Machine Learning for Android Malwares

Pietro Spadaccino, 1706250

Homework 1

1 Introduction

Our goal is to implement a malware classifier for android applications. This classifier should be able to tell if a certain sample is a malware or it is safe based on some observed features, both static, like strings or permissions, and dynamic, like system calls or run-time permissions. We will show how to implement such a classifier and how to train it on the DREBIN dataset.

2 DREBIN dataset

The DREBIN dataset includes the features of almost 130000 applications. There are 5560 samples of malwares, while the remaining are safe. The dataset reports also the family for each malware, with a total number of 179 families.
The features of each application are listed in the directory $feature\_vectors$, where each file is named with the SHA-1 of the sample and it's composed by a series of lines having the structure *<feature-class>::<feature-instance>*.
The informations about the malware family are encoded in a *.csv* file, reporting the SHA-1 of the malware and its corresponding family. We can understand if a sample is safe or not by checking if it appears inside this file, if it does then the sample is a malware.
There are 10 classes of features and a total number of feature instances of more than 500000. This half-million occurrences is not evenly split among all the classes: only three types of features ($url, activity, service\_receiver$) count the 97% of all possible feature instances, while the remaining 3% is split among the other seven; the total distribution is reported in Figure 1.

3 Problem defining

We want to implement a binary classifier for malware detection: given a list of features of an application, our classifier should be able to recognize if it's a malware or not. More precisely, we want to learn a function $\hat{f} : X \to Y$ where $Y = \{0, 1\}$ and $X \subseteq \{f_{1,1}, f_{1,2}, ..., f_{1,n_1}, f_{2,1}, ..., f_{m,n_m}\}$ where $f_{k,i}$ is a feature of the application of class k with identifier i.

