Statistical Evaluation of Malware Classification Algorithms

by

Lu Zhu

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Science Degree

in

Probability and Statistics

Carleton University
Ottawa, Ontario

Abstract

Classifying malware with learning algorithms is common in the information security community. In this thesis, the performance of five learning algorithms on malware classification is evaluated statistically.

The study is based on the malicious file collection released by Microsoft on Kaggle.com where 10K labeled malware instances (250GB) were provided. Following the work of Ahmadi et al (2016b), 1801 features in 13 feature categories were extracted and the volume of extracted data set was reduced to 90MB.

Five learning algorithms were run on the reduced data set and on a standardized data set and evaluated for accuracy and logloss. Statistical analyses using multivariate analysis of variance (MANOVA) and univariate analysis of variance (ANOVA), and graphical tool of interaction plots were employed to assess the performance of the algorithms while controlling for effect of data set used. The analyses showed that XGBoost was the best classification algorithm for accuracy and logloss.

Acknowledgements

I am deeply thankful to my advisor Dr. Shirley E. Mills, for her invaluable guidance, exceptional recommendations, continued encouragement and support all throughout of my research. I have greatly enjoyed working under her supervision.

I would like to express my gratitude to my friend Ying Zhou for inspiration, continuous motivation, and great ideas.

I would like to express my deepest and sincerest thanks to my parents Mr. Daozeng Zhu, Mrs. Jiahui Huang, my father-in-law Mr. Jinshu Yu, and mother-in-law Mrs. Hongru Sun, for their immeasurable love and support in all my endeavors.

Finally, I would like to thank my husband, Sheng Yu for all his patience, constant encouragement, invaluable advice, and taking care of our daughter throughout my studies. He has continuously been supporting all throughout these years.

To my daughter Zhile Yu whose unconditional love moves my heart with happiness and encouragement.

List of Tables

Table 1 Summary for Accuracy Score and Logloss 26

Table 2 Summary for Transformed Accuracy Score 27

Table 3 Representation of i 28

Table 4 Representation of j 28

Table 5 Representation of l 29

Table 6 MANOVA Result 1 30

Table 7 MANOVA Result 2 30

Table 8 MANOVA Result 3 30

Table 9 MANOVA Result 4 30

Table 10 MANOVA Result 5 31

Table 11 ANOVA for tr_acc Result 1 40

Table 12 ANOVA for tr_acc Result 2 40

Table 13 ANOVA for tr_acc Result 3 40

Table 14 ANOVA for logloss Result 1 40

Table 15 ANOVA for logloss Result 2 41

Table 16 ANOVA for logloss Result 3 41

Table 17 Univariate ANOVA for Test of the Interaction Effect 42

Table 18 Difference Matrix of Interaction Effects for Transformed Accuracy Score 43

Table 19 Difference Matrix of Interaction Effects for Logloss 45

Table 20 Algorithms Suitable in Malware Classification for Type of Data Set 46

List of Figures

Figure 1 Roadmap of the Thesis 3

Figure 2 A Part of Hex Code 10

Figure 3 A Part of Disassembled Code 10

Figure 4 Special Symbol "??" in Hex Code 12

Figure 5 Average Transformed Accuracy Score by Data Set 33

Figure 6 Average Transformed Accuracy Score by Learning Algorithm 34

Figure 7 Average Logloss by Data Set 36

Figure 8 Average Logloss by Learning Algorithm 37

Contents

Abstract i

Acknowledgements ii

List of Tables iii

List of Figures iv

Chapter 1 Introduction 1

Chapter 2 Literature Review 5

Chapter 3 Data 9

3.1. Raw Data 9

3.2. Extracted Features 11

3.2.1. Features from Hexadecimal Files 11

3.2.2. Features from Assembly Files 14

3.3. Extraction Implementation 16

Chapter 4 Classification Algorithms and Evaluation Standards 18

4.1. Classification Algorithms 18

4.1.1. Logistic Regression 18

4.1.2. Nearest Neighbors 19

4.1.3. Support Vector Machine 20

4.1.4. Boosted Trees – XGBoost 21

4.1.5. Neural Networks 22

4.2. Evaluation Standards 23

4.2.1. Accuracy Score 23

4.2.2. Logloss 24

Chapter 5 Algorithm Evaluation 25

5.1. Data Summary 25

5.2. Multivariate Analysis of Variance (MANOVA) 27

5.2.1. MANOVA Model 27
5.2.2. No Interaction Test 31
5.3. Interaction Plots 32
5.3.1. Accuracy Scores 32
5.3.2. Logloss 35
5.4. Univariate Analysis of Variance (ANOVA) 38
5.4.1. ANOVA Model 38
5.4.2. No Interaction Test 41
5.4.3. Tukey's Test 42
Chapter 6 Conclusion 47
Reference 50
Appendix: Python Codes for Accuracy Scores and Logloss 56

Chapter 1 Introduction

Malicious software is a major security threat for information systems and user data on personal computers and mobile devices. With the development of internet technologies, including cloud computing and the Internet of Things, the number of malicious files increases as well as the number of cyber applications. According to the quarterly threat report from McAfee Labs in December, 2016, 245 new threats appear every minute, and the total number of malware entities has grown to 644 million at the end of 2016 (McAfee Labs 2016). Techniques for malicious files to escape detection are becoming more sophisticated as are mechanisms for security protection. Obfuscation techniques are used for hiding malware in itself (O'Kane, Sezer and Kieran 2011). The enormous amount of malware and the new techniques for obfuscation represent a huge challenge to information security.

There are two categories of techniques for analyzing malicious programs - dynamic techniques and static ones (Gandotra, Bansal and Sofat 2014). Dynamic techniques analyze the interaction between a malicious program and the information system while the malware is running, while static analysis does not involve program execution. This thesis adopts static methods for malware classification. Common static methods focus on features from hexadecimal code and disassembled files. This category of analysis depends on characteristics extracted from source or binary code.

Malware analysis can be divided into two phases - malware detection and malware classification. Commonly at the first step, an executable is detected if it is malicious, and if it contains a harmful part, then at the second step it is classified into some malware family for further

research. However, in practice, a suspicious program may be categorized into a certain malware family first, which is considered to be an advantage in early malware detection (Cesare and Xiang 2010). For this thesis, the performance of learning algorithms are evaluated on malware classification, therefore all of the cases are assumed to be malware already.

In order to automatically analyze unseen malicious software before it is executed, learning algorithms are employed as heuristic-based methods (Shabtai, et al. 2009). These methods generate classifiers to learn patterns from binary code or disassembled files, and the classifiers are used to analyze unknown executable files. For this thesis, a classifier for classifying new malicious code is learnt from a training set. Five common learning algorithms are adopted for malware classification in the thesis: Logistic Regression, Nearest Neighbors, Support Vector Machine, Boosted Trees, and Neural Network.

In this thesis, the performances of the five learning algorithms were compared on malware classification. Based on the huge collection of malicious binary codes and disassembled files provided by Microsoft on Kaggle.com, a reduced data set is extracted with hexadecimal features and assembly features. The reduced data set was standardized to form what is called a standardized data set. The five learning algorithms to be compared were run on the two data sets with evaluation criteria of performance being accuracy score and logloss. The interaction effect between the factor of data set and the factor of algorithm was tested by two-way MANOVA. Then, interaction plots were employed as a graphical tool to visualize the effects of data sets and algorithms for accuracy score and logloss separately. ANOVAs were applied to test the effects of

two factors for each performance measurement. Tukey's test was applied for multiple comparisons among the combination groups of data sets and algorithms.

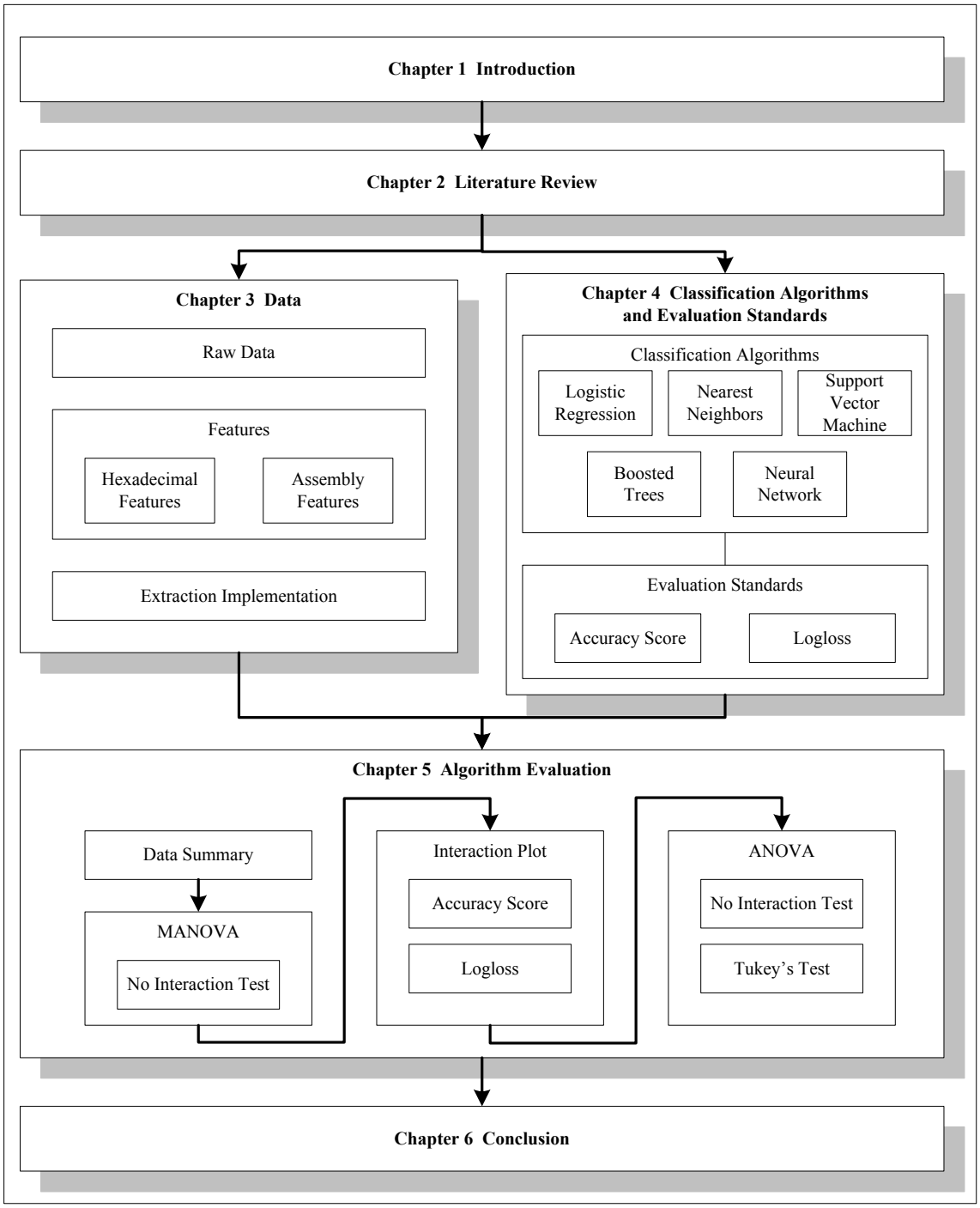

Figure 1 Roadmap of the Thesis

The thesis is organized into 6 chapters: introduction, literature review, data, methods, results, and conclusion. The roadmap of the thesis is shown in as Figure 1. Chapter 1 introduces the motivation of the study in this thesis, the research question, and structure of the thesis. In Chapter 2, earlier research and methodologies are reviewed. Chapter 3 describes the raw data and features and the implementation of feature extraction. Chapter 4 discusses the learning algorithms for malware classification and evaluation criteria used in the thesis. The results of algorithm evaluation are discussed in Chapter 5. When the no interaction hypothesis of algorithm with data set is rejected in MANOVA, the interaction plots are applied to analyze visually the effects of factors for each performance measurement. Then, quantitative results by ANOVA are used to confirm the analyses statistically. Tukey's test is employed for multiple comparisons and conclusions regarding algorithms for given performance measurement and form of data set are provided.

Chapter 2 Literature Review

Two phases are involved with static malicious code classification: feature extraction and classification.

Feature extraction is the first phase for analysis of malware classification. Shabtai et al (2009) called feature extraction 'executable files representation', which indicated feature extraction transferred raw data into readable information for machine. For this purpose, several methods were proposed to obtain information from software for input for further analysis. The *byte n-gram* feature method is one of the representations, which collects n-byte sequences from the detected file. Abou-Assaleh (2004) claimed malware detection accuracy could reach 98% with the byte n-gram method. However, byte n-gram cannot provide meaningful information. Instead the *OpCode n-gram* (Shabtai, et al. 2009) feature method is comparatively clear and easy to understand for human beings. An OpCode consists of an operational instruction and one or more operands. Bilar (2007) experimented on 67 malicious software and 20 benign samples, and the results indicated the malware OpCode frequency distribution significantly differs from that of benign OpCode. Critical OpCode sequences were distinguished with data mining algorithms for classifying malicious and benign software, whose accuracy could reach 98.4% (Siddiqui, Wang and Lee 2008). Nevertheless, malware does not include assembly codes with executable file. That means OpCode has to be drawn from disassembled executable file. But this inverse process sometimes cannot restore the original code, so OpCode cannot be caught precisely. Therefore, other representations were proposed to compensate and to represent executables, such as *string feature* method (Schultz, et al. 2001), and *function-based* feature method (Menahem, et al. 2009). Ahmadi et al. (2016b) proposed a model which focused on malware feature selection. Based on

analysis of malware behavior characteristics, malware instances were classified into malware families. Their experiments were carried out on a massive malware data set (approximate 0.5 TB) published by Microsoft in the *Kaggle* competition in 2015, and reached a 99.8% accuracy.

