

Naive Bayes Classifier for Malware Detection

HW1 - Machine Learning Sapienza

Valerio Trenta, 1856471

2018-11-18

1 Introduction

This document describes the implementation of a **Naive Bayes Classifier** in order to classify applications for **Android OS**.

Given a set of *categories* $C = C_1, C_2, \dots, C_n$, **classification** is the problem of identifying to which of these categories a new instance i belongs: a **classifier** is an *algorithm* which implements **classification**.^[1]

The classifier realized in this work makes use of **Drebin dataset**, which contains more than *120 000* applications, which are classified as *safe* if they contain *no malicious threats*, and as *malware* otherwise.

The goal of this work is to analyze and understand the dataset, in order to partition it and use it to implement and then evaluate a *machine learning program* which is able to determine if an application contains malicious threats - and is therefore classified as *malware* - or not, returning useful predictions based upon the dataset used for its training.

In paragraphs **2** and **3**, general concepts about android malwares and their structure are given, followed by a description of **Drebin dataset** used in order to train and evaluate the classifier.

Paragraph **4** introduces and defines the **classification problem** which the classifier aims to solve, along with the **target function** which it tries to approximate.

Paragraph **5** finally specifies the **classifier** properties and its implementation, while paragraph **6** outlines the *evaluation procedure* used to evaluate the predictions made by the classifier, analyzing the results in terms of *accuracy*, *precision*, *recall* and *confusion matrix*.

2 Android Malware

A **malware** is a malicious software that fulfills the deliberately harmful intent of an attacker. [2]

Malwares have different characteristics, usually depending on the operative systems the attacker plans to target, but some common features are often shared between all type of malwares, like:

- their goal: malwares are designed to damage users or systems;
- their behaviour: in order to damage a system, malwares often exploit hardware and software vulnerabilities (also known as bugs);
- their virulent ability to spread the infection by installing other malwares on a system;
- **social engineering**, which is *psychological manipulation* performed by an attacker in order to trick users into divulging confidential information.[3]

There exist many different types of malwares, depending on their goal: *Trojan*, *Ransomware*, *Dropper*, etc.; following the rapid evolution of mobile hardware and software, **mobile malware** has spread incredibly fast in the last decade.

The first known mobile virus - called "*Timofonica*" - originated in Spain and was identified in **2000**, while ten years later, in **August 2010**, **Kaspersky Lab** identified the first malware classified as *Trojan* which affects mobile phones running on **Android Operative System**, named "*Trojan-SMS.AndroidOS.FakePlayer.a*".[4]

Many antivirus programs have been developed since then, but mobile malwares change, evolve and grow so fast that the antiviruses are often outdated and left behind: the only solution is for them to update constantly and fastly. This is where *malware analysis* steps in: by developing an even deeper understanding of malwares and their features and structure, including also their evolution in time and their behaviour, better countermeasures - and thus, better antivirus programs - can be built.

Machine learning is crucial in *malware analysis*, since it makes it possible - and easier - to reach a deeper understanding of malwares' behaviour and evolution.

Every time a security defence gets updated, many variants of a single, "*original*" malware are often produced in order to minimize the effort required to evade the update: these variants share many features with the original

malware, even if a small part of their code is changed, allowing malwares to avoid detection.

A **malware family** is the set of malwares deriving from the *same original malware*: malware \mathbf{m}_1 and \mathbf{m}_2 belonging to the malware family \mathbf{M} , have a similar behaviour and exploit the same vulnerabilities, and probably share the same malicious goal.

Furthermore, malwares often use *obfuscation techniques*: encryption of their processes and different fake certificates grant them not to be detected, but **machine learning** techniques are resilient to obfuscation.

An **Android malware** is a *malicious application* for **Android OS** often disguised as a *safe application* - that is to say, an application which contains no malicious threats.

Every Android application has some common components often contained by every other application: a malware can hide its malicious features in several different components of the application.

So, in terms of *Android application*, malwares belonging to the same malware families often share the same components.

Only by looking at the source code of the application - that is to say, the **apk** pack and, in particular, strings contained in **manifest.xml** file, which can *always* be found in every application - all these components can be extracted and analyzed: by classifying these features, a machine learning program is able to detect if an application is malicious or not and, if needed, it can also detect the malware family it belongs to.

3 Drebin Dataset

Drebin is a lightweight method for *Android malware detection* developed in 2014 by the *University of Göttingen*, in Germany, which performs statistic analysis to extract **features** from **samples** of Android applications and then use them to detect and classify *malware families*.^[5]

Drebin makes use of a **Support Vector Machine** in order to perform the aforementioned classification, but can be also implemented by using any kind of classifier - **Naive Bayes** for example, which is the one implemented and discussed in this paper.

The features extracted, which are used to fit the classifier, are publicly available as **Drebin Dataset**: it consists in a file with **123,453** benign Android applications and **5,560** malicious Android applications (malwares).

