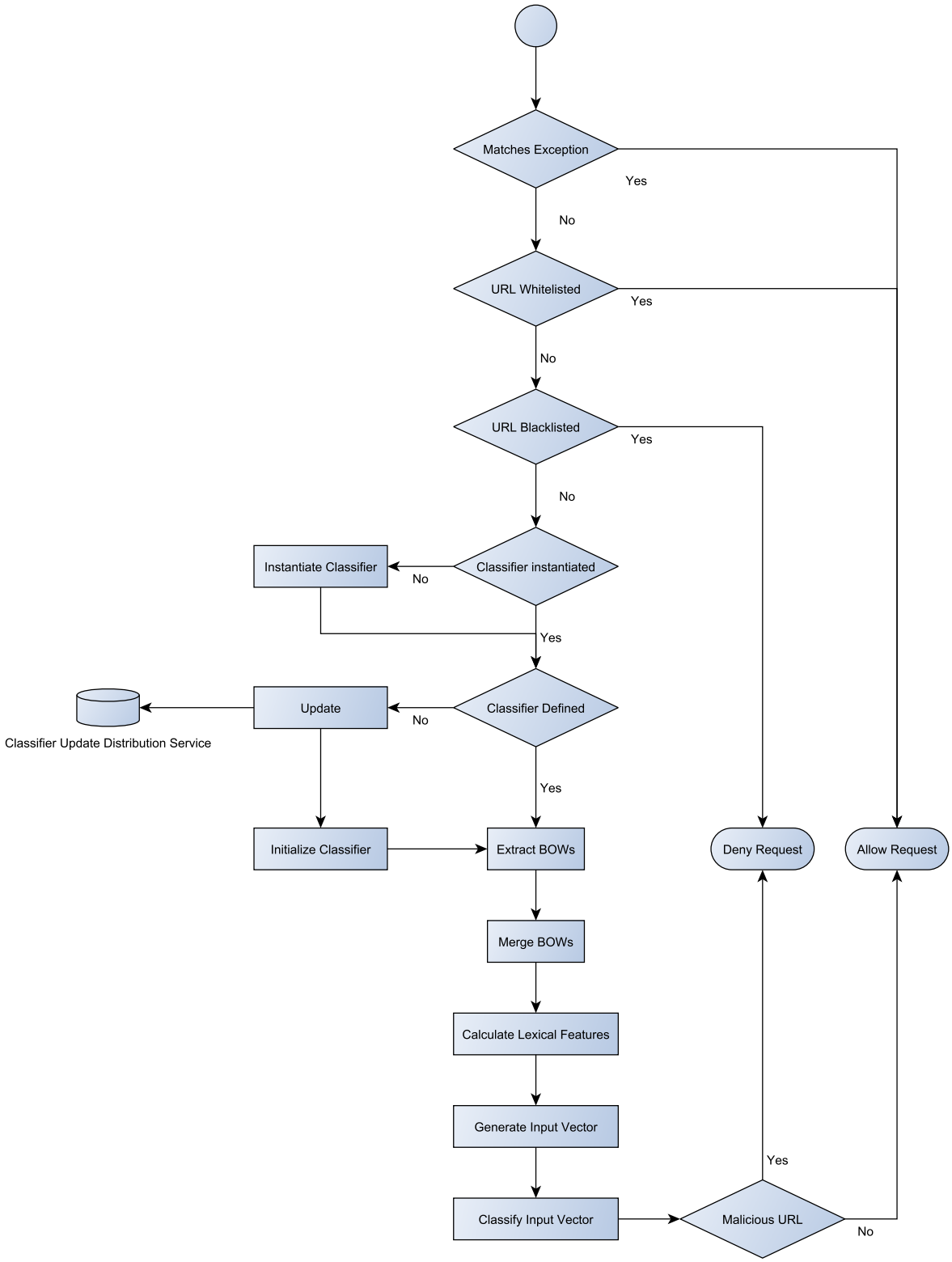


Figure 4.11: Flow diagram of the Net Defence classification algorithm.

that URLs can not be hidden from the engine by making the calls asynchronously within a JavaScript procedure. A second benefit is that this callback is executed in a blocking fashion. This means that URLs will not be fetched until after they have been classified, and can be cancelled (blocked) altogether, further protecting users from drive-by attacks.

The *checkURL* method is the callback which is executed whenever a URL is requested. An overview of the algorithm used in the *checkURL* method is depicted in Figure 4.11. Firstly, the URL is then checked against the exceptions rule list which allow for any URL that matches them to be allowed. For usability and performance reasons, the next check performed is against a user-defined whitelist. If a URL is found in this list, it is automatically allowed and the algorithm will not proceed any further. If the requested URL does not appear in this whitelist, it is checked against a user-defined blacklist. This is not a large blacklist that is defined from a resource on the internet, but a list of URLs that have been classified as benign in the past but the user has flagged them as false negatives. Similarly, if a URL has previously been classified as malicious and the user has flagged it as a false positive, it is added to the whitelist. If the URL does not match any exceptions, whitelist or blacklist entries, it is then processed by the remainder of the algorithm.

The following process encapsulates the ANN functionality around which this research is based. Firstly, the algorithm checks to see if the classifier is already instantiated from a previous request. If there is no instantiation, it will check for the data that are used to instantiate the object. If they are not found, they are requested from the CDS through the *update* functionality shown Figure 4.11. On the first execution, the classifier object is created and is an instance of the *OnlinePerceptron* class, which is stored in the library written for this extension. Once the classifier is instantiated, the data store is checked for the network definition. If this definition is not present; the update service is then invoked and the definition is downloaded and stored. This definition contains information regarding how many inputs are used, their weightings as well as the bias that the network should use. Also contained within this data is the BOW definition and the normalisation values required.

Once the classifier is instantiated and initialised, data extraction begins. The first step of this process is to extract every word from each section of the URL. A vector of boolean values is created for each section of the URL and is as long as the BOW for each section with each value set to *false*. Each word present in the requested URL is checked against this vector and, if present, the corresponding index of the vector is flipped to true. This process is repeated for each section of the URL. Once each BOW has been created, the

vectors are merged in the order of the URL structure, creating the large vector that will later be merged with another vector to form the input data for the classifier.

The final data required as input to the classifier is that of the lexical features of the URL. The URL is analyzed and a set of 20 metrics is calculated, as described in Section 3.1.1. These values are inserted into another vector and passed to an object which uses the update data to normalise the values for each field. After normalisation of this vector is complete, the vector is the merged to the end of the BOW vector which then finally constitutes the final input data passed to the classifier.

The last step within this algorithm is to pass control to the classifier which then flags the input data, and therefore the requested URL, as malicious or benign. A benign classification results in the extension allowing the request to continue uninterrupted, while a malicious classification results in the request being blocked and the user being notified, depending on the extension's settings. The options available are discussed in Section 4.7.5. Within this figure, the malicious request is simply logged to the console rather than being blocked out right. This is useful in research environments or during the testing of the classifier.

4.7.5 Functionality

While the primary purpose of this research has been to protect end-users from malicious URLs using lexical analysis; one of the foremost purposes of the Net Defence extension is to demonstrate the capabilities and possibilities of such a classification method. Figure 4.12 shows the default landing page displayed when the *options* page is requested for the extension. This page displays several metrics regarding the training of the classifier. These metrics are described briefly in Table 4.4.

Shown in Figure 4.13 is the *Settings* tab. Using this tab; the user is able to adjust the general behaviour of the extension. Different users have different levels of expertise and require different levels of interception from the classifier. A non-technical user may require the extension to simply block URLs, while expert users may not want pages blocked, rather to have notifications be displayed. Conversely, an expert user may find the behaviour of blocking pages, forcing the user to add an exception often, an irritation when doing security research. The direct result of this low usability is that the extension would eventually be disabled. Allowing the user the ability to adjust this behavior dramatically increases the usability of the application for a larger range of requirements.

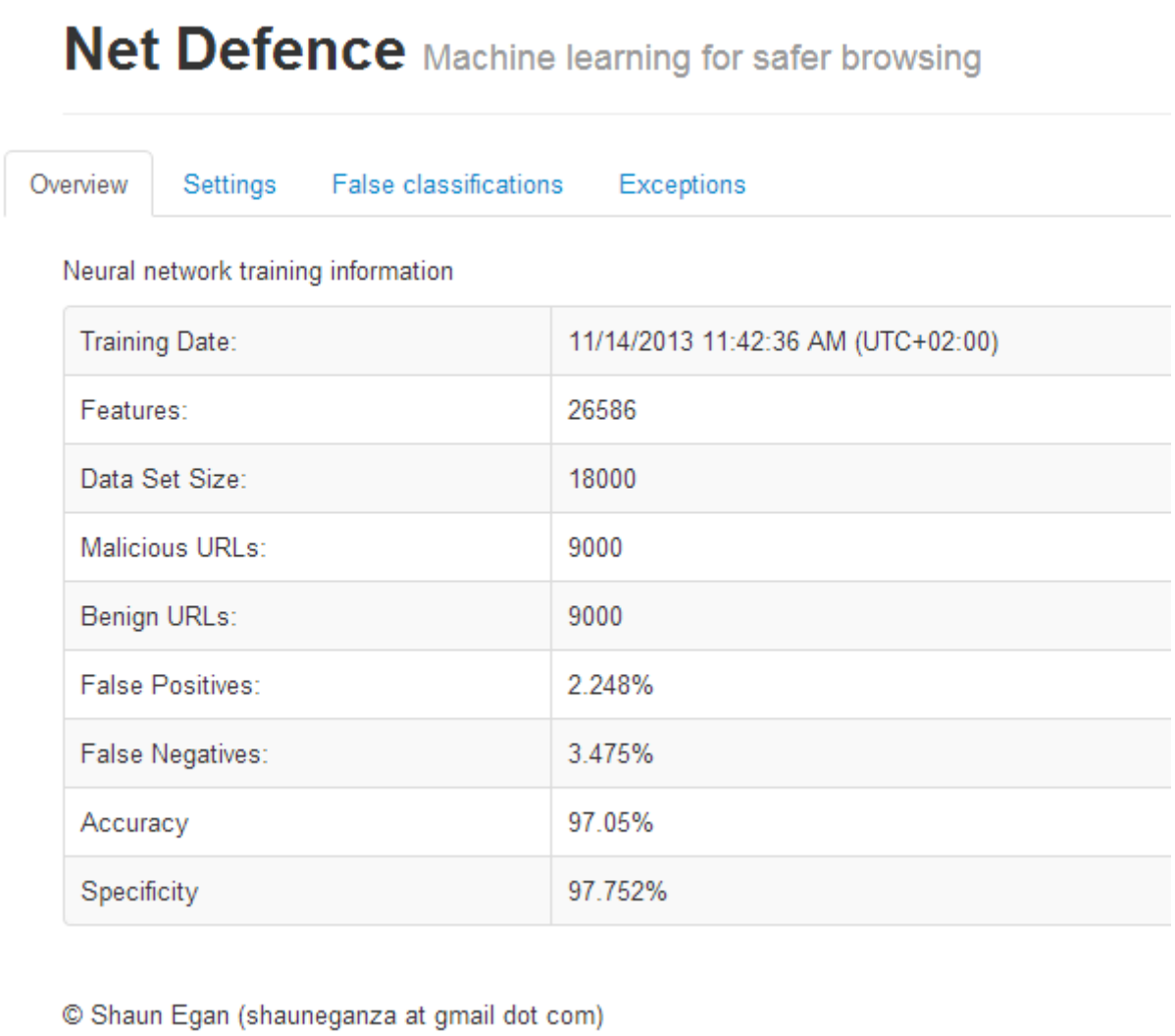


Figure 4.12: Validation details as shown by Net Defence.

Shown at the top of this page are the dates when the extension was last updated, as well as the last time that updates were checked for. This area allows for the manual checking and fetching of new updates for the classifier. Also within this section is the ability to specify a custom URL for use when checking for updates. This option is not strictly required as users should only use the official resource when checking for updates so as to increase trust in extension. This is not viable for during the testing phase and is simply present as residue from the development process.

Underneath this section is a series of check boxes which enable and disable specific behaviours. The option *Block requests for malicious URLs* allows the user to turn off malicious URL blocking which is useful when doing security research or when the user does not wish to make exceptions for a particular URL. Enabling the *Popup notifications on block* option sets the extension to show a notification whenever a URL is blocked. This

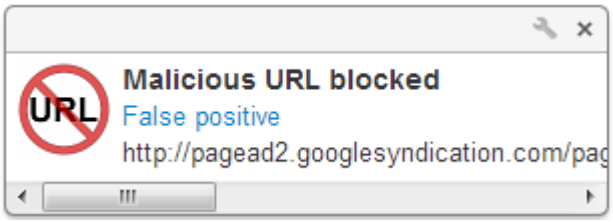
Table 4.4: Classifier metrics sent to the Chrome extension.

Training Date	The date the classifier was trained.
Features	This represents the size of the input vector used by the classifier.
Data Set Size	The complete number of URLs used to train the classifier.
Malicious URLs	The number of malicious URLs used during training.
Benign URLs	The number of benign URLs used during training.
False Positives	The false positive rate obtained by the classifier.
False Negatives	The false negative rate obtained by the classifier.
Accuracy	The accuracy of the classifier.
Specificity	The True Negative Rate (TNR).

is useful for debugging as a page request often makes several requests for other resources in the background, which the user may not be aware of. The popup is shown in Figure 4.14a and shows the URL which has been blocked and gives the option to flag to block as a false positive.

Figure 4.13: Net Defence settings tab.

When a URL is blocked, the default behavior of Chrome is to show a page indicating to the user that an extension has blocked the request. This is acceptable when the popup option is used as the popup indicates to the user which extension has blocked the request. As an alternative however, the option *Redirect on block* exists which redirects the requests to a page contained in the extension and displays the same information as the popup. An example of this page is shown in Figure 4.14b.



(a) A popup notification

(b) A redirect page notification

Figure 4.14: Net Defence alternatives for indicating a blocked URL.

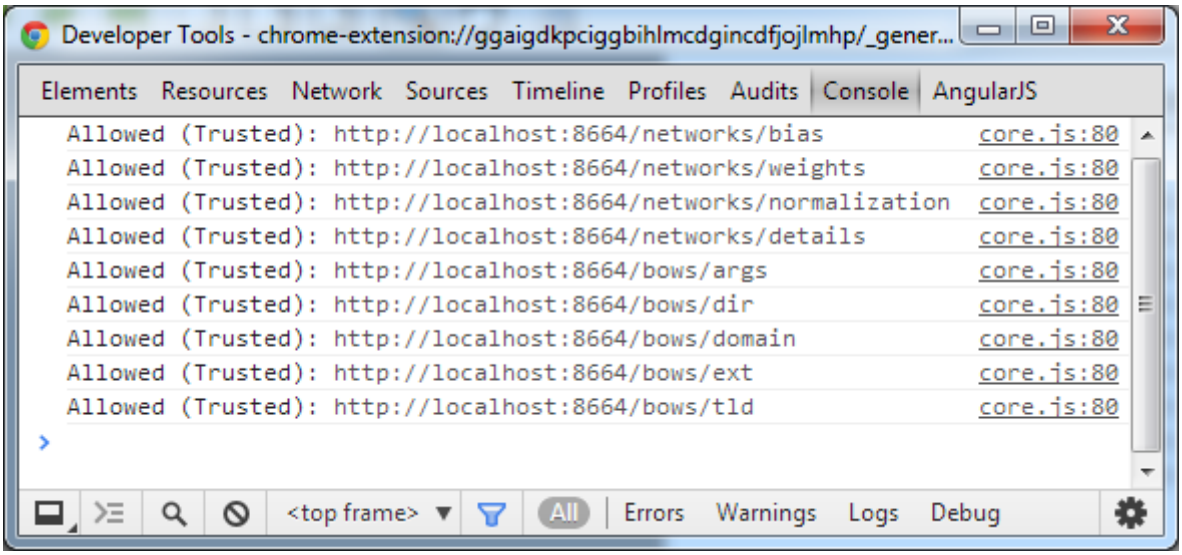
The next option available on the *Settings* tab is the *Log to console* setting. The console is a page that is generated by Google Chrome for each extension and exists as a debugging method. This page is where extensions may log messages that are intended for use by expert users or programmers when testing their extensions. The console is shown in Figure 4.15a and is displaying normal activity logged by *Net Defence*.

