Machine Learning 2017/18 Homework 1

Tufa Alexandru Daniel 1628927

19 November 2017

1 Introduction

Malware analysis is the study or process of determining the functionality, origin and potential impact of a given malware sample. Malware, or malicious software, is any computer software intended to harm the host operating system or to steal sensitive data from users. Usually we can have two types of malware analysis:

- **Static Malware Analysis** where we inspect the different available resources without executing the actual harmful software
- **Dynamic Malware Analysis** that is based on observing the behaviour of the malware while it is actually running

More often than not we want an automated system that can analyze a certain software and decide if it can be trusted or not. A way of doing this is to compare it with many other candidates that were revealed to be malicious by examining some particular features that characterize them. A natural choice to support such a process of knowledge extration is offered by **Machine Learning**.

2 Description of the problem

The goal of this paper is to analyze different machine learning algorithms with the purpose of classifying an Android application as malware or non-malware and compare their offered results. We will use as training set the same that was used in the `Drebin` [1] project. It contains 123,453 bening applications and 5560 malware, each of which is represented by a file named by its own identifying hash. The file is divided internally into rows where every row is structured as "`feature`::`value`". These features are extracted from the `manifest.xml` file and from the disassembled code and are divided in 10 sets:

- **url**: network addresses
- **feature**: a hardware or software feature that is used by the application
- **provider**: classes that provide simplified methods of adding or retrieving data from a particular content provider

- **call**: suspicious API calls
- **activity**: component of the application that interacts with the user
- **intent**: an abstract description of an operation to be performed
- **api_call**: restricted API calls
- **real_permission**: used permission
- **permission**: requested permission
- **service_receiver**: a general-purpose entry point for keeping an app running in the background for all kinds of reasons

To solve this problem we have chosen Weka [2] as a tool for our machine learning algorithms. We will see how to transform the Drebin dataset to an .arff[1] file that is compatible with the input that our program can accept.

3 Transformation of the dataset

We will see and compare two ways of transforming the data. Initially our dataset will be divided into 10 attributes, 11 if we consider the Class, and every attribute will have as possible value every concatenation of its feature values encountered during the analysis of the applications. This method will enphasize more on the `order` of certain feature values. The second methodology will be focused more on the `presence` of certain feature values and our dataset will be characterized by binary attributes.

3.1 Concatenation method

Our desired .arff file has the following form:

```
@relation malware

@attribute url {http://a.admob.com/f0?,..}
@attribute feature {android.hardware.touchscreen,..}
@attribute provider{com.google.provider.NotePad,..}
@attribute call {getDeviceId,..}
@attribute activity {.MainActivity,..}
@attribute api\_call {android/media/MediaPlayer;->start,..}
@attribute real\_permission {android.permission.INTERNET,..}
@attribute permission {android.permission.READ_PHONE_STATE,..}
@attribute service\_receiver {MtAlarmBroadcast,}
@attribute Class {malware,not-malware}

@data
```

[1]Attribute Relationship File Format

After `@data` every row will represent one application with values belonging to the attributes described before in the same order as they are listed, separated by a comma.
This transformation will be done using the python code provided. We will describe the initial state of this file and explain how and why it was decided to be changed. It uses mainly three functions (four if we want to analyze only part of the dataset): `prepareData`, `writeAttributes`, `writeData`.

- **prepareData** creates the data structure that will be used for filling the rows under the `@data` field in the .arff file. It has the following structure: data = dictionary{ nameOfTheApplication : array(0 : dictionary { feature : singleValuesConnectedUsing& } , 1 : malware/not-malware) }
To do this we create the initial structure by defining the initial dictionary and setting the first index of the internal array as a dictionary where every key is the name of the feature. Then for every file we use a temporary dictionary where the key is still the feature and the value is an array where we'll append every feature value every time we encounter it. Once we have finished scanning through the file we use this array of feature values to create the string that will be used for the main internal dictionary. This string is made by concatenating every feature value using the '&' character. After we have finished scanning thourgh all the files, given that we have saved the files in the main external dictionary using as the key the name of the file, we can scan through the `sha256_family.csv` file to associate to each file if it's a malware or not.

