VISVESVARAYA TECHNOLOGICAL UNIVERSITY

Jnana Sangama, Machhe, Belagavi-590018

A

Project Phase - 2 Report on

"Detection of Malware Using Machine Learning Algorithms"

Submitted in partial fulfillment required for
award of the Degree

Bachelor of Engineering

In

Computer Science and Engineering

Submitted by

SATYAM SINGH	**1HK19CS135**
YASH ANJANA	**1HK19CS183**
YUSUF AHMAD	**1HK19CS185**
SHUBHANSHU KUMAR	**1HK20CS409**

Under the guidance of

Prof. Tahir Naquash

Assistant Professor

Department of Computer Science & Engineering

2022-2023

Department of Computer Science & Engineering

HKBK COLLEGE *of* ENGINEERING

(Approved by AICTE & Affiliated to VTU)

22/1, Nagawara, Arabic College Post, Bangalore-45, Karnataka

Email: info@hkbk.edu.in URL: www.hkbk.edu.in

HKBK COLLEGE *of* ENGINEERING

Nagawara, Bangalore–560 045
Approved by AICTE & Affiliated to VTU

Department of Computer Science and Engineering

Certificate

Certified that the Project Work Phase-2 entitled "**Detection of Malware Using Machine Learning Algorithms**" carried out by Mr. **Satyam Singh (1HK19CS135), Yash Anjana (1HK19CS183), Yusuf Ahmad (1HK19CS185) and Shubhanshu Kumar (1HK20CS409)** are Bonafide student of HKBK College of Engineering in partial fulfilment of final year project, regards to the subject **"Project Work Phase – 2 (18CSP83)"** for the award of Bachelor of Engineering / Bachelor of Technology in **Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year **2022 – 23**. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the Report deposited in the departmental library.

The project report has been approved as it satisfies the academic requirements in respect of Project work prescribed for the said Bachelor of Engineering.

Prof Tahir Naquash	**Dr. Ashok Kumar**	**Dr. Tabassum Ara**
Guide	HOD	Principal

External Viva

Name of the Examiners	Signature with Date
1.	
2.	

ACKNOWLEDGEMENT

We would like to profoundly thank Chairman **Mr. C. M. Ibrahim**, Director **Mr. C. M. Faiz** and Management of HKBK College of Engineering for providing such a healthy environment for the successful completion of project work.

We would like to express our thanks to the Principal **Dr. Tabassum Ara**, for her encouragement that motivated us for the successful completion of Project Work. We wish to express our gratitude to **Dr. Ashok Kumar P. S**., Professor and Head of Department of Computer Science & Engineering for providing such a healthy environment for the successful completion of Project work.

We are also grateful to **Dr. Deepak N. R**., Professor, Dept of CSE and **Prof. Simran Pal R** Asst. Prof., Dept of CSE, HKBKCE. We are extremely thankful and indebted to her for sharing expertise, and sincere and valuable guidance and encouragement extended to us.

We express our heartfelt appreciation and gratitude to my Project Guide, **Prof. Tahir Naquash**, Professor of Computer Science and Engineering, HKBK College of Engineering, Bangalore, for his intellectually- motivating support and invaluable encouragement during my Project work. His comprehensive knowledge and understanding of the research topic as well as his uncompromising and sensible attitude towards research and insistence on quality work have profoundly influenced us and will definitely benefit our future work-my heartfelt thanks to his painstaking modification of this report.

We would also like to thank all other teaching and technical staffs of **Department of Computer Science and Engineering**, who have directly or indirectly helped us in the completion of this Project Work. And lastly, we would hereby acknowledge and thank our parents who have been a source of inspiration and also instrumental in the successful completion of this project.

1HK19CS135	**SATYAM SINGH**
1HK19CS183	**YASH ANJANA**
1HK19CS185	**YUSUF AHMAD**
1HK20CS409	**SHUBHANSHU KUMAR**

ABSTRACT

Current antivirus software's are effective against known viruses, if a malware with new signature is Introduced then it will be difficult to detect that it is malicious. Signature-based detection is not that effective during zero-day attacks. Till the signature is created for new (unseen) malware, distributed to the systems and added to the anti-malware database, the systems can be exploited by that malware. Research shows that over the last decade, malware has been growing exponentially, causing substantial financial losses to various organizations. Different anti-malware companies have been proposing solutions to defend attacks from these malwares. The velocity, volume, and the complexity of malware are posing new challenges to the anti-malware community. Current state-of-the-art research shows that recently, researchers and anti-virus organizations started applying machine learning and deep learning methods for malware analysis and detection. Machine learning methods can be used to create more effective anti-malware software which is capable of detecting previously unknown malware, zero-day attack etc. We propose an approach that Various machine learning methods such as Support Vector Machine (SVM), Decision tree, Random Forest and XG Boost will be used.

TABLE OF CONTENTS

ACKNOWLEDGEMENT		I
ABSTRACT		II
TABLE OF CONTENT		III
LIST OF FIGURES		IV
LIST OF TABLES		V
ABBREVIATION		1
CHAPTER 1 INTRODUCTION		2
1.1	EVOLUTION OF MALWARE	4
1.2	MALWARE DETECTION	5
1.3	NEED FOR MACHINE LEARNING	7
1.4	SUMMARY	8
CHAPTER 2 LITERATURE SURVEY		9
2.1	LITERATURE PAPER SURVEYED	10
2.2	SUMMARY	16
CHAPTER 3 METHODOLOGY		17
3.1	ALGORITHMS USED	18
3.1.1	RANDOM FOREST	19
3.1.2	DECISION TREE	24
3.1.3	GRADIENT BOOSTING	29
3.1.4	AdaBoost	34
3.1.5	GNB	
CHAPTER 4 DESIGN AND DEVELOPMENT		
4.1	DATASET	
4.2	PRE-PROCESSING	
4.3	FEATURES EXTRACTION	
4.4	FEATURES SELECTION	
CHAPTER 5 CODE SNIPPETS		
CHAPTER 6 CONCLUSION		
CHAPTER 7 REFERENCES		

LIST OF TABLES

Figure3: Methodology……………………………………………………………18

Figure3.1: Ensemble…………………………………………………………...20

Figure3.1.1: Ensemble Classifier………………………………………………21

Figure3.1.2: Bagging Ensemble Method ……………………………………....22

Figure: 3.1.3 Random Forest View..........…...………………………………..23

Figure: 3.2.1 Sample of patient data..........…...……………………………....24

Figure: 3.2.2 predict data..........…...…………………………………………25

Figure: 3.2.3 node data………………………………………………………....26

Figure: 3.3 Gradient Boost ……………………………………………………30

Figure: 3.4 Decision Stumps...…………………………………………………34

Figure: 3.4.1 Linear Regression Model………………………………………..35

Figure:4.1 Proposed ML malware detection method…………………………48

Figure 4.2 Workflow process illustration……………………………………...49

Figure 5.1: Home Page………………………………………………………..66

Figure 5.2 Program Detail……………………………………………………..66

Figure 5.3 Our Team…………………………………………………………..67

Figure 5.4 Malware Found…………………………………………………….67

Figure 5.5 Malware Not Found………………………………………………..68

Figure 5.6 Graph……………………………………………………………….68

Figure 5.7 Histogram…………………………………………………………..69

Figure 5.8 Features identified as important……………………………………69

Figure 5.9 Test Results………………………………………………………...70

LIST OF TABLES

TABLE 1 Dataset file types…………………………………………..………..11

TABLE 2 Classifiers results comparisons …………………………………….14

TABLE 3 Sample data……………………………………..……….…….24

TABLE 4 Dataset ………………………………………..…………..……36

TABLE 5 Dataset ………………………………………..…………..……39

TABLE 6 Dataset ………………………………………..…………..……39

TABLE 7 Dataset (sample weights)……………………………………...40

TABLE 8 Dataset (random numbers)…………………………………….40

ABBREVARTION

ABBREVIATION	DESCRIPTION
CNN	Convolutional Neural Network
AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
GBM	Gradient Boosting

CHAPTER 1
INTRODUCTION

CHAPTER 1

INTRODUCTION

Idealistic hackers attacked computers in the early days because they were eager to prove themselves. Cracking machines, however, is an industry in today's world. Despite recent improvements in software and computer hardware security, both in frequency and sophistication, attacks on computer systems have increased. Regrettably, there are major drawbacks to current methods for detecting and analysing unknown code samples. The Internet is a critical part of our everyday lives today. On the internet, there are many services and they are rising daily as well. Numerous reports indicate that malware's effect is worsening at an alarming pace. Although malware diversity is growing, anti- virus scanners are unable to fulfil security needs, resulting in attacks on millions of hosts. Around 65,63,145 different hosts were targeted, according to Kaspersky Labs, and in 2015, 40,00,000 unique malware artefacts were found. Juniper Research (2016), in particular, projected that by 2019 the cost of data breaches will rise to $2.1 trillion globally. Current studies show that script-kiddies are generating more and more attacks or are automated. To date, attacks on commercial and government organizations, such as ransomware and malware, continue to pose a significant threat and challenge. Such attacks can come in various ways and sizes. An enormous challenge is the ability of the global security community to develop and provide expertise in cybersecurity. There is widespread awareness of the global scarcity of cybersecurity and talent. Cybercrimes, such as financial fraud, child exploitation online and payment fraud, are so common that they demand international 24-hour response and collaboration between multinational law enforcement agencies. For single users and organizations, malware defense of computer systems is therefore one of the most critical cybersecurity activities, as even a single attack may result in compromised data and sufficient losses.

Mobile phones have become increasingly important tools in people’s daily life, such as mobile payment, instant messaging, online shopping, etc., but the security problem of mobile phones is becoming more and more serious. Due to the open-source nature of the Android platform, it is very easy and profitable to write malware using the vulnerabilities and security defects of the Android system. This is the main reason for the rapid increase in the number of malware on the Android system. The malicious behaviors of Android malware generally include sending deduction SMS, consuming traffic, stealing user's private information, downloading a large number of malicious applications, remote control, etc., threatening the privacy and property security of mobile phones users.

The number of Android malware is growing rapidly; particularly, more and more malicious software use obfuscation technology. Traditional detection methods of manual analysis and signature matching have exposed some problems, such as slow detection speed and low accuracy. In recent years, many researchers have solved the problems of Android malware detection using machine learning algorithms and had a lot of research results. With the rise of deep learning and the improvement of computer computing power, more and more researchers began to use deep learning models to detect Android malware. This paper proposes an Android malware detection model based on a hybrid deep learning model with deep belief network (DBN) and gate recurrent unit (GRU). The main contributions are as follows:

(i) In order to resist Android malware obfuscation technology, in addition to extracting static features, we also extracted the dynamic features of malware at runtime and constructed a comprehensive feature set to enhance the detection capability of malware.

(ii) A hybrid deep learning model was proposed. According to the characteristics of static features and dynamic features, two different deep learning algorithms of DBN and GRU are used.

(iii) The detection model was verified, and the detection result is better than traditional machine learning algorithms; it also can effectively detect malware samples using obfuscation technology.

1.1 EVOLUTION OF MALWARE

In order to protect networks and computer systems from attacks, the diversity, sophistication and availability of malicious software present enormous challenges. Malware is continually changing and challenges security researchers and scientists to strengthen their cyber defences to keep pace. Owing to the use of polymorphic and metamorphic methods used to avoid detection and conceal its true intent, the prevalence of malware has increased. To mutate the code while keeping the original functionality intact, polymorphic malware uses a polymorphic engine. The two most common ways to conceal code are packaging and encryption. Through one or more layers of compression, packers cover a program's real code. Then the unpacking routines restore the original code and execute it in memory at runtime. To make it harder for researchers to analyze the software, crypters encrypt and manipulate malware or part of its code. A crypter includes a stub that is used for malicious code encryption and decryption. Whenever it's propagated, metamorphic malware rewrites the code to an equivalent. Multiple transformation techniques, including but not limited to, register renaming, code permutation, code expansion, code shrinking and insertion of garbage code, can be used by malware authors.

The combination of the above techniques resulted in increasingly increasing quantities of malware, making time-consuming, expensive and more complicated forensic investigations of malware cases. There are some issues with conventional antivirus solutions that rely on signature-based and heuristic/behavioural methods. A signature is a unique feature or collection of features that like a fingerprint, uniquely differentiates an executable. Signature-based approaches are unable to identify unknown types of malware, however. Security researchers suggested behaviour-based detection to overcome these problems, which analyses the features and behaviour of the file to decide whether it is indeed malware, although it may take some time to search and evaluate. Researchers have begun implementing machine learning to supplement their solutions in order to solve the previous drawbacks of conventional antivirus engines and keep pace with new attacks and variants, as machine learning is well suited for processing large quantities of data.

1.2 MALWARE DETECTION

In such a way, hackers present malware aimed at persuading people to install it. As it seems legal, users also do not know what the programme is. Usually, we install it thinking that it is secure, but on the contrary, it's a major threat. That's how the malware gets into your system. When on the screen, it disperses and hides in numerous files, making it very difficult to identify. In order to access and record personal or useful information, it may connect directly to the operating system and start encrypting it. Detection of malware is defined as the search process for malware files and directories. There are several tools and methods available to detect malware that make it efficient and reliable. Some of the general strategies for malware detection are:

- Signature-based
- Heuristic Analysis
- Anti-malware Software
- Sandbox

Several classifiers have been implemented, such as linear classifiers (logistic regression, naïve Bayes classifier), support for vector machinery, neural networks, random forests, etc.

Through both static and dynamic analysis, malware can be identified by:

- Without Executing the code
- Behavioral Analysis

1.3 NEED FOR MACHINE LEARNING IN MALWARE DETECTION

Machine learning has created a drastic change in many industries, including cybersecurity, over the last decade. Among cybersecurity experts, there is a general belief that AI-powered anti-malware tools can help detect modern malware attacks and boost scanning engines. Proof of this belief is the number of studies on malware detection strategies that exploit machine learning reported in the last few years. The number of research papers released in 2018 is 7720, a 95 percent rise over 2015 and a 476 percent increase over 2010, according to Google Scholar,1. This rise in the number of studies is the product of several factors, including but not limited to the increase in publicly labelled malware feeds, the increase in computing capacity at the same time as its price decrease, and the evolution of the field of machine learning, which has achieved ground-breaking success in a wide range of tasks such as computer vision and speech recognition. Depending on the type of analysis, conventional machine learning methods can be categorized into two main categories, static and dynamic approaches. The primary difference between them is that static methods extract features from the static malware analysis, while dynamic methods extract features from the dynamic analysis. A third category may be considered, known as hybrid approaches. Hybrid methods incorporate elements of both static and dynamic analysis. In addition, learning features from raw inputs in diverse fields have outshone neural networks. The performance of neural networks in the malware domain is mirrored by recent developments in machine learning for cybersecurity.

1.4 SUMMARY

This section describes what is actually considered as malware, the need for malware detection and its evolution. The use of Machine Learning Algorithms to easily categorize malware using the static methods, dynamic methods, hybrid methods. Stages for malware detection: Signature based. The need to implement machine learning algorithms.

CHAPTER 2
LITERATURE SURVEY

CHAPTER 2

LITERATURE SURVEY

2.1 Literature Papers Surveyed

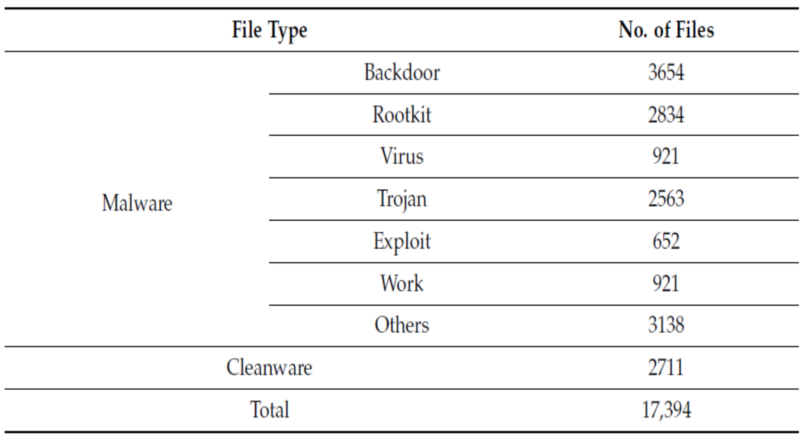
The world is vulnerable to cyberattacks because there are so many computers, smartphones, and other devices that can connect to the Internet. The surge in malware activity has led to the emergence of a plethora of malware detection techniques. When attempting to determine researchers employ a number of big data technologies and machine learning techniques to detect malicious code.

Traditional machine learning-based malware detection techniques take a long time to process, but they may successfully spot recently discovered malware. Due to the widespread use of contemporary machine learning algorithms, such as deep learning, feature engineering may become outdated. In this study, we looked at various methods for identifying and categorising malware. Researchers have developed methods to examine samples for malicious intent using machine learning and deep learning.

It is crucial to understand the significance of data security while implementing digital applications. As highlighted by Armaan (2021), models cannot function accurately without reliable data. Hence, precautions must be taken to safeguard data from potential cyber risks.