From each application, features are extracted mainly from the *manifest.xml* file and the *disassembled code*, and then divided into eight different sets (S_1, S_2, \dots, S_8) :

- S_1 : requested hardware components;
- S_2 : requested permissions;
- S_3 : app components (activities, services, content providers, broadcast receivers);
- S_4 : filtered intents;
- S_5 : restricted API calls;
- S_6 : used permissions;
- S_7 : suspicious API calls;
- S_8 : network addresses;

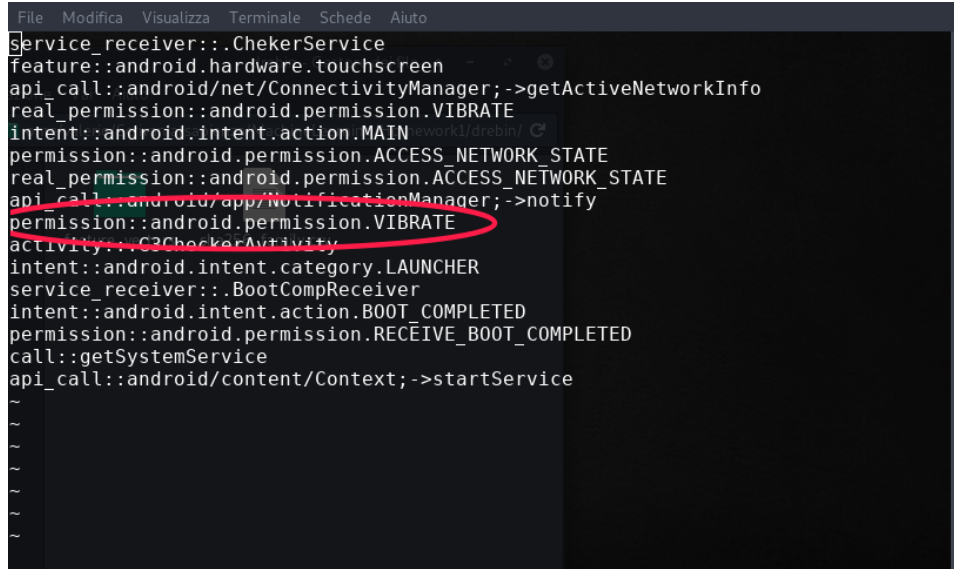
feature sets from S_1 to S_4 are extracted by *manifest.xml* of each sample, while feature sets from S_5 to S_8 are extracted from the disassembled code. Considering an application **A**, extracting S_1, S_2, \dots, S_8 sets of features from **A** can give us a deeply detailed view and description of the application, especially from a *malware detection* point of view.

Let's take, for example, feature sets S_2 and S_5 : **requested permissions** are declared in **manifest.xml** file of every Android application, since *permission system* is one of the most important security mechanisms introduced in Android. [5] Indeed, every time an application is installed, Android asks the user to grant a *permission* for each permission declared in **manifest.xml**, allowing the application itself to access *security-relevant resources*. [5] That is to say, a malicious application will surely ask the user to allow access to these important resources, so its **manifest.xml** file will be more likely to have these permissions declared in it.

Features in set S_5 are still related to the Android *permission system*: once a permission has been granted by the user, the application has also access to a series of restricted **API calls**, which can be found in the disassembled code. Sometimes, trying to avoid malware detection, an attacker could use a restricted **API call** for which the permission has not been granted or even requested, which is why it's also important to compare sets S_2 and S_5 .

The sixth set, S_6 , contains the permissions that are both requested and actually used, for which S_2 and S_5 are used as ground in order to obtain it. [5]

Every Android application in the dataset is identified by its **SHA1 hash** of the name of its **apk** pack, so every file containing the feature sets of a sample, is actually a text file whose name is the **SHA1 hash** of the sample. Every line of the text file consists in a **feature** with a **prefix** and a **value**: the first one represents the feature set the feature belongs to, while the second one states the value of the feature contained in the application.



```

File Modifica Visualizza Terminale Schede Aiuto
service_receiver::.ChekerService
feature::android.hardware.touchscreen
api_call::android/net/ConnectivityManager;->getActiveNetworkInfo
real_permission::android.permission.VIBRATE
intent::android.intent.action.MAIN
permission::android.permission.ACCESS_NETWORK_STATE
real_permission::android.permission.ACCESS_NETWORK_STATE
api_call::android/app/NotificationManager;->notify
permission::android.permission.VIBRATE
activity::C3CheckerActivity
intent::android.intent.category.LAUNCHER
service_receiver::.BootCompReceiver
intent::android.intent.action.BOOT_COMPLETED
permission::android.permission.RECEIVE_BOOT_COMPLETED
call::getSystemService
api_call::android/content/Context;->startService
~
~
~
~
~

```

Figure 1: Text file containing features from an Android application in Drebin dataset.

For example, in the file pictured in *Figure 1*, the string highlighted by the red circle is a feature identified by **prefix permission** and value *android.permission.VIBRATE*.

Each set is identified by its own prefix inside the dataset, as follows from the table in *Figure 2*.

Drebin dataset also includes another **.csv** file in which are stored all the malicious applications contained in the dataset, each one of them classified into a malware family, as pictured in *Figure 3*.

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 2: Table of prefix and corresponding set in Drebin dataset.

