Application of machine learning in malware file classification

by

Guangjie Shi

(Under the direction of Khaled Rasheed)

## Abstract

In this thesis multiple advanced machine learning algorithms, such as random forests, boosted treess, support vector machine, etc., were applied to investigate the problem of malware file classification. Feature engineering procedures were performed on a large dataset ($\sim 400G$) of malware files provided by Kaggle.com. Four different feature sets were generated: filesizes and header string frequency, byte-sequence n-grams, opcode n-grams and image features. Each of these feature sets was studied individually at first, and then different combinations of them were investigated in detail. Moreover, the importance of different features was studied and discussed as well.

Index words: Machine Learning, Malware Classification, N-gram, Local Binary Pattern, Random Forest, Support Vector Machine, Gradient Boosted Tree.

# Application of machine learning in malware file classification

by

Guangjie Shi

B.S., Xiamen University, 2011

A Dissertation Submitted to the Graduate Faculty

of The University of Georgia in Partial Fulfillment

of the

Requirements for the Degree

Master of Science

Athens, Georgia

2016

# Application of machine learning in malware file classification

by

Guangjie Shi

Approved:

| | |
|---|---|
| Major Professors: | Khaled Rasheed |
| Committee: | Roberto Perdisci<br>Kang Li |

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
August 2016

# Acknowledgments

First and foremost, I would like to express my deepest gratitude to my adviser Prof. Khaled Rasheed, for his invaluable guidance, support, and patience throughout my MS studies. I would also like to thank Prof. Roberto Perdisci and Prof. Kang Li for invaluable discussions and for serving on my committee. Last but not least, I would like to thank my family for their endless support and unconditional love throughout all these years. Ever since I was a child, my parents have provided me with excellent education opportunities, and have respected every decision I made. Without them, I would not have been able to achieve this much.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

With large financial support in recent years, the malware industry has become a well-organized market. The traditional protections of anti-malware vendors are greatly challenged by multi-player syndicates. One of the major challenges is that there is a vast amount of files that need to be evaluated for potential malicious intent. For instance, Microsoft's real-time detection anti-malware products inspect over 700 million computers monthly, which generates tens of millions of daily data points to be analyzed as potential malware [5]. The reason behind the high volume of different files is that malware authors introduce polymorphism to the malware components. Therefore, even malware files that belong to the same malware "family" can look totally different from each other by modifying or obfuscating through various tactics.

The problem we try to address in this study is whether or not we could find an effective way to classify malware files into their respective "families" by using state-of-the-art machine learning techniques. The dataset that we used in this study is posted on Kaggle.com by Microsoft [5]. This dataset contains two parts: the training set and the testing set. In the training set, there are 10,868 labeled malware files belonging to 9 different families; in the testing set, there are 10,873 unlabeled malware files. For each malware file, the raw data

contains the hexadecimal representation of the file's binary content, without the PE header. Each file also comes with a metadata manifest, which is a log containing various metadata information. These metadata manifest were generated using the IDA disassembler tool. We will describe in detail how the process of feature engineering (Ch. 3) is performed on these two types of files in order to extract meaningful information. The nine malware classes with their occurrence in the training set are listed in Table 1.1. The histogram of these nine malware families is shown in Fig. 1.1. One can tell that the training set is very unbalanced throughout different classes. For example, the C3 class has the most instances, 2942, while the C5 class has only 42 instances.

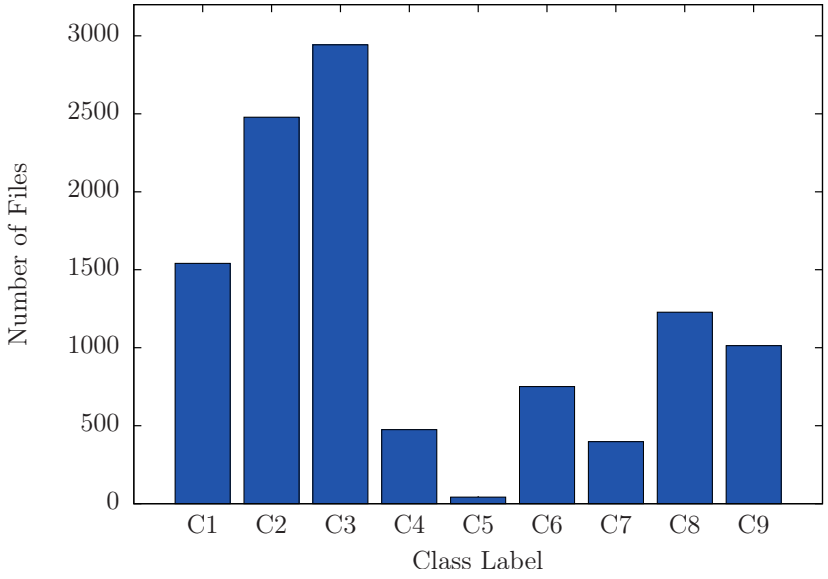


Figure 1.1: Histogram of the occurrence for each malware class in the dataset.

Table 1.1: Nine malware classes in the dataset, with their occurrence

| Class ID | Malware Family | Number of files |
|---|---|---|
| C1 | Ramnit | 1541 |
| C2 | Lollipop | 2478 |
| C3 | Kelihos_vers | 2942 |
| C4 | Vundo | 475 |
| C5 | Simda | 42 |
| C6 | Tracur | 751 |
| C7 | Kelihos_ver1 | 398 |
| C8 | Obfuscator.ACY | 1228 |
| C9 | Gatak | 1013 |

# Chapter 2

# Methods and Metrics

## 2.1 Classification Methods

### 2.1.1 Decision Trees and Ensemble of Trees

Decision trees, one of the most popular classification methods, exhibit flowchart-like structure. Each internal node of the tree performs the "if-else" test of one attribute, the outcome of which will lead to the according branch. The classification rules of decision trees are represented by all the paths from root to leaf, which represents a class label. In Fig. 2.1, we show an example of the graphical representation of a decision tree for car buying. With gathered information about a car (e.g. price, mileage and etc.), one could easily go through the decision tree from top to bottom and obtain a decision. For instance, a certified used car with mileage less than 100,000 miles and price less than $5,000 will result in a decision of purchase. This example shows very nicely the advantages of the decision tree method:

1. The graphical representation give it excellent interpretability.

2. The process of making decisions in a decision tree mimics the human decision-making process.

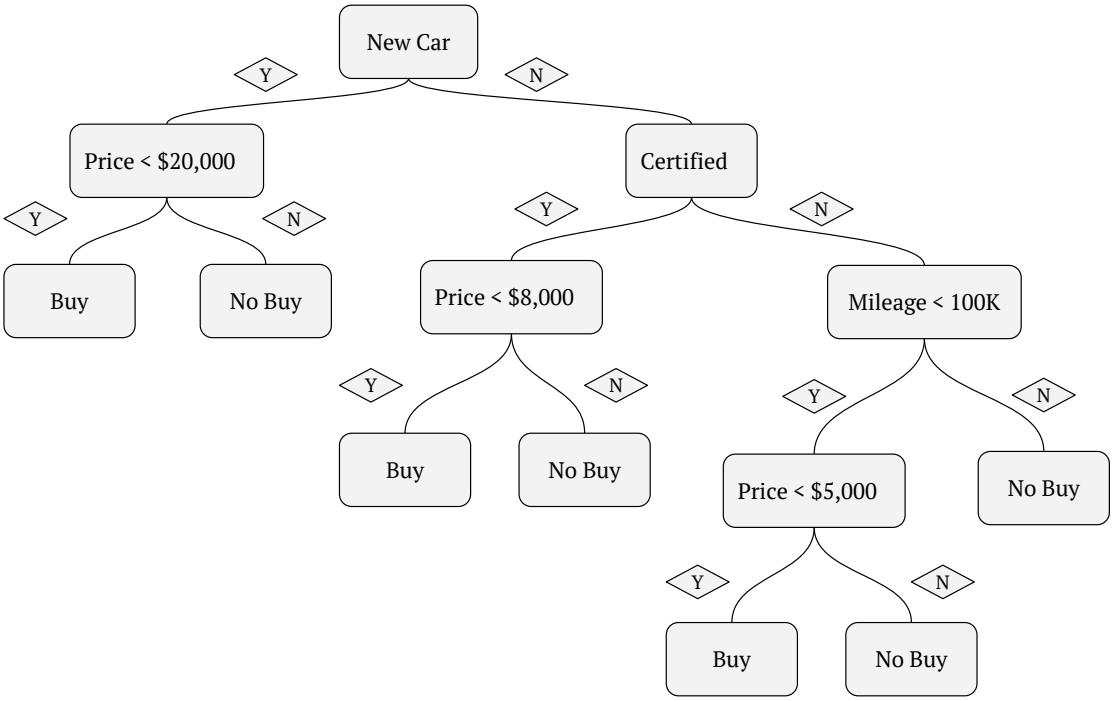


Figure 2.1: One example of the graphical representation of a decision tree.

In the case of determining which attribute is to be chosen as an internal node, decision trees have different classification criteria, the two most popular ways being Gini impurity and Cross entropy. The former is used by the CART (classification and regression tree) algorithm while the latter is adopted by the ID3, C4.5 and C5.0 tree-generation family of algorithms. For multi-class classification, one can assume there are K outcomes with values from 0,1,...,K-1. The proportion of class k observations can be calculated as:

$$f_k = \frac{1}{N} \sum_i I(y_i = k), \tag{2.1}$$

where N is the total number of observations, $y_i$ is the label of observation $i$, and $I()$ is an indicator function. Then the **Gini impurity** and **Entropy** can be respectively written as

$$I_G = \sum_k f_k \cdot (1 - f_k), \tag{2.2}$$

$$I_E = -\sum_k f_k \cdot log(f_k) \tag{2.3}$$

Even though the decision tree is a very simple idea, there is more to be considered when training one. For example, in order to avoid over-fitting the training data, one would avoid a fully grown tree by pruning a very large tree from the bottom up to its subtrees [4]. Moreover, a single decision tree is usually non-robust, or very sensitive to the training data. Therefore, it is usually used as a building block along with bagging or boosting methods to construct a more powerful classifier.

### Random Forests

Random forests, also known as random decision forests, are one of the well known ensemble learning methods, taking advantage of the simplicity of decision trees and the ensemble technique, namely the boostrap aggregating (bagging) method. For a single decision tree, the deeper it grows, the higher its variance will be due to the fact that it overfits the training set. However, the idea of the random forests algorithm is to train multiple deep decision trees on different parts of the same training set, with the goal of reducing variance. It achieves this by randomly sampling from the same training set with replacement to generate the subsets for training multiple decision trees. Moreover, at each splitting process, instead of considering all of the given features, a randomly selected subset of features will be tested. This step is essential due to the fact that there might exist one or more strong features that could dominate the others. Without this step, all of the trees that have been grown could

end up very similar to each other, i.e. highly correlated. Averaging many highly correlated estimators wouldn't reduce the variance as much as averaging many uncorrelated estimators.

