Need for Speed: Analysis of Brazilian Malware Classifiers' Expiration Date

Thesis · February 2018

DOI: 10.13140/RG.2.2.16132.83840

3 authors:

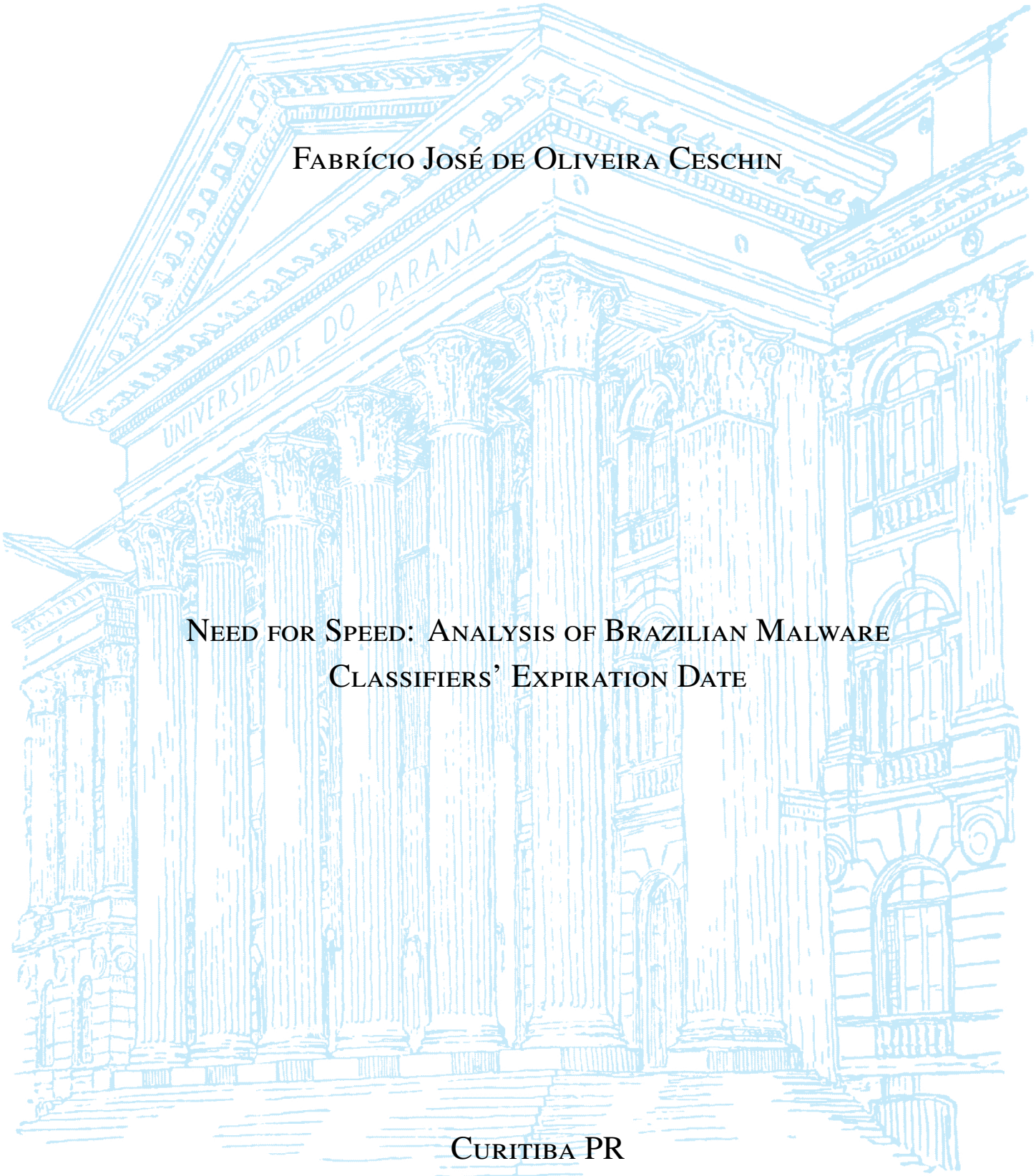
Fabrício Ceschin
Universidade Federal do Paraná
5 PUBLICATIONS **1** CITATION
SEE PROFILE

André Ricardo Abed Grégio
Universidade Federal do Paraná
56 PUBLICATIONS **223** CITATIONS
SEE PROFILE

David Menotti
Universidade Federal do Paraná
125 PUBLICATIONS **1,342** CITATIONS
SEE PROFILE

Universidade Federal do Paraná

Fabrício José de Oliveira Ceschin



Need for Speed: Analysis of Brazilian Malware Classifiers' Expiration Date

Curitiba PR
2018

Fabrício José de Oliveira Ceschin

Need for Speed: Analysis of Brazilian Malware Classifiers' Expiration Date

Tesis presented as partial requirement for the degree of Master. Graduate Program in Informatics, Sector of Exact Sciences, Universidade Federal do Paraná.

Field: *Computer Science*.

Supervisor: André Ricardo Abed Grégio.

Co-supervisor: David Menotti Gomes.

Curitiba PR
2018

C421n Ceschin, Fabrício José de Oliveira
Need for Speed: Analysis of Brazilian Malware Classifiers' Expiration Date / Fabrício José de Oliveira Ceschin. – Curitiba, 2018 .
70 f. : il. color. ; 30 cm.

Dissertação - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática, 2018 .

Orientador: André Ricardo Abed Grégio.
Coorientador: David Menotti Gomes.

1. Ciência da computação. 2. Classificação de programas. 3. Identificação de malware. 4. Aprendizado de máquina. I. Universidade Federal do Paraná. II. Grégio, André Ricardo Abed. III. Gomes, David Menotti. IV. Título.

CDD: 005.2

MINISTÉRIO DA EDUCAÇÃO
SETOR CIÊNCIAS EXATAS
UNIVERSIDADE FEDERAL DO PARANÁ
PRÓ-REITORIA DE PESQUISA E PÓS-GRADUAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO INFORMÁTICA

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da Dissertação de Mestrado de **FABRÍCIO JOSÉ DE OLIVEIRA CESCHIN** intitulada: **Need for Speed: Analysis of Brazilian Malware Classifiers' Expiration Date**, após terem inquirido o aluno e realizado a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de mestre está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

Curitiba, 27 de Fevereiro de 2018.

ANDRÉ RICARDO ABED GRÉGIO
Presidente da Banca Examinadora (UFPR)

ALTAIR OLIVO SANTIN
Avaliador Externo (PUC/PR)

FELIPE AZEVEDO PINAGÉ
Avaliador Externo (UFPR)

DAVID MENOTTI GOMES
Co-orientador - Avaliador Interno (UFPR)

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DO PARANÁ
Setor de Ciências Exatas
Coordenação de Pós-Graduação em Informática

Acknowledgments

This work was supported by the International Cooperation Program CAPES at Federal University of Paraná. Financed by CAPES – Brazilian Federal Agency for Support and Evaluation of Graduate Education within the Ministry of Education of Brazil. Also, we want to thanks Google for supporting the continuation of this work (PhD project) thought the program Google LARA - Latin America Research Award 2017.

"If you want to be successful, you need to have total dedication, get your last limit and give the best of you."Ayrton Senna

Resumo

Novos programas maliciosos são criados e liberados diariamente para enganar usuários e superar soluções de segurança, assim exigindo melhora continua nestes mecanismos (por exemplo, atualização constante de antivirus). Apesar da maioria dos programas maliciosos serem "genéricos"o suficiente para infectar o mesmo tipo de sistema operacional mundialmente, alguns deles estão relacionados às especificidades de um ciberespaço de certos países alvos. Neste trabalho, nós apresentemos uma análise de milhares de exemplares de *malware* coletados no ciberespaço brasileiro ao longo de vários anos, incluindo suas evoluções e o impacto dessas evoluções na classificação de *malware*. Nós também disponibilizamos um *dataset* desse conjunto de *malware* para permitir que outros experimentos e comparações sejam feitas pela comunidade. Este *dataset* representa o ciberespaço brasileiro e contém perfis de programas que são conhecidamente malignos e benignos, baseados em características estáticas de seus binários. Nossa análise utilizou algoritmos de aprendizado de máquina (em particular, nós avaliamos quatro algoritmos populares *off-the-shelf*: *Support Vector Machines*, *Multilayer Perceptron*, *KNN* e *Random Forest*) para classificar os programas do nosso *dataset* como maligno ou benigno (incluindo experimentos com *thresholds*) e identificar o potencial *concept drift* que ocorre quando o modelo de classificação evolui com o passar do tempo. Nós também providenciamos detalhes extensos sobre nosso *dataset*, que é composto por 38.000 programas – 20.000 rotulados como malignos, coletados de anexos de *e-mails* maliciosos/usuários infectados (coletados em ambos os casos por uma grande instituição financeira brasileira com uma rede distribuída em todo o pais entre 2013 e começo de 2017. Por uma questão de reprodutibilidade e comparação imparcial, nós disponibilizamos publicamente os vetores de características utilizados. Finalmente, nós discutimos os experimentos conduzimos, cuja análise evidencia a existência de concept drift nos programas, tanto benignos como malignos, e mostra que não é possível dizer que existe sasonalidade em nosso *dataset*.

Palavras-chave: Classificação de programas, Identificação de *malware*, Aprendizado de máquina, *Concept drift*.

Abstract

New malware variants are produced and released daily to deceive users and overcome defense solutions, thus demanding continuous improvements on these mechanisms (e.g., antiviruses constant updates). Although most malware samples are usually "generic" enough to infect the same type of operating system world-widely, some of them are tied to the specificities regarding the cyberspace of certain target countries. In this work, we present an analysis of thousands of malware samples collected in the Brazilian cyberspace along several years, including their evolution and the impact of this evolution on malware classification. We also share a labeled dataset of this Brazilian malware set to allow other experiments and comparisons by the community. This dataset is representative of the Brazilian cyberspace and contains profiles of known-bad and known-good programs based on binaries' static features. Our analysis leveraged machine learning algorithms (in particular, we evaluated four popular off-the-shelf classifiers: Support Vector Machines, Multilayer Perceptron, KNN and Random Forest) to classify the programs of our dataset as malware or goodware (including experiments with thresholds) and to identify the potential concept drift that occurs when the subject of a classification scheme evolves as time goes by. We also provide extensive details about our dataset, which is composed of $38,000$ programs – $20,000$ labeled as known malware, collected from malicious email attachments/infected users (triaged in both cases by a major Brazilian financial institution with a country-wide distributed network) between 2013 and early 2017. For the sake of reproducibility and unbiased comparison, we make the feature vectors produced from our database publicly available. Finally, we discuss the results of the conducted experiments, whose analysis evidences the existence of concept drift on programs, either goodware and malware, and shows that it is not possible to say that there is seasonality in our dataset.

Keywords: Program classification, Malware identification, Machine learning, Concept drift.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Acrônimos

VSM	Vector Space Model
KNN	*k*-nearest neighbors
MLP	Multilayer Perceptron
SVM	Support Vector Machines
t-SNE	*t*-Distributed Stochastic Neighbor Embedding
FPR	False Positive Rate
TPR	True Positive Rate

Lista de Símbolos

φ	Activation function of a neural network

Capítulo 1

Introduction

As time goes by, the threat posed by malicious codes has not been diminishing. Malware attacks become more serious – persistent, stealth, multi-targeted – and, apart from the fact that users are massively migrating to mobile platforms, still continue to infect desktop systems. In 2016, for example, the number of malware sofware increased from 5.14 millions (2015) to 6.83 millions and, in 2017, the prevision is that it exceeds the mark of 7 millions, as presented in Figure 1.1 Benzmüller (2017). This situation makes it difficult for researchers and security companies to find a suitable solution that handles the effectiveness of defense mechanisms and the complexity, sophistication, protection tools' evasiveness, and spreading by several means of malicious actors. In addiction, the uncountable amount of modern malware (variants) released daily brings the urgent need to create automated classifiers/detectors. In general, malware detection through machine learning techniques is based on extracting program characteristics in a way that they can be classified as malicious or benign, or even grouping the malicious into distinct families, depending of the attributes and datasets used.

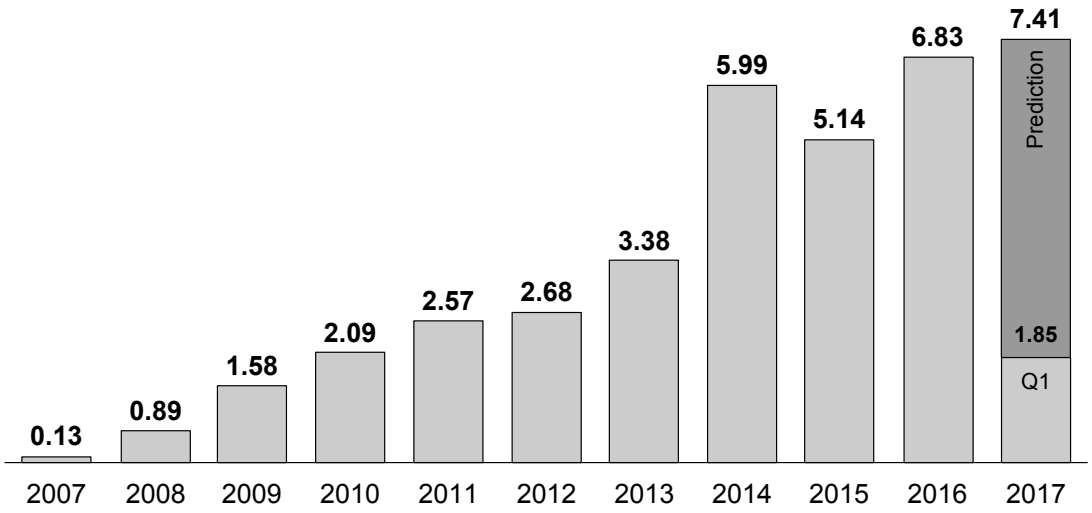


Figura 1.1: Number of malware released by year (in millions) Benzmüller (2017).

There are numerous approaches in the literature for malware classification, clustering and detection, whose premise is the use of machine learning Huang e Stokes (2016); David e Netanyahu (2015). Most of the published articles in the area presents problems of methodology and/or reproducibility, either because they do not provide the data, scripts and programs developed for samples classification and results analysis or because of the lack of representativity/transparency

in samples selection (many of these problems are described by Rossow et al. Rossow et al. (2012) in an observation of related work from 2006 to 2011). Moreover, the focus observed in the literature is in obtaining accuracy rates close to 100%, without caring about the expiration time of the produced machine learning solutions. Therefore, it can not be guaranteed that the results generated are completely exempt from bias caused by the chosen dataset, since there is no discussion about the effectiveness of the classifiers over time, nor available data for new tests by other researchers.

In this work, first, we leverage a comparison among some classic machine learning algorithms, such as Support Vector Machines (SVM), Multi-Layer Perceptron (MLP), k-Nearest Neighbors, and Random Forest. Our focus is on the problem of classification of programs into malicious or benign from a set of representative, unfiltered samples collected in Brazilian cyberspace from 2013 to 2017. Second, based on the produced results using traditional approaches (using cross-validation and thresholds), we provide a critical analysis of the expiration date of these type of classifiers, identifying when they lose effectiveness due to the malware samples evolution. This is accomplished by two experiments, which consist in training (learning) and testing (evaluation) the generated month by month, simulating a real world situation and checking the representativeness of the collected samples by month. Also, we show that it is not possible to say that there is seasonality in malware samples.

Besides that, we fill a gap in the machine learning area related to security by making publicly available the labeled attributes vector from the 38,490 programs used here (19,979 malicious and 18,511 benign), as well as the meta-information used to produce them along with all the scripts developed for extraction, processing, and analysis of data and results. In addition to the confirmation of the hypothesis on the expiration date of "malware identifiers", another contribution of our work is to highlight the positive influence that updated goodware samples can offer to the process of programs classification.

The remaining of this document is organized as follows: in Chapter 2, we present the theoretical foundation and other work related with malware classification/detection using machine learning; the dataset we used in this work (the descriptor vectors are available in Ceschin et al. (2016)), as well as the methodology used to collect them, is described in Chapter 3; we analyze the collected data in Chapter 4; we present the experiments performed, as well as the classifiers expiration analysis over time and seasonality study, in Chapter 5. Finally, we make some final considerations on this work in Chapter 6.

Capítulo 2

Background and Related Work

This Chapter introduces the background and the related work. The Section 2.1 shows the background, with all the theoretical foundation related, and Section 2.2, the related works.

2.1 Background

In this Section we present all the theoretical foundation about the topics used in this work, related to two areas of computer science: machine learning and security.

2.1.1 Information Retrieval

According to Manning et al. (2008b), information retrieval (IR) is finding material of an unstructured nature that satisfies an information need from within large collections. Briefly, in our case, it finds ways to transform unstructured data (data which do not have clear, semantically overt, easy-for-a-computer structure) into structured, converting texts to features that can be used as input of a classifier, for example. The following Subsections aim to present the steps involving information retrieval applied to texts.

Case Folding and Normalization

The first step in preprocessing texts is case folding, a way to keep all texts in lower case to maintain a standard for comparison. After that, the normalization aims to remove any special characters, accent, marks and numbers. This last step can follow several rules, such as replacing symbols and numbers with their names, for example Manning et al. (2008b).

Stop Words Removal

After normalizing the text, it's necessary to remove the stop words, words that do not have essential meanings in a text, usually the most common words in a language. With that, they are excluded from the vocabulary entirely, since they do not contribute to differ them Manning et al. (2008b).

Bag of Words

Bag of words is a representation used in natural language processing and information retrieval that counts the number of occurrences of each term, usually used in documents classification Manning et al. (2008b). Despite being a structured data, the bag of words model is

not enough to differ texts for classification tasks. This representation, in this case, is only used to compute both terms used in Vector Space Model, presented in the next Section.

Vector Space Model

Vector Space Model (VSM) is fundamental to a host of information retrieval operations, representing a text through the relative importance of its words Manning et al. (2008b); Salton et al. (1975). Given a vocabulary from a set of documents (texts), i.e., every word that appears in this set, every document i is represented by a vector $\vec{d_i} = (w_{i,1}, w_{i,2}, ..., w_{i,t})$, where $w_{i,j}$ represents the *TF-IDF* (Term Frequency – Inverse Document Frequency) of the j^{th} word in the vocabulary. The *TF-IDF* is a statistic measure used to evaluate how important a word is to a document in relation to a collection of documents Manning et al. (2008a). This measure can be obtained using the bag of words representation, cited in the previous Section, through the multiplication of two terms, TF and IDF, i.e., $TFIDF(t) = TF(t) \times IDF(t)$, where:

- *Term Frequency (TF)*, which measures how often a word/term t occurs in a text/document, as shown in Equation 2.1 below:

$$TF(t) = \frac{\text{Number of times that } t \text{ appears in the document}}{\text{Total number of words in the document}} \tag{2.1}$$

- *Inverse Document Frequency (IDF)*, which measures how important a term t is, as shown in Equation 2.2 below:

$$IDF(t) = \log_e \left(\frac{\text{Total number of documents}}{\text{Number of files that contains the word } t} \right) \tag{2.2}$$

In a nutshell, each text/document is represented by a sparse vector that contains their *TF-IDF* measures for each word in the vocabulary. The vocabulary can be reduced to a number V of words that are most present in the given documents or texts.

2.1.2 Features Normalization

As there are differences in the characteristics scale, there is a need of normalizing them. The normalization technique applied on the extracted features of this work was the *MinMax*. This technique scales every feature (characteristic) into an interval between zero and one, using the formula seen on Equation 2.3.

$$z_i = \frac{x_i - min(x)}{max(x) - min(x)} \tag{2.3}$$

In this equation, x_i is the value of the feature x from a file i, $min(x)$ is the minimum value of that feature in the training set, and $max(x)$ is the maximum value of this feature in the same set Pedregosa et al. (2011).

2.1.3 Classification

There are two types of classification problems: unsupervised learning and supervised learning. The first one (unsupervised) clusters similar samples, without necessarily knowing the number of classes, and the second (supervised) aims to establish a rule whereby we can classify a new observation into one of the existing classes (that are known) Michie et al. (1994). Briefly, in supervised learning we have a collection of labeled known samples that will be used to train a

classifier which classifies an unknown sample into one of the existing classes. As in this work we used only supervised learning, we focus in this type of classification, explaining, in the next Sections, the classifiers used.

K-Nearest Neighbors

K-Nearest Neighbors (KNN) is a distance-based model, whose classification of a new instance is based in the distance of the k nearest training samples. Thus, a new unknown sample will be classified as being of the class that most occurred among these k samples, as shown in Figure 2.1 Michie et al. (1994), where the new instance will be classified as red, when $k = 3$, green, when $k = 5$ and unknown when $k = 6$ (the result is a draw, that's why an even number is not recommended for binary problems).

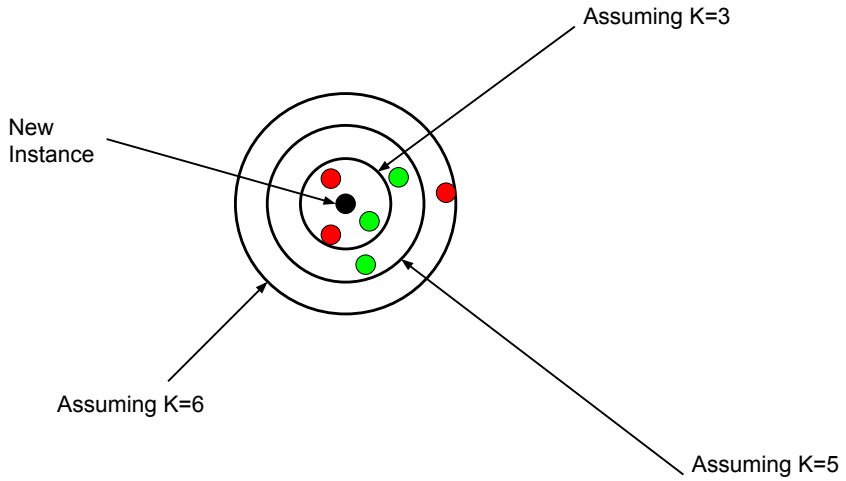


Figura 2.1: The class is computed from the k most closer neighbors.

The distance used by KNN is the Euclidean distance. Given a distance x, described by (a_1(x),a_2(x),...,a_n(x)), where a_i(x) is the i-th attribute, the distance between two instances x_i and x_j is defined by the Equation 2.4 Mellish (2017).

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n}(a_r(x_i) - a_r(x_j))^2)} \tag{2.4}$$

Multilayer Perceptron (MLP)

A neural network is a computational model of a human brain that can be composed of hundreds or thousands of neurons (processing units). In general, it's a machine that is designed to model the way in which the brain performs a particular task. The smallest component of a neural network is a perceptron, a processing unit that simulates a neuron Haykin (2009). Figure 2.2 shows the structure of a perceptron, where $x = \{x_1, x_2, ..., x_n\}$ are the signals (input), $w = \{w_1, w_2, ..., w_n\}$ are the weights, $\varphi(.)$ is the activation function and y is the output.

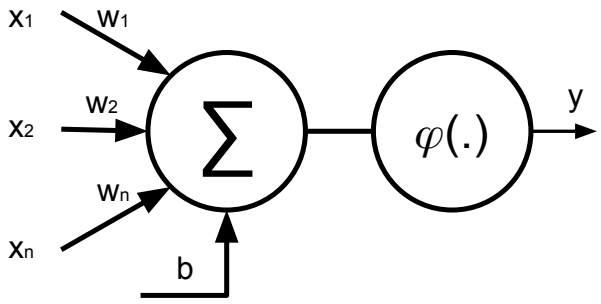


Figura 2.2: Behavior of a single perceptron, the smallest component of a neural network.

The Equation 2.5 presents the output (y) of a perceptron. The input signals are multiplied to the weights, that are adapted on an iteration-by-iteration basis, using an error-correction rule known as the perceptron convergence algorithm. It also includes an bias (b), which has the effect of increasing or lowering the net input of the activation function. The activation function is responsible for determining the shape and intensity of the values transmitted from one neuron to another (or to the output), limiting the amplitude of the output Haykin (2009).

$$y = \varphi(\sum_{i=1}^{n} w_i \times x_i + b) \tag{2.5}$$

The training of a single perceptron helps to understand better the operation of a neural network. The algorithm is the following Haykin (2009):

1. Initialize weights and bias with small random values.
2. Apply current sample input pattern and check the network output (y).
3. Calculate output error (e), comparing it (the output y) to the expected value (t_j), as shown in Equation 2.6.
$$e = t_j - y \tag{2.6}$$
4. If the output error is equal zero ($e = 0$), it means that the output is correct. In that case, a new sample is presented, going back to the step 2.
5. Otherwise, if the output error is different from zero ($e \neq 0$), it's necessary to update the weights and the bias, as shown in Equations 2.7 and 2.8, respectively.
$$w_j = w'_j + e \times x_j \tag{2.7}$$
$$b = b' + e \tag{2.8}$$
6. Go back to step 2 and present a new sample to the network. The stopping criteria can be based in iterations number, average error rate, accuracy, etc.