	A	B
1	sha256	family
2	090b5be26bcc4df6186124c2b47831eb96761fcf61282d63e13fa235a20c7539	Plankton
3	bedf51a5732d94c173bcd8ed918333954f5a78307c2a2f064b97b43278330f54	DroidKungFu
4	149bde78b32be3c4c25379dd6c3310ce08eaf58804067a9870cfe7b4f51e62fe	Plankton
5	dd11c105ec8bb3c851f5955fa53eebb91b7dc46bef4d919ee4b099e825c56325	GinMaster
6	6832234c4eae7a57be4f68271b7eecb056c4cd8352c67d2273d676208118871d	FakeDoc
7	f39f20ec060481bd89cfbd44654077fcd6404d87a1286685570334bd430e2f18	GinMaster
8	eb1bcca87ab55bd0fe0cf1ec27753fddcd35b6030633da559eee42977279b8db	FakeInstaller
9	5010f34461e309ea1bc5539bb24fcc320576ce6d677a29604f5568c0a5e6315	Opfake
10	f1c8b34879b04dc94f0a13d33c1e1272bdf9141e56e19da62c1a1b27af128604	FakeInstaller
11	4e355df8f0843afc4a7bfc294ee4b1db9e9b896269c754c2d57dcb647dcd3efb	Opfake
12	ecf9c8520e13054bcc1b1a18cc335810f7eb97bde75fc204ad050228f805216	BaseBridge

Figure 3: .csv file in Drebin dataset containing each malicious file assigned to its malware family

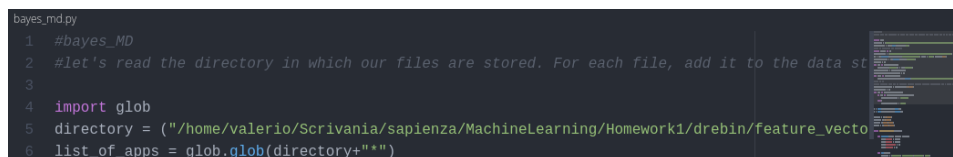
3.1 Structures defined in Python for the usage of Drebin Dataset

First things first, in order to use Drebin dataset to fit the **Naive Bayes classifier** that has to be implemented, the program comprehending the classifier has to be able to manipulate the dataset .

Suppose we want to realize the classifier in *Python* language: the first thing to do, is to make sure the program can access the dataset and split it in order to compute the fitting and evaluating operations.

All the text files of the applications of Drebin dataset are included in folder

"featurevectors" in directory "drebin". So, assuming the directory where the "drebin" folder is located is "/home/", then the directory our program will have to access in order to obtain the text files is "/home/drebin/featurevectors/*". Listing all the files contained in the directory, we can build a list of paths which lead to the text files in the dataset, as pictured in *Figure 4*.



```

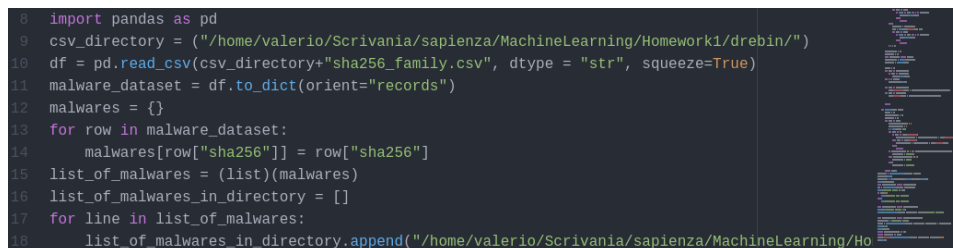
bayes_md.py
1 #bayes_MD
2 #let's read the directory in which our files are stored. For each file, add it to the data st
3
4 import glob
5 directory = ("/home/valerio/Scrivania/sapienza/MachineLearning/Homework1/drebin/feature_vecto
6 list_of_apps = glob.glob(directory+"*")

```

Figure 4: List of paths leading to text files representing the features contained by every application in the dataset.

The list of paths to the applications text files isn't sufficient by itself: we also need to read the **.csv** file containing all the malicious applications in order to implement our classifier.

Since we're only interested in malware detection - and not in malware families classification - what we need to do now is to open the file, read all the **SHA1 hash** names of the malicious files and then come up with a list of paths leading exclusively to the text files containing the features of the malware that are in the dataset, as pictured in *Figure 5*.



```

8 import pandas as pd
9 csv_directory = ("/home/valerio/Scrivania/sapienza/MachineLearning/Homework1/drebin/")
10 df = pd.read_csv(csv_directory+"sha256_family.csv", dtype = "str", squeeze=True)
11 malware_dataset = df.to_dict(orient="records")
12 malwares = {}
13 for row in malware_dataset:
14     malwares[row["sha256"]] = row["sha256"]
15 list_of_malwares = (list)(malwares)
16 list_of_malwares_in_directory = []
17 for line in list_of_malwares:
18     list_of_malwares_in_directory.append("/home/valerio/Scrivania/sapienza/MachineLearning/Ho

```

Figure 5: List of paths leading to text files of malicious applications in the dataset.