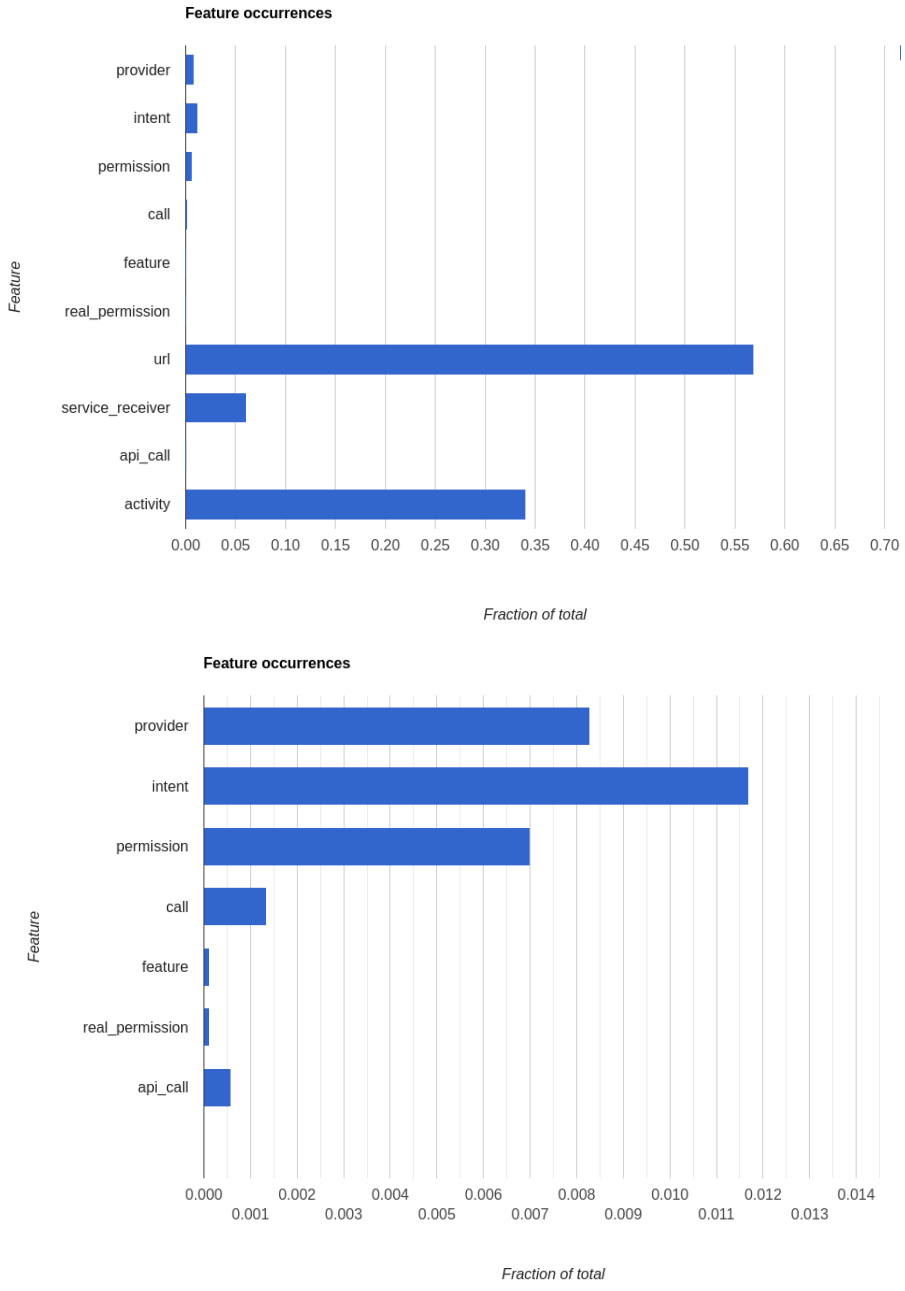

Figure 1: Distribution of 500000 occurrences of feature classes. The upper image reports all the features, while the bottom one is a zoom of the first one, reporting only classes with few elements.

4 Problem approach

After seeing the distribution of feature instances, I decided to use only a subset F for the classifiers, with $F \subset \{k_1, k_2, ..., k_m\}$. This was due to the huge difference in number between some classes of feature, and it is explained in more detail in section 6.
I thought that using a single SVM could be infefficient, because there are different types of features, so I trained one SVM classifier for each different kind of feature, trying to facilitate the linear separation by the SVM. Each classifier f_k is a binary classifier and corresponds to a certain kind of feature f_k. When analyzing an element $x \in X$, it classifies it using only features of type f_k, so each classifier is learning a function $\hat{f_k} : X_k \to Y$ where $Y = \{0, 1\}$ and $X_k \subseteq \{f_{k,1}, f_{k,2}, ..., f_{k,n_k}\}$, meaning that X_k is composed only by features of kind k of the application, for $k \in F$. In other words, each classifier is responsible to classify a sample based only on *one* feature class. Therefore, my final classification function calculates the prediction of each classifier and then outputs the final verdict:

$$y_k = clf_k(x_k) \text{ for each } k \in F \tag{1}$$

$$y = g(y_{k \in F}) \tag{2}$$

Once a prediction is made by each classifier, they are combined into a final prediction by a function g, in Equation 2. The function g is a policy that constructs the final verdict based on the output y_k of each classifier. It can be as simple as a "voting" policy, outputting the prediction of the majority of the classifiers (see section 8 for results).

5 Encoding strategy

Because of how the dataset is structured, some preprocessing was needed before training the classifier. I want to find a way to encode the content of all DREBIN dataset, so that but first I have to find a way to encode the data only once for further reuse.
I decided to use a linear SVM as classification algorithm, so we need to encode the data in a one-hot way, and, because of the high number of possible features, is more efficient to use a sparse vector, saving only the indices instead of the whole vector.
I started to implement the encoding of malware families via a vector f_{fam} and a mapping function $f_{fam} : H \to FAM$, where H is the set of all hashes of malware samples and FAM is the set of all possible families. To let this function be usable by the classifiers, the families should be one-hot encoded instead of being ASCII strings. For this purpose I use the a vector v_{fam}, that contains all possible families (as ASCII strings), one family per element with no repetitions. The idea is to make f_{fam} return a family represented not by an ASCII string,