Despite of being efficient, a single perceptron can only solve linearly separable problems, which, in most of times, are not present in the real world, i.e., the presence of non-linear problems is very common in real problems. Due to that, neural networks usually combine more than one neuron, which makes it possible to them separate non-linear problems Haykin (2009). Figure 2.3 shows two examples of problems, the first one (left) linear and the second (right), non-linear. As

the decision boundary of the perceptron is defined by a line, it solves problems like the first one. A more complex network, such as multilayer perceptron, composed of multiple neurons, can solve the second one.

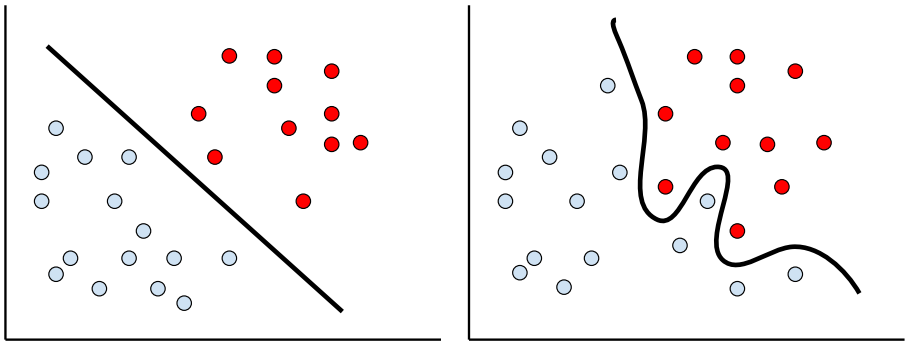


Figura 2.3: Linear (left) versus non-linear (right) problem.

A multilayer perceptron (MLP) neural network is composed by source nodes, that are the input, one layer or more of perceptrons, called hidden layers, and the output neurons. All the layers, except the input, are composed by neurons, each one of them initialized with random weights and bias. This type of network is progressive, i.e., the neurons of a certain layer are just connected to the next layer. Thus, the input passes through all existing layers. The number of input nodes is the dimensionality of the input data and the number of neurons in the output is generally composed by the number of classes of the problem (in this case, each neuron represents one class, i.e., the output value of the neuron is directly related to its respective class. The higher the value, the higher the chances of that sample being of that class) Haykin (2009). The Figure 2.4 presents a multilayer perceptron neural network with two hidden layers and three output neurons.

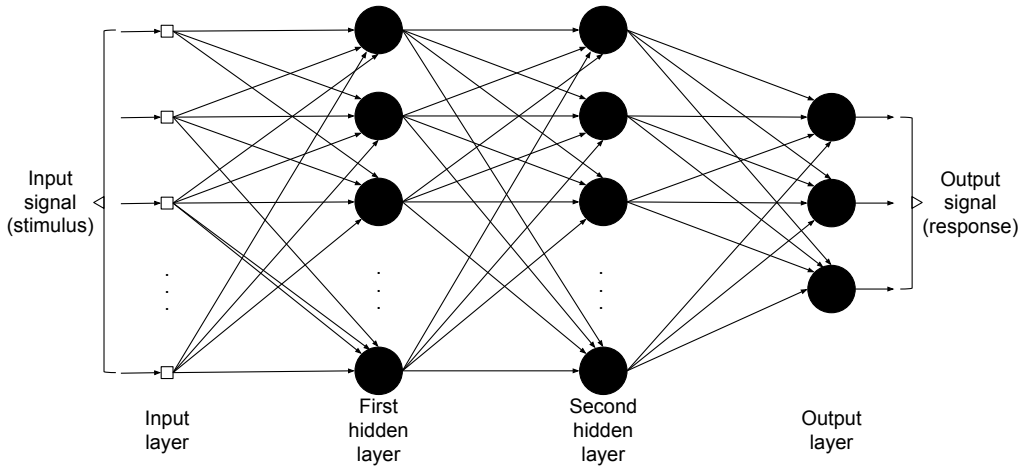


Figura 2.4: Multilayer perceptron neural network architecture with two hidden layers and three output neurons.

An important algorithm for neural networks in general is back propagation, a computationally efficient method for the training of multilayer perceptrons. Briefly, this algorithm implements gradient descent in weight space for a multilayer perceptron, efficiently computing partial derivatives of an approximating function $F(w, x)$ realized by the network, adjusting its weights according to the input Haykin (2009).

Support Vector Machine (SVM)

The Support Vector Machine (SVM), originaly developed for binary classification, finds, in linearly separable problems, the construction of a hyperplane as a decision surface (a boundary), such that the separation between the samples is maximal. When the patterns are non-linearly separable, the SVM finds a mapping function, which projects the data in a space where the data is linearly separable. The main idea of this classifier is to maximize the hyperplane margin from the training data. An optimal hyperplane is the one whose margin distance to the positive class is the same margin distance to the negative class. The Figure 2.5 illustrates an optimal hyperplane defined by the support vectors, the training samples most closer to it Cortes e Vapnik (1995); Fradkin e Muchnik (2006).

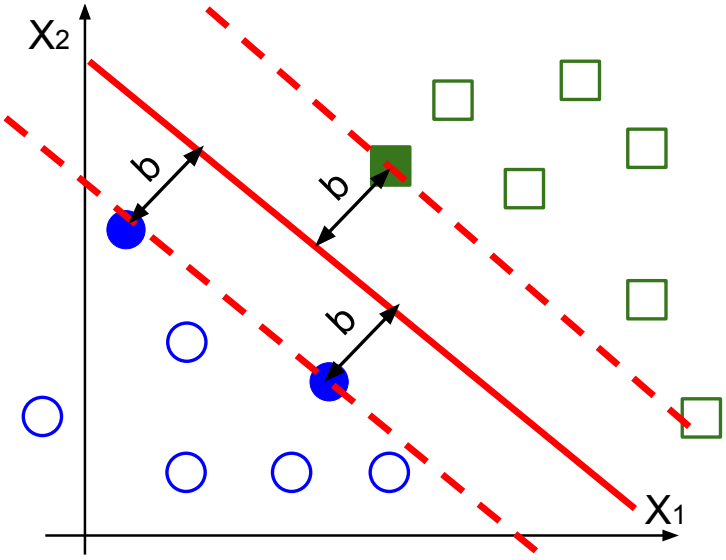


Figura 2.5: The train samples most closer to the hyperplane are called support vectors.

In most of times, the problems are not linearly separable. Due to that, it's necessary to project the data in a space where they are linearly separable, called feature space. The kernel function is responsible for this projection and this process is called kernel trick. After projected, it's possible to find a hyperplane that separates the data in that space. The Figure 2.6 exemplifies the use of the kernel trick to project the data in another dimension.

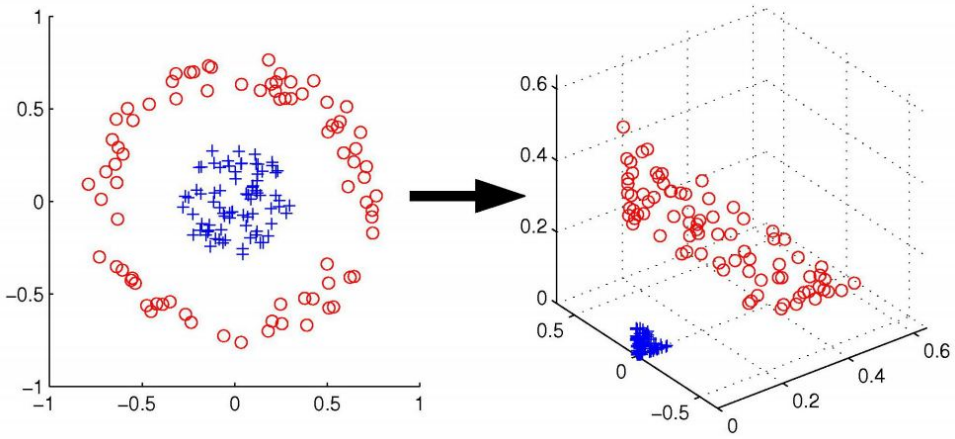


Figura 2.6: Projection sample of a non-linear separable problem, in which the data is projected in another dimension (bi-dimensional to three-dimensional) Jordan (2017).

The decision function responsible for building the hyperplane is defined by Equation 2.9, where K is the kernel function, α and b are parameters found during the training, x_i are the feature vectors and y_i, their labels.

$$f(x) = \sum_i \alpha_i y_i K(x, x_i) + b \tag{2.9}$$

Since the majority of the real problems involve more than two classes and the SVM is a binary classifier, it's necessary to use a different decision approach. The most common is the one versus all, also called one versus the rest Manning et al. (2008b). In this approach, there are q classifiers (SVMs), where q is the number of classes. Each SVM c_i is trained to the class i, using as counterexample the samples from another classes. The final decision can be made through a "vote counting" Milgram et al. (2006).

Random Forest

Random forest is a classifier that consists of a collection (ensemble) of tree-structured classifiers, each of them trained on bagged data using random selection of features (bootstrap aggregating or bagging) and cast an vote for the most popular class for a given input Breiman (2001). Figure 2.7 shows an example of a random forest, containing L trees using bagging. To better understand this ensemble. we will explain how a decision tree works.

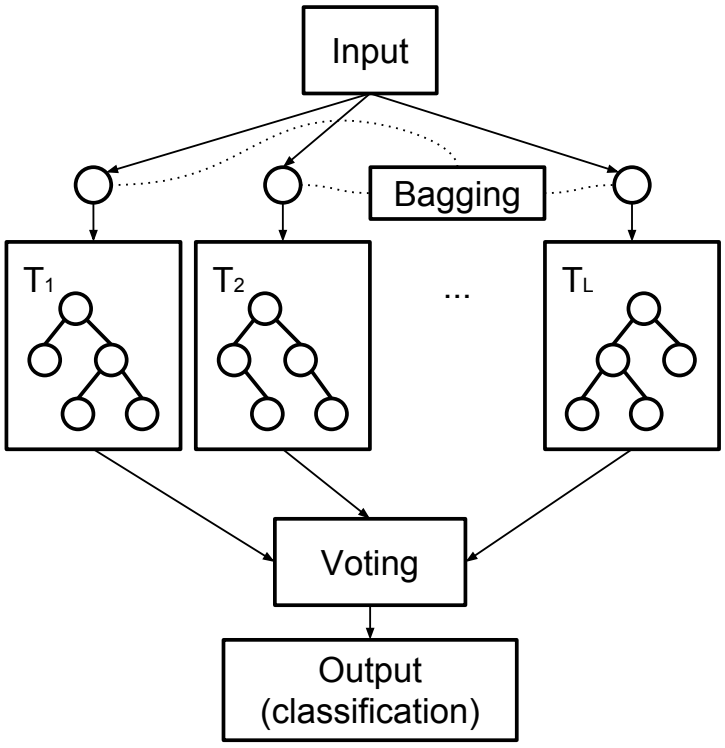


Figura 2.7: Example of a random forest containing L trees with bagging Breiman (2001).

A decision tree is basically a classifier that creates a set of if-then rules to classify new samples. These rules improve human readability, since it makes easy to understand the model and which attributes are important. The classification of new instances is done by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. Each node of the tree represents a test of some attribute of the instance, and each branch

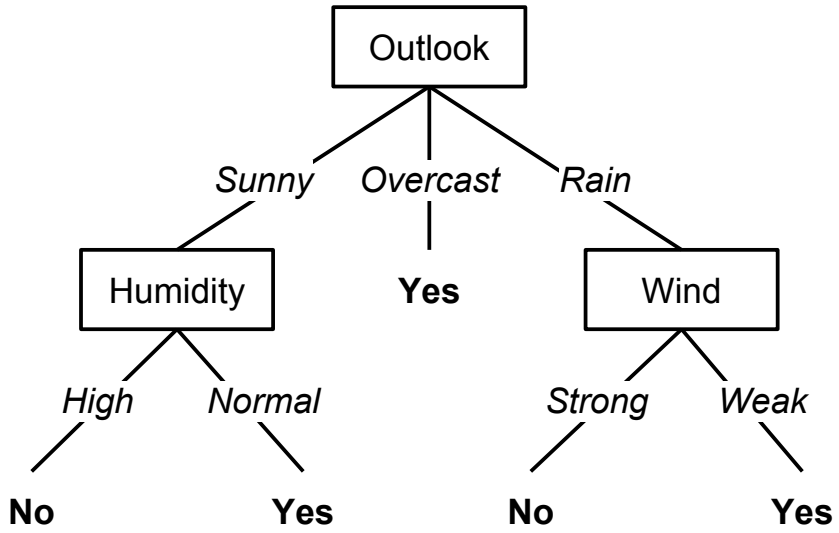


Figura 2.8: Example of a decision tree using the classical concept *PlayTennis* Mitchell (1997).

descending from that node corresponds to one of the possible values for this attribute Mitchell (1997). Figure 2.8 shows an example of a decision tree used to classify if a given moment of a day is good do play tennis (based on weather conditions).

The training of a decision tree is based in two measures: entropy and information gain. The entropy, represented by the Equation 2.10, where S is a set of training samples and c, the number of classes, measures the homogeneity of the samples.

$$Entropy(S) = \sum_{i=1}^{c} -p_i \log_2(p_i) \tag{2.10}$$

The information gain, as shown in 2.11, is used as an attribute selection measure. The tree is built according to the information gain of the attribute A, since you always pick the attribute with higher information gain as splitting attribute, to create new branches for each value of this attribute (or intervals, in the case of numerical features) Mitchell (1997).

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v) \tag{2.11}$$

The algorithm for training a decision tree (the basic algorithm, called ID3), given a set of *examples* (training samples), the *target attribute* (label, the attribute to be predicted by the tree) and the *attributes* (list of the attributes of the samples, excluding the label), is the following Mitchell (1997):

1. Create a root node for the tree.
2. If all *examples* are positive, return the single-node tree root, with $label = +$.
3. If all *examples* are negative, return the single-node tree root, with $label = -$.
4. If *attributes* is empty, return the single-node tree root, with $label$ = most common value of *target attributes* in *examples*.
5. Otherwise, do the following:

(a) A = the attribute from *attributes* that best classifies *examples*. The best attribute is the one with highest information gain, as already defined in Equation 2.11.

(b) The decision attribute for root is A.

(c) For each possible value v_i of A:

  i. Add a new tree branch below root, corresponding to the test $A = v_i$.

  ii. Let $examples_{v_i}$ be the subset of *examples* that have value v_i for A:

    A. If $examples_{v_i}$ is empty, them, below this new branch, add a leaf node with $label$ = most common value of *target attributes* in *examples*.

    B. Else, below this new branch, create a new subtree, going back to step 1 using a subset of *examples* ($examples_{v_i}$) and *attributes* ($attributes - \{A\}$).

2.1.4 t-SNE

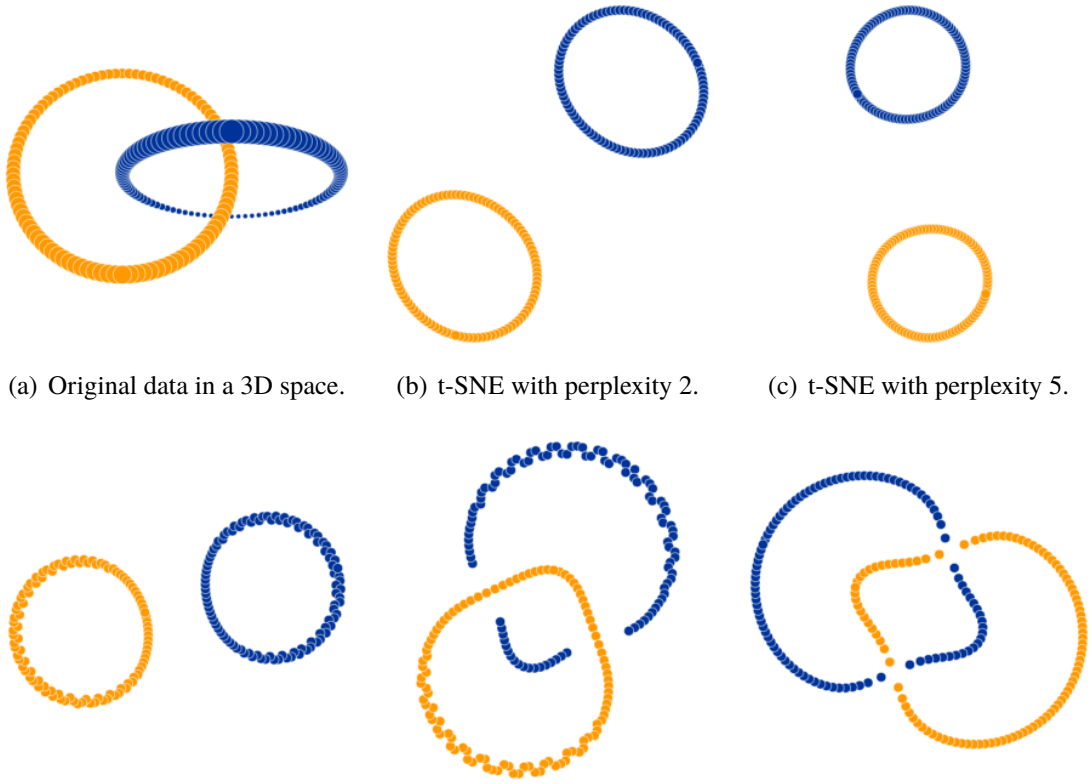
t-SNE (t-Distributed Stochastic Neighbor Embedding) is a technique for dimensionality reduction commonly used for the visualization of high-dimensional data. Since visual exploration is an essential component of data analysis and traditional visualization techniques (histograms, scatter plots, etc) only contribute to visualize few variables, it's important to understand t-SNE. Also, the visualizations produced by t-SNE are significantly better than those produced by other techniques, such as PCA (Principal Component Analysis), which is a linear algorithm, i.e., it's not able to interpret complex polynomial relationship. t-SNE, in the other hand, is based on probability distributions with random walks on neighborhood graphs to find the structure within the data van der Maaten e Hinton (2008a); van der Maaten (2014). The objective of t-SNE is to take a set of points in a high-dimensional space and find a faithful representation of those points in a lower-dimensional space (2D or 3D plane). This algorithm adapts to the underlying data, performing different transformations on different regions and aims to approximate similar examples in the new space. The t-SNE algorithm is composed by a tunable parameter called perplexity, a guess about the number of close neighbors each point has, which says how to balance the local and global aspects of the data in the new projection Wattenberg et al. (2016). Figure 2.9 shows examples of projections in a 2D space using t-SNE, with multiple perplexity values and 5.000 steps, comparing to the original data in a 3D space.

2.1.5 Concept Drift

Concept drift is the situation in which the relation between the input data and the target variable (variable that needs to be learned, which is often the class variable), changes over time. It generally happens when there is a change in a hidden context, which makes it difficult to handle, since this problem spans across different research fields. Each example in the input data is represented by a feature vector $x = [x_1, x_2, ..., x_L]$, where L is the number of features that are used to determine it's class y, according to the *a posteriori* probabilities $P(y, x)$. $P(x)$ is defined as features distribution and $P(y)$ as prior probabilities. In the literature, there are two types of concept drift, both described below Wang et al. (2011); Gama et al. (2014).

Virtual Concept Drift

The virtual concept drift happens when the distribution of the incoming data changes, i.e., $p(x)$ changes, without changing $p(y, x)$. Figure 2.10 shows an example of virtual drift with



(a) Original data in a 3D space. (b) t-SNE with perplexity 2. (c) t-SNE with perplexity 5.

(d) t-SNE with perplexity 30. (e) t-SNE with perplexity 50. (f) t-SNE with perplexity 100

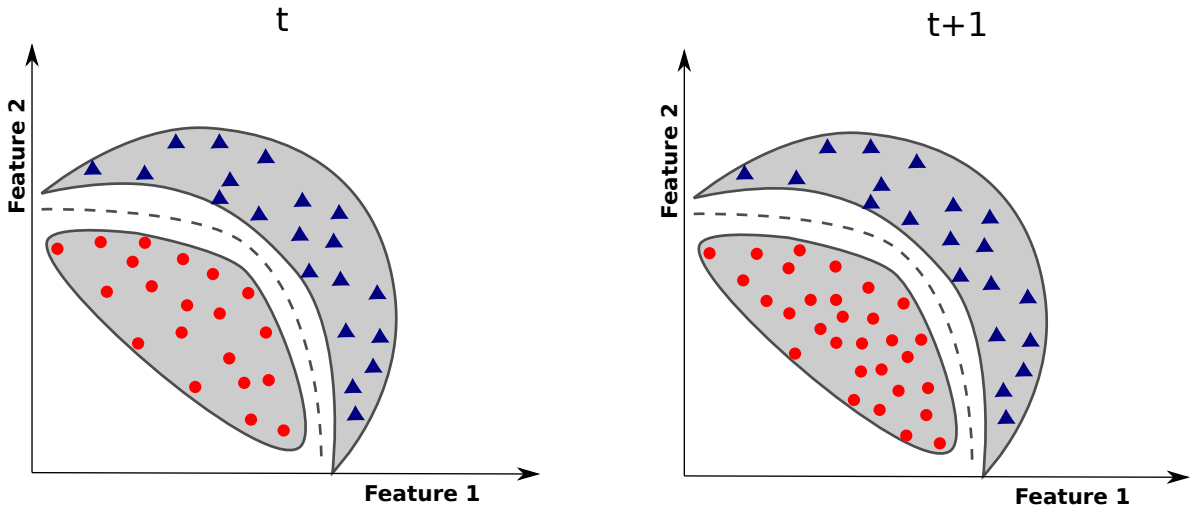
Figura 2.9: Data visualization using the original data in a 3D space and t-SNE with different values of perplexity and 5.000 steps (iterations) in a 2D space Wattenberg et al. (2016).

changes in the data distribution at times t and $t + 1$, where the red class is more prevalent but it does not lead to changes in the best boundary between them Almeida et al. (2015).

To illustrate the problem caused by the virtual concept drift, Figure 2.11 presents the distribution of a feature for the blue and red classes (presented in Figure 2.10) at both times t and $t + 1$. The vertical dashed line in both figures is the threshold that separates both classes (the misclassification cost of the red class is higher than the blue one), that are equally distributed in t (Figure 2.11(a)), different from $t + 1$ (Figure 2.11(b)), which has greater presence of the red class. Despite of the fact that the mean and standard deviation of both classes remains unchanged, with the threshold in the same position, i.e., keeping the same classifier unchanged, the probability of finding a red object as a blue one is increased, as shown in the dark red area in Figure 2.11(b) Almeida et al. (2015).

Real Concept Drift

The real concept drift happens when there is a change in $P(y, x)$ with or without changing $p(x)$, i.e., the relation between the classes and the feature vectors changes over time. A classic example is related to the e-mail spam problem, where an e-mail represented by a feature vector x_e can be considered as a spam at a given time t and cannot be at $t + 1$, due to user behavior changes. As an example of real concept drift, Figure 2.12 shows a two class problem with a *a posteriori* probabilities drift, causing changes in the boundaries at time t and $t + 1$ (Figures 2.12(a) and 2.12(b), respectively) and forcing an update in the classifier Almeida et al. (2015).



(a) Examples projection at time t.

(b) Examples projection at time $t + 1$.

Figura 2.10: Virtual concept drift example with $P_t(y) \neq P_{t+1}(y)$. Almeida et al. (2015)

2.1.6 Malware

Malware is a short definition for malicious software, including viruses, spyware and other unwanted software, that are usually used to cause damage to a single computer, server, mobile device or computer network without user's consent. They can also be used to make the device crash, to monitor and control user's activity (generally used by criminals to steal personal information, send spam and commit fraud), to make the device vulnerable to viruses and deliver unwanted ads Microsoft (2003); COMMISSION (2015).

