# Mandroid: Android malware detection with machine learning techniques

Davide Spallaccini

11 of November 2017

## 1 Introduction

In this work we present Mandroid a machine learning driven Android malware analysis tool able to detect malicious applications that have been previously analysed and registered according to the Drebin dataset format. The motivation behind this work is given by the increasing spread of malware-based cyber attacks: in the first semester of 2017 these type of attacks have seen a growth of 85% according to [3], and 19% of all cyber attacks are malware driven according to [6]. Beyond this nowadays malware is often very complex and usually requires weeks of analysis to be classified and unraveled by software experts, it employs code obfuscation techniques that make the malicious software almost a black box to the eye of the analyst, and finally is often organised in *variants* that allow the malware developer to cope with the updated security defences with minimal effort. Furthermore with the introduction and wide spread of mobile devices and application the security scene became broader.

## 2 Background

We will start with a basic definition: a malware is a malicious software that fulfils the deliberately harmful intent of an attacker, its purpose is usually to damage or trick users or systems, exploit vulnerabilities, install other malware.

In the rest of the report we will mention different machine learning algorithms, here we provide a brief overview of some of them:

**SGD** Stochastic Gradient Descent is a very efficient linear discrimination approach which works on convex function loss, and it has received at-

tention with the advent of large-scale learning. The method consists in learning a linear scoring function $f(x) = w^T x + b$ with model parameters $w$ and intercept $b$. A common choice to find the model parameters is by minimizing the regularized training error $E(w, b)$.

**SVM classifier** A support vector machine constructs a hyper-plane or set of hyper-planes in a high or infinite dimensional space, which can be used for classification, regression or other tasks. Intuitively, a good separation is achieved by the hyper-plane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier.

**Android malwares and machine learning: the Drebin dataset** The process of analysing, classifying or detecting malware requires sometimes deeper understanding about several aspects of malware, and at the same time there's the need to capture malicious aspects and traits having the broadest scope. Machine learning is then a natural choice to support such a process of knowledge extraction from data consisting of existing legal and malicious applications.

The environment that enables us to employ machine learning for these tasks is the plentiful of labelled samples collected through static and dynamic analysis of existing applications. This allows us to employ a very large training dataset to be fed to the machine learning algorithms. Many works in literature have taken this direction with a variety of approaches and results. Some of the key advantages of the employment of machine learning techniques in malware analysis are though the fact that the analysis is completely automated without requirement of human support, and the fact that these methods are in principle able to generalize fro the examples making the analysis resilient to malware variants and to obfuscation techniques.

The Android operating system allows the analyst to retrieve useful information from an Android application starting from the manifest.xml file and the apk file. In this work we will build on the work done by the creators of Drebin [1] a pretty large dataset containing labeled applications and malwares. In particular Drebin addresses the problem of malware family classification and malware detection on a dataset composed of 123.453 benign applications and 5.560 malware. It uses both static and dynamic analysis to extract relevant data from the samples, and features are extracted from the *manifest.xml* file and from the disassembled code.

## 3 Design

This work concentrates on the task of malware detection, even if other types of analysis are still possible like classification. So our goal is finding a function $MD(x)$ that has as a domain $F$ the set of all possible applications files, and as codomain the set $P, N$ where $P$ indicates that the file is a malware, while $N$ that the file is not a malware. Supervised learning allows us in general to infer the parameters of this function based on a dataset of labeled samples.

**Data selection and preprocessing** Data in the Drebin dataset is organized in a large folder containing text files and a dictionary .csv file containing the malware IDs and the respective family of each malware. Each file contains information about a single application in the format `type of info::value` in an unordered way; the various files are named after the result of the SHA256 hashing of the apk file. The features were originally classified in 8 classes: requested hardware components, requested permissions, app components, filtered intents, restricted API calls, used permissions, suspicious API call, network addresses embedded in the code. For the purpose of this work we chose to consider *all* the features given with the dataset since, as we will see later in the next paragraph we made our tests with algorithms that are able to work with very large dataset in a really efficient way allowing us to achieve pretty good results loading almost all the dataset with a running time of 5 to 80 seconds (depending on the evaluation method and algorithm employed). The work can be extended with the use of more classifiers like for example Neural Networks, and indeed our code instantiates the sklearn class *MLPClassifier* but this kind of algorithm requires further optimizations on the dimension of data in order to run in an acceptable time on a single standard machine.

