Table of Contents

Introduction

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

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. 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 these algorithms 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 123453 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”. Csv file contains 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, files are named with the hashes of the programs. Each file contains a line for each feature of the program. Features can be divided into two categories: static and dynamic.

Static features are strings extracted from the Manifest file, that provides general information about 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 invoked and all strings used (including network addresses).

Malware Detection

Description

In this section it's described the implementation of the first algorithm that, given a program, 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:

$$v \in \mathbb{R}^N \text{ s.t. } v_i = \begin{cases} 1 & if\ feature\ with\ index\ i\ appear\ in\ v \\ 0 & if\ feature\ with\ index\ i\ do\ not\ appear\ in\ v \end{cases}$$

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 $\mathbb{R}^N$ space in two subspaces:

$$\mathbb{R}^N = \mathbb{A} \oplus \mathbb{B}$$

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

$$x = \begin{cases} Malware & if\ x \in \mathbb{A} \\ Benign & if\ x \in \mathbb{B} \end{cases}$$

Once we have the equation of the hyperplane:

$$y(x) = (w_1, w_2, \dots, w_N)\, x + w_0$$

It's easy to classify a new instance:

$$x = \begin{cases} Malware & if\ y(x) > 0 \\ Benign & if\ y(x) \leq 0 \end{cases}$$

So, the learning process consists on learning the weights w_i 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 a 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 w_i of the hyperplane in 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 w_i maximizing 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 implement this first algorithm, we'll use the SVM provided by "scikit-learn", that's called LinearSVC. Scikit-learn is a python module that efficiently implements 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 ($\mathcal{X}$ parameter) with their real classification ($\mathcal{y}$ parameter). These two parameters are defined as follow:

$\mathcal{X}_{\mathrm{MxN}} = \{ x_{\mathrm{i,j}} \}$ and $\mathcal{y} \in \mathbb{R}^{\mathrm{M}}$ where:

- M is the number of programs of the training set
- N is the number of total features
- $x_{\mathrm{i,j}} = \begin{cases} 1 & if\ program\ i\ has\ feature\ with\ index\ j \\ 0 & otherwise \end{cases}$
- $\mathcal{y}_{\mathrm{i}} = \begin{cases} 1 & if\ program\ i\ is\ a\ malware \\ 0 & otherwise \end{cases}$

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

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

To build $\mathcal{X}$ and $\mathcal{y}$ matrices we'll follow these steps:

- First, we iterate over each program to build the list of all features, without repetitions
- Then we'll use a dictionary where $\langle key|value\rangle = \langle feature|idx\rangle$ to easily access the index of each features
- Then we'll divide the dataset into training set (containing the 66% of total programs) and test set (containing the 33% if total programs).
- Then for each program we'll extract all features and if the program i had the feature j we'll set to one the $x_{\mathrm{i,j}}$ element of the $\mathcal{X}$ matrix. Than if the i programs appears in the sha_families.csv file (containing all malwares with the corresponding family) we'll set the ith element of the $\mathcal{y}$ vector to one or to zero otherwise.

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

Although dataset contains different kinds of features and the number of features for each kind could be unbalanced, we'll not categorize nor preprocess them, even if intuition suggests that some kinds of features are more useful than others. Doing this way our hypothesis will not influence the evidences collected from data by the algorithm, that could find connection that maybe we didn't consider.

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":

$$y(x) = w_1 x_1 + w_2 x_2 + \ \dots + w_N x_n + w_0$$

Now remember that a program x is classified as follow:

$$x = \begin{cases} Malware & if\ y(x) > 0 \\ Benign & if\ y(x) \leq 0 \end{cases}$$

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 we'll list the k most weighted features, then for each program classified as Malware we'll compute 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 let's spend few words about choosing the value of k (the number of most weighted feature to select), that's a hyper-parameter of our algorithm. Trying with different values of k and measuring performances we'll observe that:

- If k is too small respect to the number of all features it could happen that for a program x detected as malware the intersection between the features of x and the k most weighted features is empty, in that case the algorithm cannot explain why x has been classified as malware. Considering that there are 545327 different features, experiments show 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 enough.

A value of k around one thousand is not too small neither too big and it'll provide good results even when we'll use it for family classification, as we'll see in next section.

Performances

To test the algorithm, let's simply split the dataset into training set, containing the 66% of total programs and test set, with the remaining programs. Then computing the predicted values y for the training set and comparing them with real values we'll 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

Once we get these values, we can use them to calculate the confusion matrix:

$$\begin{array}{ccc} & & \textit{Predicted Values} \\ & & \begin{array}{cc} p & n \end{array} \\ \textit{Ground truth} & \begin{array}{c} p \\ n \end{array} & \begin{pmatrix} 1708 & 118 \\ 148 & 41819 \end{pmatrix} \end{array}$$

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 enough 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.

$$\textit{Accuracy} = \frac{true\ positive + true\ negative}{true\ positive + true\ negative + false\ positive + false\ negative}$$

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

$$\textit{Precision} = \frac{true\ positives}{true\ positives + false\ positives}$$

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

$$\textit{Recall} = \frac{true\ positives}{true\ positives + false\ negatives}$$

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.

$$\textit{False Positive Rate} = \frac{false\ positives}{false\ positives + true\ negatives}$$

Family Classification

Description

Now we'll implement an algorithm for family classification, that is, given a set of features of the malware, find which family the malware belongs to. This could be interpreted again as classification problem but this time with multiple classes, where classes are all the families in the dataset and the instances to classify are the malwares. Although we could implement the algorithm with the same techniques used for malware detection, we'll give this problem a different interpretation using probability.

