MALWARE DETECTION AND ANALYSIS USING MACHINE LEARNING

CSE3502 - Information Security Management

J Component Project

20BCB0098	ROHINI DAS
20BCB0148	DHARMIK NAICKER

B.Tech. Computer Science and Engineering

School of Computer Science and Engineering
Vellore Institute of Technology
Vellore
March, 2023

Under the guidance of
Prof. Aju D
SCOPE, VIT, Vellore

INDEX

1. ABSTRACT

Malware detection is typically carried out with the use of anti-infection software, which compares each software in the system to known malware. With the aid of machine learning calculations, we might also identify malware from other threats. We could train a model to determine whether a software is malware using the known characteristics of malware. In this vein, we'll use machine learning computations to determine whether a particular software is malware or not. Since the invention of the web, information has increased dramatically. In addition, the nature of information is evolving quickly. Along with the dataset, we also release open source code for extracting highlights from more parallels so that the dataset can be supplemented with additional example highlights. This dataset makes up for a deficiency in the data security AI folks group: a big, open, and broadly applicable dataset that is amicable/pernicious and covers a few fascinating use cases.

Keywords: Malware detection, XG-Boost Algorithm, Automated malware detection, Machine Learning, Neural Networks

2. INTRODUCTION

Machine learning can be a useful tool for both helpful location heuristics and a vital discovery capability. In creating information that is differentiating between retaliatory and charitable examples, managed learning models subsequently make inappropriate use of intricate connections between document ascribes. Additionally, properly regularized AI models add up to new examples whose names and highlights follow a comparative dispersion to the preparation data. In any case, the security network as a whole agrees that the existing mark-based approach to dealing with infection identification is not currently sufficient. File infectors and stand-alone malware make up a straightforward taxonomy of malware. Computer malware (viruses, worms, Trojan horses, rootkits, botnets, backdoors, and other malicious software) spreads quickly, and traditional antivirus systems based on signature matching are unable to identify polymorphic and novel, previously undiscovered dangerous executables. The problems with dynamic examination may have a potential solution in static executable investigation. Static analysis looks at the executable's internal structures that are necessary for its proper operation. These structures cannot be removed, encoded (even though their code content may be), or mixed up without issue since they are controlled by the record type. One suggested method (solution) is combining machine learning (classification) techniques with automatic dynamic (behavior) malware analysis to identify malware effectively and efficiently. A summary of various machine learning techniques that have been presented for malware detection is provided in Additionally, it requires significantly less computer resources than dynamic inspection because it only involves parsing structures. Recently, research was conducted on Windows compact executable (PE) documents static executable examination. Particularly noteworthy are the numerous advanced persistent threat (APT) malware campaigns targeted specifically at Macs during the past four to five years.. Exploring instruments to confront these dangers currently guarantees that we will have the option to all the more likely handle the expanding danger later on. We demonstrate that XGB can be trained toward a reliable optimization objective, making the algorithm more resistant to hostile attacks. We highlight an existing tradeoff between malware detection score and robustness to adversarial attacks as a result of robust training. According to our findings, robust models are significantly less vulnerable to adversarial attacks than non-robust models.

3. LITERATURE SURVEY

3.1 Accurate and Robust Malware Detection: Running XGBoost on Runtime Data From Performance Counters (R. Elnaggar, et. al. 2022) (Base Paper)

It has recently been suggested that malware can be found using hardware performance counters (HPCs). We demonstrate how utilizing the XGBoost classifier and suitable data pretreatment can increase malware detection performance while employing HPCs by at least 15%. By classifying HPC datastreams at frequent intervals, we also demonstrate how the suggested technique can identify malware quickly (shortly after it has been released).This paper provides a multitemporal classification approach that guarantees the early identification of a large proportion of malware while maintaining generally low false positive rates. Last but not least, we demonstrate how thorough training makes the XGBoost classifier up to 50x less vulnerable to attacks from the opposite side that aim to reduce its effectiveness in detecting malware.

Strengths: The paper shows how proper data preprocessing and the use of the XGBoost classifier can be used to improve the performance of malware detection using HPCs by at least 15% and also,the proposed method can detect malware early) by classifying HPC datastreams at short time intervals.

Limitations: Malware detection techniques using hardware performance counters (HPCs) have recently been suggested. Recent research has also demonstrated that using realistic performance counter sampling techniques to identify malware is ineffective.

3.2 Android Malware Detection Using Deep Learning (Elayan & Mustafa, 2021):

Both machine learning and deep learning methods can be used to analyze static malware. In this research, virus detection in Android applications is compared using Gated Recurrent Unit (GRU), a deep learning technique, and traditional machine learning algorithms. On the website of the Canadian Institute of Cybersecurity, 347 benign samples and 365 malware samples of Android applications were combined to create the dataset. During feature engineering, the API calls and permissions features were taken out of the Android applications.A dataframe was built, with the rows representing malicious and benign APK files and the columns representing features (expressed as binary values). As machine learning classifiers, Support Vector Machines, K-Nearest Neighbors, Decision Tree, Random Forest, and Naive Bayes were used, and their performance was evaluated using standard performance criteria (namely, accuracy, recall, F score, and precision). GRU, a deep learning method, was utilized to create a binary classification model. The experimental findings showed that, among machine learning algorithms, Random Forest did the best (correctly predicted 97% of the dataset), but the deep learning classifier surpassed all of the models (accurately predicted 98% of the dataset and 99.8% of the malware samples).

Strengths: The paper shows that GRU deep learning approach enhances the performance of malware classifiers. It also shows that , both the machine learning and deep learning models based on API-calls and permissions extracted from a real world dataset, performed very well.

Limitations: The paper could have added other performance metrics such as AUC, FPR, etc, to enhance the models. The classifiers could be tested on a larger dataset. A hybrid classification approach could be taken towards malware detection, which wasn't factored in the comparison.

3.3 Detection of Malicious Software by Analyzing Distinct Artifacts Using Machine Learning and Deep Learning Algorithms (Ashik et al., 2021):

Anti-malware programmes typically utilize signature-based methods to find known malware. This, however, is useless against unidentified, disguised, or packaged malware. In order to find a feature that categorizes the executable, the relevance of structural features (such as mnemonics, instruction opcodes, API calls, etc.) of unpacked malicious and benign PE files was examined in this study. Support Vector Machine (SVM), Naive Bayes, J48 Decision Tree, Random Forest (RF), XGBoost, Deep Dense network, One-Dimensional Convolutional Neural Network (1D-CNN), and CNN-LSTM were used to conduct the experiment on four different datasets. ANOVA test and Minimum Redundancy Maximum Relevance (mRMR) were used to accomplish feature selection from the static features (4-gram mnemonic trace, instruction opcode, etc.).

- Random Forest and Adaboost models had high detection rates for dataset-1(VX-Data set).
- DNN classifier had the best results (accuracy score= 99.1 %) for dataset-2 (VirusShare Data set).

Strengths: The classifiers were less prone to inefficiencies caused by redundant features because feature selection was done before fusing the features. In comparison to simply dynamic analysis or static analysis alone, combining system calls/API with static features was more efficient and accurate. demonstrated the diversity of the classifiers using four different datasets, one of which contained harmful samples that were disguised.

Limitations: By using majority voting, ensembles of classifiers can be utilized to categorize samples that haven't been seen. The test can be extended to categorizing malware according to its family. It is possible to further explore the effectiveness of ML and DL approaches on packed samples produced through feature modifications in order to find a defense against adversary attacks.

3.4 A Novel Machine Learning Based Malware Detection and Classification Framework(Sethi et al., 2019):

A malware analysis approach for detection and classification is suggested in this paper. Static analysis is done for malware detection and classification using feature engineering, while dynamic analysis is done using the Cuckoo sandbox. Using Chi-square and Random Forest feature selection methods, the features are further condensed. The researchers collected information for PE header files (sources: VirusShare and VirusTotal). Two datasets were produced after feature extraction and selection: Macro (for a multi-class classifier) and Micro (for binary classifier). The two classifiers' models were created using the scikit-learn Python library.On the basis of accuracy, precision, recall, f-measure, and AUC, the machine learning models (KNN, Decision Tree, SVM, Random Forest) were assessed.The trial findings showed that Decision Tree, with an accuracy score of 99.11%, performed the best for the overall malware classification (binary classification). The Adware and Trojan malware families saw the highest precision rate (1.0) and recall rate from the multi-class classifier model (1 and 0.9 respectively).

Strengths: A more thorough study is possible thanks to the detection of malware as well as the classification of it by family. The feature selection significantly increased the effectiveness of the models by decreasing feature redundancy. Increasing the precision score also involves using false positives as a performance metric. The classifiers were able to tackle malware obfuscation because dynamic characteristics were used in addition to static features.

Limitations: To test the models' scalability, the experiment could be run on a sizable real-world dataset. For handling the dynamic malware analysis, deep learning methods can be examined.

