Malware Detection using Machine Learning

Abhilash Parthasarathy
University of Delaware
Newark, Delaware
abhipart@udel.edu

Vinit Singh
University of Delaware
Newark, Delaware
vinitvs@udel.edu

Mingxing Gong
University of Delaware
Newark, Delaware
mingxing.gong@udel.edu

ABSTRACT

This project uses machine learning to detect the intrusion of malware. The basic approach is a combination of extracting various features from the training set and building a machine learning model. We are using gradient boosting (xgboost) package as we learnt from the Kaggle team [1] while modelling and further we calculate the log loss error for various samples.

CONCEPTS

Machine Learning → Random forest algorithm, **Gradient boosting**.

KEYWORDS

Feature engineering, Learning Model.

I. INTRODUCTION

Our study is based on the first-place code and documents for a Kaggle competition [1]. Microsoft Malware Classification Challenge', which requires to classify malware into families according to the file content and characteristics. Following the code, we first replicate the code using the small sample datasets (10 asm files and bytes files) and we encounter working environment issues (python 2 instead of python 3, xgboost installation, pypy etc). Although finally we solved these issues, the code is running very slowly. After consulting with teaching assistant, we move all data to AWS and run the codes under UNIX environment, which greatly improves our efficiency.

Following the instructions of Kaggle code, we generate 4-gram byte, single byte frequency, instruction count, function names and Derived assembly features. We also generate three golden features (Opcode count, Header count & asm pixel intensity). There are over 71k features and it is very sparse. We use random forest to select useful features and finally around 4k features have been selected. In this report, we compare the time extraction of features varying by different sample size. And also we use simple semi-supervised model to classify these malware into 9 classes based on these features. Cross validation is used to solve overfitting issue and log loss is key metrics to assess the model efficiency. The figure 1 [1] explains the over view of our approach. So, we start off with the extracting of features then move on to gradient boosting and further developing the model. Samples are fed to the model and a processing time vs sample graph is observed later in the project.

We run single_model.sh file as the model that links feature extraction programs and the semi model program.

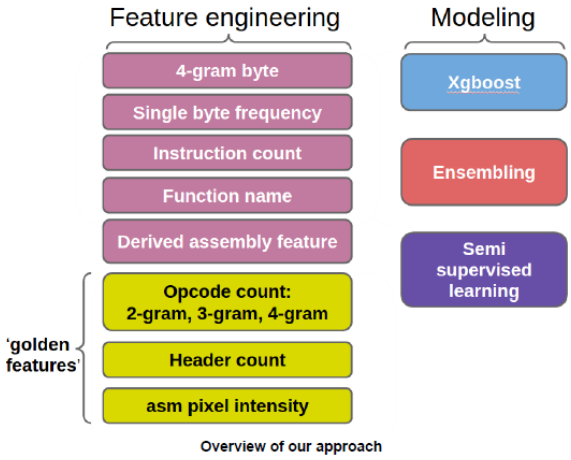

Fig 1. Overview of our approach [1]

II. Methods

Feature Extraction

1. Format of the Process Flow

a) Extract:

1. Transfer training data and test data from large data set to data folder. The size of this subset is defined by the user.
2. From this subset run createnewlabels and create a new csv file that will be used to extract features.
3. Extract all features.

b) Training Time:

1. Run the model for the subset of training data.
2. Store the time it took to run it.
3. Store Log loss error related to it.
4. Store (Size of training data, Training time, log loss error) into a csv file.

c) Visualize:

1. Use the csv file from training time to plot a graph of training size vs time.
2. Use the csv file from training time to plot a graph of training size vs log loss error.

3.These graphs are further made while the model runs sample starting from 10 to around 1000.

2. Modeling

The single model is linked with feature extraction and classification. So, after the features are extracted, the single model will classify the malware accordingly. Test data is used for verifying the efficiency of th model and log loss error of the test data. The training data is used to create the model. So, the model will classify the training data into different classes using the extracted features. The Test data will be the new malware files that will test the training model and prove the efficiency of the model. So, when the model comes across a new malware, it will extract the feature and check if it matches the existing classes.

III. RESULTS AND DISCUSSION

3.1 Log loss error

Log loss error is obtained for every sample and a graph is plotted.

