Android Malware Detection

Pietro Borrello - 1647357
Sapienza University of Rome

November 19, 2017

Abstract

Detecting malicious programs is becoming increasingly more difficult due to the variety of behaviors and forms malware nowadays presents. Hash-based and heuristic driven approaches used by current antivirus seem not to address efficiently nor effectively the problem. To address the problem Machine Learning based approaches are increasingly more used with notable success.

In this paper we want to analyze such approaches to assess the accuracy of this ML based methods, in particular in a supervised learning setting, providing the reader a systematization of knowledge on the state of art in the field.

1 Introduction

To be able to detect effectively malware programs through Supervized Machine Learning approaches, we needed some really big labeled dataset to analyze. We started from the DREBIN dataset used in the DREBIN paper [1], a 123K labeled Android application dataset, containing 5,560 malwares, resulting in a rate of 95.4% benign programs.

Each program is described through some features, statically retrieved both from the Manifest file and from the source code of the corresponding apk. The features included:

- Hardware Components
- Requested Permissions
- App Components
- Filtered Intents

- Restricted API Calls
- Used Permissions
- Suspicious API Calls
- Network Addresses

Each program has an ethereogeneus set of features, and the set of all possible features described in the dataset is huge: more than 545,000. Based on this fact, we approached malware classification as document classification, in which, given a program and his set of words/features our aim is to decide if is $Malware$ or $NotMalware$

2 Approaches

We used different approaches, trying to exploit the fact that each program has in average only 50 features out of 545,000 possible. We evaluated three different models, each with different parameters: Naive Bayes, linear Support Vector Machine and non-linear Support Vector Machine.

2.1 Naive Bayes

Our first choice was the Naive-Bayes classifier, a probabilistic model known to be suitable for text classification [2]. The Naive-Bayes model classifies an instance with the class which maximizes the probability $P(c_i|x, D)$ that the given sample x exhibits such features given its membership to that class c_i in the dataset D. Therefore the class assigned to the new instance x (represented by a set of features $< x_1, \ldots, x_n >$) will be:

$$argmax_{c_i \in C}\{P(c_i|x, D)\}$$

that is equal to

$$argmax_{c_i \in C}\{P(x|c_i, D)P(c_i|D)\}$$

Assuming the conditionally independence of the features given the class to which the sample is member of, the probability is also equal to:

$$argmax_{c_i \in C}\{P(c_i|D)\Pi_i P(x_i|c_i, D)\}$$

that is what Naive-Bayes computes.

Seeing each program as a text representing the features it has, preprocessing the dataset to be compatible with the model is straightforward. We constructed a huge data file, where for each line, each program was reported with all the features it had. To manage the features more efficiently we encoded their strings defining a map, where for every feature was defined a unique incremental number. So a program file in the form of:

```
class (taken from the attached csv file)
feature_class::feature_value_1
feature_class::feature_value_2
...
feature_class::feature_value_i
```

Would become a line in the dataset as:

`<class> id_feature_1, id_feature_2, ..., id_feature_i`

The preprocessing code is attached in the file `preprocessing.py`

Assuming each feature conditionally independent from the others, given the class, this model seems a good choice to us.

We implemented by scratch a Naive Bayes classifier based on multinomial distribution, and made it available to the public [4].

The Naive Bayes classifier at a first glance seems to perform quite well with the task. We evaluated it with K-Fold algorithm on the whole dataset. With $K = 10$ it reaches in a minute in a modern computer an accuracy of 96.4%. But what we have to take into consideration, is the distribution of the dataset, in fact there is a 95.5% probability that given a program, it is not a malware. So on a dataset which maintains the same distribution, a Dumb classifier, always replying that a program is not a malware, would reach such an accuracy of 95.5%, though having a False Negative Rate of 100%, and still a True Negative Rate of 100%. For a Malware Classifier we need to minimize the False Negative Rate, which means to minimize threats not blocked.

On Naive-Bayes we measured a good True Negative Rate of 99.8%, but a still too high False Negative Rate of 78.7% so we conclude Naive-Bayes is not the best model to apply, at least with the approach used.

2.2 Linear Classification

To improve the classification, a more complex model may be needed. Assuming the data is linearly separable (which is not foolish for large dimen-

sional datasets), the first successive choice is represented by Linear Classification, which can give better performances in case of huge sparse feature-wise dataset, as the DREBIN one. A Linear Classifier, predicts the class of a sample in a multidimensional space by a linear combination of the features. In particular it divides the space with an hyperplane into the two classes, to be able to predict the class of a new sample, based on the position in respect to that plane. To find such an hyperplane it solves an optimization problem of the form:

$$argmin_w \frac{1}{2} w^T w + C \sum_i L(y_i, w^T x_i)$$

Where w is the vector that defines the hyperplane, C is the cost (penalty for misclassified instances) and L is the loss function that measures the discrepancy between the predicted value and the actual y_i, and is specific for the model chosen.