Next step is to consider only a part of the dataset obtained by listing all the text files in **Drebin** dataset, due to the fact that processing over 120,000 files could take a considerably large amount of time, so the program will use only 30% of the whole dataset, which is around 30,000 files.

Since there is also a list of the malicious files, it is possible to initialize a *dictionary - training data* - which contains all the applications in the considered dataset and assigns them a value: **malware**, if the application

is also contained in the list of malicious files, and **safe** otherwise. This way, we have obtained a dataset where are stored all the applications we want to use to train the classifier and, furthermore, the applications are already classified according to the **Drebin** dataset.

A small amount of applications will then be used to evaluate and test the classifier, thus these 30,000 files are split in *training data* (70%) and *test data* (30%), as pictured in *Figure 6*: X_{train} represents the text files used for training the classifier, while X_{test} the ones used for evaluation, while y_{train} and y_{test} are the values of the applications - **safe** or **malwares**, that is to say the class they belong.



```

20 #let's consider only 30% of original dataset (Total: 129013), 30%->38703, 70% = 27092 // 20%
21 list_of_apps_reduced = list_of_apps[0:38703]
22 training_data = {}
23 for app in list_of_apps_reduced:
24     if app in list_of_malwares_in_directory:
25         training_data[app] = "Malware"
26     else:
27         training_data[app] = "Safe"
28
29 X = list(training_data.keys())
30 Y = list(training_data.values())
31
32 #split into train data (70%, 27092) and test/evaluation data (30%)
33 X_train = X[0:27092]
34 y_train= Y[0:27092]
35 X_test= X[27092:38703]
36 y_test = Y[27092:38703]

```

Figure 6: Reduce the dataset and then split it in training data and evaluation data.

4 Classification Problem and Target Function

Now that the dataset is ready to be processed and analyzed in order to fit the classifier, the next step is to *define* the **classification problem** we want the classifier to address and - if possible - solve.

In other words, a **target function** needs to be established.

A **target function** is a function which the **classification problem** tries to approximate in order to solve the problem itself. Being a **classification** the problem of deciding to which class an instance belongs [1], in this case the first thing to notice is that the only *two classes* of the problem are **malware** and **safe**.

Drebin dataset could allow also other classes, because malicious applications are classified according to the *family* they belong to, but this means pushing the problem into another, deeper direction, which is not the goal

of the **Naive Bayes classifier** which is going to be implemented: what we want to realize here, is **malware detection**, so just considering these two aforementioned classes is fine for now.

So, let's say the **classification problem** we want to address is the following:

*"given an Android application x in a group of Android applications X , and a set of features extracted from x (which is denoted as $S(x)$), classify x as **malware** or as **safe**."*

It should be pretty clear, at this point, that since the classes addressed are just two, the problem is a **binary classification problem**, that is to say the applications are classified in only two groups.

That's why the **target function** denoted by **F**, has only two possible output values, the aforementioned **malware** and **safe**.

Thus, we define **target function F** as follows:

$$\mathbf{F} : x \in X \rightarrow Y \text{ given } S(x) = (s_1, s_2, \dots, s_{|S(x)|})$$

where $x \in X$ is the new instance (an Android application), described by a set of features (**S(x)**) and **Y = ("Malware", "Safe")** is the set of possible classes of **F**.

Still, this target function is ambiguous: it doesn't tell much about how the features extracted from the application **x** (**S(x)**) is meant to help the classifier to approximate the **target function**.

This aspect need to be clarified step by step: that's why the fundamental role of the features extracted from each application is addressed carefully in the next subparagraph and then in paragraph 5.

4.1 Features Extraction from dataset

Extracting every set of features described in **Drebin** dataset from every application could take a very long time in terms of computational complexity of the algorithm implemented in the **Naive Bayes classifier**.

This is the main reason which actually led to consider only 4 of the sets described in paragraph 3, which are:

- S_1 : requested hardware components, prefix: **feature**;
- S_2 : requested permissions, prefix: **permission**;
- S_5 : restricted API calls, prefix: **api_call**;

- S_8 : network addresses, prefix: **url**.

These sets are actually highlighted in the first line of the **fit** function of the class implementing the **Naive Bayes classifier**, as pictured in *Figure 7*.

```
46     def fit(self):
47         features = ["feature", "permission", "api_call", "url"]
```

Figure 7: The only features addressed by the classifier are those explicated in the *"features"* set.

Of course, adding features to the set or changing the set could lead to different results.

What is important, now, is to comprehend how this features can be used in order to fit the classifier. One could use an approach similar to a *"spam or non-spam"* classifier: in this specific application, a *bag of words* is realized implementing a dictionary, in which every word from messages analyzed is stored and then classified as a *spam word* or a *non-spam word*. Then, the classifier makes his decision by assigning to each word a value, which actually represents the probability of the word of being a *spam word* or a *non-spam word*. Every new instance of a message is then classified by summing the probabilities of *spam words* and *non-spam words* contained in it and, by choosing the higher one, the message is finally classified.

This approach is still valid, even if someone may think it needs to be strengthened in **malware detection**, especially if the attacker who realized the malware is pretty smart.