3.5 Static Malware Analysis using Machine Learning Algorithms on APT1 Dataset with String and PE Header Features(Balram et al., 2019):

Static malware analysis extracts features from executable files(PE files) and evaluates them without actually running the code. Machine learning and data analytic techniques are used to classify and detect malware attacks from a large amount of data. In this paper, 6 machine learning models were implemented as malware detectors with feature engineering. The APT1 dataset was taken from Mandiant(now owned by FireEye) in 2013. It consists of 427 malicious PE files and 989 benign PE files. Feature engineering was implemented to get more information and two sets of features was found upon examination: PE header features (extracted or derived from the file's portable executable headers) and string features(possible since the files were not encrypted) . The malware classifiers: SVM, LR , RF , XGBoost, LR/XGboost, LR/RF/Naive Bayes were evaluated based on their performance on both string and PE header features. The experimental results showed that the accuracy scores of all models on string features were above 90% whereas the accuracy score averaged on 70% for PE features for this dataset.

Strengths: The experimental results indicate that the classifiers were highly accurate in predicting the true positives(the number of malicious files predicted that were actually malicious) for string features. The malware detectors prove to be effective for a small dataset, with the XGBoost and XGBoost/Linear Regression being concluded as the most efficient models.

Limitations: The experimental results were based on the model classifiers performance for only a small dataset, therefore we do not know for sure how well they perform on a large,real world dataset. The accuracy of the classifiers on the PE header features was found lacking. The experiment could be performed by adding more PE features. Not much exploration was done on other machine learning techniques such as neural networks and deep learning techniques such as CNN. There could be improvements on the classification of malware such as identifying to which family a particular malware sample belongs to.

3.6 The rise of machine learning for detection and classification of malware: Research developments, trends and challenges (Gibert et al., 2020):

Malware complexity changes as quickly as innovation advances, creating a never-ending conflict between security experts and malware developers. Because machine learning approaches can keep up with the growth of malware, current cutting-edge research focuses on their development and use in malware detection.

Strength: A series of bytes is the most basic way to express a computer programme. In other words, each byte in an input sequence is treated as a separate unit. Since this representation is unaffected by the executable file format, such as whether it is a Portable Executable (PE) file or an Executable and Linkable Format (ELF) file, etc., it can be used to represent malware independently of the operating system and hardware. The benefit of late fusion is that it makes it possible to use several models on various modalities, making it more versatile. Additionally, it is simpler to address missing modalities because each modality's predictions are created independently.

Limitations: The fact that machine learning just produces models without understanding their ramifications is one of its main flaws. It simply processes data and makes judgments using the most effective, statistically validated way. As was already said, the system receives millions of data points without having any of them particularly identified as symptoms of infection. That will be determined by the machine learning model on its own.

3.7 Use of Honeypot in Machine Learning Based on Malware Detection (Matin & Rahardjo, 2020):

A honeypot system is one that is purposefully left up as bait to entice potential attackers away from crucial systems. The purpose of a honeypot is to deflect attention away from important systems, gather information about attacker activities, and let attackers remain on the system.

Strength: A virtual honeypot is a trap system created with the aid of virtualization technology, allowing the honeypot to function on a single workstation. Virtual honeypot can react to network activity by imitating it on a single system. Another way to describe Honeypot is as a simulation engine with model behavior. Because it can be readily integrated into a system utilizing VMWare, User Mode Linux, and Microsoft Virtual PC, virtual honeypot is more frequently used. The benefit of adopting virtualization honeypot is that it is simple to isolate and repair. A virtual honeypot can also imitate several systems on a single workstation. In order to improve accuracy, machine learning can benefit from honeypot updates to training data.

Limitations: When we use a honeypot application, the honeypot lacks a clear scheme, making implementation challenging and unclear, especially when one wishes to customize it for the development of a machine learning model. If machine learning is not adequately trained, it runs the danger of using ineffective algorithms and producing few predictions. It is necessary to teach machine learning algorithms how to analyze data patterns and form inferences in order to discover abnormalities and recognise malware risks. The algorithm won't be able to distinguish between clean and malicious files when fed with a huge number of samples, hence the solution will produce incorrect results if the database is corrupt or not labeled appropriately. To avoid producing incorrect results, engineers must still intervene and optimize the algorithm.

3.8 Significant Permission Identification for Machine-Learning-Based Android Malware Detection (Li et al., 2018):

The alarming growth rate of malicious apps has become a serious issue that sets back the prosperous mobile ecosystem. A recent report indicates that a new malicious app for Android is introduced every 10 s. To combat this serious malware campaign, we need a scalable malware detection approach that can effectively and efficiently identify malware apps. Numerous malware detection tools have been developed, including system-level and network-level approaches. Malware detection using significant permissions is more advantageous to use MLDP to perform malware detection as it can be effective while notably conserving time and memory. Since the time and memory are limited in common computers, we can outsource the task to the cloud in a suitable way to boost efficiency.

The limitation was found to be that MLDP is not as efficient as SigPID (significant permission) and SigPID is much more effective detecting up to 93.62% of malware in the dataset.

3.9 Android Malware Detection Using Genetic Algorithm based Optimized Feature Selection and Machine Learning (Fatima et al., 2019):

Android platform due to open source characteristics and Google backing has the largest global market share. Being the world's most popular operating system, it has drawn the attention of cyber criminals operating particularly through wide distribution of malicious applications. This paper proposes an effectual machine-learning based approach for Android Malware Detection making use of evolutionary Genetic algorithm for discriminatory feature selection. Selected features from Genetic algorithms are used to train machine learning classifiers and their capability in identification of Malware before and after feature selection is compared. The experimentation results validate that the Genetic algorithm gives the most optimized feature subset, helping in reduction of feature dimension to less than half of the original feature-set.

Strengths: The proposed methodology attempts to make use of evolutionary Genetic Algorithm to get the most optimized feature subset which can be used to train machine learning algorithms in the most efficient way. From experimentations, it can be seen that a decent classification accuracy of more than 94% is maintained using Support Vector Machine and Neural Network classifiers while working on lower dimension feature-set, thereby reducing the training complexity of the classifiers.

Limitations: Even though it has a detection accuracy as high as 96% it has one disadvantage, being that it could run only on rooted devices, making it incapable for commercial use.

3.10 Android Malware Detection Using Machine Learning on Image Patterns(Using Random Forest algorithm) (Darus et al., 2018):

Android platform has been targeted by cyber-criminals due to the increased number of Android users in 2017. More than 8,000 Android malware were identified everyday making it difficult for the malware analyst to detect them. Traditional malware detection techniques are no longer reliable to detect newly created malware in a short period of time. In this paper, they use a different approach to detect Android malware. The Android malware will be visualized into gray scale images and their image features will be extracted using GIST descriptor. The detection will be done and compared using three different classifiers namely k-nearest neighbor (KNN), Random Forest (RF), and Decision Tree (DT).

Strength: The proposed work was able to achieve 84.14% detection accuracy by using Random Forest machine learning algorithm on image features generated from APK samples. The images were generated from 483 APK samples consisting of 183 malwares and 300 benign samples, and their features were extracted using GIST descriptor.

Limitations: 117 malware samples could not be generated into images because the APK files are either corrupted or they did not contain classes.dex file.

4. OVERALL ARCHITECTURE

The goal of this project is to identify malware that is present statically using machine learning and deep learning computations and a compact executable (PE). Information is frequently defined as a collection of data that has been transformed into a double structure for management. Huge information is typically described as a huge volume of data that may be used for analyzing and developing new advancements and dynamic ML models, albeit it depends on other factors like volume. Using deep learning and neural networks, large amounts of data can also be used to identify viruses.

4.1 Algorithms used: XG-Boost

XGBoost, which stands for Extreme Gradient Boosting, is a scalable, distributed gradient-boosted decision tree (GBDT) machine learning library. It provides parallel tree boosting and is the leading machine learning library for regression, classification, and ranking problems. XGBoost also improves upon the base GBM framework through systems optimization and algorithmic enhancements.

The XGBoost algorithm was developed as a research project at the University of Washington. The algorithm is novel in the following ways:

- Can be used to solve regression, classification, ranking, and user-defined prediction problems.
- Runs smoothly on Windows, Linux, and OS X.
- Supports all major programming languages including C++, Python, R, Java, Scala, and Julia.
- Supports AWS, Azure, and Yarn clusters and works well with Flink, Spark, and other ecosystems.

XGBoost builds upon Supervised Machine Learning. Supervised ML uses algorithms to train a model to find patterns in a dataset with labels and features and then uses the trained model to predict the labels on a new dataset's features.

4.2 Algorithms used: Random Forest

The supervised learning method includes the well-known machine learning algorithm Random Forest. It can be applied to ML Classification and Regression issues. Its foundation is the idea of ensemble learning, which is the process of mixing various classifiers to solve a challenging problem and enhance the performance of the model.

Random Forest, as the name implies, is a classifier that uses a number of decision trees on different subsets of the provided dataset and averages them to increase the dataset's predictive accuracy. Instead of depending on a single decision tree, the random forest uses forecasts from each tree and predicts the result based on the votes of the majority of predictions. Higher accuracy and overfitting are prevented by the larger number of trees in the forest.

