Applying machine learning to malware analysis: Android malware detection

Riccardo, Chiaretti, 1661390

2/11/2017

1 Description of the problem

The problem for a machine to be able to discriminate a malware software from a benign one, is a matter of great importance.
There are lots of possibilities to overcome this problem each one based on different approaches: some techniques are oriented to *signature-based* analysis which use, for example, to compare some patterns of an incoming software to some other patterns stored inside a signature database. These types of techniques however, are vulnerable to obfuscation since that, during time, a specific malware could change its own behaviour and easily "skip" a check like this.
On the other hand, using machine learning, we can carry out analysis built on the *semantic* of an application, therefore, analysis which are not subjected to obfuscation due to code appearance. Using machine learning we can afford different types of analysis such as category detection, variants selection and so on [1].
In the following, the problem we are going to deal with, is a *malware detection* problem in the form of a supervised learning: given a set of Android applications, for each of which we already know the membership class, we want to find a function that given an application, it can infer the correct membership class. Since that we have only two classes, which are malware and non-malware, this is a binary classification problem.

2 DREBIN dataset

At first, before describing the learning algorithm, we should illustrate which is the dataset on which we are going to extract the features and how this

dataset is structured.
For the learning problem, we have considered a dataset of real Android Applications called DREBIN, composed of 123,453 benign applications and 5,560 malware samples. In DREBIN dataset each application has a lightweight representation indeed, it is a text file (whose name is the output of the SHA1 hashing function given in input the apk of the application) within which there is a list of features. These features are representative for the application since they extracted from both the *"AndroidManifest.xml"* and the disassembled code of the application itself.
The format in which each feature appears is:

prefixname::value

Where **prefixname** indicates the set to which the feature belongs to (for example the prefix **feature** belongs to the set Hardware components, while the prefix **call** belongs to the set Suspicious API call and so on) and **value** represents the particular value concerned the belonging set [2].
In the following you can find a table showing the match between the prefix and the set to which it belongs to while, in Figure 1, there is an explanation of what kind of requests characterize each set.

PREFIX	**SET**
feature	S_1 Hardware components
permission	S_2 Requested permission
activity	S_3 App components
service_receiver	S_3 App components
provider	S_3 App components
service	S_3 App components
intent	S_4 Filtered intents
api_call	S_5 Restricted API calls
real_permission	S_6 Used permissions
call	S_7 Suspicious API calls
url	S_8 Network addresses

3 Selection of the dataset

Below we will explain how the features have been extracted from the DREBIN dataset, to be then processed by the learning algorithm.
In order to select a portion of files from DREBIN dataset, we use the `select_files_dataset()` function that takes in input (apart from the path

Feature set	Explanation
S_1 Hardware features	App uses %s feature %s.
S_2 Requested permissions	App requests permission to access %s.
S_3 App components	App contains suspicious component %s.
S_4 Filtered intents	Action is triggered by %s.
S_5 Restricted API calls	App calls function %s to access %s.
S_6 Used permissions	App uses permissions %s to access %s.
S_7 Suspicious API calls	App uses suspicious API call %s.
S_8 Network addresses	Communication with host %s.

Figure 1: This figure is taken from [2] and explains what kind of requests characterize each set.

in which the DREBIN directory is stored) the desired number of files to be extracted from DREBIN and, choosing randomly, returns a pair of dictionary data structures. In these dictionaries we have pairs *(key : value)* and, in particular, in the first dictionary (*app_category*) the key is the name of a file and the value is the membership class while, in the second one (*app_features*), the key is still the file name however the value is a list (an empty list for the moment) which will be filled up in the following step.
Throughout the procedure above what we did was just choose, from the entire DREBIN dataset, the names of the applications we will use as dataset, but now we should collect all the features associated to each application.
Since that it is infeasible to read the features of a file each time they are needed, the `collect_features()` function, which takes as input the output of the function `select_files_dataset()` and an additional parameter, is the one which has the task of filling up the list of features for each chosen file contained in the *app_features* dictionary. The additional parameter that it takes in input is a list containing the features, from all possible features listed in the table above, in which we are interested in. This is done so that we can let the algorithm focus on a subset of features instead of choosing the whole set.
In addition to all this stuff, we perform another operation on the feature we are reading: since that each feature appears in the form we have mentioned in the previous section and since that the prefix is useful only when choosing the subset of feature that we are going to consider, we filter the string deleting the prefix and taking in consideration only the value of that prefix. Another reason is that there is no need to associate a prefix to its value since that each value is unique and there are nott two identical values belonging to different categories.

At the end, what we will give in input to the learning algorithm from what we have computed up to now, are the two data structures containing the match between *(file : list of features)* and *(file : membership class)*.