Shown within the console log of Figure 4.15a is a list of requests that have been allowed. Each request is either prefixed with the word "Allowed" or "Denied". This indicates whether or not the extension blocked the request. The next field logged is a field that the extension used to justify the action it took. The five options for this field are shown in Table 4.5.

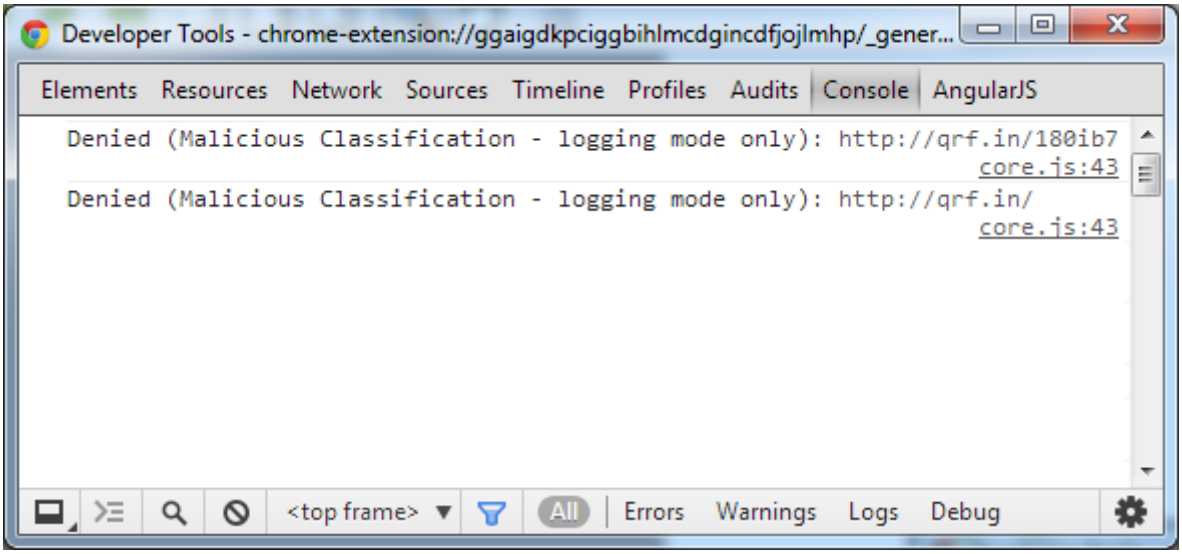
Table 4.5: Net Defence action justifications.

Trusted	A request that has been allowed as it is white listed within the extension as it matches a user-defined rule.
False Positive	This request has been allowed as the user has indicated previously is falsely classified as malicious by the classifier.
False Negative	This is used when a request is blocked because the user has flagged the URL as one which is falsely classified as benign.
Malicious Classification	When a URL is blocked with this reason being cited, it has been classified as malicious by the ANN.
Benign Classification	This reason is logged when a URL has been classified as benign by the classifier.

An example of the console when the extension blocks a request is shown in Figure 4.15b. Another status code shown within this console log is "Logging mode only". This is shown



(a) Update request



(b) Blocking a malicious URL

Figure 4.15: Net Defence console logs.

when the extension is configured to not block requests for malicious URLs.

The next option shown on this tab is *Log blocks to localStorage*. This sets the extension to insert URLs which are blocked into Chrome's local storage and is intended to be used in later revisions of the software where these can be reported to the API as training data. Another option for this data may be to submit it automatically to blacklists which are used by other services or classification mechanisms.

When a URL is requested and is classified as malicious and blocked, the user is given the ability to flag the classification as a false positive. When the user invokes this flagging, the extension will open the resource in a new tab when the option *Open resource when flagged as false positive* is enabled.

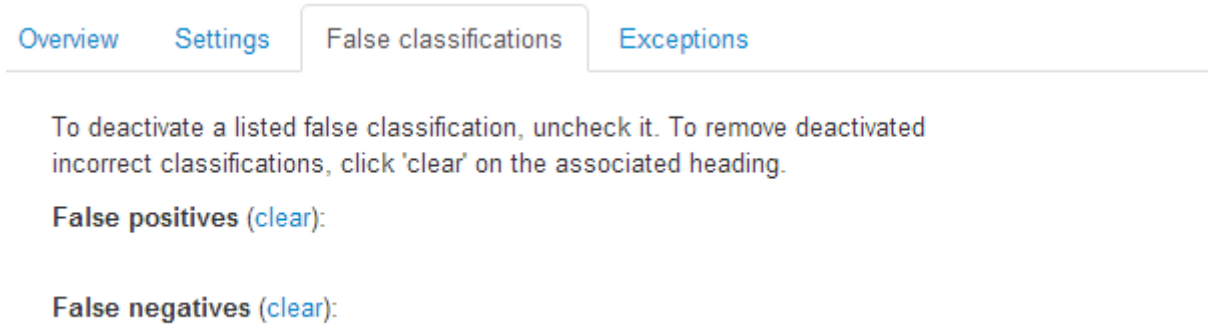


Figure 4.16: Net Defence False Classifications tab.

Shown in Figure 4.16 is the next tab available in the options page and is called "False Classifications". This tab simply lists URLs which the classifier has previously classified as benign or malicious, and the user has flagged that classification as incorrect. These lists give the user the ability to temporarily disable a classification exception, and to remove all inactive false classifications using the "clear" options for the two lists. Giving the user this ability allows users to correct themselves when accidentally flagging a classification as incorrect.

A positive classification is flagged as incorrect by clicking the "false positive" link on the popup (or redirect page) when a URL is classified as malicious; shown in Figure 4.14a and Figure 4.14b. When a classification is made and thought to be benign, no popup or redirect is executed and makes the act of flagging this classification difficult. *Net Defence* addresses this problem by using a mechanism provided by the Google Chrome extension API: the browser action. This is displayed as a small button to the right of the address bar and, when clicked, displays an option to flag a particular web page as a false

negative available by clicking on the text which says "This page is malicious". This has the additional effect of closing the tab. The browser action is shown in Figure 4.17.

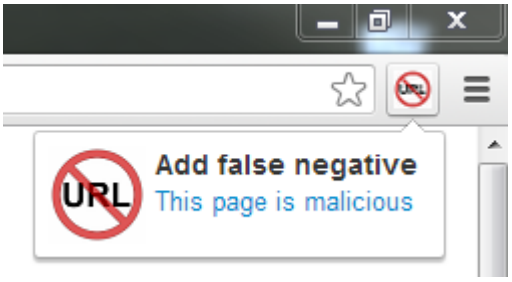


Figure 4.17: Net Defence option to add a false negative.

The final tab available within the options page of *Net Defence* is the "Exceptions" tab, shown in Figure 4.18, and is purely intended for use by expert user. This page allows a user to specify Regular Expression (REGEX) patterns that the extension should attempt to match to a URL when it is requested. When a REGEX match is made, the URL is automatically allowed as a white list entry. This page follows standard JavaScript REGEX syntax excepted for the addition of the added asterisk wild card which simplifies its use, allowing users to specify patterns without being able to write complex REGEX patterns.

These rules are useful when adding an entire class of URLs to exceptions, APIs and pre-emptively allowing sites with suspicious domain names. However, these options are dangerous if used by an inexperienced user. Within the figure are several different domains which the author found useful to allow. Many of these exceptions which apply to Google domains could be covered by the use of a single exception for **.google.**. This, however, would allow any URL with the ".google." token in any position, which is a tactic specifically employed when obfuscating URLs. It is for this reason that this page should only be used by expert users. Finally, each REGEX exception may be enabled or disabled through the use of the check box to the left of the specific exception.

4.8 Summary

Using the three entities described in this chapter (Classifier Generation Service (CGS), Classifier Distribution Service (CDS) and Net Defence); the system is able to automatically collect phishing and benign URL samples from remote sources. Because of the way that the framework has been built, this data may be fetched from different sources

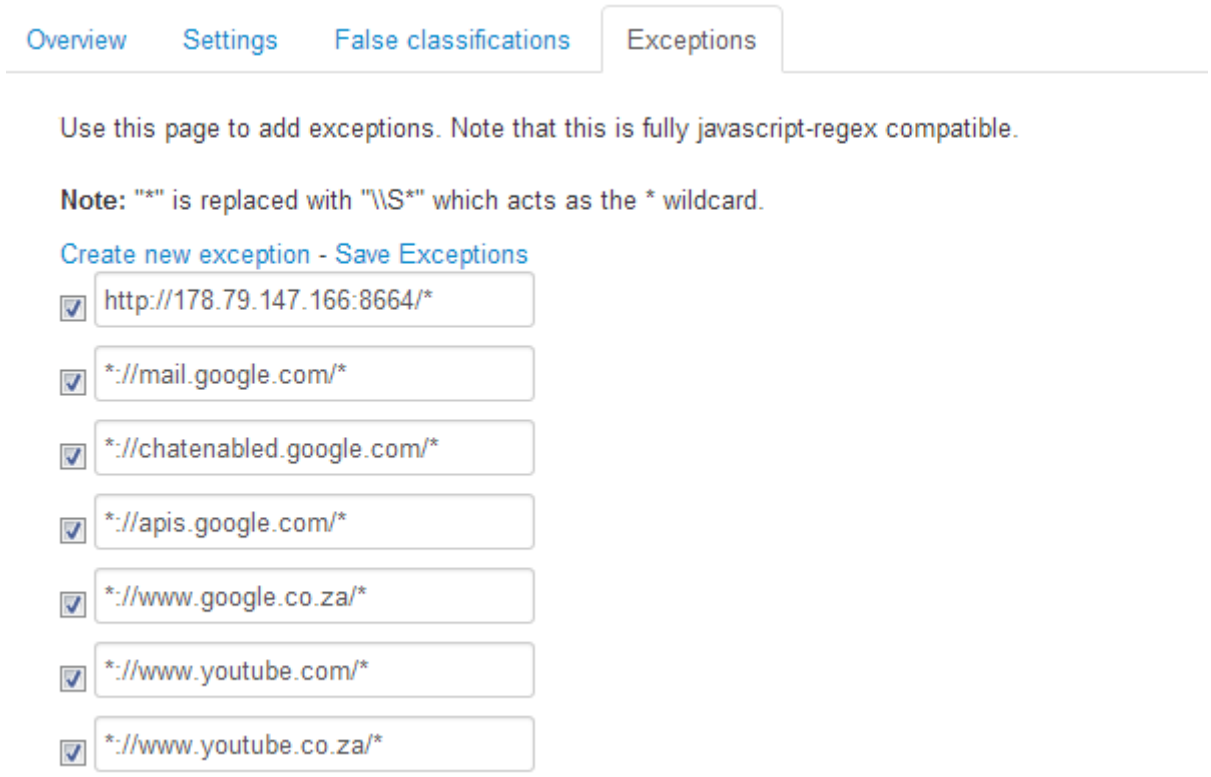


Figure 4.18: Net Defence exceptions tab.

with different intents, automatically. The data is then used to train a classifier, move it to the distribution service and deploy it to client implementations in an autonomous fashion. The ANN is then used to classify URLs in a transparent manner, while offering several protection options when a malicious classification is made. The effectiveness of these classifiers; methods of transmission and storage and best practices for generating new classifiers are tested in Chapter 5.

CHAPTER

FIVE

TESTING AND RESULTS

This chapter describes tests performed on the ANNs generated by the services described in Chapter 4 towards the goals outlined in Section 1.2. These in turn made use of the techniques described in Chapter 3. Data set composition, size and purpose are discussed in Section 5.1. Within Section 5.2 the methods used to analyse the performance and other characteristics of the classifiers are shown. Technical specifications of the computer used to run the tests described in this chapter are shown in Section 5.3.

Covered in Section 5.4 is a test designed to find an optimal size for training data sets. The speed at which classifications can be made is tested in Section 5.5. In Section 5.6, the size of the data that will need to be distributed is calculated in reference to the size of the training samples used, optimal classifiers and the effectiveness of various compression algorithms. Finally, the results of these tests and recommendations are discussed in Section 5.7.

5.1 Data sets

For reasons discussed in Sections 5.4.3 and 5.4.4; a training data set size of 18 000 samples is generated for each test. In Section 5.4; a variable training data set size is used and is discussed in Section 5.4.1. The validation set is always generated at a size of 2000 unique

Table 5.1: Summary of data sets.

Section	Data set #	Training	Validation	Testing	Balance
5.4	1 - 22	1 000 - 22 000	2 000	2 000	1:1
5.5	18	18 000	2 000	2 000	1:1
5.6	18	18 000	2 000	2 000	1:1

samples, none of which appear in the training set. Finally, for ANN analysis; a third data set called the *Testing set* is used. This set, like the validation set, is also always generated at a sample size of 2000, none of which appear in either the training or validation sets. All data sets generated for every test described in this chapter are balanced with a 1 to 1 ratio of positive and negative samples. These samples are blended together as each successive sample is the opposite classification, randomly selected from its source data set. A summary of the data sets that were generated is shown in Table 5.1 which indicates how many samples were used out of the 9 213 239[1] samples available.

5.2 Classifier performance analysis

Each classifier's performance is judged by a series of statistical metrics, each with different insights, strengths and weaknesses in terms of performance analysis. Several different metrics are used to obtain measures from different viewpoints, allowing for a thorough examination of the performance of each classifier.

Equation 5.1 shows how accuracy is calculated in terms of total number of correctly classified samples. This test does not take into account differences between correct positive and negative classifications, just how often the classification is correct in terms of the testing data set. Shown in Equation 5.2 is the calculation used to determine the classifier's TPR, or the rate at which a sample is correctly classified as positive. Also known as *sensitivity*; TPR is a measure of how effective a classifier is at identifying positive samples correctly (Boyko, 1994). Equation 5.3 shows how the False Positive Rate (FPR) is calculated. The FPR is known to measure fall-out, or how often the classifier incorrectly classifies a sample as positive. Specificity, or TNR, is also calculated. The method in which this is done is shown in Equation 5.4. This measure determines the rate at which the classifier is able to correctly identify negative samples. False Discovery Rate (FDR) is a measure of how many positive classifications are incorrect out of the total number of positive classifications made by the classifier. This measure is calculated in Equation 5.5.

[1]This consists of 14 707 phishing URLs and 9 198 532 benign URLs

Algorithm 4: Classifier performance measures.

$$ACC = \frac{TP + TN}{P + N} \tag{5.1}$$

$$TPR = \frac{TP}{P} = \frac{TP}{TP + FN} \tag{5.2}$$

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN} \tag{5.3}$$

$$TNR = \frac{TN}{N} = 1 - FPR \tag{5.4}$$

$$FDR = \frac{FP}{FP + TP} \tag{5.5}$$

$$PPV = \frac{TP}{TP + FP} \tag{5.6}$$

$$NPV = \frac{TN}{TN + FN} \tag{5.7}$$

$$F_1 = \frac{2TP}{2TP + FP + FN} \tag{5.8}$$

$$MCC = \frac{TP * TN - FP * FN}{\sqrt[2]{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{5.9}$$

Where:

- P is the number of positive samples.
- N is the number of negative samples.
- TP is the number of correctly classified positive samples.
- TN is the number of correctly classified negative samples.
- FP is the number of incorrectly classified negative samples.
- FN is the number of incorrectly classified positive samples.

Positive Prediction Value (PPV) and Negative Prediction Value (NPV) are calculated and shown in Equation 5.6 and in Equation 5.7 respectively. Given a positive prediction, PPV indicates what the likelihood is of a correct prediction. Similarly, given a negative prediction, NPV indicates what the likelihood of the prediction being correctly classified is. The F_1 score, or F-score, of a classifier is another metric which is meant to indicate a classifier's accuracy and is shown in Equation 5.8. The F_1 score uses the PPV (preci-

sion) and TPR to determine a harmonic mean value between them. Because the F_1 score does not take into account the TNR, the Matthews Correlation Coefficient (MCC) is also calculated for each classifier as shown in Equation 5.9. MCC gives a score between -1 and 1, where 1 indicates that the classifier is capable of perfectly classifying the data, -1 indicates that the classifier makes incorrect classifications for every sample and a score of 0 indicates that the classification accuracy is no better than random prediction (Baldi, Brunak, Chauvin, Andersen, and Nielsen, 2000).

5.3 Test bed

All tests performed on the classifiers described in this chapter are performed on the same desktop computer, unless otherwise stated. The test system has an Intel i7 2600k processor running at 3.4 GHz and 8 gigabytes of DDR 1600 MHz RAM. The Operating System (OS) installed during the testing period was Microsoft Windows 7 64 bit, running all available updates up to July 2013. Microsoft Visual studio 2010 is used to execute the CGS application with no patches. Python version 2.7.3 is used to execute CDS and Google Chrome version 31.0.1650.57 (m) is used to run the Net Defence extension.