#### Gradient Tree Boosting

Taking a different approach from the random forest method, the gradient tree boosting method builds an additive model in a forward stage-wise fashion. At each step, it tries to optimize the differentiable loss function by fitting estimators (trees) on the negative gradient of the multinomial deviance loss function. However, in order to obtain the optimal model, multiple parameters need to be tuned. For example, the learning rate, which determines the "step size", is very important. If the learning rate is too small, the process will converge slowly and chances are it will be trapped in some local minimas. However, if the learning rate is too large, it might overlook the global minima that we are searching for. Another key factor in this method is the number of estimators to be trained. A small amount of estimators might not be able to build up a strong model, while a large amount of estimators could just overfit the data.

### 2.1.2 Support Vector Machine

The support vector machine (SVM) [3] has served as a great tool for performing classification and regression since it was proposed decades ago. It performs binary classification by constructing a hyper-plane with the goal of optimally separating the data points into two classes. The name support vectors represent those data points that are closest to the hyper-plane. For cases where the training data can be linearly separated (hard margin), the mathematical expression of support vector machine can be written as [4]:

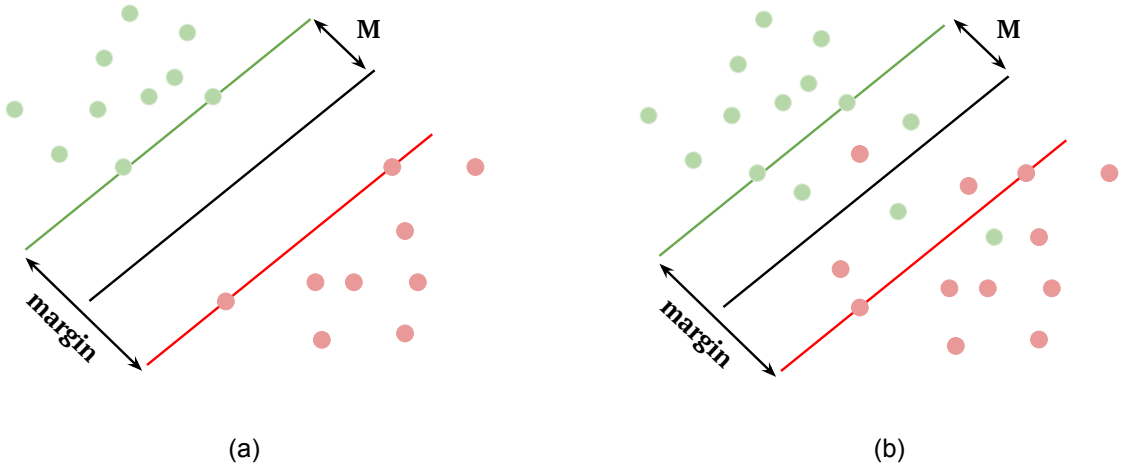


Figure 2.2: Two examples of the support vector machine, with hard margin (a) for cases where the data points are linearly separable, and soft margin (b) for cases where the data points are not linearly separable.

$$\min_{\beta,\,\beta_0} ||\beta||,$$

$$\text{subject to } y_i(\mathbf{x_i}^T\beta + \beta_0) \geq 1, i = 1, \ldots, N \ ,$$

where $y_i \in \{-1, 1\}$ is the class label, $\mathbf{x_i}$ is the feature vector of given sample point and $||\beta||^{-1} = M$ is one half of the margin (Fig. 2.2).

However, in real world problems, the training data is most likely not linearly separable. In these cases, the support vector machine can be extended by introducing the slack variable $\xi_i$, which is the proportional amount by which the prediction is on the wrong side of its

margin [4].

$$\begin{aligned} &\min_{\beta,\beta_0} ||\beta||, \\ \text{subject to } &y_i(\mathbf{x_i}^T\beta + \beta_0) \geq 1 - \xi_i, \\ &\xi_i > 0 \text{ and } \sum \xi_i < \text{constant}, \\ &i = 1, \ldots, N \ , \end{aligned}$$

Moreover, the support vector machine can also be employed to classification problems where linear boundaries are not applicable by enlarging the feature space using basis expansions, e.g. splines [4].

### 2.1.3 Naive Bayes Classifiers

Naive Bayes Classifiers are a family of simple probabilistic classifiers based on Bayes' theorem. They make the assumption that every pair of features are independent from each other given the classification. Even though the assumption is quite strong (or 'naive'), it has been extensively studied since the 1950s and has been successfully applied in problems such as text categorization. Given a feature vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, the conditional probability of a class variable y can be written as

$$P(y|\mathbf{x}) = \frac{P(y)\, P(\mathbf{x}|y)}{P(\mathbf{x})}.$$

With the naive independence assumption, we now have

$$P(y|\mathbf{x}) = \frac{P(y) \prod_i P(x_i|y)}{P(\mathbf{x})}$$

Thus the classification rule can be derived as:

$$\hat{y} = \arg\max_{y} P(y) \prod_{i} P(x_i|y) \ ,$$

given that $P(\mathbf{x})$ is constant with the same input. The difference between naive bayes classifiers mainly comes from the choice of the distribution for $P(x_i|y)$. The one we chose for this study is Multinomial naive Bayes, which assumes the data are multinomially distributed.

### 2.1.4 Artificial Neural Networks

Artificial Neural Networks (ANNs) are machine learning algorithms that are inspired by biological neural networks. An artificial neural network (as seen in Fig. 2.3) is composed of a large amount of highly interconnected elements, namely artificial neurons, simulating the biological nervous systems. As the most fundamental element, each artificial neuron takes multiple input signals and generates one output. Due to the different processing rules of artificial neurons, the different structures of the network and the different learning procedures, ANNs can be applied to a lot of very important problems, such as image processing, pattern recognition (classification), regression, etc.

## 2.2 Evaluation Metrics

### 2.2.1 Logarithmic Loss

Logarithmic loss is an evaluation metric adopted by Kaggle.com. Its multi-class formulation can be written as

$$logloss = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} y_{ij} \, log(p_{ij}), \tag{2.4}$$

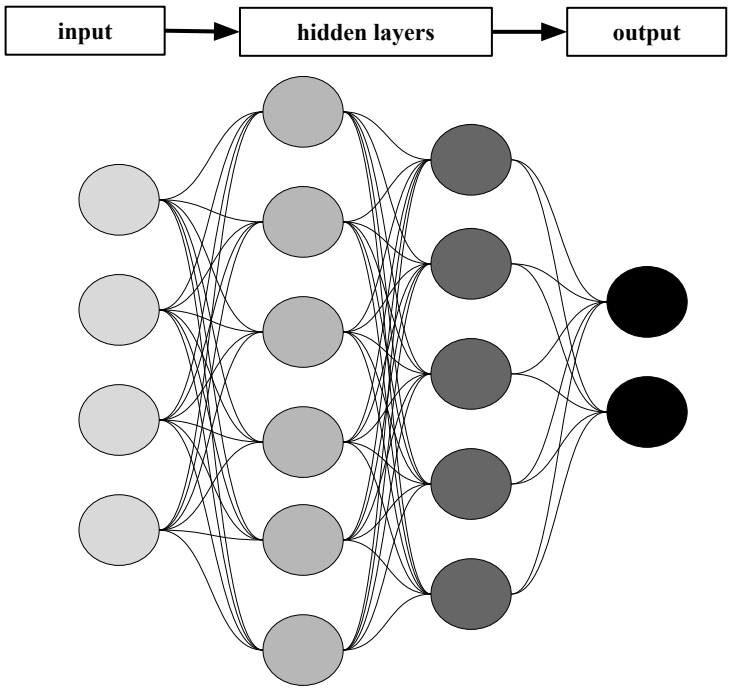


Figure 2.3: An example of Artificial Neural Networks with two hidden layers.

where N is the number of samples, M is the number of total possible classes, $p_{ij}$ is the predicted probability of assigning class j to instance i, and $y_{ij}$ is an indicator showing whether class j is the correct classification for instance i.

### 2.2.2 Precision, Recall and F1 Score

Table 2.1: An example of confusion matrix for binary classification problems.

| True Positive (TP) | False Positive (FP) |
|---|---|
| False Negative (FN) | True Negative (TN) |

$$precision = \frac{TP}{TP + FP} \tag{2.5}$$

$$recall = \frac{TP}{TP + FN} \tag{2.6}$$

$$f1\text{-}score = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{2.7}$$

In classification problems, one very important tool for evaluating models is the confusion matrix. In Table 2.1, a confusion matrix for a binary classification problem, which can easily be extended to multi-class classification, is shown. In the matrix, elements along the diagonal are those that have been correctly predicted. Some important metrics such as precision, recall and f1-score (as seen Eq. 2.5, 2.6 and 2.7) can be calculated based on this matrix.

# Chapter 3

# Feature Engineering

As described briefly in Ch. 1, two different types of files are given for each malware file: a binary file (namely **BYTE**) without PE header and a metadata manifest (namely **ASM**). Examples of these two files can be seen in Fig. 3.1. In this section, we will describe in detail how useful information is extracted and collected from these files.

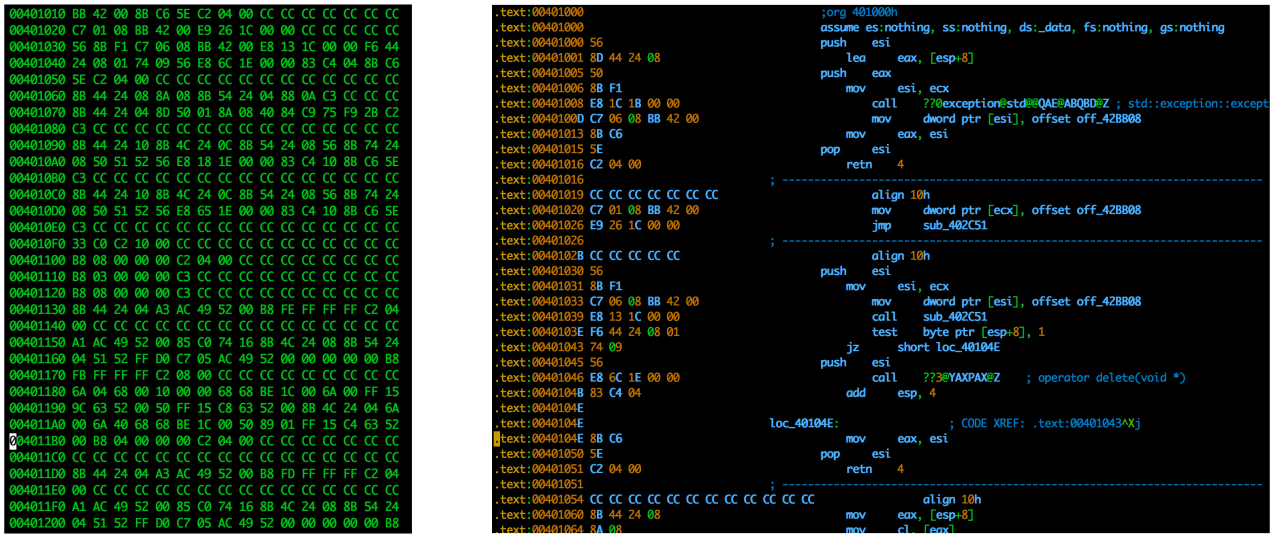


Figure 3.1: An example of the **BYTE** files on the left, and the corresponding **ASM** file on the right.