Machine learning is a cutting-edge approach that enables precise prediction by selecting relevant features. However, feature selection can be a challenging task, particularly when dealing with non-standard data. Therefore, it is essential to develop adaptable workarounds to handle such data. To effectively prevent future attacks, malware analysis must be performed to identify new rules and patterns. IT security professionals can use malware analysis tools to find patterns and create new rules to counter emerging malware types, as shown in Table 1. Overall, data security and analysis are critical components in developing effective digital applications that can accurately predict outcomes and prevent potential cyber risks.

The availability of technologies that analyze malware samples and determine their level of malignancy is a significant boon to the cybersecurity sector. These tools can help monitor security alerts and prevent malware attacks before they cause significant damage. Malware analysis tools are essential in identifying and categorizing malware, determining its capabilities, and assessing its potential impact. If malware is deemed dangerous, it must be eliminated before it can propagate further and cause additional harm. Moreover, as the number and complexity of malware threats continue to grow, businesses must find effective ways to lessen their effects. Malware analysis is becoming increasingly popular as it provides valuable insights into how malware operates, enabling IT security professionals to develop appropriate mitigation strategies. The availability of malware analysis tools is essential for businesses to safeguard their data and systems against cyber threats. These tools enable effective monitoring and prevention of malware attacks, helping businesses to reduce the risks associated with malware threats.



File Type		No. of Files
Malware	Backdoor	3654
	Rootkit	2834
	Virus	921
	Trojan	2563
	Exploit	652
	Work	921
	Others	3138
Cleanware		2711
Total		17,394

TABLE 1 Dataset file types

Chowdhury's (2018) proposed malware detection approach utilizing machine learning classification techniques, specifically incorporating N-gram and API call capabilities, has proven to be effective and dependable. However, further experiments were conducted to determine if adjusting certain parameters could increase the accuracy of malware classification. The experimental evaluation has confirmed the efficacy and dependability of the proposed technique, indicating that it outperformed other competing approaches. The results of this evaluation are presented in Table 2, where the Chowdhury [23] approach was clearly superior. Future work in this area will focus on merging a large number of features to increase detection precision while decreasing false positives. This would further enhance the accuracy and effectiveness of the malware detection approach proposed by Chowdhury. The proposed approach utilizing machine learning classification techniques, N-gram, and API call capabilities, provides a reliable and effective means of detecting malware. The results of the experiments confirm the validity of this approach, and future work will continue to build upon its strengths to further improve malware detection accuracy.

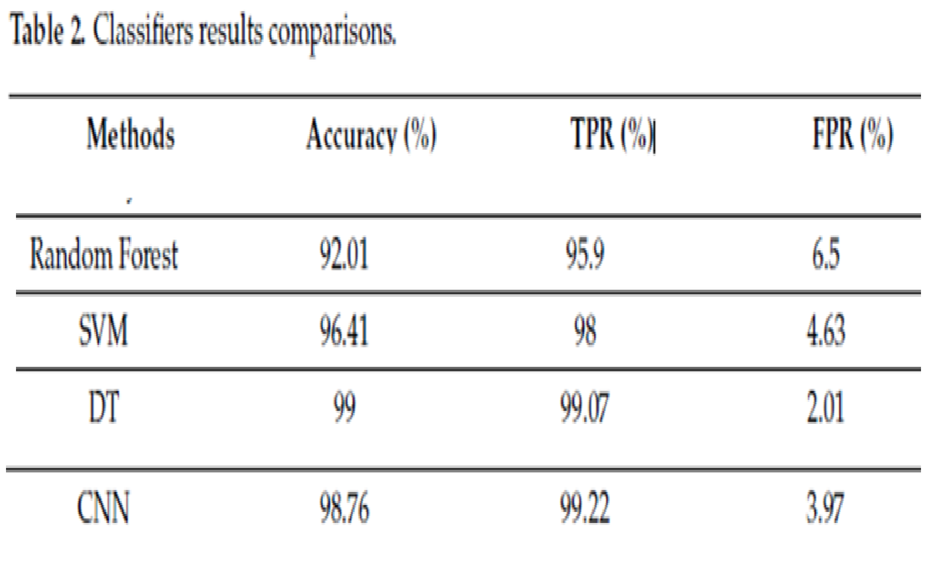


Table 2. Classifiers results comparisons.

Methods	Accuracy (%)	TPR (%)	FPR (%)
Random Forest	92.01	95.9	6.5
SVM	96.41	98	4.63
DT	99	99.07	2.01
CNN	98.76	99.22	3.97

TABLE 2 Classifiers results comparisons

The proliferation of malicious software poses a significant threat to global stability, with the prevalence of malware increasing as the number of interconnected computers exploded in the 1990s. In response to this phenomenon, multiple protective measures have been created. However, the current safeguards are not sufficient to keep up with modern threats created by malware authors to thwart security programs. In recent years, researchers have focused on using machine learning (ML) algorithm strategies for malware detection. In this research paper, a protective mechanism is presented that evaluates four ML algorithm approaches to malware detection and selects the most appropriate one. Using ML algorithm strategies for malware detection provides a promising avenue for improving current protective measures against malware. The results of this research suggest that the machine learning algorithm is a highly effective method for detecting malware and could be further developed and implemented in future protective mechanisms.

We collectively taught machine learning algorithms to distinguish between harmful and beneficial information. The DT machine learning approach was the most effective classifier we looked at, with 99% accuracy, as shown in Table 2. In order to achieve the maximum detection accuracy and the most accurate representation of malware, this experiment showed the possibility of static analysis based on PE information and selected important data elements. As the Internet evolved, malicious programmes and associated threats, or "malware," increased in prevalence and sophistication. Because of its quick spread across the Internet, malware writers now have access to a wide range of malware production tools.

The study focused on analyzing and measuring classifier performance to better understand how machine learning works in the detection of malware. Features were extracted from the recovered PE file and library information using latent analysis, and six classifiers based on machine learning techniques were evaluated. The experimental outcomes revealed that the random forest method is preferable for data categorization, with an accuracy of 99.4 percent. This indicates that the PE library is compatible with static analysis and that focusing on only a few properties could improve malware detection and characterization. The main benefit of this approach is that it reduces the risk of accidental installation of malicious software, as users can check the validity of a file before opening it. It is recommended that machine learning systems be trained and tested to determine whether a file is harmful or not.

Due to the exponential growth of malware samples, intelligent methods for efficient and effective malware detection at the cloud (server) side are urgently needed. As a result, much research has been conducted on developing intelligent malware detection systems using data mining and machine learning techniques. In these methods, the process of malware detection is generally divided into two steps:
feature extraction and classification/clustering.
We provide a comprehensive investigation on both the feature extraction and the classification/clustering steps.
We conclude that data-mining-based malware detection framework can be designed to achieve good detection performance with high accuracy while maintaining low false positives. Many developed systems have been successfully integrated into commercial anti-malware products.

The following are some practical insights from our view:

1. Based on different data sets with different feature representation methods, there is no single classifier/clustering algorithm always performing best. In other words, the performance of such malware detection methods critically depend on the extracted features, data distributions, and the categorization methods.

2. Generally, compared with individual classifiers, an ensemble of classifiers can always help improve the detection accuracy. In real applications, a successful malware detection framework should utilize multiple diverse classifiers on various types of feature representations.
3. For feature extraction, both static and dynamic analysis approaches have their own advantages and limitations. In real applications, we suggest using static analysis at first, since over 80% of the file sample collection can be well-represented by static features. If the file cannot be well-represented by static extraction, then we can try dynamic analysis. Novel features, such as file-to-file relation graphs, can also provide invaluable information about the properties of file samples.
4. In order to achieve the best detection performance in real applications, it is often better to have enough training samples with balanced distributions for both classes (malware and benign files).

This paper explored a new direction to extract the API-based dynamic features by analyzing the API calls together with their list of arguments. With the help of machine learning algorithms, we designed two methods to detect Windows malware samples and classify them into their types. The first method deals with the entire list of arguments of each API call as one feature, whereas the second method deals with each argument of each API call separately as one feature. We verified the performance of the proposed methods in malware detection using reasonable datasets of 7105malicious samples belonging to ten distinct types and 7774 benign samples. We showed that our approach outperforms other recent API arguments-based malware detection approaches in terms of accuracy, limitations, and required API information. Our experimental results showed that our malware detection approach gave accuracy of over 99.8992 %, and outperformed the state-of-the-art. Furthermore, we used the same methods to classify the malware samples into their types. The experimental results showed that our malware classification approach gave an accuracy of over 97.9548 % Our approach has promise for wide adoption in API-based malicious behavior detection for Windows platforms.

In particular, it can meet the demands of applications that are currently served by malware detectors that rely on extracting statistical information of the API-based dynamic features but desire better robustness against unfavorable sequence interruption attacks.

2.2 SUMMARY

This section concludes the information about the various survey papers and gives a detailed explanation about these papers. These papers and the detailed explanation give us information about the Detection of Malware using many Algorithms and gives us brief glimpse of the process.

CHAPTER 3
METHODOLOGY

CHAPTER 3

METHODOLOGY

To detect the unknown malware using machine learning technique, a flowchart of our approach shown in below figure. It includes pre-processing of dataset, promising feature selection, training of classifier and detection of advanced malware.

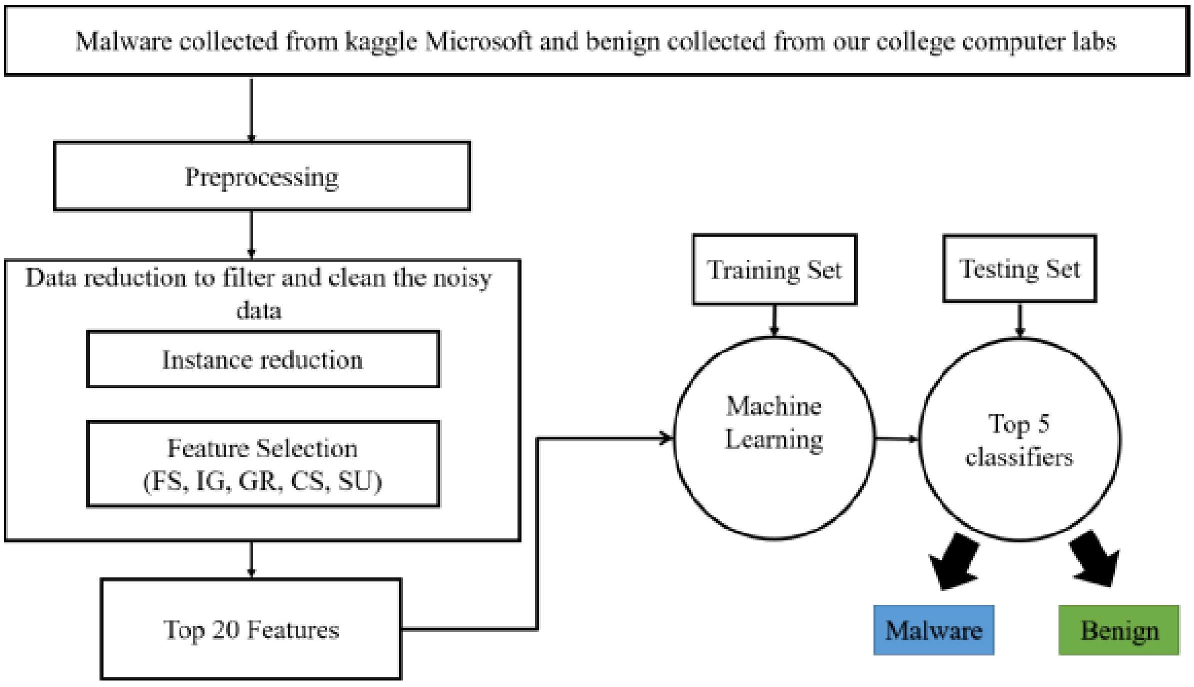

Malware collected from kaggle Microsoft and benign collected from our college computer labs
Preprocessing
Data reduction to filter and clean the noisy data
Instance reduction
Feature Selection (FS, IG, GR, CS, SU)
Top 20 Features
Training Set
Testing Set
Machine Learning
Top 5 classifiers
Malware
Benign

Figure: 3 METHODOLOGY

3.1 ALGORITMS USED:

3.1.1. RANDOM FOREST

3.1.2. DECISION TREE

3.1.3. GRADIENT BOOSTING

3.1.4. AdaBOOST

3.1.5 GNB

3.1.1. RANDOM FOREST

An effective alternative is to use trees with fixed structures and random features. Tree collections are called forests, and classifiers built in so-called random forests. The random water formation algorithm requires three arguments: the data, a desired depth of the decision trees, and a number K of the total decision trees to be built, i. The algorithm generates each of the K trees. independent, which makes it very easy to parallelize. For each tree, build a complete binary tree. The characteristics used for the branches of this tree are selected randomly, usually with replacement, which means that the same characteristic can occur more than 20 times, even in a single branch. a. the leaves of this tree, where predictions are made, are completed based on training data. The last step is the only point at which the training data is used.

The resulting classifier is just a K-lot vote, and random trees. The most amazing thing about this approach is that it actually works remarkably well. They tend to work best when all the features are at least, well, relevant, because the number of features selected for a particular tree is small. One intuitive reason that it works well is the following. Some trees will query unnecessary features. These trees will essentially make random predictions. But some of the trees will happen to question good characteristics and make good predictions (because the leaves are estimated based on training data).

Working of Random Forest Algorithm:

Before understanding the working of the random forest, we must look into the ensemble technique. ***Ensemble*** simply means combining multiple models. Thus, a collection of models is used to make predictions rather than an individual model.

***Ensemble uses two types of methods*:**

1. Bagging– It creates a different training subset from sample training data with replacement & the final output is based on majority voting. For example, Random Forest.

2. Boosting– It combines weak learners into strong learners by creating sequential models such that the final model has the highest accuracy.

For example, ADA BOOST, XG BOOST

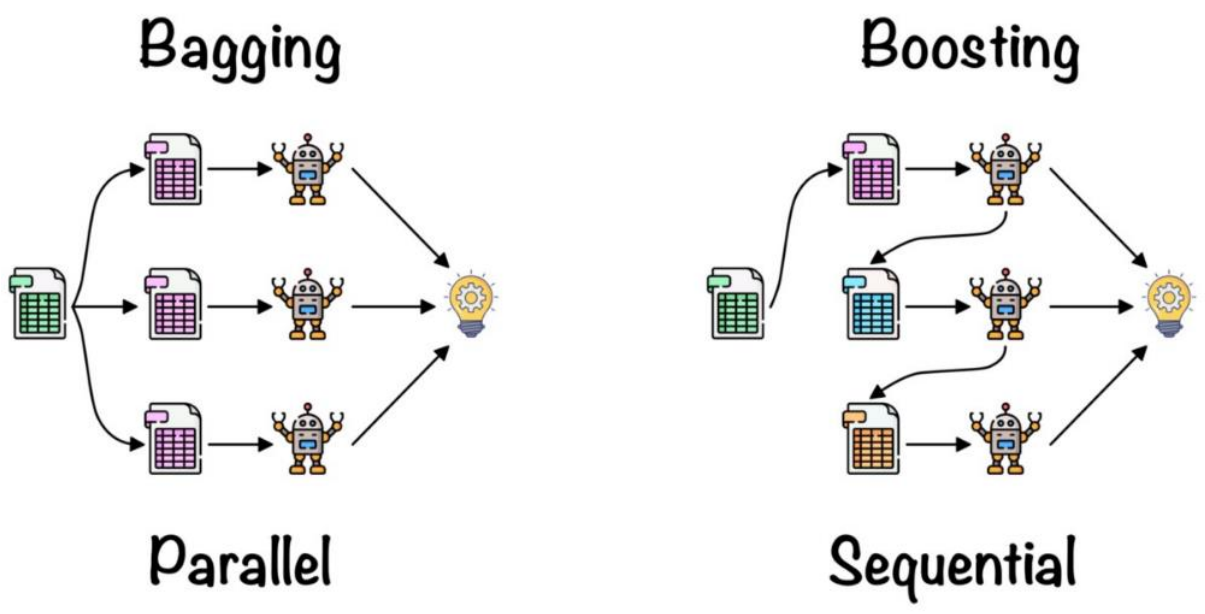


Figure: 3.1 Ensemble

As mentioned earlier, Random Forest works on the Bagging principle. Now, let's dive in and understand bagging in detail.

Bagging

Bagging, also known as *Bootstrap Aggregation* is the ensemble

technique used by random forest. Bagging chooses a random sample from the data set. Hence each model is generated from the samples (Bootstrap Samples) provided by the Original Data with replacement known as *row sampling*. This step of row sampling with replacement is called *bootstrap*. Now each model is trained independently which generates results. The final output is based on majority voting after combining the results of all models. This step which involves combining all the results and generating output based on majority voting is known as *aggregation*.