The ANNs are all generated by the framework described in Section 4.5 which is running in a test mode. This mode allows for data to be loaded from files rather than online services and saves the resulting ANN and other required data in binary data files. This data can then be loaded and the ANN instantiated using a bespoke test application which uses the testing data set to generate the required statistics.

5.4 Optimal data set size

This test is designed to determine what effect the size of the sample training data has on the accuracy of the resulting classifier. Results from this test will indicate if the framework is functioning as designed and if the accuracy of 93.1% reported in Le *et al.* (2011) when using Phishtank and Open Directory data sources is achievable. The test will also indicate if that accuracy can be improved upon by adjusting the training data set size. The results of this test are discussed in Section 5.4.5.

5.4.1 Test data

Since this test is to determine what data set sizes result in the best performing ANNs, multiple data sets with various sizes are required. The data set that generates the best performing ANNs are referred to as the *optimal* size sets. To achieve a high enough resolution, data sets were generated with size incrementing in 1 000 sample intervals. The smallest set being 1 000 samples, with the largest being 22 000 samples. The sets generated for this test are all perfectly balanced at the 50% point. For example, the 15 000 sample set has 7 500 positive samples (malicious) and 7 500 negative samples (benign).

Sets were generated using the reduction data balancing method since both malicious and benign data sets exceeded this number by enough to accommodate 2 000 sample validation and 2 000 sample testing data sets. Just as the training data sets are balanced sets, both validation and testing data sets are balanced. Each comprises 1 000 positive and 1 000 negative samples. As a result; 22 data sets were generated, each with a unique training, validation and testing data set for each test run, as shown in Table 5.1.

For each unique data set group generated in Section 5.4, 10 classifiers were trained. This allows the test to compensate for anomalies in the training data in several aspects. This resulted in 220 unique classifiers, each with different results due to the random initialisation states which are part of the training process.

Once this training process was completed, an analysis program was run which was built to do the same job as validation in terms of calculating metrics based on the performance of the ANN, but to use the testing data instead of the validation data. Finally, this program generated a results file which contained all of the relevant metrics regarding the testing of the classifiers.

The results of these tests are aggregated and graphed using the *ggplot2*[2] package for the *R-project for statistical computing*[3] application. This results in graphs that are easy to generate and display additional information such as the level of confidence of a fit in a gray band around a fit line. The fit line for each test is generated using the Linear Model (LM) method to show trends within data points.

[2] http://ggplot2.org
[3] http://www.r-project.org

5.4.2 Timings

The first tests performed as part of the objective of determining the optimal training data set size is that of timings. By knowing how increasing or decreasing data set size impacts the time taken to extract, normalise and train an ANN; questions regarding how often new classifiers can be generated and supplied as updates may be answered.

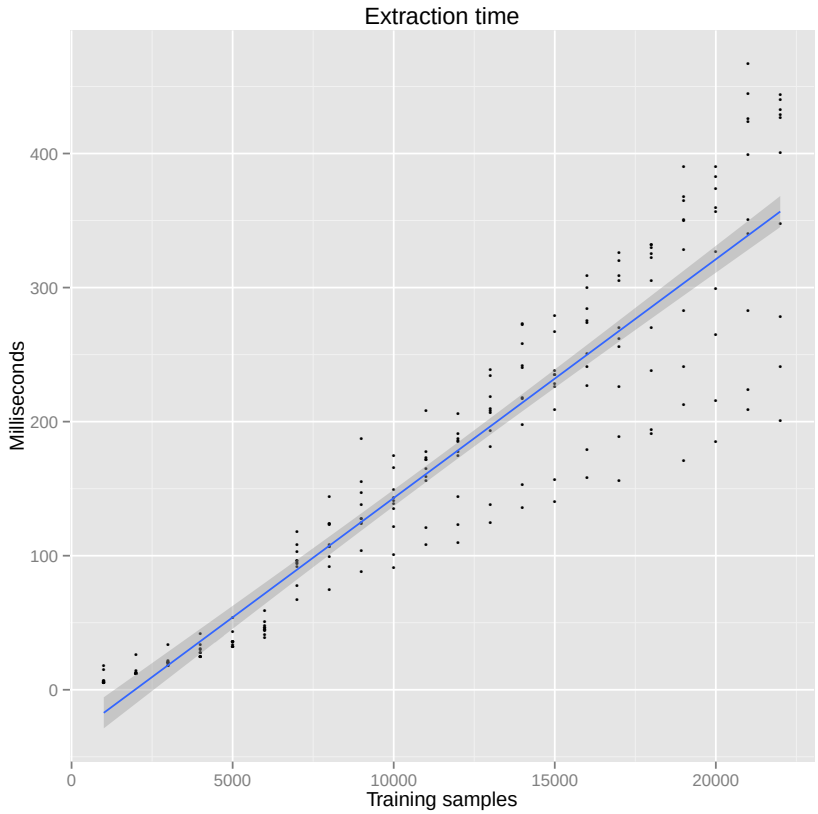
Since the size of the BOW generated for a data set increases in size as the number of training samples increases, the likelihood of new unique words being included is increased. As this number of possible unique words in each section of a URL increases and the number of words that need to be checked for expands, the size of the vector expands and the time required to extract that vector as input to the classifier increases.

As shown in Figure 5.1a, extraction time increases linearly with the number of training samples present. The various statistical metrics regarding extraction time are summarized in Table 5.2 and show that arithmetic mean time increases from 7.9 milliseconds to 364.2 milliseconds. These values are the time required to extract 2000 samples. This means that the average time taken to extract a single sample takes from 0.00395 milliseconds to 0.1821 milliseconds depending on the training data set total size. Using the fit line shown in Figure 5.1a, approximate extraction time can be estimated using Equation 5.10.

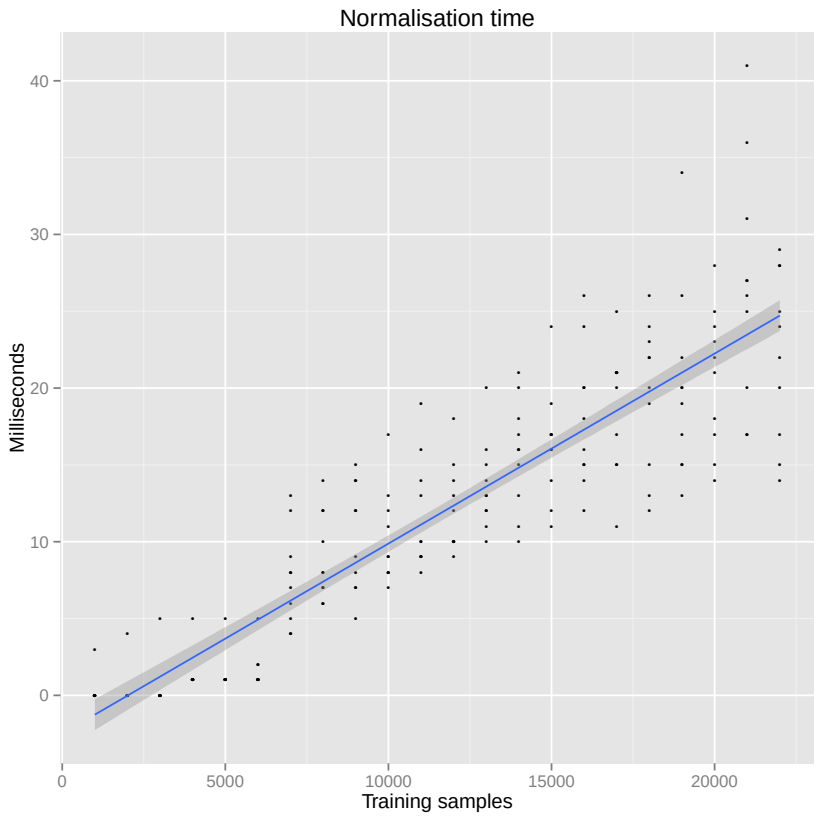
Table 5.2: Extraction time in milliseconds.

Data set #	Samples	Fastest	Slowest	Average	Std. Dev.
1	1 000	5	18	7.9	4.654
4	4 000	25	42	29.3	5.417
7	7 000	67	118	94.7	14.322
10	10 000	91	175	136.2	26.076
13	13 000	125	239	195.4	37.886
16	16 000	158	309	249.8	49.948
19	19 000	171	390	306.0	74.873
22	22 000	201	444	364.2	91.803

Like the extraction time, normalisation time also increases linearly with the number of training samples used to generate the classifier. Variation in normalisation time stays fairly stable, as shown in Figure 5.1b, and is due to the number of continuous variables within the vector being constant. While this number does not change, the length of the vector being normalised does increase with the number of training samples used due to the size of the BOW. This increased length makes seek time within the vector take longer as within this implementation, it is stored at the end of the vector.

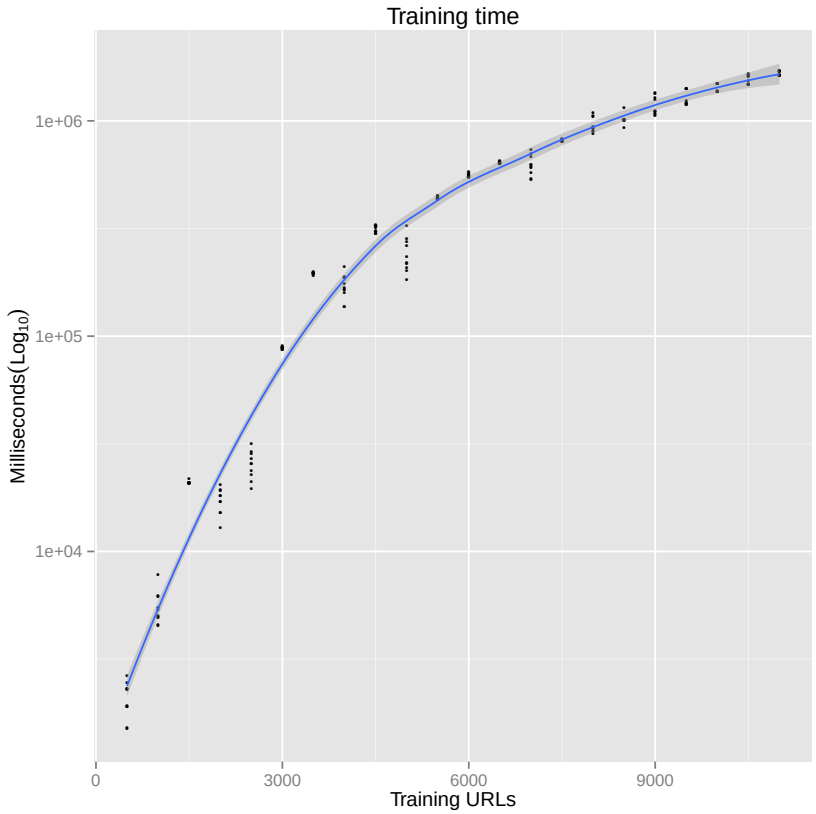


(a) Extraction time.



(b) Normalisation time.

Figure 5.1: Timing based on the number of training samples.



(c) Training time.

Figure 5.1: Timing based on the number of training samples (continued).

The longest time required to normalise a testing data set is encountered with the training data set of size 21 000 samples. The time required to normalise on this worst case scenario with 2000 samples was 41 milliseconds, or 0.0205 milliseconds per sample. Shown in Equation 5.11 is a method to estimate the time required to normalise a data set. Algorithm 5 may be used to calculate estimates of both extraction and normalisation time for a given data set size. A summary of the normalisation time data is shown in Table 5.3.

Training time increases substantially with the increase in training set size, as shown in Figure 5.1c with timings summarized in Table 5.4. This happens for a number of reasons. Firstly; as the number of training URLs increases, so does the size of the results BOW which needs to be generated as part of the overhead. Secondly; with a larger BOW, more words need to be checked for with each sample being extracted for training and validation. Normalisation time for the training data set and validation set is also increased as there are more input vectors to check and subsequently normalise. Normalisation is further slowed due to the larger size of each individual input vector due to the time required to seek to the required continuous values. Finally; the volume of data required for each

Table 5.3: Normalisation time in milliseconds.

Data set #	Samples	Fastest	Slowest	Average	Std. Dev
1	1 000	0	3	0.3	1.949
4	4 000	1	5	1.4	1.265
7	7 000	4	13	7.6	3.098
10	10 000	7	17	10.2	3.084
13	13 000	10	20	13.6	2.875
16	16 000	12	26	18.0	4.497
19	19 000	13	34	20.1	6.190
22	22 000	14	29	22.2	5.535

training epoch is higher, resulting in longer individual training iterations through the data set.

It is clear from the three tests described that the number of samples used to train an ANN has a significant impact on the time required to complete such training. This is summarized in Table 5.4 and is shown in milliseconds. This table shows that the longest time required to train a classifier took 28.5 minutes while the shortest time to train one was 1.5 seconds. This shows that for an increase of 22 times the training data used to generate the ANN, the training time slowed down by 803.06 times on average.

Table 5.4: Training time in milliseconds.

Data set #	Samples	Fastest	Slowest	Average	Std. Dev.
1	1 000	1 507	2 643	2 073.8	386.488
4	4 000	12 876	20 396	17 217.3	2299.809
7	7 000	190 217	198 625	195 573.8	2 489.005
10	10 000	182 910	325 305	241 064.4	44 538.710
13	13 000	630 926	651 593	640 250.7	5 887.714
16	16 000	875 317	1 093 692	959 349.6	78 986.540
19	19 000	1 191 612	1 419 212	1 289 620.0	108 113.500
22	22 000	1 619 702	1 712 934	1 665 385.0	39 553.880

While extraction and normalisation times stay relatively low, training time increases at a near exponential rate with the use of larger training data sets. With domain names, folder structures, file names and arguments not limited to natural language words, there is no limit to the length of the BOW. This further justifies the need to find an optimal training data set size so that training times do not become unfeasibly long in the search for accuracy. Shown in Sections 5.4.3 and 5.4.4, one such optimum value exists.

As is shown in Section 5.4.3, 18 000 samples is the best number of samples to use during training. Using a data set of this size would require 283.9 milliseconds to extract into

Algorithm 5: Time estimates for training data extraction and normalisation.

$$y_1 = 0.018x - 33.333 \tag{5.10}$$

$$y_2 = 0.0012x - 2 \tag{5.11}$$

Where:

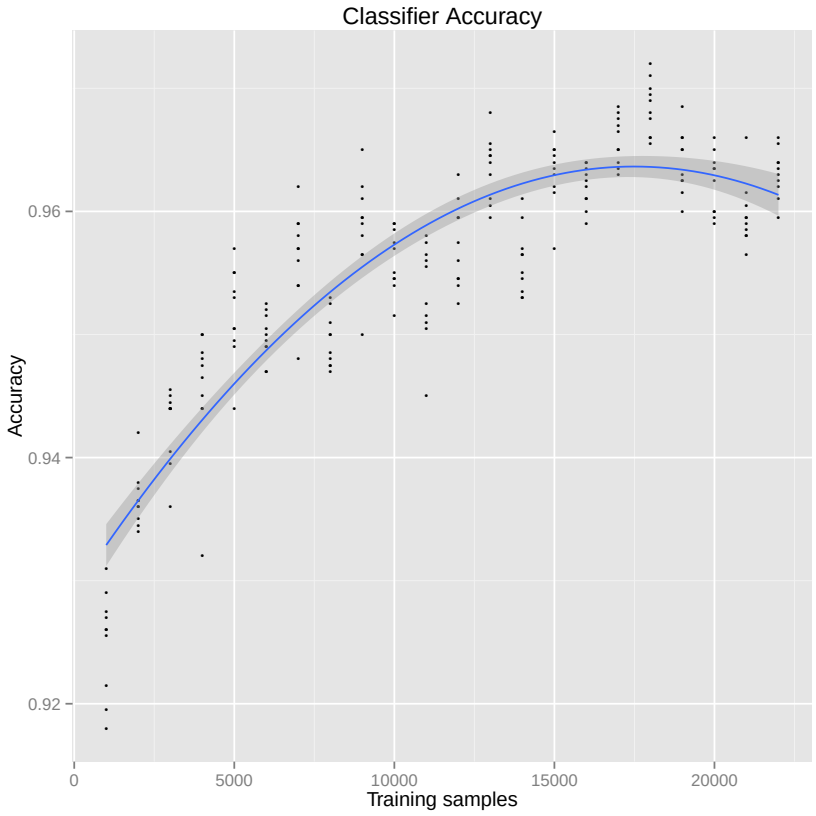
- y_1 is extraction time in milliseconds.
- y_2 is the normalisation time in milliseconds.
- x is the number of samples used for training.

input vectors, 19.6 milliseconds to normalise those vectors and a further 19.57 minutes to train the classifier on average. With this timescale, a new classifier could be trained hourly or daily.

