Analysis of Malware Prediction Based on Infection Rate Using Machine Learning Techniques

by

Safir Zawad
19241038
Raiyan Mansur
19241037
Nahian Evan
19241036
Ashub Bin Asad
15301062

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science

Department of Computer Science and Engineering
BRAC University
December 2019

Declaration

It is hereby declared that

1. The thesis submitted is our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

Safir Zawad 19241038	Raiyan Mansur 19241037
Nahian Evan 19241036	Ashub Bin Asad 15301062

Approval

The thesis/project titled "Analysis of Malware Prediction Based on Infection Rate Using Machine Learning Techniques" submitted by

1. Safir Zawad (19241038)
2. Raiyan Mansur (19241037)
3. Nahian Evan (19241036)
4. Ashub Bin Asad (15301062)

Of Fall, 2019 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc.in Computer Science on December10, 2019.

Examining Committee:

Supervisor:
(Member)

Muhammad Iqbal Hossain, PhD
Assistant Professor
Department of Computer Science and Engineering
BRAC University

Program Coordinator:
(Member)

Md. Golam Rabiul Alam, PhD
Associate Professor
Department of Computer Science and Engineering
BRAC University

Head of Department:
(Chair)

Mahbubul Alam Majumdar, PhD
Professor and Chairperson
Department of Computer Science and Engineering
BRAC University

Abstract

In this modern, technological age, the internet has been adopted by the masses. And with it, the danger of malicious attacks by cybercriminals have increased. These attacks are done via Malware, and have resulted in billions of dollars of financial damage. Which is why prevention of malware attacks has become an essential part of the battle against cybercrime. In recent years, Machine Learning has become an important tool in the field of Malware Detection, which is the first step towards removing malware from infected devices. In this thesis, we are applying machine learning algorithms to predict the malware infection rates of computers based on its features. We are using supervised machine learning algorithms and gradient boosting algorithms, such as LightGBM, Neural Networks, and Decision Tree Learning. We have collected a publicly available dataset, which was divided into two parts, one being the training set, and the other will be the testing set. After conducting four different experiments using the aforementioned algorithms, it has been discovered that LightGBM is the best model with an AUC Score of 0.73926.

Keywords: LightGBM, Neural Networks, Decision Tree Learning

Dedication

We would like to dedicate this thesis to our loving parents.

Acknowledgement

We want to dedicate our acknowledgement of gratitude to our thesis supervisor Dr. Muhammad Iqbal Hossain, PhD, Assistant Professor, Department of Computer Science and Engineering of BRAC University for his guidance for the completion of our thesis. We are thankful to CSE department, BRAC University for providing us the necessary equipment for the completion of this project.

Table of Contents

List of Figures

List of Tables

Nomenclature

The next list describes several symbols & abbreviation that will be later used within the body of the document

AUC	Area under the ROC Curve
$CART$	Classification and Regression Tree
$CHAID$	CHi- squared Automatic Interaction Detector
CNN	Convolutional Neural Network
$Cos - p$	Cascade One-Sided Perceptron
$CVScore$	Cross Validation Score
DLL	Dynamic Link Library
$Dtype$	Data Types
ESM	Enterprise Storage Management
FN	False Negative
FP	False Positive
FPR	False Positive Rate
GPU	Graphics Processing Unit
$ID3$	Iterative Dichotomiser 3
$LGBM$	Light Gradient Boosting Model
MLP	Multi-layer Perceptron
$OPCODE$	Operation Code
PE	Portable Executable
RF	Random Forest
ROC	Receiver Operating Characteristic
SVM	Support Vector Machine
TN	True Negative

TP	True Positive
TPR	True Positive Rate
USD	United States Dollar
$XGBoost$	eXtreme Gradient Boosting

Chapter 1

Introduction

1.1 Overview

Malware, or malicious software, is software created to infect a machine without the user's knowledge or consent. It is actually a generic definition for all sorts of threats that can affect a computer. A simple classication of malware consists of le infectors and stand-alone malware. The objectives of a malware could include accessing private networks, stealing sensitive data, taking over computer systems to make use of its resources, or disrupting computing or communication operations. Malware is deliberately malevolent, even when camouflaged as genuine software from a seemingly reputable source in such a way that most of them have the ability to spread itself within the network, remain undetectable, cause changes or damage to the infected system or network. Other objectives include service disruption targeting specific organizations, identity theft, or even cyberwarfare by targeting another nation state. The victims can just as easily be entrepreneurs, enterprises, large corporations, or even governments. Common types of malwares includes Worms, Trojan or Trojan horse, Adware, Ransomware etc [1] [2].

1.2 Motivation

Although significant strides have been made in the field of malware prediction, prevention, and cybersecurity in general, it appears that cybercriminals are always a step ahead of the curve. The volume and intensity of cyberattacks have only been increasing with increased dependency on the internet by individuals and organizations alike. It is projected that by 2021, global costs related to cybercrime is going to hit 6 trillion USD [3]. The future does not look bright for mankind in its battle against malware, but with so many bright minds working on cybersecurity, the damages inflicted by malware might be dampened in the near future [3].

1.3 Problem Statement

In this day and age, cybercrime is one of the biggest threats to humankind, as it causes personal and financial damage to both individuals and organizations. And the biggest tool in the hands of a cybercriminal is Malware. Through malware, Cybercriminals gain access to valuable personal data by attacking organizations that hold such data. Data breaches end up costing organizations a lot of money, and the fear of future breaches have resulted in an increase in the cost of cybersecurity [3].

At first, malware was spread mostly through CD-ROMs and Floppy Disks, which limited the damage caused by malware. However, the advent of the internet, and its rapid adoption across the globe has seen the damage caused by malware grow exponentially. And the extent of the damage caused by malware has not been more visible than in the last decade. Some of the biggest data breaches have occurred in the last decade. This increase in the sheer volume of cyberattacks via malware has resulted in a huge increase in spending on cybersecurity by organizations across the globe. 2017 saw an increase of 23 percent in cybercrime costs per organization from 2016, with the average cost per organization standing at 11.7 million USD [3]. Some malware attacks were so severe that they shut down a whole organization as we can see in 2016 Lincolnshire County Council had to shut down as a result of zero-day attack [4].

1.4 Objective

The objective of our research is to identify the model that provides the highest accuracy in predicting if individual machines would be infected by malware. It is done by taking into account the different properties of the machines, especially ones related to the machine's malware protection status, and user habits. Extensive research on Malware Prediction would enable users of these machines to be prepared beforehand for malicious attacks on their computers, and this would result in billions of dollars in savings for businesses and personal computer users throughout the world.

1.5 Thesis Structure

In Chapter 2 we have discussed about different algorithms and the algorithms we used in our research for malware prediction. In Chapter 3 we have described our dataset, a chart describing feature importance, Chi-squared test diagrams, correlation heatmap and our proposed model description. In Chapter 4 we have showed the experiments done on malware prediction using LightGBM and K-Fold Cross Validation. Next comes Chapter 5 where we have evaluated our research using AUC score. Finally, the paper concludes with a hypothesis, the difficulties we faced during our work and a few remarks.

Chapter 2

Background

2.1 Literature Review

The writers of the paper proposed a framework where multiple machine learning algorithms were used to differentiate clean files from malware files, while minimizing the number of false positives [5]. The researchers planned on working around mainly two algorithms, the cascade one-sided perceptron, and the cascade kernelized one-sided perceptron. Three datasets were used, a training dataset, a medium sized testing dataset, and an extremely large 'scaled up' dataset containing over millions of files to test out the algorithms. The datasets contained a mixture of multiple malwares, including Backdoor, Hacktool, Rootkit, Worm, Trojan, and other malwares. The main goal of the researchers was to create a framework of machine learning algorithms that detect as many malware files as possible, with a zero false positive rate. The researchers were very close to their goal, but still had a non-zero false positive rate. They concluded that malware detection using machine learning will not be replacing other techniques, but can be used in conjunction with the other techniques used at that time via the addition of some deterministic exceptions to make it fit for commercial usage. The most effective algorithms at detecting malware were found to be the cascade one-sided perceptron (COS-P) and its mapped variant (COS-P-Map). This paper was published at a time when the idea of applying machine learning algorithms in malware detection was in its infancy.
The authors of the paper were researching the use of predicting if a file is malicious or not based on a short snippet of data [6]. They tried out multiple neural networks and were able to predict if a file was malware or benign in less than 5 seconds of the execution of the file, achieving 94% accuracy. This was the first time malware prediction was done on files while it was being executed. The norm is to use an activity from executable files after it was executed. As a result, this made the advancement of endpoint security with the use of past behavioral data, instead of detecting them after they have done damage.
The paper mentions that malware detection with the help of machine learning will help detect new king of malwares and prevent users from future attacks where traditional malware detection can only detect known malwares [7]. There are two kinds of malware analysis through machine learning. Static analysis requires examining the executable file without running the file. By analyzing the portable executable headers and sections and provide a good insight about the file's functionality. This procedure is mainly used by the antivirus companies. They analyze malwares and