Assumptions for Random Forest

Since the random forest combines multiple trees to predict the class of the dataset, it is possible that some decision trees may predict the correct output, while others may not.
But together, all the trees predict the correct output. Therefore, below are two assumptions for a better Random forest classifier:

- There should be some actual values in the feature variable of the dataset so that the classifier can predict accurate results rather than a guessed result.
- The predictions from each tree must have very low correlations.

Working of Random Forest

Random Forest works in two-phase first is to create the random forest by combining N decision tree, and second is to make predictions for each tree created in the first phase.

The Working process can be explained in the below steps and diagram:
1. Select random K data points from the training set.
2. Build the decision trees associated with the selected data points (Subsets).
3. Choose the number N for decision trees that you want to build.
4. Repeat Step 1 & 2.
5. For new data points, find the predictions of each decision tree, and assign the new data points to the category that wins the majority votes.

4.3 Algorithms used: Logistic Regression

One of the most often used Machine Learning algorithms, within the category of Supervised Learning, is logistic regression. Using a predetermined set of independent factors, it is used to predict the categorical dependent variable. In a categorical dependent variable, the output is predicted via logistic regression. As a result, the result must be a discrete or categorical value. It can be True or False, Yes or No, 0 or 1, etc., but rather than delivering an exact value between 0 and 1, it delivers probabilistic values that are in the range of 0 and 1.

Logistic Regression is much similar to Linear Regression except that how they are used. Linear Regression is used for solving Regression problems, whereas Logistic regression is used for solving the classification problems. In Logistic regression, instead of fitting a regression line, we fit an "S" shaped logistic function (sigmoid), which predicts two maximum values (0 or 1). The curve from the logistic function indicates the likelihood of something such as whether the cells are cancerous or not, a mouse is obese or not based on its weight, etc.

Because it can classify new data using both continuous and discrete datasets, logistic regression is a key machine learning approach. When classifying observations using various sources of data, logistic regression can be used to quickly identify the factors that will work well.

Type of Logistic Regression

On the basis of the categories, Logistic Regression can be classified into three types:

- Binomial: In binomial Logistic regression, there can be only two possible types of the dependent variables, such as 0 or 1, Pass or Fail, etc.
- Multinomial: In multinomial Logistic regression, there can be 3 or more possible unordered types of the dependent variable, such as "cat", "dogs", or "sheep"
- Ordinal: In ordinal Logistic regression, there can be 3 or more possible ordered types of dependent variables, such as "low", "Medium", or "High".

Assumptions for Logistic Regression

- The dependent variable must be categorical in nature.
- The independent variable should not have multicollinearity.

4.4 Algorithms used: Neural Networks

Neural networks are a collection of algorithms that are intended to recognise patterns and are loosely based on the human brain. They categorize or group raw input to understand sensory data using a form of machine perception. All real-world data, including images, sounds, texts, and time series, must be converted into vectors in order for them to recognise the patterns, which are numerical and contained within.

- Neural networks help us cluster and classify. They help to group unlabeled data according to similarities among the example inputs, and they classify data when they have a labeled dataset to train on.
- Neural networks can also extract features that are fed to other algorithms for clustering and classification; so you can think of deep neural networks as components of larger machine-learning applications involving algorithms for reinforcement learning, classification and regression.

The term "stacked neural networks," or networks made up of several layers, is used to describe deep learning. Nodes comprise the layers. A node is just a location for computation; it combines input from the data with a set of coefficients, or weights (that either amplify or dampen that input), providing value to inputs in relation to the job the algorithm is attempting to learn. To evaluate if and how far a signal should advance through the network to influence the final result, such as an act of categorization, these input-weight products are added together, and the sum is then sent through a node's so-called activation function. The neuron has "activated" if the signal gets through.

A mathematical model called a neural network (NN) imitates the network organization of the human brain. There are several layers of neurons in it. Artificial neural networks, often known as neural nets, are computer architectures that draw inspiration from the biological neural networks that make up animal brains. It is composed of an Input Layer, a Hidden Layer, and an Output Layer and is based on a network of interconnected units or nodes known as artificial neurons.

5. PROPOSED METHODOLOGY

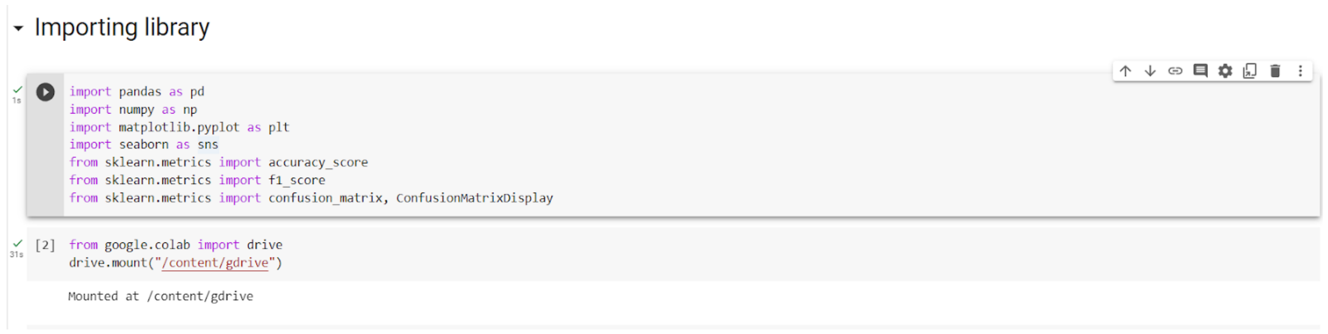
Dataset

For our project we have used a publicly accessible dataset consisting of 41,323 legitimate Windows binaries (exe, dll) and 96,724 malware files from the VirusShare website. The target variable ‘legitimate’ classifies malicious (0) and benign (1) files.

Link to dataset: https://www.kaggle.com/code/luizhemelo/malware-exploratory/data

Workflow

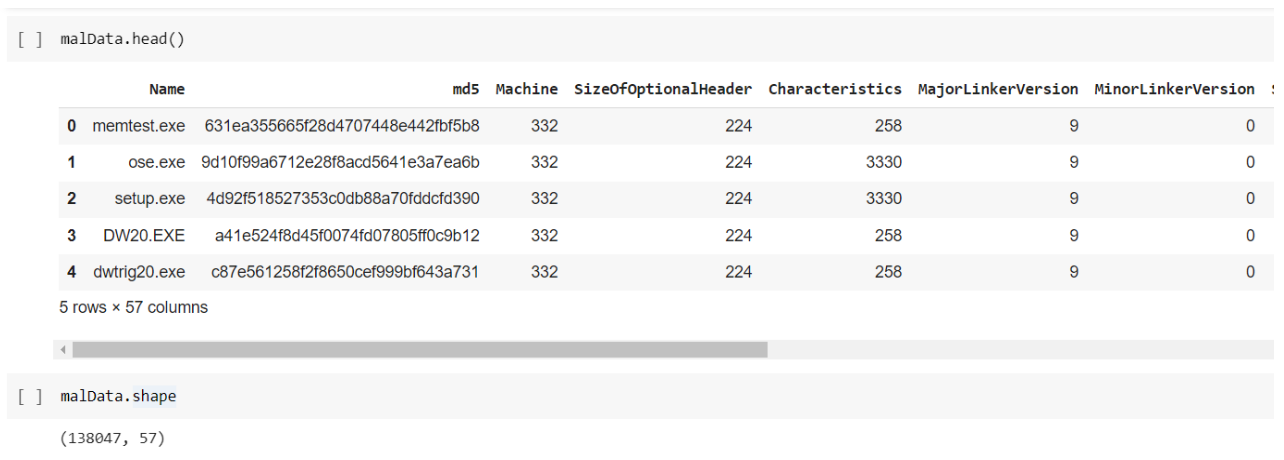
The necessary libraries were imported and the dataset was explored.