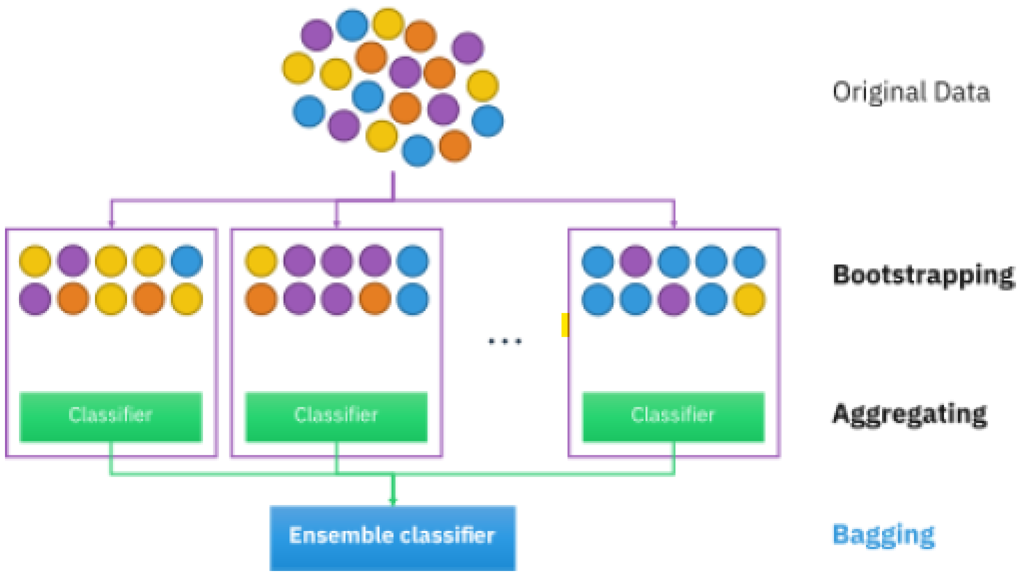


Figure: 3.1.1 Ensemble Classifier

Now let's look at an example by breaking it down with the help of the following figure. Here the bootstrap sample is taken from actual data (Bootstrap sample 01, Bootstrap sample 02, and Bootstrap sample 03) with a replacement which means there is a high possibility that each sample won't contain unique data. Now the model (Model 01, Model 02, and Model 03) obtained from this bootstrap sample is trained independently. Each model generates results as shown. Now Happy emoji is having a majority when compared to sad emoji. Thus based on majority voting final output is obtained as Happy emoji.

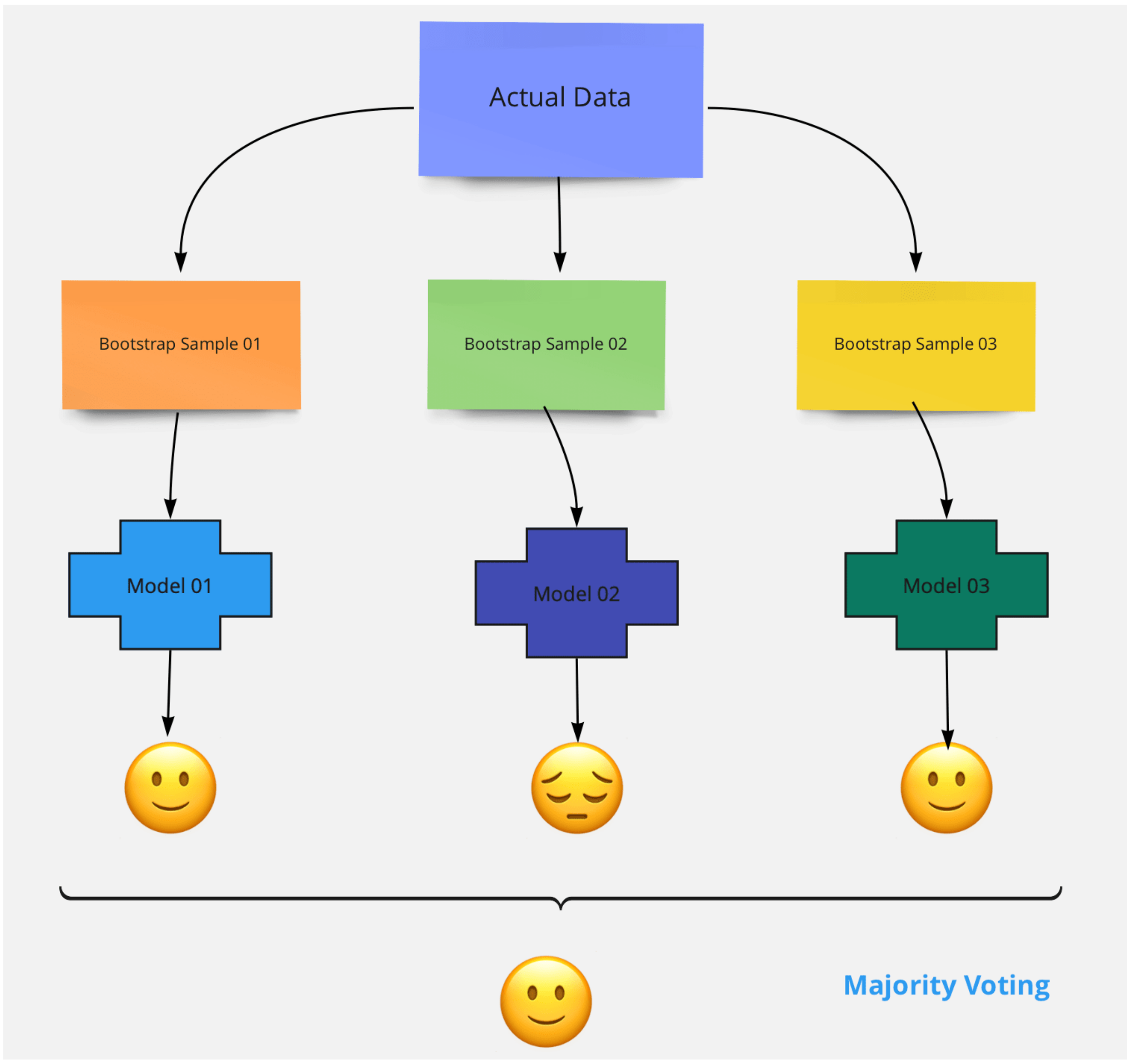


Figure: 3.1.2 Bagging Ensemble Method

Steps involved in random forest algorithm:

Step 1: In Random Forest n number of random records are taken from the data set having k number of records.

Step 2: Individual decision trees are constructed for each sample.

Step 3: Each decision tree will generate an output.

Step 4: Final output is considered based on *Majority Voting or Averaging* for Classification and regression respectively.

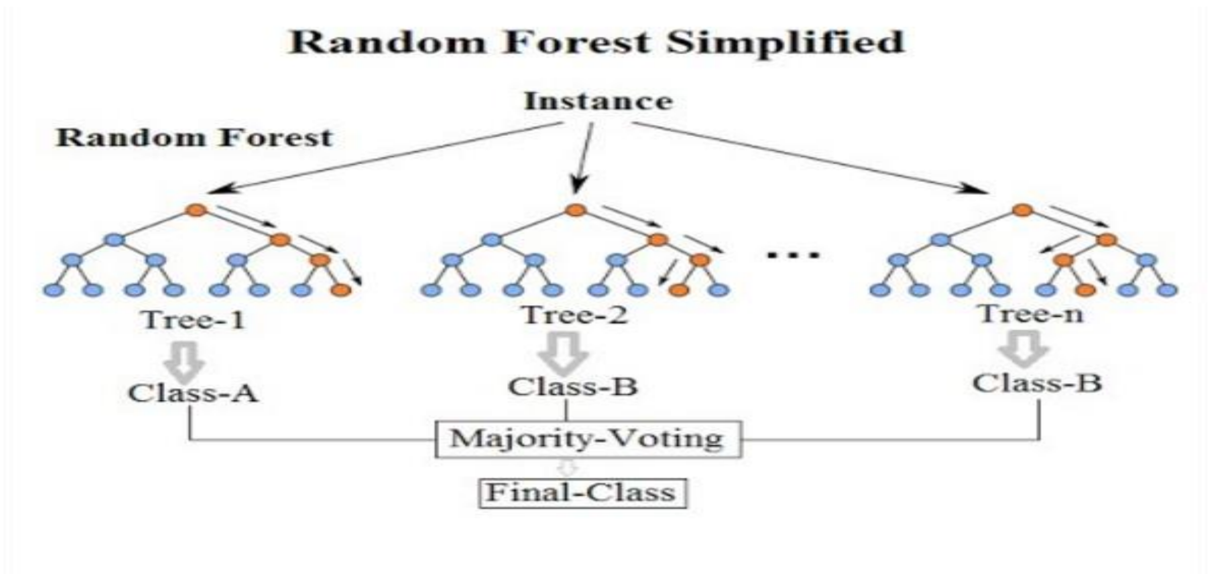


Figure: 3.1.3 Random Forest View

Important Features of Random Forest

1. Diversity- Not all attributes/variables/features are considered while making an individual tree, each tree is different.

2. Immune to the curse of dimensionality- Since each tree does not consider all the features, the feature space is reduced.

3. Parallelization-Each tree is created independently out of different data and attributes. This means that we can make full use of the CPU to build random forests.

4. Train-Test split- In a random forest we don’t have to segregate the data for train and test as there will always be 30% of the data which is not seen by the decision tree.

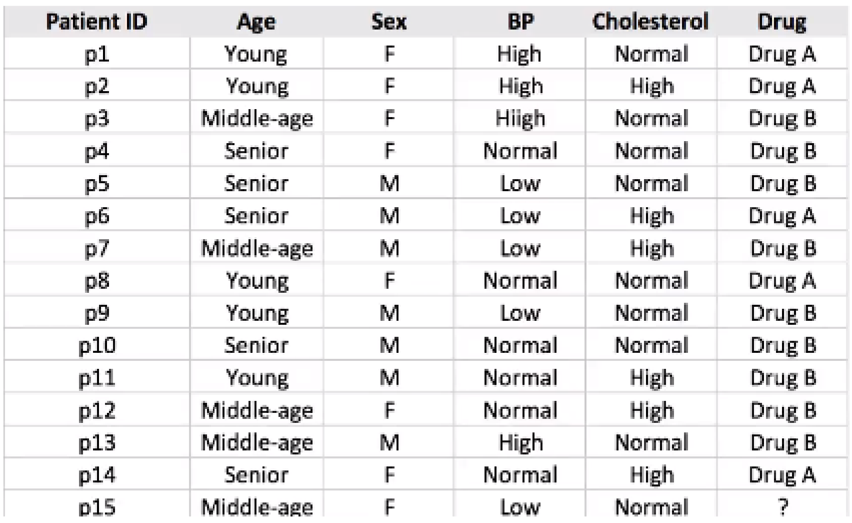
5. Stability- Stability arises because the result is based on majority voting/ averaging.

3.1.2. DECISION TREE

The decision tree Algorithm belongs to the family of supervised machine learning algorithms. It can be used for both a classification problem as well as for a regression problem.

The goal of this algorithm is to create a model that predicts the value of a target variable, for which the decision tree uses the tree representation to solve the problem in which the leaf node corresponds to a class label and attributes are represented on the internal node of the tree.

Let's take a sample data set to move further ….



Patient ID	Age	Sex	BP	Cholesterol	Drug
p1	Young	F	High	Normal	Drug A
p2	Young	F	High	High	Drug A
p3	Middle-age	F	Hiigh	Normal	Drug B
p4	Senior	F	Normal	Normal	Drug B
p5	Senior	M	Low	Normal	Drug B
p6	Senior	M	Low	High	Drug A
p7	Middle-age	M	Low	High	Drug B
p8	Young	F	Normal	Normal	Drug A
p9	Young	M	Low	Normal	Drug B
p10	Senior	M	Normal	Normal	Drug B
p11	Young	M	Normal	High	Drug B
p12	Middle-age	F	Normal	High	Drug B
p13	Middle-age	M	High	Normal	Drug B
p14	Senior	F	Normal	High	Drug A
p15	Middle-age	F	Low	Normal	?

TABLE 3 Sample data

Suppose we have a sample of 14 patient data set and we have to predict which drug to suggest to the patient A or B.

Let's say we pick cholesterol as the first attribute to split data

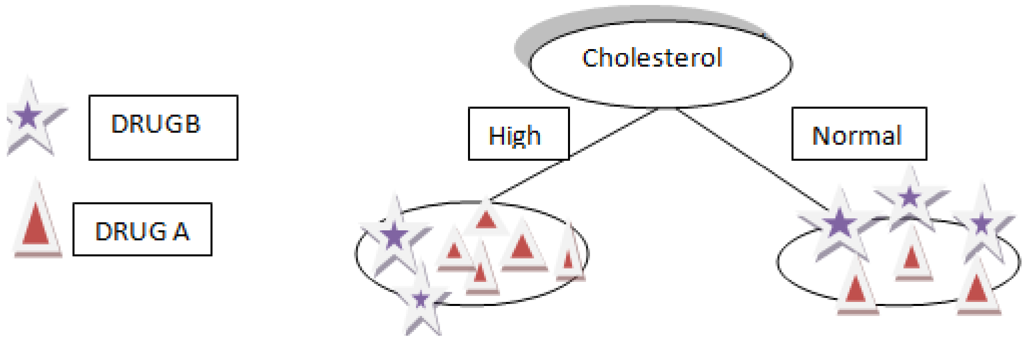


Figure: 3.2.1 Sample of patient data

It will split our data into two branches High and Normal based on cholesterol, as you can see in the above figure.

Let's suppose our new patient has high cholesterol by the above split of our data we cannot say whether Drug B or Drug A will be suitable for the patient.

Also, If the patient cholesterol is normal, we still do not have an idea or information to determine that either Drug A or Drug B is Suitable for the patient.

Let us take Another Attribute Age, as we can see age has three categories in it Young, middle age and senior let's try to split.

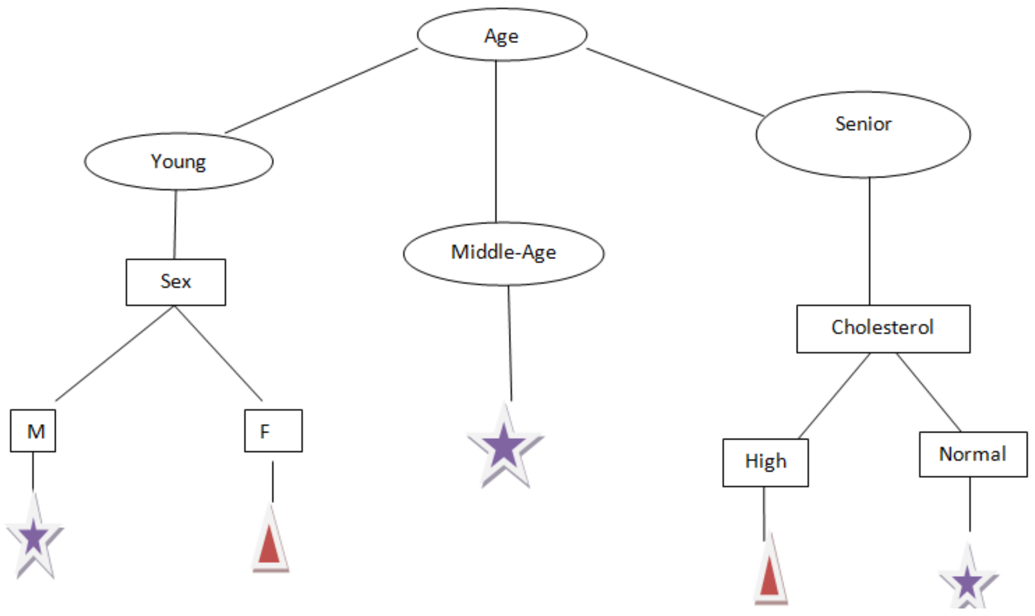


Figure: 3.2.2 predict data

From the above figure, now we can say that we can easily predict which Drug to give to a patient based on his or her reports.

Assumptions that we make while using the Decision tree:

– In the beginning, we consider the whole training set as the root.

-Feature values are preferred to be categorical, if the values continue then they are converted to discrete before building the model.

-Based on attribute values records are distributed recursively.

-We use a statistical method for ordering attributes as a root node or the internal node.

Mathematics behind Decision tree algorithm: Before going to the Information Gain first we have to understand entropy

Entropy: Entropy is the measures of impurity, disorder, or uncertainty in a bunch of examples.

Purpose of Entropy:

Entropy controls how a Decision Tree decides to split the data. It affects how a Decision Tree draws its boundaries.

"Entropy values range from 0 to 1", Less the value of entropy more it is trusting able.

