# 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

## ABSTRACT

The overall goal of the project is to detect the intrusion of malware using machine learning models. Initially, The kaggle team that won the Microsoft Competition implemented a basic approach. It is a combination of extracting various features from the training set and building a machine learning model.
We noticed that their approach to execute the scripts consumed a lot of memory and time. Hence, we came up with a solution that will reduce the executing time and memory consumption.
We follow a parallel processing approach and prove why our method is more efficient than the approach followed by the kaggle team.

## 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.
the project. The Kaggle team.
The kaggle team approach is to extract the features then further feed the selected features into a model. They also use ensemble learning to choose the right model. Our project is to optimize their approach and make it more efficient i.e reduce the processing time and RAM consumption.
So, we came up with parallelly executing the feature scripts in two different instances. The reason why we chose two instances- There are feature extraction scripts that are dependent on each other. So, we split the extraction of features in terms of dependency.
To prove our analysis, first, we ran the scripts separately one by one to observe the processing time. Second, we ran them in parallel. We compared the processing time and RAM consumption of features extracted, plotted graphs for log-loss calculation and data flow. We were able to determine that the execution time reduced by 70% compared to the Kaggle's method of feature extraction.
In brief, we did HTOP analysis first, it was not accurate enough for us to tabulate it, we will be discussing about it later in the report. So, we used python RAM analysis to calculate the RAM consumption for both serial and parallel execution of feature scripts.

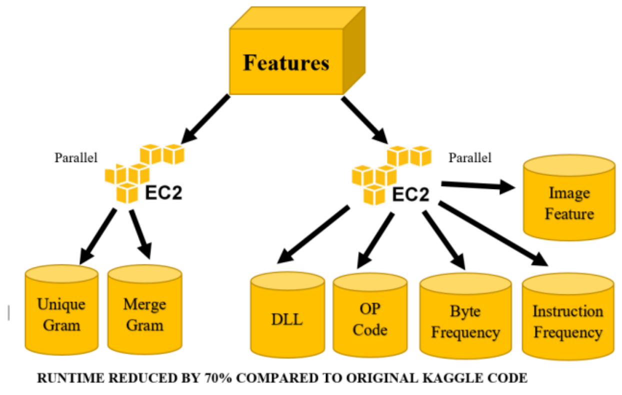


**Fig 1. Overview of our approach**

We can observe from fig 1 that there are EC-2 instances used, one for unique gram and merge gram, another for DLL, OP Code, Byte frequency, instruction frequency and image features. So, the figure describes EC 1 and EC 2 instances, unique gram and merge gram runs on EC1 and the other features run on EC 2. Merge gram feature extraction fully depends on the unique gram extraction, so it cannot be parallelly processed.

## II. Methods

### Feature Extraction and selection

Before we get started on our suggested approach, we would like to explain how feature extraction and selection is implemented. We want to get into detail about the Kaggle's method used to extract the features. It was originally discussed by another paper [2]. The Kaggle team used their methodology and performed the extraction using Xgboost (Gradient boosting package). Further, they fed the extracted and selected features to the model.
We will explain about n-grams extraction [2] to get an accurate view of feature extraction. Two feature scripts known as unique grams and merge grams are extracted. Feature extraction using n-gram analysis involves extracting all possible n-grams from the given dataset and selecting the best n-grams among them. Each such n-gram is a feature.
So [2] the Kaggle team chose information gain (IG) as the selection criterion, because it is one of the best criteria used in literature for selecting the best features. So, Kaggle divided the feature scripts into two types- Novel features such as opcode N-gram, asm file segment count and asm file pixel intensity, Single-byte frequency, byte 4-gram instruction count, function names and Derived Assembly Features (DAF).

