Unknown Malcode Detection and the Imbalance Problem

Robert Moskovitch[1], Dima Stopel[1], Clint Feher[1], Nir Nissim[1], Nathalie Japkowicz[2], Yuval Elovici[1]

[1] Deutsche Telekom Laboratories at Ben Gurion University,
Department of Information Systems Engineering,
Ben Gurion UniversityBe'er Sheva, 84105 Israel
{robertmo, stopel, clint, nirni, elovici}@bgu.ac.il
[2] School of Information Technology and Engineering,
University of Ottawa,
Ottawa, Ontario K1N 6N5, Canada
nat@site.uottawa.ca

Abstract. The recent growth in network usage has motivated the creation of new malicious code for various purposes. Today's signature-based antiviruses are very accurate for known malicious code, but can not detect new malicious code. Recently, classification algorithms were used successfully for the detection of unknown malicious code. But, these studies involved a test collection with a limited size and the same malicious: benign file ratio in both the training and test sets, a situation which does not reflect real-life conditions. We present a methodology for the detection of unknown malicious code, which examines concepts from text categorization, based on n-grams extraction from the binary code and feature selection. We performed an extensive evaluation, consisting of a test collection of more than 30,000 files, in which we investigated the class imbalance problem. In real-life scenarios, the malicious file content is expected to be low, about 10% of the total files. For practical purposes, it is unclear as to what the corresponding percentage in the training set should be. Our results indicate that greater than 95% accuracy can be achieved through the use of a training set that has a malicious file content of less than 33.3%.

Keywords: Unknown Malicious Code Detection, Machine Learning, Classification, Imbalance Problem.

Contact Person:
Robert Moskovitch
Deutsche Telekom Laboratories at Ben Gurion University, Ben Gurion University
Be'er Sheva, 84105 Israel
Email: robertmo@bgu.ac.il

1 Introduction

The term *malicious code* (malcode) commonly refers to pieces of code, not necessarily executable files, which are intended to harm, generally or in particular, the specific owner of the host. Malcodes are classified, mainly based on their transport mechanism, into four main categories: *worms*, *viruses*, *Trojans* and new group that is becoming more common, which is comprised of *remote access Trojans* and *backdoors*. The recent growth in high-speed internet connections and in internet network services has led to an increase in the creation of new malicious codes for various purposes, based on economic, political, criminal or terrorist motives (among others). Some of these codes have been used to gather information, such as passwords and credit card numbers, as well as behavior monitoring.

Current anti-virus technology is primarily based on two approaches: *signature-based* methods, which rely on the identification of unique strings in the binary code; while being very precise, it is useless against unknown malicious code [1]. Moreover, these can be overcome by a variant after checking it with a black box anti-virus check [2] The second approach involves *heuristic-based* methods, which are based on rules defined by experts, which define a malicious behavior, or a benign behavior, in order to enable the detection of unknown malcodes [3]. Other proposed methods include *behavior blockers*, which attempt to detect sequences of events in operating systems, and *integrity checkers*, which periodically check for changes in files and disks. However, besides the fact that these methods can be bypassed by viruses, their main

drawback is that, by definition, they can only detect the presence of a malcode after it has been executed.

Therefore, generalizing the detection methods to be able to detect unknown malcodes is crucial. Recently, classification algorithms were employed to automate and extend the idea of heuristic-based methods. As we will describe in more detail shortly, the binary code of a file is represented by n-grams and *classifiers* are applied to learn patterns in the code and classify large amounts of data. A classifier is a rule set which is learnt from a given *training-set*, including examples of classes, both malicious and benign files in our case. Recent studies, which we survey in the next section [4,5,6,7,8], have shown that this is a very successful strategy. However, these studies present evaluations based on test collections, having similar proportion of malicious versus benign files in the test collections (50% of malicious files). This proportion has two potential drawbacks. These proportions do not reflect real life situation, in which malicious code is commonly significantly less than 50% and additionally these studies, as will be shown later, might report optimistic results. Recent survey[1] made by McAfee indicates that about 4% of search results from the major search engines on the web contain malicious code. Additionally, it was found that above 15% of the files in the KaZaA network contained malicious code[2]. Thus, we assume that the percentage of malicious files in real life is about or less than 10%, but we also consider other possible percentages.

[1] McAfee Study Finds 4 Percent of Search Results Malicious,By Frederick Lane,June 4, 2007 [http://www.newsfactor.com/story.xhtml?story_id=010000CEUEQO]

[2] S. Shin, J. Jung, H. Balakrishnan, Malware Prevalence in the KaZaA File-Sharing Network, Internet Measurement Conference (IMC), Brazil, October 2006.

In this study, we present a methodology for *malcode categorization* based on concepts from *text categorization*. We present an extensive and rigorous evaluation of many factors in the methodology, based on eight types of classifiers. The evaluation is based on a test collection 10 times larger than any previously reported collection, containing more than 30,000 files. We introduce the class imbalance problem, which refers to domains in which the proportions of each class instances is not equal, in the context of our task, in which we evaluate the classifiers for five levels of malcode content (percentages) in the training-set and 17 (percentages) levels of malcode content in the test-set. We start with a survey of previous relevant studies. We describe the methods we used, including: concepts from *text categorization*, data preparation, and classifiers. We present our results and finally discuss them.