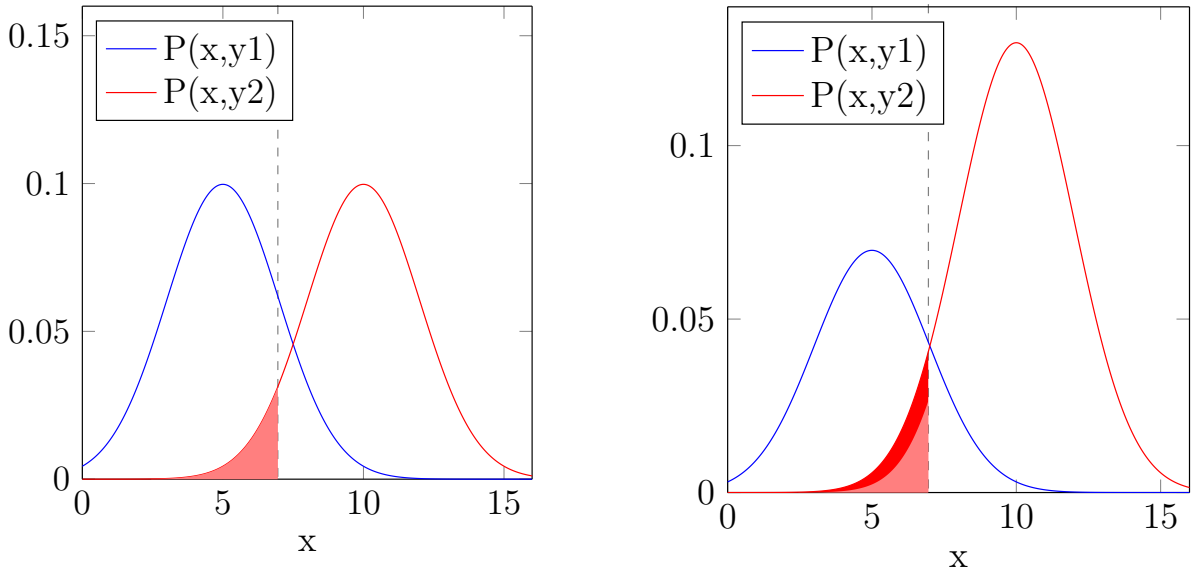
2.1.7 Portable Executable (PE)

Portable Executable (PE) is a file format for executables, dynamic libraries and others used in both 32-bit and 64-bit versions of Windows, which defines a data structure to encapsulate the necessary information for the OS to manage the wrapped executable code. This file format contains many potentially interesting structural components (attributes), as shown in Figure 2.13, for malware analysis, including the ones shown in the sections bellow Yonts (2010); Pietrek (1994).

PE File Header

Attributes extracted using information from the PE file header:

- **Machine:** identifier of the CPU the file is intended for.
- **Number Of Sections:** number of sections of the file.
- **Time Date Stamp:** the time the file was created, in seconds.
- **Pointer To Symbol Table:** file offset of the COFF (Common Object File Format) symbol table.
- **Number Of Symbols:** number of symbols in the COFF symbol table.



(a) Class distribution at time t.

(b) Class distribution at time $t + 1$.

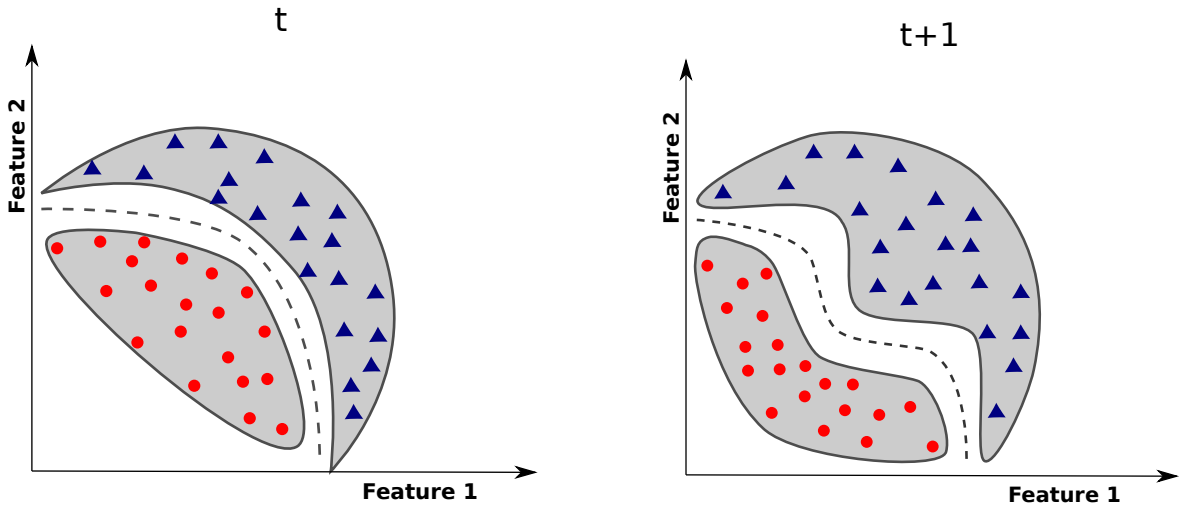
Figura 2.11: Virtual concept drift example with $P_t(y) \neq P_{t+1}(y)$ Almeida et al. (2015).

- **Size Of Optional Header:** size of the file's optional header.
- **Characteristics:** set of flags with file's information. They indicate whether were reallocations, whether the file is an executable image, and/or if the file is a dynamic link to a library.

Optional Header

Attributes extracted using information from the file optional header:

- **Magic:** file architecture's identifier.
- **Size Of Code:** combined and rounded-up size of all the code sections.
- **Size Of Initialized Data:** total size of all sections that are composed of initialized data.
- **Size Of Uninitialized Data:** size of the sections that the loader commits space in the virtual address space, but that do not take up any space in the disk file.
- **Base Of Data:** RVA (Relative Virtual Address) where the file's data section begins.
- **Base Of Code:** RVA where the file's code sections begin.
- **Image Base:** address where the file will be mapped in memory.
- **Size Of Image:** total size of the portions of the image that the loader must worry about.
- **Size Of Headers:** size of the header and the section table.



(a) Decision boundary at time t. (b) Decision boundary at time $t + 1$.

Figura 2.12: Real concept drift example Almeida et al. (2015).

Others

Other attributes that can be useful to malware analysis, extracted from the Export Directory Table (EDT), Import Library Table (ILT) and Import Directory Table (IDT):

- **Imported Dlls:** list of the dynamic libraries (DLL) used by the file.
- **Imported Symbols:** list of the functions that the file uses (every function belongs to a library listed in ImportedDlls).
- **Identify:** list of packers, compilers and/or tools used to create the file.

2.2 Related Work

The literature on detection, classification, and/or clustering of malicious programs using machine learning is extensive Gandotra et al. (2014) and can be divided in three types, according to the extracted characteristics: based on static attributes from the executable binary; on dynamic attributes obtained through running the sample in controlled environments; and hybrid approaches that combine both methods. It is also possible to divide the solutions between those that consider the detection component (they use malicious and benign programs for training and testing the classification algorithm) and those that try to classify unknown programs into pre-existing groups (families) of malicious programs. However, there is no standard dataset that can be used as a ground truth for evaluating the generated classifiers, such as the classic KDD Cup 1999 dataset for intrusion detection The UCI KDD Archive (1999), in addition to the fact that the data used in published papers is rarely available. We briefly discuss some articles that address the malware classification problem by using machine learning techniques, which represent the related work of their respective time of publication.

The first articles on automated classification of malicious programs appeared in 2005. In that same year, Gheorghescu proposed a scheme to compare viruses and to identify the unknown ones based on the distance of their observed stored behaviors Gheorghescu (2005). In 2007, Bailey et al. proposed an approach to classify malware at a large scale, whose input was

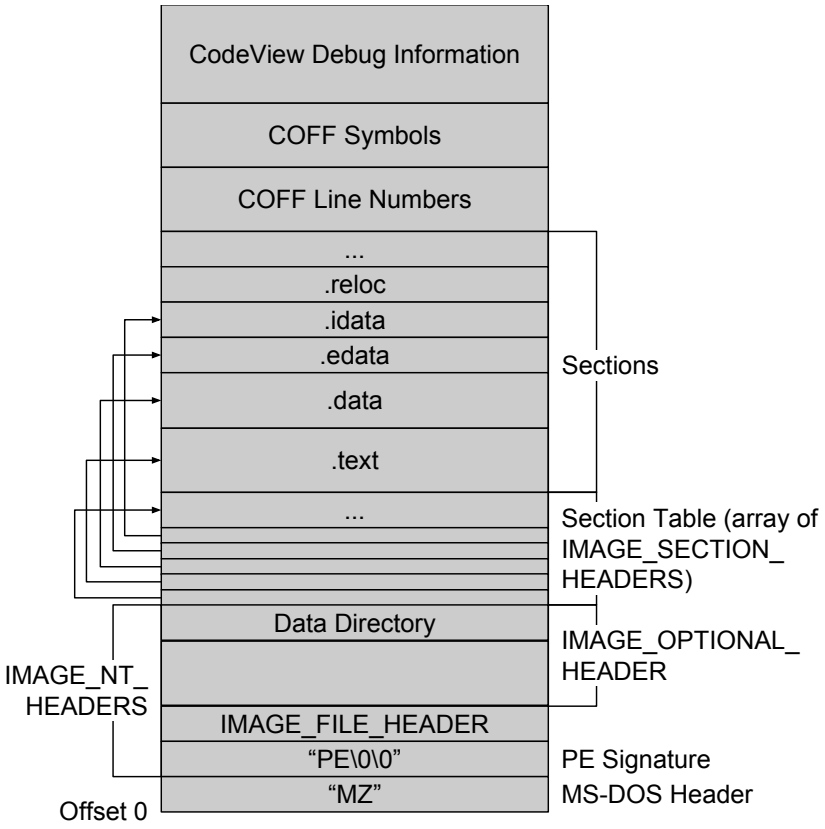


Figura 2.13: PE file format structure Pietrek (1994).

the representation of samples behavior as "state changes" meaning the interaction between the program and the target system Bailey et al. (2007). The generated profiles were then compared one by one to each other for creating candidate family groups based on distance calculation. Other important approaches that rely on malware behavior are shown in Bayer et al. (2009) and Rieck et al. (2011). In the first one, the similar behavior identification involves clustering similar samples through a Locality Sensitive Hashing (LSH) algorithm; in the latter, the goal is the same, but achieved using hierarchical clustering. Malware grouping into families of samples was also explored in Grégio et al. (2012), in which the authors use the Jaccard distance for calculating the similarity distance between profiles extracted from malicious programs debugging. The profiles contained specific instructions used for writing in memory in an attempt to identify actions that differentiate malware families.

It is important to point out that the methods presented so far only apply techniques to classify malicious programs into families and, given an unknown program (neither labeled nor present in the dataset), assign it to one of the produced groups. Therefore, the presented approaches did not consider the existence of benign programs, limiting the contribution that would be done in users protection's enhancement. We can notice limitations even in more recent articles, such as small datasets (eight thousand in Annachhatre et al. (2015), 20 thousand in Ahmadi et al. (2016)), which are built using malware sharing websites or honeypots with samples collected on the same year or a year before the article. Furthermore, it is worth noting that proposals whose datasets are composed by malicious programs only may have tendentious results, as evidenced in Li et al. (2010), which is based on an evaluation of the results achieved in the already cited article Bayer et al. (2009). Table 2.1 presents related works that are similar to this, i.e., those that use machine learning and extract features from Windows programs to classify them in two classes (malware and benign). The first article (Schultz et al. (2001)) appeared in 2001 and

was the first to introduce the concept of data mining for detecting malware, containing 4, 266 samples of both malware and goodware and using portable executable (PE) list of DLLs, function calls, number of different system calls used within each DLL, strings and non-overlapping byte sequence as features in a Naive Bayes classifier. Three years later, Kolter e Maloof (2004) used the same features, but now using overlapping sequences of four bytes in a more extensive list of classifiers, such as Support Vector Machine and Decision Tree. Years later, in 2011, Nataraj et al. (2011) extracted features using image processing techniques to visualize malware binaries as gray-scale images, classified their 9, 581 samples (9, 458 malware and 123 goodware) using a K-nearest neighbors and obtained 98.08% of accuracy. In the same year, Anderson et al. (2011) was one of the first works to use dynamic analysis technique using similarity matrix resulting from graphs that represent Markov chains in a Support Vector Machine and obtained an accuracy of 96.41%. In the next year, in 2012, the same author (Anderson et al. (2012)), using both static and dynamic features, built using control flow graph, dynamic instruction trace, system call trace and a file information feature vector, achieved an accuracy of 98.07%, a result very close to the one obtained in Nataraj et al. (2011). However, despite of having these results, most of the works cited does not have a representative set of samples. Nataraj et al. (2011), for example, only uses 9, 581 samples (only 123 goodware and 9,458 malware). In the other hand, Kantchelian et al. (2013) uses a dataset containing 188, 704 samples, where 94, 352 are goodware and 123, 845 are malware, and classifies them using an ensemble of classifiers, one for each family of unwanted behavior (they did not report the accuracy, only false negative rate – 55% – and false positive rate – 1%). Analyzing the whole picture, it's clear to see how difficult it is to compare works in this area, once each one use a different dataset and do not make them publicly available, which makes impossible to reproduce them. Also, it's evident that there are not standard metrics for this problem.

Also, there are questions involving machine-learning based approaches: the majority evaluation metrics does not consider the time, an important factor since malware samples are always evolving (they change their concept, a problem known as concept drift) in response to external (new technologies and detectors) or internal (new capabilities) pressures Singh et al. (2012). Trying to solve this problem, Roberto Jordaney e Cavallaro (2016) presents conformal evaluator, an evaluation framework that makes use of statistical metrics to capture the quality of the produced results. Roy et al. (2015) concludes that AUPRC (Area Under the Precision Recall Curve) is a better metric for comparing results of different approaches in ML-based Android malware detection. Allix et al. (2015) shows that using a random set of known malware to train a detector, as it is done in many experiments in literature, yields significantly biased results, while Kantchelian et al. (2013) says that an ensemble of classifiers is responsive to malware changes, since it is composed by multiple classifiers, each one of them representing a malware family. Jordaney et al. (2017) introduces a framework called Transcend, capable of identifying the aging of a machine learning based model, even before its performance starts to decrease, thought statistical comparisons which makes it possible to identify a concept drift in Android and Windows malware – the Windows malware analysis, however, does not use the binary classification addressed in this work, but rather the classification in families.

The current work, alternatively, aims to overcome these limitations whereas focusing in malware collected in-the-wild in Brazilian scenario, since it exhibits peculiar samples compared to those of other countries[1] Grégio et al. (2013). Thus, our goal is to provide a representative

[1] https://krebsonsecurity.com/tag/bolware/
http://blog.checkpoint.com/2016/02/18/the-return-of-the-brazilian-banker-trojan/
http://www.welivesecurity.com/2016/07/19/malicious-scripts-gaining-

Tabela 2.1: Related work table comparing datasets, features, algorithms and results.

Paper	Dataset			Features	Algorithm	Results
	Malware	Goodware	Total			
Schultz et al. (2001)	3,265	1,001	4,266	Portable Executable (PE) list of DLLs, function calls, and number of different system calls used within each DLL, strings and non-overlapping bytesequence.	Ripper and Naive Bayes	Accuracy: 97.11%
Kolter e Maloof (2004)	1,651	1,971	3,622	Portable Executable (PE) list of DLLs, function calls, and numberof different system calls used within each DLL, strings and overlapping sequences of four bytes.	Naive-Bayes, Support Vector Machine, Decision Tree and their boosted versions	AUC: 99.6%
Siddiqui et al. (2009)	1,444	1,330	2,774	Variable length instruction sequence	Decision Tree and Random Forest	Accuracy: 96%
Tian et al. (2010)	1,368	456	1,824	Behavioural features using logs of various API calls.	WEKA Classifiers	Accuracy: 97%
Firdausi et al. (2010)	220	250	470	Sparse vectors preprocessed from reports generated by a sandbox environment.	K-nearest neighbors, Naive Bayes, J48 Decision Tree, Support Vector Machine and Multilayer Perceptron Neural Network	Accuracy: 95.9%
Nataraj et al. (2011)	9,458	123	9,581	Image processing techniques to visualize malware binaries as gray-scale images.	K-nearest neighbors,	Accuracy: 98.08%
Anderson et al. (2011)	1,615	615	2,230	Similarity matrix resulting from graphs that represent Markov chains.	Support Vector Machine	Accuracy: 96.41%
Santos et al. (2011)	1,000	1,000	2,000	Byte n-gram distribution.	Semi-supervised Learning, Learning with Local and Global Consistency (LLGC)	Accuracy: 86%
Anderson et al. (2012)	780	776	1,556	Control flow graph, dynamic instruction trace, system call trace and a file information feature vector.	Support Vector Machine	Accuracy: 98.07%
Santos et al. (2013)	1,000	1,000	2,000	Sequence of operational codes and system calls,operations and raised exceptions monitoring.	K-nearest neighbors, Decision Tree, Bayesian Network and Support Vector Machine.	Accuracy: 96.60%
Islam et al. (2013)	2,398	541	2,939	Function length frequency, printable string information, API function names and API parameters.	Support Vector Machine, Decision Tree, IB1 and Random Forest.	Accuracy: 97.06%
Kantchelian et al. (2013)	123,845	94,352	188,704	Sparse 120K dimensional binary vector derived from the control flow graph of the instance using static binary analysis by the provider.	Ensemble of classifiers, one for each family of unwanted behavior.	False Negative Rate: 55% & False Positive Rate: 1%
Ghiasi et al. (2015)	850	390	1,240	API calls with registers values before and after invoking API calls	Random forest, J48, SMO and Bayesian logistic regression.	Accuracy: 92.1%, Recall: 94.6%, Precision: 93.9%
Mangialardo e Duarte (2015)	131,073	2,659	133,732	Attributes obtained throuth analisys of the Portable Executable (PE) header and API calls.	C5.0 and Random Forest.	Accuracy: 95.75%
Hu e Tan (2017)	9,183	9,998	19,181	DLL and API feature, the string feature, PE-Miner and the byte level N-Gram feature.	Random Forest	Accuracy: 97.20%, AUC: 99.57%, TPR: 97.5%, FPR: 3.07%

dataset in time and variety of samples observed in Brazilian cyberspace that allows the community to perform additional analysis and comparison with other literature work. Also, using features extracted from PE headers, which are known to have high discriminative power in both malware families and binary classification problems Yan et al. (2013), we present experiments that evidence the presence of concept drift in this dataset.

prevalence-brazil/
https://securelist.com/blog/research/74325/the-evolution-of-brazilian-malware/ http://researchcenter.paloaltonetworks.com/2016/03/banload-malware-affecting-brazil-exhibits-unusually-complex-infection-process/

Capítulo 3

Data Collection and Preprocessing

This Chapter presents details about the data collection and preprocessing, both presented in Sections 3.1 and 3.2. Briefly, we show how we collect our data and how we use them in machine learning classification algorithms.

3.1 Data Collection

This Section presents the details about the methodology used to collect our data and their extracted attributes, both presented in Sections 3.1.1 and 3.1.2.

3.1.1 Methodology

To create the dataset presented in this work, we used a set of ≈180 GB of executable files (malware and benign software) that were collected in Brazilian cyberspace or popular Internet download sites from 2013 to early 2017. We collected two classes of software to build the dataset: goodware (allegedly benign software) and malware. To obtain goodware samples, we implemented a Web crawler that downloaded software from three sources: Sourceforge Slashdot Media (2017), Softonic Softonic Internacional S.A. (2017) and CNET Download CNET (2017). We collected ≈130 GB of binary files, which we assumed benign, totaling 18, 511 unique samples. We have an established partnership for many years with a major Brazilian financial institution (which prefers to remain anonymous) who provides us with daily malware samples collected from detected infections in its corporate perimeter or that were identified by customers via phishing email attachments. As the malware samples were received by our server, we group them by day with the objective to save the temporal information. This process has been executed in an ongoing fashion since January 2013, with the exception of the period from January to July 2016, when the collection was paused due to a shutdown period in the storage server. Figure 3.1 presents the data collection scheme: the Brazilian financial institution sends daily new samples in a FTP server. These samples are then collected by a second server, which also downloads goodware samples. At the moment of this paper, we collected approximately 50 GB of malicious binary samples, totaling 26, 800 samples, from which 19, 979 are unique.

3.1.2 Extracted Attributes

We extracted as many static attributes as possible from the downloaded executables with the goal to make it available for the research community without direct malware sharing, a practice that is illegal in Brazilian law. We used Python's `pefile` libraryCarrera (2016), which

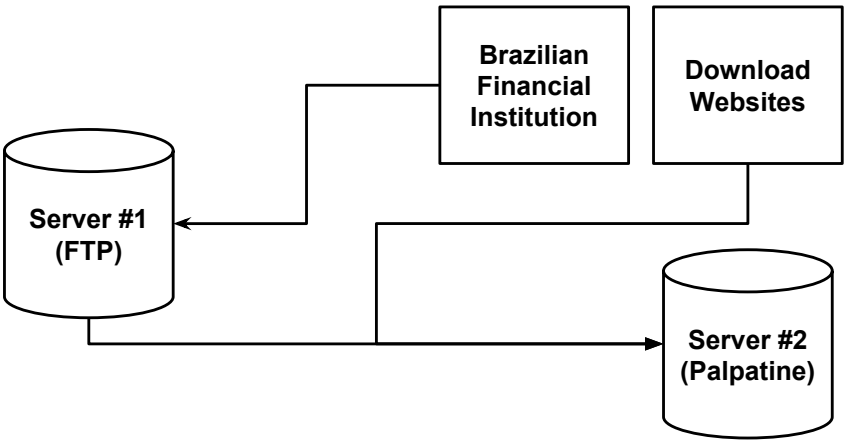


Figura 3.1: Data collection scheme.

allows us to access the attributes of PE (Portable Executable) files Pietrek (1994). Figure 3.2 presents the attributes extraction scheme: the attributes are extracted from the collected samples stored in server two and saved in a text file formatted as *csv* in the project repository.

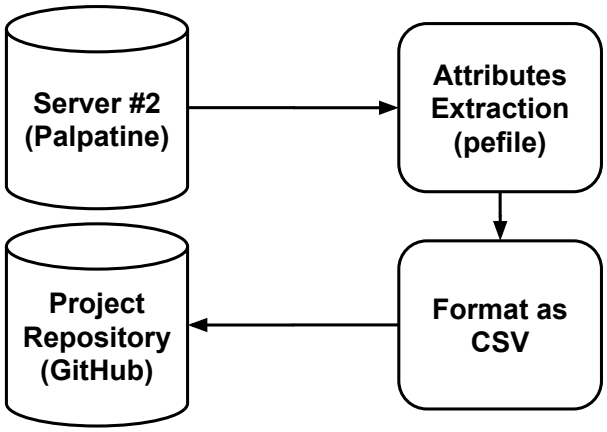


Figura 3.2: Attributes extraction scheme.

The extracted attributes Yonts (2010) for each executable are listed in the following sections. In addition to the attributes extracted from the portable executable file structure (PE file header, optional header and others), as shown in Section 2.1.7, general attributes, as the file name, size, hashes (MD5 and SHA1) and entropy (Shanon entropy – the greater the entropy, the greater the chance that the file is compressed or encrypted), were extracted.

After extracting those features, the collected attributes were recorded into text files (formatted as CSV). Since we collect malware samples every day, there is a text file for each one, containing the attributes of the collected samples for that day. For the goodware, there is a single text file containing all of them (each line represents a single sample, with its associated attributes). The produced dataset can be found in the project's repository for this work, publicly available Ceschin et al. (2016).

3.2 Preprocessing

The analysis of data obtained from Section 3.1.2 allowed us to divide the attributes into the three categories below. Thus, it is possible to decide what attributes will be used for training a classifier:

- **Unique attributes:** they are unique for each executable and are not interesting for machine learning algorithms, given that they are not discriminant for the classes, only for the files. They include the MD5 and SHA1 hashes and the name of the file (name). However, they can be interesting in a previous step, since it is possible to check for them in blacklists or whitelists of benign or malicious programs through their hashes, and discover if they try to deceive the victims by using a name of an already known-good system process or application.
- **Numerical attributes:** they represent integer or floating-point numbers that must be normalized to keep the same type of format. In this case, the numerical attributes are the following: BaseOfCode, BaseOfData, Characteristics, DllCharacteristics, FileAlignment, ImageBase, Machine, Magic, NumberOfRvaAndSizes, NumberOfSections, NumberOfSymbols, PE_TYPE, PointerToSymbolTable, Size, SizeOfCode, SizeOfHeaders, SizeOfImage, SizeOfInitializedData, SizeOfOptionalHeader, SizeOfUninitializedData, TimeDateStamp, and Entropy.
- **Textual attributes:** they comprise a set of words that must be previously processed and also normalized before used. In the presented problem, there are three set of words: Identify, ImportedDlls and ImportedSymbols.

It is worth noticing that only numerical and textual attributes were used in these experiments. The following sections deal with textual attributes pre-processing (Section 3.2.1) and normalization of all attributes (Section 3.2.2).

3.2.1 Textual Attributes Preprocessing

The textual attributes pre-processing step aims to cluster similar texts (documents). This way, programs that use the same libraries, functions, and compilers tend to be close in the features' space. To make that possible, each set of words is transformed in a document (in the context of information retrieval), where each word is separated by a space. Thus, three "documents" are created by file, one for each textual attribute (Identify, ImportedDlls and ImportedSymbols). Following that, all text is case-folded and normalized: text content was kept in lowercase and any special characters, accent, marks and numbers were removed. Therefore, we obtained statistic measures about every document based on their words. In this work, we used the Vector Space Model (VSM) representation model, shown in Section 2.1.1, which represents a document through the relative importance of its words Turney e Pantel (2010); Manning et al. (2008a). This was done to try to differentiate benign and malicious software by the programming language, compiler, packer, and imported libraries used and the functions the program invoked.

