Sommario

[Introduction 1](#_Toc529894534)

[Drebin Dataset 1](#_Toc529894535)

[Malware Detection 1](#_Toc529894536)

[Description 1](#_Toc529894537)

[Implementation 2](#_Toc529894538)

# Introduction

In this report I’ll describe the implementation of an Android malware detector and classifier based on machine learning techniques. In particular I’ll use Supervised Learning methods, where the ground truth is provided by a huge open dataset called “Drebin” containing features extracted by malwares and benign programs collected through both official and unofficial application stores.

The problem of malware detection is a binary classification problem, where the two classes in which programs must be divided into are the positive class, denoted with the symbol “+” and the negative class, denoted with “-”. The former will contain malwares and the latter benign programs.

The second problem is a different classification one, in which malwares must be divided into classes according to the “family” they belong to, where a family is a set of malwares with the same behavior that usually have similar features.

In this report I’ll describe the techniques used to solve both these problems and the relative implementation in Python.

# 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 turn 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. This is a problem of binary classification, and we’ll assume that programs are linearly separable.

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. There’re a lot of algorithms for computing these weights according to a training set, in my implementation I used SVM, that’s a method that finds the optimum value for each weight 𝓌­­­i 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 solve this 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.