2. Background

2.1 Detecting Unknown Malcode via Machine Learning

Over the past five years, several studies have investigated the direction of detecting unknown malcode based on its binary code. [4] were the first to introduce the idea of applying machine learning (ML) methods for the detection of different malcodes based on their respective binary codes. They used three different feature extraction (FE) approaches: *program header, string features* and *byte sequence features*, in which they applied four classifiers: a *signature-based method* (anti-virus), *Ripper* – a rule-based learner, *Naïve Bayes* and *Multi-Naïve Bayes*. This study found that all of the ML methods were more accurate than the signature-based algorithm. The ML methods were more than twice as accurate when the out-performing method was

Naïve Bayes, using strings, or Multi-Naïve Bayes using byte sequences. [5] introduced a framework that used the common n-gram (CNG) method and the k nearest neighbor (KNN) classifier for the detection of malcodes. For each class, malicious and benign, a representative profile was constructed and assigned a new executable file. This executable file was compared with the profiles and matched to the most similar. Two different data sets were used: the *I-worm collection*, which consisted of 292 Windows internet worms and the *win32 collection*, which consisted of 493 Windows viruses. The best results were achieved by using 3-6 n-grams and a profile of 500-5000 features. [6] presented a collection that included 1971 benign and 1651 malicious executables files. N-grams were extracted and 500 features were selected using the *information gain* measure [8]. The vector of n-gram features was binary, presenting the presence or absence of a feature in the file and ignoring the frequency of feature appearances (in the file). In their experiment, they trained several classifiers: IBK (KNN), a similarity based classifier called TFIDF classifier, Naïve Bayes, SVM (SMO) and Decision tree (J48). The last three of these were also boosted. Two main experiments were conducted on two different data sets, a small collection and a large collection. The small collection included 476 malicious and 561 benign executables and the larger collection included 1651 malicious and 1971 benign executables. In both experiments, the four best-performing classifiers were Boosted J48, SVM, boosted SVM and IBK. Boosted J48 out-performed the others. The authors indicated that the results of their n-gram study were better than those presented by [4]. Recently, [6] reported an extension of their work, in which they classified malcodes into families (classes) based on the functions in their respective payloads. In the categorization task of multiple classifications, the best results were

achieved for the classes' *mass mailer*, *backdoor* and *virus* (no benign classes). In attempts to estimate the ability to detect malicious codes based on their issue dates, these techniques were trained on files issued before July 2003, and then tested on 291 files issued from that point in time through August 2004. The results were, as expected, lower than those of previous experiments. Those results indicate the importance of maintaining the training set by acquisition of new executables, in order to cope with unknown new executables. [8] presented a hierarchical feature selection approach which enables the selection of n-gram features that appear at rates above a specified threshold in a specific virus family, as well as in more than a minimal amount of virus classes (families). They applied several classifiers: ID3, J48 Naïve Bayes, SVM- and SMO to the data set used by [4] and obtained results that were better than those obtained through traditional feature selection, as presented in [4], which mainly focused on 5-grams. Additionally, [9] presented to use the frequency of the n-grams in the files to select them as alternative to information gain based selection criterion.

2.2 The Imbalance Problem

In machine learning the data is, often, presented as a list of labeled examples, in which an example is described by a vector of features and an additional special feature which is the class (e.g., malicious/benign). Thus, the data is actually a matrix, in which each example is a row having n features and a class, which are the columns. Often there are equal numbers of examples for each class. These general proportions are important since most of the classifiers are probabilistic and thus they induce the general proportions of the classes in the dataset. For evaluation purposes the dataset is

divided into two datasets: training set, which is used to train a classifier and which actually represents the world to the learner, and a test set which represents the real life scenario. Whenever there is a significant difference in the proportions of the numbers of examples for the classes, which happens often as a result of less available examples of a specific class, it might affect the accuracy of the classifier. This case is called the class imbalance problem.

The class imbalance problem was first noticed by the machine learning research community a little over a decade ago (e.g.,[10,11,12]). As just discussed, it typically occurs when there are significantly more instances from one class relative to other classes. In such cases most standard classifiers tend to misclassify the instances of the low represented classes. In certain cases of extreme imbalances, the classifier may go as far as to classify all the data with the label of the large class, thus, completely ignoring the data from the small class. More and more researchers realized that the performance of their classifiers may be suboptimal due to the fact that the datasets are not balanced. This problem is even more important in fields where the natural datasets are highly imbalanced in the first place [13], like the problem we describe.

Over the years, the machine learning community has addressed the issue of class imbalances following two general strategies. The first one, which is classifier-independent, consists of balancing the original data set, using different kinds of undersampling or oversampling approaches. In particular, researchers have experimented with random sampling, where instances from the training set are either duplicated or eliminated at random (e.g., [14]); directed sampling, where specific instances are targeted for undersampling or oversampling with the idea of strengthening the most relevant data and weakening the least relevant ones (e.g., [14],

[10]) ; and artificial sampling, where the smaller class is oversampled with artificially generated data designed to augment the minority class without creating the risk of overfitting [15]. The second way involves modifying the classifiers in order to adapt them to the data sets. In particular, these approaches look for ways of incorporating misclassification costs into the classification process and assigning higher misclassification costs to the minority class so as to compensate for its small size. This was done for a variety of different classifiers such as Neural networks [16] Random Forests [17], and SVM [18].

However, in our problem unlike in other problems, the data is not imbalanced in the training set, but rather in real life conditions, which we reflect by the test set. Thus, we don't need an algorithm to overcome the imbalanced data, but rather to understand the optimal construction of a training set to achieve the best performance in real life conditions. Our research is, thus, more in line with the work of [19], which considers the question of what proportion of examples of each class is most appropriate for learning if a only a limited number of training instances can be used altogether. Their work, considers the case of decision tree induction on twenty-six different data sets. We, on the other hand, focus on the single problem of interest here—malcode detection—but consider eight different classifiers.

Another way in which our work relates to the research emanating from the class imbalance community concerns the choice of an evaluation metric, as discussed in Section 4.2.

3 Methods

3.1 Text Categorization

For the detection and acquisition of unknown malicious code, we suggest the use of well-studied concepts from *information retrieval* (IR) and more specific *text categorization*. In our problem, binary files (executables) are parsed and n-gram terms are extracted. Each n-gram term in our problem is analogous to a word in the textual domain. We hereby describe IR concepts which we used in this study.

