VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OPTIMALIZACE HEURISTICKÉ ANALÝZY SPUSTITELNÝCH SOUBORŮ

OPTIMIZATION OF HEURISTIC ANALYSIS OF EXECUTABLE FILES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MICHAL WIGLASZ
AUTHOR

VEDOUCÍ PRÁCE Prof. Ing. TOMÁŠ HRUŠKA, CSc.
SUPERVISOR

BRNO 2012

Abstrakt

Cílem této práce je implementace klasifikátoru, který by mohl být použit v prostředí společnosti AVG Technologies k detekci škodlivého softwaru na základě jeho chování ve virtuálním prostředí namísto současné metody detekce, kdy mají jednotlivé sledované akce ručně přiděleny váhy a jejich součet rozhoduje o tom, zda je neznámý vzorek považován za škodlivý či neškodný. Tato společnost také poskytla data použitá pro učení a testování.

V práci jsou představeny základní pojmy z klasifikace, blíže jsou představeny některé klasifikační metody a způsoby vyhodnocení úspěšnosti modelu a srovnání různých modelů a také učení s ohledem na ceny chybné klasifikace. Konkrétně je představena bayesovská klasifikace, rozhodovací stromy, neuronové sítě, Support Vector Machines a algoritmy AdaBoost a MetaCost.

Popsané klasifikační algoritmy jsou porovnány na poskytnuté databázi. Ukázalo se, že ačkoliv některé klasifikátory dosahují velmi dobrých výsledků (94% úspěšnost v případě neuronových sítí), na druhou stranu klasifikují příliš mnoho neškodných vzorků jako škodlivých – více než 9 %, přičemž společnost AVG je ochotna tolerovat maximálně 1 %. Po nastavení cen falešně pozitivní klasifikace, tak aby byla tato hodnota dodržena se ukázalo, že úspěšnost všech klasifikátorů se pohybuje okolo 62 % a větší rozdíl je pouze v čase potřebném k trénování. Na základě těchto experimentů byl k implementaci zvolen naivní bayesovský klasifikátor, který se ukázal jako nejrychlejší.

Zvolený algoritmus byl implementován v jazyce Python a jeho standardní knihovna. Jako vstupní a výstupní formát byl použit dialekt CSV. Vytvořená aplikace pokrývá všechny potřebné kroky – extrakci atributů z logů vytvořených během analýzy souborů ve virtuálním prostředí, učení klasifikátoru a odhad jeho úspěšnosti a klasifikaci neznámých vzorků. Aplikace také umožňuje nastavení vah trénovacím vzorkům na základě jejich třídy, díky čemuž lze snadno model upravovat směrem k pozitivní či negativní klasifikaci.

Výsledky aplikace byly ověřeny na poskytnuté databázi. Ukázalo se, že při nastavení vah tak, že model dosahoval požadovaného maxima 1 % neškodných vzorků ohodnocených jako škodlivých, byla jeho úspěšnost podobná jako současně používaný způsob klasifikace. Na druhou stranu učení probíhá zcela automaticky a není třeba ručně určovat váhy jednotlivých sledovaných akcí. Také se ukázalo, že eliminací některých méně důležitých atributů lze zkrátit čas potřebný k učení zhruba o jednu polovinu. Vytvořená aplikace neposkytuje pro výběr atributů žádné nástroje, nicméně pro tento úkol lze použít některý z mnoha existujících specializovaných nástrojů.

Abstract

This thesis describes the implementation of a classification tool for detection of unknown malware based on their behaviour which could replace current solution, based on manually chosen attributes' scores and a threshold. The database used for training and testing was provided by AVG Technologies company, which specializes in antivirus and security systems. Five different classifiers were compared in order to find the best one for implementation: Naïve Bayes, a decision tree, RandomForrest, a neural net and a support vector machine. After series of experiments, the Naïve Bayes classifier was selected. The implemented application covers all necessary steps: attribute extraction, training, estimation of the performance and classification of unknown samples. Because the company is willing to tolerate false positive rate of only 1% or less, the accuracy of the implemented classifier is only 61.7%, which is less than 1% better than the currently used approach. However it provides automation of the learning process and allows quick re-training (in average around 12 seconds for 90 thousand training samples).

Klíčová slova

Klasifikace, dolování z dat, strojové učení, škodlivý software, antivirus, naivní Bayes, Python.

Keywords

Classification, data mining, machine learning, malicious software, anti-virus, Naïve Bayes, Python.

Citace

Michal Wiglasz: Optimization of Heuristic Analysis of Executable Files, bakalářská práce, Brno, FIT VUT v Brně, 2012

Optimization of Heuristic Analysis of Executable Files

Declaration

I declare that this thesis is my own work that has been created under the supervision of Prof. Ing. Tomáš Hruška, CSc., and in consultation with Pavel Krčma. All sources and literature that I have used during elaboration of the thesis are correctly cited with complete reference to the corresponding sources.

. . . .
Michal Wiglasz
July 30, 2012

Acknowledgements

I would like to thank Pavel Krčma from the AVG Technologies company for valuable feedback and Ing. Jan Koutník for his lead and supervision of my thesis during my studies in Switzerland. I would also like to thank all my friends and my family for endless support and encouragement.

Contents

Chapter 1

Introduction

Malicious software or malware is a term used for all kinds of unwanted software (viruses, Trojan horses, worms etc.) which pose security thread to computer users. The most important defence against these are virus scanners. These usually work with database of descriptions of known malware, in form of signatures. When a new type of malware appears, traditional scanner will not be able to recognize it until its signature database is updated.

To be able to detect also unknown malware, heuristic analysis was introduced. Instead of looking for signatures of specific malware instances, heuristic analysis tries to look for suspicious behaviour. During analysis, several features are extracted from examined file and then these features are processed by classifier, which yields a final decision whether the file seems to be benign or a new form of malware.

AVG Technologies[1], an anti-virus company, owns a database of several millions examples of both benign and malicious files, as a training and testing data for their heuristic analyser. Part of the database was examined inside a virtual environment (*sandbox*) for performed actions. Currently, the results of this step are then processed by a simple classification algorithm, which rates each action by manually chosen score and the classification is based on the total score and a certain threshold. However the accuracy of this approach is poor, only 39.3%, which is worse than random classification.

In this thesis we try to find the best classification method suitable for this task and implement a tool which could be integrated into the sanbox. It describes some of the available classification methods and how they are trained and evaluated. Described classifiers are then trained using a data set created from sandbox reports, provided by AVG Technologies. The training and testing is done using RapidMiner, a specialized data mining tool. The results of performed experiments are evaluated and the best-performing method is implemented as a part of application which provides all necessary functions: attribute extraction from sandbox logs, training, estimation of classifier's performance and classification of unknown samples.

1.1 Goals of the Thesis

The main goal is to create a new classification tool which could be embedded into sandbox system currently used by AVG Technologies. However, the tool cannot be created without an analysis of available classification methods, which can be used for this task, because the

[1]More information about AVG Technologies company can be found on its website, `http://www.avg.com/`

their performances depends also on the data we examine. The main goal can be divided into sub-goals as follows:

1. Analyse provided data set.
2. Experiment with various classifiers using k-fold cross-validation.
3. Evaluate results and choose the classification method.
4. Implement application which would provide selected classifier and could be integrated into the sandbox system.
5. Evaluate the performance of the implemented application.

The document is structured as follows: Chapter 2 gives an overview of classification, suitable methods, presents how classifiers are evaluated and also gives some examples of machine learning approaches to malicious code detection. Chapter 3 describes the provided data set and also the current classification method used by the company and its performance. Chapter 4 presents experiments performed in order to select which classifier to implement. Chapter 5 is focused on design of the application and chapter 6 describes the implementation of the application and evaluation of the performance of the application.

Chapter 2

Data classification

Classification is a task of learning a function which maps each record from input data to one of predefined class labels based on its properties, represented as set of attributes. The algorithm implementing this task is called classifier. This term is also used for the resulting function, also called classification model. Model can be represented in various forms, for example as a set of rules, decision tree or mathematical expression.

During learning, classifier is provided by records for which we already know correct class labels. For this reason, classification is considered as a supervised learning method, which means that the classifier is provided by correct output for each training record. The main goal is to find pattern which is common to all records (or at least to most of them) belonging to the same class and at the same time not true for records from other classes. When the learning is finished, the classification model can be used either to determine classes of unknown instances or also as a description of features common to instances of the same class and different from instances of other classes.

2.1 Classification methods

This sections describes classification methods which were trained and evaluated (using provided data set) in order to select method for later implementation.

2.1.1 Bayes classifiers

Bayes classifiers calculate probability that a given sample belongs to a particular class. They are based on Bayes' theorem, named after Thomas Bayes: let X be a data tuple, described by values of n attributes, H a hypothesis, e.g. that the data tuple belongs to a particular class C. For classification, we want to determine $P(H|X)$, the probability that tuple X belongs to class C, also called the **posterior probability**. It can be estimated from the **prior probability** $P(H)$ (the probability that any tuple belongs to class C), posterior probability of X conditioned on H (the probability that tuple belonging to class C equals to X) and the **evidence** $P(X)$, which all can be estimated from training data. The Bayes' theorem is:

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)} \tag{2.1}$$

The classifier estimates prior and posterior probabilities for all m classes $C_1, C_2, \ldots, C_m$ and compares them to each other. Because value of $P(X)$ is constant, it can be omitted in comparison. The prior probability of class C_i can be either given by an expert or estimated from the training data as:

$$P(C_i) = \frac{|C_i|}{\sum_{k=1}^{m} |C_k|} \tag{2.2}$$

where $|C_i|$ represents the number of tuples of class C_i in the training set.

To simplify calculation of class-dependent posterior probability, the **Naïve Bayes** classifier assumes that all attributes' values are conditionally independent on each other, given the class of the tuple. Thus $P(X|H_i)$ can be estimated as:

$$P(X|H_i) = \prod_{k=1}^{n} P(x_k|H_i) \tag{2.3}$$

Here x_k refers to value of attribute A_k for given tuple X. The probability for single discrete attribute is given as number of tuples of class C_i having the value x_k for A_k divided by the total number of tuples in class C_i. For continuous attributes, their distribution is typically assumed as Gaussian with mean μ and standard deviation σ:

$$g(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.4}$$

so the probability is:

$$P(x_k|H_i) = g(x_k, \mu_{C_i}, \sigma_{C_i}) \tag{2.5}$$

The classifier predicts that given tuple belongs to class C_i if and only if:

$$P(H_i|X)P(H_i) > P(H_j|X)P(H_j) \text{ for } 1 \leq j \leq m, j \neq i \tag{2.6}$$

Since multiplying many probabilities could lead to floating point number underflow, the logarithms of probabilities are added instead. Because $log(xy) = log(x) + log(y)$ and logarithm is monotonic function, the classification is still given by the highest value. The products in equation 2.6 are substituted as follows:

$$P(H_i|X)P(H_i) = \log(P(H_i|X)) + \sum_{k=1}^{n} \log(P(x_k|H_i)) \tag{2.7}$$

2.1.2 Decision trees

One of the most popular classifiers are decision trees. They are constructed by repeatedly dividing training data into subsets based on attribute values. Usually in each test, a value of one attribute is compared to a constant, but sometimes two attributes are compared to each other or some function of several attributes is used. Building stops after certain condition is reached – for example the minimal data subset size or tree maximal depth can be limited.

