INTERNATIONAL
HELLENIC
UNIVERSITY

Machine Learning and Images for Malware Detection and Classification

Project: MaLiC

Konstantinos Kosmidis

SID: 3307150005

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Communications and Cybersecurity

DECEMBER 2016

THESSALONIKI – GREECE

Machine Learning and Images for Malware Detection and Classification

Project: MaLiC

Konstantinos Kosmidis

SID: 3307150005

Supervisor: Assist. Prof. Christos Kalloniatis

Supervising Committee Members: Professor Sokratis Katsikas. Acad. Assoc. Marios Gatzianas

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Communications and Cybersecurity

DECEMBER 2016

THESSALONIKI – GREECE

Abstract

This dissertation is an introduction to machine learning techniques for malware detection and classification.

The first chapter describes the past and current status of malware analysis providing basic definitions and input from the respective literature. In the second section the various types of malware, which can disastrously affect a Microsoft Windows operating system are presented. In addition, with an explanation and an introduction to malware detection and its techniques are described. The third chapter identifies and describes the role and the goal of artificial intelligence in malware detection and more precisely deep learning in malware detection. After a discussion of the malware detection's goals, an explanation of clustering and classification algorithms used in the dissertation will be presented along with the respective theoretical background. In chapters four and five the experiment set up will be presented along with the respective data sets and the expected outcomes of the research. Also, the results from every category of testing (classification and clustering) will be presented and discussed. Finally, conclusions that were raised from this dissertation, potential improvements and expansions of the tools made will be submitted in chapter six.

Acknowledgments

I would like to thank my supervisor Assistant Professor Christos Kalloniatis for his support, advice and company throughout my dissertation as I shared with him my moments of happiness and because he gave me the opportunity to work on something that I like and I love, information security and more precisely machine learning on malware analysis field. Finally, my family for supporting me with their behavior, advice and financially and gave me the appropriate ethics and personality to become what I love and to keep moving forward.

Konstantinos Kosmidis December 2016

Contents

ABSTRACT **3**

ACKNOWLEDGMENTS **4**

CONTENTS **5**

1 INTRODUCTION **8**

1.1 CURRENT SITUATION 8

1.2 STATIC MALWARE ANALYSIS 10

1.3 DYNAMIC MALWARE ANALYSIS 11

2 MALWARE **13**

2.1 WHAT IS MALWARE? 13

2.2 WHAT IS CLEANWARE? 13

2.3 BEHAVIOR-BASED DETECTION 13

2.4 SIGNATURE-BASED DETECTION 14

2.5 WHAT IS CLASSIFICATION? 15

2.6 MALWARE FAMILIES 17

2.7 NAMING MALWARE 17

2.8 TYPES OF MALWARE 19

2.8.1 Backdoor *19*

2.8.2 Botnet *19*

2.8.3 Downloader *19*

2.8.4 Information-stealing malware *19*

2.8.5 Rootkit *19*

2.8.6 Ransomware *20*

2.8.7 Worm & Virus *20*

2.8.8 Reverse Shell *20*

2.8.9 RAT – Remote Access Trojan *20*

2.8.10 Browser Hijacker *21*
2.8.11 Bootkit *21*
2.8.13 Spam Sending Malware *21*
2.9 MALWARE DETECTION 23
2.9.1 An introduction to Malware Detection and Deep learning *23*
2.9.2 Basic theory in Sandboxing *24*

3 THE ROLE OF ARTIFICIAL INTELLIGENCE IN MALWARE DETECTION 25

3.1 LITERATURE REVIEW ON ARTIFICIAL INTELLIGENT MALWARE DETECTION 25
3.2 FEATURE ENGINEERING EXPLANATION 30
3.3 HOW TO CONVERT MALWARE SAMPLES TO DIGITAL IMAGES 32

4 EXPERIMENT SET UP 35

4.1 SOFTWARE AND HARDWARE SPECIFICATIONS 35
4.2 EXPLANATION OF THE DATASET 35
4.3 THEORY OF CLASSIFICATION ALGORITHMS IMPLEMENTED- 38
4.3.1 Support Vector Machines (SVMs) *38*
4.3.2 Perceptron *40*
4.3.3 Multilayer Perceptron *41*
4.3.4 Stochastic Gradient Descent (SGD) *43*
4.3.5 Nearest Centroid *44*
4.3.6 Multinomial Bayes *44*
4.3.7 Decision Trees *45*
4.3.8 Bernoulli Restricted Boltzmann machine (RBM) *47*
4.3.9 Random Trees-Forest *48*
4.4 THEORY OF CLUSTERING ALGORITHMS IMPLEMENTED 48
4.4.1 Meanshift Clustering *48*
4.4.2 DBScan Algorithm *49*
4.4.3 KMeans *50*
4.4.4 MiniBatch KMeans *51*
4.5 GOALS OF THE EXPERIMENT AND COMPARISON CRITERIA 52
4.6 EXPECTED OUTCOMES 52

5 RESULTS OF ALGORITHMS......55

5.1 CLASSIFICATION ALGORITHMS RESULTS......55

5.1.1 Classification Results for Decision Tree Algorithm......58

5.1.2 Classification Results for Support Vector Machine (SVM) Algorithm......61

5.1.3 Classification Results for Nearest Centroid Algorithm......63

5.1.4 Classification Results for Stochastic Gradient Algorithm......65

5.1.5 Classification Results for Perceptron Algorithm......67

5.1.6 Classification Results for Multilayer Perceptron Algorithm......69

5.1.7 Classification Results for Random Forest......71

5.1.8 Classification Results for Multinomial Naive Bayes......72

5.1.9 Classification Results for Bernoulli......73

5.2 CLUSTERING ALGORITHMS RESULTS......75

5.2.1 MeanShift Clustering......75

5.2.2 DBscan Clustering......75

5.2.3 Kmeans and Minibatch Clustering......77

6. DISCUSSION......79

BIBLIOGRAPHY-REFERENCES......81

APPENDIX......89

1 Introduction

1.1 Current Situation

The Internet, nowadays, plays a crucial role in our everyday life. It has become an enormous information and communication network making people do transactions and interactions having thus a growing and increasing global market. While the Internet is growing, services like web banking, e-shopping, communication through the internet and social media are available to people for everyday tasks. On the other hand, there are people that are determined to enhance themselves by imposing novice users that perform transactions using the Internet. Malware is now a program that helps people with malicious intentions to accomplish their goals.

One of the most significant and largest vulnerabilities found, was the Heartbleed-bug, announced in Codenomicon 2014 [1], a bug discovered in the Transport Layer Security (TLS) heartbeat function. This vulnerability enabled attackers and criminals to exploit this vulnerability, therefore, allowing access to web application memory, where potential usernames and passwords, emails, and business critical documents could be stored .

Malware and malicious software are developed, programmed and registered every day. In agreement with Symantec's documentation about Ransomware and Businesses, [2] one of the most recent threats for the organizations and businesses is the ransomware Cryptowall, which locks and encrypts all the programs and files of the infected system, prompting groups or regular users to pay usually in bitcoins in order the information system to be unlocked.

There will always be threats and vulnerabilities, which malware developers and criminals will exploit. Therefore, it is important for security companies to detect the malicious programs and notify businesses and users about potential vulnerabilities. In line with the exponential growth of the Internet, the number of new malware is increasing every day, which has become difficult to analyze manually.

Analyzing the increasing number of malware requires a lot of human resources if done manually. As of 2015, the AV-Test Institute, [3] registers 390,000 new malicious programs every day, which is infeasible to analyze manually. Even more, the malware should be divided into groups or families to which their code and behavior correspond to.

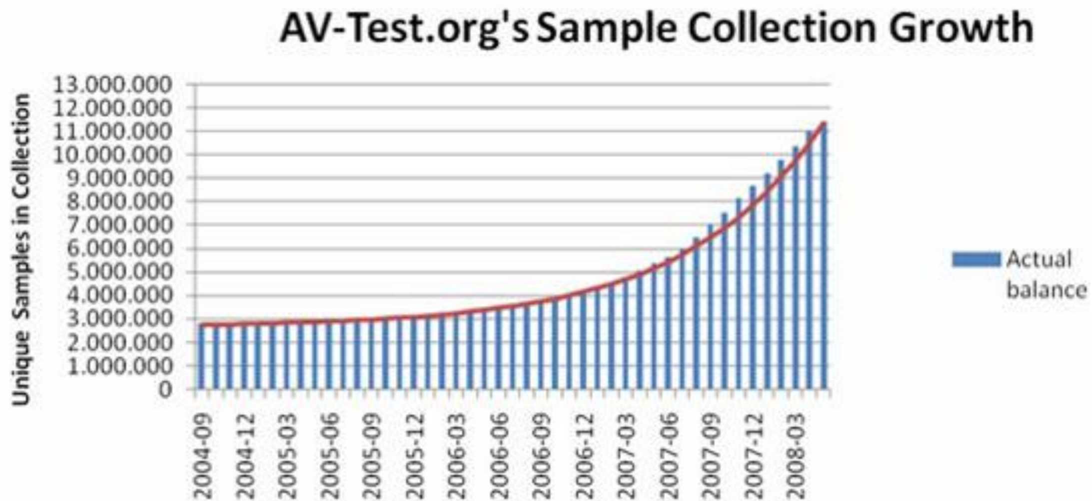


Figure 1: AV-Test's Sample Collection Growth up to 2008. [3]

Current malware development procedures are focused on stealing sensitive data from everyday users and, to a more severe extent, they target critical infrastructures. There are many ways to get infected with malware. Some of them include social engineering approaches that try to deceive users and make them click e-mail attachments.

It is well known that everyone that has an email account has dealt with spam at least once. However, the problem of spam is not always defined as irrelevant content and lack of bandwidth but also is a method to spread malware. Usually, spam emails are using a method known as driven by downloads because they want to make people click on links to websites which cyber criminals have infected with malicious code or open documents that again contain ransomware. This method is also known as spear-phishing attack. On the other hand, there is phishing that comes from spam messages also. Phishing's goal is to redirect their targets to fake websites from which confidential data is then collected.

The malicious software being developed and programmed by intruders, attackers and criminals is polymorphic and has various evasion techniques, meaning that they tend to have the code in such way that they are not detected by intrusion detection systems and

antiviruses. In other words, they use obfuscation techniques reducing the effectiveness of static analysis. Moreover, the different amount of their variants extremely concludes the influence of established protections which usually use static analysis methodologies and approaches and are unable to detect the previously unknown malicious binaries. The variants of malware have in common usual behavioral models reversing their source and intention. Static and dynamic methods retrieve and acquire behavioral models and procedures that can be later used to implement algorithms for detection and classification of unknown malware into recognized malware families using machine learning.

To tackle this problem, researchers have suggested static and dynamic analysis techniques and procedures, which depend on the observation of the behavior of the malware program's activities for detection and classification.

1.2 Static Malware Analysis

Sikorski & Honig, 2012 on their book Practical Malware Analysis [4] and in accordance with [5] and [6] state that:

Basic Static Analysis proposes the examination of an executable file with malicious intentions without viewing the behavior and the instructions on what it does. This method, determines if a file is malicious, and provides information about its construction as well as its unique signature. The basic static analysis is performed using specific software, which has several disadvantages when performed on more sophisticated malware.

Advanced Static Analysis proposes the method of reverse engineering. It is the way of revealing malware's binary and assembly language by feeding the binary into a disassembler, decompiler, and debugger and looking at binary's source code and assembly code to find out services and activities of the executable. So, that way there is a determination of what the program does step-by-step. It is the most challenging and promising part of Static Malware Analysis as solving the assembly behind the malicious software requires disassembly, programming, and specific knowledge of how Windows and Linux(Android) operating system performs.

1.3 Dynamic Malware Analysis

Sikorski & Honig, 2012 on their book Practical Malware Analysis [4] and in accordance with [5] and [6] state also that:

Basic Dynamic Analysis proposes and offers the opportunity of testing, executing and running malicious code and examining on the system to check its behavior, processes and potentially erase the infection. However, it should be mentioned that it is crucial to set up a virtual environment or virtual lab that will let a researcher to study the executed malware without damaging the actual information system or network. Even though typical dynamic analysis is a part of the malware analysis, it has some drawbacks, so the advanced method of dynamic analysis is required.

Advanced Dynamic Analysis employs a disassembler to examine the internal condition and the procedures of an executed malicious file. This technique provides an approach to acquire more detailed data from a malicious program. Similar methods are valuable for obtaining information.

There are two methods for dynamic malware analysis that can be introduced and proposed:

• Examining the dissimilarity between specified states: On this occasion, there are two states. The first state is the infection of the malware and the state after the infection. It is crucial to know how the information system was in the first state to be able to extract information about the malware while it is running. Finally, a report of the states comparing each other of the behaviors of the malware are presented.

• Monitoring running activities and services: Where, malicious programs executed are observed. More details what a researcher examines and finds through a dynamic analysis procedure.

- **RAM analysis:** There are times that malware does acts like buffer overflow, or tries to find ways to access individual processes through RAM.
- **Files modifications:** It is important to have a list of all system files before the actual infection of the system. There are malware that change or delete files. So keeping a list allow us to realize which files have been added, deleted or modified.

- **Processes and system services:** The aim is to detect if new services or procedures have been started or if something changed to processes that are already running. For example, recent evidence suggests that most of the times malware try to bypass any antivirus program that is on their way.
- **Systems changes:** These modifications happen in registry so while investigating registry and log files an examiner can discover the purpose of this malicious file
- **Search for weird URL destinations:** As analysts monitor the network, they try to find evidence that may lead to malicious websites, so unknown IP addresses should be checked through sites like virus total, sucuri and more.

However, dynamic analysis of malware must be performed in an environment that researchers are willing to sacrifice, and that is logically partitioned from other hosts on network (and, hopefully, the rest of the world). A reasonably complete overview of the behavior of a Windows program can be achieved by just monitoring its interaction with the file system, the registry, other processes, and the network.

Both of the above techniques have their advantages and disadvantages. The static analysis suggests a full inclusion and report, but sometimes it suffers from code obfuscation. The binary file should be processed accordingly before examination with a common technique called unpacking, having as a result to make researchers encounter unmanageable complexity during the analysis. Dynamic analysis is more useful and does not need the binary to be unpacked or decrypted. On the other hand, it takes time and consumers computing resources, so it raises scalability concerns. Furthermore, there are many malicious activities that might be without monitoring because there is no such a state or situation in order to generate the appropriate circumstances.

This dissertation takes a totally different path to characterize, label and analyze malware. In more general terms, a malware executable can be depicted and described as a binary string of zeros and ones. This vector can be modified into a matrix and exported and converted into an image. Nataraj et al., 2011 suggested an approach and methodology where meaningful and relevant visual similarities exist in image structure of malware and can be used to classify them to their known malware families. Existing classification methods require either disassembly or execution whereas our approach does not require

any of the two but still shows significant improvement regarding performance. Finally, it is also resilient to favorite obfuscation techniques such as section encryption.

2. Malware

2.1 What is Malware?

According to [5] and [6] malware is a program that has malicious intentions and it is made and designed for a certain purpose, to acquire access to an information system without the administrator's authorization. With more simple words malware is a software that helps an attacker complete and fulfills his malicious and crime intentions.

2.2 What is Cleanware?

Cleanware is that kind of software whose activity is not considered malicious. It is important to separate malware from clean ware, to ensure that an unknown file, is not malicious.

Examples could be:

• Opening attachments in an E-mail.

• Inserting an exterior hard drive or a USB stick to your system.

• Background detection on computer.

• Downloading a file.

2.3 Behavior-based Detection

On behavior-based malware detection, researchers analyze the malware and its behavior during run-time. The reason that sometimes there is a need to use behavior-based detection is to know the actual source code of the malware. Here the source-code can execute different code obfuscation techniques e.g. packing their code, polymorphism, and metamorphism.

2.4 Signature-based Detection

Signature-based malware detection commonly refers to static analysis, where the malware sample is analyzed and unique signatures are extracted. These can then be used to distinguish malware files from good data and is a commonly used method by AV-vendors. The problem arises when dealing with code obfuscation techniques employed by the malware. Some of the obfuscation techniques used against Signature-based Detection are listed below:

• **Packing:** Adam Kujawa on a Malwarebytes 2013 Threat Report [8] informs that when malware developers pack malware, it means that it is compressed into a binary file, in a way it is understandable only if correct decompression is used or reverse engineering techniques and it is used to bypass anti-viruses, firewalls and more. When unpack is done or decompression, the malware is loaded into memory in a human readable form. Malware can be compressed in many ways and even several times making it close to impossible to reverse-engineer the code. As recently has been presented at a Black Hat conference as a presentation [9]

• **Polymorphism:** It is the method where the malicious software has a part of its code changed after every iteration it runs on the computer, while another part remains the same.

• **Metamorphism:** It is the method that all the code of the malware is changed while it runs on an information system, but functionality is still the same.

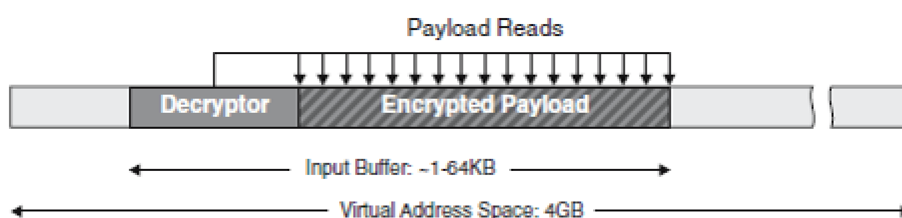


Figure 2: Malware Development [4-6]

2.5 What is Classification and Clustering?

Classification is a technique under which an object needs to be identified and be categorized to a class by scientists. This dissertation provides predefined and known types, labels and categories which separate this definition from that of clustering.

As the amount of malware samples are huge, it is easily understandable that humans are slow classifiers, so there is a need to automate the classification processes.

The goal of classification in the field of machine learning is to present as a map new input variables or samples after training to a discrete output variable or label. To perform this generalization task, the classifier, often represented as a black-box module, is first trained using a set of labeled input/output data. Plenty of models exist to represent classifiers. Among them, the most popular ones include decision trees, the naive and general Bayes classifiers, random forest, artificial neural networks and other kernel related techniques.

In data science outlier detection is the labeling of unknown information and data such as features, incidents, attacks or activities and services which do not belong to a known group and cluster in a dataset. More often the data for an anomalous detection system are from malware, botnets, etc. Three types of anomaly detection procedures and approaches exist. Unsupervised, Supervised and Semi-supervised anomaly detection techniques. An unsupervised procedure detects outliers in an unsupervised dataset by making the hypothesis that the majority of the data inside the dataset are reasonable and it looks and searches for any data that does not fit to the remaining dataset. A supervised technique has a dataset that is already labeled as" normal" or" abnormal," and it is used to train a classification algorithm to be subsequently used to detect new and similar threats. Finally, in semi-supervised a framework is presented where normal behavior from a given normal training data set is trained, and then the probability of a test supplement is tested and generated by the learned model. The most important anomaly detection techniques that have been introduced by the literature is Density-based techniques, classification and cluster algorithms (k-nearest neighbor, local outlier factor), support vector machines, neural network and computational intelligence and fuzzy logic.

2.5.1 Artificial Neural Networks

This dissertation considers Artificial Neural Networks (ANN) algorithms, which can be linear or nonlinear parametric models. These types are motivated by biological neural networks where neurons are computational units which are activated by weighted connections (axons and dendrites). ANN are broadly studied and applied in engineering and sciences, especially in pattern recognition. They mainly yield accurate results when applied to regression and classification problems.

2.5.2 Classification problem for detection

The classification task in data processing entails separating data entries into groups by assigning them a specific class (or type). The classification problem can be encountered in almost any field of work and there exists a large variety of approaches to find a solution. Some approaches worth mentioning are the decision tree models, the Bayesian classifiers (naive and general), support vector machines where the classification task can be divided into two subtasks. The first one is often referred to as the learning phase when the classifier is created from a set of labeled input/output cases. The second step is often referred to as the testing phase where the classifier is tested on a set of labeled input/output cases not yet encountered during the training period. Once the model built in the two first phases is functional, it can be used to estimate and predict the output class for new and unlabeled input values.

2.5.3 Classification problem statement

The classification problem is easy when the data entries are linearly separable (i.e. they can be separated by a hyperplane). However, it is slightly more difficult when they are not linearly separable. In this case, the data have to be represented using a nonlinear model.

2.6 Malware Families

A malware family can be considered as a group of malware, whose main source code is the same ,that has similar main functionalities, but only their behavior is changed. That is the reason why sometimes new malware are referred to as variants or updated versions of old ones by industry and researchers. For classification, an emphasis is placed on the original and same main features and behavior of the malicious samples even though they might have different practices in general.

In 2014 Microsoft recorded and registered in its database more than 236 malware families. The reason researchers need this registration and have many families and classes is that having more samples means that they have more features for supervised learning providing better results and can focus more on the performance of their algorithms. So, having a database recording these samples could be a benefit for any examiner.

2.7 Naming Malware

According to Microsoft [10] a malware with specific behavior can have more than one names from AV Vendors because they use different methods and ways to call malware and it depends on the number of samples that they collect as well as their particular behavior. One of the most common methods for calling malware samples is the CARO Naming standard.

CARO is an informal malware naming scheme developed by individuals from AV companies and researchers. Note that the CARO naming convention does not solve the problem of ambiguous class labels, but instead, tries to address the inconsistent labeling.

The idea is to create a universal standard, or syntax, for naming malware, to prevent confusion of definitions among , say. AV-vendors and users. The most complex form is as follows:

<malware type>://<platform>/<family name>. <group name>. <infective length>. <sub-variant><devolution><modifiers>

All conventions are optional except for family name since not all entries are necessarily available.

This protocol is used by Microsoft in their AV software namely, MSE or the Win8 version, Windows Defender. For MSE, the scheme used is as following:

<malware type>://<platform>/<family name>. <sub-variant>! <vendor-specific comment>

It is noticed here, that infective length, group name, and devolution is not applied in their convention.

Additionally, the modifiers have been replaced with! vendor-specific comment, which is part of the modifiers parameter also used in CARO. To give an idea of the structure a real example is listed below:

Backdoor: Win32/Caphaw.D

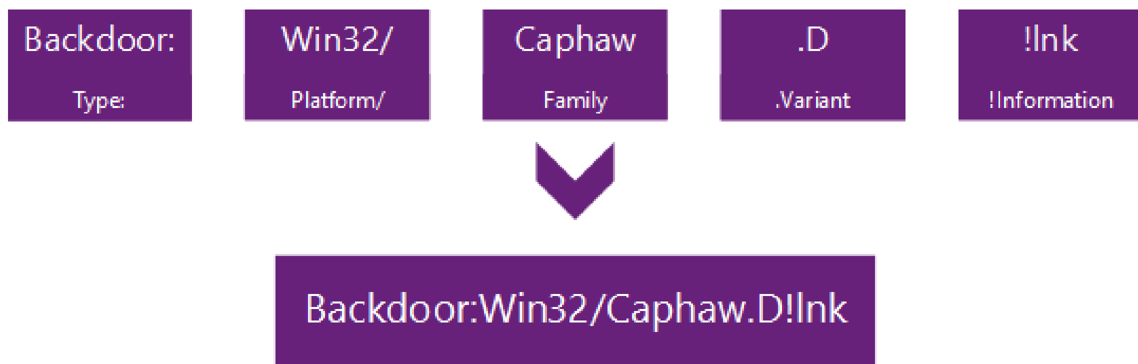


Figure 3. Detailed Explanation of Naming procedure of a malware sample by Microsoft [10]

2.8 Types of Malware

There are several types of malware as reported by [11] and below is an explanation of the essential elements and features of malware types that readers should know and understand.

2.8.1 Backdoor

A program that installs on its own to a computer system and makes "a door" to let attackers connect to the system. Backdoors create, achieve and execute code on the system with little or no authentication.

2.8.2 Botnet

A program similar to backdoor, but with the difference that the information systems affected build a network of bots that receive commands from a server known as command-and-control server.

2.8.3 Downloader

Downloaders are programs embedded in websites, information systems, personal computers whose goal is to download other malicious code

2.8.4 Information-stealing malware

These types of malware also known as keyloggers, password grabbers or sniffers are made to collect information and send this information to somewhere else. These types of programs can be considered and categorized as Riskware as they are safe when used by an authorized person in a suitable activity and status. On the other hand, if misused, or employed by an attacker, the program may affect the security of a person or a system. For example, keyloggers are often used to monitor users. This is the most common attack to acquire access to online banking systems.

2.8.5 Rootkit

Rootkits are the type of malware that is constructed to hide other code and are commonly connected with another malware, such as backdoor. This, allows the attacker to maintain remote access and make detection of the code by investigators difficult..

2.8.6 Ransomware

One of the most common malware designed to run and execute on all operation systems. Their goal is to frighten and make an infected user into buying something. Most of the time it has a user interface with instructions on how to proceed with the payments. It warns users that there is malicious code by using cryptographic algorithms on their personal information systems and that the only way to get rid of it, is to pay with digital currencies. As an exchange, they will deliver the key to decrypt user's system when it does nothing more than stealing people's money or destroying information systems.

2.8.7 Worm & Virus

Malicious code whose goal is to copy itself and infect more computers. Usually, it does not make changes to other programs. Worms, on the other hand, often search for a specific requirement to systems and when they find them they change them. The greatest infection is Stuxnet targeting SCADA systems.