Salton presented the *vector space model* [20] to represent a textual file as a *bag of words*. After parsing the text and extracting the words, a vocabulary, of the entire collection of words is constructed. Each of these words may appear zero to multiple times in a document and at least in a single document. The *vocabulary* is the vector of terms which was extracted from the entire set of documents. Each term in the vocabulary can be described by its frequency in the entire collection, often called *document frequency*, which is later used for the term weighting. For each document a vector of terms in the size of the vocabulary is created, such that each index in the vector represents the *term frequency (TF)* in the document. Formula 1 shows the definition of a normalized TF, in which the term frequency is divided by the maximal appearing term in the document with values in the range of [0-1]. An extended representation is the *TF Inverse Document Frequency (TFIDF)*, which combines the frequency of a term in the document (TF) and its frequency in the documents collection, denoted by *Document Frequency* (DF), as shown in forumla 2, in which the term's (normalized) *TF* value is multiplied by the $IDF = log\,(N/DF)$, where N is

the number of documents in the entire file collection and *DF* is the number of files in which it appears.

$$TF = \frac{term\ frequency}{\max(term\ frequency in\ document)} \quad (1)$$

$$TFIDF = TF * \log(\frac{N}{DF}) \quad (2)$$

The TF representation is actually the representation which was used in previous papers in our domain of malicious code classification. However, in the textual domain it was shown that the *tfidf* is a richer and more successful representation for terms for retrieval and categorization purposes [20], thus, we expected that using the *tfidf* weighting will lead to better performance then the *tf*. In the textual domain often the *stop words*, which are words that appear often, such as *the*, *to*, etc, are removed. These terms can be characterized by having high DF value.

3.2 Data Set Creation

We created a data set of malicious and benign executables for the Windows operating system, as this is the system most commonly used and most commonly attacked. To the best of our knowledge and according to a search of the literature in this field, this collection is the largest one ever assembled and used for research. We acquired the malicious files from the VX Heaven website[3]. The dataset contains 7688 malicious files. To identify the files, we used the Kaspersky[4] anti-virus and the Windows version of the Unix 'file' command for file type identification. The malicious files included: The files in the benign set, including executable and DLL (Dynamic Linked

[3] http://vx.netlux.org

Library) files, were gathered from machines running Windows XP operating system on our campus. More specifically the set included applications, such as messenger, visual studio executables, anti-virus applications, zipping applications, as well as windows inner and driver dlls, service packs installers and other installation files and executables related to varying applications which were installed on the machines. The benign set contained 22,735 files. The Kaspersky anti-virus program was used to verify that these files do not contain any malicious code.

3.3 Data Preparation and Feature Selection

N-grams extraction

We parsed the files using several *n-gram* sequence lengths, denoted by *n*. Vocabularies of 16,777,216, 1,084,793,035, 1,575,804,954 and 1,936,342,220, for 3-gram, 4-gram, 5-gram and 6-gram respectively were extracted. Later TF and TFIDF representations were calculated for each n-gram in each file.

In machine learning applications, the large number of features (many of which do not contribute to the accuracy and may even decrease it) in many domains presents a significant problem. Moreover, in our problem, the reduction of the number of features is crucial, but must be performed while maintaining a high level of accuracy. This is due to the fact that, as shown earlier, the vocabulary size may exceed billions of features, far more than can be processed by any feature selection tool within a reasonable period of time. Additionally, it is important to identify those terms that appear in most of the files, in order to avoid vectors that contain many zeros. Thus, we first extracted the features having the top 5,500 *document frequency* (formula 2)

[4] http://www.kaspersky.com

values as a preliminary aggressive feature selection, on which later three feature selection methods were applied. In order to check whether the *stop words* phenomena happens in our problem domain we selected the 5,500 top features and 1,000-6,500 top features from the entire list ranked by the DF. The features selected from the top 1,000-6,500, in which the top 1000 features were removed, represented the idea of stop-words in the textual domain, which we examined their potential effect here.

Feature Selection

We used a *filters* approach, in which a measure is used to quantify the correlation of each feature to the class (malicious or benign) and estimate its expected contribution to the classification task. After applying the filter each feature gets a rank which quantifies its expected contribution in the classification task, from which later the features with the top ranks are used. Note that the filter is applied on the dataset and the measure is independent of any classification algorithm, which enables to compare the performances of the different classification algorithms on the same subset of features. We used three feature selection measures. As a baseline, we used the *document frequency* measure *DF* (the amount of files in which the term appeared in), *Gain Ratio (GR)* [8] and *Fisher Score (FS)* [21].

3.3.1 Gain Ratio

Gain Ratio was originally presented by Quinlan in the context of *Decision Trees* [8], which was designed to overcome a bias in the *Information Gain (IG)* measure, and which measures the expected reduction of entropy caused by partitioning the examples according to a chosen feature. Given entropy *E(S)* as a measure of the impurity in a collection of items, it is possible to quantify the effectiveness of a

feature in classifying the training data. Formula 4 presents the formula of the entropy of a set of items *S*, based on *C* subsets of *S* (for example, classes of the items), presented by S_c. *Information Gain* measures the expected reduction of entropy caused by portioning the examples according to attribute *A*, in which *V* is the set of possible values of *A*, as shown in Formula 3. These formulas refer to discrete values; however, it is possible to extend them to continuous values attribute.

$$IG(S,A) = E(S) - \sum_{v \in V(A)} \frac{|S_v|}{|S|} \cdot E(S_v) \quad (3)$$

$$E(S) = \sum_{c \in C} -\frac{|S_c|}{|S|} \cdot \log_2 \frac{|S_c|}{|S|} \quad . \quad (4)$$

The *IG* measure favors features having a high variety of values over those with only a few. *GR* overcomes this problem by considering how the feature splits the data (Formulas 5 and 6). S_i are *d* subsets of examples resulting from portioning *S* by the *d-valued* feature *A*.