When classifying an unknown record, tree is routed from a root node, following path determined by attribute test results in internal nodes. When a leaf node is reached, its value is the result of prediction. Decision trees are also easily readable and it is visible which attributes are the most important for the prediction.

There are several methods to build the tree and choose suitable split attribute in each step. **ID3** algorithm invented by Ross Quinlan [10] uses information gain measure. It is based on calculating expected amount of information necessary to correctly classify a record, also known as entropy. Let D be the training set, $|D|$ the number of instances in D, m number of distinct classes $C_1, C_2, \ldots, C_m$ and p_i non-zero probability that a tuple from D belongs to class C_i (estimated as $|C_i|/|D|$). The amount of information needed to classify instance in D is given by:

$$\text{Info}(D) = -\sum_{i=1}^{m} p_i \log_2(p_i) \tag{2.8}$$

When a training set is divided into k subsets $D_1, D_2, \ldots, D_j$ by attribute A, the information needed for classification after split is estimated as:

$$\text{Info}_A(D) = -\sum_{j=1}^{k} \frac{|D_j|}{|D|} \times \text{Info}(D_j) \tag{2.9}$$

Information gain is the difference between the original information requirement and the requirement for each created subset, thus:

$$\text{Gain}(A) = \text{Info}(D) - \text{Info}_A(D) \tag{2.10}$$

This measure tends to favour attributes with a larger number of values. For example in extreme case, when one of the attributes had unique value for each record (e.g. ID), split using this attribute would result into maximal information gain, however it is not useful for classification. In **C4.5** algorithm [11], successor of ID3, gain ratio measure is used in attempt to overcome this bias. As a kind of normalization to information gain ratio, it also considers the number of records in each created subset. It does so by adding new “split information” value defined as:

$$\text{SplitInfo}_A(D) = -\sum_{j=1}^{k} \frac{|D_j|}{|D|} \times \log_2\left(\frac{|D_j|}{|D|}\right) \tag{2.11}$$

The gain ratio is then estimated as:

$$\text{GainRatio}(A) = \frac{\text{Gain}(A)}{\text{SplitInfo}_A(D)} \tag{2.12}$$

Gain ratio measure however tends to prefer unbalanced splits with one partition much smaller than the others. Several other methods exist, but *“no one attribute selection measure has been found to be significantly superior to others. Most measures give quite good results”* [5]. Most algorithm consider all attributes when deciding on split, but there are exceptions: for example RandomTree algorithm randomly selects subset of attributes from which the tree is built. An extension is the RandomForest algorithm which trains several decision trees, each with randomly selected subset of attributes. The class of unknown instance is determined by voting.

To avoid overfitting, which is the situation when learned tree fits training data too well and almost all training records are classified correctly but it fails to correctly classify unknown records, the tree is pruned. The pruning can be done during the building process (pre-pruning), which means that the algorithm tries to decide when to stop creating more subtrees, or after the complete tree is built (post-pruning), during which some subtrees are replaced by single leaves. Although pruning causes worse performance on the training data, it may lead to better results with unseen records [6].

2.1.3 Neural Networks

Neural networks are algorithms inspired by real biological neural systems. Like in brain, neural network consists of number of **neurons**. Each neuron realises particular function of several inputs and one output. They are organised into layers. The first layer is used for input, number of neurons in this layer is the same as number of attributes in data set. The last layer servers as output – it provides the result of classification. It usually has one output neuron for each defined class, but not necessarily – for two-class problems only one output neuron is needed, which will output “low” value for the first class and “high” for the second class. Between these layers there are one or more **hidden layers**. In **feed-forward** networks, neurons in one layer are connected only to neurons in the next layers. Usually each neuron is connected to all neurons in the next layer, but other layouts are also possible. Figure 2.1 shows an example of feed- forward network for two-class classification problem with each record having five attributes.

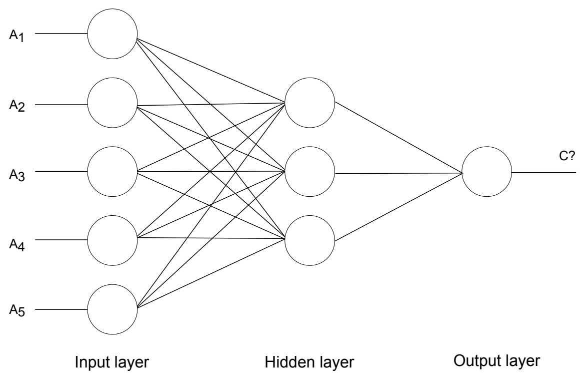


Figure 2.1. An example of feed-forward neural network with one input, one hidden and one output layer. Attributes are connected to input layer, the output layer yields predicted class label. Each connection has assigned its weight which is adjusted during training.

The connections between neurons have associated weights. In the beginning, they are set to random values and during learning their values are adjusted towards desired output of the whole network. In the **back-propagation** method, this is done by connecting each record from training set to the network inputs and computing the output of the network with current weights. The error between the correct and actual output is then propagated back to network and the weights are adjusted. The output is computed in forward direction – outputs of neurons in layer k are computed before the outputs in layer $k+1$. Each neuron implements an activation function, commonly used is the **sigmoid** function, defined as:

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.13}$$

The goal is to determine vector of weights $\mathbf{w}$ which minimizes the sum of squared error which is given as:

$$E = \frac{1}{2}\sum_{i=1}^{N}(y - f(x))^2 \tag{2.14}$$

where N is the number of records in training set, y is the instance's real class label and $f(x)$ the network's output value. To solve this problem, the **gradient descent** method is used. The weights of the connections are updated using formula:

$$w_j \leftarrow w_j - \lambda \frac{\delta E}{\delta w_j}$$

where λ is the learning rate. In back-propagation, the weights are updated in reverse direction – the weights at level $k+1$ are updated before the weights at level k.

When the whole training data set is processed, the process is repeated again. Each repetition is called an **epoch**. The total number of performed epochs can be determined manually or the learning process can stop when certain conditions are met, e.g. when the average output error or number of misclassified records during one epoch is smaller then some specified threshold.

Major drawback of neural network classification is that the knowledge represented by connections and their associated weights is difficult to interpret by humans. Also the network topology (number of neurons and layers) must by selected which may lead to long trial-and-error process until acceptable results are achieved [15] [6].

2.1.4 Support Vector Machines

Support Vector Machines (SVM) are based on finding the best hyperplane which separates instances of each class – the **maximum margin hyperplane**. They are defined for two-class problems, however there are modifications for classification with three or more classes. Because hyperplane is a kind of linear model, the data must be linearly separable, however SVM can be used also of non-linearly separable problems by mapping the original data into a different space, where the boundary between classes becomes linear. An example of maximum margin hyperplane is shown in figure 2.2.

The maximum margin hyperplane is a hyperplane which separates data, so that all instances are correctly classified and also gives the largest separation between classes. It is found using **support vectors** and **margins**. Margin is the distance between the hyperplane and nearest training instance of either class. We expect that hyperplanes with larger margin will have less classifications errors for future data, so SVM searches for the hyperplane with the largest possible margin.

If we have two-class problem with N training instances, we can denote each instance as (x_i, y_i), where $x_i = (x_i 1, x_i 2, ..., x_i d)^T$ corresponds to the attributes of i-th instance and $y_i \in -1, 1$ denotes its class. The hyperplane can be written as:

$$H : \mathbf{w} \cdot \mathbf{x} + b = 0 \tag{2.15}$$

where $\mathbf{x}$ and b are parameters of the model. Any point laying above the separating hyperplane satisfies

$$\mathbf{w} \cdot \mathbf{x_a} + b > 0 \tag{2.16}$$

Similarly, any point below the separating hyperplane satisfies

$$\mathbf{w} \cdot \mathbf{x_a} + b < 0 \tag{2.17}$$

If we label all points above the hyperplane as class $y_i = 1$ and points below it as $y_i = -1$, we can predict the class label y for any test example $\mathbf{z}$:

$$y = \begin{cases} 1, & \text{if } \mathbf{w} \cdot \mathbf{z} + b > 0; \\ -1, & \text{if } \mathbf{w} \cdot \mathbf{z} + b < 0. \end{cases} \tag{2.18}$$

Two parallel hyperplanes H_1 and H_2 which defines the "sides" of the margin can be expressed if we adjust the parameters $\mathbf{x}$ and b as:

$$H_1 : \mathbf{w} \cdot \mathbf{x} + b = 1 \tag{2.19}$$
$$H_2 : \mathbf{w} \cdot \mathbf{x} + b = -1 \tag{2.20}$$

The margin is given by the distance of these two hyperplanes. Any training instance which lies on either of these hyperplanes is called as support vector. There is always at least one for each class. The support vectors gives the most information regarding classification and they uniquely define the maximum margin hyperplane, the rest of training set is irrelevant and can be removed without changing the hyperplane.

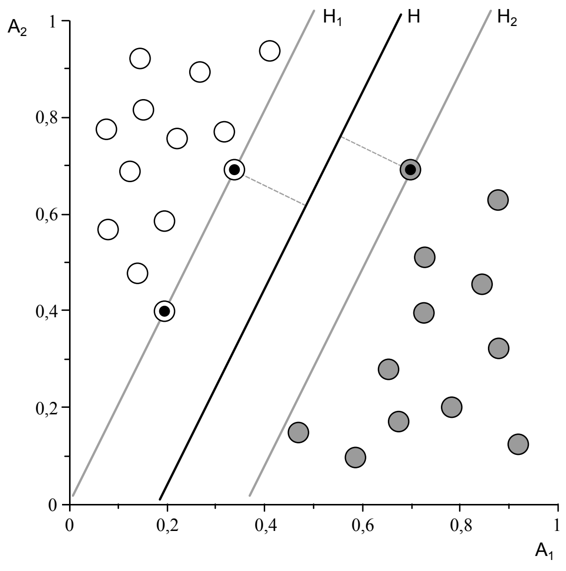


Figure 2.2. An example of a maximum margin hyperplane H in two-class problem within two-dimensional space. Points (training instances) lying on hyperplanes H_1 and H_2 are called support vectors – the rest can be omitted without changes to the hyperplane.

The process of finding the hyperplane can be rewritten that it will become a problem known as *constrained convex quadratic optimization problem.* It can be solved using general optimization software package, however for larger data sets specialized and more efficient SVM algorithms exist. Once the support vectors are found and the maximum margin hyperplane defined, the support vector machine is trained and can be used for classifying new instances [15] [6].

