Sommario

[Introduction 1](#_Toc530089375)

[Drebin Dataset 2](#_Toc530089376)

[Malware Detection 3](#_Toc530089377)

[Description 3](#_Toc530089378)

[Implementation 4](#_Toc530089379)

[Performances 6](#_Toc530089380)

[Family Classification 8](#_Toc530089381)

[Description 8](#_Toc530089382)

[Implementation 9](#_Toc530089383)

[Performances 10](#_Toc530089384)

[Conclusion 12](#_Toc530089385)

# Introduction

In this report we’ll describe the implementation of an Android malware detector and classifier based on machine learning techniques. In particular we’ll use Supervised Learning methods, so our system will learn from an open dataset containing thousands of programs to detect malwares and to classify them by its own.

Malware detection is a binary classification problem, in which malwares must be classified as positive and benevolent programs as negative.

The second aspect we’ll consider is the family classification of a malware. In fact malware developers produce variant of malwares to evade updated security defense. All variants of the same original malware belong to the same “family”. In the second part of this report we’ll see the implementation of a classifier that will learn to classify malwares according to their families.

I chose to implement the algorithm using Python to take advantage of the rich set of packages that it provides.

# Drebin Dataset

Drebin dataset is an open dataset containing features from 123,453 benign applications and 5560 malwares. The dataset is structured as follows: the root folder contains a subfolder “features\_vector” and a csv file “sha\_families” containing couples sha-family where “sha” is the sha-256 hash of the malware and “family” is the family the malware belongs to. The subfolder contains a txt file for each program, named with the hash of the program that the file is related to. Each file contains a line for each feature of the program. Features can be divided into two categories: static and dynamic.

Static features are string extracted by the Manifest file, that provides general information on the application, static features can be divided in four subcategories: hardware components, requested permissions, app components and filtered intents.

Dynamic features are obtained by disassembling the executable of the applications and they include all APIs calls in the applications and all the strings (including network addresses).

# Malware Detection

Description

In this section I’ll describe the solution to the first problem: given a program the algorithm must classify it as benign or malware.

The problem can be geometrically interpreted in a N-dimensional vector space, where N is the cardinality of the set of all features collected from all programs in the dataset.

In this space a program is a N dimensional vector:

𝓋 ∈ ℝN  s.t. 𝓋i ­­=

To represent a program in this way we must first index all features with a progressive number from 1 to N.

The solution of the problem is the hyperplane that better partition the ℝN space in two subspaces:

ℝN = 𝔸 ⊕ 𝔹

Such that a program 𝓍 can be classified according to this partition as follows:

𝓍 =

Once we have the equation of the hyperplane:

𝓎(𝓍) = (𝓌­­­1 , 𝓌­­­2 , … , 𝓌­­­N ) + 𝓌­­­0

It’s easy to classify a new instance:

𝓍 =

So the learning process consists on learning the weights 𝓌­­­i of the hyperplane. In this representation of the problem there’s an implicit assumption: the programs must be linearly separable, i.e. we’re assuming it exists a line (or an hyperplane that’s the generalization of a line in a multidimensional space) that perfectly divides instances of the two classes.

There’re a lot of algorithms that computes the coefficients 𝓌­­­i of the hyperplanein order to divide the instances in the training set. In this implementation we’ll use SVM, that’s a method that finds the optimum value for each weight 𝓌­­­i that maximizes the distance from the hyperplane to its closer point. This method is reliable because it’s able to find the optimum solution even in presence of outliers (i.e. a small number of points that are distant from the others).

Implementation

To solve the first problem, I used the “scikit-learn” implementation of the linear SVM (LinearSVC). Scikit-learn is a python library that provides efficient implementations of many machine learning algorithms.

LinearSVC object has two fundamental method: fit, to train the SVM with a given dataset and predict, to classify new instances.