Importing library

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
```

```
[2] from google.colab import drive
    drive.mount("/content/gdrive")

    Mounted at /content/gdrive
```

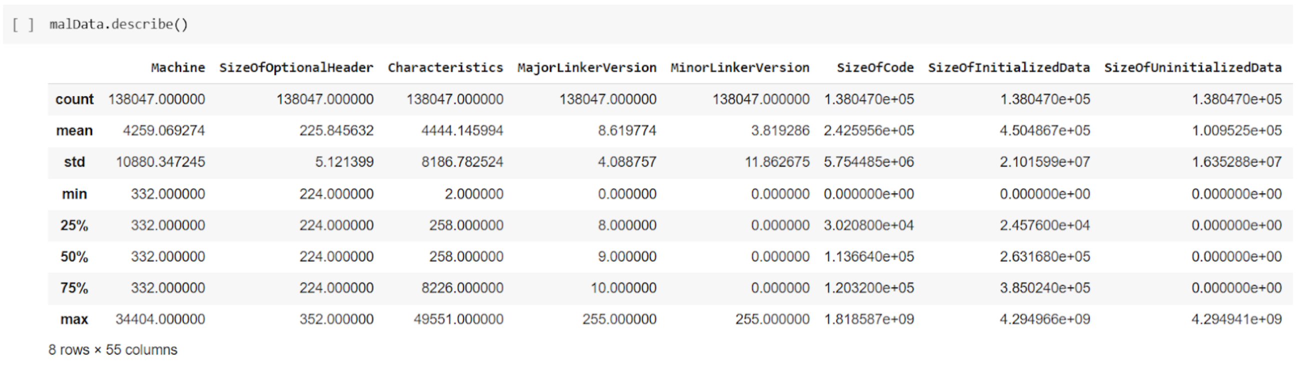


```
[ ] malData.head()
```

	Name	md5	Machine	SizeOfOptionalHeader	Characteristics	MajorLinkerVersion	MinorLinkerVersion
0	memtest.exe	631ea355665f28d4707448e442fbf5b8	332	224	258	9	0
1	ose.exe	9d10f99a6712e28f8acd5641e3a7ea6b	332	224	3330	9	0
2	setup.exe	4d92f518527353c0db88a70fddcfd390	332	224	3330	9	0
3	DW20.EXE	a41e524f8d45f0074fd07805ff0c9b12	332	224	258	9	0
4	dwtrig20.exe	c87e561258f2f8650cef999bf643a731	332	224	258	9	0

5 rows × 57 columns

```
[ ] malData.shape

(138047, 57)
```



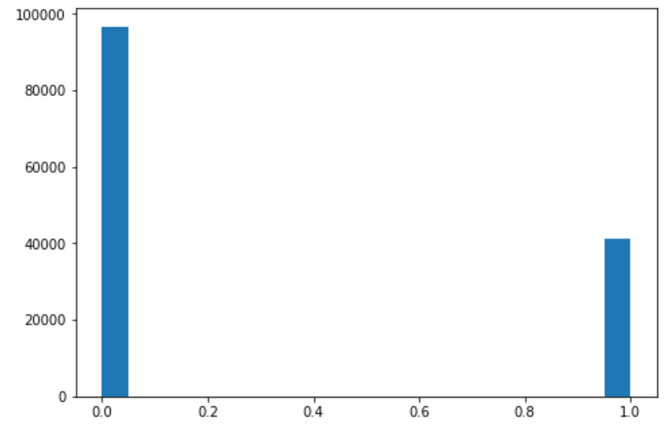
```
[ ] malData.describe()
```

	Machine	SizeOfOptionalHeader	Characteristics	MajorLinkerVersion	MinorLinkerVersion	SizeOfCode	SizeOfInitializedData	SizeOfUninitializedData
count	138047.000000	138047.000000	138047.000000	138047.000000	138047.000000	1.380470e+05	1.380470e+05	1.380470e+05
mean	4259.069274	225.845632	4444.145994	8.619774	3.819286	2.425956e+05	4.504867e+05	1.009525e+05
std	10880.347245	5.121399	8186.782524	4.088757	11.862675	5.754485e+06	2.101599e+07	1.635288e+07
min	332.000000	224.000000	2.000000	0.000000	0.000000	0.000000e+00	0.000000e+00	0.000000e+00
25%	332.000000	224.000000	258.000000	8.000000	0.000000	3.020800e+04	2.457600e+04	0.000000e+00
50%	332.000000	224.000000	258.000000	9.000000	0.000000	1.136640e+05	2.631680e+05	0.000000e+00
75%	332.000000	224.000000	8226.000000	10.000000	0.000000	1.203200e+05	3.850240e+05	0.000000e+00
max	34404.000000	352.000000	49551.000000	255.000000	255.000000	1.818587e+09	4.294966e+09	4.294941e+09

8 rows × 55 columns

```
[ ]  legit=malData[0:41323].drop(["legitimate"], axis=1)
     mal=malData[41323::].drop(["legitimate"], axis=1)

     print("The shape of the legit dataset is: %s samples, %s features "%(legit.shape[0], legit.shape[1]))
     print("The shape of the mal dataset is: %s samples, %s features" %(mal.shape[0], mal.shape[1]))
```

```
The shape of the legit dataset is: 41323 samples, 56 features
The shape of the mal dataset is: 96724 samples, 56 features
```



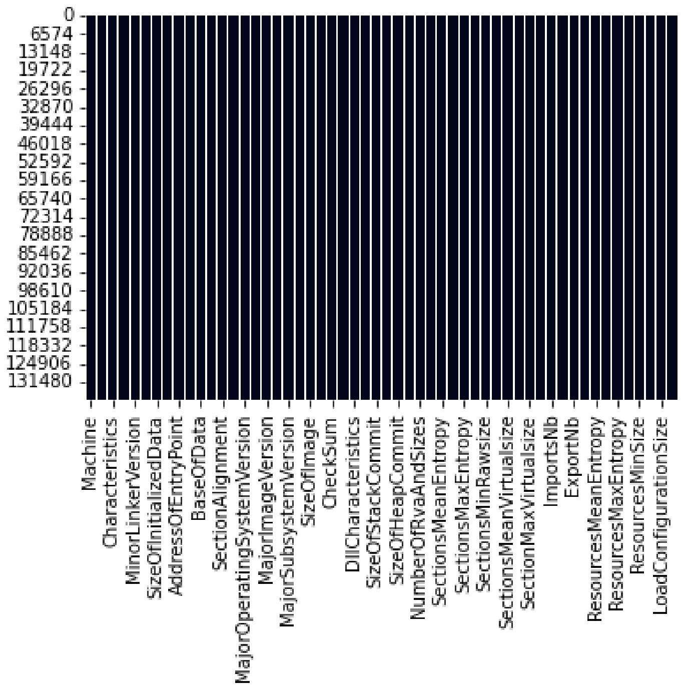
Number of samples of Malicious (0) & Benign (1)

- <u>Data cleaning</u>: The target variable was dropped, unnecessary attributes removed and the missing values checked.

```
[ ]  y=malData['legitimate']
     malData=malData.drop(['legitimate' ], axis=1)
```

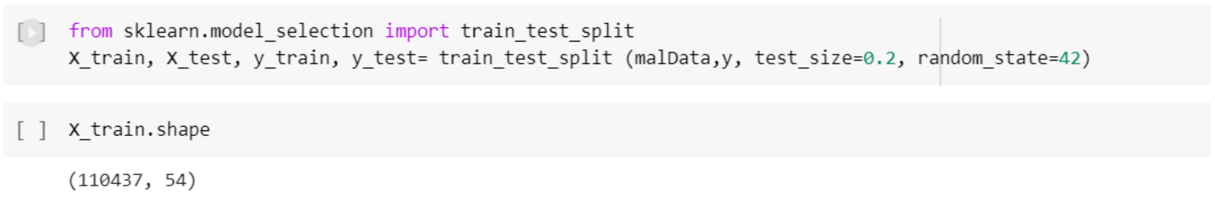
```
[ ]  malData=malData.drop(['Name'], axis=1)
     malData=malData.drop(['md5'], axis=1)
     print(" The Name and md5 variables are removed successfully")
```

```
 The Name and md5 variables are removed successfully
```



No missing values in the Dataset

- Splitting data: For the model training and comparison, the data was split into training and test datasets.



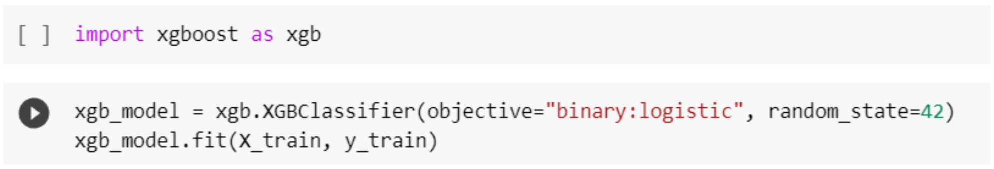
```
[ ] from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test= train_test_split (malData,y, test_size=0.2, random_state=42)
```