2.1.5 Boosting

Boosting is an example of an ensemble or combination model – it combines several models of the same type and the resulting class is determined by voting. The votes are not equal, each created model has assigned its weight, based on how well it performs. The models are trained consequently and any classification method can be used for their creation. After each model is trained, records' weights are recalculated, so incorrectly handled records are given higher weight than records handled correctly. This means that the next trained model will more focus on the misclassified instances, seeking for model which would complement the previous one.

One of widely used boosting methods is **AdaBoost** (short for Adaptive Boosting), designed especially for classification, formulated by Yoav Freund and Robert E. Schapire in [4]. At the beginning, all training records are given the same weight. Then the underlying classifier is called to generate model based on the data and weights are adjusted – decreased for correctly classified instances and decreased for the misclassified ones. Finally, next iteration is started – new model is built and training records are re-weighted. If e denotes the classifier's error on the weighted data, new weight for correctly classified instance is calculated as follows:

$$\text{weight} \leftarrow \text{weight} \cdot \frac{e}{1-e}$$

Then the weights are re-normalized, so their sum remains same as before, which causes increase of weights of misclassified instances. The modelling is stopped when the overall model error exceeds or equals 0.5 or when it equals 0, because then all weights become 0.

If the selected classification method does not support weighted instances in training set, it can be still used with boosting. In this case, an unweighted data set is created by re-sampling – each instance is selected into new data set with probability proportional to its weight. This means, that high weighted instances are replicated frequently and ones with low weight may never be selected. The data set is provided to the classifier once it reaches the size of the original weighted one [15].

2.2 Performance evaluation

Once classifier is trained, it is useful to measure how well it performs to classify previously unseen data. For this reason it is tested against unlabelled test set. The test set is randomly selected from all available data beforehand and it is not used during learning.

Counts of correctly and incorrectly classified test set records can be visualised as a two-dimensional **confusion matrix**. Each entry in this matrix represents the number of records for which the row is their real class and the column is the predicted class. For goo results, values on the main diagonal are large and all other values are small, ideally zero.

To make it more convenient to compare different classifiers, confusion matrix can be summarized into one number using a performance metric such as **accuracy** or **error rate**:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

$$\text{Error rate} = \frac{\text{Number of wrong predictions}}{\text{Total number of predictions}}$$

Accuracy (and error rate) calculated on the test set can be also used to compare two different classifiers on the same domain.

Besides accuracy, the performance of a classifier can be also observed in terms of the true positive rate (TPR, also called sensitivity) and the false positive rate (FPR). Knowing number of all positive (P), negative (N), true positive (TP) and false positive (FP) samples, these rates can be calculated as $TPR = {}^{TP}/_{P} = {}^{TP}/_{TP+FN}$ and $FPR = {}^{FP}/_{N} = {}^{FP}/_{FP+TN}$.

2.2.1 Selecting training and testing data

There are several options how to create training and testing data set. The simplest method is the **holdout**. In this method, data are partitioned into two disjoint sets, one used for training, one for testing. Main weakness of this method is that part of the data is not used to build the classifier, thus the resulting model might not be as good as when all records were used. On the other hand, the larger the training set is, the less reliable is the error estimate computed from the test set. Also, the result is dependent on the composition of the two sets, for example if all instances of one class are present in the test set but none of them are in the training set, classifier will probably perform poorly for that class. To improve error estimates, holdout method can be run more times and the resulting estimate is the mean of values from each run – this scheme is called the **repeated holdout** method. **Stratification** can also be used, which divides data into two parts keeping ratio between classes in each subset as in original data.

To ensure all records were used for both training and testing and maximize amount of data used for training, the ***k*-fold cross-validation** can be used. In this approach, data is split into k subsets and the validation is performed k times. In each run, one subset is used for testing and the others for training. The resulting error measurement is found by averaging results from each run. A special case is the **leave-one-out** method in which $k = N$, the size of the data set. In each run, exactly one record is used for testing. The advantage is that maximum of available data is used for training, however it might be computationally expensive to repeat the process N times [15].

2.3 Comparing various classifiers

Although accuracy gives some overview on how different classification methods perform compared to each other, it is good to prove that the difference between results are not given only by chance. This can be confirmed using the **Student's *t*-test**, introduced by William Sealy Gosset ("Student" was his pen name). It is a statistical method that can be used to determine whether the mean of two samples differ significantly or not. Measurements observed using the same classifier but with different data (for example each fold in k-fold cross-validation) can be considered as samples from a probability distribution – in particular ***t*-distribution** with $k-1$ degrees of freedom, where k equals to the number of measurements. This distribution looks similar to normal (Gaussian) distribution, but the defining function is different. The hypothesis (**null hypothesis**) tested is that the two models are same – the difference between mean error rates is zero. If the hypothesis can be rejected, we can conclude that the difference between the models is statistically significant and we can select the model with the lower error rate.

The test computes t-statistic with $k-1$ degrees of freedom. Let $\overline{err}(M_1)$ and $\overline{err}(M_2)$ be average error rate for model M_1 and M_2, respectively and $var(M_1 - M_2)$ the variance of the difference between these two models. The t-statistic for $k-1$ degrees of freedom is then computed as:

$$t = \frac{\overline{err}(M_1) - \overline{err}(M_2)}{\sqrt{var(M_1 - M_2)/k}} \tag{2.21}$$

The variance between the means of the two compared models is estimated as:

$$var(M_1 - M_2) = \sqrt{\frac{var(M_1)}{k_1} + \frac{var(M_2)}{k_2}} \tag{2.22}$$

To determine if models M_1 and M_2 are significantly different, we compute t and select the **significance level** – usually 5% or 1%. Based on the computed t and selected significance level, the t-distribution value z is found – either computed or looked up in table of selected values. If $t > z$ or $t < -z$, then t lies in the rejection region and the null hypothesis can be rejected. That means, that the difference between models M_1 and M_2 is statistically significant. If we cannot reject the null hypothesis (i.e. $-z < t < z$), any difference between these two models can be attributed to chance [6].

2.4 Cost-sensitive classification

When dealing with unknown samples, in some cases the cost of false positive classification might be much higher than the cost of false negative classification (or vice versa). These costs can be taken into account during the training by providing a **cost matrix** which allows to bias the model to avoid costly errors.

The cost matrix looks similar to the confusion matrix, however the cells contain assigned costs instead of number of classified examples. The values on the main diagonal are always zero (because there is no cost of correct classification).

However, some classifiers does not support cost-sensitive learning and editing them might be non-trivial. This problem can be solved by using the **MetaCost** method, proposed by Pedro Domingos [3] which treats underlying classifier as a black box and does not require any changes to it. It creates multiple training sets using sampling with replacement and learns a classifier for each of them. Probabilities that each sample belongs to each of the classes are estimated by using the trained classifiers and the training set is relabelled to optimal classes according to the given cost matrix. This process is repeated m times, producing m models. When an unknown sample is given to the resulting m models, the decision is determined by voting (as a class predicted by the greatest number of models).

2.5 Machine learning in malware detection

Machine Learning was first used to detect malware by Schultz et al. [13]. They used three feature extraction approaches (program headers, strings and byte sequences), and free classification methods (RIPPER, a rule-based classifier, Naïve Bayes, and Multi-Naïve Bayes). All of the methods were more accurate than signature-based methods, best performing being the Multi-Naïve Bayes using byte sequences with accuracy of 97.76%.

Another approach is the Common N-Gram analysis (CNG), proposed by Abou-Assaleh et al. [1]. N-grams is file substrings a fixed length n. For each class, L most frequent n-grams collected from training data represents class profile. A new instance is classified by building its class profile and using the k-nearest neighbours algorithm. They achieved 98% accuracy for several parameter configurations.

Kolter and Maloof [8] also collected n-grams, each being sequence of 4 bytes, then selecting 500 most relevant ones using information gain measure. They applied four different machine learning methods: k-nearest neighbours, Naïve Bayes, a support vector machine (SVM) and a decision tree. All of the methods, excluding k-NN were also "boosted". Best performance was achieved by boosted the decision tree, followed by SVM and boosted SVM.

They also tried to determine the kind of malware (e.g. backdoor virus), omitting benign data set, with boosted decision tree and SVM being again the best performing methods. To estimate results in an real-world environment operation, the authors also trained classifiers using malware discovered before July 2003 and tested them against files discovered between July 2003 and August 2004. The results were not as good as in their first experiment, with boosted decision tree again outperforming others classifiers.

Henchiri and Japkowicz [7] worked with same data set as Schultz et al. [13] but with different – hierarchical – feature extraction approach, selecting n-grams present at rates above threshold within specific virus family and also present in minimal amount of families. They evaluated several classifiers (decision trees, Naïve Bayes and a support vector machine), authors claimed they achieved better results than in [13].

Ding et al. [2] experimented with dynamic heuristic analysis using set of 8823 malicious and 2821 benign samples. Each of them was run inside virtual environment and monitored by API tracer for Windows API calls, resulting into a 35-dimension feature vector, each attribute representing one event (e.g. file system operations), the value indicated only whether that event occurred or not, number of these occurrences in each run was not recorded. They built two classifiers – a statistical model and Mixture of Experts model. They achieved true positives rate of 96.01% with statistical and 75.2% with Mixture of Experts model.

2.6 Machine Learning Tools

When dealing with data mining tasks, various existing tools may be used for data preprocessing, data visualisation, modelling and evaluation. Two different tools were used – Weka for data analysis and RapidMiner for training and evaluation of described classification methods on the given data set.

2.6.1 Weka

Weka, (Waikato Environment for Knowledge Analysis), is a tool developed at university of Waikato, distributed under GNU GPL license[1]. It is written in Java and can be easily run on most used operating systems and platforms and can be used as a stand-alone application or integrated into custom Java applications. It provides a collection of both supervised and unsupervised machine learning algorithms, and also tools for data pre-processing and visualisation [14].

2.6.2 RapidMiner

RapidMiner is an open source data mining tool developed by Rapid-I GmbH. It is available in two editions – Community Edition available under **GNU AGPL** license[2] and Enterprise Edition which offers extended technical support and it can be also integrated into proprietary closed-source applications. Like Weka, it is written in Java and can be integrated into other Java applications. Functionality can be extended by plug-ins – for example learners from Weka can be integrated into RapidMiner in this way [12].

Data mining tasks in RapidMiner are defined as processes consisting of various operators which are applied to input data. Each process is described in an XML file and can

[1] `http://www.gnu.org/licenses/gpl.html`
[2] `http://www.gnu.org/licenses/agpl-3.0.html`

be created either manually or using a graphical user interface. Operators can be combined into chains and in some cases even nested together making the tool universally applicable to many data mining tasks. There are more than 600 operators of various types, including basic process control (loops, branching, etc.), data transformation (attributes normalization, filtering, sampling, etc.), modelling (classification, regression, clustering, etc.) or support for evaluation of the created model. Operators define their inputs and outputs and their types (e.g. data set, model, vector etc.). The types are checked during both designing and execution of a process and when incompatible input and output are connected, RapidMiner raises an error. There are also additional outputs, not included in normal processing flow, which can be stored in a log, such as execution time or the count of how many times an operator was executed.