Suppose to have a *bag of features* for **malware** and one for **safe**. In each bag are stored the lines (features) which come from malicious and benign applications. Let's say each application has an average of 4 *features* stored. Problem is, only one of those four features could be malicious, while the other three could be features that are actually safe, but classified as malicious because they come from a malicious application.

In other words, there are some features that are actually stored in both malicious and benign applications file text since they're general features actually used in every application, so they could be classified as malware features and safe features with the same probability.

Actually, this problem is exactly the same of words in spam messages that could actually be used in non-spam messages: **Naive Bayes'** approach

guarantees excellent results. Thus, to address and solve this classification problem, an implementation of a **Naive Bayes classifier** is sufficient.

5 Naive Bayes Classifier

The purpose of this work is to implement a **Naive Bayes classifier**, which makes use of the **Drebin** dataset and the data structures defined to manipulate and analyze it in *Python* language, in order to predict whether a set of lines representing features extracted from a certain Android application x classifies x as **malware** - malicious application - or **safe** - benign application.

A **Naive Bayes classifier** approximates the **target function** using *conditional independence* and *Bayes rule*.

Assuming the **target function** defined in paragraph 4, given a dataset A of Android applications and a new instance x of an Android application described by a set of features $S(x)$, so that $x = (s_1, s_2, \dots, s_{|S(x)|})$, according to the *Bayes rule* the most probable value of **F** is:

$$y_{\text{MAP}} = \arg \max_{y_j \in Y} P(y_j | x, A)$$

being $x = (s_1, s_2, \dots, s_{|S(x)|})$, we have:

$$y_{\text{MAP}} = \arg \max_{y_j \in Y} P(y_j | (s_1, s_2, \dots, s_{|S(x)|}), A)$$

and, for *Bayes rule*:

$$y_{\text{MAP}} = \arg \max_{y_j \in Y} P((s_1, s_2, \dots, s_{|S(x)|}) | y_j, A) * P(y_j | A)$$

Naive Bayes classifier now makes an *assumption*, so to make the classification process faster and more practical:

$$P((s_1, s_2, \dots, s_{|S(x)|}) | y_j, A) = \prod_i P(s_i | y_j, A)$$

this way, the classifier assigns class to the new instance according to the following:

$$y = \arg \max_{y_j \in Y} P(y_j | A) * \prod_i P(s_i | y_j, A)$$

For each target value y_j in target function **F**, where target values are **Y** = ("Malware", "Safe"), the classifier calculates **P(y_j|A)**, and then for each feature s_i calculates **P(s_i|y_j, A)**.

Then, by calculating the *argmax* function, the instance is finally classified in **Y**.

5.1 Implementation

Every implementation of a *classifier* has at least two functions: **fit** and **predict**.

The **fit** function *trains* the classifier - that is to say, given a dataset A , classifies its instances in order to make use of them later, when a new instance $\notin A$ will have to be classified. Basically, the **fit** function behaves just like the classifying process of a **Naive Bayes classifier** aforementioned.

First, the new class **Bayes_Classifier** is defined, along with the `__init__` function required by every class defined in *Python* language, which is used to initialize the local variables of the class:

- **x_train** is the list of paths leading to the text files in the database;
- **y_train** is the list of **target values** of the elements of the dataset - that is to say, "Malware" or "Safe";
- **probMalware** is a dictionary which will link every feature to the probability of it being malicious;
- **probSafe** is a dictionary which will link every feature to the probability of it being benign;

```
38 class Bayes_Classifier(object):
39
40     def __init__(self, xtrain, ytrain):
41         self.x_train = xtrain
42         self.y_train = ytrain
43         self.probMalware = {}
44         self.probSafe = {}
```

Figure 8: Definition and initialization of classifier class.

First step of **fit** function has already been analyzed: a **features** vector is initialized containing the four features that are to be extracted from the text files.

Considering the i -th index of list **x_train**, **y_train[i]** contains the value of application **x_train[i]** - "Malware" or "Safe".

Going through these two lists, the algorithm opens each text file describing the application contained in **x_train** list looking for the features mentioned in **features** vector: if the application is classified as "Malware", then, puts

every feature contained in the text file in a **malwares** list.

Otherwise, if the application is classified as **"Safe"**, the features are extracted and put in another separated **safe** list: the whole process is represented in code lines from 53 to 71 in *Figure 11*.

Two counters are initialized: **total_malwares** and **total_safe**. They are used to count the number of total malwares and total benign applications (safe) in the dataset, which will be useful later.

```
46     def fit(self):
47         features = ["feature", "permission", "api_call", "url"]
48         total_malwares = 0
49         total_safe = 0
50         malwares = []
51         safe = []
52
53         i = 0
54         while i < len(self.x_train):
55             if self.y_train[i] == "Malware":
56                 file = open(self.x_train[i], "r")
57                 for line in file:
58                     if any(s in line for s in features):
59                         total_malwares = total_malwares + 1
60                         malwares.append(line)
61             else:
62                 continue
63         else:
64             file = open(self.x_train[i], "r")
65             for line in file:
66                 if any(s in line for s in features):
67                     total_safe = total_safe + 1
68                     safe.append(line)
69             else:
70                 continue
71         i = i + 1
```

Figure 9: Features extraction from dataset, counting malwares and safe applications in dataset and splitting them into two separated lists.