Next the classification phase is considered. A large number of classic machine learning algorithms has been proposed and developed during the past three decades, and most of them were used in malware detection and classification (Shabtai, et al. 2009), including Artificial Neural Networks (Bishop 1995), Decision Trees (Quinlan, C4.5: Programs for Machine Learning 1993), Naïve-Bayes (John and Langley 1995), Bayesian networks (Pearl 1987), Support Vector Machine (SVM) (Joachims 1998), Boosting (Schapire 1999), and K-Nearest Neighbor (KNN) (Aha, Kibler and Albert 1991). This thesis focuses on supervised learning, i.e., the training data is labelled with categories. These classification techniques use a training data set (from malicious and benign codes) to generate a classification rule, and calibrate the classifier using a test data set. *Artificial Neural Networks* (Bishop 1995) was designed as a network of many neurons cooperatively working for a specific function, like pattern recognition and classification. Every weight of neuron's inputs is modified by a learning algorithm according to the cases the network obtains. *Decision Trees* (Quinlan, C4.5: Programs for Machine Learning 1993) has a tree structure with nodes and leaves. Classifiers can be represented as trees. In order to avoid the problem of over-fitting, a *pruning* technique (Quinlan, Simplifying Decision Trees 1987) was integrated into tree generating. To simplify the model with an exponential number of feature combinations, based on *Bayes theorem*, the *Naïve-Bayes* classifier (John and Langley 1995) assumes that features are conditionally independent. In this model, the posterior probability of a category is proportional to the conditional probability of the features. Prior probability and

posterior probability can be obtained from the training data-set. Unlike Naïve-Bayes, *Bayesian networks* (Pearl 1987) do not make the assumption that the features are independent. The model is a form of directed acyclic graph which illustrates causal dependencies among variables by nodes and arcs. Bayesian networks are also known as belief networks. *Support Vector Machine* (Joachims 1998) utilizes a linear hyper-plane to divide data into two classes. This methodology can deal with a large number of features. *Boosting*, such as *Adaboost* (Schapire 1999) , is a combination of diverse classifiers. Learning is through iteration. Every iteration updates probability for each instances selected in the next iteration. *K-Nearest Neighbor* (KNN) was proposed by Aha, Kibler and Albert (1991). *k* is a small positive integer which is the number of the nearest neighbors. The classification of an object is decided by these *k* neighbors using majority vote. Euclidean distance, Manhattan distance or other distance measures could be used for distance between cases.

Numerous experiments were carried out to compare machine learning algorithms in various situations or integrate them to obtain a result. In experiments by Schultz, et al. (2001), 3301 malicious executables and 1000 benign executables were collected as the data set. The result of their experiments demonstrated that machine learning algorithms are better than signature-based detection methods for accuracy. Experiments by Kolter and Maloof (2004) reveal that Boosted J48 (a decision tree algorithm) surpasses SVM, boosted SVM and IBK (a KNN algorithm) on a small data set with 476 malware and 561 benign executables and on a larger data set with 1651 malware and 1971 benign executables. Elovici et al. (2007) used the Receiver Operating Characteristics (ROC) curve to evaluate some classification algorithms separately and a combined algorithm using a weighing method (Bauer and Kohavi 1999). The experiments were

conducted with both byte-sequence 5-grams and portable executable features, and Neural Networks, Decision Trees and Bayesian networks were tested on the training set. The merged algorithm (Bauer and Kohavi 1999) obtained the best true positive rate and best false positive rate. In the experiments (Moskovitch, Feher, et al. 2008), which employed the top 300 2-grams by true false rate, Boosted Decision Tree, Decision Tree and Artificial Neural Networks reached lower false positive rates than before.

Chapter 3 Data

This chapter describes the raw data, implementation of extraction, and the extracted features. The raw data was a collection of malicious files. Two types of malicious files were provided - binary codes and disassembled files. Therefore, features were extracted from these two types of files. Implementation was based on the previous work of Ahmadi et al (2016a).

3.1. Raw Data

The original data collection is provided by Microsoft on the Kaggle competition website. The training set contains 10,868 samples of malicious software, while 10,873 instances are in the test set. For every virus there is a *.byte file with hexadecimal representation and an *.asm file from assembly view. The real name of each virus is replaced by a unique 20-character hash value. And the portable executable (PE) header is also removed to ensure sterility. The PE header describes the rest of the file. Basic information in PE header of a Windows PE file includes a DOS header and related data, NT header, a section table and section data. It provides rich attributes of the PE file (Choi, et al. 2008). The viruses are from nine different families: Ramnit, Lollipop, Kelihos_ver3, Vundo, Simda, Tracur, Kelihos_ver1, Obfuscator.ACY, and Gatak.

Common representation methods of files can be classified into hex code and disassembled code. The data collection provides both of them. Figure 2 and Figure 3 are taken from the same virus. Figure 2 is from the hex view while Figure 3 is from the assembly view. Hex code expresses machine binary code with hexadecimal digits. This kind of code usually starts with a memory address, followed by operators and operands in machine language. Figure 2 is a snapshot of a

part of a real byte file. “00401000” is the starting address for “56 8D 44 24 08 50 8B F1 E8 1C 1B 00 00 C7 06 08” which represents instructions or data in hexadecimal code. Two hexadecimal digits need one-byte space, consequently this line occupies 16 bytes, and then the next line starts from address 00401010.

```
00401000 56 8D 44 24 08 50 8B F1 E8 1C 1B 00 00 C7 06 08
00401010 BB 42 00 8B C6 5E C2 04 00 CC CC CC CC CC CC CC
00401020 C7 01 08 BB 42 00 E9 26 1C 00 00 CC CC CC CC CC
00401030 56 8B F1 C7 06 08 BB 42 00 E8 13 1C 00 00 F6 44
00401040 24 08 01 74 09 56 E8 6C 1E 00 00 83 C4 04 8B C6
00401050 5E C2 04 00 CC CC CC CC CC CC CC CC CC CC CC CC
00401060 8B 44 24 08 8A 08 8B 54 24 04 88 0A C3 CC CC CC
00401070 8B 44 24 04 8D 50 01 8A 08 40 84 C9 75 F9 2B C2
00401080 C3 CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
```

Figure 2 A Part of Hex Code

Generally, virus executable files are always disassembled to be analysed. For each virus the original data set provides a disassembled file, which is generated from the corresponding hex code by the Interactive Disassembler (IDA) tool[1]. As Figure 3 illustrates, disassembled files translate binary machine code into assembly language which is meaningful for humans. For example, hex decimal digits ‘56’ is disassembled as instruction ‘push’ and the object register ‘esi’. In the real world, dissembling is not an easy task even if some disassembler tools are used.

```
.text:00401000 56                                                    push    esi
.text:00401001 8D 44 24 08                                           lea     eax, [esp+8]
.text:00401005 50                                                    push    eax
.text:00401006 8B F1                                                 mov     esi, ecx
```

Figure 3 A Part of Disassembled Code

[1] https://www.hex-rays.com/products/ida/. . Accessed Apirl 2017.

3.2. Extracted Features

Features for virus classification in this thesis are extracted from both *.byte files and *.asm files. There are five feature categories from hexadecimal files, and eight categories from assembly files. This section explains what these features are and why they are chosen. Generally, a PE header in a file can be informative for feature extraction (Shabtai, et al. 2009); however, this part of each file has been deleted by the data provider to prevent virus spreading.

3.2.1. Features from Hexadecimal Files

1. n-gram

An n-gram is a string of n items in a given sequence. It is used in diverse fields which have a characterizing sequence, such as gene sequencing and computational linguistics. In malicious software detection, n-gram means n-byte. A byte of binary code contains 2^8 (i.e. 256) values. A normal byte shows from "00" to "FF", which can be interpreted into informative machine language. However, a normal situation is that some contents in a file are not initialized sometimes. When accessing those addresses, an exceptional symbol "??" will appear (Figure 4), which is not in the range "00" to "FF". Since "??" is a 'no mapping' signal, the special symbol "??" is considered to be omitted when a hexadecimal file is analyzed in 1-gram method. Consequently, there are still 256 features recorded. Furthermore, 2-gram means 2-byte which contain 2^{16} values meaning 65,536-dimensional vectors need to be dealt with. In order to control computational complexity, only 1-gram is considered as an example of the n-gram method.

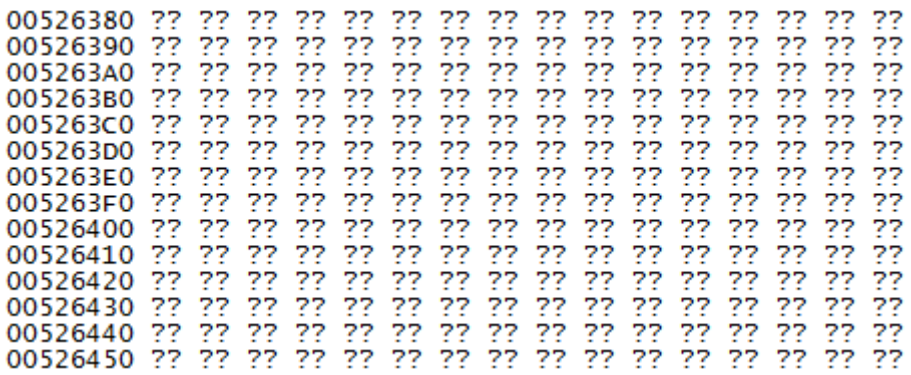

Figure 4 Special Symbol "??" in Hex Code

2. Metadata

Metadata for a file summarizes basic information about the file, such as the owner, date created, date modified and file size. The PE header always provides abundant metadata information for an executable file. Since all of the PE headers have already been removed from the raw data, in this thesis only the file size and the starting address have been extracted from *.byte files. For conformance with other features, the hexadecimal code of the starting address is transferred into decimal values.

3. Entropy

Entropy describes a measurement of change from order to disorder. According to communication theory, entropy is a numerical measure of the uncertainty of an outcome. In previous work (Baysa, Low and Stamp 2013), information entropy was used to deal with the metamorphic detection problem (metamorphic virus can edit itself when it is copied). Entropy is extracted from *.byte files and calculated at byte level. The sliding window method is adopted with the size of each window as 10,000 bytes According to the Shannon's formula (Shannon 2001), that is,

$$e_i = -\sum_{j=1}^{m} p(j) log_2 p(j)$$

where $p(j)$ is the frequency of byte j in window i, and m is the number of different bytes in the window . e_i represents the measurement of entropy in window i. In addition, some descriptive statistics are calculated, such as quantiles and mean.

4. Image Representation

By taking every byte as a pixel at a certain gray-level, a malicious executable can be visualized as a picture and classified (Nataraj, et al. 2011). The transferred pictures show the common patterns within one malware classification. Although some malware samples from different families display similar texture pattern, image representation can be used as a characteristic for malware classification. The Haralick features and the Local Binary Patterns features from the Python package *mahotas* (Coelho 2013) are adopted. The Haralick features are based on the number of a pixel with value i next to a pixel with value j. The Local Binary Patterns features are based on the comparison of a pixel's value and its neighbor's value (Mahotas Contributors 2016). There are efficient for depicting the texture in an image and effective for classification of malware (Ahmadi, et al. 2016b).

5. String Length

Hex Code from byte files is transferred to ASCII strings for malicious executable classification. ASCII of printable characters, and Line Feed (i.e. '\n') and Carriage Return (i.e. '\r') are taken into account. Because many strings are not meaningful, string length is considered instead of

string. The lengths of strings are grouped into successive intervals and counted as features for further malware classification.

3.2.2. Features from Assembly Files

1. Metadata

Similar to the metadata for hexadecimal files, the size of the file and the number of instructions are extracted as assembly file features. After being disassembled, the size of the binary code file is significantly different from the size of the corresponding assembly file. Consequently, both of their sizes are recorded.

2. Symbol

Symbols '-', '+', '*', '[', ']', '?' and '@', if they are at a certain high frequency, are characteristics of a program obfuscated to avoid malware detection. These symbols may possibly call functions indirectly or load dynamic libraries. For an indirect call, the callee's address is not always explicit in the detected object. Clues of data location obfuscation can be exposed in an indirect call, although the implementation of a call depends on the compiling environment or its own structure. Dynamic library loading happens during runtime when an executable loads an external library into memory and imports its functions. Therefore, when the executable file is analyzed by static methods, it is difficult to capture these functions.

3. Operation Code

Operation code (OpCode), also called instruction syllable, comes from assembly language that specifies the operation to be performed. In this thesis, a set of 93 common OpCodes selected by

Ahmadi et al. (2016b) is utilized as features. Those OpCodes are chosen for their high frequency in malicious software. However, these kinds of features can be affected by the technique of instruction replacement (Christodorescu, et al. 2005), which means some rich instruction sets can replace short instruction sequences with the same semantics.

4. Registers

Registers are small high-speed storages in processors. The frequencies of registers in processors are recorded for contribution to malware classification (Ahmadi, et al. 2016b).

5. Application Programming Interface

Application Programming Interfaces (APIs) and their use are adopted as features. According to the research by Ahmadi et al. (2016b), the 794 most frequent APIs are taken into account, which are obtained from approximate 500K malicious samples. Ahmadi et al. (2016b) asserted that these features could even distinguish those malicious files who do not have obvious APIs in static code but call for APIs through the `LoadLibrary` function in runtime.

6. Data Define Instruction

As aforementioned, some malicious executable files do not show any API in static analysis. Ahmadi et al. (2016b) pointed out that typically portable executable files always carry instructions - ‘db’ (defining byte), ‘dw’ (defining word) and ‘dd’ (defining double word) whose frequencies of use can be very useful for malware classification.

7. Section

An executable file contains some sections. Each line in an assembly file starts with a dot and its section name, like '.text'. Common sections are '.text' (executable instructions), '.data' (initialized data), '.bss' (declared variables) and etc. Due to techniques for detection evasion, some section names can be modified or unknown section names can be generated. Therefore, common section names are counted, and unknown sections are recorded and their proportions are calculated.