For the task we chose to use *liblinear*, an open source library designed for large-scale linear classification, inspired by *libsvm*, which provides an easy to use command line interface [3].

To make our dataset compatible with *liblinear* we needed some additional preprocessing to map text file into points in a geometrical space. We chose to represent our data accordingly to the DREBIN paper [1], as sparse vector of binary attributes, with each value of each sample on dimension i representing if the sample had the feature i, with i growing up to 545,000: an easy to do representation, given the work done to map features with unique numeric ID (going to represent dimension in the space), done for the Naive Bayes classifier.

We chose as a base truth the performances of our previous classifier, and tried to improve them. The first choice was running the trainer for the model the dummy way, without any additional parameter, using the logistic regression model provided by default.

Doing a K-Fold cross validation with $K = 10$ executing:

```
>>> train -v 10 android_malwares.libsvm
```

we had from the beginning a surprisingly good accuracy of 99.3% with a True Positive Rate of 99.6% and a better False Negative Rate of 9.4%, convincing us this will be a better choice for the model. The overall running time is still around a minute.

As a first step to improve the detection, we tried to change the model from the default logistic regression to the loss Support Vector Machine classification model. A SVM based model maps the samples into points of a multidimensional plane, trying to find a maximum margin hyperplane, to divide samples in the space in respect to their category, with the greater confidence possible. Then we used the built-in option of cost optimization which tries to change the cost factor of a misclassified instance to maximize the accuracy, automatically searching in the space $C \in \{2^{-5}, \ldots, 2^{15}\}$. A lower cost will decrease the accuracy, but a too high cost will determine overfitting, so a proper value must be chosen. Running:

```
>>> train -C -s 2 android_malwares.libsvm
```

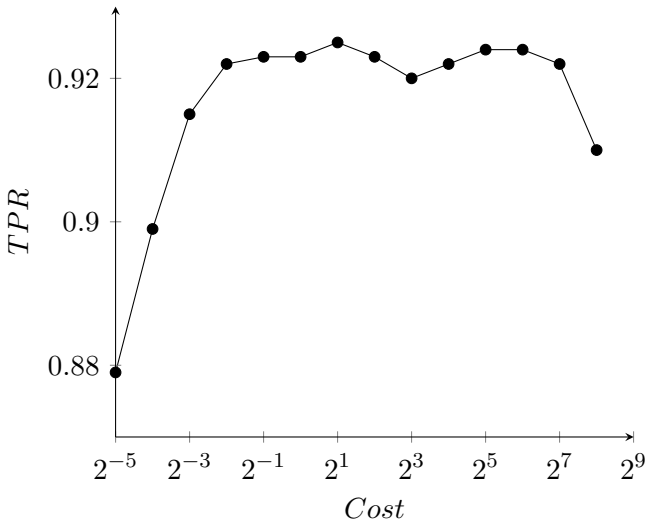
we are suggested to use $c = 2$ with a resulting accuracy of 99.44%. We then investigated the False Negative Rate, which was what we really wanted to minimize, with a simple bash script, searching for cost values around the given c.

```
#!/bin/bash

for i in 0.03125 0.0625 0.125 0.25 \
             0.5 1 2 4 8 16 32 64 128 256;
do
 echo -n '-c' $i ' ' >> costs.txt
 train -c $i -q -s 2 train_android_malwares.libsvm \
     android_malware.model
 predict test_android_malwares.libsvm \
     android_malwares.model out.txt >> costs.txt
 ./stats.py >> costs.txt
done
```

Where `stats.py` is a simple python script to compute TNR and TPR (considering $TPR = 1 - FNR$) and `train_android_malwares` and `test_android_malwares` where random subset of the dataset in the proportion of 90% to 10% to maintain the probability distributions of the two classes.

The resulting True Positive values are represented:



And for completeness of result we provide the resulting accuracies:

Cost	**Accuracy**	**TPR**
0.03125	99.2798%	0.879
0.0625	99.339%	0.899
0.125	99.3981%	0.915
0.25	99.4369%	0.922
0.5	99.4389%	0.923
1	99.4287%	0.923
2	99.44%	0.925
4	99.4%	0.922
8	99.3859%	0.920
16	99.3879%	0.923
32	99.4104%	0.924
64	99.4104%	0.924
128	99.392%	0.922
256	99.355%	0.910

After the TPR estimation, we where confident that the value that maximized accuracy, would also minimize the False Negative Rate. Therefore running:

```
>>> train -c 2 -s 2 android_malwares.libsvm
```

we obtained a classifier with a 99.44% accuracy and a good 7.5% False Negative Rate, with a training phase that only lasted 6 seconds.