store the data in their systems and analyze any suspicious software for similar pattern. If there is a match that file is considered as a malware. Dynamic analysis is running the malware in a virtual safe environment and closely examining the activities. A typical machine learning experiment in malware analysis starts with collecting a dataset of malicious executables. Then the dataset is divided into training and testing sets. One is to train the model and the other one is to test the model for unknown occurrences. Larger dataset provides more accurate results. Then the desired static features are extracted for static analysis like String n-grams, Byte sequence n-grams, PE headers, DLL libraries, OpCode etc. Different machine learning algorithms are used like Inductive rule based model, Probability based model, Multi Naïve Bayes model.
In the research paper, flow based malwares were detected using machine learning algorithms such as Convolutional Neural Network (CNN) and Random Forest (RF) [8]. In that paper, data from public data from stratosphere IPS were used. Those malwares were flow based which means malwares have various port numbers and protocols which makes them harder to detect. In the era of internet this kind of malware detection is imported and machine learning algorithms were used to detect them with higher percentage of accuracy. Many machine learning algorithms such as Convolutional Neural Network (CNN), Multi-layer Perceptron (MLP), Support vector machine (SVM) and Random Forest (RF) were applied for classification. Those were applied in 9 different malware packet data. The appliance resulted in 85% of accuracy using CNN and RF.
In the paper, the researchers used data from Symantec containing data from more than 1.4 million machines and 50 different types of malware which was collected for 2 years and from multiple nations [9]. Features were derived from both malware and devices. A non-linear model was also designed for malware detection, with the help of epidemiological and information diffusion models. The researchers showcased a new ensemble-based method, called ESM, which results in a more precise algorithm. With data collected from numerous experiments containing various malware and data from different nations, which shows that ESM has predicted infection ratios four times better than other baseline models. It even works when the number of detections is very few.

2.2 Algorithms

There are many algorithms for malware detection and prediction. Among them Decision Tree, Recurrent Neural Network, and Light Gradient Boosting Model (LightGBM) are commonly used and useful algorithms for malware detection and prediction. There has been a lot of research done on Malware Detection, but not much has been done on the Prediction of Malware Infection Rates on machines as of now. The biggest difference between Malware Detection and Prediction is that, in Malware Detection, individual files are checked and it is determined if the file is a Malware or if it is Benign. However, the prediction of malware infection rates in machines works with whole machines, and the objective is to determine using certain given metrics if the machine is vulnerable to future malware attacks. When detecting malware, you can either be right or wrong, but when predicting if malware will infect your machine, the quality of your result is very important as well.

2.2.1 LightGBM

In our thesis, we have mostly worked with the LightGBM algorithm, which is fairly new, but has many useful applications in Malware Prediction. LightGBM is a gradient boosting framework that uses based tree learning algorithms. Compared to other algorithms where trees grow horizontally, meaning it grows level-wise, LightGBM trees grow vertically, which means that it grows leaf-wise, as shown in figure 2.1. This results in a reduced amount of loss compared to a level-wise algorithm. The part where LightGBM shines is in the case of massive datasets, like the one used in our implementation. The light in LightGBM comes from the fact that it runs at a very high speed, and consumes a lot less memory. It also has support for GPU learning, which is a must in this day and age [10][11].

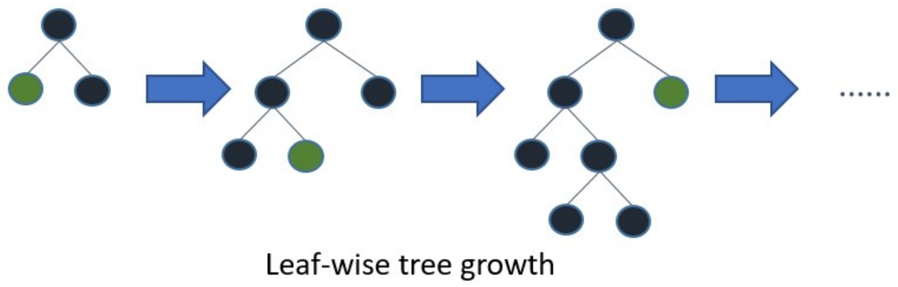


Figure 2.1: Leaf-wise Tree Growth in LightGBM

LightGBM works by first processing the dataset and making it lighter to make full, efficient use of the gradient boosting framework. Then the lighter pre-processed dataset is put into the algorithm, where it is run at light speed.

2.2.2 Neural Network

Neural Network is distinct from other algorithms for its characteristics which consists of learning from example, distributed associative memory, fault tolerance and pattern recognition [12]. Neural Network algorithm works by recognizing a pattern and the pattern is taken as numerical data or vectors. The data is given a weight and and predict a value and then compares with the ground truth to find the error. After finding the error this algorithm goes back to the error of a given model to find adjustment [13].
For a single layer of calculation, algorithm first takes an input for calculation. The input then gets multiplied by the slope parameter. Then bias is added with it. The bias indicates the layer of neural network. Then sigmoid function is applied to the product and thus we get the predicted probability. Now, in neural network there can be multiple layers. So, the general rule for calculation is to take a weight for each neuron. So the weights get multiplied by its connected input and all the products are added together with the bias. And finally sigmoid function is applied like before for finding prediction.

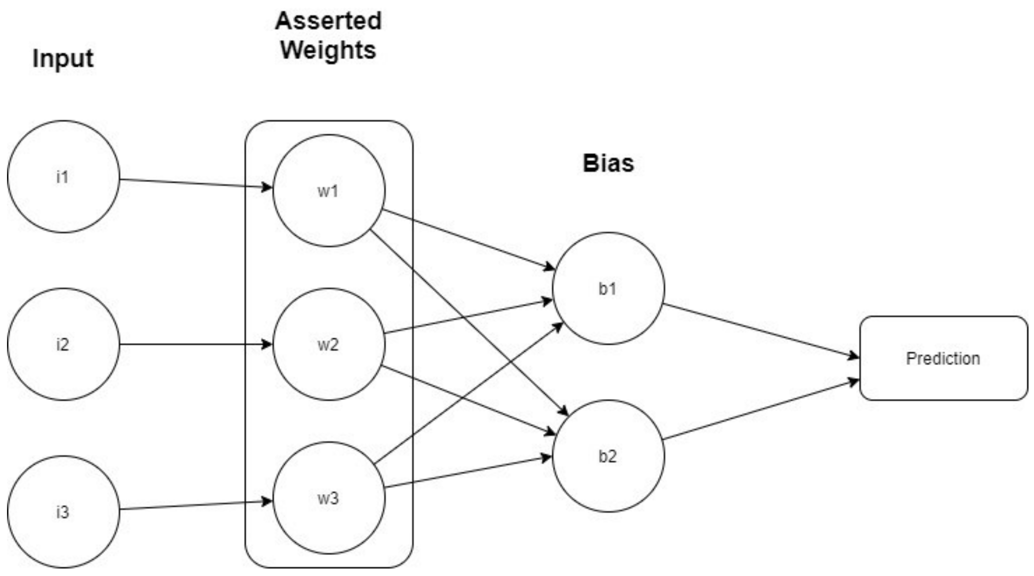


Figure 2.2: Neural Network Prediction Method

Fig 2.2 shows the demonstration of calculation for prediction in neural network. From this the algorithm gets its prediction. As said earlier, this algorithm compares the prediction with actual value to find the error. The error gets multiplied by the contribution of weights for error to get the adjustment.
So this algorithm evaluates the error and adjustment which results in fair and accurate prediction making it preferable for our dataset.

2.2.3 Decision Tree Learning

The Decision Tree algorithm uses a tree like graph of decisions with possible outcomes for calculation [14]. There are many classification algorithms for Decision tree for example ID3(Iterative Dichotomiser 3), C4.5, CART(Classification and Regression Tree), CHAID(CHi- squared Automatic Interaction Detector) [15]. Amongst these ID3 algorithm is the most popular. This algorithm creates a decision tree using top-down greedy approach and uses the strategy of learning from examples [16]. Making the tree it takes the best decision and select attribute and uses the decision attribute as node and finishes after using all the attributes. After that it calculates information gain by using the uncertainty of the variables also known as entropy [17]. CART(Classification and Regression Tree) uses Gini Index to construct the classification tree by splitting the attribute. Using these splits it can handle missing values. The calculation for this algorithm takes some attributes such as Entropy, Information Gain, Gini Index etc. Entropy measures the uncertainty of a variable.

$$E(S) = \sum_{i=1}^{c} -p_i log_2 p_i \tag{2.1}$$

It takes the probability of an element(p_i) and calculates for its randomness and for every element it does the same. The summation of all the calculation is ultimately the Entropy. It increases if the randomness increases.