8. Miscellaneous

A group of assembly codes were picked up as keywords by Ahmadi et al. (2016b) for the feature category. Some of them can indicate the number of blocks in the sample, some can reflect the access to the Windows registry, and some can reveal the number of loaded DLLs.

3.3. Extraction Implementation

The extraction was implemented using Python code, and run on Scientific Linux release 7.2 which belongs to Red Hat Linux. The Kernel version was 3.10.0-327.36.3.el7. The CPUs were Intel® Xeon® E5-2667 v2 @ 3.30GHz. Initially, the Windows operating system was used, but the choice was abandoned since no C Complier was built into the Windows operating system, which meant it was sometimes very difficult to install some Python packages for feature extraction.

The training collection of malicious files in the competition on Kaggle.com was used for feature extraction. Since the file set of malware was provided for competition, the testing collection was

not released with labels for verification. The number of instances in the training collection was 10,868.

The implementation of feature extraction was reproduced through the Python code originating from the Github repository of Ahmadi et al (2016a). As discussed in the previous sections, Python code had two tasks: for features extracted from hexadecimal files and for disassembled files. Before features were extracted, corresponding headers for the features were generated. Since more than 1000 lines of code was published by the authors, it could be expected that bugs were found in it. For example, there were six features which were calculated in the function of *asm_data_define* but for which necessary headers were not prepared in the function of *header_asm_data_define*. For robustness and portability of the source code from the original authors, code for exception handling was added. The debugged code was applied on the training collection. From 250 GB malicious files, 1801 features were extracted to form a 90 MB data set which is called 'regular data set' in this thesis. Then, the function *preprocessing.scale* from the Python package *sklearn* (Pedregosa, et al. 2011) was used on the regular data set to obtain a standardized data set. Using the function, each feature was centered and scaled to unit standard deviation.

Chapter 4 Classification Algorithms and Evaluation Standards

This chapter introduces five learning algorithms for malware classification and the evaluation standards adopted for algorithm comparison. The five learning algorithms are Logistic Regression, Nearest Neighbors, Support Vector Machine, XGBoost, and Neural Networks. They were employed in malware classification in the past (Dahl, et al. 2013) (Firdausi, et al. 2010) (Kolter and Maloof 2004) (Ahmadi, et al. 2016b) (Elovici, et al. 2007) (Moskovitch, Stopel, et al. 2008). The evaluation standards used are accuracy score (Abou-Assaleh, et al. 2004) (Siddiqui, Wang and Lee 2008) (Schultz, et al. 2001) and logloss (Ahmadi, et al. 2016b).

4.1. Classification Algorithms

4.1.1. Logistic Regression

Logistic regression is a linear model for classification (Cox 1958). In a linear model, the output value is represented as a linear combination of the input values as follow,

$$\hat{y}(w, x) = w_0 + w_1 x_1 + \cdots + w_p x_p$$

where $\hat{y}$ is the predicted value, w_0 is intercept, the vector $w = (w_1, w_2, \ldots, w_p)$ are coefficients, and x represents the explanatory variables.

In logistic regression, a sigmoid function is used to model the probability of "success" on a single trial of an experiment that can have only one of two possible outcomes, i.e.

$$f(x) = \frac{L}{1 + e^{-k(x - x_0)}}$$

where x_0 is the sigmoid's midpoint, L is the maximum value for curve, k is the steepness of the curve. For a standard logistic function, parameters are $k = 1, x_0 = 0, L = 1$ (von Seggern 2007).

In this thesis, the function *linear_model.LogisticRegression* from the Python package *sklearn* (Pedregosa, et al. 2011) was employed. It is implemented by minimizing the function (Scikit-learn Developers 2016):

$$\min_{w,c} \frac{1}{2} w^T w + c \sum_{i=1}^{n} \log\left(\exp\left(-y_i(x_i^T w + c)\right) + 1\right)$$

where c is a positive number whose default value is 1.

4.1.2. Nearest Neighbors

Although the principle of the nearest neighbor method is extremely simple, the method is very effective for classification. The basic idea of k nearest neighbors is to predict a point's label by its k nearest neighbors set. Generally, k of training cases can be specified by the user in advance, and the distance measure chosen to be appropriate for the type of data in the data set (Scikit-learn Developers 2016).

The nearest neighbor algorithm for classification uses a majority vote of the k nearest neighbors of the case in question to assign a label to that case. Large values of k could reduce the effects of noisy data but blur boundaries of categories (Scikit-learn Developers 2016). Nearest neighbors usually assumes uniform weights for every neighbor in doing the majority vote, but could use unequal weights under some conditions. In this thesis, the function

neighbors.KNeighborsClassifier from the Python package *sklearn* (Pedregosa, et al. 2011) was employed.

4.1.3. Support Vector Machine

A support vector machine, also known as support vector networks (Cortes and Vapnik 1995), is a supervised learning method for classification that constructs a hyperplane or a group of hyperplanes in a high dimensional space. The basic idea of the method is to find the hyperplane that has the largest gap or margin to the nearest training data point belonging to any class.

In this thesis, the function *SVM.SVC* from the Python package *sklearn* (Pedregosa, et al. 2011) was employed. Different *kernel functions* can be defined in a specific SVM model. The Radial Basis Function (RBF) kernel was implemented as default for the function *SVM.SVC* (Scikit-learn Developers 2016) with the expression:

$$\exp(-\gamma|x - x'|^2)$$

where γ is a kernel coefficient with default value $1/n\_features$. For multi-class classification, the function *SVM.SVC* adopted the one-against-one method (Knerr, Personnaz and Dreyfus 1990). One-against-one means for each pair of classes, one classifier will be generated. If there are n classes, then ${}^{n}C_{2} = n(n-1)/2$ classifiers will be produced. A case will be assigned to the class which obtains the most votes (Scikit-learn Developers 2016).

4.1.4. Boosted Trees – XGBoost

Extreme Gradient Boosting (XGBoost) (Chen and Guestrin 2016) is based on the original model of Gradient Boosting (Friedman 2001). The model of XGBoost is tree ensembles, which incorporates a batch of classification and regression trees (CART). In CART, instances are classified into leaves and a score is measured for every leaf. For tree ensembles, the predictions of diverse trees are summed up to obtain the final score. One advantage of ensembles is that those trees can complement each other. In mathematical form, tree ensembles can be expressed follows (Chen 2014):

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), f_k \in \mathcal{F}$$

where x_i is training data with multiple features, $\hat{y}_i$ is a prediction, $\mathcal{F}$ represents the whole space of CARTs, f is a function in $\mathcal{F}$, and K represents the number of trees. To update what have been learnt, a new tree $f_t(x_i)$ is added at each step:

$$\hat{y}_i^{(0)} = 0 \qquad t = 0$$

$$\hat{y}_i^{(1)} = f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \qquad t = 1$$

$$\hat{y}_i^{(2)} = f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \qquad t = 2$$

$$\dots$$

$$\hat{y}_i^{(t)} = \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)$$

The process is called additive training (Chen 2014).

Chen (2014) finds $f_t(x_i)$ by minimizing the objective function. A typical objective function contains two parts, the training loss function and a regularization term, i.e.

$$Obj(\Theta) = L(\theta) + \Omega(\Theta)$$

where $L(\theta)$ is the training loss and $\Omega(\Theta)$ is the regularization term. The regularization term is helpful for restraining the complexity of the model. The training loss describes differences between the predictions from the trained model and the real world. Mean squared error is generally used as training loss for regression, logloss is commonly used for classification.

In this thesis, the function *XGBClassifier* from the Python package *xgboost* (XGBoost Contributors 2015) was employed.

4.1.5. Neural Networks

Multi-layer Perceptron (MLP) is another supervised learning algorithm (Scikit-learn Developers 2016). By training on a data set, it obtains a function $f(\cdot): R^n \rightarrow R^m$, where n and m represent the number of dimensions for input and output respectively. The model has an input layer, an output layer and possibly several non-linear hidden layers. The bottom layer is the input layer made up of a set of neurons $\{x_i | x_1, x_2, \dots, x_m\}$ which represent the set of features. From the neurons in the previous layer, each neuron in the hidden layer outputs a value. The value contains two parts, a weighted linear summation $\sum_{i=1}^{n} w_i x_i$ and a non-linear activation function $g(\cdot): R \rightarrow R$.

In this thesis, the function *neural_network.MLPClassifier* from the Python package *sklearn* (Pedregosa, et al. 2011) was employed. It is implemented by modeling a MLP with one hidden layer and one hidden neuron function (Scikit-learn Developers 2016). Given training data $(X_i, y_i), X_i \in R^n, y_i \in \{0, 1\}, i = 1, \dots, n$, it trains the function $f(X) = W_2 g(W_1^T X + b_1) + b_2$, where $W_1 \in R^n$ is the weight of the input layer and $W_2 \in R$ is weight of hidden layer, $b_1, b_2 \in R$,

b_1 is the bias added to the hidden layer, and b_2 is the bias added to the output layer. Function $g(\cdot)$ is the activation function. The performance of neural networks can be improved by adding more hidden layers and nodes in the layers.

4.2. Evaluation Standards

4.2.1. Accuracy Score

An algorithm for classifying a malicious file will have the following possible classification results:

- The algorithm classifies the malicious file x into its malware family X correctly (True Positive).
- The algorithm does not classify the malicious file x into its malware family X (False Negative).
- The algorithm classifies the malicious file y into the malware family X, which does not belong to malware family X (False Positive).
- The algorithm does not classify the malicious file y into the malware family X, which does not belong to malware family X (True Negative).

In multiclass classification, accuracy means the predicted label matches exactly the corresponding true label. In this thesis, the function *metrics.accuracy_score* from the Python package *sklearn* (Pedregosa, et al. 2011) was employed for calculating the accuracy score. The default setup of *metrics.accuracy_score* calculates the fraction of correct predictions as follows (Scikit-learn Developers 2016).

$$accuracy(Y, \hat{Y}) = \frac{1}{n_{samples}} \sum_{i=0}^{n_{samples}-1} \mathrm{I}(\hat{y}_i = y_i)$$

where $\hat{Y}$ represent the vector of predicted classes, Y represent the vector of true classes, $\hat{y}_i$ represents the predicted class of the i^{th} sample, y_i is the corresponding true class, $\mathrm{I}(\hat{y}_i = y_i)$ is the indicator function, . Then, $accuracy(Y, \hat{Y})$ means the fraction of correct predictions over $n_{samples}$.

4.2.2. Logloss

Logarithmic loss (logloss) is also called logistic regression loss or cross-entropy loss. It is defined on probability estimates in order to assess the performance of classification methods. In this thesis, the function *metrics.log_loss* from the Python package *sklearn* (Pedregosa, et al. 2011) was employed for calculating logloss. This logloss is calculated using the true value (0 or 1) indicating that case i belongs in class j and the probability that case i belongs in class j (which is supported by function *predict_proba*). The equation (Scikit-learn Developers 2016) below shows logloss for the multiclass case.

$$L_{log}(Y, P) = -logPr(Y|P) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{M} y_{i,j} log(p_{i,j})$$

where Y is a $M \times N$ matrix, N is the number of observations and M represents the true labels. $y_{i,j}$ is element in Y. $y_{i,j}$ is a binary indicator for a true label. If $y_{i,j} = 1$, then observation i belongs to class j, otherwise $y_{i,j} = 0$. P is a $M \times N$ matrix of probability estimates, and $p_{i,j} = Pr(t_{i,j} = 1)$, which is the probability case i is labeled as class j.

Chapter 5 Algorithm Evaluation

In this chapter, five learning algorithms for malware classification are compared for accuracy score and logloss on the regular data set and the standardized data set. To evaluate the five algorithms, 500 observations with measurement of accuracy score and logloss were generated. Statistical analyses and graphical tools, such as multivariate analysis of variance (MANOVA), univariate analysis of variance (ANOVA) and interaction plots, were employed for comparison of the performance of the algorithms.

5.1. Data Summary

The learning algorithms to be evaluated consisted of Logistic Regression (lr), K Nearest Neighbors (knn), Support Vector Machine (svm), eXtreme Gradient Boosting (xg), and Neural Networks (nn). Two measurements were considered as criteria to assess the performance of the learning algorithms. One is accuracy score (acc), the other one is logloss. Every measurement was the average based on 5-fold cross validation. Two data sets are used – the regular data set (reg) and the standardized data set (std). Every observation consisted of an accuracy score and a logloss value. Table 1 is the summary for measures in combinations of data set and algorithm. On both regular data set and standardized data set, XGBoost obtained the highest average accuracy score and lowest average logloss. On the standardized data set, the average accuracy scores by all of the five algorithms were more than 0.95, and the average logloss were less than 0.24, which are better than those value on regular data set. In next sections, further statistical research is done to analyze the performance of the algorithms.

Table 1 Summary for Accuracy Score and Logloss

Dataset	Algorithm	Measures	N	Mean	Std Dev	Minimum	Maximum
reg	kn	acc	50	0.9098582	0.0010817	0.9074339	0.9121273
		logloss	50	0.4654603	0.0133271	0.4382603	0.4968539
	lr	acc	50	0.6650275	0.0032580	0.6600109	0.6764788
		logloss	50	1.0075350	0.0013308	1.0049871	1.0105907
	nn	acc	50	0.8499631	0.0212958	0.7931571	0.8815800
		logloss	50	5.1790551	0.7354796	4.0872420	7.1411532
	sv	acc	50	0.2202650	0.0299804	0.1620332	0.2685859
		logloss	50	1.8959201	0.000691605	1.8941876	1.8974445
	xg	acc	50	0.9957895	0.000309149	0.9950313	0.9965033
		logloss	50	0.0144330	0.000682060	0.0132755	0.0161383
std	kn	acc	50	0.9536640	0.000551862	0.9526129	0.9550977
		logloss	50	0.2377489	0.0081797	0.2205030	0.2569026
	lr	acc	50	0.9901233	0.000424966	0.9892341	0.9910745
		logloss	50	0.0495829	0.0025722	0.0450326	0.0568850
	nn	acc	50	0.9903957	0.000689243	0.9891426	0.9922708
		logloss	50	0.0495327	0.0039126	0.0412655	0.0578276
	sv	acc	50	0.9814943	0.000416410	0.9805847	0.9825175
		logloss	50	0.0613277	0.0013769	0.0585506	0.0646283
	xg	acc	50	0.9957564	0.000314183	0.9950313	0.9965033
		logloss	50	0.0144528	0.000707788	0.0131685	0.0161491

The accuracy score is the percentage of true positives. MANOVA assumes variables are normally distributed. Consequently, all accuracy scores were transformed using the arcsine transformation to fulfill the requirement of MANOVA. The arcsine of the square root of the number to be transformed is calculated (Rédei 2008) as follows:

$$X' = \sin^{-1} \sqrt[2]{X}$$

For transformed accuracy score, the summary is as Table 2.