- **writeAttributes** is used for writing the attributes in the .arff file. Initially they were written in the same way as they are scanned in the `prepareData` function (values concatenated using '&') and this generated, for all the dataset and together with the written data, a 250MB file. A lot of space was wasted by doing this verbose encoding, so instead a numerical one was chosen were for every distinct value of a certain feature a certain numerical element was linked while for features with more than one value the concatenation (still using '&') was done between numbers. This reduced the space used to 70MB, but it wasn't enough. By doing some further analysis on how the `nominal` values work in Weka we figured out that it was useless to represent complex features using concatenation, instead we assigned to this concatenation an unique numerical identifier exactly like the single valued features because they represent the same concept, a certain value for a feature that will be used by the data. This technique reduced our data to 7.5MB. An incredible improvement in comparison with considering all 130,000 files that together weigh 500MB! During the writing of the attributes we also build a data structure that will be used as an encoder for transforming the values of the feature from strings to numbers.

- **writeData** takes as input the data structure generate by `prepareData` and the attributes encoder returned by `writeAttributes` and writes the data on the .arff file using the given encoding. We have to also take into consideration that in some cases not all features are used by the application, so for missing values a '?' is to be used.

- **filterFiles** is an optional function that will be used for selecting a random subset of applications from the entire dataset

3.2 Binary attributes

In this representation we will add every distinct feature value, not the concatenation of multiple values, as a binary value to indicate if it's present or not in an application. In this way we will concentrate more on the presence of the feature values and not on their order. Every single instance will instead be represented using a sparse vector so that the attributes that are not present are assigned automatically 0. To simplify notation given that the sparse vector needs to assign values to the index of the attributes we will use the index to name the various attributes. As we can see in order to do this we have divided our code into 2 functions:

- **writeAttributes** is used for scanning through the files and write the attributes as binary variables. We maintain as data structure a dictionary of attributes where we use as key the name of the feature (e.g. `android.hardware.touchscreen`) and as value the encoding of it that is a counter that increments for every added attribute. If we haven't already written the attribute on the .arff file we write it as "@attribute `index` {0,1 }"
- **writeData** takes as input an encoder of the attributes and for every application it writes in the .arff file a single row that is to be considered a sparse vector. Every row in the application file, that is a certain feature value, represents an entry in the sparse vector and because it is present we add the index of this value, that is given by the encoder, and the numerical value `1` to indicate its presence. Automatically all values that are not present in this sparse vector are considered 0 and that is our intention. At the end of the row we also add the class of the app in consideration. The only attribute that is not represented in the .arff file by its index is the `Class` one and to obtain its index is enough to consider the number of distinct attributes because it will always be the last attribute to be considered.

4 Analysis of the ML algorithms

4.1 Naive Bayes

In order to do the classification this algorithm calculates for the malware and not malware class the probability of the given class and multiplies it by the product of the probabilities of a certain attribute given the class. In Weka this

estimations are done using a Bernoulli distribution and before the classifications it does a training on the given dataset. The probability of a class, for example the probability of the malware class, is calculated by counting all the instances that are malwares and divide it by the total number of instances. To calculate the probability of each attribute given a class, we still consider malware in our example, we could count the number of instances that are malwares and contain this attribute and divide it by the total number of instances. In some cases this would given us a zero probability[2]. To remedy this behaviour we sum the numerator numerator with 1 and the denominator with 2; these are often called parameters for Laplace smoothing. Becase we have seen two different ways of interpreting the original data, every algorithm will be run on two different models. In this case the NB algorithm will initially test the probabilities for the attributes that represent the feature values as a single attribute emphasising the order in which they appear. Two applications that, for example, do the same API calls, but in different order will be considered different. Secondly we will see things from a different perspective and care more about the presence or absence of the values of the features that will be distinct, so two applications that do the same API calls, but in different order are considered to belong to the same class. The second approach, in most of the cases, will require more time given that we have to build the model with a lot more attributes.
By analyzing the dataset we can see the original distribution: 5555 instances are malware while 123443 are not malware which gives us a distribution of 5% of malware instances and 95% of not malware. We will mantain this distribution, but consider cases with fewer instances in the dataset while we'll also see what happens if we use a distribution with 70% not malware and 30% malware. To calculate the accuracy we will use cross validation with 10 folds. For 10 times we will use a different training and classify set and we'll do the average of the obtained accuracies.