## 3.1 File Size

A simple yet effective feature that one can extract from the dataset is the size of the malware files. In Fig. 3.2 we plot the mean with standard deviation of the normalized file size for each malware class. From this plot, we can find significant variance of file sizes among different malware classes, which indicates that the file size could be a potentially good feature for distinguishing between different classes.

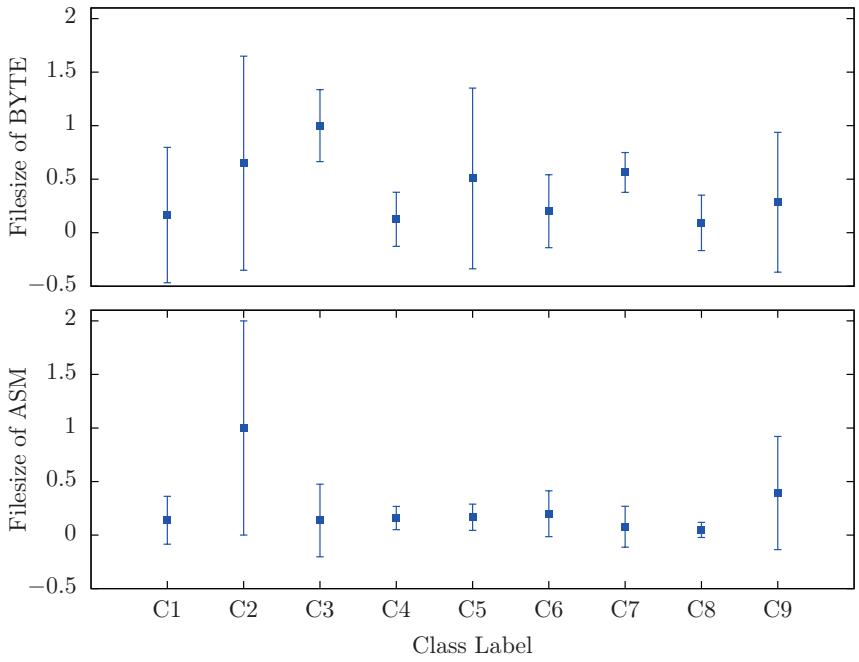


Figure 3.2: The mean values with standard deviation of the normalized file sizes for **BYTE** and **ASM** of each malware class. The y-axis indicates the standardized filesizes. That is, for each plot, all the filesizes are divided by the maximum filesize out of nine classes.

## 3.2 Header Strings

Each ASM file contains different sections, such as ".text", ".data" and so on. We first compiled a list of all of the different header strings in the training set, which resulted in 448 in total. However, most of the header strings only exist in less than 5 files, which doesn't

give much information for classification. Therefore, we picked header strings that exist in at least 5 different files, thus resulting in 66 different header strings. For each of these picked header strings, we counted the number of lines that appeared in the ASM file as our feature. Top ten header strings are shown in Table 3.1.

Table 3.1: Top ten sections in **ASM** files, with number of files that contain these sections.

| Header String | Number of files |
|---|---|
| .idata | 10339 |
| .data | 10296 |
| .text | 10291 |
| .rdata | 9255 |
| HEADER | 8341 |
| .rsrc | 6721 |
| .reloc | 2299 |
| .bss | 1047 |

## 3.3 Byte Gram

**n-gram** is a contiguous sequence of n items from a given sequence of text. For instance, given a sequence "abcda", all possible 1-gram sequences are "a", "b", "c" and "d", and all possible 2-gram sequences are "ab", "bc", "cd" and "da". Using n-gram as the analysis tool for malware analysis has been proposed by Kolter et al. [6].

For each **BYTE** file, we first counted the frequency of 1-byte gram sequences and found 257 different byte sequences in total, all of which are collected into the feature set. Following the same procedures, we calculated the frequency of 2-, 3- and 4-byte gram sequences as well. However, due to the huge amount of sequences generated, we had to make choices. For instance, in the case of the 2-byte gram sequence seen in Fig. 3.3, we only kept the top 1000 byte sequences with the highest frequencies for each **BYTE** file. Thus the mean frequencies of these byte sequences over all the malware files of the same family can be calculated. Then

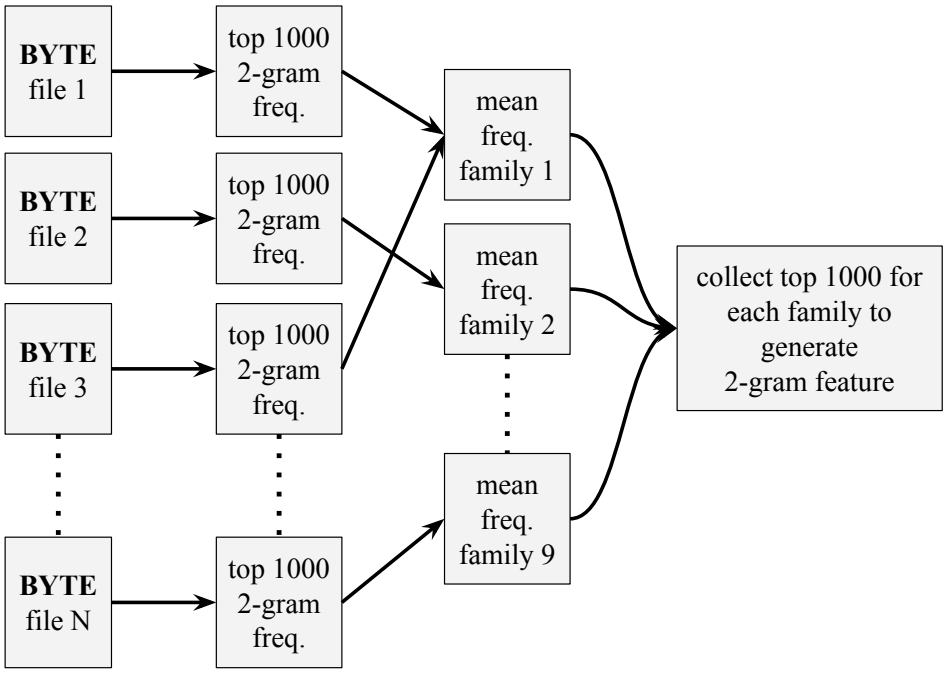


Figure 3.3: Illustration of the procedures that we applied for generating the 2-byte gram sequences as features.

the top 1000 byte sequences of each malware family are added to the feature set. The same procedures were applied to the calculation of the 3-gram and 4-gram byte sequences as well. The total number of features for each type of byte gram are listed in 3.2.

Table 3.2: Number of features for 1-, 2-, 3- and 4-byte gram sequences.

| N-gram | Number of Features |
|---|---|
| 1 | 257 |
| 2 | 4933 |
| 3 | 6144 |
| 4 | 6650 |

## 3.4 Opcode Gram

In machine language instruction, an opcode is the portion which specifies the operation to be performed. We manually collected $\sim$ 200 X86 opcodes as our opcode vocabulary. Then we generated all the 1-, 2-, 3- and 4-opcode gram sequences for each of the **ASM** files. There are 193 1-opcode gram sequences, all of which are chosen into the feature set. For 2-, 3- and 4-gram opcode sequences, due to the large amount, we followed the same procedures that were described in Sec. 3.3. The total number of features that we generated are listed in Table 3.3.

Table 3.3: Number of features for 1-, 2-, 3- and 4-opcode gram sequences.

| N-gram | Number of Features |
|---|---|
| 1 | 193 |
| 2 | 3472 |
| 3 | 5231 |
| 4 | 6633 |

## 3.5 Image Feature

### Local Binary Pattern

The local binary pattern has been an excellent visual descriptor used for image classification problems since it was proposed in 1990[2]. Particularly, it is known for its good performance in terms of texture classification. The essential idea behind it is actually very simple. Instead of taking the whole image as a high-dimensional vector, the local binary pattern focuses only on the local features. That is, for each pixel in the image, as seen in Fig. 3.4, we first compare its intensity with those of all the neighbors around. If a neighbor has a higher intensity, the neighbor will be assigned a value of 1; otherwise, it will be assigned a value of 0. Then, following a clockwise direction, a binary code as well as the corresponding decimal number can be generated according to these 0s and 1s. Then the histogram of these decimal numbers can be collected as our features. Depending on the radius, a different number of neighbors will be compared with, thus resulting in a different number of possible decimal numbers. For example, with a radius equal to 1, there are $8 \times 1$ neighbors and $2^8 = 256$ possible decimal numbers.

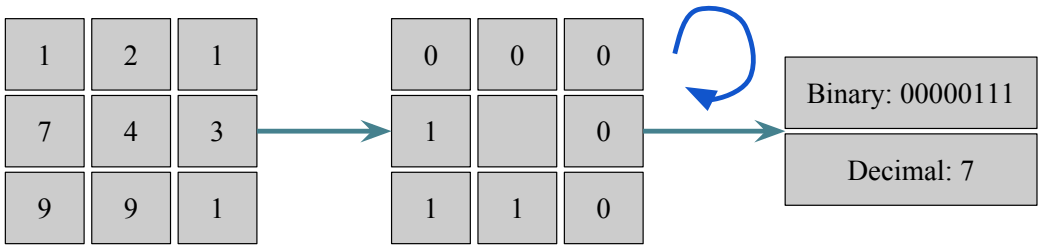


Figure 3.4: Procedures that local binary pattern applied for calculating the local feature of each pixel.

### Feature Preparation

Nataraj et al. [8, 9] have proposed a method of analyzing malwares with image processing techniques and shown the result of which is comparable with dynamic analysis in terms of accuracy. In this work, we decided to convert every BYTE file into an image feature using the local binary pattern method. In order to convert a binary file into a gray-scale image file, one has to decide the width of the image. According to ref. [7], there are several good choices of the image width depending on the file size. Therefore, we chose 512 and 1024 bytes respectively as the image widths to calculate the local binary pattern and the resulting histogram. One example can be seen in Fig. 3.5.