2.8.8 Reverse Shell

A reverse shell provides access to the attacker to the host that previously got infected or permits the connection of an infected system to the attacker. Their functions work as a backdoor on the infected host. The way a reverse shell works is that it gives to the attacker

the ability to execute and type commands as the intruder is local. Windows cmd.exe and Netcat are commonly used for making packaged reverse shells. These methods are used to hide from user's infected information system giving the time to execute commands on the infected host.

2.8.9 RAT – Remote Access Trojan

A Remote Access Trojan (RAT) is a type of malware which gives unauthorized access to an attacker and allows the control of the infected host using a backdoor. Remote Access Trojans are often distributed through free-of-charge software and are sent as an attachment by e-mail.

2.8.10 Browser Hijacker

Browser hijackers are malicious software that is designed and programmed with the aim of changing the homepage, usually the search engine provider. They are often installed through free software, and they target the more novice user that will not consider them as malicious. They are malicious as sometimes are adware or spyware having access to user's online privacy

2.8.11 Bootkit

Another type of a rootkit is bootkit. Its name was taken because it is hidden in the boot sector making it hard to be detected by antiviruses and intrusion detection systems after its infection.

2.8.12 Scareware

A type like ransomware is called scareware that most commonly tries to frighten the infected user making him/her purchase something. It comes as a mail attachment with a blackmail text or an information mail that has to do some steps to remove another virus. Because of these methods, many victims will pay for the software to have the virus removed.

2.8.13 Spam Sending Malware

Spam Sending Malware is malicious software that is a part of a botnet controlled by a command and control server operating as a distributed spam-sending network. This, happens to spread usually another malware or give the computational resources by infecting

another system for malicious activities. Sometimes ISPs take countermeasures against this botnet by disabling the victim's internet connection or marking as spam email addresses.

2.8.14 Potentially Unwanted Program (PUP)

Also, known as Potentially Unwanted Application (PUA), Potentially Unwanted Web Application (PUWA, Popups), Potentially Unwanted Software (PUS). It is usually a software that acts and has an unusual behavior with undesirable and unwanted utilities and functions but does not meet the requirements to be considered as malware. What makes PUPs complicated to be analyzed and classified is that for some people, they are considered useful but malicious for others. Most of the times a PUP can impact productivity, privacy, and security but also it can put unwanted stress on the resources of a system.

- **Unintended impact on productivity:**
  - Upsetting with regard to user experience.
  - Futility.
  - The program acts and behaves unexpected, unwelcome and unauthorized actions, which point to unwanted distractions, lost opportunities or lowered productivity.
  - Many times, operators of the affected systems should perform maintenance and cleaning procedures that take time.
- **Unwanted stress on the device's resources:**
  - Computing resources like Memory, CPU, and hard drive are used more than the usual.
  - Increased Bandwidth.
- **Compromises security:**
  - Publicity and vulnerable to unexpected, controversial and unsubstantiated content, location or applications.

- **Compromises privacy:**
  - Personal information including sensitive software is unnecessarily disclosed to unknown or unauthorized parties.

2.8.15 Spyware:

Collects information about the user's web browsing activities or favorited applications. The data collected are usually sent out to another person.

2.8.16 Trackware:

It provides the utilities in order a user or an information system to be identified by third parties, usually with a unique identifier. The most common trackware is tracking cookies.

2.8.17 Adware:

Distributes malicious code and content through a web browser, PC's Desktop or mobile applications. An alternative name for this type of malware is malvertising using known companies and their advertising banners to distribute malware through them.

2.9 Malware Detection

2.9.1 Introduction to Malware Detection and Deep Learning

Neural networks have attained a reputation throughout the years. Deep learning has performed well, suggesting solutions for advance persistent threats and zero-day protection.

This dissertation aims to discuss countermeasures that should be performed to predict attacks that should do with malware and, to address the importance of machine learning from security point of view.

Current anti-virus software detects a type of malware after its infection or after it has done the damage that it is intent to do. So, detecting malicious code using collected datasets and using neural networks and fuzzy techniques should be presented based on the behavior of their procedures. Recognizing and identifying a problem automatically is a significant problem. Researchers and analysts can examine a small number of files, so the need for large-scale techniques and classification is necessary using neural networks which give us several training algorithms that can be tested for.

The Science of Artificial Intelligence plays a major role in automatic and large-scale malware classification. Machine learning frameworks have been researched, developed and

tested to characterize and categorize malware into their malware families, using characteristics extracted and acquired from static and live analysis of the malicious software. Feature engineering and extraction methodologies require time, which does not scale well to the daily malware samples, binaries that have been analyzed and other malicious software being recorded and submitted for further investigation by researchers who collect and analyze malware or repositories that host malware. So, it is mandatory to search and find new methods for feature engineering and extraction, to perform useful classification algorithms.

Nataraj, Karthikeyan, Jacob, & Manjunath, 2011 in [12] introduced a new approach and method, called binary-texture analysis.

This process should be examined in contrast with similar and existing malware classification approaches previously published. Research results suggest that binary texture analysis provides comparable and similar results regarding accuracy obtained from experiments performed with dynamic analysis procedures. In addition, it provides and produces outcomes faster than the results produced by dynamic procedures. Furthermore, the texture-based methodology and technique shows resilience to packing techniques, and can successfully categorize a significant amount of malware with both encrypted and unencrypted fragments with the difference of considering encrypted samples as a different malware family.

2.9.2 Basic theory in Sandboxing

Oktavianto & Muhardianto, 2013 in [13] state that as technology progresses, malware became more sophisticated, more complicated and harder to analyze. So, there is a need for new ways of prevention, and that would allow us to analyze malware quickly and efficiently without compromising or infecting information systems. Sandboxing has vast applications among industry that belongs to information technology. It is a method of separating a malicious program from the rest of the information system by providing limited execution capabilities. Sandbox, allows analysts to run and execute malicious applications, files, software or codes and see the malware activities and intents. It also maintains a safe and secure environment without worrying about the changes that will take place during the process. There are several malware sandboxes for building automated malware analysis lab like Malheur and Cuckoo Sandbox.

3 The role of Artificial Intelligence in Malware Detection

3.1 Literature review for Artificial Intelligent Malware Detection

Various machine learning algorithms (Perceptron, MeanShift, DBSCAN, etc) have been conducted and developed regarding the detection, identification and classification of unidentified and unrecognized malware into known malware families. Some of these algorithms and methodologies being utilized, are described in this subsection of this chapter.

First, Schultz, Eskin, Zadok, & Stolfo, 2001 [14], applied and extracted three signature-based characteristics for malware classification: Byte sequences Portable Executables (PE) and computer variables most likely strings. The directory of DLLs, function calls and several system calls employed within each DLL used by the file and executable, are reversed engineered and abstracted from DLL records and data that are enclosed to Portable Executable files. Computer variables are processed and analyzed from the executable files established by the text computer variables that are encrypted in program files. All the sequences of n bytes that are being modified and derived from an executable file are named in general as byte sequence.

Their performance results were improved by Kolter & Maloof, 2006 in [15], using data mining methodologies and n-gram as a feature to detect malware. The algorithms chosen were Naïve Bayes, Support Vector Machines, Decision Trees with the last giving the best classification results.

Need for automation on malware classification was stressed by Kong & Yan, 2013 in [16]. Authors proposed a framework that depends on function call graph of malware. After finding a good way to extract the features based on function call graph for each malware sample, they used distance metrics to find the similarities between two malware programs. These metrics clustered and categorized the malware samples to same malware and class family while using a limited range kept the different groups separated. Having tested that approach an aggregation of classifying algorithms was utilized and suggested

that learns from pairwise malware distances to categorize malware into certain malware families.

Tian, Islam, Batten, & Versteeg, 2010 in [17] focused on classifying Trojans that use function length frequency. The amount of bytes that determines the function length is in the cipher. The performance of the algorithms shows that the function range along with its frequency are meaningful in the field of identification of malware families and is associated with other characteristics for malware classification regarding performance. WEKA library provides such algorithm for categorizing malware.

A different approach that was suggested from Santos in [18] mentions that a reasonable number of supervised executable files for malicious and benign samples were used in a semi-supervised methodology for identification and discovery of zero day exploits. This methodology utilizes and tries to perform machine learning using a lot of supervised and unsupervised cases and experiments. Learning with Local and Global Consistency (LLGC) which is a semi-supervised model, is employed, which can be trained from supervised and unsupervised data and provides an answer regarding the basic architecture presented by both supervised and unsupervised situations. A n-gram method characterizes and defines executables. Moreover, researchers conduct and assess the ideal amount of supervised situations and the effect of these situations regarding accuracy. Goal achieved of this inquiry and investigation is to decrease the amount of necessary supervised cases while achieving significant accuracy. The only downgrade is that supervised training methodologies were shown and presented better performance above or near 90%.

Another interesting research was done by Siddiqui, Wang, & Lee, 2009 in [19]. Their intention was to detect worms while examining the packets from traffic and find samples that are not yet analyzed and submitted to vendors, (also known as a term in the wild, another exciting field of detection). Before reverse engineering the samples, compilers and packers are identified and discovered. Decision Tree and Random Forest are utilized for classification and to make sequence reduction.

Zolkipli & Jantan, 2011 in [20] wanted to see malware classification from the point of view of malware behavior analysis as they believed that dynamic analysis could improve accuracy and performance. Every fragment is executed on Anubis and CWSandbox where actions of malware are identified. Analyzers generate results that use artificial intelligent and neural network depended on dynamic analysis. The malware are then

grouped into malware families. Main disadvantage of the research is today's internet traffic makes it impossible to use social analysis to achieve the desired results.

Another automatic behavior-based malware analysis framework was announced by Rieck et al., 2011 in [21] using machine learning. As most of the frameworks do, it collects many malicious samples and monitors their behavior using a sandbox virtual environment. After that conclusion and consideration, they inserted the results in a vector to implement and develop the algorithms. So, clustering was utilized to identify the families and clusters of malware with similar behavior.

The classification was focused on attaching and connecting zero-day vulnerabilities to identified clusters. This, was implemented to show and present that clustering and classification, focused and based on behavior-based analysis can process the activities of malware executables every day.

Anderson et al., 2011 in [22] presented a malware detection algorithm based on the analysis of graphs constructed after the dynamic collection of instruction traces. Modification of malware analysis framework based on Ether was used to gather samples. Methodology suggests the control of 2-grams to state the likelihood of a Markov chain graph. System of graph kernels is implemented to compute and calculate similarity vector between instances in the learning phase. Two metrics that finds and searches similarities, a Gaussian kernel, which computes and calculates the local similarity between graph edges and a spectral kernel which computes the global similarity between charts, calculates a kernel vector. Critical dissimilar behaviors of malware determine the performance of various kernel learning procedures. A disadvantage, is the high computational complexity, so the usage in actual situations and real environments is limited.

Bayer et al. in [23] suggested a method that puts effectively and automatically into classes malicious datasets. In order to apply more information sources, an extension for Anubis was implemented with taint-propagation efficiencies. An abstraction of evidences was created in addition with an observable outline for every trail, which aids as input to the Locality Sensitive Hashing (LSH) algorithm. Researchers show the scalability of their method by classifying a large dataset of malware data in a few hours.

Tian et al., 2010 in [24] utilized an automated tool for extracting API call sequences from binaries while these are running in a virtual environment. Tian utilized and applied classification methods from WEKA software to separate malicious data from good data but also, for classifying malware into their families.

Biley, in [25] constructed a classifier that explains malware's activities regarding system changes. A firewall is used to restrict and protect from the impact of any sudden and unnecessary activity during examination. A behavioral identity of malicious behavior was developed which includes network connection and processes created. To perform connection of the malware samples, a distance metric known as normalized compression distance (NCD) was applied and tested. The method performed an automated categorization of a specific set of malware samples. Biley, also measured and compared the fullness, integrity and condensation of the clusters and compared them with the clusters of AV vendors. As a disadvantage, can be considered that analysts encountered problems with consistency as the efficiency and the status are static.

A malware classification method was suggested by, Park et al., 2010 in [26] which depends on maximal component subgraph detection. First, a sandbox environment is used to execute and analyze malware samples, while system calls are taken, and a directed chart is created from these system call trays. For the comparison of the two programs the maximal common subgraph is calculated and estimated. However, there are some already known malware whose their primary ability is to gain root authorization bypassing the analysis procedure.

Another procedure for malware detection proposed by Firdausi et al., 2010 in [27] analyzes malware samples using Anubis. Machine learning is used for processing information and records into sparse vector models for classification.

Nari & Ghorbani, 2013 [28] developed a model for automatic malware classification into their specific and particular clusters depended on network performance. Their method depends on network traces applied as pcap input files to the model, that have been analyzed, processed and extracted. Afterward, a graph and plot of the network activities of malware was presented. Some features of these figures are adopted and used to classify malware using classification algorithms.

Lee et al., 2007 has developed another machine learning method in [29] regarding clustering malicious software. After the dataset is performed in a virtual environment where reports are exported, a behavioral profile is produced describes the sample's interaction with system resources. After the similarity between two profiles is computed, clustering algorithms are being applied like k-means and nearest neighbor to cluster them appropriately. In this method, the obstacle of obfuscation and execution-stalling techniques has made necessary the research of hybrid methods for better results.

Santos, Devesa, Brezo, Nieves, & Bringas, 2013 in [30] again developed a hybrid zero-day detector called OPEM, which uses and exploits characteristics gathered and collected from the analysis of malevolent code. Signature-based malware analysis obtains the static characteristics, and dynamic malware analysis captures dynamic features. Two disparate datasets are compared through different classification algorithms. This method improves the accuracy and speed of both methods when running individually. Islam et al., 2013 does something similar in [31].

Anderson et al., 2011 in [32] suggested a method, in which various information and features are utilized. Kernels based on Markov Chain graphs are being proposed for the binary file, disassembled file, and two dynamic traces. A graph-let kernel is applied and implemented for the control flow graph. A Gaussian kernel is executed for the file information data matrix. In order to find weighted connections between the data multiple kernel learning is employed. Moreover, to categorize and separate the dataset into malicious and benign files, support vector machine classifier is applied. The results have shown great performance.

As literature shows data science proposes several solutions for categorizing malware. Machine Learning is increasingly being applied in a variety of industries. No doubt that Information Security should be one of those, as the extent and complexity of networks is ever increasing. Internet and "cloud" applications generate vast data sets from performance monitoring and event logs which require scalable and flexible techniques to distil useful and actionable information.

3.2 Feature Engineering explanation

Due to the existence of large-scale malware search and retrieval and large datasets, there is a need for Cross-Validation. So, it is necessary to have a feature (data) matrix X and a label vector y. Once these two are calculated and computed by the algorithm, it is easy to split the data into training/testing and then pass them to a classification algorithm.

For data matrix x, GIST was used to calculate, measure and compute texture features, which uses a wavelet decomposition of an image taken from Nataraj et al., 2011 [7] and [33]. This feature has been proved favorable, performed and used in scene classification and object classification. In this problem, instead of scenes, there are malware samples converted to images.

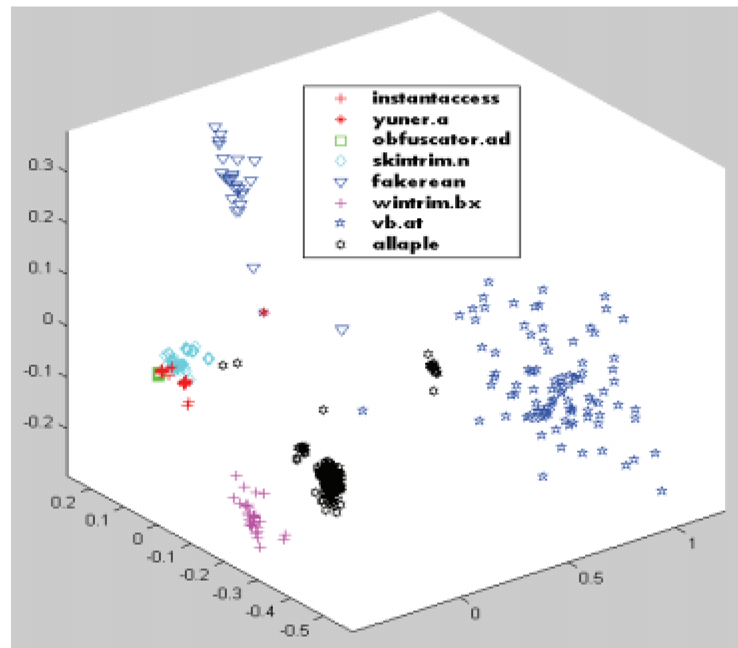


Figure 4: GIST Features projected in lower dimensions using multidimensional scaling [7]

As sarvam team on their blog [34] explains for the Malimg Dataset, the length of y is equal to the total amount of data samples meaning that in our experiment environment there are more than 9.000. The values of y depend on the number of families (classes), so in this situation, the numerical values to the different categories from 0 to 24.

A short explanation of cross-validation is necessary for understanding the methodology. Below is the actions and steps taken for every k-fold in the cross-validation technique:

- K-1 of the folds are used as training data to perform the training phase.
- Outputs are then approved and accepted as a test set in order to check and calculate accuracy for the rest of the dataset.

In general, these procedures are in a loop and the computed results after k-fold cross-validation are the average of the values calculated. The method is considered to be computationally demanding, but researchers use it for the advantage of keeping much of data the same. [36]

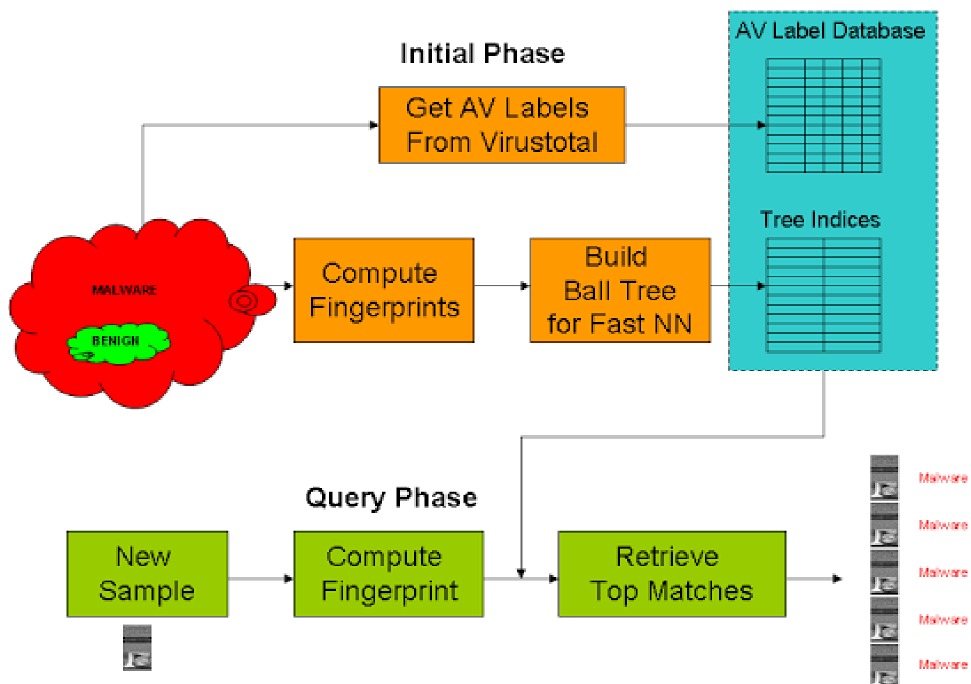


Figure 5: Diagram showing the process of identification of malware samples [34]

3.3 How to convert Malware Samples to Digital Images

The first step is to turn all the samples to digital images. The image similarity fingerprints (feature vectors) will be computed on these images. One more option could be the computation of the fingerprints in memory without saving the image on disk saving space from the hard drive.

```
import numpy, scipy, os, array
filename = 'sample';
f = open(filename,'rb');
ln = os. path.getsize(filename); # length of file in bytes
width = 256;
rem = ln%width;

a = array.array("B"); # uint8 array
a.fromfile(f,ln-rem);
f.close();

g = numpy.reshape(a,(len(a)/width,width));
g = numpy.uint8(g);
scipy.misc.imsave('sample.png',g); # save the image
```

A sample is defined as the name of the malware file that needs to be converted into digital images.

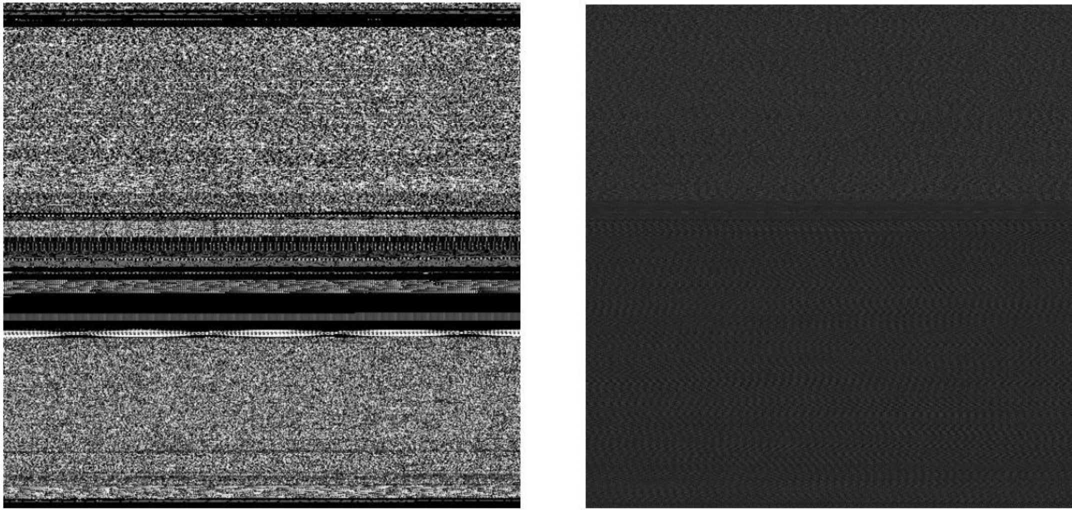
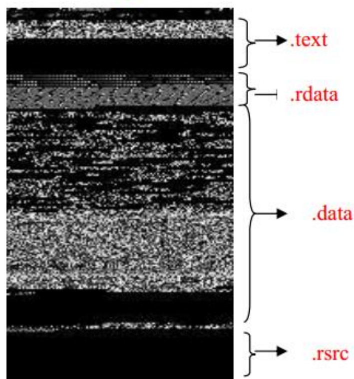


Figure 61 : Malware converted to image first picture is a byte file and the second picture is an asm file [7]



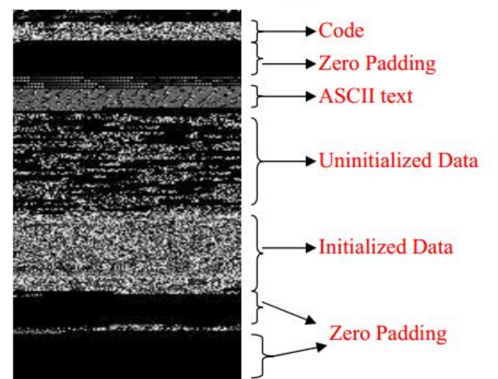


Figure 7: General structure of the information gained by converting malware to image. [7]

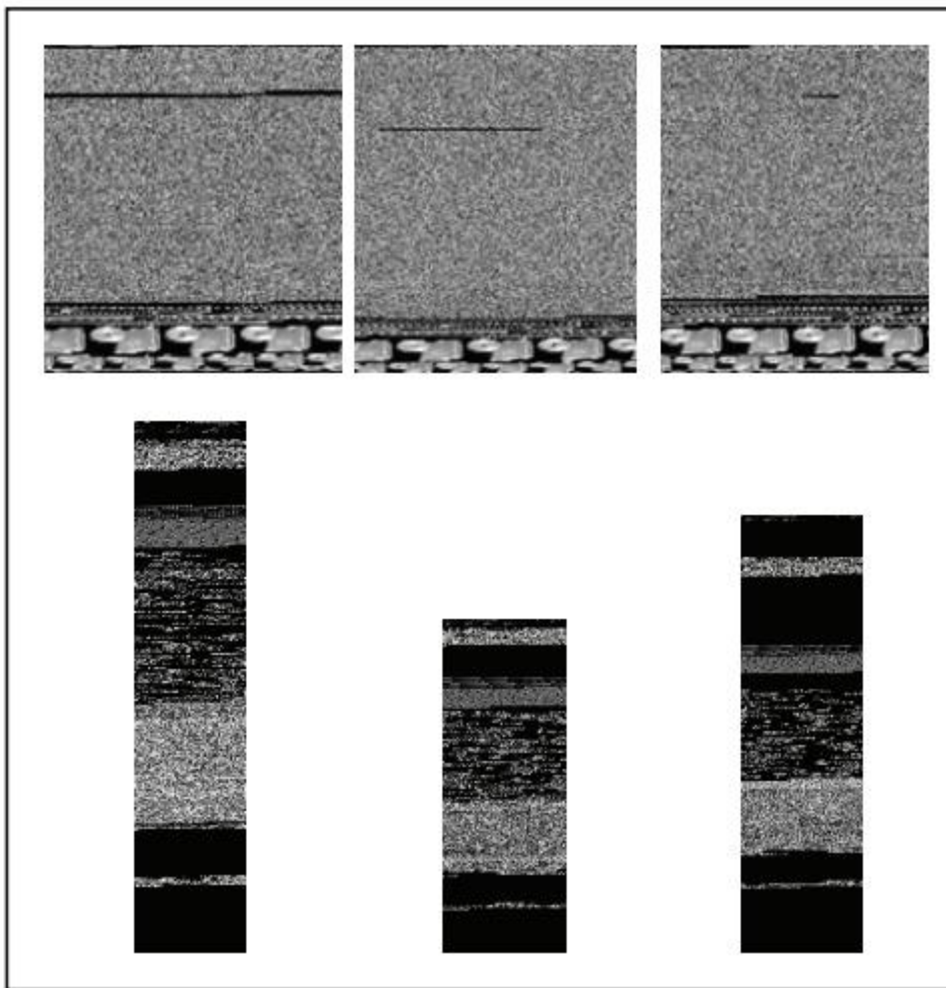


Figure 8: How malware samples are presented as images and how they appear differently even in their dimension length and width. [7]

When the conversion from the sample to image is finished, the next step is to compute a compact fingerprint for every binary. This fingerprint captures the structural/visual similarity between malware variants and is comparable with dynamic analysis.

```
import Image,leargist
im = Image.open('sample.png');
im1 = im.resize((64,64)); # for faster computation
des = leargist.color_gist(im1); # 960 values
feature = des [0:320]; # since the image is grayscale, need only first 320 values
```

In addition, packing transforms an executable, binary or asm file to a completely different form. As a result, the image extracted after packing, also appears completely different. Furthermore, there is a belief and misunderstanding that if two malware executables belonging to different malware families or classes are packed and encrypted using the

same packer, they will appear the same. *Figure 9* explains that this assumption is wrong and the Art of Unpacking in Black Hat, [9] presents more details about packing techniques.