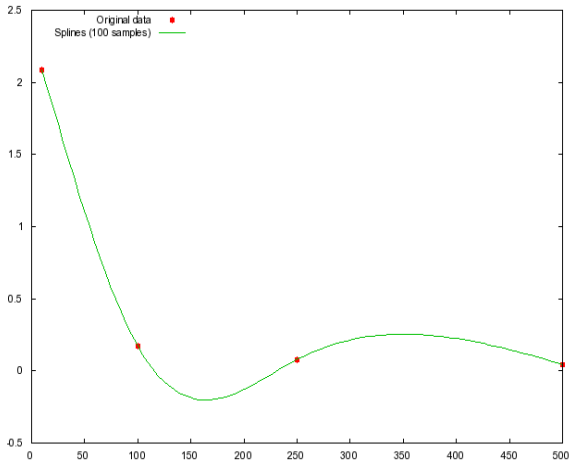

Fig 2. Log loss vs sample

So the figure 2 shows a graph between log loss obtained vs the samples used, The X-axis consists of samples 50,100….500 and the y-axis consists of log loss error from 0.5 to 2.5. So, we can observe in the graph that there is a decrease in log loss error as the sample size increases. Except, there is an increase from 150 to 250 and there is another decrase from 250 to 500. This proves our analysis that the log loss error is minimized

3.2 Processing time

3.2.1 Training time

So, we calculate the time taken to process every sample, similar to log loss error we consider samples from 50 to 500 and calculate the processing time for every sample.

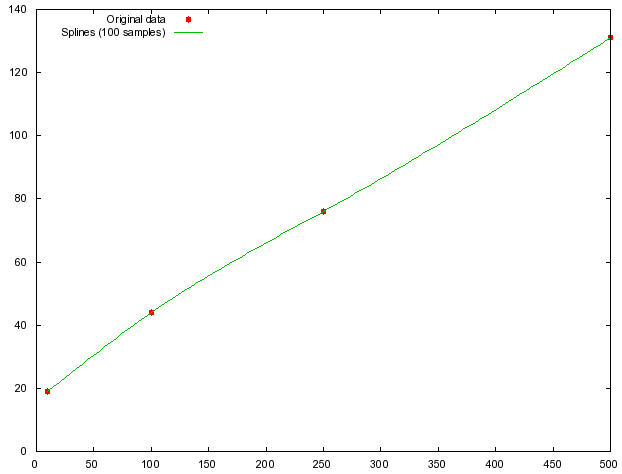

Fig 3. Train time vs sample

So, figure 3 shows us the graph between processing time and different samples used. You can see the exponential increase in the graph. We use samples 50,100,150,250,300 and we calculate the amount of time taken to process each sample. Further a graph is plotted to prove our analysis. The X axis shows the samples used and Y-axis shows the time taken to train the samples.

3.2.2 Feature extraction time

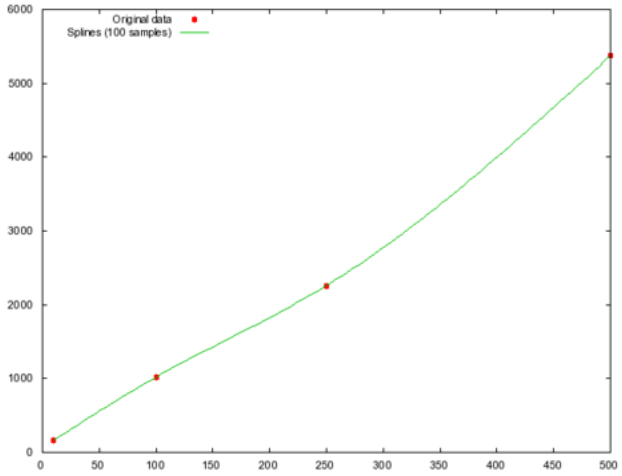

Fig 4. Extraction time vs sample

So, figure 4 shows the y-axis as extraction time and x-axis consists of samples from 50 to 500. You can see the graduate increase in extraction time as the sample size increases.

IV. CONCLUSION

In summary, we are optimizing an existing approach by the kaggle team. We ae

V. APPENDIX: Scripts

We used various scripts to extract the features, in this appendix section, we are going to explain the scripts in detail. So, the format will be the input taken by the scripts, description of the script that will explain the various functions used and finally the output of the script.

A.1 Feature extraction

A.1.1 Unique grams

Input- 4 grams and Testing data.

Description-

Loading the label-Mentioning the path where the label will be loaded. Using dictionary reader to open the specified path. Putting data in each of the output csv file.

Grams dictionary- Basically, generates gram dictionary for a file. Opening the path for the dictionary. Adding one list of grams(strings) to the dictionary. Further, using a tree and transferring the contents in dict to it. Displaying the tree as the output.