As stated previously during the *Features Extraction* subparagraph, some features classified as *malware* could be also classified as *safe*. This happens

because when an application is classified as *malware*, every single feature extracted from it is also classified as malicious, even though the malicious feature is only one out of n features of which $n-1$ are benign.

This causes the other $n-1$ safe features to be classified as *malware* and put in the **malwares** list, leading to a misjudgement.

Since there are many features that both malware and safe Android applications share, one way to achieve this goal is to delete those features from the lists in *Figure 11*. Unfortunately, while giving pretty good results with the **Drebin** dataset, this approach could never work against real threats: it would delete every feature from the lists except for the *url features*, which are of course the ones that are less likely to be found in both *safe* and *malicious* applications.

Anyway, this implementation of the **fit** function is sufficient to solve this problem, since the values of the probabilities assigned to each feature are such that they guarantee the classifier to give excellent predictions as output.

Indeed, now that the features have been extracted and classified, there's an additional step to go through: counting each occurrence of a malware and safe feature and then assign to each line a value in terms of probability, as pictured in *Figure 12*.

```

73 malware_lines = {}
74 safe_lines = {}
75 from collections import Counter
76 malware_lines = Counter(malwares)
77 safe_lines = Counter(safe)
78
79 for line in malware_lines:
80     self.probMalware[line] = (malware_lines[line]+1)/(total_malwares+2)
81 for line in safe_lines:
82     self.probSafe[line] = (safe_lines[line]+1)/(total_safe+2)
83
84 return

```

Figure 10: Assign each feature its corresponding probability of being malware or safe.

Given feature f , number of occurrences of f in malware applications $o(f)$ and total number of malware applications n , the probability of it being a *malicious feature* according to **Drebin** dataset is:

$$P(\mathbf{f} \rightarrow \text{malicious}) = \frac{o(f)+1}{n+2}$$

same value is achievable for probability of feature f being safe.

This way, dictionaries **self.probMalware** and **self.probSafe** are filled with

features and their corresponding probabilities of being classified as "malware" or "safe".

The classifier is now fit and ready to predict.

5.2 Predict function

Function **predict** takes as input a set of text files, each one of them representing an Android application and containing its features.

Its output is, of course, the one of the approximated **target function** that has been chosen, that is to say that it outputs a list of Android applications with their corresponding **target value**, which is "Malware" or "Safe".

This function makes use of three dictionaries: **result**, which is the dictionary that is given as output, **malicious_score**, which assigns to every text file (that is to say, to every application) given as input a score based upon its malicious features, and **safe_score**, which assigns a score to every application according to its benign features.

```
86     def predict(self, data):
87         result = {}
88         malicious_score = {}
89         safe_score = {}
```

Figure 11: Initializing the three dictionaries in **predict** function.

Now, for every text file in the input set, the **predict** function proceeds to open and read it: for each text file contained in input set, a new index of the two score dictionaries is initialized at zero.

Then, every file is read line by line and every feature is analyzed.

In code lines from **94** to **100**, the function updates the **scores dictionaries**: if feature f from application **A** is in dictionary **self.probSafe** but not in dictionary **self.probMalware**, then obviously only **A**'s **safe_score** is updated, same if the feature is only contained in dictionary **self.probMalware** (then only **A**'s **malicious_score** is updated).

Otherwise, if feature f is in both dictionaries, then both of them are updated according to their corresponding probabilities.

If the feature is not in the dictionaries, then the algorithm proceeds to analyze the next line of the text file.

A classic "spam or non-spam" classifier approach still works efficiently, as

shown in next paragraph, and it actually leads to a surprisingly good value of accuracy, precision and a nearly-perfect confusion matrix .

```
90     for file in data:
91         malicious_score[file] = 0
92         safe_score[file] = 0
93         f = open(file, "r")
94         for line in f:
95             if line in self.probMalware:
96                 malicious_score[file] = malicious_score[file] + self.probMalware[line]
97             elif line in self.probSafe:
98                 safe_score[file] = safe_score[file] + self.probSafe[line]
99             else:
100                 continue
101         if malicious_score[file]>=safe_score[file]:
102             result[file] = "Malware"
103         else:
104             result[file] = "Safe"
105
106     return result
```

Figure 12: Finally calculating each score dictionary and final output by confronting malicious and safe score of each file.

6 Evaluation Procedure

In order to evaluate the classifier implemented, three values have been considered: **accuracy**, **precision** and **recall**.

A **confusion matrix** has also been compiled and it is given as output by the *Python* program. The classifier has been evaluated making use of **scikit-learn** libraries.

Since only **30%** of the **Drebin dataset** has been used, **70%** of the data was actually used to train the classifier with **fit** function, while only **30%** of it - which means *10,000* samples circa - was used as *evaluation data* and given as input to the **predict** function.