Table 2 Summary for Transformed Accuracy Score

Dataset	Algorithm	N	Mean	Std Dev	Minimum	Maximum
reg	kn	50	1.2658610	0.0018883	1.2616487	1.2698403
	lr	50	0.9535832	0.0034569	0.9482744	0.9657633
	nn	50	1.1738778	0.0295390	1.0986489	1.2194928
	sv	50	0.4876360	0.0364722	0.4142827	0.5448067
	xg	50	1.5059050	0.0023879	1.5002489	1.5116292
std	kn	50	1.3538440	0.0013144	1.3513535	1.3572759
	lr	50	1.4712729	0.0021512	1.4668504	1.4761805
	nn	50	1.4726998	0.0035673	1.4664079	1.4827667
	sv	50	1.4343459	0.0015475	1.4310027	1.4381865
	xg	50	1.5056508	0.0024221	1.5002489	1.5116292

5.2. Multivariate Analysis of Variance (MANOVA)

The main purpose of using MANOVA was to determine if logloss and accuracy score were affected by the form of the data set (regular or standardized) and or by the learning algorithms, or the interaction of these two factors. The two data sets and the five algorithms were considered as the explanatory variables.

5.2.1. MANOVA Model

In this section, the effects of data sets and algorithms on the performances of learning algorithms are discussed. MANOVA is applied to model two or more continuous dependent variables using one or more categorical predictor variables. MANOVA was implemented in this thesis with both logloss and transformed accuracy scores as dependent variables, the five learning algorithms as "treatments", the regular data set and the standardized data set as the "blocks", and with interaction effects between algorithm and data set. For the model in this thesis, $\boldsymbol{Y}_{ijk}$ is a $p \times 1$ vector of measurements. In the thesis p is 2, which means two measurements, transformed

accuracy score and logloss. $\boldsymbol{Y}_{ijk}$ represents the k^{th} observation at the i^{th} level of factor data set and the j^{th} level of factor algorithm. The model can be expressed in the form

$$\boldsymbol{Y}_{ijk} = \boldsymbol{\mu} + \boldsymbol{\beta}_i + \boldsymbol{\tau}_j + (\boldsymbol{\beta\tau})_{ij} + \boldsymbol{\varepsilon}_{ijk}$$

$$i = 1, 2$$

$$j = 1, 2, \dots, 5$$

$$k = 1, 2, \dots, 50$$

where the parameter $\boldsymbol{\beta}_i$ represents the effect of data set i (Table 3), the parameter $\boldsymbol{\tau}_j$ represents the effect of learning algorithm j (Table 4), and $(\boldsymbol{\beta\tau})_{ij}$ represents the interaction effect of data set i and algorithm j. For each combination of data set i and algorithm j, there are 50 observations. And all are $p \times 1$ (i.e. 2×1) vectors.

Table 3 Representation of *i*

i	**Representation**
1	The regular data set
2	The standardized data set

Table 4 Representation of *j*

j	**Representation**
1	K Nearest Neighbors
2	Logistic Regression
3	Neural Networks
4	Support Vector Machine
5	XGBoost

Therefore, the vector of measurements $\boldsymbol{Y}_{ijk}$ can be expressed as

$$[y_{ijk1} \quad y_{ijk2}] = [\mu_1 \quad \mu_2] + [\beta_{i1} \quad \beta_{i2}] + [\tau_{j1} \quad \tau_{j2}] + [(\beta\tau)_{ij1} \quad (\beta\tau)_{ij2}] + [\varepsilon_{ijk1} \quad \varepsilon_{ijk2}]$$

The restrictions imposed are:

$$\sum_{i=1}^{2}\beta_{il}=\sum_{j=1}^{5}\tau_{jl}=\sum_{i=1}^{2}(\beta\tau)_{ijl}=\sum_{j=1}^{5}(\beta\tau)_{ijl}=0$$

$$l=1,2$$

and

$$\boldsymbol{\varepsilon}_{ijk}\sim N_2(\mathbf{0},\Sigma)$$

where i is data set, j is algorithm, k is observation, l is measure (Table 5) and Σ is the covariance matrix.

Table 5 Representation of l

l	Representation
1	Transformed accuracy score
2	Logloss

The model for the entire set of data can be expressed in matrix notation as

$$\begin{bmatrix} y_{1,1,1,1} & y_{1,1,1,2} \\ y_{1,1,2,1} & y_{1,1,2,2} \\ \vdots & \vdots \\ y_{1,1,50,1} & y_{1,1,50,2} \\ y_{1,2,1,1} & y_{1,2,1,2} \\ \vdots & \vdots \\ y_{1,2,50,1} & y_{1,2,50,2} \\ \vdots & \vdots \\ y_{2,5,1,1} & y_{2,5,1,2} \\ \vdots & \vdots \\ y_{2,5,50,1} & y_{2,5,50,2} \end{bmatrix}_{500\times 2} = \begin{bmatrix} 1&1&1&0&0&0&1&0&0&0 \\ 1&1&1&0&0&0&1&0&0&0 \\ \vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots \\ 1&1&1&0&0&0&1&0&0&0 \\ 1&1&0&1&0&0&0&1&0&0 \\ \vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots \\ 1&1&0&1&0&0&0&1&0&0 \\ \vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots \\ 1&-1&-1&-1&-1&-1&-1&-1&-1&-1 \\ \vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots&\vdots \\ 1&-1&-1&-1&-1&-1&-1&-1&-1&-1 \end{bmatrix}_{500\times 10} \begin{bmatrix} \mu_1 & \mu_2 \\ \beta_{11} & \beta_{12} \\ \tau_{11} & \tau_{12} \\ \tau_{21} & \tau_{22} \\ \tau_{31} & \tau_{32} \\ \tau_{41} & \tau_{42} \\ (\beta\tau)_{111} & (\beta\tau)_{112} \\ (\beta\tau)_{121} & (\beta\tau)_{122} \\ (\beta\tau)_{131} & (\beta\tau)_{132} \\ (\beta\tau)_{141} & (\beta\tau)_{142} \end{bmatrix}_{10\times 2} + \varepsilon$$

Using SAS, the results of MANOVA appear in Table 6, Table 7, Table 8, Table 9, and Table 10. Table 6 displays the error SSCP (the Sums of Squares and Cross Products) matrix **E**. The diagonal elements of this matrix are the error sums of squares from the corresponding univariate analyses (Table 11 and Table 14). Table 7 shows the partial correlation matrix computed from the error SSCP matrix. Table 8 displays Type III SSCP matrix **H** associated with the effect of

data set and algorithm, where Type III error means the probability of "correctly rejecting the null hypothesis for the wrong reason" (Huynh n.d.). The diagonal elements of this matrix are the model sums of squares from the corresponding univariate analyses (Type III SS for dataset*algorithm in Table 13 and Table 16). Table 9 displays the characteristic roots and vectors of $\mathbf{E^{-1}H}$. Table 10 displays the results of four multivariate tests, all based on the characteristic roots and vectors of $\mathbf{E^{-1}H}$.

Table 6 MANOVA Result 1

E = Error SSCP Matrix		
	tr_acc	**logloss**
tr_acc	0.1103152955	-1.065054347
logloss	-1.065054347	26.518887254

Table 7 MANOVA Result 2

Partial Correlation Coefficients from the Error SSCP Matrix / Prob > |r|		
DF=490	**tr_acc**	**logloss**
tr_acc	1.000000	-0.622696
logloss	-0.622696	1.000000

Table 8 MANOVA Result 3

H = Type III SSCP Matrix for dataset*algorithm		
	tr_acc	**logloss**
tr_acc	14.402363041	-19.21594688
logloss	-19.21594688	434.08853917

Table 9 MANOVA Result 4

Characteristic Roots and Vectors of: E Inverse * H, where H = Type III SSCP Matrix for dataset*algorithm E = Error SSCP Matrix			
Characteristic Root	**Percent**	**Characteristic Vector V'EV=1**	
		tr_acc	**logloss**
200.763914	92.47	3.84774814	0.15313144
16.359418	7.53	0.02778528	0.19529884

Table 10 MANOVA Result 5

MANOVA Test Criteria and F Approximations for the Hypothesis of No Overall dataset*algorithm Effect H = Type III SSCP Matrix for dataset*algorithm E = Error SSCP Matrix S=2 M=0.5 N=243.5					
Statistic	**Value**	**F Value**	**Num DF**	**Den DF**	**Pr > F**
Wilks' Lambda	0.00028551	7112.74	8	978	<.0001
Pillai's Trace	1.93743809	3793.62	8	980	<.0001
Hotelling-Lawley Trace	217.12333183	13255.5	8	696.25	<.0001
Roy's Greatest Root	200.76391423	24593.6	4	490	<.0001

5.2.2. No Interaction Test

We first test the hypothesis of no interaction effects of data sets and learning algorithms as

$$H_0: (\beta\tau)_{11} = (\beta\tau)_{12} = \cdots = (\beta\tau)_{25} = 0,$$

versus

$$H_a: at\ least\ one\ (\beta\tau)_{ij} \neq 0$$

The F statistic for the Wilk's Lambda was $F = 7112.74$ with p-value $p < 0.0001$ (Table 10). Therefore, we reject the null hypothesis of no interaction effects. Since the interaction effects of data set and algorithm are not all zero, then the effect of one factor may depend on the level of the other factor and both effect of data set and effect of learning algorithm cannot be assumed to be additive.

5.3. Interaction Plots

Interaction plots are graphical statistical tools to illustrate the effects between two or more factors. The interaction plots show interaction effects between algorithm and data set exist for accuracy score and logloss. The performance of four algorithms for the two measures was improved on the standardized data set, but the algorithm XGBoost got the same results of accuracy score and logloss on the two data sets. For accuracy score, SVM is most sensitively affected by the type of data set, while for logloss, KNN is the most affected one.

5.3.1. Accuracy Scores

Figure 5 plots average accuracy score for each algorithm by data set used. Figure 6 plots average accuracy score for each data set by algorithm used. Higher accuracy scores reveal a better performing an algorithm in some circumstance. In these figures, “kn” represents KNN, “lr” represents Logistic Regression, “nn” represents Neural Networks, “sv” represents SVM, “xg” represents XGBoost, “reg” represents the regular data set, and “std” represents the standardized data set.

The fact that these lines are not parallel indicates interaction effects between algorithm and data set are present. From Figure 5 and Figure 6, we use that standardization of the data set improved most of the results of tr_acc, especially the accuracy scores for Support Vector Machine, Logistic Regression and Neural Network. The interaction of data set with algorithm is most positive for SVM. For accuracy score, the performance of K Nearest Neighbors was boosted slightly, while performance of XGBoost was unchanged, indicating the effect of data set on these two algorithms was weak.

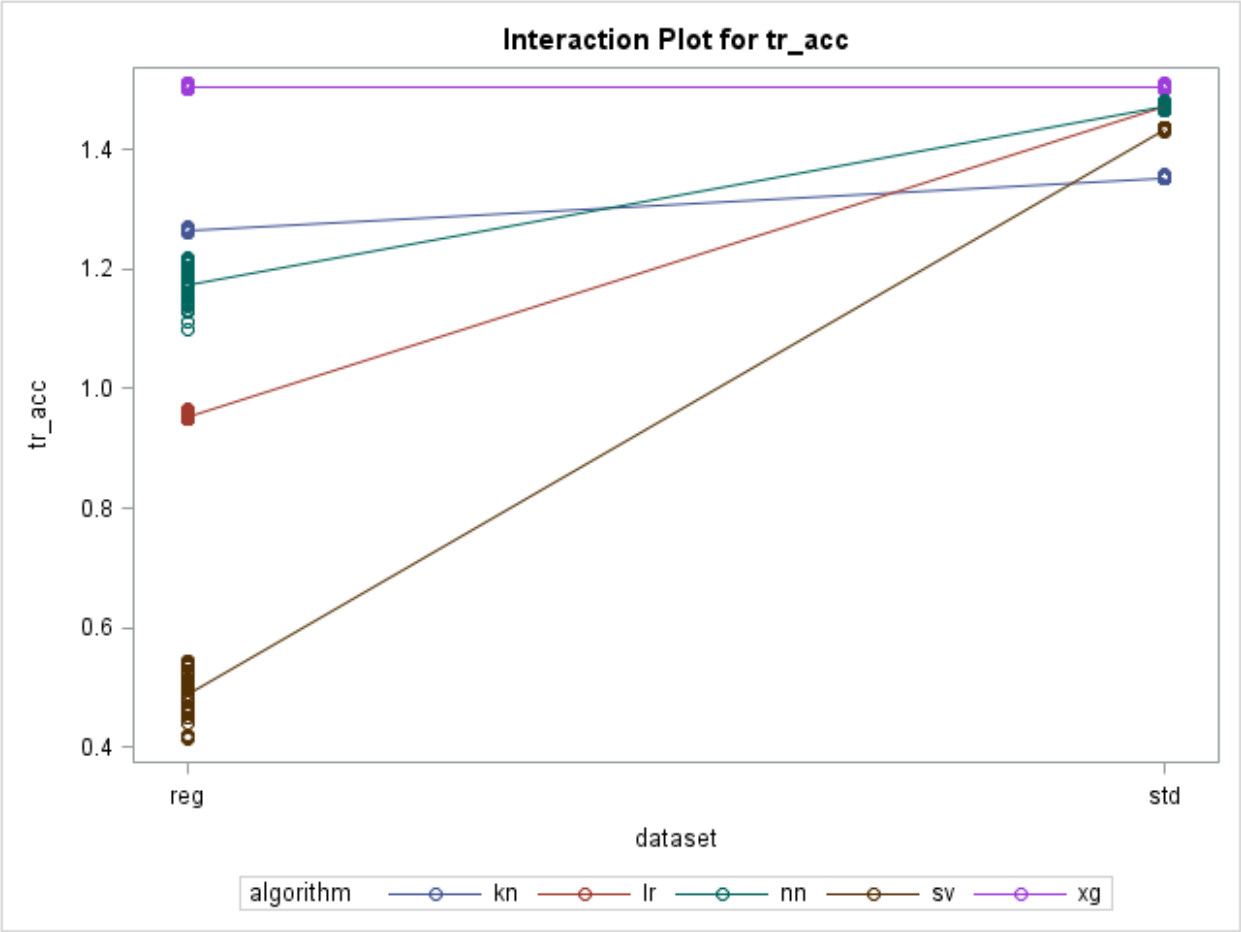

Figure 5 Average Transformed Accuracy Score by Data Set

Figure 6 shows clearly that readings for accuracy score by algorithm for the standardized data set are above those for the regular data set. This means data set has a significant effect on the accuracy scores. However, differences between the regular data set and the standardized data set vary for each algorithm. This suggests a statistically significant effect for algorithm. The varying slopes for the regular data set (as opposed to the standardized data set) suggest that algorithm effect is strongest for the regular data set. Take the steepest line of the regular data set in the segment from sv to xg as an example. Both the line of the regular data set and the line of the standardized data set go up, but the scope of the former is obviously steeper. That indicates the

factor algorithm has a more significant effect for the regular data set than for the standardized data set when the algorithm is changed from SVM to XGBoost.

