Malware Analysis

HW1 - ML Sapienza

Giuliano Abruzzo
1712313

2018

Contents

1 Introduction

The goal of the first Machine Learning homework is to analyze a list of Android applications and determine through **Machine Learning** techniques if these are malware or not.

1.1 Malware and Machine Learning role

A **malware** is a malicious software that fulfills the deliberately harmful intent of an attacker. Often a malware:

- Is designed to damage users or systems;
- Exploits Software and Hardware Vulnerabilities;
- Uses Social Engineering to trick users;
- Can install other malware;
- Is controlled by a command and control server;

The analysis of malware is the study of **malicious software** with the aim of studying and understanding different aspects of malware such as their behavior, how they evolve in time and how they intrude target systems. Malicious software are evolving and improving along with security defenses so malware continues to infect systems. For this reason, the defense-side goal is to produce defensive technologies as challenging as possible to overcome. **Machine Learning** is a natural choice to support a process of knowledge extraction from malicious software and allows to create malware analysis techniques that:

- Not need human support;
- Are resilient to obfuscation techniques;

1.2 Malware Analysis

The goal of the malware analysis is finding if an application is malware or not. We can describe it as a function MD having:

- The set F of all possible files as the domain;
- The set P,N as codomain where P represents Positive if the file is a malware and N represents Negative if the file is not a malware;

There are two types of learning:

- Supervised learning: that use a labeled training set;
- Unsupervised learning: that don't need a labeled training set;

2 Drebin

Drebin is a lightweight method for detection of Android malware that enables identifying malicious applications directly on the smartphone. Drebin performs a *broad static analysis*, gathering as many features from an application's code and manifest as possible. These features are organized in sets of strings (such as permissions, API calls, and network addresses) and embedded in a joint vector space.

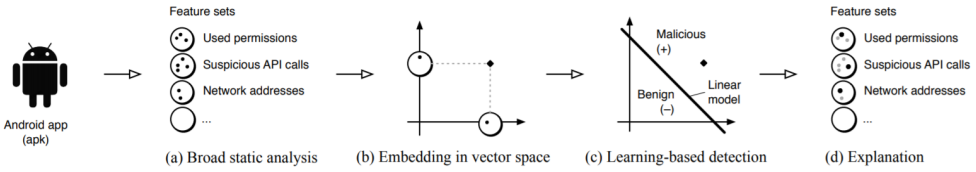

Figure 1: Schematic depiction of the analysis steps performed by Drebin [1]

The Drebin archive contains two elements:

- **feature vectors**: a folder containing 123.453 files, each file represents an application;

[1] https://www.sec.cs.tu-bs.de/pubs/2014-ndss.pdf

- **sha256 family.csv**: a CSV file containing 5.560 malware and the family they belong to;

The zip file contains the features extracted from each application and these features are stored inside a text file whose name is the SHA1 Hash of the apk.