Reduce_dictionary- This function is used to add up ngrams dictionary. Using result variable and initially transferring the contents of the dictionary to the assigned variable. The dictionary consists of labels and the names of the labels, initially it prints the existing information. The ngarms are added up in the initialized result array. The result array is printed.

Heap top -Initializing the heap Defining a counter and checking it with the root of the heap. Further transferring the labels and ngrams to the heap/Printing the heap.

Output-

Trainlabelsnew.csv containing the loaded labels which in turns consists of 4 grams.

A.2.2 Join grams

Input- Takes ngrams and testing data as the input, this program depends on unique grams.

Description

Join ngrams: Merging all the n-grams Initializing a heap. Heaps will consist of n-grams extracted. dict_all will contain the joined ngrams. dict_all will be displayed.

Num_instances: Printing the path where the dictionary consists of the instances. Num_instances are generated using the existing labels. Printing the generated instances.

Entropy, Info gain : Calculates variable ratios of the given instances for entropy. Information gain is calculated using the entropy.

Heap gain: Features of the num instances are generated. Initializing a heap and storing the generated features. Calculating heap gain by comparing the heap with the information gained earlier. Printing the heap gain.

Gen_df Generating all the features that includes the joined ngrams we calculated earlier-Printing the dictionary path and names. Building a new dictionary that will contain the ngram features, comparing it with the existing dictionary by performing a look up approach and checking if it has the same ngrams. **Generating the binary features**-Further num_instances, joined grams and the entire features are transferred to the 2 csv files as the output

Output

Trainlabels750.csv and Testlabels750.csv, 2 files for train and test each.

A.2.3 Dll

Input- testing data and dll single file

Description-

Load labels-Loading the labels and mentioning the path. A result array is created and the labels are loaded into it. Displaying the result array.

Dll single file-Two variables are used- dict_pattern that describes the pattern of dict and f_lines that consists the list itself. Then it uses an external variable 'p' that does strip and split function. Further it is checked if lines in p are similar to f_lines. Finally, pattern dictionary are displayed.

Reduce dict-Loading the labels into the trainlabelsnew.csv file Create a dict_all that will store the features generated. The variable f_name will have the .asm data and train labels Check if the features match the ones in dict_all. Print dict_all,

Num instances-The path and labels are considered. Opening the dictionary reader. Setting up counters p and n to check if the class in the label Displaying the variables p and n.

Entropy and Info gain- Calculates variable ratios of the given instances for entropy. Information gain is calculated using the entropy.

Heap gain- Initializes a heap and considers the root as the first element of the heap. Counts the items in the list and checks if the grams are in dict_all. Then it determines the sum of class labels by adding previous to the next. The heap gain is found using the information gain and checked if it is higher than the root of the heap. Result variable stores the gain, it is displayed along with the heap itself.

gen_df- First, this function tests if the data is train or test. Further, it generates a list for binary features and dict for tmp_pattern. It checks if the features are in the tmp_pattern and then adds the binary features to the list. The program adds num-instances generated to the output csv files.

File Output- testdll.csv and traindll.csv containing the dll features.

A.2.4 Instruction count

Input:. byte file Training and testing data

Description of Working: For all. byte files of both training and testing data. For row in that .byte, filecode = row - offset (first element) code is converted to two bytecodes-frequency. The two

bit code is calculated and is kept on incrementally added. Then consolidated by counting final frequency for each hex value. Then this row is added to the csv.

File Output : file consisting hex value frequency for each file

A.2.5 Image feature

This code is to generate image features of the data. As we know malware binaries can be visualized as gray-scale images, the images that belong to the same family would have similar layout and texture. In this code, we have got asm and bytes image features for each train and test groups.

A.2.6 Frequency count

Input:. byte file Training and testing data

Description of Working:

For all. byte files of both training and testing data. For row in that. Byte file code = row - offset (first element) code is converted to two bytecodes frequency this two byte code is calculated and is kept on incrementally being added Then consolidated by counting final frequency for each hex value Then this row is added to the csv file

Output:. csv file consisting hex value frequency for each file

A.2.7 get id

Description: These two codes are used to generate all the features and do the segment count. As Kaggle points out that a simple count of lines of each segment in the asm file can also provide very useful information, which can also be treated a feature for the modeling.