It's fundamnetal to highlight how the *accuracy score*, especially in this case considering this dataset, is not a highly reliable evaluation method, since by classifying al the *evaluation* samples as "**Safe**", the accuracy score is around **97%**, that is an extremely good value until the other parameters and che **confusion matrix** tell us that more than *400* malwares have not been detected.

So, since the *safe applications strongly outnumber malware applications*, what should be really evaluated here is how good is our classifier in detecting malwares.

The **accuracy** score value tells how many predictions the classifier actually got right:

$$\text{Accuracy} = \frac{\text{NumberOfCorrectPredictions}}{\text{TotalNumberOfPredictionsMade}}$$

but, as stated before, even if this metric can give us an idea of how good the predictions of the classifier are, is still pretty unreliable, being the **target values** of the **Drebin** dataset composed mainly by **safe** applications. Anyway, if the **accuracy score** exceeds **60%**, a classifier can be considered good enough.

Other metrics can - and must - be used for evaluation: **precision** is the number of correct *relevant* predictions amongst the correct predictions made - that is to say:

$$\text{Precision} = \frac{\text{NumberOfTruePositives}}{\text{NumberOfTruePositives} + \text{NumberOfFalsePositives}}$$

which means, "out of all the *positive* predictions, how many of them were actually correct?" Thus, **precision** measures in our case how many applications classified as "**Malware**" were actually *malicious applications*, so it tells us how many *safe* applications the classifier could have misclassified as *malicious*.

For example, if the classifier implemented has **0.4** precision over "**Malware**" class, it means that in **40%** of the cases in which it classifies an application as "**Malware**" it is right - and wrong in **60%** of them.

If number of false positives is zero, the classifier has a precision of **1** - and that is actually what a classifier has to aim for.

Recall is defined as follows:

$$\text{Recall} = \frac{\text{NumberOfTruePositives}}{\text{NumberOfTruePositives} + \text{NumberOfFalseNegatives}}$$

which measures how many actual positives were actually detected - that is to say, it tells us how many *malicious* or *safe* applications our classifier classifies.

If the classifier implemented has **0.8** recall over "**Safe**" class, it actually means that **80%** of the *safe applications* in the dataset were classified correctly as "**Safe**" - and the remaining **20%** were misclassified as "**Malware**".

The **confusion matrix** is often used to visualize the performance of a classifier, so it has been used as well: all the evaluation results of this work are in the following subparagraph.

6.1 Results

Before introducing the **results** obtained thanks to the implementation of the classifier given in this paper, it might be useful to reach a deeper understanding of the evaluation procedure by considering some of the results obtained by different methods.

First, for example, are the results obtained by *deleting* from the data structures implemented in the `fit` function of the **Naive Bayes classifier** - that is to say, `malware_lines` and `safe_lines` - all the features that are in both of the two structures.

The results are showed in *Figure 15* and *Figure 16*.

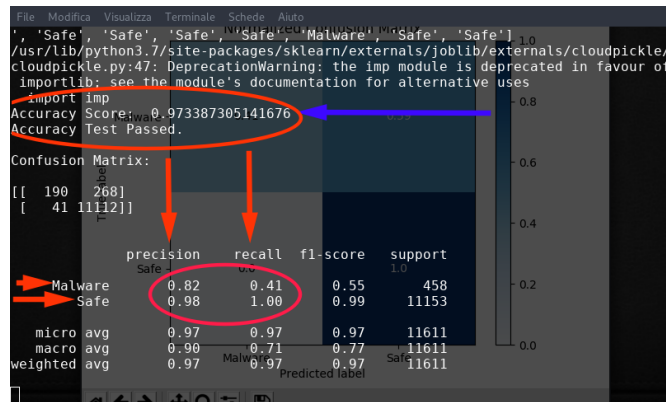


Figure 13: Accuracy, precision and recall scores for a not-so-accurate classifier.

Accuracy, precision and recall values seem to be not only acceptable, but good.

These values are actually *misleading*, if compared to the **confusion matrix** obtained:

This case was discussed previously: even if the accuracy score is very high (**97%**), it is pretty clear just by looking at the **confusion matrix** that the detection is not so accurate: every **safe** applications is surely classified correctly, but almost **60%** of the malicious applications were not detected and classified as **safe** - and that is not a good example of how a *malware detection* tool should work.

The **Naive Bayes classifier** implemented in this paper has a different, better looking **confusion matrix**, even if some scores might actually seem

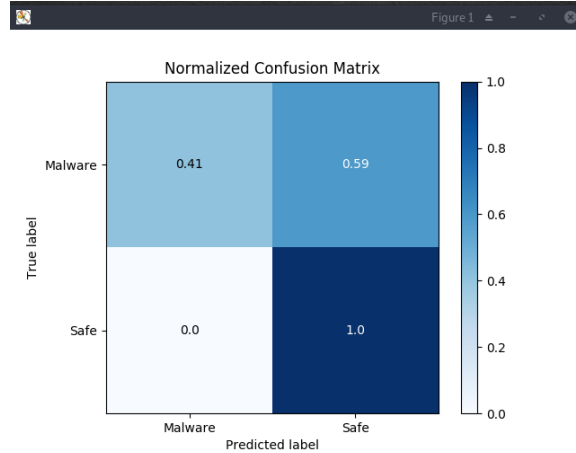


Figure 14: Confusion matrix of a not-so-accurate classifier.

bad.