```
[ ] X_train.shape

    (110437, 54)
```

- Modeling:

- 1 main model (XGBoost via library and via custom hyperparameters) was developed.
- 3 other models (Random Forest, Logistic Regression, Neural Networks) for comparison against the main model, were also designed.

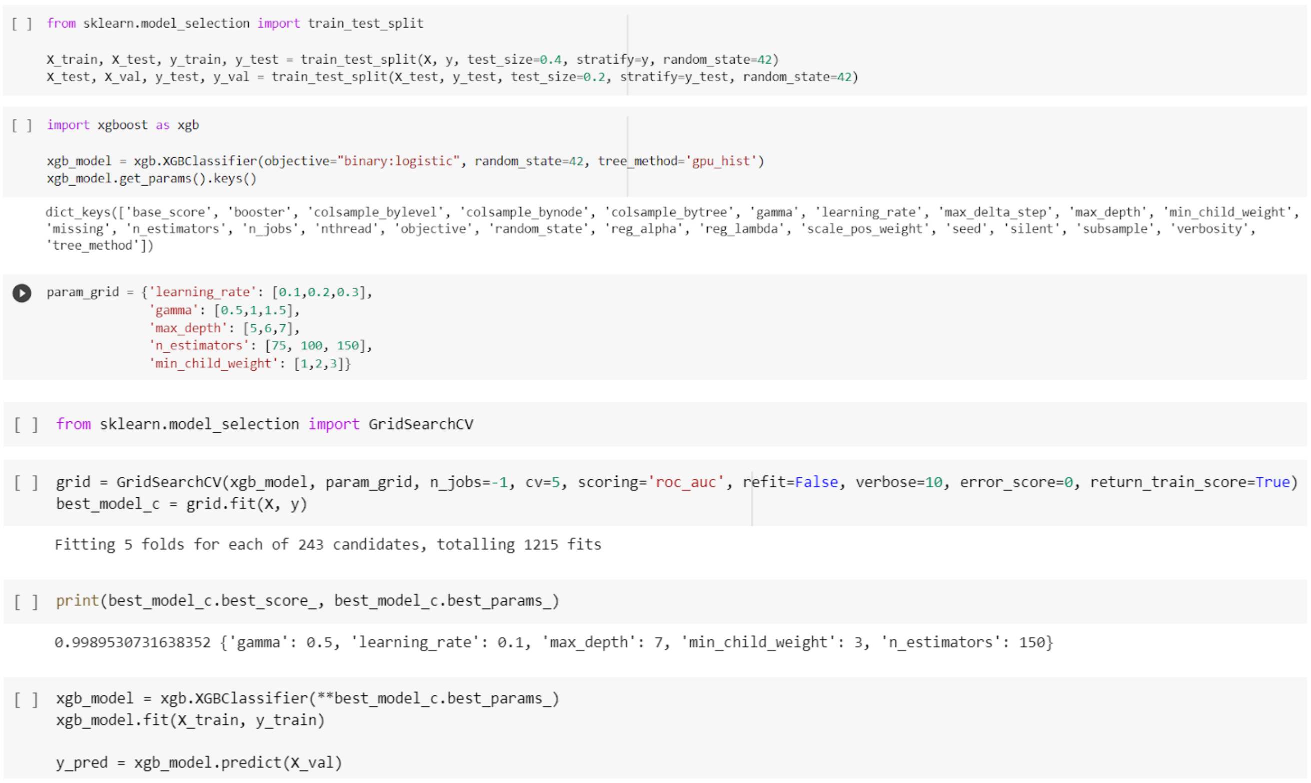
- XGBoost: Taking inspiration from our base paper, we used:
    - One basic model from the scikit-learn library:



```
[ ] import xgboost as xgb
```

```
xgb_model = xgb.XGBClassifier(objective="binary:logistic", random_state=42)
xgb_model.fit(X_train, y_train)
```

    - Then we custom designed the hyperparameters using GridSearchCV:



```
[ ] from sklearn.model_selection import train_test_split

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, stratify=y, random_state=42)
    X_test, X_val, y_test, y_val = train_test_split(X_test, y_test, test_size=0.2, stratify=y_test, random_state=42)
```

```
[ ] import xgboost as xgb

    xgb_model = xgb.XGBClassifier(objective="binary:logistic", random_state=42, tree_method='gpu_hist')
    xgb_model.get_params().keys()

    dict_keys(['base_score', 'booster', 'colsample_bylevel', 'colsample_bynode', 'colsample_bytree', 'gamma', 'learning_rate', 'max_delta_step', 'max_depth', 'min_child_weight',
    'missing', 'n_estimators', 'n_jobs', 'nthread', 'objective', 'random_state', 'reg_alpha', 'reg_lambda', 'scale_pos_weight', 'seed', 'silent', 'subsample', 'verbosity',
    'tree_method'])
```

```
param_grid = {'learning_rate': [0.1,0.2,0.3],
              'gamma': [0.5,1,1.5],
              'max_depth': [5,6,7],
              'n_estimators': [75, 100, 150],
              'min_child_weight': [1,2,3]}
```

```
[ ] from sklearn.model_selection import GridSearchCV
```

```
[ ] grid = GridSearchCV(xgb_model, param_grid, n_jobs=-1, cv=5, scoring='roc_auc', refit=False, verbose=10, error_score=0, return_train_score=True)
    best_model_c = grid.fit(X, y)

    Fitting 5 folds for each of 243 candidates, totalling 1215 fits
```

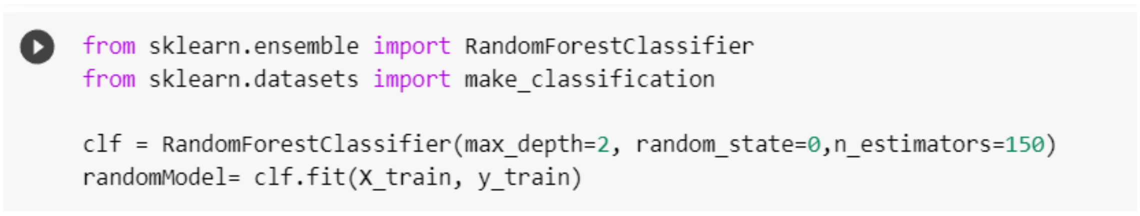
```
[ ] print(best_model_c.best_score_, best_model_c.best_params_)

    0.9989530731638352 {'gamma': 0.5, 'learning_rate': 0.1, 'max_depth': 7, 'min_child_weight': 3, 'n_estimators': 150}
```

```
[ ] xgb_model = xgb.XGBClassifier(**best_model_c.best_params_)
    xgb_model.fit(X_train, y_train)

    y_pred = xgb_model.predict(X_val)
```

❖ GridSearchCV searches for the most optimal hyperparameters, for the model, while simultaneously cross validating each combination to correct overfitting.

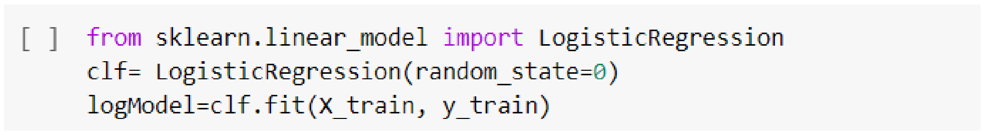
❖ Grid searches on huge data can be computationally exhaustive and time-taking, but we have optimized it by building a model that utilizes GPU in its runtime.

- Random Forest:



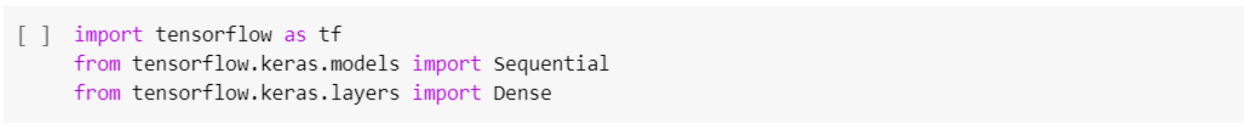
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification

clf = RandomForestClassifier(max_depth=2, random_state=0,n_estimators=150)
randomModel= clf.fit(X_train, y_train)
```

- Logistic Regression:

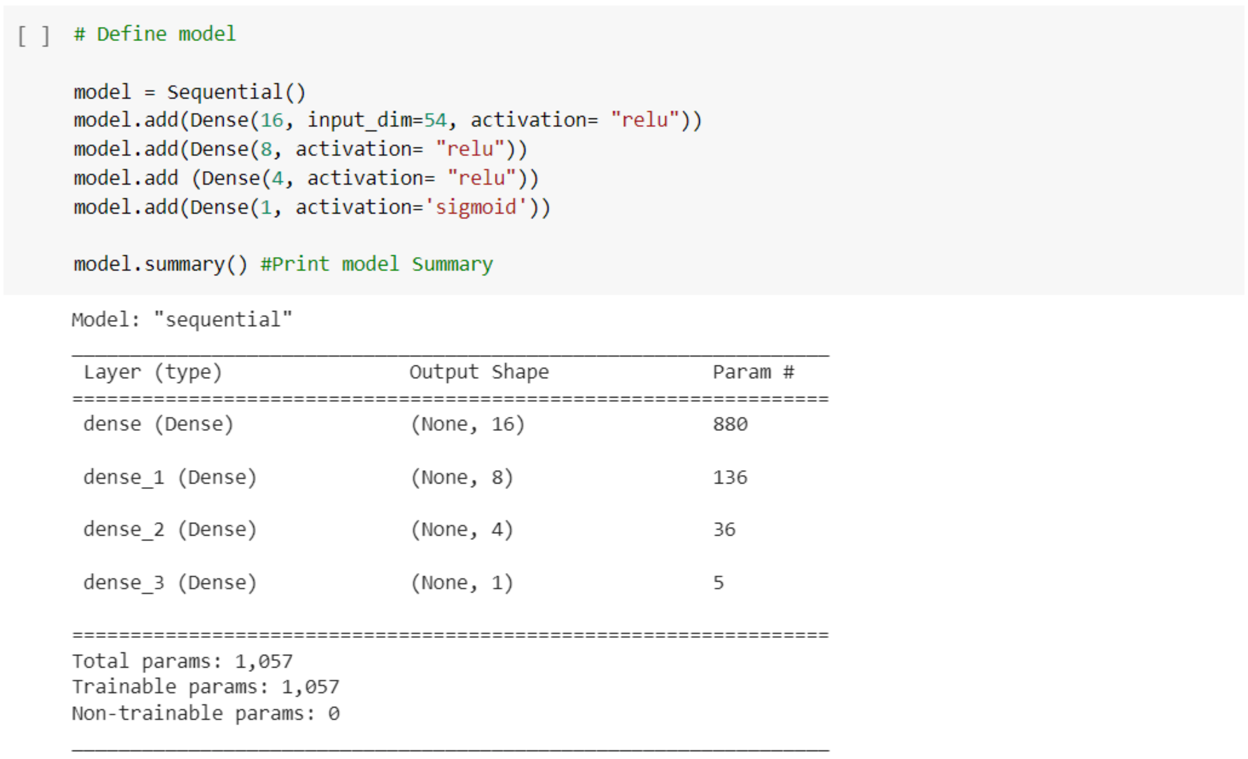


```
[ ]  from sklearn.linear_model import LogisticRegression
     clf= LogisticRegression(random_state=0)
     logModel=clf.fit(X_train, y_train)
```