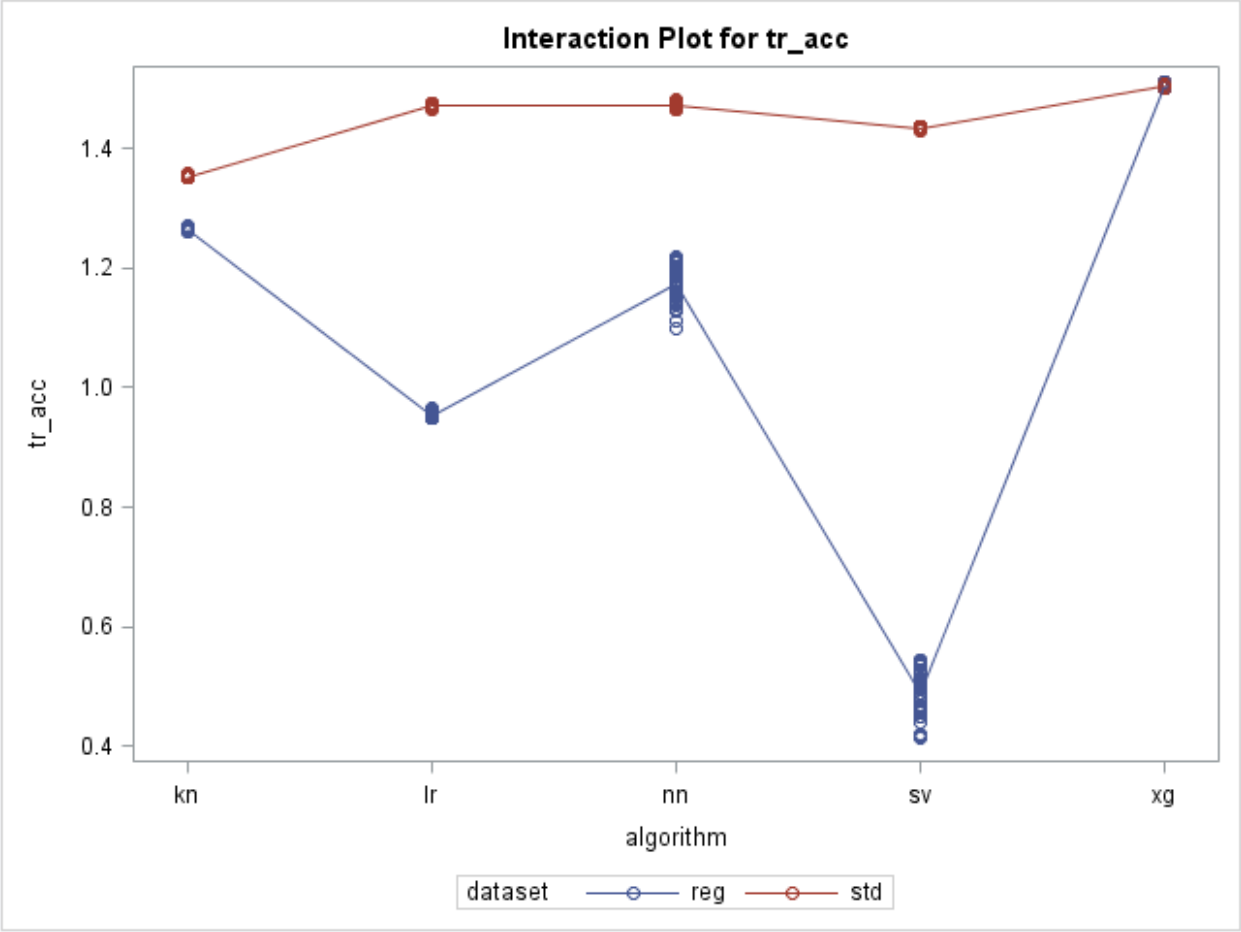

Figure 6 Average Transformed Accuracy Score by Learning Algorithm

According to Figure 5 and Figure 6, the overall performance of the standardized data set is better than that of the regular data set for the accuracy scores, however the effect of data set is not significant on XGBoost. XGBoost reaches higher accuracy scores than the other four algorithms on both the regular data set and the standardized data set. The performances of KNN on both the

two data sets are quite similar. Logistic Regression, SVM and Neural Network are influenced significantly by the type of data set, and SVM is the most sensitive among them.

5.3.2. Logloss

Figure 7 and Figure 8 display the logloss by data set and by algorithm respectively. In Figure 7 each colour represents a learning algorithm, while in Figure 8 each curve represents a data set. For logloss, low values indicate better performance of a learning algorithm, as aforementioned in section 4.2.2.

Figure 7 illustrates the effect of data set is different for the five algorithms. Four lines decline but with different slopes. The line for Neural Networks has the steepest downward slope, and compared with it, the line for SVM and the line for Logistic Regression drop gradually, which means the type of data set affects the performances of the three learning algorithms. However, the line representing KNN drops only slightly, and the line for XGBoost is horizontal. That implies the effect of data set is not so large here. Accordingly, logloss using the algorithms of Neural Networks, SVM and Logistic Regression can be enhanced significantly by choosing a standardized data set, while the choice data set does not have much effect on logloss when using the algorithms XGBoost and KNN.

In Figure 8, we clearly see the difference, by algorithm, due to data set when measuring performance in terms of logloss. Figure 8 shows the curve for the standardized data set is below that of the regular data set. Because a lower value of logloss represents a better performance of a learning algorithm, this result implies that standardizing improves the performance of most

algorithms. Clearly, based on Figure 8, the effect of algorithms cannot be rejected but the effects depend on the form of the data set used. The curve for the standardized data set does not change slope very much in each segment (especially in segment of (lr, nn), (nn, sv), (sv, xg)), whereas the curve for the regular data set has an obvious slope change in each segment. This explains the stronger effect of algorithm with the regular data set than with the standardized data set when logloss is used as measuring the performance of an algorithm.

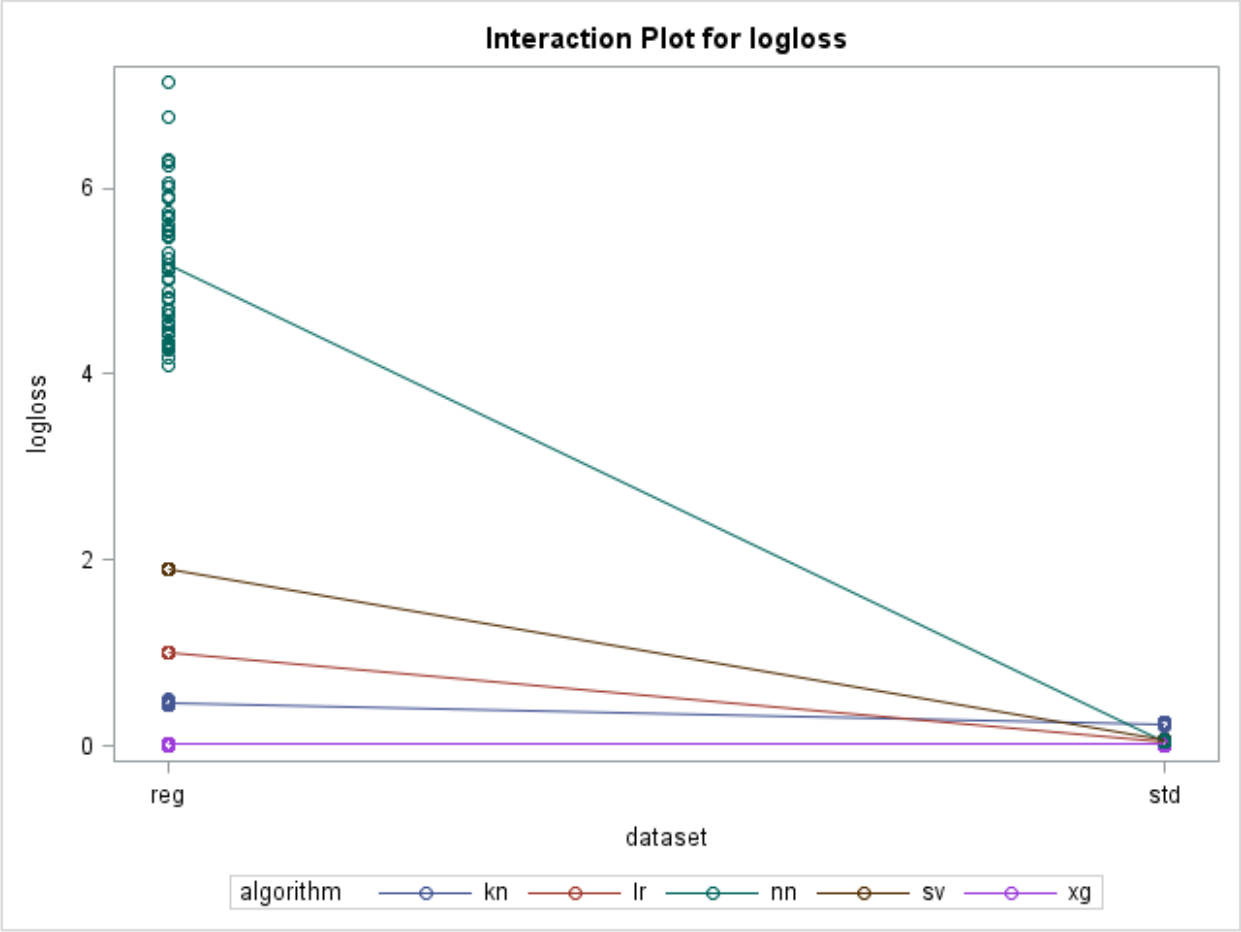

Figure 7 Average Logloss by Data Set

From these graphs we observe that, the performance of four algorithms for logloss was improved on the standardized data set, but the algorithm XGBoost got the same results of logloss on the

two data sets. The logloss performance of XGBoost was the best among the five learning algorithms no matter which data set was used. The values of logloss for the KNN algorithm on the two data sets are close to each other. Other three algorithms - Logistic Regression, SVM and Neural Network are sensitive to the type of data set, and the results of all these on logloss are very close to that of XGBoost, (the best one). Most conclusions for logloss are similar to those for accuracy score, except here the most sensitive algorithm to the type of data set is Neural Networks instead of SVM.