Information Gain shows the difference between the given data and gained information.

$$IG(Y, X) = E(Y) - E(Y|X) \tag{2.2}$$

It is the difference between entropy of data portion which is given(Y) and entropy of attribute given the data portion$(Y|X)$.
Gini index is used in CART algorithm. It measures the impurity of data elements.

$$Gini = 1 - \sum_{i=1}^{c} (p_i)^2 \tag{2.3}$$

It is calculated by taking the sum of squared probabilities of every class or element(p_i) and then subtract that by one.

Chapter 3

Database and Experimental Setup

3.1 Dataset description

The dataset that has been used for our project is the Microsoft Malware Prediction Dataset that has been used in the Microsoft Malware Prediction Competition posted on Kaggle this year. The goal of the competition was to predict the probability of a Windows machine being infected by various types of malware, based on the different properties of the machine, so that infected machines can be identified quickly and be cured in time [18].
The size of the dataset is massive, with the training dataset named 'train.csv' containing 9 million rows, and the testing dataset named 'test.csv' containing 8 million rows. There are 82 features contained in the dataset, with most being categorical, of which 23 are numerically encoded to protect the privacy of the information contained in the dataset. The variable type pie chart is given in Table 3.1 below.

	Feature	Unique_values	Percentage of missing values	Percentage of values in the biggest category	Type
28	PuaMode	2	99.974119	99.974119	category
41	Census_ProcessorClass	3	99.589407	99.589407	category
8	DefaultBrowsersIdentifier	1730	95.141637	95.141637	float16
68	Census_IsFlightingInternal	2	83.04403	83.04403	float16
52	Census_InternalBatteryType	78	71.046809	71.046809	category
71	Census_ThresholdOptIn	2	63.524472	63.524472	float16
75	Census_IsWIMBootEnabled	2	63.439038	63.439038	float16
31	SmartScreen	21	35.610795	48.379658	category
15	OrganizationIdentifier	49	30.841487	47.037662	float16
29	SMode	2	6.027686	93.928812	float16
14	CityIdentifier	107366	3.647477	3.647477	float32
80	Wdft_IsGamer	2	3.401352	69.205344	float16
81	Wdft_RegionIdentifier	15	3.401352	20.177195	float16
53	Census_InternalBatteryNumberOfCharges	41087	3.012448	56.643094	float32
72	Census_FirmwareManufacturerIdentifier	712	2.054109	30.253692	float16
69	Census_IsFlightsDisabled	2	1.799286	98.199728	float16
73	Census_FirmwareVersionIdentifier	50494	1.794915	1.794915	float32
37	Census_OEMModelIdentifier	175365	1.145919	3.416271	float32
36	Census_OEMNameIdentifier	2564	1.070203	14.428946	float16
32	Firewall	2	1.023933	96.856251	float16
46	Census_TotalPhysicalRAM	3446	0.902686	45.894971	float32
79	Census_IsAlwaysOnAlwaysConnectedCapable	2	0.799676	93.50432	float16
62	Census_OSInstallLanguageIdentifier	39	0.673475	35.636026	float16
30	IeVerIdentifier	303	0.660137	43.55601	float16

Table 3.1: Feature Table for the Dataset

3.1.1 Data Preprocessing

The dataset contained some columns that had mostly missing values, and these columns were dropped. There were 26 columns that had mostly one category in it, as in, 90 percent of it was just one category. These columns were also dropped and excluded from training and testing. The use of dtypes resulted in the reduction in size of data loaded. This was done by converting a certain type of data into a data type of smaller size [19].
For the Kernels with LGBM, frequency encoding was required to fit the database with the gradient boosting algorithm. Frequency encoding was done for variables with large cardinality, where encoding was done efficiently with categories being ranked with respect to their frequency in the dataset. After that, the encoded variables were considered as numerical during experimentation.

3.1.2 Feature Importances

Here are some Graphs and Charts displaying which features had the most importance in the construction of the LGBM boosting framework model used in our experimentation. Figure 3.1 shows that the most important features when constructing the gradient boosting model are 'Census_OSVersion', 'SmartScreen', 'OsBuildLab', 'AppVersion', and 'EngineVersion'.

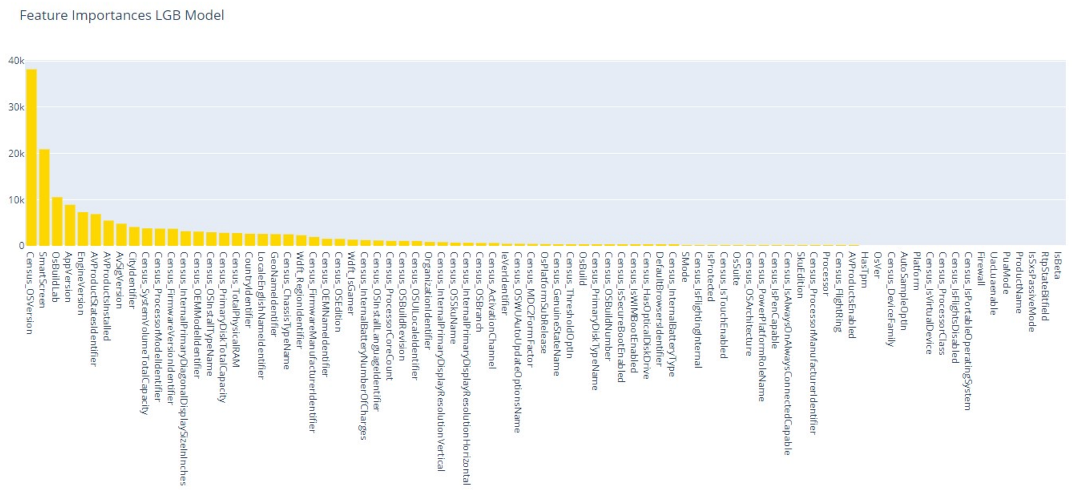


Figure 3.1: LightGBM Feature Importance Chart

3.1.3 Chi-squared Test

A chi-squared test was used to test the relationship between different, independent categorical variables. The purpose of the test is to see if the different categorical values in the dataset are independent of each other, as in, they are mutually exclusive. This is done to ensure a well done goodness of fit test, which shows how well the distribution of the columns in the dataset is. Figure 3.2 shows the Crosstab for 'OsBuild', 'HasDetections', 'Census_OSVersion', 'ProductName', Figure 3.3 displays the Crosstab for 'Census_OSVersion', 'SmartScreen', 'HasDetections', Figure 3.4 outputs the Crosstab for "OsVer', 'Platform', 'Processor', 'HasDetections', and Figure 3.5 exhibits the Crosstab for 'Census_ProcessorClass', 'HasDetections', 'SmartScreen', 'ProductName'.

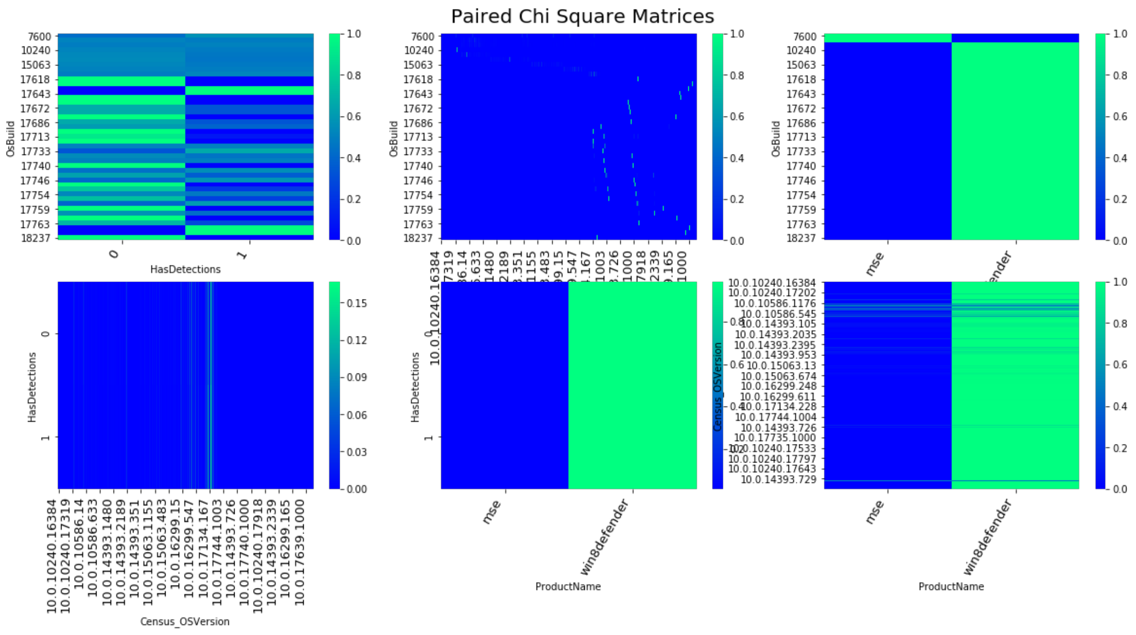


Figure 3.2: Crosstab for 'OsBuild', 'HasDetections', 'Census_OSVersion', 'ProductName'