The **accuracy score** is **88%**, and being the safe applications almost the 100% of the dataset, this is actually a good score.

Precision is surprisingly high for **"Safe"** class (should be **1** for a perfect classifier) and low for **"Malware"** class - it is actually the only value which can and should be improved.

Recall values are around **80-90%** for both classes.

The evaluation scores obtained by the classifier are represented in *Figure 16*.

The **confusion matrix** tells us that over **80%** malicious applications were actually classified as **"Malware"**, while only **19%** of them were actually not detected and classified as **"Safe"**.

On the other hand, **11%** of benign applications were misclassified as **"Malware"**, while the remaining **89%** of them were correctly detected as **"Safe"**. The **confusion matrix** is pictured in *Figure 17*.

Of course, the classifier can be improved.

Its results can get better: we have to consider that *only four features* were actually used for detection, so these results were expected.

Many malwares could hide their malicious features in different set of features that we have not chosen, and only the **30%** of the whole dataset was used, part for training and part for evaluation.

Notwithstanding this, the results on these samples are still good, and the

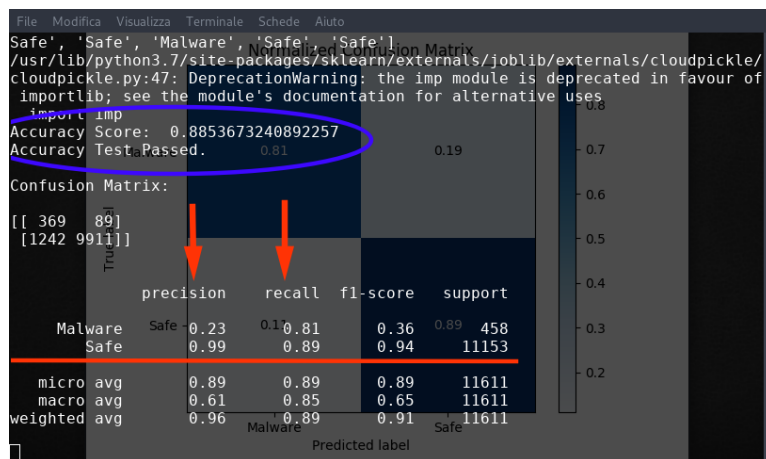


Figure 15: Accuracy, Precision and Recall measured for the Naive Bayes Classifier implemented.

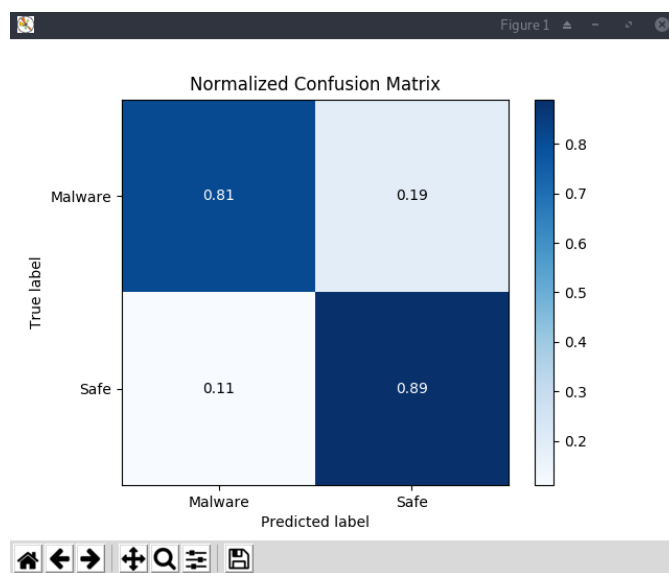


Figure 16: Confusion Matrix of the Naive Bayes Classifier implemented.

classifier implemented can be considered valid and functioning.

References

- [1] "Wikipedia - Statistical Classification."
https://en.wikipedia.org/wiki/Statistical_classification.
Accessed: 2018-10-31.
- [2] "Nikola Milosevic - History of Malware
in CoRR/abs/1302.5392 (2013)" <https://arxiv.org/abs/1302.5392>.
Accessed: 2018-10-31.
- [3] "Wikipedia - Social Engineering"
[https://en.wikipedia.org/wiki/Social_engineering_\(security\)](https://en.wikipedia.org/wiki/Social_engineering_(security)).
Accessed: 2018-10-31.
- [4] "Wikipedia - Mobile Malware"
https://en.wikipedia.org/wiki/Mobile_malware. Accessed: 2018-10-31.
- [5] "Daniel Arp, Michael Spreitzenbarth, Malte Huebner, Hugo Gascon,
and Konrad Rieck - DREBIN: Effective and Explainable Detection of
Android Malware in Your Pocket"
<https://www.tu-braunschweig.de/Medien-DB/sec/pubs/2014-ndss.pdf>.
Accessed: 2018-11-02.