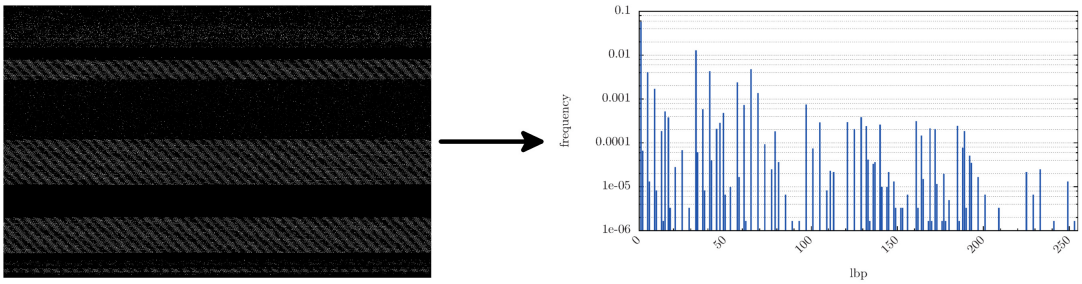


Figure 3.5: An example of converting gray-scale image file into local binary pattern histogram.

# Chapter 4

# Experiments and Results

## 4.1 Experiment Setup

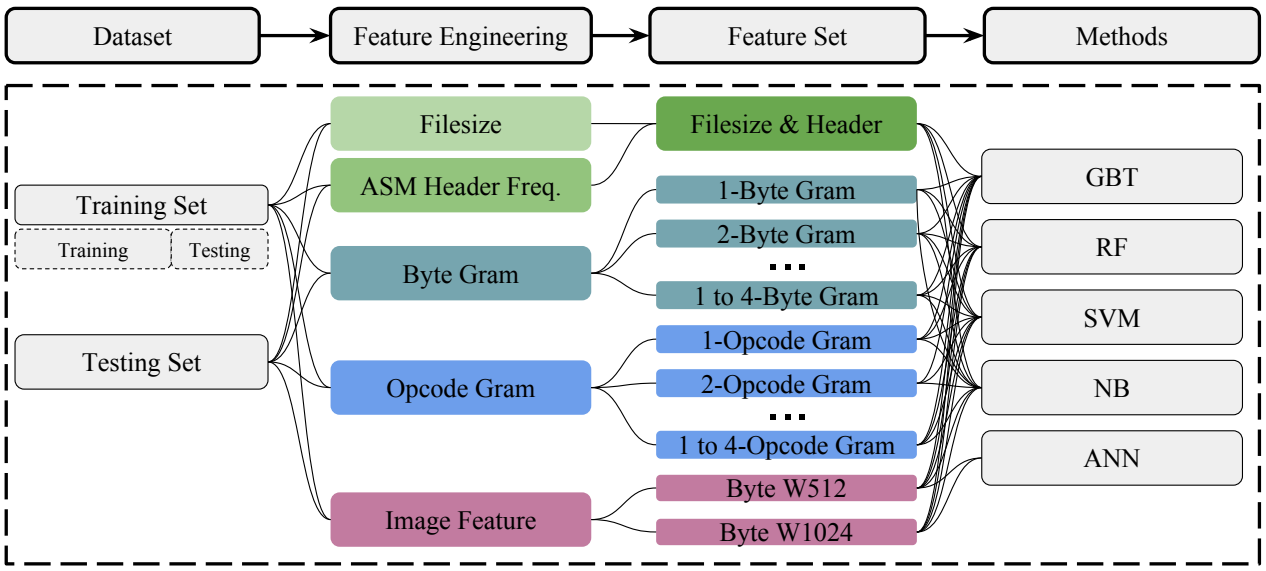


Figure 4.1: Illustration of experiment setup.

### 4.1.1 Local Training Set, Local Testing Set, Kaggle Testing Set

There are two sets of data provided by the Kaggle website: the training set and the testing set. The number of malware files are respectively 10868 and 10873 in these two dataset, with total size $\sim 400G$. The feature engineering procedures described in Chapter 3 are performed on both datasets. We will refer to the testing dataset as Kaggle Testing set, since the labels in this set are not provided and thus the results can only be checked by submitting to the website. Moreover, we split the training set into two different sets with an 80/20 split. We will refer to them as Local Training set and Local Testing set respectively.

### 4.1.2 Feature Selection

Due to the large number of features generated through our feature engineering procedures, we had to reduce the dimensionality of the feature space to reduce the training time, improve the accuracy and prevent over-fitting. The random forest method is used for this purpose in our study. In the implementation of random forest in scikit-learn, it provides the measure of feature importance with the idea that the feature on the top of a tree is relatively more important than its descendants. We only kept features that have importance scores higher than the mean value across all the features.

### 4.1.3 Software and Packages

There are a lot of well-developed open source software in the field of machine learning. In this work, we have utilized two very famous software: scikit-learn [10] and xgboost [1]. As listed in Table 4.1, we adopted the implementation of support vector machines, naive Bayes, artificial neural networks and random forests from scikit-learn and that of gradient boosted tree from the Xgboost package. To optimize the performance with the best parameters as well as prevent the overfitting, we have employed a method named GridsearchCV provided

by scikit-learn. With this method, we used 5-fold cross-validation for the search for the best parameters.

Table 4.1: Methods and packages that are employed in this study.

| Method | Package | Implementation |
|---|---|---|
| Support Vector Machine(SVM) | scikit-learn | sklearn.svm.SVC |
| Naive Bayes (NB) | scikit-learn | sklearn.naive_bayes.MultinomialNB |
| Artificial Neural Network (ANN) | scikit-learn | sklearn.neural_network.MLPClassifier |
| Random Forest (RF) | scikit-learn | sklearn.ensemble.RandomForestClassifier |
| Gradient Boosted Tree (GBT) | xgboost | XGBClassifier* |

*XGBClassifier is a wrapper of xgboost, in order to take advantage of the model selection methods provided by scikit-learn.

## 4.2 Results and Discussion

### 4.2.1 Feature: Only file size and Header string

Table 4.2: Results of different methods on model that consist of file sizes and header string frequency.

| Methods | Local Test | | | | Kaggle Test | |
|---|---|---|---|---|---|---|
| | logloss | precision | recall | f1-score | log loss | rank |
| NB | 1.36068 | 0.44045 | 0.43811 | 0.3495 | 1.358348074 | 289 |
| SVM | 0.59515 | 0.79959 | 0.71695 | 0.7402 | 0.584892249 | 272 |
| RF | 0.04954 | 0.99068 | 0.99066 | 0.9905 | 0.046072858 | 147 |
| GBT | 0.05442 | 0.99009 | 0.99019 | 0.9900 | 0.040359325 | 127 |

The first set of features we investigated are normalized file sizes and header strings frequency as described in Sec. 3.1 and 3.2. We applied four different methods: naive bayes (NB), support vector machine (SVM), random forest (RF) and gradient boosted tree (GBT) on this set of features, the results of which are shown in Table 4.2. Both RF and GBT perform fairly well in terms of both the local test and the Kaggle test. Even though RF

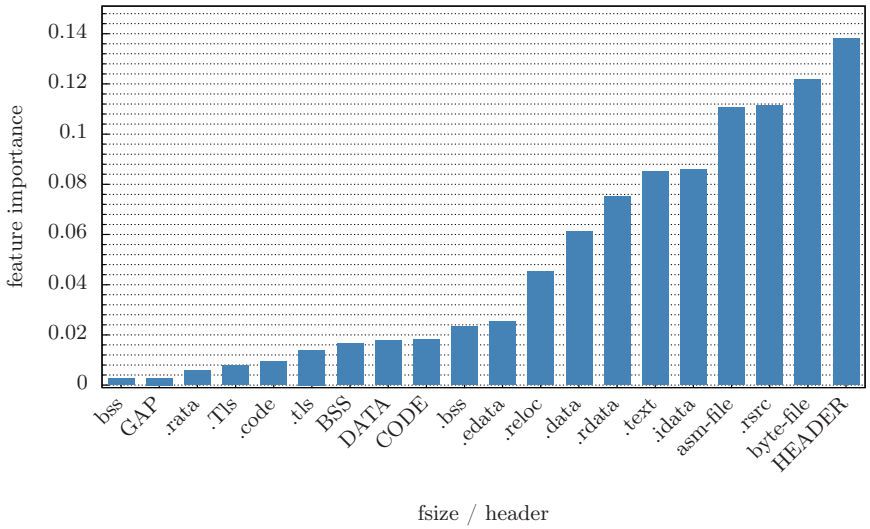


Figure 4.2: Feature importance for file size and ASM header string.

has slightly better scores compared with GBT, it turns out to be not as good as GBT on the Kaggle test. We present the top 20 features in terms of their importance scores. As we expected, both byte file size and asm file size turn out to be top features and some top 10 header string sections in Table 3.1 are very important features as well.

### 4.2.2 Feature: Only Byte Gram

In this section, we studied 1-, 2-, 3-, and 4-byte gram features and their combinations, which resulted in 15 models in total. A summary of the performance of all the methods on different models (or features) can be found in Fig. 4.3. For more details, we listed the results of four different methods: GBT, RF, SVM and NB, in Table 4.3, 4.4, 4.5 and 4.6 respectively. We found that different methods actually value features differently. The gradient boosted tree (GBT), which gave the best performance in terms of the Kaggle test in this section, achieved the best testing score with the model consisting of 1-, 3- and 4-byte gram. The random forest method works well with the same model but with worse performance. As for support vector

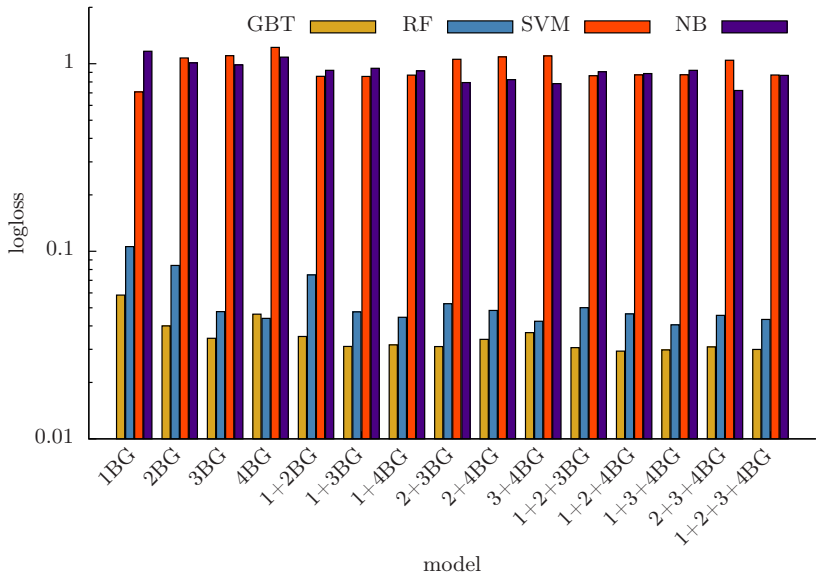


Figure 4.3: Results of 4 different methods applied on 15 different models. Vertical axis, in logscale, shows the logloss (smaller the better); while the horizontal axis shows the abbreviation of different models. BG stands for Byte Gram.

machine (SVM) and naive bayes (NB), the former scores the best with simply the 1-byte gram, while the latter works well with 2-, 3- and 4-byte gram models.