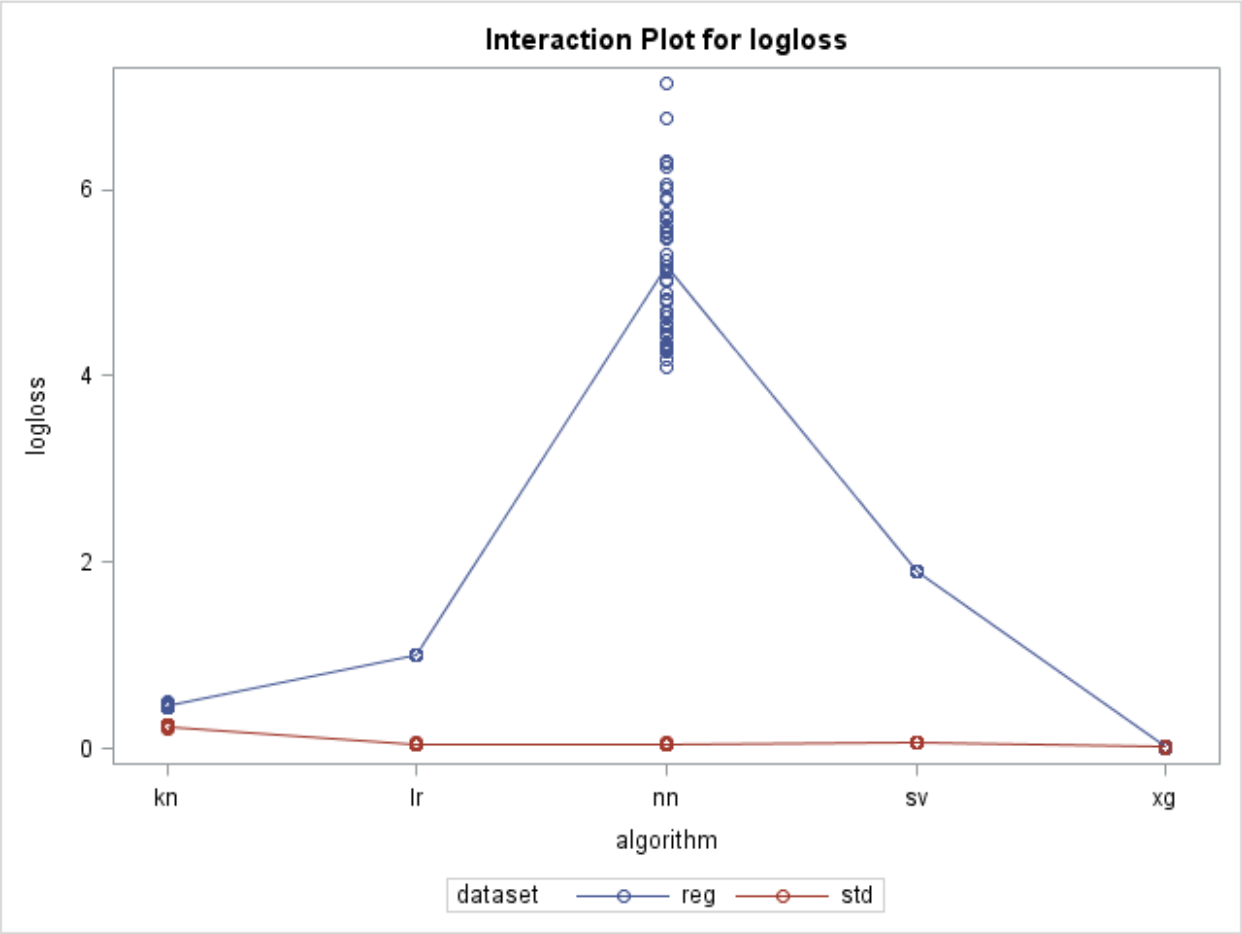

Figure 8 Average Logloss by Learning Algorithm

5.4. Univariate Analysis of Variance (ANOVA)

Since the interaction effect of data set and algorithm was significant, then separate univariate ANOVAs were performed on each measurement which was a dependent variable in the model of MANOVA in 5.2.1. The univariate ANOVA is able to test the effects of two predictor variables on one dependent variable.

5.4.1. ANOVA Model

In this section we discuss whether the interaction effects are present in both models, for accuracy score and logloss, or only on one of them. Now, univariate ANOVA models are fitted on transformed accuracy score and logloss individually. The five learning algorithms are considered as "treatments", the regular data set and the standardized data set are taken as another factor "blocks", and interaction effects between algorithm and data set are included in the model.

For the univariate model in this thesis, y_{ijk} is an observation of transformed accuracy score (or logloss), the k^{th} observation at the i^{th} level of factor data set and the j^{th} level of factor algorithm. The model used is shown as follow

$$y_{ijk} = \mu + \beta_i + \tau_j + (\beta\tau)_{ij} + \varepsilon_{ijk}$$

$$i = 1, 2$$

$$j = 1, 2, \dots, 5$$

$$k = 1, 2, \dots, 50$$

where the parameter β_i represents the effect of data set i (Table 3), the parameter $\boldsymbol{\tau}_j$ represents the effect of learning algorithm j (Table 4), and $(\beta\tau)_{ij}$ represents the interaction effect of data

set i and algorithm j. $k = 1, 2, \ldots, 50$ means for each combination of data set i and algorithm j, there are 50 observations.

The assumptions imposed are:

$$\sum_{i=1}^{2} \beta_i = \sum_{j=1}^{5} \tau_j = \sum_{i=1}^{2} (\beta\tau)_{ij} = \sum_{j=1}^{5} (\beta\tau)_{ij} = 0$$

and

$$\varepsilon_{ijk} \sim N(0, \Sigma)$$

The responses in the observation vector of transformed accuracy score (or logloss) can be written in matrix notation as

$$\begin{bmatrix} y_{1,1,1} \\ y_{1,1,2} \\ \vdots \\ y_{1,1,50} \\ \hdashline y_{1,2,1} \\ \vdots \\ y_{1,2,50} \\ \hdashline \vdots \\ \hdashline y_{2,5,50} \\ \vdots \\ y_{2,5,50} \end{bmatrix}_{500\times 1} = \left[\begin{array}{cccccccccc} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hdashline 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ \hdashline \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \hdashline 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{array}\right]_{500\times 10} \begin{bmatrix} \mu \\ \beta_1 \\ \tau_1 \\ \tau_2 \\ \tau_3 \\ \tau_4 \\ (\beta\tau)_{11} \\ (\beta\tau)_{12} \\ (\beta\tau)_{13} \\ (\beta\tau)_{14} \end{bmatrix}_{10\times 1} + \boldsymbol{\varepsilon}$$

The total degrees of freedom total are equal to the total number of observations in the analysis minus one, that is $500 - 1 = 499$. And the degrees of freedom for the effect of a factor is equal to the number of levels of the factor minus one. Therefore, for data set, $df = 2 - 1 = 1$ since there are two levels of data sets (regular and standardized). Similarly, for learning algorithm, $df = 5 - 1 = 4$ since there are five levels of algorithm. The degrees of freedom for the interaction is equal to the product of the degrees of freedom of the variables in the interaction. Thus, the degree of freedom for the interaction effect of data set and algorithm are $df = 1 \times 4 =$

4. The error degrees of freedom is equal to the total degrees of freedom minus the degree of freedom for all the effects, i.e. $\mathrm{df} = 499 - 1 - 4 - 4 = 490$.

From SAS, the results of ANOVA for transformed accuracy score are shown in Table 11, Table 12, and Table 13.

Table 11 ANOVA for tr_acc Result 1

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	9	47.38610768	5.26512308	23386.7	<.0001
Error	490	0.11031530	0.00022513		
Corrected Total	499	47.49642297			

Table 12 ANOVA for tr_acc Result 2

R-Square	Coeff Var	Root MSE	tr_acc Mean
0.997677	1.188501	0.015004	1.262468

Table 13 ANOVA for tr_acc Result 3

Source	DF	Type III SS	Mean Square	F Value	Pr > F
dataset	1	17.13008445	17.13008445	76088.6	<.0001
algorithm	4	15.85366019	3.96341505	17604.8	<.0001
dataset*algorithm	4	14.40236304	3.60059076	15993.2	<.0001

From SAS, the results of ANOVA for logloss are shown in Table 14, Table 15, and Table 16.

Table 14 ANOVA for logloss Result 1

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	9	1182.968502	131.440945	2428.69	<.0001
Error	490	26.518887	0.054120		
Corrected Total	499	1209.487389			

Table 15 ANOVA for logloss Result 2

R-Square	Coeff Var	Root MSE	tr_acc Mean
0.978074	25.92047	0.232637	0.897505

Table 16 ANOVA for logloss Result 3

Source	DF	Type III SS	Mean Square	F Value	Pr > F
dataset	1	332.0928195	332.0928195	6136.21	<.0001
algorithm	4	416.7871432	104.1967858	1925.29	<.0001
dataset*algorithm	4	434.0885392	108.5221348	2005.21	<.0001

5.4.2. No Interaction Test

The null hypothesis of no interaction effect of data sets and learning algorithms for each performance measurement is:

$$H_0: (\beta\tau)_{11} = (\beta\tau)_{12} = \cdots = (\beta\tau)_{25} = 0,$$

versus

$$H_a: at\ least\ one\ (\beta\tau)_{ij} \neq 0$$

The F ratio for an effect is calculated by dividing the mean square for the effect by the mean square error. Hence, the F-value for each of transformed accuracy score and logloss were computed using the mean square for the interaction effect and the mean square error, respectively. The F-value and the p-value for each univariate ANOVA test for interaction of algorithm with data set are displayed in Table 17. The hypotheses of no interaction effect of data set and algorithm was rejected for both transformed accuracy score and logloss. Therefore, the interaction effects between data set and algorithm are present for both performance measures.

Table 17 Univariate ANOVA for Test of the Interaction Effect

Measurement	Mean Square for Error	DF of Error	Mean Square for Interaction	DF of Interaction	F-Value	P-Value
tr_acc	0.00022513	490	3.60059076	4	15993.2	<.0001
logloss	0.054120	490	108.5221348	4	2005.21	<.0001

5.4.3. Tukey's Test

Tukey's range test or Tukey's HSD (honest significant difference) test is performed as a pairwise comparison test. Based on a studentized range distribution, Tukey's test compares all pairs of means. Tukey's test is a post-hoc test, which means it should be performed after an ANOVA test. After ANOVA indicates that not all effects are the same, and the Tukey's test clarifies which effects have significant differences.

Since the null hypothesis of no interaction effects was rejected in the ANOVAs in the previous section, Tukey's test was performed to examine further the nature of the effects. Here, μ_{ij} is the average of transformed accuracy scores (or logloss) at the level i of data set and level j of learning algorithm. The null hypothesis of no difference in the average of each performance measurement for different combinations of algorithm and data set can be expressed as:

$$H_0: \mu_{ij} = \mu_{i'j'}, ij \neq i'j'$$

$$i, i' = 1, 2$$

$$j, j' = 1, 2, \dots, 5$$

versus

$$H_a: \mu_{ij} \neq \mu_{i'j'}$$

Table 18 and Table 19 display the p-values for Tukey's test for transformed accuracy score and logloss. The p-values in grey indicate statistically significant difference for accuracy score (or logloss) for the paired comparison represented by the row and column in the table.

Table 18 Difference Matrix of Interaction Effects for Transformed Accuracy Score

	reg-kn	reg-lr	reg-nn	reg-sv	reg-xg	std-kn	std-lr	std-nn	std-sv	std-xg
reg-kn										
reg-lr	<.0001									
reg-nn	<.0001	<.0001								
reg-sv	<.0001	<.0001	<.0001							
reg-xg	<.0001	<.0001	<.0001	<.0001						
std-kn	<.0001	<.0001	<.0001	<.0001	<.0001					
std-lr	<.0001	<.0001	<.0001	<.0001	<.0001	<.0001				
std-nn	<.0001	<.0001	<.0001	<.0001	<.0001	<.0001	1.0000			
std-sv	<.0001	<.0001	<.0001	<.0001	<.0001	<.0001	<.0001	<.0001		
std-xg	<.0001	<.0001	<.0001	<.0001	1.0000	<.0001	<.0001	<.0001	<.0001	

Table 18 shows the p-values of Tukey's pairwise comparison on the combination of data set and algorithm for transformed accuracy score. For regular data set, all of the five learning algorithms differ significantly from each other for the average transformed accuracy scores. According to the inspection of average transformed accuracy scores of these algorithms, Table 2 (in section

5.1) shows that the result of XGBoost was higher than for the other four algorithms. This means that the performance of XGBoost on accuracy score is better than the other four algorithms, when they are implemented on the regular data set. For the standardized data set, there were significant differences between KNN and Logistic Regression, KNN and Neural Networks, KNN and SVM, KNN and XGBoost, and also between Logistic Regression and SVM, Logistic Regression and XGBoost, Neural Networks and SVM, Neural Networks and XGBoost, and SVM and XGBoost. This suggests that the performance XGBoost on accuracy score is significantly different from the other algorithms. Since the average of transformed accuracy score for XGBoost is higher than other learning algorithms on the standardized data set, this suggests that XGBoost can be the best choice when accuracy score on standardized data set is considered. Further, it can be noted that the hypothesis of no difference between the combination of regular data set and XGBoost and the combination of standardized data set and XGBoost cannot be rejected (p-value = 1). That indicates XGBoost is not influenced by type of data set when for accuracy score is the performance measure. This characteristic of XGBoost was confirmed by its author (Chen 2014).

Table 19 displays the p-values of Tukey's pairwise comparison on the combination of data set and algorithm for logloss. With regard to the regular data set, there are significant differences among all of the five learning algorithms for the average logloss. An inspection of average logloss (Table 1) by these five algorithms shows that XGBoost reached a lower value than the other four algorithms. Therefore, the performance of XGBoost for logloss is better than others on the regular data set. For standardized data set, KNN differs significantly from each of the other four algorithms. The p-values for the Tukey's pairwise comparisons for the four algorithms are

all beyond 0.99, which means the four algorithms perform similarly when measuring logloss on the standardized data set. Since average logloss for KNN is higher than for the other algorithms, which means worse performance, the other four algorithms - Logistic Regression, SVM, XGBoost and Neural Networks - should be chosen for lower logloss on the standardized data set.

Table 19 Difference Matrix of Interaction Effects for Logloss

	reg-kn	reg-lr	reg-nn	reg-sv	reg-xg	std-kn	std-lr	std-nn	std-sv	std-xg
reg-kn										
reg-lr	<.0001									
reg-nn	<.0001	<.0001								
reg-sv	<.0001	<.0001	<.0001							
reg-xg	<.0001	<.0001	<.0001	<.0001						
std-kn	<.0001	<.0001	<.0001	<.0001	<.0001					
std-lr	<.0001	<.0001	<.0001	<.0001	0.9991	0.0024				
std-nn	<.0001	<.0001	<.0001	<.0001	0.9991	0.0024	1.0000			
std-sv	<.0001	<.0001	<.0001	<.0001	0.9917	0.0064	1.0000	1.0000		
std-xg	<.0001	<.0001	<.0001	<.0001	1.0000	<.0001	0.9991	0.9991	0.9918	

Table 20 summarizes recommendations for algorithm choice based on data set type and performance measure. The results are in agreement with the conclusions from the discussion of the interaction plots in section 5.3. For regular data set, XGBoost gives the best performance on both accuracy score and logloss. Hence, based on the evidence from algorithm evaluation,

XGBoost can be used in malware classification on regular data set. For the standardized data set, XGBoost performs better for accuracy score than the others, while there was not much difference among Logistic Regression, SVM, XGBoost and Neural Network for logloss. Therefore, of the algorithms used in the thesis and under the default setting for most of them, XGBoost should be chosen for classifying malicious files from the five algorithms.