but by an identifier. This identifier i of a family $family$ is the index of the vector v_{fam} such that $(v_{fam})_i = family$.
Even if we are classifying an element e in a binary way, so we don't need information about its family, we can know if e is a malware by checking if its SHA is contained in the hashes of all malwares, $\text{SHA}_e \in H$. In addition, if we want to, we are free to change the classifier to recognize also the families using this same data encoding.
Now I can start to encode the features of the dataset's samples. My goal is to create a vector $v_{samples}$ where each element is a tuple $(v_{samples})_i = (sha_i, features_i)$, where sha_i is the SHA of the i-eth file and $features_i$ the encoded features in the i-eth file. The information about the SHA of the file is needed to later use the function f_{fam} described above: the idea is give $features_i$ to the classifier, and then compare its prediction $y_i \in \{0, 1\}$ with $\hat{y}_i$, where $\hat{y}_i = 1$ if $SHA_i \in H$, otherwise $\hat{y}_i = 0$.
I can encode $features_i$ as a list of one-hot encoded vectors, one for each feature class. Let be v_k a vector of size equal to n_k (the number of all possible occurrencies of the feature k), then $features_i$ is a list of vectors v_k with the following property: a component $(v_k)_j$ is set to 1 if the feature $f_{k,j}$ appears in the file i, otherwise it is 0.
This encoding respects the split between features classes, making easy to give different vectors v_k in input to different classifiers clf_k. So the pipeline for making a prediction of the i-eth file given this encoding is: for each in $features_i$ and for each vector v_k in $features_i$, calculate $y_k = clf_k(v_k)$ and then combine the predictions of all classifiers $y = g(y_{k \in F})$.

6 Prior testing

Some test scripts were made before starting to write down the actual code. Here we remember that the malware samples are a lot less than the safe samples in the dataset, around than 4%. The data on which the classifiers must be trained should have a ratio of malwares and non-malwares of around 0.5, in order to not create biases. To test if this balance is necessary, I trained some classifiers for different feature with an unbalanced dataset. The results reported in Table 3 and 4 show that this balance is needed.
Hence, if the dataset contains around 5000 malwares, the training set should have size of about 10000 samples, less than one tenth of the whole dataset.
This limit put a constraint of the type of features to use, or, in other words, which feature kind k put in the set F: if a features has a lot of different occurrencies, I may assume that is not feasible to train a good classifier on that feature. For this reason I made a script that counts all occurrencies for all class of features, the results where already mentioned in the section describing the DREBIN dataset. I then selected a subset of 5 feature classes that had the fewest number of different occurrences:

$$[call, feature, api\_call, permission, real\_permission].$$

Table 1: Confusion matrices of singularly-evaluated classifiers. The rows represent the ground truth, while the columns are the the prediction of the model. Each classifier was trained and evaluated on the same training and evaluation set, having both ≈ 0.5 ratio of malwares/non-malwares. The metrics are reported in Table 2.

	api_call		call		feature		permission		real_permission	
	F	T	F	T	F	T	F	T	F	T
F	41%	8%	40%	7%	40%	10%	46%	3%	43%	6%
T	11%	40%	11%	42%	8%	42%	5%	46%	11%	40%

Table 2: Metrics of the classifiers trained using a training set with balanced ratio of malwares/non-malwares.

	api_call	call	feature	permission	real_permission
Accuracy	88.6%	80.45%	79.3%	91.15%	84.85%
Precision	98.67%	98.41%	98.89%	99.43%	98.61%
Recall	89.29%	80.88%	79.26%	91.27%	85.37%
False positives rate	29.74%	29.07%	19.77%	11.63%	26.74%
F-measure	0.94	0.89	0.88	0.95	0.92

Then I decided to train and evaluate the performance of one classifier for each feature without combining their prediction. As shown in Table 1, the results weren't too bad and they were similar among the classifiers.

7 Implementation

The code was split into two parts, the encoding, which preprocesses the data as described previously, and the model, which implements the classifier logic.