Moreover, to better understand the importance of each feature, we plotted the feature importance for the top 20 1-, 2-, 3- and 4-byte gram respectively in Fig. 4.4, 4.5, 4.6 and 4.7. As one can see, for 1- and 2-byte gram, there are some features that have much higher importance scores compared to others. For instance, in 1-byte gram, byte sequence '00' has almost twice the importance score of the second highest 'FF'; and in 2-byte gram, byte sequence '1000' has roughly 1.5 times the importance score of the second highest '008D'. However, for 3- and 4-byte gram, the differences are much smaller. Thus, one would expect that with even higher orders of byte-gram, the differences will be reduced further.

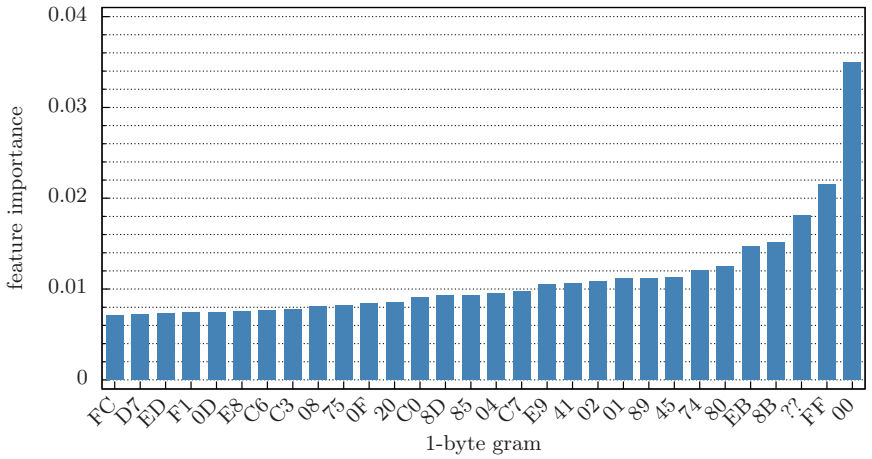


Figure 4.4: Feature importance score for top 20 1-byte gram.

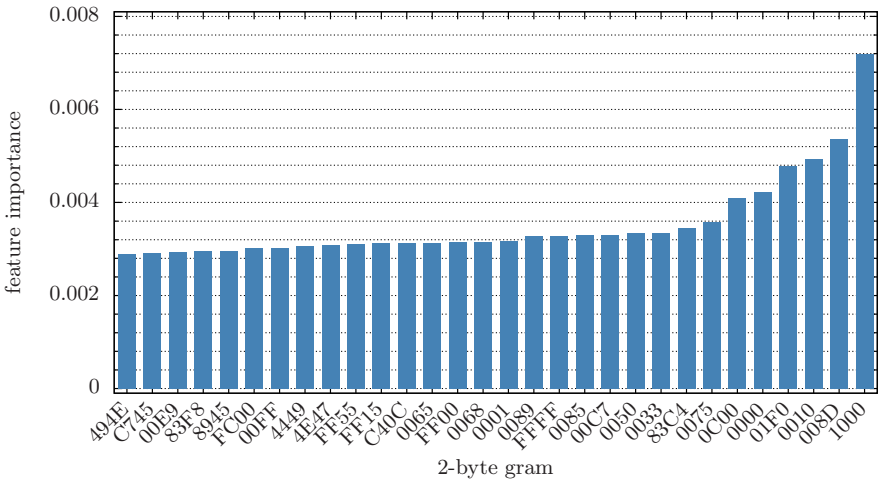


Figure 4.5: Feature importance score for top 20 2-byte gram.

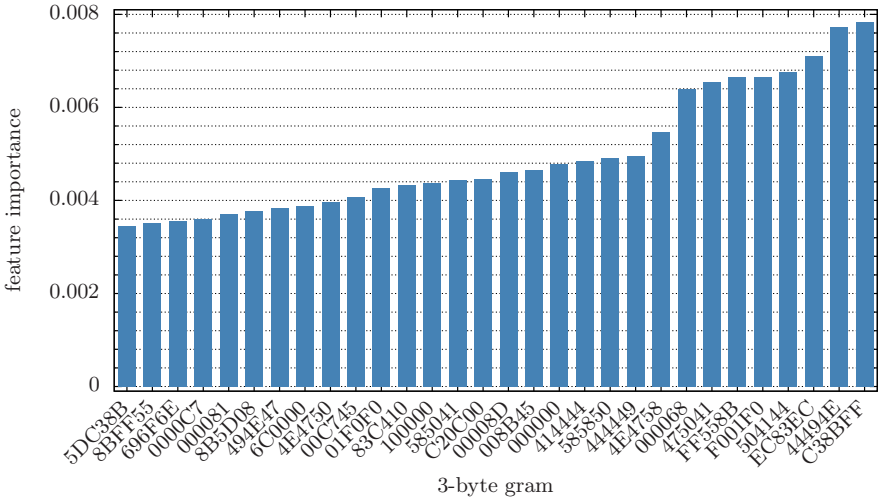


Figure 4.6: Feature importance score for top 20 3-byte gram.

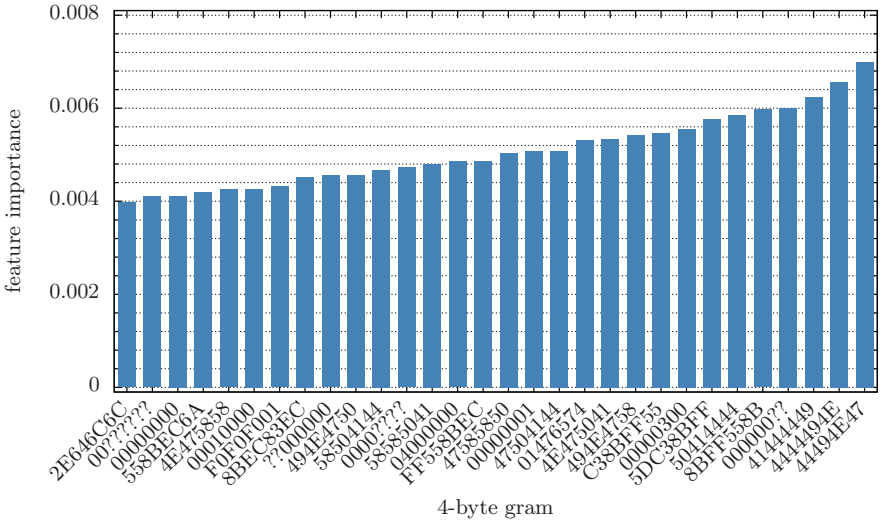


Figure 4.7: Feature importance score for top 20 4-byte gram.

Table 4.3: Results of Gradient Boosted Tree (GBT) on models that are composed of n byte gram features. The best one in terms of Kaggle test is labeled by red color.

| Features | Local Test | | | | Kaggle Test | |
|---|---|---|---|---|---|---|
| | logloss | precision | recall | f1-score | log loss | rank |
| 1-byte gram | 0.06794 | 0.98513 | 0.98505 | 0.98485 | 0.058427869 | 170 |
| 2-byte gram | 0.04349 | 0.98780 | 0.98783 | 0.98774 | 0.039999557 | 127 |
| 3-byte gram | 0.04154 | 0.99055 | 0.99069 | 0.99055 | 0.034396248 | 115 |
| 4-byte gram | 0.05211 | 0.98482 | 0.98468 | 0.98466 | 0.046222758 | 147 |
| 1- and 2-byte gram | 0.04208 | 0.98836 | 0.98837 | 0.98826 | 0.035161211 | 118 |
| 1- and 3-byte gram | 0.04143 | 0.99098 | 0.99093 | 0.99093 | 0.031085531 | 107 |
| 1- and 4-byte gram | 0.03843 | 0.98726 | 0.98720 | 0.98721 | 0.031712716 | 108 |
| 2- and 3-byte gram | 0.03551 | 0.98964 | 0.98958 | 0.98953 | 0.031070368 | 107 |
| 2- and 4-byte gram | 0.04097 | 0.99178 | 0.99180 | 0.99171 | 0.033900608 | 114 |
| 3- and 4-byte gram | 0.04409 | 0.98809 | 0.98828 | 0.98791 | 0.036863338 | 121 |
| 1-, 2- and 3-byte gram | 0.03753 | 0.99270 | 0.99262 | 0.99262 | 0.030620258 | 105 |
| 1-, 2- and 4-byte gram | 0.03744 | 0.99219 | 0.99207 | 0.99209 | 0.029774098 | 103 |
| 1-, 3- and 4-byte gram | 0.03652 | 0.99121 | 0.99108 | 0.99091 | 0.029349611 | 103 |
| 2-, 3- and 4-byte gram | 0.03631 | 0.99195 | 0.99180 | 0.99182 | 0.030929875 | 107 |
| 1-, 2-, 3- and 4-byte gram | 0.03377 | 0.99174 | 0.99171 | 0.99166 | 0.029993074 | 103 |

Table 4.4: Results of random forest (RF) on models that are composed of n byte gram features. The best one in terms of Kaggle test is labeled by red color.

| Features | Local Test | | | | Kaggle Test | |
|---|---|---|---|---|---|---|
| | logloss | precision | recall | f1-score | log loss | rank |
| 1-byte gram | 0.12392 | 0.97873 | 0.97851 | 0.97783 | 0.106005521 | 230 |
| 2-byte gram | 0.09163 | 0.98789 | 0.98783 | 0.98778 | 0.084014007 | 212 |
| 3-byte gram | 0.05562 | 0.99171 | 0.99162 | 0.99140 | 0.047645314 | 149 |
| 4-byte gram | 0.05255 | 0.99428 | 0.99414 | 0.99416 | 0.043920403 | 142 |
| 1- and 2-byte gram | 0.08731 | 0.98747 | 0.98744 | 0.98724 | 0.074824771 | 188 |
| 1- and 3-byte gram | 0.05808 | 0.99286 | 0.99275 | 0.99276 | 0.047498616 | 149 |
| 1- and 4-byte gram | 0.05048 | 0.99319 | 0.99314 | 0.99313 | 0.044474508 | 143 |
| 2- and 3-byte gram | 0.06027 | 0.99253 | 0.99242 | 0.99242 | 0.052519540 | 160 |
| 2- and 4-byte gram | 0.05666 | 0.99416 | 0.99408 | 0.99389 | 0.048351297 | 152 |
| 3- and 4-byte gram | 0.05197 | 0.99585 | 0.99578 | 0.99580 | 0.042399837 | 135 |
| 1-, 2- and 3-byte gram | 0.06068 | 0.99402 | 0.99400 | 0.99398 | 0.050079276 | 154 |
| 1-, 2- and 4-byte gram | 0.05488 | 0.99304 | 0.99300 | 0.99300 | 0.046337592 | 147 |
| 1-, 3- and 4-byte gram | 0.04950 | 0.99253 | 0.99242 | 0.99236 | 0.040534530 | 128 |
| 2-, 3- and 4-byte gram | 0.05388 | 0.99417 | 0.99408 | 0.99404 | 0.045571748 | 145 |
| 1-, 2-, 3- and 4-byte gram | 0.05117 | 0.99499 | 0.99494 | 0.99493 | 0.043304202 | 139 |