$$GR(S,A) = \frac{IG(S,A)}{SI(S,A)} \quad (5)$$

$$SI(S,A) = -\sum_{i=1}^{d} \frac{|S_i|}{|S|} \cdot \log_2 \frac{|S_i|}{|S|} \quad (6)$$

3.3.2 Fisher Score

The Fisher score ranking technique calculates the difference, described in terms of mean and standard deviation, between the positive and negative examples relative to a certain feature. Formula 7 defines the Fisher score, in which R_i is the rank of feature *i*, describing the proportion of the substitution of the mean of the feature *i* values in the positive examples (*p*) and the negative examples (*n*), and the sum of the standard deviation. The bigger the R_i, the bigger the difference between the values of positive

and negative examples relative to feature *i;* thus, this feature is more *important* for separating the positive and negative examples. This technique is described in details in [21].

$$R_i = \frac{|\mu_{i,p} - \mu_{i,n}|}{\sigma_{i,p} + \sigma_{i,n}} \quad (7)$$

Based on each feature selection measure we selected the top 50, 100, 200 and 300 features.

3.4 Classification Algorithms

We employed four commonly used classification algorithms: *Artificial Neural Networks* (ANN), *Decision Trees* (DT), *Naïve Bayes* (NB), and their boosted versions, BDT and BNB respectively, as well as Support Vector Machines (SVM) with three kernel functions. We briefly describe the classification algorithms we used in this study.

3.4.1 Artificial Neural Networks

An Artificial Neural Network (ANN) [22] is an information processing paradigm inspired by the way biological nervous systems, such as the brain, process information. The key element is the structure of the information processing system, which is a network composed of a large number of highly interconnected neurons working together in order to approximate a specific function, as shown in figure 1. An ANN is configured for a specific application, such as pattern recognition or data classification, through a *learning process* during which the individual weights of different neuron inputs are updated by a *training algorithm*, such as back-propagation. The weights are updated according to the examples the network receives, which

reduces the *error function*. Formula 8 presents the output computation of a two-layered ANN, where x is the input vector, v_i is a weight in the output neuron, g is the activation function, w_{ij} is the weight of a hidden neuron and $b_{i,o}$ is a bias. All the ANN manipulations were performed within the MATLAB(r) environment using the Neural Network Toolbox.

$$f(x) = g\left[\sum_i v_i g\left(\sum_j w_{ij} x_j + b_i\right) + b_o\right] \tag{8}$$

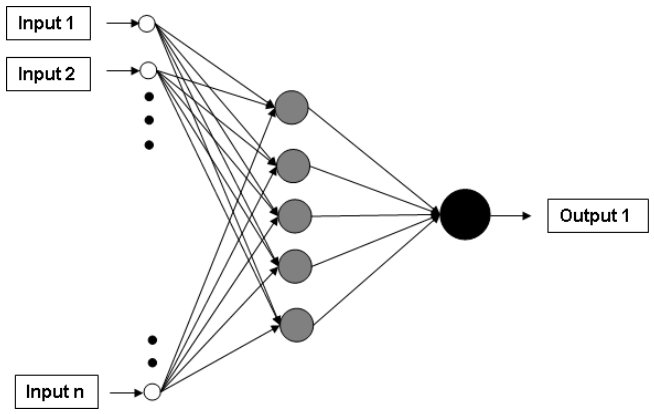

Figure 1. A typical architecture of a feed forward ANN, having five hidden neurons and a single output neuron. The number of the hidden neurons and the number of the output neurons may vary according to the analyzed data.

3.4.2 Decision Trees

Decision tree learners [24] are a well-established family of learning algorithms. Classifiers are represented as trees whose internal nodes are tests of individual features and whose leaves are classification decisions (classes). Typically, a greedy heuristic search method is used to find a small decision tree, which is induced from

the data set by splitting the variables based on the *expected information gain*. This method correctly classifies the training data. Modern implementations include pruning, which avoids the problem of over-fitting. In this study, we used J48, the Weka [24] version of the C4.5 algorithm [23]. An important characteristic of decision trees is the explicit form of their knowledge, which can be easily represented as rules.

3.4.3 Naïve Bayes

The Naïve Bayes classifier is based on the *Bayes theorem*, which in the context of classification states that the posterior probability of a class is proportional to its prior probability, as well as to the conditional likelihood of the features, given this class. If no independent assumptions are made, a Bayesian algorithm must estimate conditional probabilities for an exponential number of feature combinations. Naive Bayes simplifies this process by assuming that features are *conditionally independent*, given the class, and requires that only a linear number of parameters be estimated. The prior probability of each class and the probability of each feature, given each class is easily estimated from the training data and used to determine the posterior probability of each class, given a set of features. Empirically, Naive Bayes has been shown to accurately classify data across a variety of problem domains [25].

3.4.4 Adaboost.M1 (BDT and BNB)

Boosting is a method for combining multiple classifiers. Adaboost was introduced by [26] and among its many variants is the Adaboost.M1 that is implemented in Weka. Given a set of examples and a base classifier, it generates a set of hypotheses combined by weighted majority voting. Learning is achieved in iterations. In each

iteration a new set of instances is selected by favoring misclassified instances of previous iterations. This is done using an iteratively updated distribution that includes a probability for each instance to be selected in the next iteration. We used the Adaboost.M1 to boost J48 decision trees and Naïve Bayes.

3.4.5 Support Vector Machines

SVM is a binary classifier, which finds a linear hyperplane that separates the given examples of two classes known to handle large amounts of features. Given a training set of labeled examples in a vector format, the SVM attempts to specify a linear hyperplane that has the maximal margin, defined by the maximal (perpendicular) distance between the examples of the two classes. The examples lying closest to the hyperplane are known as the supporting vectors. The normal vector of the hyperplane (denoted as w in forumla 9, in which n is the number of the training example) is a linear combination of the supporting vectors multiplied by LaGrange multipliers (alphas). Figure 2 illustrates a two dimensional space, in which the examples (vectors) are located according to their features values in two groups based on their labels (classes +1 and -1) and the hyperplane which is derived to separate them linearly according to their label.