3.2.2 Normalization

After pre-processing all textual attributes, each of their VSM vectors are concatenated to the others attributes of the file, resulting in a new vector with $22 + 3 \times V$, where 22 is the

number of numerical attributes, and V the size of the vocabulary used over the textual attributes. However, there is still a difference in the extracted characteristics scale, leading in the need of normalizing them. The normalization technique applied on the extracted features was the MinMax, presented in Section 2.1.2. This technique scales every feature (characteristic) into an interval between zero and one, using the formula seen on Equation 2.3.

From this moment on, all the files are normalized and ready for the classification step. The Figure 3.3 shows the projection of the extracted features using t-SNE van der Maaten e Hinton (2008b)[1]. This projection contains $5,000$ samples (half of them goodware - blue - and half, malware - green). It is possible to note some clusters, in addition to some overlaps between both classes. We also projected our samples splitting them in years again (from 2013 to 2016), as shown in Figure 3.4, which helps to evidence the presence of drift in the data, since t-SNE clusters similar samples and it is clear to observe class grouping.

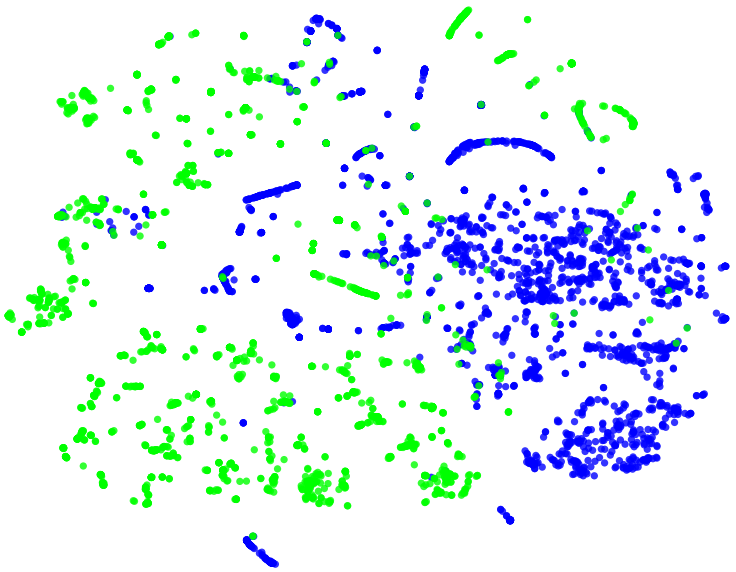
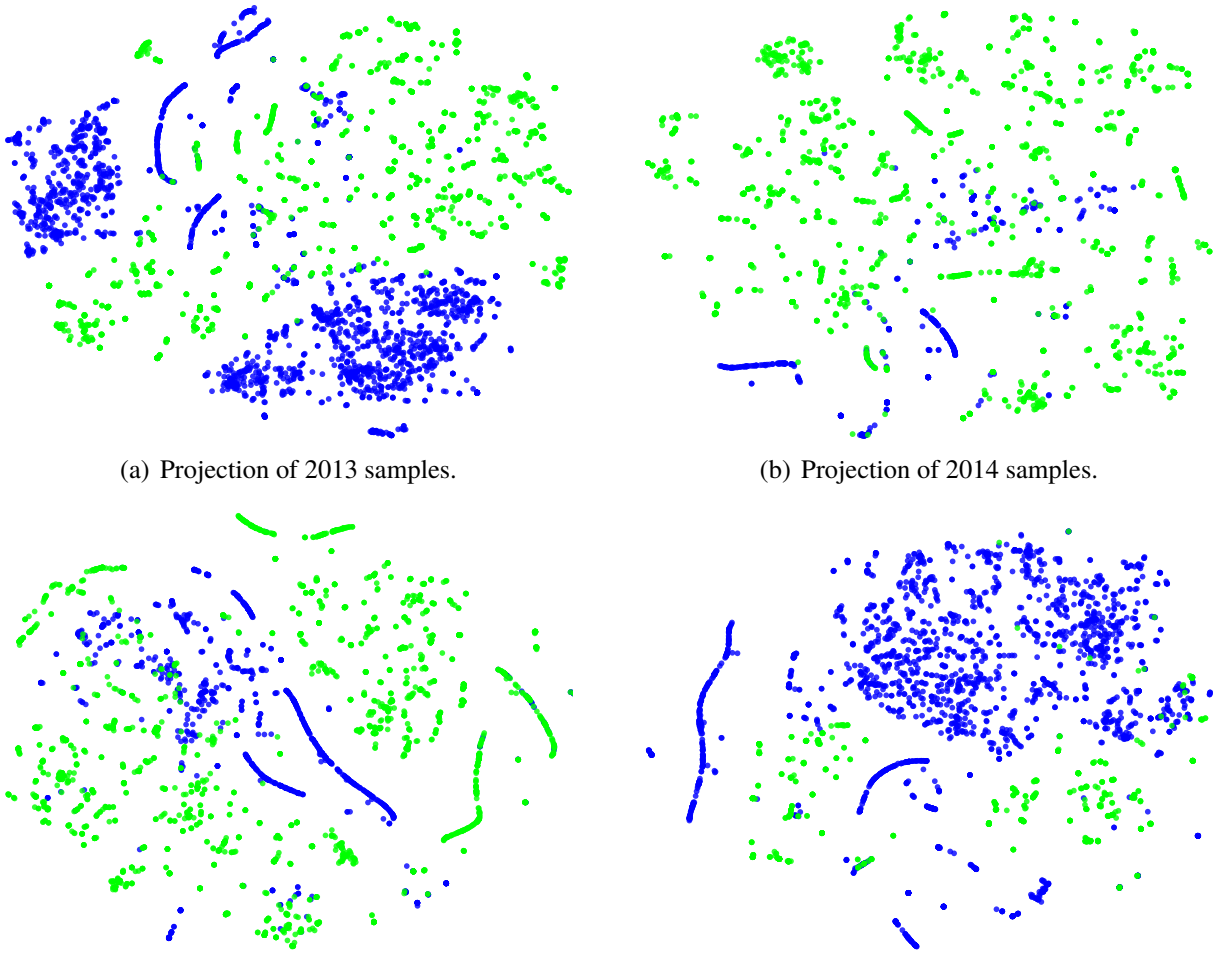


Figura 3.3: Partial dataset projection using t-SNE in a two-dimensional space, where the green and blue points represent malware and goodware, respectively.

[1] http://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html#sklearn.manifold.TSNE



(a) Projection of 2013 samples.

(b) Projection of 2014 samples.

(c) Projection of 2015 samples.

(d) Projection of 2016 samples.

Figura 3.4: Dataset projection from 2013 to 2016, where the green and blue points represent malware and goodware, respectively.

Capítulo 4

Data Analysis

In this Chapter we present the analyses performed over the collected samples of malware and benign software. First, we present the data distribution by year (Section 4.1). Then, we make a deeper analysis over the malware samples, i.e., the distribution within families, in Section 4.2). Those analyses were possible using VirusTotal API VirusTotal (2017), together with AVClass Sebastián et al. (2016), since VirusTotal can test an uploaded sample on more than sixty antiviruses, and AVClass can classify them in families based on VirusTotal reports. Moreover, we identified the most used dynamic libraries (Section 4.3) and functions (Section 4.4) by both classes to verify whether this information aids in differing goodware from malware.

4.1 Distribution by Year

The date at which a piece of malware appears in the wild is crucial for the problem of malware classification, since it is possible to observe their evolution over time. In Table 4.1 we break down our samples by year. For malware samples, the year corresponds to the year in which a sample appears in-the-wild; for goodware, the year corresponds to the timestamp that its corresponding binary file was compiled (this information can be obtained from the header file). In total, we collected 26, 800 malware and 18, 511 goodware samples. In both cases, there are more samples in 2013, with 10,075 malware and 11,930 goodware samples. Note that there were less malware samples in 2016 because the collection stopped from January to July in that year and resumed only in August. Since goodware collection occurred in early 2017, there were just three samples from this year for the goodware class. Also, we divided our samples in trimesters, as shown in Figure 4.1, a stacked bar chart. It is important to say that some goodware were compiled before 2013 and, because of that, they are not present in this chart, as well as malware samples collected in August 2016 (since they do not belong to the last trimester of 2016). Note that in the third trimester of 2013, particularly in August 2013, there is a large number of goodware comparing to other trimesters. This means that a lot of benign programs present in our set were compiled in similar dates (windows native programs, for example). Malware samples present a balanced distribution, despite the drop in the last two trimesters.

4.2 Malware Families Distribution

In this section we present two studies about the collected malware families. The first one consists of the general distribution of the samples (Section 4.2.1) while the second consists of the distribution per year (Section 4.2.2).

Tabela 4.1: Samples distribution by year.

Year	Malware	Goodware
<=2013	10,075	11,930
=2014	8,688	512
=2015	6,461	1,057
=2016	728	5,009
>=2017	848	3
Total	**26,800**	**18,511**

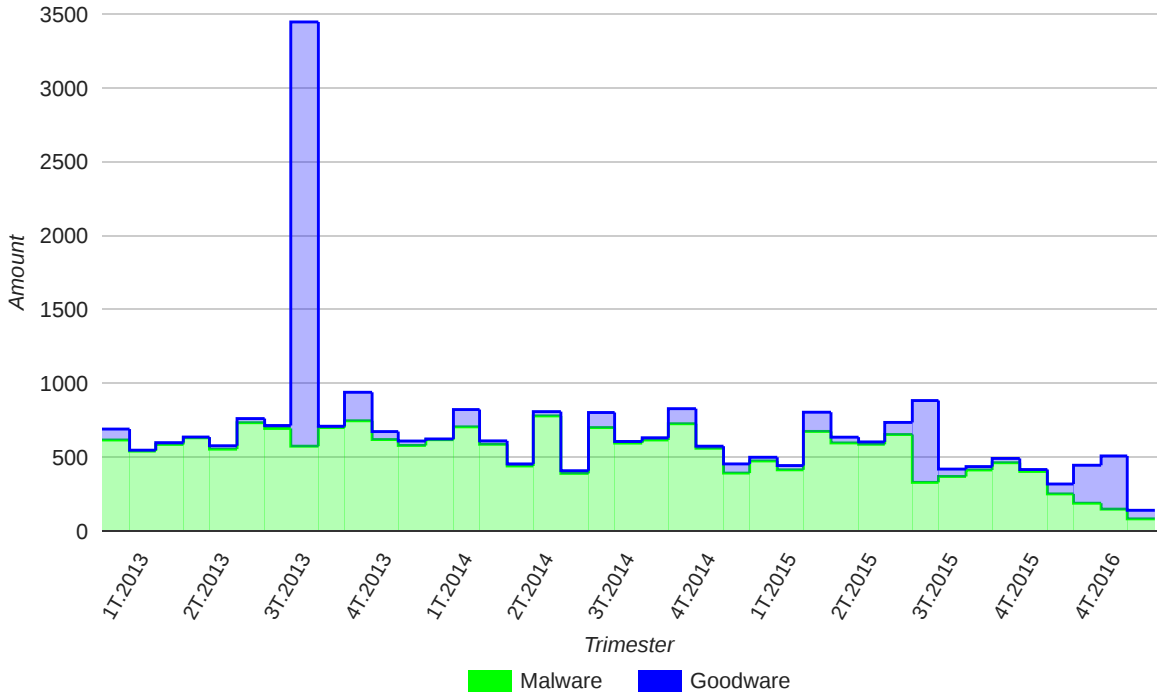


Figura 4.1: Data distribution by trimester (stacked).

4.2.1 General Distribution

Using the reports generated by VirusTotal and AVClass, we were able to obtain the malware classification per familiy. As shown in Figure 4.2, the most prevalent family in our dataset is *banload* (31.2%), a family of Trojans that steal user's banking credentials and other bank-related sensitive information and send them back to a remote attacker[1]. The next most prevalent family was *chePro* (11.2%), a class of downloaders that usually install banking malware and can be used to attack virtually any Internet banking service[2]. Then, we have *confidence* (8.7%), a malware family that could not be classified by AvClass because of the different names used by all AV engines Sebastián et al. (2016). Finally, we have *delf* (7.2%), a family of Trojans usually written in Delphi, with highly variable behavior, often associated to information stealing[3]. Notice in Figure 4.2 the high heterogeneity of this dataset, as 18.5% of the samples belong to other small families with less than 3% prevalence, such as *bestafera*, a family of banking Trojans

[1] `https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?name=TrojanDownloader%3AWin32%2FBanload`

[2] https://threats.kaspersky.com/en/threat/Trojan-Banker.Win32.ChePro/

[3] `https://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=Trojan:Win32/Delf`

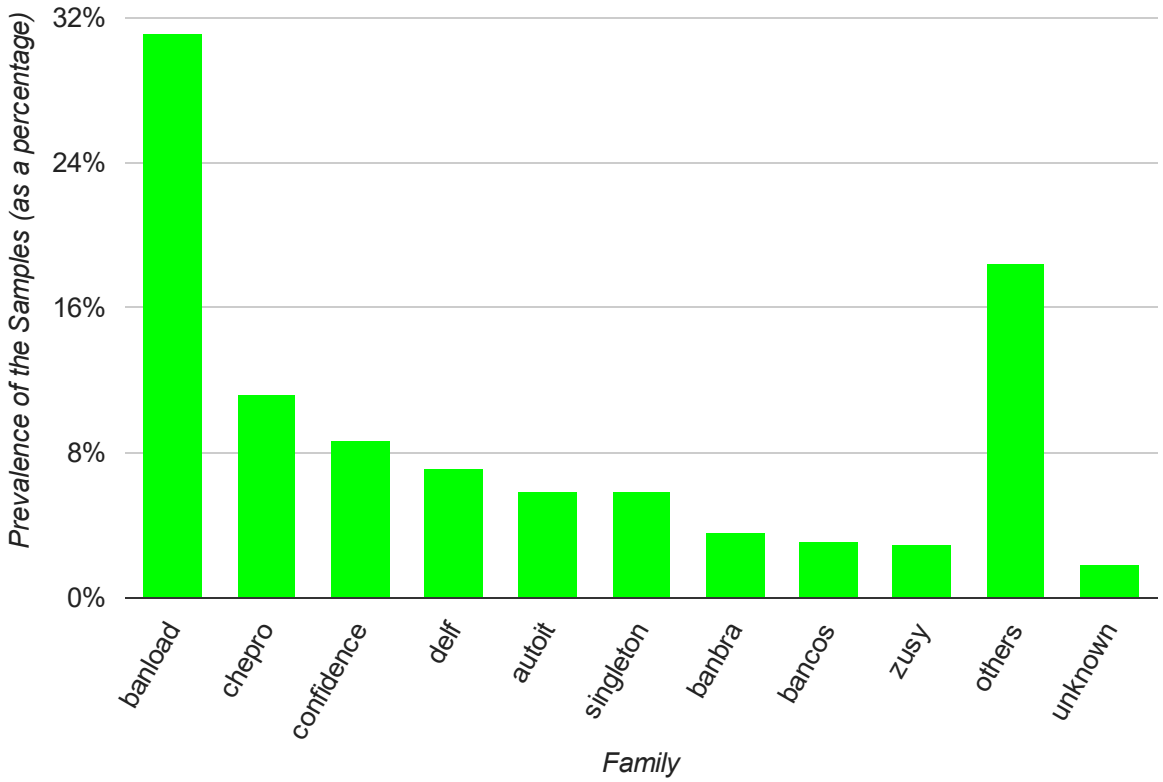


Figura 4.2: Families distribution in collected malware.

that sends information about the victim machine to a remote control server[4], *vobfus*, a family of worms that can download other malware[5], and *proxyChanger*, a malware type that can perform a variety of malicious actions[6]. Some samples could not be scanned by VirusTotal and they were labeled as "unknown" (1.8%).

4.2.2 Distribution by Year

This analysis is important because it allows us to discover whether there is a similar distribution between families as time goes by or not. In Figure 4.3, we show the distribution of the most prevalent malware families between 2013 and 2016. As already mentioned before, the *banload* family is the most prevalent across the years, reaching almost 40% of the samples in 2015. However, its prevalence in 2016 is lower than in previous years. The same was found for the following families: *chepro* (achieved 15.14% of the samples in 2014 to 0% in 2016); *confidence* (9.55% in 2013 to 4.98% in 2016); *delf* (8.71% in 2013 to 3.42% in 2016); *bancos* (5.14% in 2013 to 0.31% in 2016), a family of trojans that can steal online banking credentials[7]; *banbra* (3.41% in 2015 to 1.66% in 2016), a malware family that collect user's sensitive information[8]; *autoit* (7.79% in 2014 to less than 0.83% in 2016), a worm family that attempts to capture and steal

[4] http://telussecuritylabs.com/threats/show/TSL20160307-05

[5] https://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=Win32%2fVobfus

[6] https://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=Trojan:JS/ProxyChanger.A

[7] https://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=Win32/Bancos

[8] https://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=TrojanSpy:Win32/Banbra

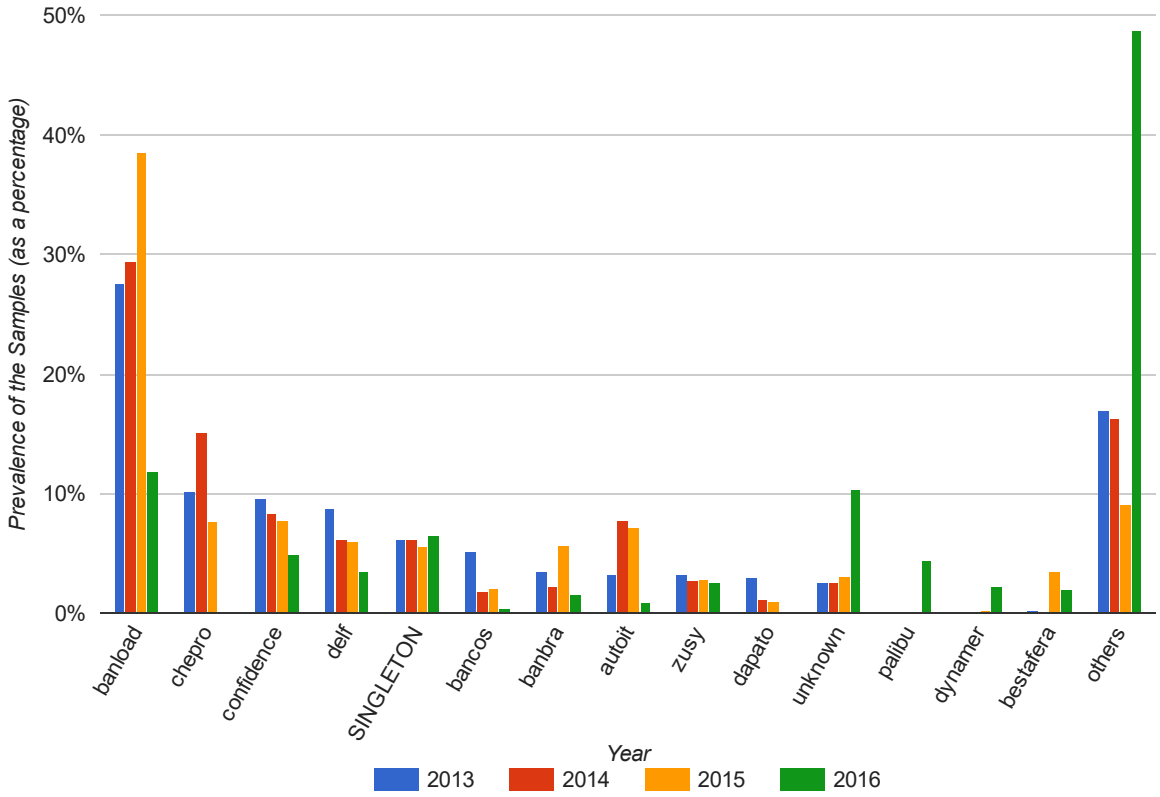


Figura 4.3: Malware families distribution by year.

authentication information for a number of different web sites or services, including Facebook and GMail[9]; and *dapato* (2.94% in 2013 to 0.1% in 2016), a malware family that downloads and installs other programs, including other malware, without user's consent[10]. Over the years 2014 – 2016 we also notice an increase in the number of unknown samples, as well as an increase in the number of samples for the following malware samples: the families *palibu*, a type of malware that performs activities without the user's knowledge (remote access connections, capture keyboard input and system information, download/upload files and other malware, perform DoS, etc)[11]; *dynamer*, a stealthy malware family[12]; and *bestafera*. One interesting thing to point out is the fragmentation occurred in 2016: there are almost 50% of samples present in different families that have less than 2% of the total samples each.

4.3 Dynamic Libraries Distribution

As discussed in Section 3.1.2, we also have the list of dynamic libraries used by every sample (both malware and goodware). In this section, we investigate how the samples use these libraries: what are the most used libraries for both classes together, and when we consider goodware and malware separated.

[9] https://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=TrojanSpy:Win32/Banbra

[10] https://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=TrojanDownloader:Win32/Dapato

[11] http://fortiguard.com/encyclopedia/virus/7278509

[12] https://www.microsoft.com/security/portal/threat/encyclopedia/Entry.aspx?Name=Trojan:Win32/Dynamer

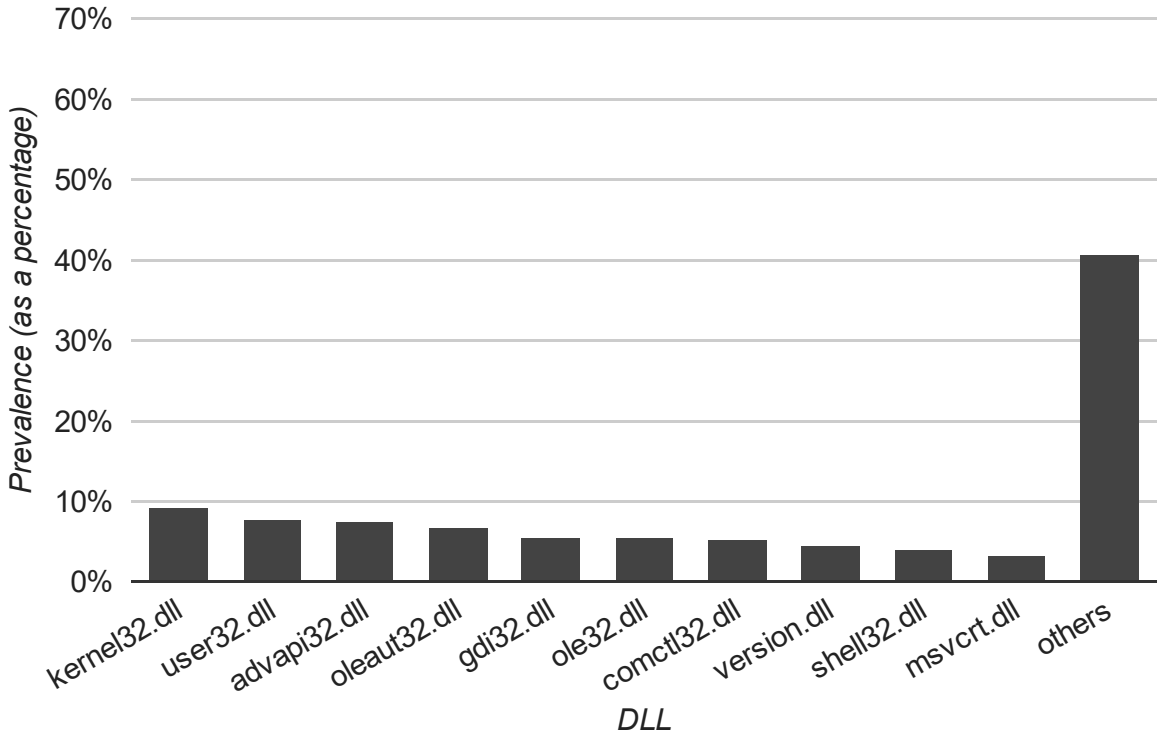


Figura 4.4: Distribution of dynamic libraries used by all the collected samples (malware and goodware).

4.3.1 General Distribution

First, we analyze our whole dataset, checking what the most used libraries are for both classes: goodware and malware. In total, 1,372 libraries are used by the collected samples. As shown in Figure 4.4, *kernel32.dll* is the most used library, being prevalent in 9.16% of the samples, followed by *user32.dll* (7.68%), *advapi32.dll* (7.53%), *oleaut32.dll* (6.73%), *gdi32.dll* (5.61%), *ole32.dll* (5.38%), *comctl32.dll* (5.33%), *version.dll* (4.55%), *shell32.dll* (4.02%) and *msvcrtl.dll* (3.19%). The remaining dynamic libraries amount to 40.82% of usage, showing a great fragmentation.

4.3.2 Goodware Distribution

Focusing on goodware samples only, we observe a scenario rather different than the one depicted in Figure 4.4. Figure 4.5 shows the distribution of the most used libraries by the goodware in our dataset. In total, these samples use 1.279 dynamic libraries. For this case, the fragmentation is even more evident, with 61.33% of libraries present in less than 3% of the samples. Again, the most used library is *kernel32.dll*, present in 7.84% of the goodware, followed by *msvcrt.dll* (5.95%), *user32.dll* (5.53%), *advapi32.dll* (5.39%), *oleaut32.dll* (3.87%), *ole32.dll* (3.75%), *ntdll.dll* (3.22%), and *mscoree.dll* (3.12%).