## Modelling

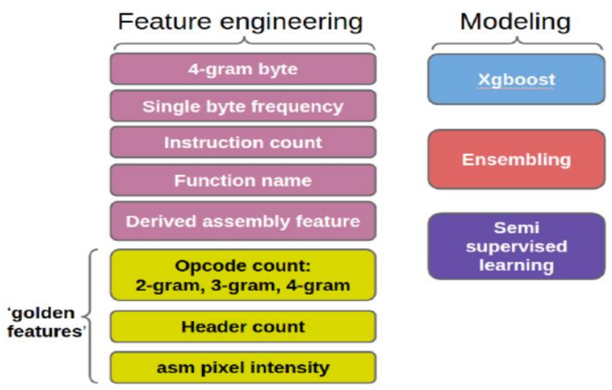
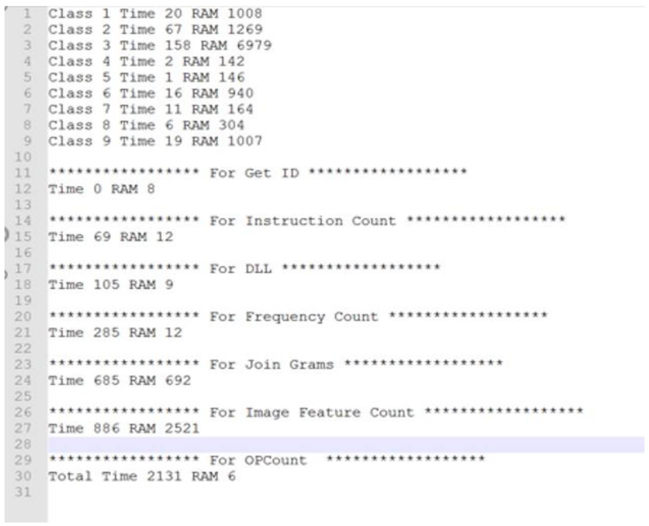


**Fig 2. Overview of the model [1]**

The main model is Gradient Boosting. We used the package Xgboost The Kaggle team combined Ensembling and semi supervised learning. The performance is boosted using Xgboost after the features are extracted and selected. Ensemble models are developed to classify them into different malware families. The strongest classifier that shows low log-loss error is chosen as the right classifier,

## III. Parallel approach

### 3.1 Log feature



```
Class 1 Time 20 RAM 1008
Class 2 Time 67 RAM 1269
Class 3 Time 158 RAM 6979
Class 4 Time 2 RAM 142
Class 5 Time 1 RAM 146
Class 6 Time 16 RAM 940
Class 7 Time 11 RAM 164
Class 8 Time 6 RAM 304
Class 9 Time 19 RAM 1007

***************** For Get ID ******************
Time 0 RAM 8

***************** For Instruction Count ******************
Time 69 RAM 12

***************** For DLL ******************
Time 105 RAM 9

***************** For Frequency Count ******************
Time 285 RAM 12

***************** For Join Grams ******************
Time 685 RAM 692

***************** For Image Feature Count ******************
Time 886 RAM 2521

***************** For OPCount  ******************
Total Time 2131 RAM 6
```

### 3.2 Why python RAM analysis is better than HTOP

Since, the HTOP analysis considers the computer Memory as well when it displays the RAM consumption, we used python instead to give an accurate reading. It gave better analysis than HTOP since it looks at a single script rather than entire load.

## III. RESULTS

### 3.1 Processing time

Consider the following table

| Data instances | Data size | Freq count | Inst count | Image feature | DLL | OP count | Non-parallel | Parallel | Time saved |
|---|---|---|---|---|---|---|---|---|---|
| 100 | 600MB | 56 | 12 | 196 | 17 | 454 | 734 | 454 | 281 |
| 250 | 2.5 GB | 115 | 27 | 382 | 49 | 1074 | 1648 | 1074 | 502 |
| 500 | 4.5 GB | 285 | 69 | 886 | 105 | 2131 | 3476 | 2132 | 1344 |

**Fig 3. Processing time**

**So in figure 3, we calculated the processing for all the feature scripts starting from unique gram to OPcount generation. Again, we used the sample size data ranging from 100 to 500. You can see that the scripts run faster in parallel. So, it takes almost double the time if you use the Kaggle team's way of executing it. The maximum time taken was by the OP count. So, if you execute the feature extraction in parallel, the entire process takes the same amount of time as executing OP count separately.**