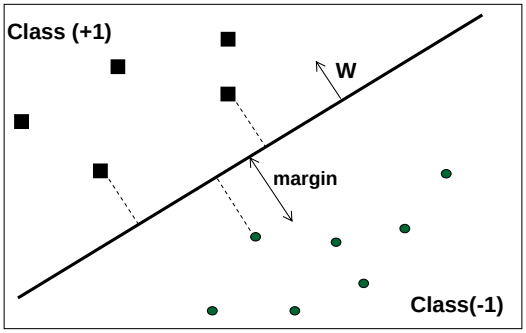

Figure 2. An SVM that separates the training set into two classes, having maximal margin in a two dimensional space.

Often the data set cannot be linearly separated, so a kernel function *K* is used. The SVM actually projects the examples into a higher dimensional space to create a linear separation of the examples. Note that when the kernel function satisfies Mercer's condition, as was explained by Burges [27], For the general case, the SVM classifier will be in the form shown in formula 9, while n is the number of examples in training set, and w is normal of the hyperplane. We examined three commonly used kernels: *Linear* (SVM-LIN), *Polynomial* (SVM-POL) and *RBF* (SVM-RBF). We used the Lib-SVM implementation[5].

$$f(x) = sign(w \cdot \Phi(x)) = sign\left(\sum_{1}^{n} \alpha_i y_i K(x_i\, x)\right) \qquad (9)$$

As it derived from the theoretic basis of SVM and was also empirically shown by Joachim [28], one should select the appropriate kernel function with the appropriate configurations of the parameters, while, usually, the more sophisticated the kernel is, the better the performance and the results are. The ranks of the sophistication are: linear, Polynomial, RBF so that linear is the simplest one, and RBF is more sophisticated than the two others. Note that with the sophistication of the kernel, the training time and the computational resources requirements are larger.

Figure 3 (which were produced by the applet which is available from the LIBSVM website [29]) illustrate the use of the kernels given the same training-set in an SVM with Linear and Polynomial kernels. While the SVM with Linear kernel (right-side) is not sophisticated enough to determine a hyperplane that separates the training-set

optimally (thus, each class vectors are located separately), the SVM with the Polynomial kernel (left-side) has successfully determined such one:

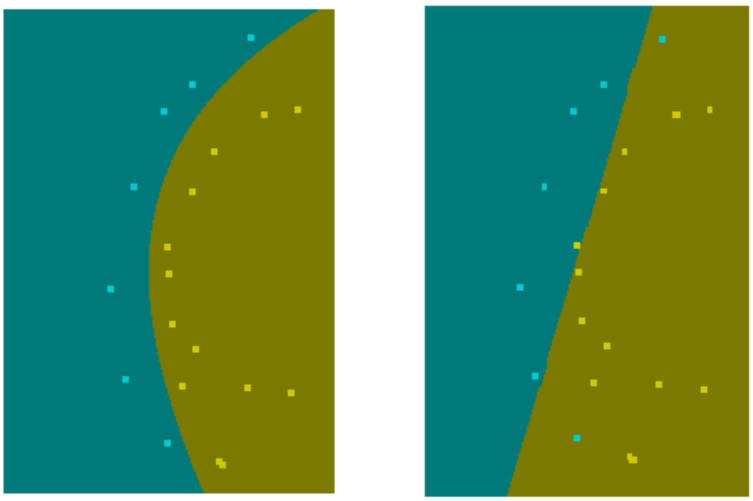

Figure 3: The Polynomial (Left) and Linear (Right) kernels applied to the same training set, The Polynomial has successfully separated the training-set hyperplane whereas the Linear hasn't.

Figure 4 illustrates the use of the RBF and Polynomial kernels in an SVM applied to the same complicated dataset. While the SVM with Polynomial kernel (left-side) is not sophisticated enough to determine hyperplane that separates the training-set, the SVM with the RBF kernel (right-side) has successfully determined such one.

[5] http://www.csie.ntu.edu.tw/~cjlin/libsvm

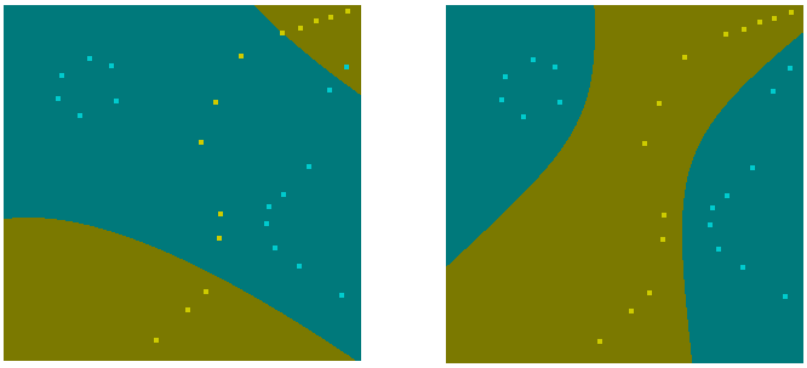

Figure 4: The Polynomial (Left) and RBF kernels (Right) are applied to the same training set. The RBF has successfully separated the data-set whereas the Polynomial hasn't, it can be observed that the RBF kernel is very sophisticated and powerful also towards complicated training-set.

4 Evaluation

4.1 Research Questions

We wanted to evaluate the proposed methodology for the detection of unknown malicious codes through two main experiments. The first experiment was designed to determine the best conditions, including four aspects:

1. Which *term representation* is better, *TF* or *TFIDF*?

2. Which n-gram is the best: *3*, *4*, *5* or *6*?

3. What is the better range for *global* feature selection, top *5500* or *1000 – 6500*?

4. Which *top-selection* is the best: *50*, *100*, *200* or *300* and which features selection: *DF*, *FS* and *GR*?

5. Which Malicious Code Percentage in the training set will be the best for any Malicious Code Percentage in the test set, and for real life conditions?