In the preprocessing code we first load the malware names in a dictionary assigning the value *True* to all the malware files. Then all the requested files are loaded and parsed one by one. In our implementation we put the possibility to decide the total number of samples and the percentage of malware among these samples. This allowed us to test with various benign app/malware ratios before evaluating the approach on the entire dataset. During the loading cycle before being added to the array of samples the file is parsed according to the following approach: each line, as it is, is considered a single feature of the file and it is stored in a dictionary structured as {`feature string: True`}; this dictionary object will be the one appended to the array of samples. The power of this approach indeed is in

the preprocess routine that uses the facilities provided by the sklearn class *DictVectorizer* that does all the necessary magic to tranform the array we built in a numpy array instance that can be fed to the training routines. The first problem in fact is to represent the data extracted from the dataset in a numerical format. We built our implementation on the ideas of the Drebin [1] work: the approach consists in characterizing each sample with an array containing all the possible features, i.e. all the possible strings present in the files; each element in the array can be 0 if the correspondent string is not present in the file, or 1 if it is. Since the number of possible features are really large we can claim that all those arrays will be sparse, so we can store them as sparse vectors where we only record the position of the elements with value 1. The resulting vector of samples will be then a sparse matrix of shape `(n_samples, n_features)` which is exactly what the sklearn classes require as input. All this work that we explained here is done automatically by the routine `fit_transform()` of the *DictVectorizer* class.

Finally we want to mention a nice optimization in the process of loading data: in Python the `pickle` package allows to serialize objects in a binary file. This means that once we have preprocessed the file for the first time we can store the result and recover it later when we launch the program with different algorithms. Just consider the timings below to have an idea of the speed-up.

```
n_malware 5000, n_goodware 95000
From pickle file: Time for fetching data:  0.09006810188293
From scratch:     Time for fetching data: 53.90184807777405
```

**Algorithm selection** During the development of the tool we experimented with different algorithms. With some good parameters most of the algorithms do a good job in terms of prediction, even if some are slightly better than others, the real difference is in the computational complexity and on the running time of the different implementations. So we decided to choose those algorithms and implementations that were faster and efficient. This was not only a question of performance, in fact since the data we extracted were pretty raw and with high dimensions we needed to use approaches that allowed us to use as much data as possible, but simultaneously running in an acceptable time. Obviously those algorithms must have in common the characteristic of being supervised learning based and suitable for the analysis of text data.

So the two main methods for classification we used are Support Vector Machines and Stochastic Gradient Descent classifiers. In particular in the case of the SVM we choose a linear kernel, always for performance reasons. For each of these algorithms we specified some parameters according to good heuristics coming from the literature. For the SVM classifier we chose a squared hinge loss function, which is simply the square of the hinge loss, the standard in SVM classifiers, then we chose 1000 as the number of iterations and 0.0001 as stopping tolerance parameter. On the other hand the parameters we chose for the SGD classifiers are: 1000 as max number of iteration, 0.001 as tolerance, and 1e-4 as the alpha parameter; this last parameter is a little bit smaller than the default one but yields a better accuracy of some small percentage points. Finally we chose the modified huber loss function instead of the standard hinge loss, this provided some minor improvements on the accuracy and brings tolerance to outliers as well as probability estimates according to the sklearn documentation.

**Implementation** For the implementation we used the scikit-learn library. The reason for the choice is that it provides a toolbox with solid implementations of a bunch of state-of-the-art models and it comes with a easy to use programming interface and a large, detailed and easy to read documentation together with useful examples and tutorials.

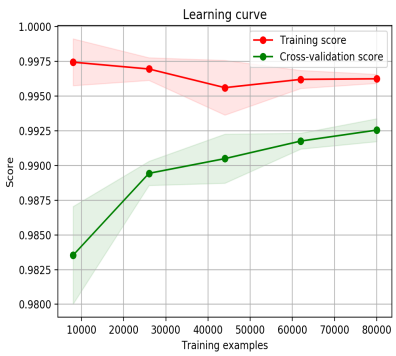
In particular for the SVM classifier with linear kernel we chose the *LinearSVC* class that wraps using C and Cython, the liblinear implementation [4], which is much more efficient than the the libsvm-based SVC counterpart with linear kernel. Some peculiarities of the specific implementations are the use of a sparse internal representation for the data that will incur a memory copy and the use of a random number generator to select features when fitting the model. For the stochastic gradient descent we used instead the class *SGDClassifier* which has an interface similar to the SVC one. The results of the profiling of this second class are very good, and the algorithm appears to be very fast.

# 4 Evaluation

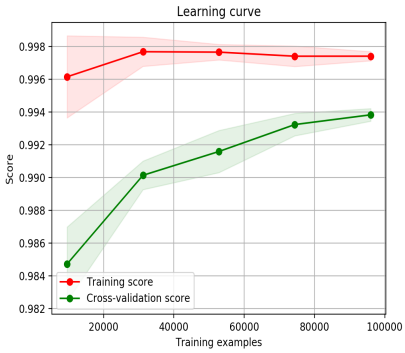
In this section we will show some of the results of our evaluation activities. It's easy to notice that SGD is quite fastes than the SVM even if they yield almost the same final accuracy. Now let us make room for the collected evaluation data. Please notice all the timings from now on in the code are measure in seconds.

```
$ python mandroid/tests.py datasets/drebin/feature_vectors
. -t SVM -p
Fetching data...
malware samples 5000, goodware samples 45000
Time for fetching data: 14.586732149124146
Time for training network: 2.6770317554473877
Confusion matrix results on test split consisting of
20% of total data:
true negative false negative
false positive true positive
[[8943   58]
 [  51  948]]
K-fold cross validation results:
[ 0.9878  0.9878  0.9874  0.9878  0.99
  0.9896  0.9876  0.9902  0.989   0.99 ]
Min/avg/max from K-fold: 0.9874, 0.98872, 0.9902
Time for 10-fold cross validation: 32.21145415306091
```