Chapter 3

Data set

The data set was generated using database of executable files owned by the AVG Technologies company, however only 100 000 malicious and 45 384 benign samples were used, due to the fact that it is computationally expensive to process the whole database (which contains millions of executables). This part of database was then analysed using the virtual environment and each example's behaviour was monitored (namely performed Windows API calls). Generated log files was then processed by specialized tool which filtered only part of sample behaviour (file system operations, network access, Windows system registry modifications, etc.) and produced final report for each file. For classification purposes, we extracted a vector of 39 numeric attributes from each report. Each attribute represents specific action and its value number of occurrences of that action.

There are many instances with null attribute vector, which means that during examination in the virtual environment, none of the monitored actions were performed. Although it is perfectly fine for benign files to not perform any of those actions, it is at least suspicious in the case of malicious files. Probably some of them can determine that they are run inside a virtual environment and so they did not perform any suspicious actions to avoid being detected. Another option is that they perform different actions which were not monitored. Despite the actual reason, these malicious set records with null attribute vector were removed from the data set.

Although the number of attributes is relatively high, not all of them may be significant for classification. Removing the least significant attributes can lead to shorter learning times without small or even no impact on classifier's performance. Weka toolbox provides several algorithms to help selecting relevant attributes. The gain ratio measure was used (which is also used to build decision trees[1]). The complete list of attributes is included in Appendix A. The attributes at the bottom of the list has zero value for almost all records in the data set – and in the case of attribute "HIDE_PROCESS" this is true for all of them. However, if the whole database of executables (and not only provided subset) was analysed, these attributes may become more significant.

3.1 Current classifier used by AVG Technologies

Current classifier is embedded into sandbox as a part of the tool which generates reports for each file. The classification method is simple: each monitored action is assigned a certain score (either negative or positive). If the total score for processed file exceeds pre-

[1]See section 2.1.2 for gain ratio definition.

determined threshold, it is marked as a possible new malware, if not, it is considered benign. Both scores and threshold were chosen manually. The result is included in the generated report. The accuracy of this approach is 39.3% for the whole data set, which is worse than classification by random choice (which has the accuracy of 50%). If the malicious samples with null attribute vector are omitted, the accuracy is higher, 61.28%. As we can see in table 3.1, although the false positives rate (i.e. the number of benign samples marked as malicious) is very small (0.93%), most of the malicious files are not detected (87.78% or 74.48% if null vector malicious samples are omitted). The scores and threshold are intentionally set to achieve false positive rate that small – 1% is the maximum value the company is willing to tolerate.

Table 3.1. Confusion matrix for current classifier. Although the false negatives rate is small, most of malicious samples are misclassified.

(a) Whole data set

	predicted malicious	predicted benign
true malicous	12.22%	87.78%
true benign	0.93%	99.07%

(b) Data set without malicious records having null attribute vector

	predicted malicious	predicted benign
true malicous	25.52%	74.48%
true benign	0.93%	99.07%

Chapter 4

Choosing classification method

To choose the classification method to implement, several experiments with different classifiers were performed using RapidMiner (briefly described in section 2.6.2).

The experiments were performed using the provided data set without malicious samples with null attribute vector. Five different classification methods described in chapter 2.1 were tested - Naïve Bayes, a decision tree (with gain ratio measure), RandomForrest algorithm, a support vector machine and a neural net. Decision tree and neural net were also boosted using AdaBoost algorithm. Performances were estimated using 20-fold cross validation, i.e. by dividing data set into 20 parts, 19 parts used for training and one part for testing, repeating this process 20 times with different testing set each time. Accuracy, absolute numbers of test instances classified correctly (true positives and negatives) and incorrectly (false positives and negatives) and the learning time were observed.

All experiments were performed with default RapidMiner settings. Besides that, decision trees were also built with various maximal depth settings (default 20, 15, 10 and 7). Four different structures of neural networks were used. All of them had one hidden layer with different numbers of neurons: 22 (default RapidMiner settings[1]), 15, 10 and 5.

When all results were gathered, classifiers can be compared to each other. To ensure that the differences are not given only by chance, the Student's t-test (described in section 2.3) were performed, using accuracies recorded for each run, giving 20 samples for each classifier.

The goal of the second set of experiments was to lower the false positive rate below 1% which is the highest value the company would tolerate. This was done by using the MetaClass operator together with the classifiers. The cost matrices were determined manually by a trial-and-error process. The false negative cost was always set to zero and the false positive cost was increased until the average false positive rate achieved value around 1%. The performance was estimated using 20-fold cross-validation, as in previous experiments. The "boosted" versions of neural nets and decision tree were not included in this set of experiment, because it turned out that boosting does not have any significant impact on the performance (however it significantly increases learning time).

[1]Default number of nodes in hidden layer is given as $n = \frac{A+C}{2} + 1$ nodes, where A is the number of attributes and C the number of classes, in our case $n = \frac{39+2}{2} + 1 = 21.5 \approx 22$

4.1 Experimental results

This section presents obtained results of the described experiments. The collection of five classifiers was trained on the data set – first with default settings and then together with the MetaCost operator (in order to achieve lower false positive rate).

4.1.1 Experiments with default settings

In general, all used classification methods performed better than the current classifier, regarding their accuracy and true positive rate. On the other hand, the average false positive rate is higher for all of them – it varies from 1.18% for Naïve Bayes to 32.71% for RandomForrest (however results for this algorithm vary a lot for each fold). Table 4.1 and figure 4.1 show results for all classifiers sorted by their accuracy.

Because training a support vector machine using standard implementation included in RapidMiner turned out to be very time consuming, this experiment was performed using “Fast Large Margin” learner instead, which is based on fast support vector learning scheme included in LIBLINEAR[2]. Although its results are similar to other SVM implementations, the training time is much better even for data sets containing millions of records and/or attributes.

The best results were achieved using neural net with default RapidMiner settings. The average accuracy is good, 93.97% ($\sigma = 0.30\%$). The true positive rate (TPR) is also the best among all methods, being 97.31% ($\sigma = 0.90\%$). However, the false positive rate (FPR) is 9.49% ($\sigma = 0.95\%$) which is approximately 10 times higher value than the one of the current classifier. Networks with different number of neurons than default in general had same or almost same accuracy, however the amount of time necessary for training was twice smaller in the case of the smallest network.

The major drawback is the learning time which is one of the highest (almost 8 hours for whole 20-fold cross validation) but it is still good compared to the time necessary to analyse samples using virtual environment (months). On the other hand, if the whole available database (millions of samples) were used for training, this might be a large issue. Another disadvantage is unclear structure of neural networks in general. For production usage, a faster learning scheme than the back-propagation would be a better choice.

The second best performing classifier were decision trees, either with default setting of maximal depth (which is 20) or adjusted to 10 – difference between these two trees was proven as statistically insignificant using the *t*-test. The accuracy is 92.93% ($\sigma = 0.33\%$) which is lower than the neural network, however, trees have an advantage of clear structure and the learned knowledge can be easily interpreted by human. The true positive rate is also lower (94.85%, $\sigma = 0.46\%$) and the false positive rate is a little better than for the neural network, but the difference is not significant. When the maximal depth is set to smaller value (which leads into smaller number of tested attributes during classification), the accuracy, TPR and FPR are also smaller, but still above 90%. Despite poorer performance, building decision trees was 12 times faster than training a neural net.

These were only two classification methods with accuracy better than 90%. The support vector machine (Fast Large Margin) is third best performing. Fourth best accuracy was achieved by RandomForrest algorithm, however results for this classifier vary a lot for each fold. This is probably due to the way RandomForrest algorithm works. Since it creates

[2]R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A Library for Large Linear Classification. *J. Mach. Learn. Res.*, 9:1871– 1874, June 2008.

number of decision trees, each of them with randomly selected subset of attributes, some of these trees are built based only on attributes which are not that useful for classification, thus the resulting tree does not perform well. Although the final decision is determined by weighted voting, it seems it could not overcome this problem.

The worst accuracy among selected classifiers was obtained by the Naïve Bayes classifier. Its results were similar to the current approach. The main reason could be that the "naive" assumption that all attributes are independent of each other is probably not true.

Two best performing methods (neural net and decision tree) with default RapidMiner settings were also trained using AdaBoost, described in chapter 2.1.5. Regarding decision trees, the difference of accuracy was statistically insignificant compared to the non-boosted version. The accuracy of the boosted neural network was even worse than the regular one and the learning took approximately twice more time.

Table 4.1. Average results for all classifiers estimated by 20-fold cross validation sorted by average accuracy. The last row shows values for current classifier which were calculated using complete data set. Neural networks had one hidden layer with different number of neurons. Decision trees were trained using different settings of maximal depth.

Classifier	Accuracy	σ	TPR	σ	FPR	σ	Learning time
Neural net (15 neurons)	94.06%	0.19%	97.39%	0.63%	9.41%	0.80%	16 min
Neural net (22 neurons)	93.97%	0.30%	97.31%	0.90%	9.49%	0.95%	24 min
Neural net (10 neurons)	93.93%	0.31%	97.42%	0.77%	9.70%	0.91%	14 min
Neural net (5 neurons)	93.61%	0.49%	96.89%	0.73%	9.79%	0.54%	12 min
Decision tree (depth 20)	92.93%	0.33%	94.85%	0.46%	9.07%	0.55%	1 min
Decision tree (depth 10)	92.93%	0.33%	94.86%	0.45%	9.08%	0.54%	44 s
Decision tree + AdaBoost	92.93%	0.33%	94.85%	0.46%	9.07%	0.55%	26 s
Decision tree (depth 7)	92.34%	0.31%	93.40%	0.54%	8.77%	0.56%	26 s
Decision tree (depth 5)	90.34%	0.33%	88.91%	0.61%	8.17%	0.61%	13 s
SVM	84.07%	1.36%	76.23%	3.70%	7.78%	1.22%	1.5 min
Neural net + AdaBoost	82.22%	1.63%	72.84%	4.37%	8.05%	1.36%	46 min
RandomForrest	72.94%	10.59%	78.38%	16.85%	32.71%	31.55%	13 s
Naïve Bayes	62.26%	0.52%	27.06%	0.94%	1.18%	0.24%	1 s
Current classifier	61.28%		25.52%		0.93%		–

4.1.2 Experiments with MetaCost

In these experiments the classifiers were tuned to achieve similar false positive rate as the method currently used by the company, i.e. around 1%, using the MetaCost method. Table 4.2 displays average accuracies, true positive rates and false positive rates, together with average learning time and the value of false positive cost used during training.

The only classifier which could not be properly tuned was the decision tree. The tree could be trained with false positive rate of either 0.37% (with average accuracy of 51.7%, which is almost the same as random classification) or 6.55% (with average accuracy of 78.99%) and nothing between; the boundary was approximately between the false positive cost values of 9.7 and 9.75. For this reason it is not included in the summary table 4.2.