Table 20 Algorithms Suitable in Malware Classification for Type of Data Set

Evaluation Measurements	Regular Data Set	Standardized Data Set
Accuracy Score	XGBoost	XGBoost
Logloss	XGBoost	Logistic Regression, SVM, XGBoost, Neural Network
Over Both Measurements	XGBoost	XGBoost

Chapter 6 Conclusion

In this thesis, five learning algorithms were compared statistically for classification performance of malicious files. The comparison was based on two measures of performance of the algorithms on two forms of data. A data set was extracted from the collection of binary files and disassembled files released by Microsoft on Kaggle.com. During extraction, hexadecimal features and assembly features were considered. In all, 13 categories of features were extracted, which provided 1801 features in total. From this data set, a standardized data set was generated. The five learning algorithms to be compared were implemented on the two data sets and assessed for accuracy score and logloss. The interaction effect between data set and algorithm was tested statistically using two-way MANOVA. After the no interaction hypothesis was rejected, interaction plots were employed as a graphical tool to exhibit the effects of data sets and algorithms on accuracy score and logloss separately. Then, ANOVAs (one for each performance measures) were conducted to test statistically the effects of data set and algorithm. After the results of the individual ANOVAs showed the presence of interaction effects, Tukey's test was employed for multiple comparisons among the combinations of data sets and algorithms, and suggested algorithms for a given performance measurement and data set were obtained.

For the algorithms and measurements discussed in this thesis, when using accuracy score as a performance measurement, XGBoost gave the highest scores on both the regular data set and the standardized data set. For the measurement of logloss, XGBoost also had the best performance on the regular data set, but on the standardized data set, Logistic Regression, SVM, XGBoost, Neural Network performed without significant difference.

In practice, any of the learning algorithms may be selected for malware classification on different types of data sets. These algorithms may have a better performance for a specific measurement or overall on several measurements. This thesis showed that the performance of malware classification depended on the combination of type of data set and the algorithms.

In summary, the contribution of this thesis are the following:

- The thesis verified the feature extraction method by debugging Python code exceeding 1000 lines (Ahmadi, et al. 2016a), which effectively reduced the raw data from 250GB to a data set of less than 100MB.
- This thesis provides a robust methodology for finding a suitable learning algorithm for malware classification on a huge collection of malicious files.
- Algorithms for Logistic Regression, Nearest Neighbors, Support Vector Machine, Boosted Trees and Neural Network were evaluated statistically on accuracy score and logloss. A recommendation of algorithm to use for given performance measurement and data set is given.

It should be noted that when the algorithms were employed on the data sets, the default parameter setting was generally used. The Python code for the algorithms are attached in Appendix: Python Codes for Accuracy Scores and Logloss. Practically, when an algorithm is applied to some specific data sets, the parameters will be adjusted for an optimal result. It is very probable that other conclusions would be reached with different parameter settings. Consequently, the results in the thesis should be interpreted with this in mind.

Although these algorithms generally obtained satisfying classification performance, unfortunately, obfuscation techniques to escape detection can affect the results of malware classification. Obfuscation techniques are used for hiding malware in itself (O'Kane, Sezer and Kieran 2011). Major obfuscating methods include metamorphism and polymorphism. The former edits a new version of code in every generation when it spreads, and the latter changes its code for every iteration. Static analysis inherently cannot overcome interference from obfuscation, therefore, false detections are possibly caused by metamorphism and polymorphism. Evaluation of learning algorithms based on dynamic analysis should be researched in the future.

Reference

Abou-Assaleh, Tony, Nick Cercone, Vlado Keselj, and Ray Sweidan. 2004. "N-gram-based Detection of New Malicious Code." *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International.* IEEE. 41-42.

Aha, David W., Dennis Kibler, and Marc K. Albert. 1991. "Instance-based Learning Algorithms." *Machine Learning* 6 (1): 37-66.

Ahmadi, Mansour, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. 2016a. *Microsoft Malware Challenge.* Accessed Apirl 2017. https://github.com/ManSoSec/Microsoft-Malware-Challenge.

—. 2016b. "Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification." *CODASPY '16 Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy.* ACM. 183-194.

Bauer, Eric, and Ron Kohavi. 1999. "An Empirical Comparison of Voting Classification Algorithms: Bagging, Boosting, and Variants." *Machine Learning* 36 (1): 105-139.

Baysa, Donabelle, Richard M. Low, and Mark Stamp. 2013. "Structural Entropy and Metamorphic Malware." *Journal of Computer* 9 (4): 179-192.

Bilar, Daniel. 2007. "Opcodes as Predictor for Malware." *International Journal of Electronic Security and Digital Forensics* 1 (2): 156-168.

Bishop, Christopher M. 1995. *Neural Networks for Pattern Recognition.* Oxford University Press.

Cesare, Silvio, and Yang Xiang. 2010. "Classification of Malware Using Structured Control Flow." *AusPDC '10 Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing.* Brisbane, Australia: Australian Computer Society, Inc. 61-70.

Chen, Tianqi. 2014. "Introduction to Boosted Trees." *University of Washington.* October 22. Accessed Apirl 2017. https://homes.cs.washington.edu/~tqchen/pdf/BoostedTree.pdf.

Chen, Tianqi, and Carlos Guestrin. 2016. "XGBoost: A Scalable Tree Boosting System." *KDD '16 Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* San Francisco, California, USA: ACM. 785-794.

Choi, Yang-seo, Ik-kyun Kim, Jin-tae Oh, and Jae-cheol Ryou. 2008. "PE File Header Analysis-based Packed PE File Detection Technique (PHAD)." *Computer Science and its Applications, 2008. CSA '08. International Symposium on.* IEEE. 28-31. doi:10.1109/CSA.2008.28.

Christodorescu, Mihai, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E Bryant. 2005. "Semantics-Aware Malware Detection." *Security and Privacy, 2005 IEEE Symposium on.* IEEE. 32-46.

Coelho, Luis Pedro. 2013. "Mahotas: Open source software for scriptable computer vision." *Journal of Open Research Software* 1 (1): e3. doi:http://doi.org/10.5334/jors.ac.

Cortes, Corinna, and Vladimir Vapnik. 1995. "Support-Vector Networks." *Machine Learning* 20 (3): 273-297.

Cox, D. R. 1958. "The Regression Analysis of Binary Sequences." *Journal of the Royal Statistical Society* 20 (2): 215-242.

Dahl, George E., Jack W. Stokes, Li Deng, and Dong Yu. 2013. "Large-Scale Malware Classification Using Random Projections and Neural Networks." *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on.* IEEE. 3422-3426.

Elovici, Yuval, Asaf Shabtai, Robert Moskovitch, Gil Tahan, and Chanan Glezer. 2007. "Applying Machine Learning Techniques for Detection of Malicious Code in Network Traffic." *Annual Conference on Artificial Intelligence.* Springer. 44-50.

Firdausi, Ivan, Charles lim lim, Alva Erwin, and Anto Satriyo Nugroho. 2010. "Analysis of Machine learning Techniques Used in Behavior-Based Malware Detection." *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on.* IEEE. 201-203.

Friedman, Jerome H. 2001. "Greedy Function Approximation: A Gradient Boosting Machine." *The Annals of Statistics* 29 (5): 1189-1232.

Gandotra, Ekta, Divya Bansal, and Sanjeev Sofat. 2014. "Malware Analysis and Classification: A Survey." *Journal of Information Security* 5 (2): 56-64. doi:10.4236/jis.2014.52006.

Huynh, Cam-Loi. n.d. "Estimation of Type III Error and Power for Directional Two-Tailed." *SAS.com.* Accessed May 2017. http://www2.sas.com/proceedings/sugi30/208-30.pdf.

Joachims, Thorsten. 1998. "Making Large-Scale Support Vector Machine Learning Practical." In *Advances in Kernel Methods - Support Vector Learning*, edited by Bernhard Schölkopf, Christopher J.C. Burges and Alexander J. Smola, 169-184. Cambridge, USA: MIT Press.

John, George H., and Pat Langley. 1995. "Estimating Continuous Distributions in Bayesian Classifiers." *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence.* Morgan Kaufmann Publishers Inc. 338-345.

Knerr, Stefan, Léon Personnaz, and Gérard Dreyfus. 1990. "Single-layer Learning Revisited: a Stepwise Procedure for Building and Training a Neural Network." *Neurocomputing.* Springer Berlin Heidelberg. 41-50.

Kolter, Jeremy Z., and Marcus A. Maloof. 2004. "Learning to Detect Malicious Executables in the Wild." *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM. 470-478.

Mahotas Contributors. 2016. "Mahotas Features." *Mahotas: Computer Vision in Python.* Accessed Apirl 2017. http://mahotas.readthedocs.io/en/latest/features.html.

McAfee Labs. 2016. "McAfee Labs Quaterly Threat Report." December. Accessed April 2017. https://www.mcafee.com/au/resources/reports/rp-quarterly-threats-dec-2016.pdf.

Menahem, Eitan, Asaf Shabtai, Lior Rokach, and Yuval Elovici. 2009. "Improving Malware Detection by Applying Multi-inducer Ensemble." *Computational Statistics & Data Analysis* 53 (4): 1483-1494.

Microsoft. 2015. *Malware Classification.* Accessed November 2016. https://www.kaggle.com/c/malware-classification.

Moskovitch, Robert, Clint Feher, Nir Tzachar, Eugene Berger, Marina Gitelman, Shlomi Dolev, and Yuval Elovici. 2008. "Unknown Malcode Detection Using OPCODE Representation." *European conference on Intelligence and Security Informatics.* Springer. 204-215.

Moskovitch, Robert, Dima Stopel, Clint Feher, Nir Nissim, and Yuval Elovici. 2008. "Unknown Malcode Detection via Text Categorization and the Imbalance Problem." *Intelligence and Security Informatics, 2008. ISI 2008. IEEE International Conference on.* IEEE. 156-161.

Nataraj, Lakshmanan, S. Karthikeyan, Gregoire Jacob, and B. S. Manjunath. 2011. "Malware Images: Visualization and Automatic Classification." *Proceedings of the 8th International Symposium on Visualization for Cyber Security.* ACM. p. 4.

O'Kane, Philip, Sakir Sezer, and McLaughlin Kieran. 2011. "Obfuscation: The Hidden Malware." *IEEE Security & Privacy* 9 (5): 41 - 47.

Pearl, Judea. 1987. "Evidential Reasoning Using Stochastic, Simulation of Causal Models." *Artificial Intelligence* 32 (2): 245-257.

Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, et al. 2011. "Scikit-learn: Machine Learning in Python." *Journal of Machine Learning Research* 12: 2825-2830.

Quinlan, Ross J. 1993. *C4.5: Programs for Machine Learning.* Morgan Kaufmann.

Quinlan, Ross J. 1987. "Simplifying Decision Trees." *International Journal of Man-Machine Studies* 27 (3): 221-234.

Rédei, George P. 2008. "Angular Transformation (Arcsine Transformation)." In *Encyclopedia of Genetics, Genomics, Proteomics and Informatics*, 98. Springer Netherlands.

Schapire, Robert E. 1999. "A Brief Introduction to Boosting." *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence.* Elsevier B.V. 1401-1406.

Schultz, Matthew G., Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo. 2001. "Data Mining Methods for Detection of New Malicious Executables." *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on.* IEEE. 38-49.

Scikit-learn Developers. 2016. "scikit-learn User Guide 0.18.1." *scikit-learn: Machine Learning in Python.* Accessed Apirl 2017. http://scikit-learn.org/stable/user_guide.html.

Shabtai, Asaf, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. 2009. "Detection of Malicious Code by Applying Machine Learning Classifiers on Static Features: A State-of-the-Art Survey." *Information Security Technical Report* 14 (1): 16-29.

Shannon, Claude E. 2001. "A Mathematical Theory of Communication." *ACM SIGMOBILE Mobile Computing and Communications Review* 5 (1): 3-55.

Siddiqui, Muazzam, Morgan C. Wang, and Joohan Lee. 2008. "Data Mining Methods for Malware Detection Using Instruction Sequences." *Proceedings of the IASTED International Conference on Artificial Intelligence and Applications, AIA 2008.* Elsevier B.V. 358-363.

von Seggern, David H. 2007. *CRC Standard Curves and Surfaces with Mathematica, 2d ed.(CD-ROM included).* Portland, OR, US: Book News, Inc.

XGBoost Contributors. 2015. *Python Package xgboost.* Accessed Apirl 2017. https://github.com/dmlc/xgboost/tree/master/python-package.