2.3 Non-Linear Support Vector Machine

Given the good results we obtained with the linear SVM based classifier, we tried to improve the classification, using a generalization of that model. Dropping the assumption of linearly separable data, Support Vector Machine can be used to classify samples even if they are non linearly separable, applying a suitable kernel function to the data, to implicitly map the dataset to a higher dimension linearly separable space.

We used *libsvm*, an integrated software for support vector classification, with a similar interface as *liblinear* [5]. Since we preprocessed our data to stick to *liblinear* file format, no additional preprocessing was needed, given the support for the same filetype.

We choose RBF Kernel (Gaussian Kernel) to map data from the non linearly separable space. A Gaussian Kernel is in the form:

$K(x_i, x_j) = exp(-\gamma \|x_i - x_j\|^2)$ with $\gamma > 0$

Our choice was driven by the fact that a the other kernels types (linear and sigmoid) can be reduced to RBF kernel for certain parameters (C, γ) [6], so analyzing parameters of such a kernel, is enough to find the optimal solution.

The first step was training the classifier with the default parameters, to have a ground truth for the optimization. Running a K-Fold 5 on the whole dataset with:

```
>>> svm-train -v 5 android_malwares.libsvm
```

resulted to a disappointing classifier with an accuracy of 95.5%, having a False Negative Rate of 100%, and a True Negative Rate of 100%: a Dumb classifier replying always "Not Malware". The worst thing was that it also increased the computation time from a minute to 2 hours.

We started heavily optimizing it, with the belief that it could do better than a linear SVM. Using the script `easy.py` included in the bundle, after 5 hours of computation the best parameters it can find were the cost $c = 128.0$ and the gamma value of the kernel $\gamma = 0.001953125$, a lot bigger than the default one equals to $1/num\_features$, resulting in a less constrained model. Then running:

```
>>> svm-train -h 0 -c 128.0, -g 0.001953125 /
      train_android_malwares.libsvm android_malwares.model
```

results in 20 minutes to a really better classifier of accuracy 99.53% a TNR of 99.7% and a FNR 7.3%.

3 Evaluation

To be able to effectively protect from malwares, a detector should minimize the rate of undiscovered threats, resulting in a False Negative Rate as little as possible. The best classifier we came up with was the non-linear SVM, with a FNR of 7.3%, which is really better with respect to the Naive Bayes classifier (FNR of 78.7%), and slightly better than the rate of the logistic regression (9.4%) and the linear SVM (7.5%).

Confronting linear and non-linear SVM, the slowdown for the training and the prediction phases, with respect to the small improvement in accuracy, becomes clear to be unacceptable. In fact in front of an improvement of $1.03x$ in the FNR for the non-linear in respect to linear, we have a $264x$ slowdown for the non-linear training phase (22 min *vs* 5 sec on average) and a $180x$ slowdown for the non-linear prediction phase (1 min 30 sec *vs* 0.5 sec on average).

The following table resumes the previous remarks, with time expressed based on K-Fold time with $K = 5$:

Model	**K-Fold Time**	**Accuracy**	**FNR**
Naive-Bayes	30sec	96.4%	78.7%
Logistic Regression	45sec	99.3%	9.4%
Linear SVM	30sec	99.44%	7.5%
Non-Linear SVM	1hour 30min	99.53%	7.3%

Given the higher time consumption for the non-linear SVM and the slight improvement it has, we concluded the best method to address such a problem should be a linear SVM with proper optimization.

Note In the examples the accuracies where estimated by a K-Fold on the whole dataset, while the training phases times where measured without K-fold to produce the model. So attention as to be given while reading about time consumption.

Note 2 Some attempts were made to also filter the attributes based on their classes for all four the models, to reduce feature complexity and improve detection, but all the improvements where negligible or not enough to justify the increase in the time for preprocessing and filtering the features.

4 Conclusion

Given the increasing number of different form of malwares, we believe the current methods to identify such threats doesn't address the problem effectively.

As an alternative, we proposed four different methods based on machine learning models to detect malicious program, and we evaluated them to show their efficacy and efficiency. We concluded that a Linear Support Vector Machine is believed to be the best choice to address such a problem, due to low time consumption and high accuracy.

References

[1] Daniel Arp, Michael Spreitzenbarth, Malte Huebner, Hugo Gascon, and Konrad Rieck *Drebin: Efficient and Explainable Detection of Android Malware in Your Pocket*, NDSS, Feb 2014.

[2] Haiyi Zhang, Di Li *Naive Bayes Text Classifier* IEEE, 2007.

[3] Rong-En Fan, Kai-Wei Chang *LIBLINEAR: A Library for Large Linear Classification* CSIE, 2017.

[4] Pietro Borrello *https://github.com/pietroborrello/SMS-BayesClassifier*, 2017.

[5] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin *A Practical Guide to Support Vector Classification* National Taiwan University, Taipei , 2003.

[6] Keerthi SS, Lin CJ. *Asymptotic behaviors of support vector machines with Gaussian kernel* Neural Comput, 2003.