Table 4.5: Results of support vector machine (SVM) on models that are composed of n byte gram features. The best one in terms of Kaggle test is labeled by red color.

| Features | Local Test | | | | Kaggle Test | |
|---|---|---|---|---|---|---|
| | logloss | precision | recall | f1-score | log loss | rank |
| 1-byte gram | 0.73383 | 0.74036 | 0.73657 | 0.70617 | 0.707840305 | 278 |
| 2-byte gram | 1.09163 | 0.27327 | 0.28932 | 0.24875 | 1.073261311 | 284 |
| 3-byte gram | 1.12277 | 0.32287 | 0.24849 | 0.25538 | 1.104820332 | 284 |
| 4-byte gram | 1.23142 | 0.40438 | 0.39279 | 0.30210 | 1.223500783 | 286 |
| 1- and 2-byte gram | 0.88550 | 0.39289 | 0.35023 | 0.33599 | 0.857133719 | 282 |
| 1- and 3-byte gram | 0.87787 | 0.57371 | 0.64279 | 0.58441 | 0.855480488 | 282 |
| 1- and 4-byte gram | 0.89377 | 0.58807 | 0.61454 | 0.55302 | 0.870126597 | 282 |
| 2- and 3-byte gram | 1.07242 | 0.46107 | 0.51729 | 0.45226 | 1.056115162 | 284 |
| 2- and 4-byte gram | 1.12241 | 0.34084 | 0.19627 | 0.18904 | 1.087907689 | 284 |
| 3- and 4-byte gram | 1.13259 | 0.46312 | 0.49438 | 0.42576 | 1.101415286 | 284 |
| 1-, 2- and 3-byte gram | 0.88585 | 0.37780 | 0.35533 | 0.34494 | 0.863722938 | 282 |
| 1-, 2- and 4-byte gram | 0.88832 | 0.51582 | 0.59049 | 0.53020 | 0.872646176 | 282 |
| 1-, 3- and 4-byte gram | 0.89918 | 0.52851 | 0.60856 | 0.54777 | 0.873518805 | 282 |
| 2-, 3- and 4-byte gram | 1.08138 | 0.33676 | 0.27699 | 0.27091 | 1.043608708 | 284 |
| 1-, 2-, 3- and 4-byte gram | 0.89801 | 0.52552 | 0.60820 | 0.54502 | 0.871688051 | 282 |

Table 4.6: Results of naive bayes (NB) on models that are composed of n byte gram features. The best one in terms of Kaggle test is labeled by red color.

| Features | Local Test | | | | Kaggle Test | |
|---|---|---|---|---|---|---|
| | logloss | precision | recall | f1-score | log loss | rank |
| 1-byte gram | 1.18588 | 0.66940 | 0.68239 | 0.62515 | 1.166548664 | 284 |
| 2-byte gram | 1.00010 | 0.76547 | 0.78909 | 0.74580 | 1.011161162 | 283 |
| 3-byte gram | 1.01351 | 0.74098 | 0.76966 | 0.72861 | 0.986500474 | 283 |
| 4-byte gram | 1.10197 | 0.71679 | 0.74820 | 0.70366 | 1.085435922 | 284 |
| 1- and 2-byte gram | 0.94656 | 0.79169 | 0.79302 | 0.76103 | 0.922949247 | 282 |
| 1- and 3-byte gram | 0.98398 | 0.73079 | 0.77244 | 0.73857 | 0.946556419 | 283 |
| 1- and 4-byte gram | 0.94308 | 0.77674 | 0.78738 | 0.75319 | 0.915573028 | 282 |
| 2- and 3-byte gram | 0.81452 | 0.75240 | 0.78778 | 0.75244 | 0.794525054 | 281 |
| 2- and 4-byte gram | 0.83145 | 0.78259 | 0.81603 | 0.78079 | 0.823162065 | 281 |
| 3- and 4-byte gram | 0.81777 | 0.74480 | 0.78585 | 0.74924 | 0.782990251 | 281 |
| 1-, 2- and 3-byte gram | 0.95900 | 0.79109 | 0.77204 | 0.73700 | 0.906776792 | 282 |
| 1-, 2- and 4-byte gram | 0.90697 | 0.78763 | 0.79571 | 0.76086 | 0.886961809 | 282 |
| 1-, 3- and 4-byte gram | 0.97343 | 0.77026 | 0.79046 | 0.75941 | 0.922360539 | 282 |
| 2-, 3- and 4-byte gram | 0.74160 | 0.80318 | 0.79636 | 0.76444 | 0.720803265 | 278 |
| 1-, 2-, 3- and 4-byte gram | 0.92980 | 0.78404 | 0.79282 | 0.76048 | 0.866792452 | 282 |

### 4.2.3 Feature: Only Opcode Gram

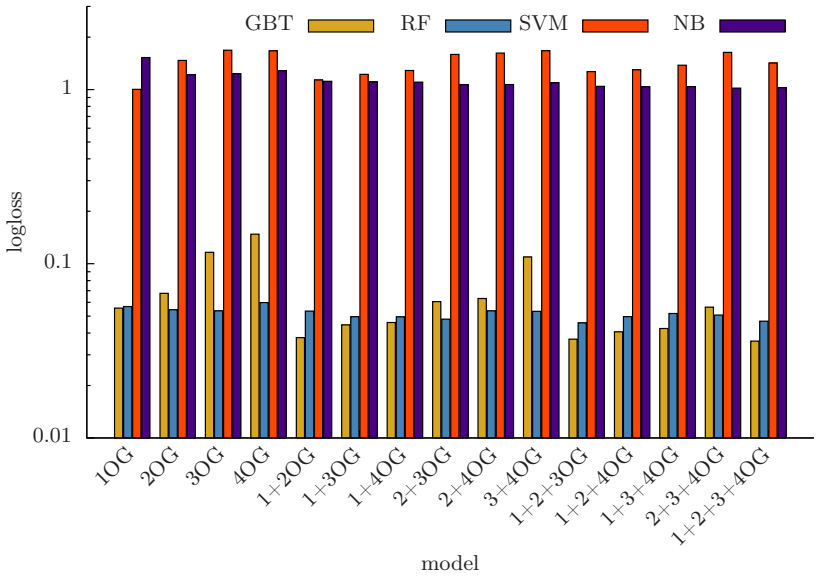


Figure 4.8: Results of 4 different methods applied on 15 different models. Vertical axis, in logscale, shows the logloss (smaller the better); while the horizontal axis shows the abbreviation of different models. OG stands for Opcode Gram.

In this section, we studied 1-, 2-, 3-, and 4-opcode gram features and their combinations, which result in 15 models in total. A summary of the performance of all the methods on different models (or features) can be found in Fig. 4.8. For more detail, we listed the results of four different methods, GBT, RF, SVM and NB, in Table 4.7, 4.8, 4.9 and 4.10 respectively. Similar to the previous section, the gradient boosted tree (GBT) scores the best in combination with the model consisting of 1-, 2-, 3- and 4-opcode gram sequences. The other three methods also favor different combinations of opcode gram features.

Moreover, for a better understanding of the importance of each feature, we plotted the feature importance for the top 20 1-, 2-, 3- and 4-opcode gram respectively in Fig. 4.9, 4.10, 4.11 and 4.12. Different from byte-gram sequences, opcode-gram sequences do not show much difference in feature importance score for 1-opcode gram. However for higher orders of opcode-gram sequences, some combinations start to become more important. For

example, in 2-opcode gram, ‘call,add’,‘add, pop’ and ‘mov, pop’ seem to be more significant than others, while in 3-opcode gram sequences, ‘mov, sub, mov’ seems to become the most influential.

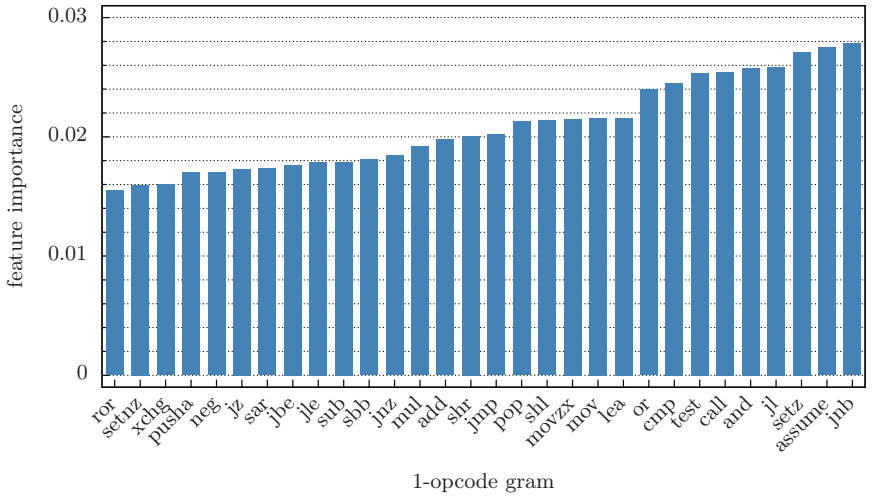


Figure 4.9: Feature importance for 1 opcode gram.