- Neural Network:



```
[ ]  import tensorflow as tf
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Dense
```



```
[ ]  # Define model

     model = Sequential()
     model.add(Dense(16, input_dim=54, activation= "relu"))
     model.add(Dense(8, activation= "relu"))
     model.add (Dense(4, activation= "relu"))
     model.add(Dense(1, activation='sigmoid'))

     model.summary() #Print model Summary
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 16)                880

 dense_1 (Dense)             (None, 8)                 136

 dense_2 (Dense)             (None, 4)                 36

 dense_3 (Dense)             (None, 1)                 5

=================================================================
Total params: 1,057
Trainable params: 1,057
Non-trainable params: 0
_________________________________________________________________
```

```
[ ] # Compile model

    model.compile(loss= "binary_crossentropy", optimizer="rmsprop", metrics=["accuracy"])
```

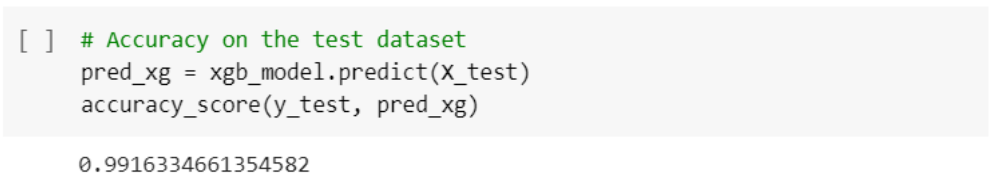
```
[ ] # Fit Model

    model.fit(X_train, y_train, epochs=5, batch_size=32)
```

```
Epoch 1/5
3452/3452 [==============================] - 16s 4ms/step - loss: 12945397.0000 - accuracy: 0.9482
Epoch 2/5
3452/3452 [==============================] - 12s 4ms/step - loss: 726055.3125 - accuracy: 0.9450
Epoch 3/5
3452/3452 [==============================] - 12s 3ms/step - loss: 742398.3125 - accuracy: 0.9338
Epoch 4/5
3452/3452 [==============================] - 11s 3ms/step - loss: 623780.7500 - accuracy: 0.9316
Epoch 5/5
3452/3452 [==============================] - 11s 3ms/step - loss: 139100.1406 - accuracy: 0.8129
<keras.callbacks.History at 0x7f06902fabd0>
```

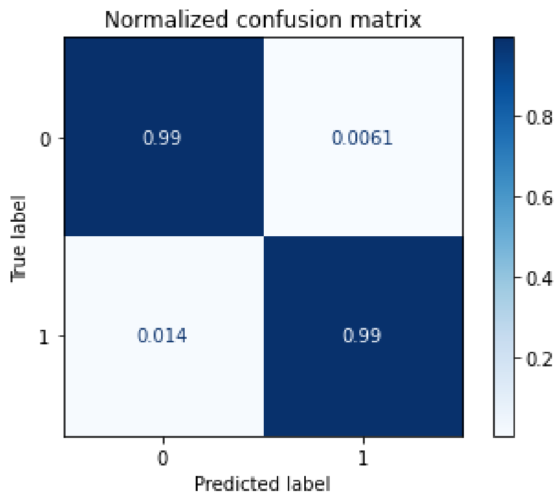
6. RESULTS & ANALYSIS

The performance metrics used are accuracy scores (both on the training and test datasets respectively) and F1 score (since the dataset is imbalanced).

- XGBoost:
  - The basic model reported an accuracy of 99.1%:

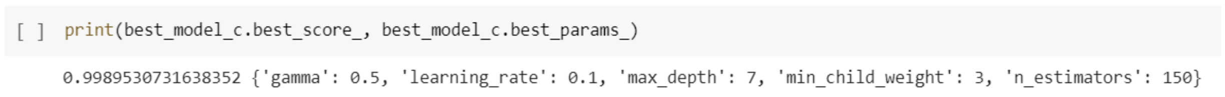


```
[ ] # Accuracy on the test dataset
    pred_xg = xgb_model.predict(X_test)
    accuracy_score(y_test, pred_xg)

    0.9916334661354582
```



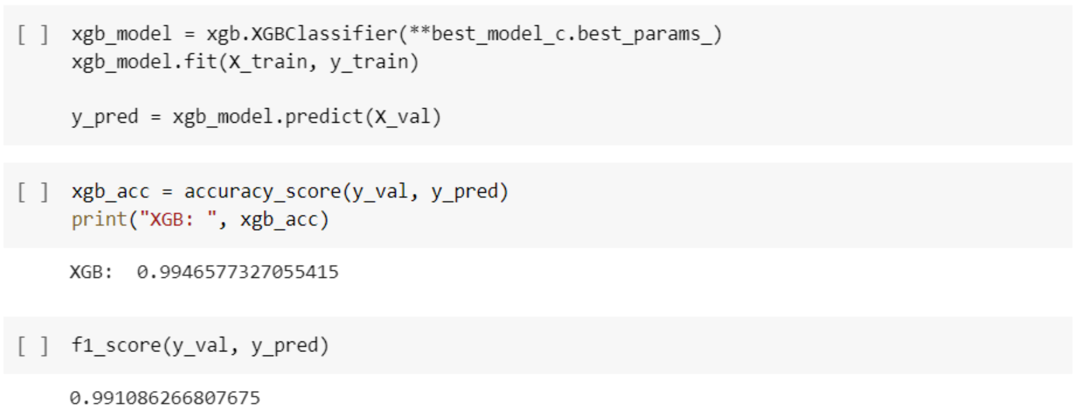
Confusion Matrix for Basic XGBoost

  - Our custom designed model using hyperparameter tuning and GridSearchCV achieved a higher accuracy with the shortlisted parameters:



```
[ ] print(best_model_c.best_score_, best_model_c.best_params_)

    0.9989530731638352 {'gamma': 0.5, 'learning_rate': 0.1, 'max_depth': 7, 'min_child_weight': 3, 'n_estimators': 150}
```

  - Our accuracy with the hyperparameter tuned model is at 99.465 %, which is greater than the basic model. Our model also achieves a good F1-score: 99.1%



```
[ ] xgb_model = xgb.XGBClassifier(**best_model_c.best_params_)
    xgb_model.fit(X_train, y_train)

    y_pred = xgb_model.predict(X_val)

[ ] xgb_acc = accuracy_score(y_val, y_pred)
    print("XGB: ", xgb_acc)

    XGB:  0.9946577327055415

[ ] f1_score(y_val, y_pred)

    0.991086266807675
```

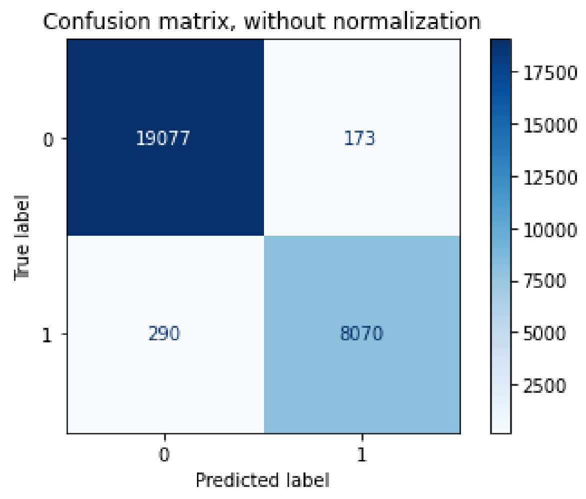
- Random Forest:

```
[ ]  from sklearn.metrics import f1_score, accuracy_score,plot_confusion_matrix, auc, confusion_matrix
```

```
[ ]  # Accuracy on the train dataset
     train_pred=randomModel.predict(X_train)
     accuracy_score(y_train, train_pred)

     0.982379093963074
```

```
[ ]  # Accuracy on the test dataset
     prediction=randomModel.predict(X_test)
     accuracy_score(y_test, prediction)

     0.983230713509598
```

```
[ ]  f1_score(y_test, prediction)

     0.9721134734686503
```