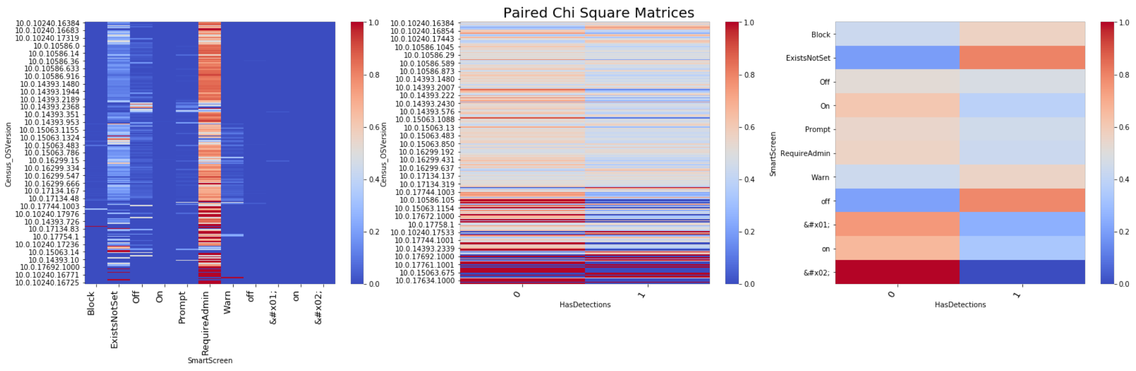


Figure 3.3: Crosstab for 'Census_OSVersion', 'SmartScreen', 'HasDetections'

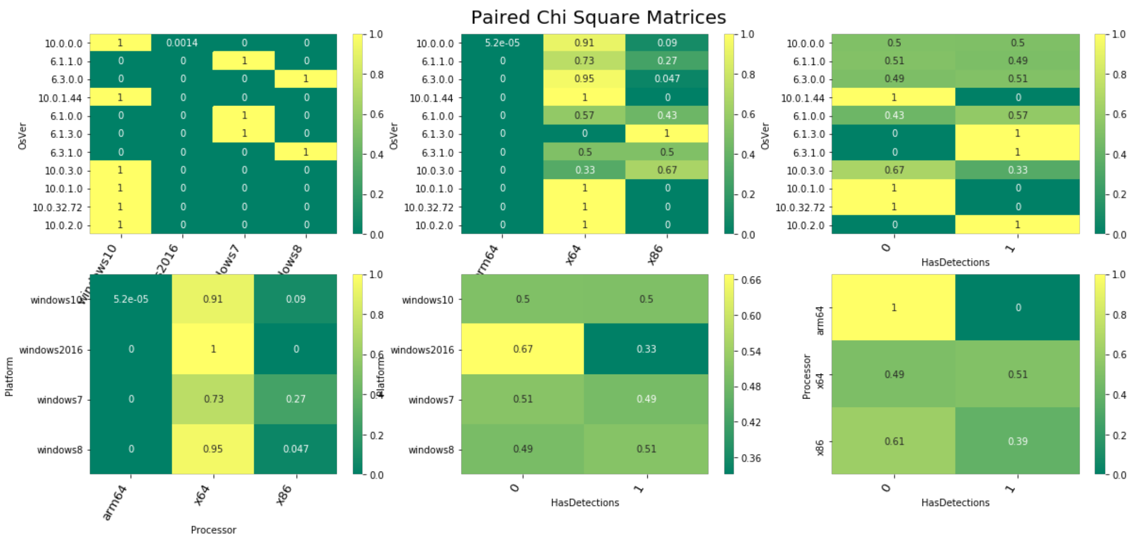


Figure 3.4: Crosstab for 'OsVer', 'Platform', 'Processor', 'HasDetections'

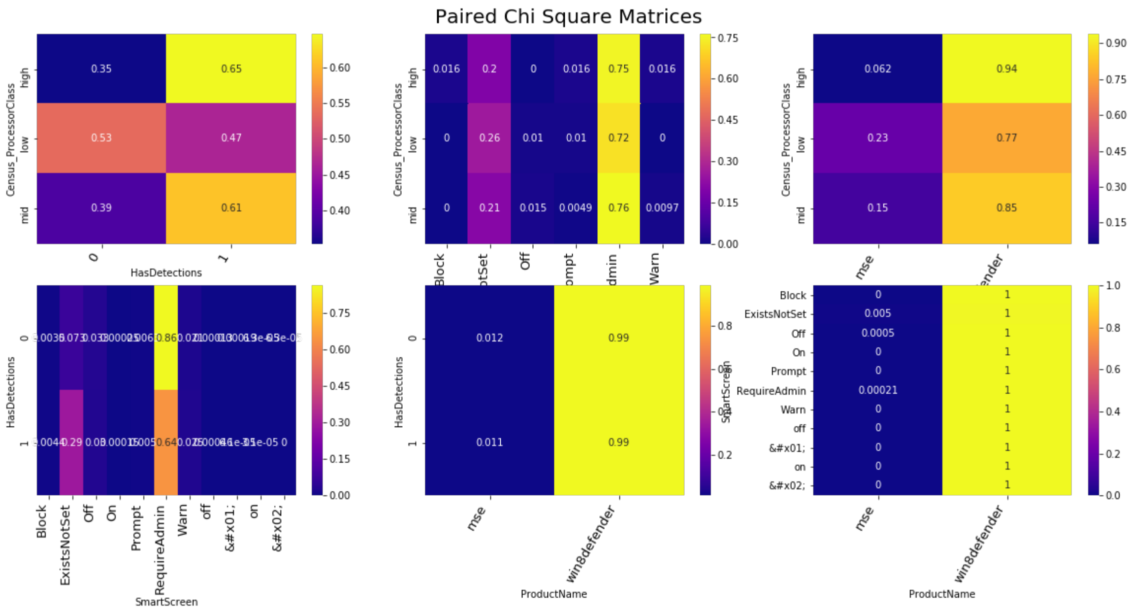


Figure 3.5: Crosstab for 'Census_ProcessorClass', 'HasDetections', 'SmartScreen', 'ProductName'

3.1.4 Correlation Heatmap

A correlation heat map is based on data analysis which uses colors by showing a bar graph uses width and height for a data visualization tool. Showing attribute to attribute relationship among the attributes which we have given as inputs. Validity of the process is 0 to 1 which means it partially good. If it crosses more than 1 or less than 0 then it should be rechecked the whole data if there is any wrong or not otherwise no relation is depending between them. We have used heatmap on the Microsoft Malware Prediction Training Database containing information of 9 million machines.