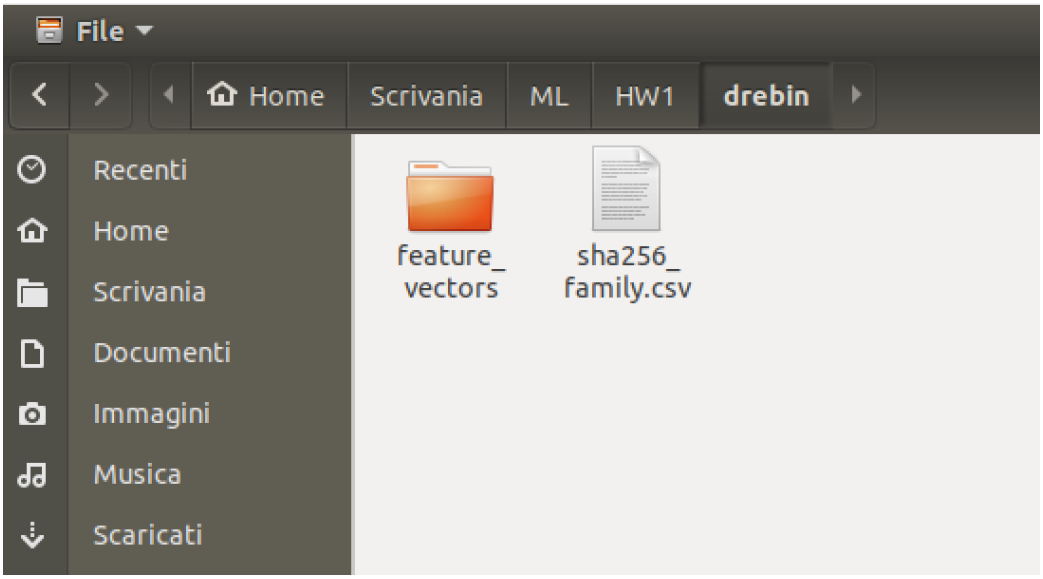

Figure 2: Drebin Archive

2.1 Features Sets

The features can be classified in 8 different sets. The first 4 groups are features that can be extracted from the AndroidManifest.xml file. The other 4 groups are features extracted from the disassembled code.

Prefix	SET
feature	S1: Hardware components
permission	S2: Requested permission
activity service_receiver provider service	S3: App Components
intent	S4: Filtered Intents
api_call	S5: Restricted API calls
real_permission	S6: Used permission
call	S7: Suspicious API calls
url	S8: Network addresses

Figure 3: Features Sets

2.2 Sha256.csv

The sha256.csv file contains a dictionary file in CSV format. In the dictionary there are the SHA1 Hash of the malware in the dataset and the family they belongs to.

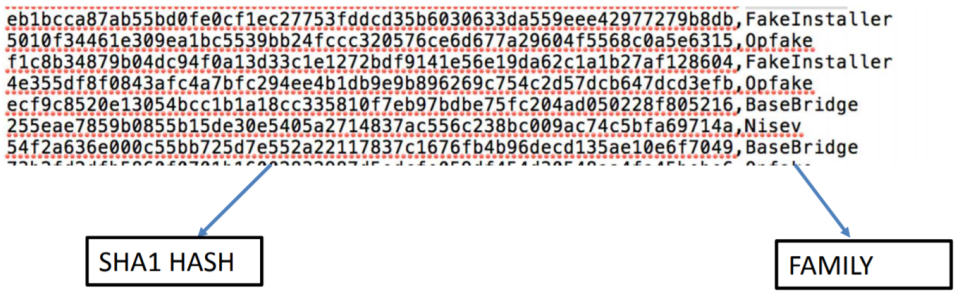

Figure 4: sha256 family.csv content

2.3 Application files

Inside the feature vectors folder, there are all the features extracted from each application. All these features are composed by a prefix and a value, the prefix represents the set the feature belongs to. For example **"Permission::android.permission.INTERNET"** belongs to the set S2 and represent the request of permission to access to the internet. A single feature is described by the pattern: ***prefix::value***.

3 Malware detection

In this chapter, we will discuss the **classifier** implemented for the classification problem written in a python script with the use of the Scikit-learn library.

3.1 Scikit-learn

Scikit-learn is a module for implement **Machine Learning** in Python. It is:

- Simple and efficient tools for data mining and data analysis;
- Accessible to everybody, and reusable in various contexts;
- Built on NumPy, SciPy, and matplotlib;
- Open source, commercially usable - BSD license;

3.2 Classification Algorithm

3.2.1 Import

In addition to the Sklearn module, the script also imports different modules:

- **Random**: used to get randomized application files;
- **OS**: used to get the list of all files in the feature vectors directory;
- **Numpy**: used to create the confusion matrix;
- **Pandas**: used to read the rows of the CSV file;

3.2.2 Sets

The script starts by creating different sets:

- **All_files**: a set that contains all the applications files;
- **All_malware**: a set that contains all the malware of the CSV file;
- **Random_malware**: a subset of All_malware with dimensions one fifth of the original and chosen randomly;
- **Random_file**: a subset of All_files with dimensions one tenth of the original and chosen randomly;
- **Non_malware**: a set created by the difference between Random_file and Random_malware;

3.2.3 Dataset

Then the script initializes the dataset list and will create the **features set** that will contain *'permission', 'call', 'url'*, and *'intent'* sets. After this, it will cycle on each app file to classify it, will scroll all the lines of the file and this line will be split into two elements: the first represents the feature set which it belongs and the second one represents the value. The script will concatenate all the values whose feature is one of those contained in the feature set and this string will be attached to the dataset list along with the file classification (type represent if it is malware or not) in form: [***"string", "type"***] The script will then shuffle the dataset through the shuffle function and will call the Kfold_cross, by attaching the dataset.