The performance of other classifiers was approximately the same – arround 62%. The differences are probably due to the fact that the classifiers were not biased to the same FPR value. The best performing classifier was the support vector machine with accuracy of 62.51% (TPR 27.43%, FPR 1.07%), followed by the Naïve Bayes classifier which had accuracy of 62.02% (TPR 26.38%, FPR 0.96%). Again, the accuracy of the RandomForrest algorithm varies a lot (from 49.1% to 63.8%), as we can see in the Box-and-whisker chart chart in figure 4.2. The main difference between the classifiers is their learning speed, ranging from just a few seconds (Naïve Bayes) to almost an hour (support vector machine).

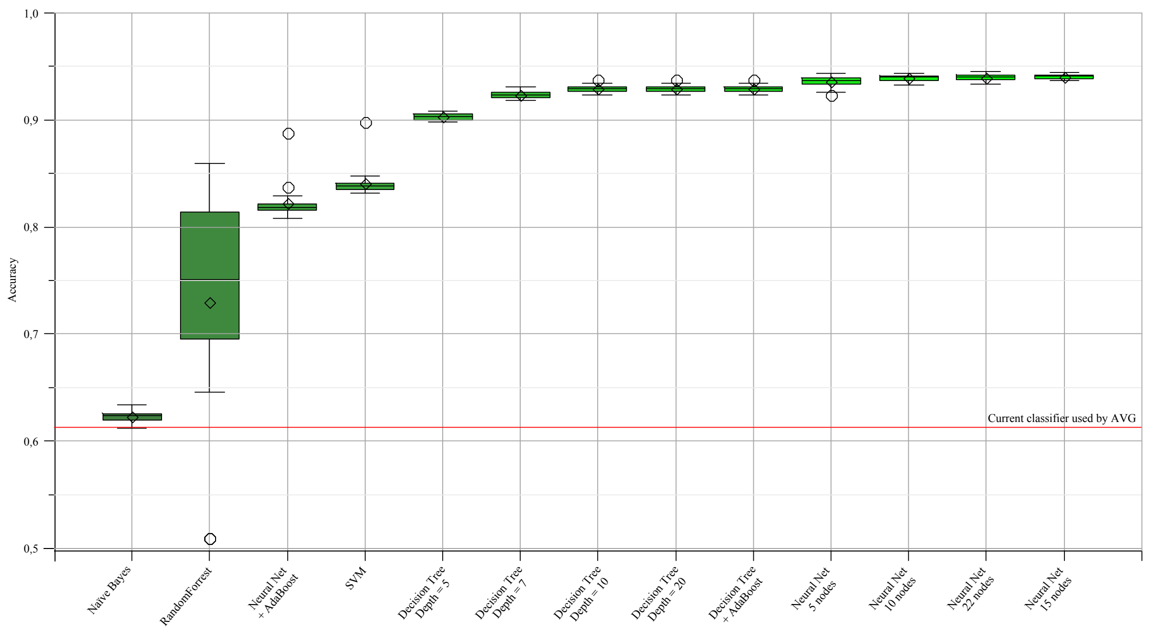


Figure 4.1. Box-and-whisker chart showing accuracy of all trained classifiers. The top and bottom of the boxes correspond with 25th and 75th percentile. The mean is plotted as a diamond, outliers as circles.

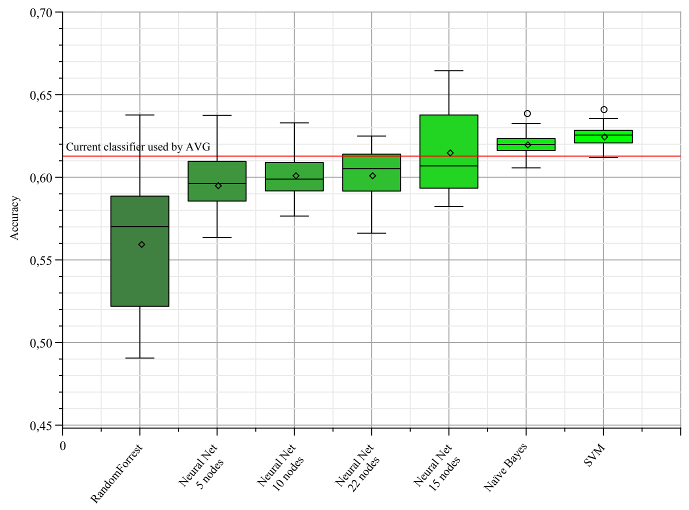


Figure 4.2. Box-and-whisker chart showing accuracy of all trained classifiers (combined with the MetaCost algorithm). The top and bottom of the boxes correspond with 25th and 75th percentile. The mean is plotted as a diamond, outliers as circles.

Table 4.2. Average results for all classifiers estimated by 20-fold cross validation sorted by average accuracy; the classifiers were biased using MetaCost method to achieve false positive rate around 1%. Neural networks had one hidden layer with different number of neurons.

Classifier	Accuracy	σ	TPR	σ	FPR	σ	FP cost	Learning time
SVM	62.51%	0.67%	27.43%	1.24%	1.07%	0.22%	4.25	53 min
Naïve Bayes	62.02%	0.74%	26.38%	1.44%	0.96%	0.22%	999 999.0	<1 min
Neural net (15 neurons)	61.55%	2.5%	25.66%	5.3%	1.17%	0.44%	12.5	48 min
Current classifier	61.28%		25.52%		0.93%		–	–
Neural net (22 neurons)	60.17%	1.53%	22.71%	3.23%	0.93%	0.26%	13.75	47 min
Neural net (10 neurons)	60.16%	1.44%	22.63%	3.08%	0.86%	0.3%	12.85	48 min
Neural net (5 neurons)	59.59%	1.91%	21.58%	4.05%	0.94%	0.36%	12.75	28 min
RandomForrest	56.04%	4.34%	14.54%	9.15%	10.13%	0.76%	0.85	1.5 min

4.1.3 Selected classifier

If accuracy would be the only criterion, neural nets are the most convenient classification method to use, followed by the decision tree. Although the neural net training is quite time consuming, there are different and faster learning schemes for neural nets than the used back-propagation.

However, the false positive rate of these two classifiers is much higher than what the company is willing to tolerate (1%). The second set of experiments showed that if the FPR is reduced to the maximal tolerated value, all tested classification methods perform with almost the same accuracy and the only difference is in the learning speed. Since only a part of the malicious and clean samples were used during the experiments, any difference in learning time could make significant impact if the complete database of millions of records would be used. Shorter learning time would allow to re-train the classifier more often, allowing more flexible reactions to new threats. For this reason the Naïve Bayes classifier was selected for implementation.

Chapter 5

Application design

The application implemented in this thesis should be easily integrated into the executable analyser currently used by the company. For this reason the design should allow easy addition of support of more types of input and output storages (for example databases of various types, remote APIs[1], etc.). The graphical interface is not included, because the application is meant to be run mainly automatically and thus the GUI is not necessary.

The application should provide these functions:

1. Attribute extraction from one or more sandbox logs.
2. Train the classifier from a training set.
3. Classify one or more unknown samples.
4. Estimate performance of the classifier using k-fold cross-validation

5.1 Attribute extraction

This process converts raw sandbox logs into tuples consisting of the ID, values of all supported attributes and the user-supplied class label. It can process many logs at once – in this case the logs are supplied as a CSV[2] file; each row contains one log. The output is also a CSV file: each row contains record ID, its attributes' values and supplied class label.

The application can be also used to extract attributes from one log at a time. In this case the input is the log file itself and the ID must can be either provided through a command line argument or it can be generated from the file name. The output can be appended to an existing CSV file or it can be saved to a INI file.

The extractor also keeps list of files created or modified during the execution in the sandbox virtual environment. This is because some actions (records in the log) may yield different attributes based on the file used during the action. For example, the line `Run file "<FILE_NAME>"` may yield either `RUN_FILE_OTHER` or `RUN_CREATED_FILE` attribute, depending on whether there is an action which created or modified the file `<FILE_NAME>` somewhere earlier in the log.

The user can also specify that samples with null attribute vector should not be written to the output CSV file. This is useful when extracting attributes from samples known to

[1]Application Programming Interface, used for communication between different systems

[2]Comma Separated Values. Each row contains one record consisting of several values, usually separated by comma, hence the name.

be malicious, because some of them might pass the analysis without being detected, as described in chapter 3.

5.2 Training and classification

The input to the training process is the training set; it can be created by the application but it is not necessary. The training set can be loaded from one or more files or directly from the application standard input. Each record consists of its ID, values of attributes and its correct class label. The output – trained Naïve Bayes classifier (described in section 2.1.1) – can be stored to a file for later usage.

To train the the classifier it is necessary to calculate the mean and the standard deviation for each attribute and class label in the training set and also prior probabilities for each class. The mean of a sample consisting of N values $x_1, x_2, \ldots, x_N$ is defined in [6] as:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{5.1}$$

and the standard deviation as:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2} = \sqrt{\frac{1}{N}\left(\sum_{i=1}^{N} x_i^2\right) - \overline{x}^2} \tag{5.2}$$

To allow bias the model towards positive or negative classification, each class label can be given a weight. Values of records from that class are then multiplied by the weight. To calculate the mean, standard deviation and prior probabilities, the number of records of each class is also multiplied by the corresponding weight. This results in the same situation as if every record appeared in the training set multiple times (depending on the weight), however unlike simple duplication of the records, this allows to use also floating point numbers as weights.

During the classification of unknown samples, the logarithms of probabilities of continuous attributes are calculated using equation 2.5:

$$\log(P(x_k|H_i)) = -\log(\sqrt{2\pi}\sigma_{C_i}) - \frac{1}{2}\left(\frac{x_k - \mu_{C_i}}{\sigma_{C_i}}\right)^2 \tag{5.3}$$

Because the term $\log(\sqrt{2\pi}\sigma_i)$ does not depend on the unknown sample, it is also calculated during training and stored in order to speed up the classification.

The input to classification is either a set of unknown samples (similarly to the training set) or just a single sample – in this case it can be provided either as a log file or as a list of attributes and their values. The output is either only the most probable class label and optionally also calculated probabilities for all class labels. If the classification is done for many samples at once, the output for each record also contain its ID and optionally the values of the attributes.

5.3 Estimation of performance of the classifier

The validation requires the same inputs as the training – the data set and the class label weights. It is done using the k-fold cross validation (see section 2.2.1) – the number of folds k can be chosen by user as a command line argument.

The application than follows the scheme described in section 2.2.1. First, the data set is partitioned into k subsets, ideally each containing the same number of records. The ratio between different classes in each subset should be kept the same as in the complete data set. The partitioning can be described as follows:

1. Prepare n buckets, where n is the number of classes in the data set.
2. Read the data set and put the sequence number of each record into corresponding bucket.
3. Calculate the target number of items of each class in each resulting subset.
4. For each of the resulting subsets, randomly pick calculated number of items (sequence numbers) from the buckets.
5. Read the data set again and replace the sequence numbers with actual records and save them to corresponding subset files.