To answer the listed questions we first performed a wide set of experiments to identify the best term representation, n-gram, global feature selection and top selection and feature selection measure. Using the best settings we performed additional set of experiments focusing on the imbalance problem.

4.2 Evaluation Measures

For evaluation purposes, we wanted to measure the accuracy of the classification algorithms, as well as the false positive and true positive which are often very important, in order to be able to tune the classifier for the best needs. For that we used the common set of measures, which included the *True Positive Rate (TPR)* measure, which is the number of *positive* instances classified correctly, as shown in formula 10, *False Positive Rate (FPR)*, which is the number of *negative* instances misclassified (formula 10), and the *Total Accuracy*, which measures the number of absolutely correctly classified instances, either positive or negative, divided by the entire number of instances shown in formula 11.

$$TPR = \frac{|TP|}{|TP| + |FN|}; \qquad FP\,R = \frac{|FP|}{|FP| + |TN|} \tag{10}$$

$$Total\ Accuracy = \frac{|TP| + |TN|}{|TP| + |FP| + |TN| + |FN|} \tag{11}$$

Total Accuracy is the most intuitive evaluation measure and is very often used in Machine Learning. It simply returns the percentage of right choices made by the classifier. One thing it does not do, however, is indicate whether the classifier is more adept at classifying positive or negative examples. This is often important

information, like in our problem where we are interested in finding out what proportion of the malcodes present in the data are actually detected by the classifier (TPR) and what proportion of the virus-free data is wrongly classified as virus data (FPR). Even if a classifier is good at detecting viruses, it might be discarded from consideration because of the large number of false alarms (high FPR) it generates. Information of this kind could not be obtained from the Total Accuracy alone, and this is why analyses of TPR and FPR results were also included.

In fact, for the imbalance analysis, *Total Accuracy* is not only misinformed, but it is, often, simply an inappropriate measure of performance. Indeed, in such circumstances, a trivial classifier that predicts every case as the majority class could achieve very high accuracy in extremely skewed domains. Several proposals have been made to address this issue including the decomposition of accuracy into its basic components (TPR and FPR) [14], the use of ROC Analysis [30] or the G-Mean [31]. In this paper, we selected to decompose accuracy into its basic components along with the use of the G-mean. This approach is conceptually simpler than using ROC Analysis and sheds sufficient light on our results.

The g-means measure (formula 13) which is often used in imbalance datasets evaluation studies, is based on the sensitivity and specificity measures (formulas 12).

$$Sensitivity = \frac{|TP|}{|TP|+|FN|};\ Specificity = \frac{|TN|}{|TN|+|FP|} \quad (12)$$

$$G-means = \sqrt{Sensitivity * Specificity} \quad (13)$$

Sensitivity is exactly the same thing as the TPR introduced earlier. The difference in name simply stems from the fact that various fields of study came up with the same

measures of success, but named them differently. The term “Sensitivity” was coined in the medical domain while TPR is the name used in the Machine Learning community. Specificity is the opposite of FPR. It measures the proportion of negative data rightly labeled as negative. In our problem, this corresponds to the proportion of uninfected data rightly labeled as such. Sensitivity and Specificity, thus, give us the same information as TPR and FPR. The G-mean, however, combines this information in a way different from the way in which Total Accuracy does. By multiplying the components together, indeed, the G-mean sheds light on whether the classifier is lacking on one or the other aspect of classification (detection of positive examples and recognition of a negative example). This is information that is not provided by Total Accuracy and which is critical, as previously discussed, in the case of class imbalances. This is why G-mean results were also provided in the class imbalance study.

5 Experiments and Results

5.1 Experiment 1

To answer the four questions, presented earlier, we designed a wide and comprehensive set of evaluation runs, including all the combinations of the optional settings for each of the aspects, amounting in 1536 runs in a 5-fold cross validation format for all eight classifiers. Note that the files in the test-set were not in the training-set presenting unknown files to the classifier.

5.1.2 Global Feature Selection versus n-grams

First we wanted to find the best terms representation, tf vs tfidf, and the global feature selection. Figure 5 presents the mean accuracy of the combinations of the term representations and n-grams. While the mean accuracies are quite similar, the top 5,500 features performed better, as did the TF representation and the 5-gram. Having the TF out-performing has meaningful computational advantages; we will elaborate on these advantages in the Discussion. Additionally, the 5-grams outperformed the other n-gram sizes.

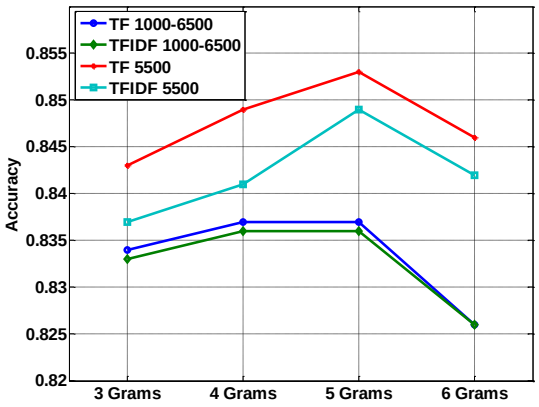

Figure 5. While their mean accuracies were quite similar, the top 5,500 features out-performed the top 1000-6500, TF out-performed TFIDF and the 5-gram out-performed the other n-gram sizes.

5.1.3 Feature Selections and Top Selections

To identify the best feature selection method and the top amount of features we calculated the mean accuracy of each option, as shown in Figure 6. Generally, the Fisher score was the superior method, starting with high accuracy, even with 50 features. Unlike the other methods, in which the 300 features out-performed, the DF's

accuracy decreased after the selection of more than 200 features, while the GR accuracy significantly increased as more features were added.