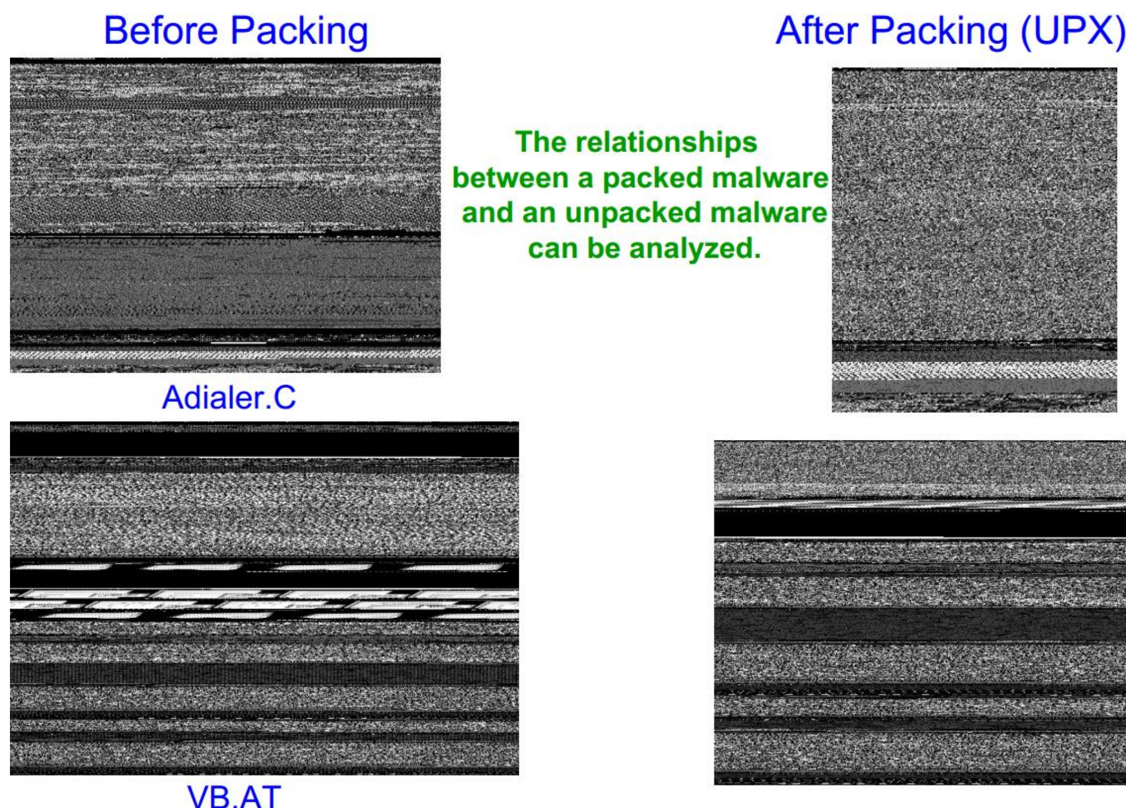


Figure 9: Shows malware images after packing and the differences, based on Nataraj's research on [7],

4 Experiment Setup

4.1 Software and Hardware Specifications

The environment used to perform tests is an Ubuntu 16.04 LTS (Xenial Xerus) system 64bit (AMD 64 bit) with 16 GB RAM and 1 TB Hard drive. To perform the experiments, Python programming language is used with some packages, libraries, and modules that help us to carry out the data analysis. The *pyleargist* package is imported to compute. For Neural Networks and the implementation of the Multilayer Perceptron experiments have

been conducted with the usage of *TensorFlow* and *Theano* libraries, information about the documentation and the functions are written in [35].

The dataset used for demonstration is the **Malimg Dataset**, from the paper [7] Nataraj et al., 2011 Malware Images: Visualization and Automatic Classification. This dataset comprises 25 malware families with varying number of variants per family.

4.2 Explanation of the Dataset

The dataset could be considered one of the most crucial parts for supervised classification as it should be correctly arranged. That is the reason that all the experiments are conducted with the Malimg Dataset. The table below, shows how this dataset is distributed and delivered. There are 25 malware families (classes), and every family has a varying number of samples. Total malware samples are 9342.

Table 1: Detailed and Categorized content of Malimg Dataset

No	Family	Family Name	No. of Variants
1	Worm	Allaple.L	1591
2	Worm	Allaple.A	2949
3	Worm	Yuner.A	800
4	PWS	Lolyda.AA 1	213
5	PWS	Lolyda.AA 2	184
6	PWS	Lolyda.AA 3	123
7	Trojan	C2Lop.P	146
8	Trojan	C2Lop.gen!G	200
9	Dialer	Instantaccess	431
10	Trojan Downloader	Swizzor.gen!I	132
11	Trojan Downloader	Swizzor.gen!E	128
12	Worm	VB.AT	408
13	Rogue	Fakerean	381
14	Trojan	Alueron.gen!J	198

15	Trojan	Malex.gen!J	136
16	PWS	Lolyda.AT	159
17	Dialer	Adialer.C	125
18	Trojan Downloader	WinTrim.BX	97
19	Dialer	Dialplatform.B	177
20	Trojan Downloader	Dontovo.A	162
21	Trojan Downloader	Obfuscator.AD	142
22	Backdoor	Agent.FYI	116
23	Worm: AutoIT	Autorun.K	106
24	Backdoor	Rbot!gen	158
25	Trojan	Skintrim.N	80

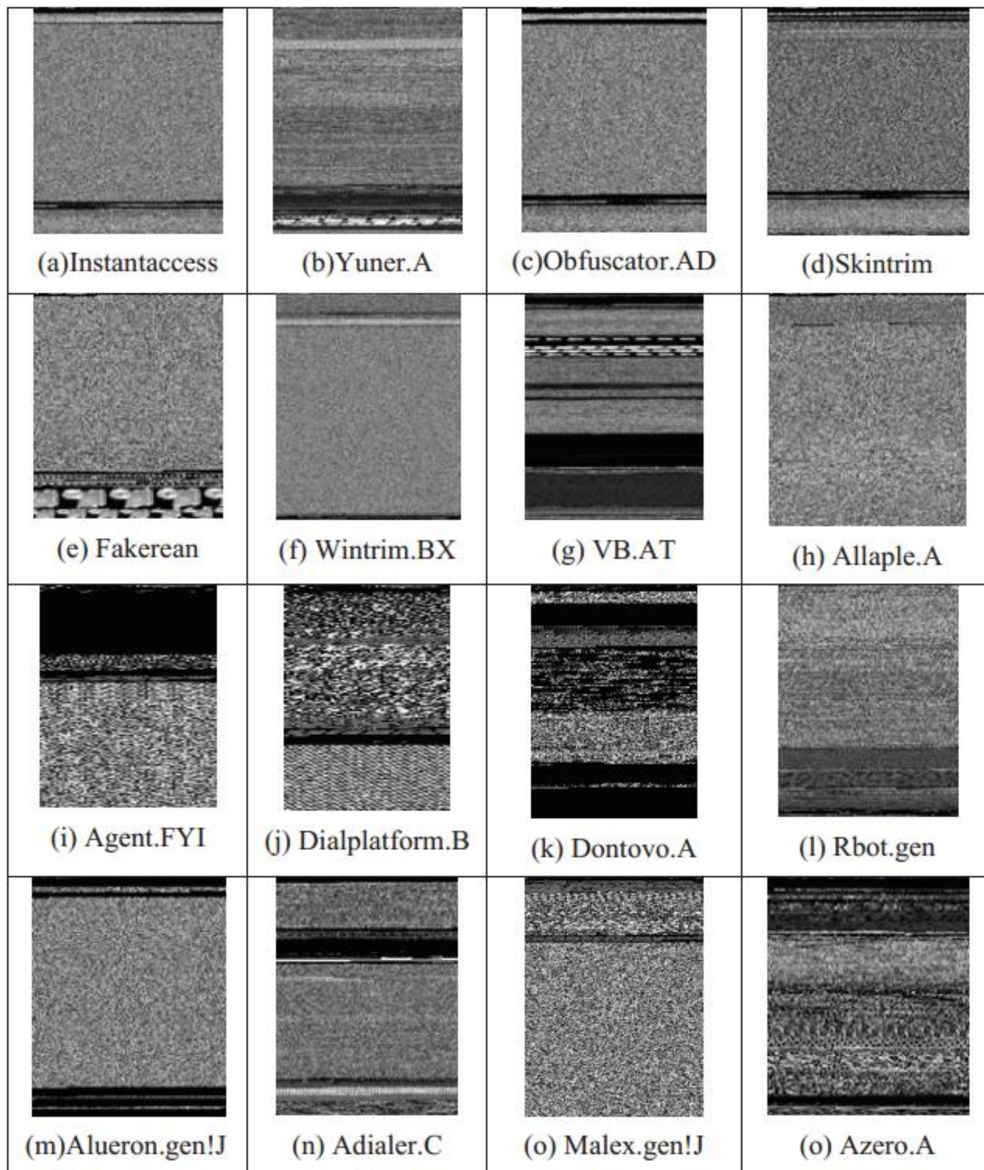


Figure 10: A representative snapshot of Malware families converted to Images [7].

4.3 Theory of Classification Algorithms implemented

According to scikit's library and OpenCV's documentation [36] an explanation of the algorithms used in this dissertation for experiment is provided below.

4.3.1 Support Vector Machines (SVMs)

Based on the bibliography and the theory explained in the documentation of scikit library and OpenCV's literature [36] below is some of the basics mathematics behind SVMs.

Support Vector Machine (SVM) applies and employs a set of supervised learning methodologies used for classification, regression, and outlier's detection. So, given a supervised learning dataset, the algorithm outputs an optimal hyperplane which classifies new examples.

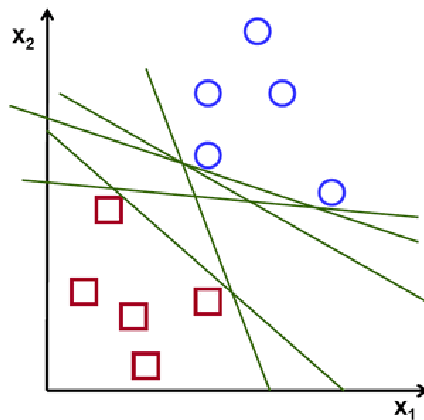


Figure 11: SVM Hyper lanes trying to find a possible solution to a problem [36]

Sometimes more than one line could be the solution to the problem, so there is a need to find a way to give a description patterned to choose which solution is better than the other. SVM algorithm's primary goal is to find the optimal separating hyperplane of the training data.

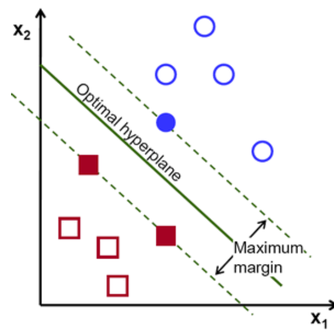


Figure 12: Optimal hyper-plane separation [36]

Support Vector Machines have their pros and cons.

The advantages of support vector machines are:

- They are productive and give satisfactory results when the number of samples is lower than the number of dimensions.
- They use support vectors, a part of the set of training points in the decision function, so it can easily be said that some kind of a memory can be achieved, making dissertation's goal more feasible.
- They are considered versatile due to different kernel functions that can be specified for the decision function. Specific kernels are provided, but it is also possible to define custom kernels.

The disadvantages of support vector machines include:

- It does not usually provide good results if the number of samples is much lower than the number of features.
- With SVMs, there are no probability estimations as these evaluations are computed using an expensive five-fold cross-validation.

4.3.2 Perceptron

Perceptron algorithm is used for supervised learning of functions that can decide, predict and calculate whether a vector of values and data that is used as an input belongs to one class or another. It is a classification algorithm that is considered as a linear classifier meaning that it tries to predict focusing on a linear predictor function combining a set of weights with the feature vector. As the data in training, the dataset is processed one at a time allowing researcher and analysts to use perceptron for online learning.

Figure 13 shows how a perceptron with a single layer is learning and categorizes the samples.

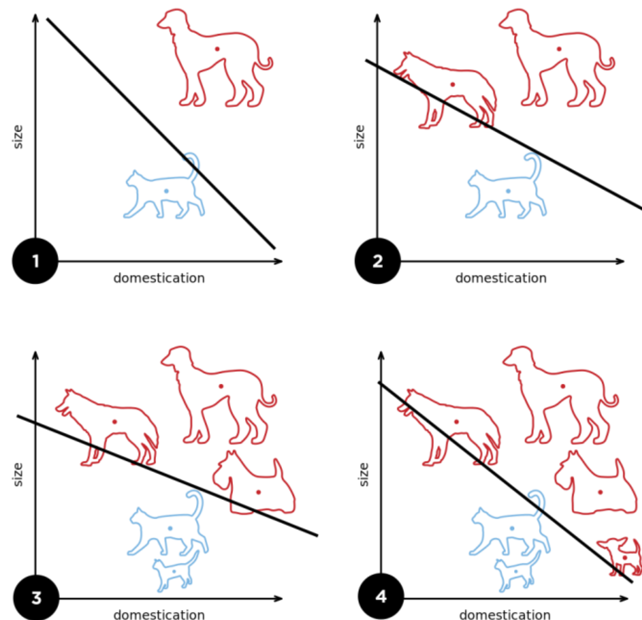


Figure 13: Steps of a perceptron finding an optimal solution. [36]

A perceptron is a linear classifier meaning that if there is a training set that is not linear it will never find an optimal solution to the problem where all the input matrices will be separated with a correct hyperlane. As a result, the training phase will stop computing and performing. So, there is a need to know the linear separability of the learning set.

However, if the learning set or the problem that is encountered is linearly separable, then the perceptron is assured and confident to find a solution, and there is a limit known as an upper threshold and there is a certain number of times that the algorithm will refine and modify its weights during the training.

As explained above a perceptron algorithm is possible and supposed to find an acceptable solution in the case of a linearly separable training and learning set, but it may still choose though any solution or the algorithm may realize many solutions of differing quality. To find a solution to this problem linear support vector machine was developed and applied.

4.3.3 Multilayer Perceptron

Multilayer Perceptron (MLP) has a structure of mainly three tiers, the input tier, the hidden tiers and the outcome tier. Every tier of MLP consists of nodes connected with the nodes from the prior and the later tier.

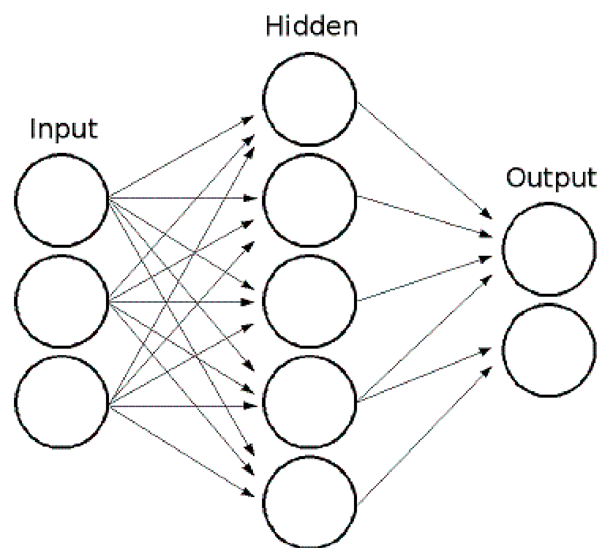


Figure 14: A model and structure of a multilayer perceptron, [36]

All layers of neurons in MLP have various and different input connections in the process of taking the results and output of data from the previous layer nodes, as a result, it produces several output links for the next layer of nodes. The numbers computed and calculated from the last segment or number of nodes are summarized with specific weights, individual for each node, after it considers the bias term to compute new results. The amount is computed with the help of the activation function that may also be dissimilar for dissimilar nodes.

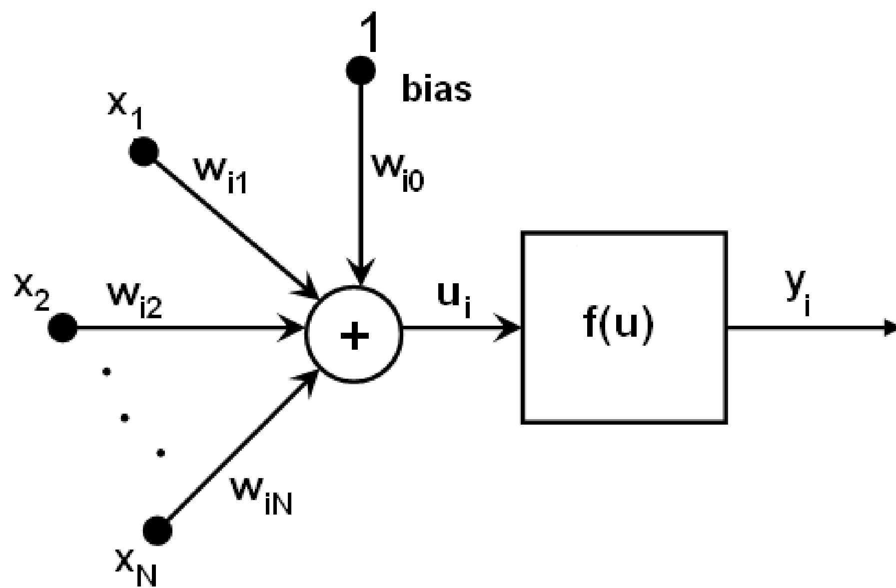


Figure 15: A more logical diagram of how multilayer perceptron works after it takes some data matrix x inputs. [36]

From the documentation of the OpenCV library below is an explanation of the MLP algorithm [36]

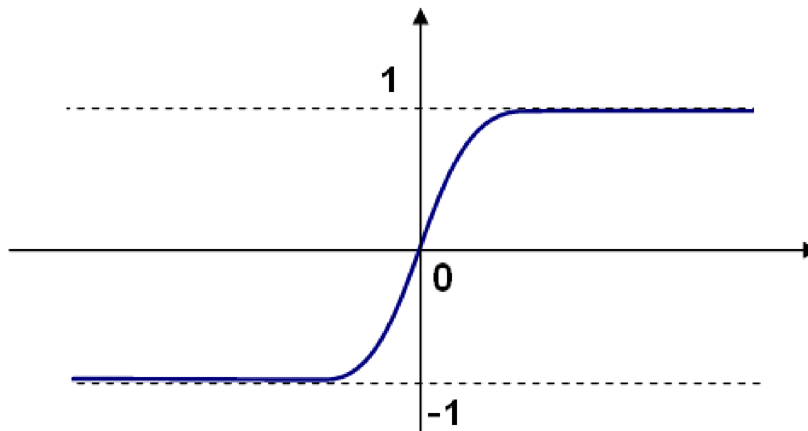


Figure 16: Bipolar sigmoid function [36]

All the neurons in Multilayer Perceptron have the same initiation functions, with the same variables that users define and the learning phase does not change them.

Procedures and steps taken for the training of the model is mentioned below:

- Set data matrix with features as initiator and activator.
- The scale of the entry tier is equal to the matrix size.
- Insert and Put initial values to the first hidden tier.
- Calculate outcomes of the hidden tier with the use of the weights and the initiation services.
- Allow and set additional results later until you compute the final tier. [36]

In order to compute the model and system, there is a need to have all the weights. The model takes an instruction set, several initial matrices with the appropriate matrices, and modifies the weights to allow and authorize the model to contribute to the inclined reaction to the provided information matrices. [36]

Having a broad network of hidden layers gives the ability for inherent system flexibility. The computation of the error on the training subset can become extremely low after summarization. However, the trained system gets the data and shows the noise in the training set. This has as a result, the percentage of the error on the test subset increases after the size of the system comes to its limit. Furthermore, MLP trains large systems, so it is tolerable to preliminary process of data, but it trains small systems for only essential features.

A disadvantage of MLP can be considered the inefficiency to handle unlimited data.

MultiLayer Perceptron performs and develops two training Multilayer models and frameworks, a random sequential back-propagation algorithm and a batch RPROP framework.

4.3.4 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is a linear classifier algorithm that it is considered efficient and simple for discriminative learning. Stochastic Gradient Descent is implemented for experiments because it performs large-scale classification with good results. Most of the times it is used for text classification and natural language processing. Gradient descent performs optimization focusing on neural networks. These algorithms, however, are often used as black-box optimizers, as there are some advantages and limitations that are hard to get unnoticed.

The advantages of Stochastic Gradient Descent are:

- Relevance and Productivity.
- It can be implemented and modified accordingly to each problem code wise.

The disadvantages of Stochastic Gradient Descent include:

- It has several hyperparameters such as the regularization parameters.
- It is impressionable to feature scaling selection.

4.3.5 Nearest Centroid

The Nearest Centroid is a classification model that shows every malware family of data by the centroid of its samples. It has many similarities with KMeans clustering algorithm. One is the feature of having no variables to choose. When families or classes have many differences, it experiences problems on non-convex classes. Nearest Centroid actions are related to the phase where labels are updated by Kmeans algorithm.

The algorithm behind Nearest neighbor can be explained below:

First of all, it is the training phase where given labeled training samples

$$\{(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)\}$$

with class labels y belongs to Y, compute the per-class centroids

$$\vec{\mu_l} = \frac{1}{|C_l|} \sum_{i \in C_l} \vec{x}_i$$

Finally, the prediction function that computes the class assigned to an observation x is

$$\hat{y} = \arg\min_{l \in \mathbf{Y}} \|\vec{\mu}_l - \vec{x}\|$$

4.3.6 Multinomial Bayes

The Naïve Bayes Classifier (NB) is a simple effective classification algorithm which has been employed and applied in the field of data analysis. The NB method depends on the Bayes's rule and is usually chosen and suggested when the dimensions of the data of the initial values is high. Naïve Bayes classifiers take for granted that the effect of a variable value on a given family or class is autonomous of the values of another variable. The Naive-Bayes activator measures and calculates dependent likelihoods of the malware families offered in the experiment and chooses the malware family with the highest value.

Relying on the accurate character of the likelihood algorithm, Naïve Bayes classifiers can be trained very efficiently in a supervised learning setting.

Moreover, Naive Bayes can be considered as a method for structuring classification models which attach class labels to case instances served as matrices of feature values, where the class labels are chosen from a supervised dataset. It cannot be considered as a single algorithm for training like classifiers, but a group of algorithms depending on a common assumption: all naive Bayes classifiers assume that the value of a particular characteristic is autonomous of the value of any other feature, given the class variable. There are some probability models where Naive Bayes classifiers can be trained very efficiently in a supervised learning experiment and environment. In many realistic experiments and operations, parameter prediction for naive Bayes models applies the method of maximum likelihood.

Even though their design can be considered naïve and messy with the conclusion of producing simple assumptions, Naive Bayes classifiers perform well in many complex situations and real life environment.

A strength of Naive Bayes is that it needs a small amount of training data to estimate and predict the parameters essential for classification. So, Naïve Bayes classifiers are incredibly fast compared to more sophisticated methods and approaches.

4.3.7 Decision Trees

A decision tree is a binary tree, and a non-parametric supervised learning method that can be utilized for classification where each child of this tree is considered as a class label but also many leaves may have the same label. Moreover, can be used for regression, meaning that a constant that is assigned for each tree leaf, to the approximation function being piecewise constant. So, decision's tree goal is to make a prediction model that calculates and computes the target variable from decision rules that come from the data features being extracted.

Some advantages of decision trees are:

- Easy to understand and to be defined.
- Trees can be visualized.
- Demands little time for the data to be arranged.