These are the results from one of the first tests with SVM. The performance with the chosen parameters was good but we notice that the learning curves where slowly increasing withe the number of samples so we decided to use the entire dataset, also considered that the time required for training was acceptable.

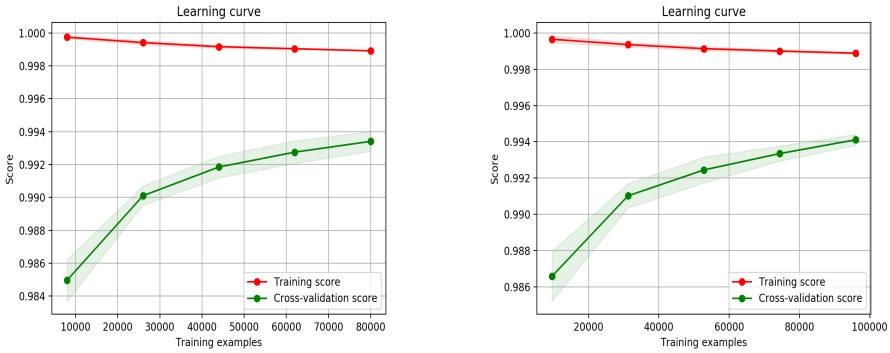



(a) SGD with 100k samples, hinge loss




(b) SGD with 123k samples, modified huber loss

Figure 1: Learning curves for Stochastic Gradient Descent classifier




(a) SVM with 100k samples (b) SVM with 123k samples

Figure 2: Learning curves for SVM classifier

Finally we provide other evaluation and performance profiling results as given as output of the mandroid tool.

```
$ python mandroid/tests.py datasets/drebin/feature_vectors
. -t SGD -p -f -plt
Fetching data from pickle serialization...
n_malware 5560, n_goodware 117440
Time for fetching data: 0.12672114372253418
Time for building learning curve: 22.014636039733887
Confusion matrix results on test split consisting of
20% of total data:
true negative false negative
false positive true positive
[[23350   129]
 [   60  1061]]
K-fold cross validation results:
[ 0.99284553  0.99365854  0.99284553  0.99308943  0.99560976
  0.99235772  0.99479675  0.99333333  0.99341463  0.99373984 ]
Min/avg/max from K-fold:
0.992357723577, 0.993569105691, 0.995609756098
Time for 10-fold cross validation: 7.738193988800049

$ python mandroid/tests.py datasets/drebin/feature_vectors
-t SVM -p -f -plt
Fetching data from pickle serialization...
```

```
n_malware 5560, n_goodware 117440
Time for fetching data: 0.08618402481079102
Time for building learning curve: 132.03317284584045
Confusion matrix results on test split consisting of
20% of total data:
true negative false negative
false positive true positive
[23345    95]
 [   65  1095]]
K-fold cross validation results:
[ 0.99276423  0.99365854  0.99398374  0.99390244  0.99520325
  0.99414634  0.99487805  0.99308943  0.99390244  0.99463415 ]
Min/avg/max from K-fold:
0.992764227642, 0.994016260163, 0.995203252033
Time for 10-fold cross validation: 62.649200201034546
```

# 5 Conclusions

In this work we compared different machine learning approaches with the purpose of detection of Android malware. In order to do this we provide a tool that allows to train a model with the data coming from the Drebin dataset and that is able to predict with an average accuracy of 99.35% if a new instance of software trace is a malware or not. In order to predict new software data about it have to be extracted according to the Drabin dataset format. Actually another student from Sapienza University is developing the `mclass` module that classifies malwares in the respective families. Future work could address the development of a deep neural network in order to further increase accuracy, and the addition of a module that automatically extracts features from an apk file before analyzing it.

## References


[1] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*. The Internet Society, 2014. URL: `http://dblp.uni-trier.de/db/conf/ndss/ndss2014.html#ArpSHGR14`.

[2] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[3] CLUSIT. Report 2017, 2017. URL: `https://clusit.it/rapporto-clusit/`.

[4] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, June 2008. URL: `http://dl.acm.org/citation.cfm?id=1390681.1442794`.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[6] SERT. Quarterly threat report q2, 2016. URL: `https://archive.org/details/perma_cc_95JP-RECR`.