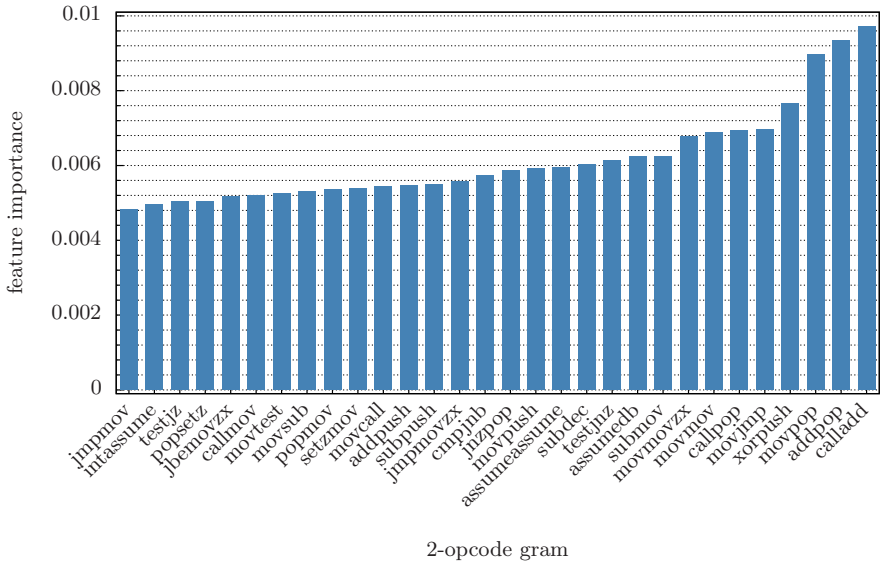


Figure 4.10: Feature importance for 2 opcode gram.

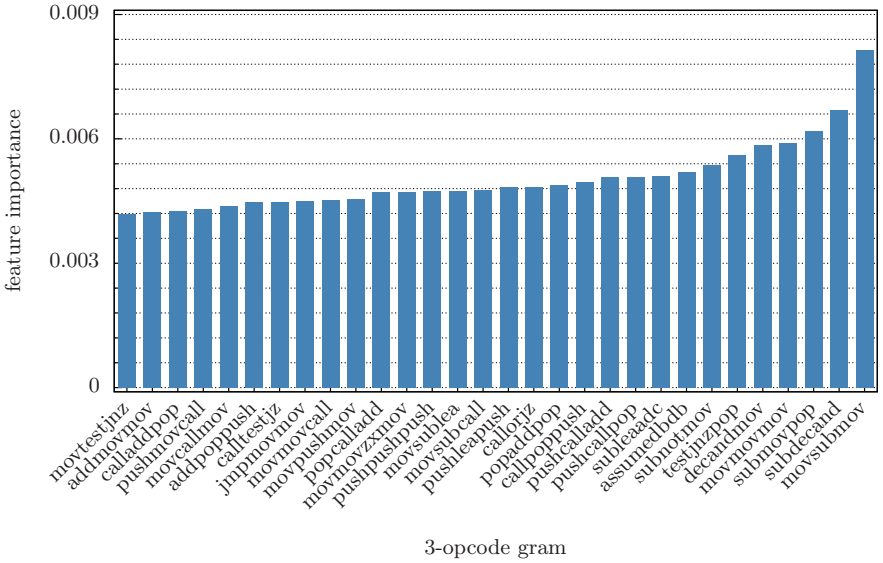


Figure 4.11: Feature importance for 3 opcode gram.

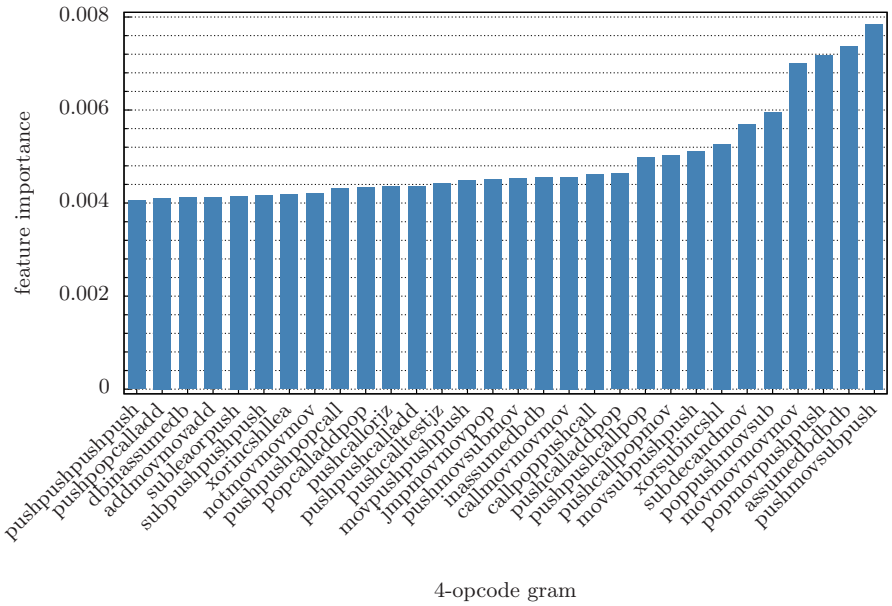


Figure 4.12: Feature importance for 4 opcode gram.

Table 4.7: Results of Gradient Boosted Tree on opcode n-gram features

| Features | Local Test | | | | Kaggle Test | |
|---|---|---|---|---|---|---|
| | logloss | precision | recall | f1-score | log loss | rank |
| 1-opcode gram | 0.06627 | 0.98437 | 0.98407 | 0.98399 | 0.055676499 | 164 |
| 2-opcode gram | 0.06610 | 0.98226 | 0.98199 | 0.98186 | 0.067582810 | 183 |
| 3-opcode gram | 0.12521 | 0.97084 | 0.96951 | 0.96920 | 0.116286780 | 232 |
| 4-opcode gram | 0.16604 | 0.96648 | 0.96429 | 0.96375 | 0.147684057 | 237 |
| 1- and 2-opcode gram | 0.05143 | 0.99045 | 0.99029 | 0.99024 | 0.037665565 | 121 |
| 1- and 3-opcode gram | 0.05157 | 0.98698 | 0.98670 | 0.98664 | 0.044563968 | 143 |
| 1- and 4-opcode gram | 0.05573 | 0.98821 | 0.98797 | 0.98799 | 0.046022506 | 147 |
| 2- and 3-opcode gram | 0.06531 | 0.98375 | 0.98352 | 0.98321 | 0.060587208 | 174 |
| 2- and 4-opcode gram | 0.06406 | 0.98307 | 0.98300 | 0.98268 | 0.063167623 | 177 |
| 3- and 4-opcode gram | 0.11966 | 0.96648 | 0.96270 | 0.96302 | 0.109332554 | 231 |
| 1-, 2- and 3-opcode gram | 0.05272 | 0.99077 | 0.99086 | 0.99074 | 0.036911443 | 121 |
| 1-, 2- and 4-opcode gram | 0.04518 | 0.98855 | 0.98833 | 0.98831 | 0.040666626 | 128 |
| 1-, 3- and 4-opcode gram | 0.05285 | 0.98964 | 0.98945 | 0.98944 | 0.042508895 | 135 |
| 2-, 3- and 4-opcode gram | 0.06593 | 0.98648 | 0.98612 | 0.98598 | 0.056368037 | 166 |
| 1-, 2-, 3- and 4-opcode gram | 0.04851 | 0.98769 | 0.98755 | 0.98754 | 0.035978248 | 120 |

Table 4.8: Results of Random Forest on opcode n-gram features

| Features | Local Test | | | | Kaggle Test | |
|---|---|---|---|---|---|---|
| | logloss | precision | recall | f1-score | log loss | rank |
| 1-opcode gram | 0.06877 | 0.99366 | 0.99363 | 0.99363 | 0.056837769 | 166 |
| 2-opcode gram | 0.06247 | 0.99182 | 0.99169 | 0.99171 | 0.054402368 | 163 |
| 3-opcode gram | 0.06906 | 0.99015 | 0.99000 | 0.99001 | 0.053657352 | 163 |
| 4-opcode gram | 0.07101 | 0.98935 | 0.98901 | 0.98903 | 0.059830554 | 172 |
| 1- and 2-opcode gram | 0.06229 | 0.99679 | 0.99676 | 0.99676 | 0.053455132 | 162 |
| 1- and 3-opcode gram | 0.06142 | 0.98960 | 0.98936 | 0.98939 | 0.049657314 | 154 |
| 1- and 4-opcode gram | 0.05982 | 0.98903 | 0.98886 | 0.98885 | 0.049557661 | 154 |
| 2- and 3-opcode gram | 0.05892 | 0.99140 | 0.99130 | 0.99130 | 0.048065831 | 151 |
| 2- and 4-opcode gram | 0.06111 | 0.99012 | 0.98989 | 0.98988 | 0.053646641 | 163 |
| 3- and 4-opcode gram | 0.06743 | 0.99049 | 0.99021 | 0.99024 | 0.053333982 | 161 |
| 1-, 2- and 3-opcode gram | 0.05808 | 0.99636 | 0.99635 | 0.99635 | 0.045790234 | 146 |
| 1-, 2- and 4-opcode gram | 0.05804 | 0.99260 | 0.99253 | 0.99254 | 0.049671613 | 154 |
| 1-, 3- and 4-opcode gram | 0.06271 | 0.99113 | 0.99083 | 0.99084 | 0.051831562 | 158 |
| 2-, 3- and 4-opcode gram | 0.06515 | 0.99269 | 0.99260 | 0.99257 | 0.050746933 | 154 |
| 1-, 2-, 3- and 4-opcode gram | 0.05870 | 0.99037 | 0.99031 | 0.99027 | 0.046784063 | 147 |

Table 4.9: Results of Support Vector Machine on opcode n-gram features

| Features | Local Test | | | | Kaggle Test | |
|---|---|---|---|---|---|---|
| | logloss | precision | recall | f1-score | log loss | rank |
| 1-opcode gram | 1.03205 | 0.86703 | 0.46427 | 0.52510 | 1.002435208 | 283 |
| 2-opcode gram | 1.52349 | 0.66821 | 0.09880 | 0.13224 | 1.468853733 | 290 |
| 3-opcode gram | 1.67486 | 0.14944 | 0.07051 | 0.09107 | 1.681416541 | 293 |
| 4-opcode gram | 1.69577 | 0.12984 | 0.06319 | 0.05275 | 1.668727274 | 193 |
| 1- and 2-opcode gram | 1.15368 | 0.75163 | 0.32609 | 0.40411 | 1.133881193 | 284 |
| 1- and 3-opcode gram | 1.31135 | 0.76577 | 0.28426 | 0.33652 | 1.221506540 | 286 |
| 1- and 4-opcode gram | 1.37267 | 0.62264 | 0.19251 | 0.20630 | 1.286477183 | 286 |
| 2- and 3-opcode gram | 1.63072 | 0.65938 | 0.08837 | 0.12453 | 1.591120403 | 292 |
| 2- and 4-opcode gram | 1.64494 | 0.29582 | 0.10473 | 0.14512 | 1.619302549 | 293 |
| 3- and 4-opcode gram | 1.66462 | 0.16743 | 0.24242 | 0.13100 | 1.671433106 | 293 |
| 1-, 2- and 3-opcode gram | 1.34362 | 0.67252 | 0.21151 | 0.24041 | 1.268720875 | 286 |
| 1-, 2- and 4-opcode gram | 1.41626 | 0.60037 | 0.17545 | 0.23020 | 1.298026947 | 286 |
| 1-, 3- and 4-opcode gram | 1.50896 | 0.64219 | 0.20220 | 0.20668 | 1.378100275 | 289 |
| 2-, 3- and 4-opcode gram | 1.65822 | 0.12220 | 0.10828 | 0.08842 | 1.633988533 | 293 |
| 1-, 2-, 3- and 4-opcode gram | 1.51366 | 0.55955 | 0.17574 | 0.18289 | 1.418311336 | 289 |