Formula for Entropy:

$$H(s) = -probablity\ of\ \log_2(p+) - -probability\ of\ \log_2(p-)$$

<u>Where</u>

$(p+)$ → % of positive class

$(p-)$→ % of negative class

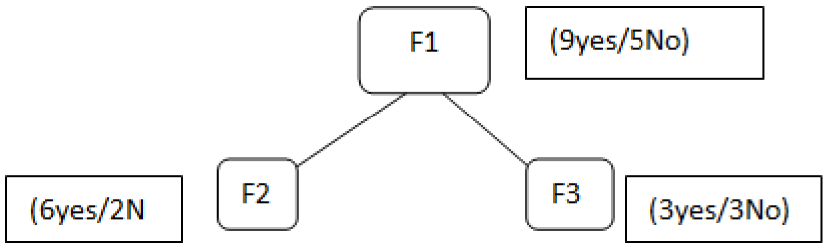


Figure: 3.2.3 node data

Suppose we have F1, F2, F3 features we selected the F1 feature as our root node F1 contains 9 yes label and 5 no label in it, after splitting the F1 we get F2 which have 6 yes/2 No and F3 which have 3 yes/3 no. Now if we try to calculate the Entropy of both F2 by using the Entropy formula. Putting the values in the formula:

$$H(s) = -\frac{6}{8}\log_2 6\backslash 8 - \frac{2}{8}\log_2\frac{2}{8} = 0.81\ bits$$

Here, 6 is the number of yes taken as positive as we are calculating probability divided by 8 is the total rows present in the F2.

Similarly, if we perform Entropy for F3 we will get 1 bit which is a case of an attribute as in it there is 50%, yes and 50% no. This splitting will be going on unless and until we get a pure subset.

What is a Pure subset?

The pure subset is a situation where we will get either all yes or all no in this case. We have performed this concerning one node what if after splitting F2 we may also require some other attribute to reach the leaf node & we also have to take the entropy of those values and add it to do the submission of all those entropy values for that we have the concept of information gain.

Information Gain: Information gain is used to decide which feature to split on at each step in building the tree. Simplicity is best, so we want to keep our tree small. To do so, at each step we should choose the split that results in the purest daughter nodes. A commonly used measure of purity is called information.

For each node of the tree, the information value measures how much information a feature gives us about the class. The split with the highest information gain will be taken as the first split and the process will continue until all children nodes are pure, or until the information gain is 0.

Formula for Information Gain:

$$Gain(S,A) = H(s) \sum \frac{/Sv/}{/s/} H(Sv)$$

The algorithm calculates the information gain for each split and the split which is giving the highest value of information gain is selected.

We can say that in Information gain we are going to compute the

average of all the entropy-based on the specific split.

Sv = Total sample after the split as in F2 there are 6 yes

S = Total Sample as in F1=9+5=14

Now calculating the Information Gain:

$$Gain(S,A) = H(F1) - \frac{8}{14} * h(F2) - \frac{6}{14} * H(F3)$$

$$= 0.91 - \frac{8}{14} * 0.81 - \frac{6}{14} * 1$$

$$= 0.05$$

Like this, the algorithm will perform this for n number of splits, and the information gain for whichever split is higher it is going to take it in order to construct the decision tree.

The higher the value of information gain of the split the higher the

chance of it getting selected for the particular split.

Gini Impurity:

Gini Impurity is a measurement used to build Decision Trees to

determine how the features of a data set should split nodes to form the tree. More precisely, the Gini Impurity of a data set is a number between 0-0.5, which indicates the likelihood of new, random data being miss classified if it were given a random class label according to the class distribution in the data set.

Entropy vs Gini Impurity

The maximum value for entropy is 1 whereas the maximum value for Gini impurity is 0.5.As the Gini Impurity does not contain any logarithmic function to calculate it takes less computational time as compared to entropy.

3.1.3. GRADIENT BOOSTING

The idea behind "gradient boosting" is to take a weak hypothesis or weak learning algorithm and make a series of tweaks to it that will improve the strength of the hypothesis/learner. This type of Hypothesis Boosting is based on the idea of Probability Approximately Correct Learning (PAC). This PAC learning method investigates machine learning problems to interpret how complex they are, and a similar method is applied to Hypothesis Boosting.

In hypothesis boosting, you look at all the observations that the machine learning algorithm is trained on, and you leave only the observations that the machine learning method successfully classified behind, stripping out the other observations. A new weak learner is created and tested on the set of data that was poorly classified, and then just the examples that were successfully classified are kept.

The Gradient Boosting Classifier depends on a loss function. A custom loss function can be used, and many standardized loss functions are supported by gradient boosting classifiers, but the loss function has to be differentiable. Classification algorithms frequently use logarithmic loss, while regression algorithms can use squared errors. Gradient boosting systems don't have to derive a new loss function every time the boosting algorithm is added, rather any differentiable loss function can be applied to the system.

Gradient boosting systems have two other necessary parts: a weak learner and an additive component. Gradient boosting systems use decision trees as their weak learners. Regression trees are used for the weak learners, and these regression trees output real values.

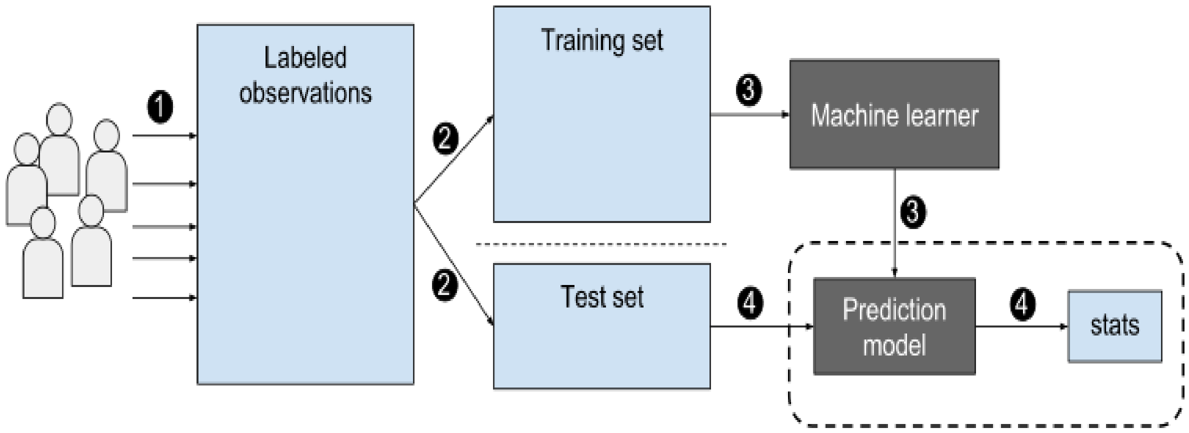


Figure: 3.3 Gradient Boost

Like other boosting methods, gradient boosting combines weak "learners" into a single strong learner in an iterative fashion. It is easiest to explain in the least-squares regression setting, where the goal is to "teach" a model ***F*** to predict values of the form

$$\hat{y} = F(x)$$

by minimizing the mean squared error

$$\frac{1}{n}\sum_i(\hat{y}_i - y_i)^2$$

where indexes over some training set of size of actual values of the output variable :

- $\hat{y}_i =$ the predicted value $F(x_i)$
- $y_i =$ the observed value
- $n =$ the number of samples in y

In many supervised learning problems, there is an output variable y and a vector of input variables x, related to each other with some probabilistic distribution. The goal is to find some function {F}(x) that best approximates the output variable from the values of input variables. This is formalized by introducing some loss function L (y, F(x)) and minimizing it in expectation:

$$\hat{F} = \arg\min_{F} \mathbb{E}_{x,y}[L(y, F(x))]$$

The gradient boosting method assumes a real-valued y. It seeks an approximation that{F}(x) in the form of a weighted sum of M functions h$_m$(x) from some class H, called base (or weak) learners:

$$\hat{F}(x) = \sum_{m=1}^{M} \gamma_m h_m(x) + \text{const.}$$

We are usually given a training set {(x {1}, y_{1}),\dots ,(x_{n},y_{n})\} of known sample values of x & corresponding values of y. In accordance with the empirical risk minimization principle, the method tries to find an approximation that{F}(x) that minimizes the average value of the loss function on the training set, i.e., minimizes the empirical risk. It does so by starting with a model, consisting of a constant function F{0}(x), and incrementally expands it in a greedy fashion:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma),$$

$$F_m(x) = F_{m-1}(x) + \left(\arg\min_{h_m \in \mathcal{H}} \left[\sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + h_m(x_i)) \right] \right)(x),$$

Unfortunately, choosing the best function h$_m$ at each step for an arbitrary loss function L is a computationally infeasible optimization problem in general. Therefore, we restrict our approach to a simplified version of the problem. The idea is to apply a steepest descent step to this minimization problem (functional gradient descent).

The basic idea behind the steepest descent is to find a local minimum of the loss function by In fact, the local maximum-descent direction of the loss function is the negative gradient. Hence, moving a small amount that the linear approximation remains valid.

Usage:

Gradient boosting can be used in the field of learning to rank. The commercial web search engines Yahoo and Yandex use variants of gradient boosting in their machine-learned ranking engines. Gradient boosting is also utilized in High Energy Physics in data analysis. At the Large Hadron Collider (LHC), variants of gradient boosting Deep Neural Networks (DNN) were successful in reproducing the results of non-machine learning methods of analysis on datasets used to discover the Higgs boson. Gradient boosting decision tree was also applied in earth and geological studies – for example quality evaluation of sandstone reservoir.

First, let’s import all required libraries.

```
# Import all relevant libraries
from sklearn.ensemble import GradientBoostingClassifier
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn import preprocessing
import warnings
```

warnings.filterwarnings("ignore")

Now let’s read the dataset and look at the columns to understand the information better.

df = pd.read_csv(FILENAME.csv')

df.head()

Example: 1 **Classification**

Steps:

Import the necessary libraries

Setting SEED for reproducibility

Load the digit dataset and split it into train and test.

Instantiate Gradient Boosting classifier and fit the model.

Predict the test set and compute the accuracy score.

Example: 2 **Regression**

Steps:

Import the necessary libraries

Setting SEED for reproducibility

Load the diabetes dataset and split it into train and test.

Instantiate Gradient Boosting Regressor and fit the model.

Predict on the test set and compute RMSE.

3.1.4. AdaBoost

AdaBoost models belong to a class of ensemble machine learning models. From the literal meaning of the word ‘ensemble’, we can easily have much better intuition of how this model works. Ensemble models take the onus of combining different models and later produce an advanced/more accurate meta model. This meta model has comparatively high accuracy in terms of prediction as compared to their corresponding counterparts. We have read about the working of these ensemble models in the article Ensemble Classifier | Data Mining.

AdaBoost, also called Adaptive Boosting, is a technique in Machine Learning used as an Ensemble Method. The most common estimator used with AdaBoost is decision trees with one level which means Decision trees with only 1 split. These trees are also called Decision Stumps.

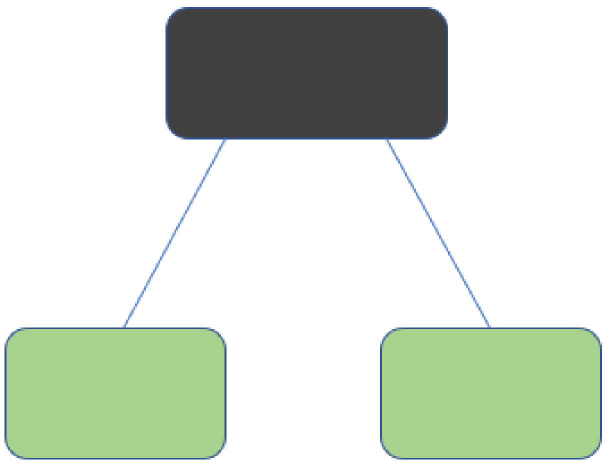


Figure: 3.4 Decision Stumps

What this algorithm does is that it builds a model and gives equal weights to all the data points. It then assigns higher weights to points that are wrongly classified. Now all the points with higher weights are given more importance in the next model. It will keep training models until and unless a lower error is received.

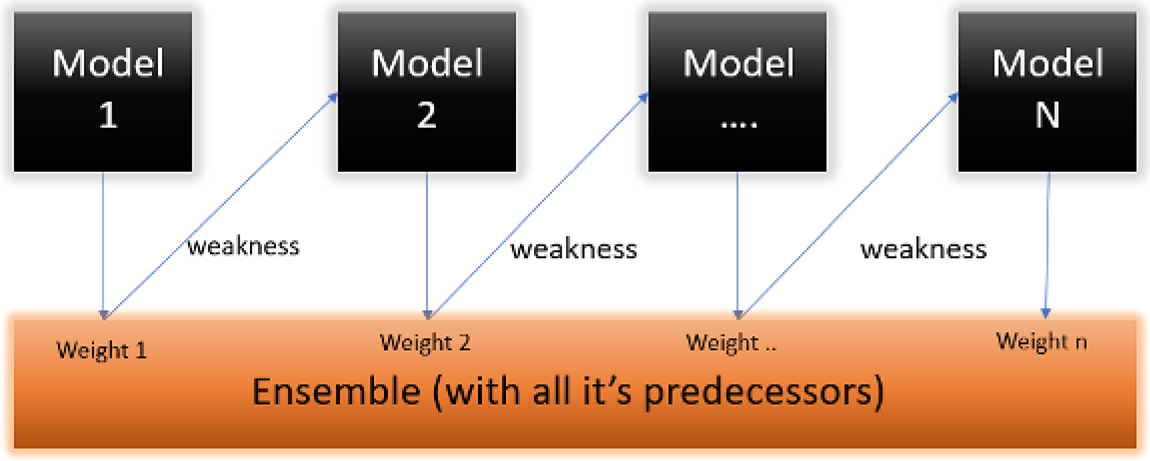


Figure: 3.4.1 Linear Regression Model

Let's take an example to understand this, suppose you built a decision tree algorithm on the Titanic dataset, and from there, you get an accuracy of 80%. After this, you apply a different algorithm and check the accuracy, and it comes out to be 75% for KNN and 70% for Linear Regression.

We see the accuracy differs when we build a different model on the same dataset. But what if we use combinations of all these algorithms to make the final prediction? We'll get more accurate results by taking the average of the results from these models. We can increase the prediction power in this way.

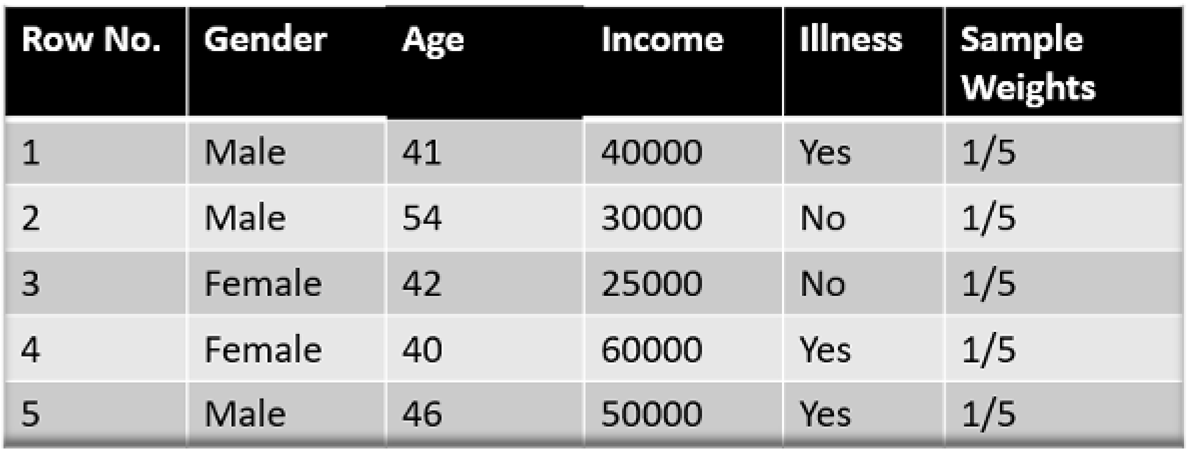
Here we will be more focused on mathematics intuition.

There is another ensemble learning algorithm called the gradient boosting algorithm. In this algorithm, we try to reduce the error instead of wights, as in AdaBoost. But in this article, we will only be focusing on the mathematical intuition of AdaBoost.

Working of the AdaBoost Algorithm

Let’s understand what and how this algorithm works under the hood with the following :

Step 1 – The Image shown below is the actual representation of our dataset. Since the target column is binary, it is a classification problem. First of all, these data points will be assigned some weights. Initially, all the weights will be equal.



Row No.	Gender	Age	Income	Illness	Sample Weights
1	Male	41	40000	Yes	1/5
2	Male	54	30000	No	1/5
3	Female	42	25000	No	1/5
4	Female	40	60000	Yes	1/5
5	Male	46	50000	Yes	1/5

TABLE 4 Dataset

The formula to calculate the sample weights is:

$$w(x_i, y_i) = \frac{1}{N}, \ i = 1, 2, \ldots . n$$

Where N is the total number of data points

Here since we have 5 data points, the sample weights assigned will be 1/5.

Step 2 – We start by seeing how well “Gender” classifies the samples and will see how the variables (Age, Income) classify the samples.

We’ll create a decision stump for each of the features and then calculate the Gini Index of each tree. The tree with the lowest Gini Index will be our first stump.

Here in our dataset, let’s say Gender has the lowest gini index, so it will be our first stump.

Step 3 – We’ll now calculate the “Amount of Say” or “Importance” or “Influence” for this classifier in classifying the data points using this formula:

$$\frac{1}{2}\log\frac{1 - Total\ Error}{Total\ Error}$$

The total error is nothing but the summation of all the sample weights of misclassified data points.