The fit method instead of taking one instance at a time takes a batch of instances (𝒳 parameter) with their real classification (𝓎 parameter). These two parameters are defined as follow:

𝒳MxN = { 𝓍i,j } and 𝓎 ∈ ℝM  where:

* M is the number of programs of the training set
* N is the number of total features
* 𝓍i,j =
* 𝓎i =

So, each row of the 𝒳 matrix represents a program, and the 𝓎 vector is the ground truth, that is the correct classification (1 for malwares and 0 for non malwares) for each program in the 𝒳 matrix. The fit method updates the weights of the SVM in order to correct classify the programs of the 𝒳 matrix according to the value indicated in the 𝓎 vector.

The predict function takes only the 𝒳 matrix and returns the 𝓎 vector with the predicted class (0 or 1) for each program in the 𝒳 matrix.

To build the 𝒳 and 𝓎 function I followed the following step:

* First, I iterated over each program to build the list of all features, without repetitions
* Then I used a dictionary where to easily access the index of each features
* Then I divided the dataset into training set (containing the 66% of total programs) and test set (containing the 33% if total programs).
* Then for each program I extracted all features and if the program *i* had the features *j* I set to one the 𝓍i,jelement of the 𝒳 matrix. Than if the *i* programs appears in the sha\_families.csv file (containing all malwares with the corresponding family) I set the ith element of the 𝓎 vector to one or to zero otherwise.

Malware detector must save the indexed list of features so it can build a new 𝒳 matrix when we give as input sets of features representing new programs to classify.

There are many advantages in using machine learning techniques for malware analysis. First of all, the process is completely autonomous. Once the algorithm learned how to classify malwares from dataset it could be able to classify new malwares, even if they are unknown. This because it could find evidences from data that would have been difficult to find by a human and that could be valid also for a completely new family of malwares.

It’s precisely for this reason that I decided by design not to divide features in categories, I put all features on the same level in order to let the algorithm decide which of them to give more importance to.

Another important point is that this algorithm is not a black box, in fact we can inspect which features were most involved in the decision. This is because of the structure of the hyperplane, or “discriminant function”:

𝓎(𝓍) = 𝓌­­­1  + 𝓌­­­2 + … + 𝓌­­­N  + 𝓌­­­0

Now remember that a program 𝓍 is classified as follow:

𝓍 =

This means that the features that most contributed to the decision of classifying a program as a malware are those with higher value of 𝓌. According to this property, in order to explain the reason of the classification I made a list of the *k* most weighted features, then for each program classified as Malware I computed the intersection between the features of the program and the *k* most weighted features. The results of the intersection are the features that most contributed to identify that program as a malware. Now I think it’s worth spending few words about choosing the value of *k* (the number of most weighted feature to select), that can be considered as an hyper-parameter of the algorithm. I tried different values of *k* and then I measured the performance:

* If *k* is too small respect to the number of all features it could happen that for a program 𝓍detected as malware the intersection between the features of 𝓍and the *k* most weighted features is empty, in that case the algorithm cannot explain why 𝓍has been classified as malware. Considering that there are 545327 different features, experiments shows that this happens when *k* < 500.
* If *k* is too big the intersection will contain too much features and the explanation could not be tight, that is the explanation is not specific for that malware respect to other malwares.

I found that a value of *k*  around one thousand is not too small neither too big and it’ll provide good results even when we’ll be use it for family classification, as we’ll see in next section.

## Performances

To test the algorithm I simply split the dataset into training set, containing the 66% of total programs and test set, with the remaining programs. Then I computed the predicted values for the training set and I compared them with real values to collect the following statistics:

* *True Positives:* malwares correctly classified as malwares
* *True Negatives:* benign programs correctly classified
* *False Positives:* benign programs classified as malwares
* *False Negatives:* malwares classified as benign programs

I used these values to calculate the confusion matrix:

*Predicted Values*

*Ground truth*

Where in the first diagonal there are the numbers of program correctly classified (true positive and true negatives) and in the second diagonal the numbers of the programs wrongly classified (starting from left bottom we have false positives and false negatives).