In fact it is not necessary to work with the sequence numbers, however storing the whole data set in the memory might not be possible for larger number of records. The work-flow with sequence numbers allows to store only one record in the memory at a time (the record which is copied from the source data set into corresponding subset).

After the subsets are created, the application trains the classifier k times, each time with different training and test set and collects the results. When this process is finished, the results are averaged and printed to the application standard output. It shows the total number of classified records, number of correctly and incorrectly classified records (in total and for each class) and the accuracy, the true positive and false positive rates. The application does not require setting which class is positive and negative – it shows rates for all classes.

The application should also support creation of the subsets separately. This allows to store the subsets into files and run the cross-validation multiple times on the same data, without the need to re-create the subsets each time from scratch.

Chapter 6

Implementation and evaluation

This chapter describes the implementation of the application described in the previous chapter and performed tests and achieved results of the implemented classifier.

6.1 Implementation

The application user interface was implemented as a set of command line scripts. Each of them starts one of the functions described in Chapter 5. The classification of unknown samples and the attribute extraction are divided into two scripts – one for processing many samples at once and one for processing only one sample. There is also one extra script for creating subsets for later use in the cross-validation.

6.1.1 Python programming language

The application was implemented using the Python language (version 2.7). This dynamic interpreted language can be run on all major platforms. Its standard library covers most of the common tasks, including CSV file format support (used for storing data sets) and easy text manipulation (and thus log parsing). The *generator functions* can be used for simple iteration over even very large data sets without the need to load them completely into the memory in exactly the same way as iterating over regular memory-based arrays[1]. Python can also be easily embedded into other applications and it can be extended by modules written in low-level languages like C or C++ which allows additional speed-up of critical parts of applications. The language implementation, CPython, is distributed under an open source license which allows also commercial usage [9].

6.1.2 Code organization

Besides the executable scripts, the code is organized into two packages. The package `classification` contains implementation of the Naïve Bayes classifier (class `NaiveBayes`) and functions for loading the model from a CSV-like file as well as saving trained model (class `NaiveBayesCsv`). The module `sampling` contains function `kfold_sample` which implements splitting data set into k subsets using stratified sampling.

[1]With some limitations, for example it is impossible to get the number of items using the `len` built-in function or to seek backwards.

The package `dataset` contains everything necessary for working with the data set – loading sandbox logs, data sets and records (module `readers`), storing data sets and records (module `writers`), and the attribute extraction (modules `extractor` and `attributes`).

The source code is formatted according to the *PEP 8 – Style Guide for Python Code*[2].

6.1.3 Reading and writing data

For reading and writing data, there are two modules containing necessary functionality – `dataset.readers` and `dataset.writers`. The readers and writers are implemented as *context managers*. Besides the context manager interface methods, each of them contain one method for storage access – `fetch_all`, returning generator yielding records one by one, and `write_record` for writers. The `dataset.readers` module also contains two helper functions – `read_attribute_file` and `read_multiple_attribute_files` which return directly the generator, without the need to create the reader class and enter corresponding context (using the `with` statement). Thanks to the use of generators, only one record is stored in the memory at a time.

The rest of the application is not dependent on the actual storage. This allows easy implementation of new data storages or file format – one only needs to write the reader and writers classes and/or functions and change the `import <module>` statements in the corresponding parts of the application (the executable scripts).

6.1.4 Attribute extraction

The extraction procedure is started by one of the scripts `extract_single.py` and `extract.py`. The former takes single log file as a input. The user can supply the file name or it can be written directly to the standard input. The extractor (object of class `AttributeExtractor`) initializes all attribute values to zero and reads the file line by line. If the line contains text characteristic for an attribute, the value of that attribute is increased by one. When the end of file is reached, the attribute vector is returned and either print to the standard output in a INI format, (generated by the function `record_to_ini` from module `dataset.writers`) or appended to a given CSV file (using an object of class `CsvAttributeWriter`).

The second script reads the logs one by one from the CSV file, using an object of class `CsvLogReader`, and repeats the same procedure for each of them. The output is in form of a CSV file, including headers and it is written either to the standard output or to a given file, using an object of class `CsvAttributeWriter`.

6.1.5 Training

The training is started by executing script `train.py`. It takes one or more CSV files containing the training set as an argument (or the training set can be written to the standard input) and optionally weights assigned to some or all of the classes. If not specified by the user, a class has weight of 1.

The learning itself is implemented in the method `train` of the class `NaiveBayes`. It reads training records from the input file(s) and stores the sum of records' weights (given as a weight of the class label) for the entire training set and also for each class label separately. Each attribute's value is multiplied by the corresponding weight and the application calculates the sum of the values and the sum of the squared values. After the whole training set is processed, these sums of values and weights are used to calculate the required

[2]Python Enhancement Proposal 8, http://www.python.org/dev/peps/pep-0008/

statistics (described in section): the prior probabilities, the mean, the standard deviation and the value of the term $\log(\sqrt{2\pi}\sigma_i)$ from the equation 5.3 for each class.

The model is not represented in the application as a object of a special type, but as a tuple consisting of 4 dictionaries[3] (prior probabilities, means, standard deviations and values of $\log(\sqrt{2\pi}\sigma)$).

6.1.6 Classification of unknown samples

Each unknown sample can be classified using the `classify_single.py` script. It takes an INI file (which can be generated by the attribute extractor) or a raw sandbox log (in this case the attributes are extracted before the classification). The output is also in INI format, with the `___label___` attribute changed to the result of the classification and optionally with new attributes `probability(<CLASS>)` containing the calculated values for each class. The INI file is read using the function `ini_to_record` from module `dataset.readers`.

If the number of unknown samples is higher, the `classify.py` script might be a better option. It takes a CSV file (in the same format as a training set) and classifies each record. The output is again a CSV file, containing each record's ID and its most probable class, and optionally also values of all attributes and/or all all calculated probabilities.

The classification itself is implemented in the method `classify` of the class `NaiveBayes`. It iterates over all classes and calculates the logarithm of the probability that the unknown sample belongs to each class. It returns the label of the class with highest calculated probability and optionally all calculated values.

6.1.7 Estimation of the classifier's performance

The k-fold cross-validation process is started using the `xvalidate.py` script. It takes a data set as an input in the CSV format. If not done beforehand, it is partitioned into k subsets, according to scheme described in section 5.3, however the records are not stored in the memory, only the mappings from records' sequence numbers to corresponding subsets are kept, to minimize memory usage.

The number of items in each subset from each class is calculated by integer division $^{N_C}\!/_{N}$, where N_C is number of records of class C and N is the total number of records. The remainder is equally distributed among subsets, so the difference between the size of the smallest and largest resulting subset is at most the number of classes. Although it can create unbalanced subsets for small data sets (e.g. 10 items of 2 classes may be divided into subsets of 4, 4 and 2 items), the difference is insignificant in the case of larger data sets.

The application can split the data set itself, however it also supports working with already created subsets. Because the splitting can be time-consuming, if several experiments are run on the same data, the necessary time can be reduced significantly by creating the subsets only once before experimenting. The data set can be split into subsets using the script `subsets.py`, which takes the data set, required number of subsets and optionally the output directory as arguments. The resulting subsets are saved to files named `subset_X.csv`, where X is number of the subset, starting from 1.

The testing is performed using the `test` method of the class `NaiveBayes`. It takes the trained model and a testing set as an argument and returns the confusion matrix as a "two-dimensional" dictionary[4]. For example, to access the number of test samples which

[3]Dictionary is a Python built-in data structure which maps keys to corresponding values.

[4]More precisely, it is a dictionary that contains another dictionaries as items' values.

have correct class label “positive” and which were classified as “negative”, one can write `my_value = matrix['positive']['negative']`.

The application calls the `test` once for each subset, used as test set, other subsets are merged together and used as the training set. The resulting values are added together. After this process is completed, the total number of classifications and number of correctly and incorrectly classified test samples are printed to the standard output, together with the accuracy. These counts are printed also for each class label. Since the application does not take the positive class label as an argument, it prints the true positive rate and the false positive rate for each class. The average learning and testing time are also recorded and printed. It is also shown how long (in average) it takes to classify one unknown record. If required by the user, the results for each of the testing sub-processes can be printed as well. An example of the output is shown in section B.2.4, in the listing B.5.

6.2 Experimental output

The performance of the implemented classifier was estimated using the 10-fold cross- validation. The goal was to observe the changes in the performance by increasing the “negative” class weight in steps of 0.1, from 1.0 to 12.0 (which results in 111 runs). The weight of the “positive” class was always set to 1 – since there are only two different classes, it is not necessary (only the ratio between the weights matters, not the absolute values). Four measures was monitored – true positive rate, false positive rate, accuracy and the average learning time necessary for one iteration of the cross-validation. The same experiment was repeated using only the 15 most significant attributes (estimated by the information gain measure) to see how the pre-selection affects the performance and learning time.

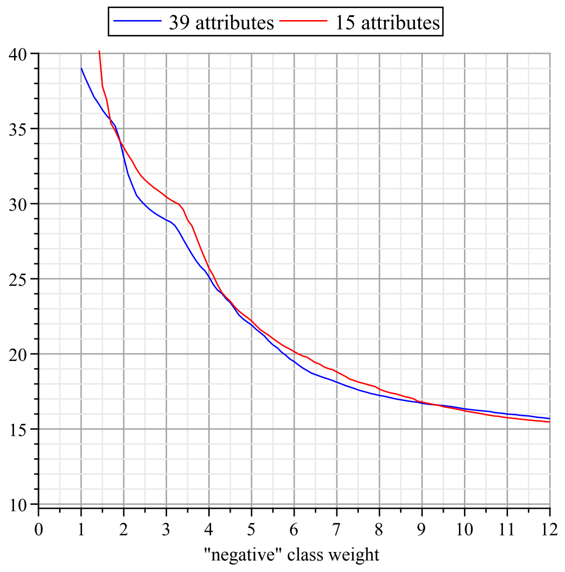
6.2.1 Results

The figure 6.1 shows the relationship between the true and false positive rates and the weight for both experiments. The rates (and also the accuracy) decreases as the weight is increased. The maximal tolerated false positive rate of 1% (0.987% to be exact) was achieved when the weight was set to 3.8, with true positive rate of 25.82%, the accuracy was 61.73%. The average learning time was 12.3 seconds.