4.3.3 Malware Distribution

Figure 4.6 shows the distribution of the most used libraries for the malware samples from our dataset. The malware samples use a total of 226 distinct libraries and, as expected, there is heterogeneity too, but in a smaller scale, since the libraries used by less than 2% of the samples in the graph amount to 19.15% of all malware samples. Also, it's possible to observe a

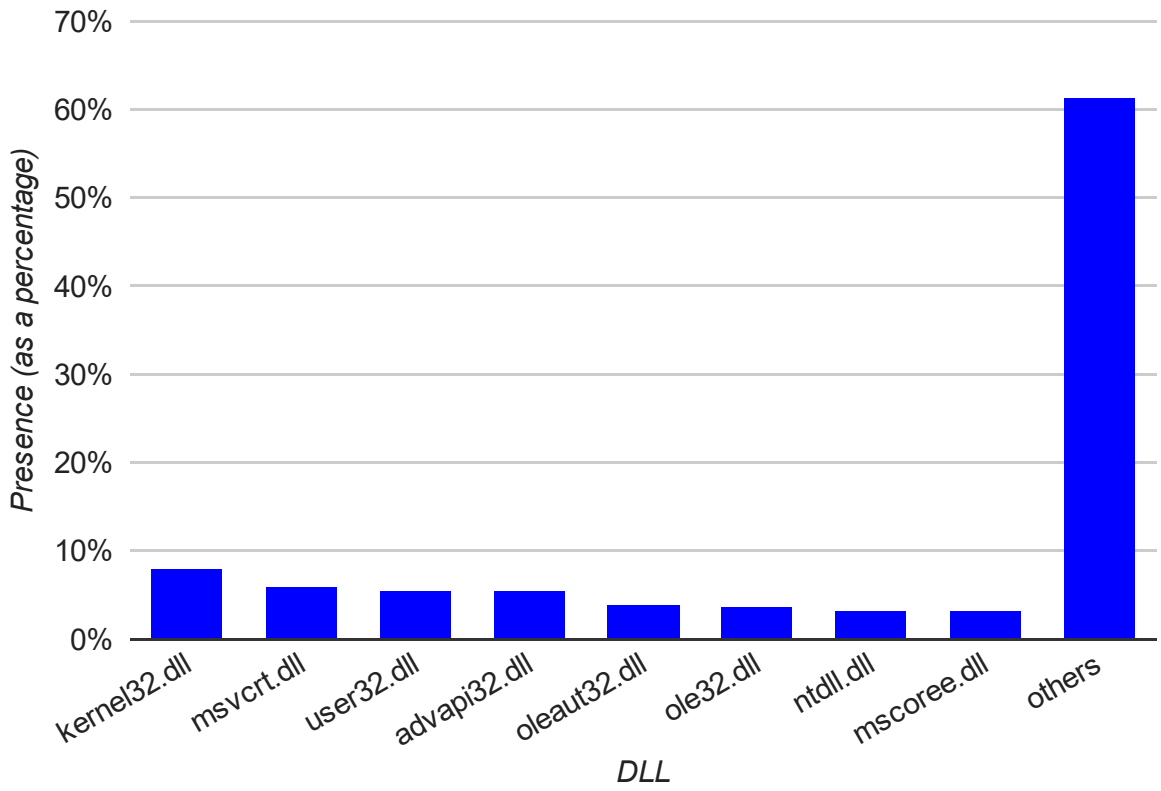


Figura 4.5: Distribution of dynamic libraries used by the collected goodware.

large prevalence of the libraries *kernel32.dll* (10.42%), *user32.dll*, (9.6%), *advapi32.dll* (9.44%), *oleaut32.dll* (9.2%), *gdi32.dll* (8.41%), *comctl32.dll* (7.89%), *version.dll* (7.36%), *ole32.dll* (6.82%), *shell32.dll* (5.36%), *wininet.dll* (3.80%) and *winspool.drv* (2.56%).

4.3.4 Goodware Versus Malware Distribution

Analyzing both classes side by side, it is possible to highlight their differences, considering the libraries most used by them. According to Figure 4.7, there are more diversity in the libraries used by goodware, since the category *others* is present in 52.4% of the samples. Further, we found that goodware uses a total of 1,276 different libraries. In contrast, malware samples libraries are not as heterogeneous as goodware: the category *others* is present in 17.47% of the malware samples and the samples use a total of only 226 different libraries. Furthermore, there are common libraries between the most prevalent of libraries used by malware and goodware: *kernel32.dll*, *user32.dll*, *advapi32.dll*, *oleaut32.dll* and *ole32.dll*. These are standard Windows libraries commonly used by every program, independent of the program intent. However, the comparison evidences that the dynamic libraries used by a program may be used to distinguish malware from goodware.

4.4 Functions Distribution

In this section, we analyze the most used functions in the used libraries when we consider the whole dataset and when we consider malware and goodware separately and side-by-side.

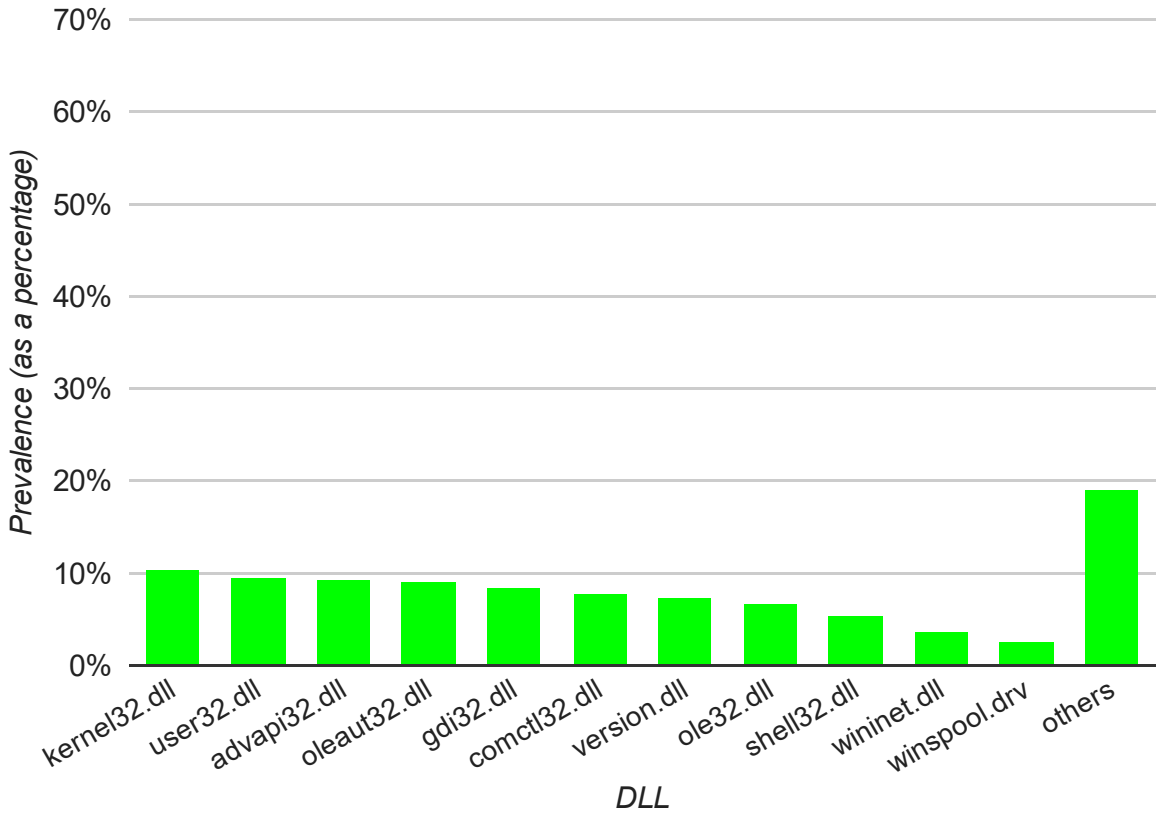


Figura 4.6: Distribution of dynamic libraries used by the collected malware.

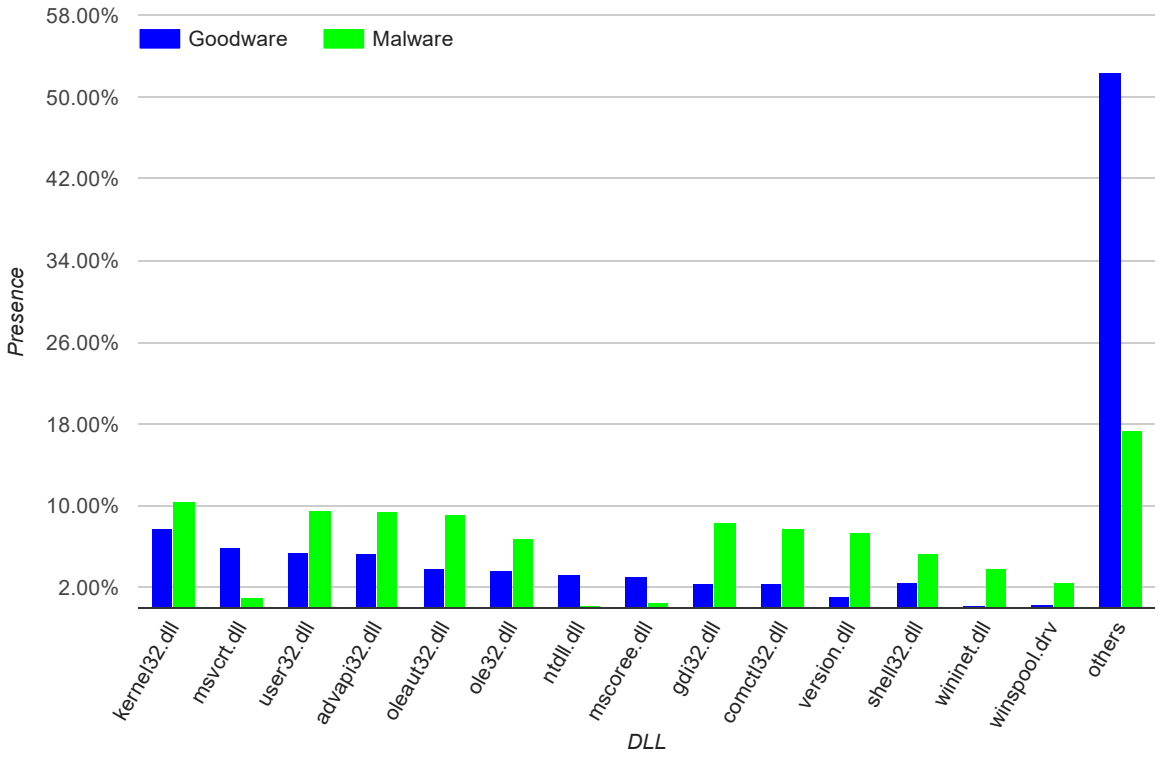


Figura 4.7: Comparison of the dynamic libraries distribution for goodware and malware.

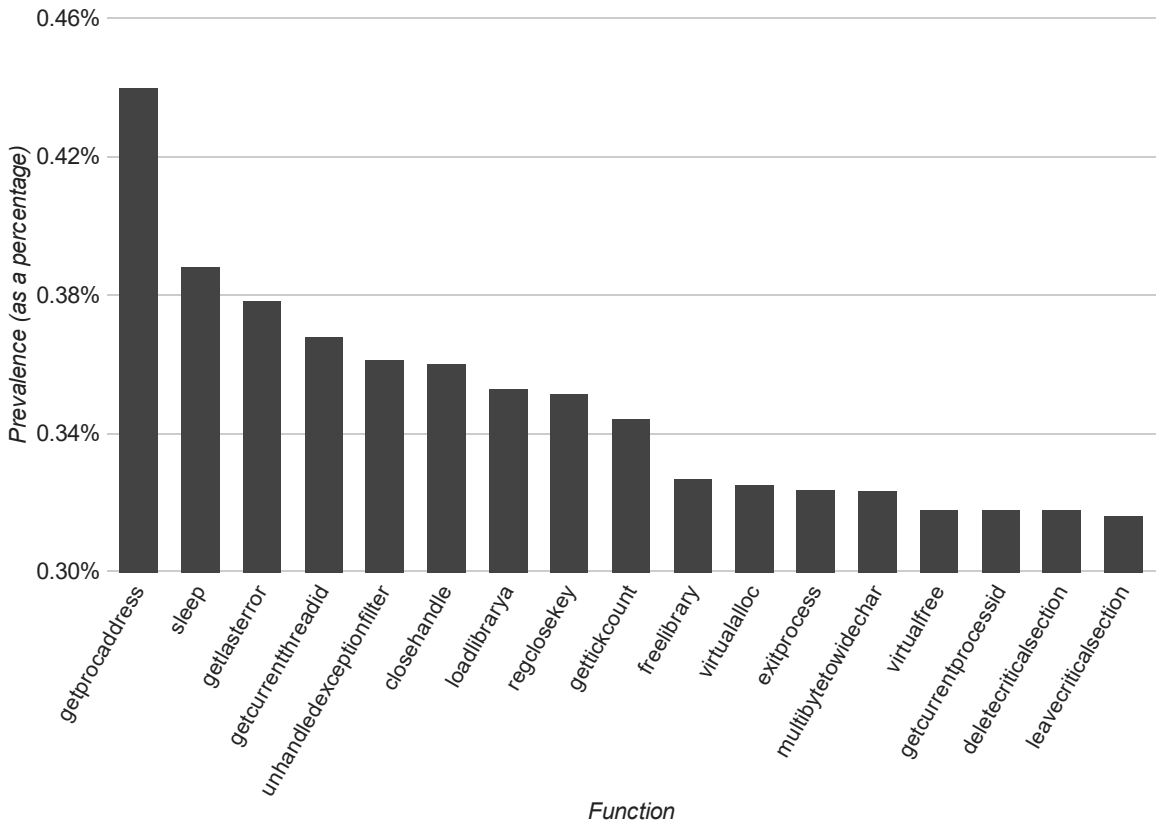


Figura 4.8: Distribution of the functions used by the collected samples.

4.4.1 General Distribution

Here we analyze the most used functions for all the samples (malware and goodware) present in our dataset. In total, 52,011 different functions are used by the samples. Figure 4.8 shows that the most used function is *getprocaddress*, implemented by *kernel32.dll* (the most common library) and this function is invoked in 0.44% of the samples. Also, *sleep* (0.39%), *getlasterror* (0.38%), *getcurrentthreadid* (0,37%), *unhandledexceptionfilter* (0.36%), *closehandle* (0.36%), *loadlibrarya* (0.35%), *gettickcount* (0.34%), *freelibrary* (0.33%), *virtualalloc* (0.33%), *exitprocess* (0.32%), *multibytetowidechar* (0.32%), *virtualfree* (0.32%), *getcurrentprocessid* (0.32%), *deletecriticalsection* (0.32%) and *leavecriticalsection* (0.32%) are implemented by *kernel32.dll*. The only function that appears between the most used function set and is not implemented by *kernel32.dll* is *regclosekey* (0.35%), which is implemented in *advapi32.dll*.

4.4.2 Goodware Distribution

Figure 4.9 shows the functions used by the goodware samples. Function *getprocaddress* is also the most used function, with 0.38% of prevalence. From the total of 47,768 functions, there are, again, a great prevalence of functions implemented by *kernel32.dll*: *loadlibrarya* (0.38%), *virtualalloc* (0.34%), *virtualfree* (0.34%), *exitprocess* (0.34%), *getlasterror* (0.28%), *sleep* (0.28%), *closehandle* (0.28%), *multibytetowidechar* (0.28%), *writefile* (0.27%), *widechar-tomultibyte* (0.27%), *freelibrary* (0.27%), *readfile* (0.27%), *getcurrentthreadid* (0.27%) and *getstdhandle* (0.27%). The function *regclosekey* (implemented by *advapi32.dll*) shows up again, with 0.29% of presence, along with *getdc* (0.28%), implemented by *user32.dll*.

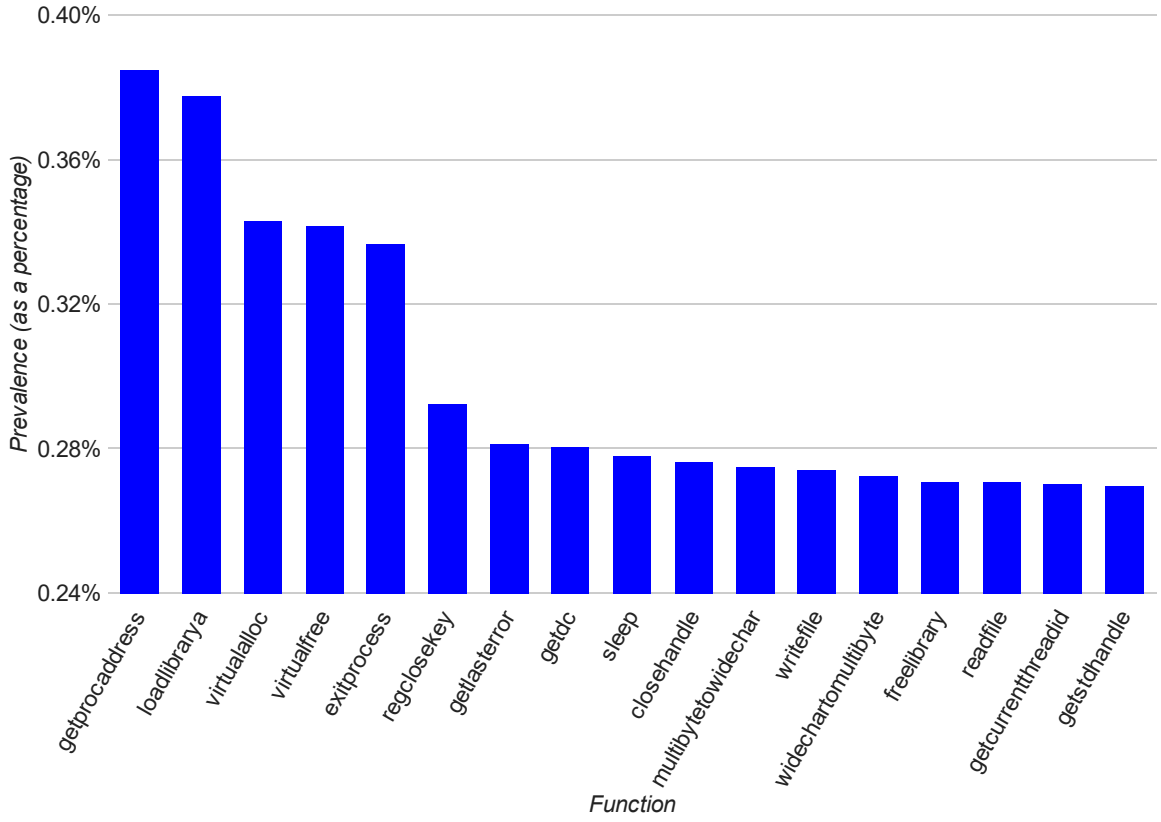


Figura 4.9: Distribution of the functions used by the collected goodware.

4.4.3 Malware Distribution

Figure 4.10 shows the functions used by the malware samples. In a total, there are 10, 546 different functions used by malware. We can see again the function *getprocaddress*, implemented by *kernel32.dll*, among the most used with 0.64% of prevalence. Other functions implemented by *kernel32.dll* are also present: *sleep* (0.62%), *gettickcount* (0.58%), *getlasterror* (0.58%), *getcurrentthreadid* (0.57%), *queryperformancecounter* (0.56%), *unhandledexceptionfilter* (0.55%), *getcurrentprocessid* (0.54%), *getsystemtimeasfiletime* (0.53%), *terminateprocess* (0.53%), *closehandle* (0.53%), *setunhandledexceptionfilter* (0.52%), *getprocaddress* (0.51%) and *freelibrary* (0.42%). Also, three other functions are also present in the list of most used functions by malware: *initterm* (0.46%) and *amsgexit* (0.43%), both implemented by *msvcrt.dll*, and *regclosekey* (0.45%), implemented by *advapi32.dll*.

4.4.4 Goodware Versus Malware Distribution

Figure 4.11 compares the most used functions by gooodware and malware samples. The function *getprocaddress* is the most used by both classes of software. The comparison of these two categories also shows that the most used functions by malware and goodware differ. For example, function *unhandledexceptionfilter* is prevalent in only of 0.06% of goodware but in 0.52% of malware. Also, function *getsystemtimeasfiletime* showed a prevalence of only 0.06% in goodware, but 0.53% in malware. Function *terminateprocess* is prevalent in only 0.09% of goodware, but in 0.53% in malware. This also provides evidence that the type of functions used by a program may be used to distinguish malware from goodware.

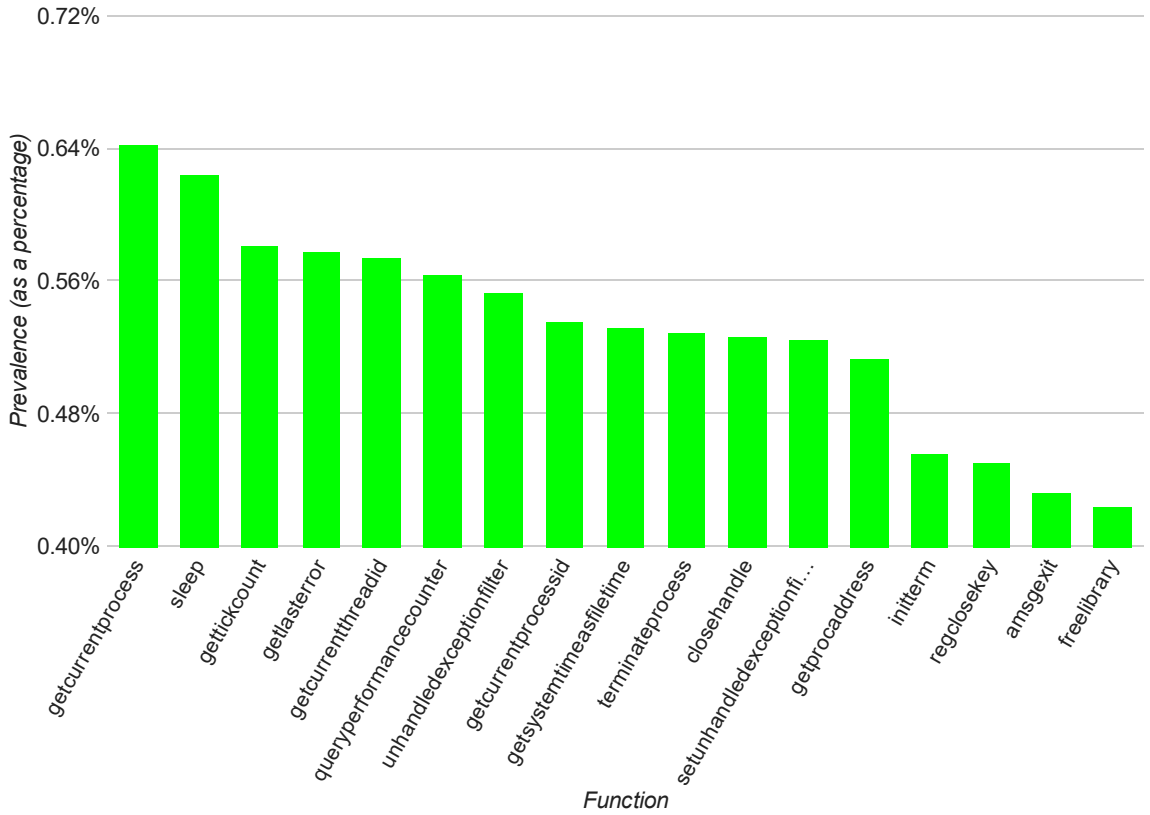


Figura 4.10: Distribution of the functions used by the collected malware.