- There is no need for data normalization
- The complexity of using the data that they considered to be predicted is logarithmic in the number of data values utilized to train the tree.
- Ability to manage both numerical and unconditional data. Other techniques are usually based and focused in analyzing datasets that have only one type of variable.
- Ability to manage multi-variant problems.
- Uses a white box method. If a given situation is apparent in a method, the explanation for the condition is easily explained by Boolean logic.
- On the other hand, in a black box model such as artificial neural network, conclusions may be harder to defined as the possible method can be the validation of a model using statistical tests about reliability.

The disadvantages of decision trees consist of:

- Decision-tree learners that can produce and make over-complex trees that do not generalize the data well-meaning that often researchers encounter overfitting.
- Decision trees can be insecure, risky and irrational because small changes or variations in the data might end up with an entirely different tree being produced and calculated. This problem can be counter measured by using decision trees within an ensemble.
- Training and learning an optimal decision tree is considered as an NP-complete problem under several visible features of optimality and even for simple concepts. Therefore, practical decision-tree learning algorithms are in a firm position on heuristic algorithms such as the greedy algorithm where locally optimal decisions are created at each tree-node. Such algorithms cannot assure or promise to produce the most acceptable optimal decision tree. This, again can be encountered and mitigated by learning multiple trees in an altogether trainer, where the features and samples are at random sampled with replacement.
- Some problems are not easy to find a solution or to train and learn because decision trees do not categorize them quickly, such as XOR, parity or multiplexer experiments.

- Decision tree trainers generate biased trees if some classes overshadow and rule. So, a suggestion is to define and refine the dataset before so as to fit with the decision tree.

4.3.8 Bernoulli Restricted Boltzmann Machine (RBM)

A restricted Boltzmann machine (RBM) is a generative stochastic artificial neural network that can learn a probability distribution over its set of inputs.

Researchers and scientists apply Restricted Boltzmann machines in deep learning networks.

Original RBM has binary-valued hidden and visible entities and involves a matrix of weights related to the relation between visible group and hidden group, but also bias weights for the hidden groups and the visible groups. Having these statements, the source of a configuration is defined as a combination of Boolean vectors.

To conclude, by definition, Boltzmann machines, probability distributions over hidden and visible models are determined regarding the energy function.

Restricted Boltzmann machines aim to finding the maximum of the results of likelihoods attached and authorized to a training subset.

Some procedures for a single data sample can be mentioned and seen below:

- Take a learning model, calculate the likelihood of the hidden groups and try an unidentified activation matrix from this likelihood classification.

- Calculate the external output and outcome of the two vectors and name it as the positive gradient.

- Gibbs sampling procedure-From first vector, try a reformation a new vector of the non-hidden groups, then retry the hidden activations from this.

- Calculate the external output and outcome of the new vectors and name it as negative gradient.

- Allow the new weight matrix be the positive gradient minus the negative gradient, calculate the time of learning phase. Renew likewise the biases [36].

4.3.9 Random Trees-Forest

Random Forest took its name from the collection of the tree predictors that are designed to handle both classification and regression problems.

The Algorithm of Random Forest has the following steps:

1st Step: Feature-data vector is integrated into the random forest classifier the random trees classifier.

2nd Step: It is classified with every tree in the forest.

3rd Step: The label of the family or class that has the most similarities commonly referred as votes, answers or replies, is outputted thus to the problem.

When a regression problem exists, the algorithm produces the average of the replies-votes for all the trees in the forest. The trees are trained with similar principles but on dissimilar training datasets.

4.4 Theory of Clustering Algorithms implemented

4.4.1 MeanShift Clustering

MeanShift clustering is also a centroid algorithm, which utilizes and functions by bringing up to date prospects by computing the mean of values for centroids within a supplied area. Proposals are refined and pre-processed in a phase to erase and get rid of almost duplicates of data to form the final set of centroids. Moreover, there is an automatic way that identifies clusters, ignoring parameter bandwidth, which enforces the area to be searched for. High scalability is not the algorithm's strong feature, as various nearest neighbor examinations are needed. MeanShift is certain to find a solution or to conclude in one. On the other hand, MeanShift will stop calculations when there is small alteration in centroids [36]

MeanShift can be specified as

1. Define a window around each data point.
2. Measure and calculate the mean of data within the window.
3. Move the center of window to the mean and repeat till convergence.

4. After every repetition the windows alter to a compressed region of the set of data

The fact that MeanShift does not make assumptions about the number of clusters or the shape of the group makes it ideal for performing dissertation's hypothesis of the recreation of the clusters of Malimg dataset but also it handles groups of non-fixed shape and number. [36]

4.4.2 DBSCAN Algorithm

The DBSCAN algorithm treats clusters as areas of high density divided by areas of low density. Thus, groups produced by DBSCAN can be any shape, and not like KMeans which takes for granted that clusters are convex shaped. When high density is mentioned, researchers talks about the core samples that are also considered as features and characteristics of the algorithm. So, a group of data is a set of gist specimens, close to each other after being calculated by a distance metric and a set of non-gist specimens that are close to a core sample. There are two parameters and characteristics on how this algorithm works, called minsamples and eps, which describes officially what densely is. Higher minsamples or lower eps indicate higher density necessary to form a cluster.

A core sample or data is in an area of the vector space when in a sample of the dataset such that there exist minsamples within neighbors of the core sample. A group is a fixed area of core samples, that can be created by picking a core fragment and find all its neighbors that again are core specimens. Such a group has a set of non-core data as well, which are data that are neighbors of a core fragment in the group but are considered as outliers.

Any gist feature is part of a group. Further, any cluster has at least minsamples points in it, following the definition of a gist fragment. DBSCAN treats as an anomaly any fragment that is not a gist feature, and does have a range and radius more than eps to any gist feature.

The first input provided to the DBSCAN algorithm is the values of data points that needs to be clustered. The second input is the definition of a distance function between the points. It worths to mention that, intensity distance function is used if pixels of the same intensity need to be clustered. The third input sets up the sensitivity of the DBSCAN,

which takes the decision if two data points are similar or different. Finally, another sensitivity factor could depend on the density of nearby data points and then decide whether a new group should start at a given data point.

The DBSCAN algorithm can recognize regions and groups to a great spatial data sets by examining the local density of database features, using only one input parameter. The DBSCAN decides what information should be categorized as anomaly. Perfomance of the algorithm is fast and scales very well with the size of a linear collection of data. The algorithm can classify nodes into independent groups that characterize the dissimilar malware families by utilizing the density distribution of nodes. *figure 18* shows that DBSCAN can discover groups of non-fixed shape. Nevertheless, groups that are near each other usually belong to the same malware family or they can be considered as variants.

Figure 18: Arbitrary shape groups after DBScan algorithm is performed. [38]

4.4.3 KMeans

K-Means clustering is an algorithm that utilizes the method of vector quantization taken from the field of signal processing and most of the time examiners use it for solving clustering problems in data science. K-Means clustering tries to transform the whole dataset to Voronoi cells by having observations and make k groups in which every observation is a part of a computed nearest mean cluster.

The problem in computer science is considered NP-hard meaning that is computationally difficult to solve. Nevertheless, k-means clustering attends to identify categories of families and classes of comparable spatial extent.

K-means is different from k-nearest neighbor classifier, as they often confused each other due to the k in the name. On the other hand, both these algorithms can be combined to find and produce better results. A serious problem is that k-means is not very extensible, and it is used for vector quantization. The parameter k is admittedly difficult to choose when not given by external constraints. Another disadvantage of the algorithm is that it cannot be used with arbitrary distance functions or on non-numerical data.

4.4.4 Minibatch KMeans

The MiniBatch KMeans is a different approach which utilizes mini-batches to decrease the calculation time, while attempting to create more excellent the impartial activity. Mini-batches are subgroups of the initial information, randomly modelled in each learning calculation. These mini-batches exceptionally decrease the number of calculation necessary to gather to a local result and explanation. Compared with other methods that decrease the union time of k-means, mini-batch k-means generates solutions that little better than the regular and typical method.

The mini-batch method performs between two major actions, similar to original k-means.

At first, b fragments are chosen at random from the dataset, to model and pattern a mini-batch. Results are later attached to the nearest centroid. Secondly, the centroids are renewed. For every fragment in the mini-batch, the attached centroid is maintained by communicating the average of the fragment and all previous fragments attached to that centroid. As a consequence, the rate of alteration for a centroid over time is reduced. These steps are performed in conjunction or until a predefined number of repetitions is achieved.

4.5 Goals of the experiment and comparison criteria

As a future work, the aim is to build a classifier tool that can classify malware samples automatically and has something like a memory and can organize labels-classes that the classifier has not yet processed or learned.

First, it is necessary to know the right classes (called labels) in the training set. There is a need for algorithms that can learn and remember previous testing and experiments. Testing and comparing algorithms are done using a test set, for which the labels are known. Many algorithms also use a validation set (mainly part of the labeled training set) to manage its learning process.

In the results, there are some miss-classifications and low prediction rate. So, a goal is to improve these algorithms to find better solutions.

Another approach that could be considered as the problem was that the classification procedure takes for granted that all fragments in the dataset are malevolent. This, means that if unknown software were injected, it would be classified as malware no matter what. Furthermore, another aim is to build models focused on larger datasets, and these models will ensure a more robust classification.

Additionally, the data should be as uniformly distributed as possible, leading to a fairer classification.

Finally, this dissertation will emphasize the necessity to improve the classification rate by choosing and take advantage of other features and using a feature selection algorithm.

To summarize the main differences, the following list has been made. The goal in general is to extend and improve the system by:

1. *Performing malware detection.*

2. *Performing classification of malware families.*

3. *Finding new and improving old features.*

4. *Applying a feature selection algorithm, that will select the most discriminative features.*

5. *Building an extensive database of malware by collecting more samples.*

6. *Retrieving a uniform sample set among the malware classes.*

Finally, comparison criteria to our dissertation except for the part of comparing our experiments and algorithms to each other is the nearest neighbor results as currently is presented on Sarvam blog. Moreover, the primary hypothesis, is if a recreation of the 25 malware families and clusters after shuffle them using the clustering algorithms presented, is possible. Finally, a comparison between Kmeans and MiniBatch Kmeans is performed and shown.

4.6 Expected Outcomes

Expected outcomes are the malware images being classified as the first dataset and arranged within the same subfolders with the same labels or at least be as near as possible to the first that is the main reason this dataset were being chosen in this dissertation to know the best-classified outcome. *Figure 23* shows the expected outcome.

The goal of clustering algorithms is to test and examine if the same 25 clusters as the original dataset can be recreated and achieved. So, the experiments try to reform the same groups and cluster from malware samples after the data is shuffled by the algorithm implemented.

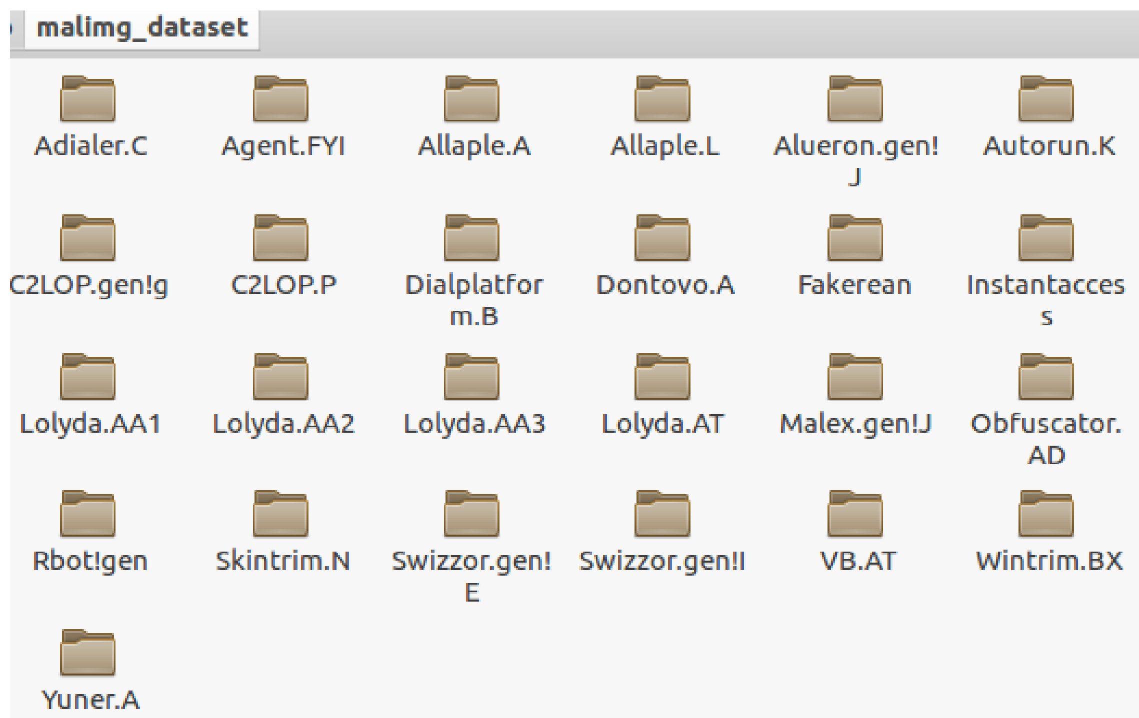


Figure 23: Folders with the names of the malware families of Malimg dataset [34]

Also, classification-wise, experiments' results tested and compared with Nataraj's results on Nearest Neighbor. So, an expected outcome could be a better accuracy machine learning algorithm. The accuracy of the Nearest Neighbor is 93.36% and from the experiments and results does not seem that an algorithm finds better results on accuracy than nearest neighbor. Performance is not something that the dissertation aims even if experiments need to run as fast as possible and be optimized as much as possible by developers. The

detection rate of accuracy and reducing misclassification and detection errors are more important. Performance wise the average extraction time is 5s and the time taken by the algorithm to classify a sample is 56s. On the other hand, the time that it takes to calculate GIST feature is 54ms and the overall classification time was 1.4s, but these numbers depend on Nataraj's proposal, and they are slightly better and revamped from the code that exported the below diagram. So, an expected outcome would be to find better and faster algorithms from this proposal.

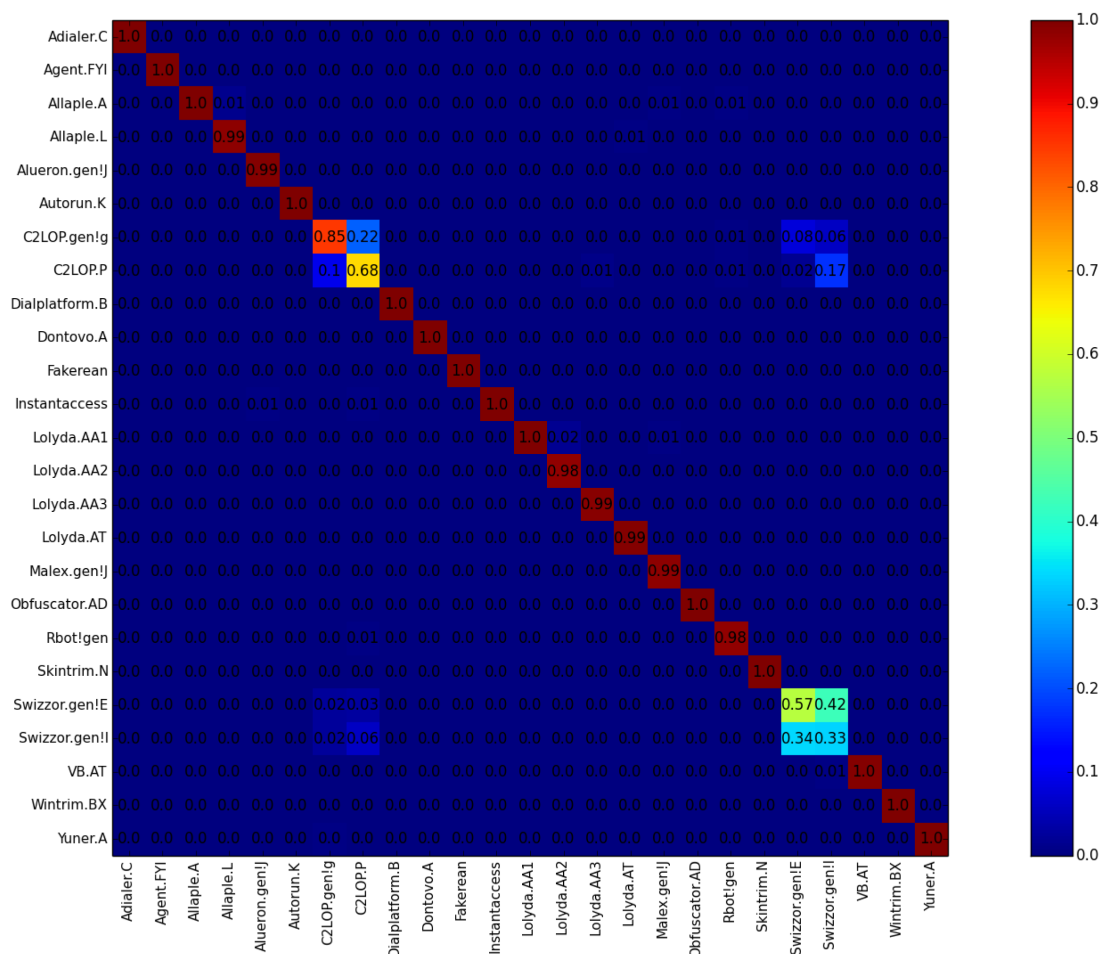


Figure 24: Diagram of the results of nearest neighbor presented in SARVAM's blog. [34]

5 Results of Algorithms

5.1 Classification Algorithms Results

The results are presented in a range from 0.00 to 0.1 meaning that having a 0.1 and in the correct label there is a 100% correct classification. On the other hand, the algorithms find similarities to other malware families even if a detection number is not classified correctly by the experiments, meaning that again is at least detected. Furthermore, on some diagrams, if it shows results that are not near the line of the confusion matrix, they are considered as errors, but if they are near the line of the confusion matrix, it may be regarded as variants even if they are classified to a different malware family. *Figure 25* shows a better view of the text below:

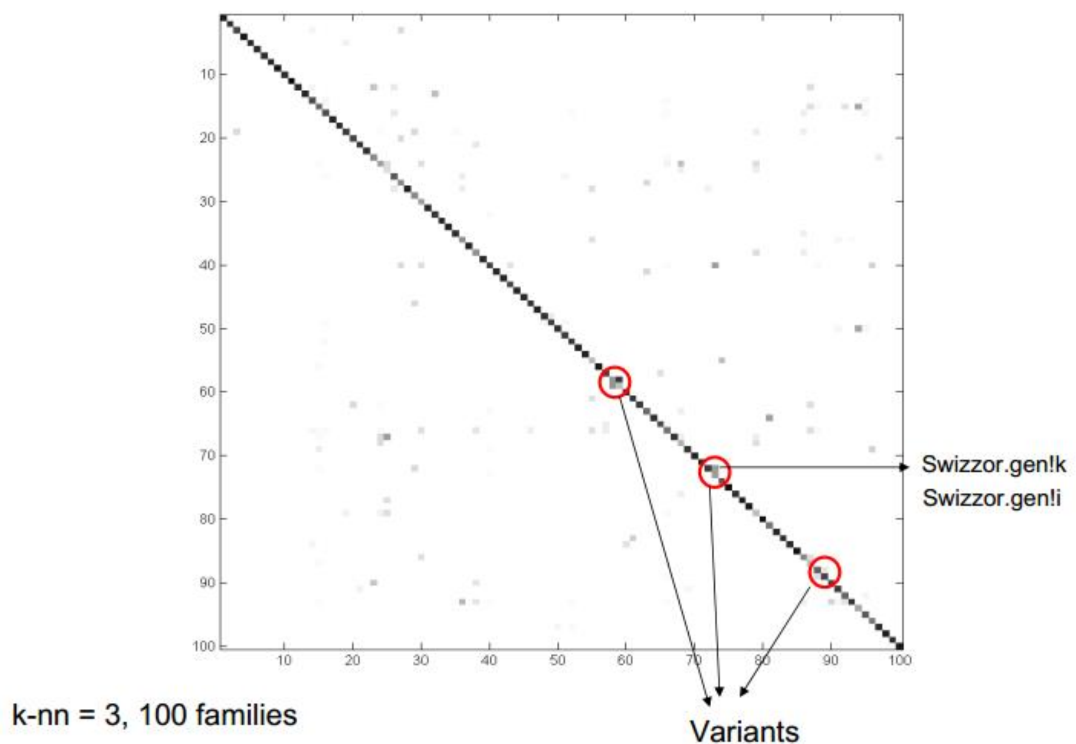


Figure 25: Nearest neighbor figure from Sarvam experiments based on Nataraj's [7] indicating the variant interpretation.

In implemented algorithms, most of the samples have been classified correctly but other samples did not. For that reason, the algorithm might have found similarities in the fingerprint feature on other malware families. Support Vector Machines (SVM) seems to have been over trained as the results were not the expected so potential changes on the initiated values should be considered. Also, how it can perform better to a different dataset, and there is a need to test more on these algorithms is the memory that they provide, and it is essential for the general classifier that has to be achieved. Same goes for BernoulliRBM and Multinomial Naive Bayes, on BernoulliRBM algorithm though there was a hypothesis to see how an unsupervised algorithm performs to a supervised and classified dataset.

For clustering algorithms, a hypothesis was made while researching if the original dataset can be achieved and recreated, meaning that the perfect result would be 25 clusters. Best estimation of clusters was from DBScan Algorithm, and then MeanShift did an evaluation of 15 groups. Then, again a hypothesis if a particular initiation is given by a researcher and a certain number of groups how the results will be using Kmeans and MiniBatchKMeans. A comparison was made of the two to identify the differences. The differences discovered can be interpreted as non-crucial having the conclusion that they perform the same way in the current dataset. These algorithms should be carried out on other datasets as well to see if they perform the same way. Also, that is the reason why malware classification is an ongoing examination and research as there is no explicitly an opinion that guaranteed that these algorithms tested will perform the same to different malware families. Each dataset is a different problem.

In information security, advanced persistent threats, malware and malicious are an ongoing battle so effort on that field and a conversation on how the community can improve algorithms should start. Moreover, according to a recent survey from Symantec advanced persistent threats and more precisely ransomware is one of the most severe and dangerous attacks on businesses and organizations in cyberspace and the loss of money in bitcoins being million dollars. Only in 2015 the new malware families for ransomware that have been discovered are more than 100 plus the previous ones, and the average ransom that they ask is at 679 dollars in bitcoins plus decryptors for Green Petya and Cryptowall are still unknown. [2] Even if better results can be seen to one algorithm than another, no one

cannot easily say that this algorithm should be used for every classification, every algorithm has each pros and cons on time and accuracy.

Below is a table that summarizes the results of all the algorithms.

Table 1: Summarized Results of the experiments of the classification algorithms

Classification Algorithms	Average Training Time (secs)	Testing Time (secs)	Average Accuracy
Decision Tree	6.53	0.00096	0.088-> 88%
Support Vector Machines	45.93	3.78	Over Trained, Re-evaluation or Results that has no meaning due to misclassification
Nearest Centroid	0.218	0.0211	0.0856-> 85.6%
Stochastic Gradient	1.291	0.0585	0.087->87% plus 2 missclassfications
Perceptron	0.817	0.0148	0.0905-> 90.5% plus 4 missclassifications
Multilayer Perceptron	16.05	0.091	0.878-> 87.8% plus one missclassification
Random Forest	1.72	0.0063	0.0916->91.6%
Multinomial Naive Bayes	0.0197	0.001445	Over Trained, Re-evaluation or Results that has no meaning due to misclassification
BernoulliRBM	208.276	0.0206	Over Trained, Re-evaluation or Results that has no meaning due to misclassification

Table 2: Summarized Results of the experiments of the clustering algorithms

Clustering Algorithms	**Cluster Estimation**	**Predefined Clusters**	**Difference**
MeanShift	15	-	-
DBScan	20	-	-
Kmeans	-	2	-
MiniBatchKMeans	-	3	-
K-Means-MiniBatch Comparison	-		3

5.1.1 Classification Results for Decision Tree Algorithm

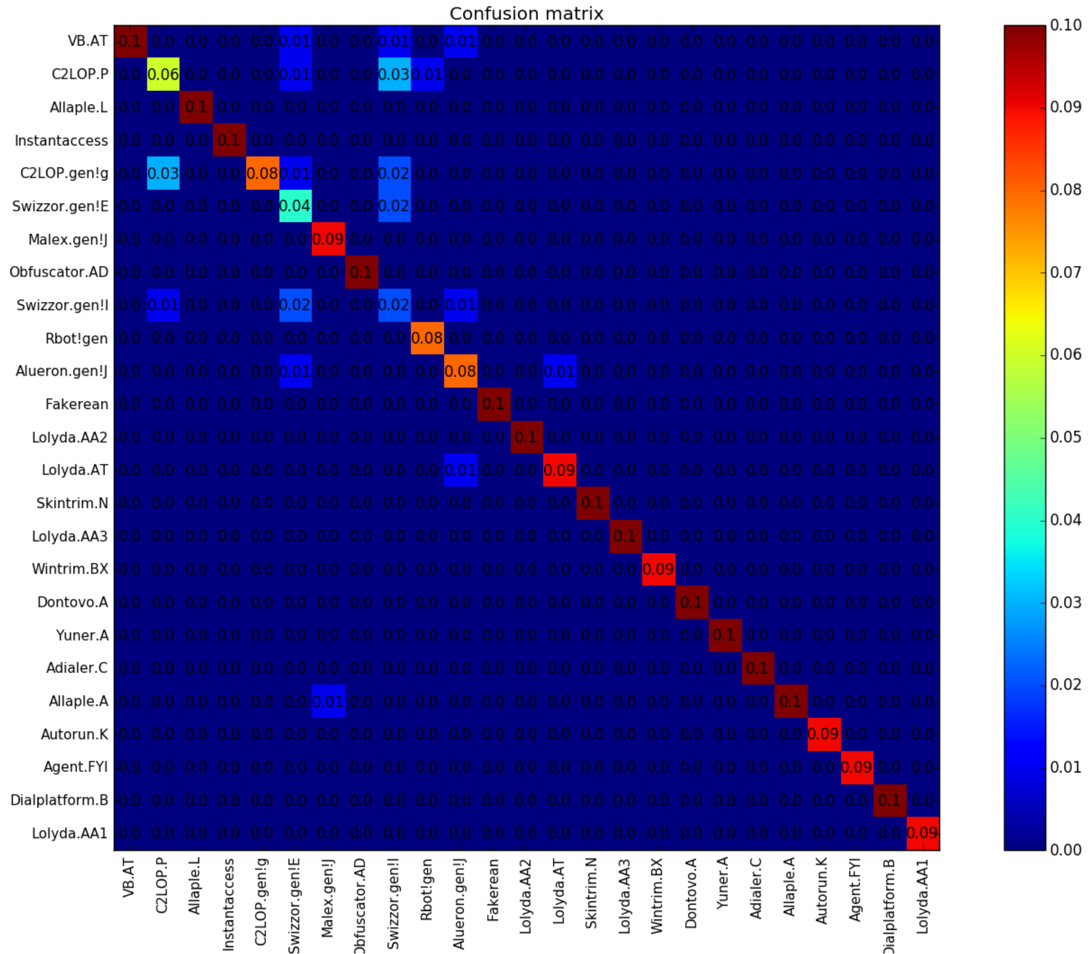


Figure 26: Decision Tree Algorithm Results

Decision tree results show that there is no misclassification but only minor accuracy errors (blue squares that are far from the confusion's matrix line) meaning that some of the samples for this algorithm have some similarities with other specimens. In malware analysis, the C2LOP.P malware family maybe has similarities with C2LOP.gen!g.According to the names, this assumption is right as they belong to same malware family and they are considered variants but some of their samples can be on the Swizzor.gen!l. Labeling wise those few samples are a detection error or more in-depth features need to be examined by research to identify if similarities exist. These goes for the other malware families that behave the same.