4.1.1 Original distribution with 4000 instances

If we mantain the original distribution and select randomnly from the dataset 4000 instances we obtain the following results:

- Concatenation: Correctly classified instances: 86.15%

  Confusion matrix:

a	b	classified as
156	17	a - malware
537	3290	b - not malware

- Binary: Correctly classified instances: 94.575%

  Confusion matrix:

[2]When there aren't such instances with that particular attribute and class

a	b	classified as
15	158	a - malware
59	3768	b - not malware

One may think that the binary technique gives us a higher accuracy but by analyzing the confusion matrix we can see how while it's good as classifying not-malware instances it's not so good as classifying the malware ones by guessing right only 15 out of 158. The first one has the opposed results with a right classification of 156 out of 173. We can see that if we use our alternate distribution the two techniques start to converge, even if the concatenation method still prevails.

4.1.2 Alternative distribution with 4000 instances

- Concatenation: Correctly classified instances: 90.3476%

  Confusion matrix:

a	b	classified as
1130	69	a - malware
317	2483	b - not malware

- Binary: Correctly classified instances: 87.5969%

  Confusion matrix:

a	b	classified as
921	278	a - malware
218	2582	b - not malware

4.2 Support Vector Machines

To see how the LIBSVM [3] library calculates the classification of the dataset we need to introduce what is the reasoning behind Support Vector Machines, or SVMs. We start from a general space and try to separate all our instances by drawing a gutter that is as large as possible so that we can make our classification as accurate as possible. We will do our calculations with respect to the line that divides the gutter in half and call "w" the vector that is perpendicular to this line. If we take an unknown vector in this space and dot cross it with w, by doing this we are projecting the vector on w thus making it also perpendicular to our reference line. We can establish a **decision rule** as follows: If $w \cdot u + b \geq 0$ then we classify this instances as positive. To aid us in discovering w and b we will add some properties that will prove useful to us. For positive sample we can say that that $w \cdot x_+ + b \geq 1$ and for negative samples $w \cdot x_- + b \leq -1$. For mathematical convenience we introduce a variable y_i that is +1 for positive samples and -1 for negative samples. So in the end we can see that for every x_i that belongs to the gutter we have $y_i(w \cdot x_i + b) - 1 \geq 0$. This is our **constraint rule**, now we need to find the main optimization one. Our objective is to have the gutter as large as possible so let's consider two general points on each side, for example a positive and a negative sample. If we do the difference of this vectors and project it on a unit vector that is perpendicular to the gutter we obtain the desired width. We have already considered this perpendicular vector as w and to obtain it's unit we just divide it by its norm. So because the width = (x_+ - x_-)$\cdot\frac{w}{\|w\|}$ and if we substitute for x_+ and x_- the previous equations we obtain that the width is $\frac{2}{\|w\|}$. The maximization of this value is equivalent to the minimzation of $\|w\|$ that is equivalent to the minimization of $\frac{1}{2}\|w\|^2$. If we put together this minimization and the constraint find before by using Lagrange multipliers we can find that w = $\sum_i \alpha_i x_i$. So w, the width of the gutter, depends exclusively on the summation of some instances and we call them **support vectors**. If we go back to the decision rule and substitute w we obtain $\sum_i \alpha_i x_i \cdot u + b \geq 0$. The decision rule depends only on the product operation and this is helpful because I can replace it with another product of the vectors represented in a different space. This mathematical contraption is called a **kernel trick** because I change a product of vectors $x_i x_j$ with a product of the same vectors in a different space $\phi(x_i)\phi(x_j)$ with ϕ representing the transformation function from a space to another space. The beauty of this method is that we don't care about what ϕ precisely is because we use instead a function that directly gives us $\phi(x_i)\phi(x_j)$. This function is called a **kernel**. There are different kernels what one can use and the one that we'll consider is the linear kernel where k(x,x') = $x^T x'$. If after using the kernel trick the space is still not linearly separable we replace the hard constraints with some soft margin constraint by introducing slack variables $\xi \geq 0$ where

- ξ=0 if the point is on or inside the correct margin boundary
- $0 < \xi \leq 1$ if the point lies inside the margin on the correct side of the decision boundary
- ξ >1 if the point lies on the wrong side of the decision boundary