In the following table there are other indicator useful for measuring performance of the algorithm.

|  |  |  |  |
| --- | --- | --- | --- |
| *Accuracy* | *Precision* | *Recall* | *False Positive Rate* |
| 99.4% | 93.5% | 92% | 0% |

Accuracy counts how many programs are correctly classified among the total of programs. This indicator is not sufficient to make an evaluation of the machine learning algorithm, especially when the two classes are not balanced. In our case there are much more benign programs than malwares, so even classifying all programs as benign the algorithm could reach a good value of accuracy.

*Accuracy =*

For this reason, we introduce the precision, that measure how much we can trust the algorithm when it classifies a program as a malware. Precision is high if there’re few false positives.

*Precision =*

Recall measures the ability to spot a malware. Recall is high if there’re few false negative

*Recall =*

The false positive rate measure how much the algorithm is prone to errors, classifying benign programs as malwares. False positive rate is low if there’re few false positives.

*False Positive Rate =*

# Family Classification

## Description

The second problem is the family classification, that is given a set of features of a malware find which family the malware belongs to. This could be interpreted again as a multiple class classification problem, where the classes are all the families in the dataset and the instances to classify are the programs, and it could be solved with the same approach we used for malware detection, but we’ll see a different interpretation and we’ll solve this second problems using different techniques: probabilistic learning.

To solve this problem I implemented a Bayes Classifier. This method uses the Bayes Theorem to compute the probability of each family given a certain feature. Then once we have the probability of every families given each feature, to classify a new malware (that’s a set of features) we just have to find the family that maximizes the probability of the family given that set of features. Let’s write some formulas to be clearer.

The classification problem can be summarized in the following formula:

With:

* *f*  is a family
* *p* is a program
* *n* is the number of features
* *m* is the number of features of program p
* *p = {s1 , s2 , . . . , sm}*
* *I =*

So, we’re looking for the family *f* that maximizes the probability of that family given the program *p* according to what we observed in the dataset. Now let’s explain the formula above:

We’re just applying the Bayes Theorem omitting the denominator since we’re only interested in computing the argmax (which is not affected by multiplicative factors)

Now we’re considering a binomial distribution

This is just a notation simplification, because in if the feature belongs to the program the first term will be the same and the second will be one, else both terms will be one, so the only terms contributing in the products will be the one where the feature belongs to the program.

In this formula we made an important assumption, i.e. we’re assuming that the features of a program are independent each other, so that we can write the joint probability distribution – that’s very hard to compute - like the product of conditional probabilities easy to compute. Obviously, this is not true because many features can be related, and if we find a certain feature in a program we will probably find also the other related to the first, but despite this assumption the result is satisfactory, so we can conclude that this method provides a good approximation.

## Implementation

As we said in the previous section to implement the family classifier, we must solve this formula:

The first thing I did was to reduce the set of features representing a program to a smaller set. In particular, remember that in the section about detection we computed the set of the most “dangerous” features, i.e. the set of all the features that most contributed to detect a program as a malware. Then we calculated the set of dangerous features also for a single program, as the intersection of the set of features of the program and the set of all the most dangerous features.

The second enhancement consists in not considering the families with few malwares, because the algorithm needs many samples of each family in order to extract the features that will cause that classification. For this reason, I replaced families with less than N families with a single “unknown” family. I experimented different values of N and I observed that with more than 50 samples per family classification is accurate enough.

In this second part we’ll represent a program with the set of its most dangerous features, rather than all features. Obviously to classify the malware family of a program the program is assumed to be a malware - or it’s a false positive, i.e. a benign program classified as a malware from our detection algorithm) - so it’s not a mistake to replace the features set with the most dangerous features set.

I made this choice for two reasons: the first one is about performance, because with a fewer number of features the computation of the formula above requires less iteration. The second reason is to avoid noise in the decision process, because only the features that contributed to identify the program as a malware should be used to classify its family while the “harmless” features that are common also with the benign programs should not.

The first term is the probability of a certain feature given the family and it can be computed as the number of malwares with the feature *si* of family *f* in the dataset - divided by the number of malwares of family f in the dataset. This first term is Bayes terminology is called “likelihood”, and during training the algorithm must learn the likelihood for each features-family pair.

