Homework 1 - Malware Analysis

Luigi Russo 1699981

November 3, 2018

1 Introduction

2 Target function

I decided to focus on the malware detection problem, i.e. given an instance of an Android Package (APK) the algorithm has to decide whether the input is a malware or not. To be more precise the goal of the homework is to train a model and find a function $f : APK \rightarrow \{good, malware\}$. The input is not actually an APK, but it has to be the feature vector of the application, as stated in the DREBIN paper.(inserisci nota). So, to be more precise the target function is: $f : \{0,1\}^n \rightarrow \{good, malware\}$, where n is the number of all the labels (features). The malware detection problem is nowadays a feature provided by the most important Android apps stores, e.g. Google Play Store. However, sometimes it is necessary to download unofficial applications (that offer more functionalities than the original ones, for instance) and these APKs are released in forum threads or in online ad-hoc repositories; the common Android user has to know, with high accuracy, whether the app is harmful or not before running it on his device.

2.1 Learning goals

2.1.1 Metrics

Before looking at the data and training the model, I decided to set the learning goals of this homework. I decided to build a classifier with high precision rather than high sensitivity: this led me to focus on the precision rather than the recall of the model; these parameters have been considered during the evaluation phase and will be pointed out in the results section. In an informal way, we can say that high precision is approximately this: if I say that it's a malware, it surely is. In fact, high precision provides a way to measure the positive predictive value: I don't want a good apk to be misclassified as malware. Most people would probably prefer high recall rather than high precision in this context, since they want to be sure that an application is not harmful when it is classified as a good apk. This is certainly the point of view of the common Android user, who downloads the applications from the store. But, if we prefer recall rather than precision, we could have some issues, due to the well-known problem of the presence of false positives:

- an Android developer, after days of hard work, could be denied to publish his harmless application on store and , in general, could never see his app deployed to users.
- there have already been legal issues (court decisions and conspicuous fines) for Anti-virus companies that misclassified software products

2.1.2 Not just one model

In many contexts it is hard to know whether the results obtained are good or not. In general it is better to compare two models, rather than consider a single model, and especially their metrics (accuracy, precision and so on). For this reason I decided to train two different models, a Naive Bayes Classifier, in particular a Bernoulli Classifier, and a Support Vector Machine (**SVM**) model. The first one is generally faster to train, but its simplicity does not always guarantees good results in terms of accuracy, recall or whatever we take as metric. In the final section of this report I will compare the two trained models on the basis of their results.

3 The dataset

The dataset used to train the model is DREBIN. Its authors used the well-known tools of the static analysis to extract features from the APK samples, both from the Manifest.xml and from the disassembled Java code. The dataset contains 123,453 harmless samples and 5,560 malware, and this unbalance has been a tricky problem to face during the training phase. Since the original dataset is provided as a collection of files (whose name is the SHA256 digest of the APK), each containing the features extracted, I had to make some preprocessing in order to train correctly the model.

3.1 Preprocessing

This is in summary what I did in the preprocessing phase: first of all I used the sha256-family.csv file to load into the main memory all the files classified as malware by at least two of VirusTotal Anti-Virus. Then, I iterated over all the files of the dataset and for each of these I build up a feature vector; I trained and evaluated the two models, computing the confusion matrix and I finally plotted the results, ready to be analyzed and compared. I am now going to expose in detail the process described above.

3.1.1 Tools used

I used Python 3.7 programming language, because of its well-documented modules and libraries: I did not want to reinvent the wheel, so I used the well-known ScikitLearn, Numpy, MatPlotLib libraries to build, train, and evaluate the classifier. In the next sections I will give further details on the precise methods used. I provided the source-code as a Jupyter Notebook (extension .ipynb) because it is easy both to read and to write snippets of code in this way. Moreover its checkpoints help to save intermediate results and show debug infos and images in the notebook itself.

3.1.2 The unbalance problem

The original dataset is very unbalanced: there are 123,453 harmless samples and 5,560 malware; and this was a big problem to handle in the training phase: for instance, when I tried to train the SVM model at the beginning, in order to provide high accuracy it did not spot correctly the malwares samples, because of the excessive difference between the number of good samples with respect to the malware ones. The model, at the end, simply classified whatever it was passed in input as a good application, thus resulting in very high accuracy (about 97%), but it was absolutely useless for the purpose of this homework. I finally decided to adopt a simple strategy, that helped me to overcome this issue: I *forced* the dataset to be balanced, fixing the ratio $\frac{\#malwares}{\#samples}$ at the beginning of the training phase. For the SVM model, I also set the **balanced** parameter to true: in this way, the model automatically adds new malware instances to the training dataset to have a well balanced dataset: but we cannot abuse of this functionality, because if the malware_ratio is very low, e.g. 0.1, this option tends to overfit the model, thus producing good results in the training phase and very bad ones in the validation counterpart. For this reason the malware_ratio has been fine-tuned and finally fixed to 0.4. because it provided very good results for both models.