The decision problem becomes

$$min\frac{1}{2}\|w\|^2 + C\sum_i \xi_i \quad s.t. \quad \xi_i \geq 0, y_i(x_i{}^T w + w_0) \geq 1 - \xi_i$$

The parameter C is a regularization parameter that control the number of errors we are willing to tolerate on the training set.
If we pass our dataset to the LIBSVM library we get the following results:

4.2.1 Original distribution with 4000 instances

- Concatenation: Correctly classified instances: 97.4244%

  Confusion matrix:

a	b	classified as
99	100	a - malware
3	3797	b - not malware

- Binary: Correctly classified instances: 98.1245%

  Confusion matrix:

a	b	classified as
156	43	a - malware
32	3768	b - not malware

4.2.2 Alternative distribution with 4000 instaces

- Concatenation: Correctly classified instances: 93.8765%

  Confusion matrix:

a	b	classified as
1016	183	a - malware
62	2738	b - not malware

- Binary: Correctly classified instances: 96.2741%

  Confusion matrix:

a	b	classified as
1137	62	a - malware
87	2713	b - not malware

4.3 Logistic Regression

To truly understand logistic regression we must talk about linear regression. The goal of linear regression is to estimate the value y' of a continuous function at x based on a dataset D composed of N observations together with the corresponding target value y. To correctly adjust the prediction of this function we will use some weights w so at the end we'll have y' = f(x,w). The simplest case we can have is a linear combination of the inputs x, in our case a numerical representation of the attributes, with the weights. To adjust the weights correctly we can compute the mean squared error[3] defined as

$$\frac{1}{m}\sum_{i}(y-y^{'})^2$$

If we minimize this error by assigning its gradient to 0 we obtain that w is equal to $(X^TX)^{-1}X^Ty$ where $(X^TX)^{-1}X^T$ is the pseudo-inverse.
We can generalize this approach to a binary classification. Because the target value y $\in \{0,1\}$ we replace the Gaussian distribution used with the linear regression with a Bernoulli distribution so we obtain p(y|x,w) = Ber(y| μ(x)). To ensure that the output values of the linear combination are constrained between 0 and 1 we pass them to the sigmoid[4] function so at the end we'll get p(y|x,w) =

[3]Deducted by using the maximum likelihood principle.
[4]sigm(x)=$\frac{1}{1+e^{-x}}$

Ber(y|sigm($w^T x$)) and is called logistic regression due to its similarity to linear regression even if it's a form of classification and not regression. After we have done this we can introduce a decision rule of the form y'(x)=1 if p(y=1|x)>0.5 so that we can do the proper classification.
These are the results obtained from the logistic regression:

4.3.1 Original distribution with 4000 instances

- Concatenation: Correctly classified instances: 96.9742%

  Confusion matrix:

a	b	classified as
85	115	a - malware
6	3793	b - not malware

- Binary: Correctly classified instances: 98.0745%

  Confusion matrix:

a	b	classified as
151	49	a - malware
28	3771	b - not malware

4.3.2 Alternative distribution with 4000 instaces

- Concatenation: Correctly classified instances: 93.2233%

  Confusion matrix:

a	b	classified as
1022	177	a - malware
94	2706	b - not malware

- Binary: Correctly classified instances: 96.2991%

  Confusion matrix:

a	b	classified as
1128	71	a - malware
77	2723	b - not malware

4.4 Comparison between the results

As we can see the Naive Bayes algorithm classifies better when using as representation method the concatenation of the feature values. With the original distribution the binary representation of the attributes fails badly, classifying only 15 malware out of 173. This however changes when we change the distribution and use fewer instances that are not malware even if the concatenation method still offers us better results. The situation changes when we use as our classification algorithm SVMs where, with the representation of the attributes using concatenation, the classifier classifies correctly only half in the original distribution. By changing the distribution it becomes better but the binary representation in this case gives us better results in both the distributions. This results are similar when using Logistic Regression with the concatenation representation performing worse when given the original distribution. When using the original distribution Naive Bayes and libsvm perform equally but under different representations,Naive Bayes by using a concatenation representation while libsvm by using a binary one. When we change our distribution we see that libsvm takes the crown with 1137 correct classification out of 1200 by using a binary representation of the dataset.

5 Extra analysis

Because the binary way of representing the data gives us a very large amount of attributes, the construction of the model may take an amount of time that is not feasible. We'll try out further algorithms considering only the concatenation method of representing the dataset.