Output: It creates two files xid_train.p and xid_test.p

B. Single Model

Description- So this is the actual model that is linked to various feature extraction and semi model.
The single model script first creates a separate folder 'gram' and it does the following for 9 iterations.
Executes unique_gram.py python script and it gives the iteration number as argument to the same.
This helps in creating unique set of grams for creating dictionaries. It asks a user input gram next, if the user chooses too, otherwise it breaks and goes to next iteration.
Overall, when we run the file, it first extracts the features unique grams, join-grams, frequency count, instruction count, image features, dll. Further, xgboost was the package we used for gradient boosting, semi model program runs. The samples are used to train the model, so the program runs over and over again, the processing time is noted and compared with the size of the samples. We have already discussed about the plot in Fig 4.0

Output- Runs for several samples starting from 10 to 500 and the processing time is noted.

VI. RELATED WORK

At first stage, we had already extracted the 4k features based on random forest and used semi supervised model to classify malware into 9 classes based on these features. Our next step is to feature selection and compare the performances of these features. Would the golden features outperform other features? Currently the feature selection is based on random forest algorithm. As we know a lot features are redundant or irrelevant, present several problems such as misleading the algorithm, overfitting and increasing the model complexity and running time. We would also try other feature selection methods, like information gain. [2]
In Kaggle, the author mainly uses Xgboost model developed by Tianqi Chen and Bing Xu. They also compared with three machine learning algorithm: Ensembling, Calibration and Semi-supervised learning. Semi-supervised learning is found to be most effective. Besides these techniques, we also found that people are using Bayesian classification-based approaches for Android malware detection [3]. As well as use of K-Nearest Neighbor (KNN) classifier for intrusion detection Compared with the three techniques we used in Kaggle, KNN is more straightforward and simple. It does not require generating so many features, that is why this method is also widely used. And also, we would like to try different machine learning algorithm, like Bayesian & KNN. Besides, we will apply to the datasets provided to assess the stability and the consistence of the model.
As per [3], Machine learning experts have focused on using Bayesian classifiers to detect unknown android malware using static analysis. As per [4], Text categorization are adopted to convert each script to vectors and calculate the similarity between both the programs. It uses K-nearest neighbor classifier used to classify the programs to observe the behavior in order to check for intrusion.
So, there are traditional tools that perform well in certain scenarios but they will not be able to classify newly formed malwares. So researchers [5] come up with a good approach that combines machine learning and crowdsourcing. So, they developed an architecture called *smartnotes* where users share their experiences of web threats and knowledge is gathered and fed to the machine learning model along with various malicious malware, given as the training set.
Despite using static features and existing tools, predicting a Malware's behavior has become a very hard task. The new malwares being generated, cannot be directly put into feature engineering. In addition, the existing tools are not enough to predict the behavior. This is where un-supervised learning comes into play. When the model encounters a malware that is very hard to recognize, it will put it into an unknown class until a new malware with the same behavior is seen. Then it can assign a new class to the malware and add labels to it. However, the log loss error and over-fitting have been the main concern and the gradient boosting model will make sure that there will be low log loss error and no over-fitting.

VII. LIMITATIONS

The main limitation of this project is that it cannot run on multiple processors because the extraction of the features is dependent on each other. The scripts are not upto date since the kaggle code is written on python 2.7 and other versions are not compatible.

VIII. ACKNOWLEDGMENT

This work was primarily done by the Kaggle team that won the Microsoft malware classification challenge (BIG 2015). Additional to gradient boosting and modelling, they also did semi supervised learning. We will be considering it further in the second part of the project.

The training set of the kaggle team was initially divided into 9 classes. They were successfully able to classify the given training set in the competition and the gradient booting method became a success.

The Kaggle team were able to give a log loss error with a very small data set. We were able to use their packages for a large data set provided to us. They were also able to avoid over-fitting of data.

We would like to thank the kaggle team for providing the scripts and xgboost package that helped us to classify the given training set.

IX. REFERENCES

[1] Jacques Cohen **(Ed.).** 1996**.** Special Issue: Digital Libraries**.** ***Commun. ACM*** **39, 11 (Nov. 1996).**

[2] Zarni Aung, Win Zaw International journal of scientific and technology research volume 2, issue 3, *Permission-Based Android Malware Detection* (MARCH 2013)

[3] Suleiman Y. Yerima, Sakir Sezer, Gavin McWilliams, ***Analysis of Bayesian classification-based approaches for Android malware detection*****, IET information security(volume: 8 Issue :1, Jan 2014)**

[4] Yihua Liao, V.Rao Vemuri, *Use of K-Nearest Neighbor classifier for intrusion detection*, Computers & Security, Volume 21, Issue 5, 1 October 2002, Pages 439-448

[5] Eugene Fink Mehrbod Sharifi Jaime G. *Carbonell Application of Machine Learning and Crowdsourcing to detection of Cybersecurity Threats* Computer Science, Carnegie Mellon University, Pittsburgh, PA