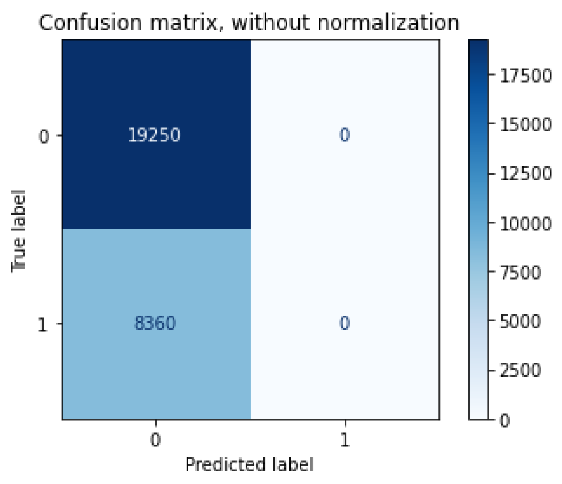
Confusion Matrix for Random Forest

- Logistic Regression:

```
[ ]  # Accuracy on the train dataset
     train_log= logModel.predict(X_train)
     accuracy_score(y_train, train_log)

     0.7015221347917817
```

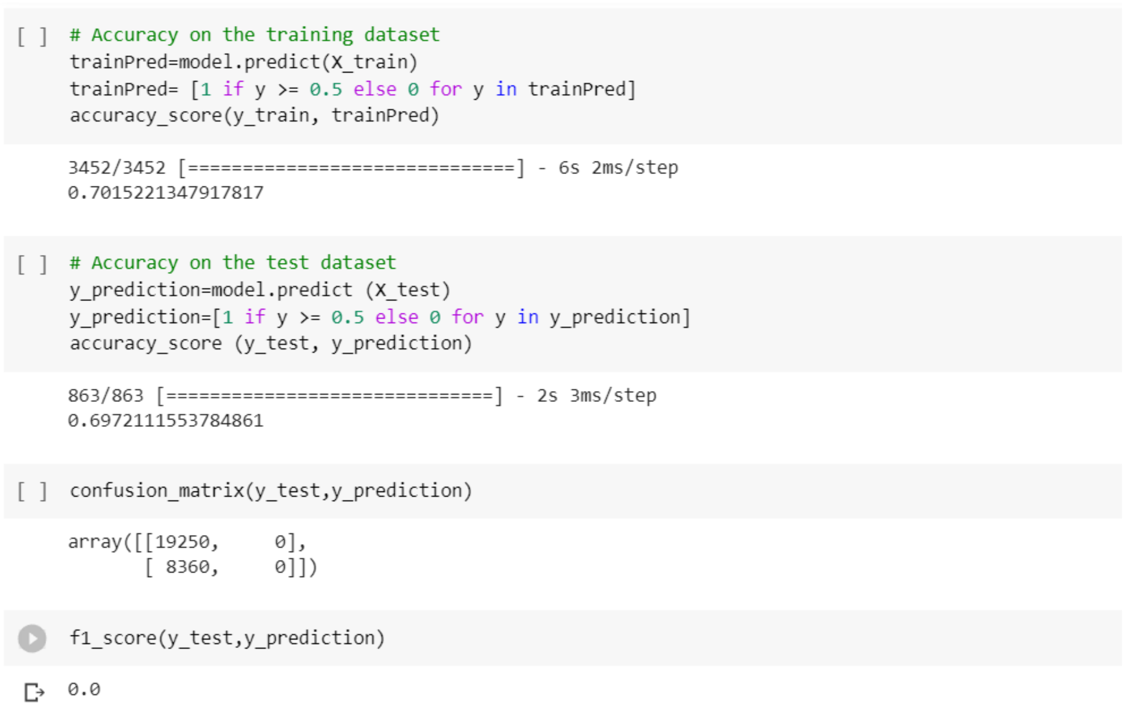
```
[ ]  # Accuracy on the test dataset
     pred=logModel.predict(X_test)
     accuracy_score(y_test, pred)

     0.6972111553784861
```

```
[ ]  f1_score (y_test, pred)

     0.0
```



Confusion Matrix for Logistic Regression

- <u>Neural Network:</u>



```
[ ] # Accuracy on the training dataset
    trainPred=model.predict(X_train)
    trainPred= [1 if y >= 0.5 else 0 for y in trainPred]
    accuracy_score(y_train, trainPred)

    3452/3452 [==============================] - 6s 2ms/step
    0.7015221347917817

[ ] # Accuracy on the test dataset
    y_prediction=model.predict (X_test)
    y_prediction=[1 if y >= 0.5 else 0 for y in y_prediction]
    accuracy_score (y_test, y_prediction)

    863/863 [==============================] - 2s 3ms/step
    0.6972111553784861

[ ] confusion_matrix(y_test,y_prediction)

    array([[19250,     0],
           [ 8360,     0]])

    f1_score(y_test,y_prediction)

    0.0
```

Comparative Analysis of Model Metrics

Model	Accuracy on Training Dataset	Accuracy on Test Dataset	F1-Score	Rank
XGBoost	99.89%	99.456%	99.1%	1
Random Forest	98.32%	98.23%	97.21%	2
Logistic Regression	70.15%	69.72%	0.0	3
ANN	70.15%	69.72%	0.0	3

7. CONCLUSION AND FUTURE WORK

The objective of our project was to develop a model which would detect and consequently prevent the entry of malware with other information packets. This is because in the modern world, the threat of malware is growing along with the growing technological advancements. Hence it is only natural that the threat to these devices known as malwares are potent and easier to infect.

Our project uses efficient Machine Learning Algorithms and one Deep Learning Model to detect these threats and achieve a high accuracy of 99.456% with our custom XGBoost Classifier Model. Among malware detection methods used, ours performed quite well compared to existing library models. The three models used were XGBoost Classifier, Random Forest, Logistic Regression and Neural Networks all of them being efficient in their own ways. Thus the time taken to train and build the models becomes shorter at the cost of the performance decreases slightly. In some cases, the performance can also increase slightly.

The performance comparison of 3 different classifiers was also presented. Apart from the XGBoost Classifier Model, the Random forest classifier produced 98.2% accuracy for the train data accuracy, 98.3% for the test data accuracy and 97.2% for the F1-score. The Logistic Regression classifier produced just 70% accuracy for the train data accuracy, 69% for the test data accuracy and a discombobulating 0% accuracy for the F1- score. The Neural Networks classifier produced a 70.1% accuracy for the train data accuracy, 69% for the test data accuracy and a 0.0 F1-score.

In conclusion, our most efficient model, the custom XGBoost Classifier, provides exceptional results which are expected to stay relevant in the following years to come. Thus our project will be useful to aid the detection of malware analysis for the foreseeable future.

REFERENCES

[1] R. Elnaggar, L. Servadei, S. Mathur, R. Wille, W. Ecker and K. Chakrabarty, "Accurate and Robust Malware Detection: Running XGBoost on Runtime Data From Performance Counters," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 41, no. 7, pp. 2066-2079, July 2022, doi: 10.1109/TCAD.2021.3102007

[2] O. N. Elayan and A. M. Mustafa, “Android malware detection using deep learning,” Procedia Comput. Sci., vol. 184, no. 2019, pp. 847–852, 2021, doi: 10.1016/j.procs.2021.03.106.

[3] M. Ashik et al., “Detection of malicious software by analyzing distinct artifacts using machine learning and deep learning algorithms,” Electron., vol. 10, no. 14, pp. 1–28, 2021, doi: 10.3390/electronics10141694

[4] K. Sethi, R. Kumar, L. Sethi, P. Bera, and P. K. Patra, “A novel machine learning based malware detection and classification framework,” 2019 Int. Conf. Cyber Secur. Prot. Digit. Serv. Cyber Secur. 2019, 2019, doi: 10.1109/CyberSecPODS.2019.8885196

[5] N. Balram, G. Hsieh, and C. McFall, “Static malware analysis using machine learning algorithms on apt1 dataset with string and PE header features,” Proc. - 6th Annu. Conf. Comput. Sci. Comput. Intell. CSCI 2019, pp. 90–95, 2019, doi: 10.1109/CSCI49370.2019.00022.