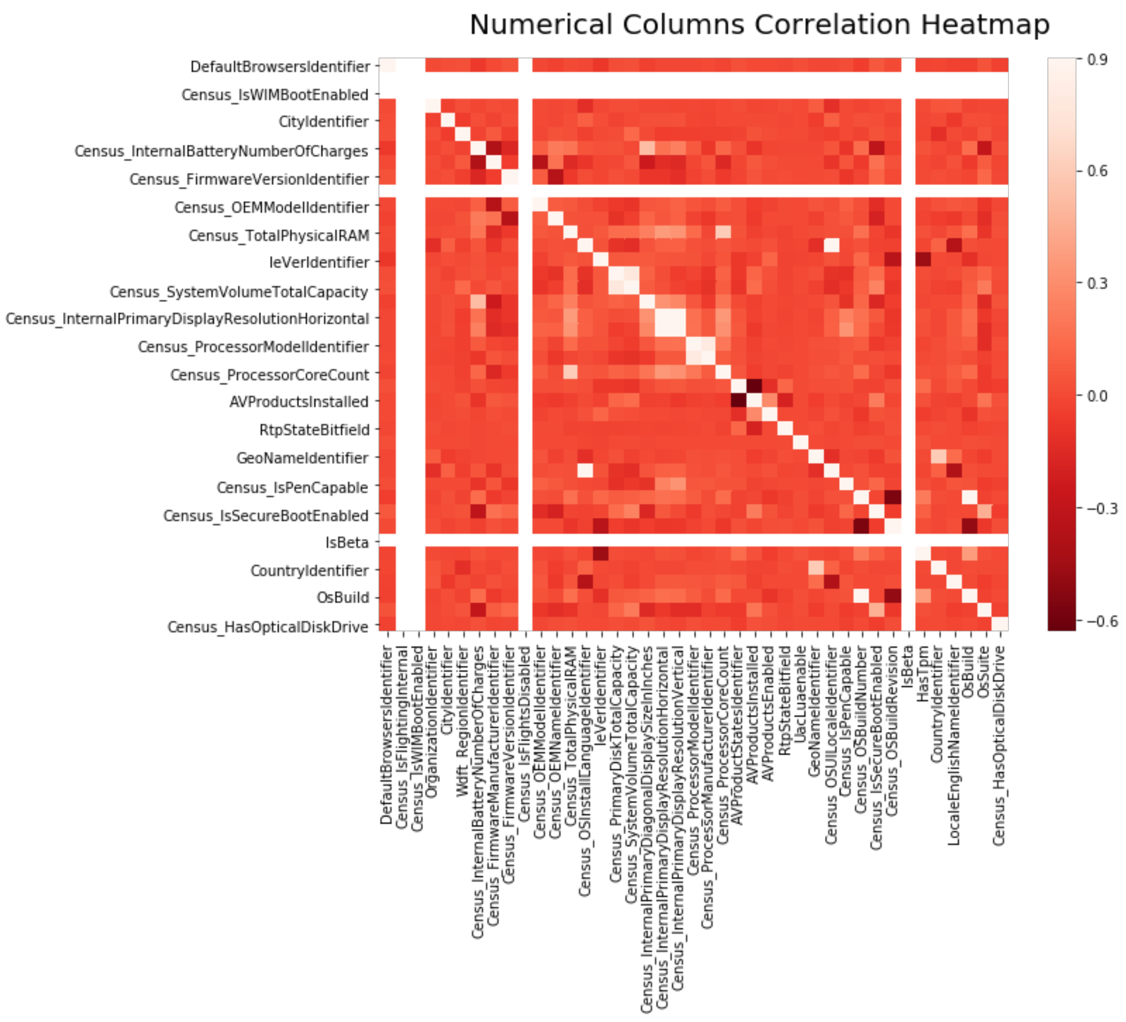


Figure 3.6: Numerical Columns Correlation Heatmap

3.2 Model description

Our experimentation requires the use of a training dataset, where the information from it is used to train the model. The trained model is then used to predict the value of 'HasDetections' in the test dataset. The metric we are going to use to determine the best performing model is the Area Under the ROC Curve (AUC) score, which calculates the area under the ROC Curve between the predicted probability values and observed values.
We start experimentation with Exploratory Data Analysis, from where we extract graphs and charts displaying many characteristics of the dataset. Then feature encoding is done to prepare the dataset for training. After that, the training dataset is used to train the model. Testing is done after that, from which we obtain results for the experiment. The work-plan flowchart is displayed below in Figure 3.7.

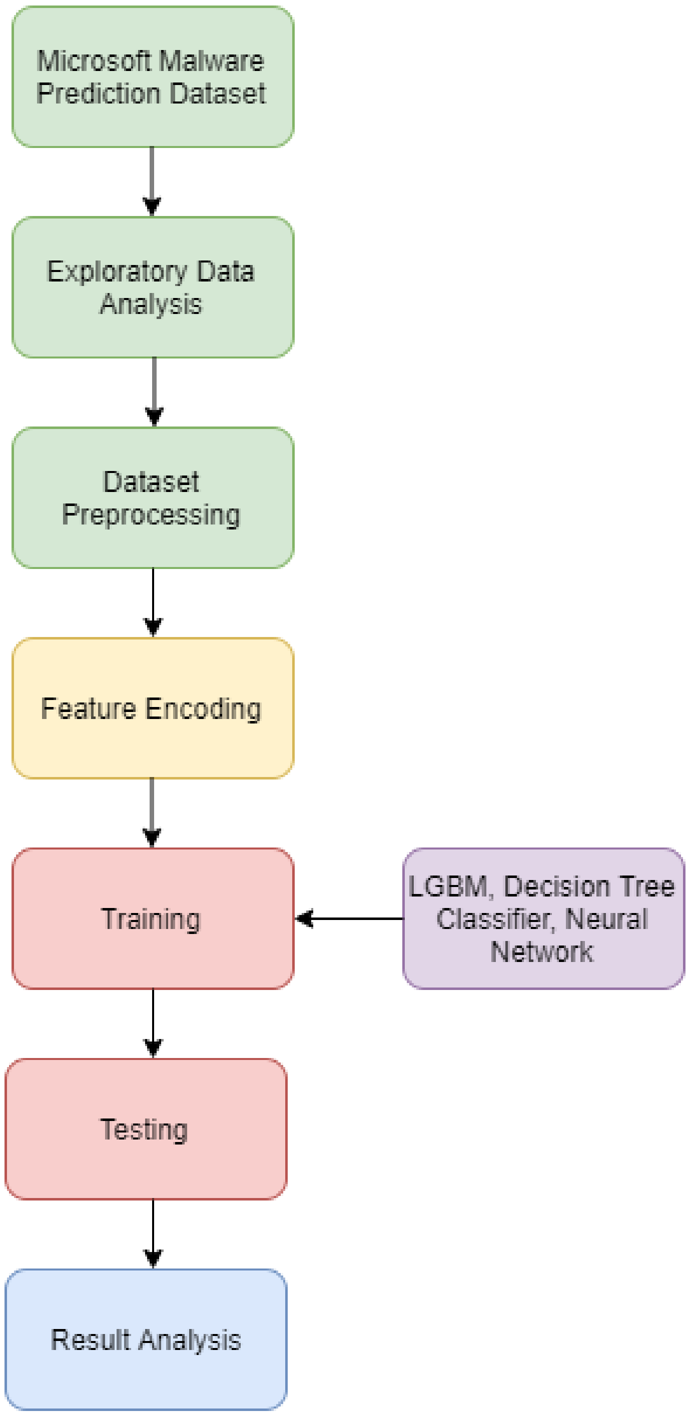

Microsoft Malware Prediction Dataset
Exploratory Data Analysis
Dataset Preprocessing
Feature Encoding
Training
LGBM, Decision Tree Classifier, Neural Network
Testing
Result Analysis

Figure 3.7: Fowchart Displaying the Workplan

Chapter 4

Experimentation

4.1 Experimental Setup

Our experimental setup consisted of a Desktop fitted with a 7th generation Intel Core i7, powerful enough to handle the large amount of data that was to be processed during our experimentation. All our work was done on Jupyter Notebook, with the use of Python 3. Most of the basic data science tools and extensions such as pandas, seaborn, sklearn were imported during experimentation. Running LightGBM required installing the LightGBM module via pip installation. Creating a neural network required installing Tensorflow and Keras.

4.2 Malware Prediction Using LGBM with K-Fold Cross Validation

We have used LightGBM on the Microsoft Malware Prediction Competition dataset so that it can predict if a machine is going to be affected by malware in the future. This is done by identifying the properties in the machine that correlate to the probability of a machine being hit by malware.

The results of our implementation is further improved with the usage of K-Fold Cross Validation. Using the K-Fold Cross Validation technique, the algorithm is run in multiple folds, where inside each fold, the algorithm is run multiple hundred times, from which the most accurate Area Under the Curve (AUC) score is taken. In our code, we have set it so that it stops for each iteration when the AUC score has not improved in 200 consecutive rounds.

We have set 5 folds for our implementation, which means that the algorithm would be run over five iterations. In our implementation, each iteration had over 700 rounds of the algorithm running. The best validation AUC score was taken from each of the five rounds and the scores were averaged. The resulting CV Score was 0.73232. The kernel took only 2 hours approximately to run.The work flow diagram is displayed in Figure 4.1.

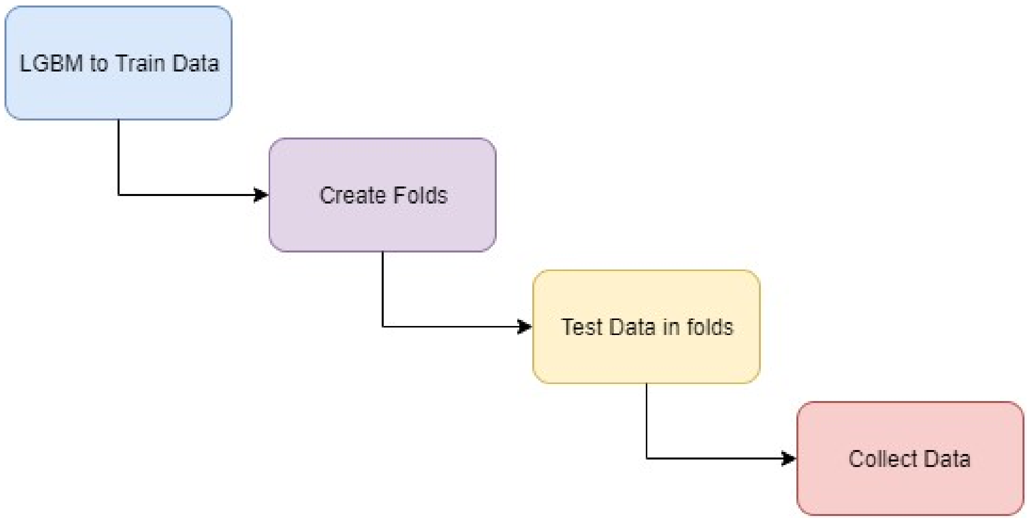


Figure 4.1: Workflow of LGBM experimentation

4.3 Malware Prediction Using LGBM and a Baseline Model with Sparse Matrix

LightGBM was also utilised in this model, because of its low memory consumption. It is necessary to use LGBM for memory management when working with a massive dataset such as the Microsoft Malware Prediction Competition dataset. This helps in saving hours of time when running the experiment on underpowered machines.
The data is converted into a sparse matrix, which can be utilised to effectively use many models, including LightGBM, XGBoost, Random Forest, and Neural Networks. In this experiment, we are pairing the Sparse Matrix with an LGBM Baseline Model. The data was also divided into small chunks of 100000 rows for memory management purposes.
Similar to the experiment above, 5 folds were used, and in these 5 iterations, each iteration ran for 500 rounds, with the best AUC score being recorded for each 100 rounds. In our code, we have set it so that it stops for each iteration when the AUC score has not improved in 100 consecutive rounds. The resulting AUC score in this experiment was 0.739256. The kernel took 2.5 hours to run. The work flow diagram is displayed in Figure 4.2.