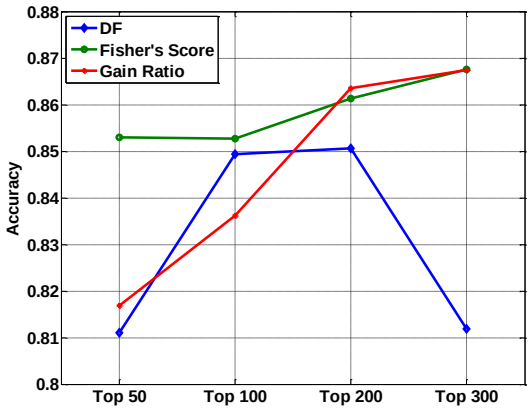

Figure 6. The Fisher score was very accurate when used with just 50 features, and maintained this high performance level as the number of features increased. When more than 200 features were added, the accuracy of GR increased significantly and the accuracy of DF decreased.

5.1.4 Classifiers

The results of each classifier under the best settings identified before for all the classifiers (section 5.1.3), including the top 300 Fisher score-selected 5-gram terms represented by TF from the top 5500 features are presented in Table 2. The BDT, DT and ANN outperform and demonstrated low false positive rates, while the SVM classifiers also performed very well. We suggest that the poor performance of the Naïve Bayes, may be explained by the independence assumption of the NB classifier.

Table 2. Classifiers performance: the BDT, DT and ANN out-performed, while maintaining low levels of false positives.

Classifier	Accuracy	FP	FN
ANN	**0.941**	0.033	0.134
DT	**0.943**	0.039	0.099
NB	0.697	0.382	0.069
BDT	**0.949**	0.040	0.110

BNB	0.697	0.382	0.069
SVM-lin	0.921	0.033	0.214
SVM-poly	0.852	0.014	0.544
SVM-rbf	0.939	0.029	0.154

5.2 Experiment 2 – The Imbalance Problem

In the second experiment, we present our main contribution in this study. In this experiment we investigated rigorously the imbalance problem in our domain and to actually answer the fifth research question. The fifth research question presents the question of what are the optimal proportions of the benign and malicious contents in the training set for varying levels of proportions in the test set, which reflect the situation in real life. This is a very useful investigation for practical purposes, since when applying this technique the proportions in the training set should be considered according to the expected proportions in the stream of the file, which was represented in this experiment by the test set.

We used the best configuration and the top 300 Fisher Score-selected 5-gram terms represented by TF from the top 5500 features. We created five levels of Malicious Files Percentage (MFP) in the training set (16.7, 33.4, 50, 66.7 and 83.4%), which represent the proportions which can be controlled when applying this technique. For example, when referring to 16.7%, we mean that 16.7% of the files in the training set were malicious and 83.4% were benign. The test set represents the 'real-life' situation, while the training set represents the set-up of the classifier, which is controlled. While we assume that a MFP above 30% (of the total files) is not a realistic proportion in the stream of real networks, but we used test set that included high percentages of malicious files in order to gain insights into the behavior of the classifiers in these situations. Our study examined 17 levels of MFP (5, 7.5, 10, 12.5, 15, 20, 30, 40, 50,

60, 70, 80, 85, 87.5, 90, 92.5 and 95%) in the test sets. Eventually, we ran all the product combinations of five proportions in the training sets and 17 test sets, for a total of 85 runs for each classifier. We created two sets of data sets in order to perform a 2-fold cross validation-like evaluation to make the results more significant.

We analyze all the results using four evaluation measures: the *True Positive Rate*, the *False Positive Rate*, the combination of them by the *Accuracy* and the *g-means* measure which is often used in imbalance problems.

5.2.1 Training-Set Malcode Percentage

In this analysis we evaluated the performance of each training-set MFP settings against the varying levels of MFP in the test-sets. Thus, for each MFP in the training set the mean values of the measures are presented. Figure 7 presents the mean accuracy (averaged over all the MFP levels in the test-sets) of each classifier for each MFP level in the training set.

All the classifiers, beside NB, demonstrated an increased FPR and TPR with the increase of the MFP in the training-set. According to the Accuracy and g-means measures the classifiers behaved similarly. ANN, BDT and DT demonstrated the highest accuracy, and relatively stable, performance across the different MFP levels, while BNB, NB and SVM-POL generally performed poorly. SVM-RBF and SVM-LIN performed well, but not consistently. They were both most accurate at the 50% level, while the accuracy of SVM-POL increased as more malicious examples were presented.

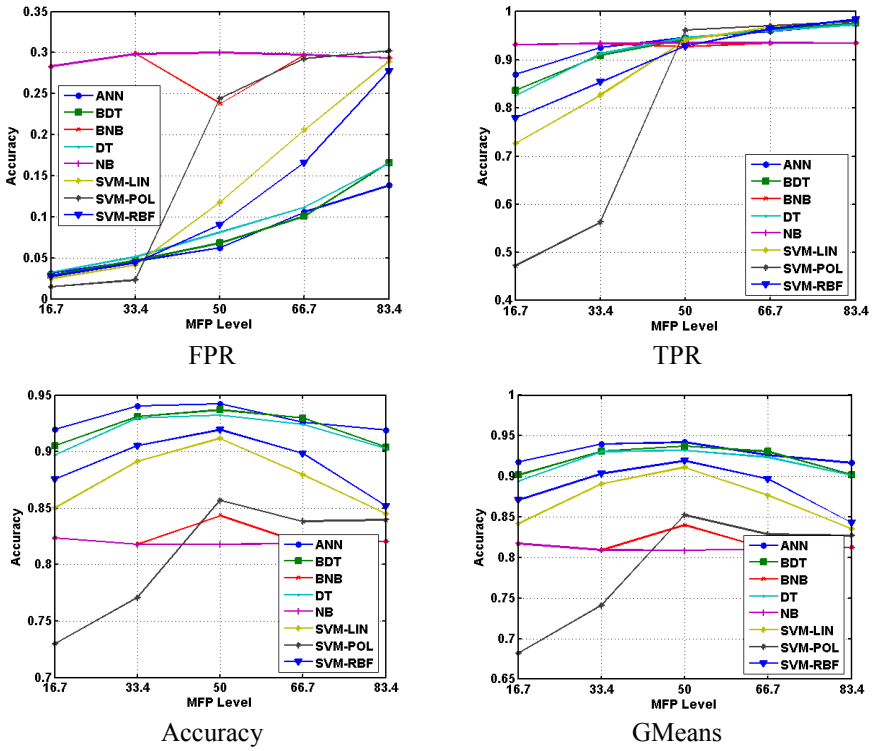

Figure 7. ANN, BNB and DT out-performed, with consistent accuracy, across the different malcode content levels.