After ten folds, the times for the above results are:

Training Times:

1st: 5.62584996223, **2nd:** 6.11614894867,

3rd: 7.76607918739, **4th:** 6.38322091103,

5th: 6.60698008537, **6th :** 6.37441205978,

7th : 6.65543818474, **8th :** 6.86663007736,

9th: 6.49542498589, **10th :** 6.36758112907

Testing Time: 0.00096

5.1.2 Classification Results for Support Vector Machine (SVM) Algorithm

Figure 27: Support Vector Machine Algorithm Classification Results

The results of Support Vector Machine are scrambled after exporting, so there is no meaning and correct explanation on this experiment. They are too misclassified so it cannot locate any similarities but also the settings being put for initialization maybe are over trained the algorithm on the test phase. From malware detection and analysis perspective, the problem was not approached correctly; it may be considered as that this is not similarities but overfitting. Further, testing for potential errors and research for this algorithm is necessary.

After ten folds, the times for the above results are:

Training Times:

1st = 45.6123859882, **2nd** = 45.6906061172,

3rd = 44.0576298237, **4th** = 44.8201007843,

5th = 43.707859993, **6th** = 45.5356550217,

7th = 47.0243530273, **8th** = 51.7937111855,

9th = 45.6938140392, **10th** = 45.3624200821

Testing Time: 3.78129291534

5.1.3 Classification Results for Nearest Centroid Algorithm

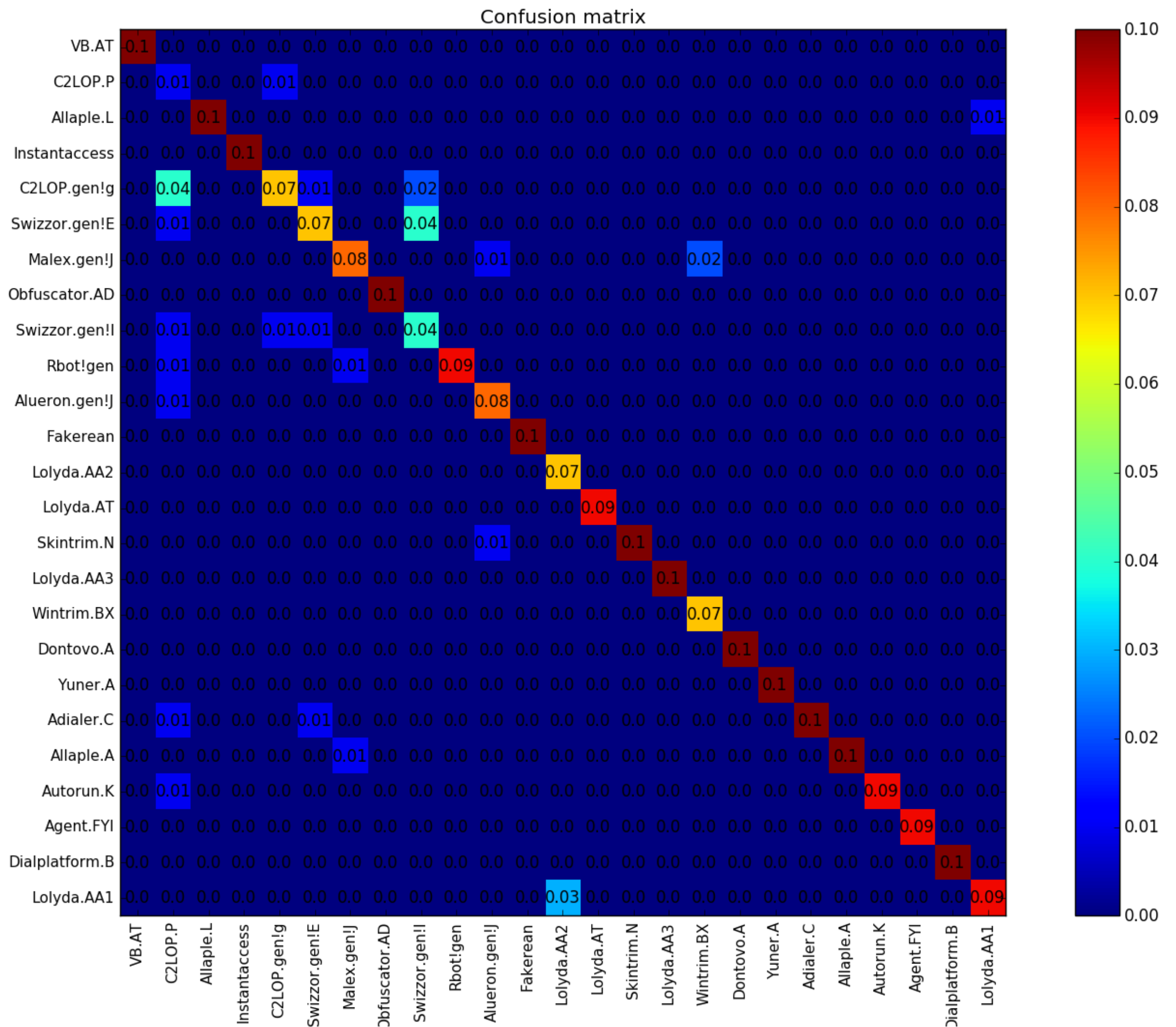


Figure 28: Nearest Centroid Algorithm Classification Results

The results of Nearest Centroid show again that there is no misclassification, but all samples are kind of classified to the correct labels except some minor errors. An emphasis to C2LOP.P should be mentioned as it seems that a small percent is detected and classified correctly but most of the samples are classified into other malware families, so this is not the best algorithm to use to counter similar samples for this kind of malware family.

After ten folds, the times for the above results are:

Training Times:

1st = 0.0834679603577 ,**2nd** = 0.0121638774872,

3rd = 0.0135629177094,**4th** = 0.0206859111786,

5th = 0.0128729343414, **6th** = 0.0129809379578,

7th = 0.0220339298248, **8th** = 0.0146219730377,

9th = 0.0130879878998 ,**10th** = 0.0122499465942

Testing Time = 0.0210800170898

5.1.4 Classification Results for Stochastic Gradient Algorithm

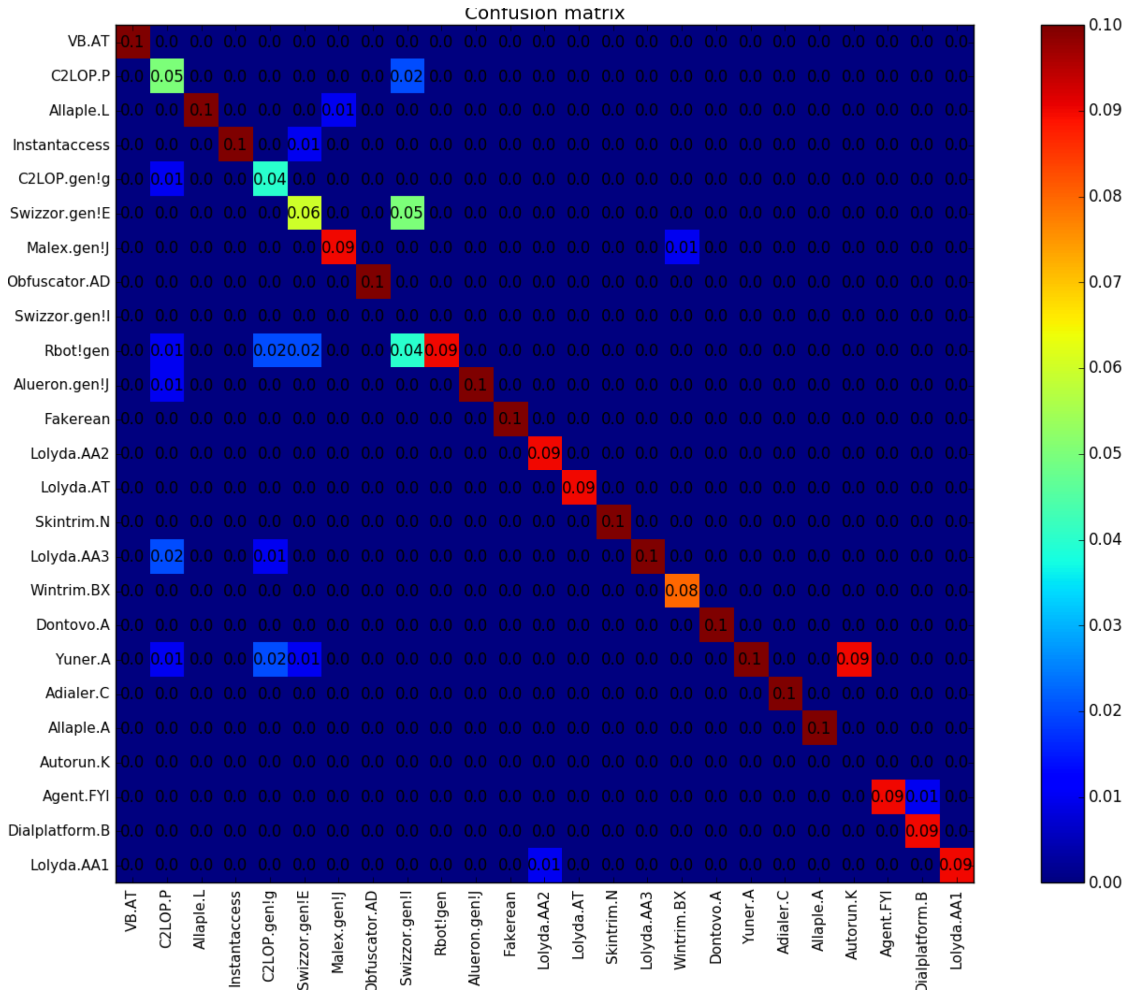


Figure 29: Stochastic Gradient Descent Algorithm Classification Results

In stochastic gradient, there is a misclassification meaning that, for malware detection it is considered as an error. Swizzor.gen!.I were much detected as the other malware family with same similarities Swizzor.gen!.E. Finally, Autorun.K was misclassified to the Yuner.A and most of its samples so it finds similarities between those two. Something that it is encountered in other experiments as well.

After ten folds, the times for the above results are:

Training Times:

1st = 1.25240087509 ,**2nd** = 1.28829216957

3rd = 1.20172715187, **4th** = 1.3295238018,

5th = 1.26634097099 ,**6th** = 1.27604484558,

7th = 1.3377199173 ,**8th** = 1.25761389732,

9th = 1.28988003731, **10th** = 1.41034507751

Testing Time = 0.0584828853607

5.1.5 Classification Results for Perceptron Algorithm

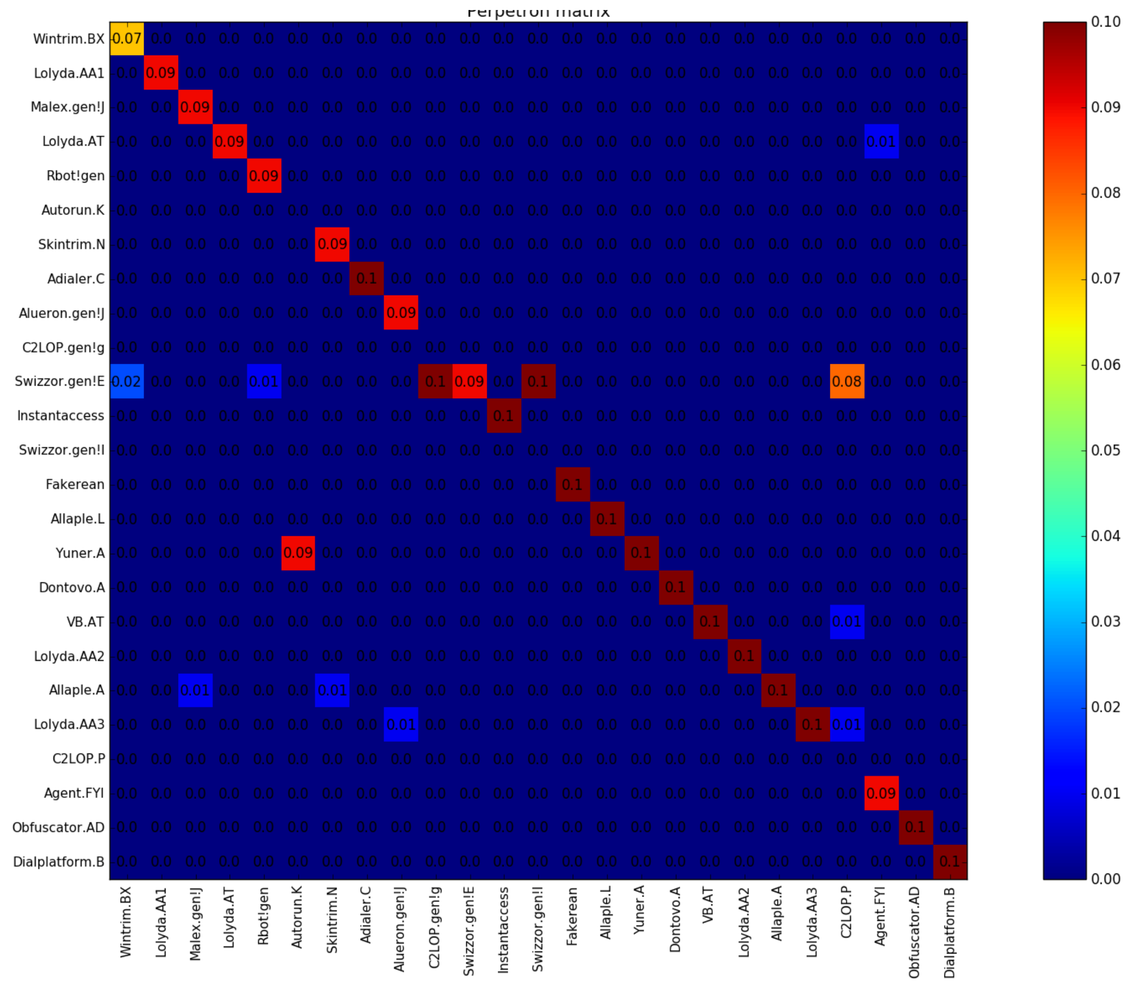


Figure 30: Perceptron Classification Algorithm Results

On Perceptron, for Autorun.K malware family most of their samples were classified into another malware family same goes to C2LOP.gen!g and C2LOP.P, and many similarities were found with Swizzor malware family, something that is also encountered to the results of other experiments with other machine learning algorithms as well.

After ten folds, the times for the above results are:

Training Times:

1st = 0.637459993362, **2nd** = 0.9637799263,**3rd** = 1.10905098915,

4th = 1.08432793617,**5th** = 1.05076909065, **6th** = 0.668966054916,

7th = 0.664897203445, **8th** = 0.669385910034,**9th** = 0.664327144623**,**

10th = 0.655335903168

Testing Time = 0.0148389339447

5.1.6 Classification Results for Multilayer Perceptron Algorithm

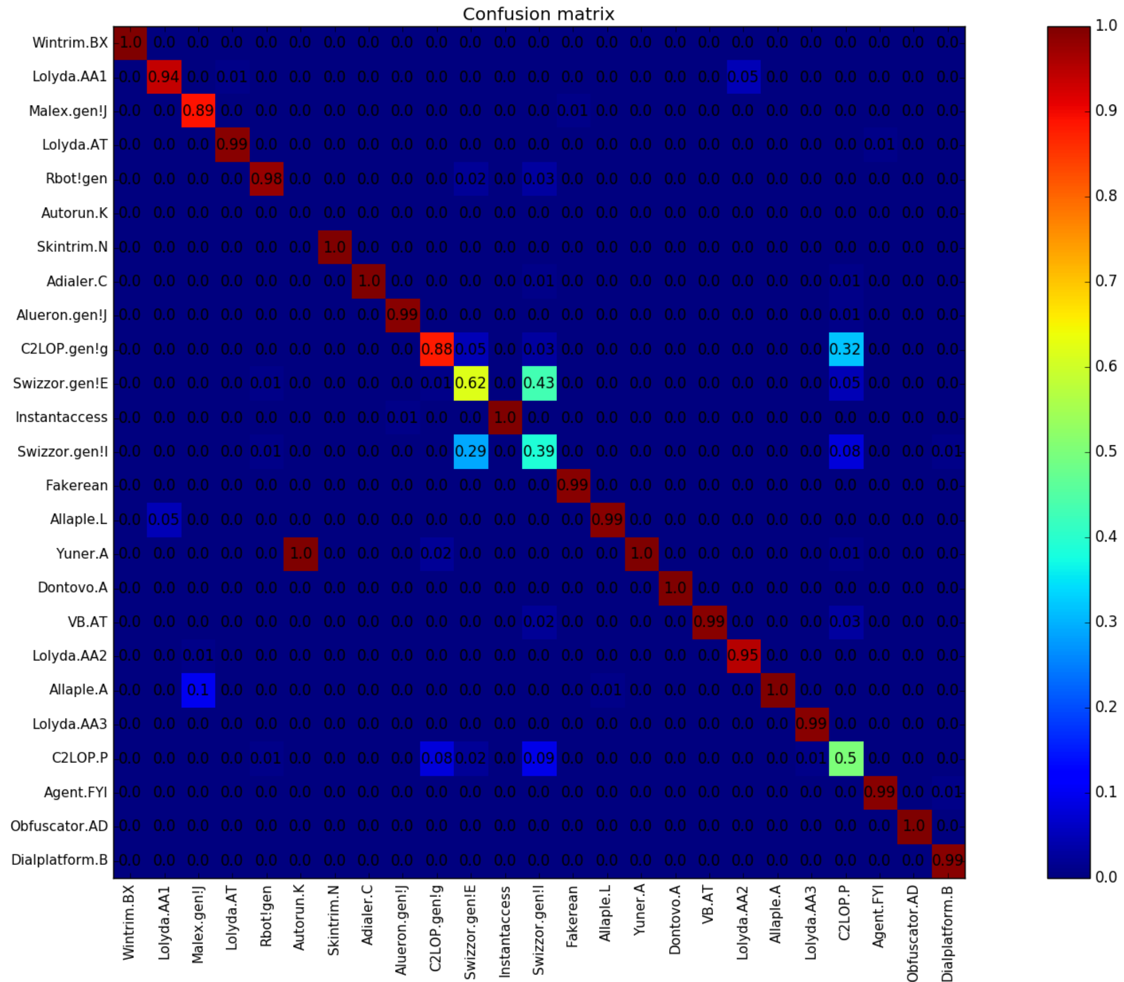


Figure 31: Multilayer Perceptron Algorithm Results

For Multilayer perceptron, Autorun.K is detected and classified to the Yuner.A malware family so training a bit more the algorithm could improve the results of this type of malware. Moreover, it is evident from the matrix produced that the Swizzor variants are being classified accordingly. As a conclusion, Neural Networks and more precisely MLP performed excellent with regard to accuracy and performance, and they should certainly need to test Deep Learning as an asset to counter these types of attacks.

After executing the MLP below is the Training and Testing Times:

Training Times:

[(945, 25)] **1st**= 30.1455199718, [(942, 25)] **2nd** = 15.0493881702,

[(938, 25)] **3rd**= 14.5006198883, [(936, 25)] **4th**= 14.3223490715,

[(935, 25)] **5th**= 13.9758219719, [(935, 25)] **6th** = 14.6204409599,

[(931, 25)] **7th**= 14.6933410168, [(929, 25)] **8th** = 14.4957091808,

[(925, 25)] **9th**= 14.37383008, [(923, 25)] **10th**= 14.3233969212

Testing Times:

[(945, 25)] **1st** = 0.123305082321, [(942, 25)] **2nd** = 0.0920000076294,

[(938, 25)] **3rd** = 0.0852298736572, [(936, 25)]**4th** = 0.0909011363983,

[(935, 25)] **5th** = 0.0852150917053, [(935, 25)] 6**th** = 0.0866298675537,

[(931, 25)] **7th** = 0.0869810581207, [(929, 25)] **8th** = 0.0924861431122,

[(925, 25)] **9th** = 0.0841000080109, [(923, 25)] **10th** = 0.0834050178528

5.1.7 Classification Results for Random Forest

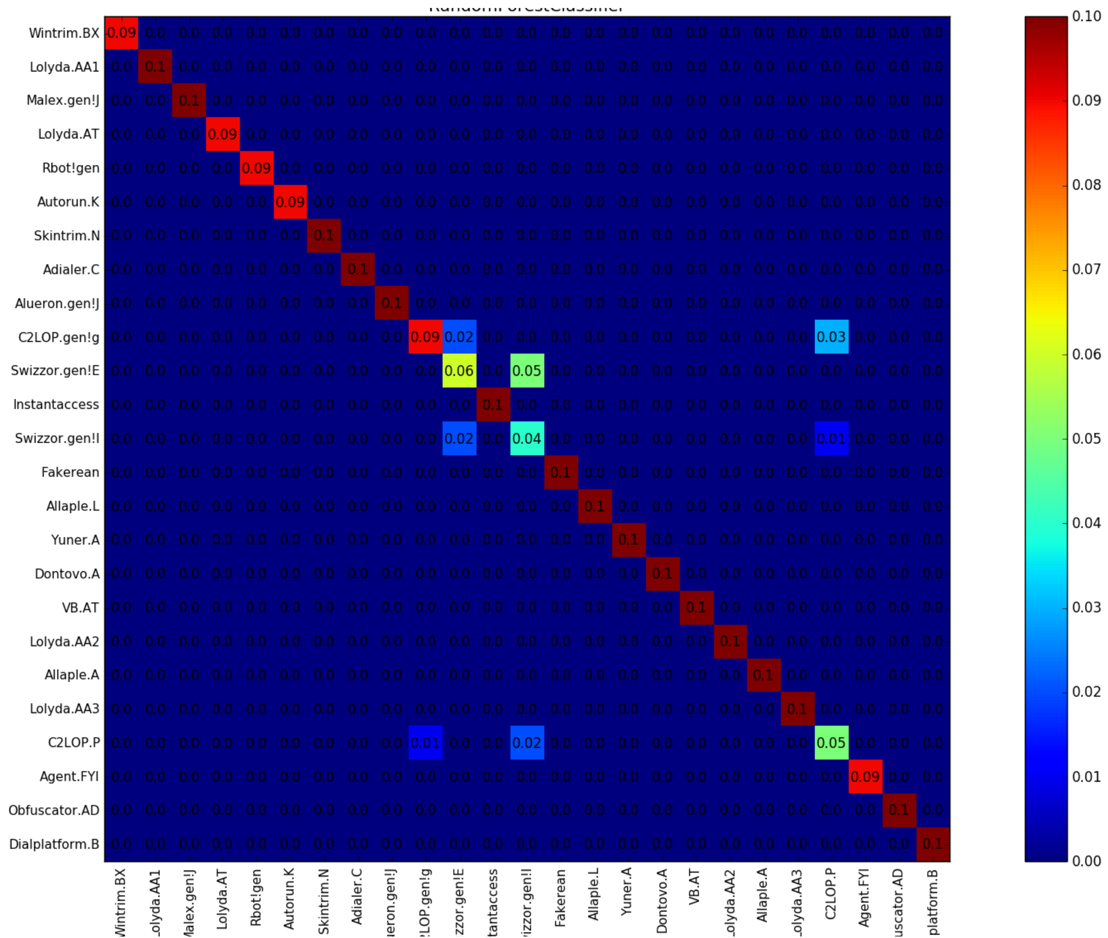


Figure 32: Random Forest Classification Results

Random Forest had the best accuracy results from all other machine learning algorithms but certainly there is always room and discussion for improvement and further research with other datasets. Everything except few blue squares is classified in the correct labels and again Swizzor variants were being detected, no misclassifications were encountered to this experiment.