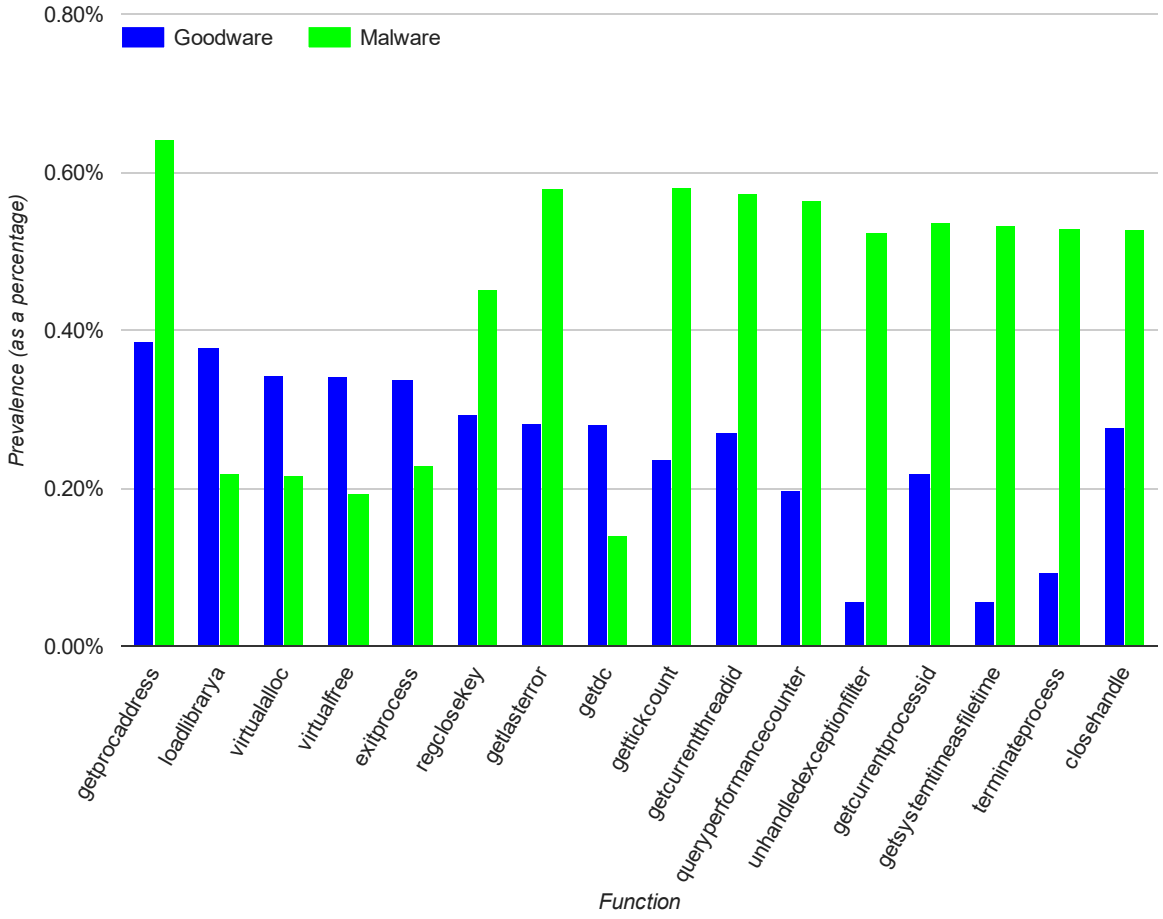


Figura 4.11: Comparison of the functions distribution of goodware and malware.

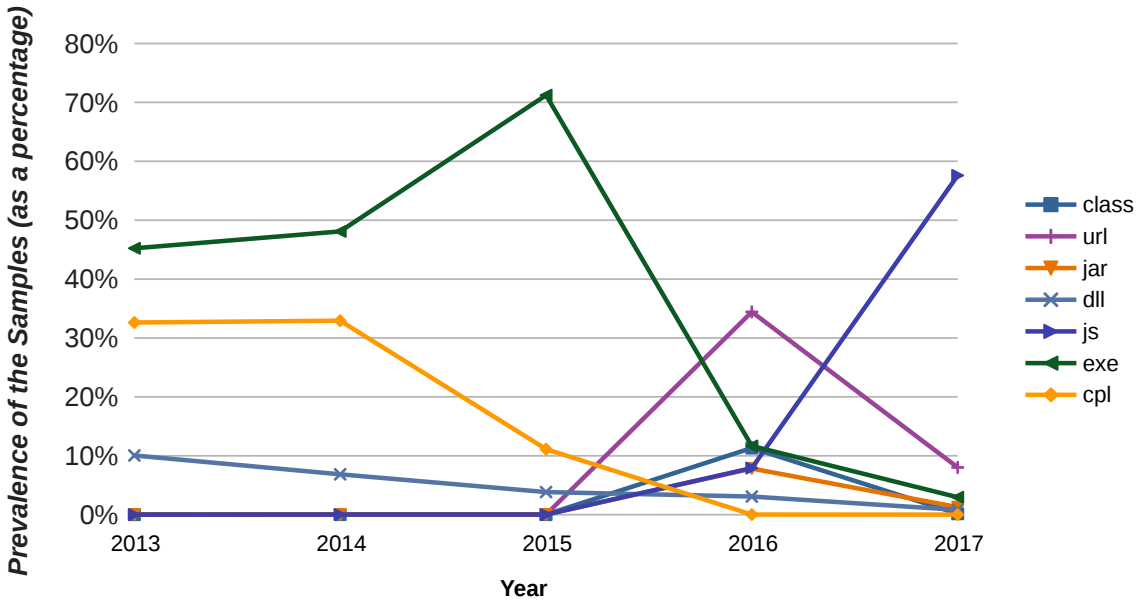


Figura 4.12: Distribution of malware file types over the years.

4.5 Malware File Types Distribution

Now taking a look at malware file types, Figure 4.12 shows their distribution over the years, from 2013 to 2017. The Windows executable format (*exe*) is the most predominant from 2013 to 2015, where it reaches its peak, with more than 70% of prevalence. After that year, its prevalence drops widely, reaching only about 3% of prevalence. The *cpl* (Control Panel file) format has a similar behavior, however its prevalence starts to drop one year before, in 2015 (in 2014, it reaches its peak, with about 32% of prevalence), dropping to almost 0% in 2017. The dynamic libraries (*dll*) have prevalence in 10% of the samples in 2013, having a considerable drop thought the next years, reaching less than 1% of prevalence in 2017. A great presence of the *url* (Universal Resource Locator, an internet shortcut file) file format is shown in 2016, with almost 35% of prevalence, having less then 1% before that and about 8% in the next year (2017). Java file types, more specifically *jar* (java archive file) and *class* (java virtual machine class file) are also between the most popular, with almost 8% and 12% of prevalence in 2016, respectively. Before that, they are similar to *url* file format, both of them with less than 1% of prevalence. In 2017, however, *jar* file format has about 1% of prevalence, with less than 1% of *class* file format. The javascript file format (*js*) has less than 1% before 2016, when it starts to increase its prevalence, with almost 8% of prevalence, increasing it massively in the next year, reaching almost 58% of prevalence.

Capítulo 5

Experiments

Using the representations (set of features) extracted in the steps shown in Section 3, we performed experiments with four classifiers: Multi-Layer Perceptron (MLP), the most frequent type of neural network used for classification purposes; Support Vector Machines (SVM); k-Nearest Neighbors (KNN), a distance based supervised clustering algorithm; and Random Forest, an ensemble of decision trees. The Multi-Layer Perceptron used is implemented by TFLearn, a high level API built on top of TensorFlow, the Google artificial intelligence library Damien et al. (2016). For these experiments, we considered an MLP that consisted of two hidden layers – the first one with $V/2$ neurons, and the second one with $V/3$, where V is the vocabulary size used by the Vector Space Model. The remaining classifiers used are implementations from the Scikit Learn library Pedregosa et al. (2011), where the chosen SVM (SVC implementation) was set to use linear kernel (default $C = 1$)[1], while KNN was with the value of K fixed in 5. We tried Several other parameters values, but the mentioned ones were those that got better overall results. In Subsection 5.1, we present the overall results obtained, followed by experiments with a predefined threshold (Section 5.2), concept drift experiments (Section 5.3) and seasonality experiments (Section 5.4).

5.1 Traditional Experiment

To validate our experiments, we used 50% of the goodware and malware samples in the testing set, while the remaining were used in the training set. We repeated this procedure ten times using cross-validation with ten partitions (folds). The size of the vocabulary V used was of 100 words. Note that this approach is considered traditional in the literature, with focus in obtaining accuracy rates close to 100%.

In Table 5.1, we present the average values obtained in this training/testing process for the measures of accuracy, recall, precision and F1-score (the harmonic mean of precision and recall) – classic measures for binary classification problems (malware/goodware) where there is class imbalanced – for each one of the four applied classifiers. It is possible to note that the Random Forest classifier achieved the best overall result, with better values for accuracy, f1-score and precision. Random Forest is usually successfully applied to other classification problems that relies on security data, such as offensive tweets, malware in general, de-anonymization, malicious Web pages, spam, among others Chatzakou et al. (2017); Mariconti et al. (2017, 2016); Caliskan-Islam et al. (2016); Stringhini et al. (2013); Canali et al. (2011); Stringhini et al. (2010). It was followed by the KNN, MLP and SVM, which was almost three percentage points bellow in

[1]We did experiments with RBF kernel and grid-search, but the results were not promising.

Tabela 5.1: Average results (10 executions) obtained by SVM, MLP, KNN and Random Forest classifiers.

Classifier	Accuracy	F1-score	Recall	Precision
SVM	95.45%	95.56%	93.11%	98.15%
MLP	97.57%	97.70%	97.69%	97.71%
KNN	97.65%	97.76%	97.51%	98.02%
Random Forest	**98.26%**	**98.34%**	**97.81%**	**98.87%**

almost all metrics. However, the presented metrics may not be interesting for real applications in this area, since false-positives can directly influence on users actual protection. This happens due to the fact that if the classifier causes a legitimate software being blocked/quarantined, it would still represent a precision of 100%. Also, this is not a good approach to evaluate this problem, since cross-validation takes into account "samples from future", i.e., data from different epochs are mixed in the training and validation sets, which can help the classifier to obtain better results (ignoring the concept drift problem). Both problems presented are addressed in the following Sections, while the Subsection below presents a way to tune the textual features extracted, aiming to improve the classifiers accuracy.

5.1.1 Tuning Textual Features

Instead of using a fixed number of most used vocabulary words V, in this Subsection we explore a new way to use the vocabulary of each textual feature using the document frequency. As already mentioned in Section 2.1.1, it's necessary to remove the stop words, words that do not have meanings in a text, in an information retrieval problem. However, in our case, our words do not have a meaning, i.e., we can not measure if they are important or not for our context – there's no set of stop words. If we think in system calls, there are ones that are used by almost all programs, just to write something for the user, for example, and do not have malicious behaviour. Thus, this kind of information is not useful to distinguish malware from goodware, but it's impracticable to build a list of libraries, system calls or compilers that are considered stop words, since it would be necessary to study which words to ignore, even with new samples appearing over the time. To work around this problem we used a lower and upper limit to ignore the terms whose document frequency is outside their range. Both parameters are *min_df*, which ignores terms that have a document frequency strictly lower than the given threshold, and *max_df*, which ignores terms that have a document frequency strictly higher than the given threshold Pedregosa et al. (2011). With that, we can ignore the most common and the rarest libraries, functions and compilers, depending on the values defined, and study the impact of this technique in a machine learning model, more specifically, random forest, the classifier that got the best results in previous Section. We tested several values for both thresholds and all the possible combinations of textual features (libraries, system calls and compilers) – combined with the numerical ones – using the same method presented in the previous Section (ten-fold cross-validation). The three best results are listed in Table 5.2. Note that using a *max_df* of 60% and 45% and a *min_df* of 15%, using only libraries (DLLs) and their combination with symbols (system calls), respectively, we achieved the same accuracy and f1-score (98.70% and 98.77%), increasing both results presented in Table 5.1 by $\approx 0.5\%$. The main difference between both cases is that using only DLLs, we achieved a better precision (99.12% versus 99.01%) and a worst recall (98.42% versus 98.52%). Using all the textual features, a *max_df* of 30% and a *min_df* of 10%, we achieved an accuracy of 98.67%, a f1-score of 98.73%, a recall of 98.48% and a precision of 98.98%. All the results

Tabela 5.2: Top three best results using thresholds based on terms document frequency.

Textual Features	DLLs, Identifiers and Symbols	DLLs	DLLs and Symbols
Max DF	30%	60%	45%
Min DF	10%	15%	15%
Accuracy	98.67%	98.70%	98,70%
F1-score	98.73%	98,77%	98,77%
Recall	98.48%	98.42%	98.52%
Precision	98.98%	99.12%	99.01%

presented in this experiment prove that using these thresholds help to improve the classifier performance. Using only libraries as textual features helped to prevent false positives (higher precision), while its combination with symbols helped to prevent false negatives (higher recall).

5.2 Experiments using Thresholds

In a scenario where a user installs a benign program, the running anti-virus must be flexible enough to do not classify this new software as malware. Hence, the number of false-positives should be decreased even that it means sacrificing the identification of some malware. In machine learning, this can be achieved using a threshold, which defines a minimum value for an item to be effectively labeled. Hence, we evaluated the classifiers' performance with distinct thresholds to decrease the number of false-positives. In Figures 5.1(a), 5.1(b), 5.1(c) and 5.1(d), we present the FPR (False-Positive Rate) and FNR (False-Negative Rate) for the SVM, Multilayer Perceptron, KNN and Random Forest classifiers, respectively (thresholds range between zero and one). As expected, the higher the threshold, the higher the number of false-negatives, i.e., the number of undetected malware grows as they are classified as goodware. On the other hand, while the false-negatives increase, the false-positives decrease, reducing the risk of classifying a goodware as malware. Particular observations regarding distinct thresholds T:

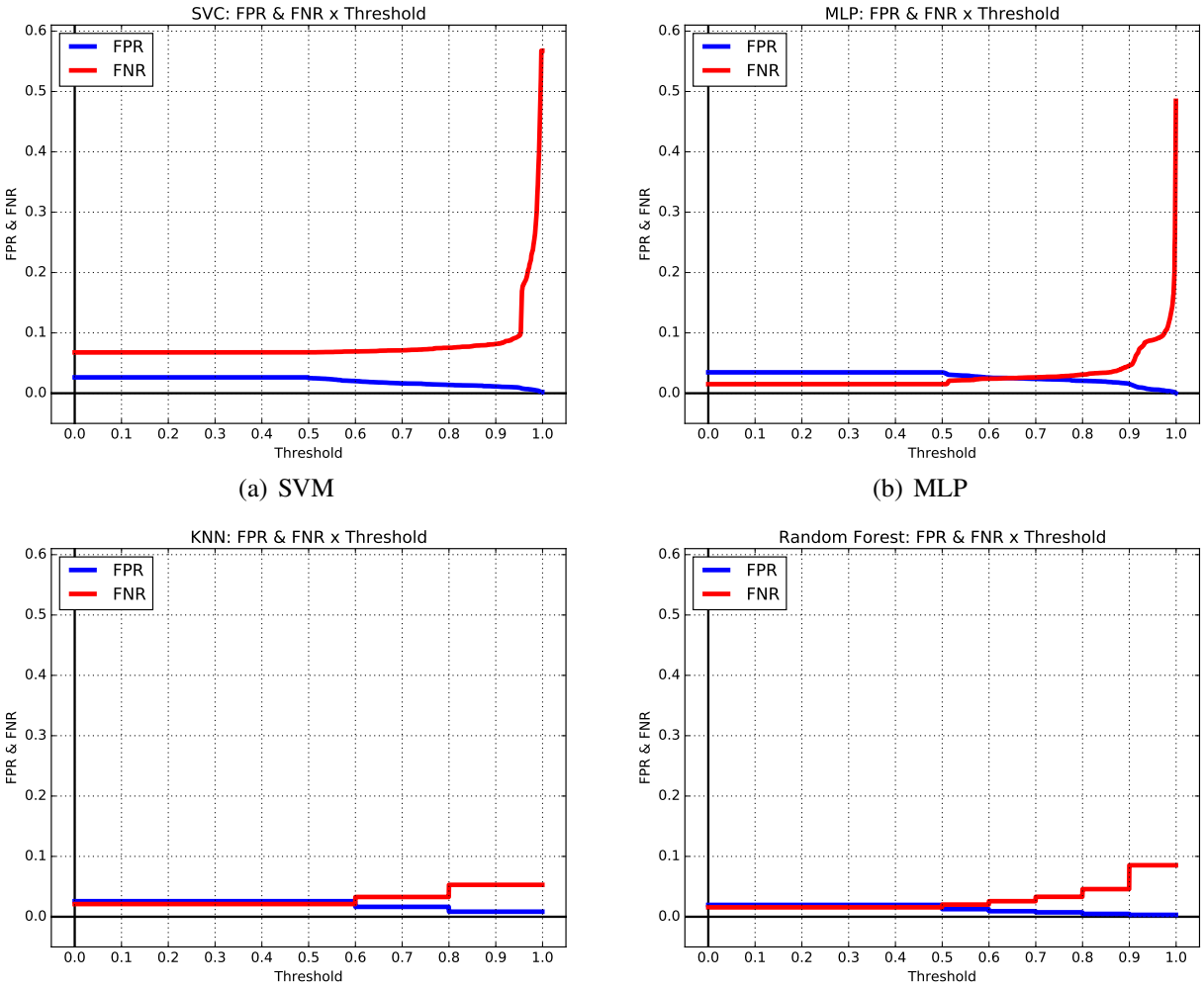
SVM's FNR starts to increase and its FPR starts to decrease when T is close to 50%. When T is close to 95%, the FNR starts to increase drastically, achieving a value close to 55% when T is close to 100%, while the FPR achieves a value close to 0%.

MLP's FNR starts to increase drastically when T is close to 90% (value close to 8%). However, in this case, the FPR is already close to 0%, showing a good result from this classifier.

KNN presented more consistent results. Its FPR starts to decrease from T = 60% and remains stable up to 80%, when the value drops to ≈0% and remains stable. The opposite can be observed in the FNR: the values keep stable at around 5% from T =80% on.

Random Forest seems to have properties even more interesting for this problem, given that at T =50%, its FPR and FNR are below 5% (really close to 0%). However, from T =90% on, although the FNR stays close to zero, the FNR almost reaches 10%.

In Figure 5.2, we show the ROC curve of the classifiers results. It clarifies what has already been previously reported in the figures: due to the threshold increase, the FPR will be lower, i.e., less samples of goodware will be misclassified. However, the side effect on the TPR is that it tends to decrease. The curve also shows the consistency of Random Forest, which maintains a low FPR and still have a TPR above 90% even with very high thresholds. The KNN behavior is also stable (it manages to maintain a FPR below 2.5% and a TPR above 93%). This



(a) SVM (b) MLP

(c) KNN (d) Random Forest

Figura 5.1: FPR & FNR vs. Threshold experiments using SVM, MLP, KNN and Random Forest.

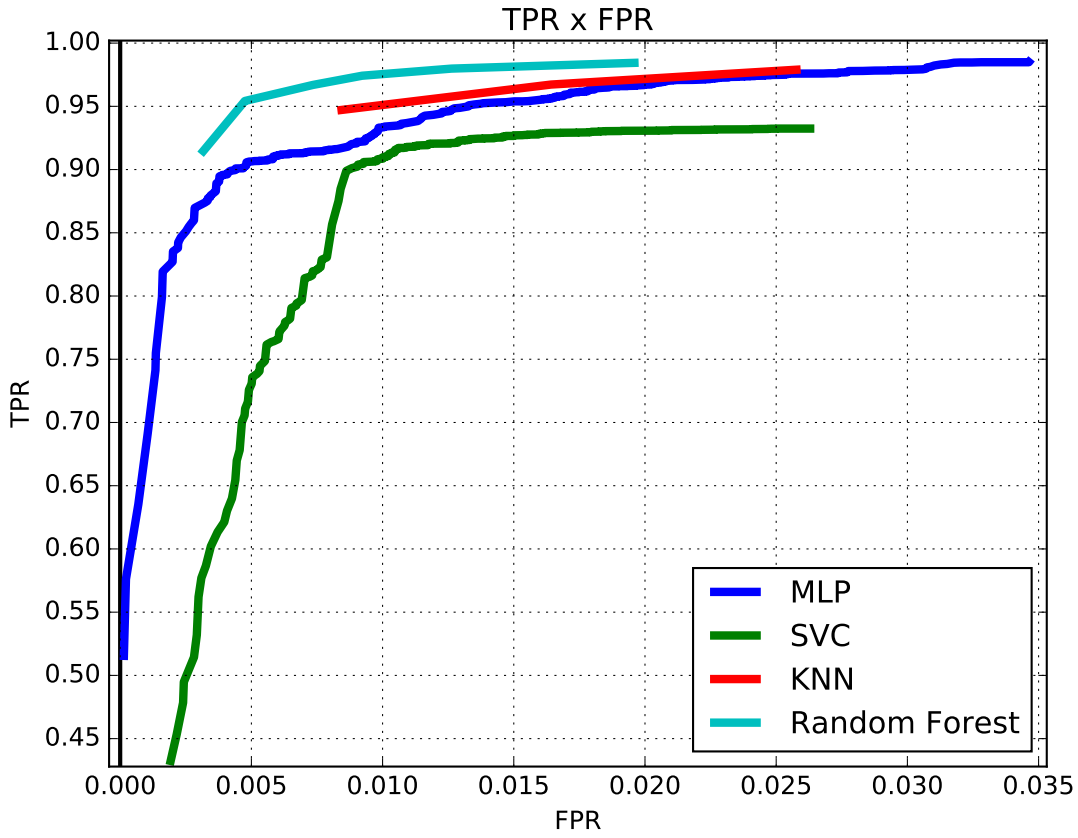


Figura 5.2: ROC curve of the classifiers results when using distinct thresholds. Random Forest achieved better tradeoff between TPR and FPR, followed by KNN, MLP and SVM.

shows that both classifiers present great certainty about the majority of the learned data. In the other hand, MLP achieved a TPR closer to Random Forest's. However, its FPR is closer to 3.5% (Random Forest achieves this rate with an FPR lesser than 2%). The worst results were achieved with SVM – maximum TPR closer to 95%, FPR greater than 2.5%. Nonetheless, if we think of an "ideal" classifier for the commercial scenario (e.g., anti-virus), when a goodware would never be classified as a malware, the MLP would also exhibit this property but with a false-negatives rate ($FNR = 1 - TPR$) close to 35%.

5.3 Concept Drift Experiments

To validate the concept drift problem presented in Section 5.1, we modified our training set with only samples created until July 2013 and our validation set with only samples created only in a particular month. This simulates a classifier that was trained using cross-validation (until July 2013) and then used to classify new samples in the future. The Figures 5.3(a) and 5.4(a) show the results for KNN, while the Figures 5.3(b) and 5.4(b) show the results for Random Forest, with accuracy, f1score, recall and precision. As expected, despite increasing in the following month (August 2013), the accuracy of both classifiers starts to decrease as time goes by reaching, as worst result, 78.81% of accuracy in August 2015. This result gives evidence that keeping machine-learning model out of date decreases its performance as time goes by. That is why cross-validation is not a valid approach to evaluate classifiers in this area, since it does not

represent the real scenario. Due to that, we propose two experiments that represent the reality, both shown in the next Subsections.

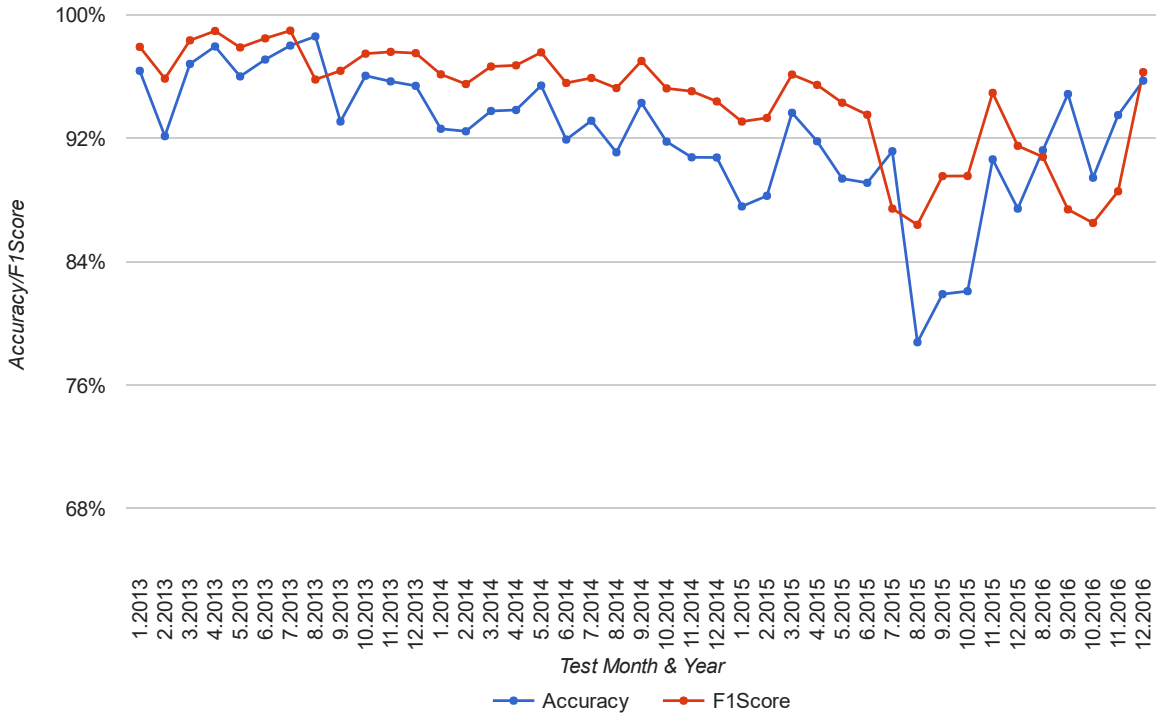
5.3.1 Experiment #1

In this experiment, we divided the samples into two groups – one containing samples created until a certain month of a year[2], used for train, and the other with samples created one month after that said month, used for validation. For example, if the month February and the year 2015, the training set contained samples that were created until February 2015 (a cumulative set), while the validation set contained samples created only in March 2015 (one month after February 2015). This simulates a real world situation, since we train with data that were already seen, i.e., malware and goodware that were created before a particular month, and test with data from the present (from the selected month). With both sets created, we used them in two off-the-shelf machine learning classifiers: KNN (*k*-nearest neighbors) and Random Forest. Figures 5.5(a) and 5.6(a) show the results for KNN, while Figures 5.5(b) and 5.6(b) show the results for Random Forest, with accuracy, f1score, recall and precision. In both cases, there is a great variance on the metrics as time goes by. In the case of KNN, the best accuracy was achieved in July 2013, with 98.61%, and the worst, in January 2013, with 78.47%. The F1score is also worst in January 2013, with 87.81%, and the best, in November 2016, with 98.78%. Looking at the recall, the worst result was also in January 2013, with 78.41%, while the best was in September 2013, with 97.99%. In the other hand, the worst precision was obtained in October 2016, with 90.38%, and the best, in February 2013, May 2013, February 2014 and November 2016, with 100%. These results show that KNN is a good classifier to prevent false positives, since it obtained 100% of precision in 4 months, and has limitations in preventing false negatives in this case, since recall never achieves 100%. In the case of Random Forest, the worst accuracy was in January 2013, with 86.50%, and the best, in July 2013, with 99.54%, while the worst f1score was 91.97% in September 2016 and the best, in September 2013, with 99.66%. Looking at the recall, the worst obtained was in January 2013, with 86.53%, while the best was obtained in September 2013, with 99.46%. The worst precision was obtained in December 2015, with 94.74%, and the best, in February 2013, December 2013, January 2014, February 2014, March 2014, June 2014, August 2016 and November 2016, with 100% in all of them. It is possible to conclude that Random Forest is a better classifier in comparison to KNN, since all the metrics were better in most cases. The significant drop and recover of all metrics through time in both classifiers evidences the existence of a concept drift in this dataset.