5.2.2 10% Malcode Percentage in the Test Set

We consider the 10% MFP level in the test set as a realistic scenario, which reflects real life conditions, in which there are 10% of malicious contents.

Figure 8 presents the mean accuracy in the 2-fold cross validation of each classifier for each MFP level in the training set, with a fixed level of 10% MFP in the test set. Thus, each point in the curve is the average of all the runs with the varying MFPs in the training sets. Accuracy levels above 95% were achieved when the training set had a MFP of 16.7%, while a rapid decrease in accuracy was observed when the MFP in the training set was increased. Thus, the optimal proportion in a training set for

practical purposes should be in the range of 10% to 40% malicious files. This is in line with [19] who concluded, from their study, that when accuracy is used, the optimal class distribution in the training set tends to be near the natural class distribution.

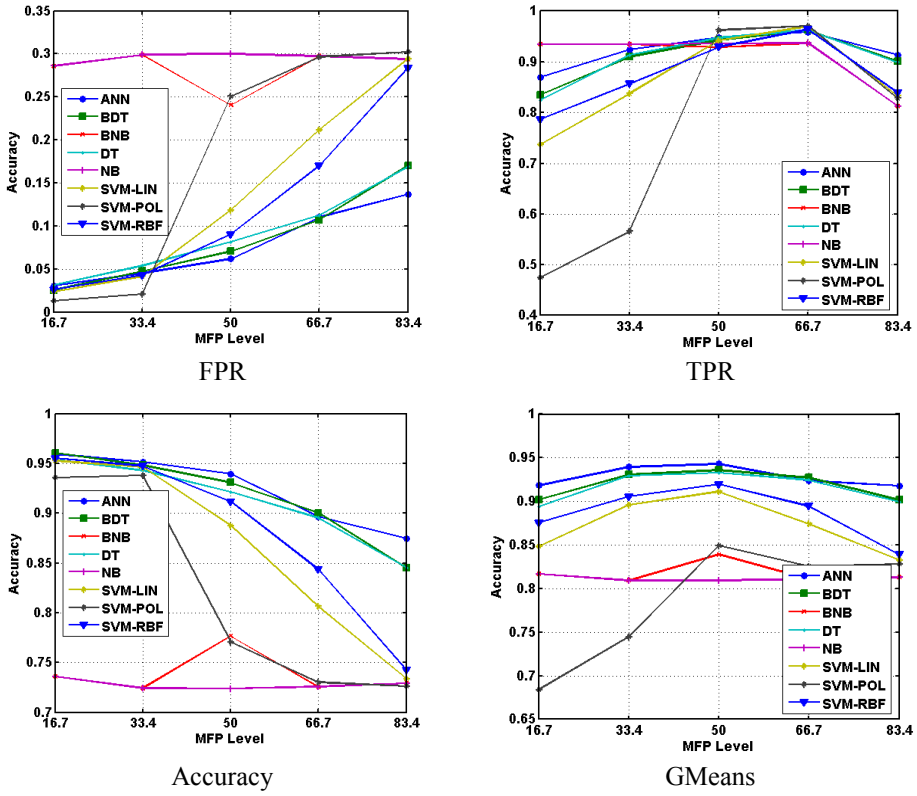

Figure 8. Greater than 95% accuracy was achieved for the scenario involving a low level (16.7%) of malicious file content in the training set.

5.2.3 Relations among MFPs in Training and Test Sets

In subsections 5.2.1 and 5.2.2 we presented the mean results of varying MFP levels of the test set (5.2.1) and for 10% MFP in the test (5.2.2) for the each MFP in the training set. Here we present specifically the accuracy for each experiment of a MFP

level in the training vs a MFP level in the test. Thus, in the following figures a three-dimension results presentation, in which the horizontal axes are the training set MFP and the test set MFP and the vertical axis is the accuracy, is given for each classifier. This presentation gives a more detailed guideline for setting the MFP in the training set for each expected MFP in the stream, reflected by the test set.

Most of the classifiers behaved optimally when the MFP levels in the training-set and test-set were similar, except for the NB and BDT, which showed low performance levels earlier. This indicates that when configuring a classifier for a real-life application, the MFP in the training-set has to be set accordingly.

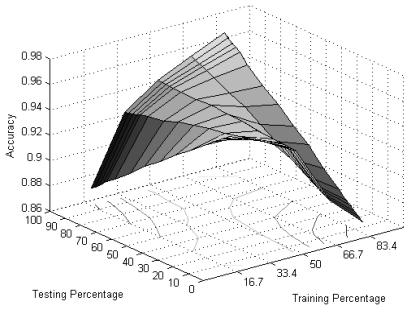

Figure 9 – ANN

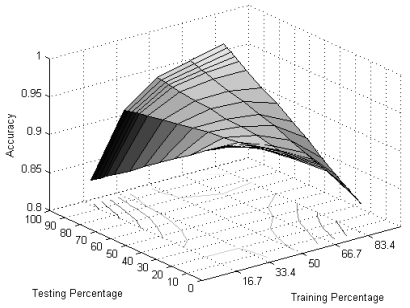

Figure 10 – DT