4 The naïve Bayes classification algorithm

The `learn_naive_bayes()` function is one of the most important since, given a list of files and the data structures containing all the files with their features and their membership class, its task is to apply the naïve Bayes approach to solve the learning problem.
The naïve Bayes classifier (NBC), which is a particular "instance" of the Bayesian learning, uses a probabilistic model and, in particular, it is based on the Bayes theorem with the strong assumption of conditional independence (hence the naïve adjective) among all features.
The criterion on which the algorithm described below has been developed, follows the same logic of the text classification algorithm where we have the analogy between the message and the Android application, and between message words and application features.
For each class of the problem at first we retrieve the subset of files, among those belonging to the training set, whose membership class is the one which we are processing and then we compute the bag of features. The bag of features or simply the vocabulary, is the set of all different features characterizing the all files in the training set; this vocabulary is used later in the algorithm to compute the probability that a feature appears in a file given the membership class.
In the second step we compute the class prior probability dividing the number of files belonging to that class by all the files in the set in input and then, for each feature in the vocabulary, we compute the (conditional probability) probability of that feature conditioned by its occurrence in that class.
What is important to store after the learning procedure, are three data structures containing:

- one of them contains the matching between class and prior probability associated to that class;
- the other two are needed to maintain the matching between feature and its conditional probability for both the classes.

All this stuff is needed in the following description since that what we are going to do, is to evaluate the output function of the learning algorithm.

5 The K-fold cross validation

Since that the chosen evaluation method is the k-fold cross validation, the entire procedure described above is done sequentially for each iteration of the loop in the `k_fold()` function.
Indeed, each time we execute the learning algorithm, we give in input to it a subset of all the dataset (called training set), while evaluating it by classifying the dataset except for the samples contained in the training set (this is called test set).
What does evaluating means? In this case it means that, given the conditional probabilities of each feature with the respect to the class and the prior probability of the classes themselves, we see how our classifier will classify instances it never processed ("it has never seen") and check if that instance has been correctly classified or not (since being a supervised learning we know to which class it belongs to).
Figure 2 shows the formula used to predict the class; in particular:

- $\hat{P}(c_{\mathrm{j}})$ is the prior probability of class c_{j} (in our case we have the malware and non-malware classes);
- $length(d)$ is the length of the list of features related to the file d;
- $\hat{P}(w_{\mathrm{i}}|c_{\mathrm{j}})$ is the conditional probability of the feature w_{i} with the respect to class c_{j};

At the end, the class predicted (v_{NB}) by our classifier is the one maximizing the product between the prior probability of the class itself and the product of conditional probabilities of features with the respect to that class.
The assumption of conditional independence among features is expressed by the second term of that formula; in the following we will see that this criterion is not the best way to predict the class of an application.

$$v_{NB} = \underset{c_j \in C}{\operatorname{argmax}} \hat{P}(c_j) \prod_{i=1}^{length(d)} \hat{P}(w_i|c_j)$$

Figure 2: This is the criterion with which the algorithm predicts the class according to the conditional and prior probabilities.

6 Comparisons between executions

In this learning problem, there are lots of combinations we can give in input to the learning algorithm, for example we can change the number of inputs files, we can give in input different subsets of all features taken in consideration by DREBIN, we can also change the parameter k of the k-fold CV.
In order to evaluate the classifier with different combinations of input parameters, we use, as evaluation metric, the *confusion matrix*: for each instance, in each row of this matrix there is the real membership class while in the column we have the class in which the classifier had classified it.
At each evaluation of the output learning function, since that it is a supervised learning, we know in advance which is the real membership class so we can discriminate if the classification is done correctly or not; thus, we should introduce the following notions:

- *true positive*: percentage of malware application that the classifier has classified correctly;
- *false positive*: percentage of non-malware samples that has been detected as malicious software;
- *true negative*: percentage of non-malware files that the classifier has correctly classified;
- *false negative*: percentage of malware software that the classifier has classified as a benign one.