5.1 Decision trees

A way of approximating our target function is by learning a function represented by decision trees. These trees classify instances by sorting them down the tree from the root to some leaf node, which provides the classification of the instance. Each node in the tree specifies a test of some attribute of the instance and each branch descending from that node corresponds to one of the possible values for the attribute. For this reason we can't use our second method for representing the data because we would have a tree with truly too many nodes. The most used algorithms are ID3, ASSISTANT and C4.5 and we'll analyze the later one because it's the one used by Weka. This algorithm does the building of decision trees from a training set by using the concept of information entropy, in the same way as the ID3 algorithm. At each node of the tree this algorithm chooses the attribute of the data that most effectively splits its set of samples into subsets enriched in one class or the other. The splitting criterion is the normalized information gain (difference in entropy). The attribute with the highest normalized information gain is chosen to make the decision. The C4.5 algorithm then recurs on the smaller sublists.
Base cases:

- if all the samples in the list belong to the same class then we create a leaf node for the decision tree saying to choose that class
- if none of the features provide any information gain then we create a decision node higher up the tree using the expected value of the class
- if we encounter an instance with a previously unseen class we create a decision node higher up the tree using the expected value

To build our decision tree we check for the base cases, afterwards, for each attribute, we find the normalized information gain ratio from splitting on that attribute. Once we have found the attribute with the highest information gain we create a decision node that splits on that best attribute. On the sublists obtained by splitting on this attribute we recur and add those nodes as children of the decision node that splits on the best attribute.
We will do our classification on 4000 instances that maintain the original distribution and also on the same number of instances by considering a distribution of 70% malware, 30% not-malware. We'll also see what happens when we use a pruned versus an unpruned tree and surprisingly observe that the unpruned version will give us better results.

5.1.1 Original distribution with 4000 instances

- Unpruned: Correctly classified instances: 95.95%
  Number of leaves: 33550
  Size of the tree: 33587
  Confusion matrix:

a	b	classified as
39	161	a - malware
1	3799	b - not malware

- Pruned: Correctly classified instances: 95%
  Number of leaves: 1
  Size of the tree: 1
  Confusion matrix:

a	b	classified as
0	200	a - malware
0	3800	b - not malware

5.1.2 Alternative distribution distribution with 4000 instances

- Unpruned: Correctly classified instances: 91.925%
  Number of leaves: 105241
  Size of the tree: 105334
  Confusion matrix:

a	b	classified as
923	277	a - malware
46	2754	b - not malware

- Pruned: Correctly classified instances: 86.55%
  Number of leaves: 1475
  Size of the tree: 1478
  Confusion matrix:

a	b	classified as
701	499	a - malware
39	2761	b - not malware

As we can see the original distribution, even if the accuracy has a pretty decent value, doesn't perform very well in classifiyng malwares both in the pruned version with 0 out of 200 and also in the unpruned version with only 39 classified correctly as malwares. The most interesting choice that we can observe has been made by the pruned version when using the original distribution. Because this distribution has very few instances that represent malwares, in the pruned version the algorithm decides to choose only 1 leaf and assing every instance as not-malware. In this way the accuracy will be high, but we have to pay the price of not classifying correctly any malware instance. If it's not very wise to use this algorithm with this distribution we see that the situation changes when we change the distribution. The performance of the algorithm is much better with the classification of the malware instances by classifying correctly 923 out of 1200 in the unpruned version and 701 out of 499 in the pruned version. We can see that because of the high number of possible values that a single attribute can assume we need a tree with more nodes to best do our generalization in order to better classify the instances and a shorter tree is worse at doing this.