7.1 Encoding

I thought that before starting experimenting with the model, it was better to first encode the whole dataset as explained in section 5. My idea is to encode

Table 3: Same as Table 1, but the training set and evaluation set had a ratio of malwares/non-malwares of about 5%, the same as the whole dataset. From here, it is clear the importance of having a balanced training set. The metrics are reported in Table 4.

	api_call		call		feature		permission		real_permission	
	F	T	F	T	F	T	F	T	F	T
F	95%	1%	96%	0%	95%	0%	95%	1%	95%	1%
T	2%	2%	4%	0%	5%	0%	1%	3%	3%	1%

Table 4: Metrics of the classifiers trained using an unbalanced training set. Notice that the accuracy is high because the evaluation set was also biased, being unbalanced as well. In this way the model outputs a lot of true negatives, thus increasing the accuracy. If we run the test with the same classifiers but on a balanced evaluation test, we can see that the accuracy drops down to around 50% for all features, while the other metrics remain similar.

	api_call	call	feature	permission	real_permission
Accuracy	97%	96%	95%	98%	96%
Precision	66.67%	$<< 1\%$	$<< 1\%$	75%	50%
Recall	50%	$<< 1\%$	$<< 1\%$	75%	25%
False positives rate	1.04%	$<< 1\%$	$<< 1\%$	1.04%	1.04%
F-measure	0.57	$<< 1$	$<< 1$	0.75	0.33

the dataset only once, save it to file as a `pickle` archive, so that the model can load it. One of the advantages of using this strategy is that once the dataset is encoded, the model don't have to encode anything, saving CPU time. Another reason could be the reduced size of files on disk and on memory: the drebin dataset unzipped is around 300 MB, while the encoded dataset pickled is just above 40 MB. One drawback is the long time to perform this encoding, almost one hour, even though it is done only once.
The Python functions that performs this encoding are located in the file `encoding.py`. They are essentially four, two for encoding families and the other two for features. In both cases, one function generates a list of all possible occurrencies as ASCII strings, while the other one-hot encodes every sample using as indices the vector provided by the previous function.
Note that a "classic" one-hot encoding would be too inefficient in terms of memory consumption: I would have huge zero-vectors with only few components set to 1. A solution to overcome this problem is to save only the indices where the vector is non-zero. In this way we have implemented a sort of sparse vector.

7.2 Model

The classifier was implemented in Python3.6 using sklearn library for the SVM. What does the model do is: loading the encoded data, perform some random split to generate training and test set, train some classifiers (one for each selected feature) and evaluate the trained classifiers combined.
Going by order, loading the encoded data is simple, just load the `.pickle` file: all the samples are now accessible as a vector $v_{samples}$ and the mapping between a sample and its family f_{fam} is available through a standard dictionary. The split between training set and test set is simple as well. If it is not needed a malware/safe ratio of 0.5 the split is made by just a random sampling, otherwise an iteration on the dataset is required to select malwares. The model don't have to perform any encoding on data, though it needs to do some preprocessing to make the training set and the test set viable for the classifiers, both the input and the ground truth vectors. For the ground truth is created a vector where

each element i is 1 if it is a malware, 0 otherwise. For the input vector this preprocessing includes transforming the vector $v_{samples} = (sha, features)$ into $v_{samples} = (features)$, then deleting all the features classes except for the one of the current classifier, and finally converting the input vector to a classic one-hot encoding representation.
Finally, the data is given to the different classifiers. Once they output their prediction, the model must combine them to obtain a final one. The question is now, in which way combine them, or, in other words, how to choose the function g that appears in equation 2. We will see that changing this policy will vary a lot the performances of the final classifier, allowing us to balance recall and false positive rate.

8 Results

In this section are reported the performances of the final classifier, expressed with the metrics of precision, accuracy, recall, false positives rate and F-measure. As the combining function g, I used a voting policy: if the number of positive predictions is greater than a threshold t then the sample is classified as positive (malware), negative otherwise (non-malware). The results are reported in Table 5 and 6 and in Figure 2.

Table 5: Confusion matrix of the final classifier using different values of the treshold t. The matrix is in the form (ground truth x value predicted), so in the rows there are T and F for samples being malware or non-malware, and in the columns there are T and F for the prediction of the classifier.