At first, in the following, we are going to analyse the output of the learning algorithm considering different number of files but varying the set of extracted features from the files of the dataset.
Before starting analysing the problem above, we should say something important about the distribution of files in the DREBIN dataset related to the approach we have developed up to now: as we have described above (Section 2) the DREBIN dataset has a "disproportionate" distribution, in the sense that, the non-malware samples are many more than malware ones.
Why is this statement so significant in this situation? We have already described the model on which is based NBC (which is a probabilistic model) so, preserving that distribution of files, we can imagine which will be the output function given whatever subset of files in DREBIN (maintaining its distribution). Indeed, as you can see in Figure 3, when we give in input to the algorithm a subset of files without changing the distribution, the accuracy is always more than the 95% and it doesn't change too much even if we

NUMER OF SAMPLES	NUMBER OF MALWARE SAMPLES	NUMBER OF NON-MALWARE SAMPLES	LIST OF FEATURES	NUMBER OF FOLDS	ACCURACY
5000	208	4792	All set of features	10	95.84%
5000	224	4476	'api_call', 'service_receiver', 'real_permission'	10	95.52%
3000	119	2881	'url', 'permission', 'service_receiver', 'real_permission'	10	96.03%
2000	60	1940	'url', 'service', 'call'	10	97.0%

Figure 3: This table shows a comparison between more executions of the algorithm maintaning the original distribution of the DREBIN dataset.

choose a different number of samples. In particular, the less malware there are, and the greater the accuracy since that the number of misclassification is exactly the number of malware in the test set.
The explanation of this behaviour is due to the original distribution of DREBIN (the percentage of benign applications is 96% while the other one is about 4%) indeed, even though the classifier will classify all the malware applications in the non-malware class, the accuracy will decrease only by 4% or 5% on the total giving us what we expected to have that is, an accuracy of about 95%. This is indeed our case (see Figure 4) because, as you can see,

		Output of learned function	
		MALWARE	NON-MALWARE
Ground truth	MALWARE	0%	100%
	NON-MALWARE	0%	100%

Figure 4: In this figure you can see the confusion matrix related to all executions of Figure 2.

the classifier is not able to distinguish a malware application from a benign one; in fact, the true positive percentage is equal to zero as well as the false negative is equal to the number of all malware files in the test set. This is because, independently from the considered features, the prior probability of the non-malware class (which expresses the probability of finding a benign file among those in the training set) is much greater than the one of malware class so using the formula in Figure 2, the NBC will always give in

output the non-malware class.
Even choosing a smaller or bigger number of files, the accuracy remains almost the same.
Another analysis which we want to mention is the one done taking in consideration the same number of malware and non-malware files. In this case the

NUMER OF SAMPLES	NUMBER OF MALWARE SAMPLES	NUMBER OF NON-MALWARE SAMPLES	LIST OF FEATURES	NUMBER OF FOLDS	ACCURACY	RECALL	FALSE POSITIVE RATE
10000	5000	5000	'api_call', 'service_receiver', 'real_permission'	10	59.31%	59.38%	40.76%
10000	5000	5000	Set of all features	10	53.99%	59.00%	51.00%

Figure 5: This table shows a comparison between two executions of the algorithm given in input the same number of malware and non-malware samples.

classifier in no longer classifying all applications as non-malware but even if the accuracy decreases a lot from the previous case, somehow it is able to recognise a malware from a benign application. In the first case, we took in

		Output of learned function	
		MALWARE	NON-MALWARE
Ground truth	MALWARE	59.38%	40.62%
	NON-MALWARE	40.76%	59,24%

		Output of learned function	
		MALWARE	NON-MALWARE
Ground truth	MALWARE	59.02%	40.98%
	NON-MALWARE	51.04%	48,96%

Figure 6: These are the confusion matrices related to executions in Figure 4. In particular, the first one is related to the execution in the first line, while the second matrix is related to the second line execution.

consideration all possible features obtaining as a result a good *recall*, that is a percentage telling how many malware the classifier has detected among those in the test set, however we also get more than 50% as *false positive rate* which means that among all benign applications, we have classified more than the half of them as malware.
A better outcome, as we can see in the first row of Figure 5, can be obtained

using a particular subset of features among all possible ones. Indeed, even if in this case we have obtained more or less the same percentage of *recall*, the *false positive rate* is ten percent less than the previous case which in turn means that, given that list of features, the classifier is able to distinguish better non-malware applications from those malware thus, increasing the accuracy too.

7 Remarks on the naïve Bayes classifier

In conclusion what we can say is that the naïve Bayes approach is not the best way to solve the problem of detecting if a software can be considered a malware or not. The reason why this is not a good classifier is due to the conditional independence assumption among features.
Indeed, we can not classify an application by only considering the occurrence of a feature in its disassembled code or in its manifest but, a better way, could be considering the presence of that feature related to the other requests done by the application itself. For example an application which accesses to the GPS *and* to the network modules can send information to a possible attacker so, it is more suspicious unlike an application which accesses to the GPS *or* to the network modules.
Thus, a better classifier could be the one using *kernels* and in particular, using *Support Vector Machines (SVM)* which are able, in this case, to represent an application according to the relation of features and requests in the application itself.

References

[1] Luca Massarelli, *Machine learning for malware analysis.*

[2] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck, *DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket.*