Here in our dataset, let's assume there is 1 wrong output, so our total error will be 1/5, and the alpha (performance of the stump) will be:

$$Performance\ of\ the\ stump\ =\ \frac{1}{2}\log_e\left(\frac{1 - Total\ Error}{Total\ Error}\right)$$

$$\alpha\ =\ \frac{1}{2}\log_e\left(\frac{1-\frac{1}{5}}{\frac{1}{5}}\right)$$

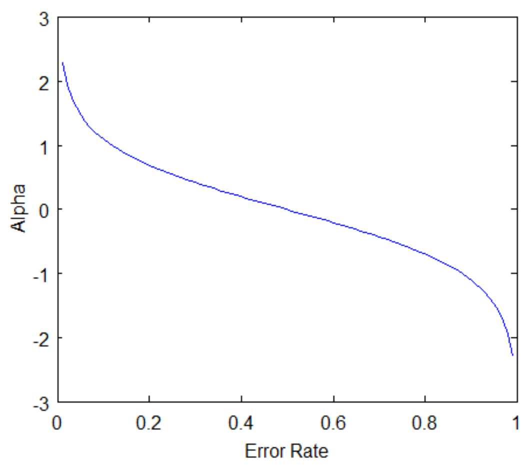
$$\alpha\ =\ \frac{1}{2}\log_e\left(\frac{0.8}{0.2}\right)$$

$$\alpha\ =\ \frac{1}{2}\log_e(4)\ =\ \frac{1}{2}*(1.38)$$

$$\alpha\ =\ 0.69$$

Note: Total error will always be between 0 and 1.

0 Indicates perfect stump, and 1 indicates horrible stump.



From the graph above, we can see that when there is no misclassification, then we have no error (Total Error = 0), so the "amount of say (alpha)" will be a large number. When the classifier predicts half right and half wrong, then the Total Error = 0.5, and the importance (amount of say) of the classifier will be 0.

If all the samples have been incorrectly classified, then the error will be very high (approx. to 1), and hence our alpha value will be a negative integer.

Step 4 – You must be wondering why it is necessary to calculate the TE and performance of a stump. Well, the answer is very simple, we need to update the weights because if the same weights are applied to the next model, then the output received will be the same as what was received in the first model.

The wrong predictions will be given more weight, whereas the correct predictions weights will be decreased. Now when we build our next model after updating the weights, more preference will be given to the points with higher weights.

After finding the importance of the classifier and total error, we need to finally update the weights, and for this, we use the following formula:

$$New\ sample\ weight\ =\ old\ weight\ *\ e^{\pm Amount\ of\ say\ (\alpha)}$$

The amount of, say (alpha) will be negative when the sample is correctly classified.

The amount of, say (alpha) will be positive when the sample is miss-classified.

There are four correctly classified samples and 1 wrong. Here, the sample weight of that datapoint is 1/5, and the amount of say/performance of the stump of Gender is 0.69.

New weights for correctly classified samples are:

$$New\ sample\ weight\ =\ \frac{1}{5}\ *\ \exp(-0.69)$$

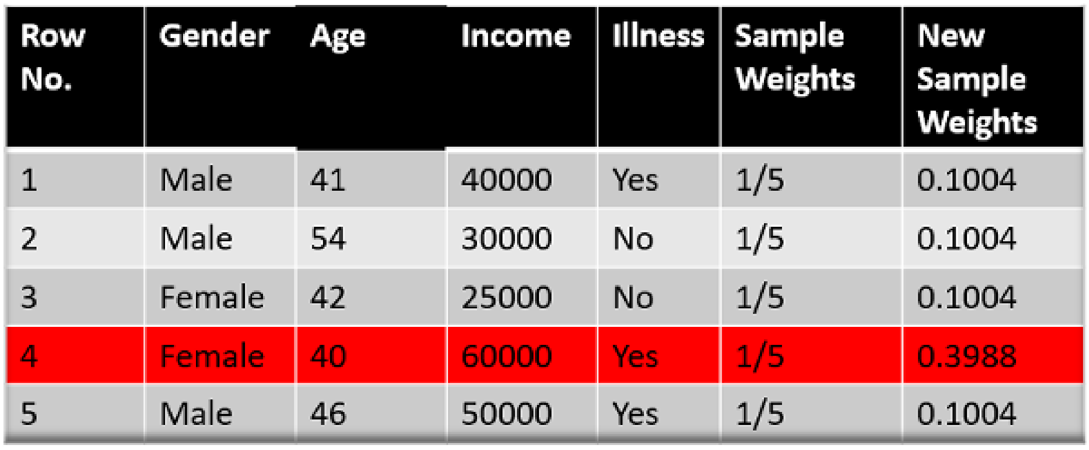
$$New\ sample\ weight\ =\ 0.2\ *\ 0.502\ =\ 0.1004$$

For *wrongly classified* samples, the updated weights will be:

$$New\ sample\ weight = \frac{1}{5} * \exp(0.69)$$

$$New\ sample\ weight = 0.2 * 1.994 = 0.3988$$

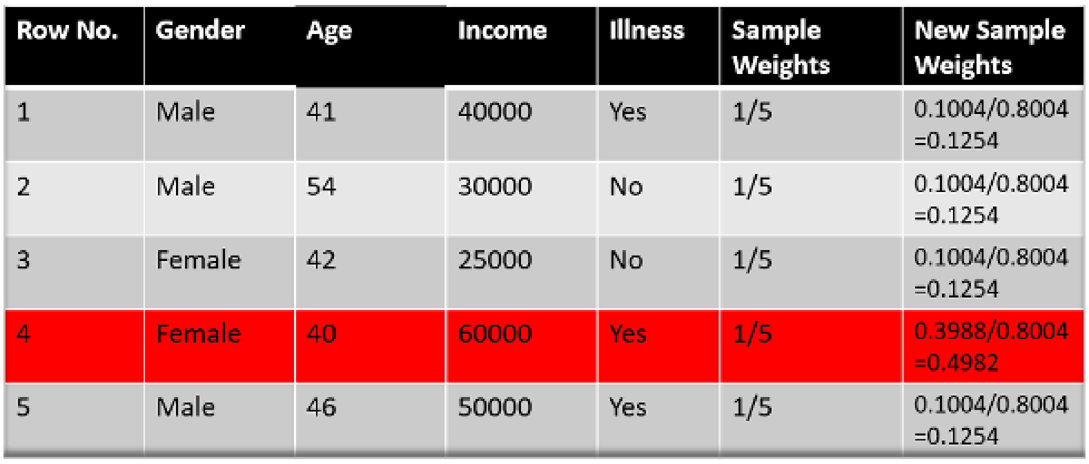
Note: See the sign of alpha when I am putting the values, the alpha is negative when the data point is correctly classified, and this decreases the sample weight from 0.2 to 0.1004. It is positive when there is misclassification, this will increase the sample weight from 0.2 to 0.3988



Row No.	Gender	Age	Income	Illness	Sample Weights	New Sample Weights
1	Male	41	40000	Yes	1/5	0.1004
2	Male	54	30000	No	1/5	0.1004
3	Female	42	25000	No	1/5	0.1004
4	Female	40	60000	Yes	1/5	0.3988
5	Male	46	50000	Yes	1/5	0.1004

TABLE 5 Dataset

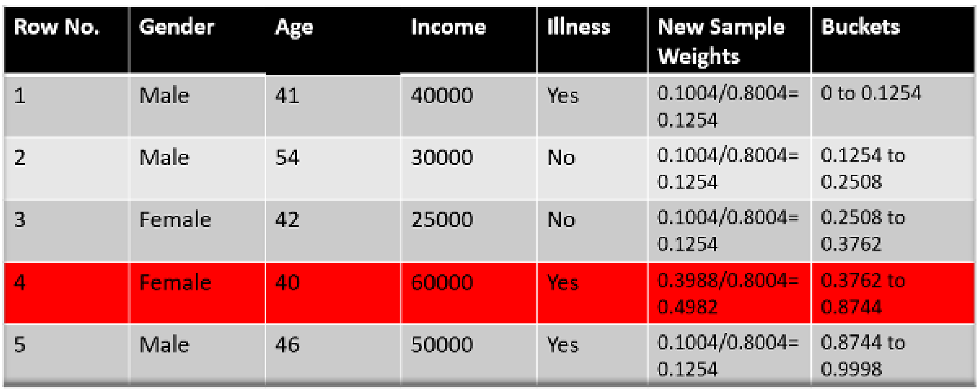
We know that the total sum of the sample weights must be equal to 1, but here if we sum up all the new sample weights, we will get 0.8004. To bring this sum equal to 1, we will normalize these weights by dividing all the weights by the total sum of updated weights, which is 0.8004. So, after normalizing the sample weights, we get this dataset, and now the sum is equal to 1.



Row No.	Gender	Age	Income	Illness	Sample Weights	New Sample Weights
1	Male	41	40000	Yes	1/5	0.1004/0.8004 =0.1254
2	Male	54	30000	No	1/5	0.1004/0.8004 =0.1254
3	Female	42	25000	No	1/5	0.1004/0.8004 =0.1254
4	Female	40	60000	Yes	1/5	0.3988/0.8004 =0.4982
5	Male	46	50000	Yes	1/5	0.1004/0.8004 =0.1254

TABLE 6 Dataset

Step 5 – Now, we need to make a new dataset to see if the errors decreased or not. For this, we will remove the “sample weights” and “new sample weights” columns and then, based on the “new sample weights,” divide our data points into buckets.



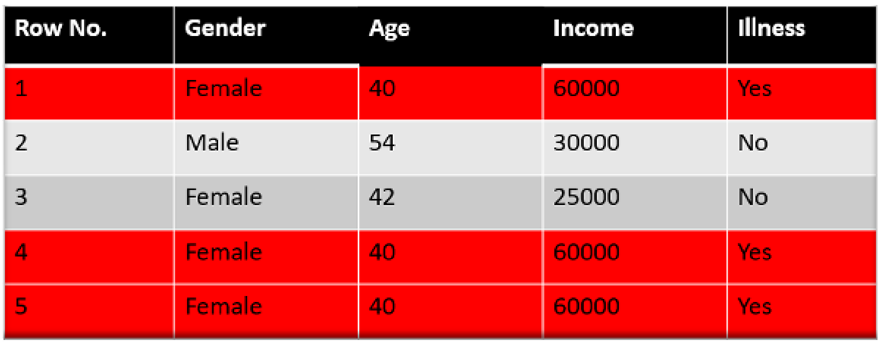
Row No.	Gender	Age	Income	Illness	New Sample Weights	Buckets
1	Male	41	40000	Yes	0.1004/0.8004= 0.1254	0 to 0.1254
2	Male	54	30000	No	0.1004/0.8004= 0.1254	0.1254 to 0.2508
3	Female	42	25000	No	0.1004/0.8004= 0.1254	0.2508 to 0.3762
4	Female	40	60000	Yes	0.3988/0.8004= 0.4982	0.3762 to 0.8744
5	Male	46	50000	Yes	0.1004/0.8004= 0.1254	0.8744 to 0.9998

TABLE 7 Dataset (sample weights)

Step 6 – We are almost done. Now, what the algorithm does is selects random numbers from 0-1. Since incorrectly classified records have higher sample weights, the probability of selecting those records is very high.

Suppose the 5 random numbers our algorithm take is 0.38,0.26,0.98,0.40,0.55.

Now we will see where these random numbers fall in the bucket, and according to it, we’ll make our new dataset shown below.



Row No.	Gender	Age	Income	Illness
1	Female	40	60000	Yes
2	Male	54	30000	No
3	Female	42	25000	No
4	Female	40	60000	Yes
5	Female	40	60000	Yes

TABLE 8 Dataset (random numbers)

This comes out to be our new dataset, and we see the data point, which was wrongly classified, has been selected 3 times because it has a higher weight.

Step 9 – Now this act as our new dataset, and we need to repeat all the above steps i.e.

Assign equal weights to all the data points.

Find the stump that does the best job classifying the new collection of samples by finding their Gini Index and selecting the one with the lowest Gini index.

Calculate the “Amount of Say” and “Total error” to update the previous sample weights.

Normalize the new sample weights.

Iterate through these steps until and unless a low training error is achieved.

Suppose, with respect to our dataset, we have constructed 3 decision trees (DT1, DT2, DT3) in a sequential manner. If we send our test data now, it will pass through all the decision trees, and finally, we will see which class has the majority, and based on that, we will do predictions
for our test dataset.

Variants:

Real AdaBoost

The output of decision trees is a class probability estimate p(x) = P(y=1 | x), the probability that x is in the positive class.

LogitBoost

LogitBoost represents an application of established logistic regression techniques to the AdaBoost method. Rather than minimizing error with respect to y, weak learners are chosen to minimize the (weighted least-squares) error of $f_t(x)$ with respect to

$$z_t = \frac{y^* - p_t(x)}{2p_t(x)(1 - p_t(x))},$$

Gentle AdaBoost

While previous boosting algorithms choose f_{t} greedily, minimizing the overall test error as much as possible at each step, GentleBoost features a bounded step size. ft is chosen to minimize and no further coefficient is applied. Thus, in the case where a weak learner exhibits perfect classification performance, GentleBoost chooses $f_t(x) = a_t/h_t(x)$ exactly equal to y, while steepest descent algorithms try to set $a_t = \infty$. Empirical observations about the good performance of GentleBoost appear to back up Schapire and Singer's remark that allowing excessively large values of *a* can lead to poor generalization performance.

3.1.5. GNB: Gaussian Naive Bayes

Naïve Bayes is a probabilistic machine learning algorithm used for many classification functions and is based on the Bayes theorem. Gaussian Naïve Bayes is the extension of naïve Bayes. While other functions are used to estimate data distribution, Gaussian or normal distribution is the simplest to implement as you will need to calculate the mean and standard deviation for the training data.

What is the Naive Bayes Algorithm?

Naive Bayes is a probabilistic machine learning algorithm that can be used in several classification tasks. Typical applications of Naive Bayes are classification of documents, filtering spam, prediction and so on. This algorithm is based on the discoveries of Thomas Bayes and hence its name.

The name “Naïve” is used because the algorithm incorporates features in its model that are independent of each other. Any modifications in the value of one feature do not directly impact the value of any other feature of the algorithm. The main advantage of the Naïve Bayes algorithm is that it is a simple yet powerful algorithm.

It is based on the probabilistic model where the algorithm can be coded easily, and predictions did quickly in real-time. Hence this algorithm is the typical choice to solve real-world problems as it can be tuned to respond to user requests instantly. But before we dive deep into Naïve Bayes and Gaussian Naïve Bayes, we must know what is meant by conditional probability.

Conditional Probability Explained

We can understand conditional probability better with an example. When you toss a coin, the probability of getting ahead or a tail is 50%. Similarly, the probability of getting a 4 when you roll dice with faces is 1/6 or 0.16.

If we take a pack of cards, what is the probability of getting a queen given the condition that it is a spade? Since the condition is already set that it must be a spade, the denominator or the selection set becomes 13. There is only one queen in spades, hence the probability of picking a queen of spade becomes 1/13= 0.07.

The conditional probability of event A given event B means the probability of event A occurring given that event B has already occurred. Mathematically, the conditional probability of A given B can be denoted as P[A|B] = P[A AND B] / P[B].

The Bayes Rule

In the Bayes rule, we go from P (X | Y) that can be found from the training dataset to find P (Y | X). To achieve this, all you need to do is replace A and B with X and Y in the above formulae. For observations, X would be the known variable and Y would be the unknown variable. For each row of the dataset, you must calculate the probability of Y given that X has already occurred.

But what happens where there are more than 2 categories in Y? We must compute the probability of each Y class to find out the winning one.

Through Bayes rule, we go from P (X | Y) to find P (Y | X)

Known from training data: P (X | Y) = P (X ∩ Y) / P(Y)

P (Evidence | Outcome)

Unknown – to be predicted for test data: P (Y | X) = P (X ∩ Y) / P(X)

P (Outcome | Evidence)

Bayes Rule = P (Y | X) = P (X | Y) * P (Y) / P (X)

The Naïve Bayes

The Bayes rule provides the formula for the probability of Y given condition X. But in the real world, there may be multiple X variables. When you have independent features, the Bayes rule can be extended to the Naïve Bayes rule. The X's are independent of each other. The Naïve Bayes formula is more powerful than the Bayes formula

Gaussian Naïve Bayes

So far, we have seen that the X's are in categories but how to compute probabilities when X is a continuous variable? If we assume that X follows a particular distribution, you can use the probability density function of that distribution to calculate the probability of likelihoods. If we assume that X's follow a Gaussian or normal distribution, we must substitute the probability density of the normal distribution and name it Gaussian Naïve Bayes. To compute this formula, you need the mean and variance of X.

$$P(X|Y=c) = \frac{1}{\sqrt{2\pi\sigma_c^2}} e^{\frac{-(x-\mu_c)^2}{2\sigma_c^2}}$$

In the above formulae, sigma and mu is the variance and mean of the continuous variable X computed for a given class c of Y.