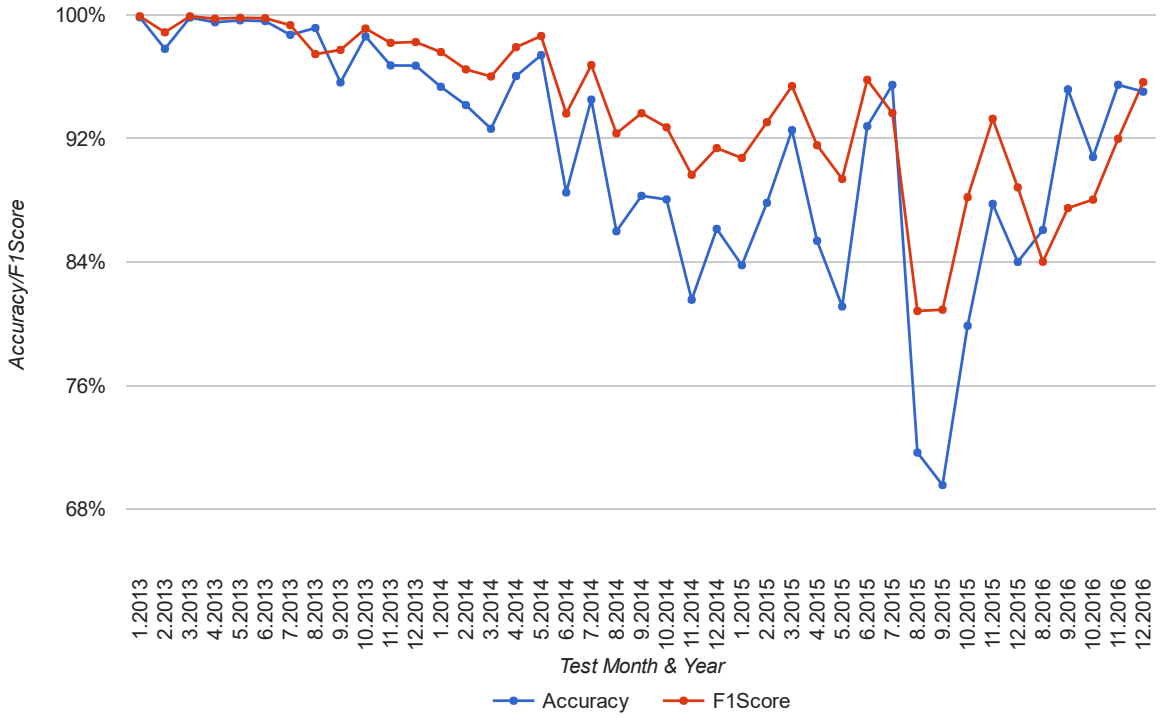
5.3.2 Experiment #2

In Experiment #2 we inverted the train and validation sets from Experiment #1. With that, the train group has samples from a certain month and year only, while the validation has samples created one month before that said month (cumulative set). For example, given the month February and the year 2015, the train set will contain samples created only that month and year, while the test set will contain samples created until a month before (January 2015). This experiment also simulates the real world, however, we use only the data from the selected month (present) to train and the others (past, before this month), for validation. With that, we can see how representative samples from a certain month are for all the data. Again, we used both sets to train the two classifiers: KNN and Random Forest. Figures 5.7(a) and 5.8(a) show the results

[2]In the case of malware, it corresponds to the year the sample was collected and that we received it from our partner; in the case of goodware, we fetched the creation date from their headers and assumed them correct.

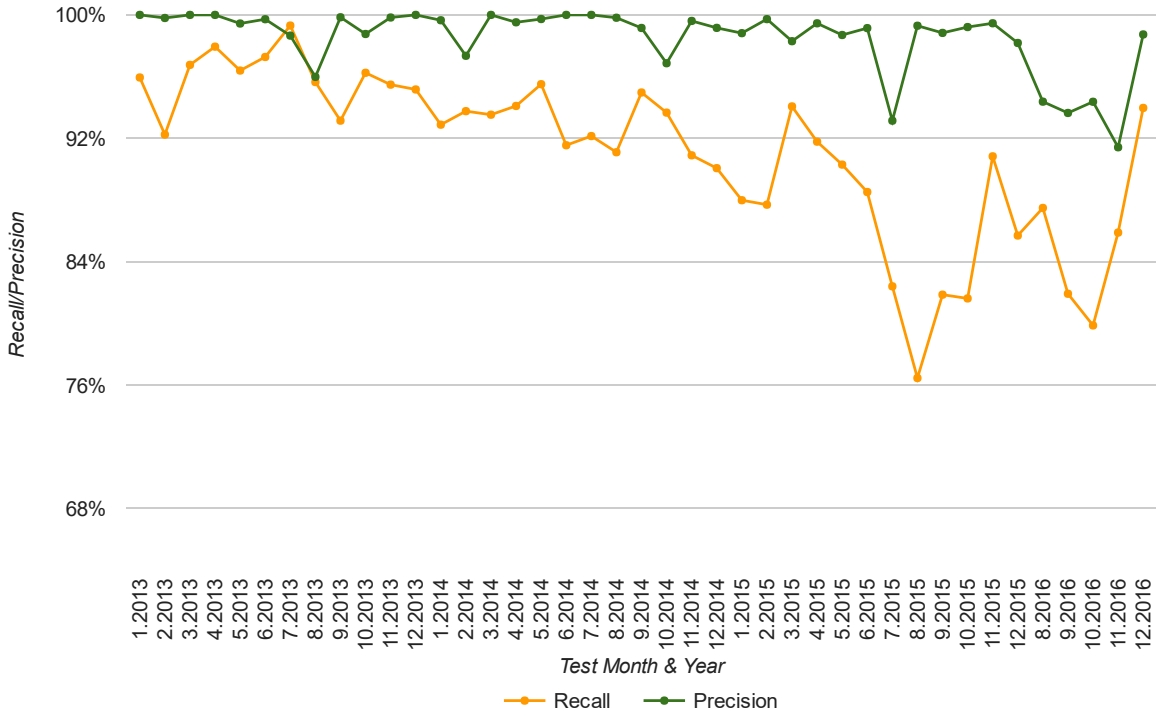


(a) Accuracy and F1Score in KNN.

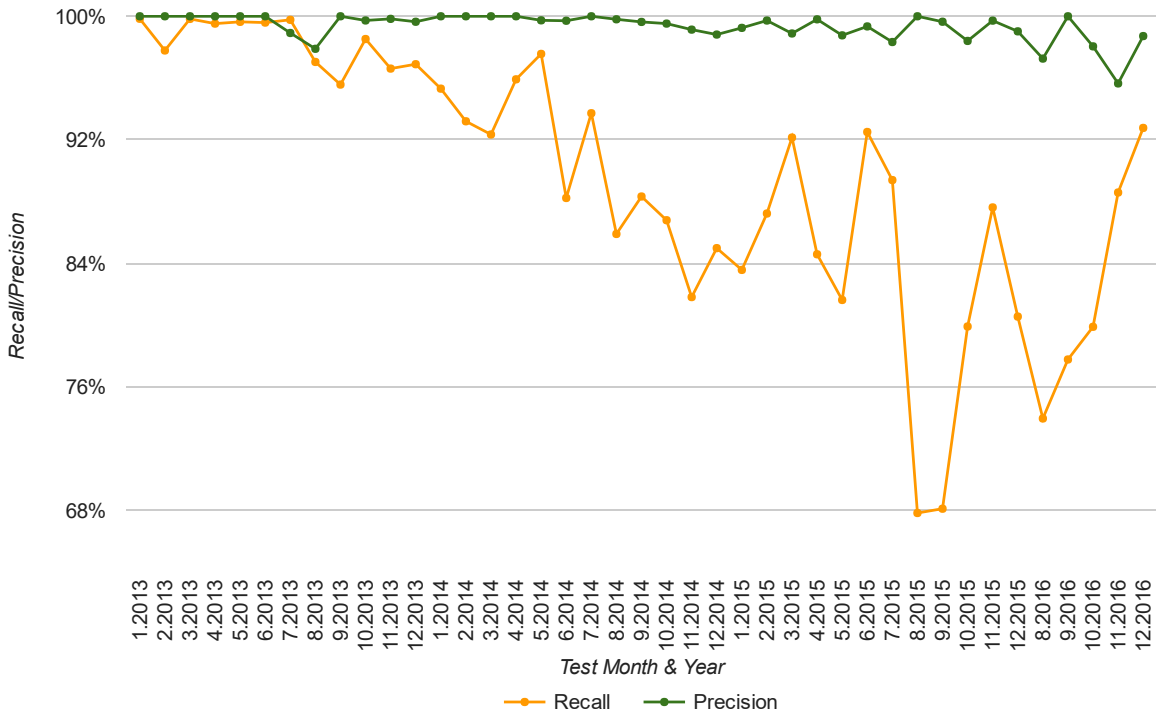


(b) Accuracy and F1Score in Random Forest.

Figura 5.3: Accuracy and f1score using samples until July 2013 to train KNN and Random Forest.



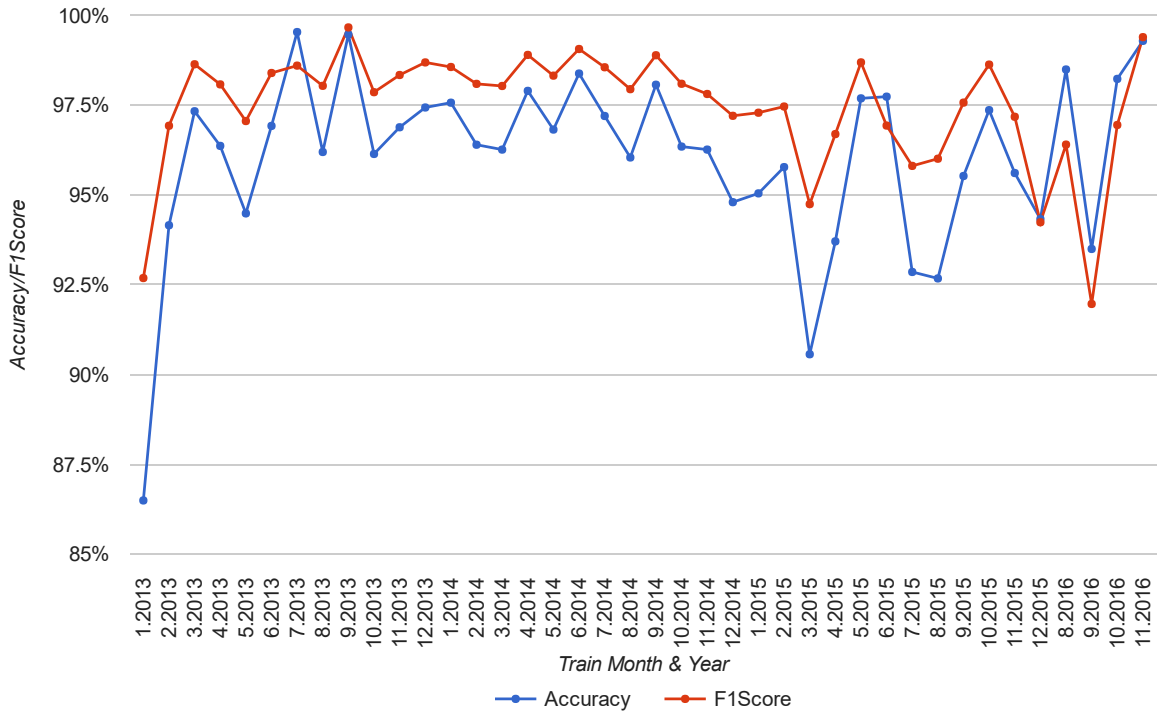
(a) Recall and Precision in KNN.



(b) Recall and Precision in Random Forest.

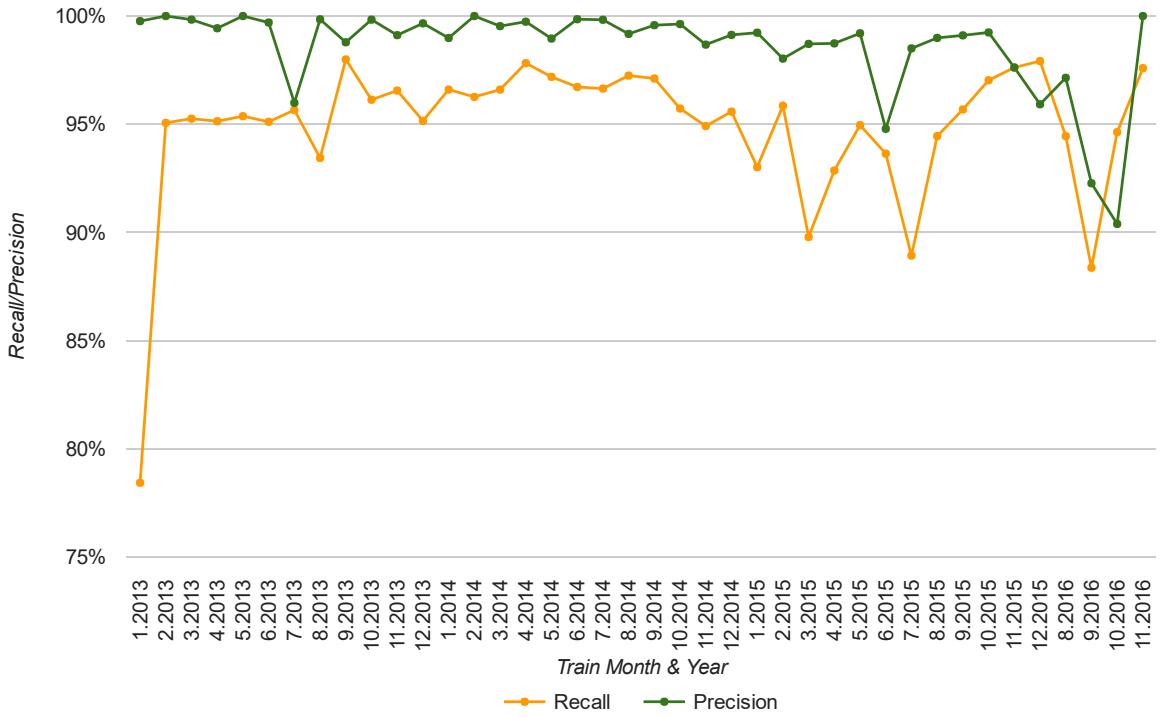
Figura 5.4: Recall and precision using samples until July 2013 to train KNN and Random Forest.

(a) Accuracy and F1Score in KNN.

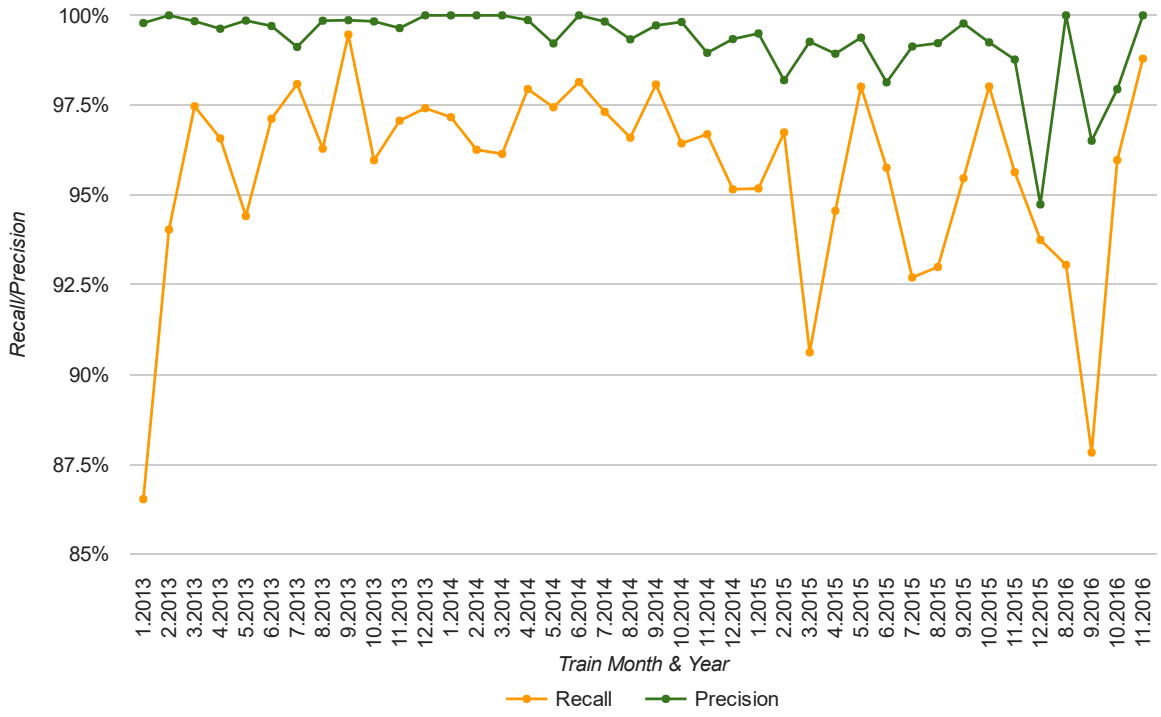


(b) Accuracy and F1Score in Random Forest.

Figura 5.5: Accuracy and f1score obtained in Experiment #1.



(a) Recall and Precision in KNN.



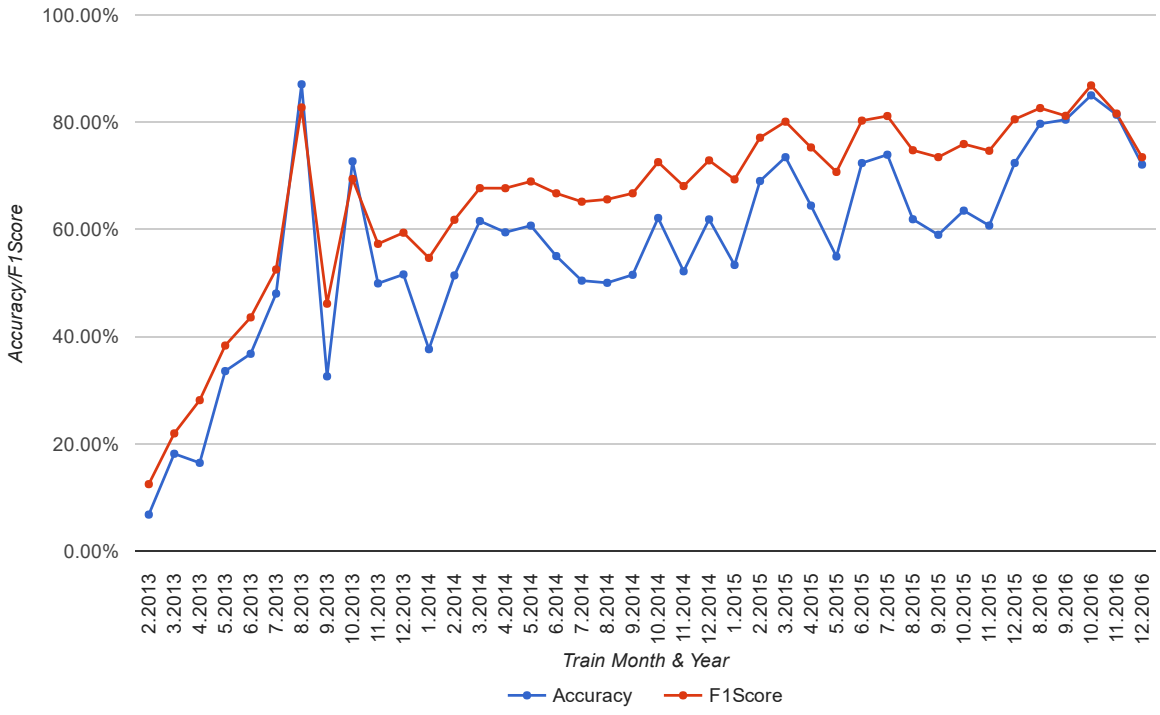
(b) Recall and Precision in Random Forest.

Figura 5.6: Recall and precision obtained in Experiment #1.

obtained by KNN, while Figures 5.7(b) and 5.8(b) show the results obtained by Random Forest, with their accuracy, f1score, recall and precision. Analyzing the results from KNN, it's possible to see a really low accuracy at the beginning (January 2013), with only 6.78%, being the best result 87.15%, in August 2013. Its best f1score was also achieved in August 2013, with 81.82%, and the worst, also in January 2013, with 12.47%. It is curious to look at recall and precision: recall achieves really good results, with 100% in January 2013, April 2013 and January 2013 (best results) and it starts to drop in December 2015, achieving 74.50% in December 2016 (worst result). Precision, in the other hand, has 6.65% in January 2013 (worst result) and it starts to increase as time goes by, despite some drops on the way, achieving 84.45% in November 2016. Even though Random Forest presents best overall results, its graphs are really similar to KNN. Random Forest achieves the best accuracy in March 2015, with 82.78%, and the worst, in January 2013, with 10.26%. Its worst f1score was also achieved in January 2013, while the best was achieved in August 2016, with 87.57%. The behavior of recall is also similar to KNN, since it starts to drop in December 2015, has values really close to 100% before that month (it never really reaches 100% as KNN, the best result was 99.99% in January 2014) and the worst result was also in December 2016, with 79.96%. Its precision is similar to KNN too, with the worst result in January 2013, with 6.88%, and the best in November 2016, with 88.36% .