	$t = 1$		$t = 2$		$t = 3$	
	F	T	F	T	F	T
F	31.5%	17.7%	40.9%	8.4%	44.6%	4.7%
T	1.3%	49.5%	2.6%	48.1%	7.3%	43.4%

Table 6: Metrics of the final classifier with different values of t.

	$t = 1$	$t = 2$	$t = 3$
Accuracy	87.9%	88.95%	80.95%
Precision	85.84%	93.92%	96.04%
Recall	90.37%	82.96%	64%
False positives rate	14.5%	5.23%	2.56%
F-measure	0.88	0.88	0.77

As expected, the value of t influences the results and some metrics. In table 5, we can see the number of false positives decreasing by increasing t. Remembering that false positives samples are non-malware classified as a malware, this is an expected behavior: if we increase the voting threshold, a sample is less likely to be classified as a malware. On the other hand we notice also that the

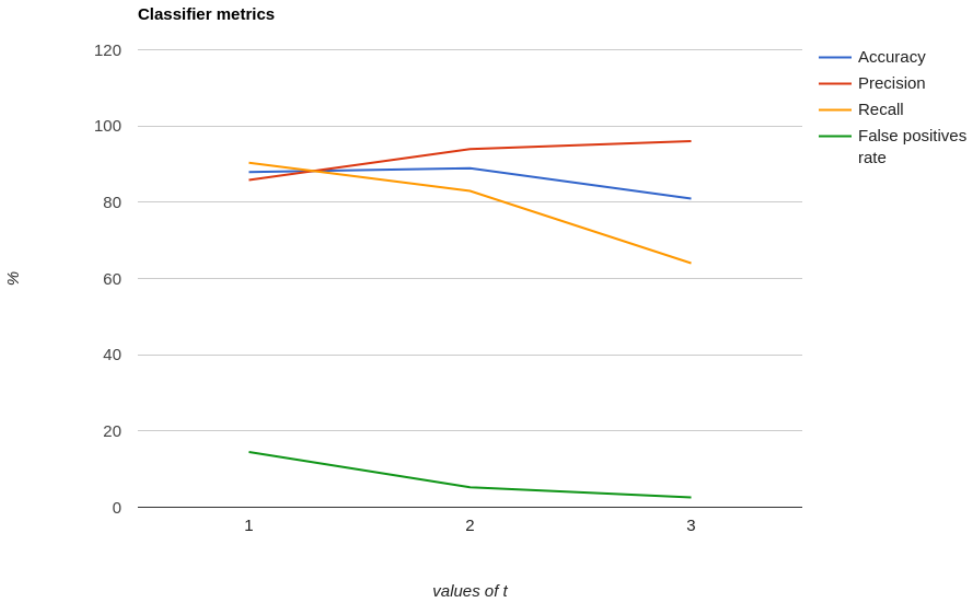

Figure 2: Metrics of the classifier changing with values of the treshold t.

number of false negatives increases, where a false negative is a malware classified as a non-malware.
This behavior influences all the metrics, and in particular false positives rate and recall: the former is the ratio between the number of false positives over the number of all non-malware samples, while the latter is number of true positives over the number of all malwares and can also be seen as: $1 -$ [false negatives rate]. For $t = 3$ we can see that the classifier has a low false positives rate and also a fairly high precision, thanks to the fact that at least 3 out of 5 classifiers must agree that a sample is a malware before classifying it as such. This implies that, in order to have a false positive, at least 3 classifiers must have done the incorrect prediction, which is unlikely. On the other hand by using $t = 1$ we have a false positive rate that is six times more than the previous one, but we have a recall of 90%, meaning that if a sample is a malware is very likely to be spotted.
Given that we are implementing a malware detector, the recall metric is probably the most important. Intuitively, if a certain sample is a malware, the recall measures how well the model spots it. Indeed, the worst thing that can happen for an antivirus is classifying a malware as a non-malware, thus having a false negative.

9 Conclusions

We have implemented a machine learning model capable of classifying android malwares. This classifier presented some good performance and, as we have shown, can be tuned to give more weight to one or another evaluation metrics.