[6] Matin, I. M. M., & Rahardjo, B. (2020). The Use of Honeypot in Machine Learning Based on Malware Detection: A Review. 2020 8th International Conference on Cyber and IT Service Management, CITSM 2020. https://doi.org/10.1109/CITSM50537.2020.9268794

[7] F. M. Darus, N. A. A. Salleh and A. F. Mohd Ariffin, "Android Malware Detection Using Machine Learning on Image Patterns," 2018 Cyber Resilience Conference (CRC), 2018, pp. 1-2, doi: 10.1109/CR.2018.8626828

[8] J. Li, L. Sun, Q. Yan, Z. Li, W. Srisa-an and H. Ye, "Significant Permission Identification for Machine-Learning-Based Android Malware Detection," in IEEE Transactions on Industrial Informatics, vol. 14, no. 7, pp. 3216-3225, July 2018, doi: 10.1109/TII.2017.2789219

[9] F. M. Darus, N. A. A. Salleh and A. F. Mohd Ariffin, "Android Malware Detection Using Machine Learning on Image Patterns," 2018 Cyber Resilience Conference (CRC), 2018, pp. 1-2, doi: 10.1109/CR.2018.8626828

[10] A. Fatima, R. Maurya, M. K. Dutta, R. Burget and J. Masek, "Android Malware Detection Using Genetic Algorithm based Optimized Feature Selection and Machine Learning," 2019 42nd International Conference on Telecommunications and Signal Processing (TSP), 2019, pp. 220-223, doi: 10.1109/TSP.2019.8769039

[11] C. Galen and R. Steele, "Performance Maintenance Over Time of Random Forest-based Malware Detection Models," 2020 11th IEEE Annual Ubiquitous Computing, Electronics & Mobile

Communication Conference (UEMCON), 2020, pp. 0536-0541, doi: 10.1109/UEMCON51285.2020.9298068

[12] M. Goyal and R. Kumar, "Machine Learning for Malware Detection on Balanced and Imbalanced Datasets," 2020 International Conference on Decision Aid Sciences and Application (DASA), 2020, pp. 867-871, doi: 10.1109/DASA51403.2020.9317206

[13] B. J. Kumar, H. Naveen, B. P. Kumar, S. S. Sharma and J. Villegas, "Logistic regression for polymorphic malware detection using ANOVA F-test," 2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS), 2017, pp. 1-5, doi: 10.1109/ICIIECS.2017.8275880

[14] https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
https://ieeexplore.ieee.org/abstract/document/8661441/

APPENDIX: CODE

```
# Importing library

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import accuracy_score
from sklearn.metrics import f1_score
from sklearn.metrics import plot_confusion_matrix

from google.colab import drive
drive.mount("/content/gdrive")

malData=pd.read_csv("/content/gdrive/MyDrive/Colab Notebooks/ISM
JComp/malware.csv", sep="|",low_memory= True)

# Exploring malware dataset
<b>malware dataset contains</b><br>
 41,323 Windows binaries (exe,dll) - legitimate<br>
 96,724 malware files from the VirusShare website <br>

  138,048 lines, in total.

https://www.kaggle.com/code/luizhemelo/malware-exploratory/data

malData.head()

malData.shape

malData.describe()

legit=malData[0:41323].drop(["legitimate"], axis=1)
mal=malData[41323::].drop(["legitimate"], axis=1)

print("The shape of the legit dataset is: %s samples, %s features
"%(legit.shape[0], legit.shape[1]))
print("The shape of the mal dataset is: %s samples, %s features"
%(mal.shape[0], mal.shape[1]))

fig = plt.figure()
ax = fig.add_axes ([0,0,1,1])
ax.hist(malData['legitimate'], 20)
plt.show()

# Data cleaning

y=malData['legitimate']
malData=malData.drop(['legitimate' ], axis=1)
```

```
malData=malData.drop(['Name'], axis=1)
malData=malData.drop(['md5'], axis=1)
print(" The Name and md5 variables are removed successfully")

# from sklearn.feature selection import SelectKBest
# from sklearn.feature_selection import chi2
# X_new = SelectKBest (chi2, k=2).fit_transform(malData, y)
# Xx_new.shape

sns.heatmap(malData.isnull(), cbar=False)

# Spliting the dataset into test and train

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test= train_test_split (malData,y,
test_size=0.2, random_state=42)

X_train.shape

# Model Building

## 1- XG Boost

import xgboost as xgb

xgb_model = xgb.XGBClassifier(objective="binary:logistic",
random_state=42)
xgb_model.fit(X_train, y_train)

y_pred = xgb_model.predict(X_train)
accuracy_score(y_train, y_pred)

# Accuracy on the test dataset
pred_xg = xgb_model.predict(X_test)
accuracy_score(y_test, pred_xg)

from warnings import simplefilter
simplefilter(action='ignore', category=FutureWarning)

#title_options =[("Confusion matrix, without normalization", None),
("Normalized confusion matrix", 'true')]
disp = plot_confusion_matrix(xgb_model, X_test, y_test, cmap=plt.cm.Blues,
normalize='true')
disp.ax_.set_title("Normalized confusion matrix")
plt.show()

### Modification 1 - Overfitting

malMod=pd.read_csv("/content/gdrive/MyDrive/Colab Notebooks/ISM
JComp/malware.csv", sep="|",low_memory= True)
```

```
malMod = malMod.drop(['Name', 'md5'], axis=1)

features = malMod.columns.to_list()
features.remove('legitimate')

X = malMod[features]
y = malMod['legitimate']

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
stratify=y, random_state=42)
X_test, X_val, y_test, y_val = train_test_split(X_test, y_test,
test_size=0.2, stratify=y_test, random_state=42)

import xgboost as xgb

xgb_model = xgb.XGBClassifier(objective="binary:logistic",
random_state=42, tree_method='gpu_hist')
xgb_model.get_params().keys()

param_grid = {'learning_rate': [0.1,0.2,0.3],
              'gamma': [0.5,1,1.5],
              'max_depth': [5,6,7],
              'n_estimators': [75, 100, 150],
              'min_child_weight': [1,2,3]}

# param_grid = {'learning_rate': [0.01, 0.1, 0.2],
                # 'max_depth': [4,5,6],
                # 'n_estimators': [100, 150, 200],
                # 'min_child_weight': [1, 2, 3]}

from sklearn.model_selection import GridSearchCV

grid = GridSearchCV(xgb_model, param_grid, n_jobs=-1, cv=5,
scoring='roc_auc', refit=False, verbose=10, error_score=0,
return_train_score=True)
best_model_c = grid.fit(X, y)

print(best_model_c.best_score_, best_model_c.best_params_)

xgb_model = xgb.XGBClassifier(**best_model_c.best_params_)
xgb_model.fit(X_train, y_train)

y_pred = xgb_model.predict(X_val)

xgb_acc = accuracy_score(y_val, y_pred)
print("XGB: ", xgb_acc)

f1_score(y_val, y_pred)
```

```
## 2  -Random Forest

from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification

clf = RandomForestClassifier(max_depth=2, random_state=0,n_estimators=150)
randomModel= clf.fit(X_train, y_train)

### Random forest Evaluation on test data

from sklearn.metrics import f1_score,
accuracy_score,plot_confusion_matrix, auc, confusion_matrix

# Accuracy on the train dataset
train_pred=randomModel.predict(X_train)
accuracy_score(y_train, train_pred)

# Accuracy on the test dataset
prediction=randomModel.predict(X_test)
accuracy_score(y_test, prediction)

f1_score(y_test, prediction)

### confusion matrix

titles_options =[("Confusion matrix, without normalization", None),
                 ("Normalized confusion matrix", 'true')]
for title, normalize in titles_options:
    disp = plot_confusion_matrix(randomModel, X_test, y_test,
                                 cmap=plt.cm.Blues,
                                 normalize=normalize)
    disp.ax_.set_title(title)
    print(title)
    print(disp.confusion_matrix)
    plt.show()

# 3 - Logistic Regression

from sklearn.linear_model import LogisticRegression
clf= LogisticRegression(random_state=0)
logModel=clf.fit(X_train, y_train)

### Model Evaluation

# Accuracy on the train dataset
train_log= logModel.predict(X_train)
accuracy_score(y_train, train_log)

# Accuracy on the test dataset
pred=logModel.predict(X_test)
accuracy_score(y_test, pred)
```

```
f1_score (y_test, pred)

### Confusion Matrix

titles_options = [("Confusion matrix, without normalization", None),
                  ("Normalized confusion matrix", 'true')]

for title, normalize in titles_options:
    disp = plot_confusion_matrix (logModel, X_test, y_test,
                                  cmap=plt.cm.Blues,
                                  normalize=normalize)
    disp.ax_.set_title(title)
    print(title)
    print(disp.confusion_matrix)
    plt.show()

# 4 - Neural Network

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Define model

model = Sequential()
model.add(Dense(16, input_dim=54, activation= "relu"))
model.add(Dense(8, activation= "relu"))
model.add (Dense(4, activation= "relu"))
model.add(Dense(1, activation='sigmoid'))

model.summary() #Print model Summary

# Compile model

model.compile(loss= "binary_crossentropy", optimizer="rmsprop",
metrics=["accuracy"])

# Fit Model

model.fit(X_train, y_train, epochs=5, batch_size=32)

## Model Evaluation

# Accuracy on the training dataset
trainPred=model.predict(X_train)
trainPred= [1 if y >= 0.5 else 0 for y in trainPred]
accuracy_score(y_train, trainPred)

# Accuracy on the test dataset
y_prediction=model.predict (X_test)
```

```
y_prediction=[1 if y >= 0.5 else 0 for y in y_prediction]
accuracy_score (y_test, y_prediction)

confusion_matrix(y_test,y_prediction)

f1_score(y_test,y_prediction)
```