### 3.2 RAM consumption

| Data instances | Data size | Unique Gram | Merge Gram | Freq Count | Instr count | Image feature | dll | Get_ID | Gen_ Opcount |
|---|---|---|---|---|---|---|---|---|---|
| 100 | 600MB | 1170 | 701 | 12 | 12 | 2344 | 8 | 8 | 105 |
| 250 | 2.5 GB | 3595 | 637 | 12 | 12 | 2251 | 9 | 8 | 91 |
| 500 | 4.5 GB | 6979 | 692 | 12 | 12 | 2521 | 9 | 8 | 243 |

**Fig 4. RAM consumption**

**So in figure 4, we calculated the RAM consumption for all the feature scripts starting from unique gram to OPcount generation.**
**Again, we used the sample size data ranging from 100 to 500. Unique gram consumes the maximum RAM, the feature generation of n grams needs a lot of memory.**

### 3.3 Unique gram time vs log loss

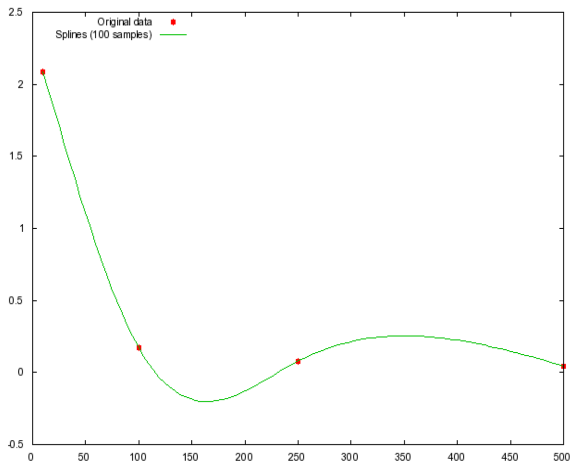
| Data instances | Data size | Uni-gram (3 gram) | Logloss (3 gram) | Uni-gram (4 gram) | LogLoss (4 gram) | Log loss difference | Time difference |
|---|---|---|---|---|---|---|---|
| 100 | 600MB | 1072 | 0.44 | 1170 | 0.46 | 0.01 | 9 secs |
| 250 | 2.5 GB | 1836 | 0.21 | 3595 | 0.19 | 0.02 | 24 secs |
| 500 | 4.5 GB | 1957 | 0.076 | 6979 | 0.073 | 0.03 | 40 secs |

**Fig 5. Unique gram vs log loss**

**So in figure 5, we calculated the log-loss and time difference for the execution of both 3 gram and 4 gram. The sample size was used ranging from 100 to 500.**

### 3.3.1 Unique gram vs log loss ( will make the graph)

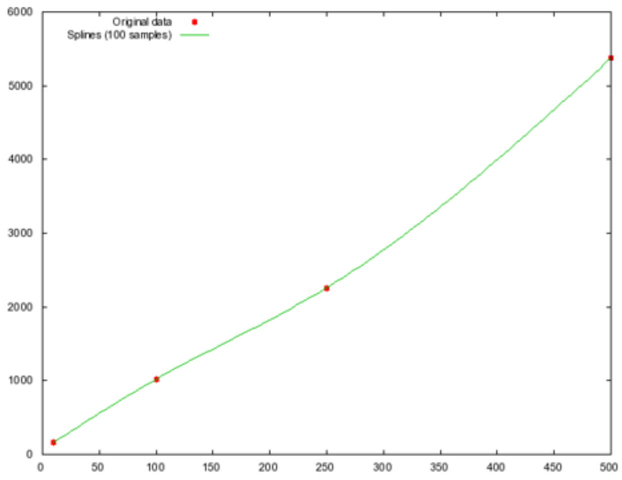
**Now, we will plot the graph between the Unique gram and the log loss**



**Graph 1. Log loss vs sample**

### 3.3.2 Extraction time vs sample

**Now, we plot the graph by calculating time taken to extract each sample**



**Graph 2. Extraction time vs Sample**

## IV. CONCLUSION

In summary, we have optimized the extraction of features and modelling approach suggested by kaggle team. We reduced the processing time and RAM consumption taken by the feature scripts and the model. We implemented a parallel processing approach to the problem. We divided the feature scripts into independent and dependent features, further we ran the independent scripts on one instance and the dependent features on the other.