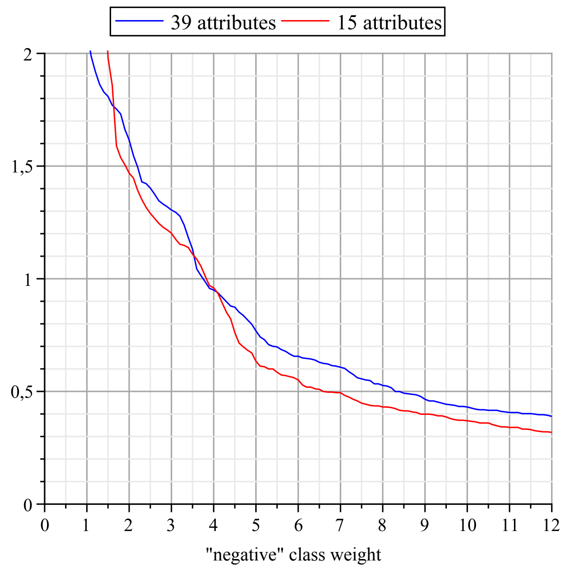
When only 15 attributes were used, the accuracy, TPR and FPR did not change much. The required false positive rate was achieved when the weight was set to 3.9 when the FPR was 0.970%. True positive rate and accuracy was a bit higher than in the previous case, 62.0% (+0.27%) and 26.38% (+0.56%) respectively, however, according to the Student’s *t*-test[5], these differences are not statistically significant. The average learning time was 5.8 seconds, approximately one half compared to the previous experiment.

Although the accuracy is only by less than one percentage point (+0.45 or +0.72 points, depending on the number of attributes used) better than in the case of the currently used method (scores and threshold), the main advantage is that it is not necessary to manually determine the scores of each of monitored actions (or attributes). It also allows to quickly bias the classifier towards positive or negative classification, as required.

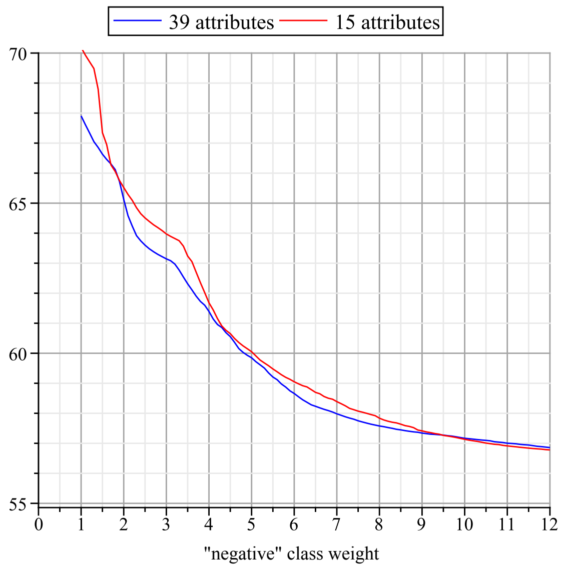
[5]See section 2.3 for description.



(a) True positive rate (%)



(b) False positive rate (%)



(c) Accuracy (%)

Figure 6.1. The relationship between the TPR, FPR and accuracy on the weight of the "negative" class.

Chapter 7

Conclusions

This thesis describes the implementation of a classification tool which could be used for detecting malicious software. The topic was proposed by an anti-virus company *AVG Technologies* which also provided the data set in a form of reports created by analysing various executable files in a virtual environment. The complete database consists of millions of samples, however only part of those were analysed, due to time expensiveness.

First, 39 features (attributes) were extracted from the reports. It turned out that about half of the samples from malicious set did not perform any of the monitored Windows API calls, thus the reports were empty. It is possible that these executables are able to detect that they are running in a virtual environment and suspend any behaviour which could be considered as malicious. Other option is that although they performed their payload, it consists of different set of Windows API calls then those monitored. Although it is very difficult to overcome the first issue (by using some kind of more advanced virtual environment or emulator), the former problem can be eliminated by monitoring larger number of API calls. Since this would means larger number of attributes in the data set, it would also lead to longer learning time. Furthermore, the significance of each feature was estimated using information gain measure included in Weka toolbox. This showed, that some of them are not used in almost all samples, however this might not be true for the whole database of executables.

Selected classification methods were then trained and evaluated using 20-fold cross validation. The classifiers used were: Naïve Bayes, decision trees (using four different maximal depth settings), the RandomForrest algorithm, a support vector machine and neural nets (using four different structures). One decision tree and one neural net was trained alse "boosted" using the AdaBoost algorithm. All experiments were performed using RapidMiner data mining software, using built-in implementations of these classifiers and their default settings.

All of the the used classifiers had better accuracy than the method currently used by the company. The best performing classifier, in terms of the accuracy, was the neural net. The experiments reveal that the net structure does not affect the performance significantly in this task (all of them had accuracy around 94%), however it does affect training time. The second best method was the decision tree induction. Although it is a little less accurate (92.9% in average), the training is about 24 times faster than in the case of the largest neural net. However, there are faster neural net training methods than the used back-propagation.

Because the achieved false positive rate was higher than the maximum the company is willing to tolerate (1%), all the classifiers were trained again, together with the MetaCost method, which allows to specify cost of the false positive classification. It turned out that when the cost is set so the FPR reaches value around 1%, the accuracy of all the classifiers does not differ as much as in the first experiments. It varied from 56.04% in the case of the RandomForrest algorithm, to 62.51% in the case of the support vector machine, however the differences may be caused by the fact, that the classifiers were not set to achieve the exactly same value of the false positive rate. The main difference between the classifiers was in the time necessary for learning and for this reason the Naïve Bayes classifier (with average training time under 1 minute) was selected for implementation.

The implemented application covers all necessary steps – attribute extraction from sandbox logs, training of the Naïve Bayes classifier, estimation of its performance using *k*-fold cross-validation, and classification of unknown samples. It also supports assigning weights to records of one or another class label which allows to fine-tune the resulting true and false positive rates.

The tests showed that when the classifier is trained so it achieves the false positive rate lower than 1%, its performance is only slightly better than the accuracy of the currently used method. On the other hand, the current solution needed to manually specify scores for each of the monitored actions and also the maximal score which can be achieved to consider unknown file as benign, which complicates addition of new attributes and/or samples to the database. The implemented application does not have this limitation – the classifier is trained fully automatically and within just couple of minutes. In the hypothetical situation when the false positive rate exceeds the tolerated maximum after new executables (or attributes) were added to the data set, it is possible to quickly re-adjust the classifier without the need of an expert (who would manually assign new scores to the attributes).

Although the implemented classifier is very fast to train – in average around 12 seconds for training set containing 90 thousand examples with 39 attributes – the necessary learning time can be shortened by omitting attributes which are less significant for the classification, however the application does not support either estimation of the attributes' significance or reduction of the attribute set before training. On the other hand however there are many existing tools which could be used for data set analysis (for example Weka or RapidMiner used in this thesis) which provide almost anything one would need – estimation of the attributes' significance, tools for identifying correlated attributes, data visualisations and much more.

Bibliography

[1] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan. N-gram-based detection of new malicious code. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, volume 2, pages 41 – 42 vol.2, Sept. 2004.

[2] Jianguo Ding, Jian Jin, P. Bouvry, Yongtao Hu, and Haibing Guan. Behavior-based proactive detection of unknown malicious codes. In *Internet Monitoring and Protection, 2009. ICIMP '09. Fourth International Conference on*, pages 72 –77, May 2009.

[3] P. Domingos. Metacost: A general method for making classifiers cost-sensitive. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 155–164. ACM, 1999.

[4] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting, 1995.

[5] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*, chapter 8.2.2, page 344. Morgan Kaufmann, 3rd edition, 2011.

[6] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 3rd edition, 2011.

[7] O. Henchiri and N. Japkowicz. A feature selection and evaluation scheme for computer virus detection. In *Data Mining, 2006. ICDM '06. Sixth International Conference on*, pages 891 –895, Dec. 2006.

[8] J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, December 2006.

[9] Python Software Foundation. „About Python“. *Python Programming Language – Official Website*. Web. 20 Jul. 2012. http://python.org/about/.

[10] J. Ross Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[11] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[12] RapidMiner. Rapid-I GmbH. http://rapid-i.com/, 2001-2012.

[13] M.G. Schultz, E. Eskin, F. Zadok, and S.J. Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38 –49, 2001.

[14] Weka. The University of Waikato. http://www.cs.waikato.ac.nz/ml/weka/, 2002-2012.

[15] Ian H. Witten and Eibe Frank. *Data Mining*. Morgan Kaufmann, 2nd edition, 2005.

Appendix A

List of extracted attributes

Gain ratio	Name
Action	
0.2388	WINSOCK_STARTUP
Initializing network sockets	
0.1628	REG_SET_RUN
Writing to the list of programs run on OS startup (CurrentVersion\Run* registry keys)	
0.1511	CONNECT
Connecting to remote computer	
0.1427	REG_SET_WINLOGON
Writing to the list of programs run on OS startup (CurrentVersion\WinLogon registry keys)	
0.1414	RUN_OTHER_FILE
Executing file not created nor modified by the monitored process	
0.1413	RUN_CREATED_FILE
Executing file previously created or modified by monitored process	
0.1369	COPY_FILE_WIN
Copying a file into C:\Windows directory	
0.1298	COPY_FILE_OTHER
Copying a file (not into C:\Windows directory)	
0.118	REG_SET_EXPLORER_RUN
Writing to the list of programs run on OS startup (Policies\Explorer\Run registry keys)	
0.1136	DNS_QUERY
Sending query to DNS server	
0.112	DELETE_OTHER_FILE
Deleting file not created nor modified by the monitored process	
0.1117	API_HOOK

Calling API hook function	
0.1099	WRITE_FILE_EXEC
Writing to exe, dll, sys or com file	
0.1094	NET_SEND
Sending data over network	
0.1017	FIREWALL_SET_EXCEPTION
Creating exception in Windows Firewall settings	
0.0974	MOVE_FILE_OTHER
Moving file to a location different than C:\Windows	
0.0946	DOWNLOAD
Downloading a file from network using the HTTP protocol	
0.0935	WRITE_FILE_OTHER
Writing to file which is neither executable, located in C:\Windows nor the HOSTS file	
0.0933	REG_SET_WINDOWS
Writing to CurrentVersion\Windows registry key	
0.0926	FIREWALL_ENABLE_EXCEPTIONS
Enabling exceptions in Windows Firewall	
0.0862	WRITE_FILE_WIN
Writing to file located in C:\Windows	
0.0845	DELETE_CREATED_FILE
Deleting file previously created or modified by monitored process	
0.0801	CREATE_CREATED_SERVICE
Registering a service driver using file previously created or modified by the monitored process	
0.0778	REG_SET_SAFEBOOT
Writing to System\CurrentControlSet \Control\Safeboot registry key	
0.0759	EMAIL
Sending an email	
0.0651	WRITE_FILE_HOSTS
Writing to HOSTS file (list of IP addresses for selected domain names, it has larger priority than Domain Name System, DNS)	
0.0641	OPEN_PROCESS
Opening another process for writing	
0.0641	CREATE_OTHER_SERVICE
Registering a service using file previously created nor modified by monitored process	
0.0623	IMAGE_LOAD
Calling undocumented IMAGE_LOAD Windows API function	

0.0613	LOAD_OTHER_DRIVER
Loading a kernel service driver using file not created nor modified by monitored process	
0.0613	WIN_OR_SYSTEM_INI
Writing to win.ini or system.ini file	
0.0588	MOVE_FILE_WIN
Moving file into C:\Windows directory	
0.0522	LISTEN
Starting listening for remote network connection	
0.0358	DELETE_SERVICE
Unregistering a service	
0.035	CREATE_OREANS32_SERVICE
Registering OREANS32 service (driver used for protecting some games)	
0.0228	REG_SET_SHELL_OPEN
Creating file type association with certain application	
0	UNLOAD_DRIVER
Unloading a kernel service driver	
0	HIDE_PROCESS
Calling HIDE_PROCESS Windows API function	
0	LOAD_CREATED_DRIVER
Loading a kernel service driver using file previously created or modified by the monitored process	

Appendix B

Application user’s manual

B.1 File formats

B.1.1 CSV files

CSV files are used for storing sandbox logs database, data sets and the resulting model. The first line always contains the column headers, other lines contains data rows. The column names are case-sensitive. Although the application can process any type of line endings, it is preferred to use Unix style line endings (`\n`).