3.1.3 Load malwares

The DREBIN dataset provides the list of all malwares in a distinct file called sha256_family.csv, that also gives further details about the malware family. I decided to load all malwares into main memory, and saving them in an hash-set (a Python set), because this data structure guarantees a constant lookup time. It was very easy, in the later phases, to check for the "harm" of a sample, simply looking to this hash-set.

3.1.4 Load labels

Once I had all malwares sha256 digests saved in an hash-set, I could focus on the features of each sample to take in consideration: the original dataset has an enormous number of features, divided and organized in 8 major sets: I decided to focus only on four of these sets, api_calls (restricted API calls, whose access requires permission), requested_hardware_componenets (GPS, camera, etc.), urls (network addresses embedded in the code) and activities. It is important to note that a different choice of these sets leads to different results (in some cases, completely different). The trade off here was to choose enough and proper sets to allow the model to correctly train, and doing it possibly fast. Adding other sets (or even all) does not guarantee better results, and can be very time and resource consuming. At the end of this phase, I had 32642 different labels and I saved them in an hash-map (a Python dictionary): in this case it is crucial to define a stable order between the labels: every label is mapped to a number $i \in \{0, 32641\}$. In the next section it will be clear how this is only the first phase of the one hot encoding process of the labels.

3.1.5 Prepare the dataset

As stated previously, I decided to use already implemented algorithms, provided by Sklearn library. Its models accept as input numpy arrays and matrices. So, iterating all over the files of the dataset, I built up the dataset matrix: each column was the feature numpy vector of a particular file; for each vector v_i, representing a sample APK, $v[i][j] = 0$ if the feature j is not present and $v[i][j] = 1$ if it is present.

As for the SVM model, the matrix was built in a similar way but the values were slightly different: for each vector v_i, $v[i][j] = -1$ if the feature j is not present and $v[i][j] = 1$ if it is present.

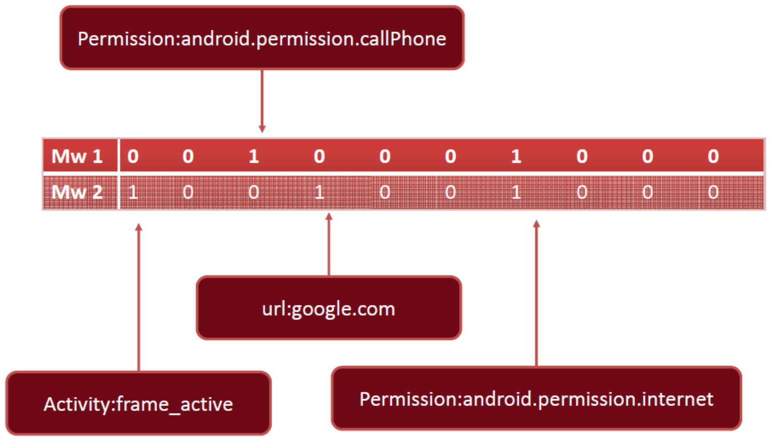

Figure 1: One Hot Encoder Bernoulli example, each feature is mapped to a position of the vector and marked as 1 if it is present, 0 if it is not.

3.2 Training the models

At this point I was able to pass the dataset to the SVM and Bernoulli Sklearn models. After the models were fitted, I used the cross_val_score and cross_val_predict methods of Sklearn to compute, in a cross validation with k = 5 folds the accuracy of the models and, later on, the confusion matrix. The results are presented in the next section.

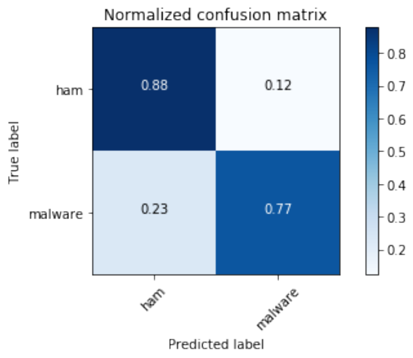

Figure 2: Bernoulli confusion matrix

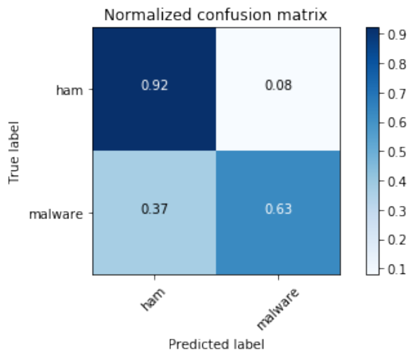

Figure 3: SVM confusion matrix

4 The results

All these results have been computed with the cross validation process.

4.0.1 Bernoulli

This model had an acuracy of 0.83, a recall of 0.81, and a precision of 0.77.

4.0.2 SVM

This model had an acuracy 0.80, a recall of 0.63, and a precision of 0.84.

The confusion matrices, built during the cross validation process, have been plotted with MatplotLib and are shown below:

5 Conclusions

Since the goal was train a model with with high precision, we could pick the SVM model as the better of the two, since it is evident that 84 ¿ 80. But, 0.8 precision is still a high value. If we look at the recall, we can note that Bernoulli performs way better than SVM 76 vs 63. Moreover, Bernoulli has been much faster to train on my machine. For these reasons I think that Bernoulli model, in this case, is the better.