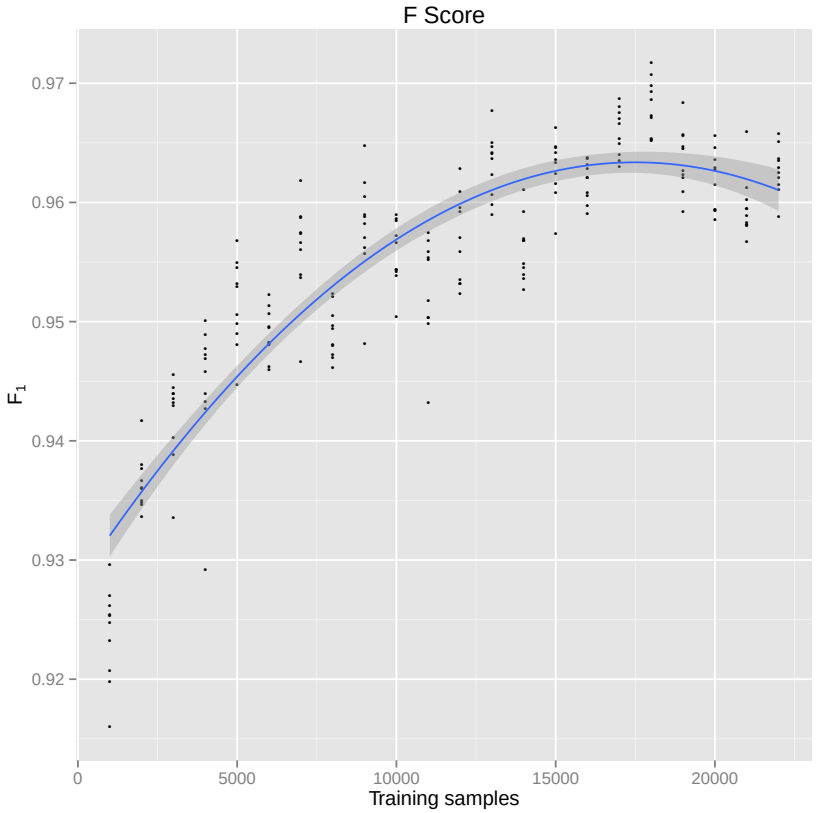
5.4.3 Accuracy

A second series of tests performed are to determine the impact that training data set size has on the performance of a classifier. As mentioned in Section 5.4, this test will show if an accuracy of 93.1% is possible, as reported in Le *et al.* (2011) using a similar method. This test will also determine if this approach is viable on a end-user level and will also help to answer the question posed at the end of Section 5.4.2. This question asked if there is an optimal number of training samples that, exceeding this number, decreases the overall performance of the classifier? This would nullify the problem of ever-increasing BOW size and the directly related increase in training time.

To this end, all of the classifiers generated were compared using three different measures of performance: accuracy in terms of cumulative error rate, f_1 score and MCC as calculated per Equations 5.1, 5.8 and 5.9. The F_1 score is a statistical measure of accuracy; reaching best accuracy at a score of 1, and a worst possible score of 0 (Beitzel, 2006). It measures accuracy in terms of precision and recall, but does not account for TNR. For this reason, the Matthews Correlation Coefficient (MCC) accuracy measure is also used, at the suggestion of Powers (2011) and can be used as a measure of a binary classifier's accuracy (Baldi *et al.*, 2000). The results of these three measures are shown in Figure 5.2a, Figure 5.2b and Figure 5.2c respectively. It is apparent from all three of these metrics, the best performing classifiers are all generated in the band of training data sets with between 13

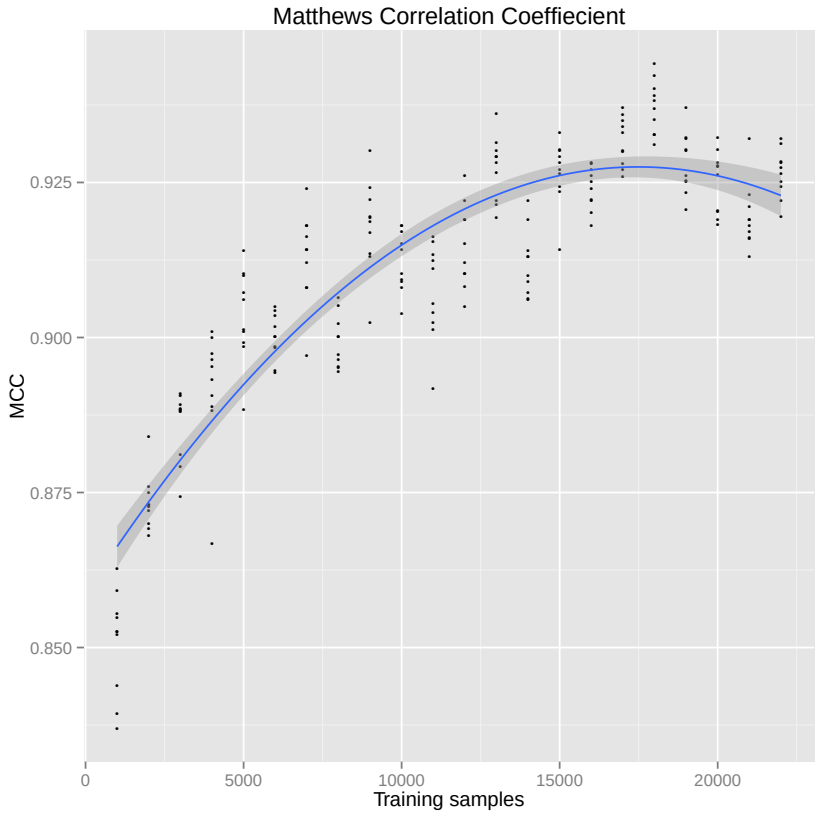


(a) Accuracy. Peaking at dataset #18.



(b) F_1 score. Peaking at dataset #18.

Figure 5.2: Accuracy metrics based on the number of training samples.



(c) Matthews Correlation Coefficient (MCC). Peaking at dataset #18.

Figure 5.2: Accuracy metrics based on the number of training samples (continued).

000 and 21 000 samples per set, with the trend moving downwards as data set sizes tends to increase after 18 000 samples.

The optimal network trained, based on all of the accuracy metrics used, occurs when using data set #18. This ANN has a cumulative error rate of 2.8% (97.2% accuracy). It is shown in Section 5.4.4 that this classifier also has the highest True Positive Rate (TPR), True Negative Rate (TNR) and prediction values of all of the classifiers generated. The least accurate classifier (91.8% accurate) is trained from 1000 samples. This shows that even a low number of training samples is capable of producing a reasonable classifier of this type, supported in Section 5.4.4, with acceptable TPR, TNR and prediction values.

Shown in the metrics above is that; given the data set is balanced in terms of classification types, that classifiers trained with datasets between 13 000 and 21 000 samples tend to perform the best, with data set #18 (18 000 samples) being optimal. It is for this reason, that all subsequent tests performed in this chapter are using the classifier trained with this sample set size that performed best of all the ANNs generated.

Table 5.5: Accuracy metrics for data set #18.

Metric	Best	Worst	Average	Std. Dev.
Accuracy	0.972	0.966	0.96845	0.0022
F Score	0.972	0.965	0.96804	0.0024
MCC	0.944	0.931	0.93725	0.0043

Table 5.6: Accuracy metrics for data set #1.

Metric	Best	Worst	Average	Std. Dev.
Accuracy	0.931	0.918	0.9251	0.0041
F Score	0.930	0.920	0.9240	0.0040
MCC	0.863	0.837	0.8510	0.0084

Shown in Table 5.5 are the performance measures training data set #18. This set contains 9 000 malicious and 9 000 benign samples and yields the best validated and separately tested classifiers of all the ANNs trained. Table 5.6 is given for comparison and shows the performance metrics for data set #1.

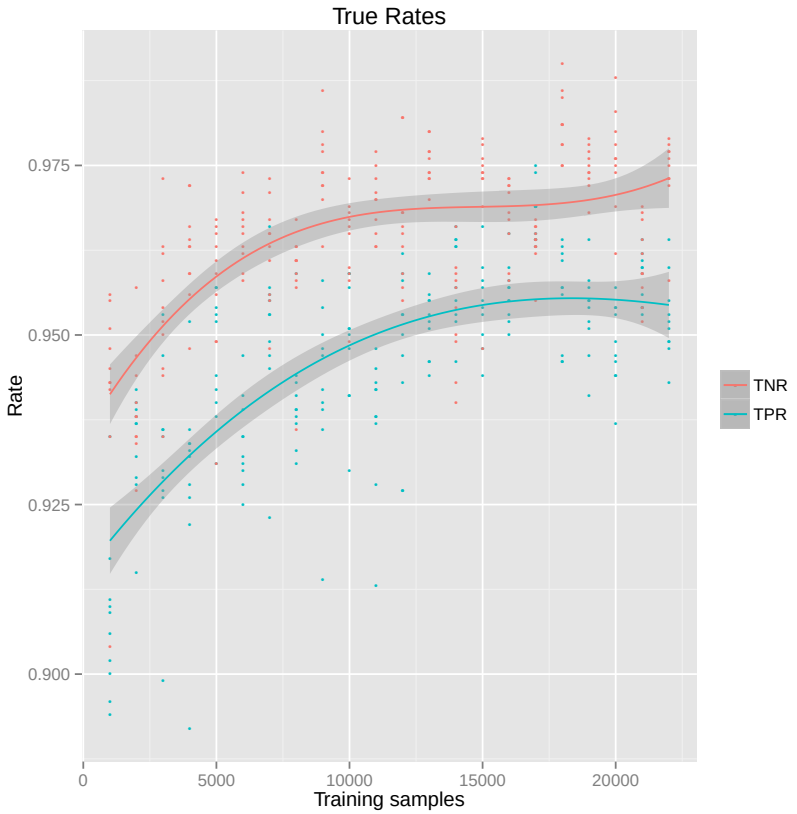
These three metrics give a good indication as to what the optimum training data set size is, but do not indicate the value of each classifier in terms of TPR and TNR. These rates are important as different applications have different requirements.

5.4.4 Prediction values

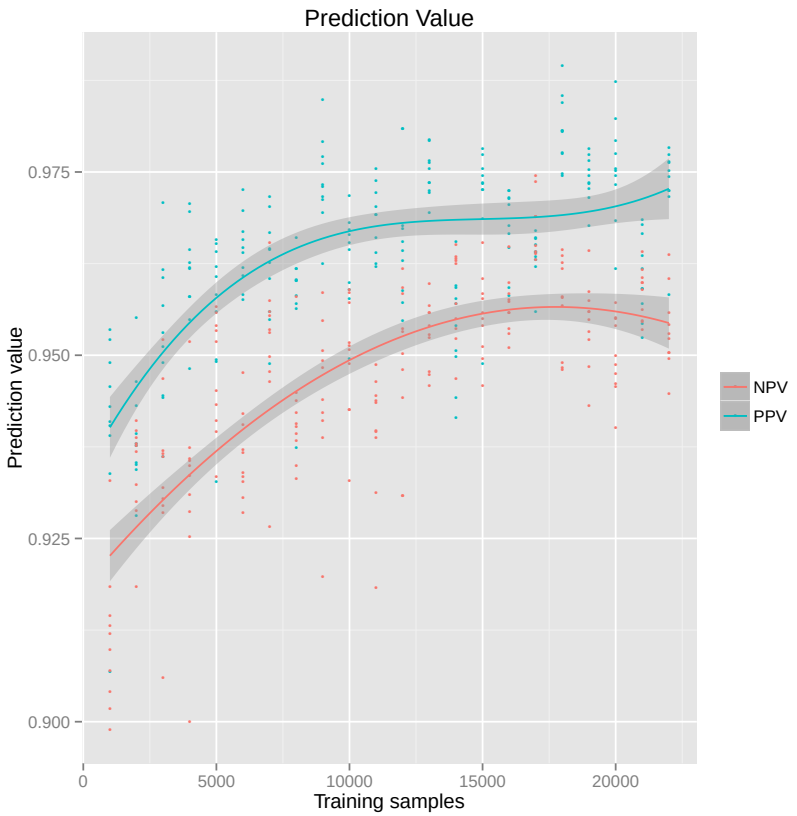
The final tests performed are regarding the prediction value and usability of the metrics discussed in Section 5.4.3. These values bring more meaning to these metrics as they show how such accuracy is achieved.

As shown in Figure 5.3a, TNR improves with the more samples used during training. The TPR hits its average maximum at data set #18 and then starts to decrease. The TNR is stable at the same sample point and gets higher with more training samples used.

Prediction value; which indicates the chance of a prediction made by the classifier being correct, is shown in Figure 5.3b. This plot is very important in relation to the True Positive (TP) and True Negative (TN) rates plotted in Figure 5.3a in that it shows that, while the TNR increases with the more samples used, the NPV starts to decrease after 18 000 samples. The opposite is true for the TPR and the PPV. While the TPR decreases after the same point, its prediction value increases.

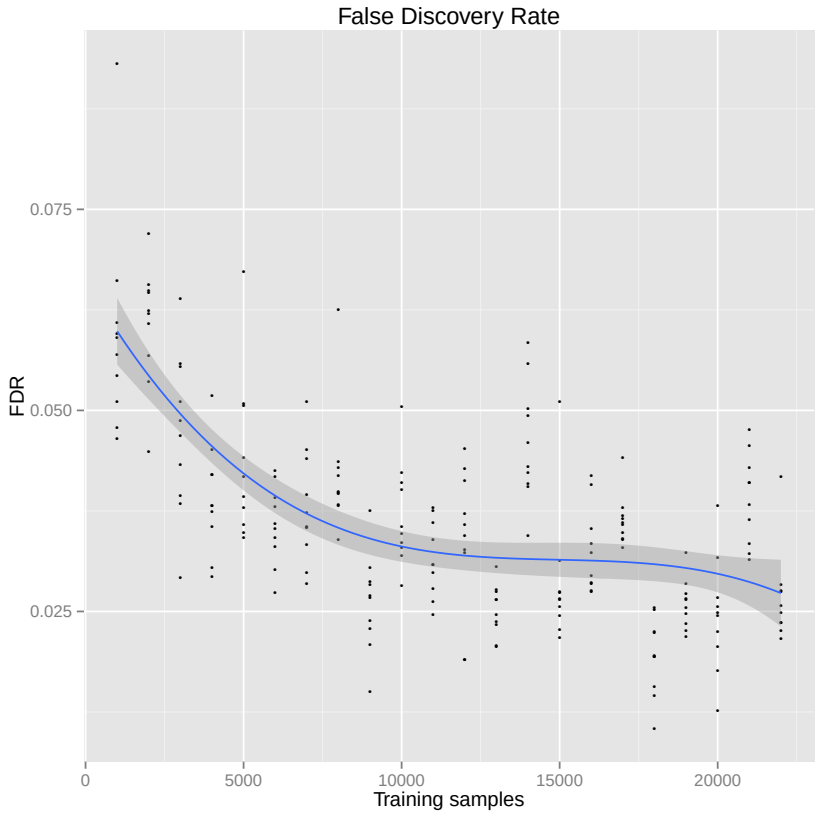


(a) True Positive Rate (TPR) and True Negative Rate (TNR).



(b) Positive Prediction Value (PPV) and Negative Prediction Value (NPV).

Figure 5.3: Prediction values.



(c) False Discovery Rate (FDR).

Figure 5.3: Prediction values (continued).

False discovery rate, as shown in Equation 5.5 and discussed in Section 5.2, indicates how often a classifier makes an incorrect positive classification of all positive classifications made. The FDR results for each network are plotted in Figure 5.3c. Like all the other tests described in this section, the best average FDRs are achieved when using 13 000 to 22 000 samples within a balanced data set as a training data set.