We compared the parallel processing method with the regular feature extraction method. We were able to observe a tremendous decrease in the time taken to execute the scripts and the RAM consumed by it. We were able to reduce the processing time by 70%. The particular feature Opcode consumed the maximum amount of RAM during the extraction.

We proved our analysis by forming tables and comparing the processing time between series and parallel. Further, we plotted the graphs to give a clearer view of our analysis.

So, from the three tables we can see the efficiency of parallel execution. Both, RAM consumption and execution time are saved.

## V. FUTURE WORK

*1.Modifying the existing data set in order to implement the kaggle mode.*

2. *Testing and training the model on the entire data set*

We could train a sample set of data to prove our analysis, in the future we can train it on the entire data set. In order to make the model as efficient as possible, the training set must be very large. Larger the training set, more efficient the model is. We were not able to implement the entire training set, since we just wanted to prove why our approach reduces extraction time and RAM consumption

*3.Further reduction of execution time.*

We can observe how the parallel processing works with an entire data set and figure out a way to reduce the RAM consumption and execution time further.

## VI. RELATED WORK

[2] The authors have a similar approach to the problem. They use feature extraction and further they select the features based on the calculation of information gain. Later, they feed it to the model. Their feature extraction technique is multi-level and they have a combination of unique features such as ngrams, assembly level functions, dll at various levels of abstraction. [1] The Kaggle have a similar approach that helped us with the project but we combined all the ngrams together and created join grams, that is fully dependent on the unique grams. The rest of the features are independent features. Kaggle team also used gradient boosting to extract the features.

The authors [3] had a similar approach but they considered the features to be different images. Different sections of the binary features were viewed as an image. The images were able to capture small changes and it helped them to identify the difference between malware and benign samples.

[4] The authors took part in the Microsoft Malware challenge competition alongside the kaggle team, their feature extraction

and classification accuracy was close to 0.098, which is impressive. In their work, Features were grouped according to different characteristics of malware behavior an fusion was performed according to a per class weighting paradigm.

[5] We propose a generic and efficient algorithm to classify malware. Our method combines the selection and the extraction of features, which significantly reduces the dimensionality of features for training and classification.

## VII. ACKNOWLEDGMENT

We would like to thank the Kaggle team that won the Microsoft malware classification challenge, for introducing us to this approach. 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.

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 could use their packages for a large data set provided to us.\

They were also able to avoid over-fitting of data. Basically, we optimized their scripts in order to reduce the processing time and RAM consumption.

We would also like to thank (that helped us understand the flow of scripts. The authors combined feature extraction and feature selection of ngrams that helped us understand the ngrams feature script created by Kaggle. To get a better view of the ngrams feature extraction, we went through the authors's work [2].

## VIII. REFERENCES

**[1]** Xiazhou Wang, Jiwei Liu, Xueer Chen, *Microsoft challenge (Big 2015)*

[2] Mohammad M Ashud, Latifur Khan, Bhavani, Springer Science Oct 23 2007 *A scalable multi-level feature extraction technique to detect Malicious executables*

**[3]** Chih-Ta Lin, Nai-Jai Wang, Han Xiao, Claudia Eckert Journal of information and Science Enginneering 31 2015 *Feature selection and extraction for malware classification*

[4] Mansour Ahmadi, Dmitriy Ulyanov, Stanislav, Mikhail, Giorgio Cornell University Library March 10 2016 *Novel feature extraction and selection for effective Malware family classification*