After ten folds, the times for the above results are

Training Times:

1st = 1.41754007339, **2nd** = 1.60521197319,**3rd** = 1.56024694443, **4th** = 1.66305494308, **5th** = 2.85804891586, **6th** = 2.15528392792,**7th** = 1.42880892754 ,**8th** = 1.46547293663 **9th** = 1.54579615593 ,**10th** = 1.53822803497

Testing Time = 0.00629901885986

5.1.8 Classification Results for Multinomial Naive Bayes

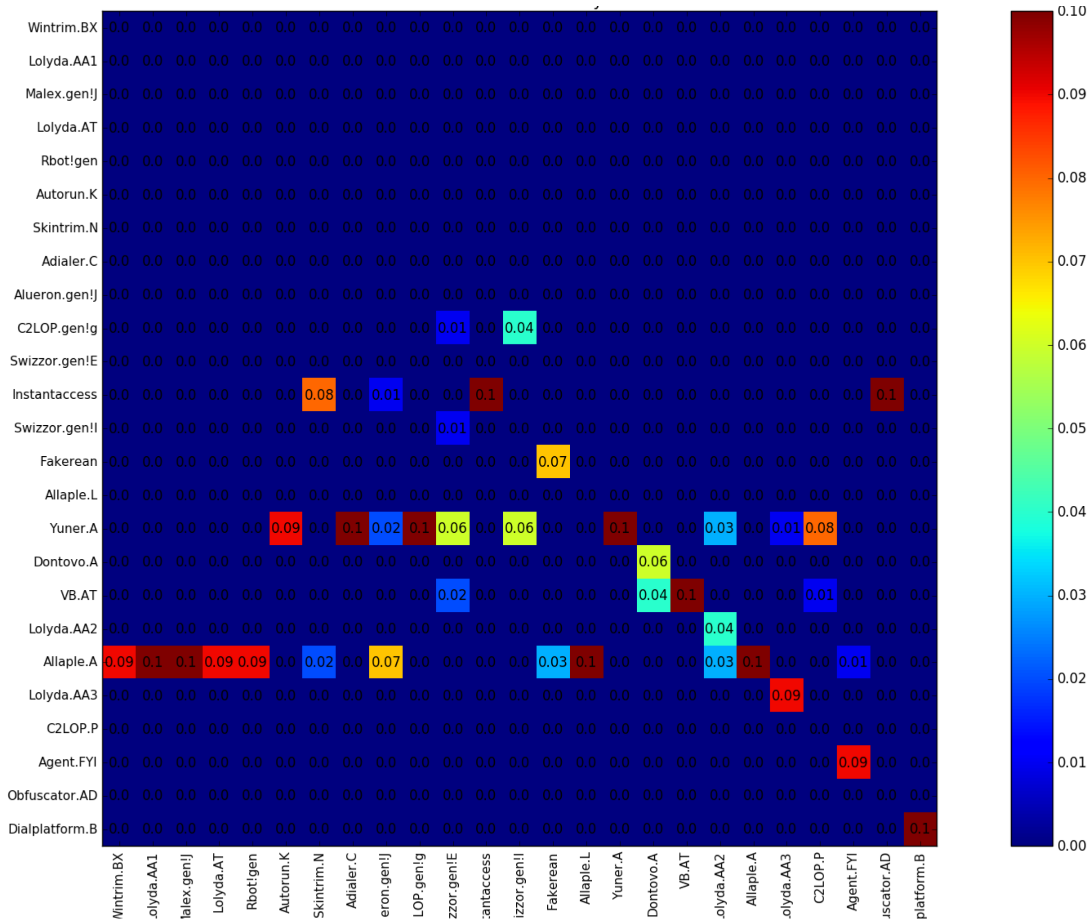


Figure 33: Multinomial Naive Bayes Algorithm Classification Results

Multinomial Naïve Bayes seems to have issues and does not provide us with precise results to identify and make conclusions. From malware detection and classification point of view, a revamp on the initial values should be considered; There is no evidence that malware families were detected and did not find similarities between the samples.

Training Times:

1st = 0.0333349704742, **2nd** = 0.0199840068817,**3rd** = 0.0204889774323,

4th = 0.020693063736, **5th** = 0.0209469795227,**6th** = 0.020231962204,

7th = 0.0201442241669, **8th** = 0.0204341411591,**9th** = 0.0210590362549

Testing Time = 0.00144481658936

5.1.9 Classification Results for Bernoulli

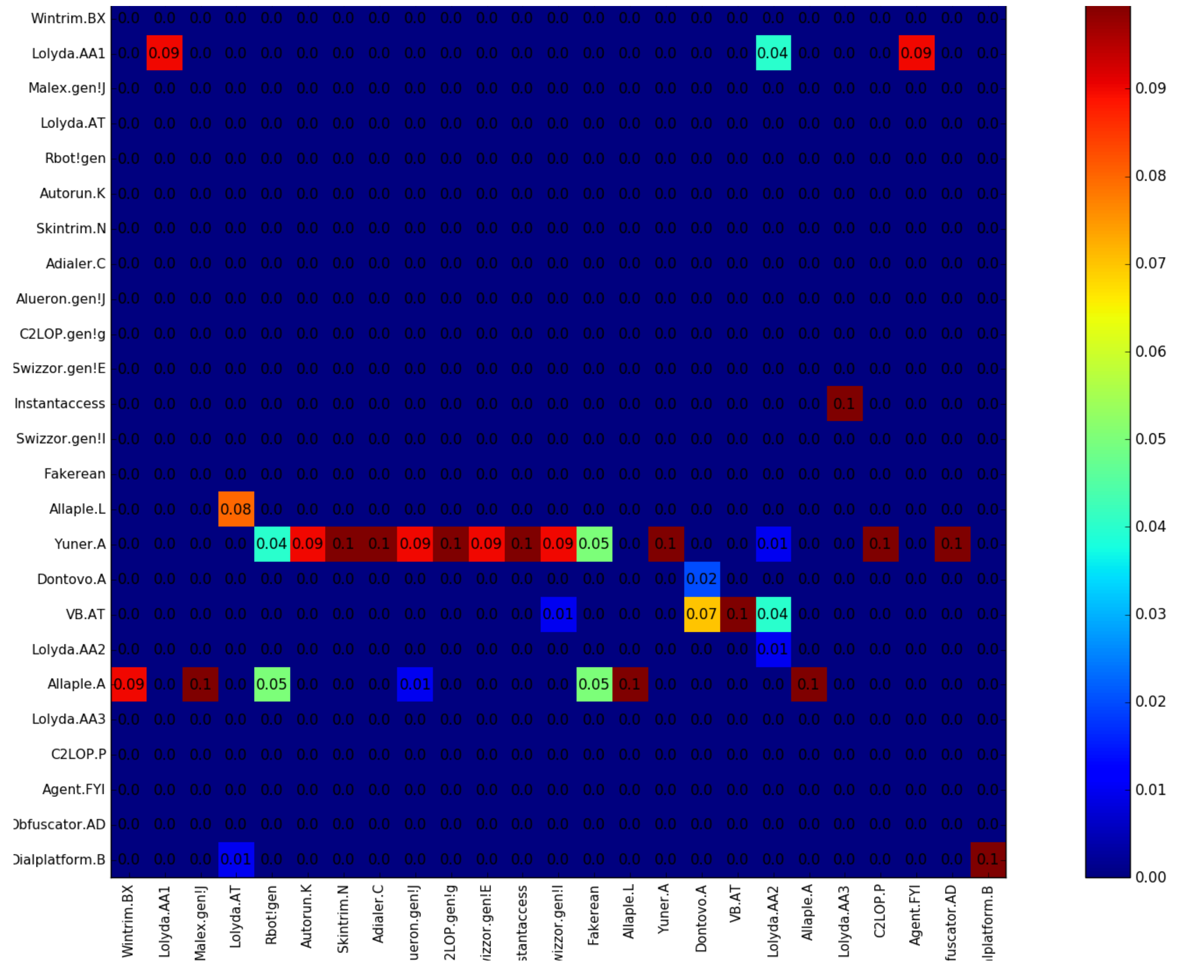


Figure 34: Restricted Boltzmann Machine Algorithm Results

Restricted Boltzmann Machines are unsupervised nonlinear feature learners and here are implemented for a supervised experiment and dataset. The hypothesis is to see how they will perform after the data is shuffled as they usually give good results to a linear problem. This method is popular to deep neural networks, and it is known as unsupervised pre-training. So here results show that RBMBernoulli does work well and there is a need to figure out new ways on how to use them or to conclude that they are no good classifiers and algorithm for malware detection.

Below is some number from the experiment performed by the algorithm.

Iteration 1: pseudo-likelihood = -25.58, time = 8.68s

Iteration 2: pseudo-likelihood = -25.26, time = 8.50s

Iteration 3: pseudo-likelihood = -25.29, time = 8.62s

Iteration 4: pseudo-likelihood = -24.75, time = 7.30s

Iteration 5: pseudo-likelihood = -25.51, time = 7.49s

Iteration 6: pseudo-likelihood = -25.99, time = 9.06s

Iteration 7: pseudo-likelihood = -25.47, time = 9.04s

Iteration 8: pseudo-likelihood = -25.87, time = 7.14s

Iteration 9: pseudo-likelihood = -25.59, time = 6.92s

Iteration 10: pseudo-likelihood = -25.17, time = 8.06s

Iteration 11: pseudo-likelihood = -25.75, time = 8.95s

Iteration 12: pseudo-likelihood = -25.93, time = 8.42s

Iteration 13: pseudo-likelihood = -25.39, time = 8.30s

Iteration 14: pseudo-likelihood = -25.30, time = 7.23s

Iteration 15: pseudo-likelihood = -24.92, time = 8.59s

Iteration 16: pseudo-likelihood = -25.58, time = 8.84s

Iteration 17: pseudo-likelihood = -25.26, time = 8.11s

Iteration 18: pseudo-likelihood = -25.30, time = 9.14s

Iteration 19: pseudo-likelihood = -25.37, time = 8.47s

Iteration 20: pseudo-likelihood = -25.35, time = 9.27s

Training Time: 208.276484966

Testing Time :0.0206489562988

5.2 Clustering Algorithms Results

5.2.1 MeanShift Clustering

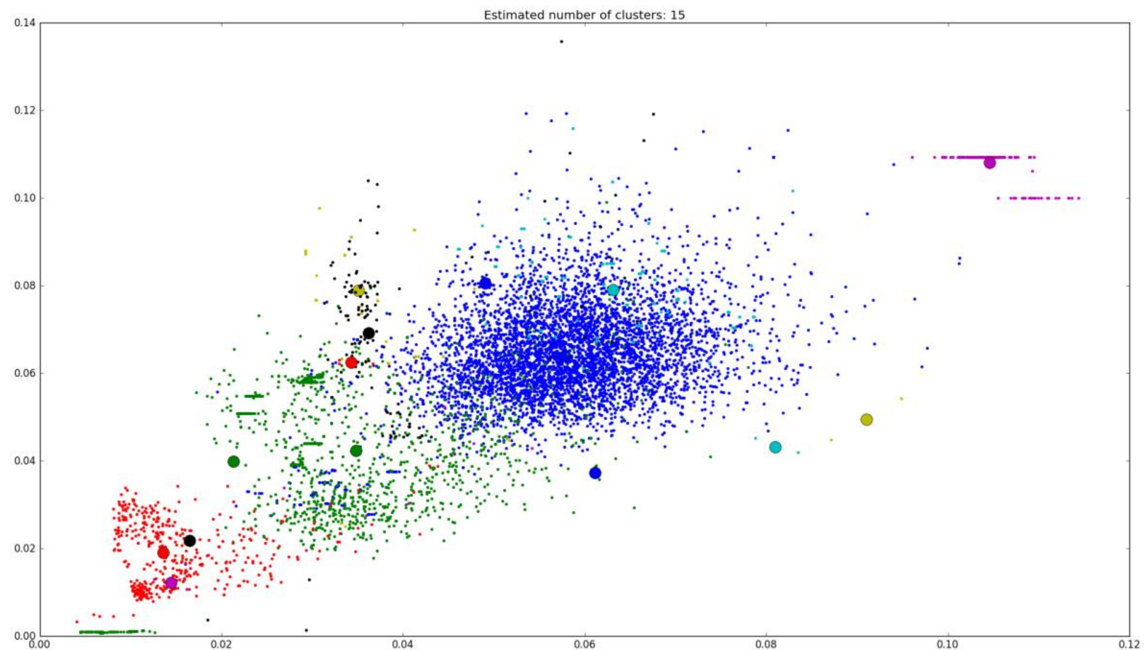


Figure 35: Estimation and Recreation of 15 malware family and clusters after performing MeanShift Algorithm.

Experiments with clustering algorithms were formed with the aim to recreate the dataset meaning that after all the samples have been shuffled by the algorithm, there was a hypothesis to estimate the number of the clusters that are going to be created without knowing if there are the same groups that are presented to the problem. So, in the MeanShift algorithm, an estimation of 15 clusters is shown, a result that is considering not right or correct one as many of the samples probably have been assembled to malware families with entirely different features and behavior. The only thing that it is worth to be mentioned is that for the red cluster are two clusters but are not near each other, meaning that these samples have similarities even if may be in another malware family due to its distance from the Centre. It is not easy to say that the algorithm is not suitable for this purpose, but it needs more investigation regarding the initial and default values

5.2.2 DBScan Clustering

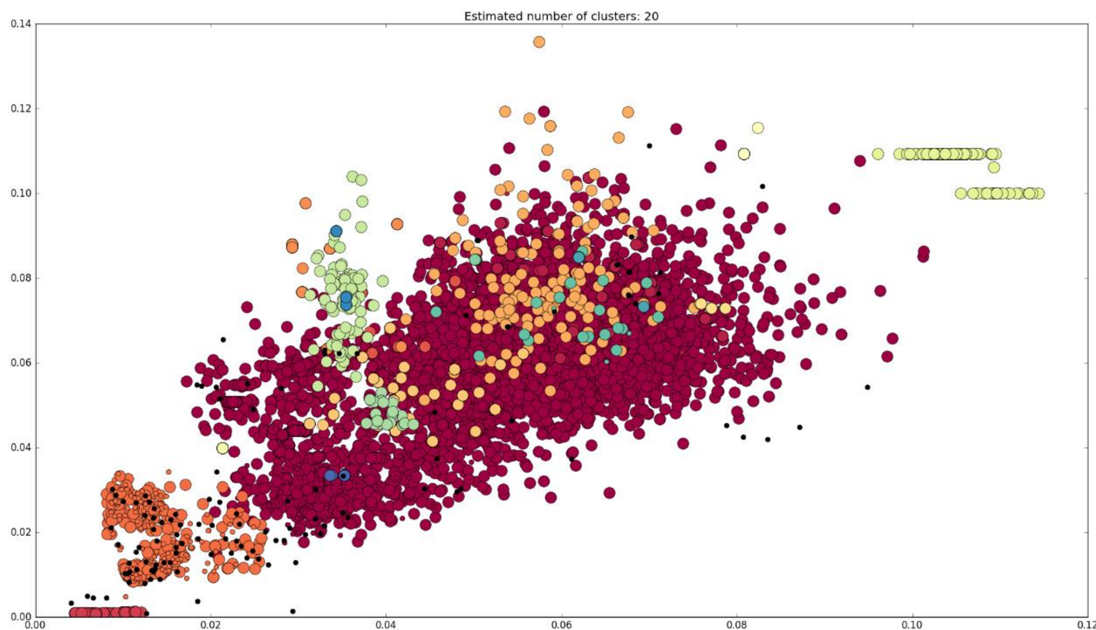


Figure 36: DBScan Algorithm's Estimation of 20 malware families and clusters.

DBScan made an estimation of 20 clusters. As a result, can be considered that it is near to the 25 clusters, but there is no clear explanation why the algorithm put the samples to other malware families. Again, more testing to the values of the algorithms can bring better results. The purple/red clusters are the biggest of all and that some samples are not so near each other so this is a potential miss clustering but also perhaps can be considered as a similarity between different malware families. Large circles are indicating core samples found by the algorithm. Smaller circles are non-core specimens that are still part of a cluster. Moreover, the outliers are shown by black points in *figure 36.*

Estimated number of clusters: 20

Homogeneity: 0.364

Completeness: 0.875

V-measure: 0.514

Adjusted Rand Index: 0.176

Adjusted Mutual Information: 0.357

Silhouette Coefficient: 0.168

5.2.3 Kmeans and Minibatch Clustering

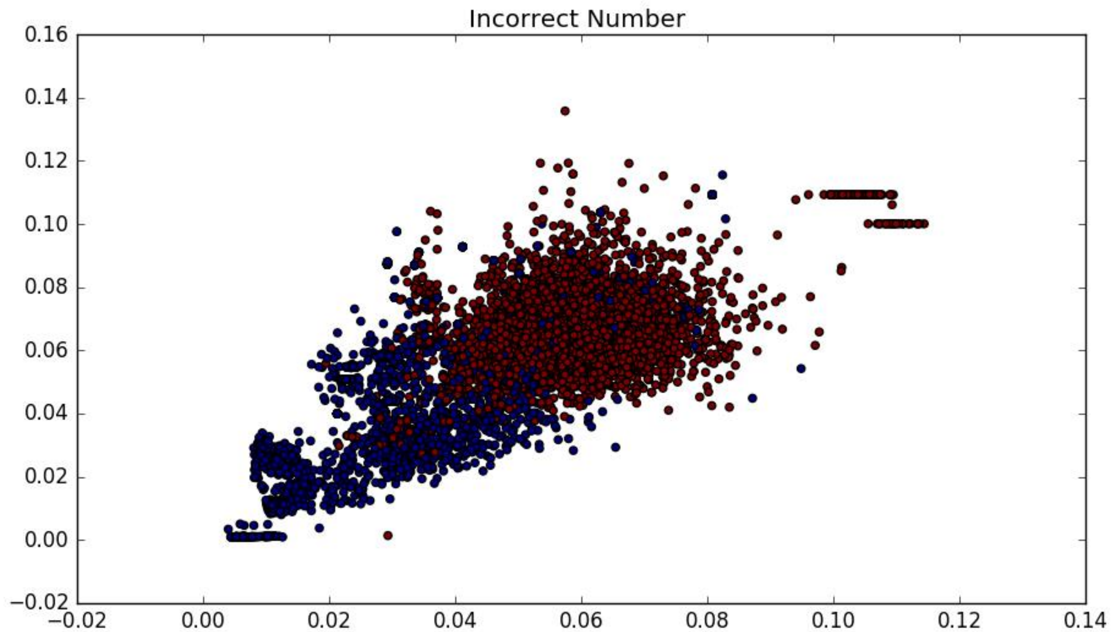


Figure 37: 2 Means Clustering Results

Using only K=2, the samples are equally classified for this dataset but again on blue clustering some samples are not near each other something that happens to red clusters. So, this means that some fingerprints found to be more similar to samples for the blue cluster rather than red's as they should be. Further investigation can show if they should be in the red as well or they are well clustered. Finally, more than K=2 clusters should be considered due to many malware samples that exist.

Figure 38: Comparison of Kmeans – MiniBatchKmeans

Performing Kmeans and MiniBatchKmeans, shown how they treat this kind of a problem and what potential differences may have. On this dataset, they reacted the same as the compare between them shown differences for only three malware samples from the around 9000 that the Malimg dataset has so, to conclude, for this dataset using those algorithms is the same.

6. Discussion

This dissertation researched and aimed at studying the application of Machine Learning and Artificial Neural Network models to the task of detecting malware and malicious activities by classifying their samples into their malware families and presenting, visualizing and converting malware to images. The main principles of featuring and classification in the case of supervised learning were stated. Datasets were introduced and processed to perform malware detection. Numerical experiments were presented to validate the proposed approach. Machine Learning models performed well on Malimg dataset while having a very high training speed, with performances comparable to other malware detection solutions. Even though several efficient solutions have been developed to cope with malware and unknown attacks related to activities on the Internet, future versions of similar malware are expected to become more sophisticated and problematic. Infection media are likely to switch from networked computers to mobile phone terminals as intruders and attackers tend to be attracted to the systems that are the most widely used. Also, newly cloud networked environments are easy targets and should be designed properly to prevent their penetration by malicious software. Moreover, a way to tackle the malware and to be more certain the botnet problem would be to efficiently implement monitoring and filtering, which is difficult due to the diversity of the Internet and the lack of economic incentives for users and ISPs to protect devices and sites.

On this dissertation's research, malware is characterized based on image feature descriptor. The performance proposed and presented for malware classification and clustering is promising. Computer vision and machine-learning techniques for malware analysis will make progress for better and innovative methodologies to analyze malware. However, an image processing based methodology to analyze malware can easily be counter measured from an investigator, or a penetration tester, researcher or attacker that wants to secure or beat the system since this approach depends on global image features. Some countermeasures are moving segments in a binary or the addition of a large number of excessive information. Research for better feature extraction and processing patterns, which consider the distinct characteristics of malware executables, is needed to bypass,

defend and tackle against such attacks. A potential expansion is to divide the image regions and characterize the local texture and spatial distribution of these texture patterns. So, feature engineering could and should become better, with more research to this field. Although clustering and classification are similar, the former is unsupervised, and the latter is supervised.

For future, there is a need to improve the algorithms suggested or propose new ones. Discover new and better clustering algorithms, as malware analysis and detection field, is new and in a research mode now. Another approach on how malware can be visualized is the development of whole new visualization algorithms. So, an interesting approach could be to test the same dataset with the new transformations and other datasets as well.

To conclude, this dissertation was to emphasize the importance for keeping investigating malware samples, their behavior and see how other algorithms work classification and clustering-wise. Malware day by day are made to bypass anti-viruses, firewalls and detection systems, and there is a need to re-evaluate current approaches and rethink how to approach them.

Bibliography-References

[1] http://www.codemicon.com/,C,(2016),*Heartbleed Bug*:[online] http://heartbleed.com/ [Accessed: July 2016].

[2] Ransomware and Businesses, Symantec [online] Available at: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/ISTR2016_Ransomware_and_Businesses.pdf [Accessed: July 2016].

[3] GmbH, A (2016) *AV-Test-The Independent IT-Security Institute*. [online] Av-test.org. Available at: https://www.av-test.org/en/press/test-results/ [Accessed: July 2016].

[4] Michael Sikorski. Andrew Honig., *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. 1st ed. No Starch Press, 2012. Print.

[5] Savan Gadhiya, Kaushal Bhavsa , Techniques for Malware Analysis, *International Journal of Advanced Research in Computer Science and Software Engineering*.[online] Available at: http://www.ijarcsse.com/docs/papers/Volume_3/4_April2013/V3I4-0371.pdf[Accessed: July 2016]

[6] Practical Malware Analysis, Kris Kendall. [online] Available at: www.blackhat.com/presentations/bh-dc-07/Kendall_McMillan/Paper/bh-dc-07-Kendall_McMillan-WP.pdf [Accessed: August 2016]

[7] Nataraj, L., Karthikeyan, S., Jacob, G., & Manjunath, B. S. (2011). Malware Images: Visualization and Automatic Classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security* (p. 4:1–4:7). New York, NY, USA: ACM. https://doi.org/10.1145/2016904.2016908

[8] Kujawa, Adam and Adam Kujawa. "Malwarebytes 2013 Threat Report". *Malwarebytes Labs*. N.p., 2016.[online] Available at: https://blog.malwarebytes.com/securityworld/2013/12/malwarebytes-2013-threat-report/ [Accessed: October 2016].

[9] The Art of Unpacking. Mark Vincent Yason. *IBM Security Systems*. [online] Available at: https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf [Accessed: July 2016]

[10] "Microsoft Malware Protection Center - Malware Naming Conventions". *Microsoft.com*. N.p., 2016. [online] Available at:https://www.microsoft.com/security/portal/mmpc/shared/malwarenaming.aspx [Accessed: September 2016].

[11] "Common Malware Types: Cybersecurity 101". *Veracode*. N.p., 2016. [online] Available at: https://www.veracode.com/blog/2012/10/common-malware-types-cybersecurity-101 [Accessed: July 2016].

"Ransomware". *En.wikipedia.org*. N.p., 2016. [online] Available at: https://en.wikipedia.org/wiki/Ransomware [Accessed: July 2016].

Hacking et al. "Malware Types Explained - Hacking Tutorials." *Hacking Tutorials*. N.p., 2016. [online] Available at:http://www.hackingtutorials.org/malware-analysis-tutorials/malware-types-explained/ [Accessed: July 2016].