5.4.5 Discussion

This test was designed to answer several questions. Is the accuracy mentioned in Le *et al.* (2011) reachable; how long does it take to train an effective classifier; how many samples are needed to train such a classifier and what accuracy is achievable? Additionally, what effect would users adding new samples make on the training of these classifiers?

As is shown in Section 5.4.3; the accuracy obtained by Le *et al.* (2011) (93.1% for an online perceptron) is possible. Section 5.4.3 shows that it is not only possible to achieve

this; but to significantly improve on it by adjusting the training data set size, improving it by a further 4.1% for a total accuracy of 97.2% (or a 2.8% cumulative error rate). This results in a better classifier built as an OP which is much simpler to implement than other techniques.

Within Section 5.4.2 it is shown that training time increases exponentially when more training samples are added. This is due to the expanding BOW size affecting performance in several facets. Additionally, as users submit new unseen data for inclusion in the training and vetting process, the resulting BOW increases in size with the increase in samples. Since there is no upper limit to the number of words that could be included within the BOW, this could be a potential problem.

However, shown in Section 5.4.3, there is an optimal range of training set sample sizes that should be used when training these classifiers. That range is from 13 000 to 21 000 samples, impacting performance of the classifier for the worse if this range is exceeded. This is a fact that is further proven in Section 5.4.4 by showing that the value of predictions made by the classifier start to decrease when exceeding this range.

Finally, the best ANNs trained as part of this research were shown to be generated using a training sample set size of 18 000 samples. Since this is considered an optimal point, it is recommended that new data retrieved from data sources (including vetted user submissions) be added to the end of the data used for training and that the most recent 18 000 samples be used. This set should comprise 50% positive and 50% negative samples with recommended separate validation and testing sets of 2000 samples each, also balanced at the 50% point.

5.5 Classification speed

The time required to extract, normalise and classify a URL is known as classification speed. This time is important as it indicates which applications this research is applicable to. The test will cover execution times in both a C# implementation as well as a Java script implementation built for Google's Chrome Browser (see, Section 4.7). The results shown in this section are discussed in Section 5.5.3. Also tested in this section is what effect processor speed has on all three stages of classification.

5.5.1 Test data

The ANN that was used for this test was the best classifier generated in Section 5.4 and the testing data generated for that classifier consisting of 2000 samples. For the Java script implementation tests, the Chrome extension described in Section 4.7 was used while following samples found in the same testing data sample set. The third test, in which classification time is tested against processor speed, uses the ten classifiers and their associated training data that were trained using data set sizes of 18 000 samples in Section 5.4. These were chosen as they were the best performing classifiers, and all ten were used to determine average times for different ANNs with the same training sample set size.

5.5.2 Tests

The first test executed used a C# implementation using the library developed and described in Section 4.5. The test was executed using the ten classifiers trained with a training data set size of 18 000 samples. Each of the ten classifiers was used to classify the testing data set and timed. Shown in Table 5.7 are the results of this test and it is shown that the average time taken to classify a single sample for a classifier of this size is 0.07908263 milliseconds with a standard deviation of 0.001730493.

Table 5.7: Classification time (MS) of 2 000 samples using the C# implementation.

Run #	Extraction	Normalisation	Classification	Total	Per sample
1	2.57715	1.68247	160.91149	165.17111	0.082585555
2	0.94895	1.62555	157.66824	160.24274	0.080121370
3	1.09362	1.57002	156.20857	158.87221	0.079436105
4	1.09827	1.53853	154.27520	156.91200	0.078456000
5	1.01428	1.61315	153.72625	156.35368	0.078176840
6	0.82579	1.47843	152.62470	154.92892	0.077464460
7	0.81273	1.49266	152.45942	154.76481	0.077382405
8	0.97333	0.96459	152.98409	154.92201	0.077461005
9	0.82555	1.50289	155.01370	157.34214	0.078671070
10	1.00392	1.66175	159.47734	162.14301	0.081071505
Average	1.117	1.513	155.535	158.165	0.079
σ	0.524	0.206	2.963	3.461	0.002

When the same test was executed using the Chrome extension the results were substantially slower, but still within imperceptible time. Chromes execution time had an

arithmetic mean of 1.4 milliseconds with a standard deviation of 1.35 milliseconds. This relatively high standard deviation is likely due to how Google Chrome loads and caches extensions when they are first run, and run subsequently.

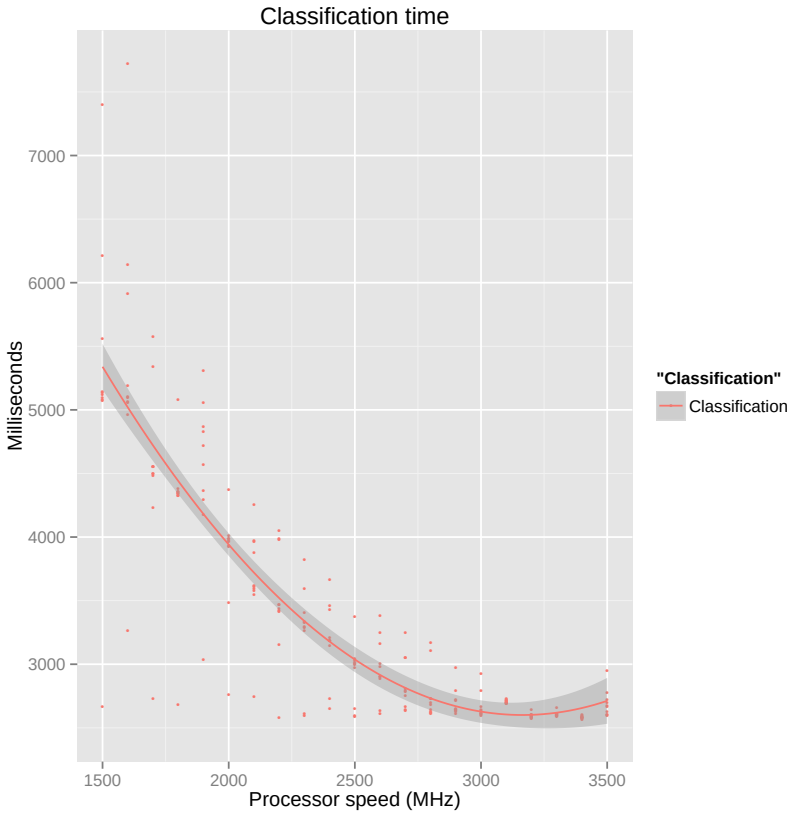
The second series of tests was designed to determine what effect processor clock speed has on the time taken to classify a sample. This test used a Virtual Machine (VM) which was created on an ESXi host to enable adjusting of processor clock speed, and had the test application installed along with all the required data and classifiers. The test was run in 100 MHz increments starting from 1 500 MHz up to 3 500 MHz. For each test run; the VM's processor speed was adjusted and the test application run, using 2 000 samples to calculate timings. This application tested each of the ten classifiers present and the data was logged. Once this was completed, the VM was shut down, its processor clock speed increased and the process repeated until all of the data was gathered. The results from this test are shown in Figure 5.4a. The timings from this test are not directly comparable with the previous test as the hardware used to run the two series is different. This is visible from the data, as tests run at the same clock speed on both hardware configurations differ, with the VM being slower.

5.5.3 Discussion

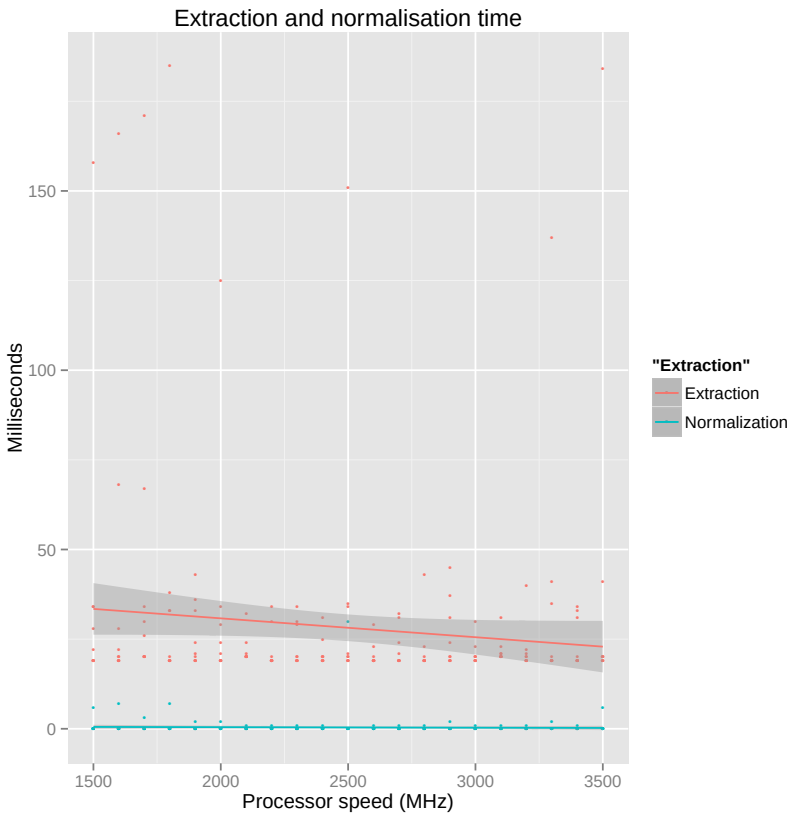
A central part of this research was to determine if these classifiers can be used in real time without adding significant overhead in terms of latency. This is because there are highly accurate services available whose downfall is that they add significant latency which adds up as more requests are made. This is detrimental to the end-user-experience and not usable in large-scale on-the-fly applications. A method of circumventing this problem is to perform these look-ups in batches, before requests are made. For obvious reasons, this approach is not very flexible and cannot classify samples seen for the first time.

The classifiers in this research are only useful if they have low enough execution times to be used by end-users or in large-scale applications such as proxies like those within tertiary education institution and large businesses. These proxies have very strict speed requirements as they can potentially serve thousands of requests per minute. End users have slightly lower requirements, requiring that the classifier executes in an imperceptible time when executing several requests for resources, such as those typical of web pages loading required resources.

It is shown in Section 5.5.2 that an optimal classifier requires 0.07908 milliseconds to classify a URL when in the form of a C# application such as the one implemented in



(a) Classification



(b) Extraction and normalisation

Figure 5.4: Time to extract, normalise and classify 2 000 samples.

Section 4.5. This equates to 1 264.54 requests a second and makes it suitable in large scale proxy applications that serve up to a theoretical maximum of 1 264.54 requests every second.

The Chrome extension was significantly slower due to it being executed as a Java script file within Chrome. It is limited by Chrome's Java script engine and the fact that it is not a compiled language. However, classifying at an average speed of 1.4 Milliseconds, it is still fast enough that it may be used on the client side without incurring significant latencies, even when requesting excessive numbers of URL.

As shown in the final tests (graphed in Figure 5.4a and Figure 5.4b), processor speed has little impact on both extraction and normalisation time when run on processors with speeds common at the time of writing. It is clear that Central Processing Unit (CPU) speed directly and significantly affects the classification step after extraction and normalisation has taken place. However, the longest time taken to classify all 2 000 samples within this test with the CPU clock set to 1 500 MHz, was 7 744 milliseconds. This equates to 3.872 milliseconds per sample, which is deemed acceptable for client side usage. This is further justified by the fact that many cellular phones have CPUs that are approaching this speed or are capable of it already.

5.6 Update size

Within this section, the size of ANNs and their descriptors are determined in relation to the size of the sample set used to classify them. This information is important as the classifier and the data required to rebuild it needs to be transmitted to the end-user as an update. If these updates are prohibitively large, it may not be possible to transmit them and it may discourage end-users from updating their implementations.

5.6.1 Data

As an indication of how the classifier data size increases as more training samples are used, the classifiers trained for the tests described in Section 5.4 were used. This data set consists of 220 sample classifiers, trained in groups of 10 per training sample set size. Also used within these tests are the 10 classifiers generated as part of the optimal set described in Section 5.4.3 which are trained using 18 000 input URLs. Finally; during compression testing, the best ANN from the optimal data set mentioned above is used.

5.6.2 Uncompressed classifiers

Shown in Figure 5.5 are the arithmetical mean sizes of the 220 classifiers. This plot shows that the size of the resulting serialized ANN and its associated data increased linearly as the number of training samples was increased. The arithmetic mean size of the optimal classifiers is 908 983.8 bytes (0.87 Megabytes (MB)).

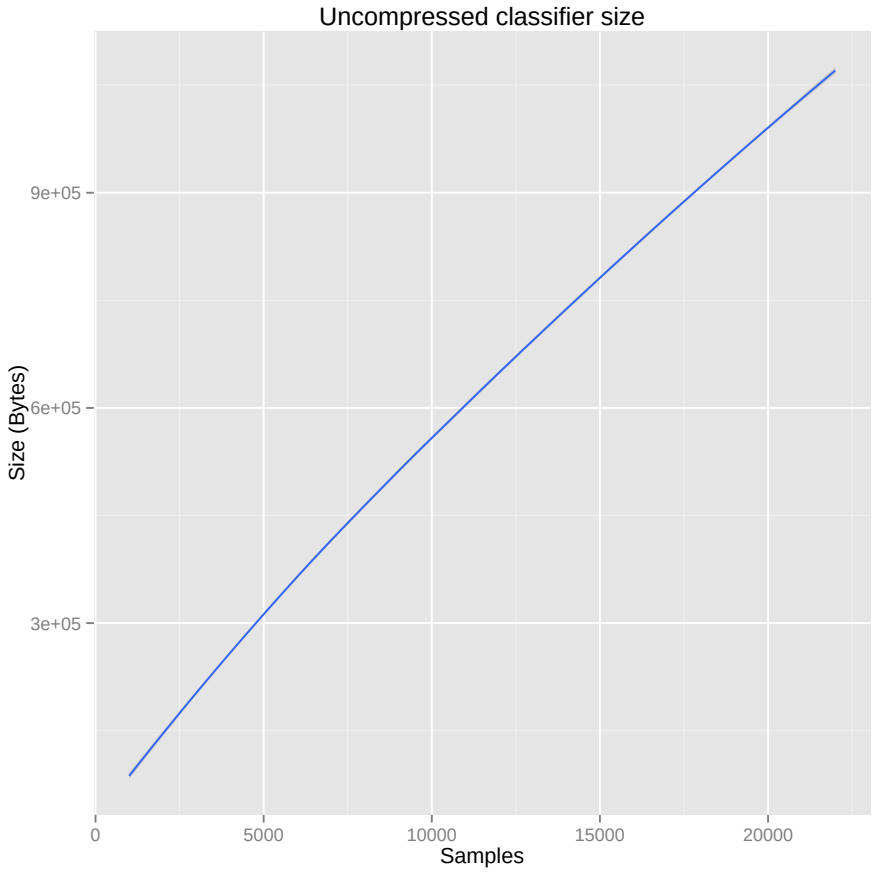


Figure 5.5: Uncompressed size of classifiers based on the number of training samples used.

5.6.3 Compression effectiveness

Further to the size analysis, the effectiveness of compression algorithms when compressing these classifiers is determined. Four different algorithms were tested: *zip*[4] using the *deflate* algorithm, *lz4*[5], *7z*[6] using the *LZMA* algorithm and finally *rar*[7]. The *zip* and *7z* formats

[4]http://www.info-zip.org/
[5]https://code.google.com/p/lz4/
[6]http://www.7-zip.org/
[7]http://www.rarlab.com/

were tested using the 7-zip program for windows. The *lz4* algorithm was tested using the application implemented on Google Code available from https://code.google.com/p/lz4/. Finally, *rar* was tested using the windows installer for the *WinRAR* program.

Table 5.8: Average compression algorithm effectiveness.

Format	Size in bytes	Compression %
Uncompressed	908984	0
zip	323837	35.62625965
Lz4	462989	50.93477993
7z	280827	30.89460321
rar	321585	35.37851051

This optimal classifier generated in Section 5.4 and all of the associated serialized data had an uncompressed size of 908 984 bytes (887.68 Kilobytes (KB)). As can be seen in Table 5.8 is that the *7s* algorithm was the most effective at compressing these classifiers and the data required to rebuild them. It compressed the classifier to 30.89% (274.25 KB) of its original size.