5.4 Seasonality Experiments

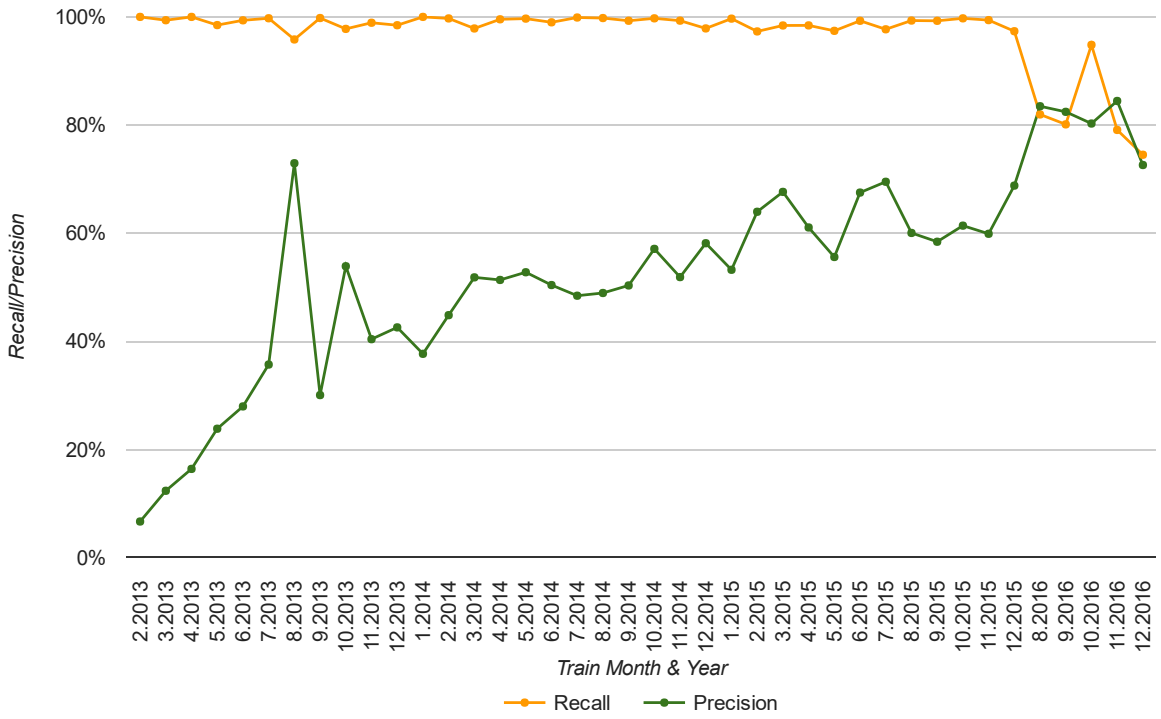
Analyzing the results obtained in previous chapter, we questioned ourselves about the possible presence of seasonality in our dataset. Seasonality is commonly present in problems like spam distribution, which generally have malware associated with them Melville et al. (2006). Despite of the fact that the spam e-mails are considered seasonal, we show in Figures 5.9(a), 5.9(b), 5.9(c) and 5.9(d) (t-SNE projection of samples year by year in January, July, September and December, respectively) that such affirmation cannot be applied to malware, i.e., only the body (the message or text) of the spams are considered seasonal and not their attachments (malware files). All the figures presented show malwares *t*-SNE projection from a single month thought the time, from 2013 to 2016 (except in January and July) and it is clear to see clusters grouping the majority of the samples that belong to a certain year. Despite of some minor overlapping of classes, we cannot generalize and say that there is seasonality in malware samples.



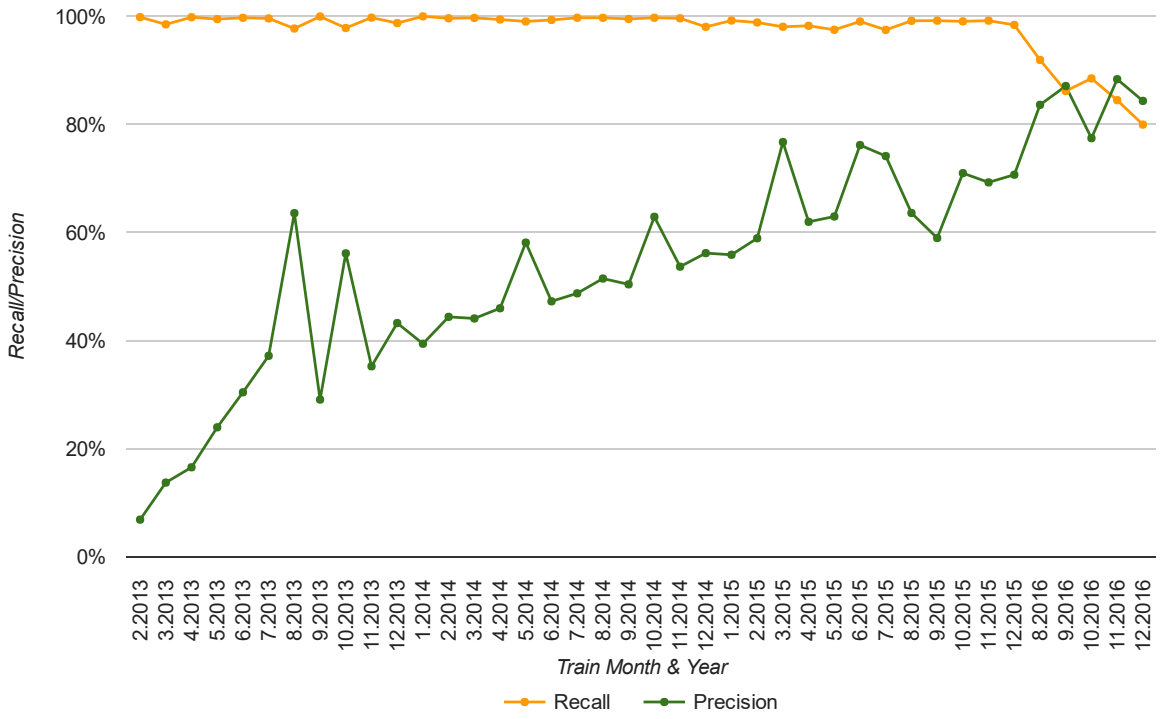
(a) Accuracy and F1Score in KNN.

(b) Accuracy and F1Score in Random Forest.

Figura 5.7: Accuracy and f1score obtained in Experiment #1.

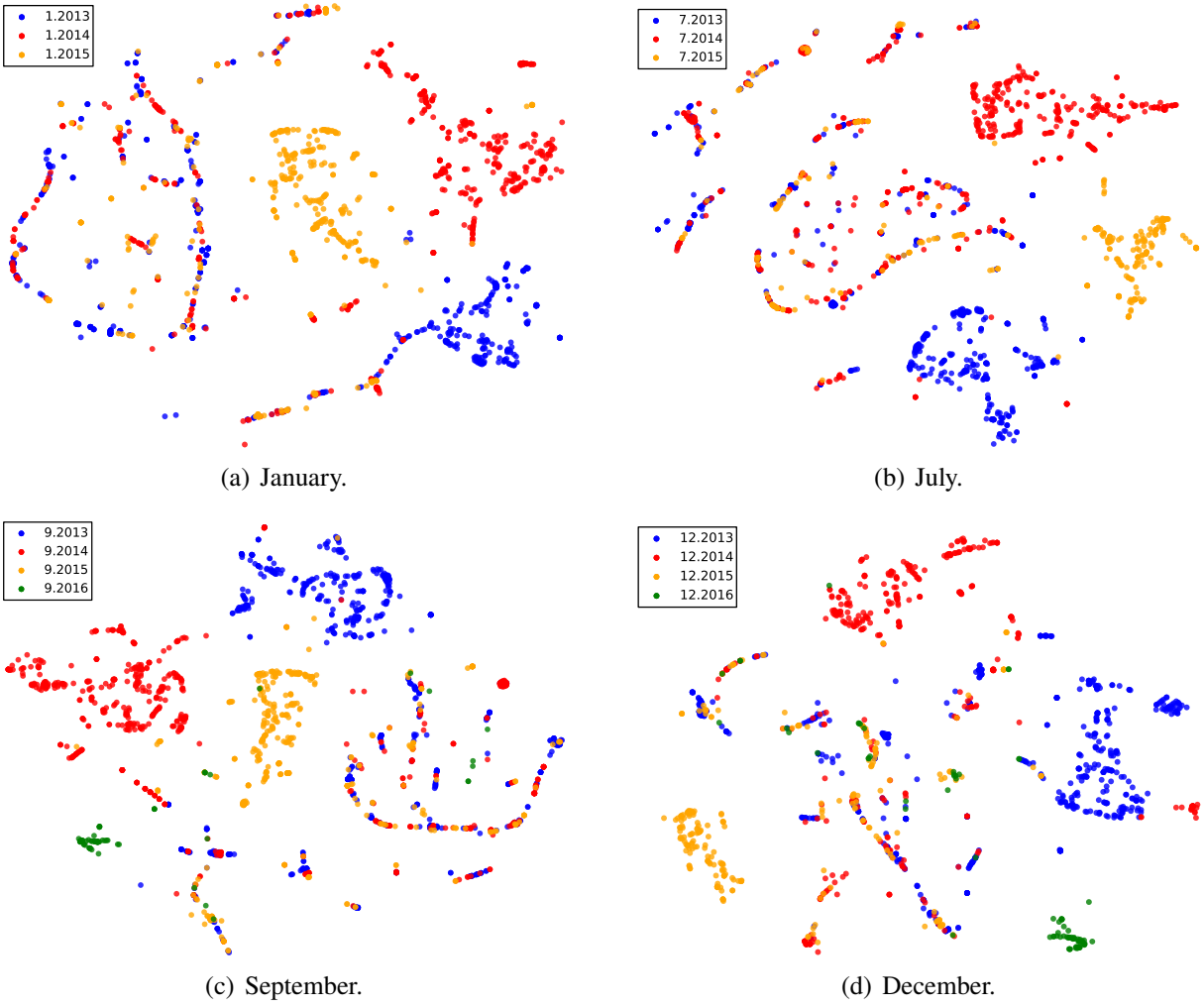


(a) Recall and Precision in KNN.



(b) Recall and Precision in Random Forest.

Figura 5.8: Recall and precision obtained in Experiment #2.



(a) January. (b) July.

(c) September. (d) December.

Figura 5.9: Projection of samples in a certain month from 2013 to 2016 using t-SNE.

Capítulo 6

Conclusion

In this work, we implemented a method to statically extract programs' attributes so as to produce comparable, labeled feature vectors. Based on these vectors, we proposed and evaluated a method for the analysis of malware classification techniques based on machine learning (SVM, MLP, KNN and Random Forest), in order to verify the classifiers' validity to correctly separate malware and goodware along the time, i.e., the classifier expiration date. We used attributes extracted from Windows PE files, either assigned to the malicious class and to the benign one. These vectors of attributes vectors and their corresponding description were obtained from the database we built for this work, which we made available to the scientific community Ceschin et al. (2016). Our goal is to allow that other researchers can conduct a fair comparison with other approaches, as well as that it is possible to reproduce the results or even to apply other techniques on this data. We can highlight the following contributions from this work:

1. A dataset containing almost 40.000 attribute vectors for malware and goodware samples.
2. We showed that malware families distribution changes as time goes by, with a growing fragmentation in the last year.
3. Also, even though sharing common libraries and functions, malware and goodware can be distinguished by them, i.e., textual features help in program classification, even more when using a lower and upper limit to ignore the terms whose document frequency is outside their range.
4. Random Forest classifier was the one that allowed us to obtain the best classification tradeoff between goodware and malware.
5. In specific scenarios (e.g., commercial or other such as the need of precision should be closer to 100%), MLP and SVM may be more interesting, as with their use it is possible to tightly control the classifier's (detector) acting point.
6. The premise of the expiration date of a classifier over time were successfully verified, validating the hypothesis about the presence of concept drift in our dataset through the two baseline experiments, which are complementary (the first one represents a real world scenario, updating the classifier with new samples as time goes by, and the second shows how representative samples from a single month are to all the collected data), showing that there is not a subset (of a single month) that represents the whole data and even updating the models as time goes by, their performance do not increase.
7. We confirmed the premise that the classic approach commonly used (ten-fold cross-validation) is not good in this specific area of malware classification.

8. We shown that it is not possible to say that malware samples are seasonal.

6.1 Future Work

As already mentioned, most research papers cited on this work do not address the concept-drift issue. Our research goals are to address this issue in an effective way, to develop advanced detectors leveraged by deep learning algorithms, and to create an up-to-date dataset that embraces as many as possible features extracted from malicious programs dynamic and statically. To do so, we will investigate how to automatically identify the occurrence of concept-drift in malware classifiers, and how to make use of execution traces from monitored programs (malicious and benign) to create deep-learning-based, real-time detection systems. With that, we also intend to answer the following questions regarding anti-malware solutions:

- What are the types of drift present in our dataset?
- When it is time to completely retrain a classifier, i.e., when it becomes outdate at the point of being unusable?
- When should we discard samples from the dataset in order to boost its performance without loosing accuracy?
- Is it possible to train behavioral models that can effectively differ between a benign and a malicious program, even in cases when the malicious behavior exhibited during the monitored execution is subtle?

The answers for these questions will drive the continuation of this work and the potential contributions of the results can help to advance the field of malware countermeasures by helping security researchers, professionals and the community to develop better solutions.

Referências Bibliográficas

Ahmadi, M., Ulyanov, D., Semenov, S. et al. (2016). Novel feature extraction, selection and fusion for effective malware family classification. Em *6th ACM Conference on Data and Application Security and Privacy (CODASPY)*, páginas 183–194.

Allix, K., Bissyandé, T. F., Klein, J. e Le Traon, Y. (2015). *Are Your Training Datasets Yet Relevant?*, páginas 51–67. Springer International Publishing, Cham.

Almeida, P., Oliveira, L., Britto, A. e Sabourin, R. (2015). Dealing with concept drifts using dynamic ensembles of classifiers. Tesis presented as partial requirement for the degree of Doctor. Graduate Program in Informatics, Sector of Exact Sciences, Universidade Federal do Paraná.

Anderson, B., Quist, D., Neil, J., Storlie, C. e Lane, T. (2011). Graph-based malware detection using dynamic analysis. *Journal in Computer Virology*, 7(4):247–258.

Anderson, B., Storlie, C. e Lane, T. (2012). Improving malware classification: Bridging the static/dynamic gap. Em *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, AISec '12, páginas 3–14, New York, NY, USA. ACM.

Annachhatre, C., Austin, T. H. et al. (2015). Hidden markov models for malware classification. *Journal of Computer Virology and Hacking Techniques*, 11(2):59–73.

Bailey, M., Oberheide, J., Andersen, J., Mao, Z. M., Jahanian, F. e Nazario, J. (2007). Automated classification and analysis of internet malware. Em *10th International Conference on Recent Advances in Intrusion Detection (RAID'07)*, páginas 178–197.

Bayer, U., Comparetti, P. M. et al. (2009). Scalable, behavior-based malware clustering. Em *the Network and Distributed System Security Symposium (NDSS)*.

Benzmüller, R. (2017). Malware trends 2017. `https://www.gdatasoftware.com/blog/2017/04/29666-malware-trends-2017`.

Breiman, L. (2001). Random forests. *Mach. Learn.*, 45(1):5–32.

Caliskan-Islam, A., Yamaguchi, F., Dauber, E., Harang, R., Rieck, K., Greenstadt, R. e Narayanan, A. (2016). When coding style survives compilation: De-anonymizing programmers from executable binaries. `https://arxiv.org/abs/1512.08546v2`.

Canali, D., Cova, M., Vigna, G. e Kruegel, C. (2011). Prophiler: A fast filter for the large-scale detection of malicious web pages. Em *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, páginas 197–206, New York, NY, USA. ACM.

Carrera, E. (2016). pefile python module. `https://github.com/erocarrera/pefile`.

Ceschin, F. et al. (2016). malware-machinelearning. https://github.com/fabriciojoc/malware-machinelearning, accessed in December 2016.

Chatzakou, D., Kourtellis, N., Blackburn, J., Cristofaro, E. D., Stringhini, G. e Vakali, A. (2017). Mean birds: Detecting aggression and bullying on twitter. https://arxiv.org/abs/1702.06877v1.

CNET (2017). CNET Downloads. http://download.cnet.com/.

COMMISSION, F. T. (2015). Malware. https://www.consumer.ftc.gov/articles/0011-malware.

Cortes, C. e Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3):273–297.

Damien, A. et al. (2016). Tflearn. https://github.com/tflearn/tflearn.

David, O. E. e Netanyahu, N. S. (2015). Deepsign: Deep learning for automatic malware signature generation and classification. Em *International Joint Conference on Neural Networks (IJCNN)*, páginas 1–8.

Firdausi, I., lim, C., Erwin, A. e Nugroho, A. S. (2010). Analysis of machine learning techniques used in behavior-based malware detection. Em *Proceedings of the 2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies*, ACT '10, páginas 201–203, Washington, DC, USA. IEEE Computer Society.

Fradkin, D. e Muchnik, I. (2006). Support vector machines for classification. *"Discrete Methods in Epidemiology", DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 70:13–20.

Gama, J. a., Žliobaitė, I., Bifet, A., Pechenizkiy, M. e Bouchachia, A. (2014). A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):44:1–44:37.

Gandotra, E., Bansal, D. e Sofat, S. (2014). Malware analysis and classification: A survey. *Journal of Information Security*, 5(2):56–64.

Gheorghescu, M. (2005). An automated virus classification system. Em *Virus Bulletin Conference*.

Ghiasi, M., Sami, A. e Salehi, Z. (2015). Dynamic vsa: a framework for malware detection based on register contents. *Engineering Applications of Artificial Intelligence*, 44(Supplement C):111 – 122.

Grégio, A. R. A., de Geus, P. L., Kruegel, C. e Vigna, G. (2012). Tracking memory writes for malware classification and code reuse identification. Em *9th International Conference, (DIMVA 2012) – Revised Selected Papers*, páginas 134–143.

Grégio, A. R. A., Fernandes, D. S., Afonso, V. M., de Geus, P. L., Martins, V. F. e Jino, M. (2013). An empirical analysis of malicious internet banking software behavior. Em *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, páginas 1830–1835, New York, NY, USA. ACM.

Haykin, S. (2009). *Neural Networks and Learning Machines*. Número v. 10 em Neural networks and learning machines. Prentice Hall.

Hu, W. e Tan, Y. (2017). On the robustness of machine learning based malware detection algorithms. Em *2017 International Joint Conference on Neural Networks (IJCNN)*, páginas 1435–1441.

Huang, W. e Stokes, J. W. (2016). Mtnet: A multi-task neural network for dynamic malware classification. Em *13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, páginas 399–418.

Islam, R., Tian, R., Batten, L. M. e Versteeg, S. (2013). Review: Classification of malware based on integrated static and dynamic features. *J. Netw. Comput. Appl.*, 36(2):646–656.

Jordan, M. I. (2017). The kernel trick, advanced topics in learning & decision making. `https://people.eecs.berkeley.edu/~jordan/courses/281B-spring04/lectures/lec3.pdf`, accessed in July 2017.

Jordaney, R., Sharad, K., Dash, S. K., Wang, Z., Papini, D., Nouretdinov, I. e Cavallaro, L. (2017). Transcend: Detecting concept drift in malware classification models. Em *26th USENIX Security Symposium (USENIX Security 17)*, páginas 625–642, Vancouver, BC. USENIX Association.

Kantchelian, A., Afroz, S., Huang, L., Islam, A. C., Miller, B., Tschantz, M. C., Greenstadt, R., Joseph, A. D. e Tygar, J. D. (2013). Approaches to adversarial drift. Em *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, páginas 99–110, New York, NY, USA. ACM.

Kolter, J. Z. e Maloof, M. A. (2004). Learning to detect malicious executables in the wild. Em *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, páginas 470–478, New York, NY, USA. ACM.

Li, P., Liu, L., Gao, D. e Reiter, M. K. (2010). On challenges in evaluating malware clustering. Em *13th International Conference on Recent Advances in Intrusion Detection (RAID'10)*, páginas 238–255, Berlin, Heidelberg. Springer-Verlag.

Mangialardo, R. J. e Duarte, J. C. (2015). Integrating static and dynamic malware analysis using machine learning. *IEEE Latin America Transactions*, 13(9):3080–3087.

Manning, C. D., Raghavan, P. e Schütze, H. (2008a). *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA.

Manning, C. D., Raghavan, P. e Schütze, H. (2008b). *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK.

Mariconti, E., Onaolapo, J., Ross, G. e Stringhini, G. (2016). What's your major threat? on the differences between the network behavior of targeted and commodity malware. Em *11th International Conference on Availability, Reliability and Security*, ARES, páginas 599–608, Los Alamitos, CA, USA. IEEE Computer Society.

Mariconti, E., Onwuzurike, L., Andriotis, P., Cristofaro, E. D., Ross, G. e Stringhini, G. (2017). Mamadroid: Detecting android malware by building markov chains of behavioral models. Em *Proceedings of the 24th Network and Distributed System Security Symposium*, NDSS. Internet Society.

Mellish, C. (2017). Machine learning, lecture notes. `http://www.inf.ufpr.br/lesoliveira/aprendizado/machine_learning.pdf`, accessed in July 2017.

Melville, N., Stevens, A., K. Plice, R. e Pavlov, O. (2006). Unsolicited commercial e-mail: Empirical analysis of a digital commons. 10:143–170.

Michie, D., Spiegelhalter, D. J., Taylor, C. C. e Campbell, J., editores (1994). *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, Upper Saddle River, NJ, USA.

Microsoft (2003). Defining malware: Faq. `https://technet.microsoft.com/en-us/library/dd632948.aspx`.

Milgram, J., Cheriet, M. e Sabourin, R. (2006). "One Against One" or "One Against All": Which One is Better for Handwriting Recognition with SVMs? Em Lorette, G., editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France). Université de Rennes 1, Suvisoft. http://www.suvisoft.com.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition.

Nataraj, L., Karthikeyan, S., Jacob, G. e Manjunath, B. S. (2011). Malware images: Visualization and automatic classification. Em *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, VizSec '11, páginas 4:1–4:7, New York, NY, USA. ACM.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. e Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Pietrek, M. (1994). Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. `https://msdn.microsoft.com/en-us/library/ms809762.aspx`, accessed in November 2016.

Rieck, K., Trinius, P., Willems, C. e Holz, T. (2011). Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.*, 19(4):639–668.

Roberto Jordaney, Zhi Wang, D. P. I. N. e Cavallaro, L. (2016). Misleading metrics: On evaluating machine learning for malware with confidence. *Technical Report 2016-1 — Royal Holloway, University of London*.

Rossow, C., Dietrich, C. J., Grier, C., Kreibich, C., Paxson, V., Pohlmann, N., Bos, H. e v. Steen, M. (2012). Prudent practices for designing malware experiments: Status quo and outlook. Em *2012 IEEE Symposium on Security and Privacy*, páginas 65–79.

Roy, S., DeLoach, J., Li, Y., Herndon, N., Caragea, D., Ou, X., Ranganath, V. P., Li, H. e Guevara, N. (2015). Experimental study with real-world data for android app security analysis using machine learning. Em *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, páginas 81–90, New York, NY, USA. ACM.

Salton, G., Wong, A. e Yang, C. S. (1975). A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620.

Santos, I., Devesa, J., Brezo, F., Nieves, J. e Bringas, P. G. (2013). *OPEM: A Static-Dynamic Approach for Machine-Learning-Based Malware Detection*, páginas 271–280. Springer Berlin Heidelberg, Berlin, Heidelberg.

Santos, I., Nieves, J. e Bringas, P. G. (2011). *Semi-supervised Learning for Unknown Malware Detection*, páginas 415–422. Springer Berlin Heidelberg, Berlin, Heidelberg.

Schultz, M. G., Eskin, E., Zadok, E. e Stolfo, S. J. (2001). Data mining methods for detection of new malicious executables. Em *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, páginas 38–, Washington, DC, USA. IEEE Computer Society.

Sebastián, M., Rivera, R., Kotzias, P. e Caballero, J. (2016). *AVclass: A Tool for Massive Malware Labeling*, páginas 230–253. Springer International Publishing, Cham.

Siddiqui, M., Wang, M. e Lee, J. (2009). Detecting internet worms using data mining techniques. 6.

Singh, A., Walenstein, A. e Lakhotia, A. (2012). Tracking concept drift in malware families. Em *Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence*, AISec '12, páginas 81–92, New York, NY, USA. ACM.

Slashdot Media (2017). SourceForge. `https://sourceforge.net/`.

Softonic Internacional S.A. (2017). Softonic. `https://en.softonic.com/`.

Stringhini, G., Kruegel, C. e Vigna, G. (2010). Detecting spammers on social networks. Em *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, páginas 1–9, New York, NY, USA. ACM.

Stringhini, G., Wang, G., Egele, M., Kruegel, C., Vigna, G., Zheng, H. e Zhao, B. Y. (2013). Follow the green: Growth and dynamics in twitter follower markets. Em *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, páginas 163–176, New York, NY, USA. ACM.

The UCI KDD Archive (1999). Kdd cup 1999 data. `https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`.

Tian, R., Islam, R., Batten, L. e Versteeg, S. (2010). Differentiating malware from cleanware using behavioural analysis. Em *2010 5th International Conference on Malicious and Unwanted Software*, páginas 23–30.

Turney, P. D. e Pantel, P. (2010). From frequency to meaning: Vector space models of semantics. *CoRR*, abs/1003.1141.

van der Maaten, L. (2014). Accelerating t-sne using tree-based algorithms. *Journal of Machine Learning Research*, 15:3221–3245.

van der Maaten, L. e Hinton, G. (2008a). Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605.

van der Maaten, L. e Hinton, G. (2008b). Visualizing high-dimensional data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605.

VirusTotal (2017). Virustotal: Free online virus, malware and url scanner. `https://www.virustotal.com/`.

Wang, S., Schlobach, S. e Klein, M. (2011). Concept drift and how to identify it. *Web Semantics: Science, Services and Agents on the World Wide Web*, 9(3):247 – 265. Semantic Web Dynamics Semantic Web Challenge, 2010.

Wattenberg, M., Viégas, F. e Johnson, I. (2016). How to use t-sne effectively. *Distill*.

Yan, G., Brown, N. e Kong, D. (2013). *Exploring Discriminatory Features for Automated Malware Classification*, páginas 41–61. Springer Berlin Heidelberg, Berlin, Heidelberg.

Yonts, J. (2010). *Building a Malware Zoo*. The SANS Institute.