Representation for Gaussian Naïve Bayes

The above formula calculated the probabilities for input values for each class through a frequency. We can calculate the mean and standard deviation of x's for each class for the entire distribution.

This means that along with the probabilities for each class, we must also store the mean and the standard deviation for every input variable for the class.

mean(x) = 1/n * sum(x)

where n represents the number of instances and x is the value of the input variable in the data.

standard deviation(x) = sqrt(1/n * sum(xi-mean(x)^2))

Here square root of the average of differences of each x and the mean of x is calculated where n is the number of instances, sum() is the sum function, sqrt() is the square root function, and xi is a specific x value.

Predictions with the Gaussian Naïve Bayes Model

The Gaussian probability density function can be used to make predictions by substituting the parameters with the new input value of the variable and as a result, the Gaussian function will give an estimate for the new input value's probability.

Naïve Bayes Classifier

The Naïve Bayes classifier assumes that the value of one feature is independent of the value of any other feature. Naïve Bayes classifiers need training data to estimate the parameters required for classification. Due to simple design and application, Naïve Bayes classifiers can be suitable in many real-life scenarios.

The **GNB** classifier is a quick and simple classifier technique that works very well without too much effort and a good level of accuracy.

3.2 SUMMARY

The methodology described in these touches on the various points and a flowchart of our approach is shown. Here, the brief description of the algorithms used in this process which includes pre-processing of dataset, promising feature selection, training of classifier and detection of advanced malware. In summary, the whole model is explained step by step along with the algorithm to describe the malware detection process.

CHAPTER 4
DESIGN & DEVELOPMENT

CHAPTER 4

DESIGN AND DEVELOPMENT

Here we introduce the various steps and components of a typical machine learning workflow for malware detection and classification, explores the challenges and limitations of such a workflow, and assesses the most recent innovations and trends in the field, with an emphasis on deep learning techniques. The proposed Design of this research study is provided below.

To provide a more complete understanding of the proposed machine learning method for malware detection, Figure 4.1 and Figure 4.2 illustrate the workflow process from start to finish.

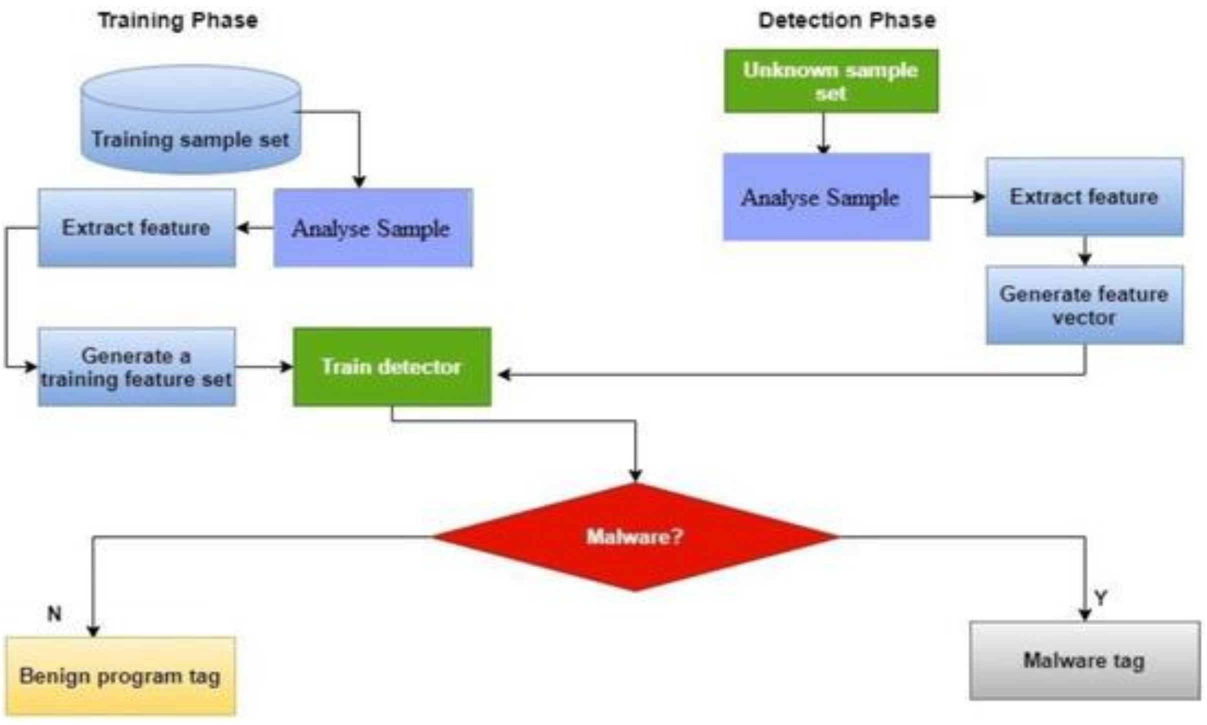


Figure 4.1 Proposed ML malware detection method.

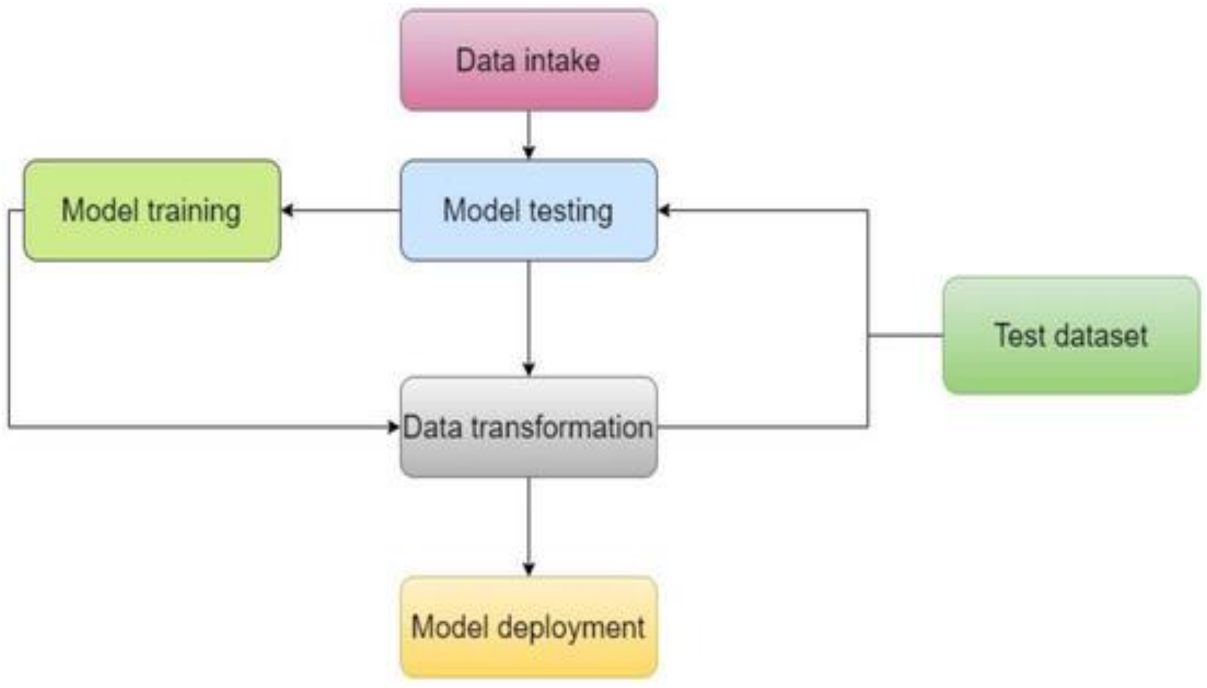


Figure 4.2 Workflow process illustration

4.1. Dataset

This study relied entirely on data provided by the Canadian Institute for Cybersecurity. The collection has many data files that include log data for various types of malware. These recovered log features may be used to train a broad variety of models. Approximate 51 distinct malware families were found in the samples. More than 17,394 data points from different locations were included; the dataset had 279 columns and 17,394 rows.

4.2. Pre-Processing

Data were stored in the file system as binary code, and the files themselves were unprocessed executables. We prepared them in advance of our research. Unpacking the executables required a protected environment, or virtual machine (VM). PEiD software automated unpacking of compressed executables.

4.3. Features Extraction

Twentieth-century datasets frequently contain tens of thousands of features. In recent years, as feature counts have grown, it has become clear that the resultant machine learning model has been overfit. To address this problem, we built a smaller set of features from a larger set; this technique is commonly used to maintain the same degree of accuracy while using fewer features. The goal of this study was to refine the existing dataset of dynamic and static features by keeping those that were most helpful and eliminating those that were not valuable for data analysis.

4.4. Features Selection

After completing feature extraction, which involved the discovery of more features, feature selection was performed. Feature selection was a crucial process for enhancing accuracy, simplifying the model, and reducing overfitting, as it involved choosing features from a pool of newly recognised qualities. Researchers have used many feature classification strategies in the past in an effort to identify dangerous code in software. As the feature rank technique is very effective at picking the right features for building malware detection models, it was extensively employed in this study.

CHAPTER 5

CODE SNIPPETS

CHAPTER 5

CODE SNIPPETS

learner.py

```
#! /usr/bin/python2
import pandas as pd #used for DATAFrames and DataFrames can hold different types data of multidimensional arrays.
import numpy as np#Numpy provides robust data structures for efficient computation of multi-dimensional arrays & matrices.
import pickle
import sklearn
import sklearn.ensemble as ske
from sklearn.model_selection import cross_validate
from sklearn import  tree, linear_model
from sklearn.feature_selection import SelectFromModel
import joblib
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import confusion_matrix
data = pd.read_csv('MalwareData.csv', sep='|') #generate df as data
X = data.drop(['Name', 'md5', 'legitimate'], axis=1).values #now droping some coloumns as axis 1(mean coloumn) and will show the values in the rows
y = data['legitimate'].values #values of legitimate data
import matplotlib.pyplot as plt
import seaborn as sns
print(data.plot())
print(plt.show())print(data.hist())

plt.xlabel("data")

plt.ylabel("d")

plt.show()

print('Researching important feature based on %i total features\n' % X.shape[1])# shape() is
use in pandas to give number of row/column


# Feature selection using Trees Classifier

fsel = ske.ExtraTreesClassifier().fit(X, y)

model = SelectFromModel(fsel, prefit=True)

X_new = model.transform(X)#now features are only 9 :)

nb_features = X_new.shape[1]#will save value 13 as shape is (138047, 13) :

X_train, X_test, y_train, y_test = sklearn.model_selection.train_test_split(X_new, y
,test_size=0.2)


#now converting in training and testing data in 20% range hahhahaha ! as total x is 138047
and testing is 138047*0.2=27610 :)

features = []

print('%i features identified as important:' % nb_features) #as mentioned above


#important features sored

indices = np.argsort(fsel.feature_importances_)[::-1][:nb_features]

for f in range(nb_features):

    print("%d. feature %s (%f)" % (f + 1, data.columns[2+indices[f]],
fsel.feature_importances_[indices[f]]))
```

```
# mean adding to the empty 'features' array the 'important features'
for f in sorted(np.argsort(fsel.feature_importances_)[::-1][:nb_features]):#[::-1] mean start
with last towards first
 features.append(data.columns[2+f])


#Algorithm comparison
algorithms = {
        "DecisionTree": tree.DecisionTreeClassifier(max_depth=10),
        #The max_depth parameter denotes maximum depth of the tree.
           "RandomForest": ske.RandomForestClassifier(n_estimators=50),#In case, of random
forest, these ensemble classifiers are            the randomly created decision trees. Each
decision tree is a single classifier and the target prediction is based on            the majority
voting method.
         #n_estimators ==The number of trees in the forest.
           "GradientBoosting": ske.GradientBoostingClassifier(n_estimators=50),
        "AdaBoost": ske.AdaBoostClassifier(n_estimators=100),


         #Ada mean Adaptive


         #Both are boosting algorithms which means that they convert a set of weak learners into
a single strong learner. They            both initialize a strong learner (usually a decision tree)
and iteratively create a weak learner that is added to the            strong learner. They differ on
how they create the weak learners during the iterative process.
           "GNB": GaussianNB()
        #Bayes theorem is based on conditional probability. The conditional probability helps us
calculating the probability          that something will happen
   }
results = {}
print("\nNow testing algorithms")
for algo in algorithms:
    clf = algorithms[algo]
    clf.fit(X_train, y_train)#fit may be called as 'trained'
    score = clf.score(X_test, y_test)
    print("%s : %f %%" % (algo, score*100))
    results[algo] = score
winner = max(results, key=results.get)
print('\nWinner algorithm is %s with a %f %% success' % (winner, results[winner]*100))

# Save the algorithm and the feature list for later predictions
print('Saving algorithm and feature list in classifier directory...')
joblib.dump(algorithms[winner], 'classifier/classifier.pkl')#Persist an arbitrary Python object
into one file.
open('classifier/features.pkl', 'wb').write(pickle.dumps(features))

#joblib works especially well with NumPy arrays which are used by sklearn so depending on
the classifier type you use you might have performance and size benefits using
joblib.Otherwise pickle does work correctly so saving a trained classifier and loading it again
will produce the same results no matter which of the serialization libraries you use
print('Saved')
```

```
# Identify false and true positive rates
clf = algorithms[winner]
res = clf.predict(X_test)
mt = confusion_matrix(y_test, res)

#A confusion matrix, also known as an error matrix,[4] is a specific table layout that allows
visualization of the performance of an algorithm, typically a supervised learning.
print("False positive rate : %f %%" % ((mt[0][1] / float(sum(mt[0])))*100))
print('False negative rate : %f %%' % ( (mt[1][0] / float(sum(mt[1]))*100)))
```

checker.py

```
#! /usr/bin/python2
#pefile is a Python module to read and work with PE (Portable Executable) files.
import pefile
import os
import array
import math
import pickle
import joblib
import sys
import argparse
import os, sys, shutil, time
import re
import pandas as pd
```

```
from flask import Flask, request, jsonify, render_template,abort,redirect,url_for
import werkzeug
import joblib
from sklearn.ensemble import RandomForestClassifier
def cutit(s,n):
   return s[n:]
app = Flask(__name__)
@app.route('/')
def home():
    return render_template('index.html')
@app.route('/uploader', methods = ['GET', 'POST'])
def upload_file():
   if request.method == 'POST':
    f = request.files['file']
        ##########################################
    # Load classifier
    clf =
joblib.load(os.path.join(os.path.dirname(os.path.realpath(__file__)),'classifier/classifier.pkl'))
    features =pickle.loads(open(os.path.join(os.path.dirname(os.path.realpath(__file__))
                                    ,'classifier/features.pkl'),'rb').read())
     ##########################################
     #tweet = request.form['tweet']
     #tweet=cutit(f.filename, 12)
    tweet =f.filename
    print(tweet)
    ##########################################
    data = extract_infos(tweet)
    pe_features = map(lambda x: data[x], features)
    pe_features = list(pe_features)

# Convert each element in the list to float
    pe_features = list(map(float, pe_features))
    res = clf.predict([pe_features])[0]
        ##########################################
     #print('The file %s is %s' % (os.path.basename(sys.argv[1]),['malicious',
'legitimate'][res]))
    return render_template('result.html', prediction = ['malicious', 'legitimate'][res])
```