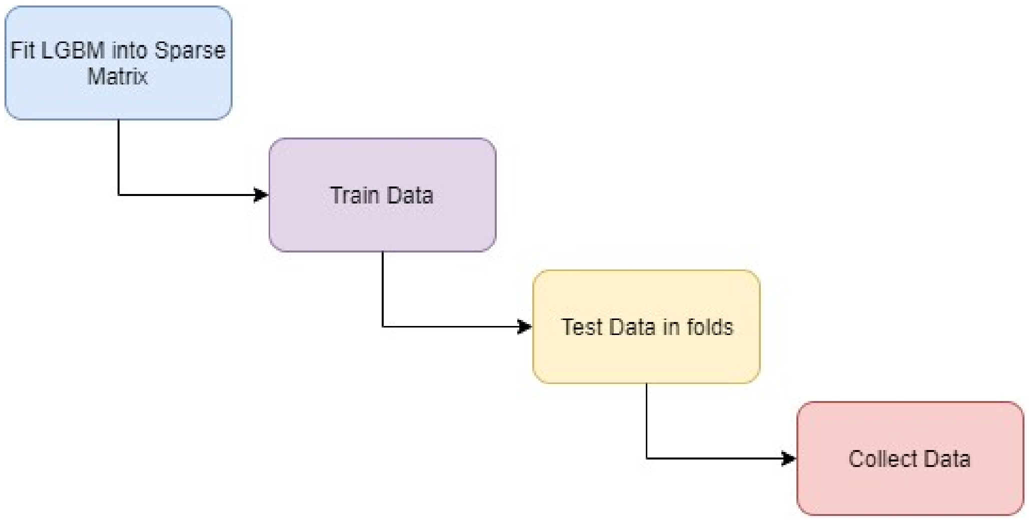


Figure 4.2: Workflow of LGBM with Sparse Matrix experimentation

4.4 Malware Prediction using a Decision Tree Classifier

The general use of decision tree algorithm is to make a training model to predict the values using training data. We applied a decision tree classifier algorithm to create the training model and predict the values of HasDetection and finding the AUC score using K-fold cross validation. However, In the dataset of microsoft malware competition could not be used directly for the algorithm, which is why we had to modify the dataset and create modified test and train dataset. We used scikit learn which needed the categories to be represented by numbers. So, the data were converted to values by finding the decision rate of every category and determining how bigger or smaller they are to a specific level.
After the dataset is converted to a suitable form for decision tree algorithm which contains integer or floating point numbers , decision tree algorithm after pre processing the training dataset. After fine tuning the required parameter the algorithm was used using 4/5 as the training data and 1/5 for test data using k-fold cross validation. After training ROC result was checked and the best sample was taken to find the best result. The AUC score found by applying this method was 0.63923 which indicates the score was average. The kernel is also very memory intensive, and required 8 hours to run. The work flow diagram is displayed in Figure 4.3.

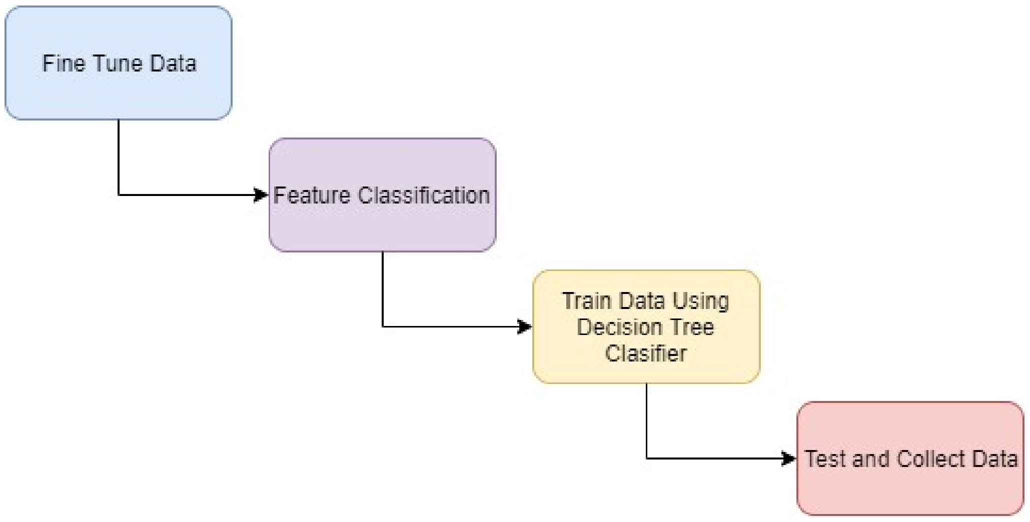


Figure 4.3: Workflow of Decision Tree experimentation

4.5 Malware Prediction using Neural Network

We applied Neural Network to our model using tensorflow backend to build and train model. The model was trained into 20 parts and after each iteration AUC score was taken and the best AUC score was stored. For prediction testing was done by parts by loading 2000000 rows in each iteration. This algorithm resulted in AUC score of 7.028363472706 which is a fair score however massive usage of memory and time consummation making it not so efficient algorithm for prediction. Time taken to run this kernel was approximately 4 hours. The work flow diagram is displayed in Figure 4.4.

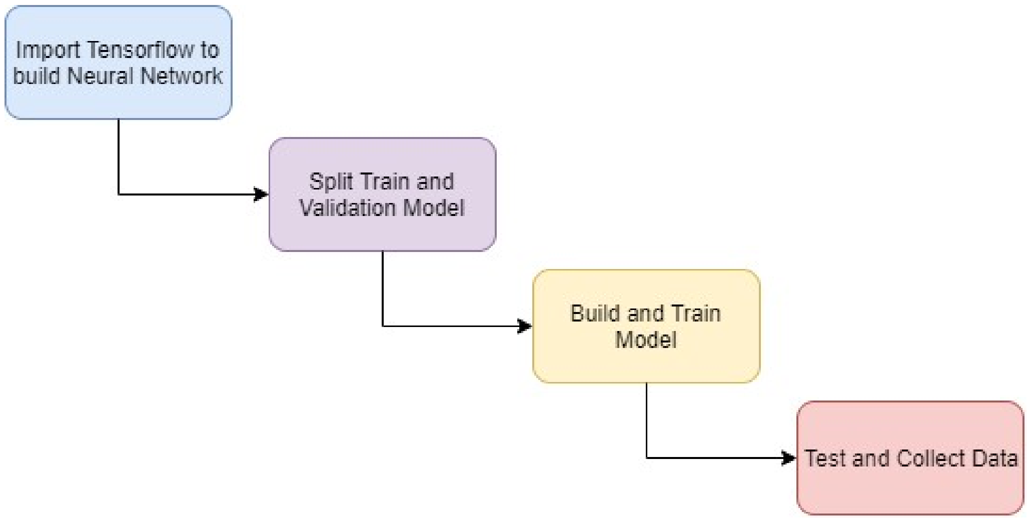

Import Tensorflow to build Neural Network
Split Train and Validation Model
Build and Train Model
Test and Collect Data

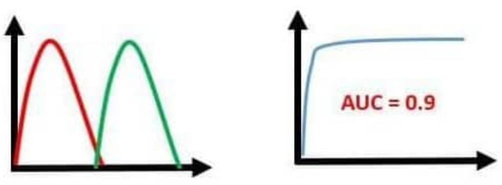
Figure 4.4: Workflow of Neural Network experimentation

Chapter 5

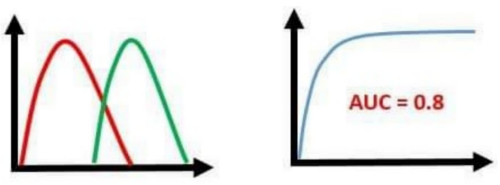
Result Analysis

5.1 Metrics used to determine result

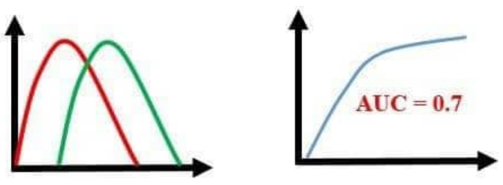
The AUC score is an indicator of the performance of models, and is the perfect metric to base our observations on. AUC Scores below 0.5 are considered to be bad, and scores between 0.5 and 0.7 are considered to be average. Scores of 0.7 and above are considered to be good scores [20]. We are using it as the defining metric for this experiment, as it is the metric used by Microsoft Corporation in its data competitions pertaining to Malware Prediction.