5.2 AdaBoost

Instead of using one strong learning method we could try using many weak or base learners in order to build simple rules, then combine this rules into a single prediction rule that will be more accurate than each single rule. Given our instances we initialize all the weights as 1 divided by the number of instances. We will train a weak learner $\mathrm{y_m(x)}$ by minimizing the weighted error function:

$$J_{\mathrm{m}} = \sum_{n=1}^{N} w_{\mathrm{n}}{}^{(\mathrm{m})} I(y_{\mathrm{m}}(x_{\mathrm{n}}) \neq t_{\mathrm{n}}) \quad with \quad I(e) = \begin{cases} 1, & \text{if e is true.} \\ 0, & \text{otherwise.} \end{cases}$$

We evaluate the error of each weight $\epsilon_{\mathrm{m}} = \frac{\sum_{n=1}^{N} w_{\mathrm{n}}{}^{(\mathrm{m})} I(y_{\mathrm{m}}(x_{\mathrm{n}} \neq t_{\mathrm{n}}))}{\sum_{n=1}^{N} w_{\mathrm{n}}{}^{(\mathrm{m})}}$ and because we want a value that ranges between $(-\infty, +\infty)$ we use the function $\alpha_{\mathrm{m}} = ln(\frac{1-\epsilon_{\mathrm{m}}}{\epsilon_{\mathrm{m}}})$. To update the data weighting coefficients we compute $w_{\mathrm{n}}{}^{(\mathrm{m})} e^{\alpha_{\mathrm{m}} I(y_{\mathrm{m}}(x_{\mathrm{m}}) \neq t_{\mathrm{n}})}$ so that if our predictions are all right we don't change the value of $\mathrm{w_n}^{(\mathrm{m}+1)}$ else we change it accordingly to how many wrong predictions we have made. Finally the output of the final classifier will simply be $Y_{\mathrm{M}}(x) = sign(\sum_{m=1}^{M} \alpha_{\mathrm{m}} y_{\mathrm{m}}(x))$. Our first base learner is the `Decision Stump` model that is a Machine Learning model consisting of a one-level decision tree. We will see that it doesn't perform very well in our case, but changing to a J48 weak learner will improve our results.

5.2.1 Original distribution with 4000 instances

- Decision Stump: Correctly classified instances: 94.85%

  Confusion matrix:

a	b	classified as
6	194	a - malware
12	3788	b - not malware

- J48 Unpruned: Correctly classified instances: 85.1%
  Number of leaves: 13386
  Size of the tree: 13395
  Confusion matrix:

a	b	classified as
132	68	a - malware
528	3272	b - not malware

- J48 Pruned: Correctly classified instances: 82.7%
  Number of leaves: 1
  Size of the tree: 1
  Confusion matrix:

a	b	classified as
154	46	a - malware
646	3154	b - not malware

5.2.2 Alternative distribution with 4000 instaces

- Decision Stump: Correctly classified instances: 82.525%

  Confusion matrix:

a	b	classified as
805	395	a - malware
304	2496	b - not malware

- J48 Unpruned: Correctly classified instances: 88.25%
  Number of leaves: 11512
  Size of the tree: 11520
  Confusion matrix:

a	b	classified as
1075	125	a - malware
345	2455	b - not malware

- J48 Pruned: Correctly classified instances: 79.225%
  Number of leaves: 5890
  Size of the tree: 5897
  Confusion matrix:

a	b	classified as
1161	39	a - malware
792	2008	b - not malware

In comparison with the other algorithms that we have used, we can see that AdaBoost is not very good at classifying not malware instances as not malwares and makes many wrong predictions about that class, but it's the best in classifying malware instances as malware with 1161 correct predictions out of 1200 total being better than libsvm, 1137 out of 1200, at this particular task, but with a total accuracy worser than most algorithms.

6 Conclusion

All our algorithms used, under certain circumstances, give similar results. It's very important, in order to obtain the best possible results, to provide to each algorithm a representation of the data that favours the way in which the classification is done by that particular algorithm. In other cases the construction of the model cannot be done in a feasible time by deciding a certain representation, so we have to rely maybe on a ”weaker” model, but nevertheless one that allows us to do a classification on as many instances as possible. To conclude we will see the results of the Naive Bayes algorithm on the entire dataset, represented using the concatenation method, that offers an accuracy that is certainly acceptable with 5201 right classifications of the malware class.

Correctly classified instances: 92.1379%

Confusion matrix:

a	b	classified as
5201	354	a - malware
9788	113655	b - not malware

References

[1] Drebin: Effective and explainable detection of android malware in your pocket. http://filepool.informatik.uni-goettingen.de/publication/sec//2014-ndss.pdf.

[2] Weka 3: Data mining software in java. https://www.cs.waikato.ac.nz/ml/weka/.

[3] Libsvm – a library for support vector machines. https://www.csie.ntu.edu.tw/ cjlin/libsvm/.