Fields are separated by comma (`,`, no spaces are allowed) and they can be enclosed in double quotes (`"`). Fields can span over multiple rows, in which case the line endings are not escaped. If double quotes character is part of field contents, it must be escaped by inserting it twice (`""`). Comments are not supported. There are many other dialects of CSV files which differ in the delimeter and escaping characters but they are not supported.

Sandbox logs database

The CSV files supplied to attribute extractor must contain at least two columns. The first column contains the record’s ID and the second column contains the log itself. The IDs are not checked for uniqueness within the file. Their names are left to user’s decision, only the order matters. If the file contains more than two columns, the rest is ignored and it does not raise an error.

Listing B.1. Example of a logs database file

```
md5,log,auxiliary
FIRST_ENTRY_ID,"first entry log line 1
first entry log line 2 with some ""quoted string""
first entry log line 3",auxiliary value
SECOND_ENTRY_ID,"second entry log line 1
second entry log line 2",auxiliary value
```

Data sets

The data set files generated and used by the application consist from at least three columns. The first column is the record’s ID and the last column contains its class label. All columns between contains the attributes’ values. The name of the ID and class label columns are arbitrary. Attributes' values must be valid floating point number literals. Nominal attributes and missing values are not supported.

Listing B.2. Example of a data set file

```
id,ATTRIBUTE_A,ATTRIBUTE_B,label
FIRST_ENTRY_ID,1.0003,4e4,positive
SECOND_ENTRY_ID,0,0,negative
```

Models

The model file format is slightly modified compared to the previous file formats. It is divided into two parts, each introduced by its own header.

The first part contains the class labels' prior probabilities. It contains two columns, named `label` and `prior`, followed by values for each available label. The second part contains calculated values for each combination of class label and attribute. It contains five columns, named `label`, `attribute`, `mean`, `stddev` and `log_factor`[1].

Listing B.3. Example of a model file

```
label,prior
positive,-0.6744398349152223
negative,-0.7122111735094135
label,attribute,mean,stddev,log_factor
positive,ATTRIBUTE_A,0.27,3.05,2.03
positive,ATTRIBUTE_B,2.12e-05,0.0,-4.46
negative,ATTRIBUTE_A,0.03,0.75,0.64
negative,ATTRIBUTE_B,0.0,0.001,-5.99
```

B.1.2 INI files – single record files

INI files are used to store attribute values of a single extracted record. They can be generated by `extract_single.py` from sandbox logs and they can used as an input for the `classify_single.py` script.

Although the INI file format is not well defined, the application follows commonly used version, however with some limitations – for example comments are not supported. The first line contains the record's ID enclosed in square brackets (similar to INI section declaration). It is followed by attributes and their values, each on separate line. The attribute name and its value are separated by single = character. There is one special attribute, `___label___`, which contains the record's correct class label.

The application supports reading only INI files containing one section (one record). The last section declaration found denotes the read ID and if any attribute is present more than once, only the last value is used. Values must be valid floating point literals, nominal attributes and missing values are not supported (an error would be raised).

Listing B.4. Example of a INI single record file

```
[RECORD_ID]
___label___=positive
ATTRIBUTE_A=1.0003
ATTRIBUTE_B=4e-4
```

[1]The value of the term $\log(\sqrt{2\pi}\sigma_i)$ from equation 5.3

B.2 Usage of executable scripts

B.2.1 Attribute extraction

```
extract.py [-h] [-o FILE] [-l LABEL] [-s] [input]
```

Extracts attributes from sandbox logs stored in a CSV file and returns results as another CSV file.

`-o FILE`
Write output CSV to `FILE`. If not specified, the result is written to the standard output.

`-l LABEL`
The class label to store for all records. If not specified, the input file name is used. Required if the input file is not specified.

`-s`
Skip records with null attribute vector. These records will not be written to the output

`input`
Input CSV file name. If not specified, the standard input is used.

```
extract_single.py [-h] [-a CSV_FILE] [-d ID] [-l LABEL] [-s] [input]
```

Extracts attributes from single sandbox log and returns results as a INI file or appends it to an existing CSV file as a new row.

`-a CSV_FILE`
Append output to `CSV_FILE` as a new row. If not specified, the result is written to the standard output as a INI file.

`-d ID`
The record's ID. If not specified, the input file name or `stdin` is used.

`-l LABEL`
The class label to store. If not specified, `unknown` is used.

`-s`
If used together with the `-a` option, if the extracted record has null attribute vector, it will not be written to the CSV file.

`input`
Input file name. If not specified, the standard input is used.

B.2.2 Training

```
train.py [-h] [-o FILE] [-w [label:weight [label:weight ...]]]
[input [input ...]]
```

Trains a Naive Bayes classifier from provided training set. The training set can be loaded from multiple files at once.

`-o FILE`
Write output model to `FILE`. If not specified, the result is written to the standard output.

`-w label:weight`
The weight of records having specified class label. This option can be repeated to specify weights for more than one class label. If the label does not exist, the option is ignored. It is not necessary to specify weights for all labels. The weights for non-specified labels are set to 1.

`input`
Input CSV file name(s). If not specified, the standard input is used.

B.2.3 Splitting dataset into subsets

```
subsets.py [-h] -n N [-o DIR] [input [input ...]]
```

Splits provided data set into n subsets. Each subset will contain approximately the same number of records and the ratio between classes in subsets are will be kept the same as in the full data set (stratified sampling). The resulting subsets are saved to files named `subset_X.csv`, where X is number of the subset, starting from 1.

`-n N`
Number of subsets. Required.

`-o DIR`
Output directory name. If it does not exist, it is created automatically. If not specified, current working directory is used.

`input`
Input CSV file name(s). If not specified, the standard input is used.

B.2.4 Performance estimation

```
xvalidation.py [-h] [-k FOLDS] [-p] [-d] [-w [label:weight [label:weight ...]]]
[input [input ...]]
```

Estimates Naive Bayes model performance using k-fold cross-validation. The training set can be loaded from multiple files at once. Commented example of the output is shown in listing B.5.

`-k FOLDS`
Number of folds. Default value is 20.

`-p`
If present, each input file is treated as an subset created from the training set, prepared for the cross-validation. The number of folds is determined by the number of input files and the option `-f` is ignored.

`-d`
If present, prints also detailed results for each fold separately.

`-w label:weight`
The weight of records having specified class label. This option can be repeated to specify weights for more than one class label. If the label does not exist, the option is ignored. It is not necessary to specify weights for all labels. The weights for non-specified labels are set to 1.

`input`
Input CSV file name(s). If not specified, the standard input is used.

Listing B.5. Example output of the cross-validation process.

```
Total number of classified samples: 92515
Average learning time: 23.659 secs
Average testing time: 18.765 secs (0.001 secs per record)
Correctly classified: 62918
Misclassified: 29597
Accuracy: 68.0084 %
------
Total CLASS_A count: 47131      Number of CLASS_A records
True CLASS_A count: 18519       Number of correctly classified CLASS_A records
True CLASS_A rate: 40.8051 %    True CLASS_A count / Total CLASS_A count
False CLASS_A count: 985        Number of CLASS_B records classified as CLASS_A
False CLASS_A rate: 2.1704 %    False CLASS_A count / Total CLASS_B count
------
Total CLASS_B count: 45384      Number of CLASS_B records
True CLASS_B count: 44399       Number of correctly classified CLASS_B records
True CLASS_B rate: 94.2034 %    True CLASS_B count / Total CLASS_B count
False CLASS_B count: 28612      Number of CLASS_A records classified as CLASS_B
False CLASS_B rate: 60.7074 %   False CLASS_B count / Total CLASS_A count
------
This part is printed only with the -d option.
k,T(CLASS_A),F(CLASS_A),T(CLASS_B),F(CLASS_B),learn_time,test_time
0,3797,222,8855,5630,18.995000,15.946000
1,3629,199,8878,5797,22.677000,27.704000
2,3842,207,8870,5584,38.670000,18.743000
3,3587,173,8904,5839,19.054000,15.031000
4,3664,184,8892,5762,18.897000,16.403000
```

B.2.5 Classification of unknown samples

`classify.py [-h] -m MODEL [-o FILE] [-a] [-f] [input [input ...]]`

Classifies many unknown samples at once. The output is in the CSV format, containing these columns:

- By default: `id`, `label`
- With the `-p` option: `id`, `probability(CLASS_A)`, `probability(CLASS_B)`, . . . , `label`
- With the `-a` option: `id`, `ATTRIBUTE_A`, `ATTRIBUTE_B`, . . . , `label`

- With both `-p` and `-a` options: `id`, `ATTRIBUTE_A`, `ATTRIBUTE_B`, ..., `probability(CLASS_A)`, `probability(CLASS_B)`, ..., `label`

`-m MODEL`
Model file, previously generated by `train.py`.

`-o FILE`
Write output CSV to `FILE`. If not specified, the result is written to the standard output.

`-p`
Include all calculated probabilities in the output.

`-a`
Include attribute values in the output.

`input`
Input CSV file name. If not specified, the standard input is used.

```
classify_single.py [-h] -m MODEL [-e] [-p] [FILE]
```

Classifies many unknown samples at once. The input can be either in INI format or a raw log. The output is in the INI format. The `___label___` attribute is replaced by the most probable class.

`-m MODEL`
Model file, previously generated by `train.py`.

`-e`
The input is a raw log file, extract attributes first.

`-p`
Include all calculated probabilities in the output. They are added as attributes named `probability(CLASS_NAME)` at the end of the file.

`input`
Input file name. If not specified, the standard input is used.

Appendix C

CD Contents

The CD contains following directories:

- `thesis` – this thesis source code in the LaTeX format
- `pdf` – this thesis in the PDF format
- `implementation` – the implemented application
- `sample_data` – sample input data
- `results` – results of all performed experiments in OpenDocument Spreadsheet format