Table 4.10: Results of Naive Bayes on opcode n-gram features

| Features | Local Test | | | | Kaggle Test | |
|---|---|---|---|---|---|---|
| | logloss | precision | recall | f1-score | log loss | rank |
| 1-opcode gram | 1.52602 | 0.54456 | 0.48566 | 0.40232 | 1.526730089 | 291 |
| 2-opcode gram | 1.22171 | 0.61043 | 0.64081 | 0.58644 | 1.217361408 | 286 |
| 3-opcode gram | 1.23700 | 0.66681 | 0.66222 | 0.62262 | 1.232014974 | 286 |
| 4-opcode gram | 1.30268 | 0.67899 | 0.63782 | 0.59650 | 1.282586933 | 286 |
| 1- and 2-opcode gram | 1.11560 | 0.70631 | 0.68548 | 0.63450 | 1.115517048 | 284 |
| 1- and 3-opcode gram | 1.10451 | 0.68594 | 0.67361 | 0.62970 | 1.104415665 | 284 |
| 1- and 4-opcode gram | 1.10789 | 0.65694 | 0.66355 | 0.61836 | 1.102746148 | 284 |
| 2- and 3-opcode gram | 1.08201 | 0.72531 | 0.67720 | 0.63662 | 1.068000043 | 284 |
| 2- and 4-opcode gram | 1.08961 | 0.72253 | 0.67065 | 0.62790 | 1.067556825 | 284 |
| 3- and 4-opcode gram | 1.11026 | 0.69884 | 0.64895 | 0.61161 | 1.094049439 | 284 |
| 1-, 2- and 3-opcode gram | 1.06745 | 0.73117 | 0.70032 | 0.66509 | 1.042596323 | 284 |
| 1-, 2- and 4-opcode gram | 1.05196 | 0.70993 | 0.68222 | 0.64819 | 1.037098949 | 284 |
| 1-, 3- and 4-opcode gram | 1.04230 | 0.75490 | 0.67721 | 0.63499 | 1.040635579 | 284 |
| 2-, 3- and 4-opcode gram | 1.02428 | 0.76622 | 0.69736 | 0.66503 | 1.019241233 | 283 |
| 1-, 2-, 3- and 4-opcode gram | 1.05560 | 0.76810 | 0.69926 | 0.67095 | 1.025521649 | 283 |

### 4.2.4 Feature: Only Image Features

Table 4.11: Results of five different methods on two image features with two different image widths: 512 and 1024 bytes. NB: naive bayes; SVM: support vector machine; RF: random forest; GBT: gradient boosted tree; ANN: artificial neural network.

| Features | Methods | Local Test | | | | Kaggle Test | |
|---|---|---|---|---|---|---|---|
| | | logloss | precision | recall | f1-score | log loss | rank |
| byte W512 image | NB | 1.89045 | 0.06812 | 0.26083 | 0.10803 | 1.894463592 | 295 |
| | SVM | 1.89524 | 0.00996 | 0.09973 | 0.01811 | 1.897860161 | 295 |
| | RF | 0.26894 | 0.94195 | 0.94043 | 0.94040 | 0.276092119 | 259 |
| | GBT | 0.29030 | 0.91883 | 0.91832 | 0.91783 | 0.262252929 | 258 |
| | ANN | 0.70867 | 0.76644 | 0.78926 | 0.76852 | 0.677654263 | 278 |
| byte W1024 image | NB | 1.89270 | 0.06352 | 0.25203 | 0.10147 | 1.894214482 | 295 |
| | SVM | 1.89813 | 0.02303 | 0.15176 | 0.03999 | 1.900646913 | 306 |
| | RF | 0.26578 | 0.93907 | 0.93631 | 0.93571 | 0.243892362 | 258 |
| | GBT | 0.26650 | 0.92578 | 0.92412 | 0.92344 | 0.237519064 | 258 |
| | ANN | 0.72745 | 0.77370 | 0.76694 | 0.74997 | 0.680402376 | 278 |

In this section, we investigated the image features for our classification task. Two different image widths, 512 and 1024 bytes, are chosen for converting BYTE files into gray-scale images. Different from previous sections, along with the naive bayes, support vector machine, random forest and gradient boosted tree, we also employed artificial neural network (ANN), which is known to be successful in image processing tasks. For ANN, due to the limitation of computing resources, only three hidden-layers and four hidden-layers architectures were used, with a various number of nodes for each layer. All the results are shown in Table 4.11. From this table, one can tell that the image features do not work well in this particular classification task. The best performance is gradient boosted tree with image width 1024 bytes. However, we see that, in combining with other features, image feature does improve our performance significantly.

### 4.2.5 Features: Mixture and Exploration

In previous sections, we found that the gradient boosted tree outperforms all of the other methods in every feature set. Therefore, in this section, we decided to focus on this method and applied it to different combinations of different feature sets, in hope that the performance of our models could be greatly enhanced. As shown in Table 4.12, we actually see the improvement by adding in more features to our model. For example, we start with the model consisting of 1-, 3- and 4-byte gram, and 1-, 2-, 3- and 4-opcode gram sequences, the result of which turns out to be much better than any of the single feature sets in previous sections. Moreover, by including the file size and header string frequency, the rank of our result jumps from 80 to 60, with the reduction of $\sim 0.005$ in logloss. Surprisingly, even though the image features did not work well as an individual feature set, we found it actually reduced the logloss when combined with other feature sets. Our best result so far had logloss of 0.0146 which ranked 53rd in the Kaggle private leaderboard.

Table 4.12: Results of Gradient Boosted Tree on different combined features. BG stands for byte gram and OG represents opcode gram.

| Features | Local Test | | | | Kaggle Test | |
|---|---|---|---|---|---|---|
| | logloss | precision | recall | f1-score | log loss | rank |
| 1-4 BG, 1-4 OG | 0.02546 | 0.99536 | 0.99530 | 0.99529 | 0.020328240 | 78 |
| above + file size and header string | 0.02001 | 0.99189 | 0.99178 | 0.99161 | 0.020171480 | 78 |
| above + byte W1024 image | 0.02185 | 0.99320 | 0.99305 | 0.99298 | 0.015425664 | 59 |
| 1,3,4 BG, 1-4 OG | 0.02736 | 0.99504 | 0.99497 | 0.99499 | 0.021093468 | 80 |
| above + file size and header string | 0.02469 | 0.99405 | 0.99388 | 0.99392 | 0.015750215 | 60 |
| above + byte W1024 image | 0.02243 | 0.99385 | 0.99379 | 0.99373 | 0.014611916 | 53 |

### 4.2.6 Further thoughts on feature selection

Since there are different types of features being generated in this work, one can use deep learning techniques as a way of performing feature selection. For this purpose, we have performed some preliminary work using three-layer autoencoder networks. An autoencoder is one type of neural network used to learn a representation (encoding) for a set of data. Typically, an autoencoder can be used for the purpose of dimensionality reduction. We have combined all of the byte-gram features (1, 2, 3 and 4-byte gram) into a feature set, each record of which contains 17984 fields. Two different three-layer autoencoders are applied on this combined feature set, with 100 and 1000 hidden nodes respectively. For each case, after training, the values of the nodes in the hidden layer are selected as the final features for our classification task. When compared with the result of using random forests as the feature selection tool (rank 103 as seen in Table 4.3), the best result of this method only obtained a rank of 183 using the same classification method. Given the fact that we have only applied very simple autoencoders, one would expect better results when an autoencoder with multiple layers is considered. However, in this case, tremendous computing resources are required for training such networks.

# Chapter 5

# Conclusion

In this work, we applied multiple advanced machine learning algorithms including random forests, boosted trees, and support vector machines to investigate the problem of malware file classification. We performed different feature engineering procedures on a large dataset ($\sim 400G$) of malware files provided by Kaggle.com, and obtained four feature sets: filesizes and header string frequency, byte-sequence n-grams, opcode n-grams and image features.

First, we applied each machine learning method to each of these feature sets. As expected, both byte filesize and asm filesize turned out to be pretty good features in malware file classification. For byte-sequence n-grams, we generated 1, 2, 3 and 4-byte grams, all the combinations of which are examined and discussed. We found that the combination of 1,3 and 4-gram combined with GBT gives the best result with logloss of 0.0293. Similarly, for opcode n-grams, we examined all different combinations, and found that the combination of 1, 2, 3 and 4-gram with GBT performs the best with logloss of 0.0360. For the image feature, we first converted each malware file into gray-scale image, and then applied the local binary pattern to generate the histogram features. However, all the methods we applied on this feature set did not give good results compared to previous ones.

In the next step, we combined different feature sets, in hope that the overall performance

would be boosted. The best combination we found contained 1, 3 and 4-byte grams, 1, 2, 3 and 4-opcode grams, filesizes and header strings, and image features. This combined feature set, with GBT, gave the best performance overall with logloss of 0.0146. To our surprise, even though the image feature set does not work well alone, it does improve the overall performance when combined with other features.

# Bibliography


[1] Chen, T., and Guestrin, C. Xgboost: A scalable tree boosting system. *CoRR abs/1603.02754* (2016).

[2] chen He, D., and Wang, L. Texture unit, texture spectrum, and texture analysis. *IEEE Transactions on Geoscience and Remote Sensing 28*, 4 (Jul 1990), 509–512.

[3] Cortes, C., and Vapnik, V. Support-vector networks. *Machine Learning*, 20 (1995), 273–297.

[4] Hastie, T., Tibshirani, R., and Friedman, J. *The elements of statistical learning : Data mining, inference, and prediction*. New York: Springer Verlag, 2009.

[5] Kaggle.com. Microsoft malware classification challenge (big 2015), Feb 2015.

[6] Kolter, J., and Maloof, M. Learning to detect malicious executables in the wild. *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2004), 470–478.

[7] Nataraj, L. http://sarvamblog.blogspot.com/, 2014.

[8] Nataraj, L., Karthikeyan, S., Jacob, G., and Manjunath, B. Malware images: Visualization and automatic classification. *Proceedings of the 8th International Symposium on Visualization for Cyber Security 4* (2011).

[9] Nataraj, L., Yegneswaran, V. Porras, P., and Zhang, J. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence* (2011), 21–30.

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