#The phrase File Entropy is used to measure the amount of data which is present in a selected file. For example, if you have some files and desire to calculate the entropy value for that, then it will be very simple by accessing the methods of File Entropy and its calculation process.

```
def get_entropy(data):
    if len(data) == 0:
        return 0.0
    occurences = array.array('L', [0] * 256)
    for x in data:
        occurences[x if isinstance(x, int) else ord(x)] += 1
    entropy = 0
    for x in occurences:
        if x:
            p_x = float(x) / len(data)
            entropy -= p_x * math.log(p_x, 2)
    return entropy
def get_resources(pe):
    """Extract resources :
    [entropy, size]"""
    resources = []
    if hasattr(pe, 'DIRECTORY_ENTRY_RESOURCE'):
        try:
            for resource_type in pe.DIRECTORY_ENTRY_RESOURCE.entries:
                if hasattr(resource_type, 'directory'):
                    for resource_id in resource_type.directory.entries:
                        if hasattr(resource_id, 'directory'):
                            for resource_lang in resource_id.directory.entries:
                                data = pe.get_data(resource_lang.data.struct.OffsetToData,
                                                   resource_lang.data.struct.Size)
                                size = resource_lang.data.struct.Size
                                entropy = get_entropy(data)

                                resources.append([entropy, size])
        except Exception as e:
            return resources
    return resources
def get_version_info(pe):
    """Return version infos"""
    res = {}
    for fileinfo in pe.FileInfo:
        if fileinfo.Key == 'StringFileInfo':
            for st in fileinfo.StringTable:
                for entry in st.entries.items():
                    res[entry[0]] = entry[1]
        if fileinfo.Key == 'VarFileInfo':
            for var in fileinfo.Var:
                res[var.entry.items()[0][0]] = var.entry.items()[0][1]
    if hasattr(pe, 'VS_FIXEDFILEINFO'):
        res['flags'] = pe.VS_FIXEDFILEINFO.FileFlags
        res['os'] = pe.VS_FIXEDFILEINFO.FileOS
        res['type'] = pe.VS_FIXEDFILEINFO.FileType
        res['file_version'] = pe.VS_FIXEDFILEINFO.FileVersionLS
        res['product_version'] = pe.VS_FIXEDFILEINFO.ProductVersionLS
        res['signature'] = pe.VS_FIXEDFILEINFO.Signature
        res['struct_version'] = pe.VS_FIXEDFILEINFO.StrucVersion
    return res


def extract_infos(fpath):
    res = {}
    pe = pefile.PE(fpath)
    res['Machine'] = pe.FILE_HEADER.Machine
    res['SizeOfOptionalHeader'] = pe.FILE_HEADER.SizeOfOptionalHeader
    res['Characteristics'] = pe.FILE_HEADER.Characteristics
    res['MajorLinkerVersion'] = pe.OPTIONAL_HEADER.MajorLinkerVersion
    res['MinorLinkerVersion'] = pe.OPTIONAL_HEADER.MinorLinkerVersion
    res['SizeOfCode'] = pe.OPTIONAL_HEADER.SizeOfCode
    res['SizeOfInitializedData'] = pe.OPTIONAL_HEADER.SizeOfInitializedData
    res['SizeOfUninitializedData'] = pe.OPTIONAL_HEADER.SizeOfUninitializedData
    res['AddressOfEntryPoint'] = pe.OPTIONAL_HEADER.AddressOfEntryPoint
    res['BaseOfCode'] = pe.OPTIONAL_HEADER.BaseOfCode
    try:
        res['BaseOfData'] = pe.OPTIONAL_HEADER.BaseOfData
    except AttributeError:
        res['BaseOfData'] = 0
    res['ImageBase'] = pe.OPTIONAL_HEADER.ImageBase
    res['SectionAlignment'] = pe.OPTIONAL_HEADER.SectionAlignment
    res['FileAlignment'] = pe.OPTIONAL_HEADER.FileAlignment
    res['MajorOperatingSystemVersion'] =
pe.OPTIONAL_HEADER.MajorOperatingSystemVersion
    res['MinorOperatingSystemVersion'] =
pe.OPTIONAL_HEADER.MinorOperatingSystemVersion
    res['MajorImageVersion'] = pe.OPTIONAL_HEADER.MajorImageVersion
    res['MinorImageVersion'] = pe.OPTIONAL_HEADER.MinorImageVersion
    res['MajorSubsystemVersion'] = pe.OPTIONAL_HEADER.MajorSubsystemVersion
    res['MinorSubsystemVersion'] = pe.OPTIONAL_HEADER.MinorSubsystemVersion
    res['SizeOfImage'] = pe.OPTIONAL_HEADER.SizeOfImage
```

```
res['SizeOfHeaders'] = pe.OPTIONAL_HEADER.SizeOfHeaders
res['CheckSum'] = pe.OPTIONAL_HEADER.CheckSum
res['Subsystem'] = pe.OPTIONAL_HEADER.Subsystem
res['DllCharacteristics'] = pe.OPTIONAL_HEADER.DllCharacteristics
res['SizeOfStackReserve'] = pe.OPTIONAL_HEADER.SizeOfStackReserve
res['SizeOfStackCommit'] = pe.OPTIONAL_HEADER.SizeOfStackCommit
res['SizeOfHeapReserve'] = pe.OPTIONAL_HEADER.SizeOfHeapReserve
res['SizeOfHeapCommit'] = pe.OPTIONAL_HEADER.SizeOfHeapCommit
res['LoaderFlags'] = pe.OPTIONAL_HEADER.LoaderFlags
res['NumberOfRvaAndSizes'] = pe.OPTIONAL_HEADER.NumberOfRvaAndSizes

# Sections
res['SectionsNb'] = len(pe.sections)
entropy = list(map(lambda x: x.get_entropy(), pe.sections))
res['SectionsMeanEntropy'] = sum(entropy) / float(len(entropy))
res['SectionsMinEntropy'] = min(entropy)
res['SectionsMaxEntropy'] = max(entropy)
raw_sizes = list(map(lambda x: x.SizeOfRawData, pe.sections))
res['SectionsMeanRawsize'] = sum(raw_sizes) / float(len(raw_sizes))
res['SectionsMinRawsize'] = min(raw_sizes)
res['SectionsMaxRawsize'] = max(raw_sizes)
virtual_sizes = list(map(lambda x: x.Misc_VirtualSize, pe.sections))
res['SectionsMeanVirtualsize'] = sum(virtual_sizes) / float(len(virtual_sizes))
res['SectionsMinVirtualsize'] = min(virtual_sizes)
res['SectionMaxVirtualsize'] = max(virtual_sizes)
```

```
# Imports
try:
    res['ImportsNbDLL'] = len(pe.DIRECTORY_ENTRY_IMPORT)
    imports = sum([x.imports for x in pe.DIRECTORY_ENTRY_IMPORT], [])
    res['ImportsNb'] = len(imports)
    res['ImportsNbOrdinal'] = sum(1 for x in imports if x.name is None)
except AttributeError:
    res['ImportsNbDLL'] = 0
    res['ImportsNb'] = 0
    res['ImportsNbOrdinal'] = 0

# Exports
try:
    res['ExportNb'] = len(pe.DIRECTORY_ENTRY_EXPORT.symbols)
except AttributeError:
    # No export
    res['ExportNb'] = 0
# Resources
resources = get_resources(pe)
res['ResourcesNb'] = len(resources)
if len(resources) > 0:
    entropy = list(map(lambda x: x[0], resources))
    res['ResourcesMeanEntropy'] = sum(entropy) / float(len(entropy))
    res['ResourcesMinEntropy'] = min(entropy)
    res['ResourcesMaxEntropy'] = max(entropy)
```

```
        sizes = list(map(lambda x: x[1], resources))
        res['ResourcesMeanSize'] = sum(sizes) / float(len(sizes))
        res['ResourcesMinSize'] = min(sizes)
        res['ResourcesMaxSize'] = max(sizes)
    else:
        res['ResourcesNb'] = 0
        res['ResourcesMeanEntropy'] = 0
        res['ResourcesMinEntropy'] = 0
        res['ResourcesMaxEntropy'] = 0
        res['ResourcesMeanSize'] = 0
        res['ResourcesMinSize'] = 0
        res['ResourcesMaxSize'] = 0
    # Load configuration size
    try:
        res['LoadConfigurationSize'] = pe.DIRECTORY_ENTRY_LOAD_CONFIG.struct.Size
    except AttributeError:
        res['LoadConfigurationSize'] = 0
    # Version configuration size
    try:
        version_infos = get_version_info(pe)
        res['VersionInformationSize'] = len(version_infos.keys())
    except AttributeError:
        res['VersionInformationSize'] = 0
    return res
if __name__ == '__main__':  app.run(debug = True)
```

CHAPTER 6

SNAPSHOTS

CHAPTER 6

SNAPSHOTS

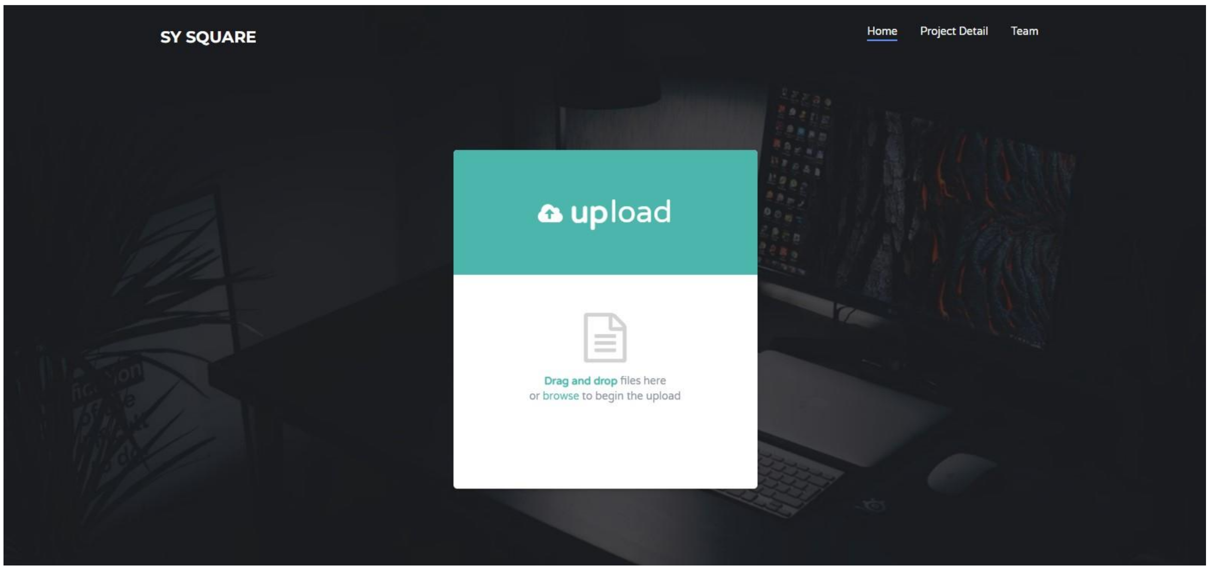
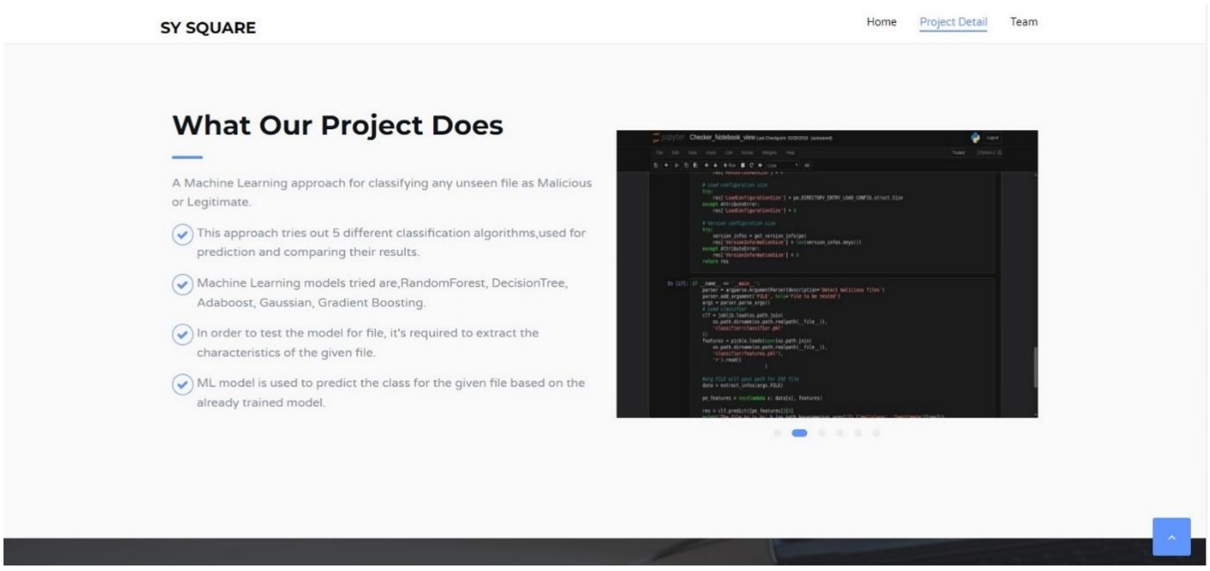


Figure 5.1 Homepage



What Our Project Does

A Machine Learning approach for classifying any unseen file as Malicious or Legitimate.

- ✓ This approach tries out 5 different classification algorithms,used for prediction and comparing their results.
- ✓ Machine Learning models tried are,RandomForest, DecisionTree, Adaboost, Gaussian, Gradient Boosting.
- ✓ In order to test the model for file, it's required to extract the characteristics of the given file.
- ✓ ML model is used to predict the class for the given file based on the already trained model.

```
# Load configuration size
try:
    res['LoadConfigurationSize'] = pe.DIRECTORY_ENTRY_LOAD_CONFIG.struct.Size
except AttributeError:
    res['LoadConfigurationSize'] = 0

# Version configuration size
try:
    version_infos = get_version_info(pe)
    res['VersionInformationSize'] = len(version_infos.keys())
except AttributeError:
    res['VersionInformationSize'] = 0
return res

In [17]: if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Detect malicious files')
    parser.add_argument('FILE', help='File to be tested')
    args = parser.parse_args()
    # Load classifier
    clf = joblib.load(os.path.join(
        os.path.dirname(os.path.realpath(__file__)),
        'classifier/classifier.pkl'
    ))
    features = pickle.loads(open(os.path.join(
        os.path.dirname(os.path.realpath(__file__)),
        'classifier/features.pkl'),
        'r').read()
                )

    #arg.FILE will pass path for EXE file
    data = extract_infos(args.FILE)

    pe_features = map(lambda x: data[x], features)

    res = clf.predict([pe_features])[0]
```