3.2.4 KFold

The script will call the **KFold function** of the **SKlearn module** [2]. This function will split dataset into 30 consecutive folds and will return to us the *training indexes* and the *test indexes*. After, the script will create and populate 4 different lists starting from the dataset and the indexes:

- **Train_msg**: a list that will contain all the values of the train set;

[2] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html

- **Train_classe**: a list that will contain all the *"malware"* or *"not_malware"* of the train set;
- **Test_msg**: a list that will contain all the values of the test set;
- **Test_classe**: a list that will contain all the *"malware"* or *"not_malware"* of the test set;

3.2.5 Classifier

When the script finishes creating the four lists, it will call the SKlearn **CountVectorizer** [3] function which will create the matrix of token counts starting from the *Train_msg* list. This matrix will then be given as an input to the **Multinomial naive Bayes classifier**[4] of the SKlearn library through the fit function together with the *Test_msg* list. I choose to use the *Multinomial naive Bayes classifier* instead of the *Bernoulli naive Bayes classifier* because is more accurate. After the fit function is finished, the script will find the matrix of the token counts of the *Test_msg*. This matrix will be given as an input to the **predict** function of the classifier and it will return a list called *predicted* that will then be compared with the list of *Test_class* to see the **accuracy** of the classifier.

4 Performance Evaluation

4.1 Confusion Matrix

In a classification problem with many classes, we can compute how many times an instance of a class C_i is classified in class C_j. Main diagonal contains accuracy for each class, outside the diagonal there are the errors.

	Malware	***Benign***
Malware	**TP**	**FN**
Benign	**FP**	**TN**

[3] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
[4] https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html

We based on the convention that if an app is classified as a **malware** this is a **positive** sample, while a **not_malware** is a **negative** sample. We can have 4 cases:

- **TP: True Positive**, I detect that it's malware, and it's true;
- **FP: False Positive**, I detect that it's malware, and it's false;
- **TN: True Negative**, I detect that it's not malware, and it's true;
- **FN: False Negative**, I detect that it's not malware, and it's false;

4.2 Accuracy

Accuracy basically shows the percentage of malware that have been **correctly classified** but is not always a good performance metric, in fact, in some cases, is not enough to assess the performance of a classification method. The accuracy has been calculated through the function of the sklearn module: *accuracy_score*.

- Accuracy is:

$$1 - ErrorRate \tag{1}$$

- Where Error Rate is:

$$\frac{FN + FP}{TP + TN + FP + FN} \tag{2}$$

- And in conclusion Accuracy can be calculated as:

$$\frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

In our test the average value of the Accuracy is:

- **Accuracy avg value: 88.2 %**

4.3 Recall

Recall represents the ability to avoid false negatives. Recall has been calculated through the function of the sklearn module: *recall_score*.

- Recall is:

$$\frac{TP}{TP + FN} \tag{4}$$

Recall it's equal to 1 if FN = 0. In our test the average value of the Recall is:

- **Recall avg value: 0.85**

4.4 Precision

Precision represents the ability to avoid false positives. Precision has been calculated through the function of the sklearn module: *precision_score*.

- Precision is:

$$\frac{TP}{TP + FP} \tag{5}$$

Precision it's equal to 1 if FP = 0. In our test the average value of the Precision is:

- **Precision avg value: 0.40**

4.5 False-Positive Rate

False-Positive Rate represent how many files I wrongly consider as a malware among all the benign files.

- False-Positive Rate is:

$$\frac{FP}{FP + TN} \tag{6}$$

In our test the average value of the False-Positive Rate is:

- **False-Positive Rate avg value: 0.14**

4.6 F-Measure

F-Measure can be interpreted as a weighted average of precision and recall.

- F-Measure is:

$$\frac{2 * (Precison * Recall)}{Precision + Recall} \tag{7}$$

In our test the average value of the F-Measure is:

- **F-Measure avg value: 0.53**