Appendix: Python Codes for Accuracy Scores and Logloss

```
import os, numpy, math, random
import xgboost as xgb
from pandas.io.parsers import read_csv
from sklearn.metrics import accuracy_score, log_loss
from sklearn.model_selection import KFold
from sklearn import linear_model, svm
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier

def get_current_directory():
    current_path = os.getcwd()
    return current_path

class dataSetInformaion:
   'Common base class for all employees'
   dataSetName = list()
   rowsName = None
   data = None
   classLabel = None

   def __init__(self, dataSetName, data, classLabel):
      self.dataSetName = dataSetName
      self.data = data
      self.classLabel = classLabel

DATASET_PATH = get_current_directory()[0:get_current_directory().rindex('/')] +
'/Dataset/'
TRAIN_ID_PATH = DATASET_PATH +'trainLabels.csv'
COMBINED_PATH_CSV = DATASET_PATH + 'combination/'

savePath = COMBINED_PATH_CSV

class_label = read_csv(TRAIN_ID_PATH, delimiter=',')
new_idx = numpy.argsort(class_label.ix[:,0])
class_label = class_label.ix[new_idx,1]
class_label = class_label.reset_index(drop=True)


# Read the combined datasets
os.chdir(savePath)
std_jointFile = read_csv('/home/luzhu/classify/Dataset/combination/std_NewTrain.csv',
delimiter=',')
std_joint = dataSetInformaion('Standardized Dataset', std_jointFile,
class_label[0:std_jointFile.shape[0]])

reg_jointFile = read_csv('/home/luzhu/classify/Dataset/combination/reg_NewTrain.csv',
delimiter=',')
reg_joint = dataSetInformaion('Regular Dataset', reg_jointFile,
class_label[0:reg_jointFile.shape[0]])

print 'Run cross-validation ...'

print std_jointFile.shape
print type(std_jointFile)
print reg_jointFile.shape
print type(reg_jointFile)

for i in range(0, 50):
    print 'Regular DataSet ========================== ', 'cross validation result'
```

```
    tr_accuraciesMeans = []
    loglossesMeans = []

    rng = random.randint(0, 65535)
    print 'rng is', rng

    print 'Logistic Regression ========================== ', 'cross validation result'
    accuracies = []
    logLosses = []
    kF = KFold(n_splits=5, shuffle=True, random_state=rng)
    for trainIndex, testIndex in kF.split(reg_joint.data):
        trainKF = reg_joint.data.ix[trainIndex, :]
        trainID = reg_joint.classLabel.ix[trainIndex]
        myModel = linear_model.LogisticRegression(multi_class='multinomial',
class_weight='balanced', solver='sag', max_iter=10000)
        myFit = myModel.fit(trainKF, trainID)
        actualLabels = reg_joint.classLabel.ix[testIndex]

        predictProbability = myFit.predict_proba(reg_joint.data.ix[testIndex, :])
        logLoss = log_loss(actualLabels, predictProbability)
        logLosses.append(logLoss)

        predictedLabels = myFit.predict(reg_joint.data.ix[testIndex, :])
        acc = accuracy_score(actualLabels, predictedLabels)
        accuracies.append(acc)
    accuraciesMean = numpy.mean(accuracies)
    loglossesMean = numpy.mean(logLosses)
    tr_accuraciesMean = math.asin(math.sqrt(accuraciesMean))
    tr_accuraciesMeans.append(tr_accuraciesMean)
    loglossesMeans.append(loglossesMean)

    print 'K Nearest Neighbors ========================== ', 'cross validation result'
    accuracies = []
    logLosses = []

    kF = KFold(n_splits=5, shuffle=True, random_state=rng)
    for trainIndex, testIndex in kF.split(reg_joint.data):
        trainKF = reg_joint.data.ix[trainIndex, :]
        trainID = reg_joint.classLabel.ix[trainIndex]
        myModel = KNeighborsClassifier(n_neighbors = 34)
        myFit = myModel.fit(trainKF, trainID)
        actualLabels = reg_joint.classLabel.ix[testIndex]

        predictProbability = myFit.predict_proba(reg_joint.data.ix[testIndex, :])
        logLoss = log_loss(actualLabels, predictProbability)
        logLosses.append(logLoss)

        predictedLabels = myFit.predict(reg_joint.data.ix[testIndex, :])
        acc = accuracy_score(actualLabels, predictedLabels)
        accuracies.append(acc)

    accuraciesMean = numpy.mean(accuracies)
    loglossesMean = numpy.mean(logLosses)
    tr_accuraciesMean = math.asin(math.sqrt(accuraciesMean))
    tr_accuraciesMeans.append(tr_accuraciesMean)
    loglossesMeans.append(loglossesMean)

    print 'Support Vector Machine ========================== ', 'cross validation
result'
    accuracies = []
    logLosses = []

    kF = KFold(n_splits=5, shuffle=True, random_state=rng)
```

```
    for trainIndex, testIndex in kF.split(reg_joint.data):
        trainKF = reg_joint.data.ix[trainIndex, :]
        trainID = reg_joint.classLabel.ix[trainIndex]
        myModel = svm.SVC(decision_function_shape='ovo', probability=True,
class_weight='balanced')
        myFit = myModel.fit(trainKF, trainID)
        actualLabels = reg_joint.classLabel.ix[testIndex]

        predictProbability = myFit.predict_proba(reg_joint.data.ix[testIndex, :])
        logLoss = log_loss(actualLabels, predictProbability)
        logLosses.append(logLoss)

        predictedLabels = myFit.predict(reg_joint.data.ix[testIndex, :])
        acc = accuracy_score(actualLabels, predictedLabels)
        accuracies.append(acc)

    accuraciesMean = numpy.mean(accuracies)
    loglossesMean = numpy.mean(logLosses)
    tr_accuraciesMean = math.asin(math.sqrt(accuraciesMean))
    tr_accuraciesMeans.append(tr_accuraciesMean)
    loglossesMeans.append(loglossesMean)

    print 'XGBoost ========================== ', 'cross validation result'
    accuracies = []
    logLosses = []

    kF = KFold(n_splits=5, shuffle=True, random_state=rng)
    for trainIndex, testIndex in kF.split(reg_joint.data):
        trainKF = reg_joint.data.ix[trainIndex, :]
        trainID = reg_joint.classLabel.ix[trainIndex]
        myModel = xgb.XGBClassifier()
        myFit = myModel.fit(trainKF, trainID)
        actualLabels = reg_joint.classLabel.ix[testIndex]

        predictProbability = myFit.predict_proba(reg_joint.data.ix[testIndex, :])
        logLoss = log_loss(actualLabels, predictProbability)
        logLosses.append(logLoss)

        predictedLabels = myFit.predict(reg_joint.data.ix[testIndex, :])
        acc = accuracy_score(actualLabels, predictedLabels)
        accuracies.append(acc)

    accuraciesMean = numpy.mean(accuracies)
    loglossesMean = numpy.mean(logLosses)
    tr_accuraciesMean = math.asin(math.sqrt(accuraciesMean))
    tr_accuraciesMeans.append(tr_accuraciesMean)
    loglossesMeans.append(loglossesMean)

    print 'Neural Network ========================== ', 'cross validation result'
    accuracies = []
    logLosses = []

    kF = KFold(n_splits=5, shuffle=True, random_state=rng)
    for trainIndex, testIndex in kF.split(reg_joint.data):
        trainKF = reg_joint.data.ix[trainIndex, :]
        trainID = reg_joint.classLabel.ix[trainIndex]
        myModel = MLPClassifier()
        myFit = myModel.fit(trainKF, trainID)
        actualLabels = reg_joint.classLabel.ix[testIndex]

        predictProbability = myFit.predict_proba(reg_joint.data.ix[testIndex, :])
        logLoss = log_loss(actualLabels, predictProbability)
        logLosses.append(logLoss)
```

```
        predictedLabels = myFit.predict(reg_joint.data.ix[testIndex, :])
        acc = accuracy_score(actualLabels, predictedLabels)
        accuracies.append(acc)

    accuraciesMean = numpy.mean(accuracies)
    loglossesMean = numpy.mean(logLosses)
    tr_accuraciesMean = math.asin(math.sqrt(accuraciesMean))
    tr_accuraciesMeans.append(tr_accuraciesMean)
    loglossesMeans.append(loglossesMean)

    f = open('/home/luzhu/classify/Dataset/alg_obs.txt', 'a')
    f.write('tr_acc_reg')
    for i in range(0, len(tr_accuraciesMeans)):
        f.write(' ' + str(tr_accuraciesMeans[i]))
    f.write('\n')
    f.write('logloss_reg' + ' ')
    for i in range(0, len(loglossesMeans)):
        f.write(' ' + str(loglossesMeans[i]))
    f.write('\n')
    f.close()


    print 'Standardizaed DataSet ========================== ', 'cross validation
result'
    tr_accuraciesMeans = []
    loglossesMeans = []

    print 'Logistic Regression ========================== ', 'cross validation result'
    accuracies = []
    logLosses = []
    kF = KFold(n_splits=5, shuffle=True, random_state=rng)
    for trainIndex, testIndex in kF.split(std_joint.data):
        trainKF = std_joint.data.ix[trainIndex, :]
        trainID = std_joint.classLabel.ix[trainIndex]
        myModel = linear_model.LogisticRegression(multi_class='multinomial',
class_weight='balanced', solver='sag', max_iter=10000)
        myFit = myModel.fit(trainKF, trainID)
        actualLabels = std_joint.classLabel.ix[testIndex]

        predictProbability = myFit.predict_proba(std_joint.data.ix[testIndex, :])
        logLoss = log_loss(actualLabels, predictProbability)
        logLosses.append(logLoss)

        predictedLabels = myFit.predict(std_joint.data.ix[testIndex, :])
        acc = accuracy_score(actualLabels, predictedLabels)
        accuracies.append(acc)

    accuraciesMean = numpy.mean(accuracies)
    loglossesMean = numpy.mean(logLosses)
    tr_accuraciesMean = math.asin(math.sqrt(accuraciesMean))
    tr_accuraciesMeans.append(tr_accuraciesMean)
    loglossesMeans.append(loglossesMean)

    print 'K Nearest Neighbors ========================== ', 'cross validation result'
    accuracies = []
    logLosses = []

    kF = KFold(n_splits=5, shuffle=True, random_state=rng)
    for trainIndex, testIndex in kF.split(std_joint.data):
        trainKF = std_joint.data.ix[trainIndex, :]
        trainID = std_joint.classLabel.ix[trainIndex]
        myModel = KNeighborsClassifier(n_neighbors=34)
```

```
        myFit = myModel.fit(trainKF, trainID)
        actualLabels = std_joint.classLabel.ix[testIndex]

        predictProbability = myFit.predict_proba(std_joint.data.ix[testIndex, :])
        logLoss = log_loss(actualLabels, predictProbability)
        logLosses.append(logLoss)

        predictedLabels = myFit.predict(std_joint.data.ix[testIndex, :])
        acc = accuracy_score(actualLabels, predictedLabels)
        accuracies.append(acc)

    accuraciesMean = numpy.mean(accuracies)
    loglossesMean = numpy.mean(logLosses)
    tr_accuraciesMean = math.asin(math.sqrt(accuraciesMean))
    tr_accuraciesMeans.append(tr_accuraciesMean)
    loglossesMeans.append(loglossesMean)

    print 'Support Vector Machine ========================== ', 'cross validation
result'
    accuracies = []
    logLosses = []
    kF = KFold(n_splits=5, shuffle=True, random_state=rng)
    for trainIndex, testIndex in kF.split(std_joint.data):
        trainKF = std_joint.data.ix[trainIndex, :]
        trainID = std_joint.classLabel.ix[trainIndex]
        myModel = svm.SVC(decision_function_shape='ovo', probability=True,
class_weight='balanced')
        myFit = myModel.fit(trainKF, trainID)
        actualLabels = std_joint.classLabel.ix[testIndex]

        predictProbability = myFit.predict_proba(std_joint.data.ix[testIndex, :])
        logLoss = log_loss(actualLabels, predictProbability)
        logLosses.append(logLoss)

        predictedLabels = myFit.predict(std_joint.data.ix[testIndex, :])
        acc = accuracy_score(actualLabels, predictedLabels)
        accuracies.append(acc)

    accuraciesMean = numpy.mean(accuracies)
    loglossesMean = numpy.mean(logLosses)
    tr_accuraciesMean = math.asin(math.sqrt(accuraciesMean))
    tr_accuraciesMeans.append(tr_accuraciesMean)
    loglossesMeans.append(loglossesMean)

    print 'XGBoost ========================== ', 'cross validation result'
    accuracies = []
    logLosses = []

    kF = KFold(n_splits=5, shuffle=True, random_state=rng)
    for trainIndex, testIndex in kF.split(std_joint.data):
        trainKF = std_joint.data.ix[trainIndex, :]
        trainID = std_joint.classLabel.ix[trainIndex]
        myModel = xgb.XGBClassifier()
        myFit = myModel.fit(trainKF, trainID)
        actualLabels = std_joint.classLabel.ix[testIndex]

        predictProbability = myFit.predict_proba(std_joint.data.ix[testIndex, :])
        logLoss = log_loss(actualLabels, predictProbability)
        logLosses.append(logLoss)

        predictedLabels = myFit.predict(std_joint.data.ix[testIndex, :])
        acc = accuracy_score(actualLabels, predictedLabels)
        accuracies.append(acc)
```

```
accuraciesMean = numpy.mean(accuracies)
loglossesMean = numpy.mean(logLosses)
tr_accuraciesMean = math.asin(math.sqrt(accuraciesMean))
tr_accuraciesMeans.append(tr_accuraciesMean)
loglossesMeans.append(loglossesMean)

print 'Neural Network ========================== ', 'cross validation result'
accuracies = []
logLosses = []

kF = KFold(n_splits=5, shuffle=True, random_state=rng)
for trainIndex, testIndex in kF.split(std_joint.data):
    trainKF = std_joint.data.ix[trainIndex, :]
    trainID = std_joint.classLabel.ix[trainIndex]
    myModel = MLPClassifier()
    myFit = myModel.fit(trainKF, trainID)
    actualLabels = std_joint.classLabel.ix[testIndex]

    predictProbability = myFit.predict_proba(std_joint.data.ix[testIndex, :])
    logLoss = log_loss(actualLabels, predictProbability)
    logLosses.append(logLoss)

    predictedLabels = myFit.predict(std_joint.data.ix[testIndex, :])
    acc = accuracy_score(actualLabels, predictedLabels)
    accuracies.append(acc)

accuraciesMean = numpy.mean(accuracies)
loglossesMean = numpy.mean(logLosses)
tr_accuraciesMean = math.asin(math.sqrt(accuraciesMean))
tr_accuraciesMeans.append(tr_accuraciesMean)
loglossesMeans.append(loglossesMean)

f = open('/home/luzhu/classify/Dataset/alg_obs.txt', 'a')
f.write('tr_acc_std')
for i in range(0, len(tr_accuraciesMeans)):
    f.write(' ' + str(tr_accuraciesMeans[i]))
f.write('\n')
f.write('logloss_std' + ' ')
for i in range(0, len(loglossesMeans)):
    f.write(' ' + str(loglossesMeans[i]))
f.write('\n')
f.close()
```