5.6.4 Discussion

There are several reasons why the size of the classifiers covered in this research are important. If the data required to rebuild them at client side are too large, it may be prohibitive to download updates frequently, or at all. This is an obvious issue in the field of online security. Additionally, if the data sets are too large it may make a centralised update distribution service unfeasible due to bandwidth requirements or may add unnecessary strain on this service in terms of bandwidth, processing and storage requirements. Additionally, delta transmission is not applicable as even classifiers trained with the same data set have different weightings. Classifiers built periodically have different input vectors, meaning that the entire OP definition is different from instance to instance, which in turn means that delta transmission would transmit the entire classifier with every update.

In Section 5.6.2 it is shown that the size in terms of bytes increases linearly with an increase in the number of training samples used to create the ANN. However, shown in Section 5.4.3, there is an upper limit in the number of URLs that should be used to generate these classifiers so as to avoid decreased performance. The optimal number was shown to be 18 000 samples.

The size of all the resources required to rebuild the optimal classifier from this range was shown in Section 5.6.2 to be 0.87 MB. While this number is low enough to not deter users

from downloading updates regularly, compression will still offer relief to the bandwidth requirements of a centralised update distribution service, such as that developed in Section 4.6.

To this end, it is shown in Section 5.6.3 that, when using the the LZMA algorithm with 7z, it is possible to reduce the size of the required data to 30.89% of its original size. This results in a reduction of bandwidth required by the distribution service of up to 69.11% in the transmission of updates. Assuming the service receives a new update every 6 hours and has 10 000 clients updating constantly; the uncompressed version would require 33.86 GB of bandwidth every day. Using compressed data, this would result in a bandwidth usage of 10.46 GB of data. On a per-client basis; uncompressed updates would result in 3.47 MB of usage every day, while compressed updates would require 1.07 MB of data usage. If this compression were to be done once, when storing a new update on the service, it would add negligible processing requirements to the overall process.

Due to the complexity required to implement timing of these various algorithms as well as the relatively small sizes of the trained classifiers, testing of the time required to execute these algorithms was not executed.

5.7 Summary

Shown in Table 5.9 is a summary of the metrics for the best ANN trained during the testing phase of this research. The recommendations within this chapter are based upon the group of classifiers to which this ANN belongs.

Table 5.9: Summary of best outcomes.

Samples	Training time (ms)	Accuracy	Sensitivity	Specificity	Compressed size (B)
13 000	635 455	96.80%	0.959	0.977	241 667
14 000	624 402	96.10%	0.963	0.959	227 701
15 000	821 476	96.65%	0.960	0.973	241 869
16 000	875 317	96.40%	0.957	0.971	254 771
17 000	1 010 783	96.85%	0.974	0.963	267 968
18 000	1 111 618	97.20%	0.963	0.981	280 538
19 000	1 203 640	96.85%	0.964	0.973	293 488
20 000	1 482 392	96.60%	0.954	0.978	305 950
21 000	1 533 894	96.60%	0.964	0.968	318 939

When training a classifier for distribution to end-users, a random data set should be selected. As shown in Section 5.4.3, this data set should be 18 000 samples in size,

balanced with a 1 to 1 ratio of positive to negative samples. These samples should be chosen at random from the total data set. A validation set should be chosen that contains no samples which appear in the training data set. It is highly important that these sets are mutually exclusive of each other. This validation set should also be randomly selected and be balanced in terms of positive and negative samples. A testing set should be constructed by the same method.

Additionally, these data sets should be constructed multiple times for each update iteration. This will help to rule out anomalies in the training data set and, due to its random nature, contribute towards finding a data set which best represents a real-world distribution. Multiple ANNs should be trained per data set. This is because the initialisation of these ANNs is random, with final trained versions performing differently. From that large pool of resulting ANNs, the best should be chosen with regards to cumulative error rate, sensitivity and specificity. Within this framework, cumulative error rate is used to identify the best performing classifier.

Shown in Section 5.4.2 is that timing for ANNs created using 18 000 training samples is less than half an hour. Given a system that has 8 cores, it is possible to generate 32 ANNs in a two hour period. Given that there should be multiple random data sets generated; assuming ten are available, 320 ANNs could be trained in a 20 hour period. This is a reasonable distribution of randomness, allowing the best classifier to be chosen. With a dedicated day once a week, any shifts in trends in URL obfuscation can be accounted for.

Classifiers should then be transmitted using the 7z compressed format, discussed in Section 5.6. This will reduce bandwidth requirements on both the update distribution service as well as client implementations by 69% for each update. As a result, clients are more likely to update as each update will be quicker to download and clients with low bandwidth requirements will be less likely to avoid updates due to bandwidth restrictions.

Client implementations using the C# implementation and a classifier such as the one described here, will be able to classify URLs in 0.079 milliseconds with a positive classification have a 96.3% chance of being accurate and a negative classification having a 98.1% chance of being correct. The test application built for this chapter uses the C# DLL implementation of this research, while the Chrome extension discussed in Section 4.7 is an example of an alternative implementation. It has been used extensively during the writing of this thesis and has shown to be highly accurate, while being transparent during normal, benign usage. This shows that this is a viable, final layer protection against phishing URLs.

CHAPTER

SIX

CONCLUSION

The research discussed within this document has covered a method of identifying malicious URLs using ANNs on the client side based on the lexical features of the URL. It has also detailed a method of building a framework to generate these classifiers in an automated fashion; a method that yields the best classifiers given available training data. Methods for storing and transmitting the resulting classifiers were shown as well as an example implementation that uses an ANN to prevent users from accessing suspicious URLs.

The training process was evaluated to determine how quickly these classifiers can be generated for use within an update schedule. The resulting classifiers were tested in terms of several accuracy measures using real-world samples of both positive and negative classifications to determine their effectiveness in a near real-world environment. The classifiers were also evaluated in terms of classification speed to determine what impact they would have on user-experience when using an implementation of this kind. Finally, the information required to be stored and sent to client implementations was used to determine the effectiveness of compression algorithms in an effort to determine the feasibility of a centralized, implementation-agnostic update service.

Background information that is relevant to this research was provided in Chapter 2. This information covered concepts regarding the modern structure of URLs as a method of locating resources on the internet with no knowledge of the content of that resource. Theoretical information regarding what phishing is, how it abuses the structure of a URL

to mask its intent as well as how prevalent it is, was discussed. Additionally, several methods of detecting phishing attacks were shown. Modern approaches to implementing Artificial Neural Networks (ANNs) were presented as well as methods for determining their accuracy when generalising to unseen data. Also covered within this chapter was background information regarding the various technologies used within the implementation of this framework.

In Chapter 3 the algorithmic choices used in the logical formatting of the training, validation and testing data used in this document were discussed. The learning method used when training classifiers was chosen and the algorithm by which this learning was implemented was discussed. The validation and testing framework was also covered, showing how the results from testing are calculated reliably using statistical methods.

An extensible framework for the implementation of a service to generate the classifiers covered in this research was built and shown in Chapter 4. A sample implementation that is able to generate Online Perceptrons (OPs) was developed and tested: a REST service which is able to store these classifiers as well as act as an update service for client implementations was built. Finally, a client implementation which uses this REST service is implemented in the Google Chrome Browser. It is capable of blocking attempts to access phishing resources with a high degree of accuracy by using these classifiers.

Chapter 5 discusses the tests which were performed on the framework and the classifiers it generates to determine their usefulness in the real-world. The framework was tested to determine how much time is required to train usable classifiers. The resulting classifiers were tested for accuracy against how many training samples had been used for each classifier. This was to determine how many samples are needed to generate the best possible classifier and to understand what the accuracy of such a classifier might be in a real world environment. Classification speed was also tested and discussed as it affects the user-experience implications of using this method of identifying malicious URLs. Also determined within this chapter; is the effectiveness of compression algorithms when applied to this kind of data, to determine the impact of compression on an update distribution service.

The formal objectives for this research are outlined in Section 1.2. They are to create an automated framework that is capable of generating ANNs, storing and distributing them. The system implemented for this purpose must be used to identify what data is needed and demonstrate a method of exposing it to client applications while being implementation agnostic. Another requirement is to create a sample end-user implementation that is

capable of demonstrating the capabilities of this research as well as show what options are available to a process once a positive classification has been made.

Other research goals were to determine if the accuracy stated in Le *et al.* (2011) is reachable and the impact that differing numbers of training samples has on this accuracy. The size of the data required to be transmitted for use by end-user applications as well as the effectiveness of various compression algorithms is also to be determined. Finally, the time required to make a classification is to be determined to identify what the effect of using this classification method has on user-experience.

As discussed in this document, the Classifier Generation Service (CGS) is capable of collecting the required data and generating ANNs for use in the lexical analysis of URLs. The Classifier Distribution Service (CDS) shows that REST is an effective method of exposing classifier updates while placing few technological requirements on clients wishing to utilise this service. This document has identified the data that is required to be exposed through this method and, through the use of the CGS, CDS and Net Defence applications, has shown that it works as an end-to-end solution. The Net Defence Google Chome extension has shown how an implementation of this research may function, demonstrated its effectiveness and has also covered several options for action to take when a positive classification is made.

Through multiple rounds of classifier generation and testing, it has been shown that the accuracy claimed in Le *et al.* (2011) is not only attainable, but improvable when using an optimised training data set. It has been shown that classifications made by ANNs of this structure are in imperceptible time, thus making little impact on user experience. It has also been shown that compression using the *7z* algorithm performs best and makes a significant impact in terms of storage and bandwidth requirements on both the distribution service and client implementations.

6.1 Future work

While every effort was made to test this framework and the classifiers that it produces in an *as-close-as-possible* to real-world environment, the fact remains that they were tested in a controlled academic environment. This is useful for testing ideal situations but does not necessarily generalise to the real-world. As a result, future work is recommended that tests the classifiers with data that originates from real-world sources, and the impact that these classifiers have in practice. Work is suggested that increases the scope of the

framework through growing it to include data sources that would train classifiers capable of identifying different types of malicious URLs and combining them, thus allowing for a general classification method for all types of malicious URLs.

6.1.1 Framework functionality

As this framework was developed as a proof of concept, there are several improvements that should be made. These improvements range from increased functionality, usability and security. Without these improvements, the framework suffers from very limited scope. There are several improvements that can be made which include:

- A genetic algorithm could be implemented as part of the ANN generation process. This will allow an automated approach to generating and validating several classifiers at once and will allow for different starting weight configurations to be tested and 'bred' to improve performance and training time. Processors with multiple cores may be leveraged to improve the performance of this process, decreasing the time required to train and find the best possible classifier. The parallelism of Graphics Processing Units (GPUs) may be utilized for this purpose, such as the work shown in Nottingham (2011). As this document deals with a small academic environment, it was not deemed as necessary functionality just reach the goals stated in Section 1.2.
- The service for storing and transmitting updates to client implementations was built as a proof of concept. This service should be adapted in order to compress the data that it stores and transmits as this has been shown to be highly effective. Communication with regards to storing new updates as well as transmitting updates should be implemented over Hyper Text Transfer Protocol Secure (HTTPS) instead of HTTP to improve security and trust. A method for the implementation of partial updates should be explored, as well as processes for implementing update notifications for client applications. While these are useful functions, they are well researched topics and are not necessary within the scope of this research.

6.1.2 Classifiers

The learning method selected to create the classifiers was the *Online Perceptron* method for reasons discussed in Section 3.6. While other methods represent improvements over

the OP, the OP training algorithm is deemed sufficient for the purposes of this research. The following training algorithms should be researched and implemented:

- The Confidence Weighted (CW) and AROW learning methods should be implemented in the form of inducers. This is appropriate as it will allow for further testing regarding these methods which have better training characteristics when compared to the OP method. Implementing these algorithms will also allow for more options to be available to any implementations using the library that is created.
- While this research focused primarily on the detection of phishing URLs, these classifiers have been shown to be able to reliably detect URLs which point to malware resources. They may be further extended to detect botnet command and control servers. This could be done in combination with other classification techniques, such as those discussed in Stalmans (2013). For this reason, separate classifiers should be trained and used together to protect end-users from a wider range of threats. Another form of classifier that could be used to identify suspicious URLs is one that is trained to detect algorithmically generated domain names. Again, this could be combined with the other classifiers to make the scope of protection wider.

6.1.3 Testing

The Net Defence Google Chrome extension was implemented as a proof of concept, to show that client implementations are feasible. However, it was not implemented as an extensively testable application as this is considered outside of the scope of research for this document. It is for this reason that extensive user testing was not performed. These tests should be performed as future research as they will show the impact this classifiers will have on the user-experience. These tests include:

- User tests should be performed to determine how transparent the classifier is when browsing legitimate resources as well as how effective it is at warning users when trying to access fraudulent resources. Additionally, over-all perceived performance should be measured.
- Testing in real-world environments is critical for determining the applicability of this research. This should include tests in environments where performance is crucial, such as use in a proxy within a education institution or large business.

6.1.4 Data

Finally, extended data sources should be found. While the data used in the research was highly useful, they do not represent the full range of URLs used in modern internet technologies. Many modern sites use technologies such as Asynchronous JavaScript and XML (AJAX) to make asynchronous API calls for normal operations. While testing the Net Defence implementation, it became apparent that the classifier is sensitive to these API calls, making false positive classifications. For this reason, data that is more representative of real-world data should be found and used during the training of these classifiers. The best approach may be to use a proxy with strict rules to generate a reasonably clean data set and then to use the AROW training algorithm to generate the classifier due to its ability to handle potentially noisy data sets over the current perceptron implementation.

REFERENCES

Aaron, G. and Rasmussen, R. *Global phishing survey: Trends and domain name use in 2h2012*. Technical report, Anti Phishing Working Group, http://apwg.org/, April 2012. Last accessed: 05/11/2013.
URL `http://docs.apwg.org/reports/APWG_GlobalPhishingSurvey_2H2012.pdf`

Aaron, G. and Rasmussen, R. *Global phishing survey: Trends and domain name use in 1h2013*. Technical report, Anti Phishing Working Group, http://apwg.org/, September 2013. Last accessed: 05/11/2013.
URL `http://docs.apwg.org/reports/APWG_GlobalPhishingSurvey_1H2013.pdf`

Anandpara, V., Dingman, A., Jakobsson, M., Liu, D., and Roinestad, H. *Phishing IQ tests measure fear, not ability.* In **Dietrich, S. and Dhamija, R.**, editors, *Financial Cryptography and Data Security*, volume 4886 of *FC'07/USEC'07*, pages 362–366. Springer-Verlag, Berlin, Heidelberg, February 2007. ISBN 9783540773658.

Androutsopoulos, I., Koutsias, J., V. Chandrinos, K., Paliouras, G., and D. Spyropoulos, C. *An evaluation of naive bayesian anti-spam filtering.* In **Mántaras, R. L. d. and Plaza, E.**, editors, *Proceedings of the Workshop on Machine Learning in the New Information Age (11th European Conference on Machine Learning)*, pages 9–17. Springer-Verlag, May 2000. ISBN 3540676023.

Arlot, S. and Celisse, A. *A survey of cross-validation procedures for model selection. Statistics Surveys*, 4:40–79, 2010.
URL `http://www.di.ens.fr/sierra/pdfs/2010_Arlot_Celisse_SS.pdf`

Baldi, P., Brunak, S., Chauvin, Y., Andersen, C. A., and Nielsen, H. *Assessing the accuracy of prediction algorithms for classification: an overview. Bioinformatics*, 16(5):412–424, 2000.
URL `http://bioinformatics.oxfordjournals.org/content/16/5/412`

Barth, A., Felt, A. P., Saxena, P., and Boodman, A. *Protecting browsers from extension vulnerabilities.* In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, NDSS '10. The Internet Society, March 2010.
URL `http://www.internetsociety.org/doc/protecting-browsers-extension-vulnerabilities`

Batista, G. E., Prati, R. C., and Monard, M. C. *A study of the behavior of several methods for balancing machine learning training data. ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD) Explorations Newsletter*, 6(1):20–29, June 2004. ISSN 1931-0145. doi: 10.1145/1007730.1007735.
URL `http://sci2s.ugr.es/keel/dataset/includes/catImbFiles/2004-Batista-SIGKDD.pdf`

Battle, R. and Benson, E. *Bridging the Semantic Web and Web 2.0 with Representational State Transfer (REST). The Journal of Web Semantics*, 6(1):61–69, February 2008. ISSN 1570-8268. doi: 10.1016/j.websem.2007.11.002.
URL `www.websemanticsjournal.org/index.php/ps/article/download/135/133`

Beitzel, S. *On understanding and classifying web queries.* Ph.D. thesis, Illinois Institute of Technology, 2006. AAI3220872.