[12] A Comparative Assessment of Malware Classification using Binary Texture Analysis and Dynamic Analysis-Lakshmanan Nataraj, Vinod Yegneswaran, Phillip Porras, Jian Zhang *Proceedings of ACM CCS Workshop on Artificial Intelligence and Security(AISEC,2011)*.

[13] Oktavianto, D., & Muhardianto, I. (2013). *Cuckoo Malware Analysis*.

Packt Publishing

[14] Schultz, M. G., Eskin, E., Zadok, F., & Stolfo, S. J. (2001). Data mining methods for detection of new malicious executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001* (pp. 38–49). https://doi.org/10.1109/SECPRI.2001.924286

[15] Kolter, J. Z., & Maloof, M. A. (2006). Learning to Detect and Classify Malicious Executables in the Wild. *J. Mach. Learn. Res.*, *7*, 2721–2744.

[16] Kong, D., & Yan, G. (2013). Discriminant Malware Distance Learning on Structural Information for Automated Malware Classification. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 1357–1365). New York, NY, USA: ACM. https://doi.org/10.1145/2487575.2488219

[17] Tian, R., Islam, R., Batten, L., & Versteeg, S. (2010). Differentiating malware from clean ware using behavioral analysis. *In 2010 5th International Conference on Malicious and Unwanted Software* (pp. 23-30).https://doi.org/10.1109/MALWARE.2010.5665796

[18] Santos, I., Devesa, J., Brezo, F., Nieves, J., & Bringas, P. G. (2013). OPEM: A Static-Dynamic Approach for Machine-Learning-Based Malware Detection. In Á. Herrero, V. Snášel, A. Abraham, I. Zelinka, B. Baruque, H. Quintián, E. Corchado (Eds.), *International Joint Conference CISIS'12-ICEUTE´12-SOCO´12 Special Sessions* (pp. 271–280). Springer Berlin Heidelberg. http://link.springer.com/chapter/10.1007/978-3-642-33018-6_28

[19] Siddiqui, M., Wang, M. C., & Lee, J. (2009). Detecting Internet worms using data mining techniques. *Journal of Systemics, Cybernetics, and Informatics*, *6*(6), 48–53.

[20] Zolkipli, M. F., & Jantan, A. (2011). An approach for malware behavior identification and classification. In *2011 3rd International Conference on Computer Research and Development* (Vol. 1, pp. 191–194).

https://doi.org/10.1109/ICCRD.2011.5764001

[21] Rieck, K., Trinius, P., Willems, C., & Holz, T. (2011). Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, *19*(4), 639–668. https://doi.org/10.3233/JCS-2010-0410

[22] Anderson, B., Quist, D., Neil, J., Storlie, C., & Lane, T. (2011). Graph-based malware detection using dynamic analysis. *Journal in Computer Virology*, *7*(4), 247–258. https://doi.org/10.1007/s11416-011-0152-x

[23] Bayer, U., Comparetti, P.M., Hlauschek, C. and Kruegel, C. (2009) Scalable, Behavior-Based Malware Clustering. *Proceedings of the 16th Annual Network and Distributed System Security Symposium.*

[24] Biley. Worm Detection by Combination of Classification With Neural Networks Retrieved from http://www.iajet.org/iajet_files/vol.3/no.2/Worm%20Detection%20by%20Combination%20of%20Classification%20With%20Neural%20Networks.pdf

[25] Park, Y., Reeves, D., Mulukutla, V., & Sundaravel, B. (2010). Fast Malware Classification by Automated Behavioral Graph Matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research* (p. 45:1–45:4). New York, NY, USA: ACM. https://doi.org/10.1145/1852666.1852716

[26] Firdausi, I., Lim, C., Erwin, A., & Nugroho, A. S. (2010). Analysis of Machine Learning Techniques Used in Behavior-Based Malware Detection. *In 2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies (pp. 201–203).* https://doi.org/10.1109/ACT.2010.33

[27] Santos, I., Nieves, J. and Bringas, P.G. (2011) Collective Classification for Unknown Malware Detection. *Proceedings of the International Conference on Security and Cryptography*, Seville, 18-21 July 2011, 251-256

[28] Nari, S., & Ghorbani, A. A. (2013). Automated malware classification based on network behavior. In *2013 International Conference on Computing, Networking and Communications (ICNC)* (pp. 642–647). https://doi.org/10.1109/ICCNC.2013.6504162

[29] Lee, T., Mody, J., Lin, Y., Marinescu, A., & Polyakov, A. (2007, June 14). Application behavioral classification. Retrieved from http://www.google.com/patents/US20070136455

Lee, T., and Mody, J.J. (2006) Behavioral Classification. *Proceedings of the European Institute for Computer Antivirus Research Conference* (EICAR'06)

[30] Islam, R., Tian, R., Batten, L. M., & Versteeg, S. (2013). Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications*, *36*(2), 646–656. https://doi.org/10.1016/j.jnca.2012.10.004

[31] Tian, R., Batten, L. and Versteeg, S. (2008) Function Length as a Tool for Malware Classification. *Proceedings of the 3rd International Conference on Malicious and Unwanted Software*, Fairfax, 7-8 October 2008, 57-64

[32] Anderson, B., Storlie, C. and Lane, T. (2012) Improving Malware Classification: Bridging the Static/Dynamic Gap. *Proceedings of 5th ACM Workshop on Security and Artificial Intelligence* (*AISec*), 3-14

[33] Oliva, A. & Torralba, A. (2001). Modeling the Shape of the Scene: A Holistic Representation of the Spatial Envelope∗. *International Journal Of Computer Vision, 2001*, *Vol. 42*(3), 145-175. Available at: https://people.csail.mit.edu/torralba/code/spatialenvelope/

[34] SARVAM: Search and Retrieval of Malware- Lakshmanan Nataraj, Dhilung Kirat, Giovanni Proc. *Annual Computer Security Applications Conference (ACSAC) Workshop on Next Generation Malware Attacks and Defense (NGMAD*), New Orleans, Dec. 2013, http://sarvam.ece.ucsb.edu (Vigna, 2013).

Sarvam (2016) *Blog about Malware Classification*-[online] Available at: http://sarvamblog.blogspot.gr/ [Accessed: October 2016].

[35] "Tutorial — Lasagne 0.2. Dev1 Documentation". *Lasagne.readthedocs.org*. N.p., 2016. [online] Available at: http://lasagne.readthedocs.org/en/latest/user/tutorial.html [Accessed: October 2016]

Tensorflow. *TensorFlow*. N.p., 2016. Tensor Flow Documentation and Tutorials: [online]Available at: https://www.tensorflow.org/versions/master/tutorials/index.html [Accessed: November 2016].

[36] "Scikit-Learn: Machine Learning in Python — Scikit-Learn 0.18.1 Documentation". *Scikit-learn.org*. N.p., [online]. Available at: http://scikit-learn.org/stable/index.html [Accessed: October2016]

Scikit-neuralnetwork.readthedocs.io (2016). *Scikit-neuralnetwork documentation* [online] Available at: http://scikit-neuralnetwork.readthedocs.io/en/latest/index.html# [Accessed: October 2016].

Welcome to OpenCV documentation! — OpenCV 2.4.13.1 documentation. (2016). *Docs.opencv.org*. [online] Available at : http://docs.opencv.org/2.4/index.html

[Accessed: September 2016]

Machine learning. (2016). *En.wikipedia.org*. [online] Available at:

https://en.wikipedia.org/wiki/Machine_learning [Accessed: October 2016]

[37] Ruder, Sebastian. "An Overview Of Gradient Descent Optimization Algorithms." *Blog*. N.p., 2016. [online] Available at: http://sebastianruder.com/optimizing-gradient-descent/ [Accessed: October 2016]

[38] Bäcklund, Henrik, Anders Hedblom, and Niklas Neijman. *A Density Based Spatial Clustering of Application with Noise*. 1st ed. the Linkopings Universitet, 2011. [online] Available at:http://webstaff.itn.liu.se/~aidvi/courses/06/dm/Seminars2011/DBSCAN(4).pdf [Accessed: October 2016]

Further Reading and Related Content

[1] Implementation of Malware Analysis using Static and Dynamic Analysis Method Syarif Yusirwan S, Yudi Prayudi, Imam Riadi.[online] Available at:http://research.ijcaonline.org/volume117/number6/pxc3902943.pdf [Accessed: September 2016]

[2] Hardikar, Aman. Malware 101-Viruses.*SANS Institute InfoSec Reading Room.* 1st ed. 2008. [online] Available at: https://www.sans.org/reading-room/whitepapers/incident/malware-101-viruses-32848 [Accessed: July 2016]

[3] A Survey on Automated Dynamic Malware Analysis Techniques and Tools.[online].Available at: https://www.sba-research.org/wpcontent/uploads/publications/malware_survey.pdf [Accessed: November 2016]

[4] Malware Analysis and Classification: A Survey- Gandotra, E., Bansal, D. and Sofat, S. *Journal of Information Security, 5, 56-64.* doi: 10.4236/jis.2014.52006 (2014).

[5] Eilam, Eldad, and Elliot J Chikofsky. *Reversing*. 1st ed. Indianapolis, IN: Wiley, 2005. Print.

[6] Hoglund, Greg, and James Butler. *Rootkits: Subverting the Windows Kernel*. 1st ed. Addison Wesley Professional, 2005. Print.

[7] Erickson, Jon. *Hacking*. 1st ed. San Francisco, Calif.: No Starch Press, 2008. Print.

[8] Makan, Keith. *Penetration Testing with The Bash Shell*. 1st ed. Birmingham, UK: Packt Pub., 2014. Print.

[9] "Microsoft Malware Classification Challenge (BIG 2015) | Kaggle". *Kaggle.com*. N.p., 2016. [online] Available at: https://www.kaggle.com/c/malware-classification [Accessed: July 2016].

[10] Hasherezade's Malware to Image Converter. [online] Available at: https://github.com/hasherezade/crypto_utils/blob/master/file2png.py [Accessed: November 2016]

[11] "Malware." *En.wikipedia.org*. N.p., 2016. [online]. Available at: https://en.wikipedia.org/wiki/Malware [Accessed: July 2016]

[12] "Malware Analysis." *En.wikipedia.org*. N.p., 2016. [online] Available at: https://en.wikipedia.org/wiki/Malware_analysis [Accessed: August 2016]

[13] «Python Software Foundation Wiki Server." *Wiki.python.org*. N.p., 2016. [online] Available at: https://wiki.python.org [Accessed: September 2016]

[14] Xiaozhou Wang, Jiwei Liu, Xueer Chen. *Microsoft Malware Classification Challenge (BIG 2015):* First Place Team: Say No to Overfitting

[15] Grégio, A., de Geus, P., Kruegel, C., & Vigna, G. (2013). Tracking Memory Writes for Malware Classification and Code Reuse Identification. *Detection Of Intrusions And Malware, And Vulnerability Assessment*, 134-143. http://dx.doi.org/10.1007/978-3-642-37300-8_8

[16]Andrew Davis, Matt Wolff, Deep Learning On Disassembly Data *Black Hat 2015 USA* [online]Available at https://www.blackhat.com/docs/us-15/materials/us-15-Davis-Deep-Learning-On-Disassembly.pdf [Accessed: September 2016].

[17] Sarvam.ece.ucsb.edu (2016). *Sarvam: Search and Retrieval for Malware* [online] Available at http://sarvam.ece.ucsb.edu/. [Accessed: July 2016].

[18] Bailey, Katherine. "Adventures Learning Neural Nets And Python." *ohAI*. N.p., 2016. [online] Available at: https://katbailey.github.io/post/neural-nets-in-python/ [Accessed: October 2016]

[19] Bailey, M., Oberheide, J., Andersen, J., Mao, Z. M., Jahanian, F., & Nazario, J. (2007). Automated classification and analysis of Internet malware. In *Recent Advances in Intrusion Detection - 10th International Symposium, RAID 2007, Proceedings.* (Vol. 4637 LNCS, pp. 178-197). (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Vol. 4637 LNCS).

[20] Park, Y., Reeves, D., Mulukutla, V., & Sundaravel, B. (2010). Fast malware classification by automated behavioral graph matching. *Proceedings Of The Sixth Annual Workshop On Cyber Security And Information Intelligence Research - CSIIRW '10*. http://dx.doi.org/10.1145/1852666.1852716

[21] David, O. & Netanyahu, N. (2015). DeepSign: Deep learning for automatic malware signature generation and classification. *2015 International Joint Conference On Neural Networks (IJCNN)*. http://dx.doi.org/10.1109/ijcnn.2015.7280815

[22] Zhang, Y., Pang, J., Yue, F., & Cui, J. (2010). Fuzzy Neural Network for Malware Detect. *2010 International Conference On Intelligent System Design And Engineering Application*. http://dx.doi.org/10.1109/isdea.2010.314

[23] Poster: Deep Learning for Zero-Day Flash Malware Detection. 1st ed. Wookhyun Jung, Sangwon Kim, Sangyong Choi, 2015. [online]Available at: http://www.ieee-security.org/TC/SP2015/posters/paper_34.pdf [Accessed: August 2016]

[24] Making Machine Learning Anomaly Detectors in Real Life. Clarence Chio: Available at https://www.youtube.com/watch?v=-aL8mn9pJ_s [Accessed: November 2016]

Appendix

Source Code

1.Dataset Arrangement

```
import os, glob, numpy

os. chdir('/home/user/Desktop/malimg_dataset') # the parent folder with sub-folders

list_fams = os. listdir (os. getcwd ()) # vector of strings with family names

no_imgs = [] # No. of samples per family

for i in range(len(list_fams)):

  os. chdir(list_fams[i])

  len1 = len(glob. glob('*.png')) # assuming the images are stored as 'png'

  no_imgs.append(len1)

  os. chdir ('..')


total = sum(no_imgs) # total number of all samples

y = numpy. zeros(total) # label vector


temp1 = numpy.zeros(len(no_imgs) + 1)

temp1[1: len(temp1)] = no_imgs

temp2 = int(temp1[0]) # now temp2 is [0 no_imgs]

for jj in range(len(no_imgs)):

  temp3 = temp2 + int(temp1[jj + 1])

  for ii in range(temp2, temp3):

    y[ii] = jj

  temp2 = temp2 + int(temp1[jj + 1])
```

2. Features Computation

```
import Image, leargist

X = numpy. zeros((sum(no_imgs), 320)) # Feature Matrix
cnt = 0
for i in range(len(list_fams)):
  os.chdir(list_fams[i])
  img_list = glob. glob('*.png') # Getting only 'png' files in a folder
  for j in range(len(img_list)):
    im = Image.open(img_list[j])
    im1 = im.resize((64, 64), Image.ANTIALIAS); # for faster computation
    des = leargist. color_gist(im1)
    X[cnt] = des [0:320]
    cnt = cnt + 1
  os. chdir ('..')
import random
from sklearn. cross validation import StratifiedKFold
from sklearn. utils import shuffle

n_samples, n_features = X. shape
p = range(n_samples) # an index array, 0: n_samples
random. seed (random. random ())
random. Shuffle(p) # the index array is now shuffled

X, y = X[p], y[p] # both the arrays are now shuffled
```

```
kfold = 10 # no. of folds (better to have this at the start of the code)

skf = StratifiedKFold (y, kfold) # indices='true'

# Stratified KFold: This first divides the data into k folds. Then it also makes sure that the distribution of the data in each fold follows the original input distribution
# Note: in future versions of scikit. learn, this module will be fused with kfold

skfind = [None] * len(skf) # indices
cnt = 0
for train_index in skf:
  skfind[cnt] = train_index
  cnt = cnt + 1


conf_mat = numpy. zeros((len(no_imgs), len(no_imgs))) # Initializing the Confusion Matrix

n_neighbors = 1 # better to have this at the start of the code
# 10-fold Cross Validation

for i in range(kfold):
  train_indices = skfind[i][0]
  test_indices = skfind[i][1]
  clf = []
  X_train = X[train_indices]
  X_val = X[train_indices]
```

```
y_train= y[train_indices]
y_val = y[train_indices]
X_test = X[test_indices]
y_test = y[test_indices]
```

3.Multilayer Perceptron with Theano-experiment that had some issues

```
############################################################
####################
# Training
# Hyper-parameters. These were set by cross-validation,
# using a GridSearchCV. Here not performing cross-validation
to
# save time.
# More components tend to give better prediction performance,
but larger
# fitting time
# Training RBM-Logistic Pipeline
# Training Logistic regression

import time
import theano
import theano. tensor as T
import lasagne
from lasagne. regularization import regular-
ize_layer_params_weighted, l2, l1
import numpy as np
import matplotlib. pyplot as plt
from numpy import *
```

```
# Uses Lasagne to train a multi-layer perceptron, adapted from
# http://lasagne.readthedocs.org/en/latest/user/tuto-rial.html
def lasagne_mlp (X_train, y_train, X_val, y_val, X_test, y_test, hidden_units=25, num_epochs=500, l2_param = 0.01, use_dropout=True):

  # Prepare Theano variables for inputs and targets
  input_var = T. tensor3('inputs')
  target_var = T. ivector('targets')

  print("Building model and compiling functions...")
  # Input layer
  network = lasagne. layers. InputLayer (shape= (None, 1, 400),
                 input_var=input_var)


 if use_dropout:
    # Apply 20% dropout to the input data:
    network = lasagne. layers. DropoutLayer (network, p=0.2)

  # A single hidden layer with number of hidden units as specified in the
  # parameter.
  l_hid1 = lasagne. layers. DenseLayer (
      network, num_units=hidden_units,
      nonlinearity=lasagne. nonlinearities. rectify,
      W=lasagne.init. GlorotUniform ())
```

```
if use_dropout:
    # Dropout of 50%:
    l_hid1_drop = lasagne. layers. DropoutLayer (l_hid1,
p=0.5)
    # Fully-connected output layer of 10 softmax units:
    network = lasagne. layers. DenseLayer (
      l_hid1_drop, num_units=10,
      nonlinearity=lasagne. nonlinearities. Softmax)
  else:
    # Fully-connected output layer of 10 softmax units:
    network = lasagne. layers. DenseLayer (
      l_hid1, num_units=10,
      nonlinearity=lasagne. nonlinearities. softmax)


 # Loss expression for training
  prediction = lasagne.layers.get_output(network)
  loss = lasagne. objectives. categorical_crossentropy (pre-
diction, target_var)
  loss = loss. mean ()
  # Regularization.
  l2_penalty = lasagne. regularization. Regular-
ize_layer_params_weighted ({l_hid1: l2_param}, l2)
  loss = loss + l2_penalty
  # Update expressions for training, using Stochastic Gradi-
ent Descent.
  params = lasagne.layers.get_all_params (network, traina-
ble=True)
  updates = lasagne. updates. nesterov_momentum (
      loss, params, learning_rate=0.01, momentum=0.9)
```

```
# Loss expression for evaluation.
test_prediction = lasagne.layers.get_output (network, de-
terministic=True)
test_loss = lasagne.objectives.categorical_crossen-
tropy(test_prediction,
                            target_var)
test_loss=test_loss. mean ()
# Expression for the classification accuracy:
test_acc = T. mean (T. eq (T. argmax (test_prediction,
axis=1), target_var),
         dtype=theano.config. floatX)

# Compile a function performing a training step on a mini-
batch (by giving
# the updates dictionary) and returning the corresponding
training loss:
train_fn=theano. function ([input_var, target_var], loss,
updates=updates)

# Compile a second function computing the validation loss
and accuracy:
val_fn = theano. function ([input_var, target_var],
[test_loss, test_acc])

# Finally, launch the training loop.
print ("Starting training...")
# Keep track of training and validation cost over the
epochs
epoch_cost_train=np. empty (num_epochs, dtype=float32)
epoch_cost_val=np. empty (num_epochs, dtype=float32)
```

```
 #iterate over epochs:
 for epoch in range(num_epochs):
   # In each epoch,do a full pass over the training data:
   train_err = 0
   # want to keep track of the deterministic (feed-forward)
   # training error.
   train_err_ff = 0
   train_batches = 0
   start_time = time. time ()
   for batch in iterate_minibatches (X_train, y_train, 50,
shuffle=True):
      inputs, targets = batch
      err, acc = val_fn(inputs, targets)
      train_err_ff += err
      train_err += train_fn(inputs, targets)

     train_batches += 1



   # And a full pass over the validation data:
   val_err = 0
   val_acc = 0
   val_batches = 0
   for batch in iterate_minibatches (X_val, y_val, 50, shuf-
fle=False):
      inputs, targets = batch
      err, acc = val_fn (inputs, targets)
      val_err += err
      val_acc += acc
```

```
      val_batches += 1

    epoch_cost_train[epoch] = train_err_ff / train_batches
    epoch_cost_val[epoch] = val_err / val_batches
    # print the results for this epoch:
    print ("Epoch {} of {} took {:.3f}s".format(
      epoch + 1, num_epochs, time.time() - start_time))
    print("    training  loss:\t\t{:.6f}".format(train_err /
train_batches))
    print("    validation  loss:\t\t{:.6f}".format(val_err /
val_batches))
    print("  validation accuracy:\t\t{:.2f} %".format(
      val_acc / val_batches * 100))


 # After training,compute and print the test error:
  test_err = 0
  test_acc = 0
  test_batches = 0
  for batch in iterate_minibatches(X_test, y_test, 50, shuf-
fle=False):
    inputs, targets = batch
    err, acc = val_fn(inputs, targets)
    test_err += err
    test_acc += acc
    test_batches += 1
  print("Final results:")
  print("       test   loss:\t\t\t{:.6f}".format(test_err   /
test_batches))
  print("  test accuracy:\t\t{:.2f} %".format(
```

```
    test_acc / test_batches * 100))
  return epoch_cost_train, epoch_cost_val


# This function was copied verbatim from the Lasagne tutorial
at
#http://lasagne.readthedocs.org/en/latest/user/tuto-
rial.html
def iterate_minibatches (inputs, targets, batchsize, shuf-
fle=False):
  assert len(inputs) == len(targets)
  if shuffle:
    indices = np. arange(len(inputs))
    np. random. shuffle(indices)
  for start_idx in range (0, len(inputs) - batchsize + 1, batch-
size):
    if shuffle:
      excerpt = indices[start_idx:start_idx + batchsize]
    else:
      excerpt = slice(start_idx, start_idx + batchsize)
    yield inputs[excerpt], targets[excerpt]



epoch_cost_train, epoch_cost_val = lasagne_mlp(X_train,
y_train, X_val, y_val, X_test,
 y_test, hidden_units=800, num_epochs=500, l2_param=0,
use_dropout=True)


plt.style.use('bmh')
plt. plot(range(len(epoch_cost_train)), epoch_cost_train,
label="Training error")
```

```
plt. plot(range(len(epoch_cost_val)), epoch_cost_val, label="Validation error")

plt. xlabel ("Num epochs")
plt. ylabel("Cost")
```