*for each family f:*

*for each feature s:*

*=*

The second term is called “prior probability” and it’s computed as the number of malwares of family f divided by the number of malwares in the dataset. The algorithm will learn the “prior probability” for all families as follows:

*for each family f:*

*=*

Once the algorithm has learnt all these terms the family of a new program *p* can be computed as:

Where I’m considering only the dangerous features of program *p*.

## Performances

To measure performances once again I split the dataset into training test (66% of the dataset) and test set (33% of the dataset) where the dataset is the set of malwares with their correct classification.

I computed the accuracy as well classified malwares divided by the total of classified malwares:

*accuracy = 0.85= 85%*

Then I computed the confusion matrix, that’s defined as follows:

*M = {ai,j} s.t. ai,j = (number of malwares of family i classified as malwares of family j)*

From this definition follows that the trace of the confusion matrix is the number of well predicted elements while the sum of the other elements is the number of wrongly predicted malwares. This is the confusion matrix of this algorithm:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | BB | DD | DKF | ELL | FD | FI | FR | GP | GN | GM | GL | IN | KM | MT | OP | PL | SP | UF |
| A | 28 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 1 |
| BB | 0 | 93 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9 | 0 | 0 | 0 | 0 | 0 | 5 |
| DD | 0 | 0 | 31 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| DKF | 0 | 0 | 0 | 229 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| ELL | 0 | 0 | 0 | 1 | 10 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 0 |
| FD | 0 | 0 | 0 | 0 | 0 | 40 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| FI | 0 | 0 | 0 | 0 | 0 | 0 | 248 | 0 | 0 | 0 | 0 | 1 | 3 | 0 | 0 | 52 | 0 | 0 | 4 |
| FR | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 22 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| GP | 0 | 1 | 0 | 4 | 0 | 0 | 0 | 0 | 10 | 0 | 1 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 |
| GN | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 30 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| GM | 0 | 1 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 109 | 1 | 6 | 0 | 0 | 0 | 0 | 0 | 0 |
| GL | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 20 | 3 | 0 | 0 | 0 | 0 | 0 | 2 |
| IN | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 48 | 0 | 0 | 0 | 0 | 0 | 0 |
| KM | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 51 | 0 | 0 | 0 | 0 | 0 |
| MT | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 21 | 0 | 0 | 0 | 0 |
| OP | 0 | 0 | 0 | 0 | 0 | 1 | 3 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 187 | 0 | 0 | 1 |
| PL | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 213 | 0 | 4 |
| SP | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 18 | 0 |
| UF | 7 | 5 | 1 | 5 | 2 | 2 | 32 | 1 | 0 | 1 | 0 | 7 | 48 | 1 | 0 | 15 | 2 | 0 | 204 |

Families:

|  |  |
| --- | --- |
| A | Adrd |
| BB | BaseBridge |
| DD | DroidDream |
| DKF | DroidKongFu |
| ELL | ExploitLinuxLotoor |
| FD | FakeDoc |
| FI | FakeInstaller |
| FR | FakeRun |
| GP | Gappusin |
| GN | Genimi |
| GM | GinMaster |
| GL | Glodream |
| IN | Iconosys |
| KM | Kmin |
| MT | MobileTx |
| OP | Opfake |
| PL | Plankton |
| SP | SendPay |
| UF | UnknownFamily |

# Conclusion

In this analysis we saw the power of machine learning algorithm applied to malware detection and classification. Now let’s make a comparison with the approach used by classic antiviruses.

First of all antiviruses uses hand-coded strategies for detection, that sometimes are “ad-hoc” for a particular kind of malware. Machine learning version instead learns to detect malwares through examples, and for this reason it could be more effective in detecting unknown malwares, since its strategy is more general than the classic approach.

Another advantage of machine learning is that the computationally hard work is done in the learning phase, so once the algorithm has learned it can classify a program in milliseconds.

However, even though machine learning method has shown excellent results, this analysis is limited to static features (or at least dynamics feature obtained by disassembling the application) and cannot inspect the behavior of the program at run time.