Figure 5.2 Program Detail

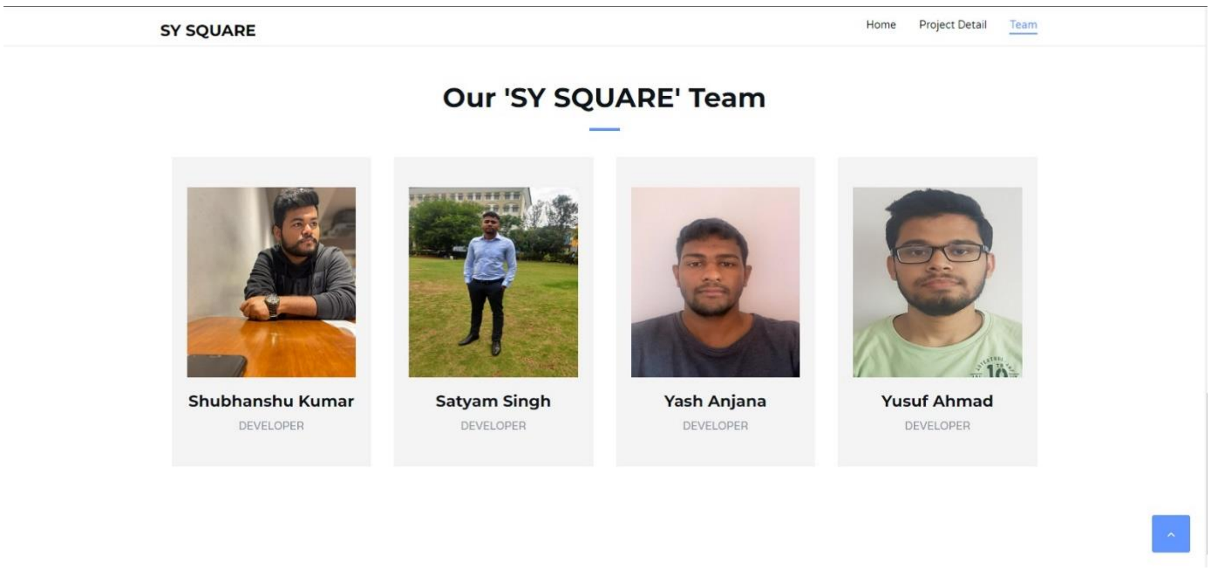


Figure 5.3 Our Team

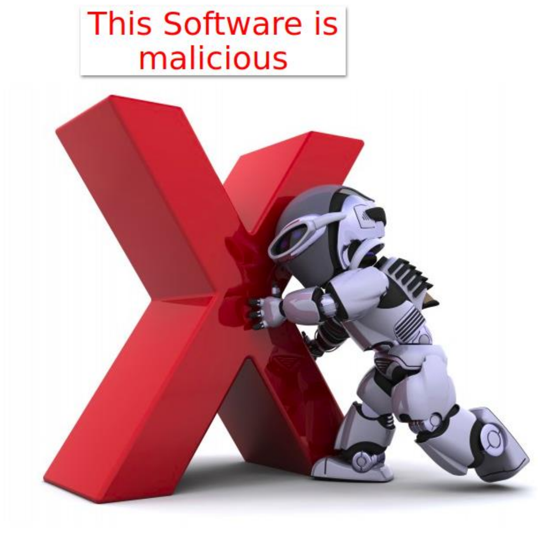


Figure 5.4 Malware Found

Figure 5.5 Malware Not Found

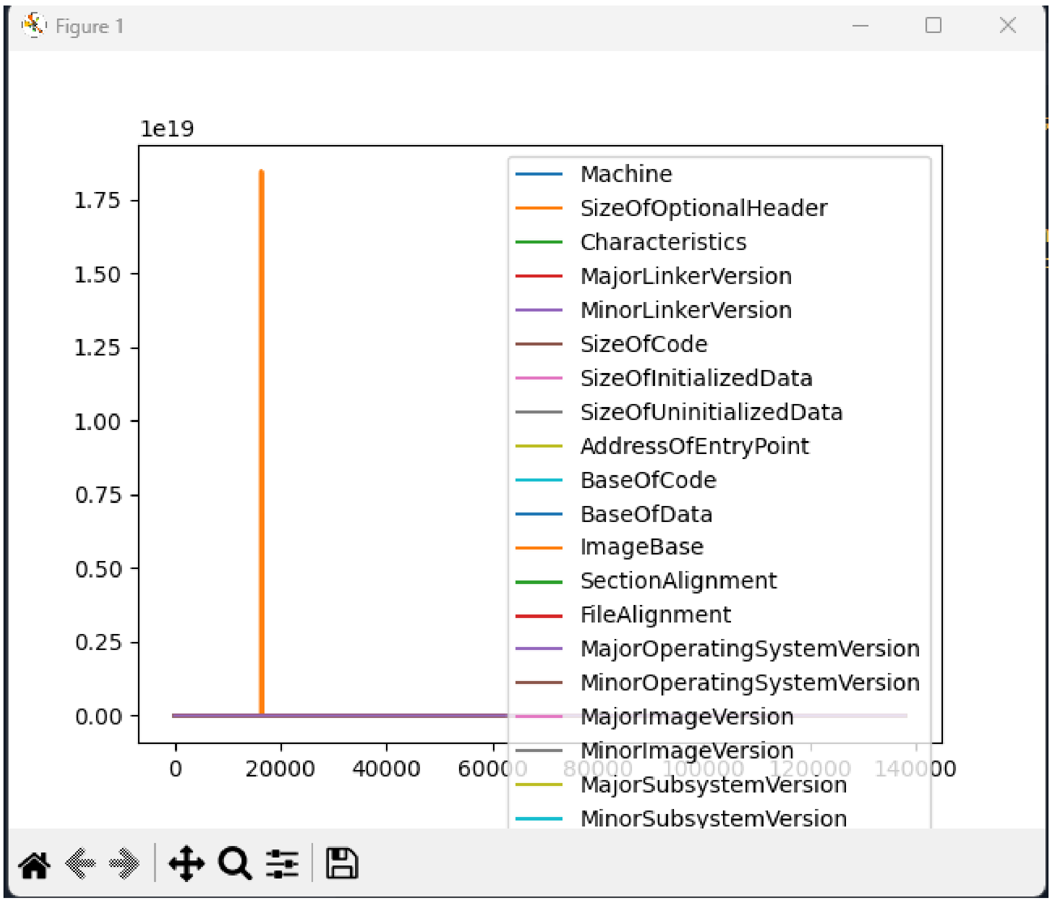


Figure 5.6 Graph

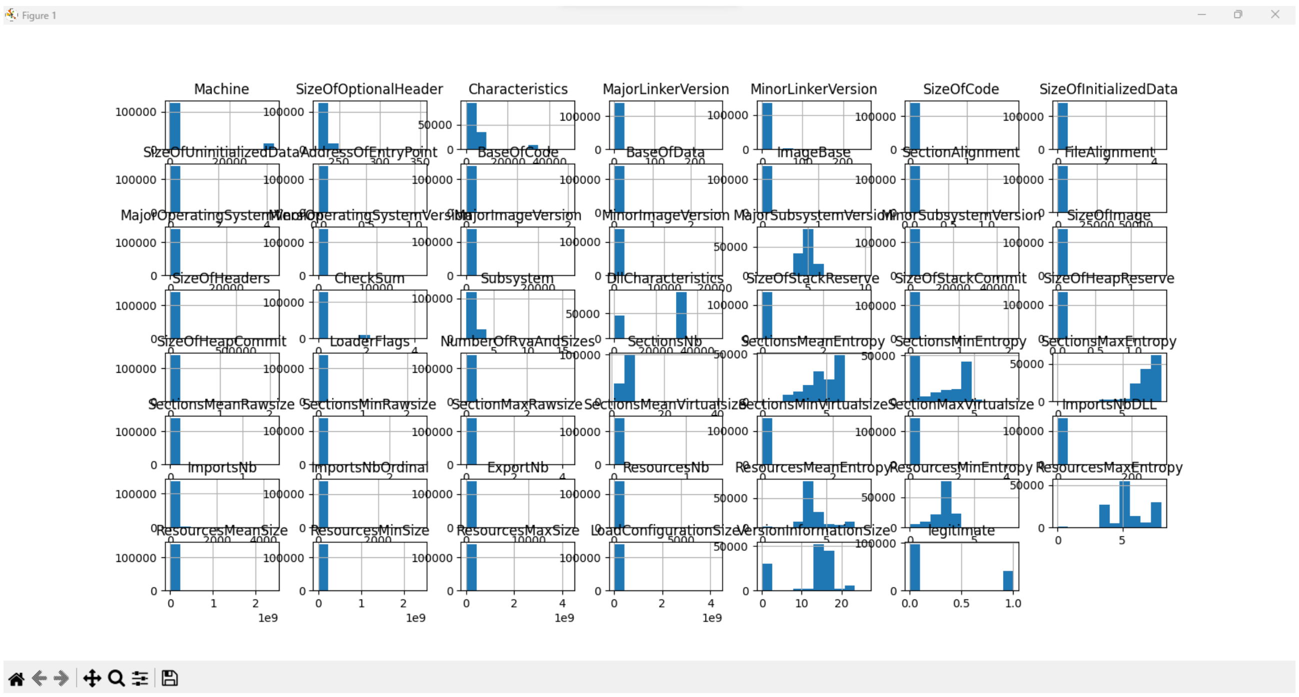
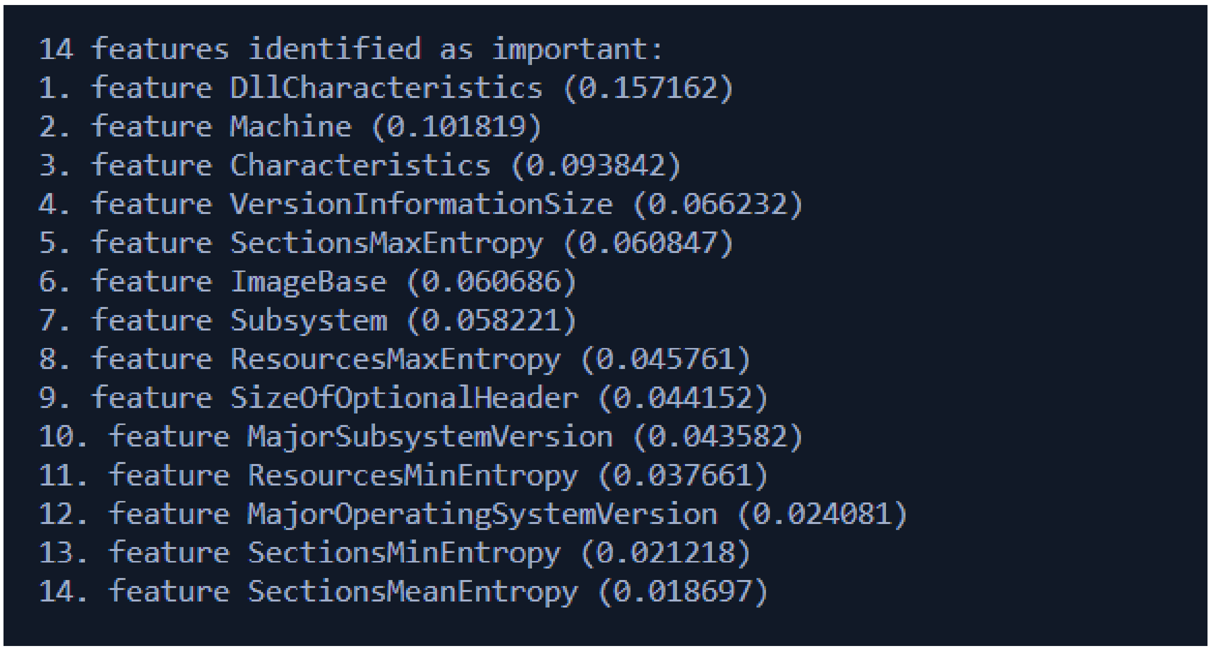
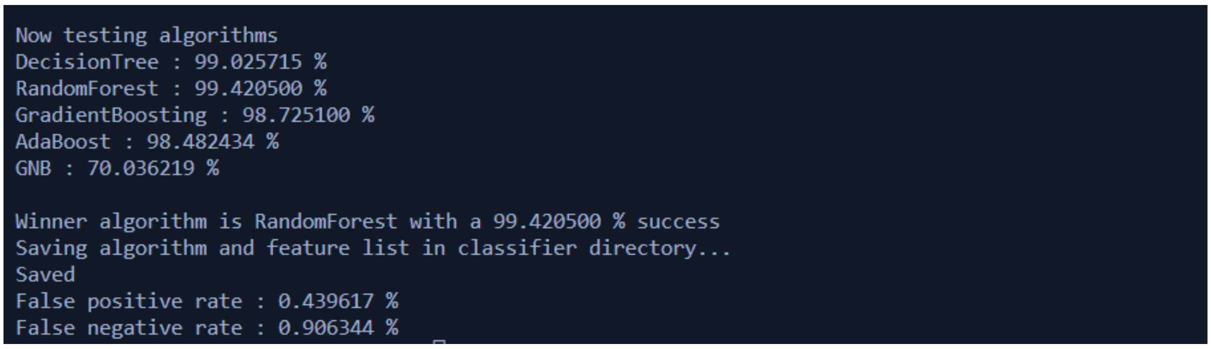


Figure 5.7 Histogram



```
14 features identified as important:
1. feature DllCharacteristics (0.157162)
2. feature Machine (0.101819)
3. feature Characteristics (0.093842)
4. feature VersionInformationSize (0.066232)
5. feature SectionsMaxEntropy (0.060847)
6. feature ImageBase (0.060686)
7. feature Subsystem (0.058221)
8. feature ResourcesMaxEntropy (0.045761)
9. feature SizeOfOptionalHeader (0.044152)
10. feature MajorSubsystemVersion (0.043582)
11. feature ResourcesMinEntropy (0.037661)
12. feature MajorOperatingSystemVersion (0.024081)
13. feature SectionsMinEntropy (0.021218)
14. feature SectionsMeanEntropy (0.018697)
```

Figure 5.8 Features identified as important



```
Now testing algorithms
DecisionTree : 99.025715 %
RandomForest : 99.420500 %
GradientBoosting : 98.725100 %
AdaBoost : 98.482434 %
GNB : 70.036219 %

Winner algorithm is RandomForest with a 99.420500 % success
Saving algorithm and feature list in classifier directory...
Saved
False positive rate : 0.439617 %
False negative rate : 0.906344 %
```

Figure 5.9 Test Results

CHAPTER 7

CONCLUSION

CHAPTER 7

CONCLUSION

The study presented a protective mechanism that evaluated four machine learning (ML) algorithm approaches to malware detection and selected the most appropriate one. The results showed that compared to other classifiers, decision tree (DT) (99.1%), Random Forest (99.42%), Gradient Boosting (98.8%), AdaBoost (98.56%) and (GNB) (70.54%) performed well in terms of detection accuracy. The performance of DT, Random Forest, and Gradient boosting algorithms in detecting malware on a small false positive rate (FPR) (DT = 2.01%, Random Forest = 0.43%, and Gradient Boosting = 4.63%) in a given dataset was compared. The study evaluated and quantified the detection accuracy of an ML classifier that used static analysis to extract features based on PE data and compared it with two other ML classifiers.

The Random Forest machine learning method had the highest accuracy (99.42%) among all the classifiers evaluated. In addition to potentially providing the highest detection accuracy and accurately characterizing malware, static analysis based on PE information and carefully selected data showed promise in experimental findings. The significant benefit of this approach is that it does not require executing anything to determine if data are malicious. Overall, the study highlights the importance of using machine learning techniques in malware detection and characterisation. The results demonstrate that DT, Random Forest, and Gradient Boosting algorithms can effectively identify dangerous versus benign data, and static analysis based on PE information and carefully selected data can improve malware detection accuracy.

FUTURE ENHANCEMENT

Though the proposed application has covered various aspects, we will add a few more in future. This can be made more accurate with adding more data set and further on adding more algorithms with better performance can add on to the accuracy of the model. It can also be hosted on the web for real time analysis of files on the cloud.

CHAPTER 8

REFERENCES

CHAPTER 8

REFERENCES

1. Nikam, U.V.; Deshmuh, V.M. Performance evaluation of machine learning classifiers in malware detection. In Proceedings of the 2022 IEEE International Conference on Distributed Computing and Electrical Circuits and Electronics (ICDCECE), Ballari, India, 23–24 April 2022; pp. 1–5.
2. Akhtar, M.S.; Feng, T. IOTA based anomaly detection machine learning in mobile sensing. EAI Endorsed Trans. Create. Tech. 2022, 9, 172814.
3. Sethi, K.; Kumar, R.; Sethi, L.; Bera, P.; Patra, P.K. A novel machine learning based malware detection and classification framework. In Proceedings of the 2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security), Oxford, UK, 3–4 June 2019; pp. 1–13.
4. Abdulbasit, A.; Darem, F.A.G.; Al-Hashmi, A.A.; Abawajy, J.H.; Alanazi, S.M.; Al-Rezami, A.Y. An adaptive behavioral-based increamental batch learning malware variants detection model using concept drift detection and sequential deep learning. IEEE Access 2021, 9, 97180–97196.
5. Feng, T.; Akhtar, M.S.; Zhang, J. The future of artificial intelligence in cybersecurity: A comprehensive survey. EAI Endorsed Trans. Create. Tech. 2021, 8, 170285.
6. Sharma, S.; Krishna, C.R.; Sahay, S.K. Detection of advanced malware by machine learning techniques. In Proceedings of the SoCTA 2017, Jhansi, India, 22–24 December 2017.
7. Chandrakala, D.; Sait, A.; Kiruthika, J.; Nivetha, R. Detection and classification of malware. In Proceedings of the 2021 International Conference on Advancements in Electrical, Electronics, Communication, Computing and Automation (ICAECA), Coimbatore, India, 8–9 October 2021; pp. 1–3.
8. Zhao, K.; Zhang, D.; Su, X.; Li, W. Fest: A feature extraction and selection tool for android malware detection. In Proceedings of the 2015 IEEE Symposium on Computers and Communication (ISCC), Larnaca, Cyprus, 6–9 July 2015; pp. 714–720.
9. Akhtar, M.S.; Feng, T. Detection of sleep paralysis by using IoT based device and its relationship between sleep paralysis and sleep quality. EAI Endorsed Trans. Internet Things 2022, 8, e4.

10. Gibert, D.; Mateu, C.; Planes, J.; Vicens, R. Using convolutional neural networks for classification of malware represented as images. J. Comput. Virol. Hacking Tech. 2019, 15, 15–28.
11. Firdaus, A.; Anuar, N.B.; Karim, A.; Faizal, M.; Razak, A. Discovering optimal features using static analysis and a genetic search based method for Android malware detection. Front. Inf. Technol. Electron. Eng. 2018, 19, 712–736.
12. Dahl, G.E.; Stokes, J.W.; Deng, L.; Yu, D.; Research, M. Large-scale Malware Classification Using Random Projections And Neural Networks. In Proceedings of the International Conference on Acoustics, Speech and Signal Processing-1988, Vancouver, BC, Canada, 26–31 May 2013; pp. 3422–3426.
13. Akhtar, M.S.; Feng, T. An overview of the applications of artificial intelligence in cybersecurity. EAI Endorsed Trans. Create. Tech. 2021, 8, e4.
14. Akhtar, M.S.; Feng, T. A systemic security and privacy review: Attacks and prevention mechanisms over IOT layers. EAI Endorsed Trans. Secur. Saf. 2022, 8, e5.
15. Anderson, B.; Storlie, C.; Lane, T. "Improving Malware Classification: Bridging the Static/Dynamic Gap. In Proceedings of the 5th ACM Workshop on Security and Artificial Intelligence (AISec), Raleigh, NC, USA, 19 October 2012; pp. 3–14.
16. Varma, P.R.K.; Raj, K.P.; Raju, K.V.S. Android mobile security by detecting and classification of malware based on permissions using machine learning algorithms. In Proceedings of the 2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), Palladam, India, 10–11 February 2017; pp. 294–299.
17. Akhtar, M.S.; Feng, T. Comparison of classification model for the detection of cyber-attack using ensemble learning models. EAI Endorsed Trans. Scalable Inf. Syst. 2022, 9, 17329.
18. Rosmansyah, W.Y.; Dabarsyah, B. Malware detection on Android smartphones using API class and machine learning. In Proceedings of the 2015 International Conference on Electrical Engineering and Informatics (ICEEI), Denpasar, Indonesia, 10–11 August 2015; pp. 294–297.
19. Tahtaci, B.; Canbay, B. Android Malware Detection Using Machine Learning. In Proceedings of the 2020 Innovations in Intelligent Systems and Applications Conference (ASYU), Istanbul, Turkey, 15–17 October 2020; pp. 1–6.
20. Baset, M. Machine Learning for Malware Detection. Master's Dissertation, HeriotWatt University, Edinburg, Scotland, December
2016.