Bengio, Y. and Grandvalet, Y. *No unbiased estimator of the variance of k-fold cross-validation. The Journal of Machine Learning Research (JMLR)*, 5:1089–1105, December 2004. ISSN 1532-4435.

Berghel, H. *Phishing mongers and posers. Communications of the ACM (CACM)*, 49(4):21–25, April 2006. ISSN 0001-0782.
URL `http://doi.acm.org/10.1145/1121949.1121968`

Berners-Lee, T. *Uniform Resource Locators (URL).* December 1994. RFC 1738.
URL `http://www.ietf.org/rfc/rfc1738.txt`

Berners-Lee, T. *Uniform Resource Identifiers (URI): Generic syntax.* January 2005. RFC 3986.
URL `http://www.ietf.org/rfc/rfc3986.txt`

Boyko, E. J. *Ruling out or ruling in disease with the most sensitive or specific diagnostic test: short cut or wrong turn? Medical Decision Making (MDM)*, 14(2):175–179, 1994.
URL `http://mdm.sagepub.com/content/14/2/175.short`

Brody, R. G., Mulig, E., and Kimball, V. *Phishing, pharming and identity theft. Academy of Accounting and Financial Studies Journal*, 11(3):43–56, 2007.

Cawley, G. C. and Talbot, N. L. *Efficient leave-one-out cross-validation of kernel Fisher discriminant classifiers. Pattern Recognition*, 36(11):2585–2592, 2003.

Chandrasekaran, M., Chinchani, R., and Upadhyaya, S. *Phoney: Mimicking user response to detect phishing attacks.* In *Proceedings of the International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, WoWMoM '06, pages 668–672. IEEE Computer Society, Washington, DC, USA, June 2006a. ISBN 0-7695-2593-8. doi: 10.1109/WOWMOM.2006.87.

Chandrasekaran, M., Narayanan, K., and Upadhyaya, S. *Phishing email detection based on structural properties.* In *Proceedings of the New York State Cyber Security Conference (NYSCSC)*, NYSCSC '06, pages 1–7. June 2006b.
URL `http://www.albany.edu/wwwres/conf/iasymposium/proceedings/2006/chandrasekaran.pdf`

Chapelle, O., Vapnik, V., Bousquet, O., and Mukherjee, S. *Choosing multiple parameters for support vector machines. Machine Learning*, 46(1-3):131–159, March 2002. ISSN 0885-6125.
URL `http://dx.doi.org/10.1023/A:1012450327387`

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. *Smote: Synthetic Minority Over-sampling Technique. Journal of Artificial Intelligence Research (JAIR)*, 16(1):321–357, June 2002. ISSN 1076-9757.

Chou, N., Ledesma, R., Teraguchi, Y., Boneh, D., and Mitchell, J. C. *Client-side defense against web-based identity theft.* In *Proceedings of the 11th Annual Symposium on Network and Distributed System Security (NDSS)*, NDSS '04. The Internet Society, February 2004. ISBN 1891562185, 1891562177.
URL `http://www.internetsociety.org/doc/client-side-defense-against-web-based-identity-theft`

Chun, W. J. *Core Python Programming.* Prentice Hall, Upper Saddle River, New Jersey, USA, 2nd edition, September 2006. ISBN 0132269937.

Cormack, G. V. and Lynam, T. R. *Online supervised spam filter evaluation. ACM Transactions on Information Systems (TOIS)*, 25(3):1–31, July 2007. ISSN 1046-8188. doi: 10.1145/1247715.1247717.

Crammer, K., Dredze, M., and Pereira, F. *Exact convex confidence-weighted learning.* In **Koller, D., Schuurmans, D., Bengio, Y., and Bottou, L.**, editors, *Advances in Neural Information Processing Systems 21 (NIPS)*, NIPS '08, pages 345–352. Curran Associates Incorporated, 2008. ISBN 9781605609492.
URL http://www.cs.jhu.edu/~mdredze/publications/cw_nips_08.pdf

Crammer, K., Kulesza, A., and Dredze, M. *Adaptive regularization of weight vectors.* *Machine Learning*, 91(2):1–33, May 2009. ISSN 0885-6125. doi: 10.1007/s10994-013-5327-x.
URL http://www.cs.jhu.edu/~mdredze/publications/nips09_arow.pdf

Dhamija, R. and Tygar, J. D. *The battle against phishing: Dynamic security skins.* In *Proceedings of 1st Symposium on Usable Privacy and Security (SOUPS)*, SOUPS '05, pages 77–88. ACM, New York, NY, USA, July 2005. ISBN 1-59593-178-3. doi: 10.1145/1073001.1073009.
URL http://cups.cs.cmu.edu/soups/2005/2005proceedings/p77-dhamija.pdf

Dhamija, R., Tygar, J. D., and Hearst, M. *Why phishing works.* In **Grinter, R., Rodden, T., Aoki, P., Cutrell, E., Jeffries, R., and Olson, G.**, editors, *Proceedings of the Special Interest Group on Computer-Human-Interaction conference on Human Factors in Computing Systems (CHI)*, CHI '06, pages 581–590. ACM, New York, NY, USA, April 2006. ISBN 1-59593-372-7. doi: 10.1145/1124772.1124861.608060.

Egan, S. and Irwin, B. *An evaluation of lightweight classification methods for identifying malicious URLs.* In *Proceedings of Information Security South Africa (ISSA)*, ISSA '11, pages 1–6. August 2011a. ISBN 978-1-4577-1481-8. doi: 10.1109/ISSA.2011.6027532.
URL http://icsa.cs.up.ac.za/issa/2011/Proceedings/Full/49_Paper.pdf

Egan, S. and Irwin, B. *High speed classification of malicious URLs.* In *Proceedings of the 14th Annual South African Telecommunication Networks and Application Conference (SATNAC)*, SATNAC '11. September 2011b.
URL http://www.satnac.org.za/proceedings/2011/papers/Work_In_Progress/Internet_Services_and_Applications/237.pdf

Egan, S. and Irwin, B. *An analysis and implementation of methods for high speed lexical classification of malicious URLs.* In *Proceedings of Information Security South Africa (ISSA)*, ISSA '12. August 2012a.

URL http://icsa.cs.up.ac.za/issa/2012/Proceedings/Research/58_ResearchInProgress.pdf

Egan, S. and Irwin, B. *Normandy: A framework for implementing high speed lexical classifcation of malicious URLs.* In *Proceedings of the 15th Annual South African Telecommunication Networks and Application Conference (SATNAC)*, SATNAC '12. September 2012b.
URL http://www.satnac.org.za/proceedings/2012/papers/WIP_4.Internet_Services_End_User_Applications/175.pdf

Elser, D. and Pekrul, M. *Inside the password-stealing business: the who and how of identity theft.* 2009. Last accessed: 03/11/2013.
URL http://www.mcafee.com/us/resources/reports/rp-inside-password-stealing-biz.pdf

EMC. *Hacktivism and the case of something phishy.* Technical report, RSA, 2013. Last accessed: 11/11/2013.

Estabrooks, A., Jo, T., and Japkowicz, N. *A multiple resampling method for learning from imbalanced data sets. Computational Intelligence*, 20(1):18–36, 2004. doi: 10.1111/j.0824-7935.2004.t01-1-00228.x.
URL http://sci2s.ugr.es/docencia/doctoM6/Estrabrooks.pdf

Fette, I., Sadeh, N., and Tomasic, A. *Learning to detect phishing emails.* In *Proceedings of the 16th International World Wide Web Conference (WWW)*, WWW '07, pages 649–656. ACM, New York, NY, USA, May 2007. ISBN 978-1-59593-654-7. doi: 10.1145/1242572.1242660.
URL http://www.cs.cmu.edu/~tomasic/doc/2007/FetteSadehTomasicWWW2007.pdf

Fielding, R. T. *Architectural styles and the design of network-based software architectures.* Ph.D. thesis, University of California, 2000. AAI9980887.

Flanagan, D. *JavaScript: The Definitive Guide.* O'Reilly, Sebastopol, California, USA, 3rd edition, July 1998. ISBN 1565923928.

Freund, Y. and Schapire, R. E. *Large margin classification using the Perceptron algorithm. Machine Learning*, 37(3):277–296, December 1999. ISSN 0885-6125. doi: 10.1023/A:1007662407062.
URL http://cseweb.ucsd.edu/~yfreund/papers/LargeMarginsUsingPerceptron.pdf

Gardner, M. and Dorling, S. *Artificial Neural Networks (the multilayer perceptron)–a review of applications in the atmospheric sciences. Atmospheric environment*, 32(14-15):2627–2636, 1998. ISSN 1352-2310.
URL http://research.eeescience.utoledo.edu/lees/papers_pdf/Gardner_1998_AtmEnviron.pdf

Garera, S., Provos, N., Chew, M., and Rubin, A. D. *A framework for detection and measurement of phishing attacks.* In *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM)*, WORM '07, pages 1–8. ACM, New York, NY, USA, November 2007. ISBN 978-1-59593-886-2. doi: 10.1145/1314389.1314391.
URL http://mmnet.iis.sinica.edu.tw/botnet/file/20100524/20100524_1.pdf

Google. *Google chrome extensions overview.* 2013a. Last accessed: 24/10/2013.
URL https://developer.chrome.com/extensions/overview.html

Google. *Google chrome manifest overview.* 2013b. Last accessed: 24/10/2013.
URL https://developer.chrome.com/extensions/manifest.html

Google. *Google safe browsing api.* 2013c. Last accessed: 24/10/2013.
URL https://developers.google.com/safe-browsing/?csw=1

Guo, H. and Viktor, H. L. *Learning from imbalanced data sets with boosting and data generation: the databoost-im approach. ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD) Explorations Newsletter*, 6(1):30–39, June 2004. ISSN 1931-0145. doi: 10.1145/1007730.1007736.

Hong, J. *The state of phishing attacks. Communications of the ACM (CACM)*, 55(1):74–81, January 2012. ISSN 0001-0782. doi: 10.1145/2063176.2063197.
URL http://cacm.acm.org/magazines/2012/1/144811-the-state-of-phishing-attacks/fulltext

Hornik, K., Stinchcombe, M., and White, H. *Multilayer feedforward networks are universal approximators. Neural Networks*, 2(5):359–366, July 1989. ISSN 0893-6080. doi: 10.1016/0893-6080(89)90020-8.
URL http://weber.ucsd.edu/~hwhite/pub_files/hwcv-028.pdf

Jagatic, T. N., Johnson, N. A., Jakobsson, M., and Menczer, F. *Social phishing. Communications of the ACM (CACM)*, 50(10):94–100, October 2007. ISSN 0001-0782. doi: 10.1145/1290958.1290968.
URL http://www.indiana.edu/~phishing/social-network-experiment/phishing-preprint.pdf

Jain, A. K., Mao, J., and Mohiuddin, K. M. *Artificial Neural Networks: A tutorial. Computer*, 29(3):31–44, March 1996. ISSN 0018-9162. doi: 10.1109/2.485891.

Japkowicz, N. and Stephen, S. *The class imbalance problem: A systematic study. Intelligent Data Analysis Journal (IDA)*, 6(5):429–449, October 2002. ISSN 1088-467X.

Kearns, M. and Ron, D. *Algorithmic stability and sanity-check bounds for leave-one-out cross-validation.* In *Proceedings of the 10th Annual Conference on Computational Learning Theory (COLT)*, COLT '97, pages 152–162. ACM, New York, NY, USA, July 1999. ISBN 0-89791-891-6. doi: 10.1145/267460.267491.
URL `http://www.eng.tau.ac.il/~danar/Public-pdf/leave-one-out-long.pdf`

Kernighan, B. *The C programming language.* Prentice Hall, Upper Saddle River, New Jersey, USA, 2nd edition, April 1988. ISBN 0131103709.

Kirda, E. and Kruegel, C. *Protecting users against phishing attacks with antiphish.* In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC)*, COMPSAC '05, pages 517–524. IEEE Computer Society, Washington, DC, USA, July 2005. ISBN 0-7695-2413-3. doi: 10.1109/COMPSAC.2005.126.
URL `http://cs.ucsb.edu/~chris/research/doc/compsac05_antiphish.pdf`

Kohavi, R. *A study of cross-validation and bootstrap for accuracy estimation and model selection.* In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2 of *IJCAI'95*, pages 1137–1145. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, August 1995. ISBN 1-55860-363-8.

Le, A., Markopoulou, A., and Faloutsos, M. *Phishdef: Url names say it all.* In *Proceedings of 30th International Conference on Computer Communications (INFOCOM)*, INFOCOM '11, pages 191–195. IEEE Computer Society, Washington, DC, USA, April 2011. ISBN 978-1-4244-9919-9. ISSN 0743-166X. doi: 10.1109/INFCOM.2011.5934995.

Li, K. and Zhong, Z. *Fast statistical spam filter by approximate classifications.* In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, SIGMETRICS '06, pages 347–358. ACM, New York, NY, USA, 2006. ISBN 1-59593-319-0. doi: 10.1145/1140277.1140317.

Liberty, J. *Programming C#: Building .NET applications with C#.* O'Reilly, Sebastopol, California, USA, 1st edition, December 2001. ISBN 0596001177.

Lippmann, R. *An introduction to computing with neural nets. IEEE Acoustics, Speech and Signal Processing Society (ASSP) Magazine*, 16(1):7–25, March 1988. ISSN 0163-

5964.
URL http://doi.acm.org/10.1145/44571.44572

Litan, A. *Phishing attack victims likely targets for identity theft.* May 2004. Last accessed: 30/10/2013.
URL http://www.gartner.com/id=431660

Ma, J., Saul, L. K., Savage, S., and Voelker, G. M. *Beyond blacklists: learning to detect malicious web sites from suspicious URLs.* In *Proceedings of the 15th International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, KDD '09, pages 1245–1254. ACM, New York, NY, USA, June 2009a. ISBN 978-1-60558-495-9. doi: 10.1145/1557019.1557153.
URL http://cseweb.ucsd.edu/~saul/papers/kdd09_url.pdf

Ma, J., Saul, L. K., Savage, S., and Voelker, G. M. *Identifying suspicious URLs: an application of large-scale online learning.* In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, ICML '09, pages 681–688. ACM, New York, NY, USA, June 2009b. ISBN 978-1-60558-516-1. doi: 10.1145/1553374.1553462.
URL http://cseweb.ucsd.edu/~savage/papers/ICML09.pdf

McGrath, D. K. and Gupta, M. *Behind phishing: An examination of phisher modi operandi.* In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, volume 8 of *LEET '08*, pages 1–8. USENIX Association, Berkeley, CA, USA, 2008.

Microsoft. *Microsoft Developer Network naming guidlines.* 2013a. Last accessed: 24/10/2013.
URL http://msdn.microsoft.com/en-us/library/x2dbyw72(v=vs.71).aspx

Microsoft. *Microsoft smart screen.* 2013b. Last accessed: 24/10/2013.
URL http://windows.microsoft.com/en-US/windows-vista/SmartScreen-Filter-frequently-asked-questions

Milletary, J. *Technical trends in phishing attacks.* Technical report, US-CERT, 2005. Last accessed: 1/11/2013.

Mohamed, A. *Phishing and social engineering techniques.* April 2013. Last accessed: 03/11/2013.
URL http://resources.infosecinstitute.com/phishing-and-social-engineering-techniques/

Nadel, L. and Stein, D. L. *1993 lectures in complex systems.* Westview Press, Boulder, Colorado, USA, 1st edition, August 1995. ISBN 0201483688.

Nottingham, A. *GPF: A framework for the general packet classification on GPU co-processors.* Master's thesis, Rhodes University, Grahamstown, South Africa, October 2011.

Pautasso, C., Zimmermann, O., and Leymann, F. *Restful web services vs. "big" web services: making the right architectural decision.* In *Proceedings of the 17th International World Wide Web Conference (WWW)*, WWW '08, pages 805–814. ACM, New York, NY, USA, April 2008. ISBN 978-1-60558-085-2. doi: 10.1145/1367497.1367606.

Peduzzi, P., Concato, J., Kemper, E., Holford, T. R., and Feinstein, A. R. *A simulation study of the number of events per variable in logistic regression analysis. Journal of Clinical Epidemiology*, 49(12):1373–1379, 1996.