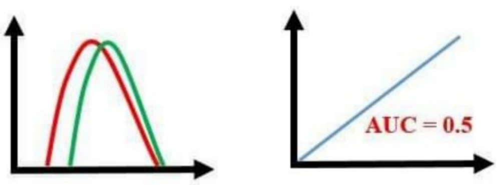
(a) Great Score



(b) Very Good Score



(c) Good Score



(d) Bad Score

Figure 5.1: Visualisation of AUC Scores

The figures in 5.1 shows how AUC is calculated[20]. Green curve indicates True Positive(TP)where the Red curve indicates False positive(FP). Area under threshold is False Negative on the left and false positive on the right. In fig (a) False positive and negative area is so low that it does a good job determining the prediction hence it has a good AUC score of 0.9. In fig (b) it is slightly worse than before but still does a good job to find the prediction making the AUC score 0.8. Similarly, the next figure gives less accurate result and finally in fig (d) a lot of area is under the threshold so it does a poor job on predicting as the FP and FN is high hence a poor AUC score of 0.5.

The AUC-ROC curve visualise the performance for all threshold values. It is plotted using the value of True Positive Rate(TPR) and False Positive Rate(FPR). TPR is simply the sensitivity which is division True Positive by addition of True Positive and False Negative. FPR is the value which for which specificity is needed. It is gained by division of False Positive by summation on False Positive and True Negative. By subtracting it from 1 we get FPR.

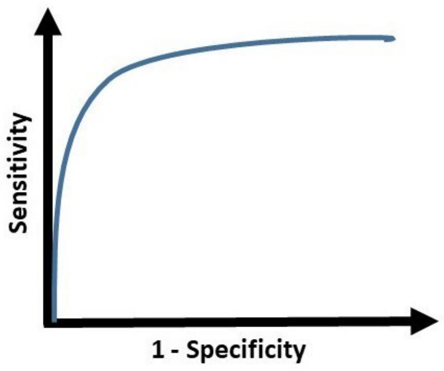


Figure 5.2: Sample AUC-ROC Curve

In Fig 5.2 we can see a sample AUC-ROC curve where Sensitivity or TPR is plotted on the Y-axis and FPR is plotted on the X-axis.

In the table listed below are the AUC Scores of each of the experiments that we have run during our research.

Experiment Title	AUC Score
Malware Prediction Using LGBM with K-Fold Cross Validation	0.73232
Malware Prediction Using LGBM and a Baseline Model with Sparse Matrix	0.73926
Malware Prediction using a Decision Tree Classifier	0.63923
Malware Prediction using Neural Network	0.70284

Table 5.1: AUC Score Table

Experiment Title	Time Taken (approx)
Malware Prediction Using LGBM with K-Fold Cross Validation	2 hrs
Malware Prediction Using LGBM and a Baseline Model with Sparse Matrix	2.5 hrs
Malware Prediction using a Decision Tree Classifier	8 hrs
Malware Prediction using Neural Network	4 hrs

Table 5.2: Table of Time Taken to Run Experimentation

5.2 Comparative Analysis

During experimentation, we faced huge difficulties with the sheer size of the dataset, which resulted in very large runtimes for the kernels. The use of LightGBM alleviated this problem, using its gradient boosting capabilities to speed up the runtime significantly. As can be seen on the AUC Score table, the kernels where LGBM was used clearly had a higher score, which shows that the use of LightGBM resulted in a higher Area under the ROC Curve between the predicted probability and actual probability. This means that LGBM had a higher accuracy compared to both Decision Tree Classifiers and Neural Networks. The kernel which utilised Neural Networks also provided a satisfactory AUC Score of above 0.7 but at a cost of a higher runtime, which makes the process less efficient. The Decision Tree Classifier performed the worst in both AUC Score, and runtime, as it had a very average AUC Score of .63923, and also required more than 8 hours to run the complete kernel. The LGBM kernel where the model was fitted into a Sparse Matrix performed better than the one where LGBM was paired with K-Fold Cross Validation. All in all, when it comes to finding out Malware Infection Rates in machines, the experiment where we had LightGBM fitted into a Sparse Matrix performed the best out of all the experiments.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In our research, we conducted four experiments with the use of three different algorithms - LightGBM, Decision Tree Classifier, and Neural Networks, along with Cross Validation techniques on the dataset we collected from the Microsoft Malware Prediction Dataset. Prior to doing the experimentation, Exploratory Data Analysis was done on the dataset, where we discovered Feature Importances, compared between different independent categorical variables using Chi-squared test, and plotted a Numerical Columns Correlation Heatmap for the Dataset. After that, the algorithms were applied on the dataset during experimentation. We applied the Area Under the ROC Curve (AUC) Score as the defining metric for our experimentation. After analysis of the results of all the experiments, LightGBM fitted into a Sparse Matrix provided the best results, as it obtained the highest AUC Score, and also had great efficiency, taking a comparatively low amount of time to run. This shows the viability of LightGBM as the leading Gradient Boosting algorithm in the prediction of malware infection rates in machines. It has great potential to be implemented in future Malware Prediction and Protection systems, which would result in more foolproof protection from malicious cyberattacks, and billions of dollars in savings for large corporations and small businesses worldwide.

6.2 Future Work

The main objective of our research is to discover the best algorithm for predicting malware infection rates in machines. We have used both traditional algorithms like the Decision Tree Classifier, and Neural Networks, as well as a more modern gradient boosting model in LightGBM. We plan on doing more experimentation using a rival gradient boosting algorithm called XGBoost, and compare it with the LGBM module. We are also eagerly awaiting for the release of another comprehensive dataset like the one we used from Microsoft Corporation, which till date is the most detailed dataset pertaining to our topic presently available on the internet. This way we could further cement our hypothesis, and assist in the global fight against Cybercrime.

Bibliography

[1] *Malware*, May 2019. [Online]. Available: https://www.veracode.com/security/malware.

[2] *What are malware, viruses, spyware, and cookies, and what differentiates them ?: Symantec connect*. [Online]. Available: https://www.symantec.com/connect/articles/what-are-malware-viruses-spyware-and-cookies-and-what-differentiates-them.

[3] Cybercrimemag, *Cybercrime damages $6 trillion by 2021*, Dec. 2018. [Online]. Available: https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/.

[4] G. Burton, *Total it shut down at lincolnshire county council over zero-day attack: Computing*, May 2017. [Online]. Available: https://www.computing.co.uk/ctg/news/2443531/total-it-shut-down-at-lincolnshire-county-council-over-zero-day-attack.

[5] D. Gavriluţ, M. Cimpoeşu, D. Anton, and L. Ciortuz, "Malware detection using machine learning", in *2009 International Multiconference on Computer Science and Information Technology*, IEEE, 2009, pp. 735–741.

[6] M. Rhode, P. Burnap, and K. Jones, "Early-stage malware prediction using recurrent neural networks", *computers & security*, vol. 77, pp. 578–594, 2018.

[7] M. Baset, "Machine learning for malware detection", PhD thesis, MSc. Dissertation, School of Mathematical and Computer Sciences, Heriot-Watt ..., 2016.

[8] M. Yeo, Y. Koo, Y. Yoon, T. Hwang, J. Ryu, J. Song, and C. Park, "Flow-based malware detection using convolutional neural network", in *2018 International Conference on Information Networking (ICOIN)*, IEEE, 2018, pp. 910–913.

[9] C. Kang, N. Park, B. A. Prakash, E. Serra, and V. Subrahmanian, "Ensemble models for data-driven prediction of malware infections", in *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, ACM, 2016, pp. 583–592.

[10] *Features*. [Online]. Available: https://lightgbm.readthedocs.io/en/latest/Features.html.

[11] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, "Lightgbm: A highly efficient gradient boosting decision tree", in *Advances in Neural Information Processing Systems*, 2017, pp. 3146–3154.

[12] R. E. Uhrig, “Introduction to artificial neural networks”, in *Proceedings of IECON’95-21st Annual Conference on IEEE Industrial Electronics*, IEEE, vol. 1, 1995, pp. 33–37.

[13] C. Nicholson, *A beginner’s guide to neural networks and deep learning.* [Online]. Available: https://skymind.ai/wiki/neural-network.