Now we'll go through the implementation of 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 family 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:

$argmax_f \; \{P(f|p)\}$

$= ① \quad argmax_f\{\ P(p|f) * \ P(f)\ \}$

$= \ ② \ argmax_f \left\{ \prod_{i=1}^{n} P(si|f)^I * \left(1 - P(si|f)\right)^{1-I} * \ P(f) \ \right\}$

$= \ ③ \ argmax_f\{ \ \prod_{i=1}^{m} P(s_i|f) * \ P(f) \ \ \}$

Where:

- f is a family
- p is a program
- n is the number of features
- m is the number of features of program p
- $p = \{s_1 , s_2 , . . . , s_m\}$
- $I = \begin{cases} 1 \ if \ s_i \in p \\ 0 \ if \ s_i \ \notin p \end{cases}$

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:

① 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)

② Now we're considering $P(p|f)$ a binomial distribution

③ This is just a notation simplification, because in ② if the feature belongs to the program the first term won't change and the second will be one, else both terms will be one, so the only terms contributing in the products will be the ones where the feature belongs to the program.

In this formula 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. This assumption of course is not completely 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 of the real solution.

Implementation

As we already observed, to classify the program p we must solve this formula:

$$argmax_f \{P(f|p)\} = argmax_f\{ \prod_{i=1}^{m} P(s_i|f) * P(f) \}$$

In order to be able to compute the formula above the algorithm must learn all the terms in the right side for all features and for all families. Now we'll see in detail how the algorithm can learn each of those.

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 s_i of family *f* in the dataset divided by the number of malwares of family f in the dataset. This first term according to Bayes terminology is called "likelihood", and during training the algorithm must learn the likelihood for each feature-family pair.

```
for each family f:
        for each feature s:
                $P(s|f) = \frac{number\ of\ malwares\ with\ feature\ s\ of\ family\ f}{number\ of\ malwares\ of\ family\ f}$
```

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:
        $P(f) = \frac{number\ of\ malwares\ of\ family\ f}{number\ of\ malwares\ in\ the\ dataset}$
```

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

$$f = argmax_f\{ \prod_{i=1}^{m} P(s_i|f) * P(f) \}$$

Where the m features are those of program p, i.e. $p = \{s_1, \ldots, s_m\}$.

Now that we know how the algorithm is implemented let's look at some optimizations. The first thing we're going to do is to reduce the set of features representing a program to a smaller set. 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.

So, 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 family 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.

This choice is due to two reasons: the first one is time optimization, because with a fewer number of features the computation of the formula above requires less iteration. The second reason is accuracy, 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 second enhancement consists in not considering the families with few malwares, because the algorithm needs many samples of each family to learn to correctly classify instances of that family. For this reason, we'll replace families with less than N programs with a single "unknown" family. Trying different values of N and we'll observe that with more than 50 samples per family classification is accurate enough.

Performances

To measure performances again we 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.

Accuracy is computed as well classified malwares divided by the total of classified malwares:

$$accuracy = \frac{well\ predicted}{total\ predictions} = 0.85 = 85\%$$

The confusion matrix is defined as follows:

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

The trace of the confusion matrix is the number of exact predictions while the sum of the other elements is the number of incorrect predictions. 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

The example described in this report showed how simple machine learning algorithms can be applied to the problem of malware analysis, and despite we made some assumptions that are not completely true - like linear separability of data and the independency of the features of a program - results are more than encouraging.

Now let’s compare the approach we saw in this report with classic antiviruses to understand the most important differences:

First, antiviruses use hand-coded strategies for detection, that sometimes are “ad-hoc” for a specific kind of malware. Machine learning algorithms instead learn to detect malwares through examples, and for this reason they could be more effective in detecting unknown malwares, since their strategy is more general.

Another advantage of machine learning is that the computationally hard work is done in the learning phase that happens before the distribution. Once the algorithm has learned and it’s packed into a mobile application it can classify a program in few milliseconds, while antiviruses often slow down system performances because their algorithms are slower and usually require more resources.

However, our analysis is static, i.e. features extraction – from the manifest and from disassembling the executable – happens before the installation, antiviruses instead inspect applications also while they’re running. This is one limit of our implementation because sometimes the nature of a program can be discovered only by observing its behavior while it’s running.