Powers, D. *Evaluation: From precision, recall and f-measure to ROC, informedness, markedness and correlation. Journal of Machine Learning Technologies*, 2(1):37–63, 2011.
URL `http://www.bioinfo.in/uploadfiles/13031311552_1_1_JMLT.pdf`

Python-Software. *Python documentation glossary.* 2013. Last accessed: 01/11/2013.
URL `http://docs.python.org/2/glossary.html`

Ramachandran, A., Dagon, D., and Feamster, N. *Can dns-based blacklists keep up with bots?* In *Proceedings of the 3rd Conference on Email and Anti-Spam (CEAS)*, CEAS '06. July 2006.
URL `http://ceas.cc/2006/14.pdf`

Rao, C. and Wu, Y. *Linear model selection by cross-validation. Journal of Statistical Planning and Inference*, 128(1):231–240, 2005. doi: 10.1080/01621459.1993.10476299.

Riedmiller, M. *Advanced supervised learning in multi-layer perceptrons - from backpropagation to adaptive learning algorithms. Computer Standards and Interfaces*, 16(3):265–278, 1994.

Rosenblatt, F. *The perceptron: a probabilistic model for information storage and organization in the brain. Psychological Review*, 65(6):386, 1958.
URL `http://blogimg.chinaunix.net/blog/upfile2/090505101133.pdf`

Rowley, H. A., Baluja, S., and Kanade, T. *Neural network-based face detection. IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 20(1):23–38, January 1998. ISSN 0162-8828. doi: 10.1109/34.655647.

Sahami, M., Dumais, S., Heckerman, D., and Horvitz, E. *A bayesian approach to filtering junk e-mail.* In *Proceedings of the Workshop on Learning for Text Categorization*, volume 62, pages 98–105. July 1998.
URL `http://cs294-28mulespam.googlecode.com/svn-history/r113/trunk/paper/references/sahami-bayesian-approach.pdf`

Schult, W. and Polze, A. *Aspect-Oriented Programming with C# and .NET.* In *Proceedings of the 5th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, ISORC '02, pages 241–248. IEEE Computer Society, Washington, DC, USA, April 2002. ISBN 0-7695-1558-4.

Sheng, S., Wardman, B., Warner, G., Cranor, L., Hong, J., and Zhang, C. *An empirical analysis of phishing blacklists.* In *Proceedings of the 6th Conference on Email and Anti-Spam (CEAS)*, CEAS '09. July 2009.
URL `http://ceas.cc/2009/papers/ceas2009-paper-32.pdf`

Stalmans, E. *DNS Traffic Based Classifiers for the Automatic Classification of Botnet Domains.* Master's thesis, Rhodes University, Grahamstown, South Africa, June 2013.

Svozil, D., Kvasnicka, V., and Pospichal, J. *Introduction to multi-layer feed-forward neural networks. Chemometrics and Intelligent Laboratory Systems*, 39(1):43–62, 1997.
URL `http://ich.vscht.cz/~svozil/pubs/nn_intro.pdf`

Tetko, I. V., Livingstone, D. J., and Luik, A. I. *Neural network studies. 1. comparison of overfitting and overtraining. Journal of Chemical Information and Computer Sciences*, 35(5):826–833, 1995. doi: 10.1021/ci00027a006.

Vapnik, V. and Chapelle, O. *Bounds on error expectation for support vector machines. Neural Computation*, 12(9):2013–2036, September 2000. ISSN 0899-7667. doi: 10.1162/089976600300015042.

West, A. G. and Lee, I. *Towards the effective temporal association mining of spam blacklists.* In *Proceedings of the 8th Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference (CEAS)*, CEAS '11, pages 73–82. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0788-8. doi: 10.1145/2030376.2030385.

Windley, P. J., Daley, D., Cutler, B., and Tew, K. *Using reputation to augment explicit authorization.* In *Proceedings of the 2007 ACM Workshop on Digital Identity Management (DIM)*, DIM '07, pages 72–81. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-889-3. doi: 10.1145/1314403.1314416.

Wu, M., Miller, R. C., and Garfinkel, S. L. *Do security toolbars actually prevent phishing attacks?* In *Proceedings of the Special Interest Group on Computer-Human-Interaction conference on Human Factors in Computing Systems (CHI)*, CHI '06, pages 601–610. ACM, New York, NY, USA, April 2006. ISBN 1-59593-372-7. doi: 10.1145/1124772.1124863.

Zhang, L., Zhu, J., and Yao, T. *An evaluation of statistical spam filtering techniques. ACM Transactions on Asian Language Information Processing (TALIP)*, 3(4):243–269, December 2004. ISSN 1530-0226. doi: 10.1145/1039621.1039625.

Zhang, Y., Hong, J. I., and Cranor, L. F. *Cantina: a content-based approach to detecting phishing web sites.* In *Proceedings of the 16th International World Wide Web Conference (WWW)*, WWW '07, pages 639–648. ACM, New York, NY, USA, May 2007. ISBN 978-1-59593-654-7. doi: 10.1145/1242572.1242659.

APPENDIX

A

FRAMEWORK COMPONENTS AND INTERFACES

Contained within this appendix is a list and short description of each of the components contained in the *Network Generation Framework* as described in Section 4.5. As a primary concern of this framework, extendability is an important goal, as well as maintainability. This is achieved through the use of dependancy injection using interfaces as a method of swapping implementations without modifying the framework. Part of the research is to create a proof of concept which is to use the supplied interfaces in a working implementation. All of the source code described here is available for download, the details of which are supplied in Appendix E.

Section A.1 to Section A.8 describe interfaces and their defined method contracts, while the Section A.9 to Section A.13 describe the test implementation and justifications for their design decisions. Further than this, the components are listed in no particular order. The final sections within this appendix discuss two classes which do not implement interfaces, but are used to aid in dependency injection and configuration management. The relationship between all of the entities described here is depicted in Figure 4.4.

A.1 Extractor

The *Extractor* interface defines a single *getArray()* method which takes the various types of BOWs as parameters. Each URL within the framework, imported as training, validation,

malicious or benign data is represented as an *Extractor* instance. This interface is meant to allow the framework to request each extractor object is able to create a data array that represents an input vector for an ANN.

A.2 NetworkBackup

The *NetworkBackup* interface defines two methods, the *backupNetwork()* and *loadNetwork()*. On each iteration through the training data, the ANN is evaluated by the framework, and if found to be the best network so far, it is saved. This interface allows the method of saving and restoring of the ANN to be abstracted. The first method, *backupMethod()* takes the ANN to be saved, as well as the configuration instance which contains the path to the destination of where the data should be saved, as parameters. The *loadNetwork()* method takes the configuration instance as a parameter and returns an instantiation of the ANN found at the location found within the configuraiton instance.

A.3 NetworkDataBalancer

The *balance()* method defined by the *NetworkDataBalancer* interface takes 2 arrays of any type as input. Implementations of this interface should balance the data using the desired method, modifying the array that are passed as parameter references.

A.4 DataFetcher

The *DataFetcher* interface defines a *fetchDataArray()* method which implementing classes should implement to fetch URLs from a data source. These urls are returned in an array of strings.

A.5 NetworkInducer

As the main component of this framework, a network inducer is responsible for generating ANNs. This interface should be implemented as the main component of any application

implementing this research. It defines two methods, the first being *initInducer()* allows the application to perform initialisation operations such as data loading, data extraction and BOW extraction. The second method defined, *generateNetwork()* returns a trained and validated ANN suitable for updating client applications.

A.6 NetworkPersistor

This interface is defines a method interface which is used by the framework to store trained ANNs on a resource which is accessible by client consumer applications. The abstraction of persistance is to allow the developer to decide method by which these updates are transmitted. Developers may add features such as compression, partial transfer or differing transmission methods such as File Transfer Protocol (FTP).

A.7 NetworkValidator

The *Validator* interface abstracts validation of ANNs and defines methods which, appart from ANN validation, allow for various statistics to be fetched. These statistics are universal to different validation methods and as such, are useful when trying to determine the best resulting network from a training run. Statistics which may be requested from *Validator* instances are *False Positive Rate (FPR)*, *False Negative Rate(FNR)*, *accuracy* and *specificity*.

A.8 Logger

Allowing applications to choose their own method of recieving or outputting logs from the framework is achieved through the use of the *Logger* interface. It has two logging methods which allow it to simply log messages through the *log()* method, or to log training and validation progress through the *logProgress()* method.

A.9 ConsoleLogger

The *Logger's* purpose is to allow applications to log to various outputs at different stages of the training process. The *ConsoleLogger* is in implementation of the *Logger* interface and

sends the frameworks output promts to *stdout*. The label parameter is used to show which component is logging while the message parameter is the actual data to be outputted. The *logProgress()* method displays two progress bars, one which shows the progression from 0 to the threshold for iterations allowed by the program, and a second visualizes the accuracy of the best performing network version so far.

A.10 DataRemovalNetworkBalancer

As an implementation of the *NetworkDataBalancer*, the *DataRemovalNetworkBalancer* takes two arrays of any type and adjusts them until they are equal size. As discussed in Section 2.4.3, this is to avoid unnatural bias as a result of unequal data sets. Within this implementation, this is done by finding the smallest the of the two data sets, and removing items from the larger of the two sets. This is accomplished by selecting the all of the items in the larger array that appear first up to the index of the smallest array size.

A.11 FileDataFetcher

This implementation of the *DataFetcher* class fetches URLs from a file. Each URL is separated by a new line and carriage return character. Four instances of this class are used within this implementation, two which fetch malicious and benign training data, and two which fetch malicious and benign validation data. Other implementations may be used in conjunction with this implementation which can fetch data from a database, web service or other resources.

A.12 FileNetworkBackup

Like the *FileDataFetcher*, this implementation uses data files to achieve its goals. It implements the *NetworkBackup* interface and is used to store different itterations of an ANN during the training phase of execution. When an iteration through the training data is completed, the ANN is validated, at which point this class is used to instantiate the best ANN stored. The accuracy of the current iteration is compared against the newly instantiated one and is then stored to the file if its accuracy is higher.

A.13 SinglePassNetworkValidator

This implementation of the *NetworkValidator* uses a set of URLs not used by the training process. Each of these URLs is classified by the network, and the *SinglePassNetworkValidator* counts each incorrect classification made by the network. These metrics are then used to calculate the accuracy of the ANN.

A.14 NetworkTrainingData

Both the training and validation data are loaded by implementations of the *DataFetcher* interface, these implementations are all stored within the *NetworkTrainingData* object. This object acts as a container for these, potentially different, instances and also has the ability to validate these instances by checking their types. Using this approach, all data can be found with a single reference to this instance, which makes the data source a clean and maintainable object to handle. This object is held in the configuration object, discussed in the following section (Section A.15).

A.15 NetworkTrainingConfiguration

This object is central to the framework's dependency injection model. It uses an XML file (see Appendix B) to load the frameworks configuration, defining which implementations of each interface to use. Each implementation is instantiated and stored as a public member of the *NetworkTrainingConfiguration* instance. Once the loading is completed, the instance is validated to make sure that all required implementations are loaded. The framework is then passed this instance for access to all of the contained implementations. This object is central to this model and is used in all aspects of training, validation and transmission of the ANN.

APPENDIX

B

CLASSIFIER GENERATION SERVICE CONFIGURATION

Listing B.1 shows an example configuration file for use with the Classifier Distribution Service (CDS). This file is used to select which implementations of the the framework's instances to use through dependency injection and affects what behavior the framework has for each configurable option within the training and distribution of ANNs.

Listing B.1: configuration.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<network>
  <training>
    <epochs>100</epochs>
    <error>0.004</error>
    <rate>1</rate>
    <normalizer>DescriptorNormalizer</normalizer>
    <balancer>DataRemoval</balancer>
    <logger>ConsoleLogger</logger>
    <persistence resource="http://localhost:8664/">
       RestNetworkPersister</persistence>
    <backup method="FileNetworkBackup">network.vnn</backup>
    <data>
      <training>
        <malicious>malicious.csv</malicious>
        <benign>benign.csv</benign>
      </training>
      <validation>
        <validator>SinglePassValidator</validator>
        <malicious>malicious_validation.csv</malicious>
        <benign>benign_validation.csv</benign>
      </validation>
    </data>
  </training>
</network>
```

APPENDIX

C

NET DEFENCE

Shown in Listing C.1 is the manifest file used to determine the security options for the Net Defence Google Chrome extension. Additionally, this file sets other configurable parameters such as icons and the file that must be used to set user options for the extension.

The algorithm used by the Net Defence extension to determine if a URL will be blocked is shown in Listing C.2. Within this algorithm a function is created which first checks if the currently requested URL has been permanently allowed by the user. If it hasn't been allowed, it is then checked against a list of both false positives and false negatives and allowed access or denied if it matches a URL in the appropriate list. The final check is to use the ANN to predictively determine the nature of the resource based on its lexical features and is allowed or denied based on the outcome of this classification.

Listing C.1: Net Defence manifest file

```
{
    "name": "Net Defence",
    "manifest_version": 2,
    "description": "Net Defence",
    "icons": {
        "16": "img/urlblock16.png",
        "48": "img/urlblock48.png",
        "128": "img/urlblock128.png"
    },
    "permissions": [
        "webRequest",
        "webRequestBlocking",
        "<all_urls>",
        "storage",
        "notifications",
        "tabs"
    ],
    "web_accessible_resources" : [
        "img/urlblock48.png"
    ],
    "background": {
        "scripts": [
            "js/lib/URLExtractor.js",
            "js/lib/OnlinePerceptron.js",
            "js/lib/Inducer.js",
            "js/lib/DescriptorNormalizer.js",
            "js/core.js"
        ]
    },
    "browser_action": {
        "default_icon": "img/urlblock16.png",
        "default_popup": "html/falsenegative.html"
    },
    "options_page": "html/interface.html"
}
```

Listing C.2: Event listener registration and high-level classification algorithm

```
var checkURL = function(details) {
    var urlString = details.url.toString();
    var settings = JSON.parse(localStorage['appDetails']);

    if (isWhiteListed(urlString)) {
        return allowRequest(details, settings, "Trusted");
    } else if (isFalsePositive(urlString)) {
        return allowRequest(details, settings, "False
           Positive");
    } else if (isFalseNegative(urlString)) {
        return blockRequest(details, settings, "False
           Negative");
    } else if (isMalicious(urlString)) {
        return blockRequest(details, settings, "Malicious
           Classification");
    } else {
        return allowRequest(details, settings, "Benign
           Classification");
    }
};

chrome.webRequest.onBeforeRequest.addListener(checkURL, {urls
    : ["<all_urls>"]}, ["blocking"]);
```

APPENDIX

D

LIST OF PUBLICATIONS

Shown here are relevant publications by the author of this document during the time in which this research was taking place:

Egan, S. and Irwin, B. *High speed classification of malicious URLs.* In *Proceedings of the 14th Annual South African Telecommunication Networks and Application Conference (SATNAC)*, SATNAC '11. September 2011b.
URL http://www.satnac.org.za/proceedings/2011/papers/Work_In_Progress/Internet_Services_and_Applications/237.pdf

Egan, S. and Irwin, B. *An evaluation of lightweight classification methods for identifying malicious URLs.* In *Proceedings of Information Security South Africa (ISSA)*, ISSA '11, pages 1-6. August 2011a. ISBN 978-1-4577-1481-8. doi: 10.1109/ISSA.2011.6027532.
URL http://dx.doi.org/10.1109/ISSA.2011.6027532

Egan, S. and Irwin, B. *Normandy: A framework for implementing high speed lexical classification of malicious URLs.* In *Proceedings of the 15th Annual South African Telecommunication Networks and Application Conference (SATNAC)*, SATNAC '12. September 2012b.
URL http://www.satnac.org.za/proceedings/2012/papers/WIP_4.Internet_Services_End_User_Applications/175.pdf

Egan, S. and Irwin, B. *An analysis and implementation of methods for high speed lexical classification of malicious URLs.* In *Proceedings of Information Security South Africa (ISSA)*, ISSA '12. August 2012a.
URLhttp://icsa.cs.up.ac.za/issa/2012/Proceedings/Research/58_ResearchInProgress.pdf

APPENDIX

E

SOURCE CODE

All of the source code written for this research is freely available at GitHub from https://github.com/VanguardZA/

The following repositories are available for download:

- Classifier Generation Service (CGS)
- Classifier Distribution Service (CDS)
- Net Defence
- Training Data

The CGS requires a Microsoft Windows installation running Microsoft .NET 3.5. Additionally, Microsoft C# is required to compile the application. The CDS requires any OS that is capable of running Python with the *webpy*[1] library installed. Finally, Net Defence has no requirements and is written to work with the Google Chrome browser[2].

[1] http://webpy.org

[2] This will not work on mobile installations as extensions are disabled on the mobile version of Chrome.