[14] A. Navada, A. N. Ansari, S. Patil, and B. A. Sonkamble, “Overview of use of decision tree algorithms in machine learning”, in *2011 IEEE control and system graduate research colloquium*, IEEE, 2011, pp. 37–42.

[15] B. Gupta, A. Rawat, A. Jain, A. Arora, and N. Dhami, “Analysis of various decision tree algorithms for classification in data mining”, *International Journal of Computer Applications*, vol. 163, no. 8, pp. 15–19, 2017.

[16] J. Quinlan, *Machine learning: An artificial intelligence approach: Michalski, r. s., carbonell, jg, mitchell, t. m., eds*, 1983.

[17] *Decision tree tutorials notes: Machine learning.* [Online]. Available: https://www.hackerearth.com/practice/machine-learning/machine-learning-algorithms/ml-decision-tree/tutorial.

[18] *Microsoft malware prediction.* [Online]. Available: https://www.kaggle.com/c/microsoft-malware-prediction.

[19] J. Dietle, *”microsoft malware prediction” and its 9 million machines*, Mar. 2019. [Online]. Available: https://towardsdatascience.com/microsoft-malware-prediction-and-its-9-million-machines-22e0fe8c80c8.

[20] J. D’Souza, *Let’s learn about auc roc curve!*, Mar. 2018. [Online]. Available: https://medium.com/greyatom/lets-learn-about-auc-roc-curve-4a94b4d88152.

Appendix

Malware Prediction Using LGBM with K-Fold Cross Validation

```
folds = KFold(n_splits=5, shuffle=True, random_state=15)
oof = np.zeros(len(train))
categorical_columns = [c for c in categorical_columns if c not in
                      ['MachineIdentifier']]
features = [c for c in train.columns if c not in
          ['MachineIdentifier']]
predictions = np.zeros(len(test))
start = time.time()
feature_importance_df = pd.DataFrame()
start_time= time.time()
score = [0 for _ in range(folds.n_splits)]

for fold_, (trn_idx, val_idx) in
enumerate(folds.split(train.values,target.values)):
    print("fold n {}".format(fold_))
    trn_data = lgb.Dataset(train.iloc[trn_idx][features],
                           label=target.iloc[trn_idx],
                           categorical_feature = categorical_columns
                          )
    val_data = lgb.Dataset(train.iloc[val_idx][features],
                           label=target.iloc[val_idx],
                           categorical_feature = categorical_columns
                          )

    num_round = 10000
    clf = lgb.train(param,
                    trn_data,
                    num_round,
                    valid_sets = [trn_data, val_data],
                    verbose_eval=100,
                    early_stopping_rounds = 200)

    oof[val_idx] = clf.predict(train.iloc[val_idx][features],
                   num_iteration=clf.best_iteration)
    fold_importance_df = pd.DataFrame()
    fold_importance_df["feature"] = features
    fold_importance_df["importance"] =
    clf.feature_importance(importance_type='gain')
    fold_importance_df["fold"] = fold_ + 1
    feature_importance_df =
    pd.concat([feature_importance_df, fold_importance_df], axis=0)

    # we perform predictions by chunks
    initial_idx = 0
    chunk_size = 1000000
    current_pred = np.zeros(len(test))
    while initial_idx < test.shape[0]:
        final_idx = min(initial_idx + chunk_size, test.shape[0])
        idx = range(initial_idx, final_idx)
        current_pred[idx] = clf.predict(test.iloc[idx][features],
        num_iteration=clf.best_iteration)
        initial_idx = final_idx
    predictions += current_pred / min(folds.n_splits, max_iter)

    print("time elapsed:
    {:<5.2}s".format((time.time() - start_time) / 3600))
    score[fold_] = metrics.roc_auc_score(target.iloc[val_idx],
    oof[val_idx])
    if fold_ == max_iter - 1: break



if (folds.n_splits == max_iter):
    print("CV score: {:<8.5f}".format(metrics.roc_auc_score
                                        (target, oof)))
else:
     print("CV score: {:<8.5f}".format(sum(score) / max_iter))
```

Malware Prediction Using LGBM and a Baseline Model with Sparse Matrix

```
print('\nLightGBM\n')

for train_index, test_index in skf.split(train_ids, y_train):

    print('Fold {}\n'.format(counter + 1))

    train = load_npz('train.npz')
    X_fit = vstack([train[train_index[i* m:(i+1)* m]]
    for i in range(train_index.shape[0] // m + 1)])
    X_val = vstack([train[test_index[i* m:(i+1)* m]]
    for i in range(test_index.shape[0] //  m + 1)])
    X_fit, X_val = csr_matrix(X_fit, dtype='float32'),
    csr_matrix(X_val, dtype='float32')
    y_fit, y_val = y_train[train_index], y_train[test_index]

    del train
    gc.collect()

    lgb_model = lgb.LGBMClassifier(max_depth=-1,
                                   n_estimators=30000,
                                   learning_rate=0.05,
                                   num_leaves=2**12-1,
                                   colsample_bytree=0.28,
                                   objective='binary',
                                   n_jobs=-1)


    lgb_model.fit(X_fit, y_fit, eval_metric='auc',
                  eval_set=[(X_val, y_val)],
                  verbose=100, early_stopping_rounds=100)


    lgb_train_result[test_index] += lgb_model.predict_proba(X_val)[:,1]

    del X_fit, X_val, y_fit, y_val, train_index, test_index
    gc.collect()

    test = load_npz('test.npz')
    test = csr_matrix(test, dtype='float32')
    lgb_test_result += lgb_model.predict_proba(test)[:,1]
    counter += 1

    del test
    gc.collect()



print('\nLigthGBM VAL AUC Score:
      {}'.format(roc_auc_score(y_train, lgb_train_result)))
```

Malware Prediction Using Decision Tree

```
def fine_tune_decision_tree(training_set, k_fold):
    results = dict()
```

```
avg_grade = dict()
std_grade = dict()
min_grade = dict()
max_grade = dict()
best_sample_leaf = 0
best_grade = 0.5
for min_samples_leaf in [200, 400, 500, 600, 800, 1000]:
    features = [c for c in training_set.columns if c not in
                  ['MachineIdentifier', 'HasDetections']]
    dt = DecisionTreeClassifier(min_samples_leaf=min_samples_leaf)
    results[min_samples_leaf] = []

    # Train and tests the data on k_fold splits
    and store the results
    for train_indices, test_indices in k_fold.split(training_set):
        print('Start fitting')
        dt.fit(training_set[features].iloc[train_indices],
        training_set['HasDetections'].iloc[train_indices])
        print('End fitting - Start predicting')
        prob = dt.predict_proba(training_set[features].iloc
        [test_indices])
        fpr, tpr, thresholds =
        sklearn.metrics.roc_curve(training_set
        ['HasDetections'].iloc[test_indices], prob[:, 1])
        results[min_samples_leaf].append
        (sklearn.metrics.auc(fpr, tpr))
        print('End predicting')

    grade = np.mean(results[min_samples_leaf])
    if grade > best_grade:
        best_grade = grade
        best_sample_leaf = min_samples_leaf
    avg_grade[min_samples_leaf] = grade
    std_grade[min_samples_leaf] = np.std(results[min_samples_leaf])
    min_grade[min_samples_leaf] = np.min(results[min_samples_leaf])
    max_grade[min_samples_leaf] = np.max(results[min_samples_leaf])

# Now plot the result.
n_leafs = avg_grade.keys()
avgs = [avg_grade[l] for l in n_leafs]
stds = [std_grade[l] for l in n_leafs]
plt.figure()
plt.errorbar(n_leafs, avgs, stds)
plt.title('decision tree classifier k fold results')
plt.xlabel('number of minimum sample in a leaf')
plt.ylabel('ROC curve area')
plt.show()
```

```
    return DecisionTreeClassifier(min_samples_leaf=best_sample_leaf)

# Define decision tree predictor and fine tune its variables
k_fold = KFold(n_splits=5, shuffle=True)
classifier = fine_tune_decision_tree(training_set, k_fold)
```

Malware Prediction Using Neural Networks

```
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout,
                                    BatchNormalization,
                                    Activation
from tensorflow.keras.callbacks import LearningRateScheduler
from tensorflow.keras.optimizers import Adam

#SPLIT TRAIN AND VALIDATION SET
X_train, X_val, Y_train, Y_val = train_test_split(
    df_train[cols], df_train['HasDetections'], test_size = 0.5)

# BUILD MODEL
model = Sequential()
model.add(Dense(100,input_dim=len(cols)))
model.add(Dropout(0.4))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dense(100))
model.add(Dropout(0.4))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer=Adam(lr=0.01),
              loss="binary_crossentropy", metrics=["accuracy"])
annealer = LearningRateScheduler(lambda x: 1e-2 * 0.95 ** x)

# TRAIN MODEL
model.fit(X_train,Y_train, batch_size=32, epochs = 20,
callbacks=[annealer, printAUC(X_train, Y_train)],
validation_data = (X_val,Y_val), verbose=2)
```