4.Multilayer Perceptron with Tensor Flow

```
###############################################################################
# Training
# Hyper-parameters. These were set by cross-validation,
# using a GridSearchCV.Not performing cross-validation to save time.
# More components tend to give better prediction performance, but larger fitting time
# Training RBM-Logistic Pipeline
# Training Logistic regression

import tensorflow as tf
import numpy as np

# This function was copied verbatim from the Tensor Flow tutorial at
# https://www.tensorflow.org/versions/master/tutorials/index.html
def dense_to_one_hot (labels_dense, num_classes=10):
    """Convert class labels from scalars to one-hot vectors."""
    num_labels = labels_dense.shape[0]
    index_offset = np. arange(num_labels) * num_classes
    labels_one_hot = np. zeros ((num_labels, num_classes))
```

```
    labels_one_hot. flat [index_offset + labels_dense. ravel
()] = 1
    return labels_one_hot


# Adapted from the Tensor Flow tutorial at
#  https://www.tensorflow.org/versions/master/tutorials/in-
dex.html
class DataSet(object):
    def __init__ (self, images, labels):
        assert images. shape [0] == labels. shape [0], (
            "images. shape: %s labels. shape: %s" % (images.
shape,
                                                      labels.
shape))
        self. _num_examples = images. shape [0]
        self. _images = images
        self. _labels = labels
        self. _epochs_completed = 0
        self. _index_in_epoch = 0

    @property
    def images(self):
        return self. _images

    @property
    def labels(self):
        return self. _labels

    @property
    def num_examples(self):
```

```
        return self._num_examples

    @property
    def epochs_completed(self):
        return self. _epochs_completed

    def next batch (self, batch_size):
        """Return the next `batch_size` examples from this
data set."""
        start = self. _index_in_epoch
        self. _index_in_epoch += batch_size
        if self. _index_in_epoch > self. _num_examples:
            # Finished epoch
            self. _epochs_completed += 1
            # Shuffle the data
            perm = np. arange (self. _num_examples)
            np. random. shuffle(perm)
            self. _images = self. _images[perm]
            self. _labels = self. _labels[perm]
            # Start next epoch
            start = 0
            self._index_in_epoch = batch_size
            assert batch_size <= self._num_examples
        end = self._index_in_epoch
        return      self._images[start:end],      self._la-
bels[start:end]


def   read_data_sets(train_images,   train_labels,   valida-
tion_images, validation_labels, test_images, test_labels):
```

```
    class DataSets(object):
        pass

    data_sets = DataSets ()
    data_sets.    train    =    DataSet    (train_images,
dense_to_one_hot(train_labels))
    data_sets.  validation  =  DataSet  (validation_images,
dense_to_one_hot(validation_labels))
    data_sets.test       =       DataSet       (test_images,
dense_to_one_hot(test_labels))
    return data_sets

# Adapted from the Tensor Flow tutorial at
#https://www.tensorflow.org/versions/master/tutorials/in-
dex.html
def tensorFlowBasic (X_train, y_train, X_val, y_val, X_test,
y_test):
    sess = tf. InteractiveSession ()
    x = tf. placeholder ("float", shape= [None, 400])
    y_ = tf. placeholder ("float", shape= [None, 10])
    W = tf. Variable (tf. zeros ([400, 10]))
    b = tf. Variable (tf. zeros ([10]))
    sess.run (tf. initialize_all_variables())
    y = tf.nn. softmax (tf. matmul (x, W) + b)
    cross_entropy = -tf. reduce_sum(y_ * tf.log(y))
    train_step = tf. train. GradientDescentOptimizer (0.01).
minimize(cross_entropy)
    mydata = read_data_sets (X_train, y_train, X_val, y_val,
X_test, y_test)

    for i in range (1000):
```

```
        batch = mydata. train. next_batch (50)
        train_step.run (feed_dict= {x: batch [0], y_: batch
[1]})

    correct_prediction = tf. equal (tf. argmax (y, 1), tf.
argmax (y_, 1))
    accuracy = tf. reduce_mean (tf. cast (correct_predic-
tion, "float"))
    return accuracy. eval (feed_dict= {x: mydata.test. im-
ages, y_: mydata.test. labels})

def weight_variable(shape):
    initial = tf. truncated_normal (shape, stddev=0.1)
    return tf. Variable(initial)

def bias_variable(shape):
    initial = tf. constant (0.1, shape=shape)
    return tf. Variable(initial)
def conv2d(x, W):
    return tf.nn. conv2d (x, W, strides= [1, 1, 1, 1], pad-
ding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool (x, ksize=[1, 2, 2, 1],
                          strides=  [1,  2,  2,  1],  pad-
ding='SAME')
def tensorFlowCNN (X_train, y_train, X_val, y_val, X_test,
y_test, add_second_conv_layer=True):
    x = tf. placeholder ("float", shape= [None, 400])
    y_ = tf. placeholder ("float", shape= [None, 10])
    sess = tf. InteractiveSession ()
```

```
# First Convolutional Layer
W_conv1 = weight_variable ([5, 5, 1, 32])
b_conv1 = bias_variable ([32])
x_image = tf. reshape (x, [-1, 20, 20, 1])
h_conv1 = tf.nn. relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
if add_second_conv_layer:
    # Second Convolutional Layer
    W_conv2 = weight_variable ([5, 5, 32, 64])
    b_conv2 = bias_variable ([64])
    h_conv2 = tf.nn. relu (conv2d (h_pool1, W_conv2) + b_conv2)
    h_pool2 = max_pool_2x2(h_conv2)

    # Densely Connected Layer
    W_fc1 = weight_variable ([5 * 5 * 64, 1024])
    b_fc1 = bias_variable ([1024])
    h_pool2_flat = tf. reshape (h_pool2, [-1, 5 * 5 * 64])
    h_fc1 = tf.nn. relu (tf. matmul (h_pool2_flat, W_fc1) + b_fc1)
else:
    # Densely Connected Layer
    W_fc1 = weight_variable ([10 * 10 * 32, 1024])
    b_fc1 = bias_variable ([1024])
    h_pool1_flat = tf. reshape (h_pool1, [-1, 10 * 10 * 32])
    h_fc1 = tf.nn. relu (tf. matmul (h_pool1_flat, W_fc1) + b_fc1)
```

```
        # Dropout
    keep_prob = tf. placeholder("float")
    h_fc1_drop = tf.nn. dropout (h_fc1, keep_prob)
    # Softmax
    W_fc2 = weight_variable ([1024, 10])
    b_fc2 = bias_variable ([10])
    y_conv = tf.nn. softmax (tf. matmul (h_fc1_drop, W_fc2)
+ b_fc2)
# Train the model
    mydata = read_data_sets (X_train, y_train, X_val, y_val,
X_test, y_test)
    cross_entropy = -tf. reduce_sum (y_ * tf.log(y_conv))
    train_step = tf. train. AdamOptimizer(1e-4). mini-
mize(cross_entropy)
    correct_prediction = tf. equal (tf. argmax (y_conv, 1),
tf. argmax (y_, 1))
    accuracy = tf. reduce_mean (tf. cast (correct_predic-
tion, "float"))
    sess.run (tf. initialize_all_variables ())
    for i in range (1000):
        batch = mydata. train. next_batch (50)
        if i % 100 == 0:
            train accuracy = accuracy. eval (feed_dict= {
                x: batch [0], y_: batch [1], keep_prob:
1.0})
            print("step %d, training accuracy %g" % (i,
train_accuracy))
        train_step.run (feed_dict= {x: batch [0], y_: batch
[1], keep_prob: 0.5})


    return accuracy. eval (feed_dict= {
```

```
        x: mydata.test. images, y_: mydata.test. labels,
keep_prob: 1.0})
accuracy = tensorFlowCNN (X_train, y_train, X_val, y_val,
X_test, y_test)
```

5. Multinomial Naive Bayes

```
from sklearn. naive_bayes import MultinomialNB
import time
conf_mat = numpy. zeros((len(no_imgs), len(no_imgs))) # In-
itializing the Confusion Matrix
n_neighbors = 1 # better to have this at the start of the
code
# 10-fold Cross Validation
for i in range(kfold):
    train_indices = skfind[i][0]
    test_indices = skfind[i][1]
    clf = []
    clf = MultinomialNB ()
    X_train = X[train_indices]
    y_train = y[train_indices]
    X_test = X[test_indices]
    y_test = y[test_indices]


    # Training
    tic = time. time ()
    clf.fit (X_train, y_train)
    toc = time. time ()
    print "training time= ", toc - tic # roughly 2.5 secs
```

```
    # Testing
y_predict = []
tic = time. time ()
y_predict = clf. predict(X_test) # output is labels and not indices
toc = time. time ()
print "testing time = ", toc - tic # roughly 0.3 secs
```

6. DBScan Clustering Algorithm

```
import time
from sklearn. cluster import DBSCAN
from sklearn import metrics
import numpy as np
from sklearn. preprocessing import StandardScaler
conf_mat = numpy. zeros((len(no_imgs), len(no_imgs))) # Initializing the Confusion Matrix
# Compute DBSCAN
db = DBSCAN (eps=0.3, min_samples=10). fit(X)
core_samples_mask = np. zeros_like (db. labels_, dtype=bool)
core_samples_mask [db. core_sample_indices_] = True
labels = db. labels_
# Number of clusters in labels, ignoring noise if present.
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)

print ('Estimated number of clusters: %d' % n_clusters_)
print ("Homogeneity: %0.3f" % metrics.homogeneity_score(y, labels))
print ("Completeness: %0.3f" % metrics.completeness_score(y, labels))
print ("V-measure: %0.3f" % metrics.v_measure_score(y, labels))
```

```
print ("Adjusted Rand Index: %0.3f"
   %metrics. adjusted_rand_score (y, labels))
print ("Adjusted Mutual Information: %0.3f"
   %metrics. adjusted_mutual_info_score (y, labels))
print ("Silhouette Coefficient: %0.3f"
   %metrics. silhouette_score (X, labels))
```

7.Random Forest Classifier

```
from sklearn. ensemble import RandomForestClassifier
import time
conf_mat = numpy. zeros((len(no_imgs), len(no_imgs)))  # Initial-
izing the Confusion Matrix
n_neighbors = 1 # better to have this at the start of the code
# 10-fold Cross Validation
for i in range(kfold):
    train_indices = skfind[i][0]
    test_indices = skfind[i][1]
    clf = []
    clf = RandomForestClassifier(n_estimators=10)
    X_train = X[train_indices]
    y_train = y[train_indices]
    X_test = X[test_indices]
    y_test = y[test_indices]
         # Training
    tic = time. time ()
    clf.fit (X_train, y_train)
    toc = time. time ()
    print "training time= ", toc - tic # roughly 2.5 secs


# Testing
```

```
y_predict = []
tic = time. time ()
y_predict = clf. predict(X_test) # output is labels and not
indices
toc = time. time ()
print "testing time = ", toc - tic # roughly 0.3 secs
```

8.MeanShift Clustering

```
import numpy as np
from sklearn. cluster import MeanShift, estimate_bandwidth
# Compute clustering with MeanShift
# The following bandwidth can be automatically detected us-
ing
bandwidth = estimate_bandwidth (X, quantile=0.3)
ms = MeanShift (bandwidth= bandwidth)
ms.fit(X)
labels = ms.labelsy
cluster_centers = ms. cluster_centersy

labels_unique = np. unique(labels)
n_clusters_ = len(labels_unique)

print ("number of estimated clusters : %d" % n_clusters_)
```

9.Kmeans-MiniBatch

```
from sklearn. cluster import KMeans
import numpy as np
import matplotlib. pyplot as plt
from sklearn. cluster import MiniBatchKMeans, KMeans
```

```
from   sklearn.   metrics.pairwise   import   pairwise_dis-
tances_argmin

np. random. seed (0)

batch_size = 45

centers = [[1, 1], [-1, -1], [1, -1]]

n_clusters = len(centers)

k_means = KMeans (init='k-means++', n_clusters=3, n_init=10)

t0 = time. time ()

k_means.fit(X)

t_batch = time. time () - t0

k_means_labels = k_means. labels_

k_means_cluster_centers = k_means. cluster_centers_

k_means_labels_unique = np. unique(k_means_labels)


##########################################################
##################

# Compute clustering with MiniBatchKMeans


mbk   =   MiniBatchKMeans   (init='k-means++',   n_clusters=3,
batch_size=batch_size,

                      n_init=10,     max_no_improvement=10,
verbose=0)

t0 = time. time ()

mbk.fit(X)

t_mini_batch = time. time () - t0

mbk_means_labels = mbk. labels_

mbk_means_cluster_centers = mbk. cluster_centers_

mbk_means_labels_unique = np. unique(mbk_means_labels)

##########################################################
##################
```

```
# Plot result
fig = plt. figure (figsize= (8, 3))
fig. subplots adjust (left=0.02, right=0.98, bottom=0.05,
top=0.9)
colors = ['#4EACC5', '#FF9C34', '#4E9A06']
# same colors for the same cluster from the
# MiniBatchKMeans and the KMeans algorithm. Let's pair the
cluster centers per closest one.
order = pairwise_distances_argmin (k_means_cluster_centers,
                                  mbk_means_cluster_cen-
ters)
# KMeans
ax = fig.add_subplot (1, 3, 1)
for k, col in zip(range(n_clusters), colors):
    my_members = k_means_labels == k
    cluster_center = k_means_cluster_centers[k]
    ax. plot (X [my_members, 0], X [my_members, 1], 'w',
            markerfacecolor=col, marker='.')
    ax. plot (cluster_center [0], cluster_center [1], 'o',
markerfacecolor=col,
            markeredgecolor='k', markersize=6)
ax.set_title('KMeans')
ax.set_xticks (())
ax.set_yticks (())
plt.text(-3.5, 1.8,  'train time: %.2fs\ninertia: %f' % (
    t_batch, k_means. inertia_))
ax = fig.add_subplot(1, 3, 2)
for k, col in zip(range(n_clusters), colors):
    my_members = mbk_means_labels == order[k]
    cluster_center = mbk_means_cluster_centers[order[k]]
```

```
    ax. plot (X [my_members, 0], X [my_members, 1], 'w',
            markerfacecolor=col, marker='.')
    ax. plot (cluster_center [0], cluster_center [1], 'o',
markerfacecolor=col,
            markeredgecolor='k', markersize=6)
ax.set_title('MiniBatchKMeans')
ax.set_xticks (())
ax.set_yticks (())
plt.text (-3.5, 1.8, 'train time: %.2fs\ninertia: %f' %
         (t_mini_batch, mbk. inertia_))

# Initialise the different array to all False
different = (mbk_means_labels == 4)
ax = fig.add_subplot (1, 3, 3)

for l in range(n_clusters):
    different += ((k_means_labels == k) != (mbk_means_labels
== order[k]))

identic = np. logical_not(different)
ax. plot (X [identic, 0], X [identic, 1], 'w',
        markerfacecolor='#bbbbbb', marker='.')
ax. plot (X [different, 0], X [different, 1], 'w',
        markerfacecolor='m', marker='.')
ax.set_title('Difference')
ax.set_xticks (())
ax.set_yticks (())

plt. show ()
```

10. Comparison

```
from sklearn import cluster, datasets
from sklearn. neighbors import kneighbors_graph
from sklearn. preprocessing import StandardScaler
import time

colors = np. array ([x for x in 'bgrcmykbgrcmykbgrcmykbgrcmyk'])
colors = np. hstack([colors] * 20)

clustering_names = [
    'MiniBatchKMeans', 'AffinityPropagation', 'MeanShift',
    'SpectralClustering', 'Ward', 'AgglomerativeCluster-ing',
    'DBSCAN', 'Birch']

plt. figure(figsize=(len(clustering_names) * 2 + 3, 9.5))
plt. subplots_adjust (left=.02, right=.98, bottom=.001, top=.96, wspace=.05,
                    hspace=.01)

plot_num = 1


    # normalize dataset for easier parameter selection
X = StandardScaler (). fit_transform(X)

    # estimate bandwidth for mean shift
bandwidth = cluster. estimate_bandwidth (X, quantile=0.3)
```

```
    # connectivity matrix for structured Ward
connectivity = kneighbors_graph (X, n_neighbors=10, in-
clude_self=False)
    # make connectivity symmetric
connectivity = 0.5 * (connectivity + connectivity. T)

    # create clustering estimators
ms = cluster. MeanShift (bandwidth=bandwidth, bin_seed-
ing=True)
two_means = cluster. MiniBatchKMeans(n_clusters=2)
ward = cluster. AgglomerativeClustering (n_clusters=2, link-
age='ward',
                                           connectiv-
ity=connectivity)
spectral = cluster. SpectralClustering (n_clusters=2,

eigen_solver='arpack',
                                          affinity="near-
est_neighbors")
dbscan = cluster. DBSCAN(eps=.2)
affinity_propagation = cluster. AffinityPropagation (damp-
ing=.9,

preference=-200)

average_linkage = cluster. AgglomerativeClustering (
        linkage="average", affinity="cityblock", n_clus-
ters=2,
        connectivity=connectivity)

birch = cluster. Birch(n_clusters=2)
```

```
clustering_algorithms = [
        two_means, affinity_propagation, ms, spectral, ward, average_linkage,
        dbscan, birch]

for name, algorithm in zip(clustering_names, clustering_algorithms):
        # predict cluster memberships
    t0 = time. time ()
    algorithm.fit(X)
    t1 = time. time ()
    if hasattr (algorithm, 'labels_'):
        y_pred = algorithm. labels_.astype(np.int)
    else:
        y_pred = algorithm. predict(X)

        # plot
    plt. subplot (4, len(clustering_algorithms), plot_num)

    plt.  scatter  (X[:,  0],  X[:,  1],  color=colors[y_pred].tolist(), s=10)

    if hasattr(algorithm, 'cluster_centers_'):
        centers = algorithm.cluster_centers_
        center_colors = colors[:len(centers)]
        plt.scatter(centers[:, 0], centers[:, 1], s=100, c=center_colors)
    plt.xlim(-2, 2)
    plt.ylim(-2, 2)
    plt.xticks(())
```

```
    plt.yticks(())
    plt.text(.99, .01, ('%.2fs' % (t1 - t0)).lstrip('0'),
                transform=plt.gca().transAxes, size=15,
                horizontalalignment='right')
    plot_num += 1

plt.show()


    # normalize dataset for easier parameter selection
```

11.Affinity Propagation

```
from sklearn.cluster import AffinityPropagation
from sklearn import metrics

############################################################
###################
# Generate sample data
centers = [[1, 1], [-1, -1], [1, -1]]

############################################################
###################
# Compute Affinity Propagation
af = AffinityPropagation(preference=-50).fit(X)
cluster_centers_indices = af.cluster_centers_indices_
labels = af.labels_

n_clusters_ = n_clusters_ = len(set(labels)) - (1 if -1 in
labels else 0)

print('Estimated number of clusters: %d' % n_clusters_)
```

```
print("Homogeneity:  %0.3f"  %  metrics.homogeneity_score(y,
labels))

print("Completeness: %0.3f" % metrics.completeness_score(y,
labels))

print("V-measure:  %0.3f"  %  metrics.v_measure_score(y,  la-
bels))

print("Adjusted Rand Index: %0.3f"

      % metrics.adjusted_rand_score(y, labels))

print("Adjusted Mutual Information: %0.3f"

      % metrics.adjusted_mutual_info_score(y, labels))

print("Silhouette Coefficient: %0.3f"

     %  metrics.silhouette_score(X,  labels,  metric='sqeu-
clidean'))

###########################################################
###################

# Plot result

import matplotlib.pyplot as plt

from itertools import cycle


plt.close('all')

plt.figure(1)

plt.clf()

colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')


for k, col in zip(range(n_clusters_), colors):

    class_members = labels == k

    cluster_center = X[cluster_centers_indices[k]]

    plt.plot(X[class_members, 0], X[class_members, 1], col
+ '.')

    plt.plot(cluster_center[0],   cluster_center[1],   'o',
markerfacecolor=col,
```

```
                markeredgecolor='k', markersize=14)
    for x in X[class_members]:
        plt.plot([cluster_center[0],  x[0]],  [cluster_cen-
ter[1], x[1]], col)

plt.title('Estimated number of clusters: %d' % n_clusters_)
plt.show()
```

12.Perceptron

```
from sklearn.linear_model import SGDClassifier
import time

conf_mat = numpy.zeros((len(no_imgs), len(no_imgs)))  # In-
itializing the Confusion Matrix

n_neighbors = 1  # better to have this at the start of the
code

# 10-fold Cross Validation

for i in range(kfold):
    train_indices = skfind[i][0]
    test_indices = skfind[i][1]
    clf = []
    clf  =  SGDClassifier(loss='perceptron',  eta0=1,  learn-
ing_rate='constant', penalty=None)


    X_train = X[train_indices]
    y_train = y[train_indices]
    X_test = X[test_indices]
```

```
    y_test = y[test_indices]

    # Training
    tic = time.time()
    clf.fit(X_train, y_train)
    toc = time.time()
    print "training time= ", toc - tic  # roughly 2.5 secs

    # Testing
y_predict = []
tic = time.time()
y_predict = clf.predict(X_test)  # output is labels and not
indices
toc = time.time()
print "testing time = ", toc - tic  # roughly 0.3 secs
```

13. Support Vector Machines

```
from sklearn import svm
import numpy
import time
conf_mat = numpy.zeros((len(no_imgs), len(no_imgs)))  # In-
itializing the Confusion Matrix
```

```
n_neighbors = 1  # better to have this at the start of the
code

# 10-fold Cross Validation

for i in range(kfold):
    train_indices = skfind[i][0]
    test_indices = skfind[i][1]
    clf = []
    clf = svm.SVC()
    X_train = X[train_indices]
    y_train = y[train_indices]
    X_test = X[test_indices]
    y_test = y[test_indices]

    # Training
    tic = time.time()
    clf.fit(X_train, y_train)
    toc = time.time()
    print "training time= ", toc - tic  # roughly 2.5 secs


# Testing
y_predict = []
tic = time.time()
y_predict = clf.predict(X_test)  # output is labels and not
indices
toc = time.time()
print "testing time = ", toc - tic  # roughly 0.3 secs
```

14.Stochastic Gradient

```
from sklearn.linear_model import SGDClassifier
import time
conf_mat = numpy.zeros((len(no_imgs), len(no_imgs)))  # In-
itializing the Confusion Matrix

n_neighbors = 1  # better to have this at the start of the
code

# 10-fold Cross Validation

for i in range(kfold):
    train_indices = skfind[i][0]
    test_indices = skfind[i][1]
    clf = []
    clf = SGDClassifier(loss="hinge", penalty="l2")
          X_train = X[train_indices]
    y_train = y[train_indices]
    X_test = X[test_indices]
    y_test = y[test_indices]


    # Training
    tic = time.time()
    clf.fit(X_train, y_train)
    toc = time.time()
    print "training time= ", toc - tic  # roughly 2.5 secs
```

```
    # Testing
y_predict = []
tic = time.time()
y_predict = clf.predict(X_test)  # output is labels and not
indices
toc = time.time()
print "testing time = ", toc - tic  # roughly 0.3 secs
```

15.Decision Tree Algorithm

```
from sklearn import tree
import time
conf_mat = numpy.zeros((len(no_imgs), len(no_imgs)))  # In-
itializing the Confusion Matrix
n_neighbors = 1  # better to have this at the start of the
code
# 10-fold Cross Validation
for i in range(kfold):
    train_indices = skfind[i][0]
    test_indices = skfind[i][1]
    clf = []
    clf = tree.DecisionTreeClassifier()


 X_train = X[train_indices]
 y_train = y[train_indices]
    X_test = X[test_indices]
    y_test = y[test_indices]

# Training
```

```
tic = time.time()
clf.fit(X_train, y_train)
toc = time.time()
print "training time= ", toc - tic  # roughly 2.5 secs
# Testing
y_predict = []
tic = time.time()
y_predict = clf.predict(X_test)  # output is labels and not
indices
toc = time.time()
print "testing time = ", toc - tic  # roughly 0.3 secs
```

16.Nearest Centroid Algorithm

```
from sklearn.neighbors.nearest_centroid import NearestCen-
troid
import time
conf_mat = numpy.zeros((len(no_imgs), len(no_imgs)))  # In-
itializing the Confusion Matrix
n_neighbors = 1  # better to have this at the start of the
code
# 10-fold Cross Validation
for i in range(kfold):
    train_indices = skfind[i][0]
    test_indices = skfind[i][1]
    clf = []
    clf = NearestCentroid()


    X_train = X[train_indices]
    y_train = y[train_indices]
```

```
    X_test = X[test_indices]
    y_test = y[test_indices]

    # Training
    tic = time.time()
    clf.fit(X_train, y_train)
    toc = time.time()
    print "training time= ", toc - tic  # roughly 2.5 secs

# Testing
y_predict = []
tic = time.time()
y_predict = clf.predict(X_test)  # output is labels and not indices
toc = time.time()
print "testing time = ", toc - tic  # roughly 0.3 secs
```

17. RBM Bernoulli Algorithm

```
from sklearn.neural_network import BernoulliRBM
from sklearn.pipeline import Pipeline
from sklearn import linear_model
from sklearn.pipeline import Pipeline
import time
conf_mat = numpy.zeros((len(no_imgs), len(no_imgs)))  # Initializing the Confusion Matrix

n_neighbors = 1 # better to have this at the start of the code
# 10-fold Cross Validation

for i in range(kfold):
```

```
    train_indices = skfind[i][0]
    test_indices = skfind[i][1]
    clf = []

    logistic = linear_model.LogisticRegression()
    rbm = BernoulliRBM(random_state=0, verbose=True)

clf = Pipeline(steps=[('rbm', rbm), ('logistic', logistic)])

############################################################
####################
# Training
# Hyper-parameters. These were set by cross-validation,
# using a GridSearchCV. not performing cross-validation to
save time.
rbm.learning_rate = 0.06
rbm.n_iter = 20
# More components tend to give better prediction performance,
but larger fitting time
rbm.n_components = 100


# Training RBM-Logistic Pipeline
# Training Logistic regression
X_train = X[train_indices]
y_train = y[train_indices]
X_test = X[test_indices]
y_test = y[test_indices]

# Training
```

```
tic = time.time()
clf.fit(X_train, y_train)
logistic_classifier = linear_model.LogisticRegres-
sion(C=100.0)
logistic_classifier.fit(X_train, y_train)
toc = time.time()
print "training time= ", toc - tic  # roughly 2.5 secs
# Testing
y_predict = []
y_predict1=[]
tic = time.time()
y_predict = clf.predict(X_test)
y_predict=logistic_classifier.predict(X_test)  # output is
labels and not indices
toc = time.time()
print "testing time = ", toc - tic  # roughly 0.3 secs
```

18. General print as computed matrix for results

```
# Compute confusion matrix
from sklearn.metrics import confusion_matrix
cm = []
cm = confusion_matrix(y_test, y_predict)
conf_mat = conf_mat + cm

conf_mat = conf_mat.T  # since rows and  cols are inter-
changed
avg_acc = numpy.trace(conf_mat) / sum(no_imgs)
conf_mat_norm = conf_mat / no_imgs  # Normalizing the con-
fusion matrix
```

```
import matplotlib.pyplot as plt
plt.imshow(conf_mat_norm, interpolation='nearest')
plt.title('Confusion matrix')
plt.colorbar()
plt.show()
plt.savefig('confusion_matrix.png')

conf_mat2 = numpy.around(conf_mat_norm,decimals=2) # round-
ing to display in figure
plt.imshow(conf_mat2,interpolation='nearest')
for x in xrange(len(list_fams)):
  for y in xrange(len(list_fams)):
    plt.annotate(str(conf_mat2[x][y]),xy=(y,x),ha='cen-
ter',va='center')
plt.xticks(range(len(list_fams)),list_fams,rota-
tion=90,fontsize=11)
plt.yticks(range(len(list_fams)),list_fams,fontsize=11)
plt.title('RandomForestClassifier')
plt.colorbar()
plt.show()
plt.savefig('confusion_matrix.png')
```